PowerServer 2021 Help

PowerServer 2021

Contents

Installation

Contents

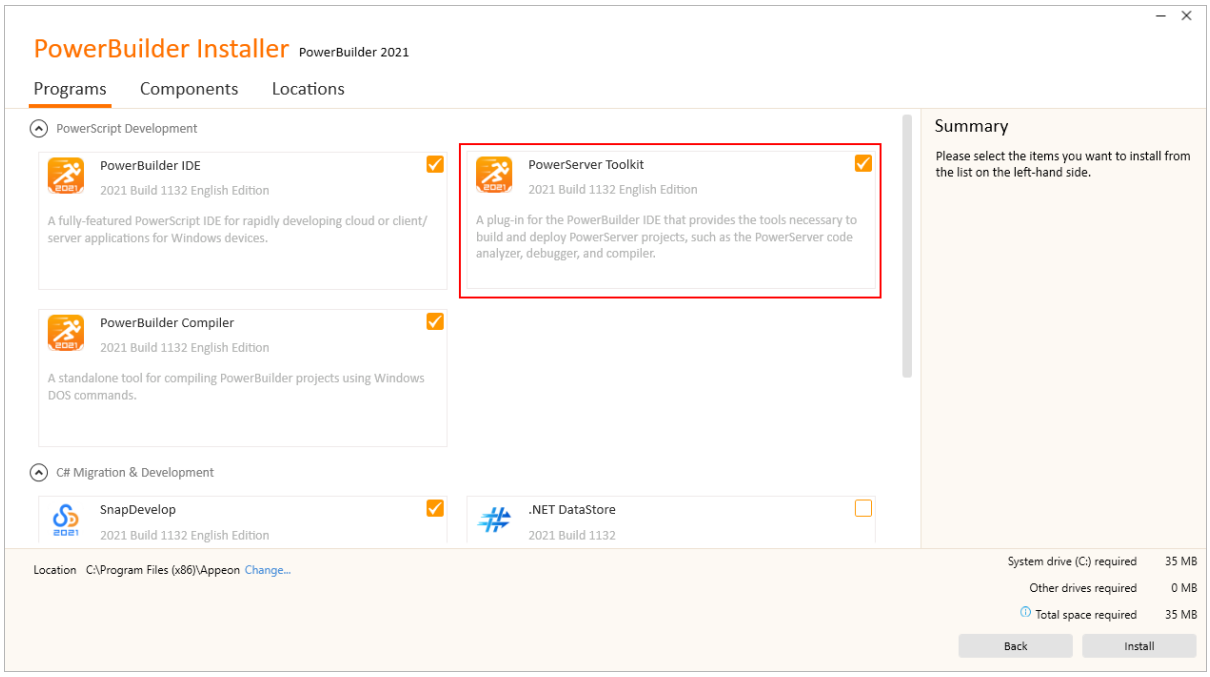
1 PowerServer components

PowerServer 2021 is comprised of two parts:

- PowerServer Toolkit -- Provides utilities for 1) Creating and managing PowerServer projects; 2) Analyzing and compiling the application; 3) Generating and deploying the application web files (PBD files and supporting files) to the web server; 4) Generating a PowerServer Web APIs solution; 5) Compiling and running the PowerServer Web APIs (in local environment); 6) Running a PowerServer project; 6) Generating build files from existing PowerServer projects for auto-build, etc.

  PowerServer Toolkit is provided as a component in the PowerBuilder Installer and installed as a plug-in to the PowerBuilder IDE. For how to run the PowerBuilder Installer, refer to Installation Guide for PowerBuilder IDE. The PowerServer Toolkit is by default installed to %AppeonInstallPath%\Common\PSToolkit\[version]\.

  Figure 1.1:

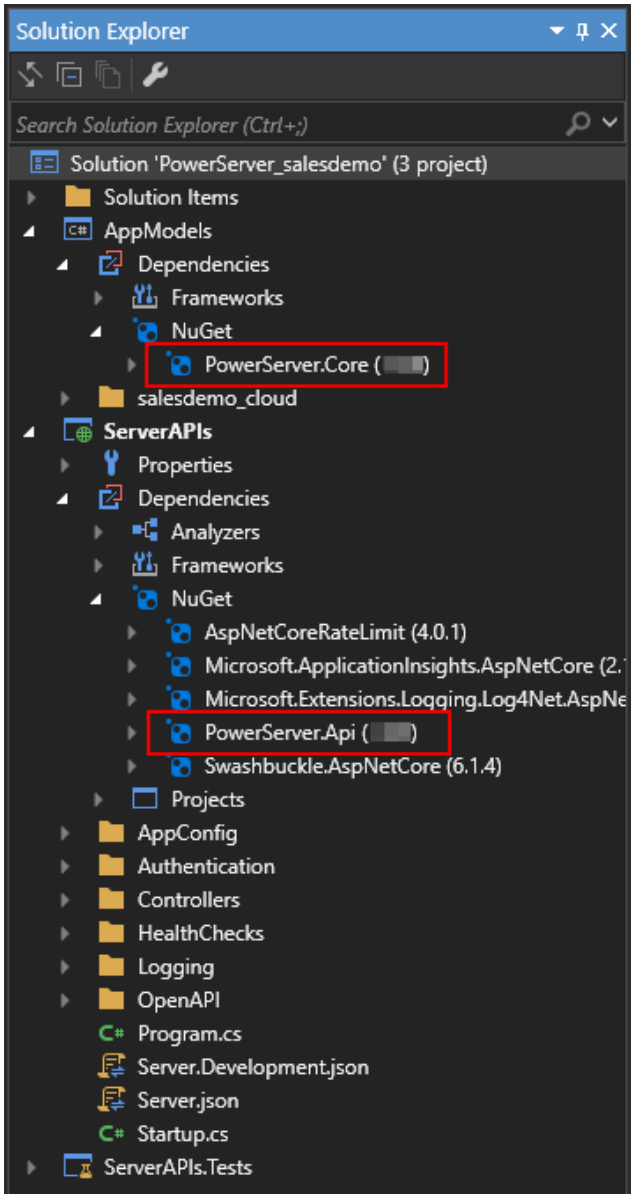


- The PowerServer NuGet packages -- The runtime library for the PowerServer Web APIs that supports data processing, authorization, licensing etc. When you launch the PowerServer C# solution, the PowerServer NuGet packages will be automatically downloaded and installed from the NuGet website (https://www.nuget.org). Please make sure the computer can connect to the NuGet website (https://www.nuget.org).

  The PowerServer NuGet packages can be downloaded to SnapDevelop or any other .NET IDE such as Visual Studio. No matter which .NET IDE you are using, the instructions on how to install, update, and uninstall the PowerServer packages are the same as all the other NuGet packages. For detailed instructions, please refer to this documentation https://docs.microsoft.com/nuget/quickstart/install-and-use-a-package-in-visual-studio.

  The PowerServer NuGet packages are free to download but must be activated before it can work properly. For more, refer to Import license and activate PowerServer.

Figure 1.2:



Note

PowerServer 2021 will only work with PowerBuilder 2021. Before deploying your application with PowerServer 2021, make sure 1) your application is upgraded to be compatible with PowerBuilder 2021; and 2) you have a PowerBuilder CloudPro license (paid or trial).

Note

PowerServer 2020 or earlier cannot be upgraded to PowerServer 2021; and applications deployed with PowerServer 2020 or earlier cannot work with PowerServer 2021.

---

2 Installation requirements

2.1 Client PC

To run the installable cloud app, install the following OS and Web browser:

- Windows 10 or 8.1, or Windows Server 2019, 2016, or 2012 R2
- Google Chrome, Mozilla Firefox, or Microsoft Edge (Chromium-based)

2.2 Development PC

It is recommended that PowerBuilder IDE, PowerBuilder Runtime, PowerServer Toolkit, and PowerBuilder Compiler are the same version and build.

For installation instructions, refer to Installation Guide for PowerBuilder IDE.

Note

You must have administrator privileges to run the PowerBuilder Installer and install some components.

Table 2.1:

To	Install the following
Build and deploy the PowerServer project	• Windows 10 or 8.1 • PowerBuilder IDE 2021 • PowerBuilder Runtime 2021 • PowerServer Toolkit 2021
Build and deploy the PowerServer project using the **PBAutoBuild210.exe** command	• Windows 10 or 8.1, or Windows Server 2019, 2016, or 2012 R2 • PowerBuilder Runtime 2021 • PowerServer Toolkit 2021 • PowerBuilder Compiler 2021 (or PowerBuilder IDE 2021)
Compile and publish the PowerServer Web APIs	• Windows 10 or 8.1 • SnapDevelop 2021 or Visual Studio 2019 The computer must be able to connect to the NuGet site (https://www.nuget.org), in order to download the packages required for compilation.

2.3 Web Server

The app files can be hosted in the following Web servers:

- Windows IIS

  For how to install and configure IIS, refer to Setting up IIS.

- Windows/Linux Apache

  For how to install and configure Apache in Windows, refer to Setting up Apache on Windows.

  For how to install and configure Apache in Linux, refer to Setting up Apache on Linux.

- Windows/Linux Nginx

  For how to configure Nginx in Windows, refer to Setting up Nginx on Windows.

  For how to configure Nginx in Linux, refer to Setting up Nginx on Linux.

* Kestrel is not recommended to be used as the Web server for hosting the app files.

* Any version within the support period is supported.

2.4 .NET Server

The PowerServer Web APIs is an ASP.NET Core 3.1 app; it can be hosted and deployed like any other ASP.NET Core app.

The following are the most popular hosting environments:

- Windows/Linux Docker

  For how to publish the PowerServer Web APIs to Docker, refer to Tutorial 2: Hosting Web APIs in Docker Containers.

- Kubernetes

- Windows IIS

  For how to publish the PowerServer Web APIs to IIS, refer to Tutorial 3: Hosting Web APIs in IIS.

- Windows/Linux Kestrel (with or without a reverse proxy server)

  For how to run the PowerServer Web APIs on Kestrel, refer to Tutorial 4: Hosting Web APIs in Kestrel.

* Any version within the support period is supported.

For a complete list of supported environments, refer to https://docs.microsoft.com/aspnet/core/host-and-deploy/?view=aspnetcore-3.1.

2.5 Database Server

The installable cloud apps can work with the following databases:

- Oracle 12c, 18c, or 19c

  PowerBuilder and/or PowerServer will automatically download the required driver (Oracle.ManagedDataAccess.Core 2.19.110) from https://www.nuget.org, or you will be asked to specify the location of the driver if https://www.nuget.org cannot be connected.

- PostgreSQL 11.3, 12, or 13

- SQL Server 2016, 2017, or 2019

- SQL Anywhere (ODBC) 16 (16.0.0.2043 or later) or 17

  If SQL Anywhere is on a different machine from PowerBuilder, make sure to enable the connection pooling setting in the ODBC driver. Connection pooling is enabled by default if SQL Anywhere is on the same machine as PowerBuilder.

- ASE (ODBC) 16.0

  ASE databases can only be connected using the ODBC driver in the PowerServer runtime environment. This is different from the PowerBuilder runtime environment where the ASE database is connected using the native driver. See ASE database for the differences caused by this driver change.

- MySQL 5.6, 5.7, or 8.0

  PowerBuilder and/or PowerServer will automatically download the required driver (MySql.Data 8.0.25) from https://www.nuget.org, or you will be asked to specify the location of the driver if https://www.nuget.org cannot be connected.

- Informix 12.x or 14 (Beta feature) *

  PowerBuilder and/or PowerServer will automatically download the required driver (IBM.Data.DB2.Core 2.2.0.100) from https://www.nuget.org, or you will be asked to specify the location of the driver if https://www.nuget.org cannot be connected.

  * Beta means the feature has not been fully tested, has known bugs, and does not receive standard technical support. We will collect reported bugs and try to address in a future version.

SQL Anywhere and ASE databases can be connected using the ODBC driver only. The other databases are connected using the native database driver.

2.6 Network

Same as any other web applications, for installable cloud apps, the Web APIs must be published to a PowerServer that locates on the same LAN as the database server. If the database is not on the same network as the Web APIs, every request has to go a long way from PowerServer to the database, it is highly possible that there will be performance and security issues.

Quick Start

Contents

1 Overview

PowerBuilder 2021 introduces a new project type: PowerServer. With the PowerServer project type, PowerBuilder applications can be deployed as installable cloud applications.

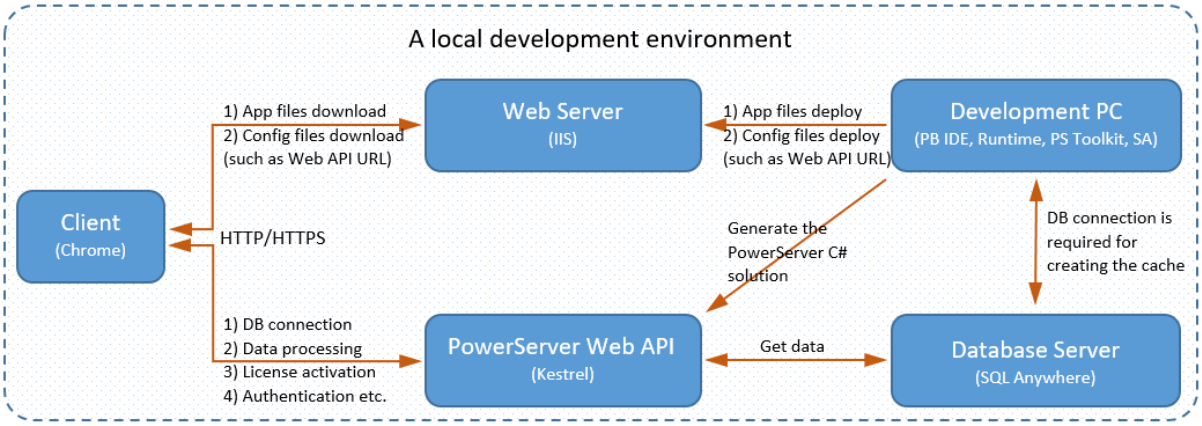
This tutorial helps you to quickly get started with PowerServer. By going through this tutorial, you will get a basic understanding of the key tasks required for deploying a PowerServer project based on the Example Sales App (SalesDemo) provided in the PowerBuilder Installer.

2 Preparing a local development environment

In order to quickly get started with PowerServer, we will use a local development machine for all roles (development, client, Web server, .NET server, and database server).

Therefore, "a local development environment" in this Quick Start guide does not mean the development PC only; it means all roles in one machine, as illustrated in the following graph. And it can only represent one supported environment (not all), for example, the IIS web server is used as an example here (although Apache and Nginx web servers are also supported), the SQL Anywhere database is used as an example (although PostgreSQL, SQL Server, Oracle etc. are also supported), Chrome is used as an example (although Firefox and Edge are also supported).

Figure 2.1:



The following steps will guide you through preparing such an environment.

Step 1: Prepare a Windows 10 (64-bit) machine.

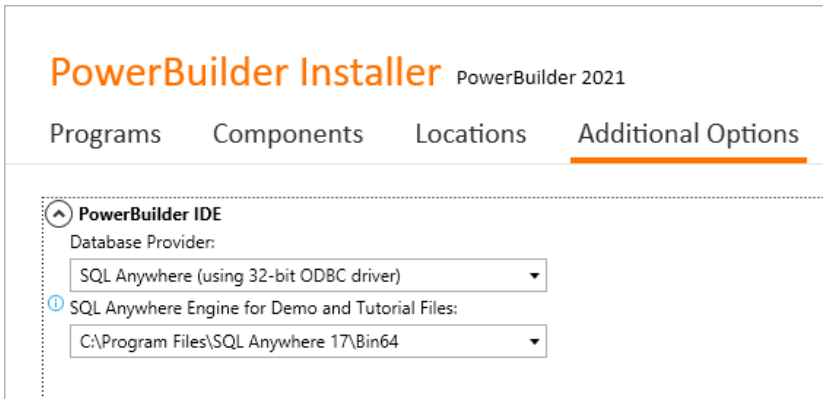
PowerBuilder IDE can only run on Windows 10 or 8.1.

Step 2: Make sure this machine has Internet connection.

Step 3: Install the following software to this machine.

Table 2.1:

Role	Requires the following software
Client	• **Install Google Chrome** The installable cloud app must run through Google Chrome, Mozilla Firefox, or Microsoft Edge (Chromium-based) for the first time.
Database Server Development	• **Install SQL Anywhere 17 (or PostgreSQL 11.3, 12, or 13)** You can download the installer for the free trial of SQL Anywhere developer edition (or the installer for PostgreSQL). If you install the PostgreSQL demo database, the steps are the same as using the SQL Anywhere demo database.
Development	Download the PowerBuilder Installer executable from the Downloads page on the Appeon User Center (login is required) and then run the PowerBuilder Installer to install the following programs or components:

Role	Requires the following software
	• **PowerBuilder Runtime 2021** • **PowerServer Toolkit 2021** • **PowerBuilder IDE 2021** During the PowerBuilder IDE installation, double check that the SQL Anywhere engine (or PostgreSQL engine) is already installed and selected in the following screen; this will automatically install the demo database according to the selected engine and create the ODBC data source required for running the PowerBuilder demo application. **Figure 2.2:**  The demo database file is automatically installed to %Public%\Documents \Appeon\PowerBuilder 21.0\ and the corresponding ODBC data source is automatically created during the PowerBuilder installation. • If SQL Anywhere engine is installed and selected, the demo database file is **pbdemo2021.db** and the ODBC data source is **PB Demo DB V2021**. • If PostgreSQL engine is installed and selected, the demo database file is **pbpostgres2021.dmp** and the ODBC data source is **PB Postgres V2021**. Alternatively, you can download the database file from https://github.com/Appeon/PowerBuilder-Project-Example-Database and create the ODBC data source manually (instructions are provided here).
Web Server	• **Install Windows IIS** Follow the next section Installing IIS in Windows 10 to install and verify IIS. Windows IIS will be used as the Web server in this tutorial to host the client-side of the installable cloud app. You can also use Windows/Linux Apache and Windows/Linux Nginx (instructions are provided here).
.NET Server	• Nothing needs to be installed

Role	Requires the following software
	In the development environment, we will directly run the PowerServer Web APIs on the ASP.NET Core Kestrel web server (a light-weight web server automatically included and enabled in the ASP.NET Core project); and as Kestrel is by default included in the PowerServer Web APIs, there is no need to install any other software. Alternatively, you can publish PowerServer Web APIs to a dedicated hosting environment such as Docker, IIS etc. (as described in tutorial 2 and tutorial 3).

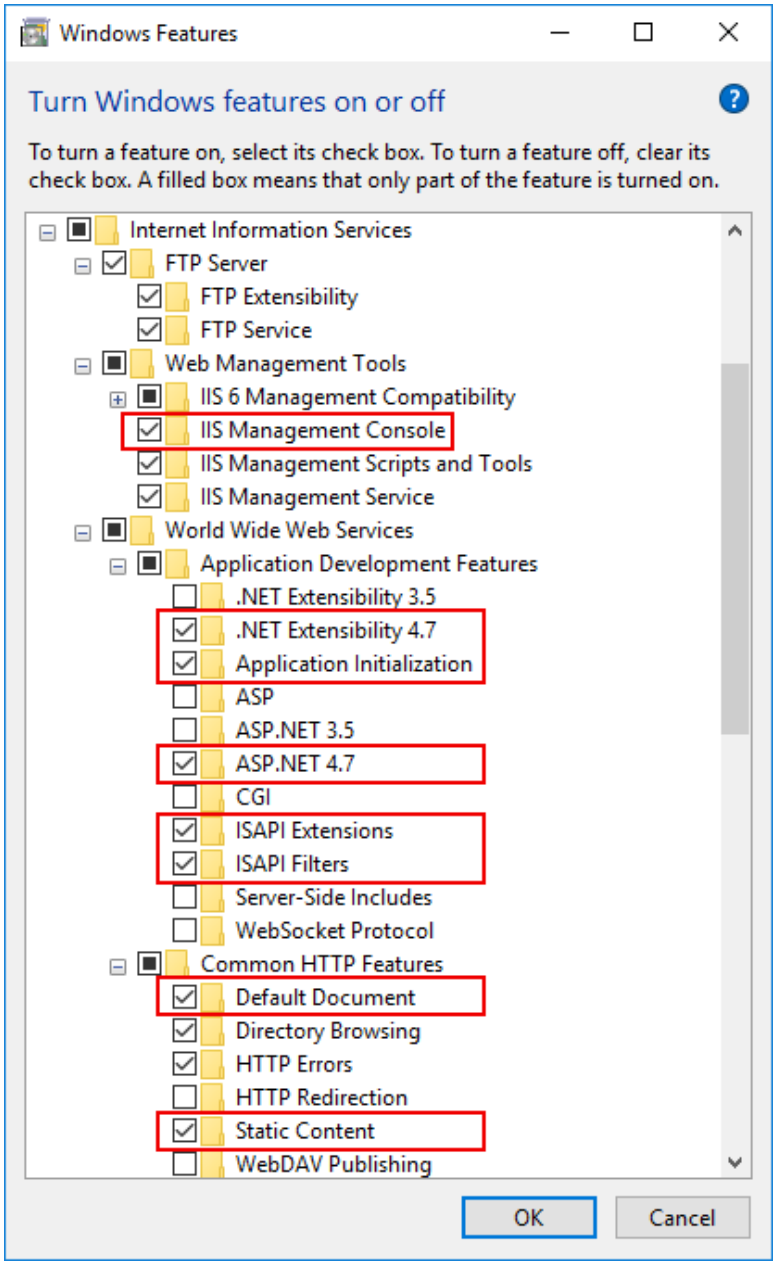
2.1 Installing IIS in Windows 10

Step 1: In Windows 10, navigate to Control Panel > Programs > Programs and Features > Turn Windows features on or off.

Step 2: Expand the **Internet Information Services** node and make sure the following features are selected.

- IIS Management Console
- .NET Extensibility 4.7
- Application Initialization
- ASP.NET 4.7
- ISAPI Extensions
- ISAPI Filters
- Default Document
- Static Content

Figure 2.3:

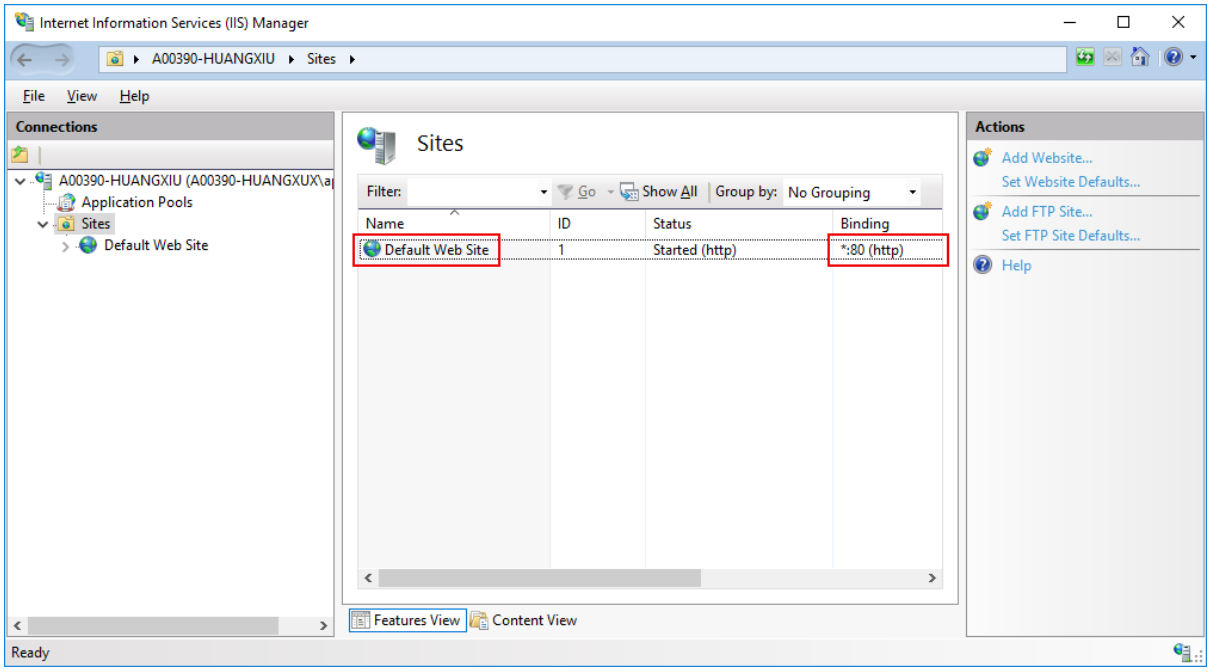


Step 3: Click **OK** to install the selected features.

After IIS is installed, a **Default Web Site** (with port 80) is automatically created.

We will use the **Default Web Site** (with port 80) in this tutorial. You can also create new websites with different port numbers (instructions are provided here).

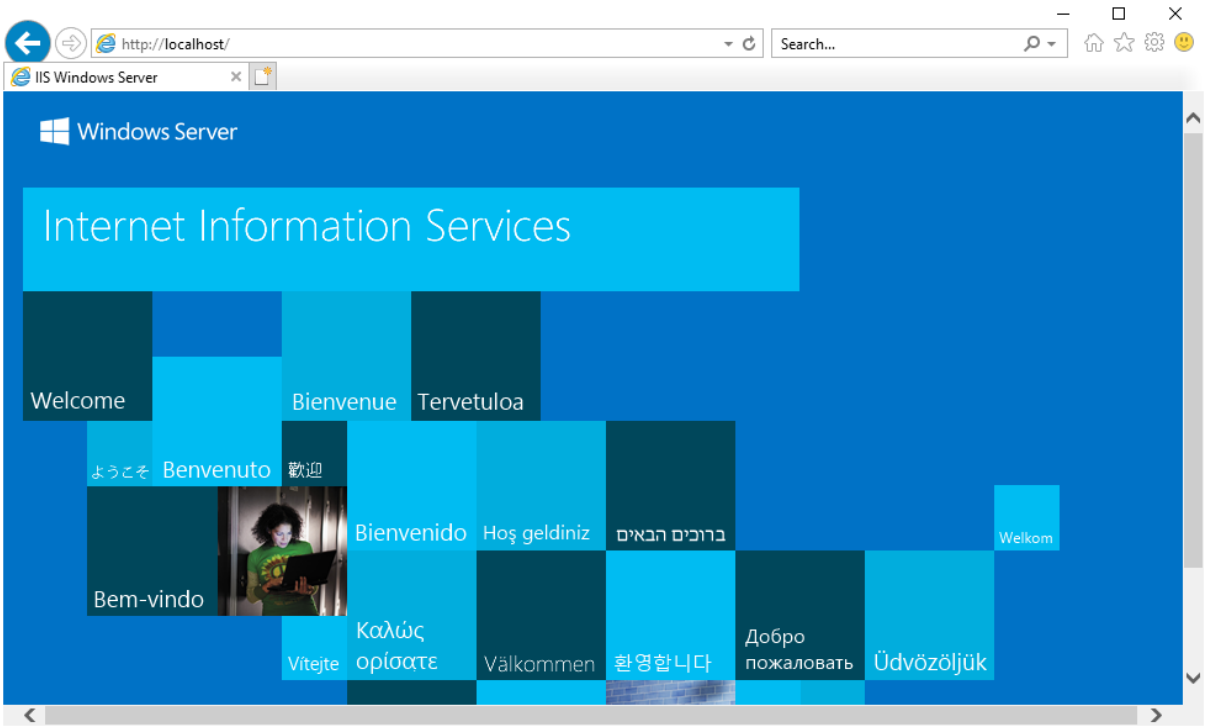
Figure 2.4:



Step 4: Open a Web browser and input "http://localhost:80/" in the address bar.

If the IIS welcome screen displays, the **Default Web Site** is working properly.

Figure 2.5:



3 Verifying the example sales app

Step 1: Select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**. The SalesDemo workspace is loaded in the PowerBuilder IDE.

Note: Run as administrator is recommended as administrator rights are required when performing some tasks later (such as uploading files to server).

Step 2: Click the **Run** button in the PowerBuilder toolbar and make sure the application can run and data can be retrieved successfully. Close the application after verifying it.

4 Minimal efforts: Deploying the sample PowerServer project

The Example Sales App (SalesDemo) contains a sample PowerServer project in the salesdemo.pbl: salesdemo_cloud. Following the instructions in this chapter, and using the sample project, you can get the application deployed to PowerServer and then run as an installable cloud app in a few steps. Alternatively, you can follow the instructions in the chapter Creating and deploying a new PowerServer project to try the full steps of creating and then deploying a PowerServer project from the very beginning.

4.1 Updating the sample PowerServer project based on your environment

Step 1: Open the sample PowerServer project in the painter.

1. Load the SalesDemo workspace in the PowerBuilder IDE by selecting Windows **Start | Appeon PowerBuilder 2021**, and then right-clicking **Example Sales App** and selecting **More | Run as administrator**.

2. Locate the **salesdemo_cloud** project file in **salesdemo.pbl**, and double click to open it in the painter.

   The project file contains multiple tabs: General, Libraries, External Files, Runtime, Signing, Client Deployment, Run Options, and Web APIs. Most settings in the tabs are pre-configured and can stay as-is. You only need to follow the instructions in the subsequent steps to adjust a few settings based on your environment.

Step 2: Update the server configuration with the following steps:

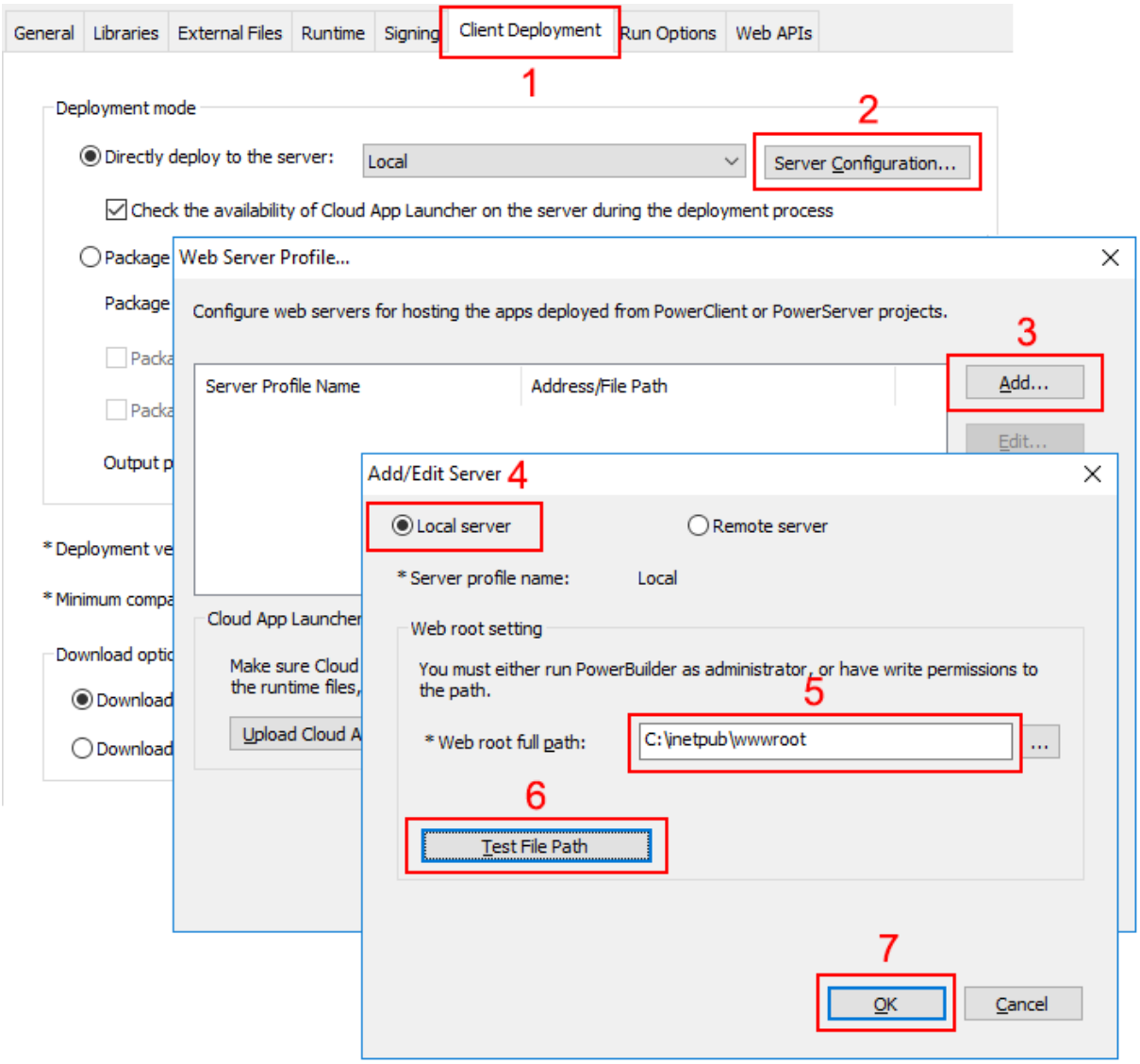
1. Click the **Client Deployment** tab in the PowerServer project painter.

2. In the **Deployment mode** section, click the **Server Configuration** button. In the **Web Server Profile** window that appears, click the **Add** button.

3. In the **Add/Edit Server** window, select **Local server**, set the **Web root full path** (in this tutorial, C:\inetpub\wwwroot), and then click **Test File Path** to ensure the path is valid.

   This tutorial assumes your OS is installed to the C drive and the IIS Web root is C:\inetpub\wwwroot. If you encounter any errors when configuring the Web server profile, refer to Permission errors when configuring the Web server profile.

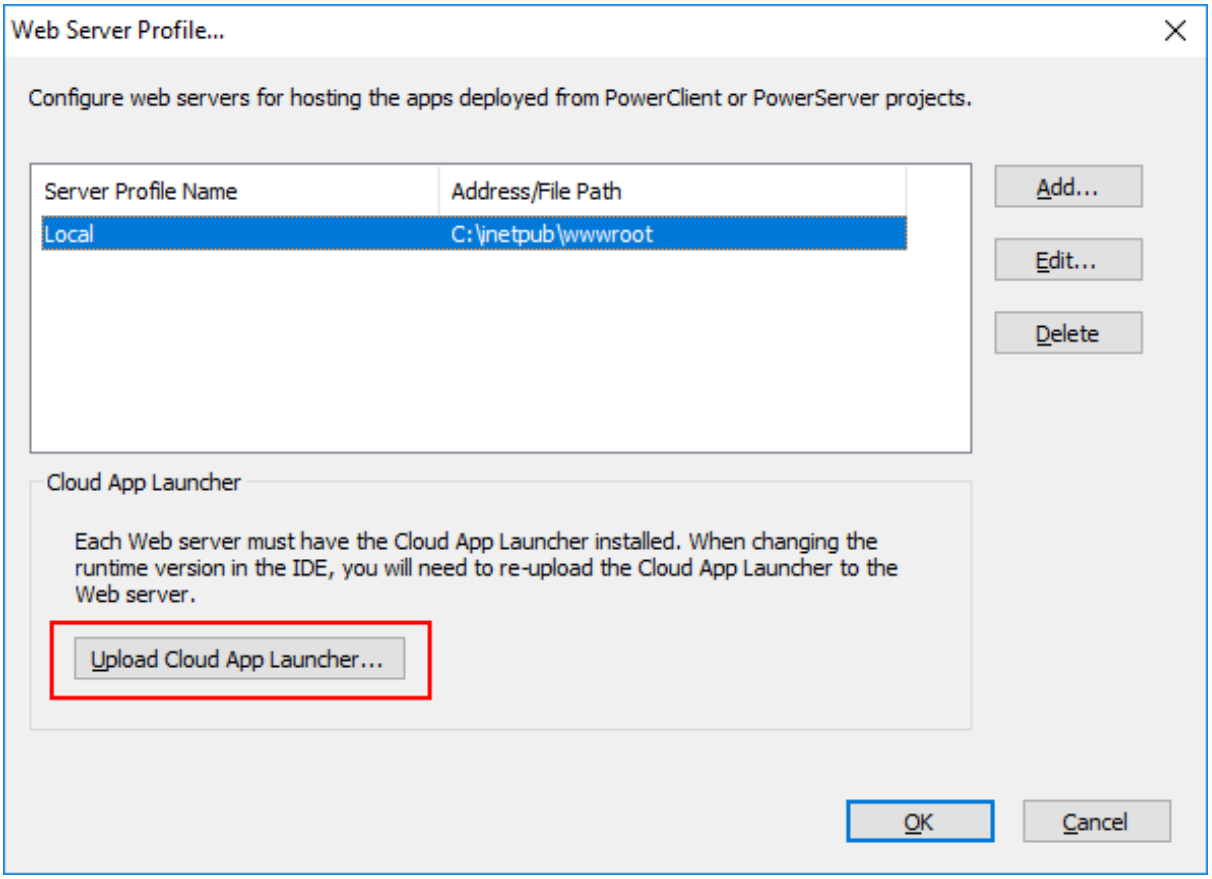
Figure 4.1:



4. Click **OK** to save the server profile and return to the **Web Server Profile** window.

5. Click the **Upload Cloud App Launcher** button.

Figure 4.2:



6. In the **Upload Cloud App Launcher and Runtime** window that appears, make sure the following are selected: **Local**, **Upload the runtime files for the apps**, **32-bit**, and **64-bit**.

7. Click **Upload** and make sure the upload is successful. This section Uploading the cloud app launcher and the runtime files has more details about this window.

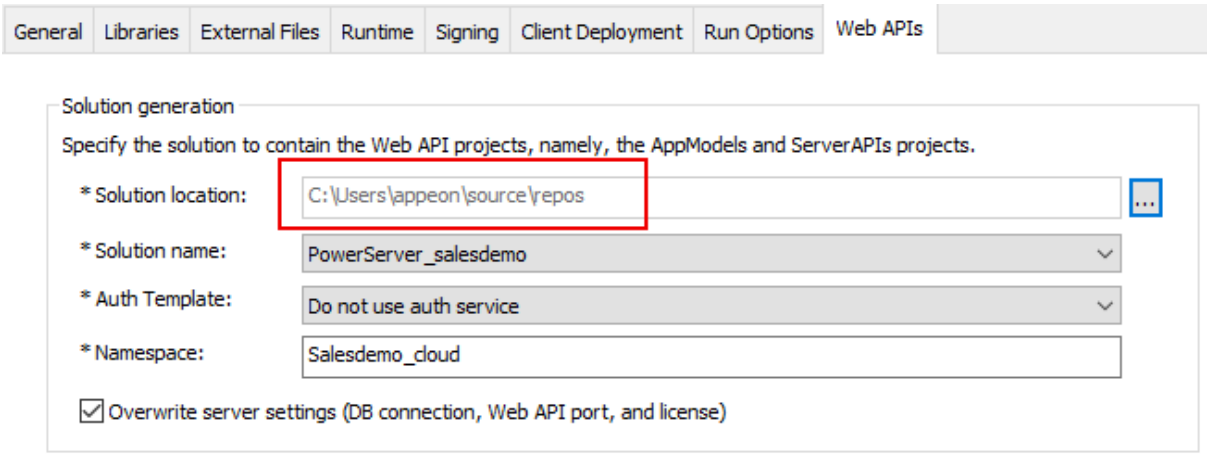
Step 3: Update the PowerServer solution path with the following steps:

1. Click **Web APIs** in the PowerServer project painter.

   In the **Solution location** field, the default location is set to [current user]\source\repos.

2. If the location in the **Solution location** does not exist on the current machine, please select a valid one.

Figure 4.3:



Step 4: Use the default Web API URL "http://localhost:5000" in the **Web APIs** tab. Make sure the port number is not occupied by another program.

In this tutorial, the Web APIs will be running on the local computer, in order to quickly get started and running.

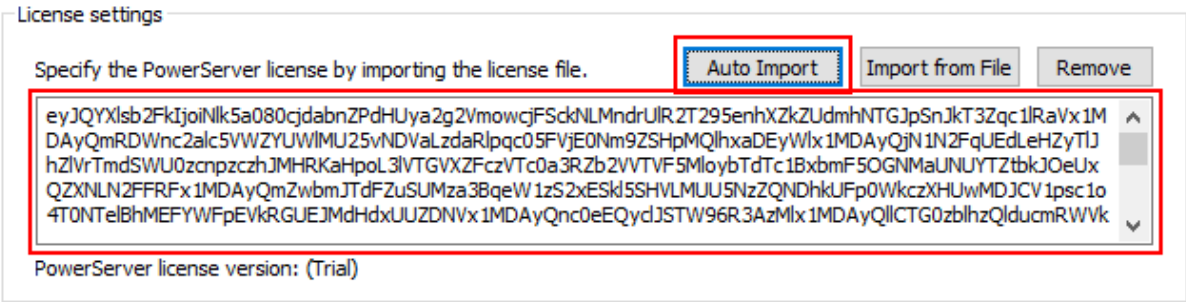
If you plan to apply a web debugging proxy tool to debug the deployed application or want to publish the PowerServer Web APIs to a dedicated server, then use the actual IP address.

Figure 4.4:

Step 5: Import a valid PowerServer license in the **Web APIs** tab.

You can import a valid license into the project settings using **Auto Import** (importing the current PowerBuilder CloudPro or trial license), or **Import from File** (file from the License Management page on https://account.appeon.com).

Figure 4.5:



Step 6: Double check the Database Configuration in the **Web APIs** tab.

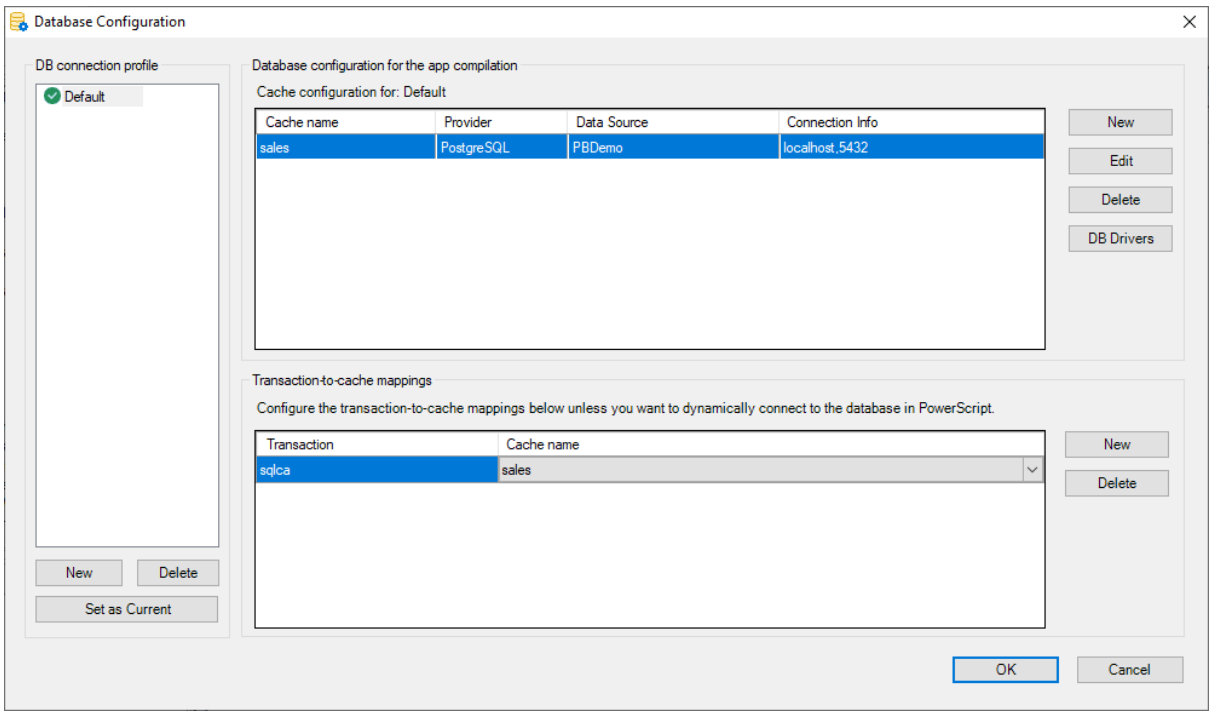
1. Click the **Database Configuration** button in the **Web APIs** tab in the PowerServer project painter.

2. Click **DB Drivers** in the upper part to make sure the SQL Anywhere driver (or PostgreSQL driver) and the option "I have read and agree to the license ..." both are selected.

3. Select the "Sales" cache and then click the **Edit** button besides the selected cache name.

   If you use SQL Anywhere as the demo database, no change is needed to the database configuration. If you use PostgreSQL as the demo database, the default login account is postgres (user)/postgres (password). Please double check the connection.

Figure 4.6:



4.2 Building and deploying the PowerServer project

Step 1: Click the **Save** button ( ) in the toolbar.

Step 2: Click the **Build & Deploy PowerServer Project** button ( ) in the toolbar to build and deploy the project.

4.3 Starting the Web APIs

Step 1: Make sure your computer can connect to the NuGet site (https://www.nuget.org).

The packages required for compiling and running the Web APIs must be downloaded from the NuGet site first.

Step 2: Click the **Compile & Run Web APIs** button ( ) in the toolbar to compile and run the Web APIs on the local computer.

This will run the Web APIs directly on Kestrel (a light-weight web server included and enabled automatically in every ASP.NET Core project).

To deploy Web APIs to a dedicated hosting environment such as Docker or IIS, refer to Tutorial 2: Hosting Web APIs in Docker Containers and Tutorial 3: Hosting Web APIs in IIS.

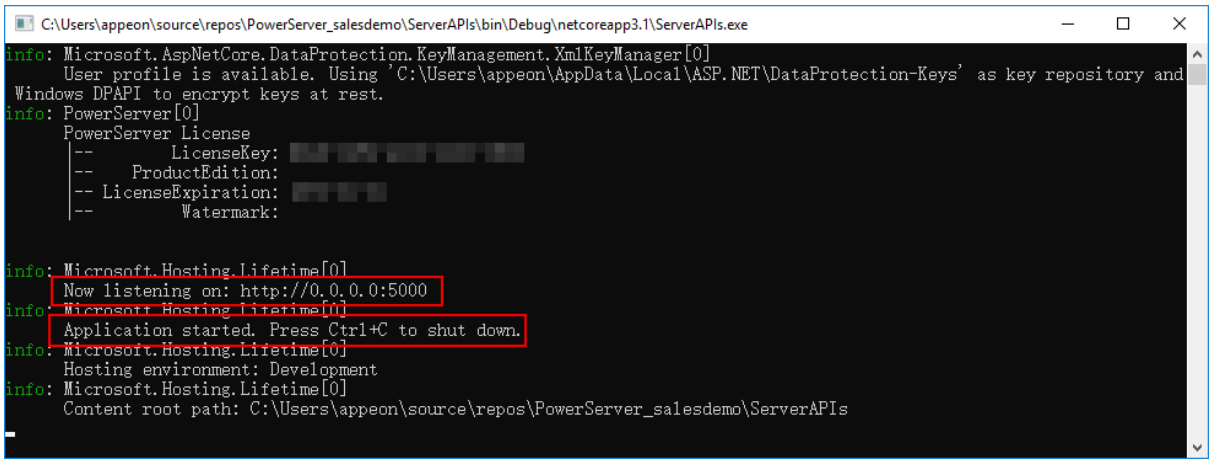
Step 3: Check the Output window and make sure build is successful.

Step 4: Make sure the API console window displays "Application started...".

Also notice "Now listening on: http://0.0.0.0:5000" in the console window. This is the URL for accessing the Web APIs. You can use "localhost" or the IP address to access the Web APIs running on the local computer. The port number can be modified in the **launchSettings.json** in the PowerServer C# solution and will take effect in the development environment.

When the installable cloud application is run later, you can view the logs in the console window to check if the requests and responses are processed successfully.

Figure 4.7:



4.4 Running the installable cloud application

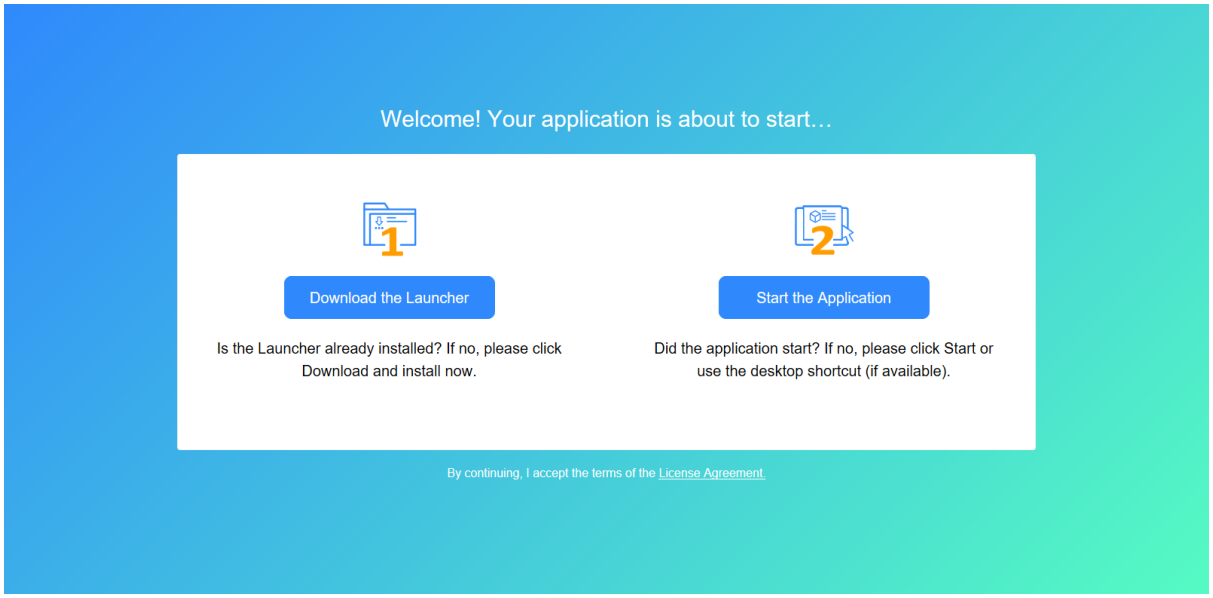
Step 1: Click the **Run PowerServer Project** button ( ) in the toolbar to run the application.

For more information about running the application, refer to Run the installable cloud application.

Step 2: In the app entry page that appears, click **Download the Launcher** to download and install the launcher.

After the launcher is installed, the application should automatically start, if not, click **Start the Application** in the entry page to start the application.

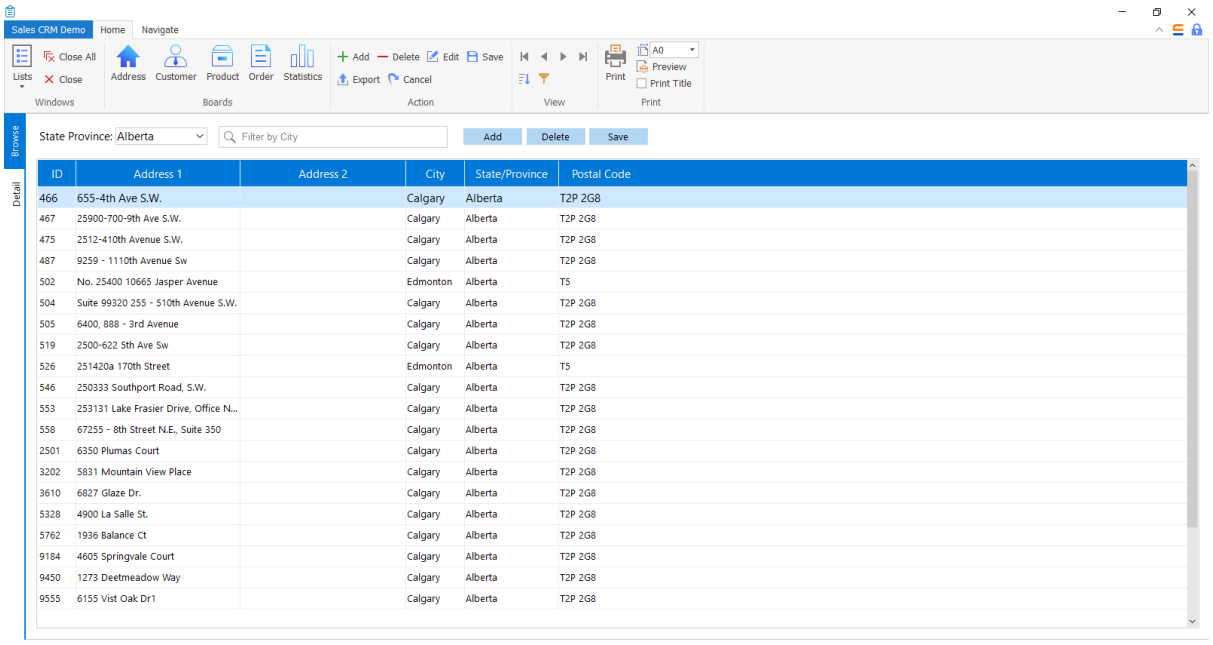
Figure 4.8:



Step 3: When the application main window displays, click the **Address** button in the application toolbar. Data should be successfully displayed.

You can view the logs in the API console window to check if the Web API requests and responses are successful.

Figure 4.9:



5 Full experience: Creating and deploying a new PowerServer project

This chapter guides you to try the full steps of creating, deploying and then running a PowerServer project (Example Sales App) from the very beginning. During the process, you can get a better understanding of each setting in the PowerServer project. Alternatively, if you hope to have a really quick experience on deploying PowerServer projects, you may start with the sample PowerServer project (salesdemo_cloud) provided in the salesdemo.pbl. For more information, see Deploying the sample PowerServer project.

5.1 Creating the PowerServer project

5.1.1 Creating a new PowerServer project

Step 1: In the PowerBuilder System Tree view, right click the **SalesDemo** workspace and select **New**. In the **New** dialog, select the **Project** tab and then select **PowerServer**.

The PowerServer project painter is opened.

Configure the PowerServer project painter according to the instructions below. Some tab pages that do not need to be configured in this tutorial will be skipped directly. For detailed information on how to use each tab page, refer to the How-to guides.

5.1.2 Configuring the General tab

Step 1: On the **General** tab, input "salesdemo_cloud_new" in **App name** as the application name.

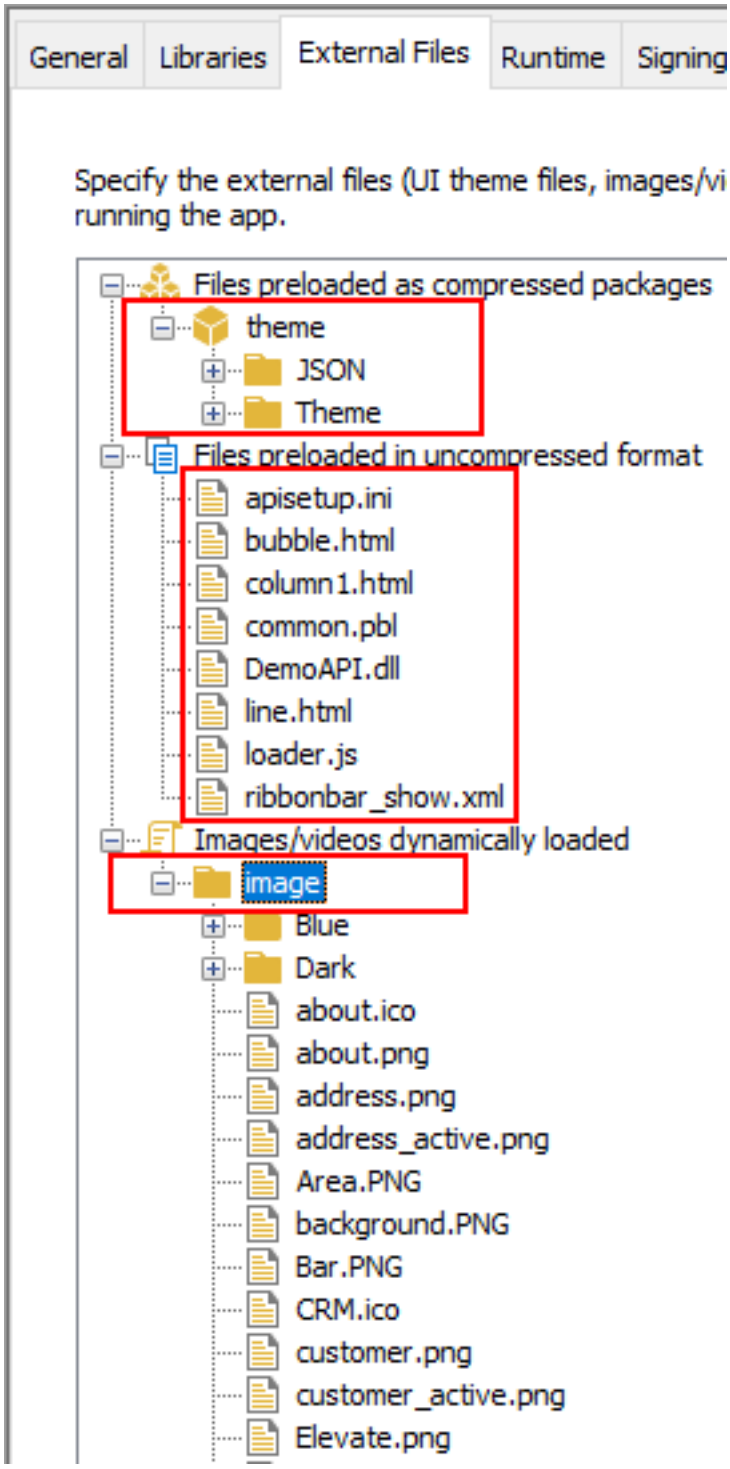
5.1.3 Configuring the External Files tab

Step 1: On the **External Files** tab, select **Files preloaded as compressed packages** and then click **Create Package**. Input a package name (for example "theme"). Then select this package and click **Add Folder** to add these two folders one by one: **JSON** and **Theme**, as shown in the figure below.

Step 2: Select **Files preloaded in uncompressed format** and then click **Add Files** to add the files as shown in the figure below.

Step 3: Select **Images/videos dynamically loaded** and then click **Add Folder** to add the following folder: **image**.

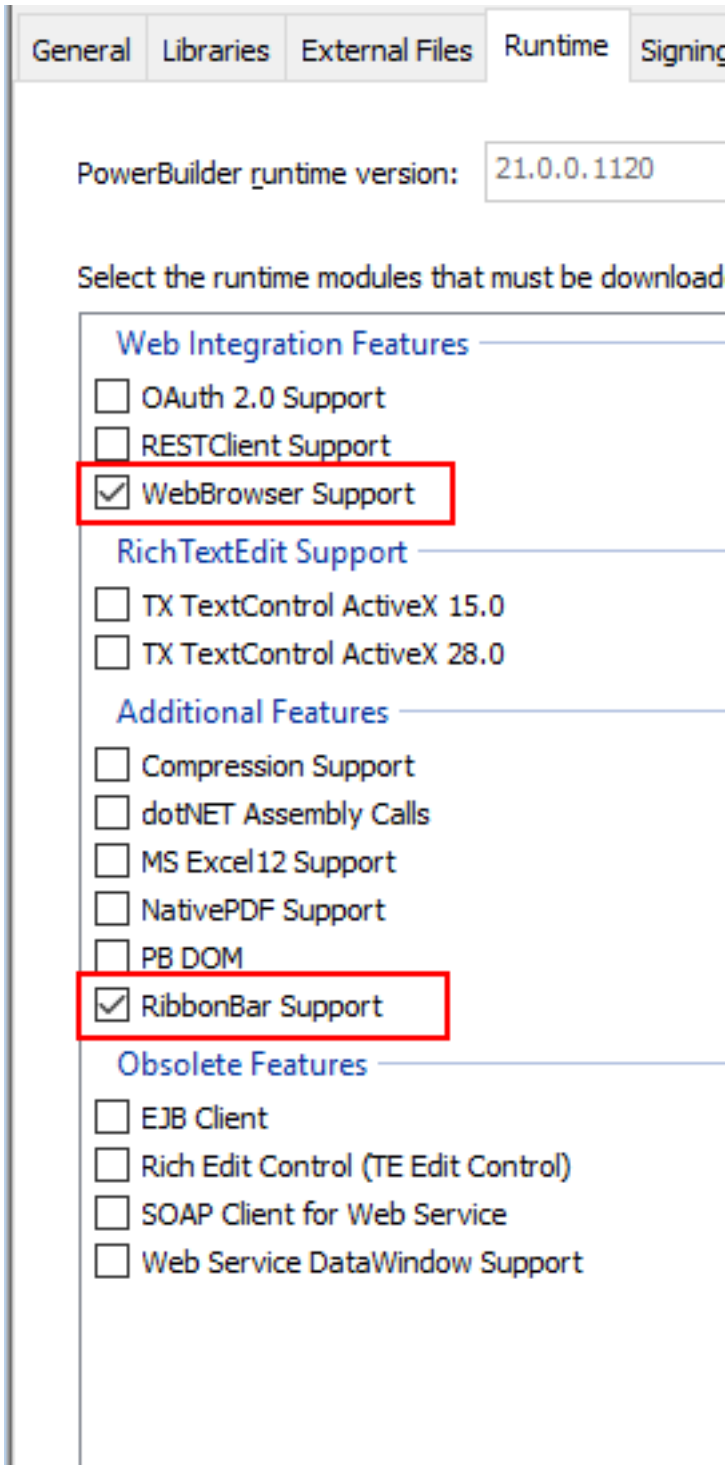
Figure 5.1:



5.1.4 Configuring the Runtime tab

Step 1: On the **Runtime** tab, select **WebBrowser Support** and **RibbonBar Support**.

Figure 5.2:



5.1.5 Configuring the Client Deployment tab

Step 1: Select the **Client Deployment** tab in the PowerServer project painter.

Step 2: In the **Deployment mode** section, click the **Server Configuration** button.

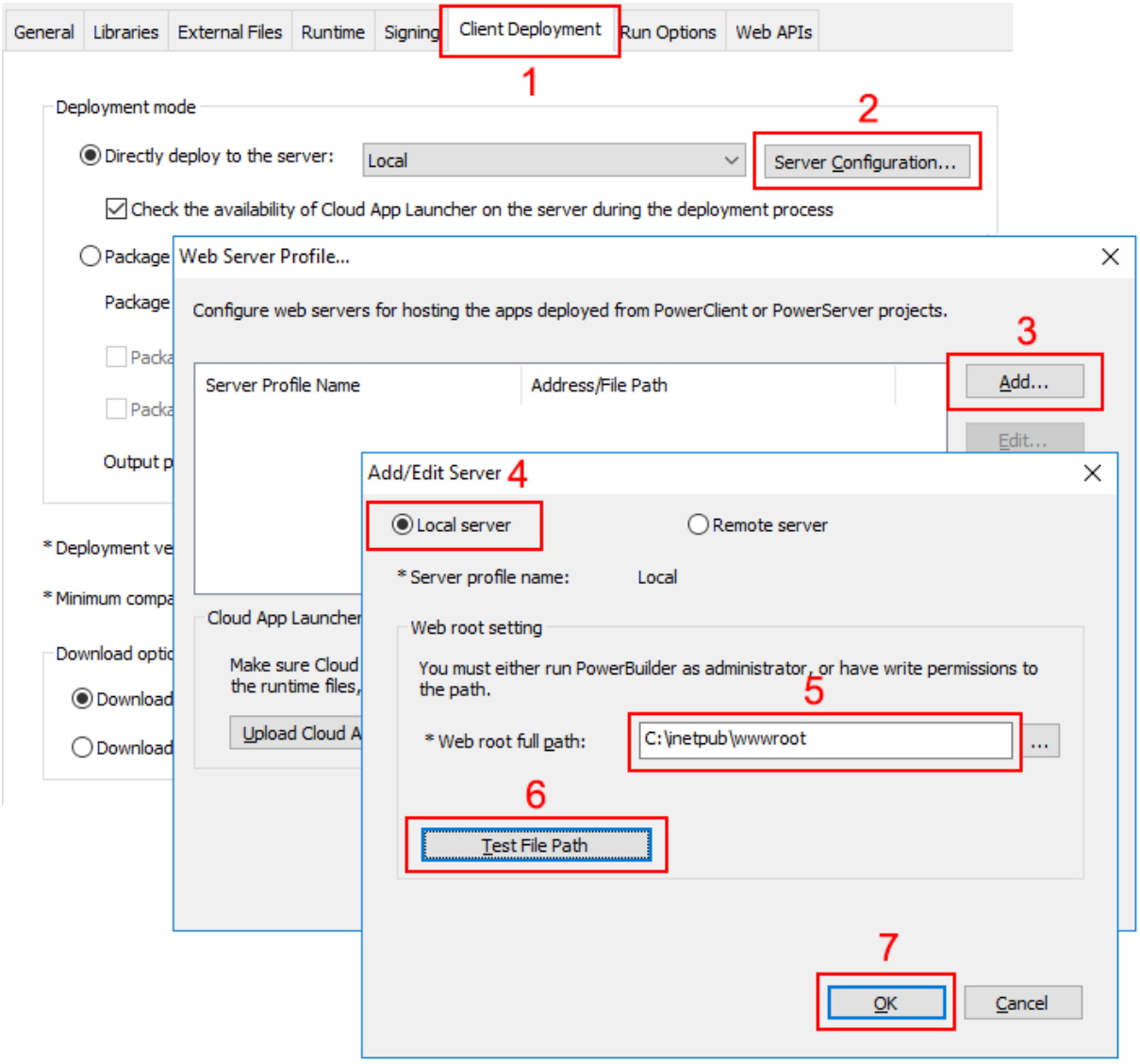
Step 3: In the **Web Server Profile** window that appears, click the **Add** button.

Step 4: In the **Add/Edit Server** window, select **Local server**, set the **Web root full path** (in this tutorial, C:\inetpub\wwwroot), and then click **Test File Path** to ensure the path is valid.

This tutorial assumes your OS is installed to the C drive and the IIS Web root is C:\inetpub \wwwroot.

If you encounter any errors when configuring the Web server profile, refer to Permission errors when configuring the Web server profile.

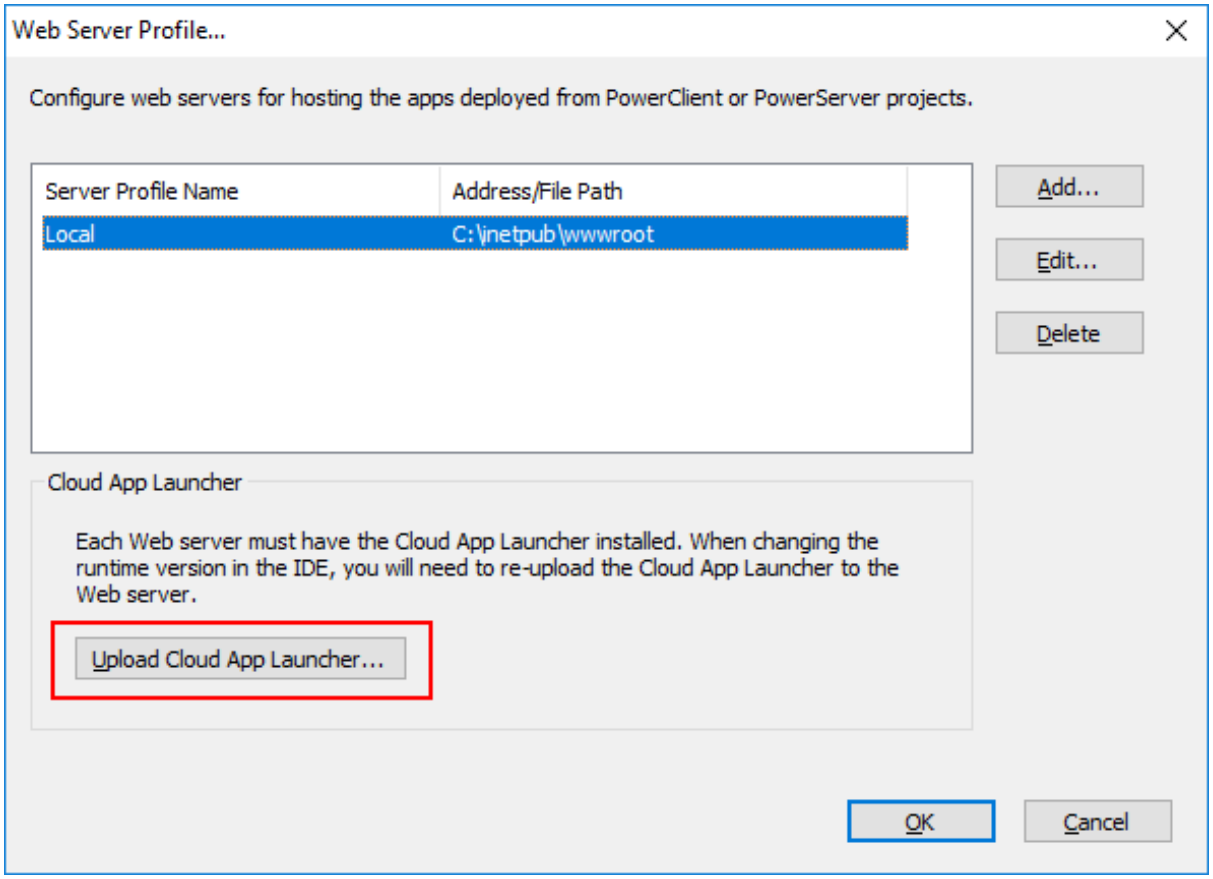
Figure 5.3:



Step 5: Click **OK** to save the server profile and return to the **Web Server Profile** window.

Step 6: Click the **Upload Cloud App Launcher** button.

Figure 5.4:

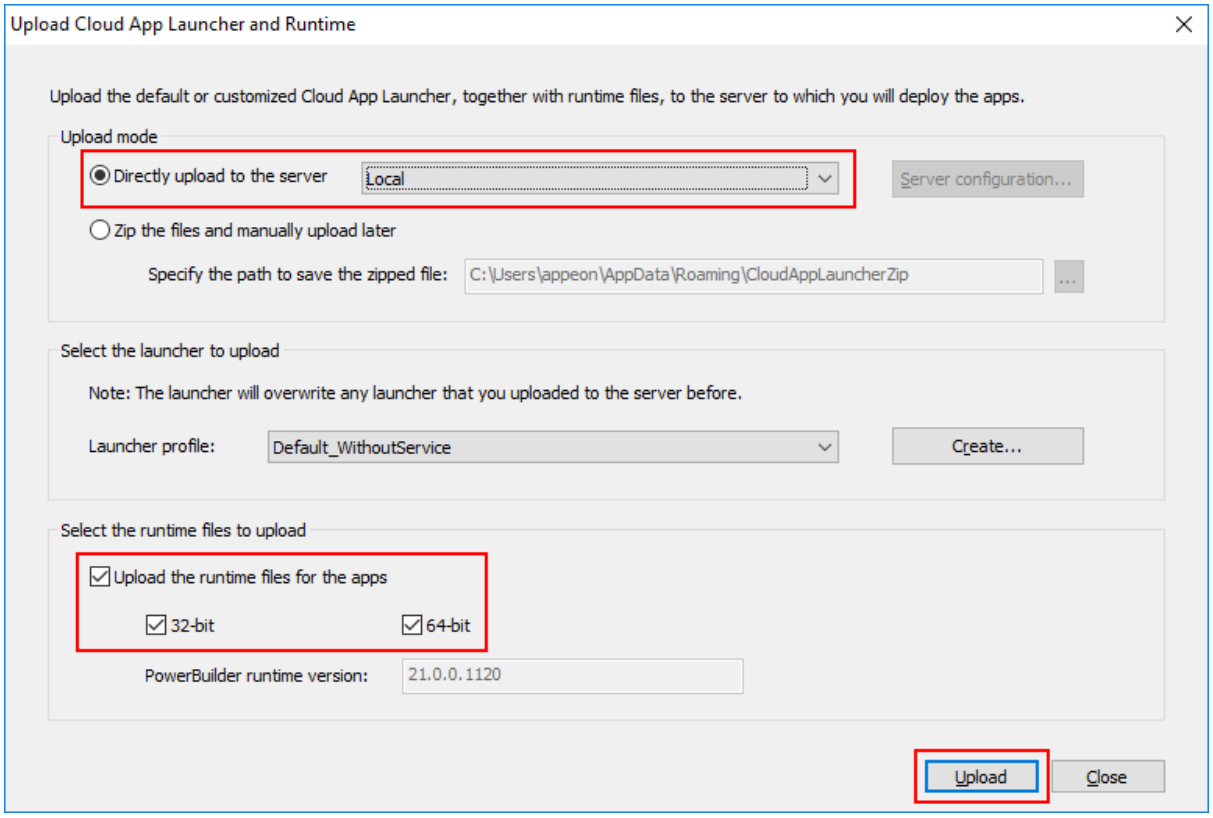


Step 7: In the **Upload Cloud App Launcher and Runtime** window that appears, select **Directly upload to the server** and **Local**, and then make sure the following are selected: **Local**, **Upload the runtime files for the apps**, **32-bit**, and **64-bit**.

Step 8: Click **Upload** and make sure the upload is successful.

This section Uploading the cloud app launcher and the runtime files has more details about this window.

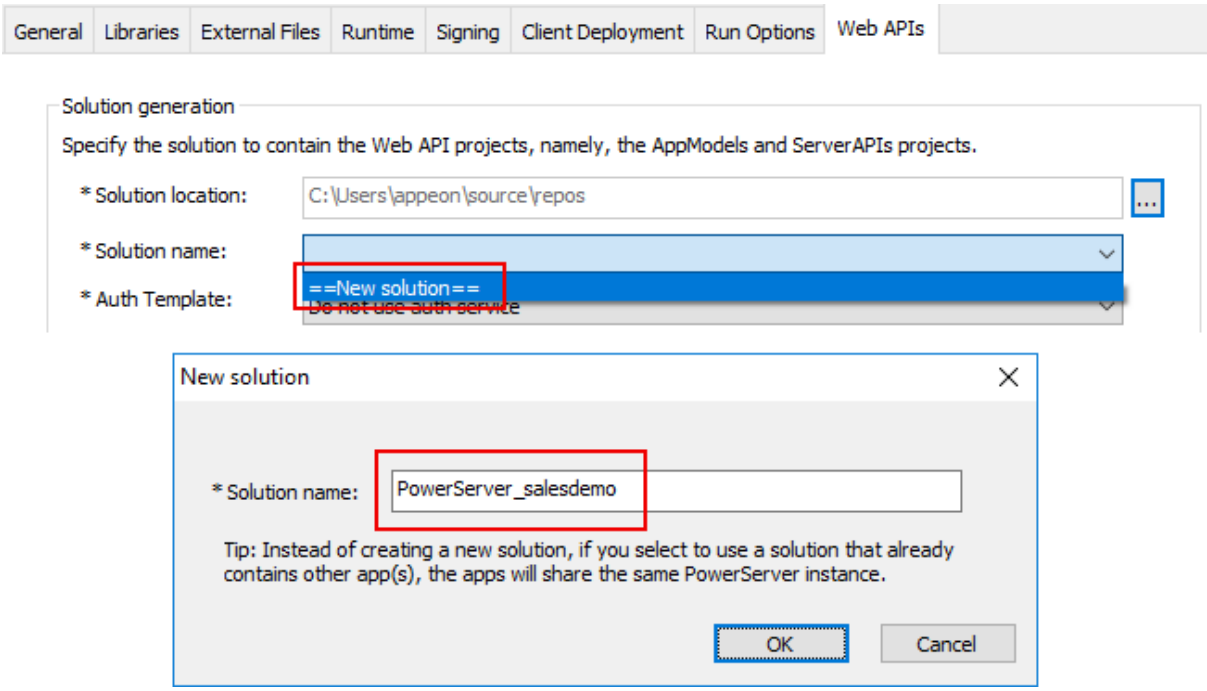
Figure 5.5:



5.1.6 Configuring the Web APIs tab

Step 1: On the **Web APIs** tab, select "New solution" from the **Solution name** list; in the **New solution** dialog, click **OK** to use the default solution name.

Figure 5.6:

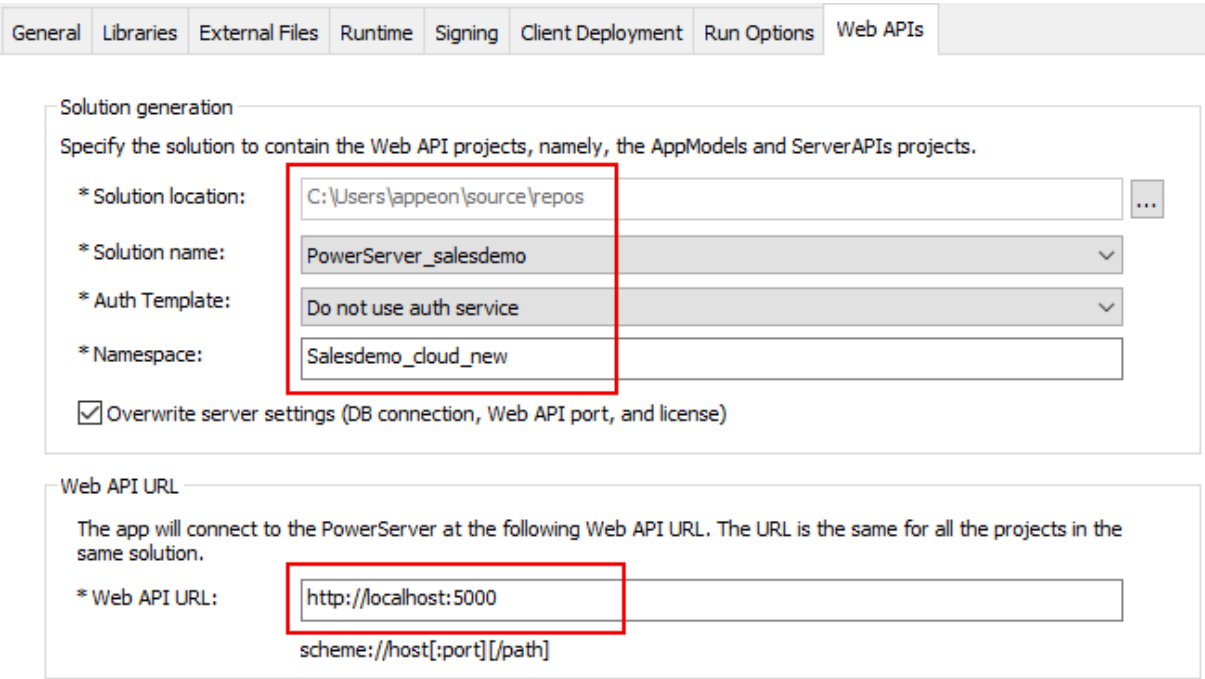


Step 2: Use the default Web API URL "http://localhost:5000". Make sure the port setting in the Web API URL is not occupied by another program.

In this tutorial, the Web APIs will be running on the local computer, in order to quickly get started and running.

If you plan to apply a web debugging proxy tool to debug the deployed application or want to publish the PowerServer Web APIs to a dedicated server, then use the actual IP address.

Figure 5.7:

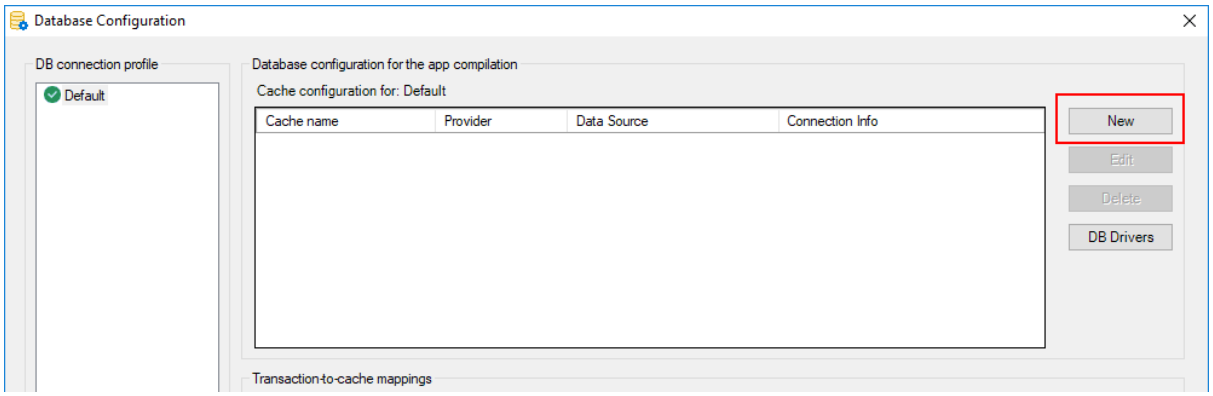


Step 3: Click the **Database Configuration** button at the bottom of the **Web APIs** tab.

Step 4: In the **Database Configuration** dialog, click **DB Drivers** in the upper part to make sure the SQL Anywhere driver (or PostgreSQL driver) and the option "I have read and agree to the license ..." both are selected.

Step 5: In the **Database Configuration** dialog, click **New** in the upper part to create the database connection that will be used by the Web APIs.

Figure 5.8:

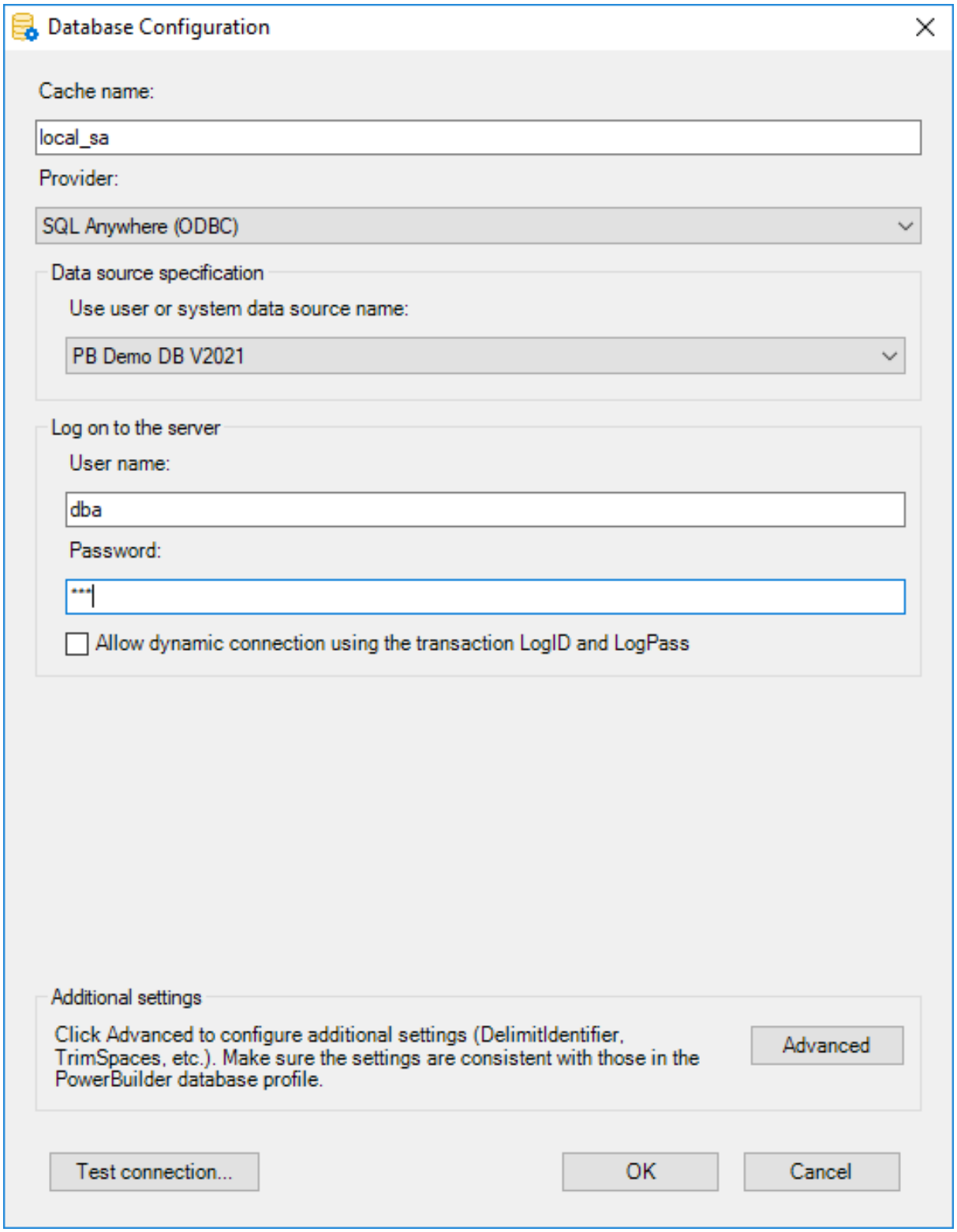


Create the database connection with the following settings:

- Specify any text (for example "local_sa") as the database cache name.

- Specify **SQL Anywhere (ODBC)** as the database provider.
- Select **PB Demo DB V2021** as the data source.
- Specify "**dba**" as the user name and "**sql**" as the password.
- Click **Test Connection** to make sure the database can be connected successfully.

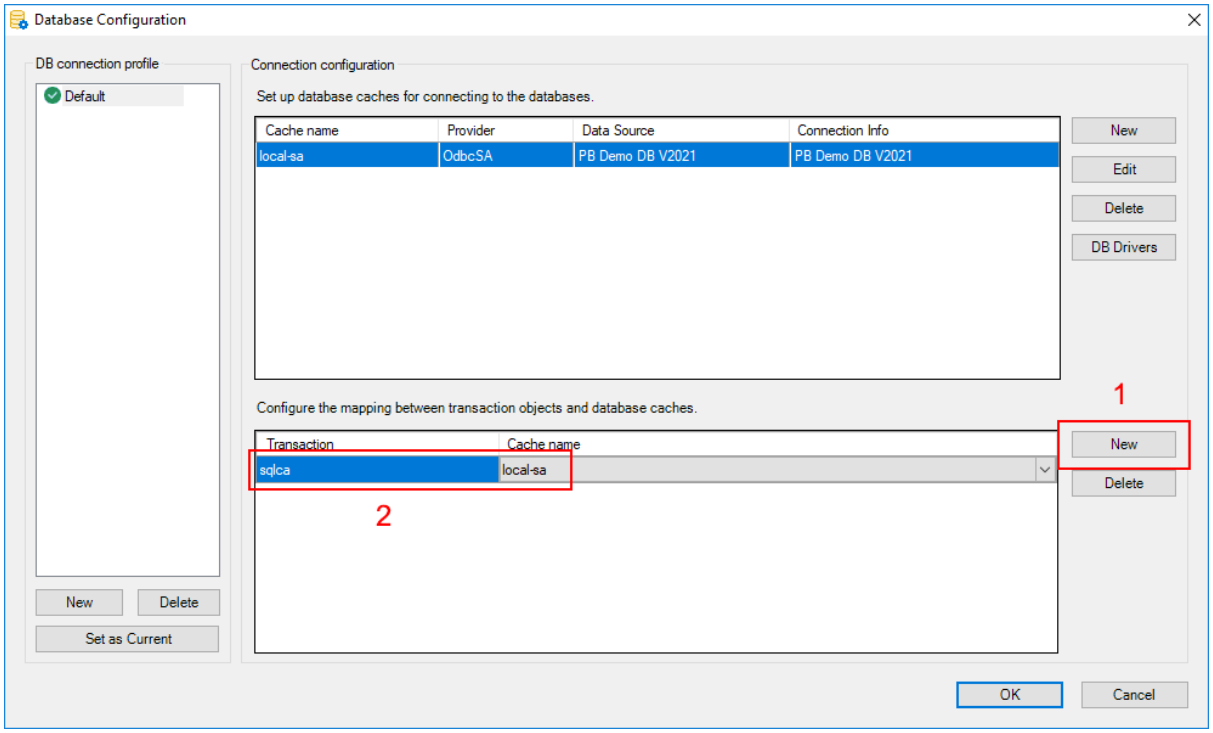
Figure 5.9:



Step 6: Click **OK** to save settings and go back to the **Database Configuration** dialog; and then click **New** in the lower part to map the transaction object with the database cache.

Step 7: Input "sqlca" as the transaction object that maps to the database cache.

Figure 5.10:



5.1.7 Importing the PowerServer license

The imported license file will be deployed along with the Web APIs project, and will be activated when the Web APIs starts.

First of all, make sure you have a valid license for PowerServer 2021 GA.

- ⊗ If you have a preview or beta license, the preview or beta license will no longer work with the GA version.

- ⊙ If you already have a PowerBuilder CloudPro license (no matter which version it is), the CloudPro license will automatically work with the GA version. Each PowerBuilder CloudPro subscription includes a developer license of PowerServer, which supports a maximum of 5 user sessions (user session = installable cloud app). You will need to purchase a production license of PowerServer in order to use the production server and more user sessions.

- ⊙ If you have no PowerBuilder CloudPro license, you can apply for a trial license at https://www.appeon.com/psfreetrial, or purchase a production license of PowerServer from https://www.appeon.com/pricing.

Once you have a valid license, you can import the license and deploy it along with the Web APIs project. The license will be activated when the Web APIs starts.

To import the license automatically:

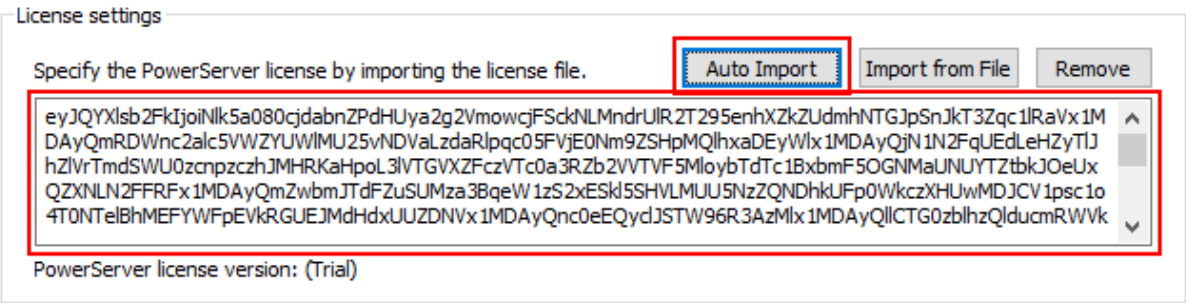
To activate PowerServer using the developer license or trial license included in the PowerBuilder CloudPro subscription, you can obtain the license automatically from the Appeon website according to the current PowerBuilder IDE login account.

1. Make sure the computer can connect to the Appeon sites (through port number 80): https://api.appeon.com and https://api2.appeon.com.

2. Go to the **Web APIs** tab of the PowerServer project painter, and then click **Auto Import** to automatically import the license.

   PowerBuilder will automatically obtain the developer or trial license of PowerServer (according to your PowerBuilder IDE login account) from the Appeon sites and then import the license here.

   Figure 5.11:

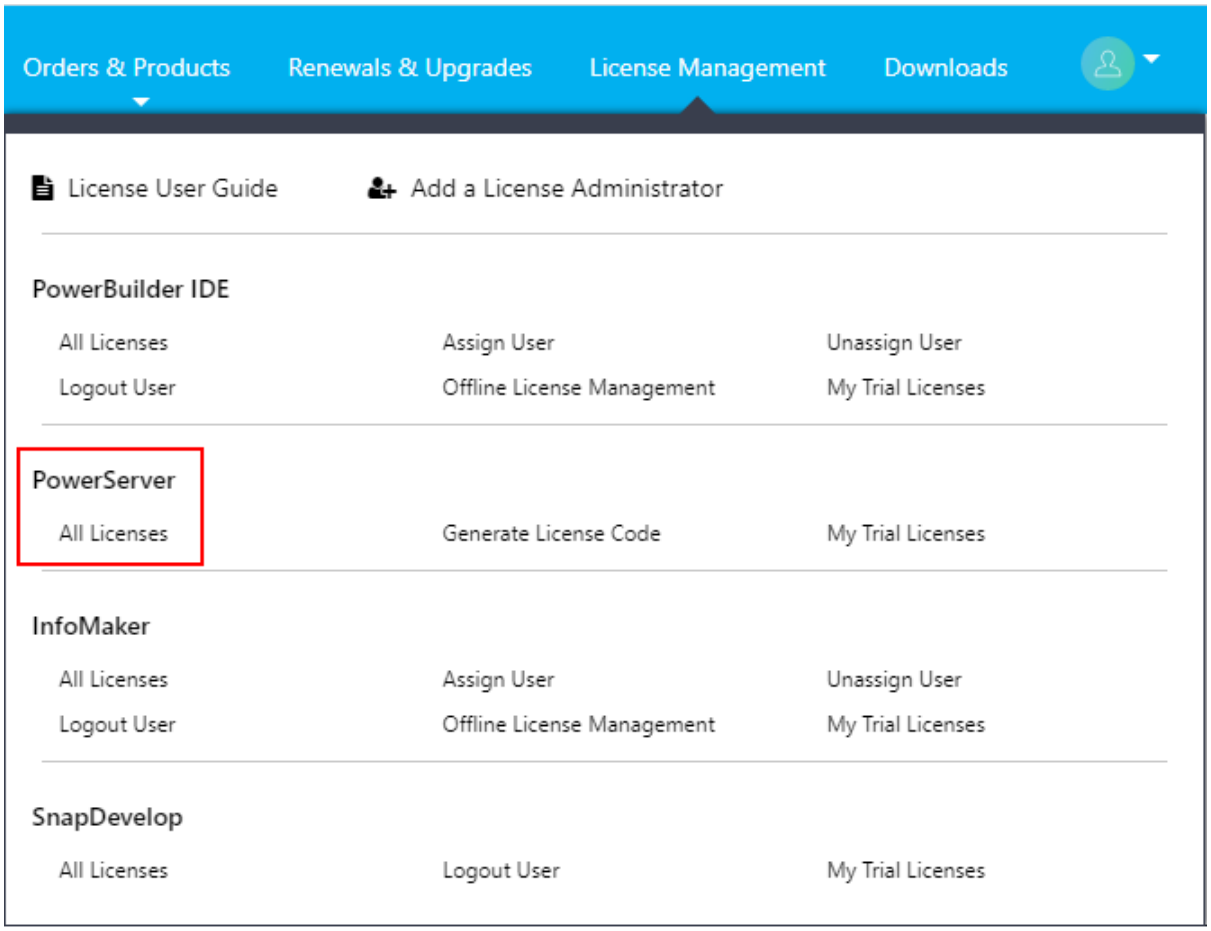


To import the license manually:

You can also export the license file from the Appeon website manually and then import the license here.

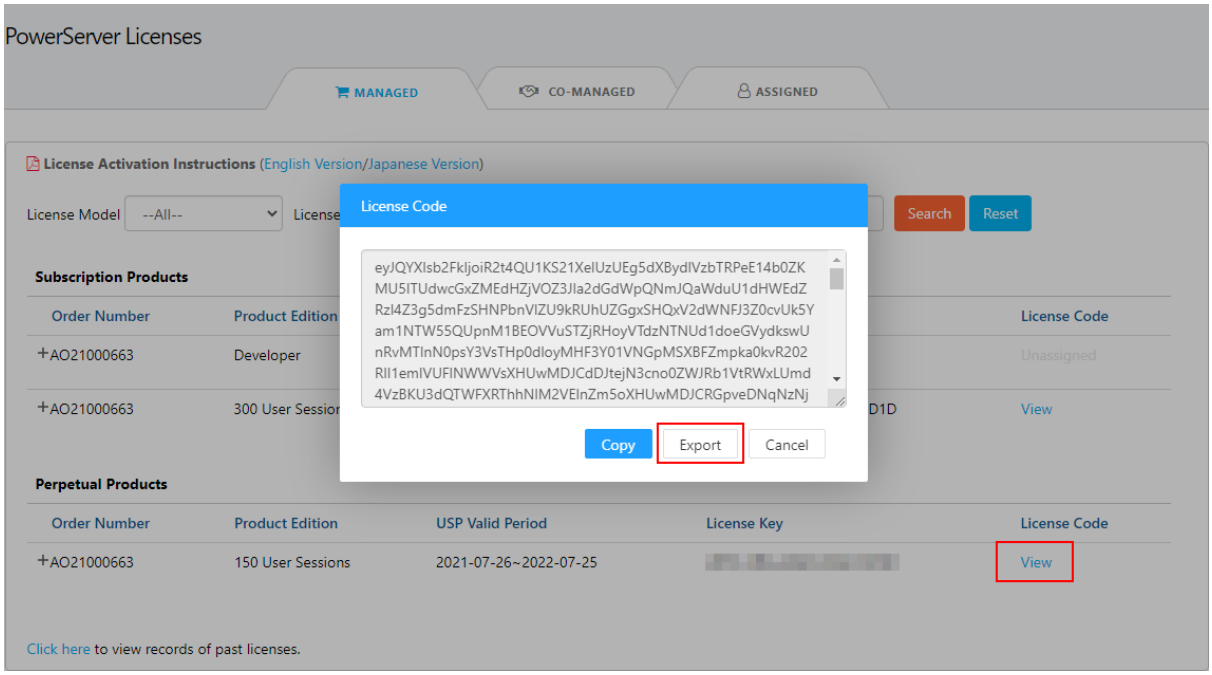
1. Log into the Appeon User Center, click **License Management**, and then click **All Licenses** under **PowerServer**.

Figure 5.12:



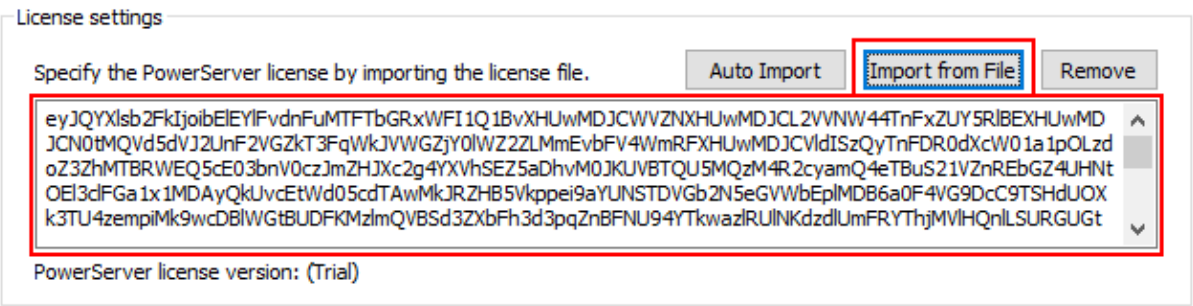
2. Click **View**, and then click **Export** to export the license code to a TXT file ([LicenseKey].txt) and save the file on the local machine.

Figure 5.13:



3. Go to the **Web APIs** tab of the PowerServer project painter, and then click **Import form File** to select and import the [LicenseKey].txt file.

Figure 5.14:



5.2 Building and deploying the PowerServer project

Step 1: Click the **Save** button ( ) in the toolbar and then enter a name for the PowerServer project object.

A PowerServer project object will be created under the specified library.

Step 2: Click the **Build & Deploy PowerServer Project** button ( ) in the toolbar to build and deploy the project.

Select **ODB ODBC | PB Demo DB V2021** from the **Database Profiles** dialog box if you are prompted to connect to a database profile.

5.3 Starting the Web APIs

Step 1: Make sure your computer can connect to the NuGet site (https://www.nuget.org).

The packages required for compiling and running the Web APIs must be downloaded from the NuGet site first.

Step 2: Click the **Compile & Run Web APIs** button ( ) in the toolbar to compile and run the Web APIs on the local computer.

This will run the Web APIs directly on Kestrel (a light-weight web server included and enabled automatically in every ASP.NET Core project).

To deploy Web APIs to a dedicated hosting environment such as Docker or IIS, refer to Tutorial 2: Hosting Web APIs in Docker Containers and Tutorial 3: Hosting Web APIs in IIS.

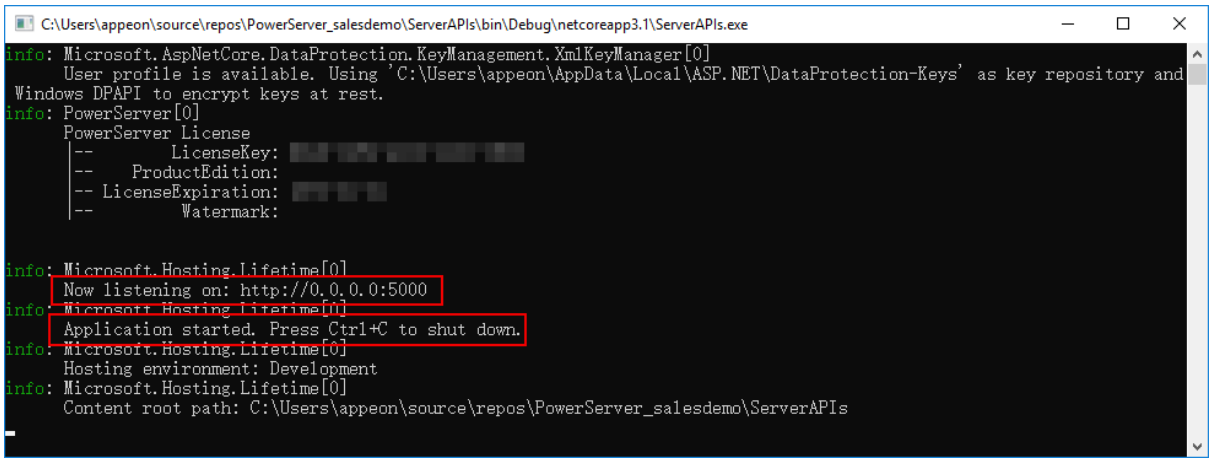
Step 3: Check the Output window and make sure build is successful.

Step 4: Make sure the API console window displays "Application started...".

Also notice "Now listening on: http://0.0.0.0:5000" in the console window. This is the URL for accessing the Web APIs. You can use "localhost" or the IP address to access the Web APIs running on the local computer. The port number can be modified in the **launchSettings.json** in the PowerServer C# solution and will take effect in the development environment.

When the installable cloud application is run later, you can view the logs in the console window to check if the requests and responses are processed successfully.

Figure 5.15:



5.4 Running the installable cloud application

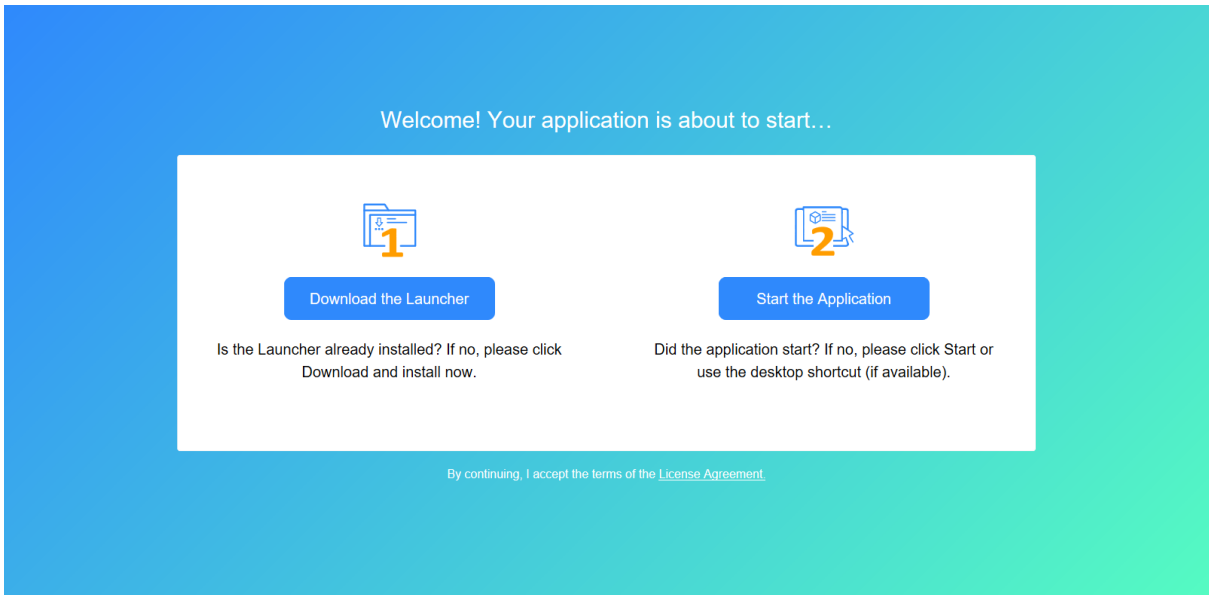
Step 1: Click the **Run PowerServer Project** button ( ) in the toolbar to run the application.

For more information about running the application, refer to Run the installable cloud application.

Step 2: In the app entry page that appears, click **Download the Launcher** to download and install the launcher.

After the launcher is installed, the application should automatically start, if not, click **Start the Application** in the entry page to start the application.

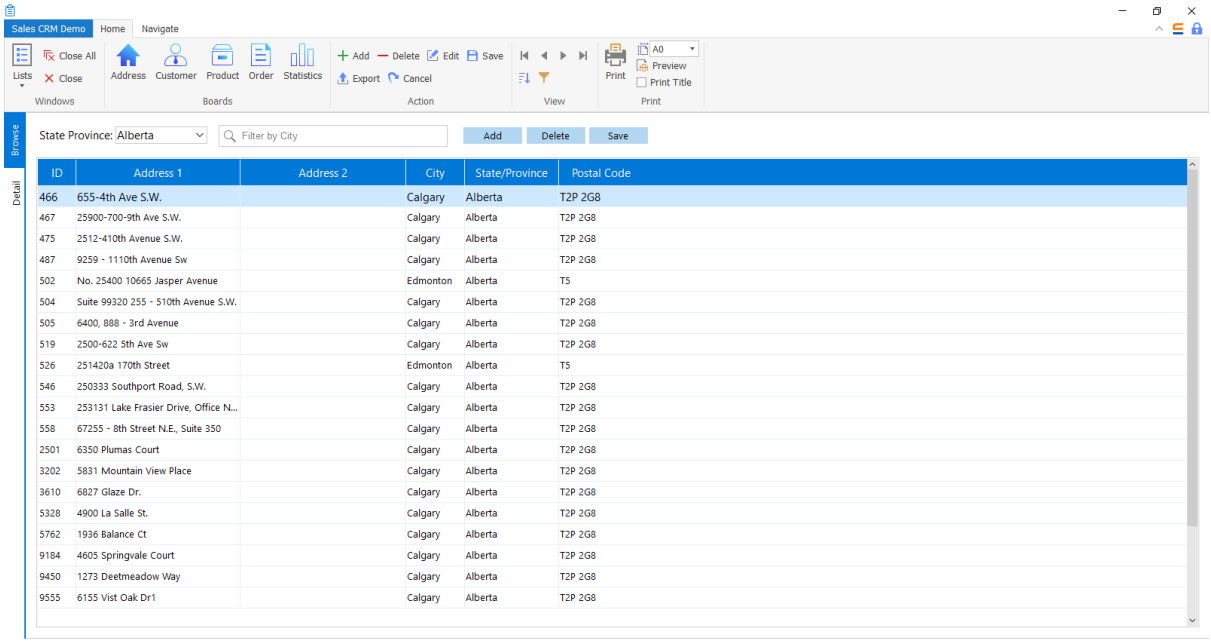
Figure 5.16:



Step 3: When the application main window displays, click the **Address** button in the application toolbar. Data should be successfully displayed.

You can view the logs in the API console window to check if the Web API requests and responses are successful.

Figure 5.17:



How-to Guides

Contents

1 Overview

The following tasks give a comprehensive overview of what you can perform for a PowerServer project:

1. Create the PowerServer project.
2. Define the PowerServer project.
3. Configure the Web server for deployment.
4. Upload the cloud app launcher and the runtime files.
5. Configure the Web API settings.
6. Configure the database connection.
7. Import license and activate PowerServer.
8. Analyze the unsupported features.
9. Build and deploy the PowerServer project.
10. Compile and run the Web APIs.
11. Check the status of Web APIs.
12. Run the installable cloud application.
13. Customize the app entry page.
14. Customize the deployed app using commands.
15. View the API documentation.
16. Get/Kill user sessions.
17. Package the client app.
18. Undeploy the client app.
19. Uninstall the client app.

2 Create the PowerServer project

Recommendation: It is recommended that you launch PowerBuilder IDE as an administrator; otherwise PowerBuilder IDE may not have full permissions to read/write the folder under the Web server.

To create a PowerServer project:

1. Select **File**>**New** or click the **New** button in the PowerBar to open the **New** dialog box.
2. Select the **Project** tab.
3. Select the target in which you want to create the project from the **Target** drop-down list.
4. Select the **PowerServer** project type and click **OK**.

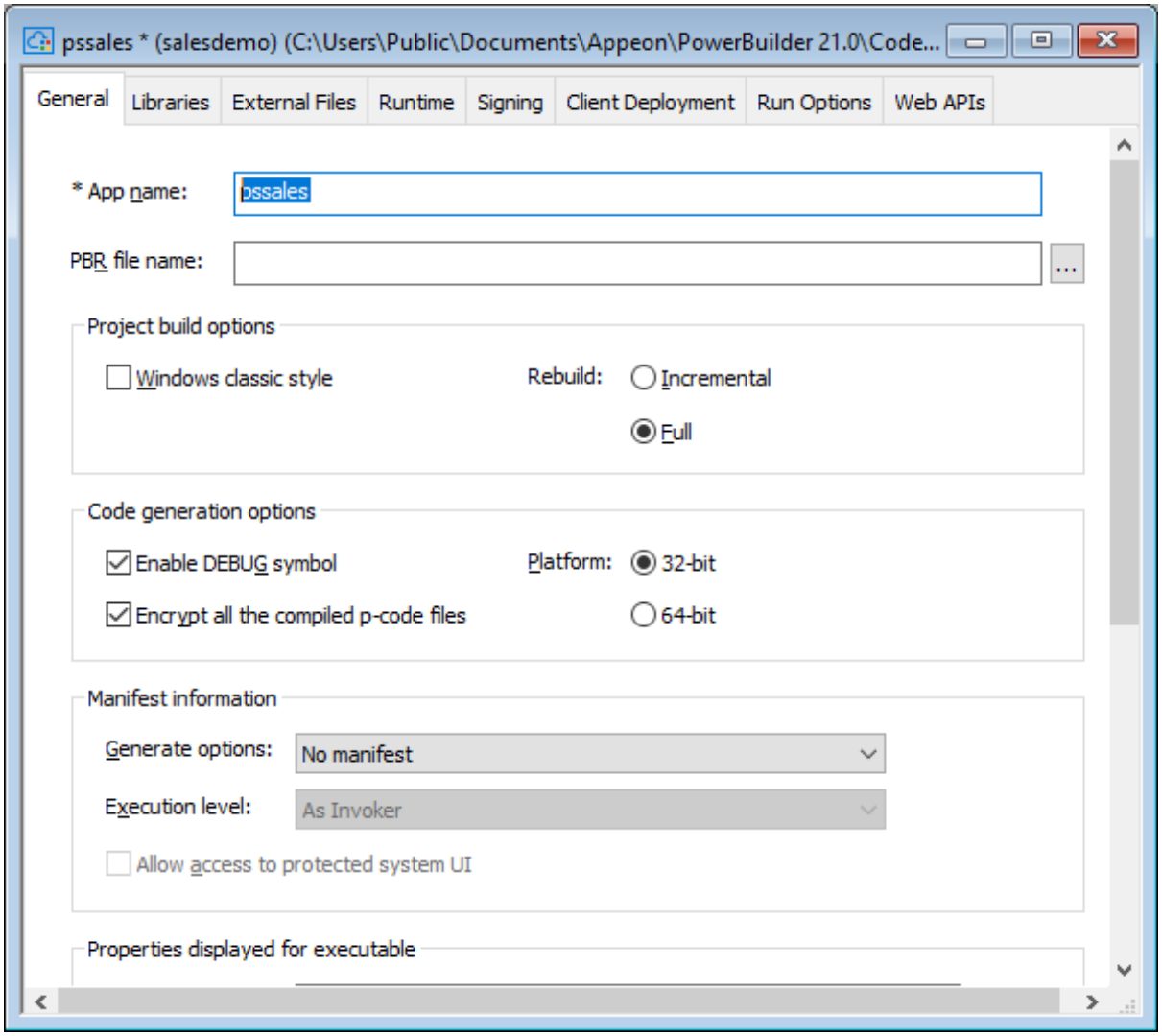
   The **Project** painter for PowerServer opens so that you can specify the various properties of your application.
5. When you have finished defining the project object, save the object by selecting

   File>**Save** from the menu bar or by clicking the **Save** button ( ) in the toolbar. PowerBuilder saves the project as an independent object in the specified library. Like other objects, projects are displayed in the System Tree and the Library painter.

3 Define the PowerServer projects

Once you have created a PowerServer project, you can open it from the System Tree and modify the properties if necessary. The Project painter for the PowerServer project looks like this.

Figure 3.1:



The following describes each of the pages and options you can specify in the Project painter for PowerServer.

General page

Table 3.1: General page

Option	What you specify
App name	Specify a name for the application.
PBR file name	(Optional) Specify a PowerBuilder resource file (PBR) for your application if you dynamically reference resources (such as bitmaps and icons) in your scripts and you want the resources included in the application instead of having to distribute the resources separately.

Option	What you specify
	You can type the name of a PBR file in the box or click the button next to the box to browse your directories for the PBR file you want to include. The PBR file as well as the resources it references must reside in the application directory or subdirectory; and only relative paths of the PBR file and the resources will be accepted. For more about PBRs, see Distributing resources in PowerBuilder User Guide.
Windows classic style	Select this to add a manifest file to the application that specifies the appearance of the controls as an application resource. By default, this option is not selected, which means the Windows flat style is used and the 3D effect of some controls will be removed to have a "flat" look, for example, the 3D lowered border of Column and Computed Field in the DataWindow object, the background color of Button, the BackColor and TextColor of the tooltip, and the TabBackColor of tab header will not take effect. If you still want the 3D effect, you should select the "Windows classic style" option when deploying the application. **Note** If you have applied a theme to the application, you should not check the "Enable Windows Classic Style in the IDE" option in the System Options or the "Windows classic style" option in the project painter and the PB.INI file (if any) should not contain such setting, otherwise, the application UI will be rendered in the Windows classic style instead of the selected theme.
Rebuild	Specify either Full or Incremental to indicate whether you want to regenerate and redeploy all object files to the Web server. If you choose Incremental, PowerBuilder regenerates and redeploys only objects that have changed, and objects that reference any objects that have changed, since the last time you built the application. As a precaution, regenerate all objects before rebuilding your project.
Enable DEBUG symbol	Select to enable any code that you placed in DEBUG conditional code blocks. For more information, see Using the DEBUG preprocessor symbol in PowerBuilder User Guide.
Encrypt all the compiled p-code files	Select whether to encrypt the object files when compiled from the PowerBuilder dynamic libraries.
Platform	Select if the application can run on 32-bit or 64-bit machines.
Manifest Information	Select whether to generate a manifest file (either external or embedded) and to set the execution level of the application. For further information, see Attaching or embedding manifest files in PowerBuilder User Guide.

Option	What you specify
Properties displayed for executable	Specify your own values for the Product name, Company name, Description, Copyright, Product version, and File version fields associated with the application file and with machine-code DLLs. These values become part of the Version resource associated with the application file, and most of them display on the Version tab page of the Properties dialog box for the file in Windows Explorer. The Product and File version string fields can have any format.
Executable version used by installer	Specify the product version and file version (in numeric values) that will be used by Microsoft Installer to determine whether a file needs to be updated. The four numbers can be used to represent the major version, minor version, point release, and build number of your product. They must all be present. If your file versioning system does not use all these components, you can replace the unused numbers with zeros. The maximum value for any of the numbers is 65535.

Libraries page

Table 3.2: Libraries page

Page	What you specify
Libraries page	Specify a PBR file for a dynamic library if it uses resources (such as bitmaps and icons) and you want the resources included in the dynamic library instead of having to distribute the resources separately. You can type the name of a PBR file in the box or click the button next to the box to browse your directories for the PBR file you want to include. The PBR file as well as the resources it references must reside in the application directory or subdirectory; and only relative paths of the PBR file and the resources will be accepted.

External Files page

Table 3.3: External Files page

Page	What you specify
External Files page	Specify the custom user external files and/or the resource files that are referenced in the PowerScript. Make sure all these files are placed in the same folder or sub-folder of the application target (.pbt) file. **Files preloaded as compressed packages** and **Files preloaded in uncompressed format** The custom user external files will be downloaded from the server before the application starts. It is recommended that you deploy the files which stay unchanged most of the time (such as UI theme files) as one compressed package, so that it can be transferred faster; and deploy the files which may be modified frequently (such as INI files) as individual files, or deploy them as a separate package. • To deploy files as one compressed package, select **Files preloaded as compressed packages** from the list box, then click **Create Package** to

Page	What you specify
	create a package, and then click **Add Folder** or **Add Files** to add the folder or files under this package. • To deploy files as individual files, select **Files preloaded in uncompressed format** from the list box, and then click **Add Folder** or **Add Files** to add the folder or files under it. The custom user external files may include the following: • INI files (including pb.ini, pblab.ini, pbodb.ini etc.) You can specify the update strategy for the INI file by clicking the **INI Configuration** button. More details are provided below. • DLL/OCX files (requiring no administrator rights to register) You can specify which DLL/OCX files can be registered by Regsvr32 or Regasm by clicking the **DLL & OCX Registration** button. More details are provided below. • XML files or image files used by the UI theme or external functions • text files, PDF files or any other files used by the external function **Images/videos dynamically loaded** The resource files (such as images, videos etc.) are downloaded from the server at the moment when they are used by the application. You can select **Images/videos dynamically loaded** and then click **Add Folder** or **Add Files** to add the folder or files under it. **Note** The read-only files added under **Files preloaded in uncompressed format** or **Images/videos dynamically loaded** will lose its read-only attribute after transferred to the server via FTP. This seems to be a common issue with FTP transfer. **DLL & OCX Registration** If the DLL/OCX files need to be registered and can be registered by Regsvr32 or Regasm without requiring the administrator rights, you can click **DLL & OCX Registration** to select the DLL/OCX files so that they can be registered by Regsvr32 or Regasm automatically before the application starts; if the DLL/OCX files need to be registered but cannot be registered by Regsvr32 or Regasm or they need to be registered using administrator rights, you can specify the registration commands in **Preload Event** in the **Run Options** tab. **INI Configuration**

Page	What you specify
	When the application is updated, the INI file can be updated with the specified strategy. Click the **INI Configuration** button and then select one or more INI file and configure the strategy for them at one time; or select and configure for the INI file one by one. • Overwrite update -- The INI file on the client will be updated if the INI file downloaded from the server has been updated, and changes made to the local INI file will be lost. • Merge update -- The INI file on the client will be merged with the INI file downloaded from the server, so changes made to the local INI file will be preserved and merged into the INI file downloaded from the server. But notice that any setting that exists in the local INI file while does not exist in the downloaded INI file will be removed. • Do not update -- Once the INI file is downloaded to the client, it shall never be updated with the INI file downloaded from the server. **Note** The external files cannot contain any file that has the same name as the application, or the PBD or p-code file to be generated, otherwise duplicate name error occurs. For example, [appname].exe, [appname].xml, [appname].manifest file etc. cannot be added to External Files. For another example, test.pbl will be deployed as test.pbd, therefore, test.pbd cannot be added to External Files.

Runtime page

Table 3.4: Runtime page

Page	What you specify
Runtime page	Select the runtime files according to the features used in the application. The files will be downloaded from the server to the client, for the application to run. The deployment tool does not actually deploy the files, instead it notifies the application to download such files (corresponding to the runtime version displayed) from the server directly. The runtime version displayed on this page can be configured in the IDE > System Options dialog. And you will need to make sure the corresponding version of PowerBuilder Runtime is uploaded to the server when you upload the Cloud App Launcher to the server.

Signing page

Table 3.5: Signing page

Page	What you specify
Signing page	Select whether to digitally sign the application executable file (*appname*.exe). If you want to digitally sign the application executable file, you can specify the settings required for signing under the "Use the SignTool utility from the Windows SDK" option, for example, SignTool location, signing certificate, certificate password, signature algorithm, and URL of the time stamp server. And make sure Microsoft's SignTool has been installed on the current machine. Or you can place the signing scripts in a file (with file extension as .cmd) and then select the file for the "Use your own signing script" option. For example, to sign the application executable file (*appname*.exe) using Microsoft's SignTool, you may create a cmd file that includes the following scripts: `signtool.exe sign /f mycert.pfx /p password /d "My application" /du http://www.mytest.com /fd sha256 /tr "http://timestamp.digicert.com" /td sha256 mytest.exe` After the executable file is generated and before it is deployed to the server, PowerBuilder will sign the executable file using your own signing scripts or using the SignTool settings you specified. Make sure the PowerBuilder user has the appropriate rights to access the time stamp server and sign files.

Client Deployment page

Table 3.6: Client Deployment page

Option	What you specify
Deployment mode	Select to deploy the client app to a local server or a remote server. If you have not configured the server yet, click the **Server Configuration** button and follow instructions in Configuring the Web server for deployment to configure the server. If the option "Check the availability of Cloud App Launcher on the server during the deployment process" is selected, the deployment process will be terminated if no Cloud App Launcher is detected on the target server. For how to upload the app launcher and runtime files, refer to Uploading the app launcher and runtime files. You can also choose to package the client app as an executable installer or a zipped file, and then install the client-side to the Web servers. For more about packaging a client app, refer to Package the client app.
Deployment version	The deployment version number is used by the server to determine whether to perform an install or update for the application. It is recommended to increment the deployment version number every time when the application is updated and re-deployed.

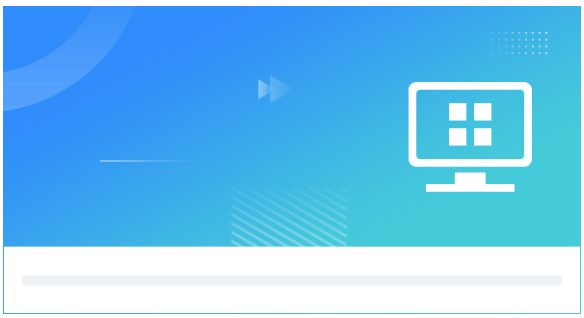
Option	What you specify
Available time and Expiration time	Schedule the time for the deployment version to be accessible or inaccessible to end users. However, if the available time or expiration time is reached and the app is still open, the app will not get updated, until the app is closed or the session times out. Therefore, it is recommended that the session timeout feature should be enabled (for apps deployed via PowerServer) or implemented (for apps deployed via PowerClient).
Minimum compatible version	Specify the lowest compatible version for the application. If the current version installed is older than it, a forced update will be performed, or the application will stop running.
Download options	Specify when to download the application files -- before the application starts or at the moment when they are called by the application at runtime. If you select "Download the app files as necessary", the following files will be downloaded before the app runs: 1) the PowerBuilder Runtime files, 2) the application executable, and 3) the files you selected to be preloaded in the External Files page; the other files will be downloaded at the moment they are called by the app. If you select "Download all the app files at app startup", the runtime files, app executable, the application files, and external files are all downloaded at the startup, except for the image files that are set to be dynamically loaded in the External Files settings.
App entry page settings	Specify which mode (with or without background service) will be run by default when the user accesses the application by inputting http://*IPAddress*/*AppName*. **IMPORTANT**: This setting must be consistent with the app launcher which is uploaded to the server, otherwise the application will fail to run. If you have changed the mode and uploaded the launcher again, make sure you also change the mode here accordingly, and ask the end user to clear the browser cache if the app launcher fail to run on the client. • If you have uploaded the app launcher **with** background service, then you should select "**Startup with background service**" (and keep "Deploy auto.html..." selected and "Deploy manual.html..." unselected). In such case, the user can input http://*IPAddress*/*AppName* or http://*IPAddress*/*AppName*/auto.html to access the application. The user should **not** input http://*IPAddress*/*AppName*/manual.html, otherwise it will lead to a "page not found" error or an infinite searching for files. • If you have uploaded the app launcher **without** background service, then you should select "**Startup without background service**" (and keep "Deploy manual.html..." selected and "Deploy auto.html..." unselected).

Option	What you specify
	In such case, the user can input http://*IPAddress*/*AppName* or http://*IPAddress*/*AppName*/manual.html to access the application. The user should **not** input http://*IPAddress*/*AppName*/auto.html, otherwise it will lead to a "page not found" error or an infinite searching for files. • If you have uploaded the app launcher **with and without** background service, then you can choose the default startup mode between "Startup with background service" and "Startup without background service" and then select both the "Deploy manual.html..." and "Deploy auto.html..." options. In such case, the user can input http://*IPAddress*/*AppName*/manual.html to run the application without background service, and input http://*IPAddress*/*AppName*/auto.html to run the application with background service; or input http://*IPAddress*/*AppName* to run the application in the default startup mode. The visual displays of the app entry page are customizable. For how, refer to Customize the app entry page.

Run Options page

Table 3.7: Run Options page

Option	What you specify
Commandline arguments	Specify the command line arguments for the application. The arguments will be directly passed to the application when the application is run. And the arguments will be automatically saved and updated to the app startup icon on the desktop and the app shortcut menu in Windows start. The arguments specified here cannot be modified at runtime. If you want to modify the argument at runtime, you can specify the argument in the application URL (for example, http://localhost/salesdemo/?arg=1). You can also pass arguments to the EXE directly. If there are multiple arguments, please include them in quotation marks or separate them with a delimiter (instead of a space), for example, C:\Users\<username>\AppData\Roaming\PBApps\Applications\localhost_<appname>\<appname>.exe **"parm1 parm2 parm3"** C:\Users\<username>\AppData\Roaming\PBApps\Applications\localhost_<appname>\<appname>.exe **parm1/parm2/parm3**
Show the loading animation before the app runs	Specify whether to show an animation (as shown below) when the application prepares for startup. The animation will disappear when the application's first window displays. This option should not be selected if the application starts with no user interface; otherwise the animation will not disappear.

Option	What you specify
	Figure 3.2:  You can deploy your own animation to replace the default animation (as shown above). For how, refer to Customize the app entry page.
Validate the application integrity before the app runs	Specify whether to validate the hash of every object file before they are loaded, so that files changed illegally will not be run.
App shortcut	You can specify whether to create the following shortcuts: • Desktop shortcut -- Specify whether to create an application shortcut icon on the client desktop. • Start menu shortcut -- Specify whether to create an application start shortcut menu in the Windows start menu. • App uninstall shortcut -- Specify whether to create an application uninstall shortcut menu in the Windows start menu. You can also customize the app shortcut name and the shortcut icon (the icon file must be added to the **External Files** tab first before it can be selected here).
Preload event	(Optional) Specify the commands that will be executed immediately after files are downloaded and before the application starts. For example, you can specify commands to register DLL/OCX files that cannot be registered by Regsrv32 or Regasm or require administrator rights to register; or any other commands that need to be executed with administrator rights. If the commands need to be executed with the administrator rights, you should select the **Run as administrator** option. You can specify how often the commands should be executed: for only one time when the application is launched for the first time or when the application is updated, or every time when the application runs. The commands can be any Windows commands or user-defined commands. For example, suppose there is a DLL file from the application that needs to be registered on the client, you can enter the following commands:

Option	What you specify

	`cd /d "C:\Windows\Microsoft.NET\Framework\v4.0.30319"`
	`regasm "%AppData%\Appeon\PBCloud \demo.appeon.com_app1\EncryptDecryptClass.dll" /tlb:testappeon.tlb /codebase /nologo`
	
	Note: As the commands are executed silently, any commands that will pause the execution and wait for user input will cause the application to wait endlessly.

Running app from IDE	Specify how the application can be launched from the PowerBuilder IDE (when you select the **Run PowerServer Project** button in the toolbar or from right-clicking the PowerServer project in the System Tree). You can specify the host name, port number, connection type (HTTP or HTTPS), and/or arguments. You can also specify to start the application from the Cloud App Launcher if the Cloud App Launcher is installed, or from a Web browser if the Cloud App Launcher is not already installed. If the Cloud App Launcher is not installed on the current machine, even if you have specified to start the application from the Cloud App Launcher, the Web browser will start to install the Cloud App Launcher and run the application. The arguments specified here will be appended to the application URL and then passed to the application via the URL, for example, http://localhost/salesdemo/?arg1&arg2. Note that the arguments appended to the application URL cannot contain special characters such as "?", "#", as they have special meanings in HTML URL; if you want to use these characters in the argument name or value, you can specify them in the **Commandline arguments** as static arguments on this same page, so that they can be passed to the application directly instead of being sent as part of the URL.

Web APIs page

Table 3.8: Web APIs page

Option	What you specify
Web API Generation	Specify the location, name, authentication template, and namespace for the PowerServer C# solution. The namespace can only contain characters, numbers, and underscores, and the first character must be a capital letter or underscore. If the PowerServer C# solution has already been created before, you can select an existing solution from the **Solution name** list, and then deploy the app to the existing solution; if you re-deploy an app to an existing solution, the application data models and ESQLs will be updated in the solution, and if you deploy a new app to an existing solution, the application data models and ESQLs will be added to the existing solution. You can also choose whether to overwrite the server settings (such as license, launch settings etc.) and the authentication template in the solution. Apps deployed to the same solution can share settings including

Option	What you specify
	PowerServer license, Web API URL, database connection settings etc. and can take advantage of additional features such as authorization, file server etc. that are developed by users. See Configure the Web API settings for more details.
Web API URL	Specify the URL for accessing the PowerServer Web APIs. It is highly recommended that you specify an HTTPS URL for the production environment. **Important** The port number in the Web API URL will be deployed to the PowerServer C# solution; so that when PowerServer Web API starts, it starts at the specified port number. And the complete Web API URL will be deployed to the Web server, so that the client knows where to call the PowerServer Web APIs.
License settings	You can click **Auto Import** to directly obtain and import the license from the Appeon sites, or click **Import from File** to select and import the license file. See Import license and activate PowerServer for more details.
Database Configuration	Click the **Database Configuration** button to configure the database connection for the application deployment and runtime. The database connection is required 1) when converting the PowerBuilder DataWindow objects to C# models during the deployment process; and 2) when accessing data from the database at application runtime. See Configure the database connection for more details.

4 Configure the Web server for deployment

A Web server is required to host the client-side of the installable cloud app deployed from the PowerServer project. If you have not set up any Web server yet, you can follow this tutorial Setting up a Web server.

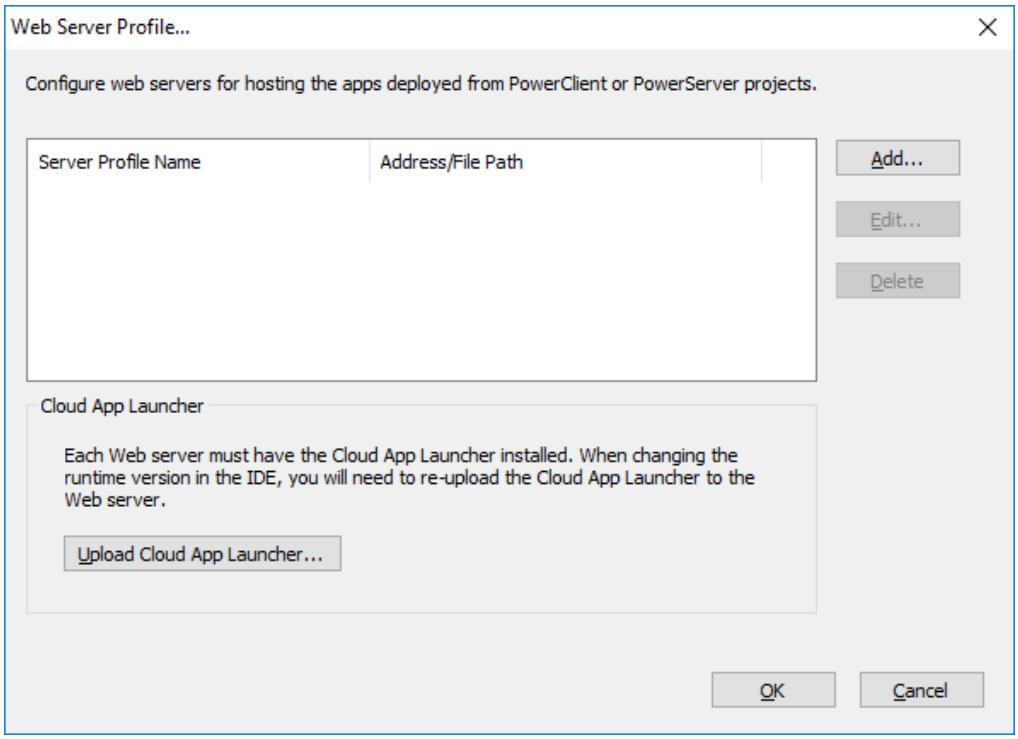
Any type of Web server (such as IIS, Apache, Nginx etc.) is supported. You can set up FTP on the server, so that you can remotely deploy the app to the server. For how to configure FTP on a server running against IIS, refer to Creating an IIS FTP site. For how to configure SSL on a server running against IIS, refer to Configure an SSL-based FTP server.

To configure a deployment server:

1. Select **Tools**>**Web Server Profile** from the menu bar to open the **Web Server Profile** window.

2. In the **Web Server Profile** window, click the **Add** button.

Figure 4.1:



3. In the **Add/Edit Server** window, select **Local server** or **Remote server**.

   For a local server, set the **Web root full path** (for example %systemdrive%\inetpub\wwwroot for IIS), and then click **Test File Path** to ensure the path is valid.

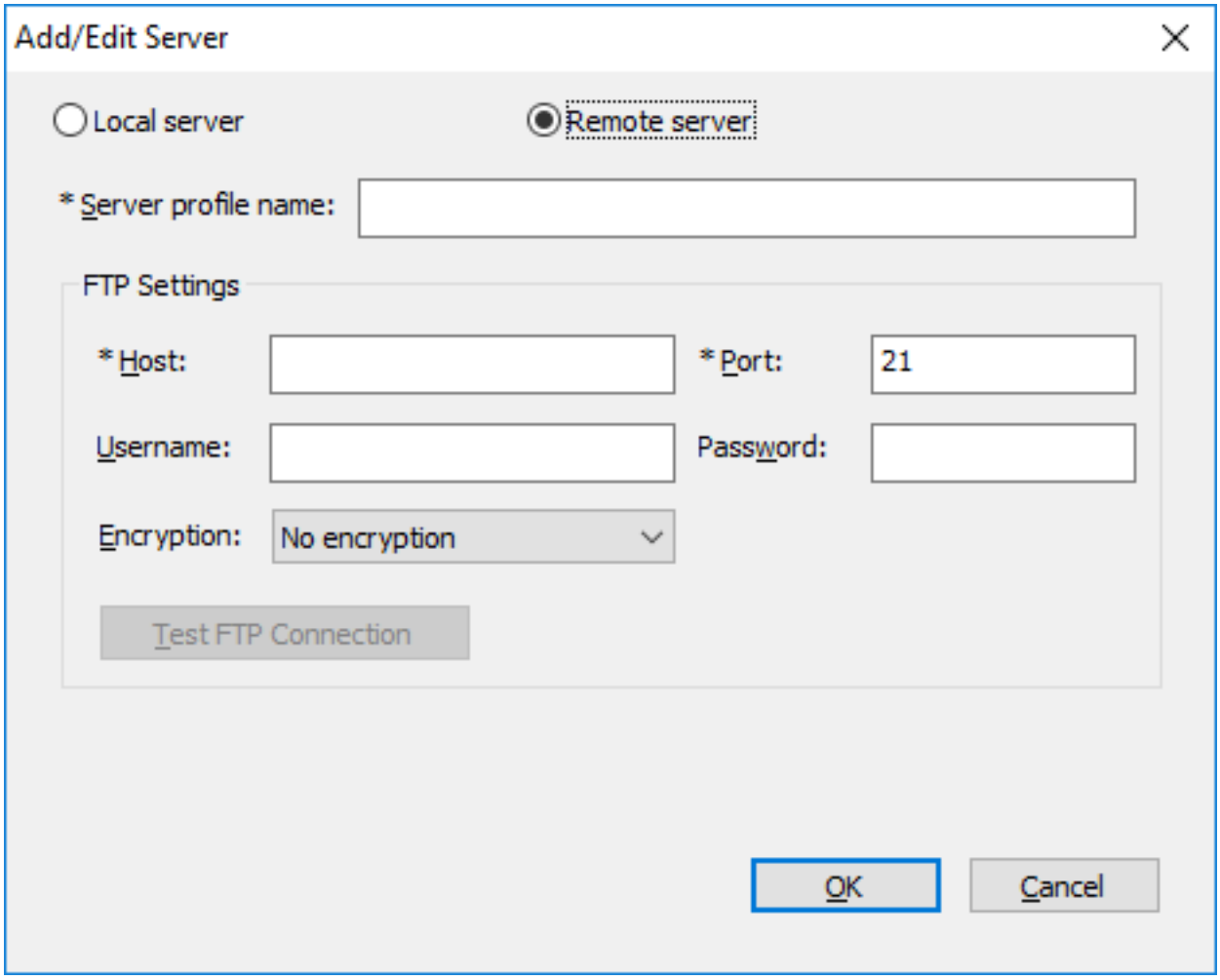
Note

If you intend to deploy to a local Web server, make sure you run PowerBuilder as administrator or have write permissions to the specified directory (administrator rights are required when transferring files to a local Web server).

For a remote server, specify a profile name and the connection settings for the FTP site (including host name, port number, FTP username, FTP password, and encryption), and then click **Test FTP Connection** to ensure the connection is successful.

4. Click **OK**. The server profile will be created.

Figure 4.2:



The server configuration will be used by all PowerServer projects; therefore if you have changed the server settings, you will need to upload the app launcher if no launcher has been uploaded to that server or directory.

Note

If you intend to deploy to the Web server through a proxy server, make sure the proxy server and the FTP server have the same encoding, otherwise, the multi-byte characters in the file/folder name will become unrecognizable after deployed to the server.

Note

As PowerBuilder is designed to be case-insensitive, therefore, in a case-sensitive system like Linux, some app files (such as images) may not be found or loaded. To

avoid any issue caused by the case of file name, make sure to configure the Linux server to ignore case-sensitive.

5 Upload the cloud app launcher and the runtime files

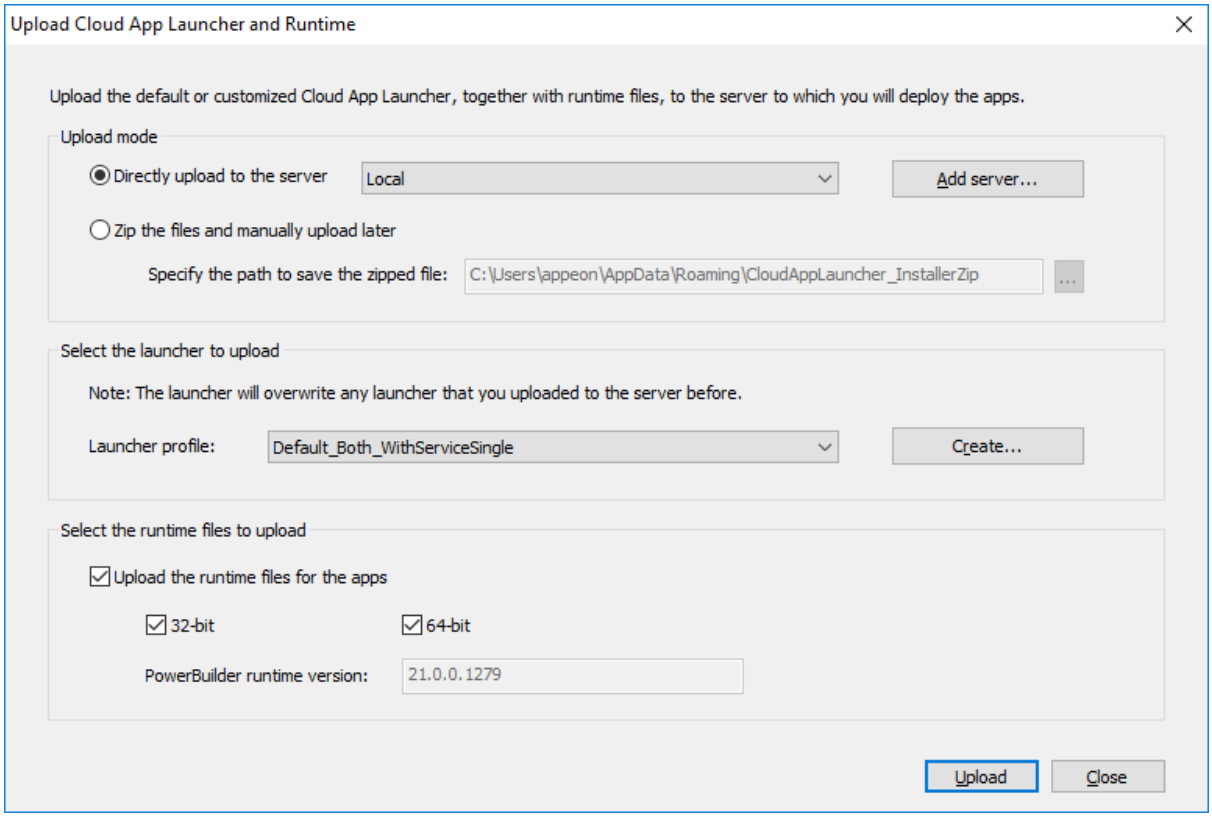
The app launcher and the runtime files must be uploaded to the Web server, and then installed to the client when the application is run for the first time. The app launcher and the runtime files will be used by all apps that are deployed to the same server and directory.

Note: there will be only one app launcher in the specified server and directory, although there can be multiple versions of runtime files. The app launcher will be overwritten without notice by the one uploaded later to the same server and directory.

To upload the app launcher and runtime files:

1. Select **Tools**>**Upload Cloud App Launcher** from the menu bar. The **Upload Cloud App Launcher and Runtime** window appears.

   Figure 5.1:



2. In the **Upload Cloud App Launcher and Runtime** window, select whether to directly upload the app launcher and runtime files to the server or only create a zip package and manually upload it to the server later.

   - To directly upload the app launcher and runtime files to the server, select a local server or a remote server where the app launcher and the runtime files will be uploaded.

   - To create a zip package which will be manually uploaded later, specify where the zip package will be created.

IMPORTANT: the app launcher and runtime files must be uploaded to the same server and directory where the application will be deployed. If you have not configured the server yet, follow instructions in Configure the Web server for deployment to configure the server first.

3. Select the runtime files (32-bit and/or 64-bit) to upload.

   The version of runtime files is determined by the runtime version selected in the IDE > System Options. Multiple versions of runtime files can co-exist on the same server and directory.

4. Select or create an app launcher to upload.

   You can select an existing app launcher from the **Launcher profile** list:

   - Default_WithoutService -- This profile specifies the launcher without the background service. It contains the following default settings:
     - **Launcher without background service** is selected.
   - Default_WithServiceSingle -- This profile specifies the launcher with the background service which supports single Windows user by default. It contains the following default settings:
     - **Launcher with background service** is selected.
     - **Single user** is selected.
   - Default_WithServiceMulti -- This profile specifies the launcher with the background service which supports multiple Windows users by default. It contains the following default settings:
     - **Launcher with background service** is selected.
     - **Multiple users** is selected.
   - Default_Both_WithServiceSingle -- This profile specifies the launcher with the background service and the launcher without the background service; and the launcher with the background service supports single Windows user by default. This profile contains the following default settings:
     - **Launcher without background service** is selected.
     - **Launcher with background service** is selected.
     - **Single user** is selected.
   - Default_Both_WithServiceMulti -- This profile specifies the launcher with the background service and the launcher without the background service; and the launcher with the background service supports multiple Windows users by default. This profile contains the following default settings:

- **Launcher without background service** is selected.
- **Launcher with background service** is selected.
- **Multiple users** is selected.

Or you can create your own launcher by clicking the **Create** button, if you want to customize the launcher settings.

- Specify a profile name for your new launcher.
- Specify where to save your new launcher on the local machine.
- On the **General** tab, specify the title and the logo (ICO format) that will be shown in the launcher.
- On the **Advanced Options** tab, specify where to install the application on the client. The path in the **App path** field will be used as the default installation path. If you want to allow the user to select where to install the application during the installation process, you can select "Allow the user to change the path".

  IMPORTANT: If you want to set a different path as the default path instead of %AppData%\PBApps, you should NOT include the system variable (such as %windir %, %temp% etc.) other than %AppData%, because currently only the %AppData% variable is supported.

- On the **Advanced Options** tab, specify which app launcher will be uploaded and installed: launcher without background service, or launcher with background service, or both. When **Launcher with background service** is selected, you can specify the launcher with background service supports single Windows user by default or supports multiple Windows users by default, and/or if you want to allow the user to select which user option to support during the installation process, you can select "Allow the user to change the option".

- On the **Signing** tab, select whether to digitally sign the launcher executable file (CloudAppLauncher_Installer.exe).

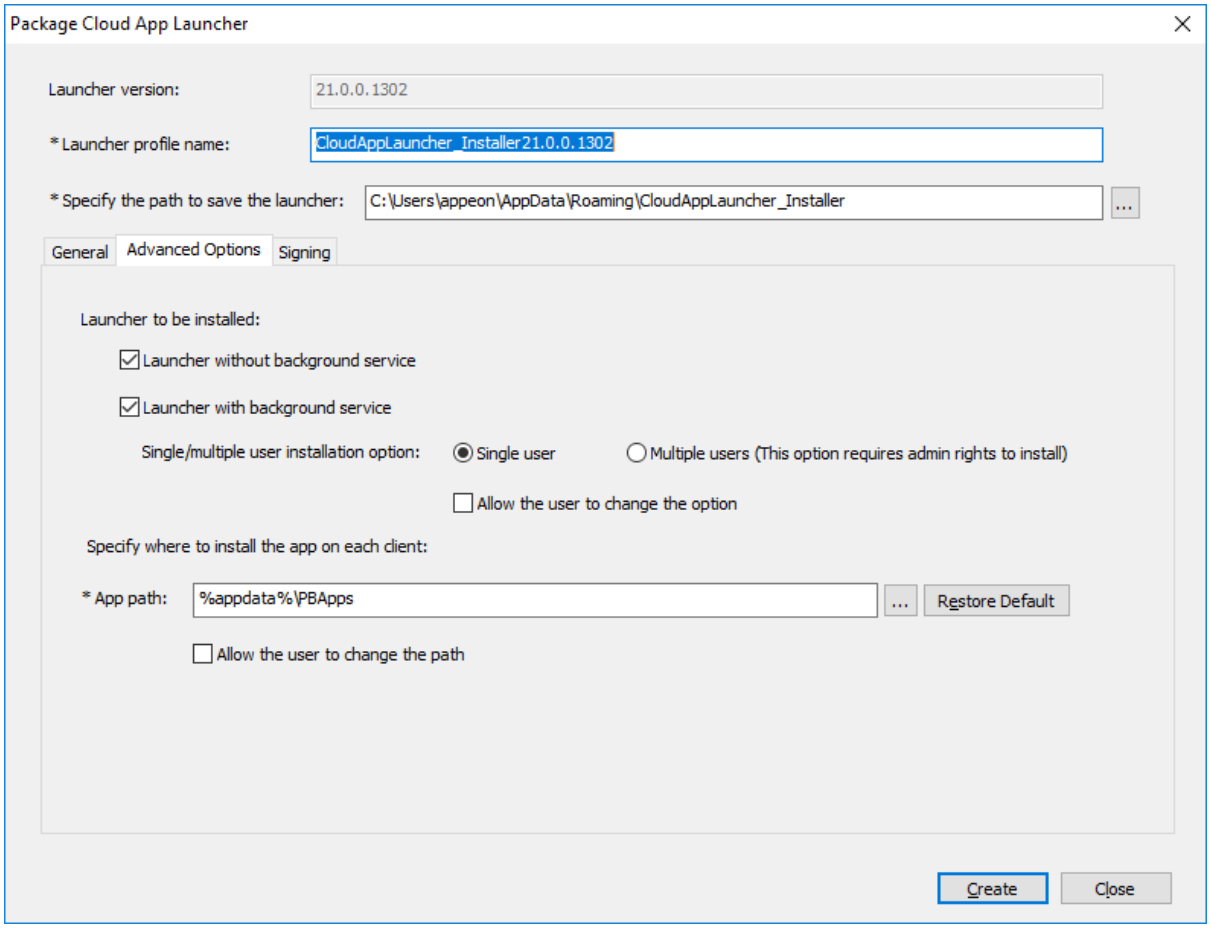
  If you want to digitally sign the launcher executable file, you can specify the settings required for signing under the "Use the SignTool utility from the Windows SDK" option, for example, SignTool location, signing certificate, certificate password, signature algorithm, and URL of the time stamp server. And make sure Microsoft’s SignTool has been installed on the current machine.

  Or you can place the signing scripts in a file (with file extension as .cmd) and then select the file for the "Use your own signing script" option. For example, to sign the executable file (CloudAppLauncher_Installer.exe) using Microsoft’s SignTool, you may create a cmd file that includes the following scripts:

```
signtool.exe sign /f mycert.pfx /p password /d "My app launcher" /du http://
www.mytest.com /fd sha256 /tr "http://timestamp.digicert.com" /td sha256
 CloudAppLauncher_Installer.exe
```

After the executable file is generated and before it is uploaded to the server, PowerBuilder will sign the executable file using your own signing scripts or using the SignTool settings you specified.

Figure 5.2:



Tip

To remove a launcher profile, go to the path where the launcher is saved (by default, C:\Users\appeon\AppData\Roaming\CloudAppLauncher), go into the folder which corresponds to the launcher version and then delete the sub-folder that is named after the profile.

5.1 About cloud app launcher

You can determine which type of cloud app launcher you want to upload to the server:

- Launcher without background service: This launcher program does NOT use a background service. As such, it should be easier to install and use and does not require administrator rights. However, it has certain dependency on the browser, which may result in different installation experience depending on the browser used and its configuration. If there are multiple users on the client machine, the launcher will need to be installed for each user.

- Launcher with background service: The launcher program uses a background service. If there are multiple users on the client machine, the launcher will need to be installed for each user, and only the first installation requires administrator rights to install and start the service. If the launcher is installed on the machine for the first time by a user without administrator rights, a window will pop up for inputting the administrator user name and password; after that, the other users also need to install the launcher but they do not need to have administrator rights, and all users will use the service started by the first installation. This launcher type does NOT have dependency on the browser.

When and why administrator rights are required?

When the cloud app launcher is first installed on the client, it needs to add the following entries to the registry:

- A registry entry for the protocol
- A registry entry for starting the launcher
- A registry entry for information that will be used by uninstall

Adding entries to the registry does not need administrator rights, unless the launcher with the background service which supports multiple Windows users (that is, the Default_WithServiceMulti or Default_Both_WithServiceMulti launcher) is installed, in such case, the launcher must be registered using administrator rights, so that the launcher can be started as a system-level service and used by all Windows users.

Silent installation of cloud app launcher

The cloud app launcher will be automatically downloaded and installed to the client when the application is run for the first time. If you want to silently install the cloud app launcher to the client before the application runs, you can get the installation package of the cloud app launcher from the PowerBuilder Runtime installation directory (for example, C:\Program Files (x86)\Appeon\Common\PowerBuilder\Runtime 21.0.0.1311\CloudAppInstall\default), and then run the following command to silently install the cloud app launcher.

```
CloudAppLauncher_Installer.msi /qn
```

6 Configure the Web API settings

To configure the Web API settings:

1. Select the **Web APIs** tab in the PowerServer project painter.

2. Select to create a new solution or select an existing solution from the **Solution name** list.

   New solution vs. existing solution

   Depending on whether multiple applications will use the same PowerServer solution or each application will use its own PowerServer solution, you can choose to create a new C# solution or an existing solution. If you want one PowerServer solution to be used by all applications, you can choose an existing solution; and then deploy the app (as well as the others) to this solution. If you re-deploy an app to an existing solution, the application data models and ESQLs will get updated in the solution, and if you deploy a new app to an existing solution, the application data models and ESQLs will be added to the existing solution.

   You can also select whether to overwrite the server settings (such as database configurations, license, Web API port etc.) in the existing solution. Apps deployed to the same solution can share settings such as the PowerServer license, Web API port, database configurations etc. and can take advantage of new developments added by the user such as authorization, file server etc.

   For more information about the PowerServer C# solution, see About the PowerServer C# solution.

3. Select a template type from the **Auth Template** list.

   - **Do not use auth service**: Provides no authentication template.

   - **Use built-in JWT server**: Includes a built-in authentication server that supports JWT or bearer tokens. See *Tutorial 6: Authenticating your apps* > Using JWT for more information.

   - **Use built-in OAuth server**: Includes a built-in authentication server based on IdentityServer4 framework that works with the OAuth 2.0 authorization flows. See *Tutorial 6: Authenticating your apps* > Using OAuth 2.0 for more information.

   - **Use built-in AWS Cognito server**: Includes a built-in authentication server that works with the Amazon Cognito user pool. See *Tutorial 6: Authenticating your apps* > Using Amazon Cognito for more information.

   - **Use external auth service**: Includes templates that can be easily extended to support the other identity providers that work with the OAuth flows or JWT, such as Azure AD or Azure AD B2C. See *Tutorial 6: Authenticating your apps* > Using other auth servers for more information.

4. Input a name as the namespace for the PowerServer C# solution.

The namespace can only contain characters, numbers, and underscores, and the first character must be a capital letter or underscore.

5. Specify the Web API URL.

   It is highly recommended that you specify an **HTTPS** URL for the production environment.

   This URL will be deployed to two areas:

   - The port number in the Web API URL will be deployed to the PowerServer C# solution; so that when the PowerServer Web APIs starts in the development environment, it starts at this port number. You can change this port number in the PowerServer C# solution > **ServerAPIs** project > **Properties** > **launchSettings.json** > **"applicationUrl"** setting.

   - The complete Web API URL will be deployed to the Web server, so that the client knows where to call the PowerServer Web APIs. If you want the client to call the PowerServer Web APIs running at a different URL, you can change the Web API URL using the **CustomizeDeploy.dll** tool. See this section for more details.

Figure 6.1:

General | Libraries | External Files | Runtime | Signing | Client Deployment | Run Options | Web APIs

Solution generation

Specify the solution to contain the Web API projects, namely, the AppModels and ServerAPIs projects.

* Solution location: C:\Users\appeon\source\repos ...

* Solution name: PowerServer_salesdemo

* Auth Template: Do not use auth service

* Namespace: Pssales

☑ Overwrite server settings (DB connection, Web API port, and license)

Web API URL

The app will connect to the PowerServer at the following Web API URL. The URL is the same for all the projects in the same solution.

* Web API URL: https://172.16.100.104:5009

scheme://host[:port][/path]

License settings

Specify the PowerServer license by importing the license file. Auto Import | Import from File | Remove

PowerServer license version: (empty)

Database Configuration...

7 Configure the database connection

Before you can build and deploy a PowerServer project, you MUST configure the database connection in the **Database Configuration** window (from the PowerServer project painter > **Web APIs** tab > **Database Configuration** button).

Database connection is required 1) when converting the PowerBuilder DataWindow objects to C# models during the deployment process; and 2) when accessing data from the database at application runtime. The database information (including the cache settings and the transaction-to-cache mappings) will be deployed to the **ServerAPIs** project in the PowerServer C# solution.

This section talks about creating a database connection cache and mapping it with the transaction object in the **Database Configuration** window. (You may want to consider the other database connection methods as discussed in Working with Database Connections).

Note that you only need to map the transaction objects that already exist in the PowerBuilder application.

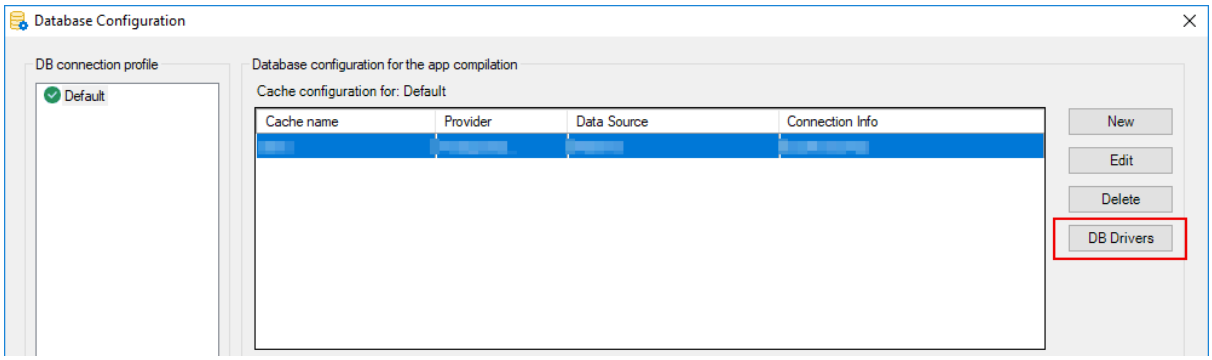
First of all, you must select the required database driver and agree to the driver license terms as the driver must be downloaded from the NuGet site to the PowerServer C# solution. You must do this no matter where you will create the database connection (in the PowerServer project settings > **Database Configuration** window or in the PowerServer C# solution > **ServerAPIs** project).

To select the required database driver:

1. Click the **Database Configuration** button at the bottom of the **Web APIs** tab.

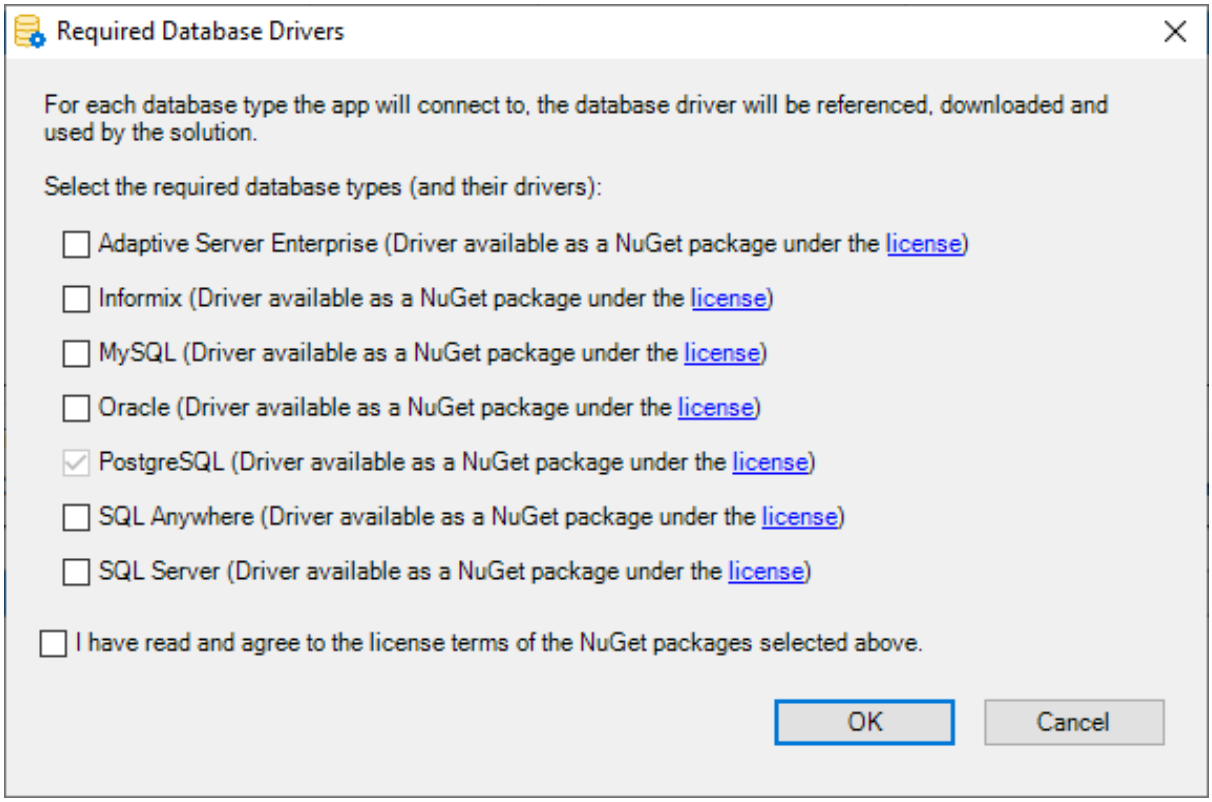
2. Click **DB Drivers** in the **Database Configuration** window.

Figure 7.1:



3. In the **Required Database Drivers** window, select the driver and the option "I have read and agree to the license ..."; and then click **OK**.

Figure 7.2:



To configure the database connection in the Database Configuration window:

1. Click the **Database Configuration** button at the bottom of the **Web APIs** tab.

2. In the **Database Configuration** dialog, you can create various DB connection profiles which include database connections to be used in different environments, for example, database connections for the development environment, testing environment, production environment, etc.

   Each DB connection profile will have a corresponding **Applications.[DBConnectionProfile].json** created in the PowerServer C# solution > **ServerAPIs** project > **AppConfig** for storing its settings such as database connection cache(s), transaction-to-cache mapping(s) etc. For example, the default **Applications.json** stores the settings of the "Default" connection profile, **Applications.Test.json** stores the settings of the "Test" connection profile, **Applications.Production.json** stores the settings of the "Production" connection profile.

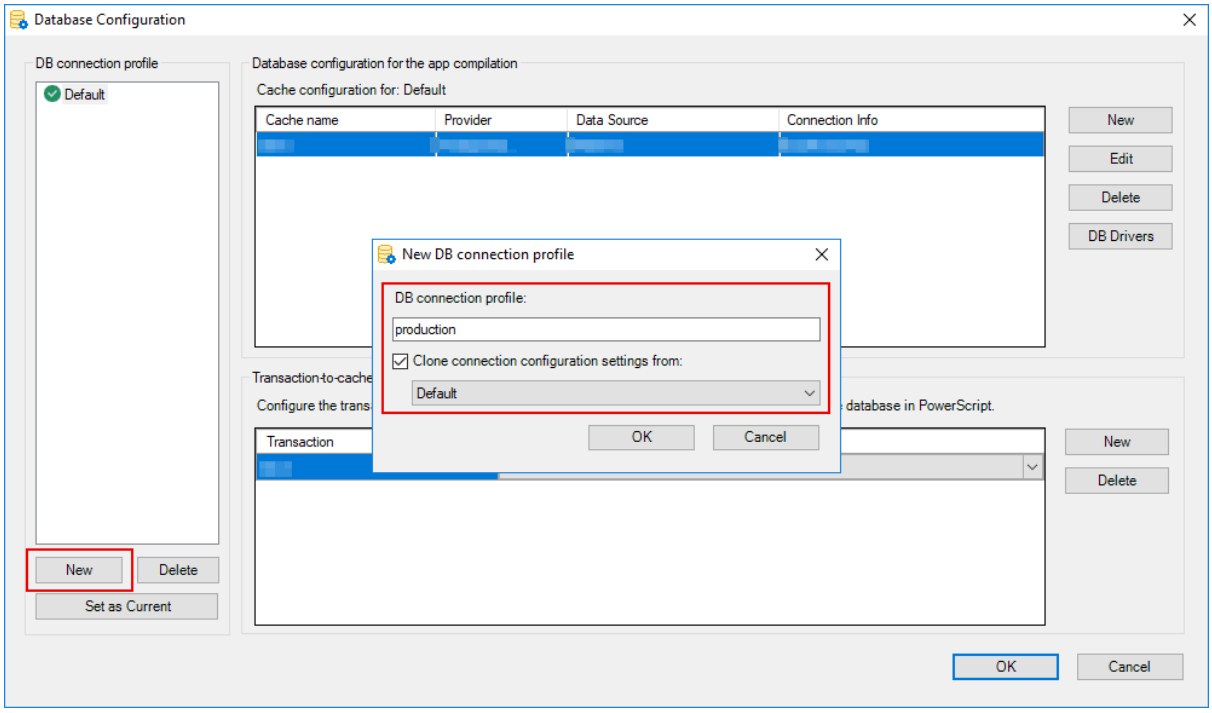
   You can then decide which connection profile to be used in the application by selecting the profile and clicking the **Set as Current** button. The name of the current profile will be stored to the "POWERSERVER_ENVIRONMENTTYPE" setting in the **ServerAPIs** project > **AppConfig** > **AppConfig.json**.

   To create a new DB connection profile:

   - Click **New** in the **DB connection profile** group.

- In the **New DB connection profile** dialog box, specify a name for the DB connection profile, for example, *production*.
- To create the new connection profile from an existing profile, you can select the check box below and then select an existing profile to clone from.

Figure 7.3:



3. In the **Database Configuration** dialog, you can create the connection cache that connects with the database.

   For example, you can establish a connection with the SQL Anywhere database for the PowerBuilder demo using the following settings:

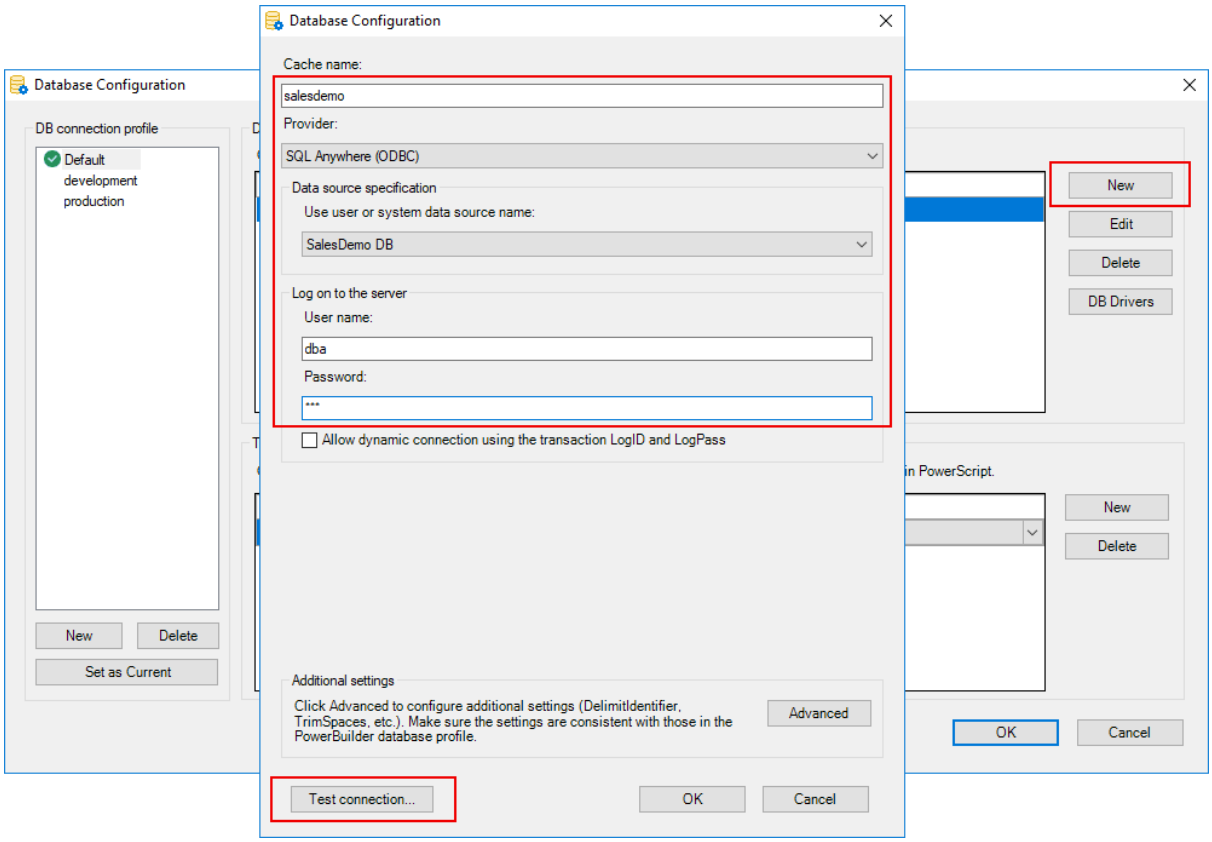
   - Click **New** in the upper part of the **Connection configuration** group.
   - In the Database Configuration dialog box, specify any text as the cache name.
   - Specify **SQL Anywhere (ODBC)** as the database provider.
   - Select the data source.
   - Specify the user name (for example, dba) and password (for example, sql).
   - Click **Test Connection** to make sure the database can be connected successfully.

   The "Allow dynamic connection using the transaction LogID and LogPass" option allows the application to use the LogID and LogPass property values of the Transaction object to log in to the database server as shown in the example below (instead of using the values in the User name and Password fields). For more, refer to Using LogID and LogPass properties.

```
Transaction.LogId = "sa"
Transaction.LogPass = "Appeon123!@#"
```

The **Advanced** button contains additional important settings for the database driver such as DelimitIdentifier, TrimSpaces, etc. If your database has such settings, make sure to click the **Advanced** button to configure those settings.

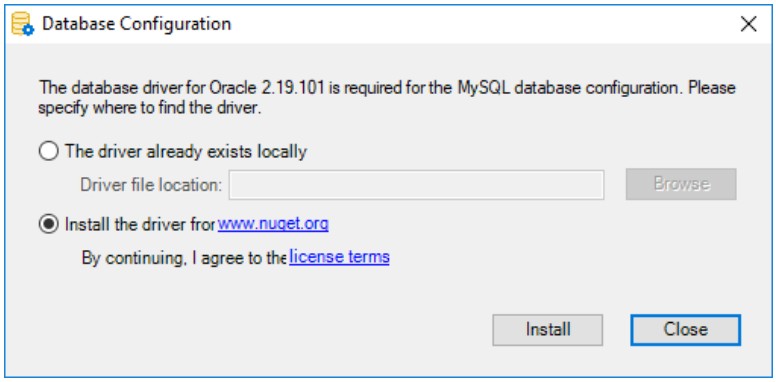
Figure 7.4:



If you select **MySQL**, **Oracle**, or **Informix** from the **Provider** listbox, you will be asked to specify a location for the required driver (MySql.Data 8.0.25, Oracle.ManagedDataAccess.Core 2.19.110, or IBM.Data.DB2.Core 2.2.0.100) or allow PowerBuilder to install the required driver from the NuGet website.

The packages downloaded from the NuGet website will be stored to %USERPROFILE%\.nuget\packages and cached in %USERPROFILE%\.sd\19.0\dbDrives\, so they can be automatically loaded when the database connection is created.

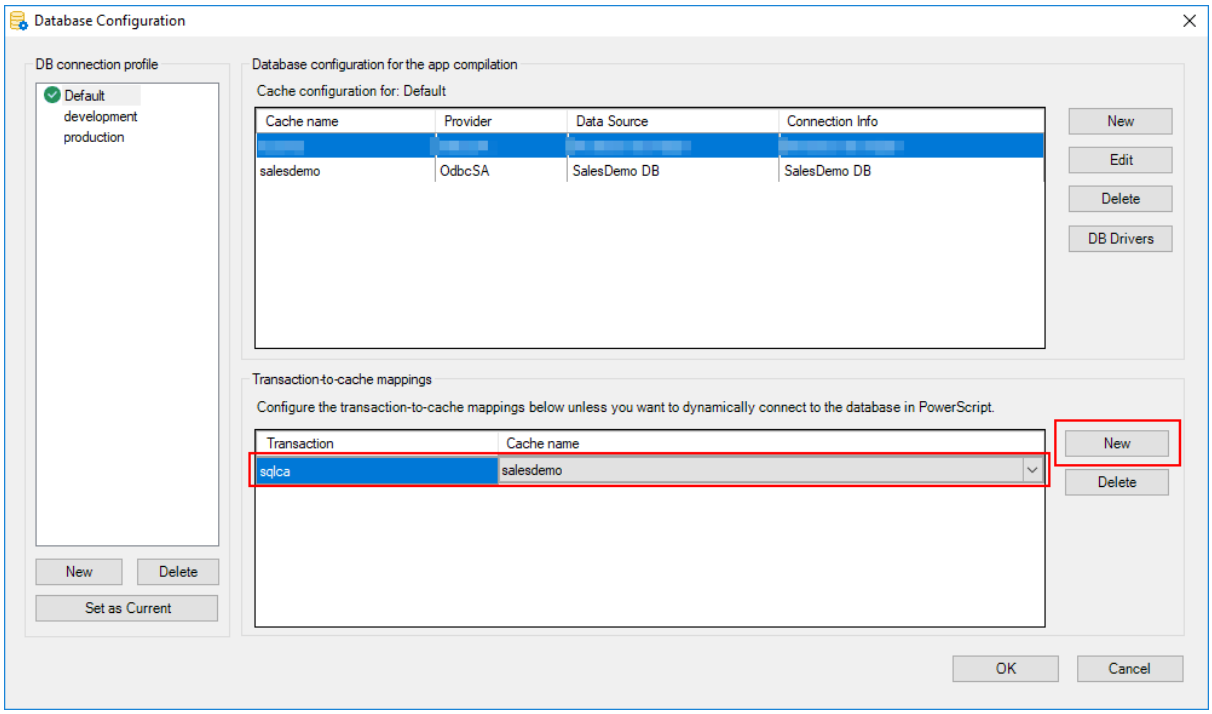
Figure 7.5:



4. After the database cache is created, you can map the transaction object with the cache in the **Database Configuration** dialog. To do this:

   - Click **New** in the lower part of the **Connection configuration** group.
   - Input the transaction object name (for example "sqlca") and then select the cache to map with.

Figure 7.6:



Rather than making static mappings of the cache and the transaction object (as shown above), you can also create dynamic mappings by using the DBParm CacheName property. For more details, see Working with Database Connections.

To manually configure the database connection in the ServerAPIs project:

When the PowerServer project is built and deployed in the PowerBuilder IDE, the cache settings (including database server host/port, database name, login ID, password, advanced

settings etc.) and the transaction-to-cache mappings configured in the Database Configuration window will be deployed and stored in PowerServer and you can manually change these settings in the PowerServer C# solution. To do this:

1. Open the PowerServer C# solution > **ServerAPIs** project > **AppConfig** > **Applications.json** or **Applications.[DBConnectionProfile].json** file.

   The Applications.json file contains the configuration of the "Default" DB connection profile. If you have another connection profile, the profile name is added in the middle of the file name. For example, Applications.Development.json file contains the configuration of the "Development" DB connection profile.

2. In the **Applications.json** (or **Applications.[DBConnectionProfile].json**) file, locate the "Applications" block > [application name] > "CloudTransactions". This is where the transaction-to-cache mapping(s) is stored.

   In the following example, the "sqlca" transaction object is mapped to the "local-sa" database cache. You can modify the existing mapping, or create a new mapping by making a copy of the existing one.

```
  "Applications": {
    "pssales": {
      "CloudTransactions": {
        "sqlca": {
          "CacheName": "local-sa"
        }
      },
      ...
```

3. In the **Applications.json** (or **Applications.[DBConnectionProfile].json**) file, locate the "Connections" block. This is where the cache(s) is stored.

   In the following example, there are two caches "local-sa" and "local-postgresql" under the "Default" cache group; and each cache contains the database connection information that are configured and deployed from the Database Configuration window. You can modify the existing cache, or create a new cache by making a copy of the existing one.

```
...
  "Connections": {
    "Default": {
      "local-sa": {
        "ConnectionType": "Odbc",
        "OdbcName": "PB Demo DB V2021",
        "OdbcDriver": "SqlAnywhere",
        "UserID": "dba",
        "Password":
 "eyJQYXlsb2FkIjoiYlx1MDAyQkxocTNiMUtWSzhBY1FCbVltU0FBPT0iLCJUaW1lc3RhbXAiOjE2MjU2NDYwNDcsIlN
        "CommandTimeout": 30,
        "OtherOptions": "",
        "DynamicConnection": false
      },
      "local-postgresql": {
        "ConnectionType": "PostgreSql",
        ...
      }
    }
  }
```

But notice that the PowerServer C# solution will be updated every time when the PowerServer project is built and deployed in the PowerBuilder IDE. If you manually modify the settings in **Applications.json** (or **Applications.[DBConnectionProfile].json**), and want to keep these changes, you should use the "Overwrite server settings (DB connection, Web API port, and license)" option properly. For more information, refer to What settings will be deployed to the solution.

8 Import license and activate PowerServer

First of all, make sure you have a valid license for the PowerServer 2021 GA version.

- ⊗ If you have a preview or beta license, the preview or beta license will no longer work with the GA version.
- ⊙ If you already have a PowerBuilder CloudPro license (no matter which version it is), the CloudPro license will automatically work with the GA version. Each PowerBuilder CloudPro subscription includes a developer license of PowerServer, which supports a maximum of 5 user sessions (user session = installable cloud app). You will need to purchase a production license of PowerServer in order to use the production server and more user sessions.
- ⊙ If you have no PowerBuilder CloudPro license, you can apply for a trial license at https://www.appeon.com/psfreetrial, or purchase a production license of PowerServer from https://www.appeon.com/pricing.

Once you have a valid license, you can import the license and deploy it along with the Web APIs project. The license will be validated later when the PowerServer Web APIs is run.

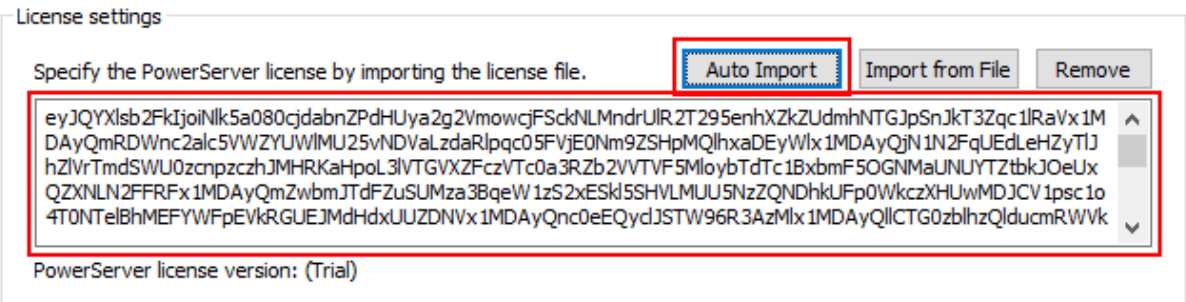
To import the license automatically:

To activate PowerServer using the developer license or trial license included in the PowerBuilder CloudPro subscription, you can obtain the license automatically from the Appeon website according to the current PowerBuilder IDE login account.

1. Make sure the computer can connect to the Appeon sites (through port number 80): https://api.appeon.com and https://api2.appeon.com.
2. Go to the **Web APIs** tab of the PowerServer project painter, and then click **Auto Import** to automatically import the license.

   PowerBuilder will automatically obtain the developer or trial license of PowerServer (according to your PowerBuilder IDE login account) from the Appeon sites and then import the license here.

   Figure 8.1:

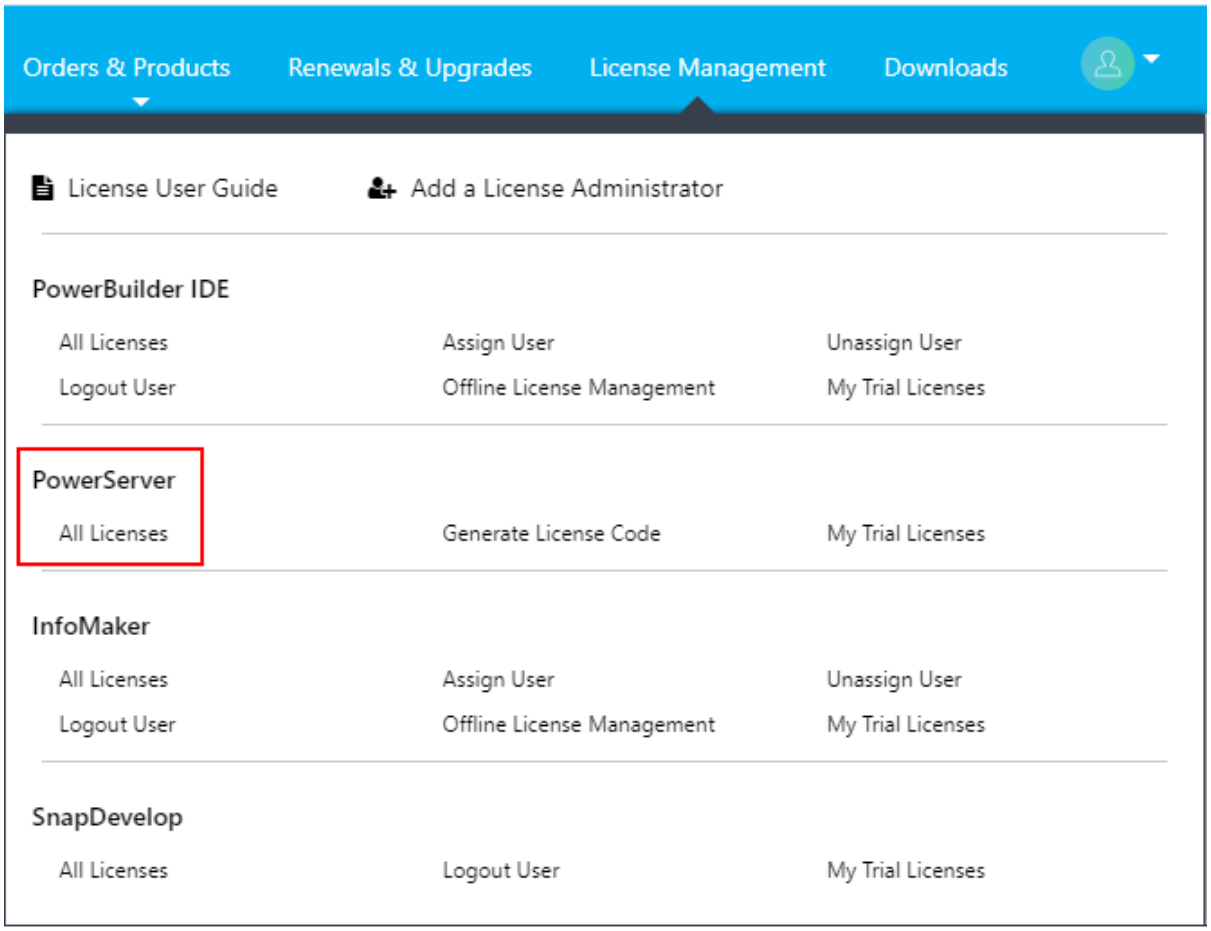


To import the license manually:

You can also export the license file from the Appeon website manually and then import the license here.

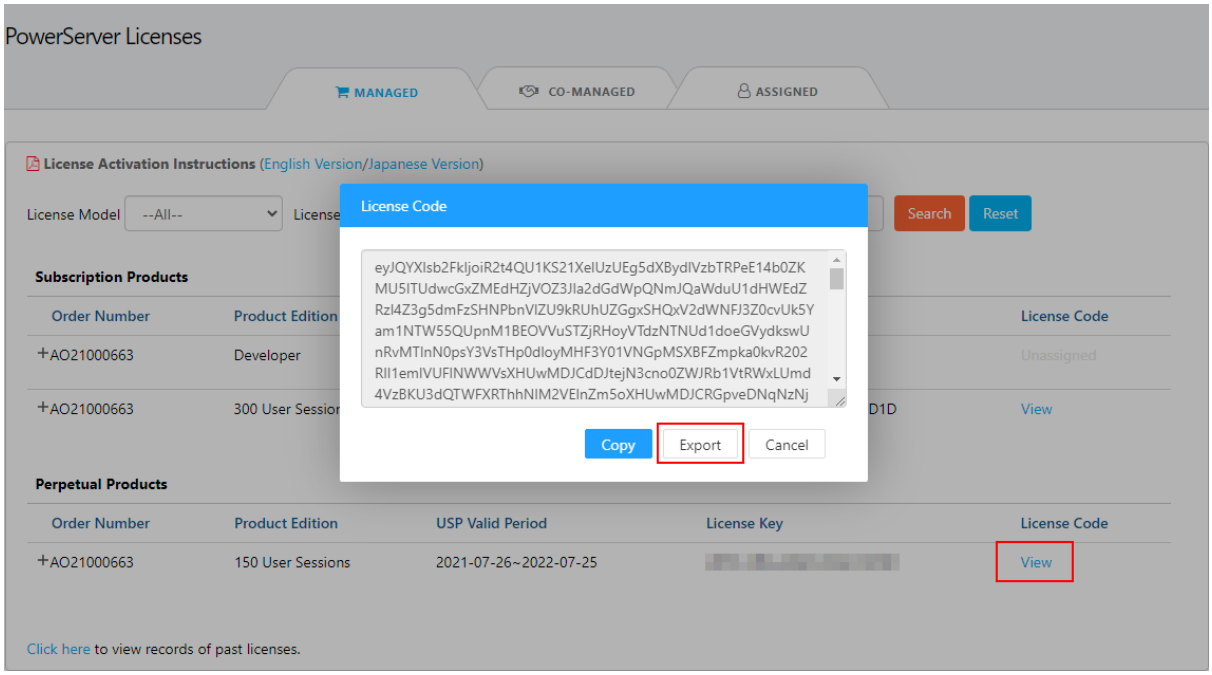
1. Log into the Appeon User Center, click **License Management**, and then click **All Licenses** under **PowerServer**.

Figure 8.2:



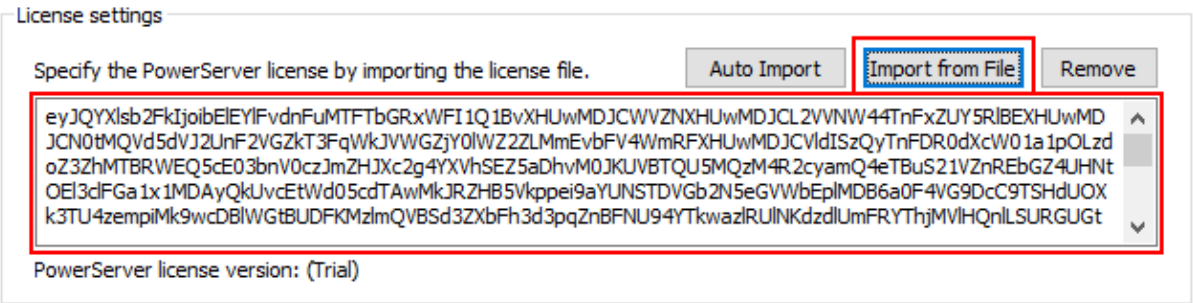
2. Click **View**, and then click **Export** to export the license code to a TXT file ([LicenseKey].txt) and save the file on the local machine.

Figure 8.3:



3. Go to the **Web APIs** tab of the PowerServer project painter, and then click **Import form File** to select and import the [LicenseKey].txt file.

Figure 8.4:



If there are multiple PowerServer projects that will use different PowerServers, then you will have to import the license to every project before deployment. The license will be deployed along with the PowerServer Web APIs (in the PowerServer C# solution > **ServerAPIs** project > **Server.json**).

The license will be automatically validated when the PowerServer Web APIs is run. Please make sure the .NET server can connect to the following Appeon websites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) so that the Appeon license server can successfully validate the license and activate the PowerServer packages.

9 Analyze the unsupported features

Refer to **Unsupported Features Guide** > How to detect unsupported features for details.

10 Build and deploy the PowerServer project

To build and deploy a PowerServer project:

1. Before building and deploying the application, make sure to close any antivirus tool on the development machine.

2. Click the **Build & Deploy PowerServer Project** button ( ) in the toolbar, or right-click the PowerServer project in the System Tree and then select **Build & Deploy PowerServer Project** to build and deploy the application to the server. Or select **Deploy PowerServer Project** if you have already built the application before.

   The application executable file (as well as the PBD files) is generated under %TEMP%\pbappscache\temp\[appname] (for example, C:\Users\appeon\AppData\Local\Temp\pbappscache\temp\pssales) on the development machine, then digitally signed, and deployed to the server.

   The PowerServer C# solution is generated under the specified location (by default C:\Users\[username]\source\repos).

   Note

   After the application is deployed to the server, do not manually change the application folder name on the server, otherwise the application uninstall program will fail to run.

The build & deploy process is composed of the following tasks:

Table 10.1:

Process	What does the process do?
Build	1) Generates or updates the PowerServer C# solution (using the specified Namespace and Auth Template). 2) Converts PowerBuilder DataWindow objects to .NET DataStore models and parses the embedded SQL statements and adds them to the **AppModels** project. 3) Configures or updates the PowerServer Web API compilation environment. 4) Compiles the scripts and analyzes unsupported features. 5) Generates the PBD files, app executable file, and other application files.
Deploy	Adds the server settings (app name, Web API port, PowerServer license, and database configurations) to the **ServerAPIs** project of the PowerServer C# solution.
Deploy or Package	Uploads the app files (PBD files, app executable file, external files etc.) and settings (runtime file list, Web API URL, and other project settings) to the Web server, or creates an application package that includes these files and settings.

You can also build and deploy the project using commands (see Build the PowerServer project with commands for details).

10.1 What is the PowerServer C# solution

The PowerServer C# solution is generated during the build process. After the solution is generated, you can click the **Open C# Solution in SnapDevelop** button () in the toolbar to launch the PowerServer C# solution in SnapDevelop. Or go to the location where the solution is generated; and double click **PowerServer_[appname].sln** to launch the solution in SnapDevelop or other C# editor such as Visual Studio.

The PowerServer C# solution is an ASP.NET Core solution which contains three projects:

- The **AppModels** project contains the C# models (converted from the PowerBuilder DataWindows) and the embedded SQLs (ESQL) from the PowerBuilder application.
- The **ServerAPIs** project contains the PowerServer Web APIs which is RESTFul APIs for handling the database connections, data processing, PowerServer license activation, and advanced features such as file server etc.
- The **ServerAPIs.Tests** project contains a number of test cases which can check if the PowerServer Web APIs is running correctly after the **ServerAPIs** project is modified. See Running the ServerAPIs.Tests project for more details.

The **ServerAPIs** project contains a number of configurable files and controllers. The following highlights the important files and settings only. For complete descriptions, refer to the **readme.txt** file under **Solution Items**.

- **Properties\launchSettings.json**: This file contains the environment settings for running the PowerServer Web APIs in the local development environment, for example, the *commandName* key specifies the web server to launch (the value "Project" indicates that the Kestrel web server will be launched), and the *applicationURL* key specifies the host name and port number for the web server. For description of the settings in this file, See https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-3.1#development-and-launchsettingsjson.
- **AppConfig**
  - **AppConfig.xml**: This file contains the DB connection profile that is currently selected and deployed from the **Database Configuration** window.
  - **Applications.json** or **Applications.[DBConnectionProfile].json**: This file contains the basic information of the deployed applications and the database connection cache settings.

    For each DB connection profile configured in the **Database Configuration** window, an **Applications.[DBConnectionProfile].json** file is created. **Applications.json** is for the "Default" DB connection profile.

    - "Applications": This block includes the mappings of transactions and connection caches, timeout values for transaction, session, and request, and run mode (0-normal mode, 1-test mode) of each deployed application.
    - "Connections": This block includes the database connection cache name, database type, data source settings, and some advanced settings.

- **Authentication**: This folder contains the template and built-in server for the selected authentication type. For more information, refer to Tutorial 6: Authenticating your apps.

- **Controllers**

  - **ApplicationController.cs**: This file provides APIs for dynamically adding, modifying or removing the application settings.

  - **ConnectionController.cs**: This file provides APIs for dynamically adding, modifying or removing the database connections such as cache or cache group.

  - **LicenseController.cs**: This file provides APIs for dynamically accessing the license information.

  - **SessionController.cs**: This file provides APIs for getting all user sessions or killing a particular user session. For more information, see Get/Kill user sessions.

  - **StatisticsController.cs**: This file provides APIs for getting statistics of the request and transaction.

  - **TransactionController.cs**: This file provides APIs for getting all transactions or rolling back a particular transaction.

  For documentation of these APIs, refer to View the API documentation.

- **HealthChecks**: Refer to the **readme.txt** file under **Solution Items** for more information. For more information, refer to Check the status of Web APIs.

- **Logging**

  - **log4net.xml**: This file contains the logging settings for PowerServer. The "RollingFile" appender specifies the location, size, and backup of the log file, the "TraceAppender" appender specifies to generate the trace log, the "ConsoleAppender" appender specifies to print the logging information in the console and sets the font color of the logging information. For detailed syntax, refer to Apache Log4Net Manual.

  - **Logging.json** or **Logging.Development.json**: This file specifies the log level (Trace, Debug, Information, Warning, Error, Critical, and None), what level of logging information will be printed in the API console, and what type of PowerServer information (SQL, transaction, and session) will be logged.

    Logging.json will take effect in the production environment (for example, when Web APIs are published and running in IIS, docker etc.); and the default log level is warning. **Logging.Development.json** will take effect in the development environment (for example, when Web APIs is running from the SnapDevelop IDE or the PowerBuilder IDE); and the default log level is information.

- **OpenAPI**: The OpenAPI Specification for implementing the API documentation.

- **Server.json** or **Server.Development.json**: This file contains settings related with the server. As you can use multiple environments in ASP.NET Core (read more), there can be

multiple configuration files, for example, **Server.json** will take effect in the production environment (for example, when Web APIs are published and running in IIS, docker etc.), while **Server.Development.json** will take effect in the development environment (for example, when Web APIs is running from the SnapDevelop IDE or the PowerBuilder IDE).

- **AllowedHosts**: This setting specifies the host names to bind with PowerServer. See Host filtering for more.

- **PowerServer**: This block specifies the settings for PowerServer.

  - LicenseKey & LicenseCode: PowerServer license information.

  - EncryptedSensitiveData: Sensitive data refers to the database login password which is used to create the database connection cache in **AppConfig\Applications.json**. If the password is an encrypted value (encrypted by the CustomizeDeploy.dll tool), this setting should be set to True; if the password is not encrypted (still a plain-text string), this setting should be set to False so that PowerServer will encrypt the password for you and store the encrypted value to the database. If this setting is set to True and you input a plain-text password, the plain-text password will be stored to the database.

  - AppModelsAssemblyNames: AppModels assembly name.

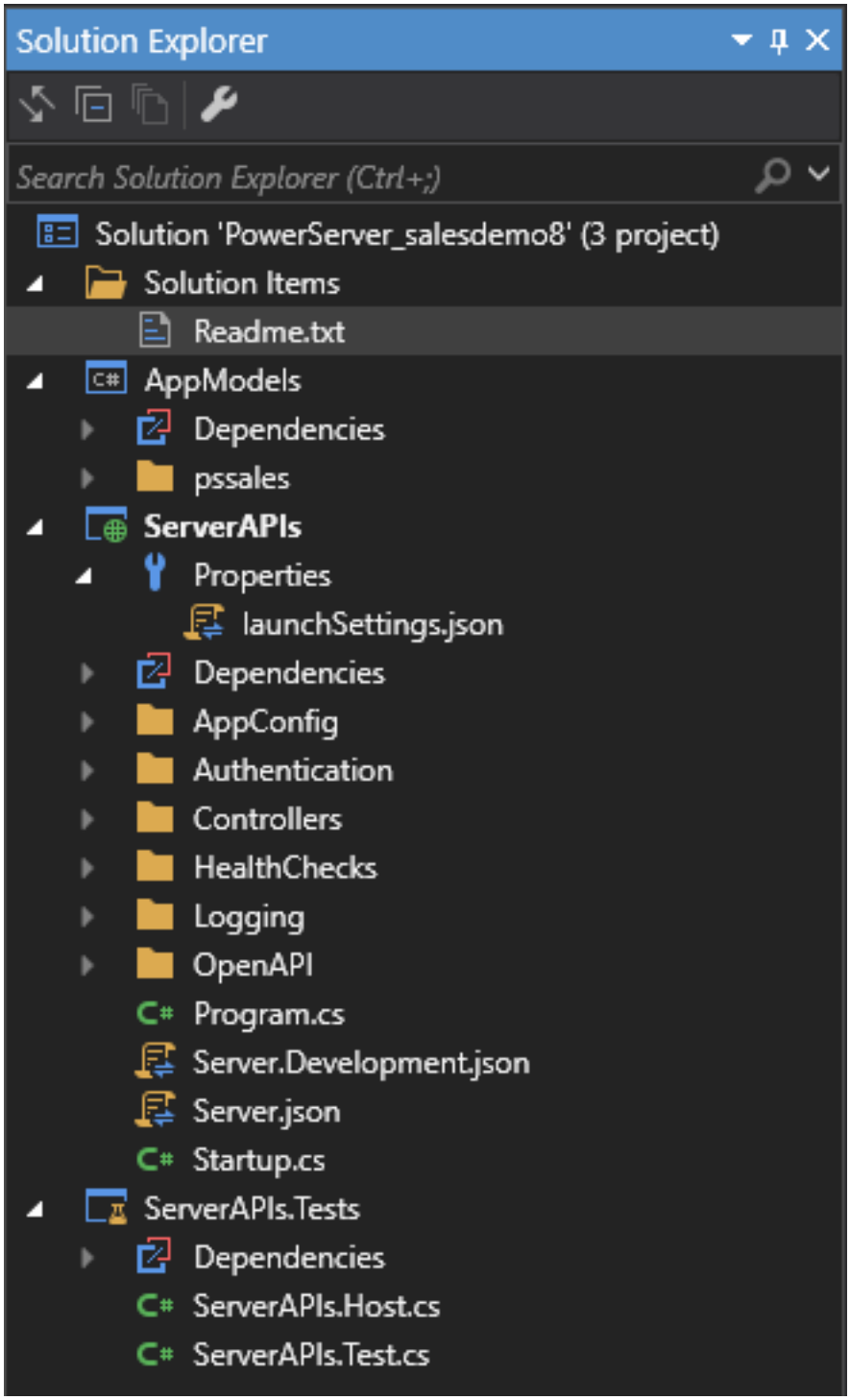
  - ProxyOptions: The IP address and login credentials of the proxy server. If the Web API host server connects to Internet through a proxy server, you will need to configure the proxy server settings here.

  - EmailOptions: This block must be configured first if you want to get notifications for license expiration. The settings include the SMTP server settings, sender email settings, and recipients.

  - StatisticsOptions: This block determines which type of transaction statistics will be generated and cached in the memory. Some settings are disabled by default to lower memory usages. You can also take advantage of the **StatisticsController** APIs in PowerServer NuGet package to get the statistics.

For files that are not mentioned here, refer to the **readme.txt** file under **Solution Items** for more information.

Figure 10.1:



10.2 What settings will be deployed to the solution

Although the PowerServer C# solution allows you to make changes to it, you will have to be aware that some settings in the solution might be updated every time when the PowerServer project is built and deployed in the PowerBuilder IDE.

Table 10.2:

Settings from the Web APIs tab	Will be updated to	By	Overwrite strategy
Auth Template	ServerAPIs project > Authentication	Build	When selecting a different authentication template, you will be prompted whether to overwrite the existing authentication.
Namespace	AppModels project	Build	Overwritten all the time.
Port number in the "Web API URL" field (**Note**: the complete URL of Web API is deployed to the Web server all the time)	ServerAPIs project > Properties > launchSettings.json > port number in "applicationUrl"	Deploy	Determined by the "Overwrite server settings (DB connection, Web API port, and license)" option. When the option is selected, the port number in "applicationUrl" will be updated, otherwise it will not be changed.
License settings	ServerAPIs project > Server.json > "PowerServer" > "LicenseKey" and "LicenseCode"	Deploy	Determined by the "Overwrite server settings (DB connection, Web API port, and license)" option. When the option is selected, both "LicenseKey" and "LicenseCode" will be updated, otherwise they will not be changed.
Database Configuration window > the current DB connection profile	ServerAPIs project > AppConfig > AppConfig.json	Deploy	Overwritten all the time.
Database Configuration window > database caches configured in all DB connection profiles	ServerAPIs project > AppConfig > Applications.json or Applications.[DBConnectionProfile].json > "Connections" > "Default" > [cache name] ("Default" refers to the default cache group.)	Deploy	Determined by the "Overwrite server settings (DB connection, Web API port, and license)" option. New caches will be added regardless if this option is selected or not. If there are multiple caches, there will be multiple [cache name] blocks. When this option is selected, and if a cache with the same name is configured in the Database Configuration window, the corresponding [cache name] block will be overwritten; if there is no cache with the same name in the Database Configuration window, the [cache name] block will not be overwritten. When this option is not selected, all [cache name] blocks will not be overwritten.

Settings from the Web APIs tab	Will be updated to	By	Overwrite strategy
			By the way, if you create new cache groups (besides "Default"), these new cache groups will not be overwritten regardless if this option is selected or not.
Database Configuration window > transaction-to-cache mappings configured in all DB connection profiles	ServerAPIs project > AppConfig > Applications.json or Applications.[DBConnectionProfile].json > "Applications" > [app name] > "CloudTransactions" > [transaction name]	Deploy	Determined by the "Overwrite server settings (DB connection, Web API port, and license)" option. New transactions will be added regardless if this option is selected or not. If there are multiple transaction objects, there will be multiple [transaction name] blocks. When this option is selected, and if a transaction object with the same name is configured in the Database Configuration window, the corresponding [transaction name] block will be overwritten; if there is no transaction with the same name in the Database Configuration window, the [transaction name] block will not be overwritten. When this option is not selected, all [transaction name] blocks will not be overwritten.

10.3 Build & deploy using commands

Instead of building and deploying the PowerServer project from the PowerBuilder IDE, you can also build and deploy the project using the **PBAutoBuild210** command. For step-by-step guidance, refer to Tutorial 7: Building your PowerServer project with commands. The **PBAutoBuild210** command can integrate with Jenkins to automate the build and deployment process for PowerServer projects. Refer to the Jenkins user documentations for how to use Jenkins.

To build and deploy the PowerServer project with commands:

1. Export the configurations of the PowerServer project to the JSON file.

   1. Click the **Export PowerServer Build File** button ( ) in the toolbar if the PowerServer project painter opens, or right click the PowerServer project object and then select **Export Build File**.

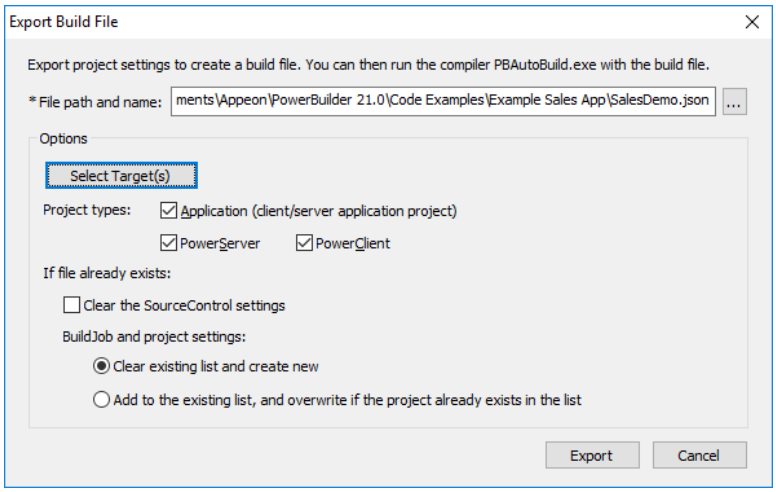
   2. In the **Export Build File** dialog box, specify where to save the exported file.

      If you right click the workspace and then select **Export Build File**, you can select one or more targets to export the build file for, and then select one or more project types to export if there are more than one type of project objects in the selected target(s). If you select more than one project type, the corresponding project objects

will be exported to the build file > "Projects" > [project object name]. If the selected targets contain project objects with the same name, only the object listed last will be added to the build file.

Specify how to overwrite the following settings if the build file already exists: the source code settings, build job settings, and project settings.

Figure 10.2:



3. Click **Export**.

The exported JSON file includes every single setting that is required for compiling, packaging and deploying the PowerServer project, for example,

- Project settings such as project type, platform (32-bit or 64-bit), build options etc.
- Library list
- Version information
- Run options
- ...

You can find a complete list of settings under the "Projects" block in the exported file.

The exported file also contains a "BuildPlan" block which provides additional configurations:

- "SourceControl" -- provides settings for downloading and merging source code from the source control server (including SVN, Git, and/or VSS).

  Merging will not only merge the source code but also upgrade the source code to the current version. However, it will not check or upgrade the PBD files used in the library list (you will need to replace the PBD files with the corresponding version).

- "BuildJob" -- contains the location of the selected PowerBuilder application target(s) and the name of the project object(s).

- Both the "SourceControl" and "BuildJob" blocks contain a "PreCommand" setting and a "PostCommand" setting which allow you to specify commands that can be executed before and/or after that particular block is executed.

  You can specify the commands or command file in "PreCommand" or "PostCommand". For example,

  ```
  "PreCommand": "SourcePre.bat"
  ```

  The commands in "PreCommand" and "PostCommand" can be executed in synchronous (default) or asynchronous mode, and the command window can be visible or invisible (default). For example,

  ```
  "PostCommand": "postcmd.bat /show /async"
  ```

Note: The relative path specified in the file is relative to the path of the JSON file.

2. Execute the **PBAutoBuild210.exe** file and the JSON file in a command line to automatically build and deploy the project. For example,

   ```
   PBAutoBuild210 /f c:\pssales.json /l deploy.log /le error.log /lu unsupport.log
   ```

   The **PBAutoBuild210.exe** file supports the following parameters:

   - /f -- specifies the configuration file. The configuration file (in JSON format) can be directly exported from the PowerBuilder IDE, as described in step 1.

     ```
     PBAutoBuild210 /f c:\pssales.json
     ```

   - /l -- writes the logging information to a file.

   - /le -- writes the error information to a file.

   - /lu -- writes the unsupported PowerScript features to a file. For example,

     ```
     PBAutoBuild210 /f c:\pssales.json /l deploy.log /le error.log /lu
      unsupport.log
     ```

     The relative path specified in the parameter is relative to the path of the configuration file. In the above example, the three log files will be generated under the same path as the configuration file.

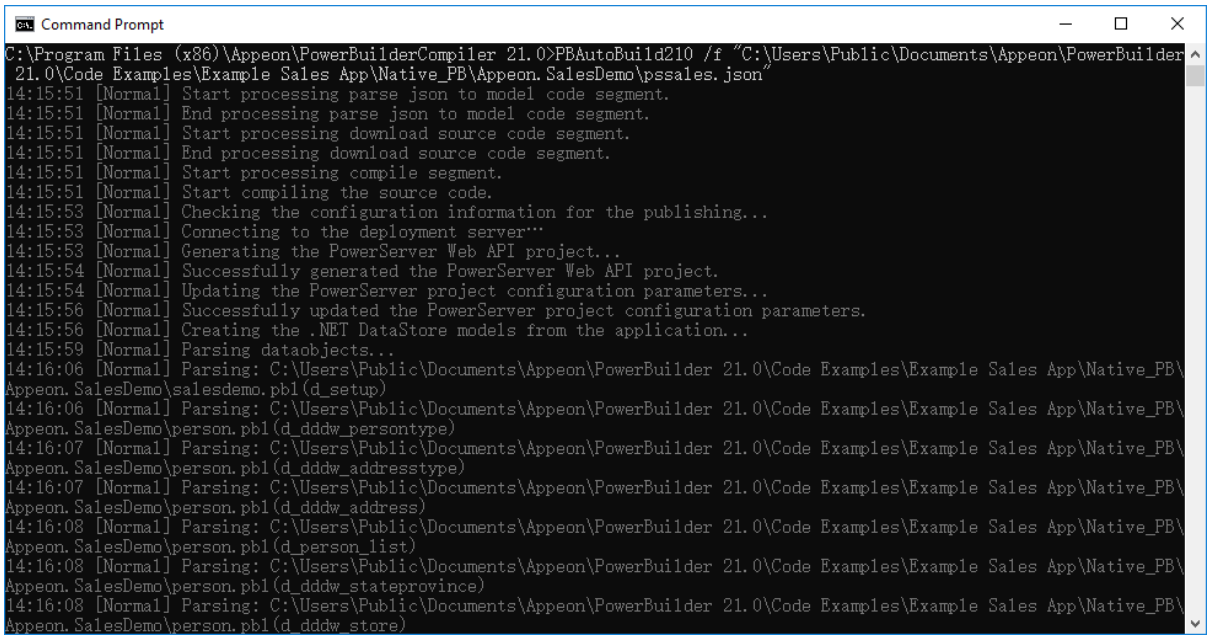
   - /p -- specifies the password for logging into SVN, Git, or VSS. This will generate an encrypted value based on the password. If the password contains the double quotation mark ("), use the escape character \" to replace ".

     ```
     PBAutoBuild210 /p 123456
     ```

   - /h or /? -- displays the help information.

     ```
     PBAutoBuild210 /h
     ```

Figure 10.3:



Note

The handling of PB.INI is the same in PBAutoBuild and OrcaScript:

If the application relies on a property in PB.INI to run, for example, [RichText] PageSizeAsControlSize=1, the user needs to copy the PB.INI file to the directory where the application executable resides.

If the compilation of PBAutoBuild or OrcaScript relies on a property in PB.INI, for example, [PB] DashInIdentifiers=0, the user needs to copy the PB.INI file to the directory where PBAutoBuild210.exe or pbc210.exe resides.

10.4 Run the ServerAPIs.Tests project

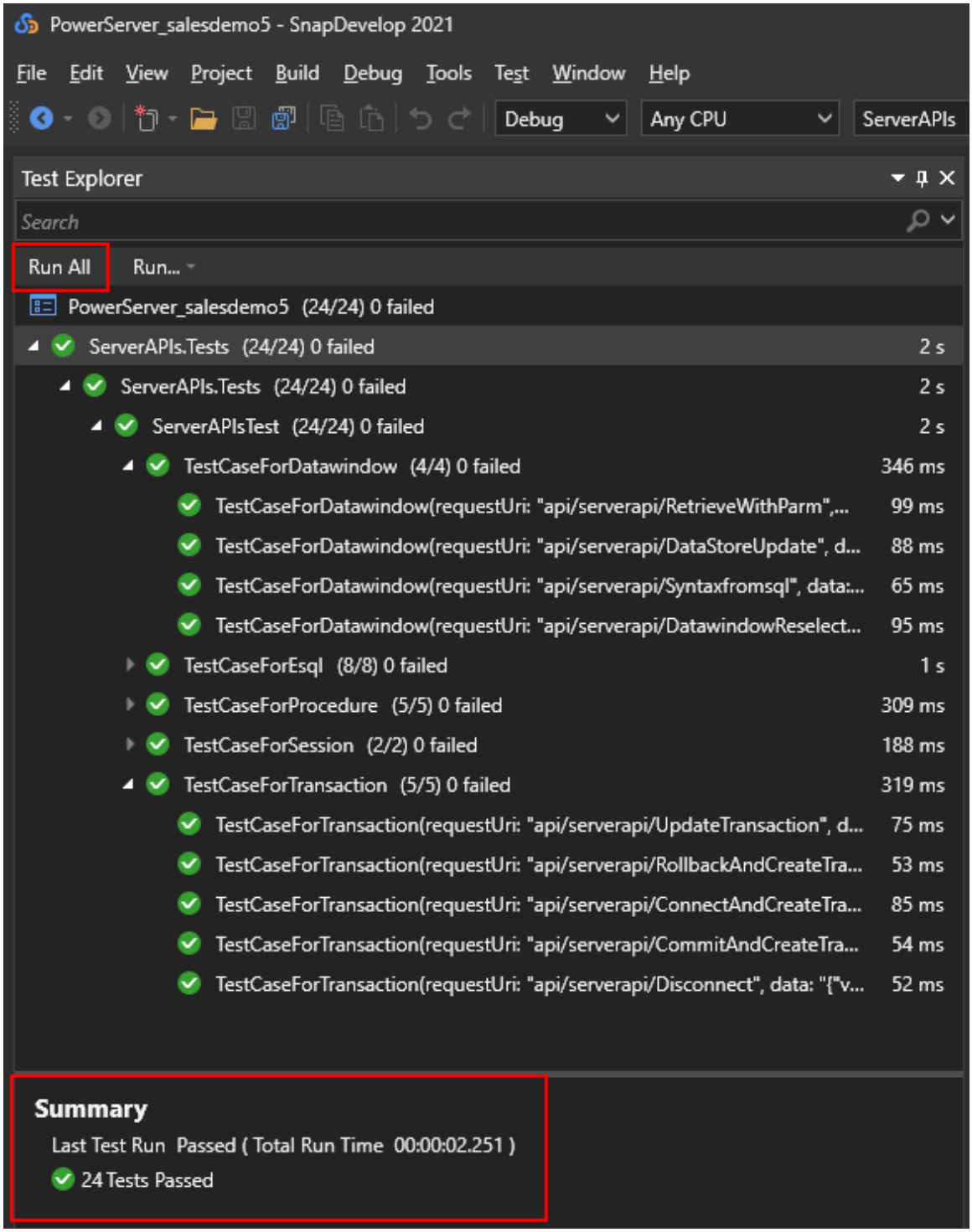
ServerAPIs.Tests is a unit test project generated with the PowerServer C# solution; it contains a number of XUnit.net tests which can check if the APIs in the **ServerAPIs** project can work correctly. The following categories of test cases are provided to check the corresponding APIs:

- DataWindows
- Embedded SQLs
- Stored procedures
- Sessions
- Transactions

To run the test case in the **ServerAPIs.Tests** project, select the **Test** > **Test Explorer** menu in the SnapDevelop IDE, and then click **Run All** in **Test Explorer** to run all the test cases, or right click the test case you want to run and then select **Run Selected Tests**.

Check the **Summary** window to make sure all tests have passed successfully.

Figure 10.4:



11 Compile and run the Web APIs

The PowerServer Web APIs is created during the build & deploy process and are ready to compile and run locally immediately after deployment.

Important

The **Compile & Run Web APIs** button ( ) in the toolbar can only run the Web APIs on the **LOCAL** machine, and detailed logs will be generated for development and debugging purpose. For optimal runtime performance, you can publish Web APIs to IIS instead of running Web APIs locally.

To compile and run the Web APIs:

1. Make sure your computer can connect to the NuGet site (https://www.nuget.org), so that the packages required for compiling and running the Web APIs can be successfully downloaded from the NuGet site.

2. Click the **Compile & Run Web APIs** button ( ) in the toolbar.

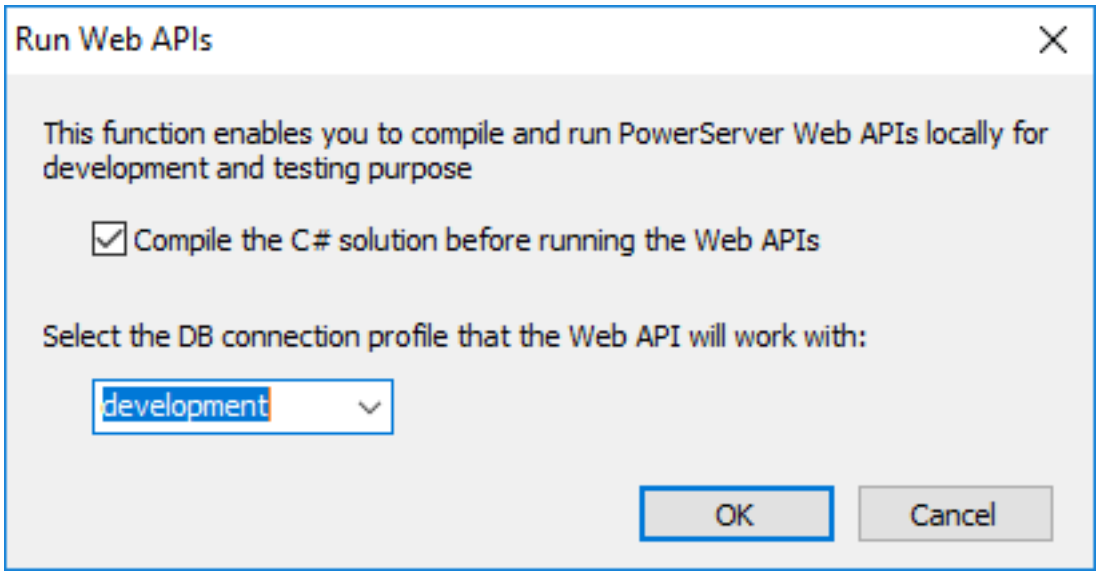
3. Select whether to compile the PowerServer C# solution before running the Web APIs (it is selected by default).

   For the first time to run the Web APIs, the compile option must be selected. After the successful compiling and running of Web APIs, this option can be de-selected to save time, unless the Web APIs project has been changed or an error has occurred.

4. Select the DB connection profile that the Web API will work with. The DB connection profile that is currently selected in the **Database Configuration** window will be selected by default.

5. Click **OK** to compile and run the Web APIs.

Figure 11.1:

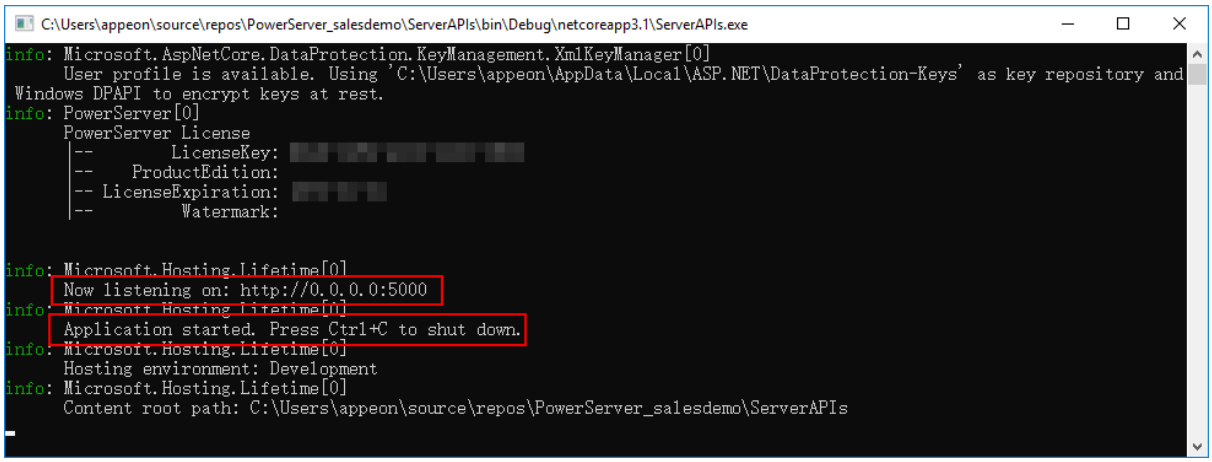


6. Check the **Output** window and make sure build is successful.

7. Make sure the API console window displays "Application started...". Also notice "Now listening on: http://0.0.0.0:5009" in the console window. This is the URL for accessing the Web APIs. The port number can be modified in the **ServerAPIs\Properties \launchSettings.json** in the PowerServer C# solution.

   When the installable cloud application is run later, you can view the logs in the console window to check if the requests and responses are processed successfully.

   Figure 11.2:

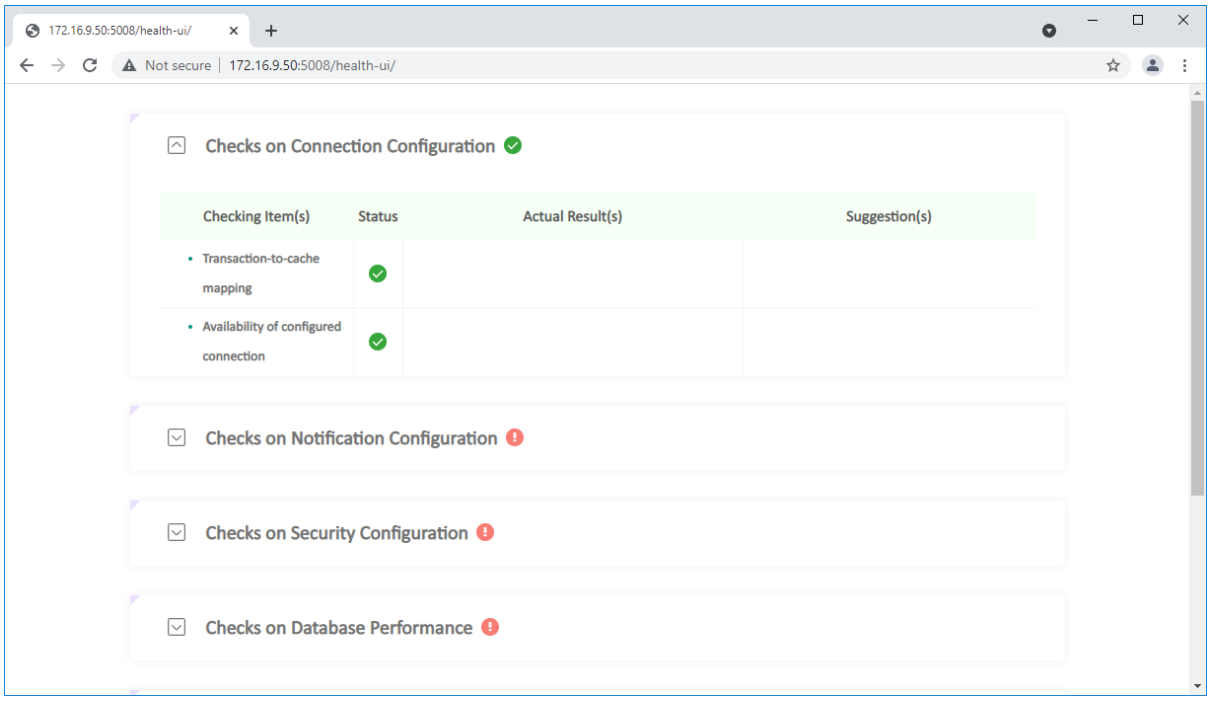


12 Check the status of Web APIs

When the Web APIs is running, you can check the health status of Web APIs by running http://[Web-API-URL]/health-ui in a Web browser, for example, http://localhost:5009/health-ui/.

Expand each block to view the details, especially the **Suggestion** section.

Figure 12.1:



13 Run the installable cloud application

Note: IE and Edge Legacy (EdgeHTML-based) browsers should not be used to run the installable cloud app, as Microsoft will end support for IE and Edge Legacy soon. You can use one of the following supported browsers: Chrome, Firefox, and the new Edge browser (Chromium-based).

- (For developers) Run the application by right-clicking the PowerServer project in the System Tree and then select **Run PowerServer Project**.

  Or click the **Run PowerServer Project** button ( ) in the toolbar. The **Run PowerServer Project** button will be available in the toolbar when the Project painter for PowerServer is opened; if more than one Project painter for PowerServer is opened, then the settings in the currently active painter will be used to run the application. And the application will be run in the Web browser or in the Cloud App Launcher according to the configurations in the **Run Options** tab in the painter. However, if Cloud App Launcher is not installed, then the default Web Browser will be run to install the Cloud App Launcher and run the application.

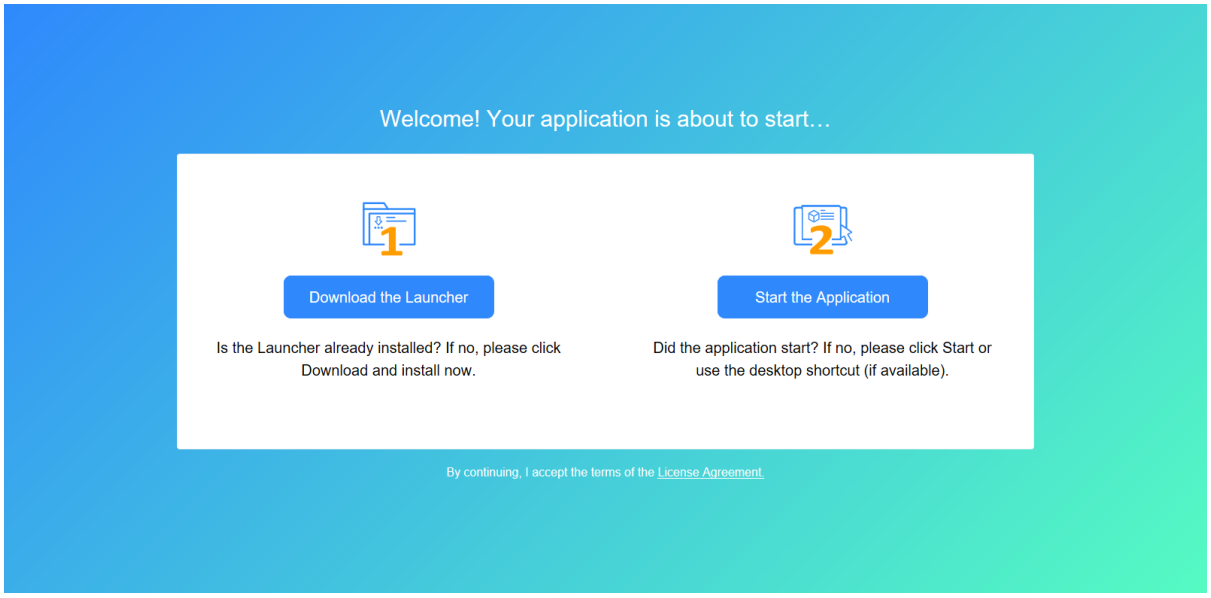
- (For developers and end users) Run the application in a Web browser for the first time.

  The user can input the application URL http://*IPAddress*/*AppName* in a Web browser to access the application. The IP address should point to the Web server where the app files are deployed. This URL can run the application with or without background process, depending on which startup mode the developer has selected as the default.

The cloud app launcher and the application must be installed through the Web browser for the first time. After that, users can directly double click the application icon on the desktop or the application shortcut on the Windows Start menu to run the application (the shortcut icon and menu are created by default unless the developer has changed the default settings in the Project painter for PowerServer).

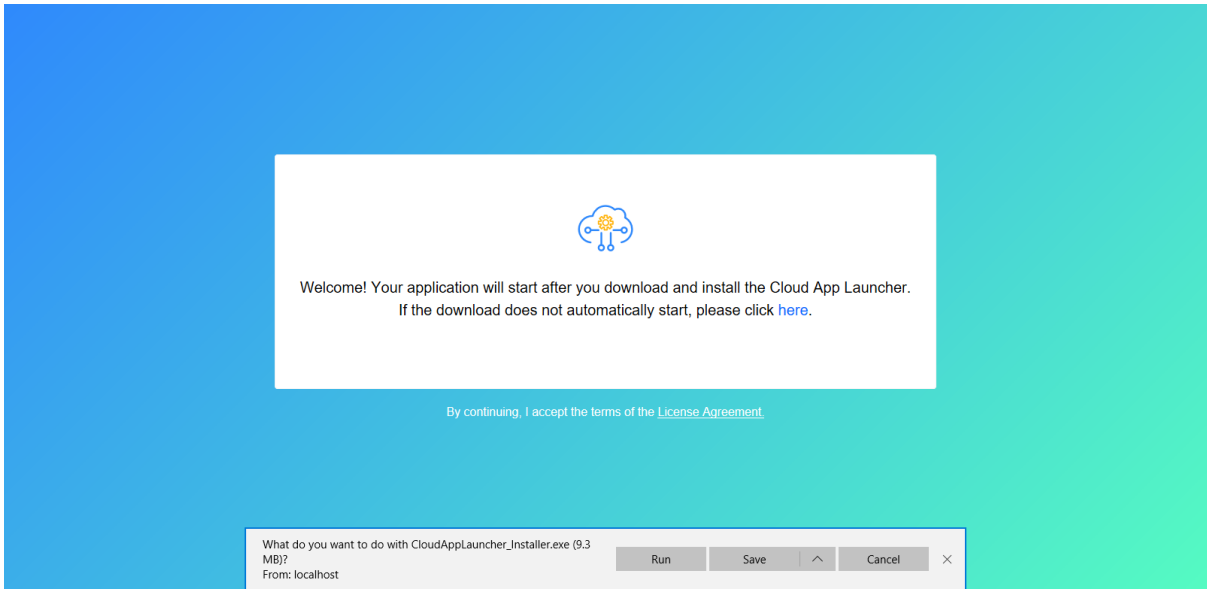
If the application is started without the background service, the user will be asked by the Web browser whether to run the app launcher. This is a browser behavior. Select **Allow**. Then the following app entry page displays. If the download does not start automatically, click **Download the Launcher** to download and install the cloud app launcher first, and then click **Start the Application** to download, install, and start the application.

Figure 13.1:



If the application is started with the background service, the following app entry displays. If download does not start automatically, click to download and install the cloud app launcher.

Figure 13.2:



You can view the logs in the API console window to check if the Web API requests and responses are successful.

Note

The virus-detection software McAfee WebAdvisor may block the CloudAppLauncher.exe file during the installation process. You can try adding the domain as a trusted site. To add the domain as a trusted site in McAfee WebAdvisor:

1. Right click the WebAdvisor add-on and select **Options**.
2. Under **Manage your trusted sites**, add the domain and click the + symbol.

3. Close and re-open the browser and run the installation again.

Note

If there is no response or progress when running the application, the CloudAppLauncher.exe file might be blocked by the Windows SmartScreen. You can try to turn off Windows SmartScreen in Control Panel > System and Security > Security and Maintenance > Change SmartScreen settings.

Note

Every time when the application launches, it needs to connect to the Web server to check updates, therefore, please make sure Web server is running and can be connected all the time.

14 Customize the app entry page

If you want to customize the license agreement and the visual displays (such as color, icon, text etc.) in the app entry page, you can make changes to the files under the %AppeonInstallPath%\PowerBuilder [version]\HTML folder, and then deploy the application again. The changes will apply to all applications deployed after the change is made.

Or you can directly make changes to the files under the application folder on the server, if you want to change that particular application only; but once you re-deploy that application, the changes will be lost.

- license.html is the template for license agreement.
- auto.html, autoconnect.html, autodownload.html, autoinit.html, and index.html are templates for applications started with background service.
- manual.html, manualconnect.html, manualdownload.html, and index.html are templates for applications started without background service.

Customize the loading animation

You can also deploy your own animation to replace the default animation (if you have selected "Show the loading animation before the app runs").

To deploy your own animation,

1. Prepare a GIF format of your animation and name the file as "**loading_ica.gif**". Only GIF format is supported currently.
2. Place "**loading_ica.gif**" under the same directory as the application target (.pbt) file.
3. Add "**loading_ica.gif**" under **Files preloaded as compressed packages** or **Files preloaded in uncompressed format** in the **External Files** page.

Important

If you have customized any file(s), it is strongly recommended that you manage these files separately, for example, back up the files somewhere to prevent file lost or overwritten after product upgrades or app deployments, or make files easily in sync if more than one developer will deploy the application.

---

15 Customize the deployed app using commands

When the application is deployed (from the PowerBuilder IDE) or installed (from the packaged executable installer or zipped file) to the Web server, the app files and config files are generated with hash codes, to prevent files changed illegally from running. Therefore, you cannot directly change the deployed settings/files on the Web server, instead you will have to make changes in the project painter and then deploy the application again, or modify the settings/files using commands (the **CustomizeDeploy.dll** tool).

The **CustomizeDeploy.dll** tool allows you to:

- Change the External Files -- The "External Files" refers to the packages, folders, and files (such as INI files, DLL/OCX etc.) that are deployed from the **External Files** tab of the PowerServer project painter.
- Change the Web API URL -- The Web API URL is stored on the Web server, so that the client knows where to call the PowerServer Web APIs at runtime. You may want to change the Web API URL value, if you want the client to call the PowerServer Web APIs running on a different URL.
- Encrypt the database password -- You can encrypt the database login password used in the PowerServer project painter > **Database Configuration**, or in the PowerServer C# solution > **ServerAPIs** project > **AppConfig** > **Applications.json** file. You can use the encrypted string instead of the plain-text string to protect sensitive information.

You can find the **CustomizeDeploy.dll** tool in the "1.01" sub-folder of the application folder after the application is deployed or installed to the Web server (either from the PowerBuilder IDE or from the packaged executable installer or zipped file).

Important

Prerequisites

To execute the **CustomizeDeploy.dll** file using the dotnet command, you will need to install the ASP.NET Core Runtime 3.1 or later.

To run CustomizeDeploy.dll in Windows Web server:

1. Install the ASP.NET Core Runtime 3.1 or later.
2. Open the command prompt. (You'd better run the command prompt using an administrator by right-clicking it and then selecting "Run as administrator").
3. Navigate to the Web server root folder > [application] folder > "1.01" (for example, C:\inetpub\wwwroot\pssales\1.01).
4. Execute the CustomizeDeploy.dll file using the dotnet command.

To run CustomizeDeploy.dll in Linux Web server:

1. Install the ASP.NET Core Runtime 3.1 or later.

```
dnf install aspnetcore-runtime-5.0
```

2. Go to the Web server root folder > [application] folder > "1.01" (for example, /var/www/html/pssales/1.01, or /usr/share/nginx/html/pssales/1.01).

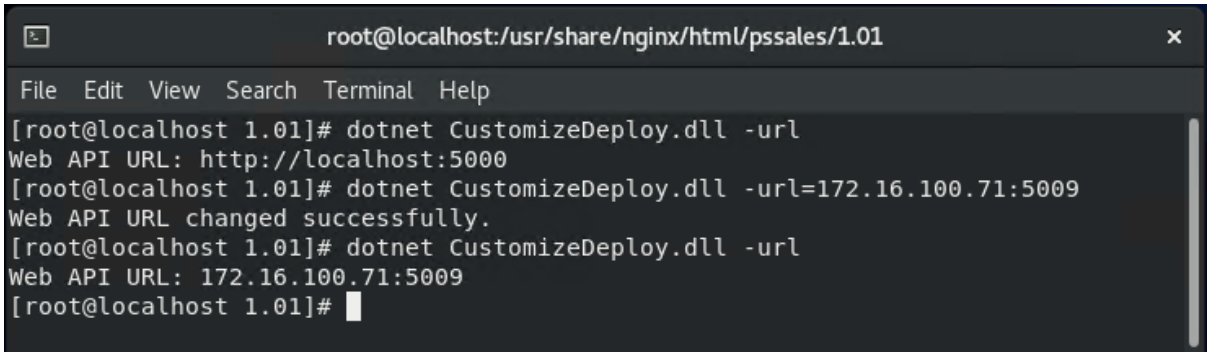
3. Right click the blank area within the folder and then select Open in Terminal.

4. Execute the CustomizeDeploy.dll file using the dotnet command. Notice that the command and file name are all case-sensitive in the Linux OS.

   For example,

   ```
   dotnet CustomizeDeploy.dll -url=https://172.16.100.71:5009
   ```

   Figure 15.1:

15.1 Change the External Files

To replace the External Files (such as INI, DLL/OCX etc.) for a deployed application:

1. Install the ASP.NET Core Runtime 3.1 or later.

2. Open the command prompt. (You'd better run the command prompt using an administrator by right-clicking it and then selecting "Run as administrator").

3. Navigate to the Web server root folder > [application] folder > "1.01" (for example, C:\inetpub\wwwroot\pssales\1.01).

4. Execute the CustomizeDeploy.dll file using the dotnet command.

   ```
   dotnet CustomizeDeploy.dll -src=<source file> -dest=<destination file>
   ```

   The "**src**" argument should point to the new file that you want to use to replace the old file.

   The "**dest**" argument should point to the old file that you want to replace with the new file.

   When external files are deployed to the server, they are appended with the file extension ".zip", but they are not compressed files (the only exception is package). For example, if **apisetup.ini** is selected in the External Files tab, it will be deployed as **apisetup.ini.zip** to the server, however, **apisetup.ini.zip** is not a compressed file and it can be directly opened in a text editor just like **apisetup.ini**.

   And to replace the file, you should prepare the source file without .zip extension.

   For example,

```
dotnet CustomizeDeploy.dll -src=/new/apisetup.ini -dest=apisetup.ini.zip
```

Or

```
dotnet CustomizeDeploy.dll -src=/new/new.ini -dest=apisetup.ini.zip
```

The only exception is the package which is indeed compressed as the zip format (with file extension ".zip.zip"). Therefore, you should prepare the source file for the package in the compressed zip format.

For example,

```
dotnet CustomizeDeploy.dll -src=/new/theme.zip -dest=theme.zip.zip
```

Or

```
dotnet CustomizeDeploy.dll -src=/new/aaa.zip -dest=theme.zip.zip
```

The tool will replace the package as a whole (and refresh the hash code of the package) and it will not validate the individual files included in the package. Therefore you need to make sure the files included in the package are correct and complete.

15.2 Change the Web API URL

To change the Web API URL for a deployed application:

1. Install the ASP.NET Core Runtime 3.1 or later.
2. Open the command prompt. (You'd better run the command prompt using an administrator by right-clicking it and then selecting "Run as administrator").
3. Navigate to the Web server root folder > [application] folder > "1.01" (for example, C:\inetpub\wwwroot\pssales\1.01).
4. Execute the CustomizeDeploy.dll file using the dotnet command.

   - Syntax 1

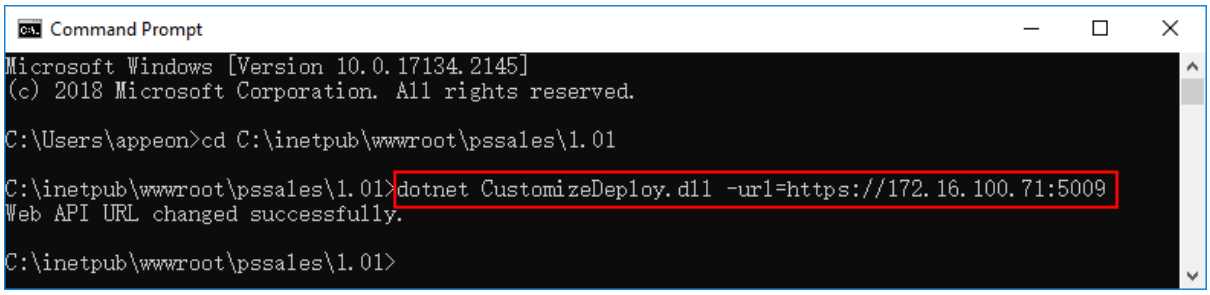
     ```
     dotnet CustomizeDeploy.dll -url=<URL>
     ```

     The "**url**" argument should point to the new Web API URL that you want to change to.

     For example

     ```
     dotnet CustomizeDeploy.dll -url=https://172.16.100.71:5009
     ```

     Figure 15.2:



- Syntax 2

```
dotnet CustomizeDeploy.dll -url
```

If the "**url**" argument is not set with any value, it will get the current URL. Therefore, you can run this command to check what the current Web API URL is.

Figure 15.3:

```
C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -url
Web API URL: https://172.16.100.71:5009

C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -url=https://www.appeon.com:6666
Web API URL changed successfully.

C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -url
Web API URL: https://www.appeon.com:6666
```

15.3 Encrypt the database password

To encrypt a password, you can execute the CustomizeDeploy.dll file located in the PowerBuilder IDE installation folder (%AppeonInstallPath%\PowerBuilder [version]\Pstools \CustomizeDeploy) or in the deployed application on the Web server.

To execute the CustomizeDeploy.dll file located in the PowerBuilder IDE installation folder, there is no need to install the ASP.NET Core Runtime as it is already installed with PowerServer Toolkit.

To execute the CustomizeDeploy.dll file located in the deployed application on the Web server, you will need to install the ASP.NET Core Runtime first, as explained in the following steps.

To encrypt a password (such as the database login password) for the database connection cache:

1. Install the ASP.NET Core Runtime 3.1 or later.

2. Open the command prompt. (You'd better run the command prompt using an administrator by right-clicking it and then selecting "Run as administrator").

3. Navigate to the PowerBuilder IDE installation folder (%AppeonInstallPath% \PowerBuilder [version]\Pstools\CustomizeDeploy) or Web server root folder > [application] folder > "1.01" (for example, C:\inetpub\wwwroot\pssales\1.01).

4. Execute the CustomizeDeploy.dll file using the dotnet command.

```
dotnet CustomizeDeploy.dll -encrypt=<string>
```

```
dotnet CustomizeDeploy.dll -encrypt=<string> -outfile=<output file>
```

The "**encrypt**" argument should be set to the value that needs to be encrypted.

The "**outfile**" argument can save the encrypted value in the specified location and file.

Figure 15.4:

```
C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -encrypt=123456
eyJQYX1sb2FkIjoibVp1am5HXHUwMDJCNUY3UnZJZW1iejh6XHUwMDJCR1E9PSIsI1RpbWVzdGFtcCI6MTYyNzQ0MDg
wNywiU21nbmF0dXJ1IjoiYk5tSE5vb1IvcWViSUdhXHUwMDJCdmZXZ1RwQ1NSRG02WWVsWkpjZHR4MjJMaFU3S1E5TG
9ZNi9KQWNGcFBWN0RqbUx5NjJWR1dpY3pzbnY4UFx1MDAyQjI0d29vbXF3PT0ifQ==

C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -encrypt=123456 -outfile=pwd.txt

Encryption succeeded.
```

Figure 15.5:

pwd.txt - Notepad

File Edit Format View Help

eyJQYXlsb2FkIjoiclE0eWNLd3J2V3JMOXNIMlRxdEZhZz09IiwiVGltZXN0YW1wIjoxNjI3NDQxMDc3LCJTaWduYXR1cmUi
OiJQdjBcdTAwMkJ6WWc0Wjh6dERzRTdGb0M0bUN1QmZSSG9OUHZvRnptZEFZb3liaHZzMWwwZ0tmeXVveVM0em1J
eWhCakRQbFJWbXViWEZZMkx6MXJjcGVYcUVBPT0ifQ==

When creating a database connection cache in the PowerServer project painter > **Database Configuration**, or in the PowerServer C# solution > **ServerAPIs** project > **AppConfig** > **Applications.json** file, instead of inputting the plain text of the database login password, you can encrypt it using the above command, and then input the encrypted value instead of the plain text.

Note

If special characters are contained in the value, use double quotation marks (in Windows) or single quotation marks (in Linux) to include the entire value. For example,

In Windows, to encrypt the value *post!gres*

```
dotnet CustomizeDeploy.dll -encrypt="post!gres"
```

If the quotation mark is contained as part of value, then place the escape character \ before the quotation mark. For example, to encrypt the value *postgr”es*

```
dotnet CustomizeDeploy.dll -encrypt="postgr\”es"
```

---

16 Support cookie validation

You can now set a cookie to the cloud app launcher and the application; and the cookie will be automatically carried in the HTTP request header of every client request.

Once a cookie is set to the cloud app launcher and the application, the cookie can be validated against the validation scripts or the SSO server etc. And based on the validation results, the launcher and/or the application can be determined whether to allow to download the requested files and/or connect with the database.

Notice that

1. Currently you can only set the name and value for a cookie, and cannot set the other cookie attributes (including Domain, Expires, Path etc.); and you must set the cookie in the key-value pairs, for example, "key1=value1; key2=value2".

2. The cookie must be passed into the launcher and the application by the index.html file, therefore, you will have to start the application from the index.html (by accessing the app URL in the Web browser); you cannot start the application from the app shortcut on the desktop or start menu.

3. Make sure the cookie will stay valid if you select "Download the app files as necessary" because files will be downloaded only when requested. Set an appropriate expiration period for the cookie.

To set a cookie to the cloud app launcher and the application,

You can modify the JavaScript file (launcher.js) on the Web server to set a cookie to the cloud app launcher.

launcher.js is located in the application folder > "js" sub-folder on the Web server, for example, C:\inetpub\wwwroot\salesdemo_cloud\js\launcher.js.

For example, the following JavaScript will set the cookie by obtaining the cookie from document.cookie. However, if the cookie is set to HttpOnly, it cannot be accessed from document.cookie by JavaScript.

```
function getCookie(){
 var strCookie = "";
 strCookie = document.cookie;
 return strCookie;
}

function getCmdline(Url){
 var strCookie = getCookie();
 var strUrl = Url;
 if(strCookie.length > 0)
 {
  strUrl += " -cookie ";
  strUrl += strCookie;
 }
 return strUrl;
}
```

For example, the following JavaScript will set the cookie by obtaining the cookie from the application URL, for example, http://localhost:5000/test?name=admin;pw=123.

```
function getCookie(){
```

```
 var strCookie = "";
 strCookie = window.parent.parent.location.search.split("?")[1];
 return strCookie;
}

function getCmdline(Url){
 var strCookie = getCookie();
 var strUrl = Url;
 if(strCookie.length > 0)
 {
  strUrl += " -cookie ";
  strUrl += strCookie;
 }
 return strUrl;
}
```

17 View the API documentation

The documentation for PowerServer Web APIs is formatted using the OpenAPI Specification (formerly Swagger Specification). Each API is described with the operations (GET and POST) and the operation parameters; and developers can easily try out and adopt the API.

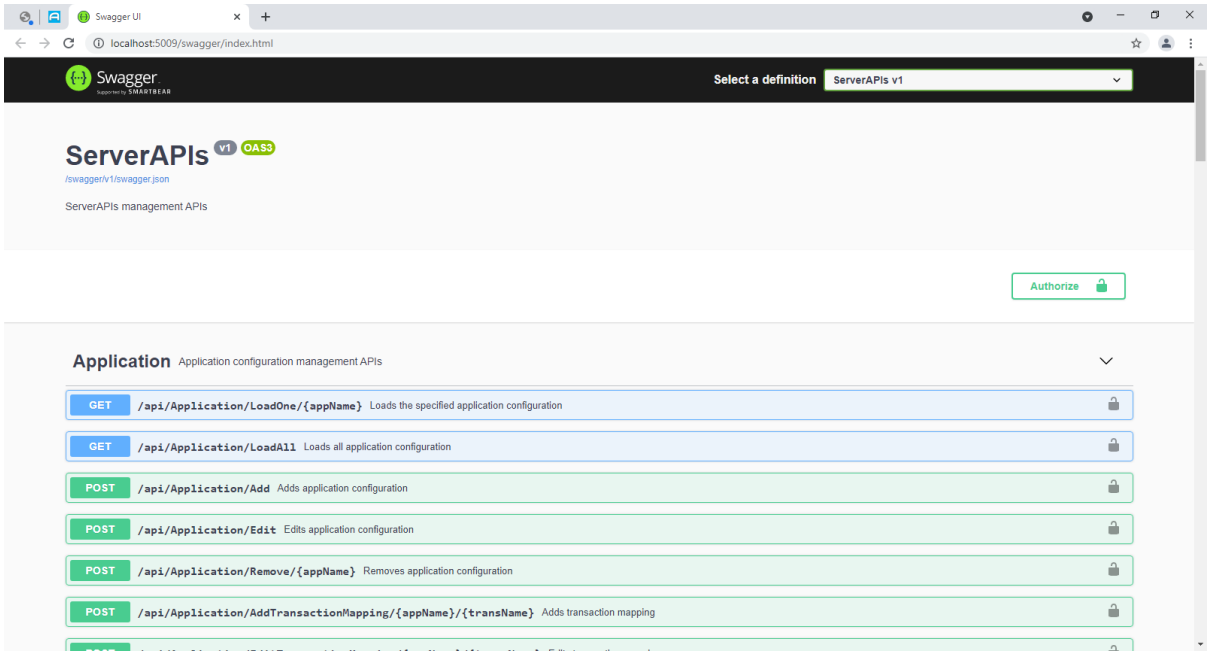
The PowerServer Web APIs here mainly refers to the management APIs in the **ServerAPIs** project > **Controllers** folder.

- **ApplicationController.cs**: This file provides APIs for dynamically adding, modifying or removing the application settings.
- **ConnectionController.cs**: This file provides APIs for dynamically adding, modifying or removing the database connections such as cache or cache group.
- **LicenseController.cs**: This file provides APIs for dynamically accessing the license information.
- **SessionController.cs**: This file provides APIs for getting all user sessions or killing a particular user session. For more information, see Get/Kill user sessions.
- **StatisticsController.cs**: This file provides APIs for getting statistics of the request and transaction.
- **TransactionController.cs**: This file provides APIs for getting all transactions or rolling back a particular transaction.

To view the API documentation, run the **ServerAPIs** project (by clicking **Run** in the PowerServer C# solution).

The Swagger UI for the API documentation will be launched automatically in the Web browser. However, the Swagger UI may not be successfully loaded until the **ServerAPIs** project completes all the startup process. You may see Swagger UI refresh a few times before the API documentation is successfully loaded; or you may need to refresh the browser to load the API documentation.

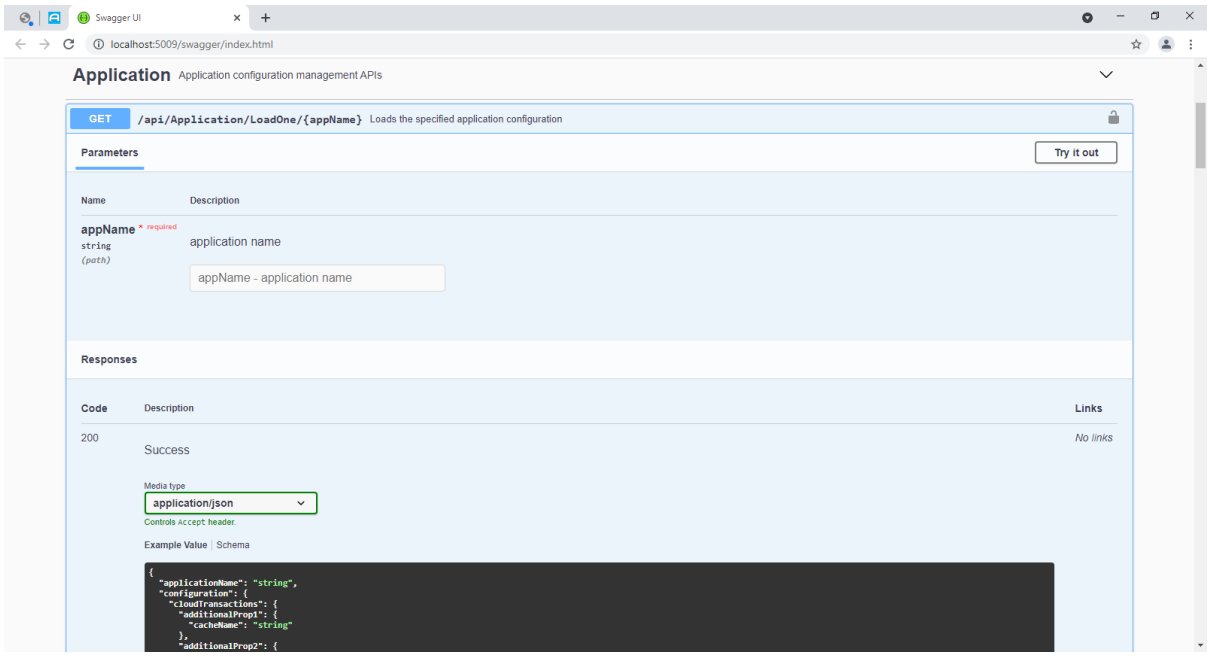
Figure 17.1:



By expanding each API, you can view the details of the API and try out the API (by clicking the **Try it out** button on the top right corner).

For code examples of calling the API in PowerScript, refer to Get/Kill user sessions.

Figure 17.2:



18 Get/Kill user sessions

You can use the following functions or APIs to manage user session(s) for the installable cloud application:

- The **GetSessionID** function of the PowerBuilder Application object -- Gets the session ID of the current client.
- The **LoadAll** API provided by the SessionController.cs file in the PowerServer C# solution -- Gets the session IDs of all clients. Refer to View the API documentation for how to view the documentation of this API.
- The **KillByID** API provided by the SessionController.cs file in the PowerServer C# solution -- Kills the session(s) according to the ID. Refer to View the API documentation for how to view the documentation of this API.

To get the session ID of the current application client, you can write PowerScripts as below:

```
String ls_SessionID
ls_SessionID = Getapplication().GetSessionID()
```

To get all user sessions, you can write PowerScripts as below:

```
//-------------------------------------
LoadAll-----------------------------------------

httpclient  lhc_client
string ls_url
string ls_json

lhc_client = create httpclient

//GetSessions
ls_url = "http://localhost:5000/api/Session/LoadAll"
//This URL should be replaced with the actual IP address and port number of
 PowerServer Web APIs
//If there are multiple .NET servers, obtain one by one
//lhc_client.SetRequestHeader("Authorization", $token, true)  //If authentication
 is enabled
lhc_client.sendrequest("Get",ls_url)

if lhc_client.getresponsestatuscode() = 200 then
                lhc_client.getresponsebody(ls_json)
                //parse the json
                //wf_getsessions(ls_json)
end if

//-------------------------------------------------------------------------------------------------
```

To kill the specified user session, you can write PowerScripts as below:

Step 1: Get all the user sessions first.

Step 2: Kill the specified session according to the session ID.

The session information returned will look like this:

```
//
[{"sessionid":"8e3f5c6d-7515-4377-9a45-0e3349fcbfd2","application":"SalesApp","sessionstate":"Act
```

```
//-------------------------------------
KillByID-----------------------------------------------

httpclient  lhc_client
string ls_url
string ls_sessionid

lhc_client = create httpclient

//GetSessions
//lhc_client.SetRequestHeader("Authorization", $token, true)  //If authentication
 is enabled
ls_url = "http://localhost:5000/api/Session/KillById"
//This URL should be replaced with the actual IP address and port number of
 PowerServer Web APIs

ls_sessionid = "8e3f5c6d-7515-4377-9a45-0e3349fcbfd2"
ls_url += "/"+ls_sessionid

lhc_client.sendrequest("post",ls_url)

if lhc_client.getresponsestatuscode() = 200 then
                messagebox("succeed",ls_sessionid +" was killed")
end if

//-------------------------------------------------------------------------------------------------
```

19 Package the client app

When deploying the PowerServer project as an installable cloud app, you can choose to package the client-side as an executable installer or a zipped file, and then install the client to the Web servers.

To package the client app:

1. Go to the **Client Deployment** tab of the PowerServer project painter, and then click **Package the compiled app and manually deploy later**.

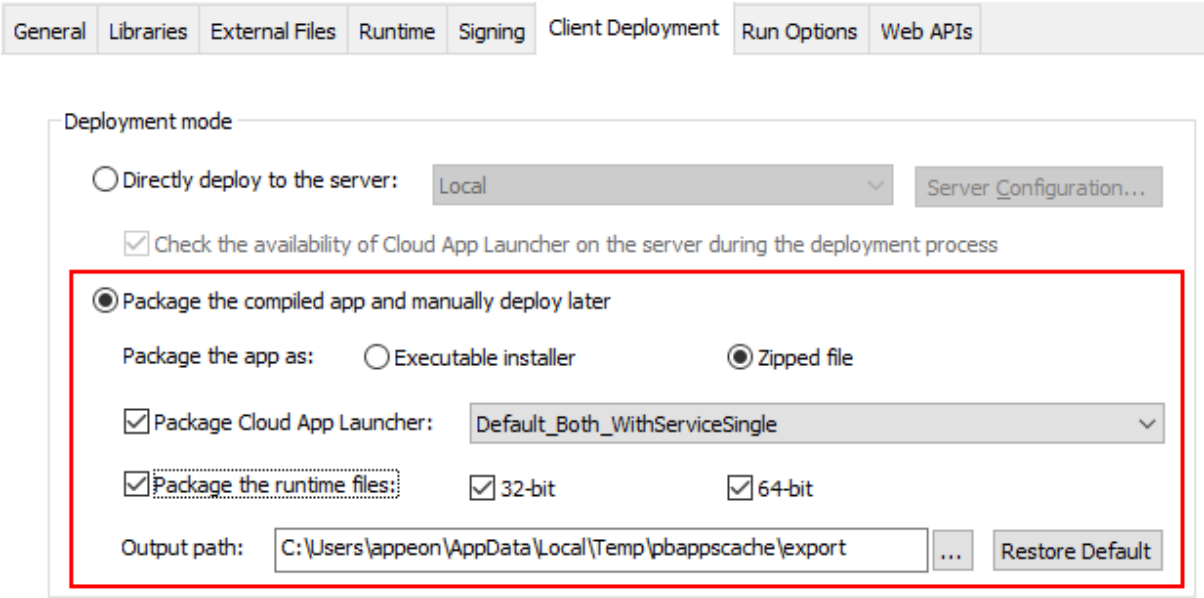
2. Specify to generate the package as an executable installer or a compressed zip file, and select whether to package the cloud app launcher and the PowerBuilder Runtime files.

   If you select **Zipped file**, an *appname*_Installer.zip file is generated in the specified path. You can copy the zip file to the server and then decompress it to the Web root.

   If you select **Executable installer**, an *appname*_Installer.exe file is generated in the specified path. You can copy the executable file to the server and then run it to install the application to the Web root.

3. Specify the location where the package will be generated.

   Figure 19.1:



4. Save the project settings and then click the **Build & Deploy PowerServer Project** button ( ) or **Deploy PowerServer Project** ( ) button in the toolbar to generate the package.

Note

Do not manually change the name of the installed or de-compressed application folder on the server, otherwise the application uninstall program will fail to run.

---

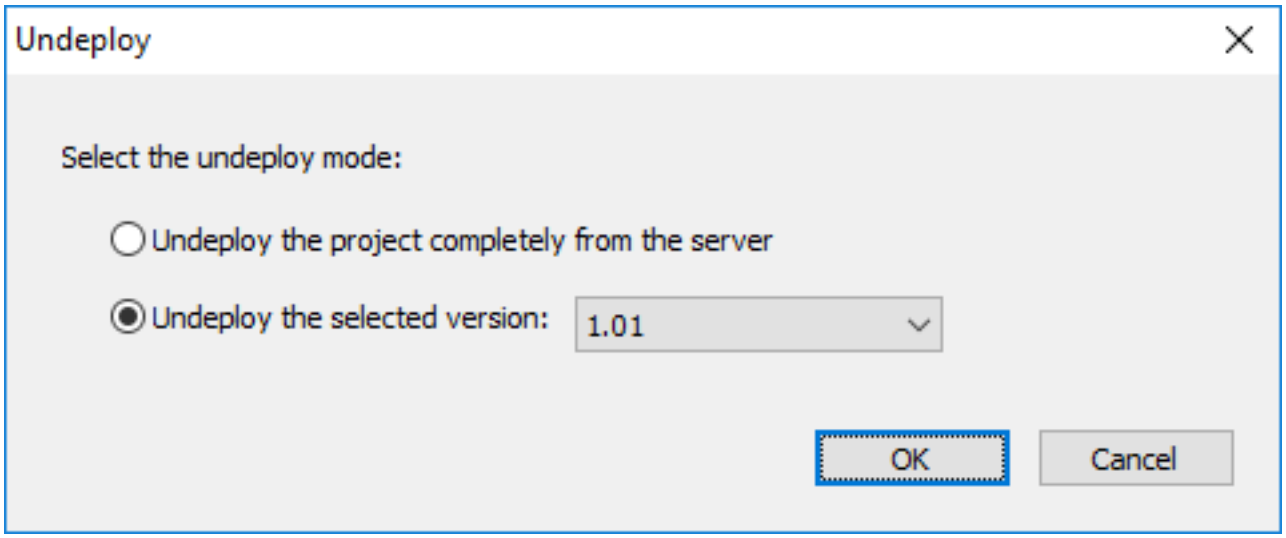
20 Undeploy the client app

To undeploy the client app from the server:

1. Right-click the PowerServer project in the System Tree and then select **Undeploy PowerServer Project** from the popup menu.

2. Select whether to remove the entire project (all deployed versions) or only the selected version from the server.

Figure 20.1:



21 Uninstall the client app

To uninstall the client app from the client machine:

1. Uninstall the application by selecting the **Uninstall** shortcut menu from the Windows Start | [appname]. The **Uninstall** shortcut menu and the [appname] menu are available only when the developer selected to create the **Start menu shortcut** and **App uninstall shortcut** in the **Run Options** page of the Project painter for PowerServer.

   If the **Uninstall** shortcut menu is not available, you can run the **Uninstall.exe** file in the application folder, for example, %AppData%\PBApps\Applications\localhost_pssales \Uninstall.exe (%AppData%\PBApps is configurable when uploading the Cloud App Launcher and runtime files).

   Note: If the application folder name (which is named after [appname]) on the server has been changed manually, the application uninstall program will fail to run.

   The uninstall program will automatically remove the following:

   - The application shortcuts on the desktop and the Windows start menu.
   - The application folder under %AppData%\PBApps\Applications, for example, %AppData%\PBApps\Applications\localhost_pssales.

     The application folder contains all of the application files and any external files (such as UI theme files, DLLs/OCXs, images/videos, INIs etc.) that are deployed with PowerServer. This folder will be automatically deleted during the uninstall process.

   However, the uninstall program will **NOT** automatically remove the following:

   - The registration information of DLL/OCX files in the Windows registry.

     If you have selected to register the DLL/OCX files (using Regsvr32 by default), you will need to remove the registry information manually. Follow instructions in step 4 below.
   - The runtime files under %AppData%\PBApps\Applications\Runtime.

     The PowerBuilder Runtime files are used by all deployed apps on the client machine. You can manually delete the runtime files if they are no longer used.
   - The download folder under %AppData%\PBApps, for example, %AppData% \PBApps\Download.

     This folder stores the download statistics of the app and runtime files. It can be manually deleted.

2. Uninstall the cloud app launcher by uninstalling **Cloud App Launcher** from Control Panel\Programs\Programs and Features.

   If the cloud app launcher without background service is uninstalled, the %LocalAppData %\Launcher folder will be removed.

   If the cloud app launcher with background service is uninstalled, the %LocalAppData% \LauncherWithService folder will be removed.

3. Uninstall the cloud app service by uninstalling **Cloud App Launcher Service** from Control Panel\Programs\Programs and Features.

   The cloud app launcher service is installed only when the launcher with background service which supports multiple Windows users is installed.

4. Remove the registry information of DLL/OCX files.

   The registry information of DLL/OCX files (or any other files that are installed and registered by your own) will not be automatically removed during the application uninstall process.

   To clean up the registry information of the DLL/OCX files, you can write scripts (a sample shown below) and place them in a file named **ManualUninstall.cmd**, place the **ManualUninstall.cmd** file under the same directory as the application target (.pbt) file, add **ManualUninstall.cmd** under **Files preloaded as compressed packages** or **Files preloaded in uncompressed format** in the **External Files** page, and then deploy the application.

   The scripts in **ManualUninstall.cmd** will be automatically run when the application uninstall program is run. (If the file requires administrator rights to unregister, you should run the application uninstall program with administrator rights.)

   You can also add scripts in **ManualUninstall.cmd** to clean up any other files that are installed and registered by your own.

   The following is a sample script for unregistering DLL/OCX files that are registered by Regsvr32:

```
set Driver=%~d0
set HOMEDIR=%~dp0
%Driver%
cd %HOMEDIR%
regsvr32 /u .\dllname
regasm   /unregister .\AssemblyName
```

Tutorials

Contents

1 Tutorial 1: Deploying your PowerServer project to production environment

1.1 Overview

In the Quick Start guide, we use the simplest scenario (all roles in the local development environment) to quickly get started with the PowerServer deployment; now in this tutorial, we will walk through the deployment process in a more production-like environment, using individual machines as the client, development PC, Web server, database server, and .NET server.

It is recommended that before you go through this tutorial you have a successful result with the Quick Start guide first, so that you have basic concepts of the whole deployment process.

1.2 Task 1: Setting up the client machine

Set up the client machine with the following OS and software:

- Windows 10
- Google Chrome

  This tutorial takes Google Chrome as an example. You can also use Firefox or the new Edge browser (Chromium-based).

1.3 Task 2: Setting up the database server

1.3.1 Preparations

In this tutorial, we will set up the **salesdemo** SQL Anywhere database server running in an independent machine.

Set up the database server with the following OS and software:

- Windows Server 2019 (64-bit)
- SQL Anywhere 17

  Click here to download the installer for the free trial of SQL Anywhere developer edition.

  This tutorial takes SQL Anywhere database as an example. You can also install the other databases by following the documentation from the vendor.

  Note that the SQL Anywhere database can only be connected through an ODBC driver. You will need to create the same ODBC data source in both the development PC and the .NET server. The data source in the development PC is for converting DataWindows to models, and the data source in the .NET server is for accessing data from the database. The following sections have detailed instructions for how to create the ODBC data source.

  Important

  For optimal runtime performance, it is highly recommended that the PowerServer Web APIs should be published to a server that locates on the same LAN as the

database server. If the database is not on the same network as the Web APIs, every request has to go a long way from PowerServer to the database, it is highly possible that there will be performance and security issues.

1.3.2 Configuring Windows Defender Firewall

If Windows Defender Firewall or any antivirus tool is turned on, make sure to configure them to allow the database server port (**2638** in this tutorial or the port number you choose to use) to go through, otherwise, connection errors may occur.

You would need to configure the firewall/antivirus settings on the following servers:

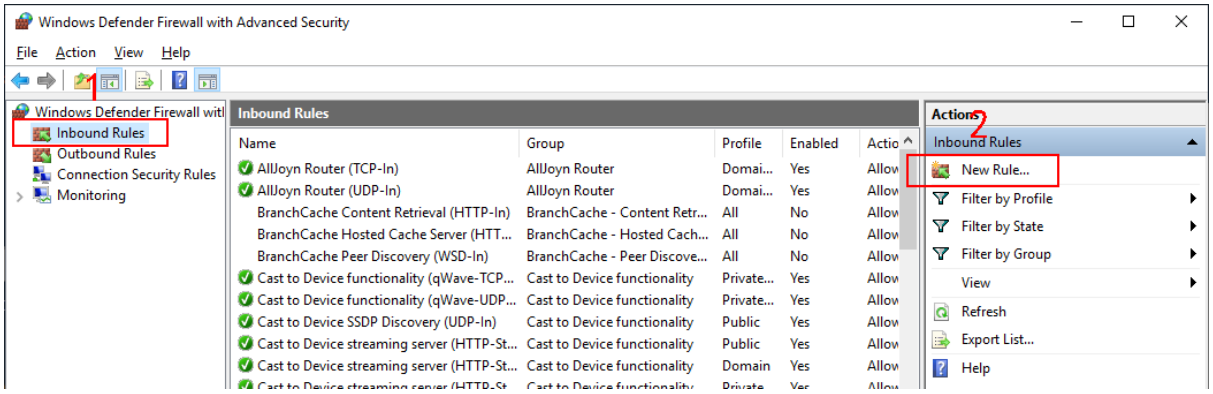
- Database server -- to allow the database server port (2638 in this tutorial)
- .NET server -- to allow the .NET server port (5009 in this tutorial)
- Web server -- to allow the FTP server port (21 in this tutorial) (this only affects the FTP connection during the app deployment)

The following steps configure the firewall settings on the database server (you can take the same steps to configure the other servers):

Step 1: Open Windows Defender Firewall and then click **Advanced settings**.

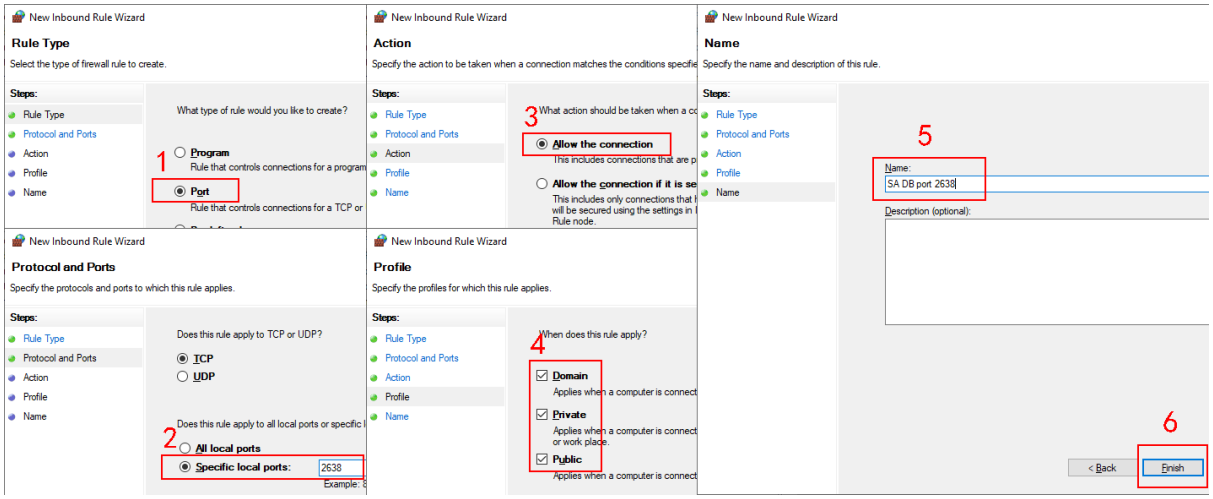
Step 2: Select **Inbound Rules** and then click **New Rule**.

Figure 1.1:



Step 3: Create a new rule which allows connections through the database server port (2638 in this tutorial).

Figure 1.2:



1.3.3 Starting the database

Step 1: Download the database file (**pbdemo2021_for_sqlanywhere.zip**) from https://github.com/Appeon/PowerBuilder-Project-Example-Database.

Or copy the database file (**pbdemo2021.db**) from the PowerBuilder demo installation folder (%Public%\Documents\Appeon\PowerBuilder 21.0\) to the database server, if you have installed PowerBuilder IDE according to Task 4.

Step 2: Start this database file on the **salesdemo** database server using **SQL Anywhere Network Server** (dbsrv17.exe).

The database server must be started as a network server (not personal server) in order to support network connections.

```
"C:\Program Files\SQL Anywhere 17\Bin64\dbsrv17.exe" -x tcpip(port=2638) -n
 salesdemo "C:\DB\pbdemo2021.db"
```

1.4 Task 3: Setting up the Web server

1.4.1 Overview

The client-side of the installable cloud app can be hosted in the following Web servers:

- Windows IIS
- Windows/Linux Apache
- Windows/Linux Nginx

This tutorial will take Windows IIS as an example. For detailed instructions of the other Web servers, refer to Setting up a Web server.

1.4.2 Preparations

In this tutorial, we will set up a Web server and an FTP server running on the same IIS instance.

Step 1: Set up the Web server with the following OS and software:

- Windows Server 2019 (64-bit)
- Microsoft IIS

  The next section Installing Web Server (IIS) has detailed installation instructions.

Step 2: Configure Secure Sockets Layer (SSL) for the Web server, so that HTTPS can be used to secure the connections between the client and the Web server.

For how to configure SSL on IIS, refer to https://docs.microsoft.com/en-us/iis/manage/configuring-security/how-to-set-up-ssl-on-iis.

Step 3: Configure Windows Defender Firewall on the Web server to allow the FTP port (21 in this tutorial).

The section "Configuring Windows Defender Firewall" has detailed instructions.

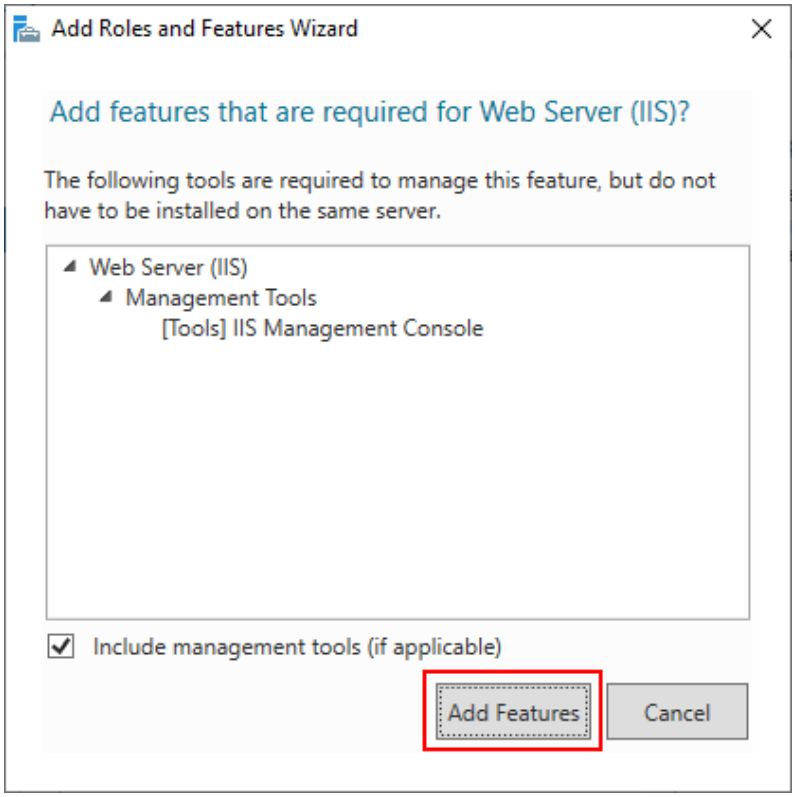
1.4.3 Installing Web Server (IIS)

Step 1: In Windows Server 2019, open **Server Manager**, and then select **Add roles and features**.

Step 2: In the **Add Roles and Features Wizard**, click **Next** several times until the **Server Roles** section displays.

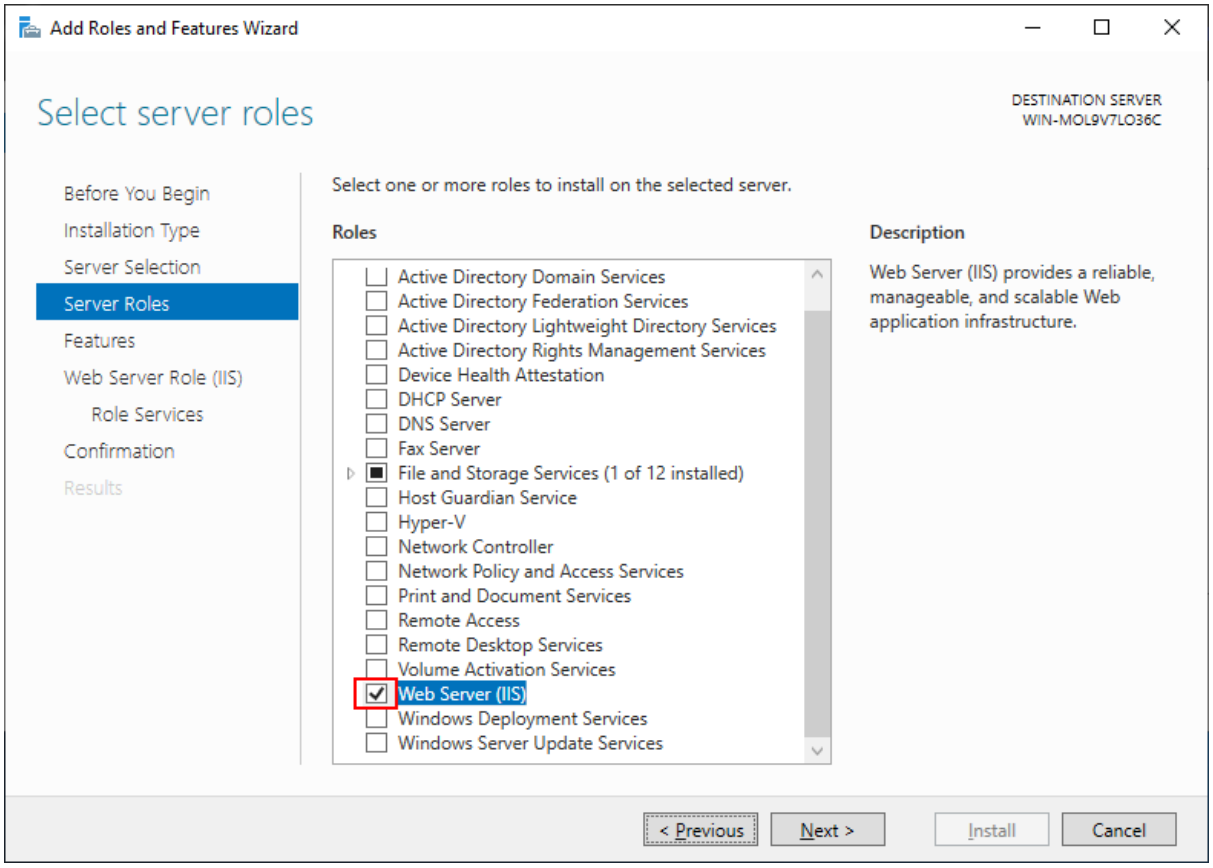
Step 3: Click the check box of **Web Server (IIS)**; and then click **Add Features** when asked whether to add features required for Web server.

Figure 1.3:



Step 4: Make sure the check box of **Web Server (IIS)** is selected.

Figure 1.4:

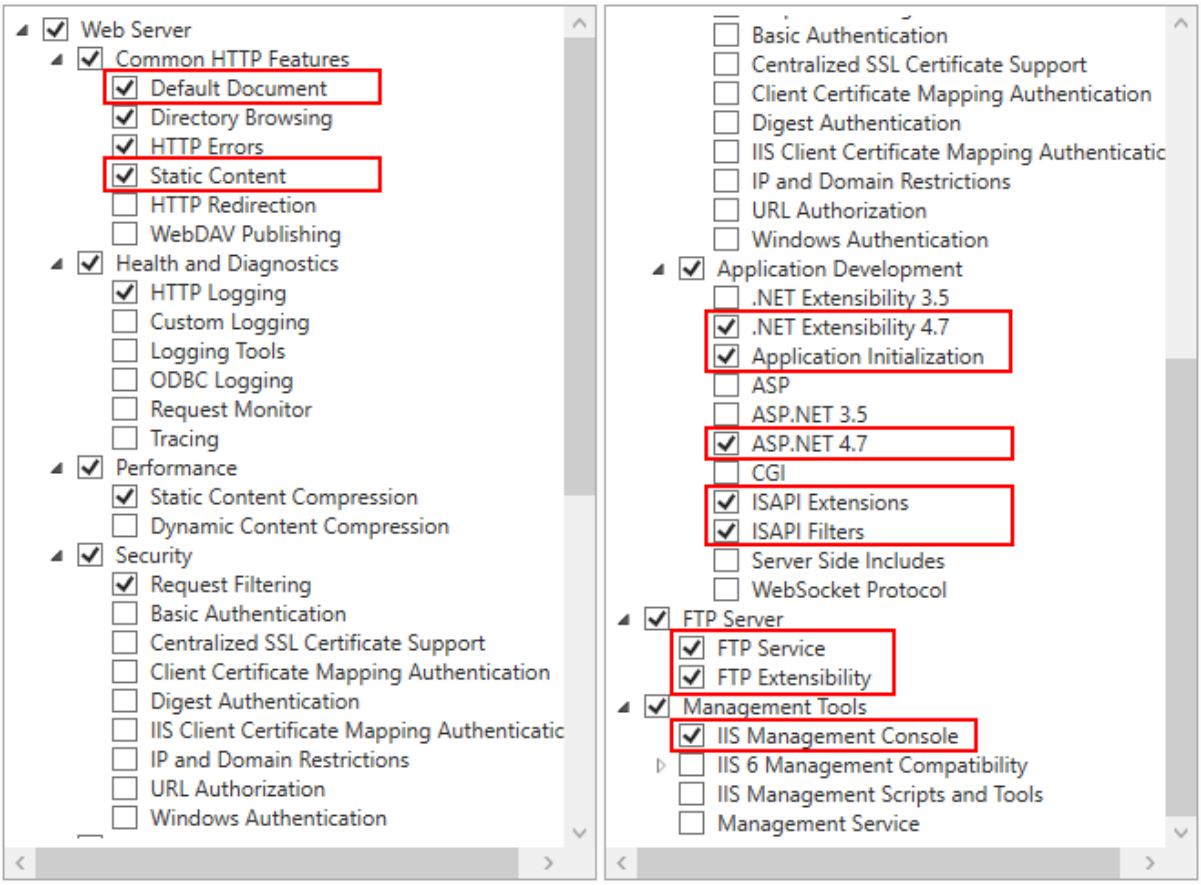


Step 5: Click **Next** until the **Role Services** section displays. Make sure the following role services are selected.

- Default Document
- Static Content
- .NET Extensibility 4.7
- Application Initialization
- ASP.NET 4.7
- ISAPI Extensions
- ISAPI Filters
- IIS Management Console
- **FTP Service**
- **FTP Extensibility**

  FTP Service & **FTP Extensibility** must be enabled if you want to create an IIS FTP site for transferring files from a remote development machine to the Web server.

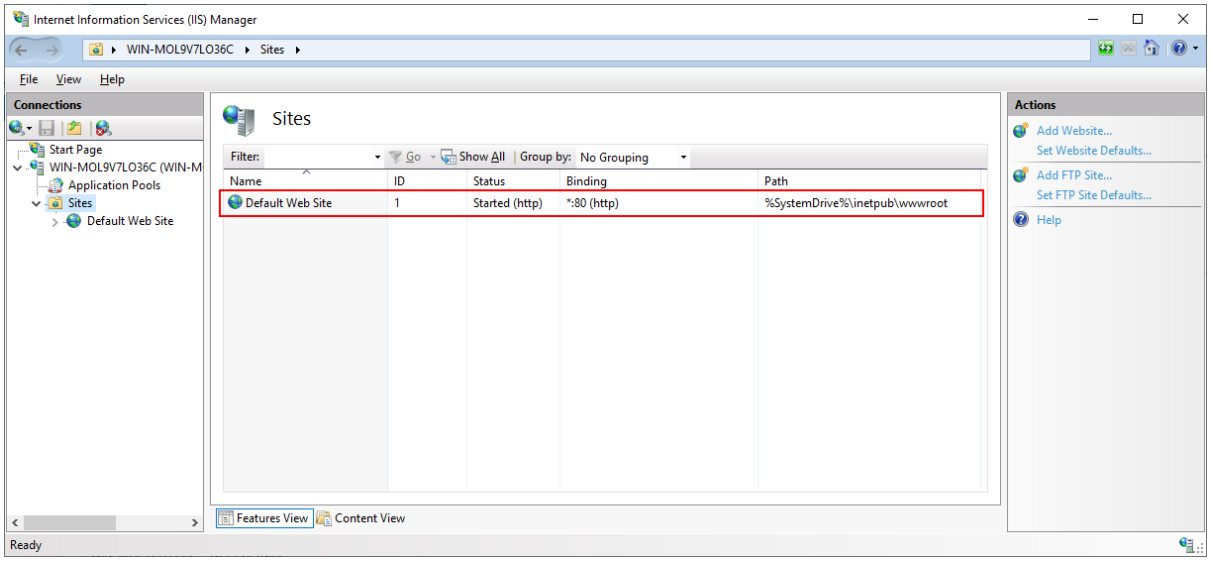
Figure 1.5:



Step 6: Click **Next** and then click **Install**.

After IIS is installed, a **Default Web Site** (with port 80) is automatically created (you could also create new websites with different port numbers).

Figure 1.6:



Step 7: Open a Web browser and run the following URLs to access the **Default Web Site**.

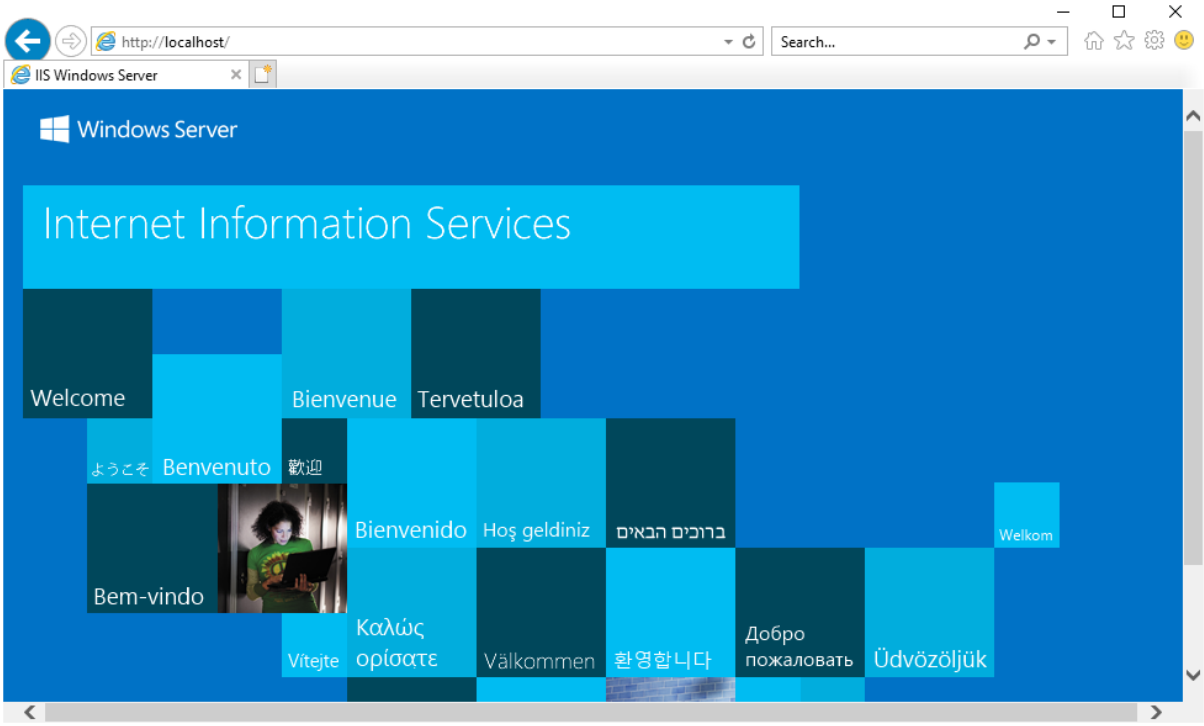
http://localhost:80/

http://*your_server_ip*:80/

TIP: You can use "localhost" or the IP address to access the IIS website on the local computer. To obtain the IP address, open a command prompt window and then type ipconfig<Enter>. Write down the IP address as it is needed when you configure the Web server profile in PowerBuilder.

If the IIS welcome screen displays, the IIS website is working properly.

Figure 1.7:



Also remember the physical path for Default Web Site which is **C:\inetpub\wwwroot** by default (or any other path you have changed to). This is where the client app will be deployed, or the FTP site will point to.

1.4.4 Deploying app files to Web Server

1.4.4.1 Overview

To deploy the client app from the local development PC to the remote Web server, you can choose:

- Method 1: Deploy the client app to the remote server through the FTP protocol.

  Step 1: Set up an FTP server (the FTP server's physical path must point to the Web root of the Web server).

  Method 1: Creating an IIS FTP site will walk you through how to set up an FTP server.

  Step 2: Deploy the client app from the development machine to the remote server through the FTP server.

"Task 4: Setting up the development PC" has detailed instructions.

- Method 2: Package the client app and then install (or copy) it to the Web root of the Web server.

  Method 2: Packaging and copying the client app will walk you through how to package the client app and then install (or copy) it to the Web server Web root.

1.4.4.2 Method 1: Creating an IIS FTP site

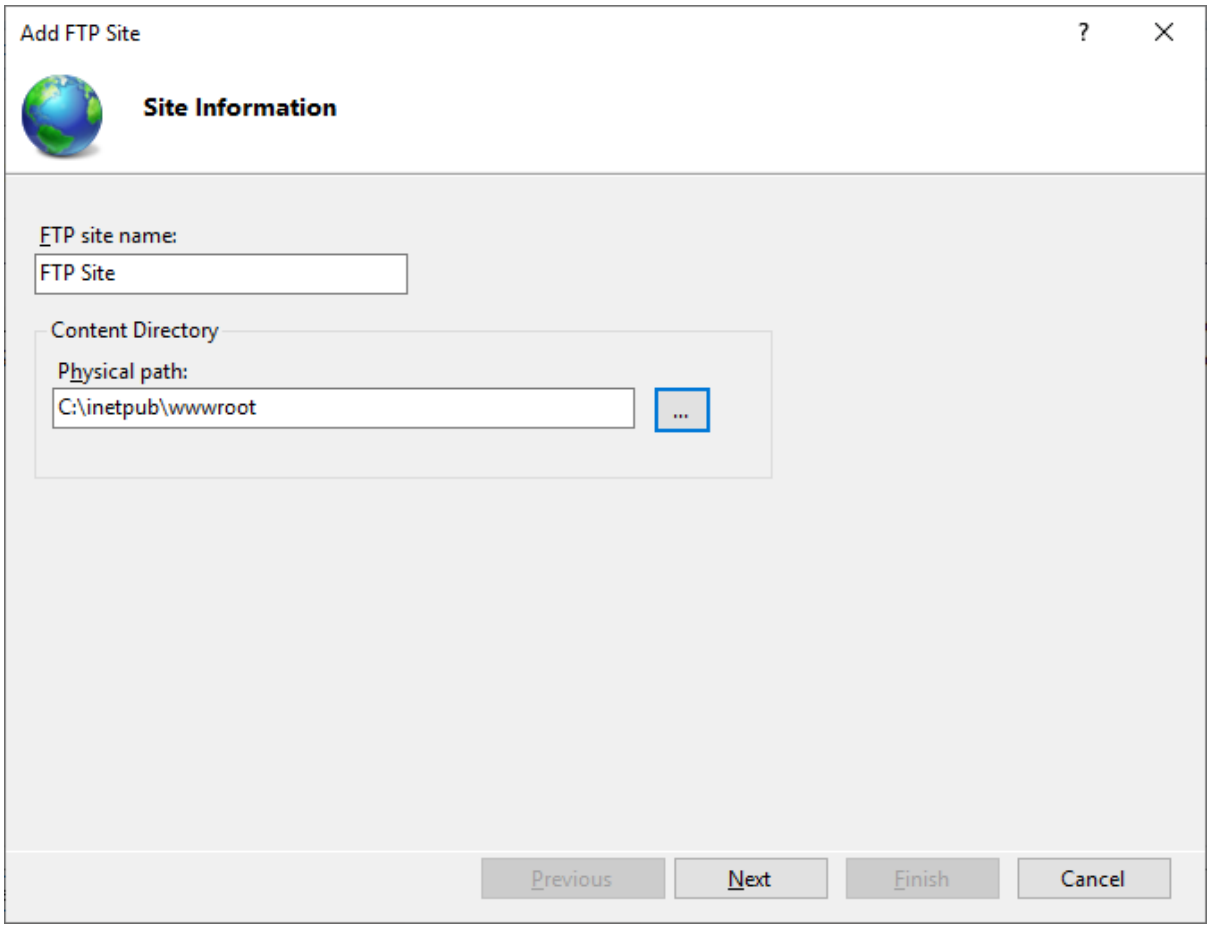
The following steps will walk you through setting up an IIS FTP site on the Web server, so that PowerBuilder can deploy files to the remote server through the FTP protocol.

In the previous section, if you have selected to enable **FTP Service** & **FTP Extensibility**, you can create an IIS FTP site to be used by the remote deployment.

Step 1: In the IIS Manager, right click **Sites**, select **Add FTP Site**.

Step 2: Specify a name for the FTP site, and set the physical path to the Web root of the IIS Web server (**C:\inetpub\wwwroot** in this tutorial). Click **Next**.

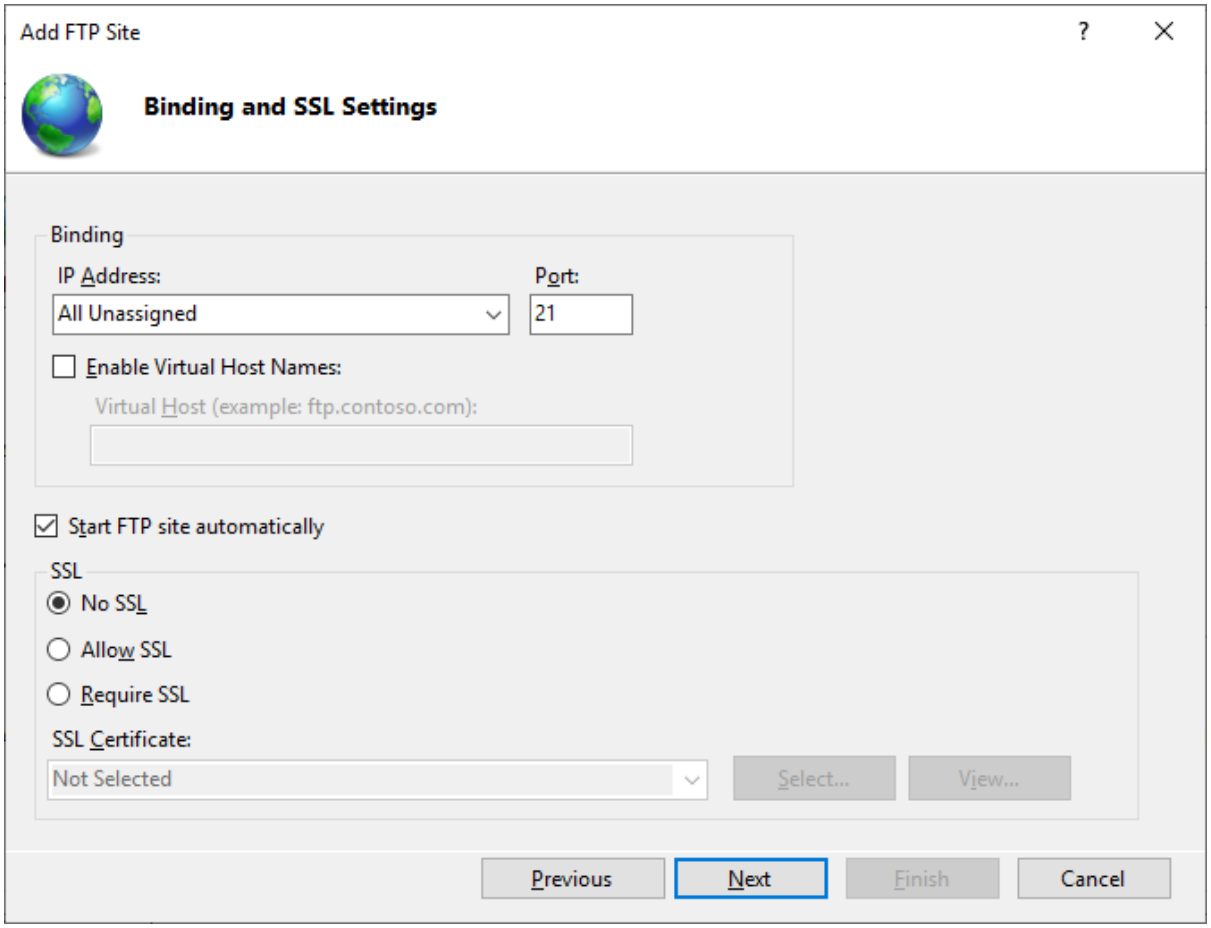
Figure 1.8:



Step 3: Use the default port 21 (or specify a different port if you like). If no certificate is available, you can select **No SSL**. Use the default values for the other settings. Click **Next**.

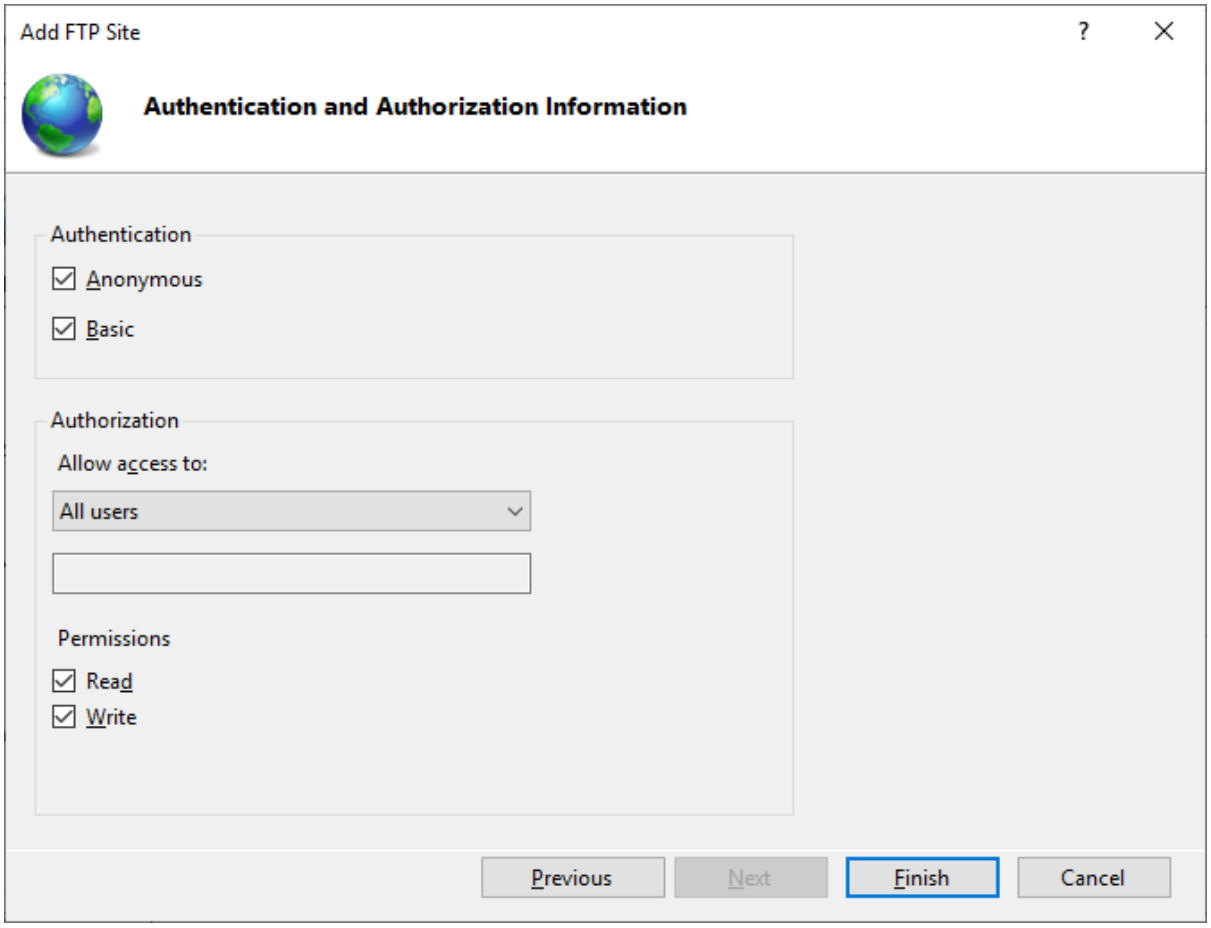
For how to configure SSL on an IIS FTP site, refer to Configure an SSL-based FTP server.

Figure 1.9:



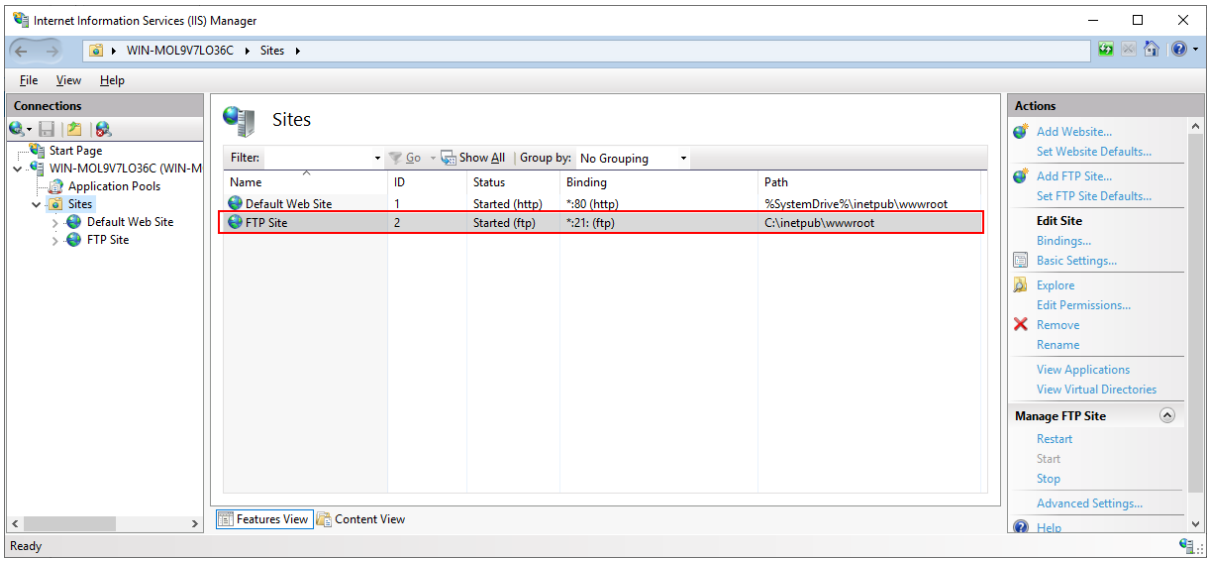
Step 4: Select **Anonymous** and **Basic** authentication. Select **All users** or specify the users that are allowed to access the FTP site, and then select the **Read** and **Write** permissions. Click **Finish**.

Figure 1.10:



The FTP site is created.

Figure 1.11:

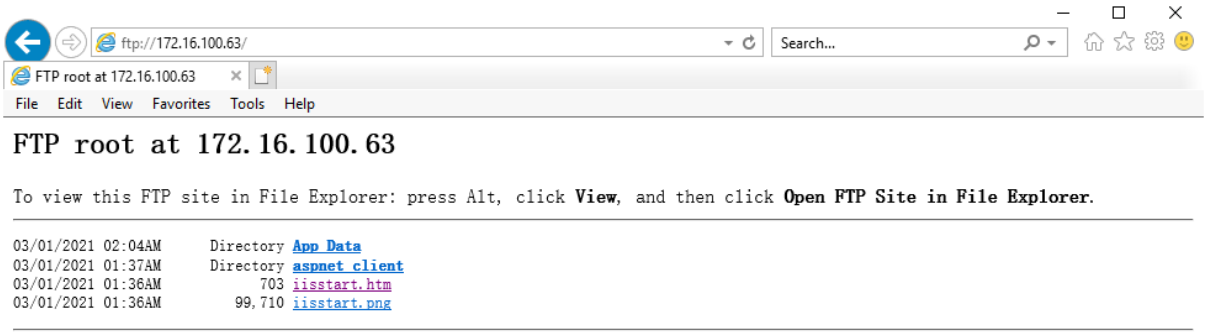


Step 5: Open a Web browser and run the following URL to access the FTP site.

ftp://*your_server_ip*:21/

If the FTP root displays, then the FTP site is working properly.

Figure 1.12:

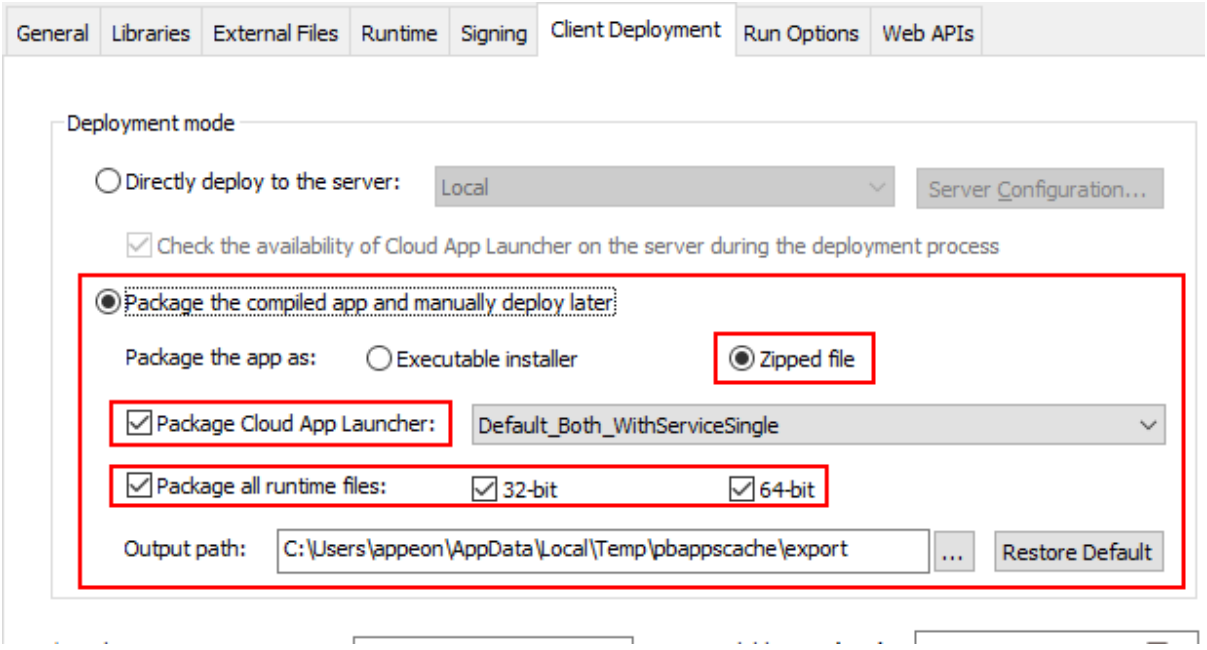


1.4.4.3 Method 2: Packaging and copying the client app

Before you take the steps below to package the client app, make sure you have built the application successfully by following instructions in the next section "Task 4: Setting up the development PC".

Step 1: In the PowerServer project painter, select the **Client Deployment** tab, then select **Package the compiled app and manually deploy later**, and then select **Zipped file**, **Package Cloud App Launcher**, and **Package all runtime files**.

Figure 1.13:



Step 2: Save the project settings and then click the **Build & Deploy PowerServer Project** or **Deploy PowerServer Project** button in the toolbar to generate the package.

When the packaging process is completed, the folder that contains the generated package will be displayed.

Step 3: Copy and extract the generated zipped file to the Web root of the Web server.

1.5 Task 4: Setting up the development PC

1.5.1 Preparations

Set up the development machine with the following OS and software (install the software in the order listed):

- Windows 10 (64-bit)
- SQL Anywhere 17
- PowerBuilder IDE 2021
- PowerBuilder Runtime 2021
- PowerServer Toolkit 2021
- SnapDevelop 2021 (optional)
- Google Chrome (optional)

1.5.2 Creating the ODBC data source

A database connection needs to be established between the development PC and the database server (for converting DataWindows to models), and between the .NET server and the database server (for retrieving data). Currently the SQL Anywhere database can only be connected through an ODBC driver, therefore, you will need to create the same ODBC data source in both:

- the development PC, and
- the .NET server

In Task 2: Setting up the database server, we have successfully set up the **salesdemo** SQL Anywhere database server in an individual machine. In this tutorial, we will create an ODBC data source on the development PC that connects to this database server. (You will take the same steps to create the same ODBC data source on the .NET server later. The same ODBC data source means the data source has the same name, for example, "**SalesDemo DB**" in this tutorial)

Step 1: Install SQL Anywhere 17.

Step 2: Create a 64-bit ODBC data source and name it as "**SalesDemo DB**". The data source name must be the same in both the development PC and the .NET server.

IMPORTANT: Make sure you use the 64-bit version of ODBC administrator to create the data source, because only the 64-bit ODBC data sources can be selected for the PowerServer project.

Step 3: Click **Test Connection** to ensure the connection settings are correct.

Figure 1.14:

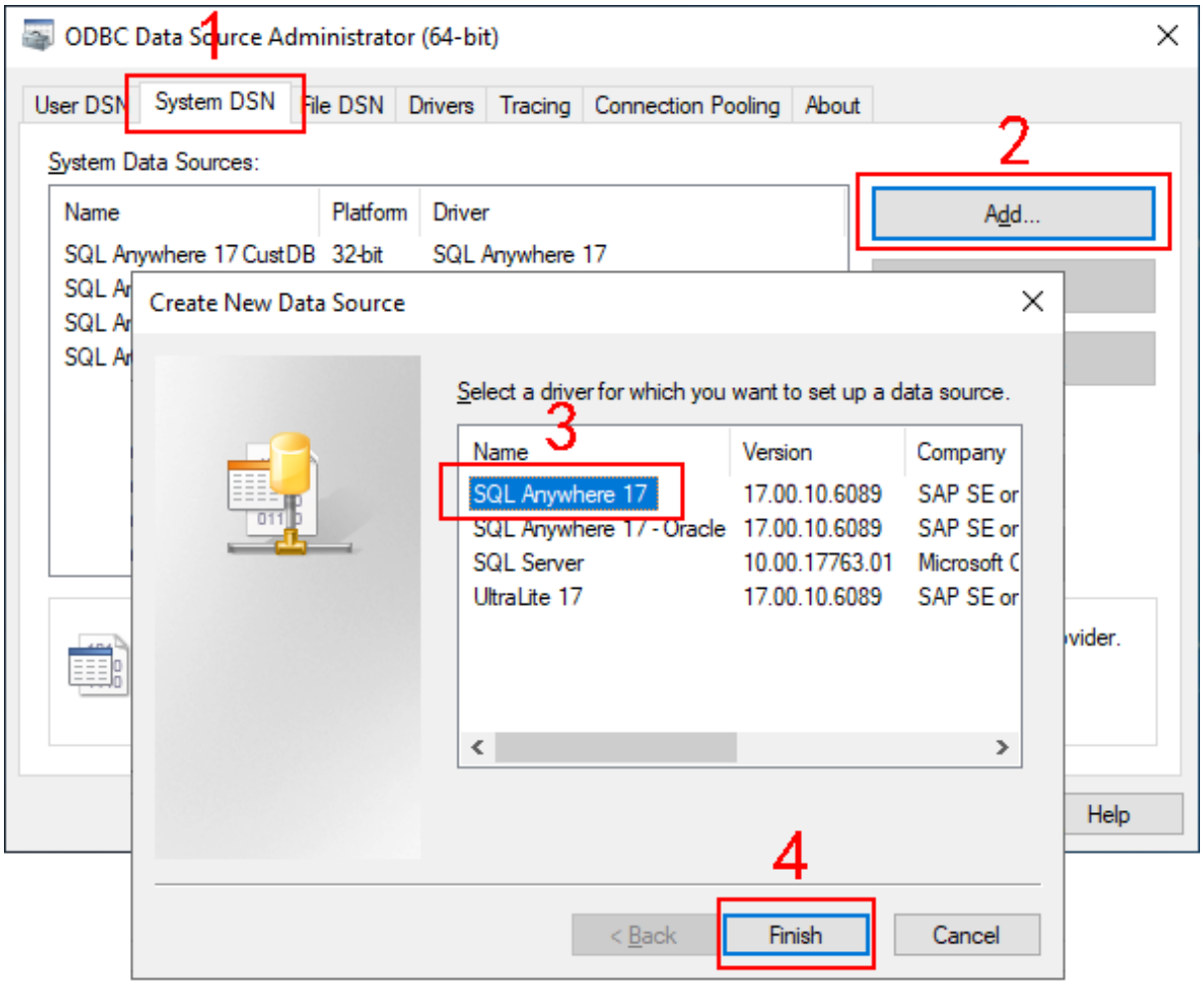
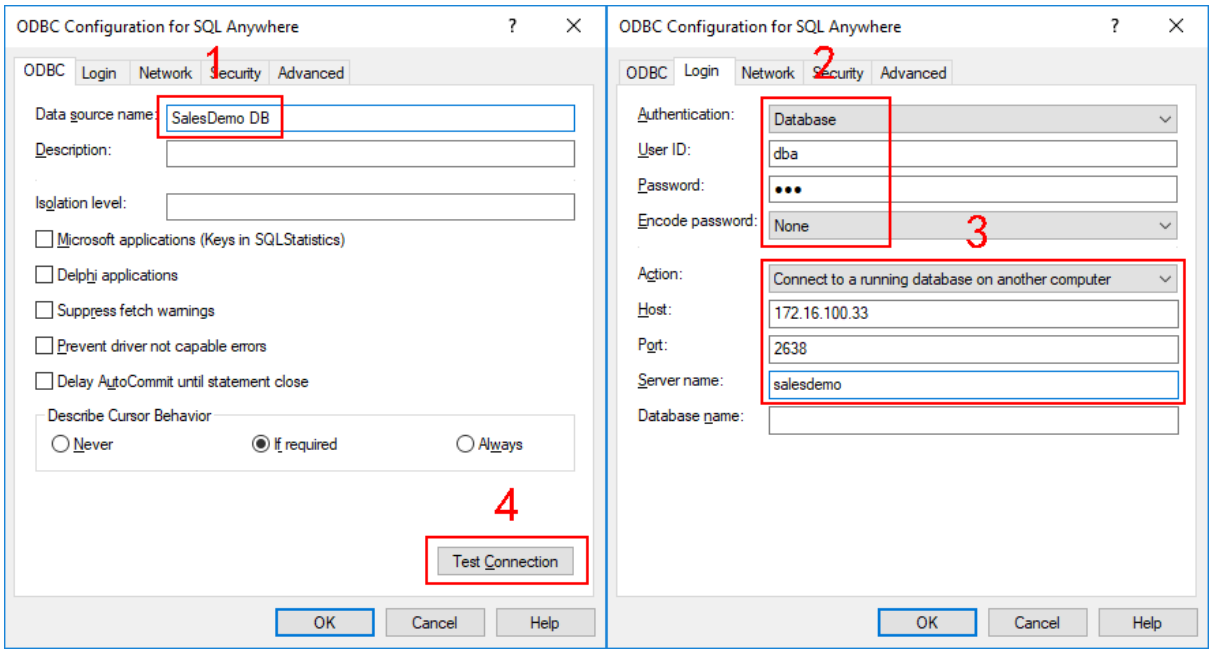


Figure 1.15:



1.5.3 Creating a Web server profile for remote deployment

Step 1: Go to the development PC, select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**. The SalesDemo workspace is loaded in the PowerBuilder IDE.

Step 2: Select **Tools**>**Web Server Profile** from the PowerBuilder menu bar to open the **Web Server Profile** window.

Step 3: In the **Web Server Profile** window, click **Add**.

Step 4: Select **Remote server**, and then specify the settings for connecting to the FTP site.

In this tutorial, specify the following values (or the values you chose):

Server profile name: Any text, for example, Remote IIS Web Server, Remote Apache HTTP Server, Remote Nginx, etc.

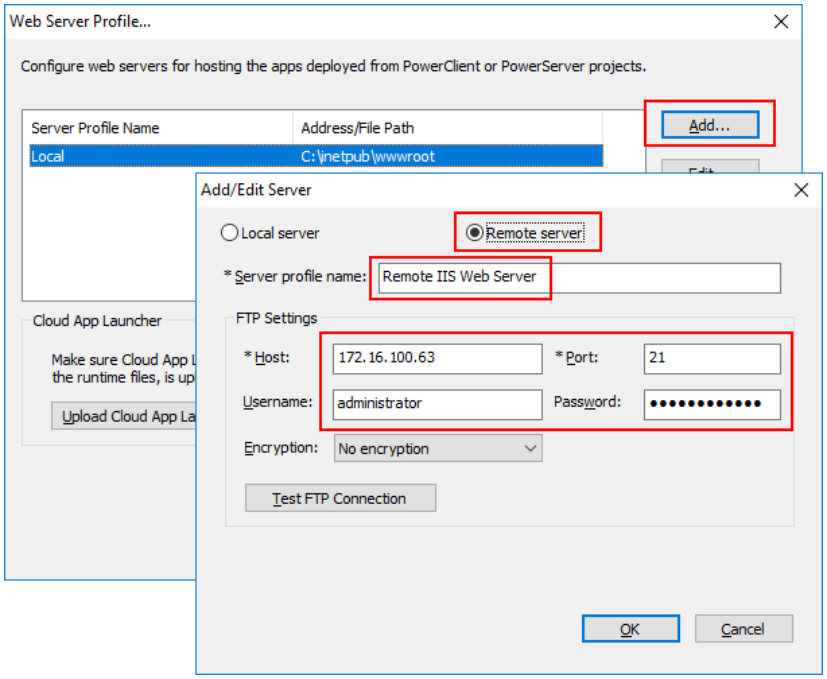
Host: IP address or host name of the FTP site, for example, 172.16.100.63.

Port: Port number of the FTP site, for example, 21.

Username: Windows user name.

Password: Windows user password.

Figure 1.16:



Step 5: Click **Test FTP Connection** and make sure connection to the FTP site is successful.

Tip

In case connection errors occur, try the following to resolve:

- Check if the Windows Defender Firewall on the FTP server allows the FTP port (21 in this tutorial) to go through.

- Check if the port (21 in tutorial) is occupied by any other program.

  Tip: You can execute the command "netstat -ano | findstr 21" to check if the port number is occupied by any other program.

- Input a username and password for logging to the FTP site, instead of using anonymous login.

- Check if the user has read and write permissions to the FTP root.

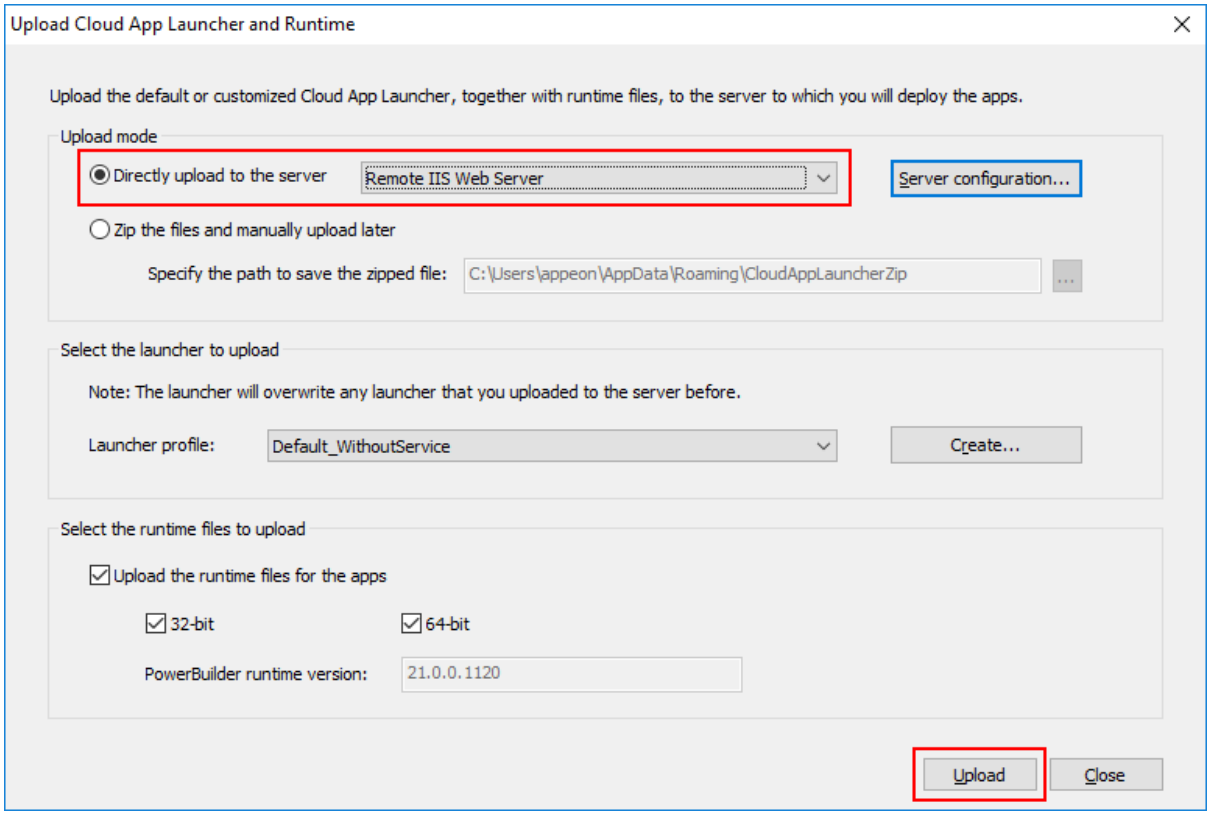
1.5.4 Uploading the cloud app launcher and the runtime files to the remote server

Step 1: Select **Tools**>**Upload Cloud App Launcher** from the PowerBuilder menu bar.

Step 2: In the **Upload Cloud App Launcher and Runtime** window that appears, select **Directly upload to the server** and then select a server profile (for example, "Remote IIS Web Server") from the listbox.

Step 3: Keep the other settings as default and click **Upload**.

Figure 1.17:



Step 4: When the upload is finished, go to the Web server and verify the "CloudAppPublisher" folder exists under the Web root (in this tutorial, the Web root for IIS is C:\inetpub\wwwroot, for Apache is C:\Apache24\htdocs, and for Nginx is C:\nginx-1.19.10\html).

1.5.5 Modifying and re-deploying the PowerServer project

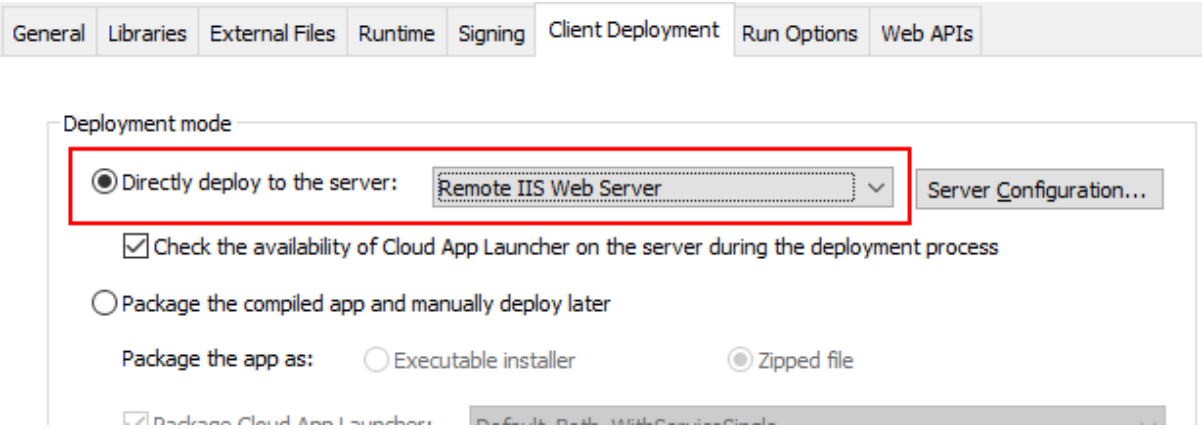
The following modifications are made to the PowerServer project created in the Quick Start guide. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

Step 1: Select the profile for the remote server (instead of the local server).

On the **Client Deployment** tab of the PowerServer project painter, select "**Directly deploy to the server**" and then select a server profile (such as "Remote IIS Web Server", "Remote Apache HTTP Server", or "Remote Nginx") in the **Deployment mode** section.

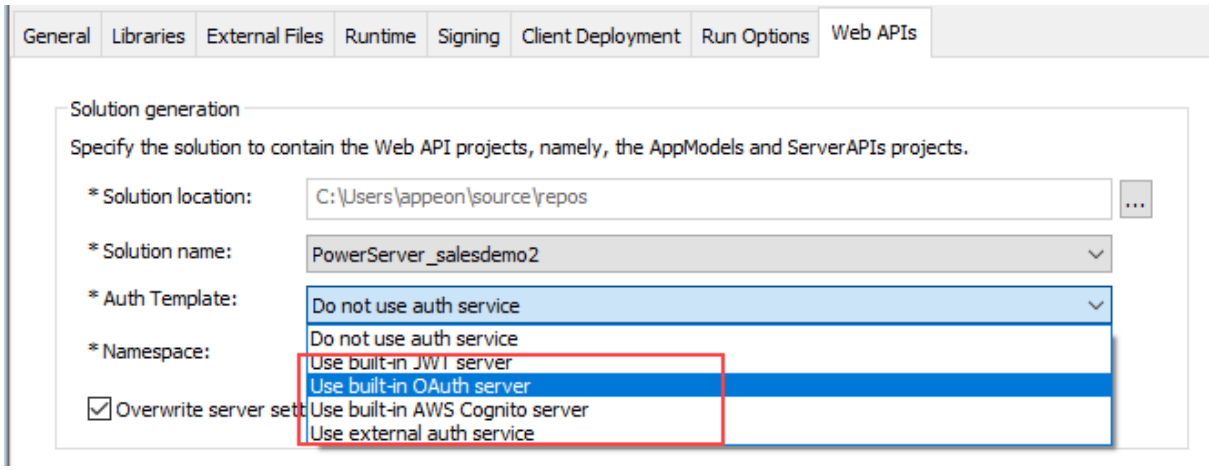
The server profile is created in the section Creating a Web server profile for remote deployment.

Figure 1.18:



Step 2: Specify the auth template to use in the production environment. It is strongly recommended for security concerns, that in the production environment, you shall safeguard the server resources through implementing an authentication server with PowerServer.

Figure 1.19:



Select a template type from the **Auth Template** list.

- **Use built-in JWT server**: Includes a built-in authentication server that supports JWT or bearer tokens. See *Tutorial 6: Authenticating your apps* > Using JWT for more information.

- **Use built-in OAuth server**: Includes a built-in authentication server based on IdentityServer4 framework that works with the OAuth 2.0 authorization flows. See *Tutorial 6: Authenticating your apps* > Using OAuth 2.0 for more information.

- **Use built-in AWS Cognito server**: Includes a built-in authentication server that works with the Amazon Cognito user pool. See *Tutorial 6: Authenticating your apps* > Using Amazon Cognito for more information.

- **Use external auth service**: Includes templates that can be easily extended to support the other identity providers that work with the OAuth flows or JWT, such as Azure AD or Azure AD B2C. See *Tutorial 6: Authenticating your apps* > Using other auth servers for more information.

Step 3: Specify the Web API URL. The Web API URL is used by the client app to call the Web APIs.

On the **Web APIs** tab of the PowerServer project painter, specify the Web API URL, for example, https://172.16.100.71:5009. This indicates that the client app will call the Web APIs running on the server at https://172.16.100.71:5009. It is highly recommended that you specify an HTTPS URL for the production environment.

Important

1. Make sure the Web API is running on the specified IP address (or host name) and port number. For how to start the Web API, see the next section.

2. If the IP address and port number of the .NET server are changed later, you will need to modify the settings here and then deploy the project again (using the "Deploy PowerServer Project" option).

Figure 1.20:

Step 4: Select the "**SalesDemo DB**" ODBC data source (created in Creating the ODBC data source).

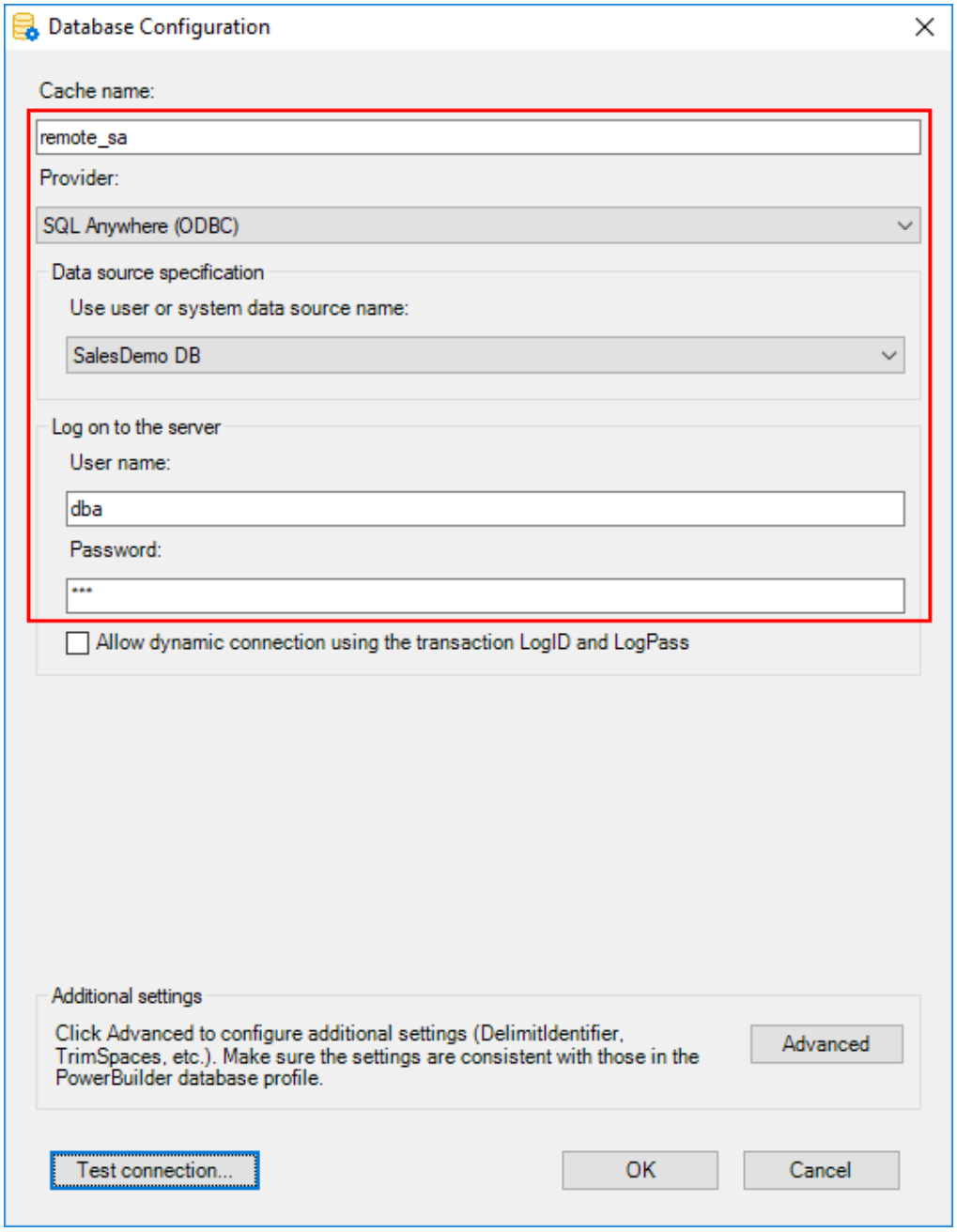
At the bottom of the **Web APIs** tab of the PowerServer project painter, click the **Database Configuration** button.

In the **Database Configuration** window, click **DB Drivers** in the upper part to make sure the corresponding database driver and the option "I have read and agree to the license ..." both are selected.

In the **Database Configuration** window, click **New** in the upper part to create the database connection that will be used by the deployment.

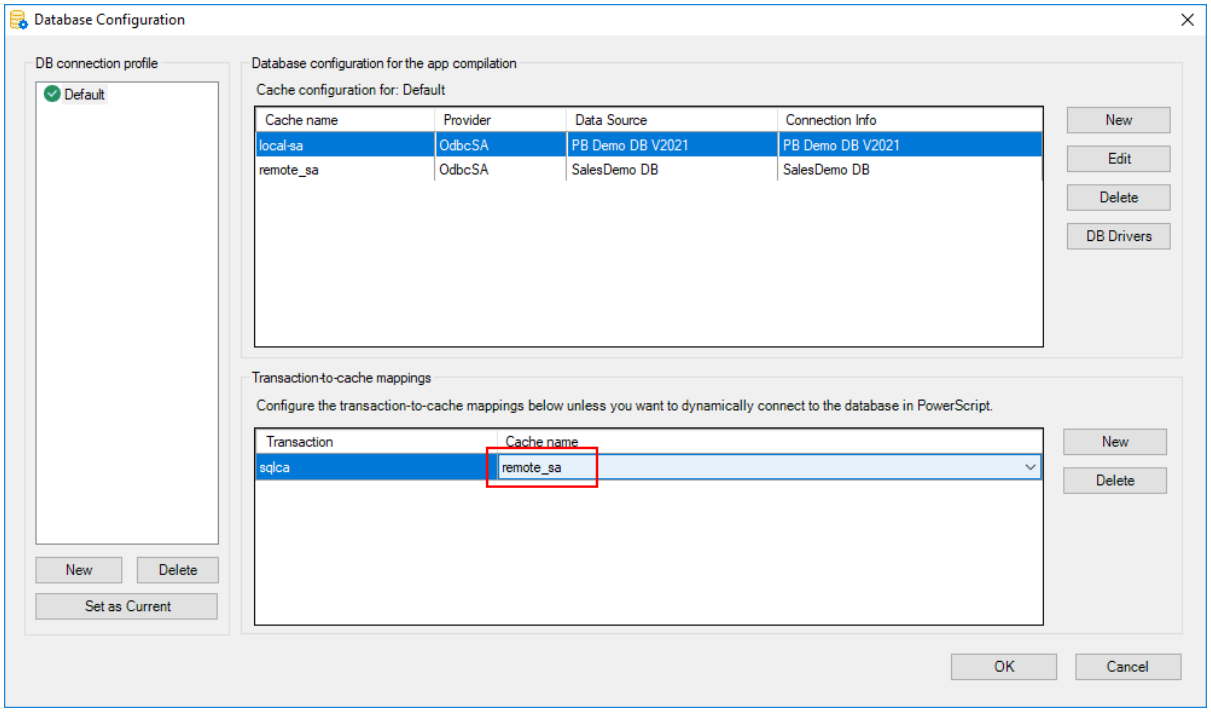
In the dialog box that displays, configure the database connection with the following settings:

Figure 1.21:



Then select the database cache you created just now to map with the "sqlca" transaction object.

Figure 1.22:



Step 5: Save the PowerServer project settings.

Step 6: Build and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) for the changes to take effect.

When the deployment is finished, go to the Web server and verify that the application folder (for example, "pssales") exists under the Web root.

Step 7: Go to the specified location (C:\Users\appeon\source\repos in this tutorial) and copy the PowerServer C# solution folder to the .NET server.

1.6 Task 5: Setting up the auth server

It is strongly recommended for security concerns, that in the production environment, you shall safeguard the server resources through implementing an authentication server with PowerServer.

If you have selected an auth template (build-in JWT server, built-in OAuth server, or built-in AWS Cognito server) in the project settings, make sure to follow the relevant instructions to modify the PowerBuilder client app and re-deploy the PowerServer project. If you have selected to use built-in AWS Cognito server, you also need to make changes to the deployed authentication template.

- **Use built-in JWT server**: Includes a built-in authentication server that supports JWT or bearer tokens. See *Tutorial 6: Authenticating your apps* > Using JWT for more information.

- **Use built-in OAuth server**: Includes a built-in authentication server based on IdentityServer4 framework that works with the OAuth 2.0 authorization flows. See *Tutorial 6: Authenticating your apps* > Using OAuth 2.0 for more information.

- **Use built-in AWS Cognito server**: Includes a built-in authentication server that works with the Amazon Cognito user pool. See *Tutorial 6: Authenticating your apps* > Using Amazon Cognito for more information.

If you have selected to use external auth service in the project settings, see *Tutorial 6: Authenticating your apps* > Using other auth servers for more information on how to incorporate the other auth server that work with the OAuth flows or JWT, such as Azure AD or Azure AD B2C.

1.7 Task 6: Setting up the .NET server

1.7.1 Preparations

This tutorial starts the Web APIs directly (using the built-in Kestrel server), you can also deploy the Web APIs to a more secure and manageable environment such as Docker Container, IIS etc. as described in the following tutorials.

- Tutorial 2: Hosting Web APIs in Docker Containers
- Tutorial 3: Hosting Web APIs in IIS
- Tutorial 4: Hosting Web APIs in Kestrel

In this tutorial, we will set up a .NET server running in an independent machine.

Step 1: Set up the .NET server with the following OS and software:

- Windows Server 2019 (64-bit)
- SQL Anywhere 17
- SnapDevelop 2021

Step 2: (**IMPORTANT**) Configure Secure Sockets Layer (SSL) for the .NET server, so that HTTPS can be used to secure the connections between the client and the .NET server.

Step 3: Make sure the .NET server can connect to the NuGet site: https://www.nuget.org (for installing PowerServer NuGet packages) and the following Appeon sites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) (for validating the PowerServer license).

> **Note**
>
> If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Step 4: Configure Windows Defender Firewall on the .NET server to allow the port (5009 in this tutorial or any port number you choose). The section "Configuring Windows Defender Firewall" has detailed instructions.

Important

For optimal runtime performance, it is highly recommended that the PowerServer Web APIs should be published to a server that locates on the same LAN as the database server. If the database is not on the same network as the Web APIs, every request has to go a long way from PowerServer to the database, it is highly possible that there will be performance and security issues.

1.7.2 Creating the ODBC data source

A database connection needs to be established between the development PC and the database server (for converting DataWindows to models), and between the .NET server and the database server (for retrieving data). Currently the SQL Anywhere database can only be connected through an ODBC driver, therefore, you will need to create the same ODBC data source in both:

- the development PC, and
- the .NET server

Step 1: Install SQL Anywhere 17.

Step 2: Create a 64-bit ODBC data source and name it as "**SalesDemo DB**". The data source name must be the same in both the development PC and the .NET server. The data source should connect to the **salesdemo** SQL Anywhere database server (which is set up in Task 2: Setting up the database server).

IMPORTANT: Make sure you use the 64-bit version of ODBC administrator to create the data source, because only the 64-bit ODBC data sources are supported by PowerServer.

Step 3: Click **Test Connection** to ensure the connection settings are correct.

Figure 1.23:

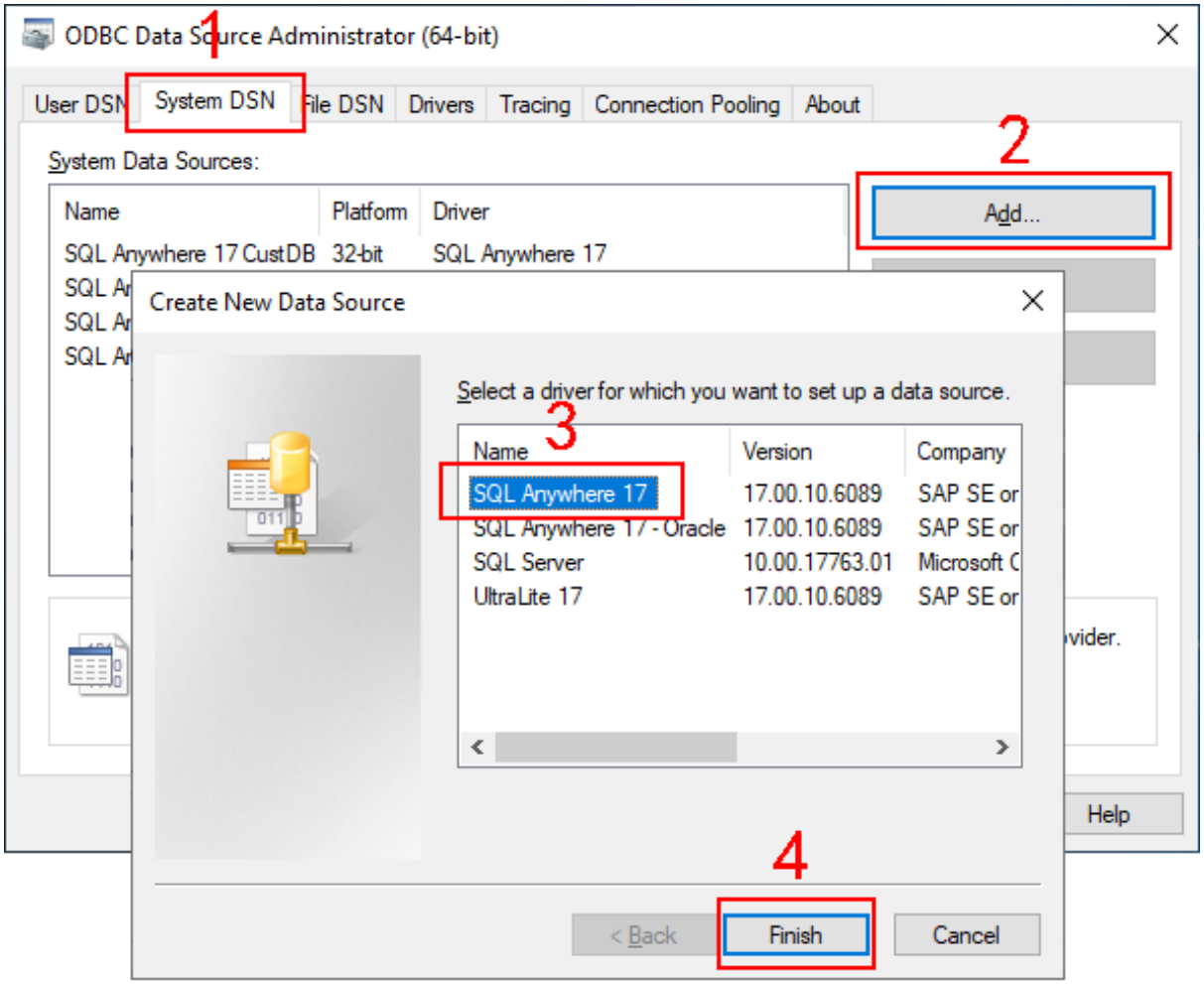
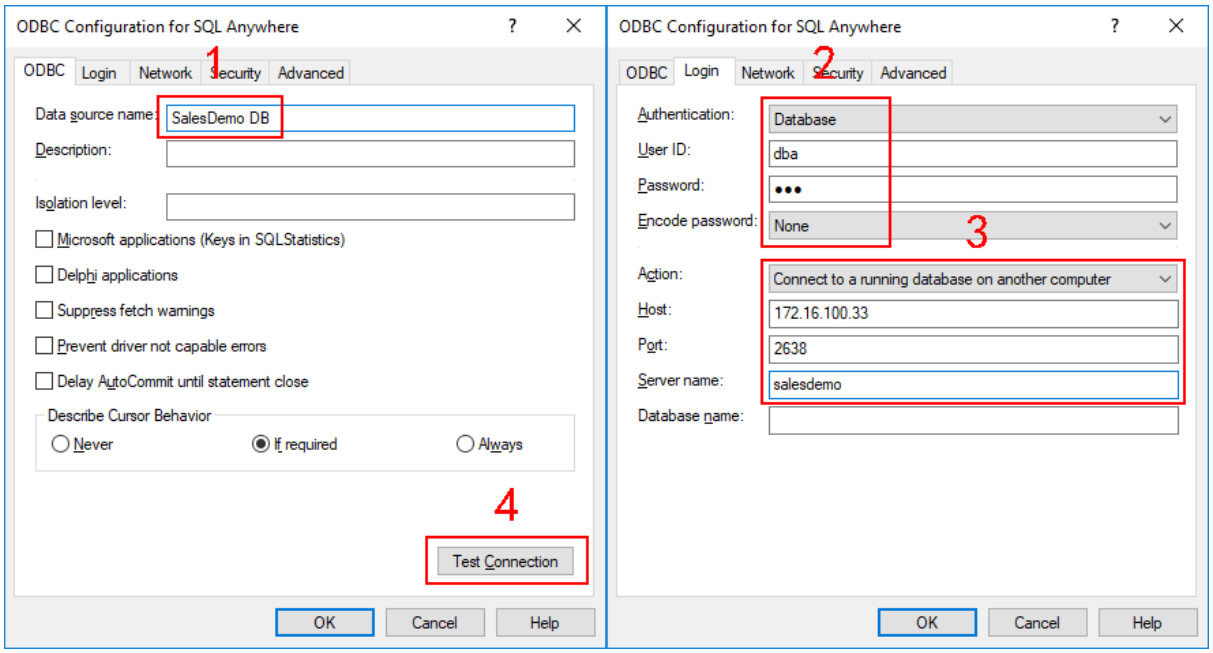


Figure 1.24:



1.7.3 Publishing the Web APIs

Step 1: Copy the PowerServer C# solution from the development PC (C:\Users\appeon \source\repos in this tutorial) to the .NET server.

Step 2: Double click **PowerServer_[appname].sln** to launch the solution in SnapDevelop. Log in to SnapDevelop if required.

The PowerServer C# solution will connect to the NuGet site (https://www.nuget.org) to download and install the required packages from the NuGet site.

Step 3: Click **Run** from the SnapDevelop toolbar to start the Web APIs (using the built-in Kestrel server) immediately.

You can also deploy the Web APIs to a hosting environment, for example, publish to a folder on the hosting server (like Docker), or publish to a process manager such as IIS etc., as described in the following tutorials.

- Tutorial 2: Hosting Web APIs in Docker Containers
- Tutorial 3: Hosting Web APIs in IIS
- Tutorial 4: Hosting Web APIs in Kestrel (and using a reverse proxy)

2 Tutorial 2: Hosting Web APIs in Docker Containers

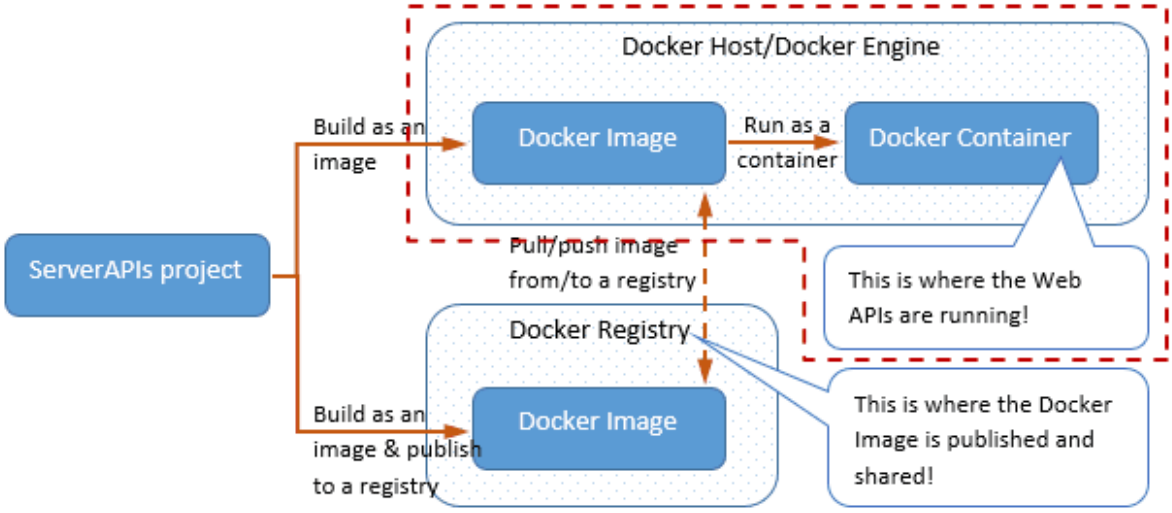
The PowerServer Web APIs is an ASP.NET Core app; it can be hosted and deployed like any other ASP.NET Core app described in https://docs.microsoft.com/aspnet/core/host-and-deploy/?view=aspnetcore-3.1.

This tutorial takes Docker as an example to show you how to publish and host the Web APIs in a Docker Container; it will reuse part of the configurations in the Quick Start and Tutorial 1, thus, it is strongly recommended that you have completed the Quick Start guide and Tutorial 1 first.

2.1 Task 1: Setting up Docker

2.1.1 Setting up a docker host (Docker Engine)

Figure 2.1:



The docker host is where the docker image is built and the docker container is run. The **ServerAPIs** project will be built and published as a docker image first, and then the docker image will be run as a docker container. The Web APIs is actually hosted and run in the docker container.

Step 1: Set up a docker host (also called Docker Engine in the SnapDevelop IDE).

To set up a docker host/Docker Engine, refer to https://docs.docker.com/engine/install/.

In this tutorial, a Docker Engine has already been set up in a Linux server (suppose its IP address and port number are 172.25.100.20:2375).

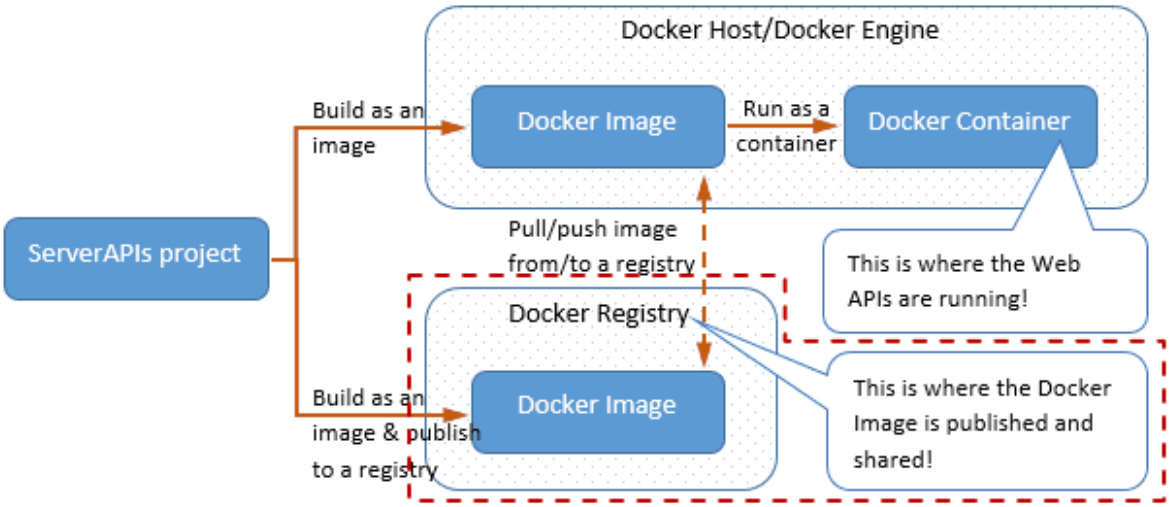
Write down this information as it will be required when you build the **ServerAPIs** project as a docker image (in the later section Publishing Web APIs to Docker).

Step 2: Make sure the docker host machine can connect to the following Appeon sites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) (for validating the PowerServer license).

If the docker host machine connects to Internet via a proxy server, refer to Configure Docker to use a proxy server for detailed instructions.

2.1.2 Setting up a docker registry

Figure 2.2:



A docker registry is the repository where the docker image is published and shared. You may choose from the following registries:

- Docker Hub -- Docker's official registry, it is the default registry when you install Docker. You can connect to the public registry (hub.docker.com:443) that anyone can use or a your own private registry. You will be required to log into Docker Hub before you can store the image. For more about Docker Hub, refer to https://docs.docker.com/docker-hub/.
- A self-hosted Docker Registry -- Your own registry created using the open-source Docker Registry. For more about Docker Registry, refer to https://docs.docker.com/registry/.

Step 1: Set up a docker registry.

In this tutorial, a self-hosted Docker Registry has already been set up in a Linux server (suppose its IP address and port number are 172.25.100.20:5000).

Write down this information as it will be required when you build and publish the **ServerAPIs** project as a docker image (in the later section Publishing Web APIs to Docker).

To know more about Docker, we recommend you start by understanding the Docker Architecture.

2.2 Task 2: Setting up the database server

2.2.1 Preparations

This tutorial takes PostgreSQL database as an example. You can also install other databases by following the documentation from the vendor.

In this tutorial, we will set up a database server with the **PBDemo** PostgreSQL database running in an independent machine.

Step 1: Set up the database server with the following OS and software:

- Windows Server 2019 (64-bit)
- PostgreSQL 12

  Click here to download the installer for PostgreSQL.

Step 2: Configure Windows Defender Firewall on the database server to allow the database server port (**5432** in this tutorial or any port number you choose). The section "Configuring Windows Defender Firewall" has detailed instructions.

2.2.2 Starting the database

Step 1: Download the database file (**pbdemo2021_for_postgresql.zip**) from https://github.com/Appeon/PowerBuilder-Project-Example-Database.

Or copy the database file (**pbpostgres2021.dmp**) from the PowerBuilder demo installation folder (%Public%\Documents\Appeon\PowerBuilder 21.0\) to the database server, if you have installed PowerBuilder IDE according to Task 3: Setting up the development PC.

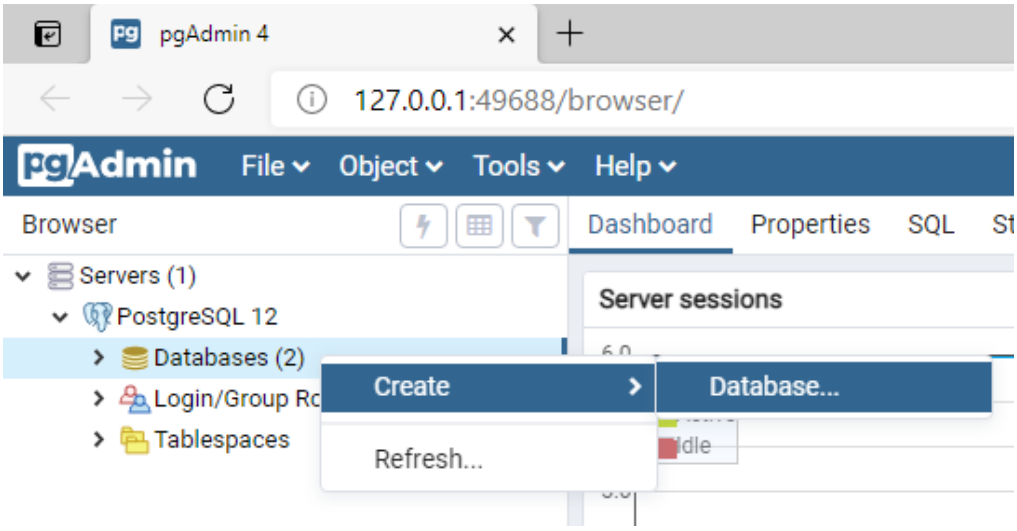
Step 2: Restore and run the database in the management tool for PostgreSQL.

1. Select Windows **Start** menu | **PostgreSQL 12** | **pgAdmin 4**.

   pgAdmin 4 is a Web application. If pgAdmin 4 cannot run in Internet Explorer (the default Web browser in Windows Server 2019), you can install and try Google Chrome.

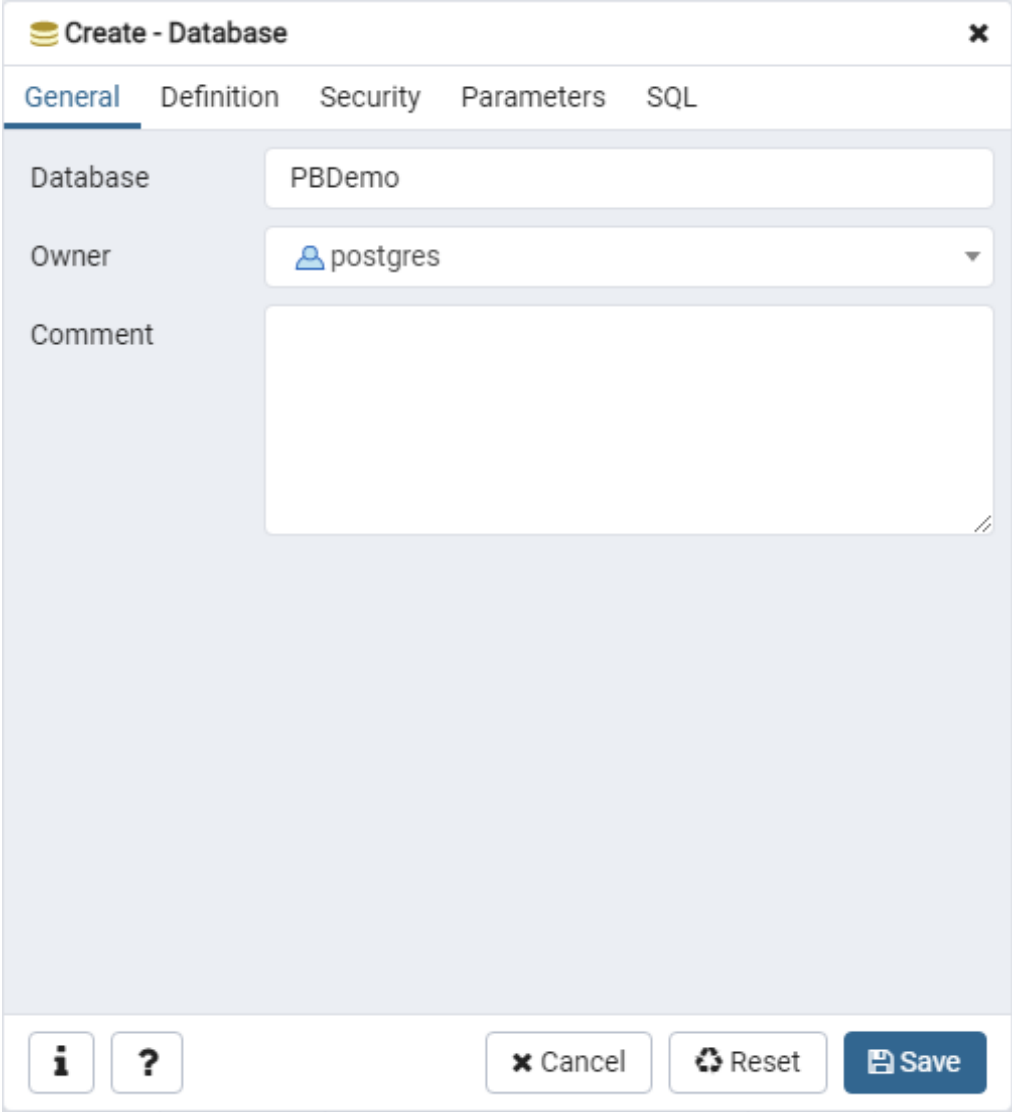
2. Expand **Servers** | **PostgreSQL**, right click **Databases**, and select **Create** | **Database**.

Figure 2.3:



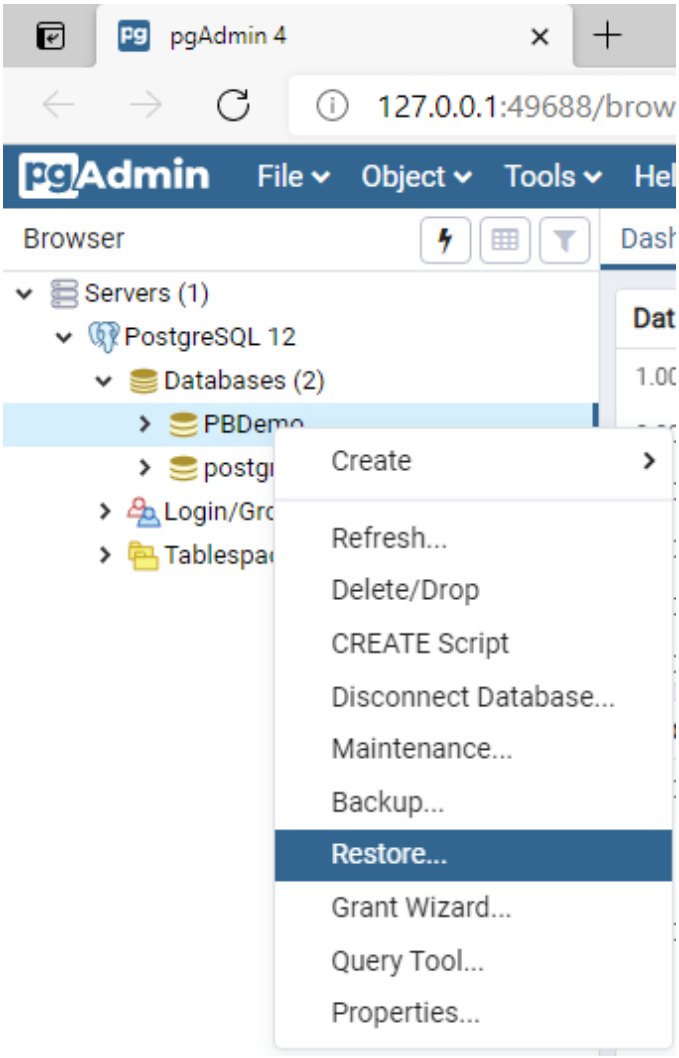
3. Input **PBDemo** in the **Database** field and click **Save**.

Figure 2.4:



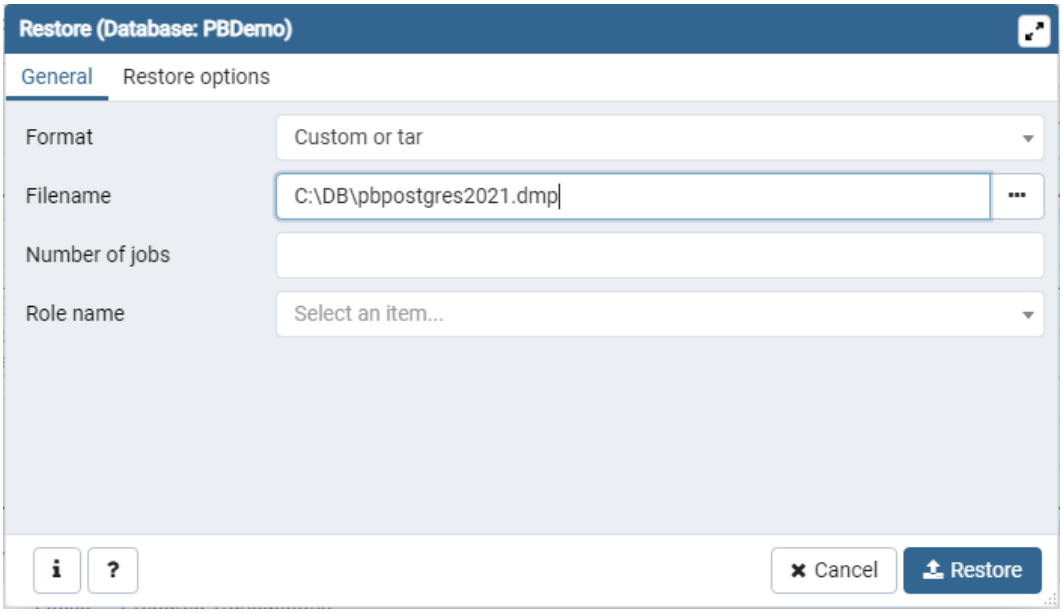
4. Right click **PBDemo** that was just created, and select **Restore**.

Figure 2.5:



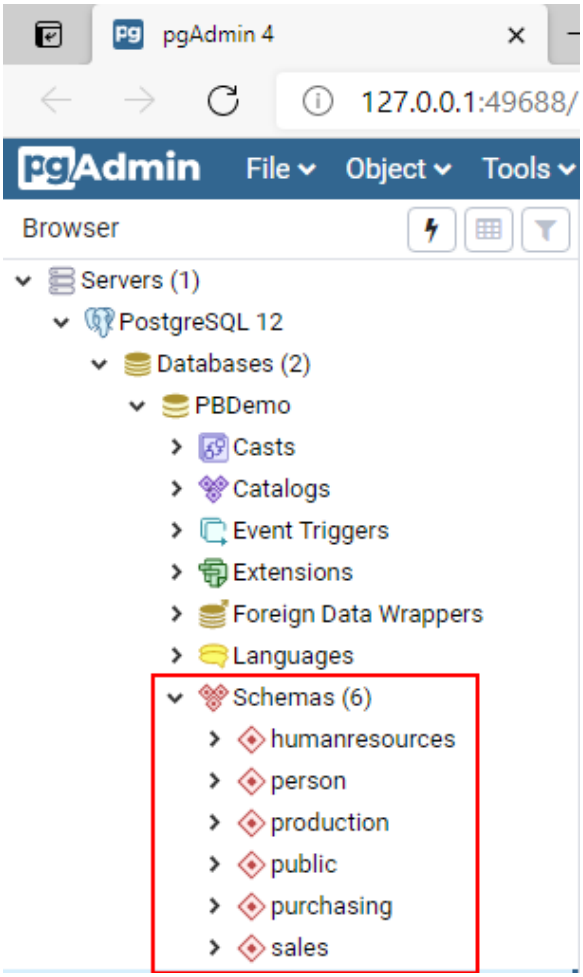
5. Select the **pbpostgres2021.dmp** file and click **Restore**.

Figure 2.6:



After the database file is restored, you will be able to view the following schemas:

Figure 2.7:



6. Open the **pg_hba.conf** file in a text editor and add the following line. The **pg_hba.conf** file is located in %PostgreSQL%\12\data. This enables the database server to allow remote connections.

```
host    all             all             0.0.0.0/0               md5
```

2.3 Task 3: Publishing to Docker

2.3.1 Preparing the development PC

Set up the development machine with the following OS and software (install the software in the order listed):

- Windows 10 (64-bit)

- PostgreSQL 12

  During installation, make sure the **Command Line Tools** component is selected to install, and specify and write down the following information:

  Data Directory: C:\Program Files\PostgreSQL\12\data by default

  Database Superuser: postgres by default

  Password for Database Superuser: (this password is set during installation) postgres in this tutorial

  Port Number: 5432 by default

- PostgreSQL ODBC driver (32-bit)

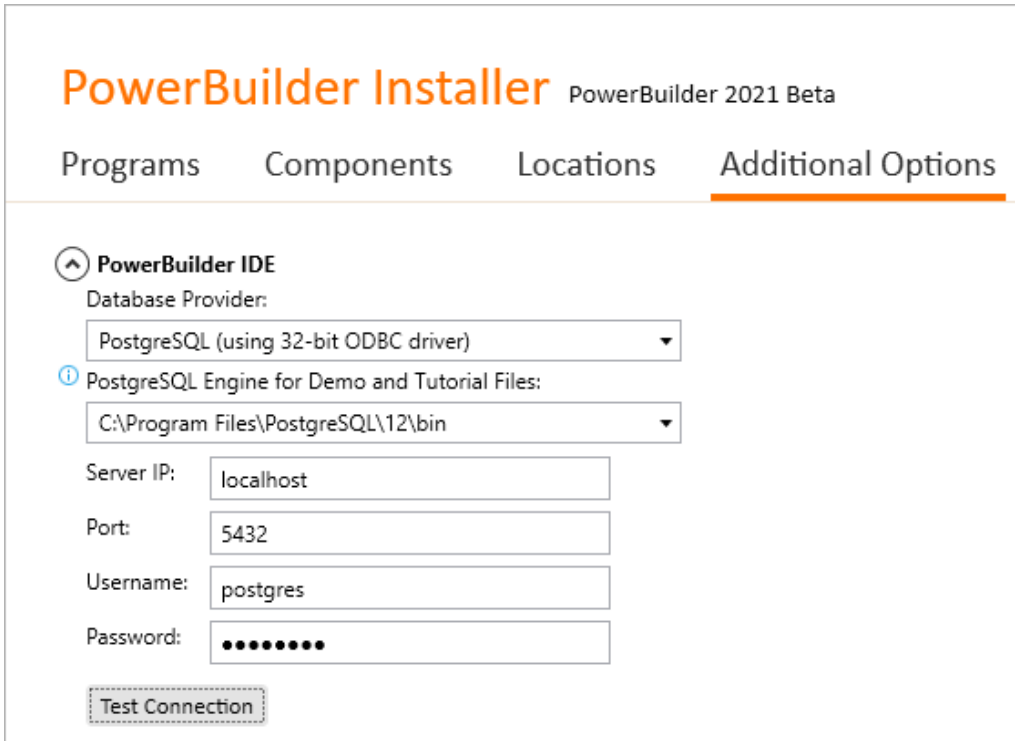
  The 32-bit version of PostgreSQL ODBC driver is required by the PowerBuilder IDE to establish database connection with the PostgreSQL database; therefore the PostgreSQL ODBC driver (32-bit) must be installed on the development PC.

- PowerBuilder IDE 2021

  During installation, make sure to select the PostgreSQL engine for the PowerBuilder demo database.

  The PowerBuilder demo database file for PostgreSQL (**pbpostgres2021.dmp**) will be installed to the %Public%\Documents\Appeon\PowerBuilder 21.0\ directory.

Figure 2.8:



- PowerBuilder Runtime 2021
- PowerServer Toolkit 2021
- SnapDevelop 2021
- Google Chrome (optional)

2.3.2 Modifying and re-deploying the PowerServer project

The following modifications are made to the PowerServer project created in the Quick Start guide and modified in Tutorial 1. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide and Tutorial 1 to create one.

Step 1: Specify where the Web APIs is actually hosted and run. This tells the client app where and how to call the Web APIs.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the docker container where the Web APIs is running. The host name (or IP address) of the docker container should be the same as that of the docker host/Docker Engine. The port number is what will be specified later when the docker container is run, for example, https://172.16.100.20:5009. This indicates that the client app will call the Web APIs running on the docker container at https://172.16.100.20:5009.

It is highly recommended that you specify an HTTPS URL for the production environment.

Important

1. Make sure the docker container is run at the same host name (or IP address) and port number. For how to run the image as a container, see the next section Publishing Web APIs to Docker.

2. If the host name and port number of the docker container are changed later, you will need to modify the settings here and then deploy the project again (using the "Build & Deploy PowerServer Project" option).

Figure 2.9:

Step 2: Configure the database connection.

1. At the bottom of the **Web APIs** tab of the PowerServer project painter, click the **Database Configuration** button.

2. In the **Database Configuration** window, click **DB Drivers** in the upper part to make sure the corresponding database driver and the option "I have read and agree to the license ..." both are selected.

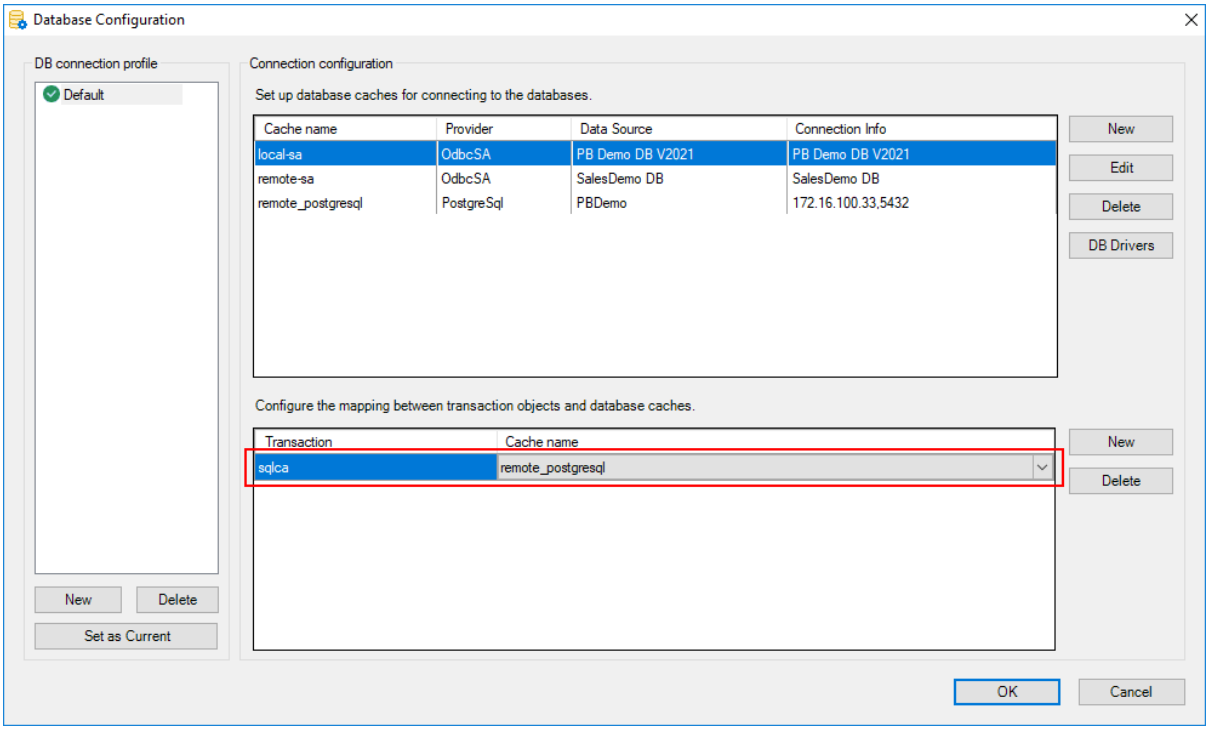
3. In the **Database Configuration** window, click **New** in the upper part to create the database connection that will be used by the deployment.

4. In the dialog box that displays, configure the database connection settings (using the **PBDemo** PostgreSQL database in this tutorial).

Figure 2.10:

5. Select the database cache you created just now and map it to the "sqlca" transaction object.

Figure 2.11:



Step 3: Save the PowerServer project settings.

Step 4: Build and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) for the changes to take effect.

2.3.3 Editing the pg_hba.conf file

Open the **pg_hba.conf** file in a text editor and add the following line.

The **pg_hba.conf** file is located in %PostgreSQL%\12\data. This enables the database server to allow remote connections.

```
host    all             all             0.0.0.0/0               md5
```

2.3.4 Publishing Web APIs to Docker

Step 1: Open the PowerServer C# solution in SnapDevelop.

Click the **Open C# Solution in SnapDevelop** button ( ) in the toolbar to launch the PowerServer C# solution in SnapDevelop. Or go to the location where the PowerServer C# solution is generated; and double click **PowerServer_[appname].sln** to launch the solution in SnapDevelop.

At startup, the solution will install/update the dependencies. Wait until the **Dependencies** folder completes the install/update. (Make sure the machine can connect to the NuGet site: https://www.nuget.org in order to successfully install PowerServer NuGet packages).

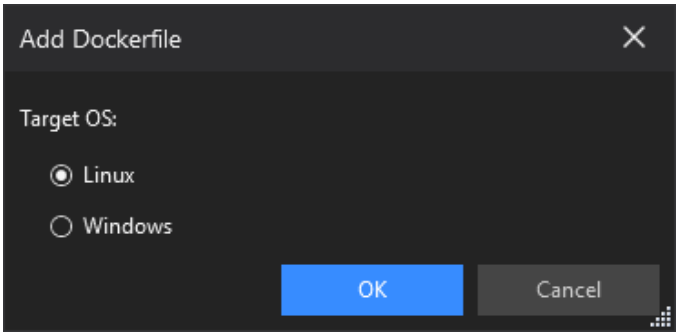
Step 2: Add docker support to the **ServerAPIs** project.

1. In the Solution Explorer, right click on the **ServerAPIs** project node, and select **Add** > **Docker Support**.

2. In the **Add Dockerfile** dialog, select the target OS: **Linux** or **Windows**, and click **OK**. The target OS indicates the platform where Docker Engine and Docker Container are running. In this tutorial, select **Linux**.

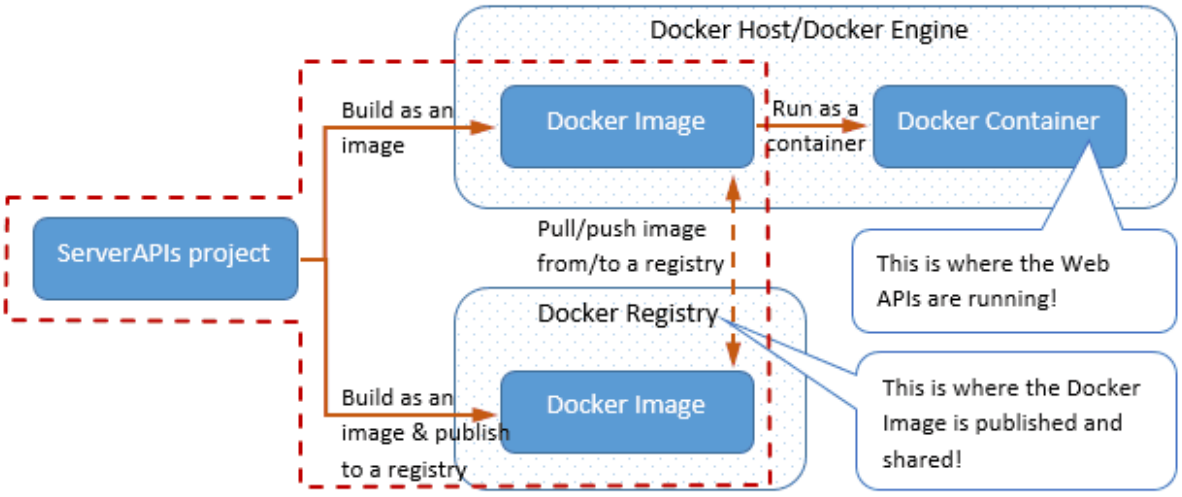
   A file named **Dockerfile** is automatically created according to the selected OS and added under the **ServerAPIs** project. This file contains all the commands required for building a docker image appropriate for the selected OS.

   Figure 2.12:



Step 3: Build and publish the **ServerAPIs** project as a docker image.

Figure 2.13:



1. In the Solution Explorer, right click on the **ServerAPIs** project node, and select **Publish**.

2. In the window that appears, select **Docker**, and then click **Start** to configure for publish.

   a. Keep **Publish to Personal Repository** checked if you are connecting to your own repository (not part of an organization). If the repository is owned by an organization, clear the checkbox, and enter the organization name.

   b. In the **Engine** field, select the machine where Docker Engine is installed.

If you select **localhost**, make sure you have installed Docker Engine on the local machine; if you select a remote machine, make sure you have installed Docker Engine to that machine and configured Docker Engine to allow remote connection. See Setting up a docker host (Docker Engine) for more.

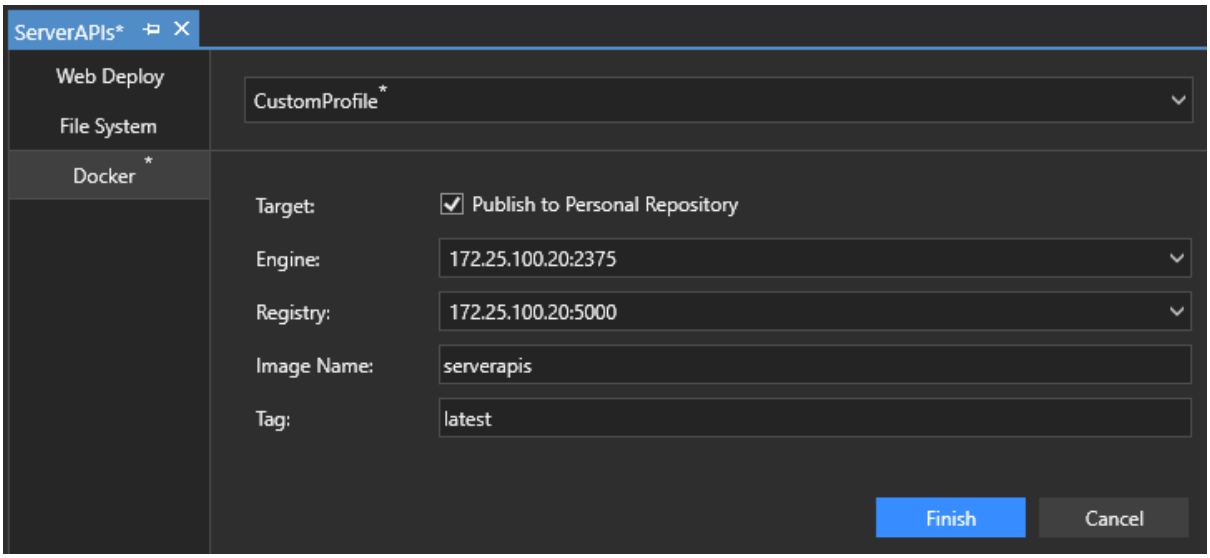
c. In the **Registry** field, specify where to store the docker image: Docker Hub or a self-hosted Docker Registry. See Setting up a docker registry for more.

   If you specify a repository in Docker Hub, you will need to enter your Docker username and password.

d. In the **Image Name** field, enter a name for the docker image you want to create for the project.

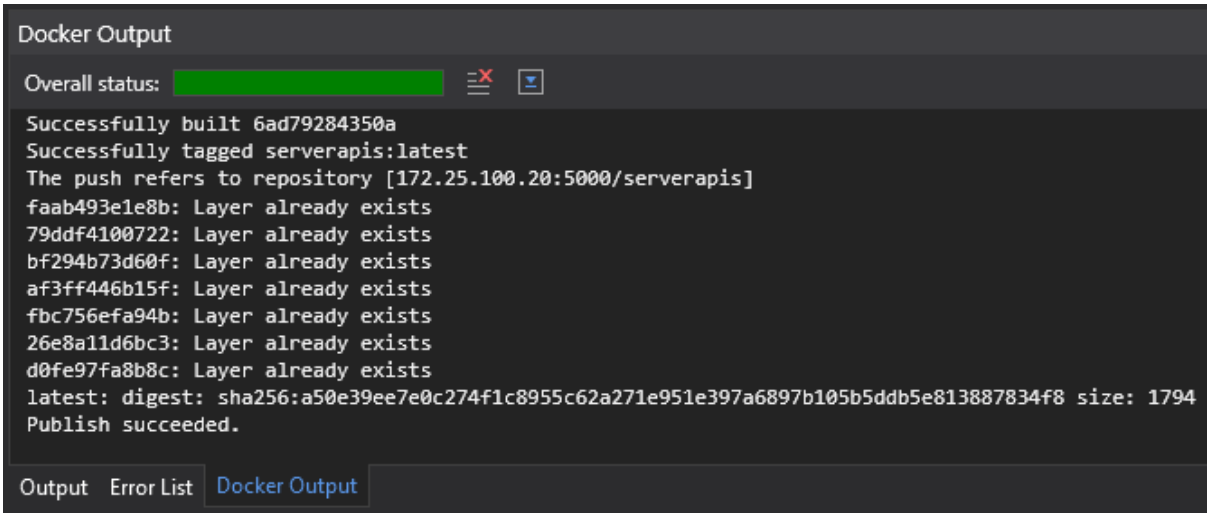
e. Click **Finish** to start building the project as an image and publishing the image to the specified Docker Engine and docker registry.

Figure 2.14:



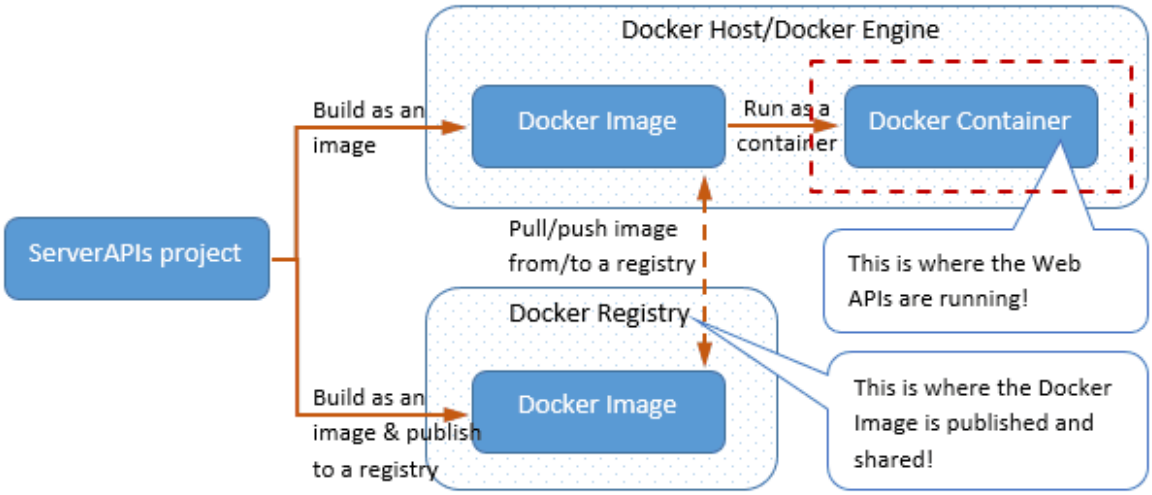
Check the **Docker Output** window and make sure the publish is successful.

Figure 2.15:



Step 4: Run the docker image as a docker container.

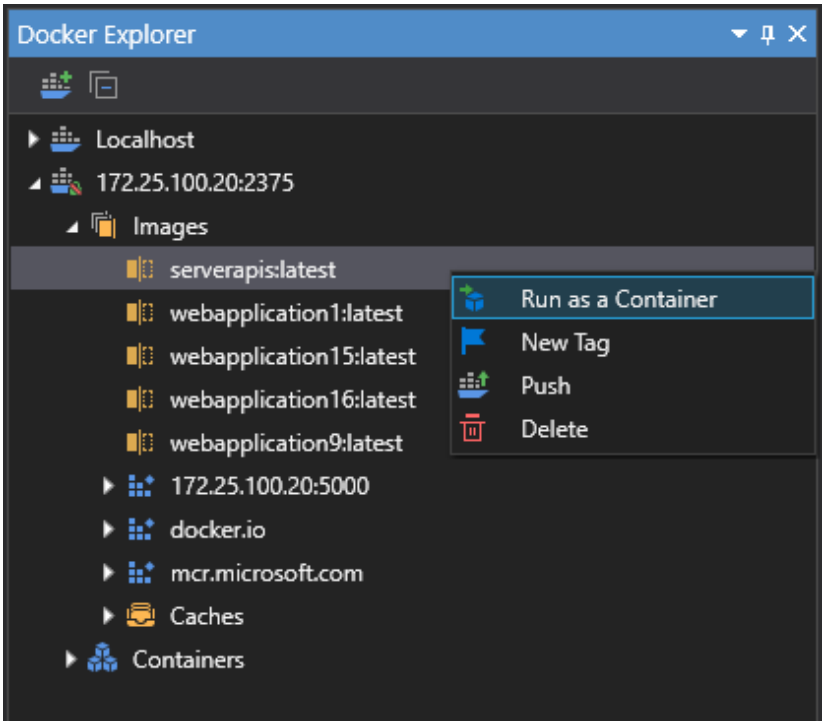
Figure 2.16:



1. In SnapDevelop, select **View** > **Docker Explorer** to open the Docker Explorer.

2. In the Docker Explorer, expand the node for the machine where Docker Engine is, and then expand **Images** and find the image that is created for the project, right click it and select **Run as a Container**.

Figure 2.17:



3. In the window that appears, specify the following settings for the container, and click **OK**.

   - Specify a name for the container.

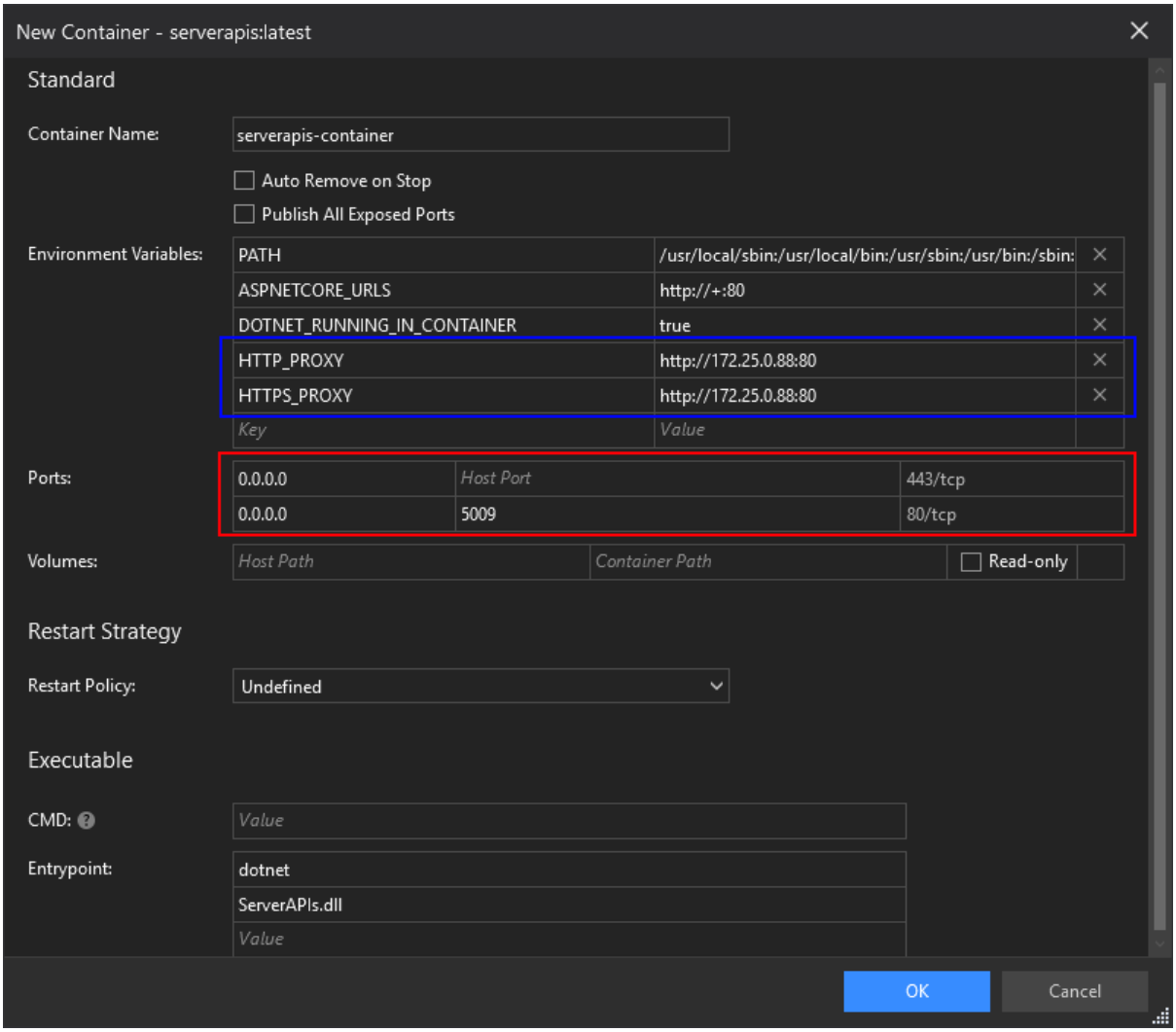
   - Specify the port number for the Web APIs in the container. Leave the IP address with the default value 0.0.0.0 which will automatically point to the IP address for Docker Engine where the container is running.

     IMPORTANT:

     1. The IP address and port number must match with the Web API URL specified on the **Web APIs** tab of the PowerServer project painter. And the actual IP address (instead of 0.0.0.0) should be specified in the Web API URL (view **Web API URL**).

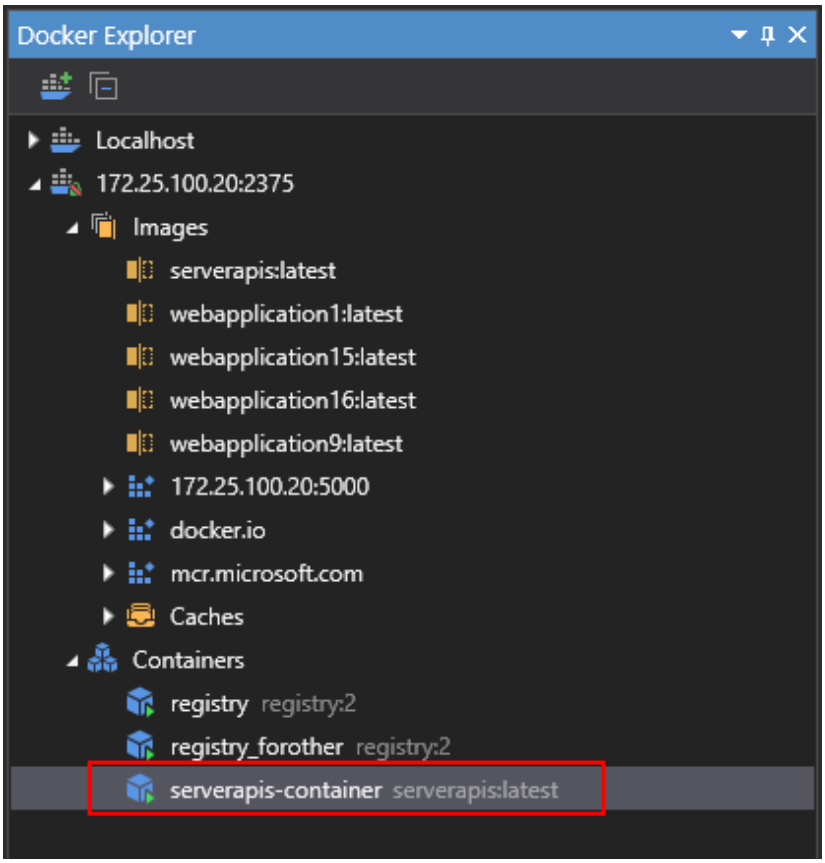
     2. If the docker host machine connects to Internet via a proxy server, configure the proxy settings as the environment variables (as shown in the blue frame below); or refer to Configure Docker to use a proxy server for detailed instructions.

Figure 2.18:



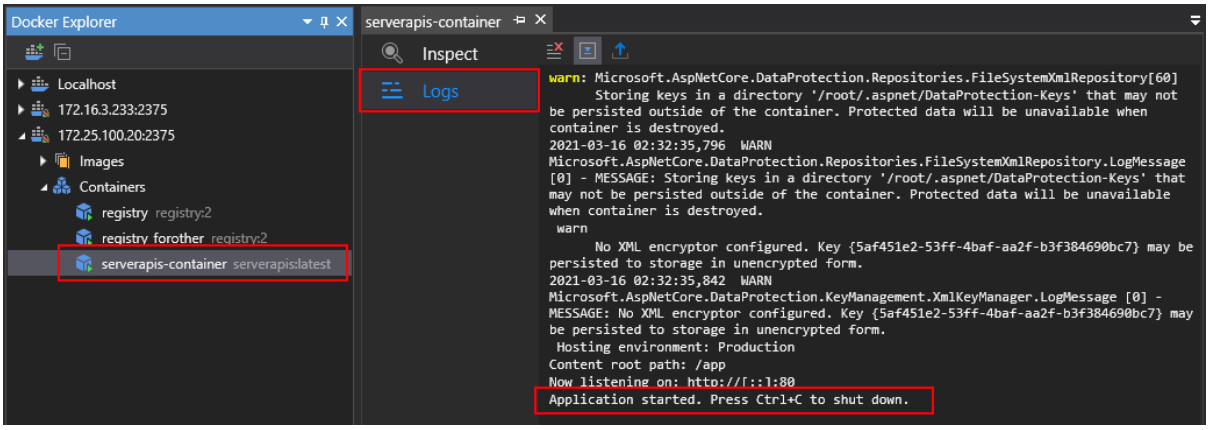
The container is started and added under the **Containers** node. You can stop, restart, or delete the container, or execute commands using the right-click context menu.

Figure 2.19:



If you double click the container, the container configuration and log will be displayed on the right. The Logs section displays valuable logging information of the Web APIs at runtime.

Figure 2.20:



2.3.4.1 Specifying Web API URL

Specify where the Web APIs is hosted and run.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the docker container where the Web APIs is running, for example, https://172.16.100.20:5009. It is highly recommended that you specify an HTTPS URL for the production environment.

IMPORTANT: if the host name and port number of the docker container are changed later, you only need to update the Web API URL and then deploy the project again (using the "Deploy PowerServer Project" option) (it is not necessary to update or re-deploy Web APIs to Docker).

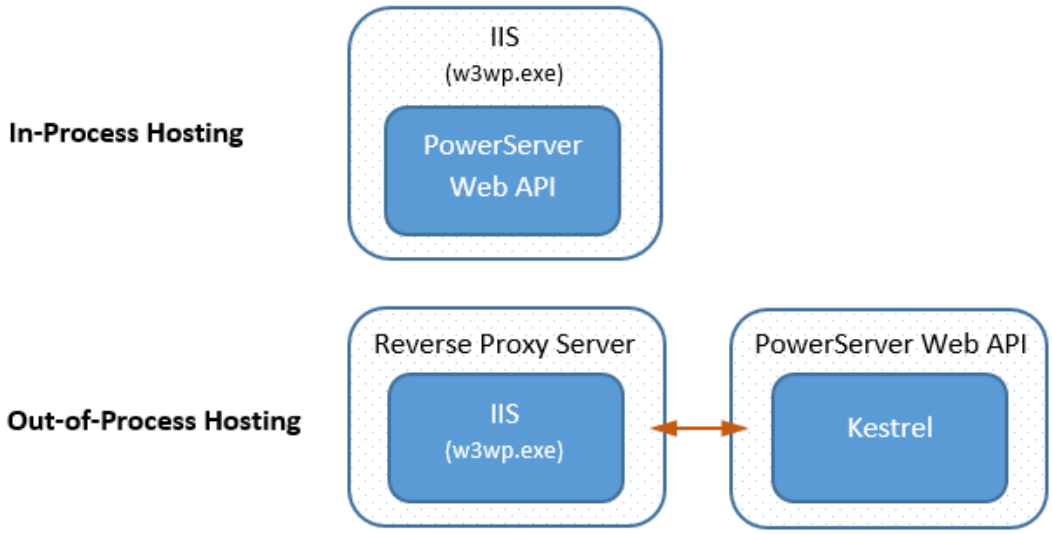
Figure 2.21:

3 Tutorial 3: Hosting Web APIs in IIS (in-process hosting)

3.1 Overview

The PowerServer Web APIs can be directly hosted inside of an IIS Application pool and run in the same process as its IIS worker process (w3wp.exe); this is known as in-process hosting. It is different from the out-of-process hosting which runs the PowerServer Web APIs in a process separate from the IIS worker process and forwards the requests made to the IIS reverse proxy to the Kestrel server.

Figure 3.1:



This tutorial talks about the in-process hosting. The configuration of IIS reverse proxy server for the out-of-process hosting will be discussed in Using Kestrel with IIS reverse proxy server.

To implement the in-process hosting of the PowerServer Web APIs in IIS, you will need to publish Web APIs to IIS using the following methods:

- Web Deploy -- directly publishes Web APIs to the specified IIS website. You can deploy to the IIS website on the local or remote server.

  To deploy to an IIS website on the local server (e.g. IIS on Windows 10), you will need to set up the server in this way:

  1. Install IIS
  2. Create an IIS website
  3. Install Web Deploy 3.6 (or later) & ASP.NET Core Hosting Bundle 3.1

  To deploy to an IIS website on the remote server (e.g. IIS on Windows Server 2019), you will need to set up the server in this way:

1. Install IIS
2. Create an IIS website
3. Configure IIS
4. Install Web Deploy 3.6 (or later) & ASP.NET Core Hosting Bundle 3.1

- File System -- publishes Web APIs to a local folder. You need to manually copy the published folders and files to the web root of the IIS website later.

  This will require you to set up the server in this way:

  1. Install lIS
  2. Create an IIS website
  3. Install ASP.NET Core Hosting Bundle 3.1

3.2 Preparations

In this tutorial, we will set up a server running on IIS in an independent machine, and then publish and host the Web APIs in the IIS running on this server.

Step 1: Set up the server with the following OS and software (install the software in the order listed).

- Windows Server 2019 (64-bit)
- Microsoft IIS

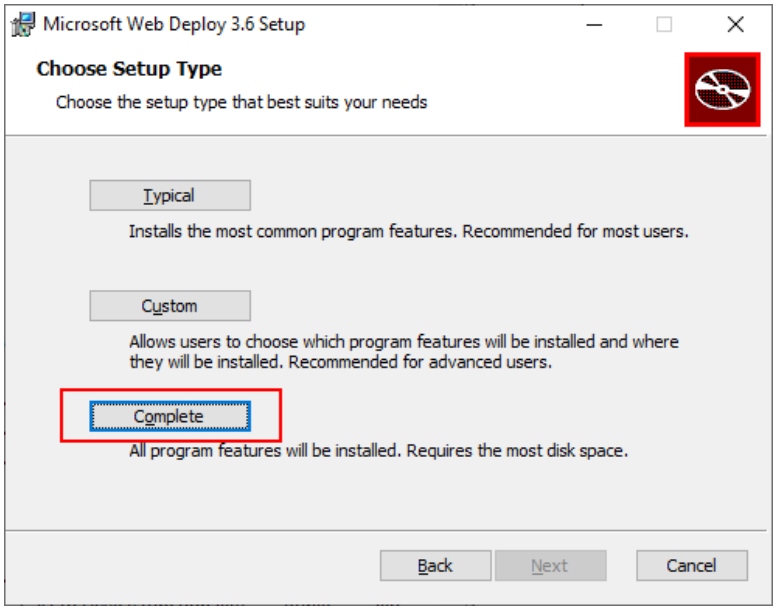
  Follow the section below to install and configure IIS.

- Web Deploy 3.6 (or later)

  Download and install from https://www.microsoft.com/download/confirmation.aspx?id=43717.

  IMPORTANT: Make sure to select the **Complete** setup type when installing Web Deploy.

Figure 3.2:



When the installation is complete, select **Control Panel** > **System and Security** > **Administrative Tools** > **Services**, and make sure "**Web Deployment Agent Service**" is running.

Important

Web Deploy must be installed **after** IIS is installed. If you have installed Web Deploy before IIS, uninstall Web Deploy and then choose the **Complete** setup type to install it again; do not use the **Modify** feature to re-install Web Deploy.

- ASP.NET Core Hosting Bundle 3.1

  Download and install from https://dotnet.microsoft.com/download/dotnet/thank-you/runtime-aspnetcore-3.1.13-windows-hosting-bundle-installer.

Step 2: Make sure the .NET server can connect to the NuGet site: https://www.nuget.org (for installing PowerServer NuGet packages) and the following Appeon sites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) (for validating the PowerServer license).

Note

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Step 3: Configure Windows Defender Firewall on the .NET server to allow the .NET server port (81 in this tutorial or any port number you choose). The section "Configuring Windows Defender Firewall" has detailed instructions.

3.3 Installing IIS

3.3.1 Windows Server OS

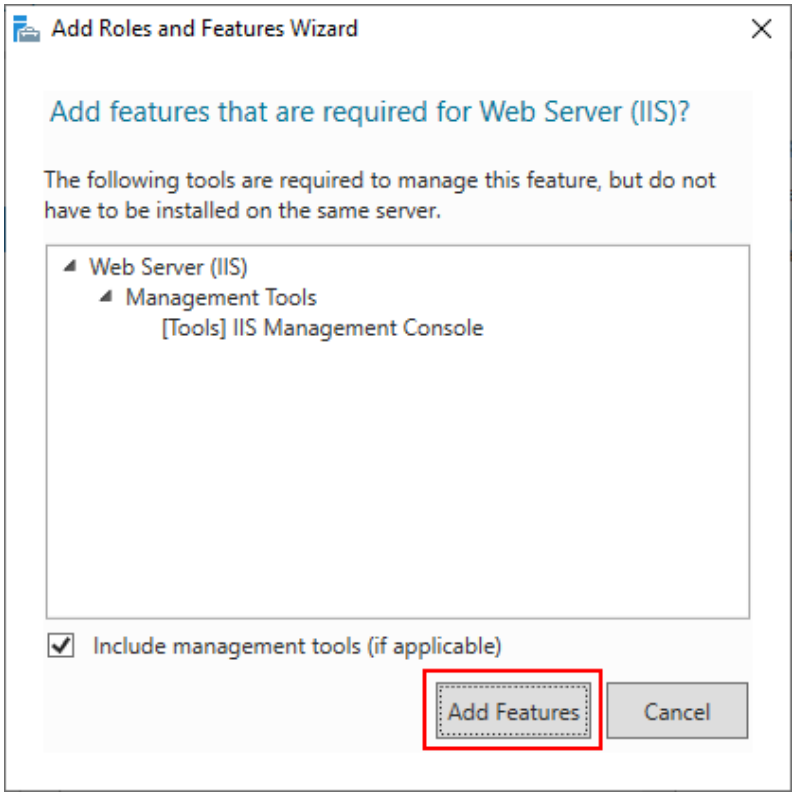
The following steps take Windows Server 2019 as an example:

Step 1: In Windows Server 2019, open **Server Manager**, and then select **Add roles and features**.

Step 2: In the **Add Roles and Features Wizard**, click **Next** several times until the **Server Roles** section displays.

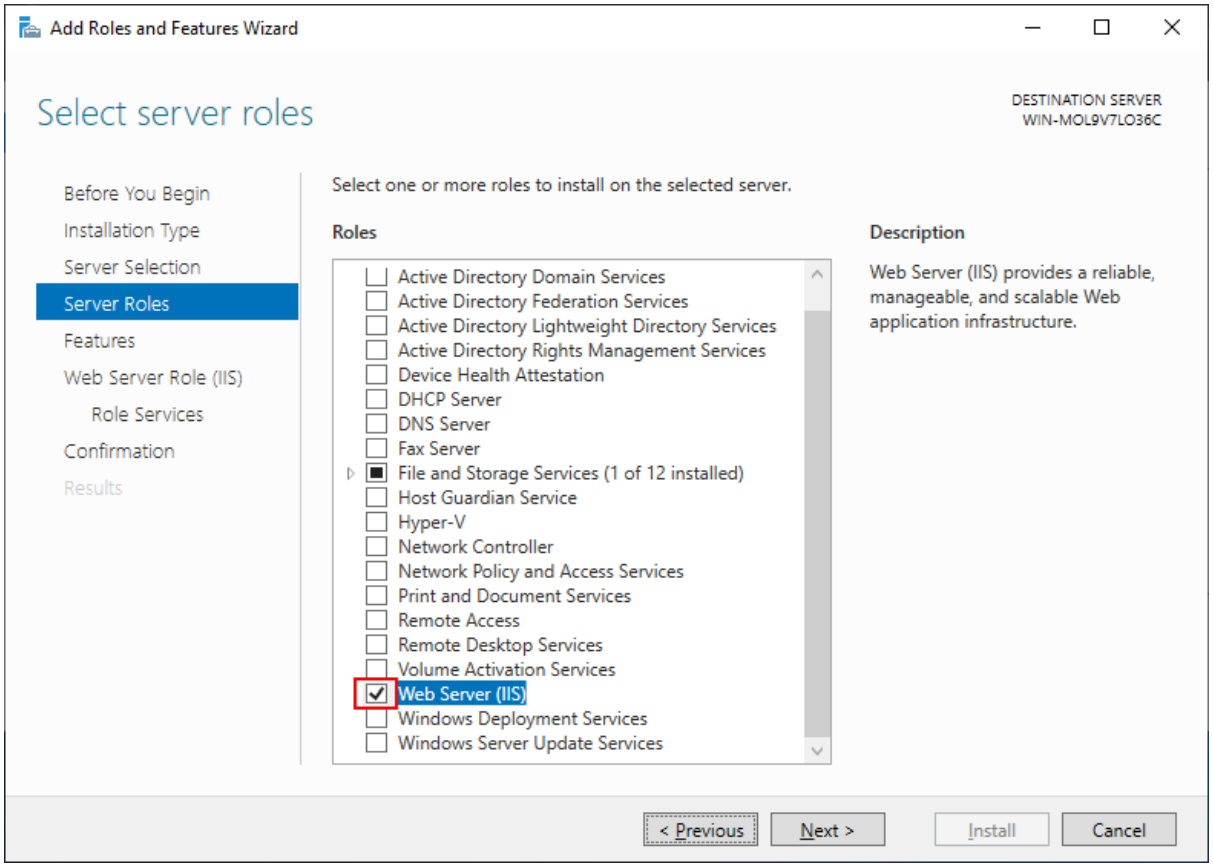
Step 3: Select the check box of **Web Server (IIS)**; and then click **Add Features** when asked whether to add features required for Web server.

Figure 3.3:



Step 4: Make sure the **Web Server (IIS)** check box is selected, and click **Next**.

Figure 3.4:

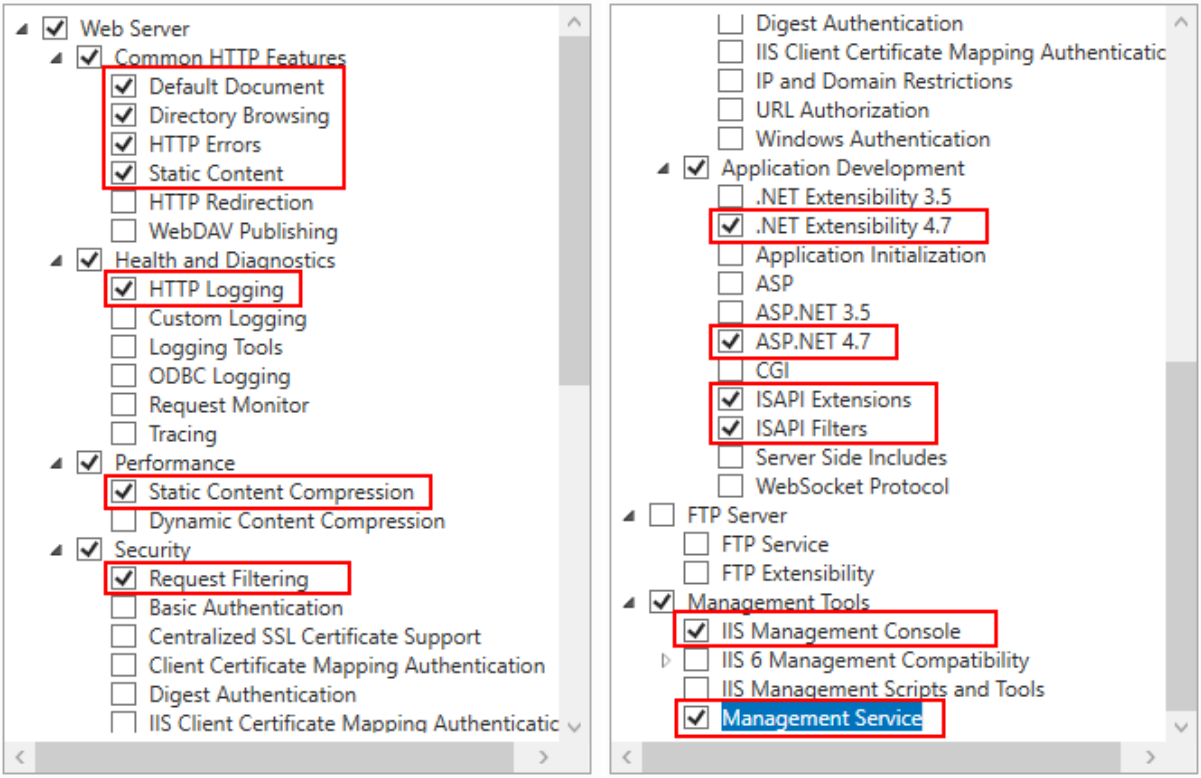


Step 5: Click **Next** until the **Role Services** section displays. Make sure the following role services are selected.

IMPORTANT: **Management Service** must be selected and installed otherwise the IIS remote management will not be supported. If IIS remote management is not supported, then you will not be able to deploy to IIS from a remote computer; you will only be able to deploy to IIS from the local computer.

Management Service is only available on Windows Server OS; and is **not** available on Windows Desktop OS (such as Windows 10); which means if you have installed IIS on Windows 10, you can only do a local deployment (instead of remote deployment) to IIS.

Figure 3.5:



Step 6: Click **Next** and then click **Install**.

After IIS is installed, a **Default Web Site** (with port 80) is automatically created (you could also create new websites with different port numbers).

Step 7: Open a Web browser and run the following URLs to access the **Default Web Site**.

http://localhost:80/

http://*your_server_ip*:80/

TIP: You can use "localhost" or the IP address to access the IIS website on the local computer. To obtain the IP address, open a command prompt window and then type ipconfig<Enter>.

If the IIS welcome screen displays, the IIS website is working properly.

3.3.2 Windows Desktop OS

The following steps take Windows 10 as an example:

Step 1: In Windows 10, navigate to Control Panel > Programs > Programs and Features > Turn Windows features on or off (left side of the screen).

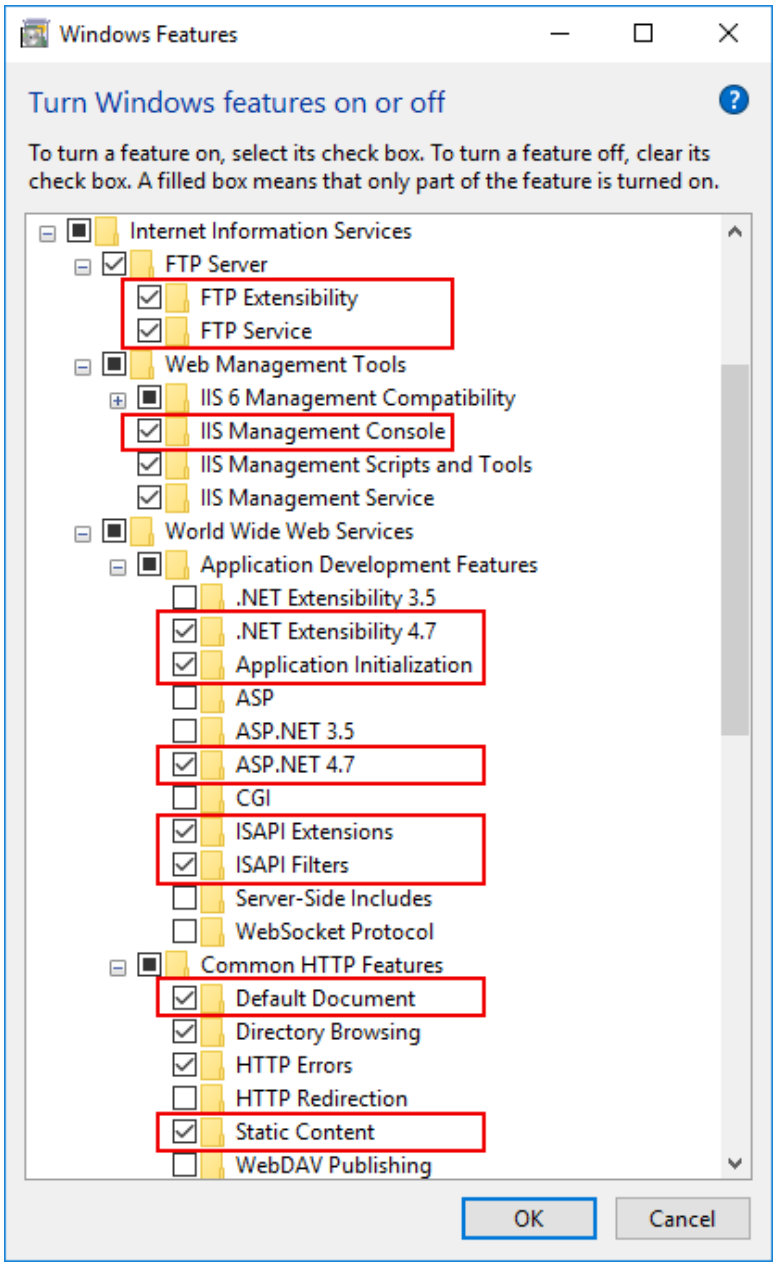
Step 2: Expand the **Internet Information Services** node and make sure the following features are selected.

- **FTP Service**
- **FTP Extensibility**

  FTP Service & **FTP Extensibility** must be enabled if you want to create an IIS FTP site for transferring files from a remote development machine to the Web server.

- IIS Management Console
- .NET Extensibility 4.7
- Application Initialization
- ASP.NET 4.7
- ISAPI Extensions
- ISAPI Filters
- Default Document
- Static Content

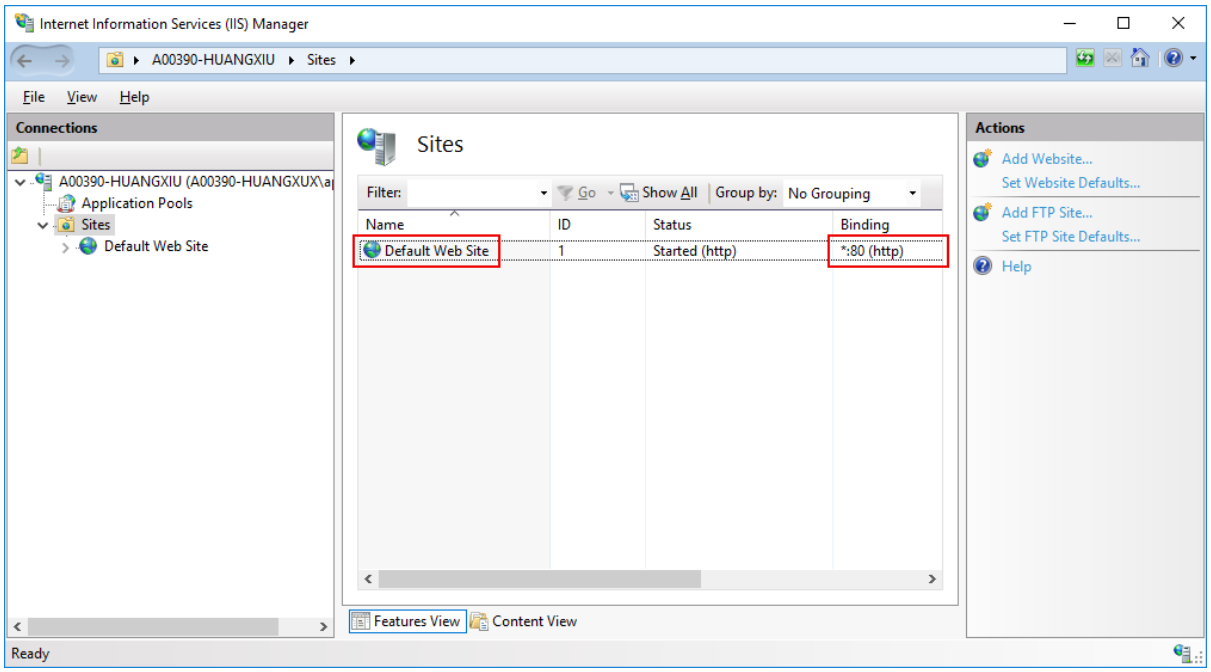
Figure 3.6:



Step 3: Click **OK** to install the selected features.

After IIS is installed, a **Default Web Site** (with port 80) is automatically created (you could also create new websites with different port numbers).

Figure 3.7:



Step 4: Open a Web browser and run the following URLs to access the **Default Web Site**.

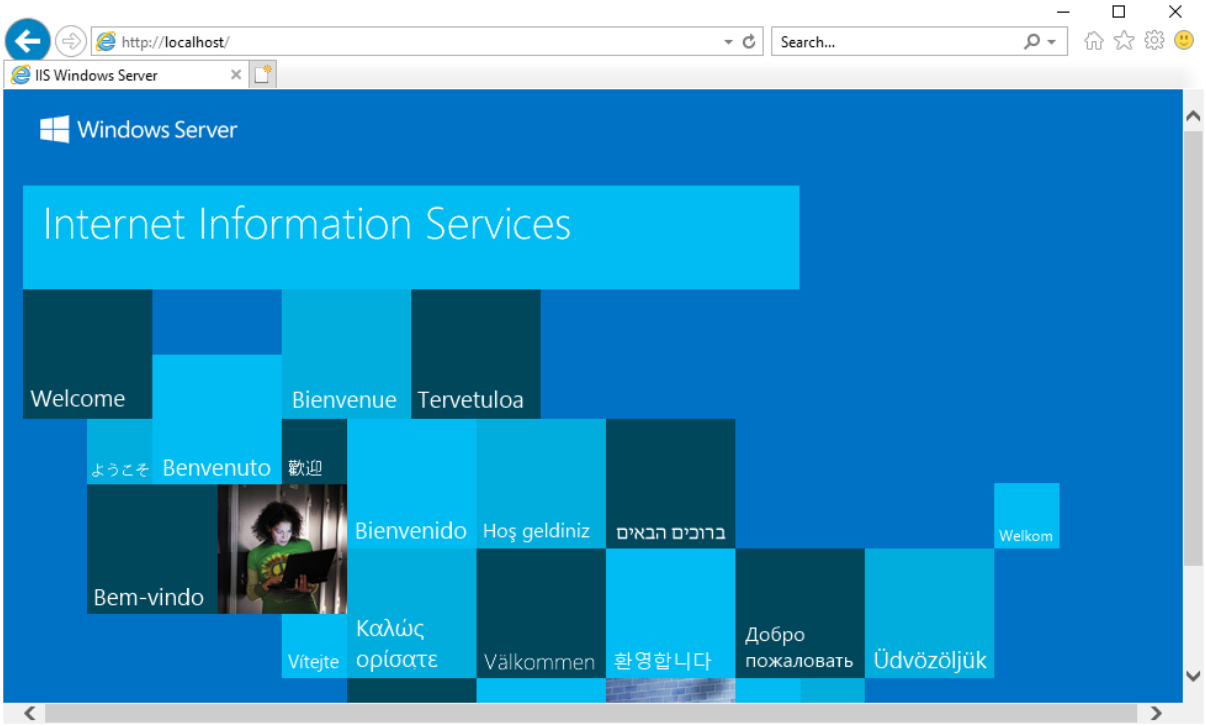
http://localhost:80/

http://*your_server_ip*:80/

TIP: You can use "localhost" or the IP address to access the IIS website on the local computer. To obtain the IP address, open a command prompt window and then type ipconfig<Enter>.

If the IIS welcome screen displays, the IIS website is working properly.

Figure 3.8:



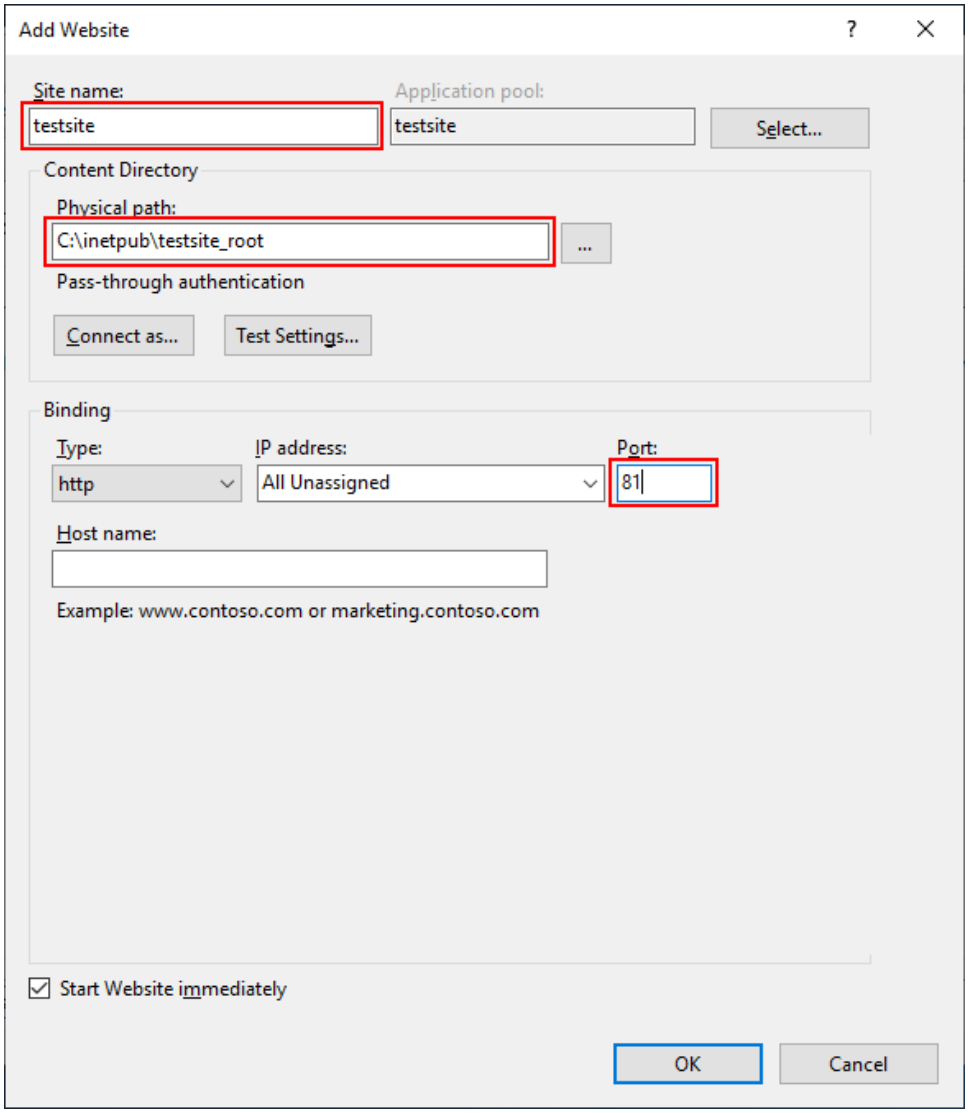
3.4 Creating an IIS website

Step 1: In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.

Step 2: Specify the following values and the click **OK** to create the site.

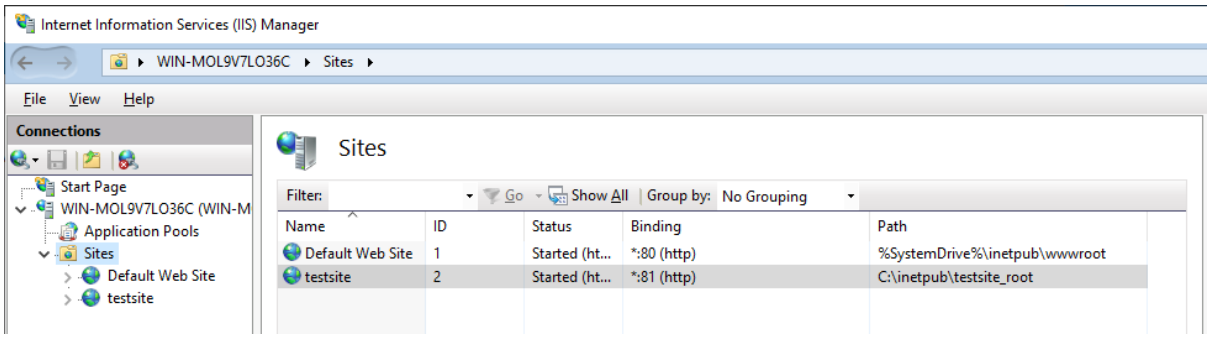
- **Site name**: *testsite* in this example
- **Physical path**: C:\inetpub\testsite_root in this example. This is where the folders and files of Web APIs will be published.
- **Port**: *81* in this example.

Figure 3.9:



The IIS website is created and started.

Figure 3.10:



Open a Web browser and run the following URLs to access the new website.

http://localhost:81/

http://*your_server_ip*:81/

If the IIS welcome screen displays, then the website is working properly.

3.5 Configuring IIS

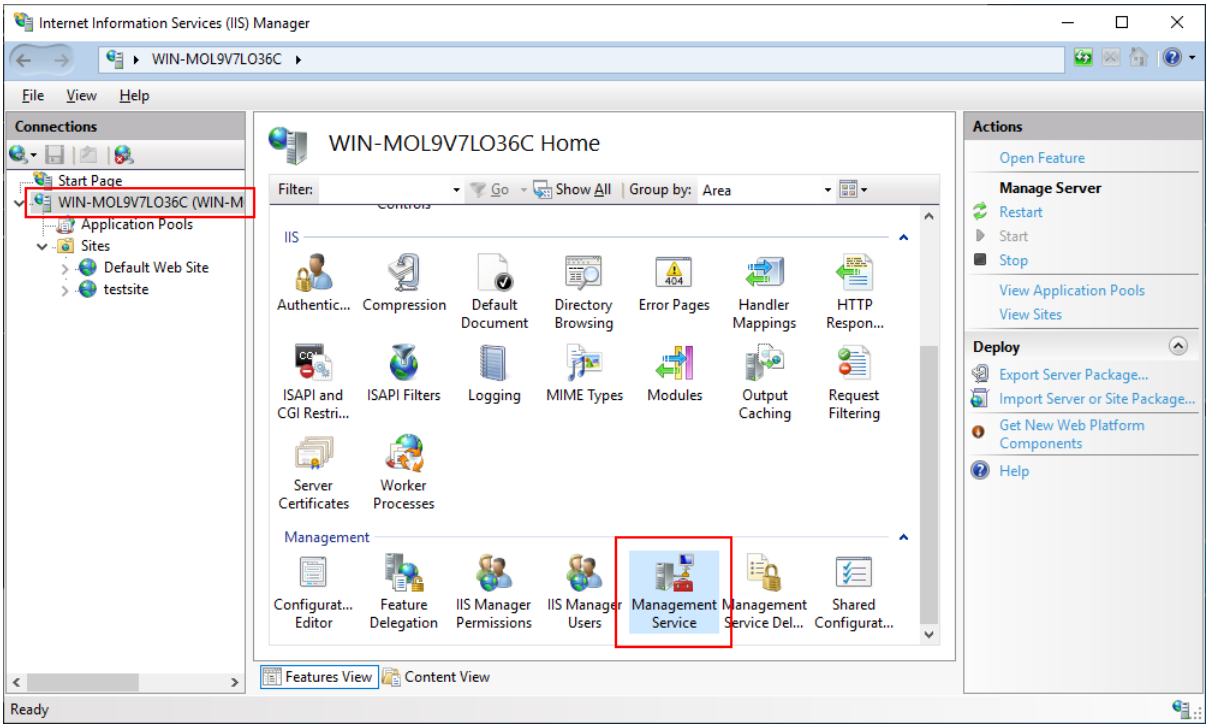
This section is to configure IIS to support remote deployment. You can skip this section if you will deploy to IIS from the local computer, for example, if you want to deploy to IIS on Windows 10 which supports only local deployment, or if you want to deploy to a local folder first and then manually copy the published files to IIS.

Step 1: Enable remote connections for the IIS server.

1. In IIS Manager, select the server's node in the **Connections** panel, and then double click **Management Service** on the **Features View**.

   Note: The **Management Service** feature is available only when you select the **Management Service** feature when installing IIS.

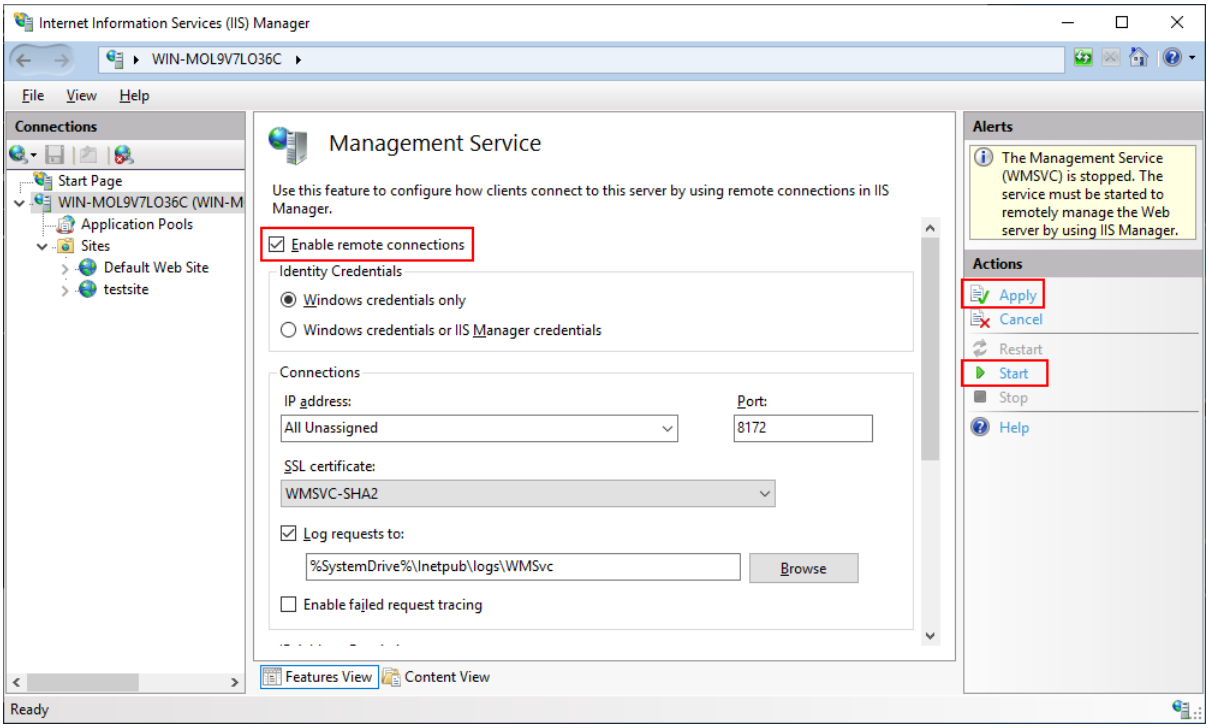
   Figure 3.11:



2. Select the check box of **Enable remote connections**, and then click **Apply** and **Start** in the **Actions** pane to start the management service.

   If Management Service is already running, click **Stop** in the **Actions** pane to stop the service first before you can make changes to it.

Figure 3.12:

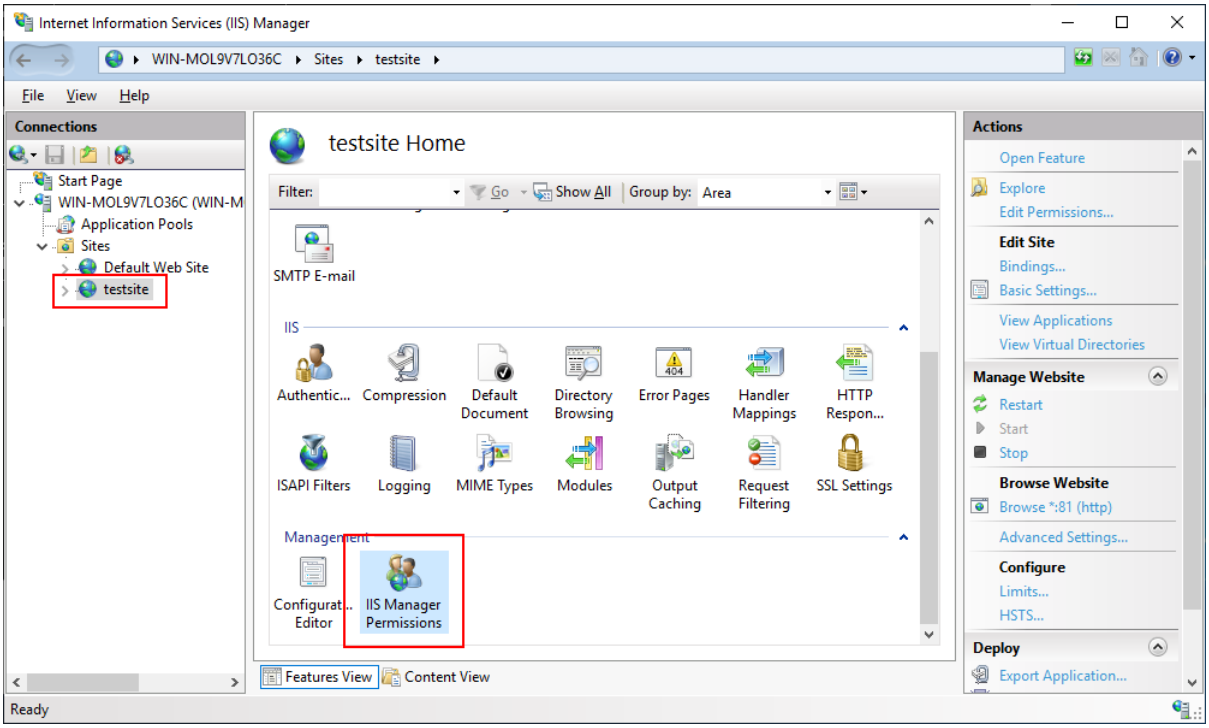


3. Select **Control Panel** > **System and Security** > **Administrative Tools** > **Services**, and make sure the "**Web Management Service**" service is running.

Step 2: Configure the IIS website to allow the Windows user to connect to the site.

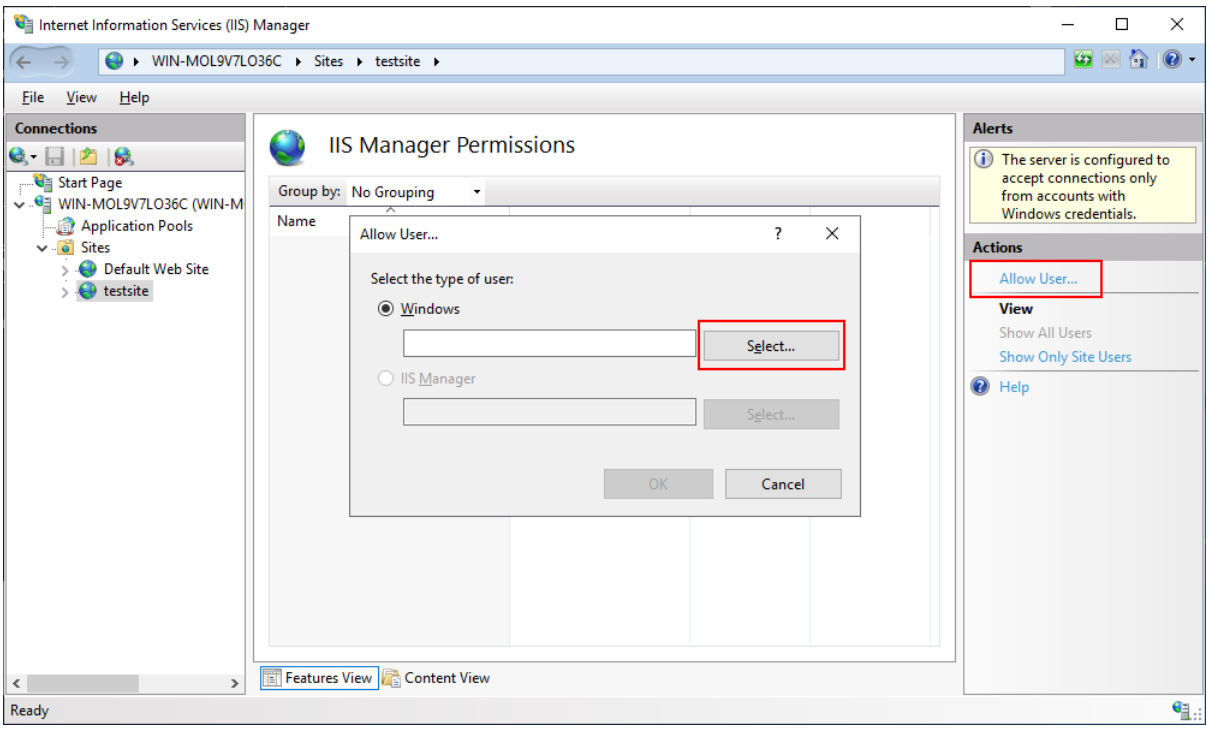
1. In IIS Manager, select the website in the **Connections** panel, and then double click **IIS Manager Permissions** on the **Features View**.

Figure 3.13:



2. Click **Allow User** on the **Actions** pane. In the **Allow User** dialog, click **Select**.

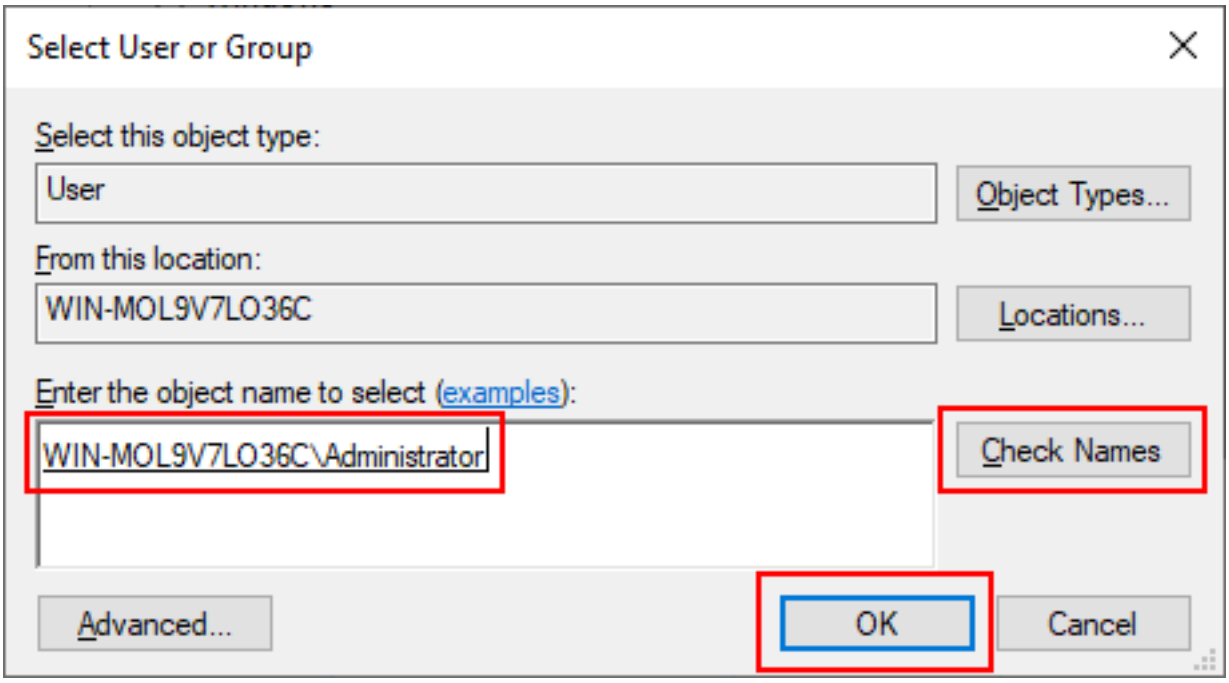
Figure 3.14:



3. Enter the Windows user name, click **Check Names** and then click **OK**.

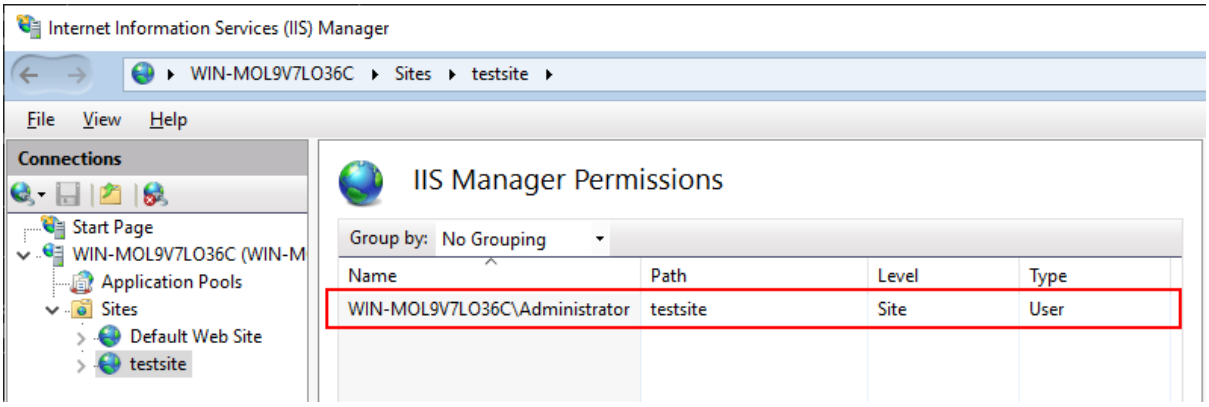
   IMPORTANT: Make sure to use a Windows user that has Full Control over the site's root folder so that it can create files and folders underneath.

Figure 3.15:



The Windows user is added to **IIS Manager Permissions**. This Windows user is allowed to connect to the "testsite" site now.

Figure 3.16:



3.6 Configuring SSL on IIS

It is highly recommended that you configure Secure Sockets Layer (SSL) for IIS, so that HTTPS can be used to secure the connections between the client and IIS.

For how to configure SSL on IIS, refer to https://docs.microsoft.com/en-us/iis/manage/configuring-security/how-to-set-up-ssl-on-iis.

3.7 Publishing Web APIs to IIS

The following uses the Web Deploy method to publish Web APIs to IIS:

Step 1: On the development machine, open the PowerServer C# solution in SnapDevelop. Log in to SnapDevelop if required.

Click the **Open C# Solution in SnapDevelop** button () in the toolbar to launch the PowerServer C# solution in SnapDevelop. Or go to the location where the PowerServer C# solution is generated; and double click **PowerServer_[appname].sln** to launch the solution in SnapDevelop.

At startup, the solution will install/update the dependencies. Wait until the **Dependencies** folder completes the install/update. (Make sure the machine can connect to the NuGet site: https://www.nuget.org in order to successfully install PowerServer NuGet packages).

Step 2: In the **Solution Explorer**, right click on the **ServerAPIs** project node, and select **Publish**.

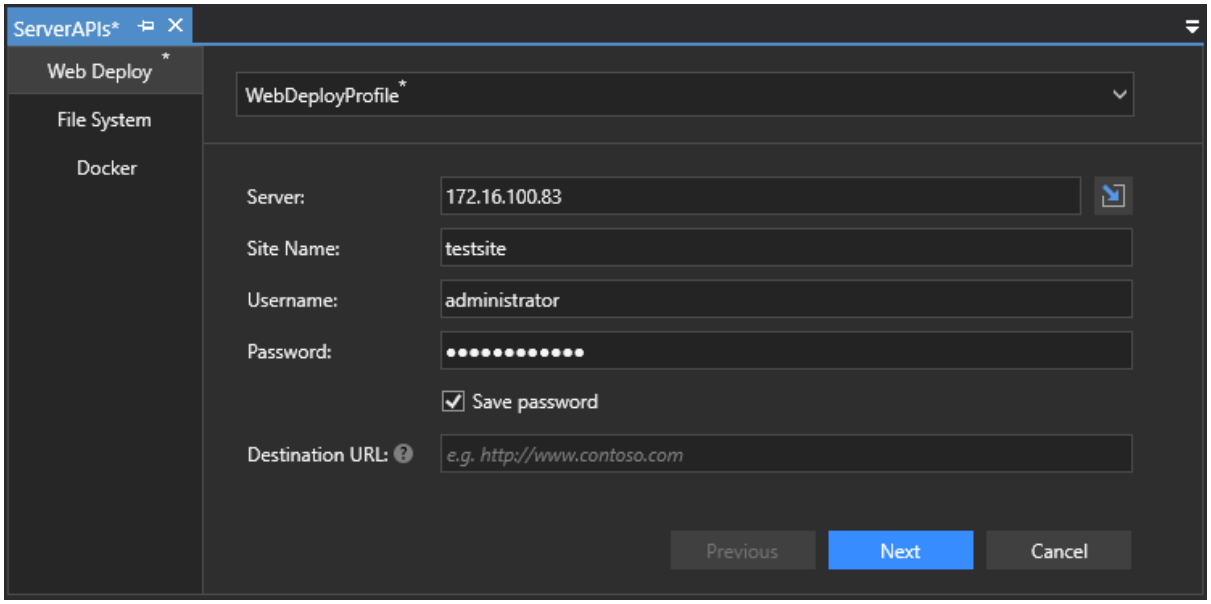
Step 3: In the window that appears, select **Web Deploy**, and click **Start**.

Step 4: Configure the Web deploy profile, and click **Next**.

The following figure shows the settings for deploying to an IIS website on a remote server.

Note: Input the Windows user name that you have configured to allow to connect to the site.

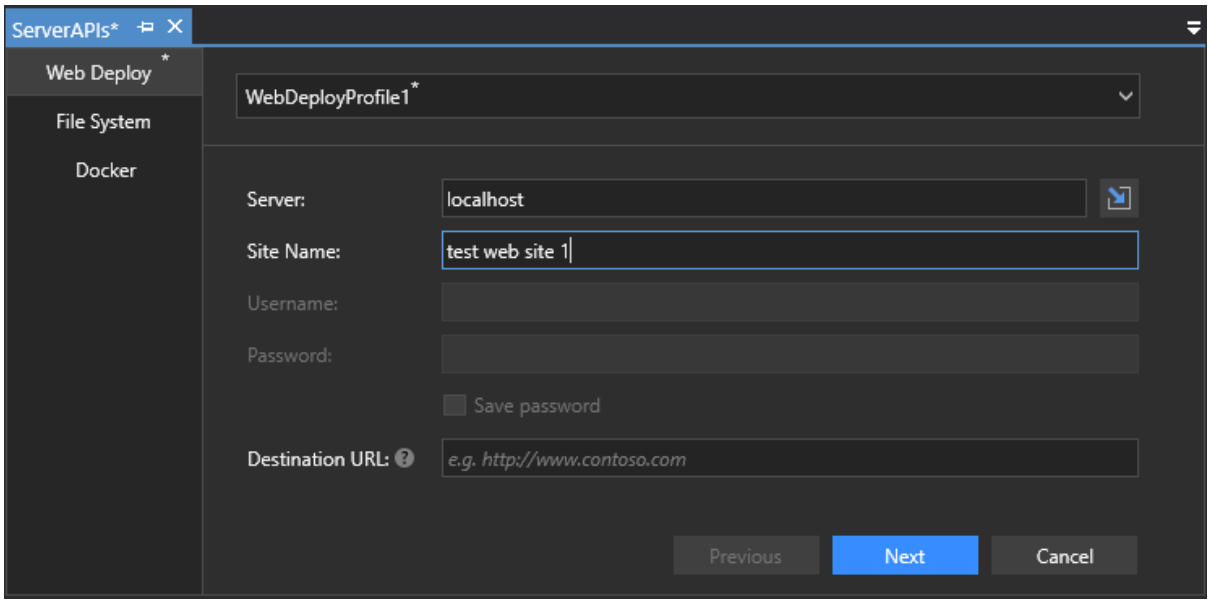
Figure 3.17:



The following figure shows the settings for deploying to an IIS website on the local machine.

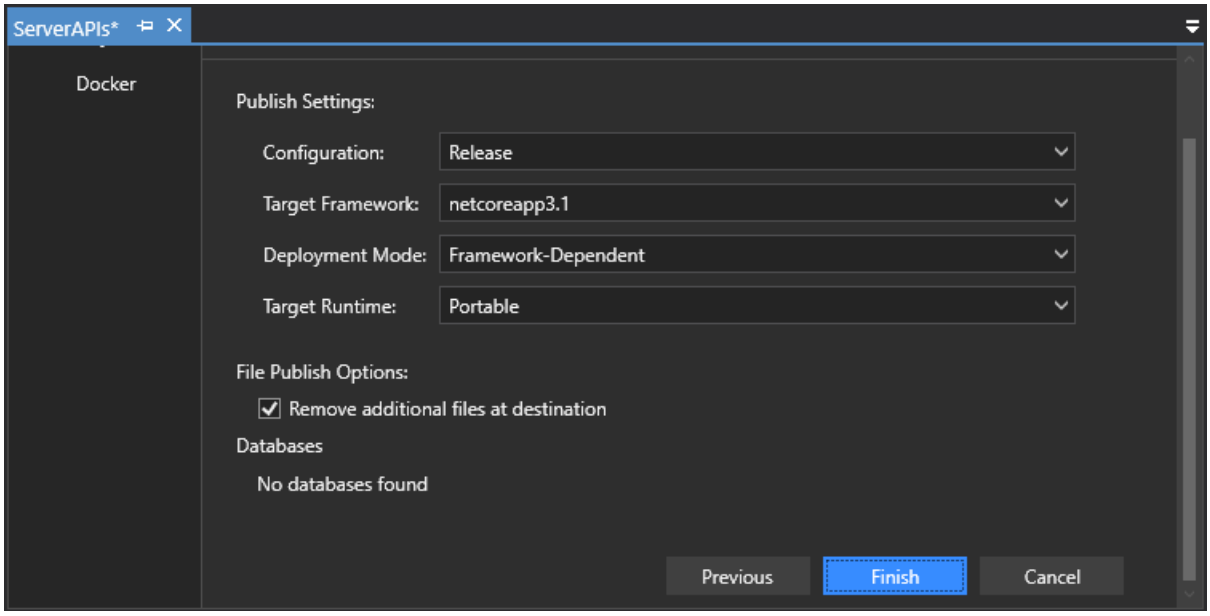
It is not necessary to input username and password when connecting to a local site.

Figure 3.18:



Step 5: Keep the others as default settings and click **Finish**.

Figure 3.19:



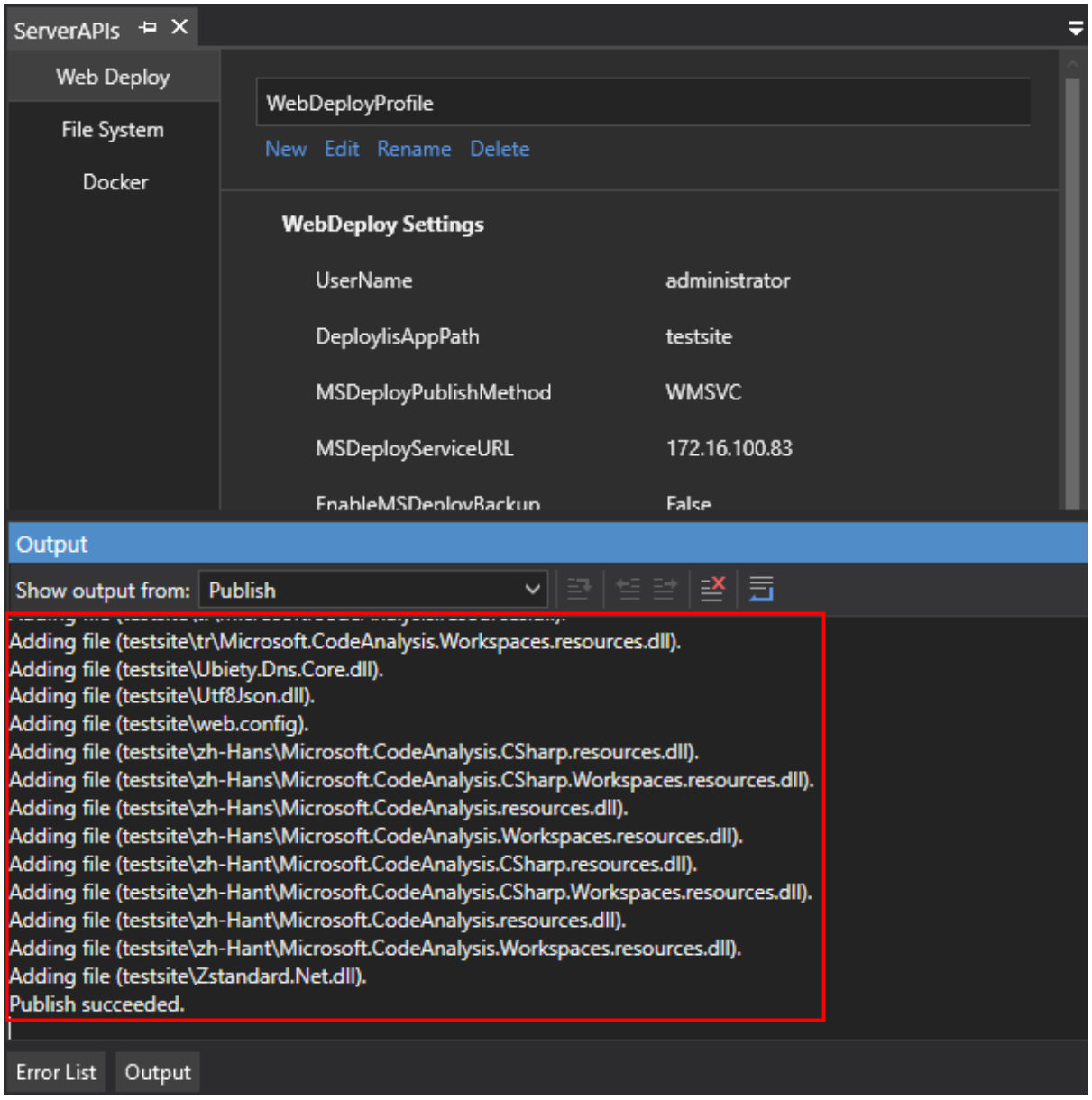
Publishing begins automatically. If any error or failure is reported in the **Output** window, click the link provided at the end to view more details and possible solutions.

Step 6: Make sure publishing was successful as shown in the figure below.

After that you can specify the URL (for example, https://172.16.100.83:81) as the **Web API URL** in the **Web APIs** tab of the PowerServer project painter and then build and deploy the project again.

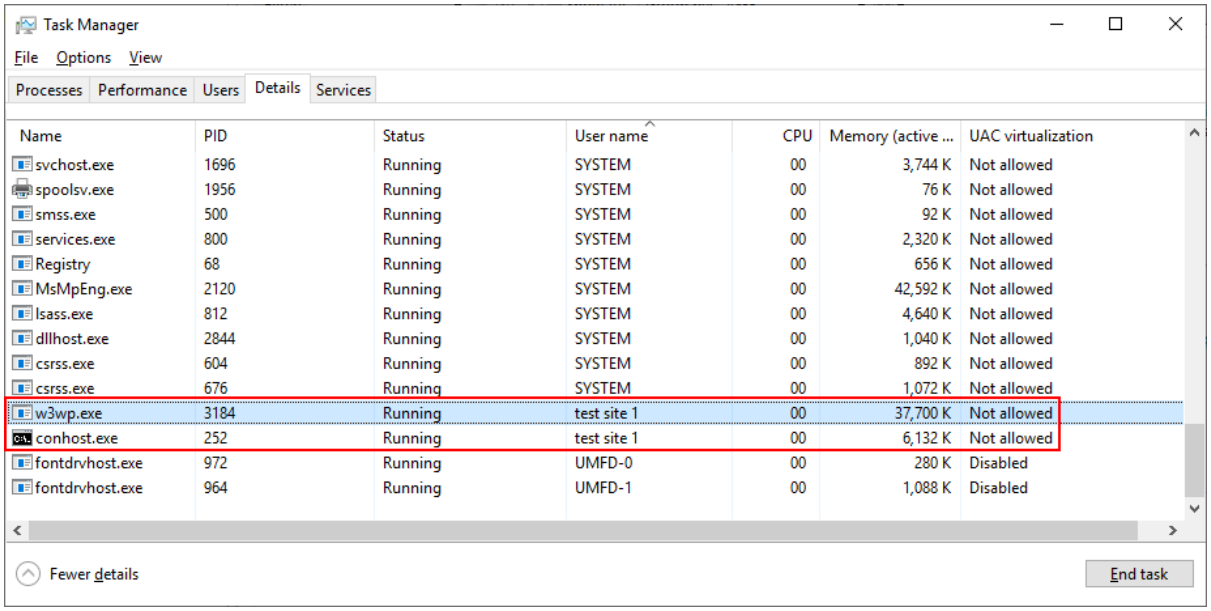
If you use the SQL Anywhere database or ASE database, also set up the corresponding ODBC data source in the server where Web APIs is published and running.

Figure 3.20:



When you run the installable cloud application later, the following two processes will be started in the server and they will launch the PowerServer Web APIs automatically.

Figure 3.21:


Task Manager
File Options View
Processes Performance Users Details Services
Name PID Status User name CPU Memory (active ... UAC virtualization
svchost.exe 1696 Running SYSTEM 00 3,744 K Not allowed
spoolsv.exe 1956 Running SYSTEM 00 76 K Not allowed
smss.exe 500 Running SYSTEM 00 92 K Not allowed
services.exe 800 Running SYSTEM 00 2,320 K Not allowed
Registry 68 Running SYSTEM 00 656 K Not allowed
MsMpEng.exe 2120 Running SYSTEM 00 42,592 K Not allowed
lsass.exe 812 Running SYSTEM 00 4,640 K Not allowed
dllhost.exe 2844 Running SYSTEM 00 1,040 K Not allowed
csrss.exe 604 Running SYSTEM 00 892 K Not allowed
csrss.exe 676 Running SYSTEM 00 1,072 K Not allowed
w3wp.exe 3184 Running test site 1 00 37,700 K Not allowed
conhost.exe 252 Running test site 1 00 6,132 K Not allowed
fontdrvhost.exe 972 Running UMFD-0 00 280 K Disabled
fontdrvhost.exe 964 Running UMFD-1 00 1,088 K Disabled
Fewer details
End task

4 Tutorial 4: Hosting Web APIs in Kestrel

4.1 Overview

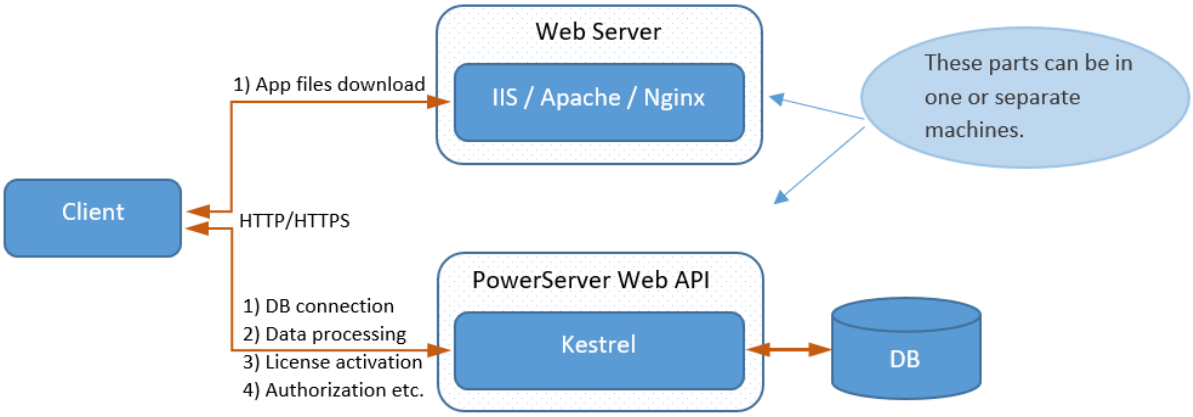
Kestrel is the default web server used for ASP.NET Core applications. When a new ASP.NET Core project is created, it includes the Kestrel web server by default. The Kestrel web server provides better request processing performance to ASP.NET Core applications as it is an open-source, cross-platform and light-weight web server; but it does not have advanced features of web servers like IIS, Nginx, Apache etc.

The PowerServer Web APIs, which is a standard ASP.NET Core application, can be hosted in the Kestrel web server with or without using a reverse proxy server.

In the following graph, the PowerServer Web APIs is hosted in Kestrel and Kestrel is used as an edge (Internet-facing) server without a reverse proxy server.

- Kestrel serves the dynamic content (such as data processing tasks) from the PowerServer Web APIs.
- The web server (such as IIS, Apache, Nignx etc.) and the Kestrel server can reside in the same or different machine.

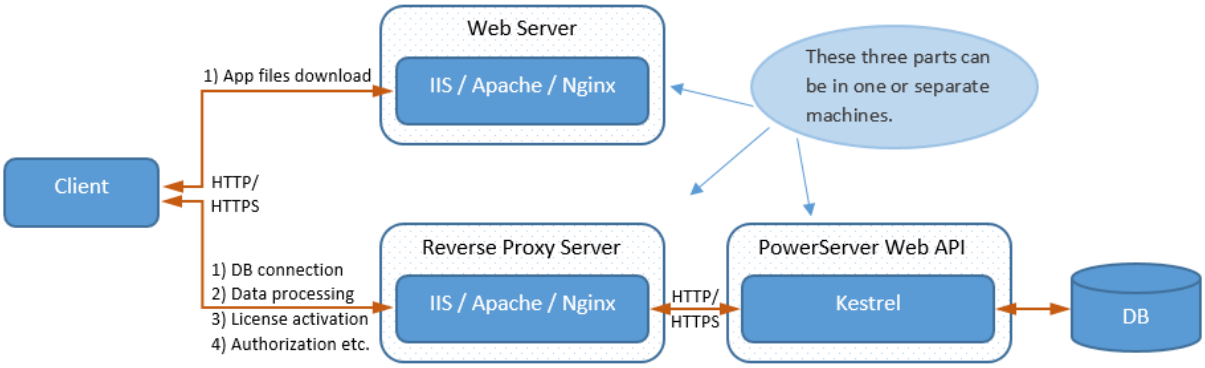
Figure 4.1:



In the following graph, the PowerServer Web APIs is hosted in Kestrel, and Kestrel is used in a reverse proxy configuration.

- Kestrel serves the dynamic content (such as data processing tasks) from the PowerServer Web APIs.
- The reverse proxy server (such as IIS, Nginx, Apache etc.) forwards the requests to the PowerServer Web APIs running in Kestrel. The reverse proxy server may reside on a dedicated machine or may be deployed alongside a web server.
- The web server (such as IIS, Apache, Nignx etc.), the reverse proxy server, and the Kestrel server can reside in the same or different machine.

Figure 4.2:



4.2 About PowerServer Web APIs and Kestrel

As aforementioned, Kestrel is by default used by the ASP.NET Core project templates, therefore it is automatically included and enabled in the PowerServer Web APIs, and there is no need to install or configure Kestrel.

In the *launchSettings.json* file of the **ServerAPIs** project of the PowerServer C# solution, the *commandName* key has the value **Project** which indicates that the Kestrel web server will be launched; and the *applicationUrl* key specifies the host name and port number for Kestrel.

For detailed description of the settings in *launchSettings.json*, see https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-3.1#development-and-launchsettingsjson.

```
"ServerAPIs": {
  "commandName": "Project",
  "launchBrowser": true,
  "launchUrl": "swagger",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  },
  "applicationUrl": "http://0.0.0.0:6000/"
}
```

4.3 Running Web APIs on Kestrel

As Kestrel is by default included and enabled in the PowerServer Web APIs, when the PowerServer Web APIs runs, it automatically runs on Kestrel.

You can run PowerServer Web APIs on Kestrel using the following methods:

- (In the development environment) Launch the PowerServer Web APIs from the SnapDevelop IDE (by clicking the Run button in the PowerServer C# solution).
- (In the development environment) Execute the "dotnet run --project PowerServer_salesdemo\ServerAPIs\ServerAPIs.csproj" command,
- (In the production environment) Publish the PowerServer Web APIs from the SnapDevelop IDE to a folder, copy the folder to the production server, and then run the app.

  The Web APIs will be compiled as an ASP.NET Core app and all files (such as configuration files, assembly files, dependencies, .NET runtime etc.) required to run the

app will be copied to the publish folder. See this section for step-by-step instructions on how to publish the Web APIs to a folder.

After that, copy the folder to the server and then run the app:

```
dotnet <app_assembly>.dll
```

The PowerServer Web APIs can be run as a service just like any other ASP.NET Core app, so that it can be automatically run without needing you to log into the PC to start it.

To run the PowerServer Web APIs as a service in Windows, refer to https://docs.microsoft.com/aspnet/core/host-and-deploy/windows-service?view=aspnetcore-3.1&tabs=visual-studio.

To run the PowerServer Web APIs as a service in Linux, refer to https://docs.microsoft.com/aspnet/core/host-and-deploy/linux-nginx?view=aspnetcore-3.1#create-the-service-file.

4.4 Using a reverse proxy server

4.4.1 Configuring Apache reverse proxy server (Windows)

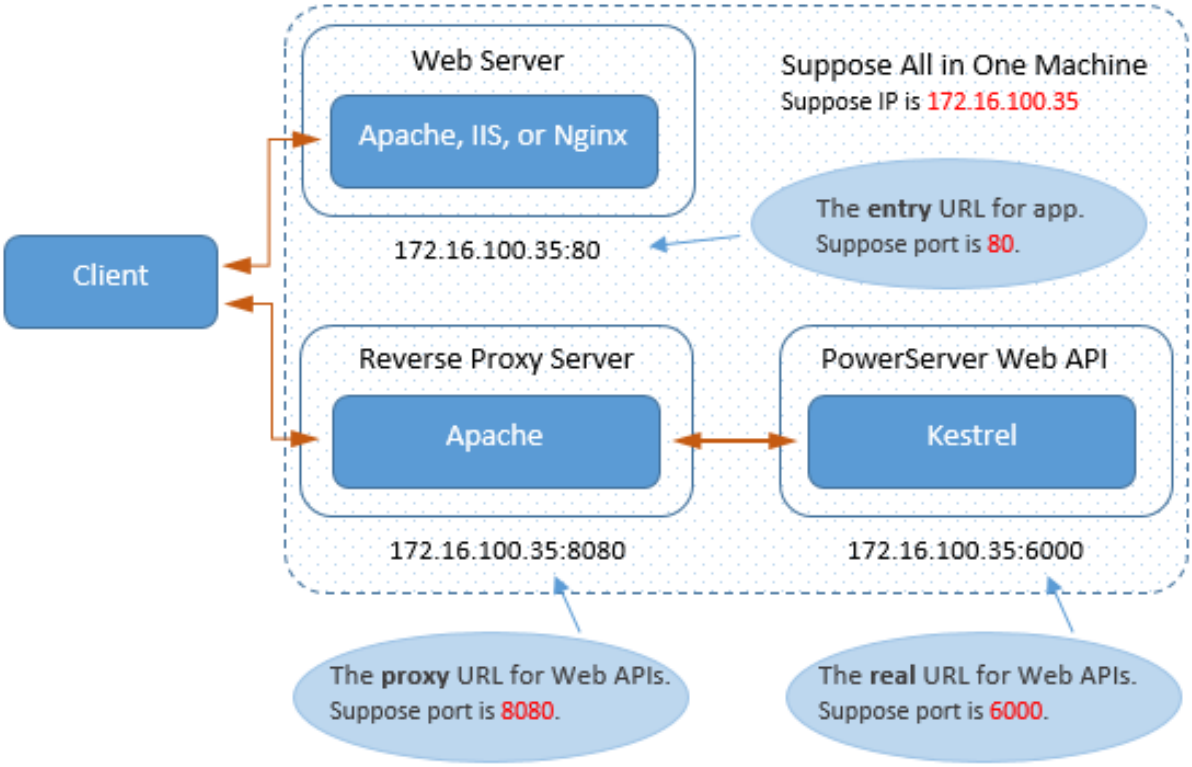
4.4.1.1 Preparations

In this tutorial, we will learn how to set up Apache on Windows and use it as the reverse proxy server to redirect requests to the PowerServer Web APIs running on the Kestrel server.

The Apache reverse proxy server can be set up on the same or different server from the PowerServer Web APIs and Kestrel. In this tutorial, the same server will be used.

In this tutorial, we will configure and use the following server environment and URLs. Be careful to use the correct port number and make sure the port is not occupied by any other program.

Figure 4.3:



Step 1: Set up the server with the following OS and software (install the software in the order listed).

- Windows Server 2019 (64-bit)
- Visual C++ Redistributable
- Apache HTTP Server 2.4.47

  The section Installing Apache HTTP Server has detailed installation instructions.

Step 2: Make sure the server can connect to the NuGet site: https://www.nuget.org (for installing PowerServer NuGet packages) and the following Appeon sites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) (for validating the PowerServer license).

> **Note**
>
> If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Step 3: Configure Windows Defender Firewall on the server to allow the port number (80 and 8080 in this tutorial or any port number you choose). The section "Configuring Windows Defender Firewall" has detailed instructions.

4.4.1.2 Configuring Apache

This section is to configure Apache as a reverse proxy server in a Windows machine.

Step 1: Go to the ..\Apache24\conf folder and open the httpd.conf file in a text editor.

Step 2: Add the following scripts to the end of the httpd.conf file.

This is to configure Apache as a reverse proxy server which will redirect requests made to the URL: https://172.16.100.35:8080/ to the PowerServer Web APIs running on Kestrel at https://172.16.100.35:6000/.

```
# Listen on port 8080 or any port you choose. Make sure it is not used by any other
 program.
<VirtualHost *:8080>
 ProxyPreserveHost On
 ErrorLog logs\ps-error.log
 CustomLog logs\ps-access.log common
 # Pass all requests received at the root https://172.16.100.35/8080 to
 https://172.16.100.35:6000/ (PowerServer Web APIs running on Kestrel server) and
 in reverse.
 ProxyPass / https://172.16.100.35:6000/
 ProxyPassReverse / https://172.16.100.35:6000/
</VirtualHost>
```

Step 3: Locate the following line in the httpd.conf file and specify the port number: 80 (or any port you choose) is used to access the static Web files on the Apache HTTP server, 8080 is used to access Web APIs (according to the reverse proxy setting in step 2, requests made to 8080 will be forwarded to 6000.)

Change

```
Listen 80
```

To

```
Listen 80
Listen 8080
```

Tip: In Windows, you can execute the command "netstat -ano | findstr 8080" to check if the port number is occupied by any other program.

Step 4: Make sure the following lines are **NOT** commented out in the httpd.conf file.

```
LoadModule negotiation_module modules/mod_negotiation.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
```

Step 5: Check if any syntax error in httpd.conf.

```
cd C:\Apache24\bin
httpd -t
```

Step 6: Restart Apache for the changes to take effect.

```
httpd -k restart
```

Step 7: View the ..\Apache24\logs\error.log file to make sure Apache is started successfully.

```
[Wed Jun 02 00:46:00.547040 2021] [mpm_winnt:notice] [pid 1556:tid 696] AH00455:
 Apache/2.4.47 (Win64) configured -- resuming normal operations
[Wed Jun 02 00:46:00.547040 2021] [mpm_winnt:notice] [pid 1556:tid 696] AH00456:
 Apache Lounge VS16 Server built: Apr 24 2021 11:08:47
[Wed Jun 02 00:46:00.547040 2021] [core:notice] [pid 1556:tid 696] AH00094: Command
 line: 'c:\\apache24\\bin\\httpd.exe -d C:/Apache24'
```

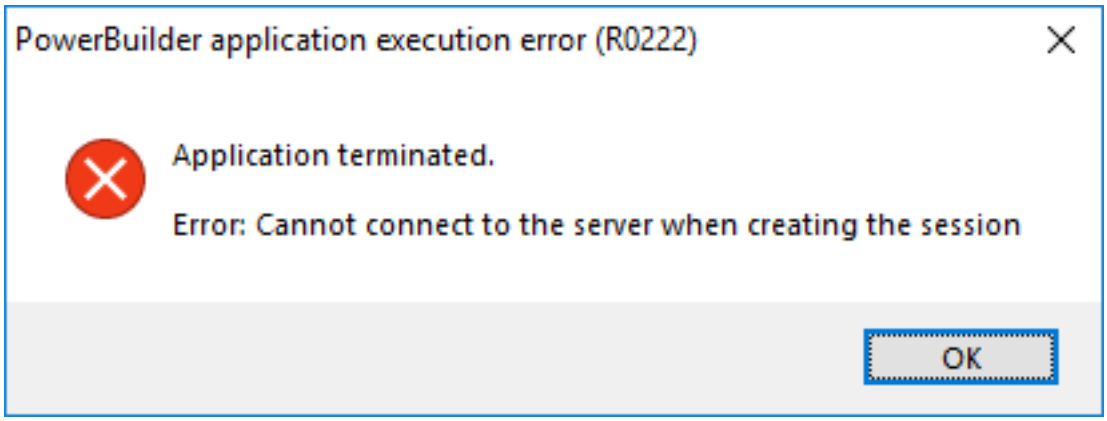
```
[Wed Jun 02 00:46:00.547040 2021] [mpm_winnt:notice] [pid 1556:tid 696]
 AH00418: Parent: Created child process 4860
[Wed Jun 02 00:46:01.143540 2021] [mpm_winnt:notice] [pid 4860:tid 728]
 AH00354: Child: Starting 64 worker threads.
```

Step 8: If you have set up a firewall on the server, configure the firewall to allow port 8080 (by following instructions in "Configuring Windows Defender Firewall").

Note

If the firewall blocks the port number, you will have the following error when running the application.

Figure 4.4:



4.4.1.3 Modifying and re-deploying the PowerServer project

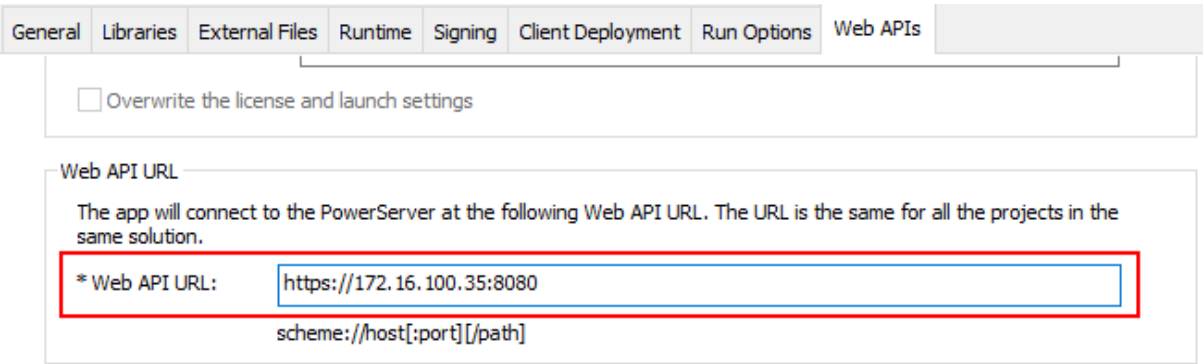
The following modifications are made to the PowerServer project created in the Quick Start guide. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

Step 1: Modify the Web API URL to point to the Apache reverse proxy server.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the Apache reverse proxy server, for example, https://172.16.100.35:8080 in this tutorial. It is highly recommended that you specify an HTTPS URL for the production environment.

All requests for PowerServer Web APIs will be first made to https://172.16.100.35:8080 and then redirected by the Apache reverse proxy server to the PowerServer Web APIs running on Kestrel server (for example, https://172.16.100.35:6000).

Figure 4.5:



Step 2: Select a Web server for deploying the app files.

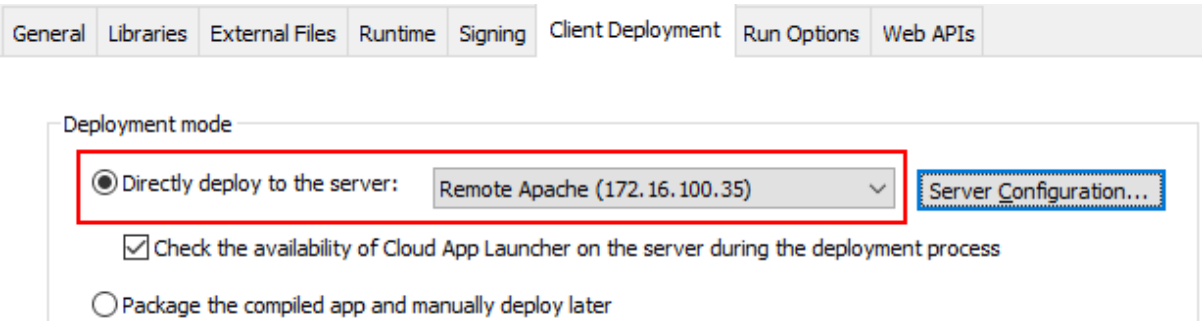
On the **Client Deployment** tab of the PowerServer project painter, select a local or remote Web server (IIS, Apache, Nginx, etc.) you have configured properly.

The Web server and the Apache reverse proxy server can reside in the same or different machine. If the Web server is an Apache HTTP server, it can be the same or different server instance with the Apache reverse proxy server.

In this tutorial, we use the same Apache server instance as the Apache HTTP server and the reverse proxy server.

- If you choose the "Directly deploy to the server" option, make sure you have configured the FTP settings properly for the server. See Setting up Apache on Windows > Installing FTP server for detailed instructions.
- If you choose the "Package the compiled app and manually deploy later" option, follow the instructions in Packaging and copying the client app.

Figure 4.6:



Step 3: Save the PowerServer project settings and then build and deploy the PowerServer project for the changes to take effect.

4.4.1.4 Starting Web APIs (in development environment)

In this tutorial, we will directly run the PowerServer Web APIs in the development environment, by using either of the following methods:

- Execute the "dotnet run --project PowerServer19\ServerAPIs\ServerAPIs.csproj" command, or
- Open the PowerServer C# solution in the SnapDevelop IDE and then click the **Run** button.

PowerServer Web APIs is running as a standalone console application on its own internal Kestrel web server.

Make sure the PowerServer Web APIs is running on the correct IP address and port number. For example, https://172.16.100.35:6000/ in this tutorial. You may modify the port number in the *launchSettings.json* file of the **ServerAPIs** project of the PowerServer C# solution when running in the development environment.

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Figure 4.7:

```
C:\Users\appeon\source\repos\PowerServer19\ServerAPIs\bin\Debug\netcoreapp3.1\ServerAPIs.exe
      Sending HTTP request POST https://apipsoa.appeon.com/connect/token
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[101]
      Received HTTP response headers after 253.5188ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[101]
      End processing HTTP request after 253.9184ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[100]
      Start processing HTTP request POST https://apips.appeon.com/api/v1/license/startup
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[100]
      Sending HTTP request POST https://apips.appeon.com/api/v1/license/startup
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[101]
      Received HTTP response headers after 303.4155ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[101]
      End processing HTTP request after 303.9532ms - 200
info: PowerServer[0]
      * * * * * PowerServer License * * * * * *
      *  LicenseKey: DSLD-IGFG-AII2-JJ2I-IE36 *
      * LicenseType: Subscription            *
      *     Session: 10                      *
      *      Expire: 2021-07-01              *
      * * * * * * * * * * * * * * * * * * * *

info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://0.0.0.0:6000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\appeon\source\repos\PowerServer19\ServerAPIs
```

When you run the application (https://172.16.100.35:80/pssales in this tutorial), you will be able to see from the console that the requests are going through successfully and the requests are originally made to the Apache proxy server (https://172.16.100.35/8080 in this tutorial).

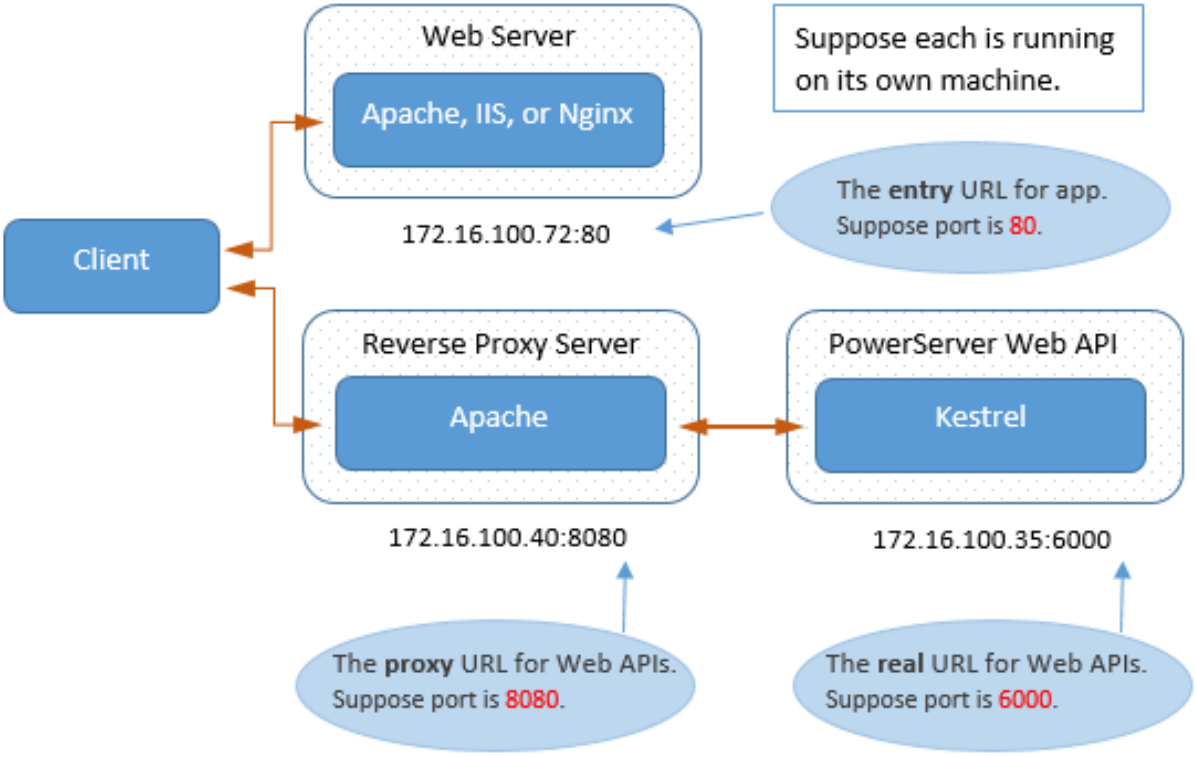
4.4.2 Configuring Apache reverse proxy server (Linux)

4.4.2.1 Preparations

In this tutorial, we will learn how to set up Apache on Linux and use it as the reverse proxy server to redirect requests to the PowerServer Web APIs running on the Kestrel server.

In this tutorial, we will configure and use the following server environment and URLs. Be careful to use the correct port number and make sure the port is not occupied by any other program.

Figure 4.8:



Step 1: Set up the reverse proxy server with the following OS and software (install the software in the order listed).

- CentOS 8 (64-bit)
- Apache HTTP Server

  The section Installing Apache HTTP Server has detailed installation instructions.

Step 2: Configure the CentOS user account: you can either use the root account or create a new account with administrative privileges.

Step 3: Set up a firewall on the server and make sure the firewall allows the port (80 and 8080 in this tutorial or any port number you choose) to go through.

Step 4: Make sure the server can connect to Internet during the installation of Apache HTTP Server.

4.4.2.2 Configuring Apache

This section is to configure Apache as a reverse proxy server in a Linux machine.

Step 1: Go to the /etc/httpd/conf folder and open the httpd.conf file in a text editor.

Step 2: Add the following scripts to the end of the httpd.conf file.

This is to configure Apache as a reverse proxy server which will redirect requests made to the URL: https://172.16.100.40:8080/ to the PowerServer Web APIs running on Kestrel at https://172.16.100.35:6000/.

```
# Listen on port 8080 or any port you choose. Make sure it is not used by any other
 program.
<VirtualHost *:8080>
```

```
 ProxyPreserveHost On
 # Pass all requests received at the root https://172.16.100.40/8080 to
 https://172.16.100.35:6000/ (PowerServer Web APIs running on Kestrel server) and
 in reverse.
 ProxyPass / https://172.16.100.35:6000/
 ProxyPassReverse / https://172.16.100.35:6000/
</VirtualHost>
```

Step 3: Locate the following line in the httpd.conf file and specify the port number: 80 (or any port you choose) is used to access the static Web files on the Apache HTTP server, 8080 is used to access Web APIs (according to the reverse proxy setting in step 2, requests made to 8080 will be forwarded to 6000.)

Change

```
Listen 80
```

To

```
Listen 80
Listen 8080
```

Tip: In CentOS, you can execute the command "netstat -anp | grep 8080" to check if the port number is occupied by any other program.

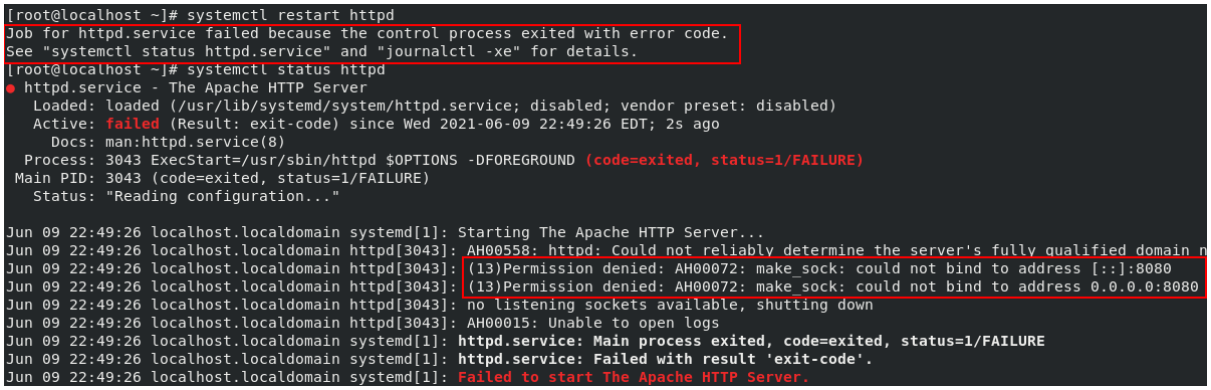
Step 4: Run the following command to add port 8080 to "http_port_t":

```
$ sudo semanage port -a -t http_port_t -p tcp 8080
```

Note

If the port is not properly added, you may see the following error when you start and check the status of Apache:

Figure 4.9:



Step 5: If you have set up a firewall on the server, run the following command to permanently enable port 8080:

```
$ sudo firewall-cmd --permanent --zone=public --add-port=8080/tcp
```

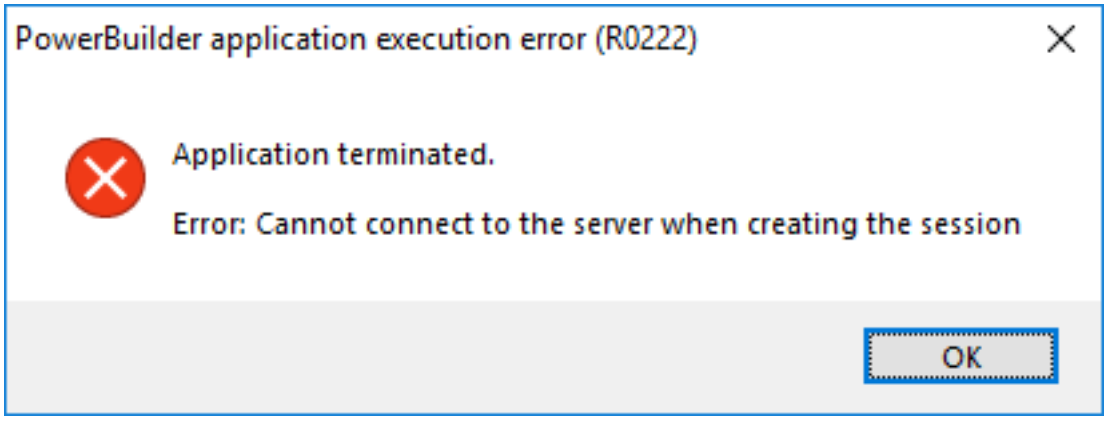
and the following command to reload the firewall service:

```
$ sudo firewall-cmd --reload
```

Note

If the firewall blocks the port number, you may have the following error when running the application.

Figure 4.10:



Step 6: Check if any syntax error in httpd.conf, and then restart Apache for the changes to take effect.

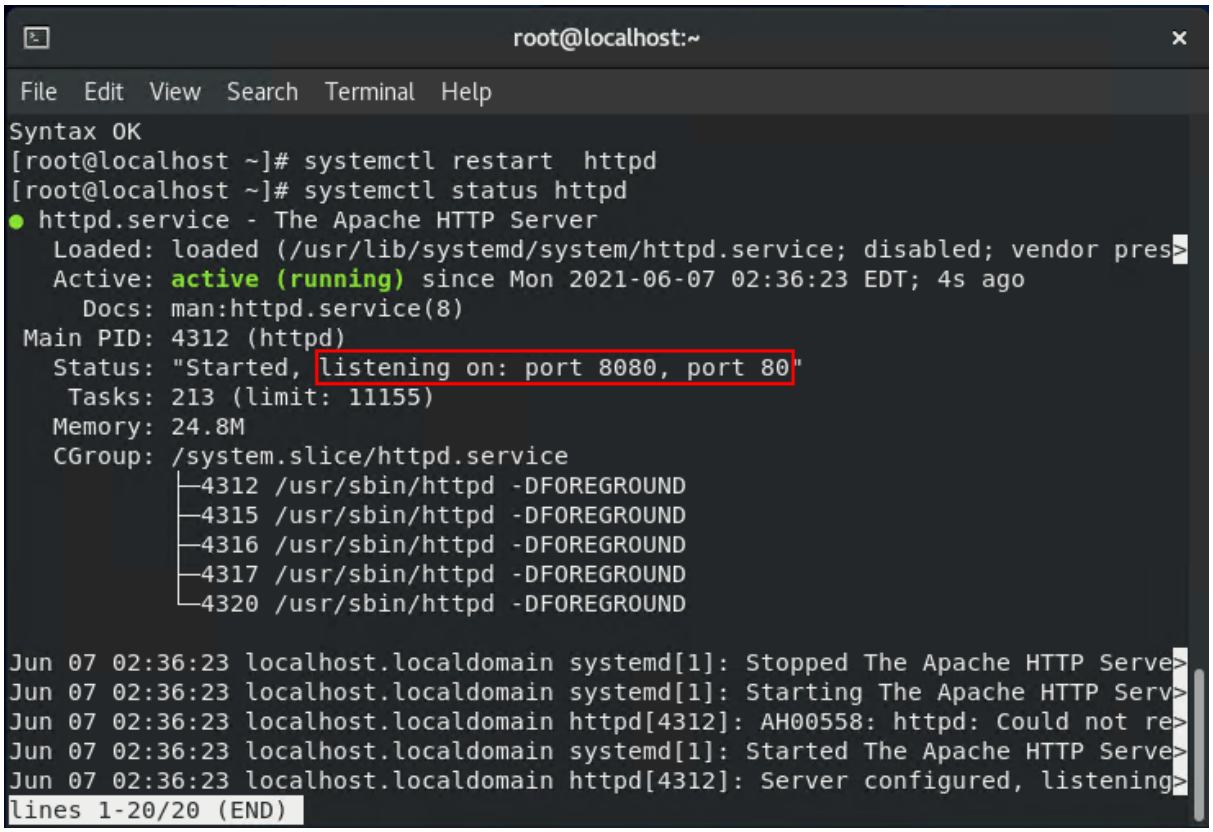
```
$ sudo apachectl configtest
```

```
$ sudo systemctl restart httpd
```

Step 7: Verify that Apache is running.

```
$ sudo systemctl status httpd
```

Figure 4.11:



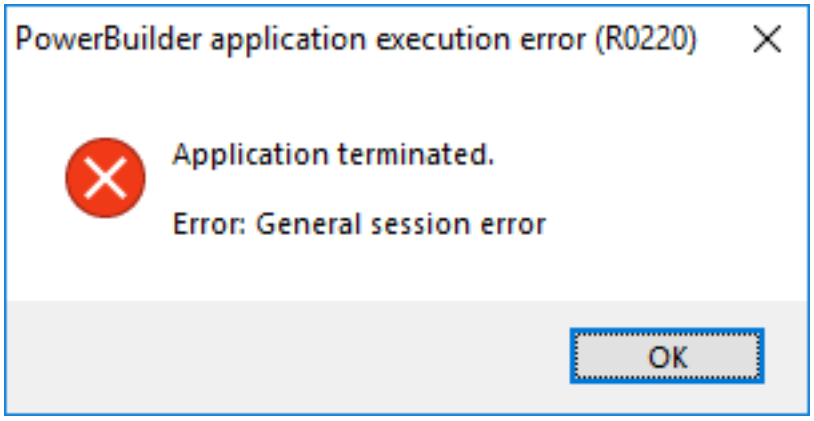
Step 8: Run the following command to allow Apache to make outbound connections.

```
$ sudo /usr/sbin/setsebool -P httpd_can_network_connect 1
```

Note

If Apache is not allowed to make outbound connections, you may encounter the following error when running the application,

Figure 4.12:



and may have the following errors in the \var\log\httpd\error_log.log file.

```
[Tue Jun 08 05:21:42.408866 2021] [proxy:error] [pid 4025:tid
 140605678085888] (13)Permission denied: AH00957: HTTP: attempt to connect
 to 172.16.100.35:6000 (172.16.100.35) failed
[Tue Jun 08 05:21:42.408952 2021] [proxy_http:error] [pid 4025:tid
 140605678085888] [client 172.16.100.35:56187] AH01114: HTTP: failed to make
 connection to backend: 172.16.100.35
```

4.4.2.3 Modifying and re-deploying the PowerServer project

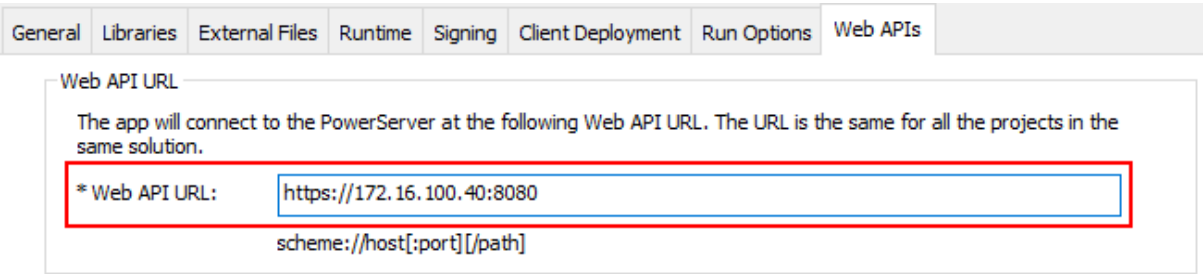
The following modifications are made to the PowerServer project created in the Quick Start guide. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

Step 1: Modify the Web API URL to point to the Apache reverse proxy server.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the Apache reverse proxy server, for example, https://172.16.100.40:8080. It is highly recommended that you specify an HTTPS URL for the production environment.

All requests for PowerServer Web APIs will be first made to https://172.16.100.40:8080 and then redirected by the Apache reverse proxy server to the PowerServer Web APIs running on Kestrel server (for example, https://172.16.100.35:6000).

Figure 4.13:



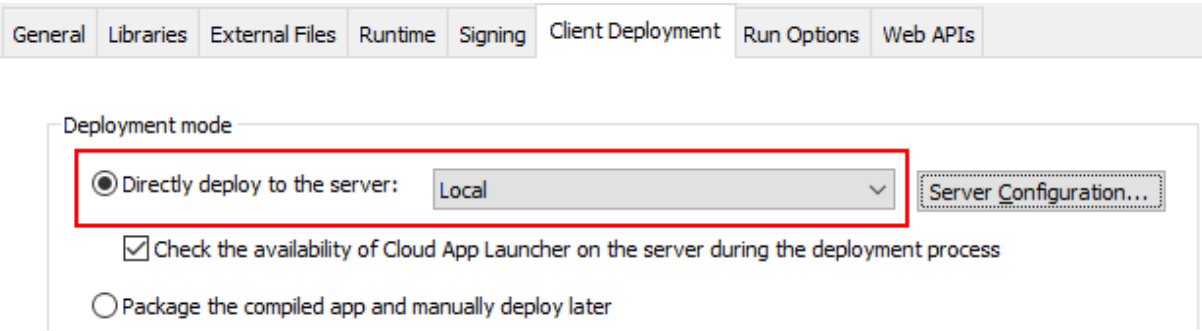
Step 2: Select a Web server for deploying the app files.

On the **Client Deployment** tab of the PowerServer project painter, select a local or remote Web server (IIS, Apache, Nginx, etc.) you have configured properly.

The Web server and the Apache reverse proxy server can reside in the same or different machine. If the Web server is an Apache HTTP server, it can be the same or different server instance with the Apache reverse proxy server. If you want to deploy the app files to the Apache HTTP server which uses the same server instance as the Apache reverse proxy server on a Linux machine, you can choose "Package the compiled app and manually deploy later" (see Packaging and copying the client app for detailed instructions).

In this tutorial, we choose to deploy the app files to a local IIS Web server.

Figure 4.14:



Step 3: Save the PowerServer project settings and then build and deploy the PowerServer project for the changes to take effect.

4.4.2.4 Starting Web APIs (in development environment)

In this tutorial, we will directly run the PowerServer Web APIs in the development environment, by using either of the following methods:

- Execute the "dotnet run --project PowerServer19\ServerAPIs\ServerAPIs.csproj" command, or
- Open the PowerServer C# solution in the SnapDevelop IDE and then click the **Run** button.

PowerServer Web APIs is running as a standalone console application on its own internal Kestrel web server.

Make sure the PowerServer Web APIs is running on the correct IP address and port number. For example, https://172.16.100.35:6000/ in this tutorial. You may modify the port number in the *launchSettings.json* file of the **ServerAPIs** project of the PowerServer C# solution when running in the development environment.

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Figure 4.15:

```
C:\Users\appeon\source\repos\PowerServer19\ServerAPIs\bin\Debug\netcoreapp3.1\ServerAPIs.exe
      Sending HTTP request POST https://apipsoa.appeon.com/connect/token
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[101]
      Received HTTP response headers after 253.5188ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[101]
      End processing HTTP request after 253.9184ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[100]
      Start processing HTTP request POST https://apips.appeon.com/api/v1/license/startup
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[100]
      Sending HTTP request POST https://apips.appeon.com/api/v1/license/startup
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[101]
      Received HTTP response headers after 303.4155ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[101]
      End processing HTTP request after 303.9532ms - 200
info: PowerServer[0]
      * * * * * PowerServer License * * * * * *
      *  LicenseKey: DSLD-IGFG-AII2-JJ2I-IE36 *
      * LicenseType: Subscription            *
      *     Session: 10                      *
      *      Expire: 2021-07-01              *
      * * * * * * * * * * * * * * * * * * * *

info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://0.0.0.0:6000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\appeon\source\repos\PowerServer19\ServerAPIs
```

When you run the application (https://172.16.100.72:80/pssales in this tutorial), you will be able to see from the console that the requests are going through successfully and the requests are originally made to the Apache proxy server (https://172.16.100.40/8080 in this tutorial).

Figure 4.16:

```
C:\Users\appeon\source\repos\PowerServer19\ServerAPIs\bin\Debug\netcoreapp3.1\ServerAPIs.exe
      Now listening on: http://0.0.0.0:6000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\appeon\source\repos\PowerServer19\ServerAPIs
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 POST http://172.16.100.40:8080/api/ServerApi/CreateSession application/json;charset=UTF-
8 507
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'PowerServer.Api.ServerApiController.CreateSessionAsync (PowerServer.Api)'
info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[3]
      Route matched with {action = "CreateSession", controller = "ServerApi"}. Executing controller action with signatur
e System.Threading.Tasks.Task`1[Microsoft.AspNetCore.Mvc.ActionResult] CreateSessionAsync(PowerServer.Core.RequestSessio
n) on controller PowerServer.Api.ServerApiController (PowerServer.Api).
info: Microsoft.AspNetCore.Mvc.NewtonsoftJson.NewtonsoftJsonResultExecutor[1]
      Executing JsonResult, writing value of type '□?'.
info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[2]
      Executed action PowerServer.Api.ServerApiController.CreateSessionAsync (PowerServer.Api) in 217.1775ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'PowerServer.Api.ServerApiController.CreateSessionAsync (PowerServer.Api)'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 332.6014ms 200 application/json; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 POST http://172.16.100.40:8080/api/ServerApi/VerificationFiles application/json;charset=
UTF-8 0
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'PowerServer.Api.ServerApiController.VerificationFiles (PowerServer.Api)'
info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[3]
```

4.4.3 Configuring Nginx reverse proxy server (Windows)

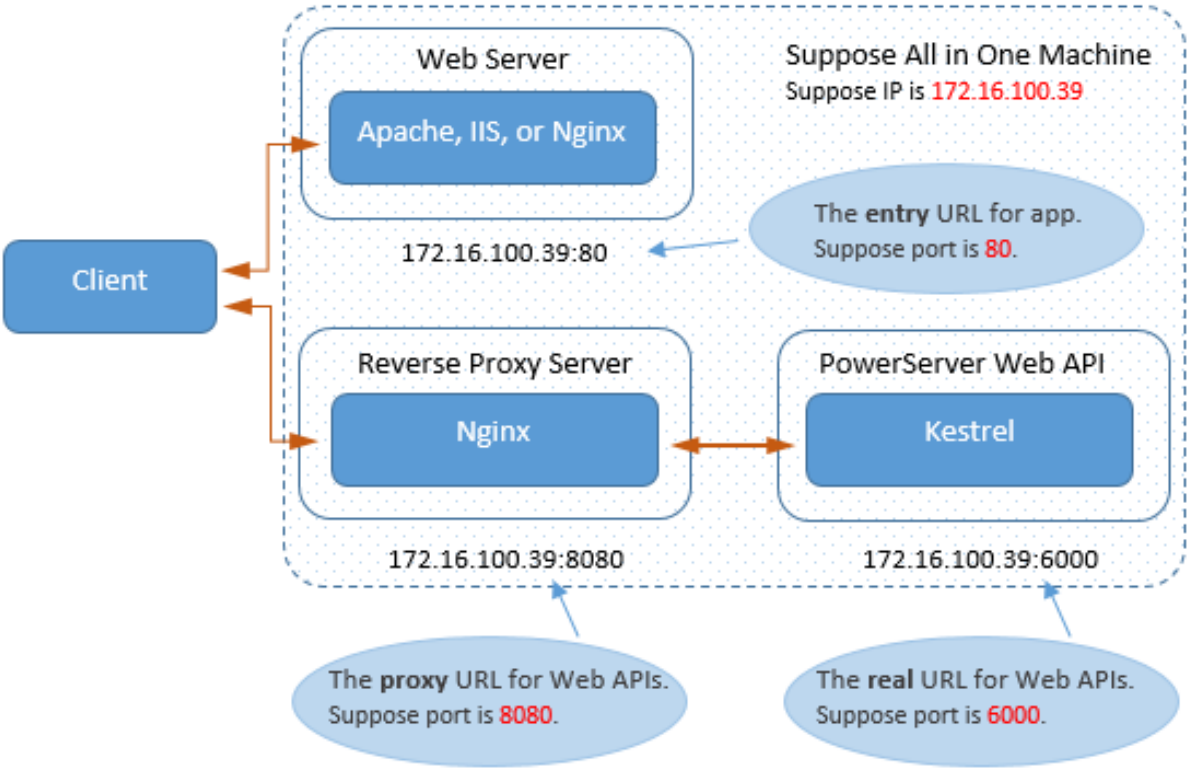
4.4.3.1 Preparations

In this tutorial, we will learn how to set up Nginx on Windows and use it as the reverse proxy server to redirect requests to the PowerServer Web APIs running on the Kestrel server.

The Nginx reverse proxy server can be set up on the same or different server from the PowerServer Web APIs and Kestrel. In this tutorial, the same server will be used.

In this tutorial, we will configure and use the following server environment and URLs. Be careful to use the correct port number and make sure the port is not occupied by any other program.

Figure 4.17:



Step 1: Set up the server with the following OS and software (install the software in the order listed).

- Windows Server 2019 (64-bit)
- Nginx 1.19.10

  The section Installing Nginx has detailed installation instructions.

Step 2: Make sure the server can connect to the NuGet site: https://www.nuget.org (for installing PowerServer NuGet packages) and the following Appeon sites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) (for validating the PowerServer license).

Note

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Step 3: Configure Windows Defender Firewall on the server to allow the port number (80 and 8080 in this tutorial or any port number you choose). The section "Configuring Windows Defender Firewall" has detailed instructions.

4.4.3.2 Configuring Nginx

This section is to configure Nginx as a reverse proxy server in a Windows machine.

Step 1: Go to the ..\nginx-1.19.10\conf folder and open the nginx.conf file in a text editor.

Step 2: Locate the "server" block and add another "server" block as shown below.

This is to configure Nginx as a reverse proxy server which will redirect requests made to the URL: https://172.16.100.39:8080/ to the PowerServer Web APIs running on Kestrel at https://172.16.100.35:6000/.

```
    server {
        listen  8080;
        location / {
           proxy_set_header Host $http_host;
           proxy_pass  https://172.16.100.35:6000;
        }
    }
```

Figure 4.18:

```
        # concurs with nginx's one
        #
        #location ~ /\.ht {
        #    deny  all;
        #}
    }

    server {
        listen  8080;
        location / {
           proxy_set_header Host $http_host;
           proxy_pass  https://172.16.100.35:6000;
        }
    }

    # another virtual host using mix of IP-, name-, and port-based configuration
    #
    #server {
    #    listen       8000;
    #    listen       somename:8080;
    #    server_name  somename  alias  another.alias;

    #    location / {
```

Tip: In Windows, you can execute the command "netstat -ano | findstr 8080" to check if the port number is occupied by any other program.

Step 3: Check if any syntax error in the Nginx configuration file, and then restart Nginx for the changes to take effect.

```
nginx -t
```

```
nginx -s reload
```

Step 4: Verify that the Nginx processes are running.

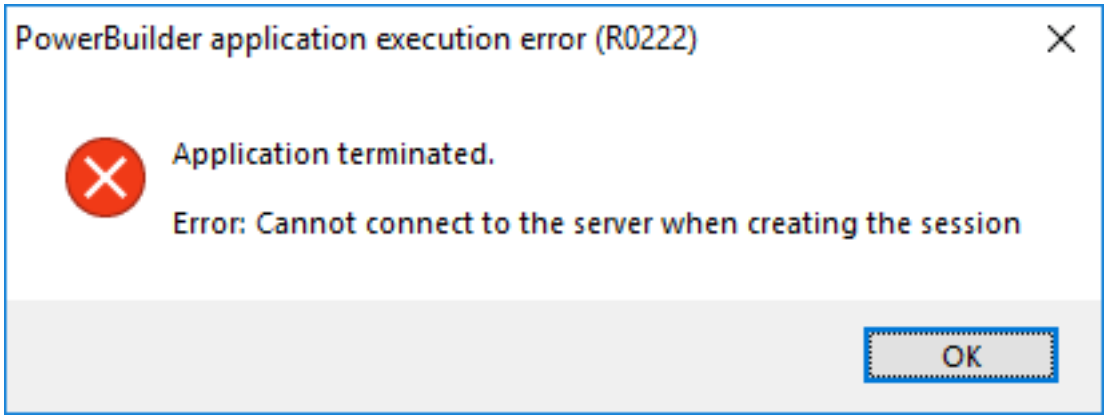
```
tasklist /fi "imagename eq nginx.exe"
```

Step 5: If you have set up a firewall on the server, configure the firewall to allow port 8080 (by following instructions in "Configuring Windows Defender Firewall").

Note

If the firewall blocks the port number, you will have the following error when running the application.

Figure 4.19:



4.4.3.3 Modifying and re-deploying the PowerServer project

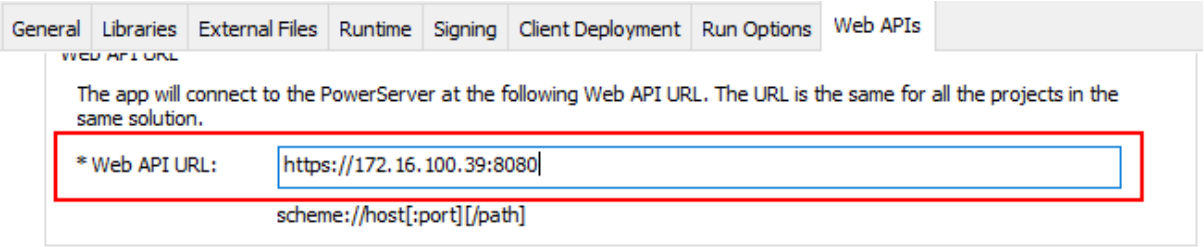
The following modifications are made to the PowerServer project created in the Quick Start guide. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

Step 1: Modify the Web API URL to point to the Nginx reverse proxy server.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the Nginx reverse proxy server, for example, https://172.16.100.39:8080 in this tutorial. It is highly recommended that you specify an HTTPS URL for the production environment.

All requests for the PowerServer Web APIs will be first made to https://172.16.100.39:8080 and then redirected by the Nginx reverse proxy server to the PowerServer Web APIs running on Kestrel server (for example, https://172.16.100.35:6000).

Figure 4.20:



Step 2: Select a Web server for deploying the app files.

On the **Client Deployment** tab of the PowerServer project painter, select a local or remote Web server (IIS, Apache, Nginx, etc.) you have configured properly.

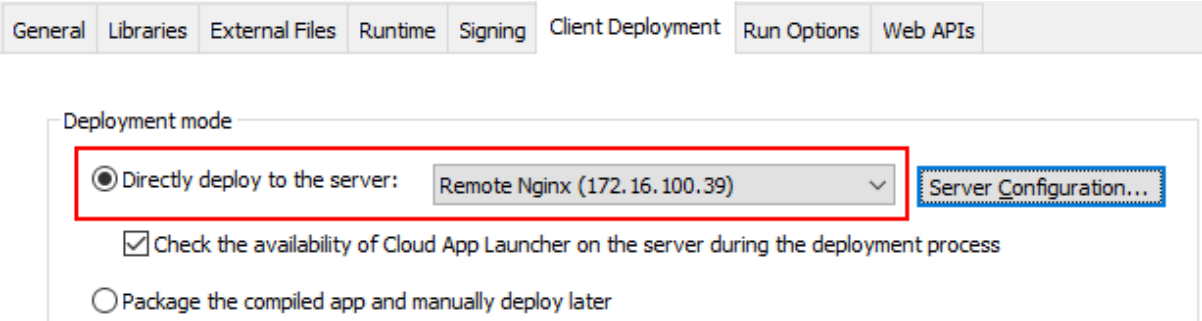
The Web server and the Nginx reverse proxy server can reside in the same or different machine. If the Web server is an Nginx HTTP server, it can be the same or different server instance with the Nginx reverse proxy server.

In this tutorial, we use the same Nginx server instance as the Nginx HTTP server and the reverse proxy server.

- If you choose the "Directly deploy to the server" option, make sure you have configured the FTP settings properly for the server. See Setting up Nginx on Windows > Installing FTP server for detailed instructions.

- If you choose the "Package the compiled app and manually deploy later" option, follow the instructions in Packaging and copying the client app.

Figure 4.21:



Step 3: Save the PowerServer project settings and then build and deploy the PowerServer project for the changes to take effect.

4.4.3.4 Starting Web APIs (in development environment)

In this tutorial, we will directly run the PowerServer Web APIs in the development environment, by using either of the following methods:

- Execute the "dotnet run --project PowerServer19\ServerAPIs\ServerAPIs.csproj" command, or

- Open the PowerServer C# solution in the SnapDevelop IDE and then click the **Run** button.

PowerServer Web APIs is running as a standalone console application on its own internal Kestrel web server.

Make sure the PowerServer Web APIs is running on the correct IP address and port number. For example, https://172.16.100.35:6000/ in this tutorial. You may modify the port number in the *launchSettings.json* file of the **ServerAPIs** project of the PowerServer C# solution when running in the development environment.

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Figure 4.22:

```
C:\Users\appeon\source\repos\PowerServer19\ServerAPIs\bin\Debug\netcoreapp3.1\ServerAPIs.exe
      Sending HTTP request POST https://apipsoa.appeon.com/connect/token
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[101]
      Received HTTP response headers after 253.5188ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[101]
      End processing HTTP request after 253.9184ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[100]
      Start processing HTTP request POST https://apips.appeon.com/api/v1/license/startup
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[100]
      Sending HTTP request POST https://apips.appeon.com/api/v1/license/startup
info: System.Net.Http.HttpClient.PowerServer.ClientHandler[101]
      Received HTTP response headers after 303.4155ms - 200
info: System.Net.Http.HttpClient.PowerServer.LogicalHandler[101]
      End processing HTTP request after 303.9532ms - 200
info: PowerServer[0]
      * * * * * PowerServer License * * * * * *
      *  LicenseKey: DSLD-IGFG-AII2-JJ2I-IE36 *
      * LicenseType: Subscription            *
      *     Session: 10                      *
      *      Expire: 2021-07-01              *
      * * * * * * * * * * * * * * * * * * * *

info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://0.0.0.0:6000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\appeon\source\repos\PowerServer19\ServerAPIs
```

When you run the application (https://172.16.100.39:80/pssales in this tutorial), you will be able to see from the console that the requests are going through successfully and the requests are originally made to the Nginx proxy server (https://172.16.100.39/8080 in this tutorial).

Figure 4.23:

```
C:\Users\appeon\source\repos\PowerServer19\ServerAPIs\bin\Debug\netcoreapp3.1\ServerAPIs.exe
      Now listening on: http://0.0.0.0:6000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\appeon\source\repos\PowerServer19\ServerAPIs
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.0 POST http://172.16.100.39:8080/api/ServerApi/CreateSession application/json;charset=UTF-
8 507
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'PowerServer.Api.ServerApiController.CreateSessionAsync (PowerServer.Api)'
info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[3]
      Route matched with {action = "CreateSession", controller = "ServerApi"}. Executing controller action with signatur
e System.Threading.Tasks.Task`1[Microsoft.AspNetCore.Mvc.ActionResult] CreateSessionAsync(PowerServer.Core.RequestSessio
n) on controller PowerServer.Api.ServerApiController (PowerServer.Api).
info: Microsoft.AspNetCore.Mvc.NewtonsoftJson.NewtonsoftJsonResultExecutor[1]
      Executing JsonResult, writing value of type '□?'.
info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[2]
      Executed action PowerServer.Api.ServerApiController.CreateSessionAsync (PowerServer.Api) in 212.0545ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'PowerServer.Api.ServerApiController.CreateSessionAsync (PowerServer.Api)'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 322.4584ms 200 application/json; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.0 POST http://172.16.100.39:8080/api/ServerApi/VerificationFiles application/json;charset=
UTF-8 0
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'PowerServer.Api.ServerApiController.VerificationFiles (PowerServer.Api)'
info: Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker[3]
```

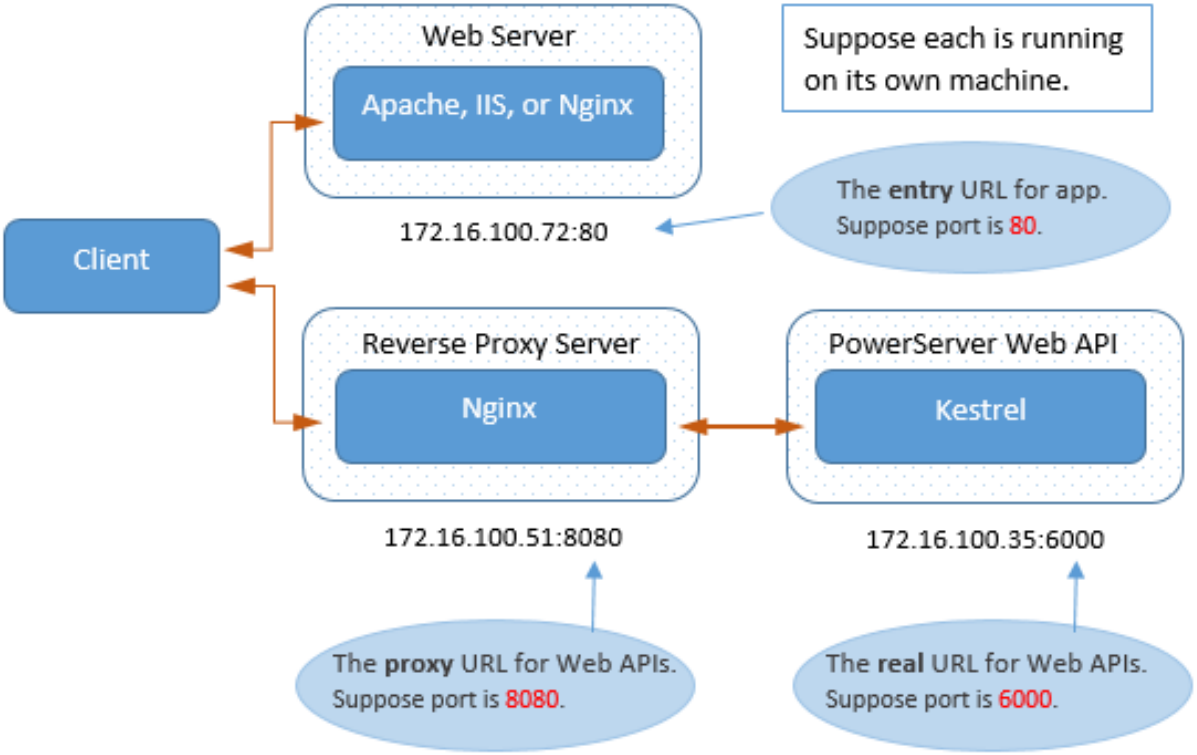
4.4.4 Configuring Nginx reverse proxy server (Linux)

4.4.4.1 Preparations

In this tutorial, we will learn how to set up Nginx on Linux and use it as the reverse proxy server to redirect requests to the PowerServer Web APIs running on the Kestrel server.

In this tutorial, we will configure and use the following server environment and URLs. Be careful to use the correct port number and make sure the port is not occupied by any other program.

Figure 4.24:



Step 1: Set up the reverse proxy server with the following OS and software (install the software in the order listed).

- CentOS 8 (64-bit)
- Nginx

  The section Installing Nginx has detailed installation instructions.

Step 2: Configure the CentOS user account: you can either use the root account or create a new account with administrative privileges.

Step 3: Set up a firewall on the server and make sure the firewall allows the port (80 and 8080 in this tutorial or any port number you choose) to go through.

Step 4: Make sure the server can connect to Internet during the installation of Nginx.

4.4.4.2 Configuring Nginx

This section is to configure Nginx as a reverse proxy server in a Linux machine.

Step 1: Go to the /etc/nginx/ folder and open the nginx.conf file in a text editor.

Step 2: Locate the "server" block and add another "server" block as shown below.

This is to configure Nginx as a reverse proxy server which will redirect requests made to the URL: https://172.16.100.51:8080/ to the PowerServer Web APIs running on Kestrel at https://172.16.100.35:6000/.

```
    server {
        listen  8080;
        location / {
           proxy_set_header Host $http_host;
           proxy_pass  https://172.16.100.35:6000;
        }
    }
```

Tip: In CentOS, you can execute the command "netstat -anp | grep 8080" to check if the port number is occupied by any other program.

Step 3: Run the following command to add port 8080 to "http_port_t":

```
$ sudo semanage port -a -t http_port_t -p tcp 8080
```

Note

If the port is not properly added, you may see the following error when Nginx starts:

Figure 4.25:

```
[root@localhost ~]# systemctl restart nginx.service
Job for nginx.service failed because the control process exited with error code.
See "systemctl status nginx.service" and "journalctl -xe" for details.
```

and may have the following error in the \var\log\nginx\error.log file.

```
2021/06/09 05:26:29 [emerg] 4107#0: bind() to 0.0.0.0:8080 failed (13:
 Permission denied)
```

Step 4: If you have set up a firewall on the server, run the following command to permanently enable port 8080:

```
$ sudo firewall-cmd --permanent --zone=public --add-port=8080/tcp
```

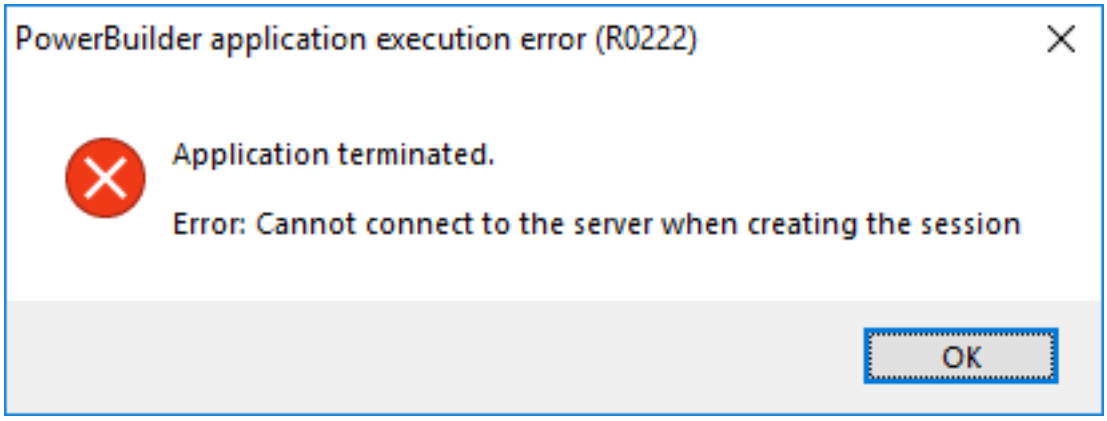
and the following command to reload the firewall service:

```
$ sudo firewall-cmd --reload
```

Note

If the firewall blocks the port number, you will have the following error when running the application.

Figure 4.26:



Step 5: Check if any syntax error in the Nginx configuration file, and then restart Nginx for the changes to take effect.

```
$ sudo nginx -t
```

```
$ sudo systemctl restart nginx
```

Step 6: Verify that Nginx is running.

```
$ sudo systemctl status nginx
```

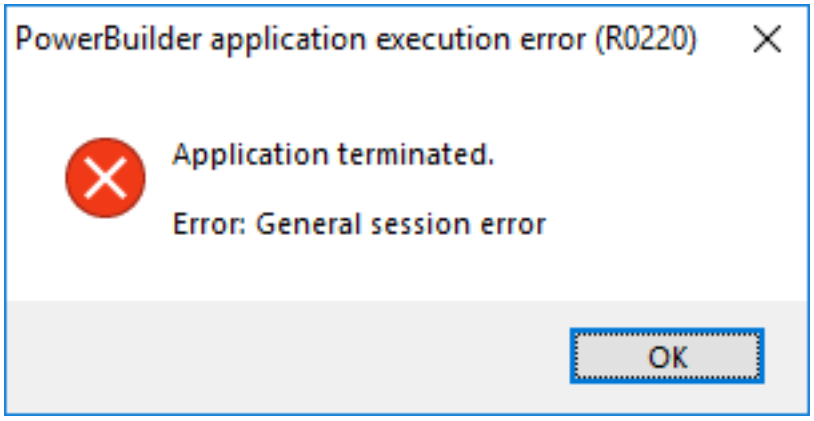
Step 7: Run the following command to allow Nginx to make outbound connections.

```
$ sudo setsebool -P httpd_can_network_connect 1
```

Note

If Nginx is not allowed to make outbound connections, you may encounter the following error when running the application,

Figure 4.27:



and may have the following errors in the \var\log\nginx\error.log file.

```
2021/06/09 02:38:02 [crit] 5364#0: *2 connect() to 172.16.100.35:6000
 failed (13: Permission denied) while connecting to upstream, client:
 172.16.100.35,
server: _, request: "POST /api/ServerApi/CreateSession HTTP/1.1",
 upstream: "http://172.16.100.35:6000/api/ServerApi/CreateSession", host:
 "172.16.100.51"
```

4.4.4.3 Modifying and re-deploying the PowerServer project

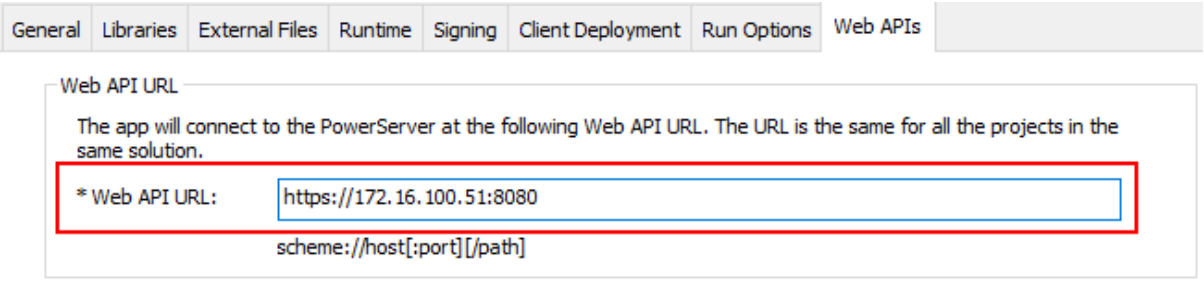
The following modifications are made to the PowerServer project created in the Quick Start guide. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

Step 1: Modify the Web API URL to point to the Nginx reverse proxy server.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the Nginx reverse proxy server, for example, https://172.16.100.51:8080. It is highly recommended that you specify an HTTPS URL for the production environment.

All requests for the PowerServer Web APIs will be first made to https://172.16.100.51:8080 and then redirected by the Nginx reverse proxy server to the PowerServer Web APIs running on Kestrel server (for example, https://172.16.100.35:6000).

Figure 4.28:



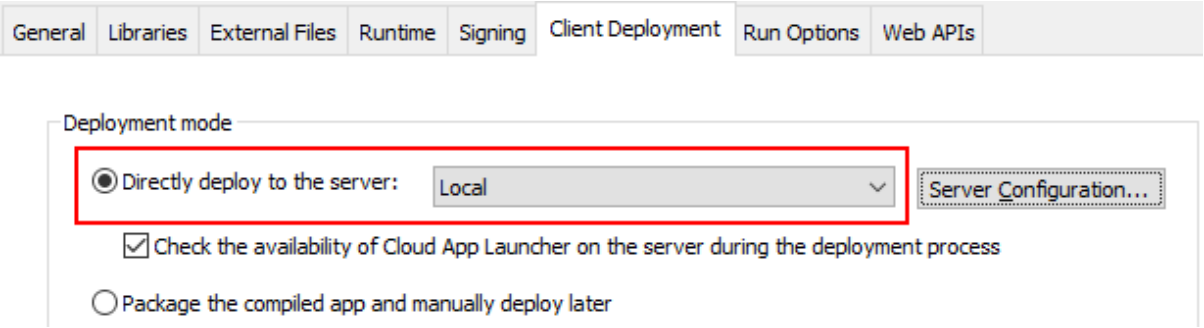
Step 2: Select a Web server for deploying the app files.

On the **Client Deployment** tab of the PowerServer project painter, select a local or remote Web server (IIS, Apache, Nginx, etc.) you have configured properly.

The Web server and the Nginx reverse proxy server can reside in the same or different machine. If the Web server is an Nginx HTTP server, it can be the same or different server instance with the Nginx reverse proxy server. If you want to deploy the app files to the Nginx HTTP server which uses the same server instance as the Nginx reverse proxy server on a Linux machine, you can choose the "Package the compiled app and manually deploy later" option (see Packaging and copying the client app for detailed instructions).

In this tutorial, we choose to deploy the app files to a local IIS Web server.

Figure 4.29:



Step 3: Save the PowerServer project settings and then build and deploy the PowerServer project for the changes to take effect.

4.4.4.4 Starting Web APIs (in development environment)

In this tutorial, we will directly run the PowerServer Web APIs in the development environment, by using either of the following methods:

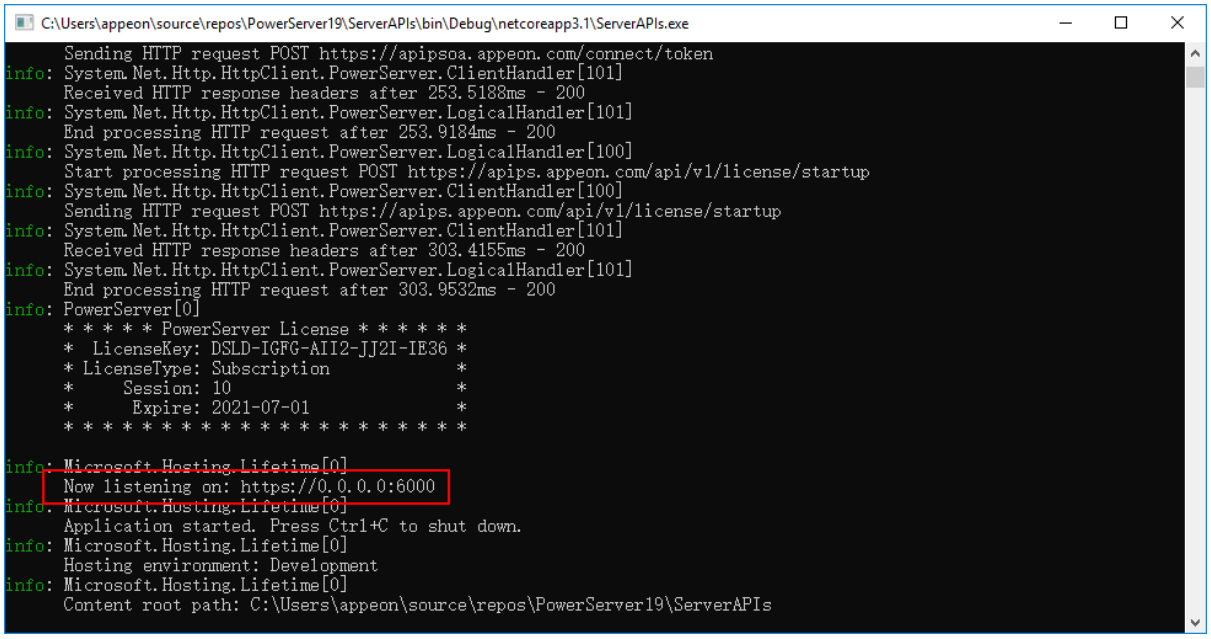
- Execute the "dotnet run --project PowerServer19\ServerAPIs\ServerAPIs.csproj" command, or
- Open the PowerServer C# solution in the SnapDevelop IDE and then click the **Run** button.

PowerServer Web APIs is running as a standalone console application on its own internal Kestrel web server.

Make sure the PowerServer Web APIs is running on the correct IP address and port number. For example, https://172.16.100.35:6000/ in this tutorial. You may modify the port number in the *launchSettings.json* file of the **ServerAPIs** project of the PowerServer C# solution when running in the development environment.

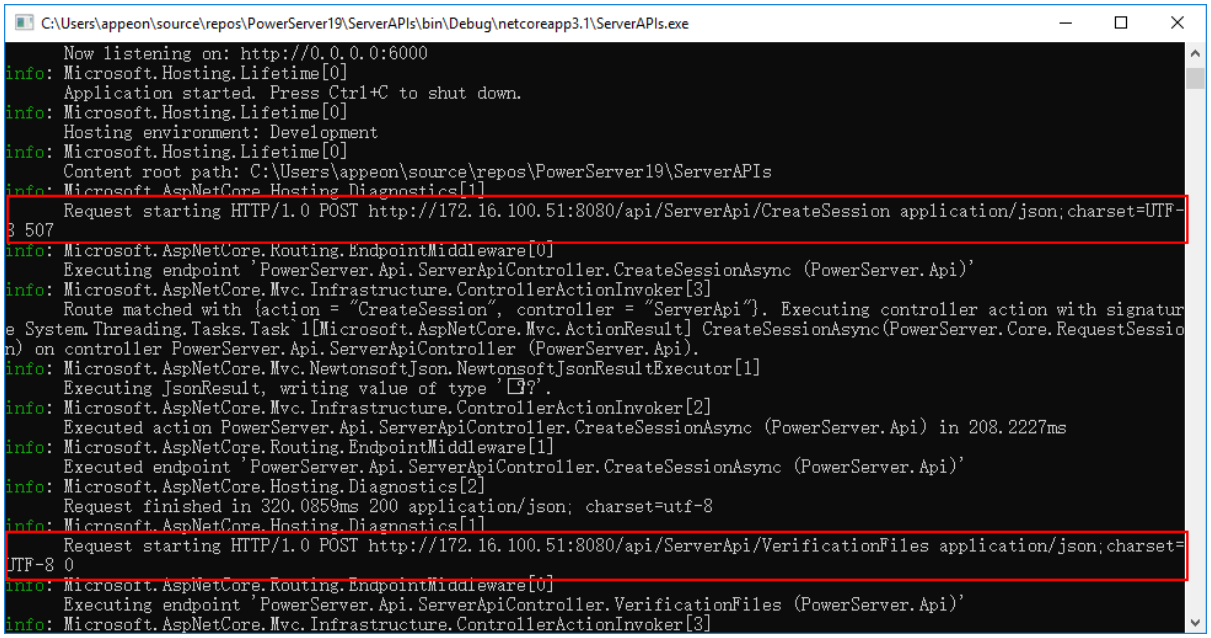
If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Figure 4.30:



When you run the application (https://172.16.100.72:80/pssales in this tutorial), you will be able to see from the console that the requests are going through successfully and the requests are originally made to the Nginx proxy server (https://172.16.100.51/8080 in this tutorial).

Figure 4.31:



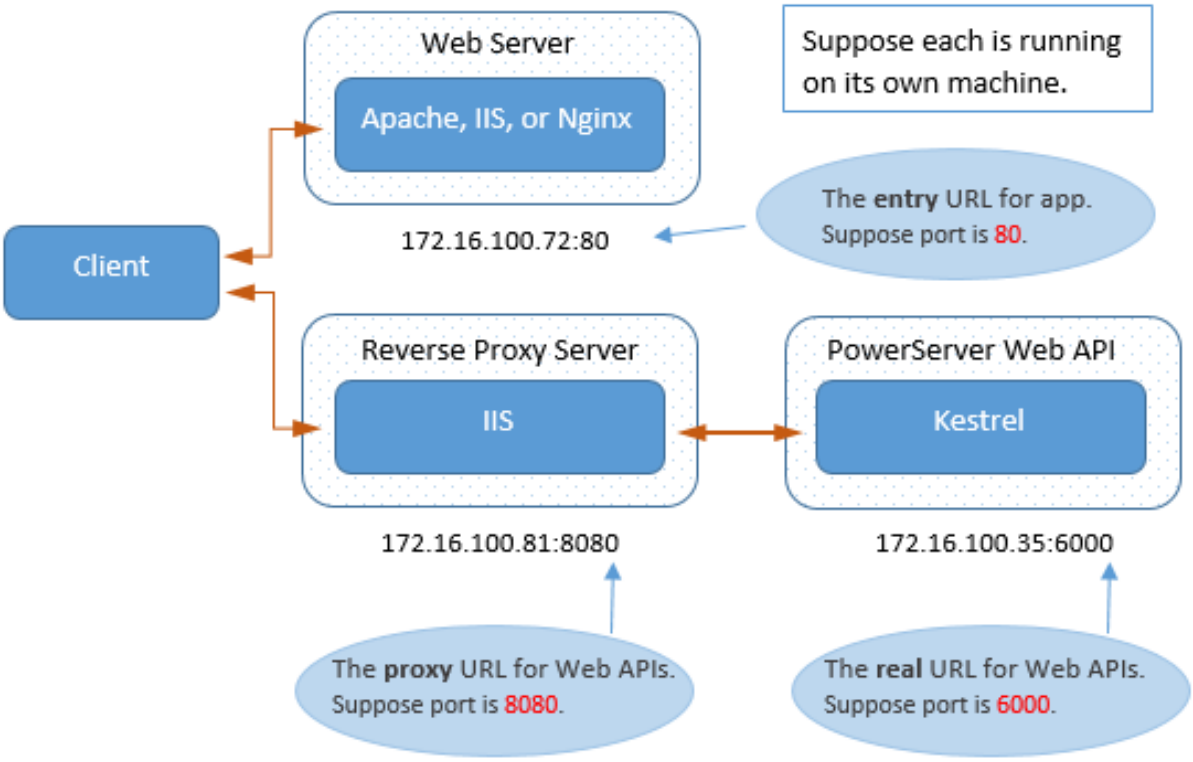
4.4.5 Configuring IIS reverse proxy server

4.4.5.1 Preparations

In this tutorial, we will learn how to set up Windows IIS as the reverse proxy server which redirects requests to the PowerServer Web APIs running on the Kestrel server. This is also known as the IIS out-of-process hosting which runs the PowerServer Web APIs in a process separate from the IIS worker process and forwards the requests made to the IIS reverse proxy to the Kestrel server.

In this tutorial, we will configure and use the following server environment and URLs. Be careful to use the correct port number and make sure the port is not occupied by any other program.

Figure 4.32:



Step 1: Set up the reverse proxy server with the following OS and software (install the software in the order listed).

- Windows Server 2019 (64-bit)
- IIS

  The section Installing Web Server (IIS) has detailed installation instructions.
- IIS URL Rewrite

  Download and install the URL Rewrite extension.

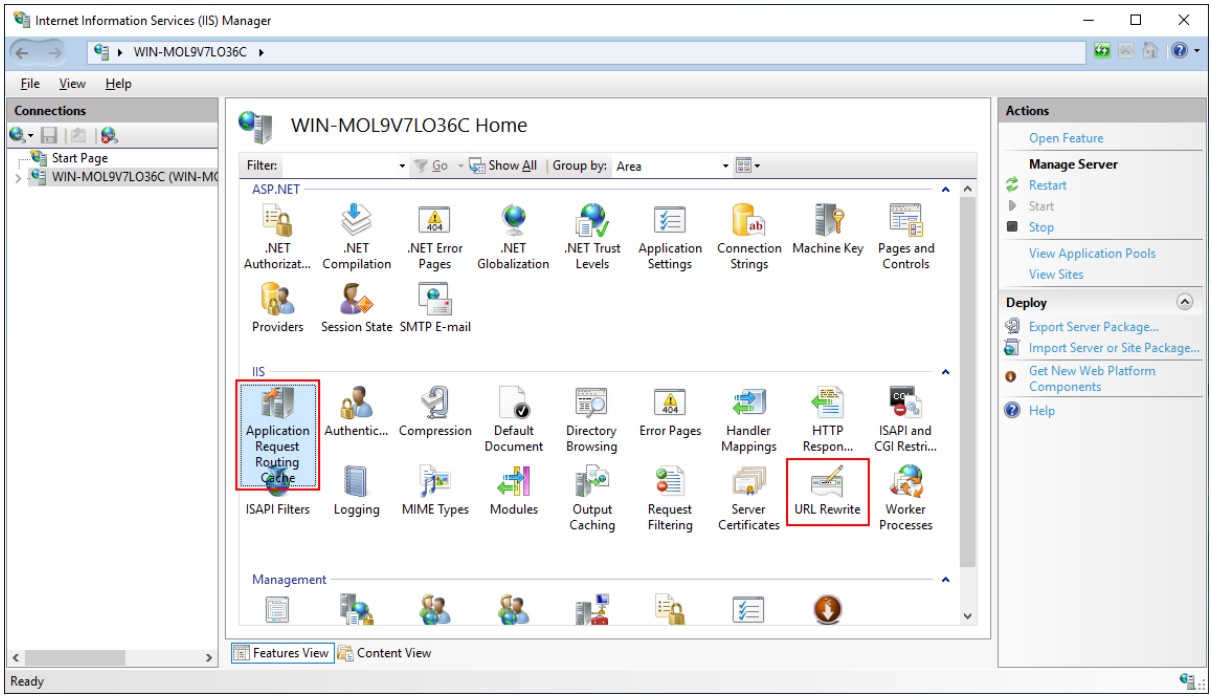
  URL Rewrite must be installed prior to ARR, as ARR depends on URL Rewrite.

- IIS Application Request Routing (ARR)

  Download and install the Application Request Routing extension.

  After installation, you should be able to see the **Application Request Routine Cache** and **URL Rewrite** features in the IIS manager.

Figure 4.33:



Step 2: Make sure the server can connect to the NuGet site: https://www.nuget.org (for installing PowerServer NuGet packages) and the following Appeon sites (through port number 80): https://apips.appeon.com and https://apipsoa.appeon.com (or https://apips.appeon.net and https://apipsoa.appeon.net) (for validating the PowerServer license).

Note

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

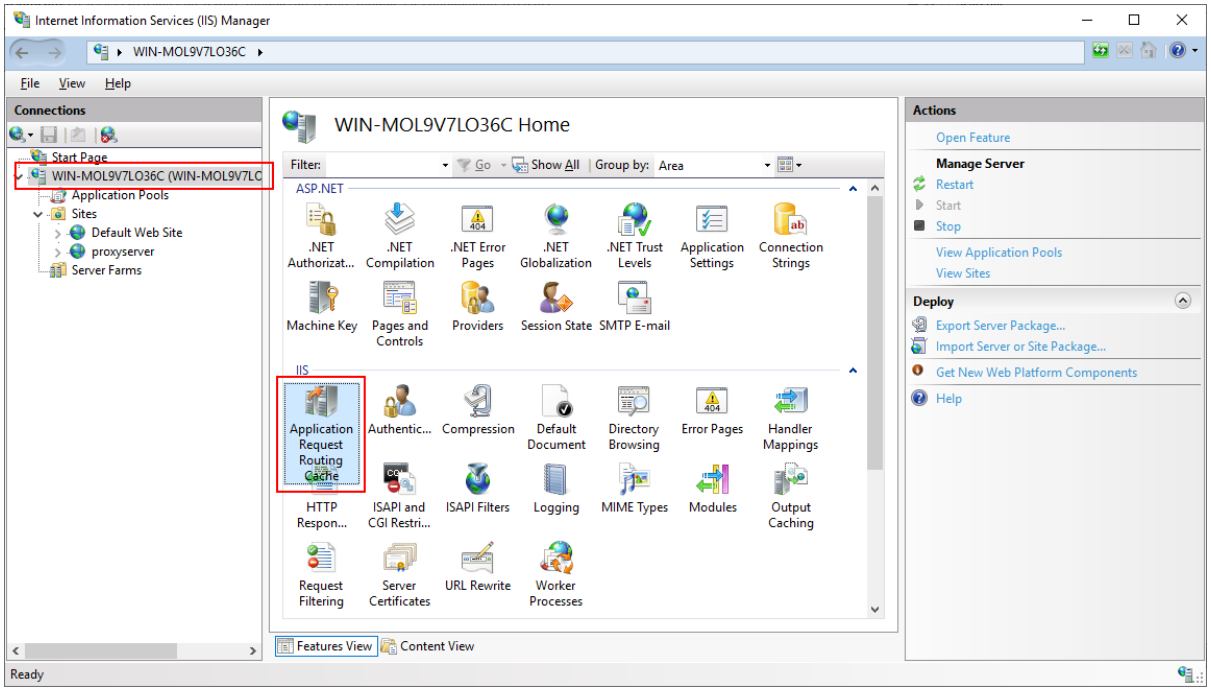
Step 3: Configure Windows Defender Firewall on the server to allow the port number (80 and 8080 in this tutorial or any port number you choose). The section "Configuring Windows Defender Firewall" has detailed instructions.

4.4.5.2 Configuring IIS

This section is to configure IIS as a reverse proxy server.

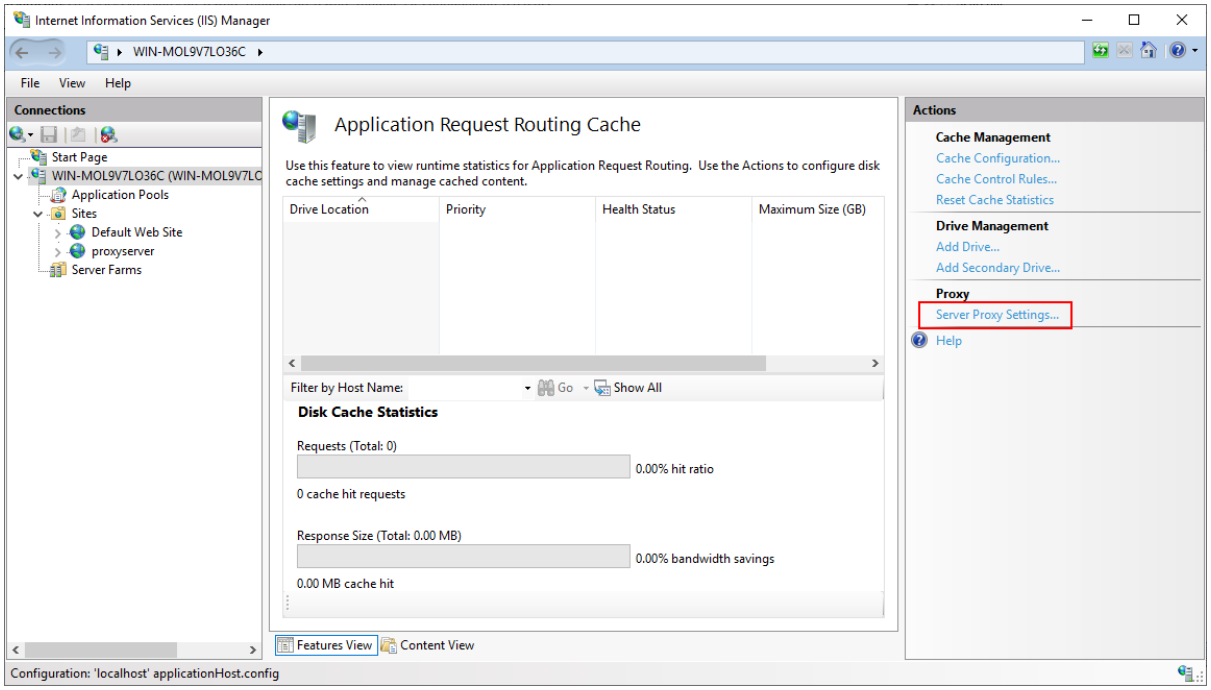
Step 1: Open the IIS manager, select the server in the **Connections** pane, and then double click **Application Request Routing Cache** to open the feature.

Figure 4.34:



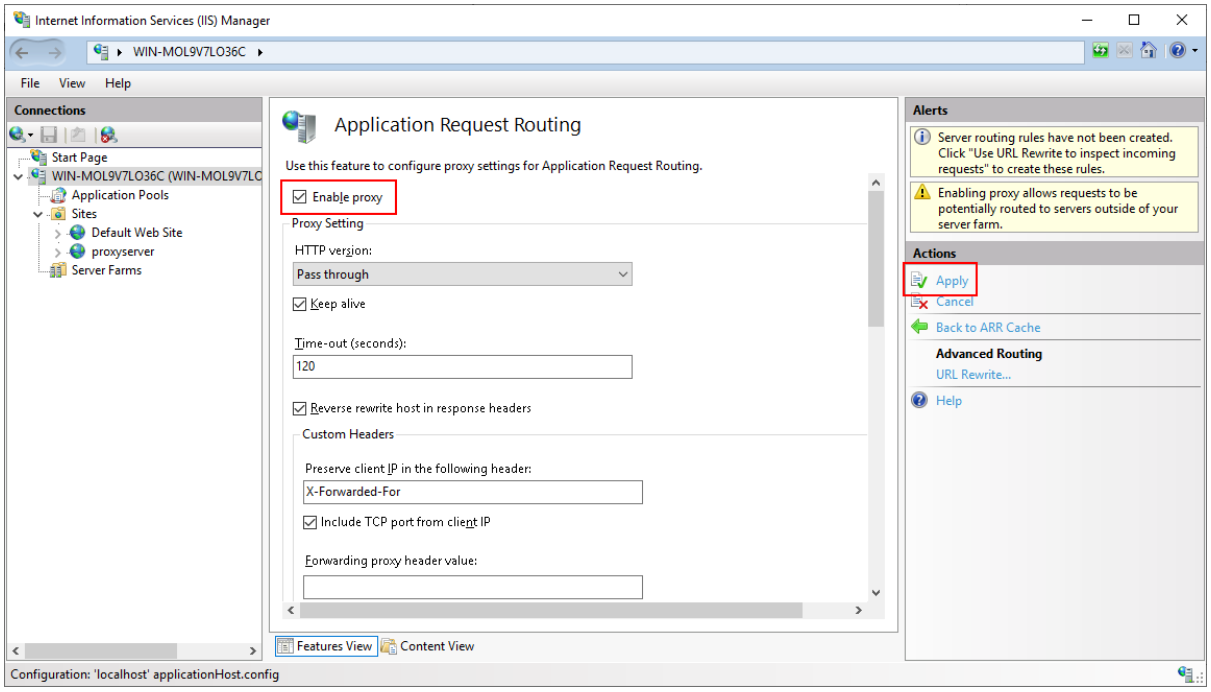
Step 2: In the **Actions** pane, click **Server Proxy Settings**.

Figure 4.35:



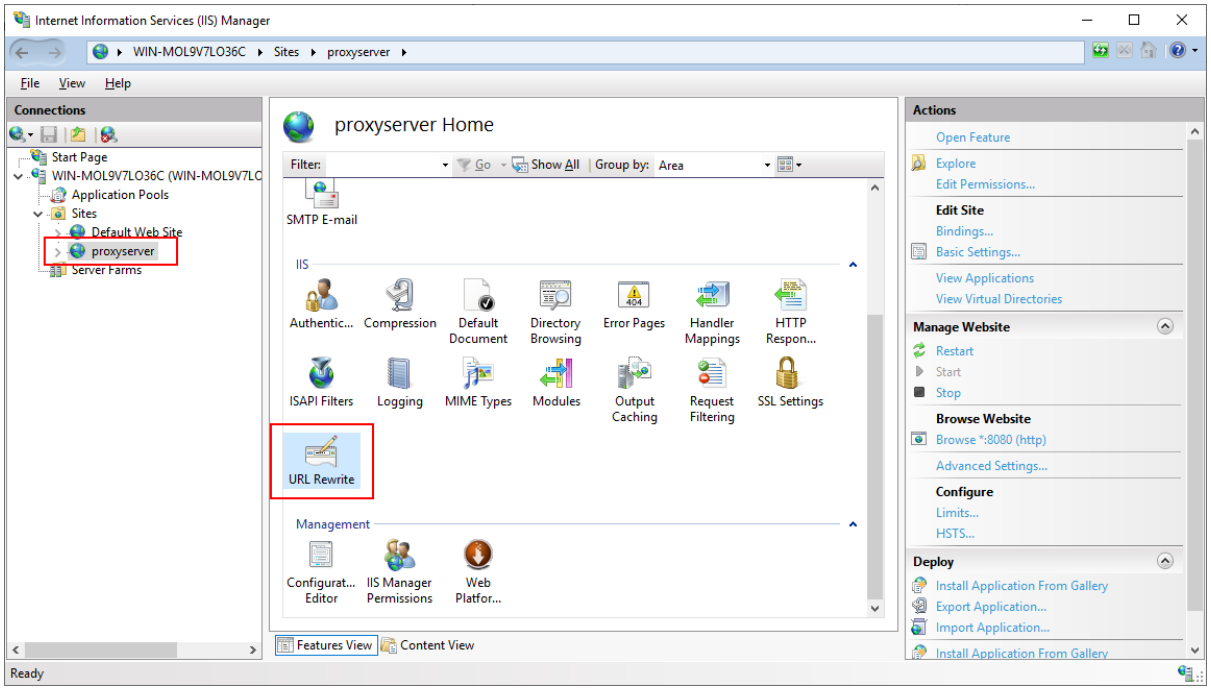
Step 3: On the **Application Request Routing** page, select **Enable Proxy**; and then in the **Actions** pane, click **Apply**. This enable ARR as a proxy at the server level.

Figure 4.36:



Step 4: Select the website (listening on port 8080 in this tutorial) in the **Connections** pane, and then double click **URL Rewrite** to open the feature.

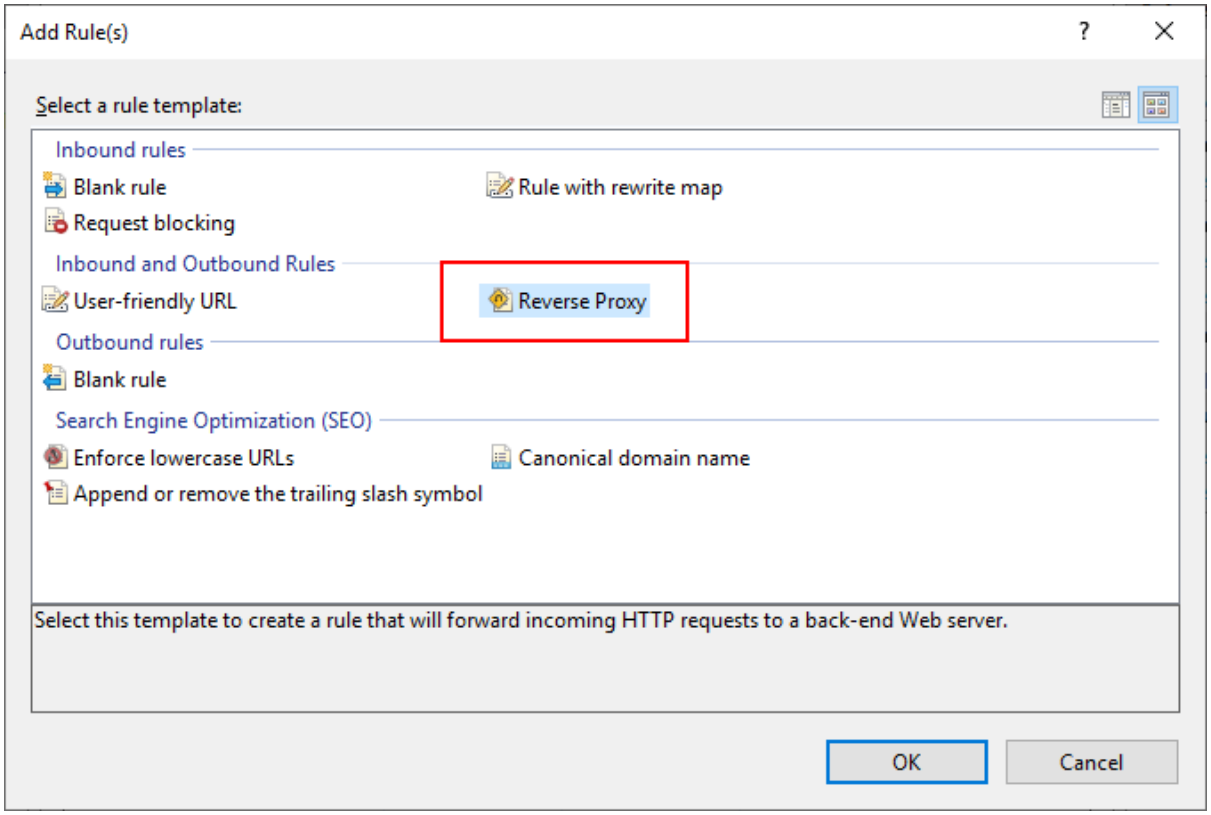
Figure 4.37:



Step 5: In the **Actions** pane, click **Add Rule(s)**.

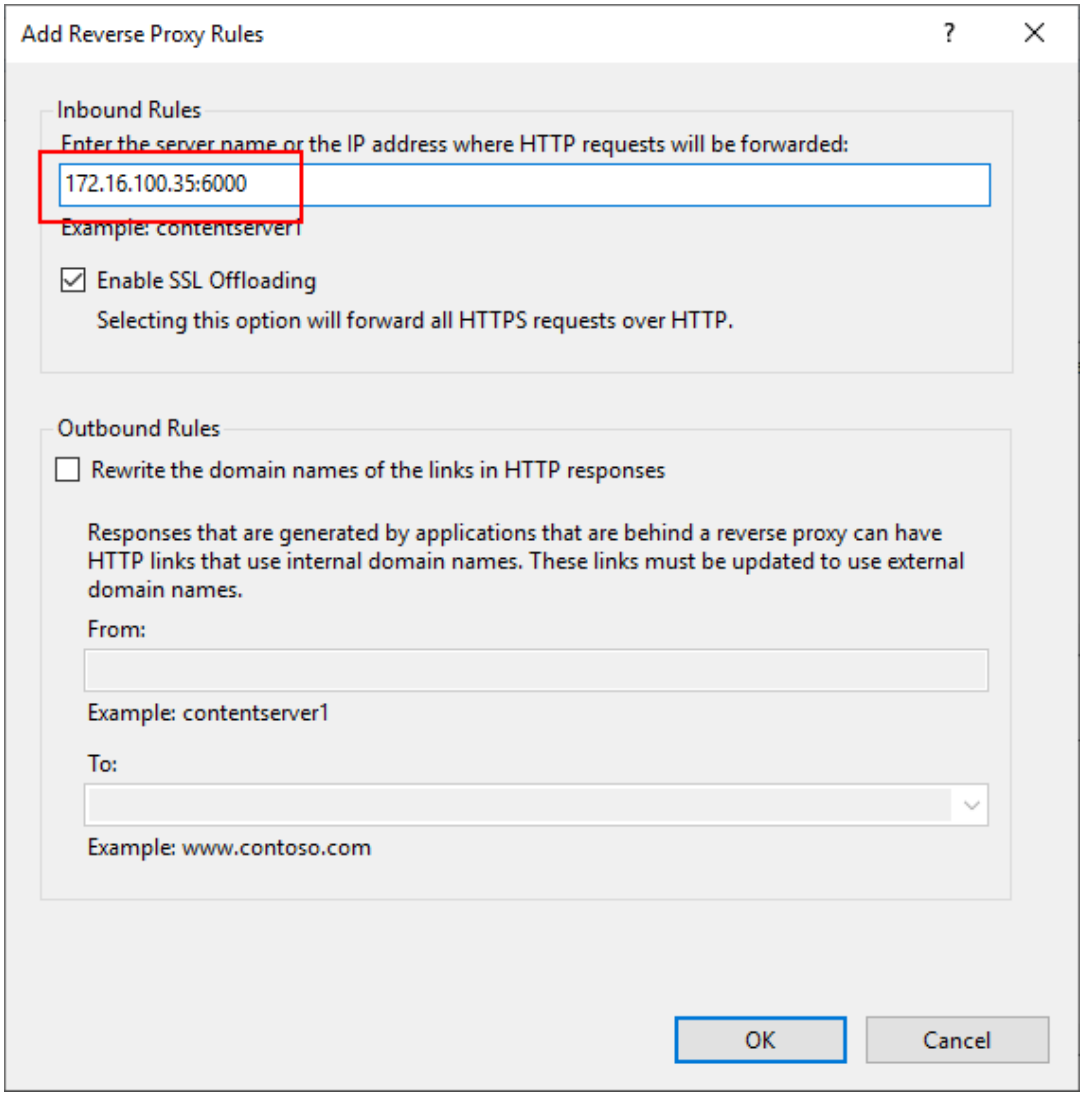
Step 6: In the **Add Rule(s)** dialog, select **Reverse Proxy** and click **OK**.

Figure 4.38:



Step 7: In the **Add Reverse Proxy Rules** dialog, input the URL of the PowerServer Web API running on the Kestrel server (https://172.16.100.35:6000/ in this tutorial). Click **OK**.

Figure 4.39:



4.4.5.3 Modifying and re-deploying the PowerServer project

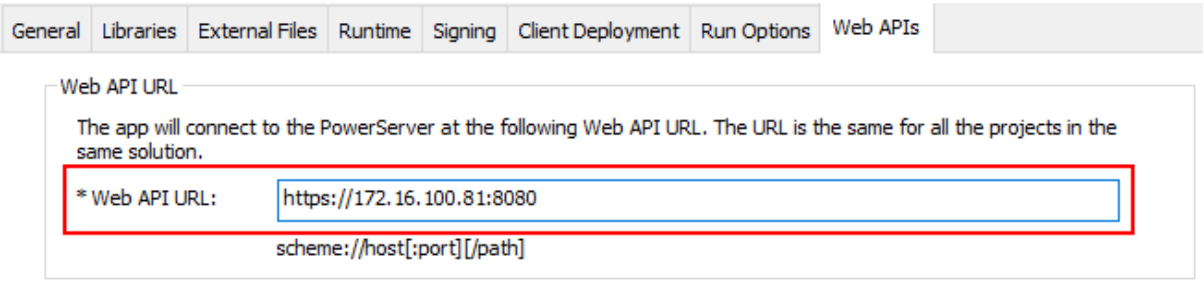
The following modifications are made to the PowerServer project created in the Quick Start guide. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

Step 1: Modify the Web API URL to point to the IIS reverse proxy server.

On the **Web APIs** tab of the PowerServer project painter, specify the URL of the IIS reverse proxy server, for example, https://172.16.100.81:8080. It is highly recommended that you specify an HTTPS URL for the production environment.

All requests for the PowerServer Web APIs will be first made to https://172.16.100.81:8080 and then redirected by the IIS reverse proxy server to the PowerServer Web APIs running on the Kestrel server (for example, https://172.16.100.35:6000).

Figure 4.40:



Step 2: Select a Web server for deploying the app files.

On the **Client Deployment** tab of the PowerServer project painter, select a local or remote Web server (IIS, Apache, Nginx, etc.) you have configured properly.

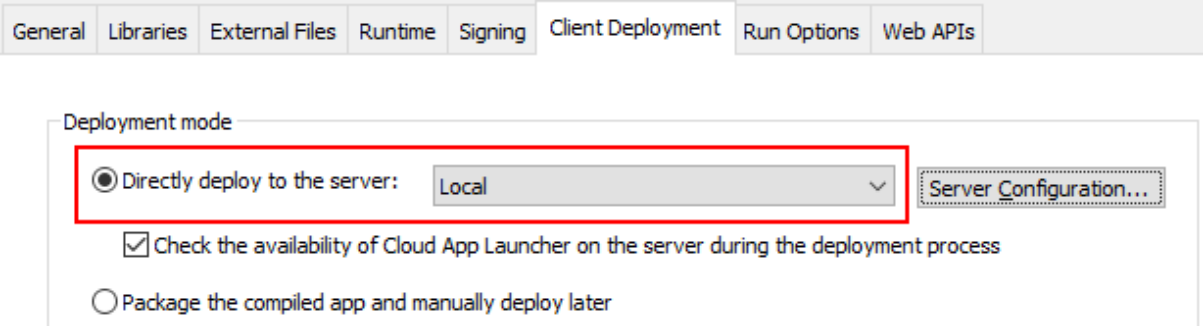
The Web server and the IIS reverse proxy server can reside in the same or different machine. If the Web server is an IIS HTTP server, it can be the same or different server instance with the IIS reverse proxy server.

In this tutorial, we choose to deploy the app files to a local IIS Web server.

To use the same IIS server instance as the IIS HTTP server and the reverse proxy server, you can choose from these two options:

- If you choose the "Directly deploy to the server" option, make sure you have configured the FTP settings properly for the server. See Setting up IIS > Creating an IIS FTP site for detailed instructions.
- If you choose the "Package the compiled app and manually deploy later" option, follow the instructions in Packaging and copying the client app.

Figure 4.41:



Step 3: Save the PowerServer project settings and then build and deploy the PowerServer project for the changes to take effect.

4.4.5.4 Starting Web APIs (in development environment)

In this tutorial, we will run the PowerServer Web APIs in the development environment, by using either of the following methods:

- Execute the "dotnet run --project PowerServer19\ServerAPIs\ServerAPIs.csproj" command, or

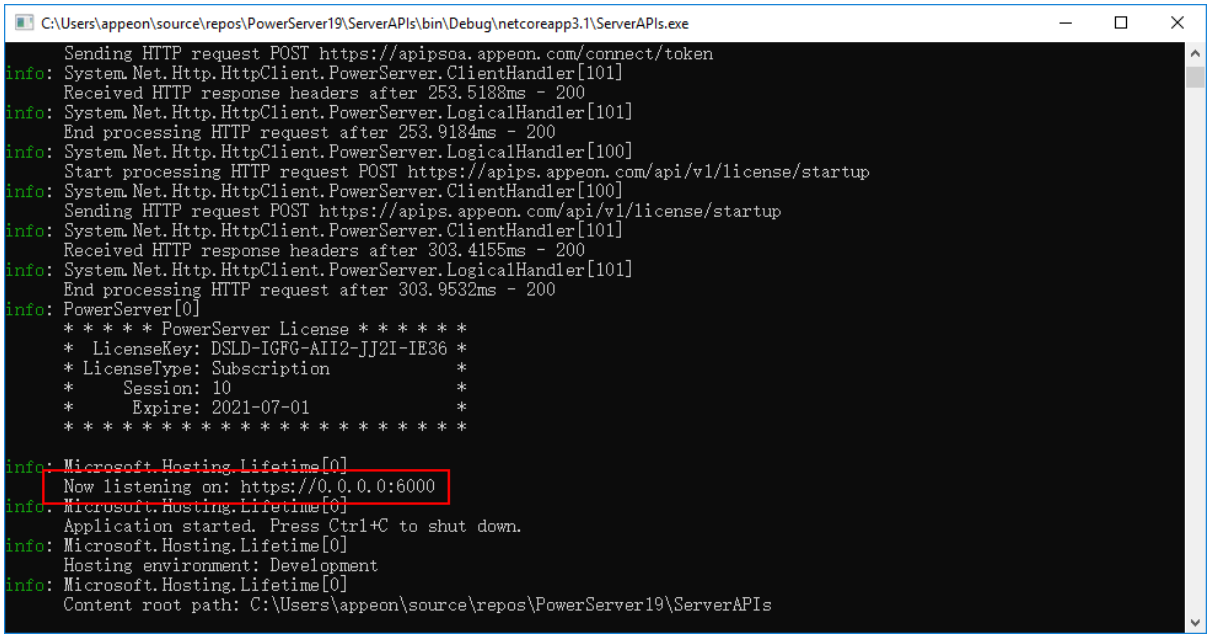
- Open the PowerServer C# solution in the SnapDevelop IDE and then click the **Run** button.

PowerServer Web APIs is running as a standalone console application on its own internal Kestrel web server.

Make sure the PowerServer Web APIs is running on the correct IP address and port number. For example, https://172.16.100.35:6000/ in this tutorial. You may modify the port number in the *launchSettings.json* file of the **ServerAPIs** project of the PowerServer C# solution when running in the development environment.

If the server connects to Internet through a proxy server, make sure to configure the proxy server settings in the PowerServer Web API as well (the **ServerAPIs** project > **Server.json** file > "**ProxyOptions**" block).

Figure 4.42:



When you run the application (https://172.16.100.72:80/pssales in this tutorial), you will be able to see from the console that the requests are going through successfully.

5 Tutorial 5: Load-balancing PowerServer Web APIs

5.1 Overview

PowerServer Web APIs provides no clustering function to support load-balancing or fail-over; but you can install and configure a third-party server (such as Nginx, Apache, IIS, AWS ALB, Azure Application Gateway, AWS EKS (K8S), Azure AKS (K8S) etc.) as a load balancer to direct requests to a group of .NET servers. (Fail-over is currently unsupported.)

PowerServer Web APIs uses cookie to achieve session persistence (it returns a cookie when each user session is created and then includes the cookie in each request from that user session). Therefore, you will need to configure the third-party server to support the following:

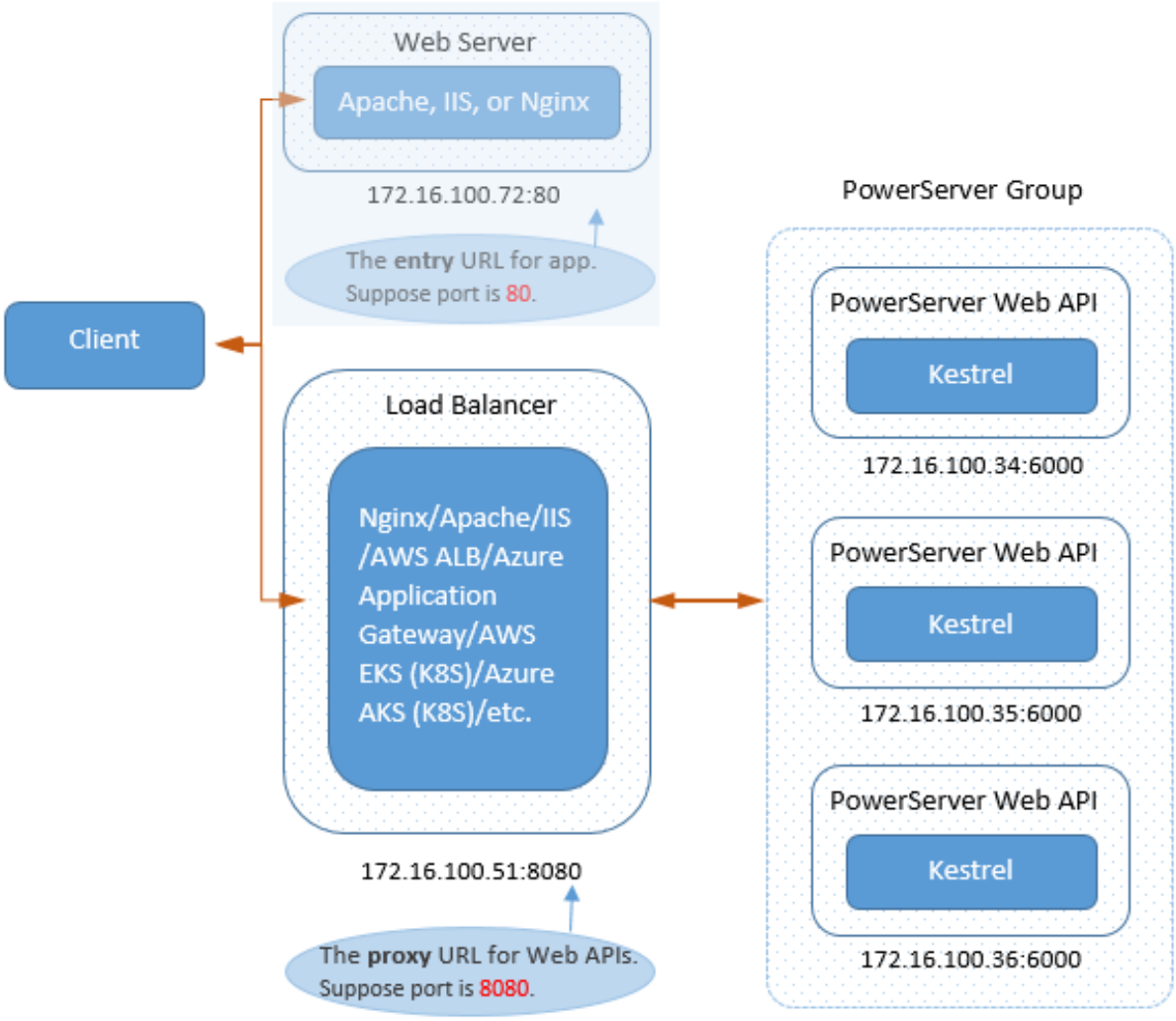
- "sticky" or "persistent" sessions (this ensures the requests from the same user will always be directed to the same PowerServer Web APIs)

- the cookie timeout value must be equal to or greater than the session timeout value (this ensures the cookie stays valid during a session)

  The session timeout value is by default 3600 seconds (it is set in the *Applications.json* file in the PowerServer C# solution).

When you configure the Web API URL for the application, you should point to the URL of the load balancer (for example, https://172.16.100.51:8080 in the following graph).

Figure 5.1:



5.2 Configuring Nginx as a load balancer

This tutorial will walk you through configuring Nginx as a load balancer to direct client requests to a group of PowerServer Web APIs. You can choose one of the following methods:

- Use the Nginx third-party module (Nginx Sticky Module) to support session persistence via cookies.

- Use Nginx Plus that supports session persistence via cookies.

  Nginx Plus is a commercial product.

- Use the IP hash load-balancing method to support session persistence via IP address.

  With IP hash, the client's IP address is used as a hashing key to determine which PowerServer Web APIs should be selected for the client's request. This ensures the requests from the same user session is always directed to the same PowerServer Web APIs. However, the IP-hash-based session persistence cannot guarantee that user sessions are evenly distributed across servers. For example, there may be situations where a lot of user

sessions are coming with the same IP address (behind proxies) and all these user sessions will go to the same server, which might cause unbalanced load.

5.2.1 Using Nginx Sticky Module

This tutorial will walk you through configuring Nginx + Nginx Sticky Module as a load balancer to direct client requests to a group of PowerServer Web APIs. You will have to configure Nginx + Nginx Sticky Module as a load balancer and use the sticky cookie to support session persistence. With sticky cookies, the requests from the same user session are always directed to the same PowerServer Web APIs.

Step 1: Download the source code of Nginx and Nginx Sticky Module separately.

- Download the source code of Nginx from https://nginx.org.
- Download the source code of Nginx Sticky Module from https://bitbucket.org/nginx-goodies/nginx-sticky-module-ng/src/master/.

Step 2: Re-compile Nginx to include the Nginx Sticky Module.

```
./configure ... --add-module=/absolute/path/to/nginx-sticky-module-ng
make
make install
```

Step 3: Check if any syntax error in the Nginx configuration file, and then restart Nginx for the changes to take effect.

```
nginx -t
```

```
systemctl restart nginx
```

Step 4: Configure Nginx to direct requests to the PowerServer Web APIs group using the sticky cookie load-balancing method.

1. Open the nginx.conf file in a text editor (nginx.conf is located in /etc/nginx/ in Linux).
2. Under the "server" block that defines the virtual server, add another "server" block and "upstream" block that define the server group.

   - The "upstream" directive defines the PowerServer Web APIs group.

     In the following example, the "upstream" block consists of two server configurations; it could consists of more.

     The "upstream" block also consists of the "sticky" directive which defines that the sticky-cookie load-balancing method will be used when determining which server in the group the request will be directed to.

   - The "listen" directive specifies the port number for the requests. The Web API URL should point to this port number.
   - The "proxy_pass" directive forwards the request to the server group defined in the "upstream" directive, therefore, it should match with the upstream name.

   The following configuration defines a PowerServer Web APIs group named **webapi** which consists of three .NET servers: https://172.16.100.34:6000/,

https://172.16.100.35:6000/, and https://172.16.100.36:6000/ and requests made to the URL: https://<server>:8090/ will be redirected to the PowerServer Web APIs group.

```
upstream webapi
       {
        server 172.16.100.34:6000;
        server 172.16.100.35:6000;
        server 172.16.100.36:6000;
        sticky name=route hash=sha1 expires=1h;
       }
server {
        listen       8090;
        server_name  localhost;

        location / {
          proxy_pass https://webapi;
          proxy_set_header Host $host;
          proxy_set_header X-Real-IP $remote_addr;
          proxy_set_header X-Forwarded-For $remote_addr;
          }
        }
```

5.2.2 Using Nginx Plus

This tutorial will walk you through configuring Nginx Plus as a load balancer to direct client requests to a group of PowerServer Web APIs. You will have to configure Nginx Plus as a load balancer and use the sticky cookie to support session persistence. With sticky cookies, the requests from the same user session are always directed to the same PowerServer Web APIs.

Step 1: Install Nginx Plus. Nginx Plus is a commercial product. You will need to purchase it first or apply for a trial version of Nginx Plus.

Step 2: Configure Nginx Plus to direct requests to the PowerServer Web APIs group using the sticky cookie load-balancing method.

1. Open the nginx.conf file in a text editor.

2. Add an "http" block that defines the server group.

   - The "listen" directive specifies the port number for the requests. The Web API URL should point to this port number.

   - The "proxy_pass" directive forwards the request to the server group defined in the "upstream" directive, therefore, it should match with the upstream name.

   - The "upstream" directive defines the PowerServer Web APIs group.

     In the following example, the "upstream" block consists of three server configurations; it could consists of more.

     The "upstream" block also consists of the "sticky" directive and defines the cookie name and timeout value. The cookie timeout value must be equal to or greater than the session timeout value (which is 3600 seconds by default). In the following example, the cookie timeout value is set to 1 hour (which is 3600 seconds).

For more information about the sticky cookie and the other load-balancing methods (such as sticky route and sticky learn), refer to https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/#enabling-session-persistence.

The following configuration defines a PowerServer Web APIs group named **servergroup** which consists of three .NET servers: https://172.16.100.34:6000/, https://172.16.100.35:6000/, and https://172.16.100.36:6000/, and requests made to the URL: https://<server>:8080/ will be redirected to the PowerServer Web APIs group.

```
http {
    server {
        listen  8080;
        location / {
           proxy_set_header Host $http_host;
           proxy_pass https://servergroup;
        }
    }
    upstream servergroup {
        sticky cookie srv_id expire=1h path=/;
        server https://172.16.100.34:6000;
        server https://172.16.100.35:6000;
        server https://172.16.100.36:6000;
        }
}
```

5.2.3 Using IP hash load-balancing

The IP-hash-based session persistence cannot guarantee that user sessions are evenly distributed across servers. For example, there may be situations where a lot of user sessions are coming with the same IP address (behind proxies) and all these user sessions will go to the same server, which might cause unbalanced load. Therefore consider the impact carefully before you decide to go this way.

To configure Nginx as a load balancer and use the IP hash load-balancing method,

Step 1: Follow the sections below to install Nginx.

- Setting up Nginx on Windows > "Preparations" and "Installing Nginx" sections
- Setting up Nginx on Linux > "Preparations" and "Installing Nginx" sections

Step 2: Configure Nginx to direct requests to the PowerServer Web APIs group using the IP hash load-balancing method.

1. Open the nginx.conf file in a text editor (nginx.conf is located in the ..\nginx-1.19.10\conf folder in Windows, or /etc/nginx/ in Linux).
2. Under the "server" block that defines the virtual server, add another "server" block and "upstream" block that define the server group.
   - The "listen" directive specifies the port number for the requests. The Web API URL should point to this port number.
   - The "proxy_pass" directive forwards the request to the server group defined in the "upstream" directive, therefore, it should match with the upstream name.

- The "upstream" directive defines the PowerServer Web APIs group.

  In the following example, the "upstream" block consists of three server configurations; it could consists of more.

  The "upstream" block also consists of the "ip_hash" directive which defines that the IP hash load-balancing method will be used when determining which server in the group the request will be directed to.

The following configuration defines a PowerServer Web APIs group named **webapi** which consists of three .NET servers: https://172.16.100.34:6000/, https://172.16.100.35:6000/, and https://172.16.100.36:6000/, and requests made to the URL: https://<server>:8080/ will be redirected to the PowerServer Web APIs group.

```
server {
    listen  8080;
    location / {
       proxy_set_header Host $http_host;
       proxy_pass https://webapi;
    }
}
upstream webapi{
    ip_hash;
    server https://172.16.100.34:6000;
    server https://172.16.100.35:6000;
    server https://172.16.100.36:6000;
    }
```

5.3 Configuring IIS as a load balancer

This tutorial will walk you through configuring IIS as a load balancer to direct client requests to a group of PowerServer Web APIs. You will have to configure IIS to support sticky sessions.

Step 1: Follow the sections below to install IIS.

- Setting up IIS > "Preparations" and "Installing Web Server (IIS)" sections

Step 2: Follow the sections below to install the extensions required by IIS to work as proxy server and load balancer.

- Configuring IIS reverse proxy server > "Preparations" section

Step 3: Configure IIS as a proxy server and load balancer which redirects requests to the PowerServer Web APIs group.

1. Create a new website "proxyserver" which binds to port number 8080. You can also use the existing Default Web Site (with port 80).

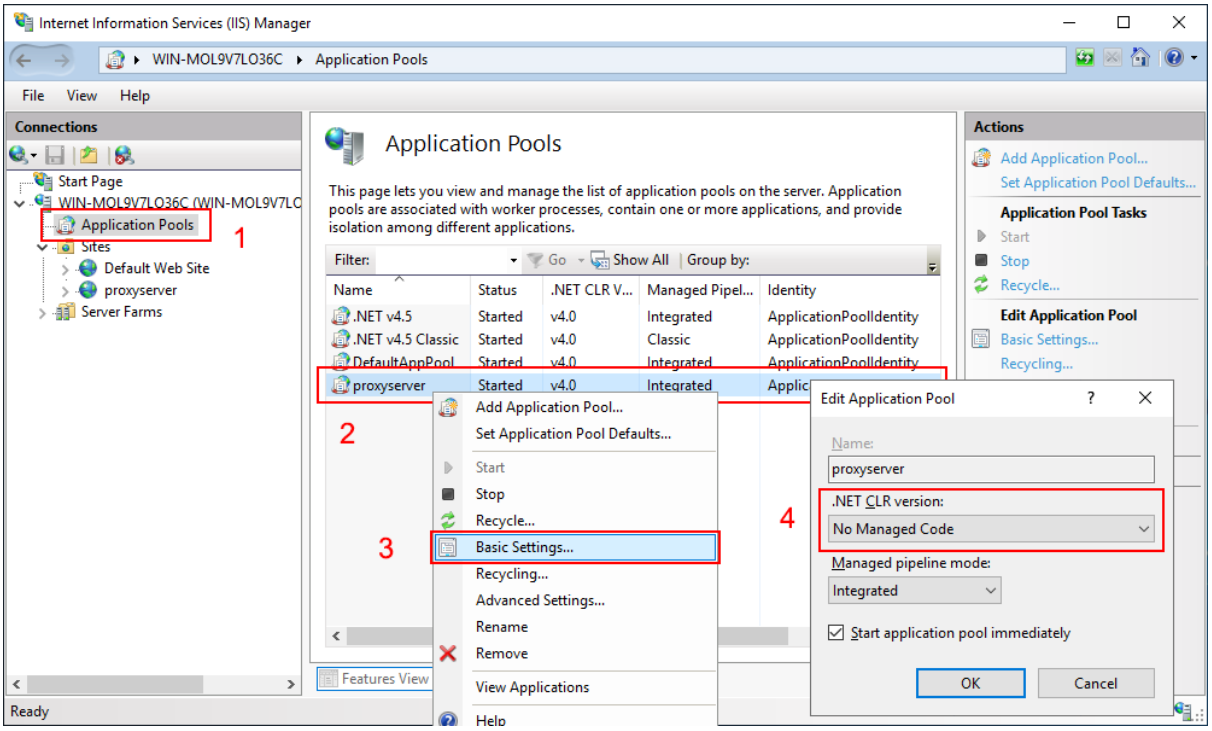
2. Set the application pool to "No managed code".

   a. Select **Application Pools** in the **Connections** pane.

   b. Right click "proxyserver" in the list of application pools and then select **Basic Settings**.

c. In the **Edit Application Pool** window, select **No Managed Code** from the **.NET CLR version** list box, and then click **OK**.

Figure 5.2:



3. Set the cookie timeout value to a value equal to or greater than the session timeout value which is 3600 seconds (60 minutes) by default.

Figure 5.3:

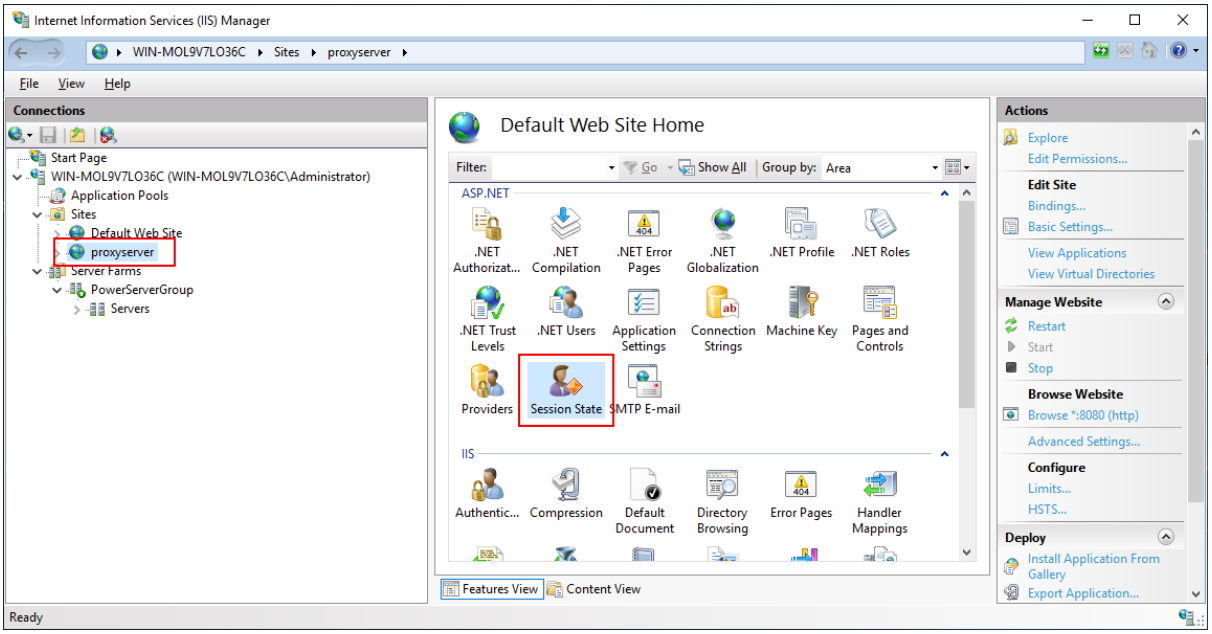
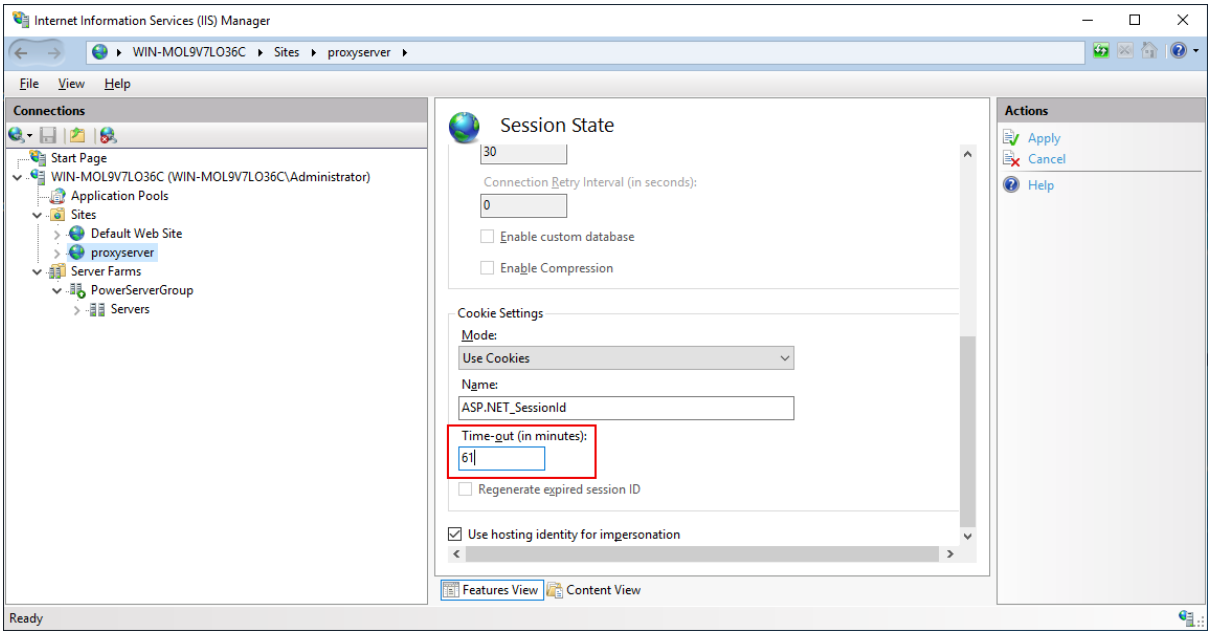


Figure 5.4:



4. Create a server farm that includes the group of PowerServer Web APIs.

   a. Right click **Server Farms** in the **Connections** pane, and then select **Create Server Farm**.

      The **Server Farms** node will not be available if "IIS Application Request Routing (ARR)" is not installed.

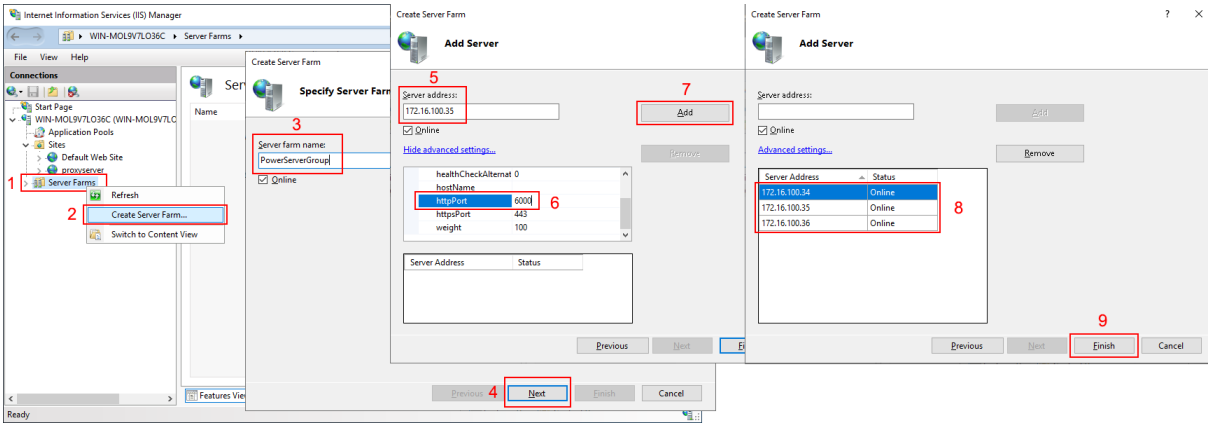
   b. In the **Create Server Farm** window, specify the server farm name and then click **Next**.

   c. Add the server instance by inputting the host name or IP address of PowerServer Web APIs, clicking **Advanced settings** to specify the port number of PowerServer Web APIs, and then clicking **Add**.

   d. Repeat the previous step to add the server instances one by one and then click **Finish**.

   e. Select **Yes** when asked whether to automatically create a URL rewrite rule.

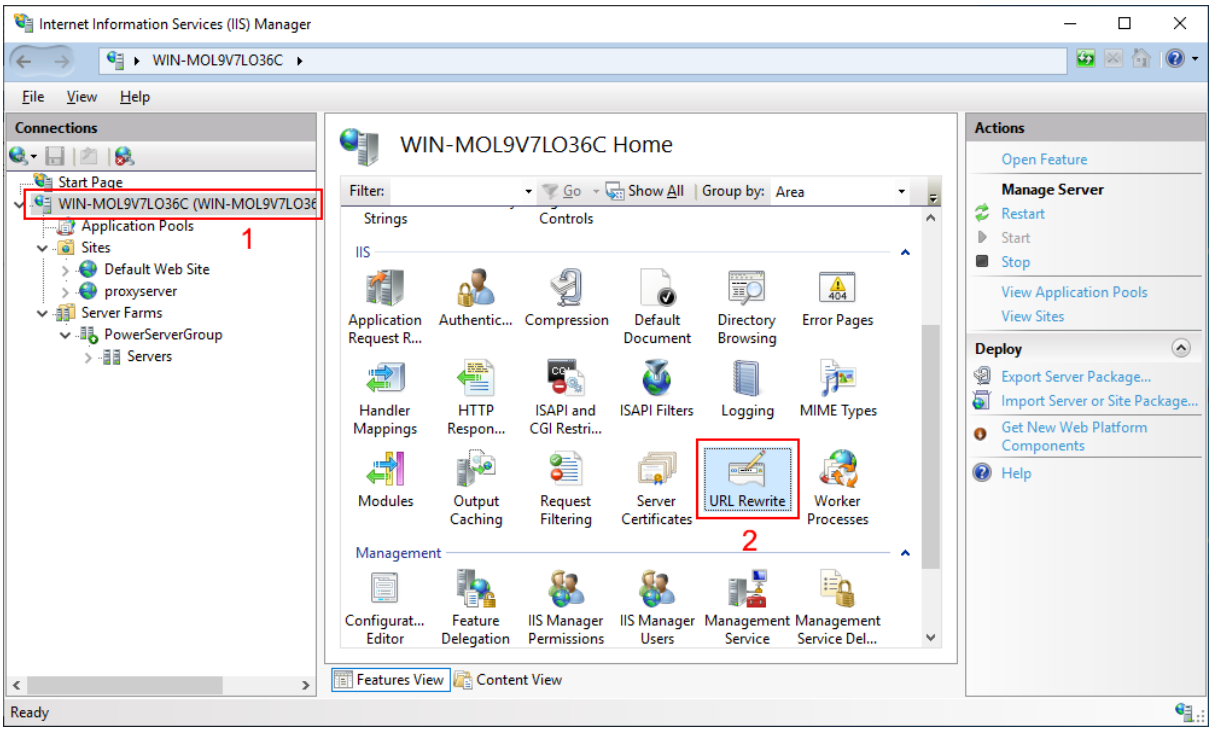
Figure 5.5:



5. Modify the URL rewrite rule.

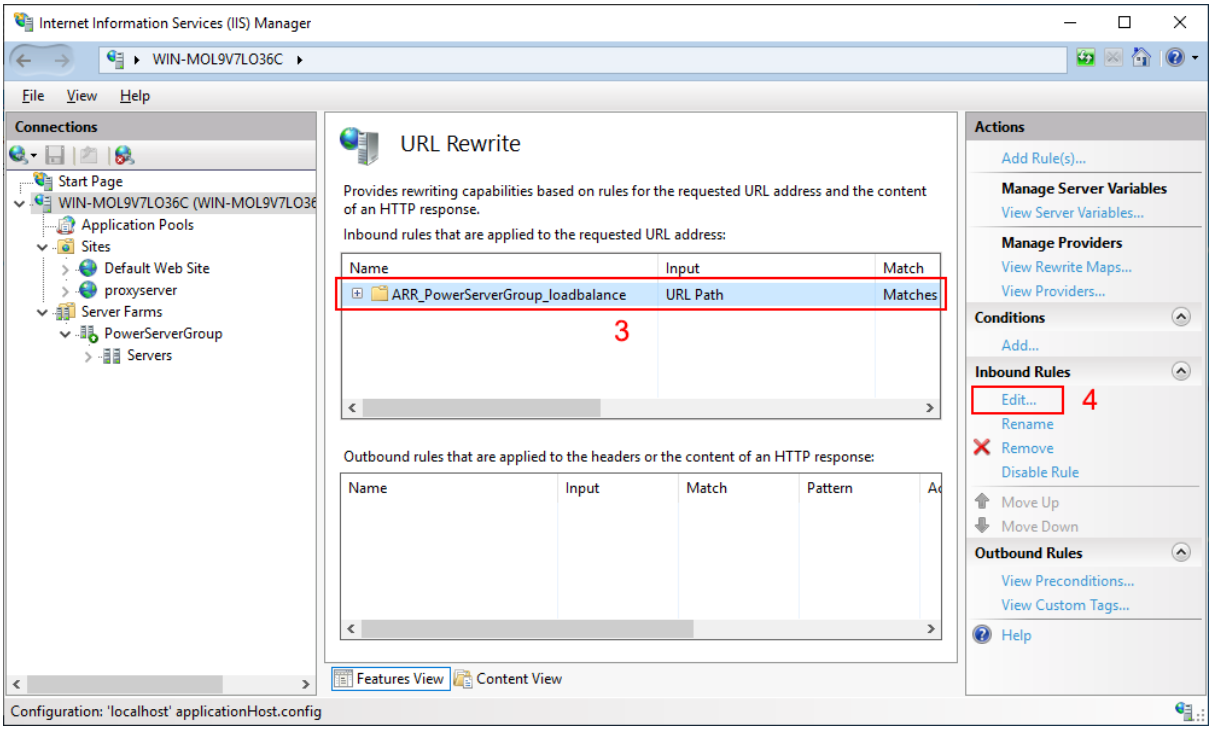
   a. Select the server in the **Connections** pane and then double click **URL Rewrite** in the features view to open the feature.

Figure 5.6:



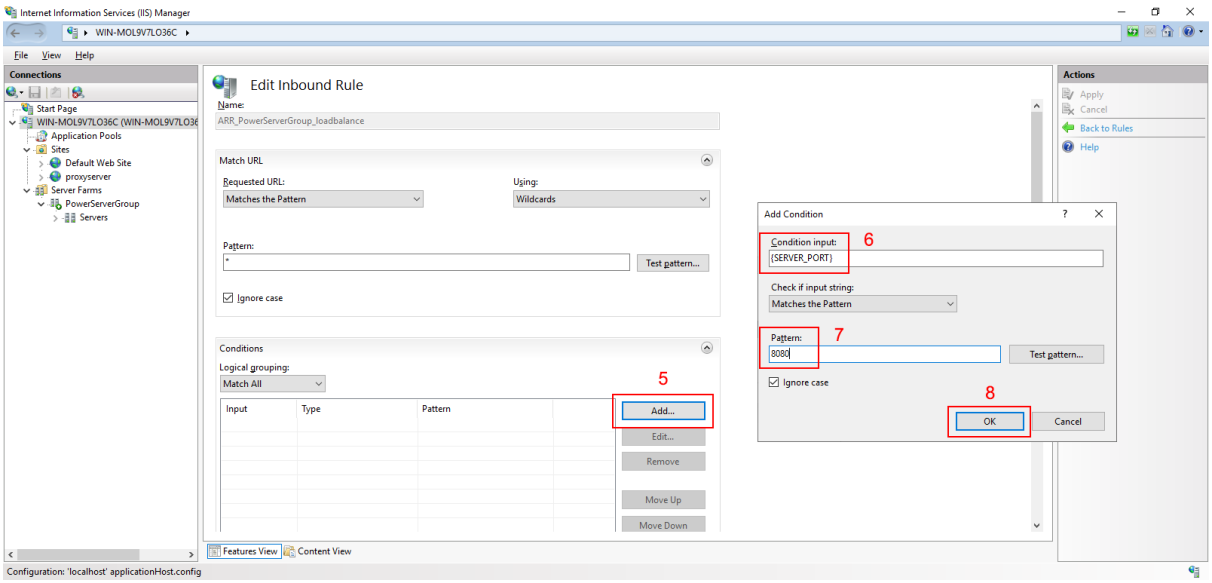
   b. Select the "ARR_PowerServerGroup_loadbalance" rule (this rule is automatically created when you create the server farm) and then click **Edit** from the **Actions** pane.

Figure 5.7:



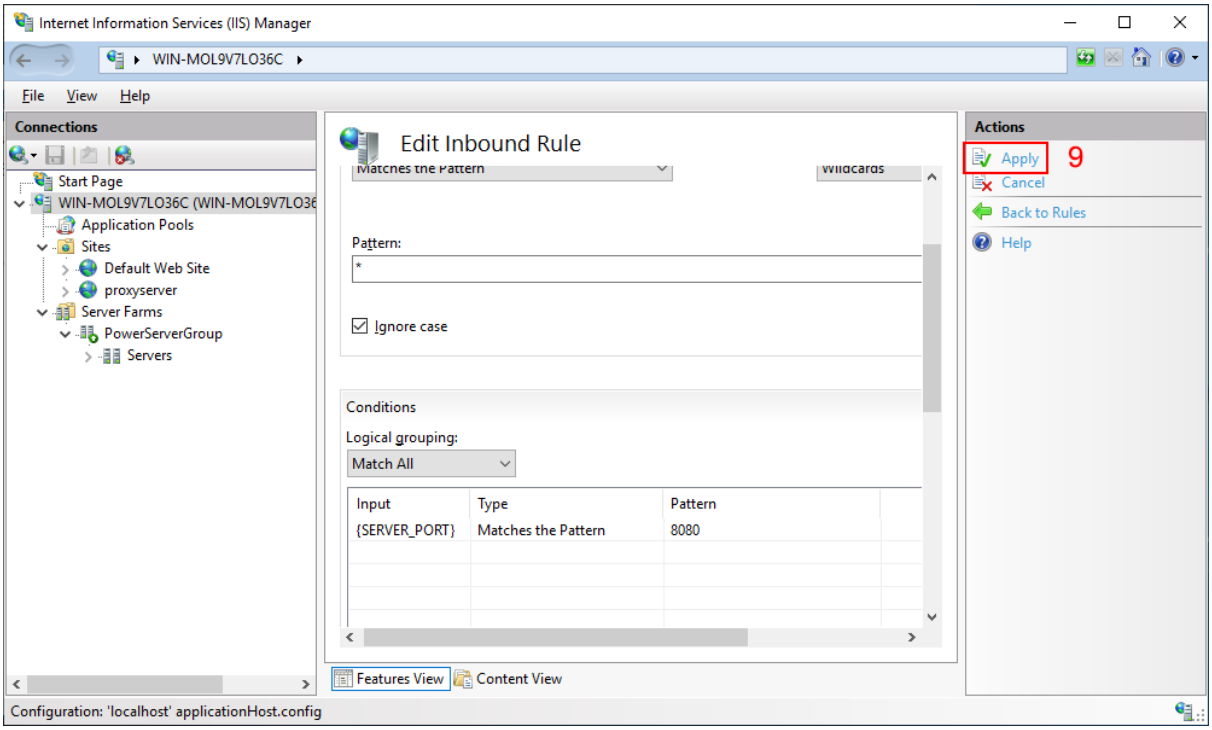
c. In the **Edit Inbound Rule** window, expand the **Conditions** block and then click **Add**. In the **Add Condition** dialog, input "{SERVER_PORT}" to the **Condition input** field and "8080" (port of "proxyserver" website) to the **Pattern** field, and click **OK**.

Figure 5.8:



d. Click **Apply** for the changes to take effect.

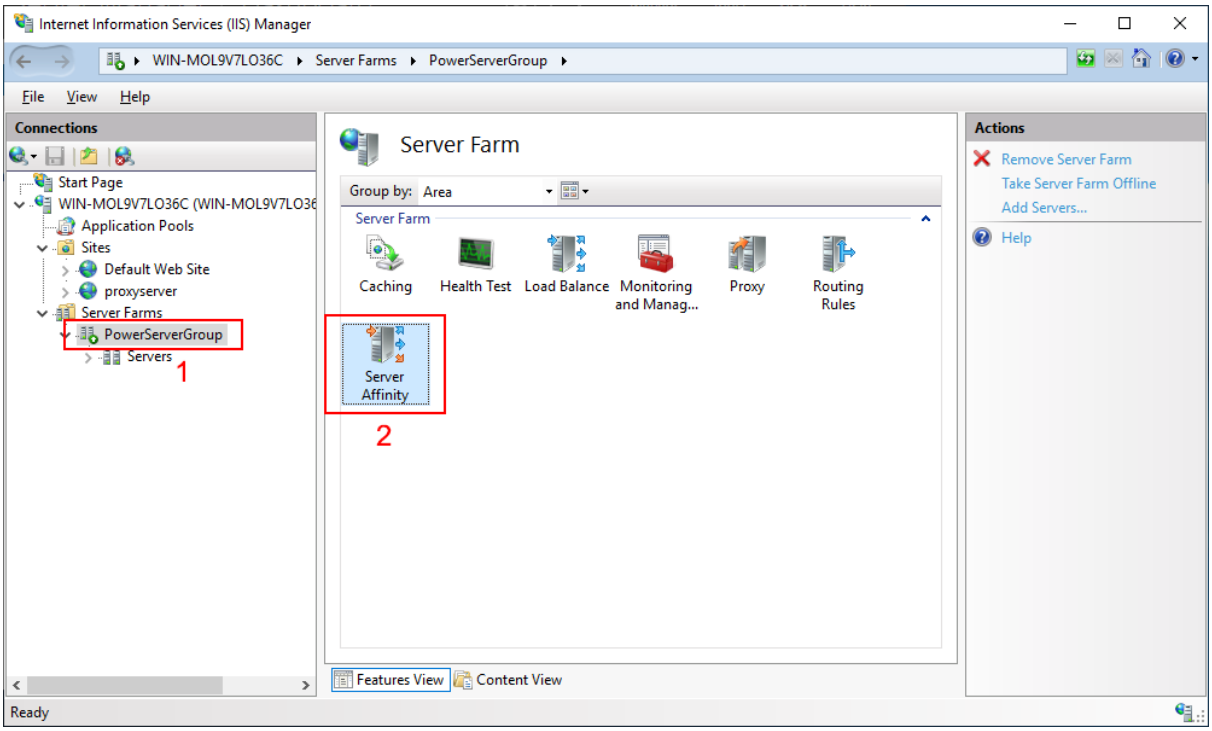
Figure 5.9:



6. Configure **Server Affinity** of the server farm to support sticky sessions.

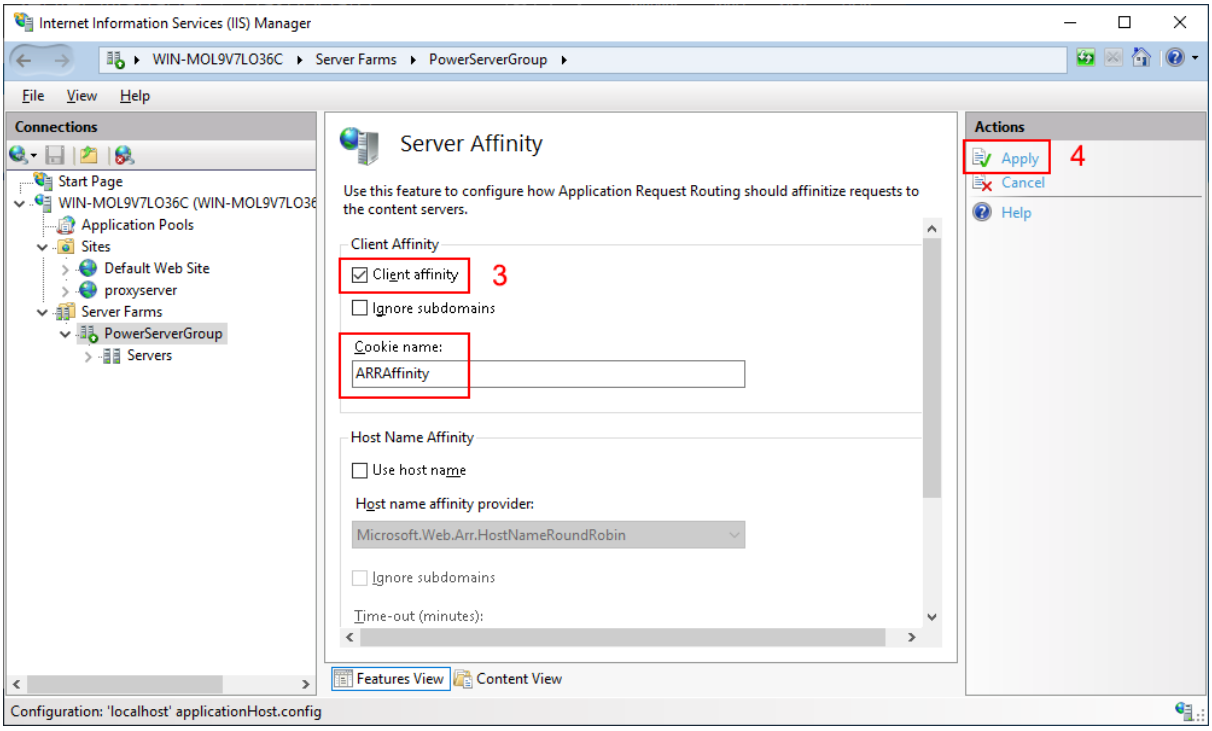
   a. Select the "PowerServerGroup" server farm in the **Connections** pane, and then double click **Server Affinity** in the features view to open the feature.

Figure 5.10:



b. Select the check box of **Client affinity**, keep the **Cookie name** as default "ARRAffinity", and then click Apply.

Figure 5.11:



5.4 Configuring Apache as a load balancer

This tutorial will walk you through configuring Apache as a load balancer to direct client requests to a group of PowerServer Web APIs. You will have to configure Apache as a load balancer and use the "Request Counting" load balancer scheduler algorithm and the cookie in order to support sticky sessions.

Step 1: Follow the sections below to install Apache 2.4 (The load balancing feature is available in Apache 2.2 or later).

- Setting up Apache on Windows > "Preparations" and "Installing Apache HTTP Server" sections
- Setting up Apache on Linux > "Preparations" and "Installing Apache HTTP Server" sections

Step 2: Configure Apache to direct requests to the PowerServer Web APIs group using the "Request Counting" load balancer scheduler algorithm and the cookie.

1. For Windows Apache, make sure the following lines are NOT commented out in the httpd.conf file.

   mod_proxy, mod_proxy_http, mod_proxy_balancer, mod_lbmethod_byrequests (the "Request Counting" algorithm), and mod_headers (stickyness cookie) must be enabled in order to have the load-balancing ability.

```
LoadModule headers_module modules/mod_headers.so
LoadModule status_module modules/mod_status.so
LoadModule slotmem_shm_module modules/mod_slotmem_shm.so
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

2. Add the following lines to the end of the httpd.conf file.

   The "Header" directive provides load balancing with stickyness using mod_headers.

   The "Max-Age" attribute specifies the number of seconds until the cookie expires. This value must be greater than the session timeout value (which is 3600 seconds by default).

   The "BalancerMember" directive specifies the URL of the server instance in the group.

   The "stickysession" attribute specifies the name of the cookie.

   For more information, refer to https://httpd.apache.org/docs/2.4/mod/mod_proxy_balancer.html.

```
Header add Set-Cookie "ROUTEID=.%{BALANCER_WORKER_ROUTE}e; Max-Age=3700; path=/"
 env=BALANCER_ROUTE_CHANGED

ProxyRequests Off
<Proxy balancer://mycluster>
BalancerMember https://172.16.100.34:6000 route=server1
BalancerMember https://172.16.100.35:6000 route=server2
BalancerMember https://172.16.100.36:6000 route=server3
ProxySet stickysession=ROUTEID
</Proxy>
ProxyPass / balancer://mycluster/
```

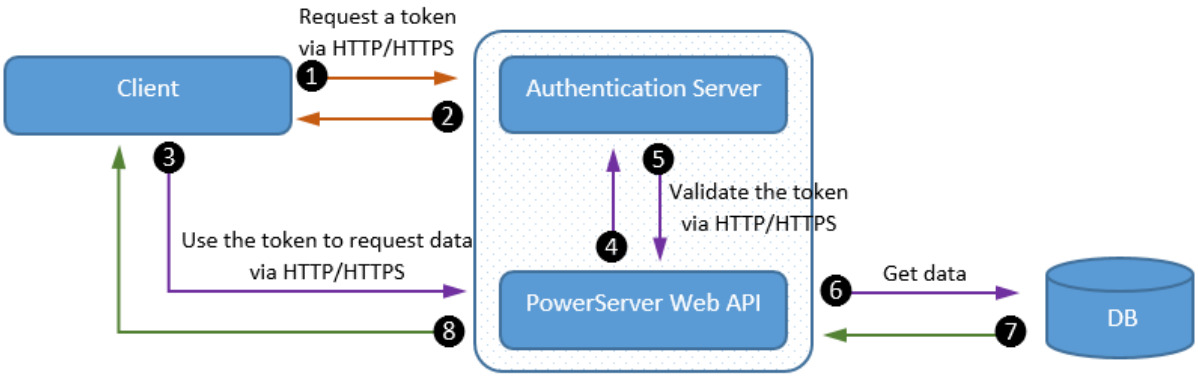
6 Tutorial 6: Authenticating your apps

6.1 Overview

The PowerServer Web APIs can include a built-in authentication server which can be used easily to authenticate the installable cloud apps. You can select which of the following authentication server to be built into the PowerServer Web APIs. And since the authentication server is built into the PowerServer Web APIs, it has the same URL as the PowerServer Web APIs and it runs automatically when the PowerServer Web APIs runs.

- **Use built-in JWT server**: Includes a built-in authentication server that supports JWT or bearer tokens. See Using JWT for more information.
- **Use built-in OAuth server**: Includes a built-in authentication server based on IdentityServer4 framework that works with the OAuth 2.0 authorization flows. See Using OAuth 2.0 for more information.
- **Use built-in AWS Cognito server**: Includes a built-in authentication server that works with the Amazon Cognito user pool. See Using Amazon Cognito for more information.
- **Use external auth service**: Includes templates that can be easily extended to support the other identity providers that work with the OAuth flows or JWT, such as Azure AD or Azure AD B2C. See Using other auth servers for more information.

Figure 6.1:



1. The client sends the user name and password (from the INI file or login window) to the authentication server.
2. The authentication server validates the user (against the DefaultUserStore.cs file, the authentication database, or the LDAP server); and if validation is successful, it authorizes and returns a token to the client.
3. The client sends a request that includes the token to the PowerServer Web API.
4. The PowerServer Web APIs validates the token with the authentication server; and if validation is successful, it gets data from the database.

The following tokens are supported:

- JSON Web Token (JWT) (**recommended**)
- Bearer token

For OAuth 2.0, the following authorization flows are recommended:

- Client Credentials
- Resource Owner Password

The PowerBuilder client application will implement the authentication process (such as getting a valid token, accessing data with the token etc.) using the PowerBuilder RestClient, OAuthClient, JsonParser, TokenRequest, and TokenResponse objects, and the Application.SetHttpRequestHeader function. See the code example in the following sections for more details. And it calls the Application.BeginSession function to create the user session in a manual way (instead of the automatic way) in order to include the token information. See the "Start session manually by code" section for more details.

6.2 Using JWT

6.2.1 Preparations

Before making changes to the PowerBuilder client app, let's follow the steps below to make sure 1) the PowerBuilder application can run successfully, 2) the app has been deployed as an installable cloud app successfully, and 3) the PowerServer C# solution (including the built-in JWT server) has been successfully generated.

In this tutorial, we will take Sales Demo as an example.

Step 1: Select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**.

Step 2: When the SalesDemo workspace is loaded in the PowerBuilder IDE, click the **Run** button in the PowerBuilder toolbar.

Step 3: When the application main window is opened, click the **Address** icon in the application ribbon bar and make sure data can be successfully retrieved.

Step 4: Create and configure a PowerServer project for the Sales Demo app (detailed instructions are provided in the Quick Start guide).

IMPORTANT: In the **Web APIs** tab, select **Use built-in JWT server** from the **Auth Template** list box.

Step 5: Deploy the application as an installable cloud app. The PowerServer C# solution is generated, but the installable cloud app cannot run yet because further settings and changes are required, as explained in the subsequent sections.

The PowerServer C# solution contains a built-in JWT server and the authentication class files as shown below.

- The built-in JWT server authenticates the user credential and returns a token. The built-in server is included in the **ServerAPIs** project; it runs automatically when the PowerServer Web APIs (the **ServerAPIs** project) runs.

- **DefaultUserStore.cs** defines two users by default. You can change this file to define more users. These users will be used by the built-in server to validate the users passed from the client.

  The users can also be defined and stored in the database. Refer to Validate username and password against a database for more information.

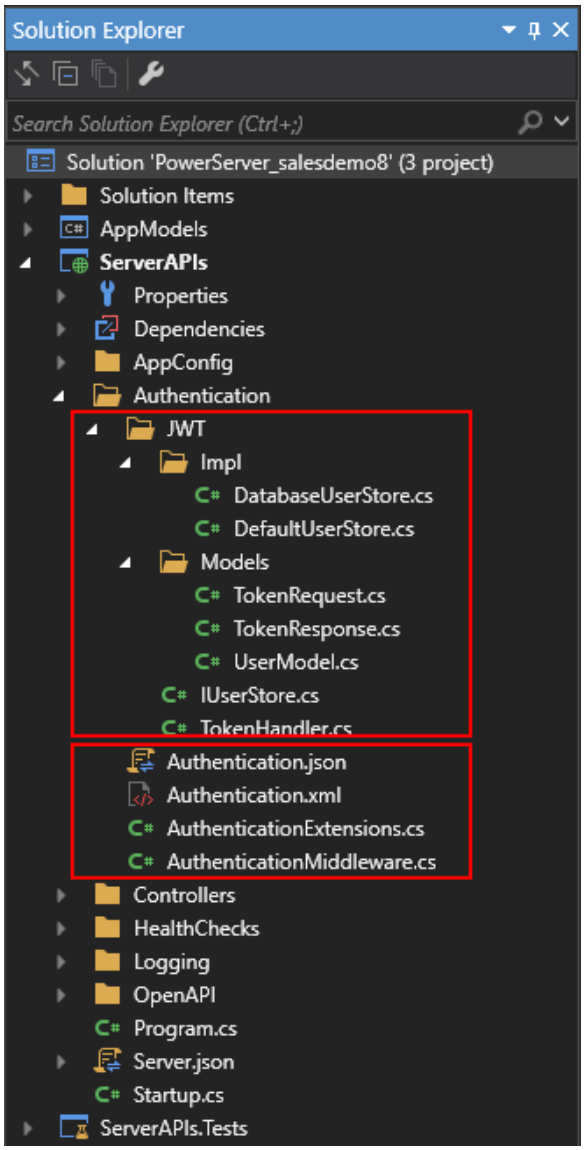
- **DatabaseUserStore.cs** can be added with scripts to connect with an authentication database where the users are defined and stored.

- The authentication class and configuration files will be used by the PowerServer Web APIs to validate the token passed from the client and, if validation is successful, data will be obtained from the database.

  - **Authentication.json** contains the settings for enabling the authentication feature ("PowerServer:EnableAuthentication") and specifying the JWT token information ("JwtSetting").

    The "PowerServer:EnableAuthentication" setting is set to **true** by default. Setting it to **false** will turn off the authentication feature. The "JwtSetting" block is used to specify the token information including the issuer, audience and security key.

Figure 6.2:



6.2.2 Modifying the PowerBuilder client app

6.2.2.1 Purpose

In this section, we will modify the PowerBuilder application source code and the PowerServer project settings to achieve the following results:

- Sends the user credentials and/or password to the JWT server and gets a token from the JWT server if authentication is successful.
- Uses the token to access data from the PowerServer Web API.
- Refreshes the token when necessary.

6.2.2.2 Add scripts

Step 1: Declare the following global variables.

```
//Token expiresin
```

```
Long gl_Expiresin
//Refresh token clockskew
Long gl_ClockSkew = 3
```

Step 2: Define a global function and name it **f_Authorization()**.

Select from menu **File** > **New**; in the **New** dialog, select the **PB Object** tab and then select **Function** and click **OK** to add a global function.

This global function uses the HTTP Post method to send the user credentials to the authentication server and then gets the token from the HTTP Authorization header.

Add scripts to the **f_Authorization()** function to implement the following scenario: When the application starts, the application uses the username and password from the login window to get the token, and when the token expires, the login window displays for the user to input the username and password again.

Note: The following scripts use the username and password from the INI file instead of from the login window. You can change the scripts to use the login window after you implement the login window and return the username and password to the **f_Authorization()** function.

```
//Integer f_Authorization() for password
//UserName & Password are passed from the login window
RestClient lrc_Client
String  ls_url, ls_UserName, ls_UserPass, ls_PostData, ls_Response, ls_expires_in
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn
JsonParser ljson_Parser

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//login window can be implemented to return username & password according to actual
 needs.
//Open(w_login)
//Return UserName & Password

ls_UserName = ProfileString("CloudSetting.ini", "users", "userName", "")
ls_UserPass = ProfileString("CloudSetting.ini", "users", "userPass", "")

If IsNull ( ls_UserName ) Or Len ( ls_UserName ) = 0 Then
 MessageBox( "Tips", "UserName is empty!" )
 Return li_rtn
End If
If IsNull ( ls_UserPass ) Or Len ( ls_UserPass ) = 0 Then
 MessageBox( "Tips", "Password is empty!" )
 Return li_rtn
End If

ls_PostData = '{"Username":"' + ls_UserName + '", "Password":"' + ls_UserPass +
 '"}'
lrc_Client = Create RestClient
lrc_Client.SetRequestHeader("Content-Type","application/json")
li_Return = lrc_Client.GetJWTToken( ls_Url, ls_PostData, ls_Response )
If li_Return = 1 and Pos ( ls_Response, "access_token" ) > 0 Then
 ljson_Parser = Create JsonParser
 ljson_Parser.LoadString(ls_Response)
 ls_TokenType = ljson_Parser.GetItemString("/token_type")
 ls_AccessToken = ljson_Parser.GetItemString("/access_token")
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
```

```
 //Set Global Variables
 gl_Expiresin = Long (ljson_Parser.GetItemNumber("/expires_in"))

 li_rtn = 1
Else
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) )
End If

If IsValid ( ljson_Parser ) Then DesTroy ( ljson_Parser )
If IsValid ( lrc_Client ) Then DesTroy ( lrc_Client )

Return li_rtn
```

Step 3: Insert a timing object (**timing_1**) to the application and add the following scripts to the **Timer** event of **timing_1**.

1) Open the application object and then select from menu **Insert** > **Object** > **Timing** to add a timing object to the application.

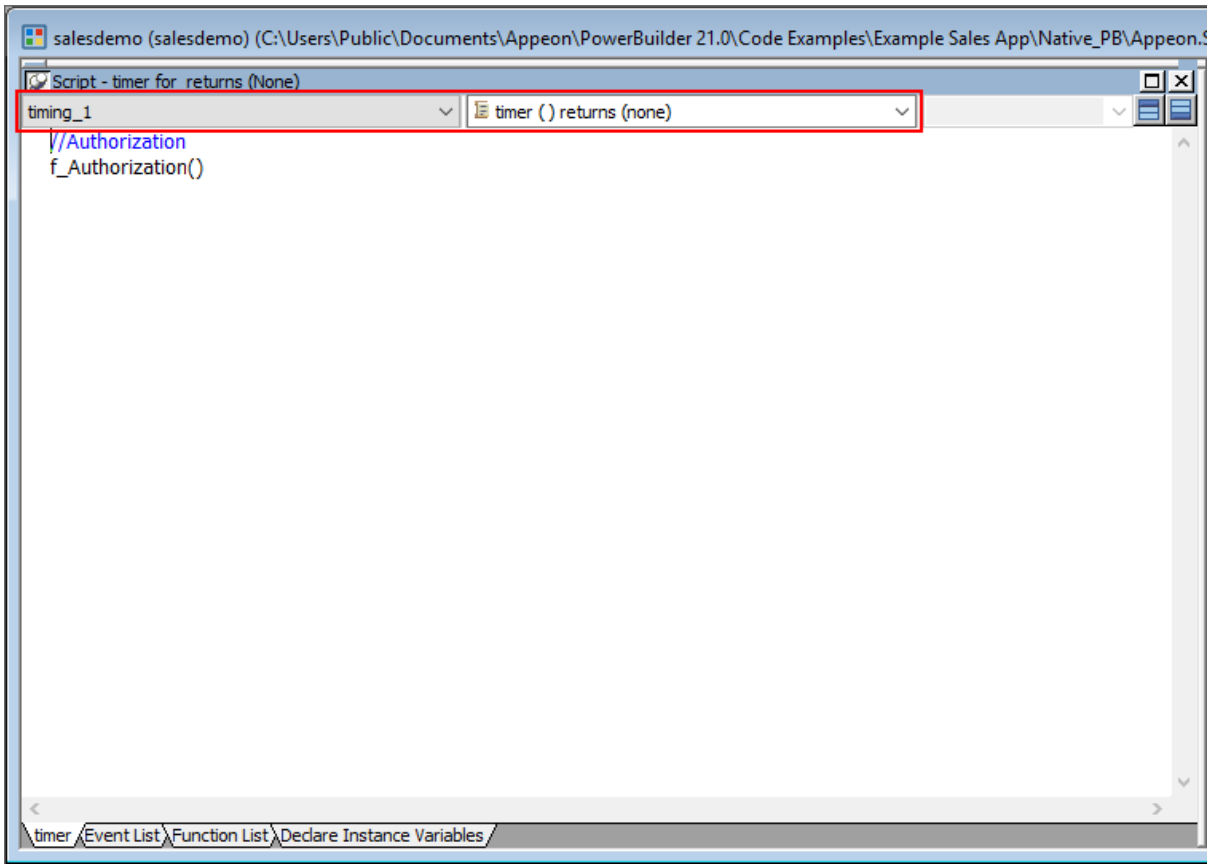
2) Add the following scripts to the **Timer** event of **timing_1**.

```
//Authenticates the user
f_Authorization()
```

When displayed in the source editor, the **Timer** event looks like this:

```
event timer;//Authenticates the user
f_Authorization()
end event
```

Figure 6.3:

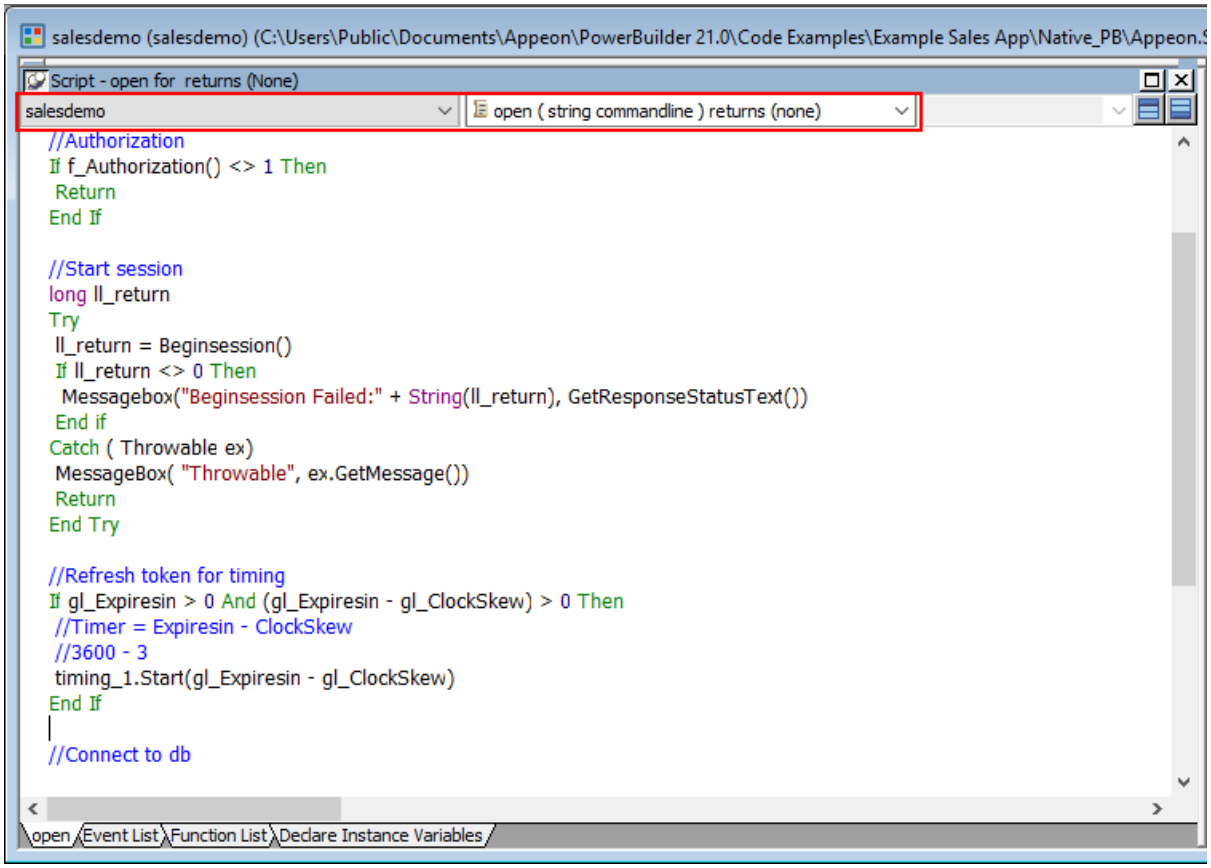


Step 4: Add the following scripts to the application **Open** event.

Place the scripts before the database connection is established. The scripts get the token from the JWT server and then start the user session (using the **BeginSession** function) to include the token information in the session.

```
//Authenticates the user and returns the token
If f_Authorization() <> 1 Then
 Return
End If

//Starts the session
long ll_return
Try
 ll_return = Beginsession()
 If ll_return <> 0 Then
  Messagebox("Beginsession Failed:" + String(ll_return),
 GetHttpResponseStatusText())
 End if
Catch ( Throwable ex)
 MessageBox( "Throwable", ex.GetMessage())
 Return
End Try

//Refreshes the token for timing
If gl_Expiresin > 0 And (gl_Expiresin - gl_ClockSkew) > 0 Then
 //Timer = Expiresin - ClockSkew
 //7200 - 3
 timing_1.Start(gl_Expiresin - gl_ClockSkew)
End If

//Connects to db
```

Figure 6.4:

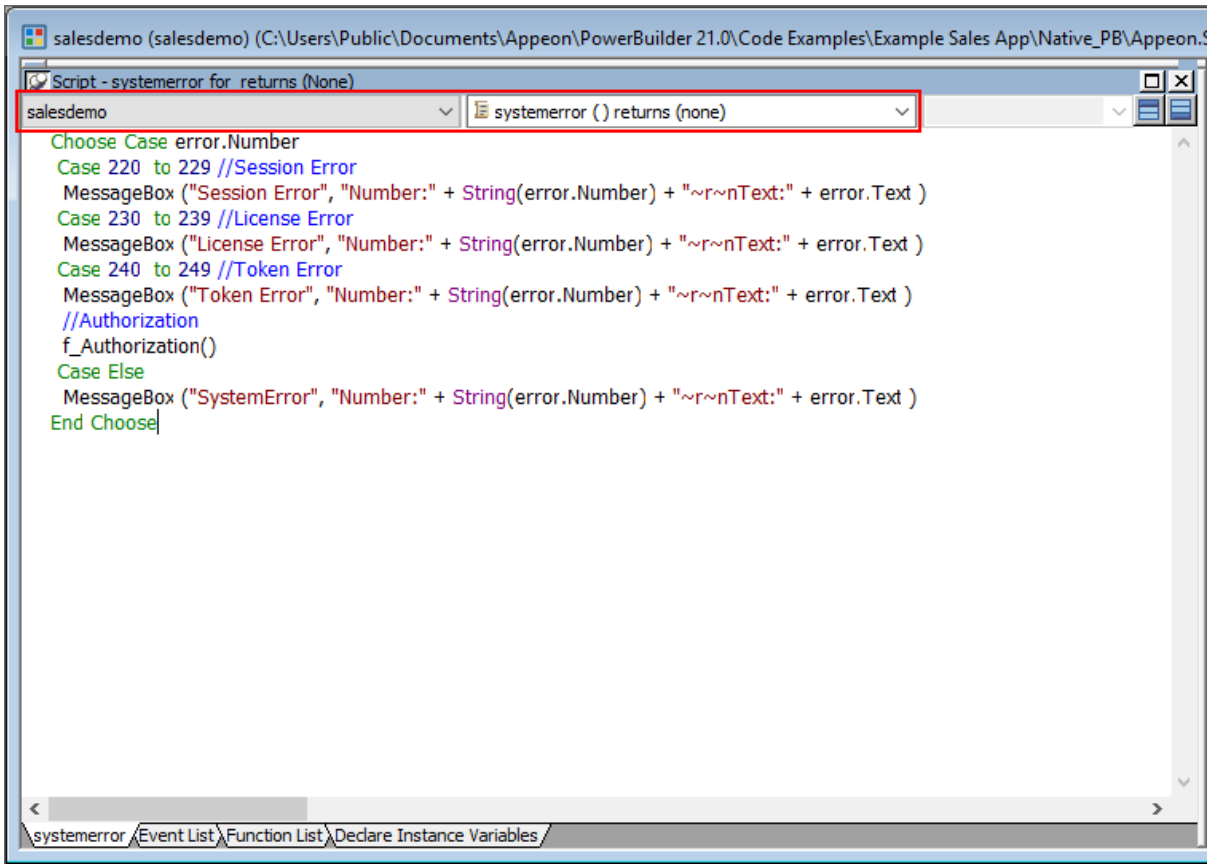


Step 5: Add the following scripts to the **SystemError** event.

The scripts will trigger the **SystemError** event when the session or license encounters an error; and if the token is invalid or expires, the scripts will call the **f_Authorization** function to get the token again.

```
Choose Case error.Number
 Case 220  to 229 //Session Error
  MessageBox ("Session Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 230  to 239 //License Error
  MessageBox ("License Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 240  to 249 //Token Error
  MessageBox ("Token Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
  //Authorization
  f_Authorization()
 Case Else
  MessageBox ("SystemError", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
End Choose
```

Figure 6.5:



6.2.2.3 Add an INI file

Create an INI file in the same location as the PBT file and name it **CloudSetting.ini**.

Specify the URL for requesting the token from the JWT server in the **CloudSetting.ini** file. Notice that **TokenURL** points to the "/connect/token" API of the built-in JWT server, and

the JWT server root URL (for example, https://localhost:5000/) is the same as the URL of PowerServer Web API. If you change the PowerServer Web API URL, change the root URL here accordingly.

```
[Setup]
TokenURL=https://localhost:5000/connect/token
```

To get the username and password from the INI file (instead of from the login window), you need to add the following section to the **CloudSetting.ini** file and set the user name and password accordingly.

```
[users]
userName=alice
userPass=alice
```

6.2.2.4 Start session manually by code

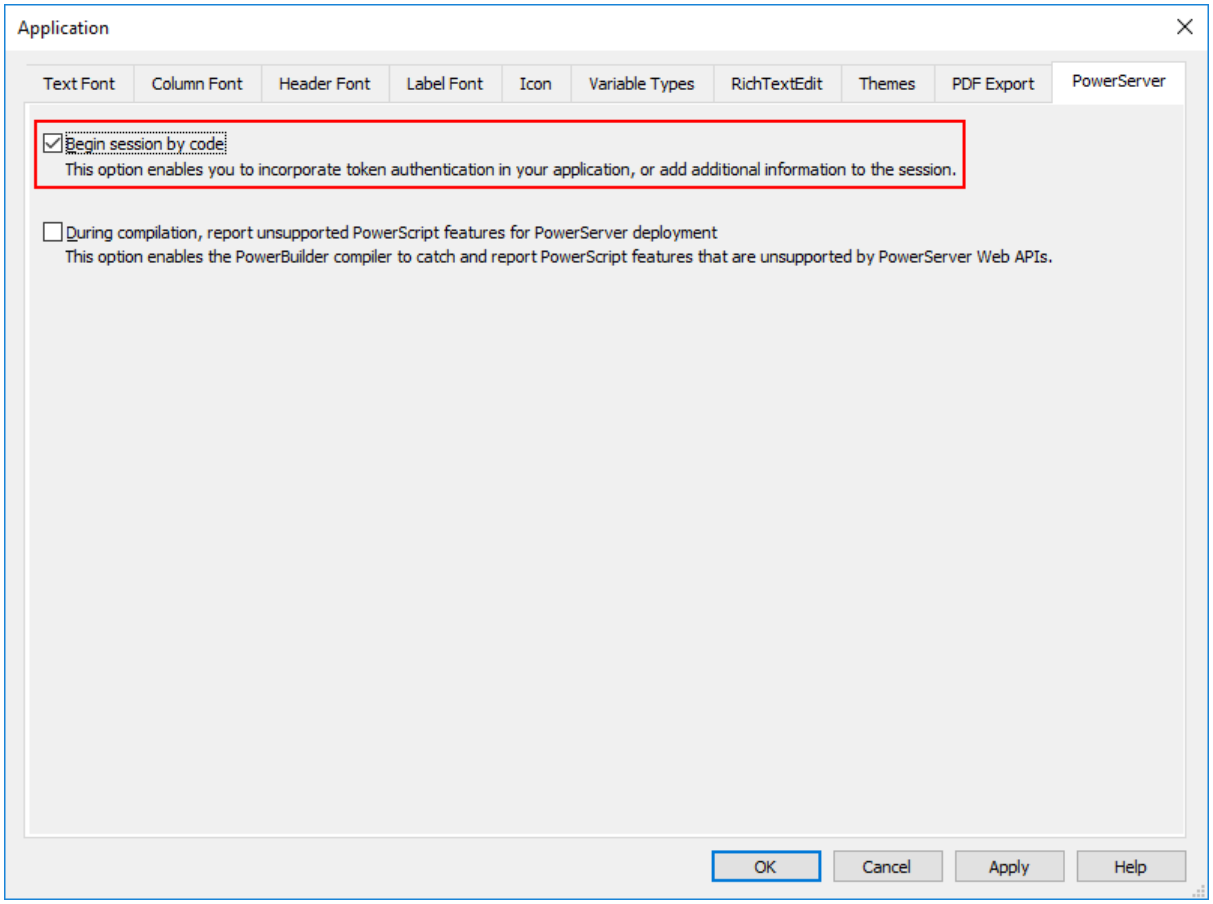
By default, the user session is automatically created when the application starts; and the session includes no token. For the session to include the token, the session must be started manually by code instead of automatically.

To start the session manually by code,

Step 1: Enable "**Begin session by code**" in the PowerBuilder IDE. (Steps: Open the application object painter, click **Additional Properties** in the application's **Properties** dialog; in the **Application** dialog, select the **PowerServer** tab and then select the **Begin session by code** option and click **Apply**.)

After this option is enabled, when the **BeginSession** function in the application **Open** event is called, it will create a session that includes the token information (See scripts in step 4 in "Add scripts").

Figure 6.6:

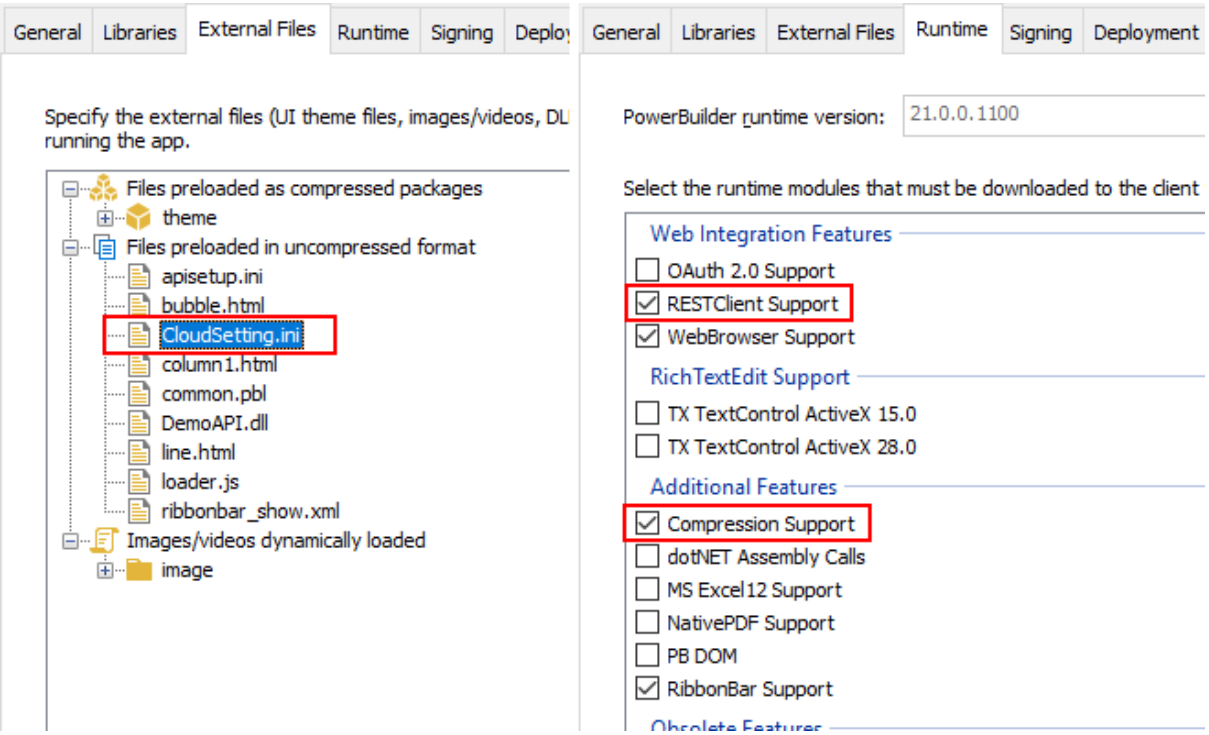


6.2.2.5 Modify and re-deploy the PowerServer project

Step 1: Add the INI file **CloudSetting.ini** to the **Files preloaded in uncompressed format** section under the **External Files** tab.

Step 2: Select **RESTClient Support** and **Compression Support** under the **Runtime** tab.

Figure 6.7:



Step 3: Double check the URL of the PowerServer Web APIs in the **Web APIs** tab. Make sure the port number is not occupied by any other program.

Tip: You can execute the command "netstat -ano | findstr *portnumber*" to check if the port number is occupied by any other program.

The built-in JWT server will run at the same URL as the PowerServer Web API. If the PowerServer Web API URL is changed, change the JWT server root URL accordingly in the INI file.

Step 4: Double check that **Use built-in JWT server** is selected from the **Auth Template** list box in the **Web APIs** tab.

Step 5: Save the changes and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) so that the above settings can take effect in the installable cloud app.

6.2.3 Appendix

6.2.3.1 Validate username and password against a database

When the username and password are passed from the application to the built-in JWT server, the JWT server will by default authenticate them against users defined in the **DefaultUserStore.cs** file. If users are defined in an authentication database instead of **DefaultUserStore.cs**, you can choose to

- Have JWT server to connect to the authentication database and authenticate the user every time when a token is requested (see this section for details); or
- Populate users from the database to the user list of the DefaultUserStore.cs file, and the user list will be cached and used to authenticate the user when a token is requested.

The benefit of populating and caching the user list is the JWT server does not need to connect to the authentication database every time when a user is authenticated, but the downside is if the users in the authentication database are updated, the PowerServer Web APIs needs to be restarted to refresh the user list.

This section will show you how to populate and cache the user list of the DefaultUserStore.cs file.

Step 1: Open the **DatabaseUserStore.cs** file and add the following scripts. Suppose a SQL Server database will be connected. Modify the database connection string according to your environment.

```
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using IdentityModel;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Logging;
using SnapObjects.Data;
using SnapObjects.Data.SqlServer;

namespace ServerAPIs
{
    public class DatabaseUserStore : IUserStore
    {
        private readonly ILogger _logger;
        private readonly List<UserModel> _users;

        public DatabaseUserStore(ILogger<DatabaseUserStore> logger)
        {
            _logger = logger;
            _users = new List<UserModel>();
            string Constr = @"Data Source=172.16.1.10,1433;Initial
 Catalog=pb_cloud;Integrated Security=False;User
 ID=sa;Password=1234;Pooling=True;Min Pool Size=0;Max Pool
 Size=100;MultipleActiveResultSets=False;Encrypt=False;TrustServerCertificate=False;ApplicationIn
            SqlServerDataContext _context = new SqlServerDataContext(new
 SqlConnection(Constr));
            string sql = "select username,password from users where isValid = 1";
            var users = _context.SqlExecutor.Select<DynamicModel>(sql);
            foreach (var u in users)
            {
                _users.Add(new UserModel
                {
                    Username = u.GetValue<string>(0),
                    Password = u.GetValue<string>(1),
                    Claims = new[]
                    {
                        new Claim(JwtClaimTypes.Name, u.GetValue<string>(0)),
                        new Claim(JwtClaimTypes.Scope, "serverapi"),  //this script
 is added because scope is enabled by default
                    },
                });
            };

        }

        public UserModel ValidateCredentials(string username, string password)
        {
            var user = _users.FirstOrDefault(x => x.Username == username &&
 x.Password == password);
            if (user != null)
```

```
            {
                _logger.LogInformation($"User <{username}> logged in.");

                return user;
            }
            else
            {
                _logger.LogError($"Invalid login attempt.");

                return default;
            }
        }
    }
}
```

To connect with a database type different from SQL Server, add the following namespace accordingly.

```
using SnapObjects.Data.MySql;
```

```
using SnapObjects.Data.Oracle;
```

```
using SnapObjects.Data.PostgreSql;
```

```
using SnapObjects.Data.Odbc;
```

Step 2: Open the **AuthenticationExtensions.cs** file and modify the script to inject the **DatabaseUserStore** class instead of the **DefaultUserStore** class.

```
//services.AddSingleton<IUserStore, DefaultUserStore>();
services.AddSingleton<IUserStore, DatabaseUserStore>();
```

6.3 Using OAuth 2.0

6.3.1 Preparations

Before making changes to the PowerBuilder client app, let's follow the steps below to make sure 1) the PowerBuilder application can run successfully, 2) the app has been deployed as an installable cloud app successfully, and 3) the PowerServer C# solution (including the built-in OAuth server) has been successfully generated.

In this tutorial, we will take Sales Demo as an example.

Step 1: Select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**.

Step 2: When the SalesDemo workspace is loaded in the PowerBuilder IDE, click the **Run** button in the PowerBuilder toolbar.

Step 3: When the application main window is opened, click the **Address** icon in the application ribbon bar and make sure data can be successfully retrieved.

Step 4: Create and configure a PowerServer project for the Sales Demo app (detailed instructions are provided in the Quick Start guide).

IMPORTANT: In the **Web APIs** tab, select **Use built-in OAuth server** from the **Auth Template** list box.

Step 5: Deploy the application as an installable cloud app. The PowerServer C# solution is generated, but the installable cloud app cannot run yet because further settings and changes are required, as explained in the subsequent sections.

The PowerServer C# solution contains a built-in OAuth server and the authentication class files as shown below.

- The built-in OAuth server uses the **IdentityServer4** framework. It is included in the **ServerAPIs** project; it will run automatically when the PowerServer Web APIs (the **ServerAPIs** project) runs. You can use another OAuth server (such as Google OAuth 2.0 Authorization Server) instead of using the built-in server. In this tutorial, we will use the built-in server to authenticate the user credentials and return the token.

  - **DefaultConfig.cs** defines two OAuth authorization flows: client credentials and resource owner password; and in each flow defines client ID, client name, grant type, client secret, scope etc. If you want to define different credentials, you can change this file accordingly.

  - **DefaultUserStore.cs** defines two users by default. You can change this file to define more users.

    The users can also be defined and stored in the database or LDAP server. Refer to Validate username and password against a database or Validate username and password against an LDAP server for more information.

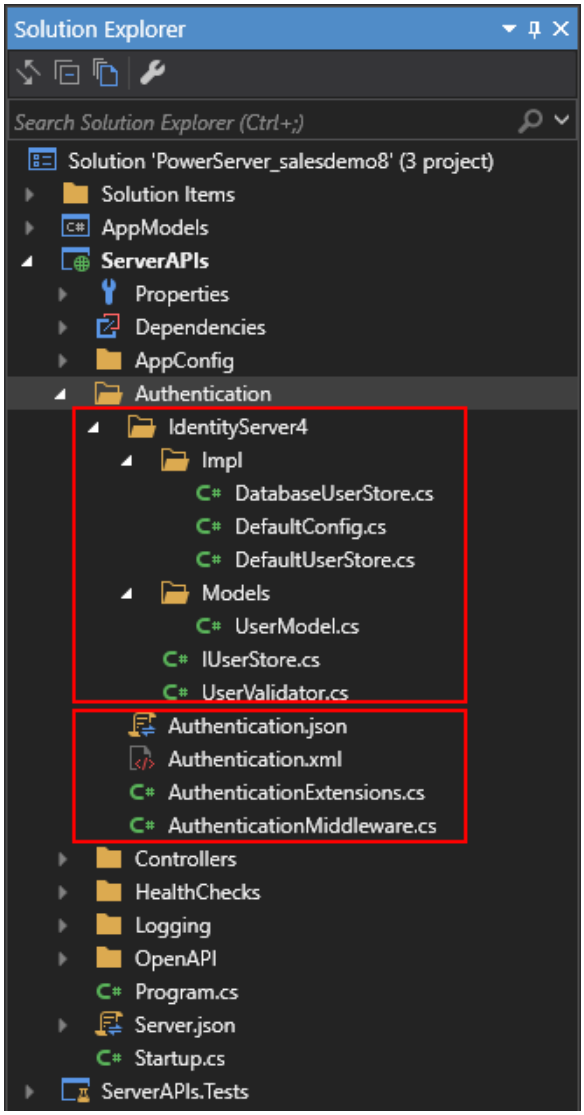
  - **DatabaseUserStore.cs** can be added with scripts to connect with an authentication database where the users are defined and stored.

- The authentication class and configuration files will be used by the PowerServer Web APIs to validate the token (passed from the client) against the OAuth server and, if validation is successful, data will be obtained from the database.

  - **Authentication.json** contains the settings for enabling the authentication feature ("PowerServer:EnableAuthentication") and specifying the URL of the OAuth server ("Authority").

    The "PowerServer:EnableAuthentication" setting is set to **true** by default. Setting it to **false** will turn off the authentication feature. The "Authority" setting is set to the OAuth server URL which is the same as Web API URL by default, as the built-in OAuth server resides in the PowerServer Web API. If you set up your own OAuth server, change the URL in this file accordingly.

Figure 6.8:



6.3.2 Modifying the PowerBuilder client app

6.3.2.1 Purpose

In this section, we will modify the PowerBuilder application source code and the PowerServer project settings to achieve the following results:

- Sends the user credentials and/or password to the OAuth server and gets a token from the OAuth server if authentication is successful.
- Uses the token to access data from the PowerServer Web API.
- Refreshes the token when necessary.

6.3.2.2 Add scripts

Step 1: Declare the following global variables.

```
//Token expiresin
```

```
Long gl_Expiresin
//Refresh token clockskew
Long gl_ClockSkew = 3
```

Step 2: Define a global function and name it **f_Authorization()**.

Select from menu **File** > **New**; in the **New** dialog, select the **PB Object** tab and then select **Function** and click **OK** to add a global function.

This global function uses the HTTP Post method to send the user credentials to the authentication server and then gets the token from the HTTP Authorization header.

Add scripts to the **f_Authorization()** function according to the following scenarios:

- Scenario 1: Supports Client Credentials (GrantType="client_credentials") and gets the client ID and secret from the application.
- Scenario 2: Supports Resource Owner Password (GrantType="password") and gets the username and password from a login window.
- Scenario 3: Supports Resource Owner Password (GrantType="password") and gets the username and password from the INI file.

Scripts for scenario 1:

Supports Client Credentials (GrantType="client_credentials") and gets the client ID and secret from the application.

When the application starts, the application uses the client ID and secret stored in the application to get the token from the OAuth server, and when the token expires, it automatically refreshes the token.

```
//integer f_Authorization() for client_credentials
//The URL for requesting token is specified in the INI file
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String ls_url, ls_UserName, ls_UserPass
String ls_TokenType, ls_AccessToken
String ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "client"
ltr_Request.clientsecret = "511536EF-F270-4058-80CA-1C89C192F69A"
ltr_Request.scope = "serverapi"
ltr_Request.granttype = "client_credentials"

loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
```

```
 //Set Global Variables
 gl_Expiresin = ltr_Response.getexpiresin( )
 li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Scripts for scenario 2:

Supports Resource Owner Password (GrantType="password") and gets the username and password from a login window.

When the application starts, the client ID and secret stored in the application as well as the username and password from the login window will be sent to the OAuth server to get the token, and when the token expires, the login window displays for the user to input the username and password again.

The username and password will be passed to the OAuth server and validated against the DefaultUserStore.cs file (to validate against a database or LDAP server rather than DefaultUserStore.cs, refer to Validate username and password against a database or Validate username and password against an LDAP server for more information).

The following scripts will work only after you implement a login window and return the username and password to the **f_Authorization()** function.

```
//Integer f_Authorization() for password from login window
//The URL for requesting token is specified in the INI file
//username & password are passed from the login window
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String ls_url, ls_UserName, ls_UserPass
String ls_TokenType, ls_AccessToken
String ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "ro.client"
ltr_Request.clientsecret = "08692CED-944D-4DA9-BFEF-0FE503C203AC"
ltr_Request.scope = "serverapi"
ltr_Request.granttype = "password"

//login window can be implemented to return username & password according to actual
 needs
//Open(w_login)
//Return UserName & Password

If IsNull ( ls_UserName ) Or Len ( ls_UserName ) = 0 Then
 MessageBox( "Tips", "UserName is empty!" )
 Return li_rtn
End If
```

```
If IsNull ( ls_UserPass ) Or Len ( ls_UserPass ) = 0 Then
 MessageBox( "Tips", "Password is empty!" )
 Return li_rtn
End If

ltr_Request.UserName = ls_UserName
ltr_Request.Password = ls_UserPass

loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
 //Set Global Variables
 gl_Expiresin = ltr_Response.getexpiresin( )

 li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Scripts for scenario 3:

Supports Resource Owner Password (GrantType="password") and gets the username and password from the INI file.

When the application starts, the client ID and secret stored in the application as well as the username and password from the INI file will be sent to the OAuth server to get the token, and when the token expires, it automatically refreshes the token.

The username and password will be passed to the OAuth server and validated against the DefaultUserStore.cs file (to validate against a database or LDAP server rather than DefaultUserStore.cs, refer to Validate username and password against a database or Validate username and password against an LDAP server for more information).

```
//Integer f_Authorization() for password from INI file
//The URL for requesting token is specified in the INI file
//username & password are passed from the INI file
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String  ls_url, ls_UserName, ls_UserPass
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer  li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "YourClientIdThatCanOnlyRead"
ltr_Request.clientsecret = "yoursecret1"
```

```
ltr_Request.scope = "scope.readaccess"
ltr_Request.granttype = "password"

//From CloudSetting.ini
ls_UserName = ProfileString("CloudSetting.ini", "users", "userName", "")
ls_UserPass = ProfileString("CloudSetting.ini", "users", "userPass", "")
If IsNull ( ls_UserName ) Or Len ( ls_UserName ) = 0 Then
 MessageBox( "Tips", "UserName is empty!" )
 Return li_rtn
End If
If IsNull ( ls_UserPass ) Or Len ( ls_UserPass ) = 0 Then
 MessageBox( "Tips", "Password is empty!" )
 Return li_rtn
End If
ltr_Request.UserName = ls_UserName
ltr_Request.Password = ls_UserPass


loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application sets the authorization header
 Getapplication().SetHttpRequestHeader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
 //Set the global variables
 gl_Expiresin = ltr_Response.getexpiresin( )

 li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Failed", "Return: " + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Step 3: Insert a timing object (**timing_1**) to the application and add the following scripts to the **Timer** event of **timing_1**.

1) Open the application object and then select from menu **Insert** > **Object** > **Timing** to add a timing object to the application.

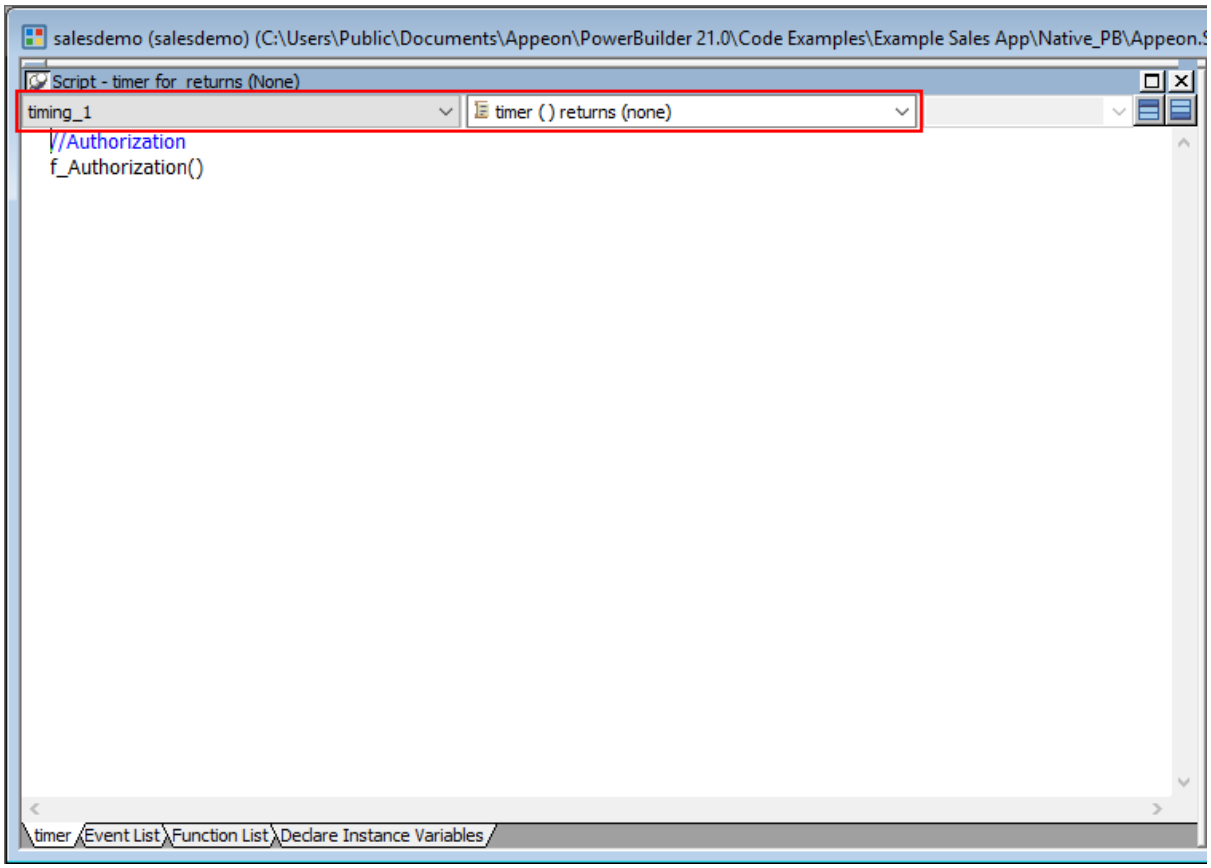
2) Add the following scripts to the **Timer** event of **timing_1**.

```
//Authenticates the user
f_Authorization()
```

When displayed in the source editor, the **Timer** event looks like this:

```
event timer;//Authenticates the user
f_Authorization()
end event
```

Figure 6.9:



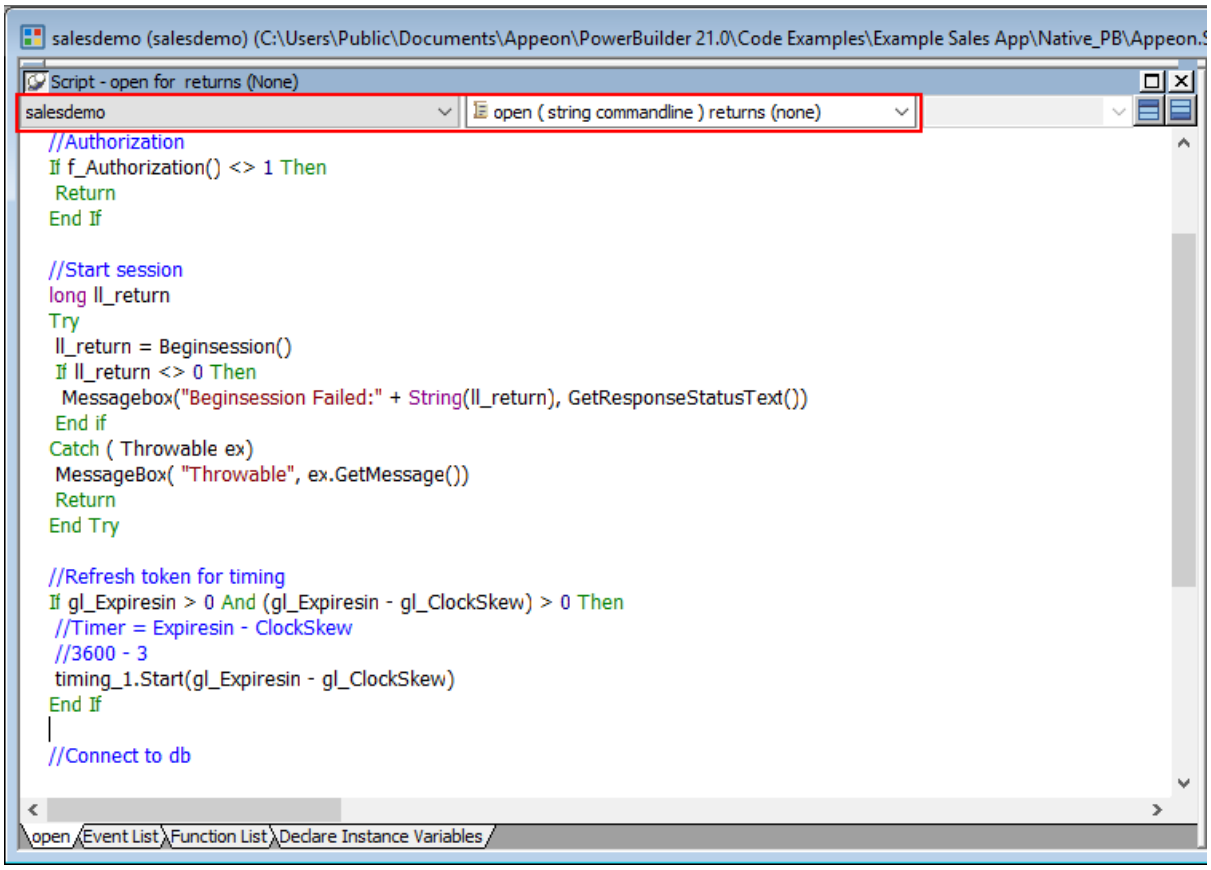
Step 4: Add the following scripts to the application **Open** event.

Place the scripts before the database connection is established. The scripts get the token from the OAuth server and then start the user session (using the **BeginSession** function) to include the token information in the session.

```
//Authenticates the user and returns the token
If f_Authorization() <> 1 Then
 Return
End If

//Starts the session
long ll_return
Try
 ll_return = Beginsession()
 If ll_return <> 0 Then
  Messagebox("Beginsession Failed:" + String(ll_return),
 GetHttpResponseStatusText())
 End if
Catch ( Throwable ex)
 MessageBox( "Throwable", ex.GetMessage())
 Return
End Try

//Refreshes the token for timing
If gl_Expiresin > 0 And (gl_Expiresin - gl_ClockSkew) > 0 Then
 //Timer = Expiresin - ClockSkew
 //3600 - 3
 timing_1.Start(gl_Expiresin - gl_ClockSkew)
End If
```

```
//Connects to db
```

Figure 6.10:

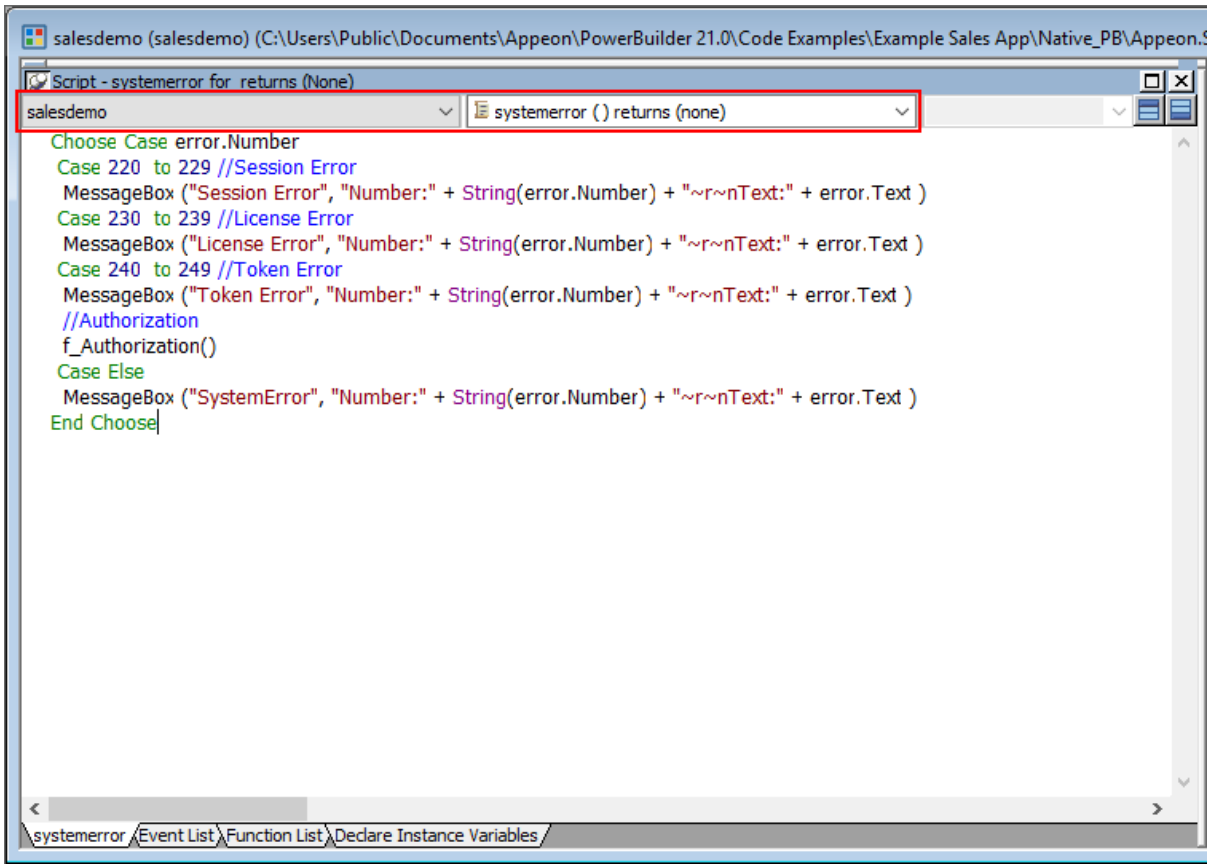


Step 5: Add the following scripts to the **SystemError** event.

The scripts will trigger the **SystemError** event when the session or license encounters an error; and if the token is invalid or expires, the scripts will call the **f_Authorization** function to get the token again.

```
Choose Case error.Number
 Case 220  to 229 //Session Error
  MessageBox ("Session Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 230  to 239 //License Error
  MessageBox ("License Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 240  to 249 //Token Error
  MessageBox ("Token Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
  //Authorization
  f_Authorization()
 Case Else
  MessageBox ("SystemError", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
End Choose
```

Figure 6.11:



6.3.2.3 Add an INI file

Create an INI file in the same location as the PBT file and name it **CloudSetting.ini**.

Specify the URL for requesting the token from the OAuth server in the **CloudSetting.ini** file. Notice that **TokenURL** points to the "/connect/token" API of the built-in OAuth server, and the OAuth server root URL (for example, https://localhost:5000/) is the same as the URL of PowerServer Web API. If you change the PowerServer Web API URL, change the root URL here accordingly.

```
[Setup]
TokenURL=https://localhost:5000/connect/token
```

To support "scenario 3" which supports Resource Owner Password (GrantType="password") and gets the username and password from the INI file, you need to add the following section to the **CloudSetting.ini** file and set the user name and password accordingly.

```
[users]
userName=alice
userPass=alice
```

6.3.2.4 Start session manually by code

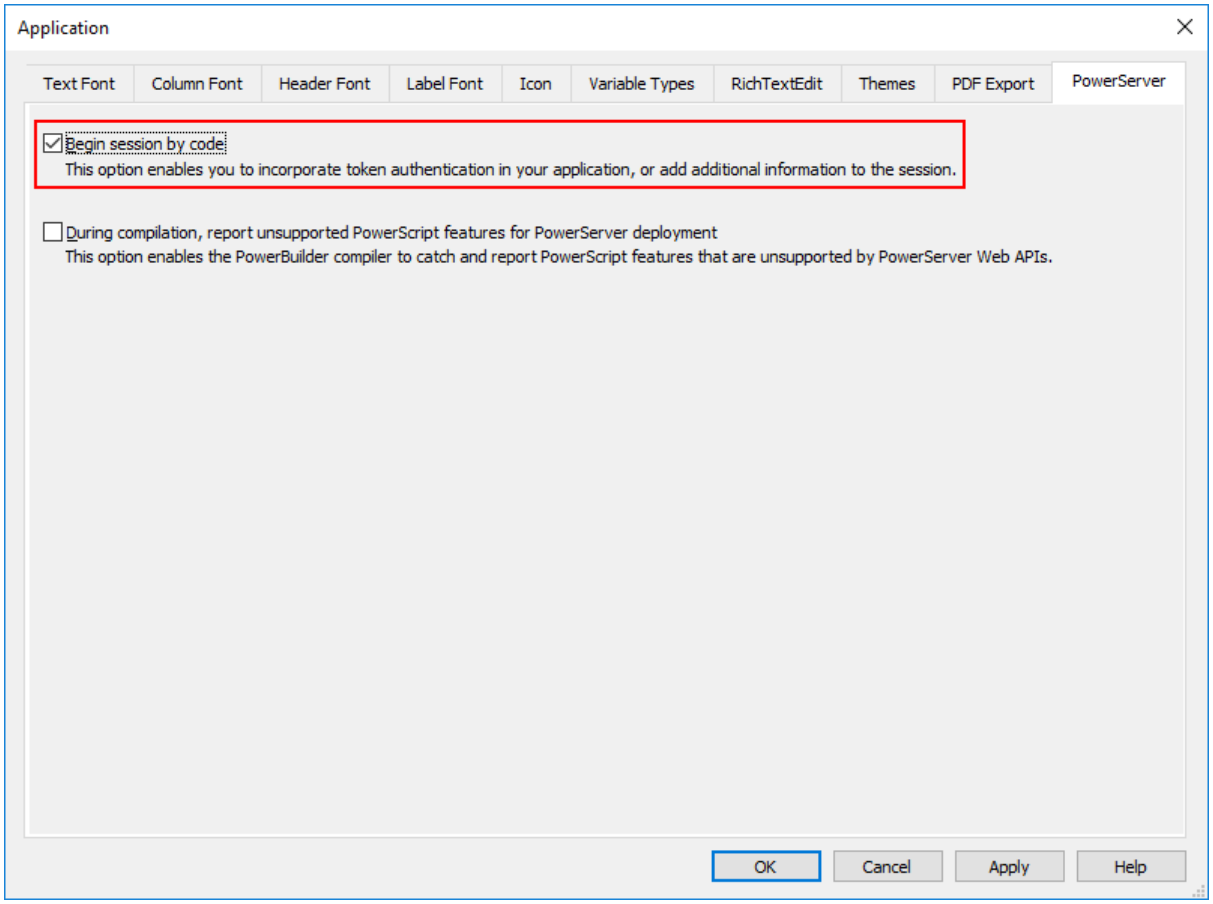
By default, the user session is automatically created when the application starts; and the session includes no token. For the session to include the token, the session must be started manually by code instead of automatically.

To start the session manually by code,

Step 1: Enable "**Begin session by code**" in the PowerBuilder IDE. (Steps: Open the application object painter, click **Additional Properties** in the application's **Properties** dialog; in the **Application** dialog, select the **PowerServer** tab and then select the **Begin session by code** option and click **Apply**.)

After this option is enabled, when the **BeginSession** function in the application **Open** event is called, it will create a session that includes the token information (See scripts in step 4 in "Add scripts").

Figure 6.12:

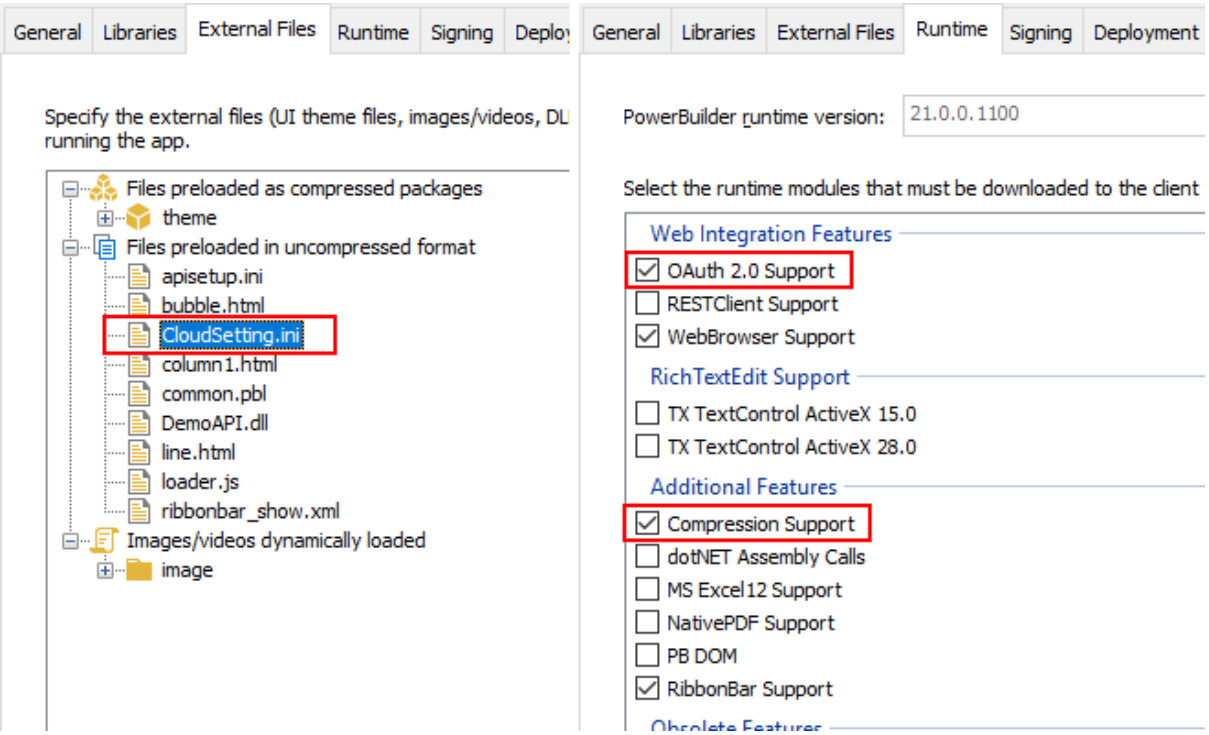


6.3.2.5 Modify and re-deploy the PowerServer project

Step 1: Add the INI file **CloudSetting.ini** to the **Files preloaded in uncompressed format** section under the **External Files** tab.

Step 2: Select **OAuth 2.0 Support** and **Compression Support** under the **Runtime** tab.

Figure 6.13:



Step 3: Double check the URL of the PowerServer Web APIs in the **Web APIs** tab. Make sure the port number is not occupied by any other program.

Tip: You can execute the command "netstat -ano | findstr *portnumber*" to check if the port number is occupied by any other program.

The built-in OAuth server will run at the same URL as the PowerServer Web APIs. If the PowerServer Web API URL is changed, change the OAuth server root URL accordingly in the INI file.

Step 4: Double check that **Use built-in OAuth server** is selected from the **Auth Template** list box in the **Web APIs** tab.

Step 5: Save the changes and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) so that the above settings can take effect in the installable cloud app.

6.3.3 Appendix

6.3.3.1 Validate username and password against a database

When the username and password are passed from the application to the built-in OAuth server, the OAuth server will by default authenticate them against the users predefined in DefaultUserStore.cs. For security concern, you might want the OAuth server to authenticate against the users stored in a database instead of DefaultUserStore.cs.

To do that, you will need to modify the **UserValidator.cs** file so that every time when a token is requested, the OAuth server will connect to the database and authenticate the user. (Another option is to populate and cache users from the database to the user list of the DefaultUserStore.cs file. See this section for details.)

Step 1: Add namespaces. Suppose a SQL Server database will be connected. The following namespaces need to be added.

```
using System;
using Microsoft.Data.SqlClient;
using SnapObjects.Data;
using SnapObjects.Data.SqlServer;
```

To connect with a database type different from SQL Server, add the following namespace accordingly.

```
using SnapObjects.Data.MySql;
```

```
using SnapObjects.Data.Oracle;
```

```
using SnapObjects.Data.PostgreSql;
```

```
using SnapObjects.Data.Odbc;
```

Step 2: Add the connection string to connect to the database and authenticate the username and password against the users in the database.

Below is the complete scripts of the **UserValidator.cs** file (suppose the users are stored in the "users" table in a SQL Server database).

```
using System.Threading.Tasks;
using IdentityServer4.Models;
using IdentityServer4.Validation;
using System;
using Microsoft.Data.SqlClient;
using SnapObjects.Data;
using SnapObjects.Data.SqlServer;

namespace ServerAPIs
{

    public class UserValidator : IResourceOwnerPasswordValidator
    {
        public Task ValidateAsync(ResourceOwnerPasswordValidationContext context)
        {
            //To validate username and password against a SQLServer database, set
 the connection string as below
            String Constr = @"Data Source=172.16.1.10,1433;Initial
 Catalog=pb_cloud;Integrated Security=False;User
 ID=sa;Password=1234;Pooling=True;Min Pool Size=0;Max Pool
 Size=100;MultipleActiveResultSets=False;Encrypt=False;TrustServerCertificate=False;ApplicationIn
            SqlServerDataContext _context = new SqlServerDataContext(new
 SqlConnection(Constr));
            string sql = "select username from users where isValid = 1 and username
 = '" + context.UserName + "' and password = '" + context.Password + "'";
            var users = _context.SqlExecutor.Select<DynamicModel>(sql);

            if (users.Count >= 1)
            {
                //If validation is successful, returns the user
                context.Result = new GrantValidationResult(subject:
 context.UserName, authenticationMethod: "custom");
            }
            else
            {
                //If validation failed, returns the error
                context.Result = new
 GrantValidationResult(TokenRequestErrors.InvalidGrant, "Incorrect username or
 password.");
```

```
            }

            return Task.CompletedTask;
        }
    }
}
```

6.3.3.2 Validate username and password against an LDAP server

When the username and password are passed from the application to the built-in OAuth server, the OAuth server will by default authenticate them against the users predefined in DefaultUserStore.cs. For security concern, you might want the OAuth server to authenticate against the users stored in an LDAP server instead of DefaultUserStore.cs.

To do that,

Step 1: Install the **Microsoft.Windows.Compatibility** NuGet package first, as the following sample scripts make references to this package.

Step 2: Modify the **UserValidator.cs** file.

Here is a sample script of the **UserValidator.cs** class that connects to an LDAP server to authenticate the user credentials.

```
using IdentityServer4.Models;
using IdentityServer4.Validation;
using System;
using System.DirectoryServices;
using System.Threading.Tasks;
namespace ServerAPIs
{
    public class UserValidator: IResourceOwnerPasswordValidator
    {
        public Task ValidateAsync(ResourceOwnerPasswordValidationContext context)
        {
            string strError = string.Empty;
            bool lb_succes = false;
            string ls_server = "ldap.appeon.com";
            string ls_user = context.UserName;
            string ls_pass = context.Password;
            using (DirectoryEntry adsEntry = new DirectoryEntry("LDAP://" +
 ls_server, ls_user, ls_pass, AuthenticationTypes.Secure))
            {
                using (DirectorySearcher adsSearcher = new
 DirectorySearcher(adsEntry))
                {
                    adsSearcher.Filter = "(SAMAccountName=" + ls_user + ")";
                    adsSearcher.PropertiesToLoad.Add("cn");
                    try
                    {
                        SearchResult adsSearchResult = adsSearcher.FindOne();
                        if (adsSearchResult == null)
                        {
                            lb_succes = false;
                        }
                    }
                    catch (Exception ex)
                    {
                        strError = ex.Message;
                    }
                    finally
                    {
                        adsEntry.Close();
```

```
                    }
                }
            }
            if (strError.Length == 0)
            {
                lb_succes = true;
            }

            if (lb_succes)
            {
                context.Result = new GrantValidationResult(subject:
 context.UserName, authenticationMethod: "custom");
            }
            else
            {
                context.Result = new
 GrantValidationResult(TokenRequestErrors.InvalidGrant, "Incorrect
 username,password or server.");
            }
            return Task.CompletedTask;
        }

    }
}
```

6.3.3.3 Test the OAuth server

Test the OAuth server by sending a request which includes the grant type, scope, client ID, client secret, and user credentials.

1. Right click in the code block of a method, and select **Run Test(s)** from the popup menu.

   The Web API Tester is launched.

2. In the Web API Tester, click the plus (+) sign to create a new request:

   URL: http://localhost:5000/connect/token

   HTTP method: POST

   Content-Type: application/x-www-form-urlencoded

   Request (when grant type is client credentials):

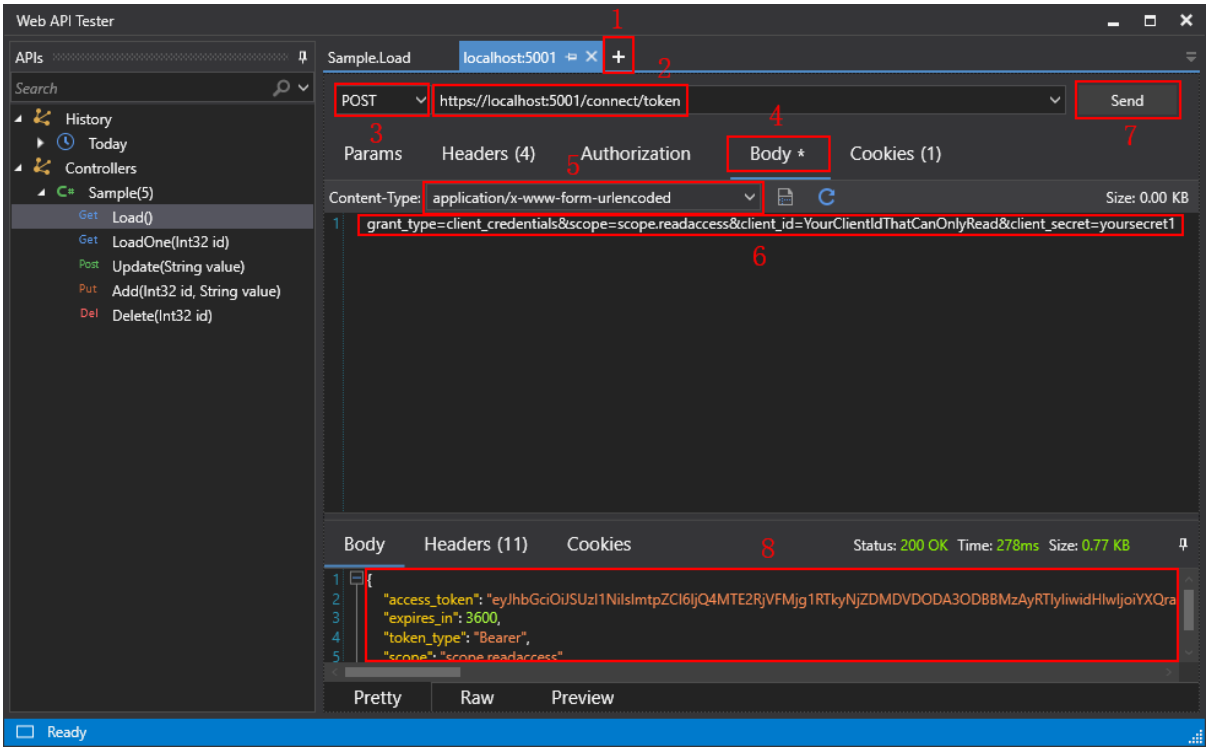
   ```
   grant_type=client_credentials&scope=scope.readaccess&client_id=YourClientIdThatCanOnlyRead&cli
   ```

   Or request (when grant type is resource owner password):

   ```
   grant_type=password&scope=scope.readaccess&client_id=YourClientIdThatCanOnlyRead&client_secret=
   ```

3. Click **Send** to send the request, and the OAuth server returns the token information if validation is successful.

Figure 6.14:



6.4 Using Amazon Cognito

6.4.1 Preparations

Before making changes to the PowerBuilder client app, let's follow the steps below to make sure 1) the PowerBuilder application can run successfully, 2) the app has been deployed as an installable cloud app successfully, and 3) the PowerServer C# solution (including the built-in Amazon Cognito server) has been successfully generated.

In this tutorial, we will take Sales Demo as an example.

Step 1: Select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**.

Step 2: When the SalesDemo workspace is loaded in the PowerBuilder IDE, click the **Run** button in the PowerBuilder toolbar.

Step 3: When the application main window is opened, click the **Address** icon in the application ribbon bar and make sure data can be successfully retrieved.

Step 4: Create and configure a PowerServer project for the Sales Demo app (detailed instructions are provided in the Quick Start guide).

IMPORTANT: In the **Web APIs** tab, select **Use built-in AWS Cognito server** from the **Auth Template** list box.

Step 5: Deploy the application as an installable cloud app. The PowerServer C# solution is generated, but the installable cloud app cannot run yet because further settings and changes are required, as explained in the subsequent sections.

The PowerServer C# solution contains a built-in Amazon Cognito server and the authentication class files as shown below.

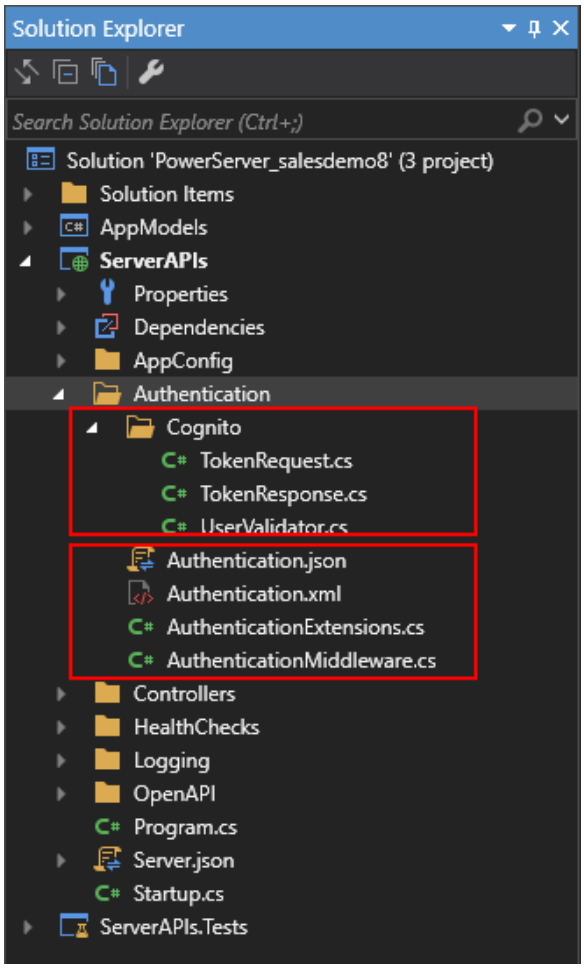
- The built-in Amazon Cognito server authenticates the user credential with the Cognito service and returns an identity token. The built-in server is included in the **ServerAPIs** project; it will run automatically when the PowerServer Web APIs (the **ServerAPIs** project) runs.

- The authentication class and configuration files will be used by the PowerServer Web APIs to validate the token passed from the client; and if validation is successful, data will be obtained from the database.

  - **Authentication.json** contains the settings for enabling the authentication feature ("PowerServer:EnableAuthentication") and specifying the Amazon Cognito user pool ("AWS").

    The "PowerServer:EnableAuthentication" setting is set to **true** by default. Setting it to **false** will turn off the authentication feature. The "AWS" block is used to the specify the Amazon Cognito user pool including region, user pool ID, user pool client ID, and user pool client secret.

Figure 6.15:



6.4.2 Creating the Amazon Cognito user pool

This tutorial uses the Amazon Cognito user pool as an SAML identity provider for the installable cloud app.

The following outlines the key steps for creating the Amazon Cognito user pool. For complete and detailed instructions, please refer to Getting Started with User Pools.

When the user pool is created successfully, gather the information such as region, user pool ID, user pool client ID, and user pool client secret which are required by the built-in server later (as shown below).

```
"AWS": {
    "Region": "us-west-2",
    "UserPoolId": "us-west-2_5wyOzYn1d",
    "UserPoolClientId": "4linbauf6d58b552r6lc3gbpkc",
    "UserPoolClientSecret": "1prlm08gm3aptlokcbai88ekiegff9mqbc98nhebfart5g4a3cr2"
  }
```

Step 1: Set up the AWS Single Sign-On (SSO).

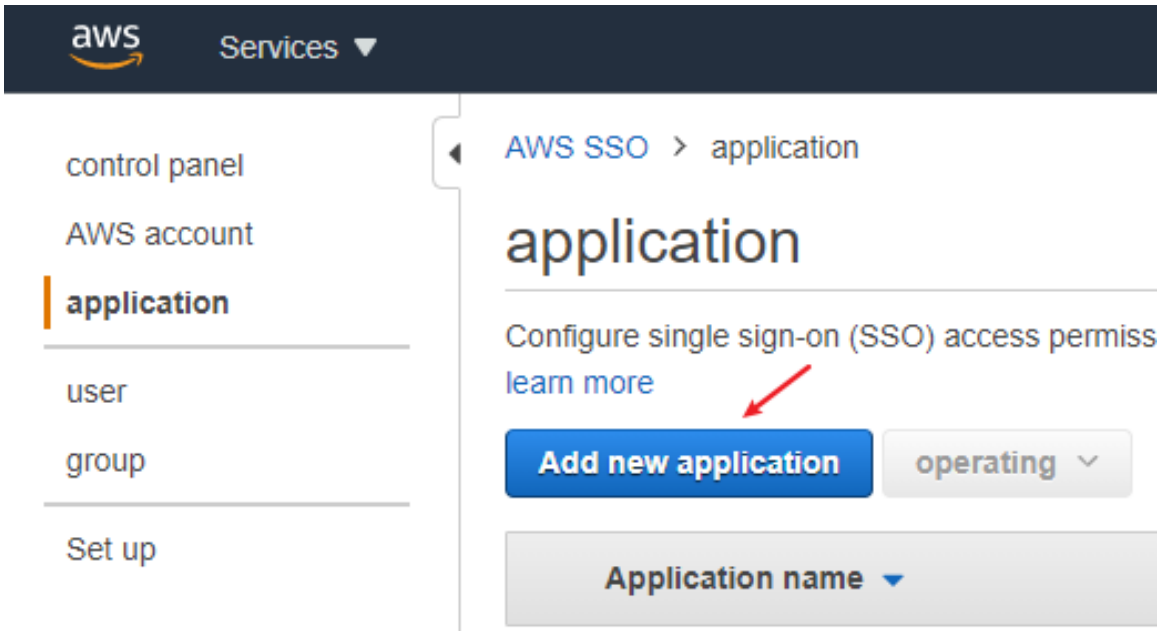
Before you can set up AWS Single Sign-On (SSO), you must:

- Have first set up the AWS Organizations service and have All features set to enabled. For more information about this setting, see Enabling All Features in Your Organization in the AWS Organizations User Guide.
- Sign in with the AWS Organizations management account credentials before you begin setting up AWS SSO. These credentials are required to enable AWS SSO. For more information, see Creating and Managing an AWS Organization in the AWS Organizations User Guide. You cannot set up AWS SSO while signed in with credentials from an Organization's member account.

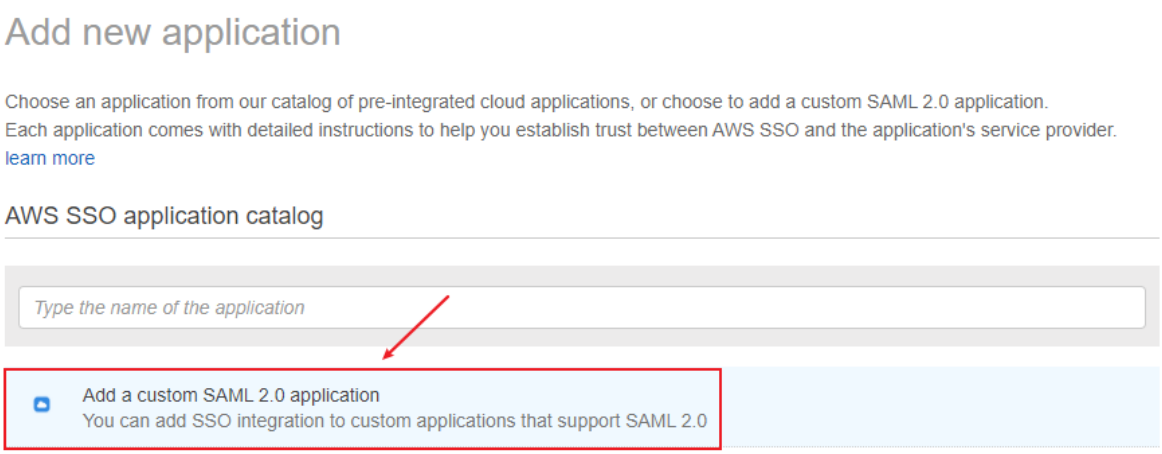
For more details, refer to AWS SSO prerequisites.

Step 2: Get the SAML 2.0 metadata.

1) Add a new application.



2) Add a custom SAML 2.0 application.

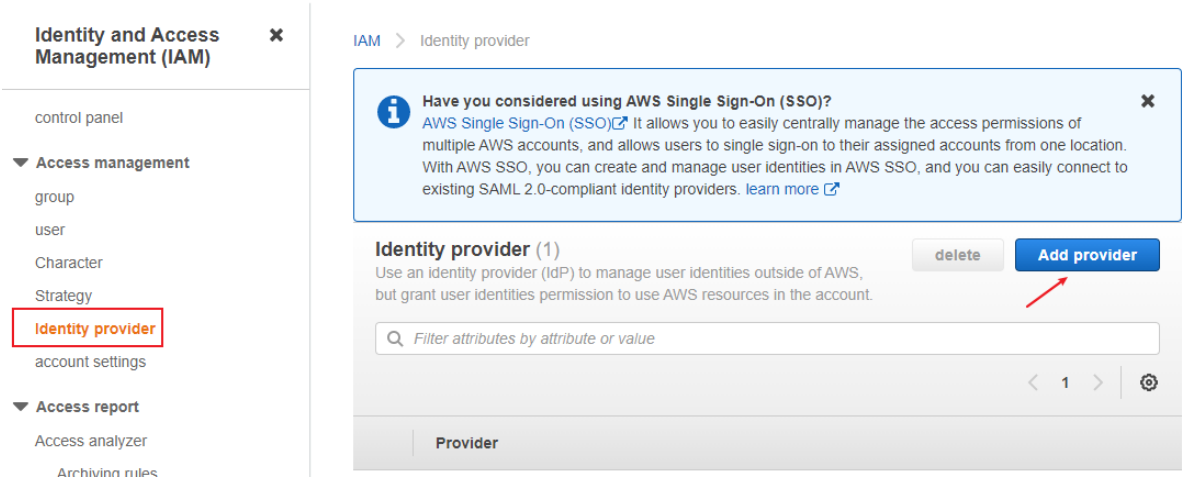


3) After filling in the configuration, save it, and then download the SAML metadata file or save the metadata file URL.

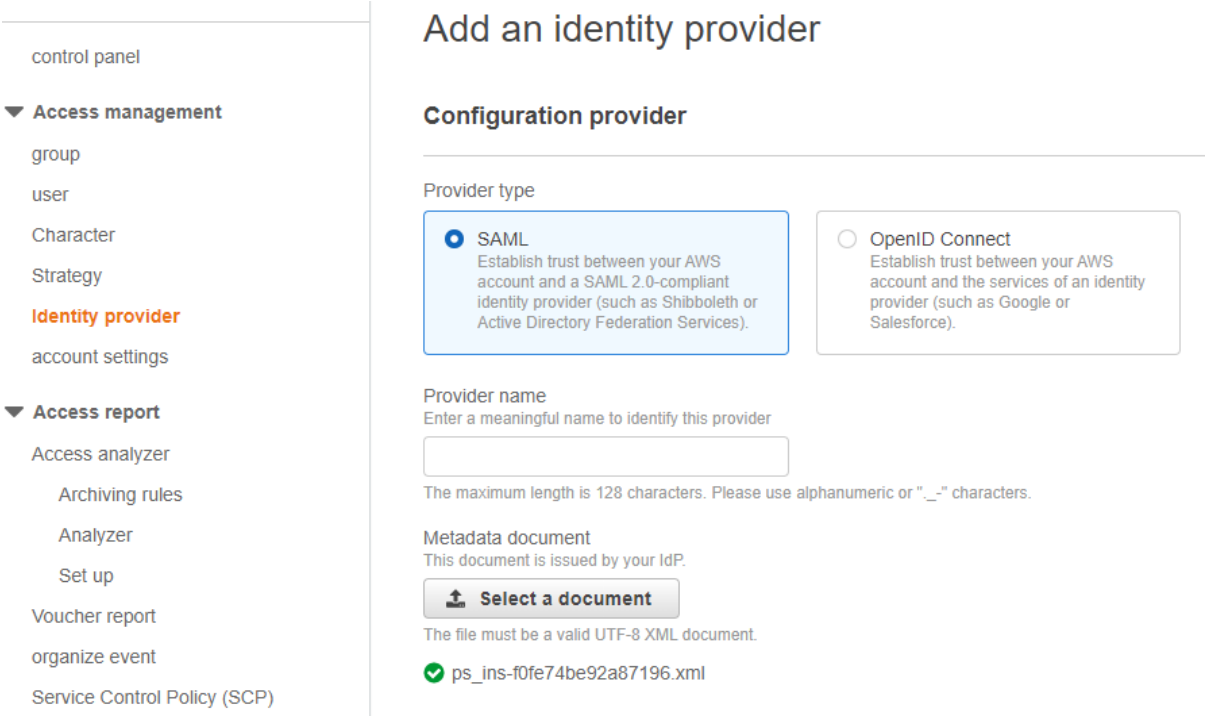
For more details, refer to [AWS Single Sign-On](#).

Step 3: Add an identity provider.

1) Click **Add provider**.



2) Select **SAML** and then upload the SAML metadata file you just got.



Step 4: Create the user pool.

a. Go to the Amazon Cognito console. You might be prompted for your AWS credentials.

b. Choose **Manage User Pools**.

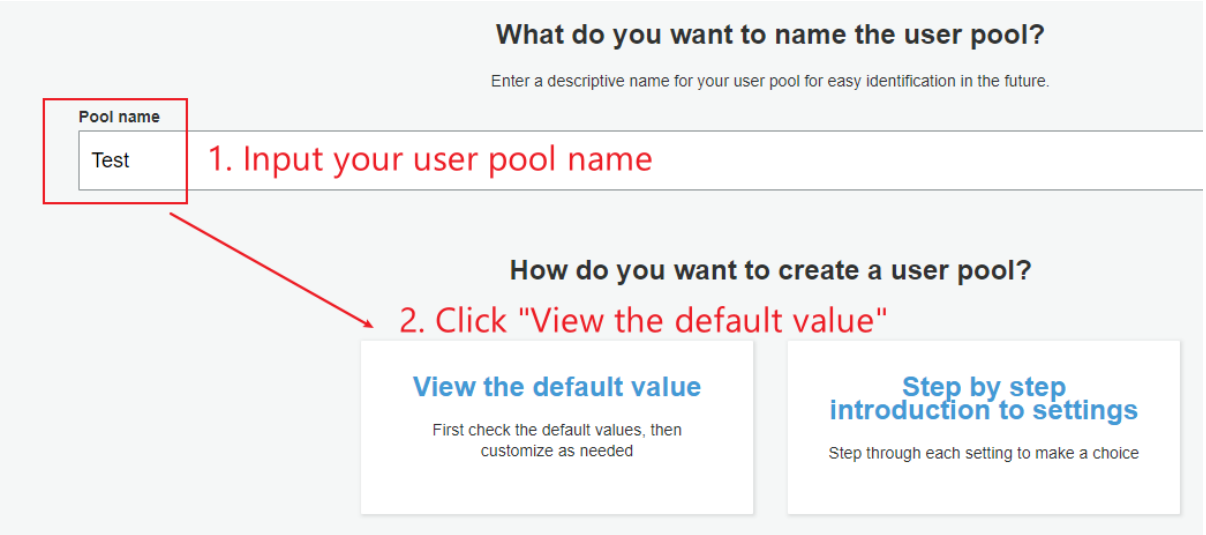
c. In the top-right corner of the page, choose **Create a user pool**.

d. Provide a name for your user pool, and choose **Review defaults** to save the name.

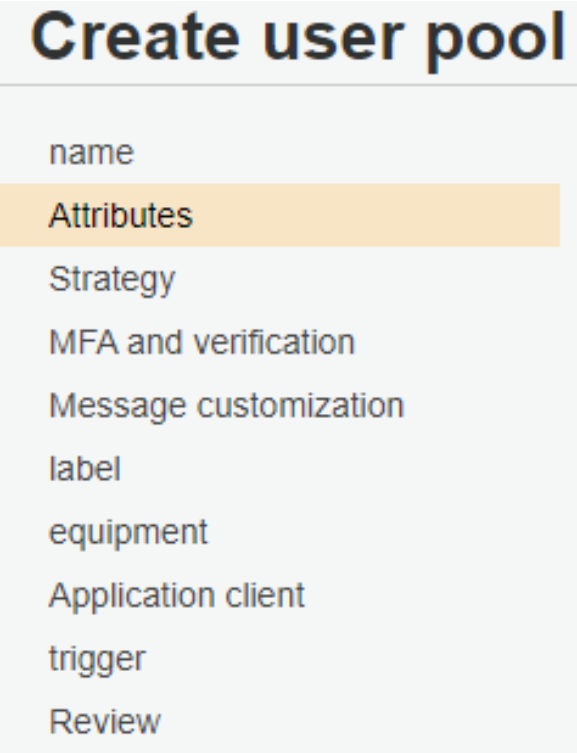
e. In the top-left corner of the page, choose **Attributes**, choose **Email address or phone number** and **Allow email addresses**, and then choose **Next step** to save.

f. In the left navigation menu, choose **Review**.

g. Review the user pool information and make any necessary changes. When the information is correct, choose **Create pool**.



Fill in the following configuration as required.



Tips: It is recommended to modify the configuration (for example, "Attributes") that cannot be modified after pool creation.

Step 5: Create the user pool application client.

a. On the navigation bar on the left-side of the page, choose **App clients** under **General settings**.

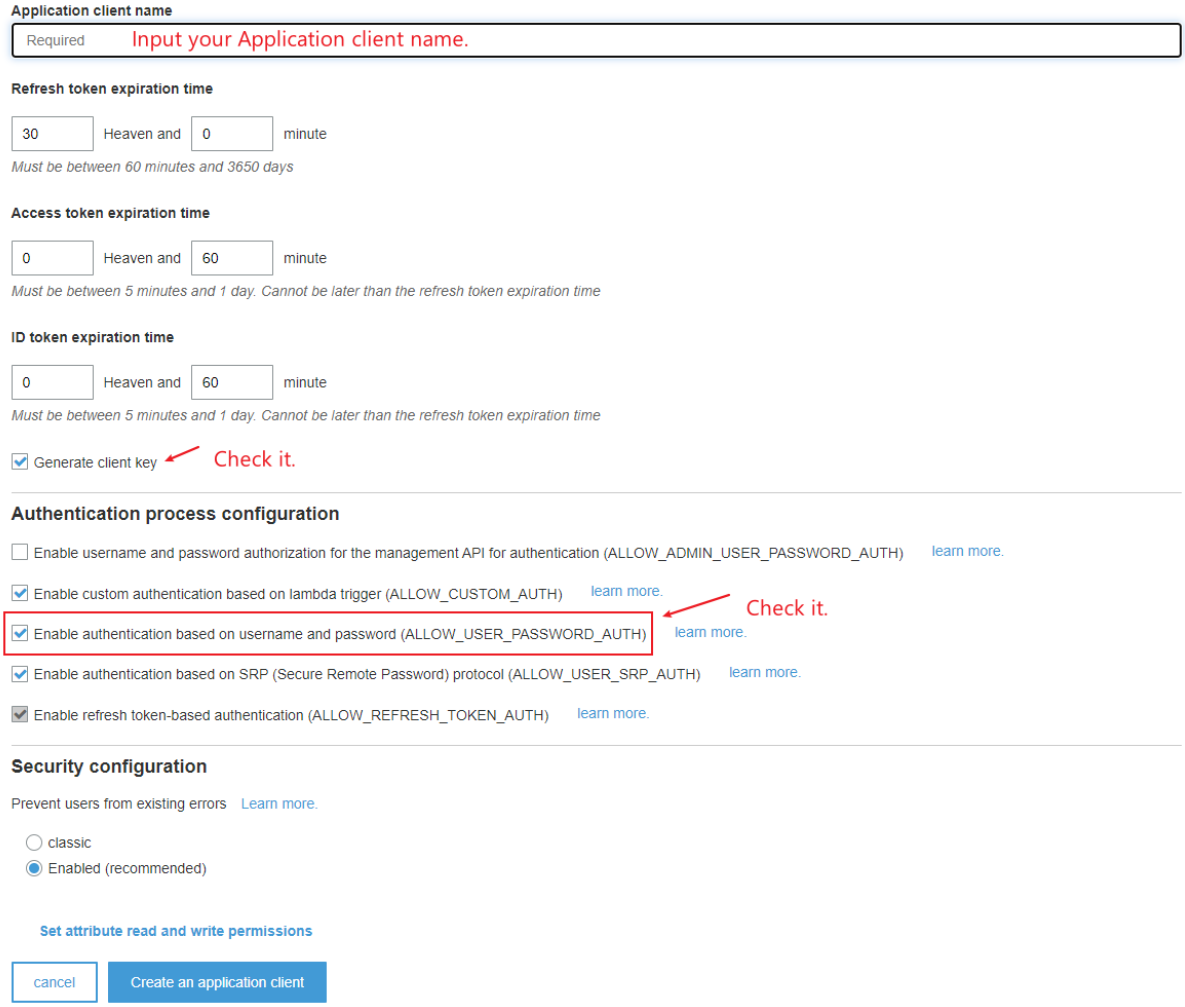
b. Choose **Add an app client**.

c. Give your app a name.

d. Check **Generate client key**.

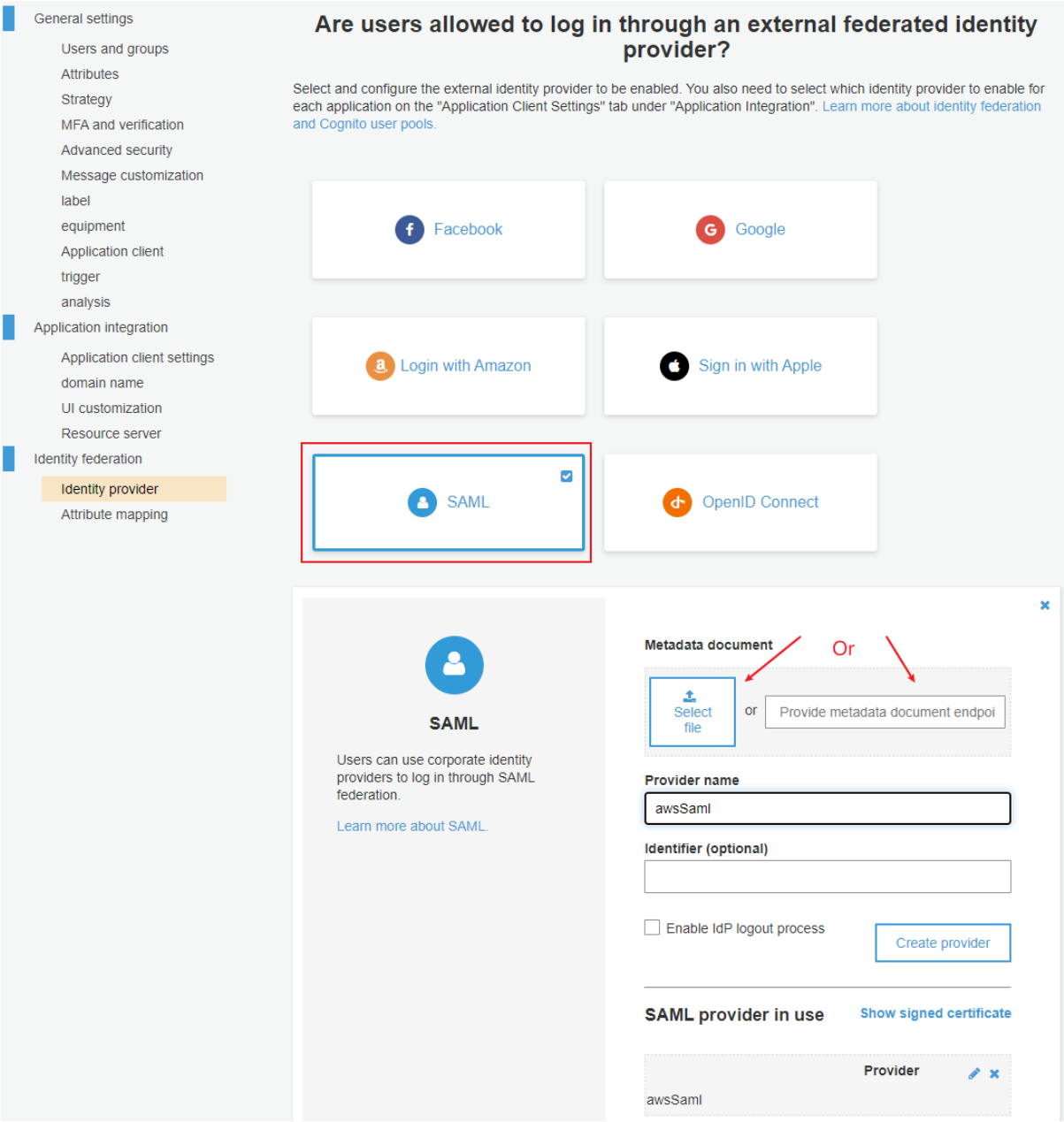
e. Check **Enable authentication based on username and password (ALLOW_USER_PASSWORD_AUTH)**.

f. Choose **Create an application client**.



Step 6: Configure the SAML identity provider.

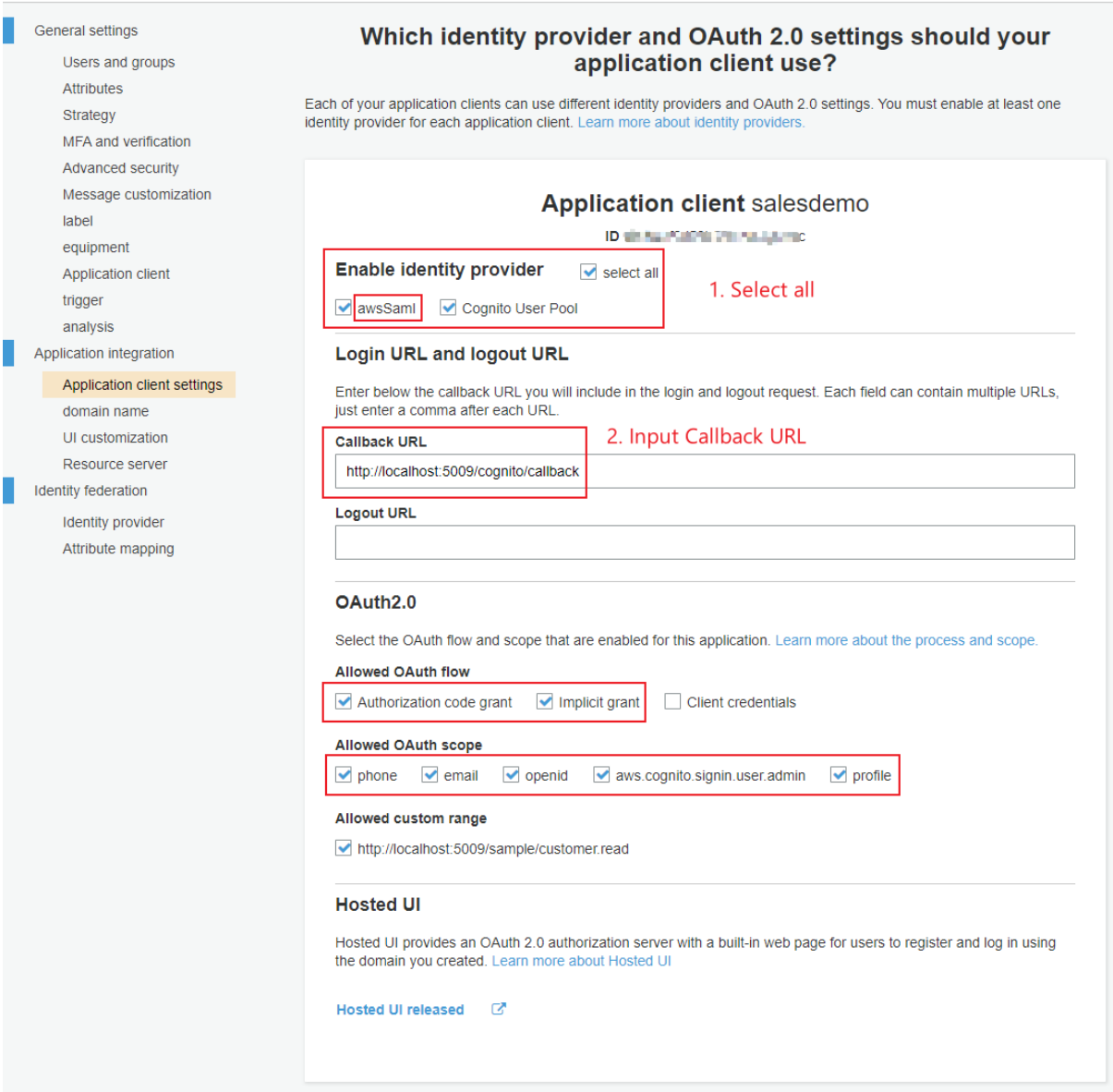
Open the identity provider configuration page of the user pool, choose **SAML**, select the SAML metadata file downloaded in step 2 or the terminal node URL of the metadata file.



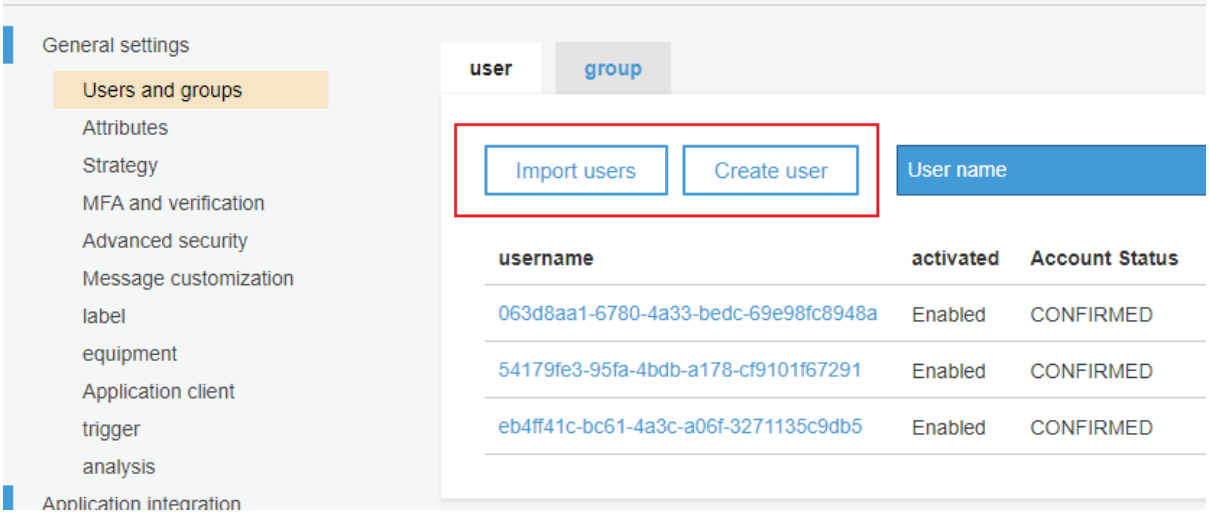
Step 7: Configure the application integration settings.

a. Configure domain name. You can configure the Amazon Cognito domain name or your own domain name.

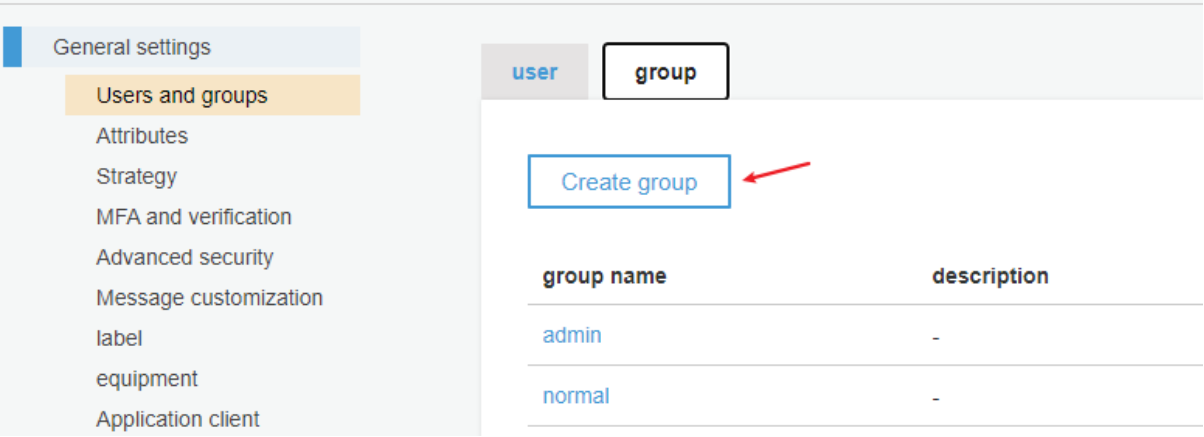
b. Configure the application client settings, select all options under the **Enable identity provider**, enter the callback URL and the logout URL, select **Authorization code grant** and **implicit grant** under the **Allowed OAuth flow**, select all options under the **Allowed OAuth scope**, save the settings and click to publish Hosted UI.



Step 8: Import or create users.



Step 9: Create a group (optional).



6.4.3 Modifying the PowerBuilder client app

6.4.3.1 Purpose

In this section, we will modify the PowerBuilder application source code and the PowerServer project settings to achieve the following results:

- Gets the user credential from the application login window, then authenticates it with the Amazon Cognito User Pools and gets an identity token.
- Uses the identity token to access data from the PowerServer Web API.
- Refreshes the identity token when necessary.

6.4.3.2 Add scripts

Step 1: Declare the following global variables.

```
//Token expiresin
Long gl_Expiresin
//Refresh token clockskew
Long gl_ClockSkew = 3
```

Step 2: Define a global function and name it **f_Authorization()**.

Select from menu **File** > **New**; in the **New** dialog, select the **PB Object** tab and then select **Function** and click **OK** to add a global function.

This global function uses the HTTP Post method to send the user credentials to the authorization server and then gets the identity token from the HTTP Authorization header.

Add scripts to the **f_Authorization()** function to implement the following scenario: When the application starts, the application uses the username and password from the login window to get the token, and when the token expires, the login window displays for the user to input the username and password again.

The following scripts hard code the username and password instead of getting them from the login window. You can change the scripts to use the login window after you implement the login window and return the username and password to the **f_Authorization()** function.

```
//Integer f_Authorization() for password
//UserName & Password are passed from the login window
RestClient lrc_Client
```

```
String  ls_url, ls_UserName, ls_UserPass, ls_PostData, ls_Response, ls_expires_in
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn
JsonParser ljson_Parser

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//login window can be implemented to return username & password according to actual
 needs.
//Open(w_login)
//Return UserName & Password

ls_UserName = "admin@test.com"
ls_UserPass = "appeon123"

If IsNull ( ls_UserName ) Or Len ( ls_UserName ) = 0 Then
 MessageBox( "Tips", "UserName is empty!" )
 Return li_rtn
End If
If IsNull ( ls_UserPass ) Or Len ( ls_UserPass ) = 0 Then
 MessageBox( "Tips", "Password is empty!" )
 Return li_rtn
End If

ls_PostData = '{"username":"' + ls_UserName + '", "password":"' + ls_UserPass +
 '"}';
lrc_Client = Create RestClient
lrc_Client.SetRequestHeader("Content-Type","application/json")
li_Return = lrc_Client.GetJWTToken( ls_Url, ls_PostData, ls_Response )
If li_Return = 1 and Pos ( ls_Response, "access_token" ) > 0 Then
 ljson_Parser = Create JsonParser
 ljson_Parser.LoadString(ls_Response)
 ls_TokenType = ljson_Parser.GetItemString("/token_type")
 ls_AccessToken = ljson_Parser.GetItemString("/access_token")
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
 //Set Global Variables
 gl_Expiresin = Long (ljson_Parser.GetItemNumber("/expires_in"))

 li_rtn = 1
Else
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) )
End If

If IsValid ( ljson_Parser ) Then DesTroy ( ljson_Parser )
If IsValid ( lrc_Client ) Then DesTroy ( lrc_Client )

Return li_rtn
```

Step 3: Insert a timing object (**timing_1**) to the application and add the following scripts to the **Timer** event of **timing_1**.

1) Open the application object and then select from menu **Insert** > **Object** > **Timing** to add a timing object to the application.

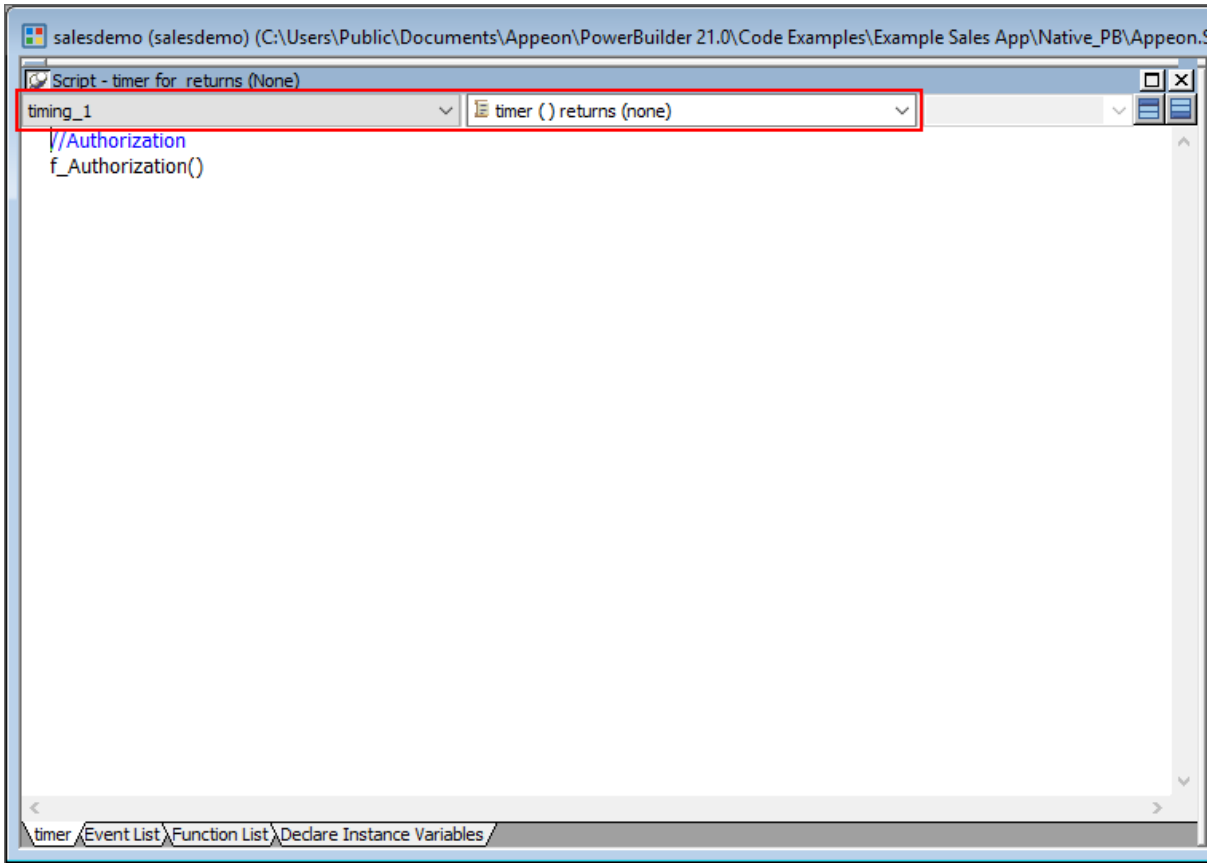
2) Add the following scripts to the **Timer** event of **timing_1**.

```
//Authenticates the user
f_Authorization()
```

When displayed in the source editor, the Timer event looks like this:

```
event timer;//Authenticates the user
f_Authorization()
end event
```

Figure 6.16:



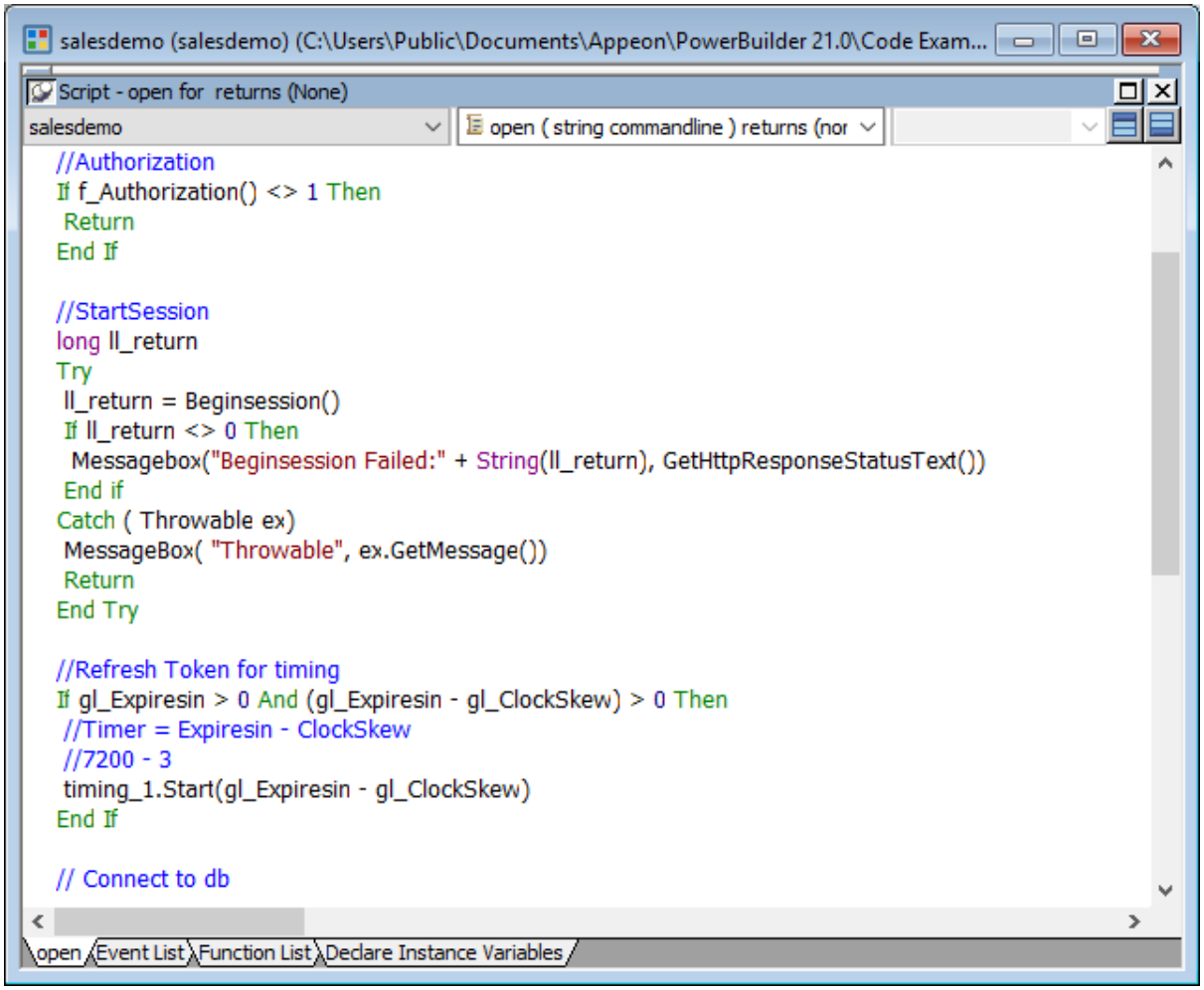
Step 4: Add the following scripts to the application **Open** event.

Place the scripts before the database connection is established. The scripts get the token from the built-in Cognito server and then start the user session (using the **BeginSession** function) to include the token information in the session.

```
//Authenticates the user and returns the token
If f_Authorization() <> 1 Then
 Return
End If

//Starts the session
long ll_return
Try
 ll_return = Beginsession()
 If ll_return <> 0 Then
  Messagebox("Beginsession Failed:" + String(ll_return),
 GetHttpResponseStatusText())
 End if
Catch ( Throwable ex)
 MessageBox( "Throwable", ex.GetMessage())
 Return
End Try

//Refreshes the token for timing
If gl_Expiresin > 0 And (gl_Expiresin - gl_ClockSkew) > 0 Then
 //Timer = Expiresin - ClockSkew
```

```
 //7200 - 3
 timing_1.Start(gl_Expiresin - gl_ClockSkew)
End If

//Connects to db
```

Figure 6.17:



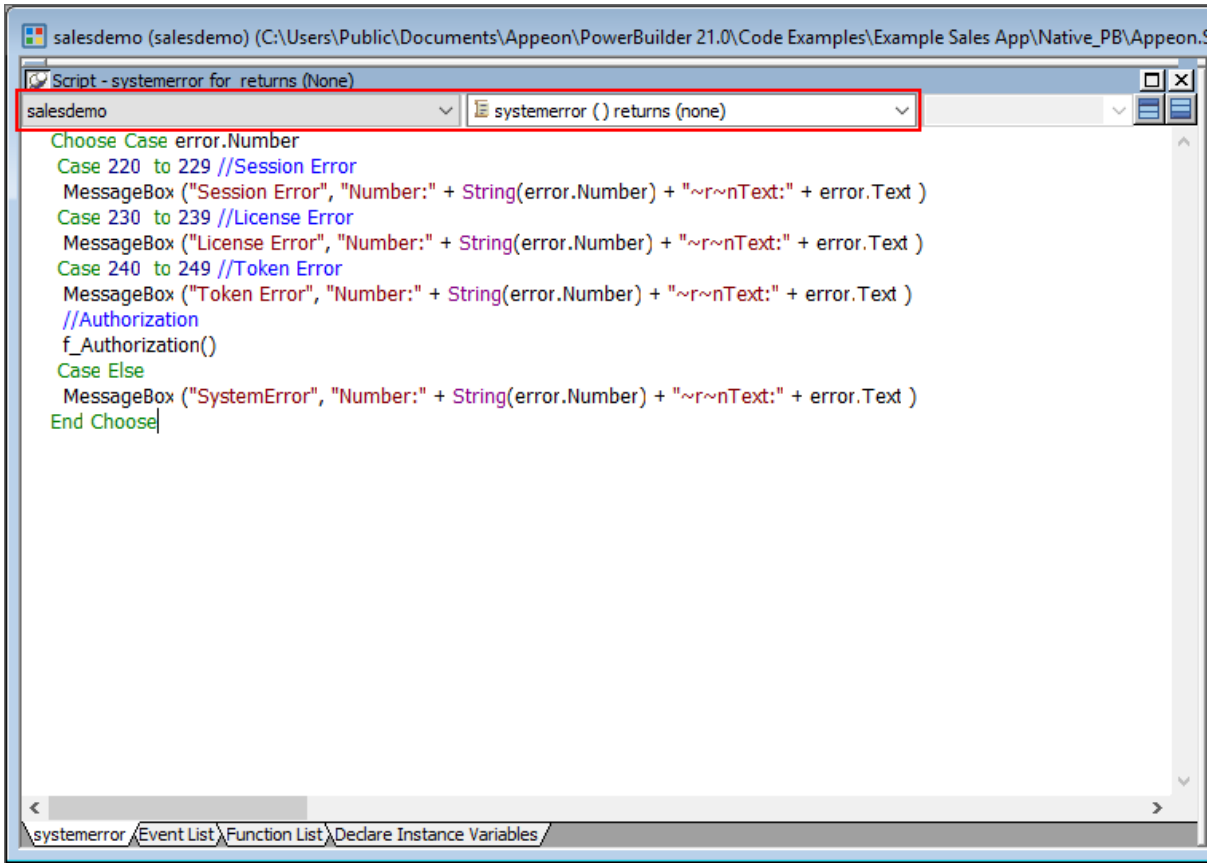
Step 5: Add the following scripts to the **SystemError** event.

The scripts will trigger the **SystemError** event when the session or license encounters an error; and if the token is invalid or expires, the scripts will call the **f_Authorization** function to get the token again.

```
Choose Case error.Number
 Case 220  to 229 //Session Error
  MessageBox ("Session Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 230  to 239 //License Error
  MessageBox ("License Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 240  to 249 //Token Error
  MessageBox ("Token Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
  //Authorization
  f_Authorization()
 Case Else
  MessageBox ("SystemError", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
```

```
End Choose
```

Figure 6.18:



6.4.3.3 Add an INI file

Create an INI file in the same location as the PBT file and name it **CloudSetting.ini**.

The INI file specifies the URL for requesting the token from the Amazon Cognito server. Notice that **TokenURL** points to the "/connect/token" API of the built-in Cognito server, and the Cognito server root URL (for example, https://localhost:5000/) is the same as the URL of PowerServer Web API. If you change the PowerServer Web API URL, change the root URL here accordingly.

```
[Setup]
TokenURL=https://localhost:5000/connect/token
```

6.4.3.4 Start session manually by code

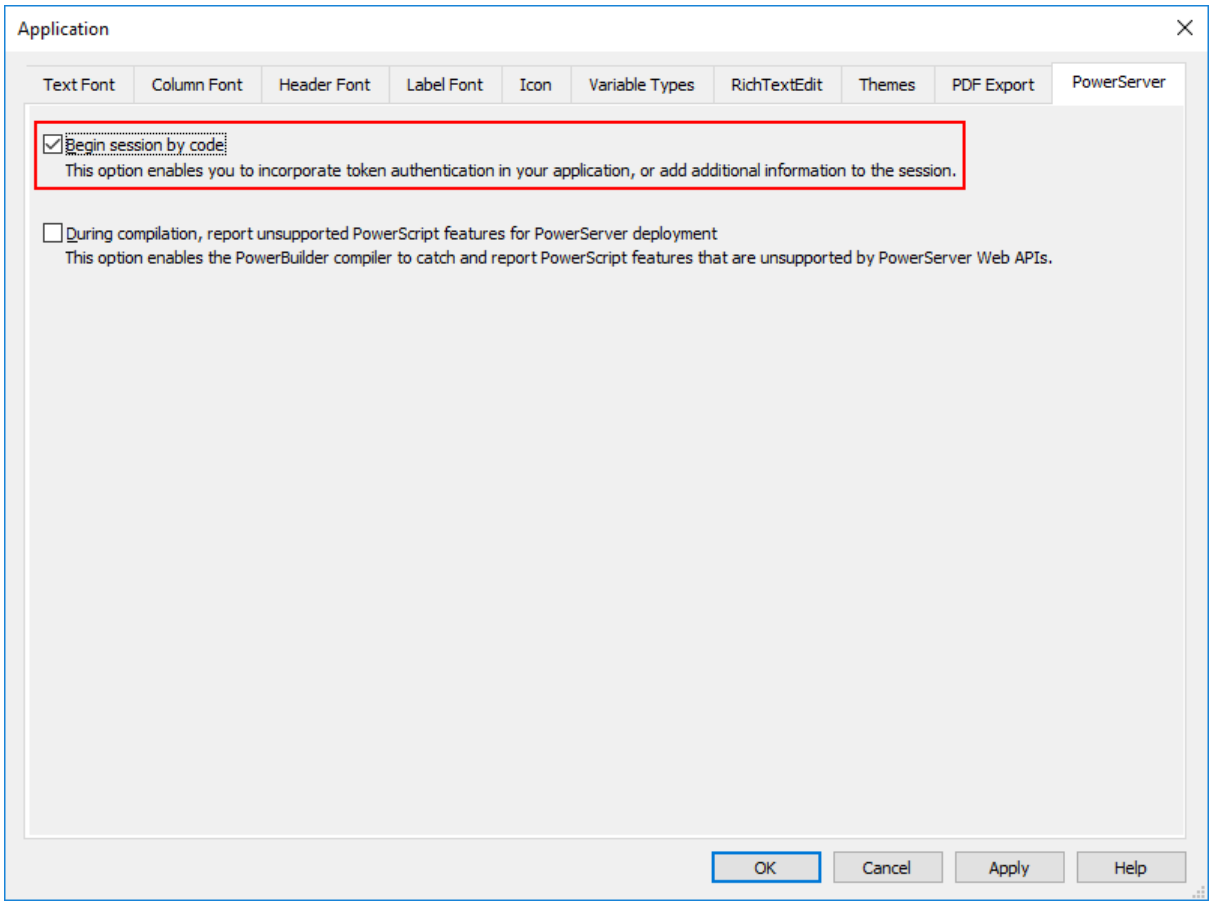
By default, the user session is automatically created when the application starts; and the session includes no token. For the session to include the token, the session must be started manually by code instead of automatically.

To start the session manually by code,

Step 1: Enable "**Begin session by code**" in the PowerBuilder IDE. (Steps: Open the application object painter, click **Additional Properties** in the application's **Properties** dialog; in the **Application** dialog, select the **PowerServer** tab and then select the **Begin session by code** option.)

After this option is enabled, when the **BeginSession** function in the application **Open** event is called, it will create a session that includes the token information (See scripts in step 4 in "Add scripts").

Figure 6.19:

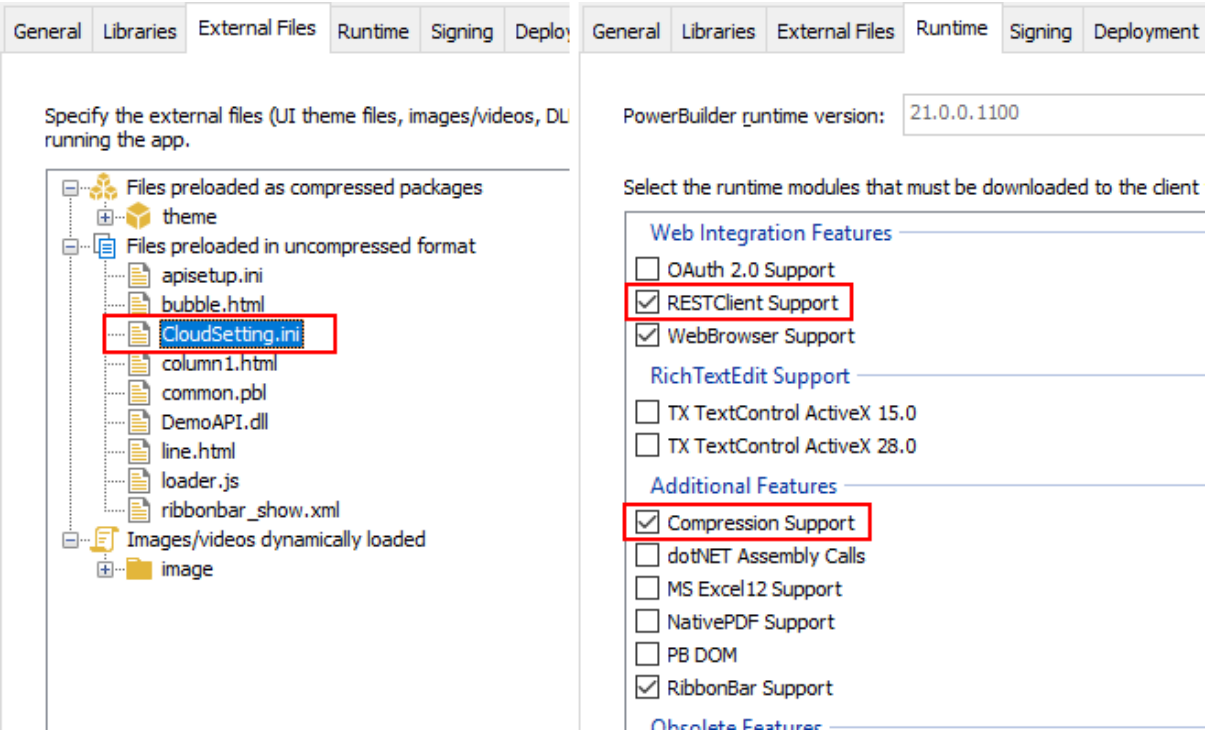


6.4.3.5 Modify and re-deploy the PowerServer project

Step 1: Add the INI file **CloudSetting.ini** to the **Files preloaded in uncompressed format** section under the **External Files** tab.

Step 2: Select **RESTClient Support** and **Compression Support** under the **Runtime** tab.

Figure 6.20:



Step 3: Double check the URL of the PowerServer Web APIs in the **Web APIs** tab. Make sure the port number is not occupied by any other program.

Tip: You can execute the command "netstat -ano | findstr *portnumber*" to check if the port number is occupied by any other program.

The built-in Cognito server will run at the same URL as the PowerServer Web API. If the PowerServer Web API URL is changed, change the root URL accordingly in the INI file.

Step 4: Double check that **Use built-in AWS Cognito server** is selected from the **Auth Template** list box in the **Web APIs** tab.

Step 5: Save the changes and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) so that the above settings can take effect in the installable cloud app.

6.4.4 Modifying the authentication template

The AWS Cognito user pool must be provided in the built-in Cognito server so that the PowerServer Web APIs can use it to validate the identity token passed from the client. And if validation is successful, it can get data from the database.

Note

The authentication template will be restored if the "**Auth Template**" option is changed and the PowerServer C# solution is re-built from the PowerBuilder IDE. Therefore, do not change the "**Auth Template**" option if you have made changes to the template in the solution.

Open the **Authentication.json** file and specify the AWS Cognito user pool (including region, user pool ID, user pool client ID, and user pool client secret) that will be used to validate the identity token passed from the client.

```
"AWS": {
    "Region": "us-west-2",
    "UserPoolId": "us-west-2_5wyOzYn1d",
    "UserPoolClientId": "4linbauf6d58b552r6lc3gbpkc",
    "UserPoolClientSecret": "1prlm08gm3aptlokcbai88ekiegff9mqbc98nhebfart5g4a3cr2"
  }
```

Figure 6.21:

6.4.5 (Optional) Testing the Cognito server

Test the built-in Amazon Cognito server by sending a request.

1. Right click in the code block of a method, and select **Run Test(s)** from the popup menu.

   The Web API Tester is launched.

2. In the Web API Tester, click the plus (+) sign to create a new request:

   URL: http://localhost:5000/Cognito/getToken

   HTTP method: POST

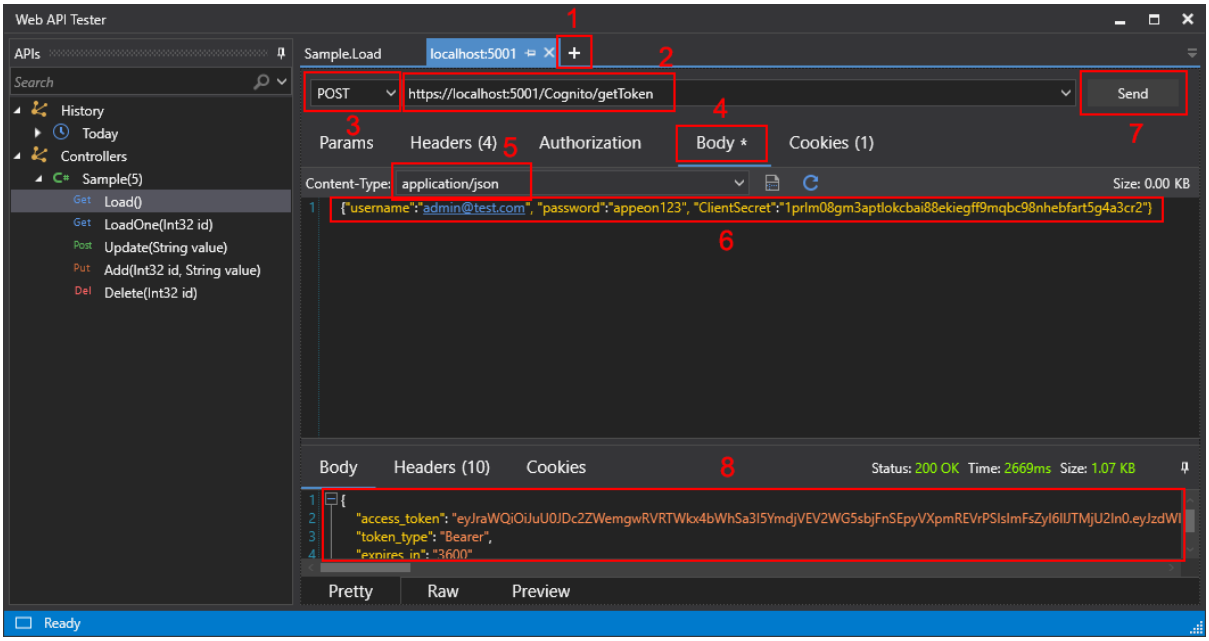
   Content-Type: application/json

   Request:

   ```
   {"username":"admin@test.com", "password":"appeon123",
    "ClientSecret":"1prlm08gm3aptlokcbai88ekiegff9mqbc98nhebfart5g4a3cr2"}
   ```

3. Click **Send** to send the request, and the API returns the token information if validation is successful.

Figure 6.22:



6.5 Using other authentication servers

PowerServer 2021 provides templates that can be easily extended to support the other identity providers that work with the OAuth flows or JWT, such as Azure AD or Azure AD B2C.

6.5.1 Azure Active Directory (AD)

6.5.1.1 Preparations

Before making changes to the PowerBuilder client app, let's follow the steps below to make sure 1) the PowerBuilder application can run successfully, 2) the app has been deployed as an installable cloud app successfully, and 3) the PowerServer C# solution has been successfully generated.

In this tutorial, we will take Sales Demo as an example.

Step 1: Select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**.

Step 2: When the SalesDemo workspace is loaded in the PowerBuilder IDE, click the **Run** button in the PowerBuilder toolbar.

Step 3: When the application main window is opened, click the **Address** icon in the application ribbon bar and make sure data can be successfully retrieved.

Step 4: Create and configure a PowerServer project for the Sales Demo app (detailed instructions are provided in the Quick Start guide).

IMPORTANT: In the **Web APIs** tab, select **Use external auth service** from the **Auth Template** list box.

Step 5: Deploy the application as an installable cloud app. The PowerServer C# solution is generated, but the installable cloud app cannot run yet because further settings and changes are required, as explained in the subsequent sections.

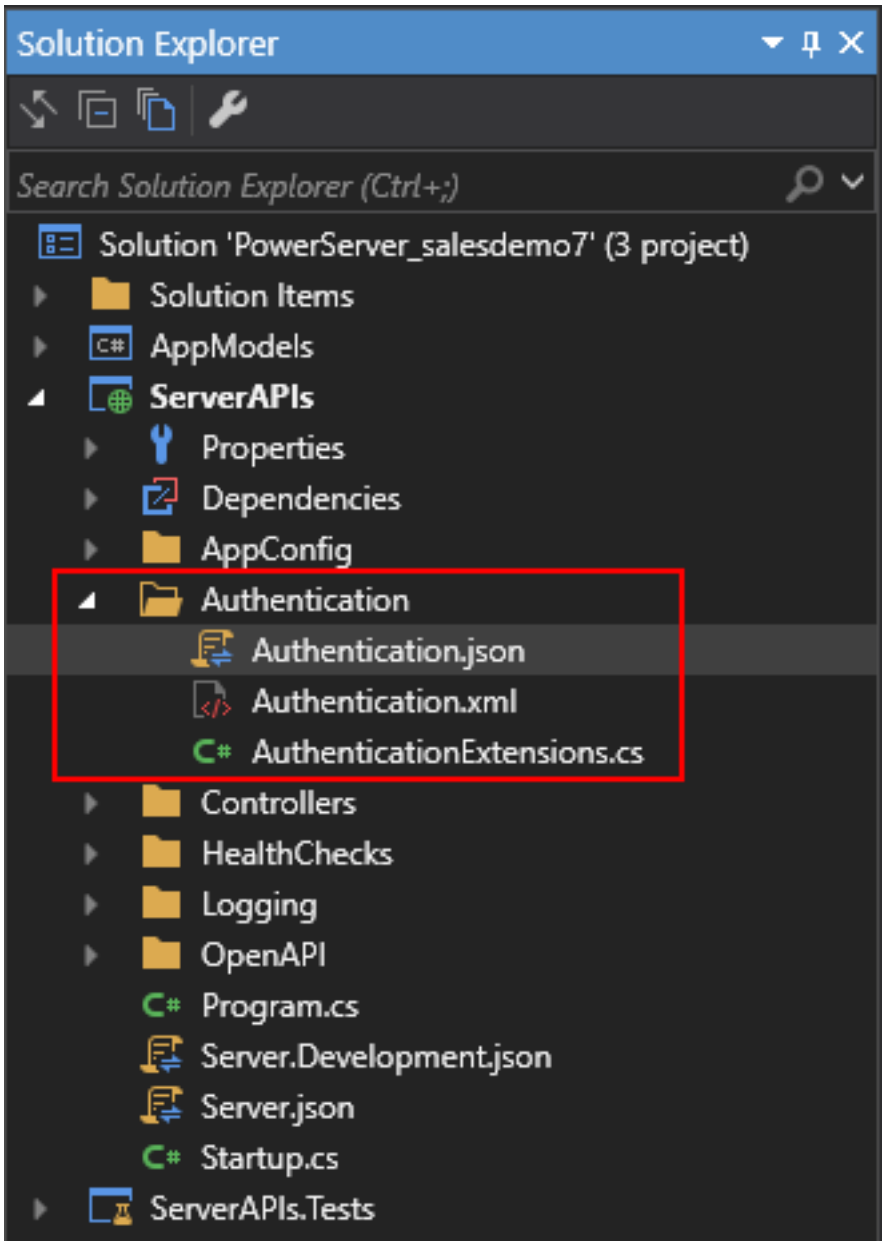
The PowerServer C# solution provides templates for configuring the address of the authentication server like Azure AD or Azure AD B2C.

- **Authentication.json** contains the settings for enabling the authentication feature ("PowerServer:EnableAuthentication") and specifying the address of the authentication server ("Authentication:Authority"). The PowerServer Web APIs will validate the token against the authentication server; and if validation is successful, data will be obtained from the database.

  The "PowerServer:EnableAuthentication" setting is set to **true** by default. Setting it to **false** will turn off the authentication feature.

  The "Authentication:Authority" setting is set for JWT by default; you can set the address of Azure AD and Azure AD B2C.

Figure 6.23:



6.5.1.2 Creating an Azure AD tenant

The following outlines the key steps for setting up an Azure AD tenant and registering an application with the Microsoft identity platform. For complete and detailed instructions, please refer to Quickstart: Set up a tenant and Quickstart: Register an application.

During the process of creating the tenant, gather the following information:

- **Tenant ID**: for example, 0ffb9ae0-c080-4913-aa94-ed08b5de4d40
- **Primary domain**: for example, powerservertest.onmicrosoft.com
- **Application (client) ID**: for example, 49cddad2-721d-4fbc-bd64-1cfa2b183e00
- **Client secret**: for example, 2ig8hfliVu.u1kl_79RbyZuh~.X_b~e~3M
- **Application ID URI**: for example, api://49cddad2-721d-4fbc-bd64-1cfa2b183e00
- **Scope**: for example, 49cddad2-721d-4fbc-bd64-1cfa2b183e00/.default

The above information will be used later.

6.5.1.3 Modifying the PowerBuilder client app

6.5.1.3.1 Purpose

In this section, we will modify the PowerBuilder application source code and the PowerServer project settings to achieve the following results:

- Gets the user credential from the application login window, then authenticates it with the Azure AD tenant and gets a token.
- Uses the token to access data from the PowerServer Web API.
- Refreshes the token when necessary.

6.5.1.3.2 Add scripts

Step 1: Declare the following global variables.

```
//Token expiresin
Long gl_Expiresin
//Refresh token clockskew
Long gl_ClockSkew = 3
```

Step 2: Define a global function and name it **f_Authorization()**.

Select from menu **File** > **New**; in the **New** dialog, select the **PB Object** tab and then select **Function** and click **OK** to add a global function.

This global function uses the HTTP Post method to send the user credentials to the authorization server and then gets the identity token from the HTTP Authorization header.

Add scripts to the **f_Authorization()** function to implement the following scenario:

- Scenario 1: Supports Client Credentials (GrantType="client_credentials") and gets the client ID and secret from the application.

- Scenario 2: Supports Resource Owner Password (GrantType="password") and gets the username and password from a login window.

Scripts for scenario 1:

When the application starts, the application uses the client ID and secret stored in the application to get the token from Azure AD, and when the token expires, it automatically refreshes the token.

```
//Integer f_Authorization() for client_credentials
//UserName & Password from login window
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String  ls_url, ls_UserName, ls_UserPass
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")
//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "49cddad2-721d-4fbc-bd64-1cfa2b183e00"
ltr_Request.clientsecret = "2ig8hfliVu.u1kl_79RbyZuh~~.X_b~~e~~3M"
ltr_Request.granttype = "client_credentials"
ltr_Request.Scope = "49cddad2-721d-4fbc-bd64-1cfa2b183e00/.default"

loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " " +
 ls_AccessToken, true)
 //Set Global Variables
 gl_Expiresin = ltr_Response.getexpiresin( )

li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Scripts for scenario 2:

When the application starts, the client ID and secret stored in the application as well as the username and password from the login window will be sent to Azure AD to get the token, and when the token expires, the login window displays for the user to input the username and password again.

The following scripts hard code the username and password instead of getting them from the login window. You can change the scripts to use the login window after you implement the login window and return the username and password to the **f_Authorization()** function.

```
//Integer f_Authorization() for password
//UserName & Password from login window
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String  ls_url, ls_UserName, ls_UserPass
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "49cddad2-721d-4fbc-bd64-1cfa2b183e00"
ltr_Request.clientsecret = "2ig8hfliVu.u1kl_79RbyZuh~~.X_b~~e~~3M"
ltr_Request.scope = "49cddad2-721d-4fbc-bd64-1cfa2b183e00/.default"
ltr_Request.granttype = "password"

//login window can be implemented to return username & password according to actual
 needs
//Open(w_login)
//Return UserName & Password

ls_UserName = "appeon2@powerservertest.onmicrosoft.com"
ls_UserPass = "Test2008aaBB"

If IsNull ( ls_UserName ) Or Len ( ls_UserName ) = 0 Then
 MessageBox( "Tips", "UserName is empty!" )
 Return li_rtn
End If
If IsNull ( ls_UserPass ) Or Len ( ls_UserPass ) = 0 Then
 MessageBox( "Tips", "Password is empty!" )
 Return li_rtn
End If

ltr_Request.UserName = ls_UserName
ltr_Request.Password = ls_UserPass

loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
 //Set Global Variables
 gl_Expiresin = ltr_Response.getexpiresin( )

 li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Step 3: Insert a timing object (**timing_1**) to the application and add the following scripts to the **Timer** event of **timing_1**.

1) Open the application object and then select from menu **Insert** > **Object** > **Timing** to add a timing object to the application.

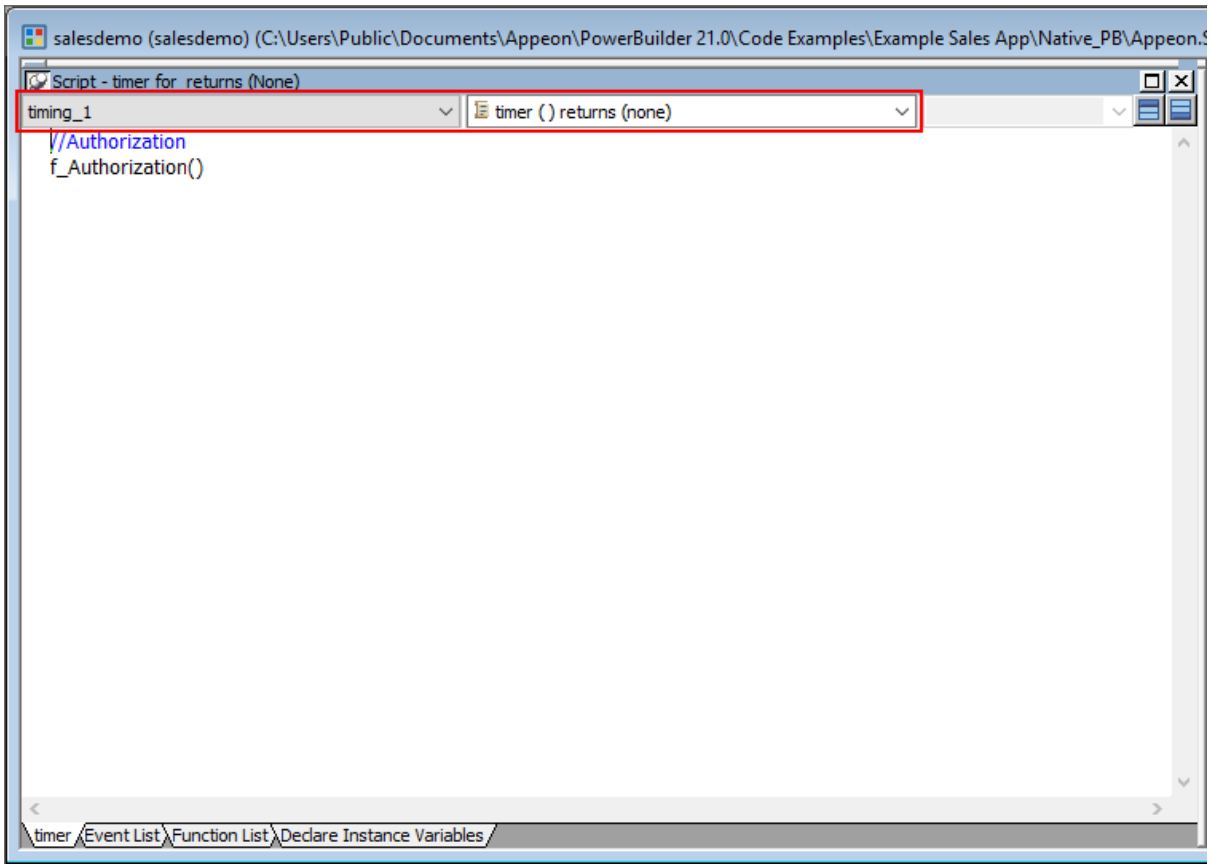
2) Add the following scripts to the **Timer** event of **timing_1**.

```
//Authenticates the user
f_Authorization()
```

When displayed in the source editor, the Timer event looks like this:

```
event timer;//Authenticates the user
f_Authorization()
end event
```

Figure 6.24:



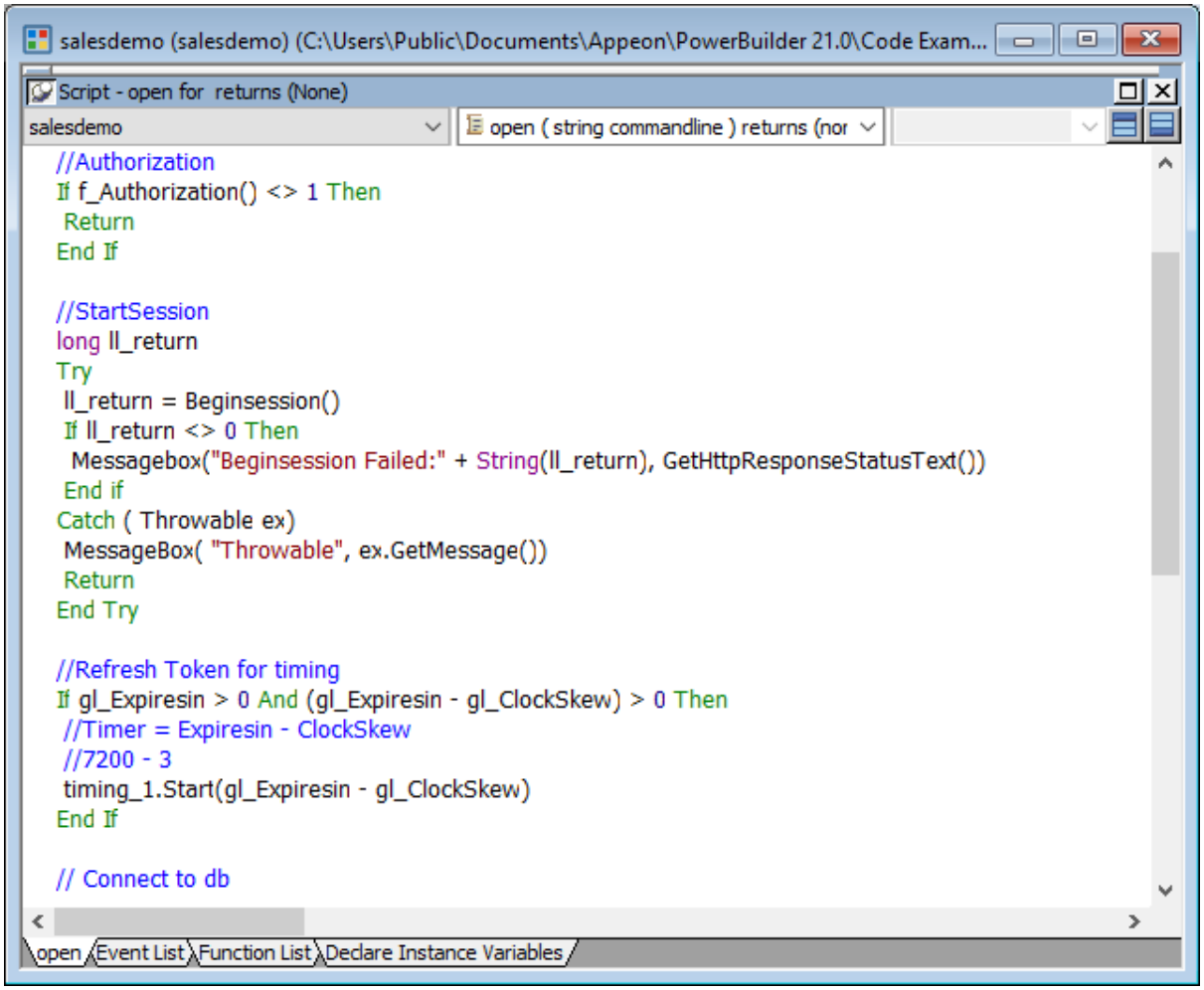
Step 4: Add the following scripts to the application **Open** event.

Place the scripts before the database connection is established. The scripts get the token from Azure AD and then start the user session (using the **BeginSession** function) to include the token information in the session.

```
//Authenticates the user and returns the token
If f_Authorization() <> 1 Then
 Return
End If

//Starts the session
long ll_return
Try
 ll_return = Beginsession()
```

```
 If ll_return <> 0 Then
  Messagebox("Beginsession Failed:" + String(ll_return),
 GetHttpResponseStatusText())
 End if
Catch ( Throwable ex)
 MessageBox( "Throwable", ex.GetMessage())
 Return
End Try

//Refreshes the token for timing
If gl_Expiresin > 0 And (gl_Expiresin - gl_ClockSkew) > 0 Then
 //Timer = Expiresin - ClockSkew
 //7200 - 3
 timing_1.Start(gl_Expiresin - gl_ClockSkew)
End If

//Connects to db
```

Figure 6.25:



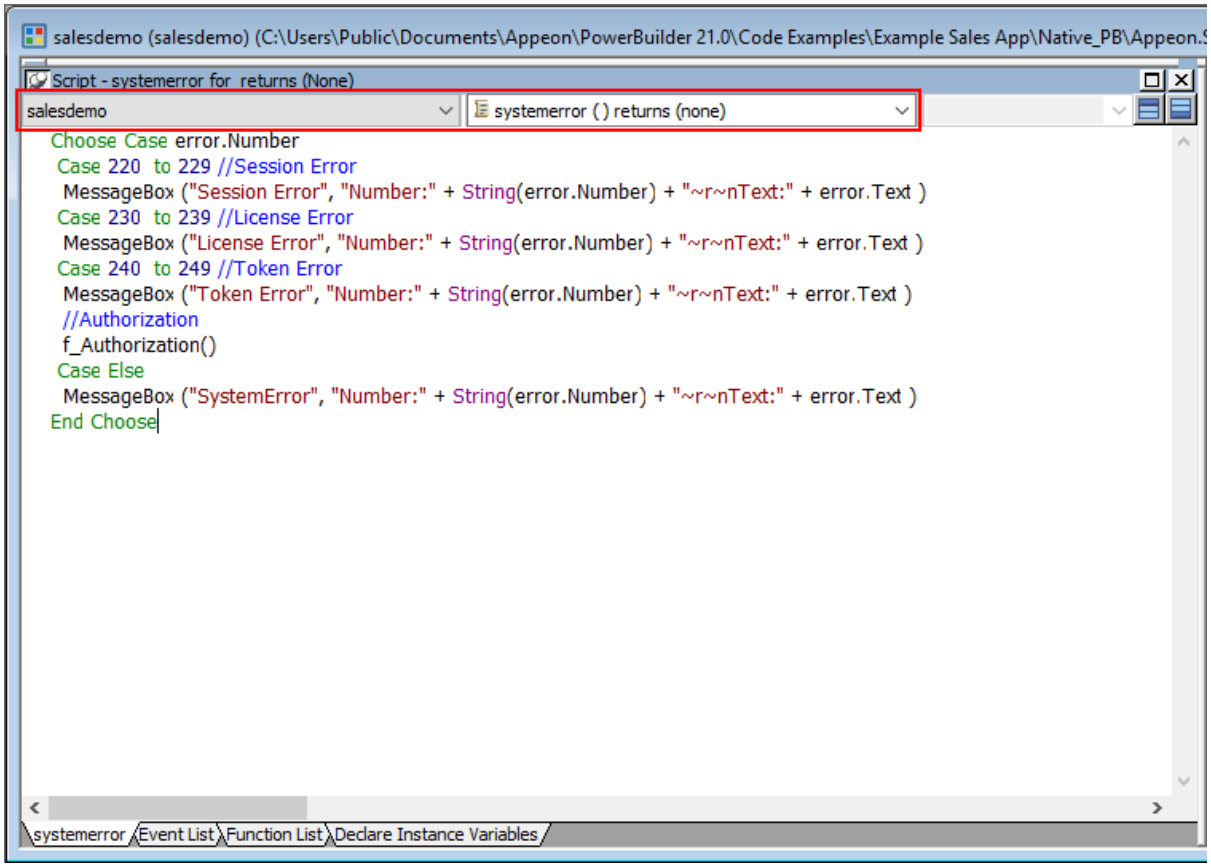
Step 5: Add the following scripts to the **SystemError** event.

The scripts will trigger the **SystemError** event when the session or license encounters an error; and if the token is invalid or expires, the scripts will call the **f_Authorization** function to get the token again.

```
Choose Case error.Number
 Case 220  to 229 //Session Error
  MessageBox ("Session Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
```

```
 Case 230  to 239 //License Error
  MessageBox ("License Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 240  to 249 //Token Error
  MessageBox ("Token Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
  //Authorization
  f_Authorization()
 Case Else
  MessageBox ("SystemError", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
End Choose
```

Figure 6.26:



6.5.1.3.3 Add an INI file

Create an INI file in the same location as the PBT file and name it **CloudSetting.ini**.

The INI file specifies the URL for requesting the token from Azure AD.

```
[Setup]
TokenURL=https://login.microsoftonline.com/0ffb9ae0-c080-4913-aa94-ed08b5de4d40/
oauth2/v2.0/token
```

6.5.1.3.4 Start session manually by code

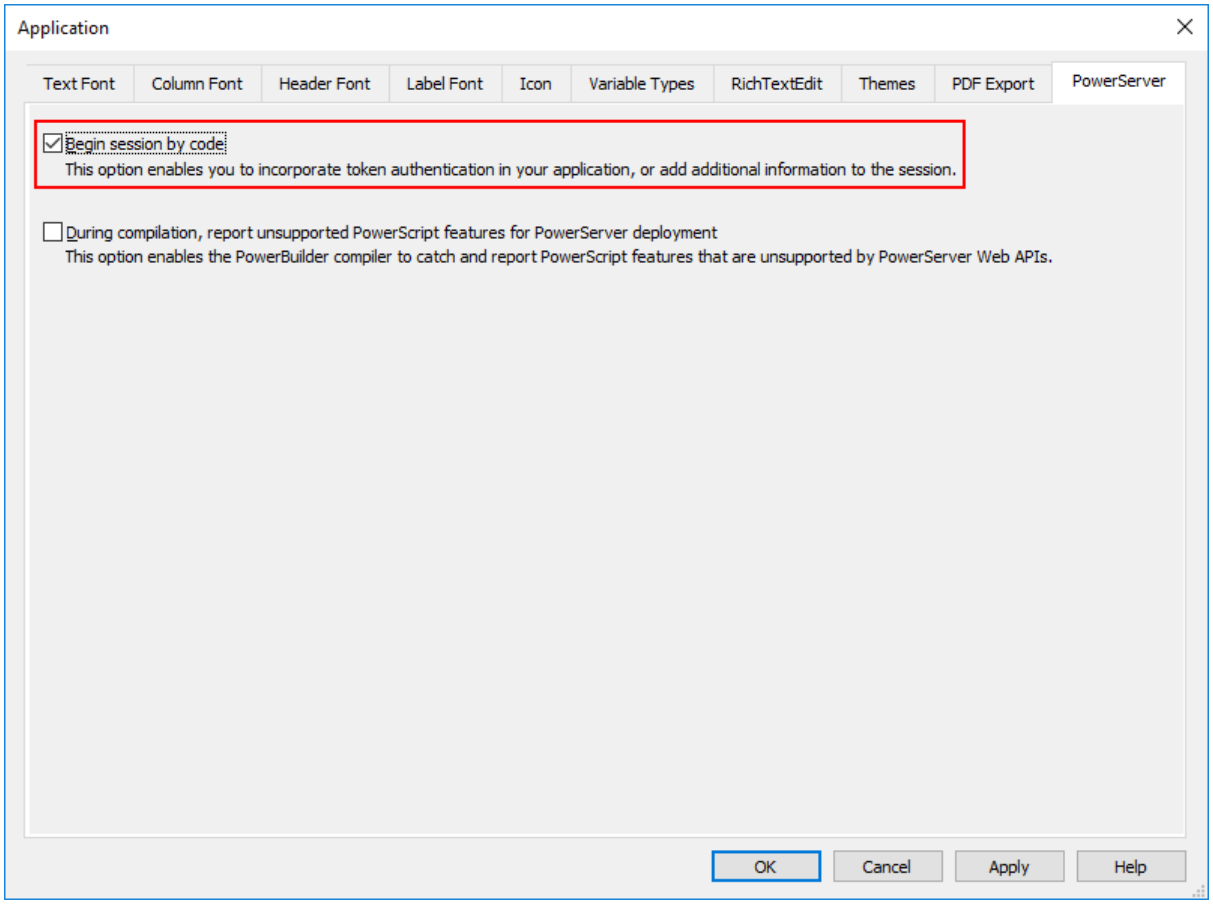
By default, the user session is automatically created when the application starts; and the session includes no token. For the session to include the token, the session must be started manually by code instead of automatically.

To start the session manually by code,

Step 1: Enable "**Begin session by code**" in the PowerBuilder IDE. (Steps: Open the application object painter, click **Additional Properties** in the application's **Properties** dialog; in the **Application** dialog, select the **PowerServer** tab and then select the **Begin session by code** option, and click **Apply**.)

After this option is enabled, when the **BeginSession** function in the application **Open** event is called, it will create a session that includes the token information (See scripts in step 4 in "Add scripts").

Figure 6.27:

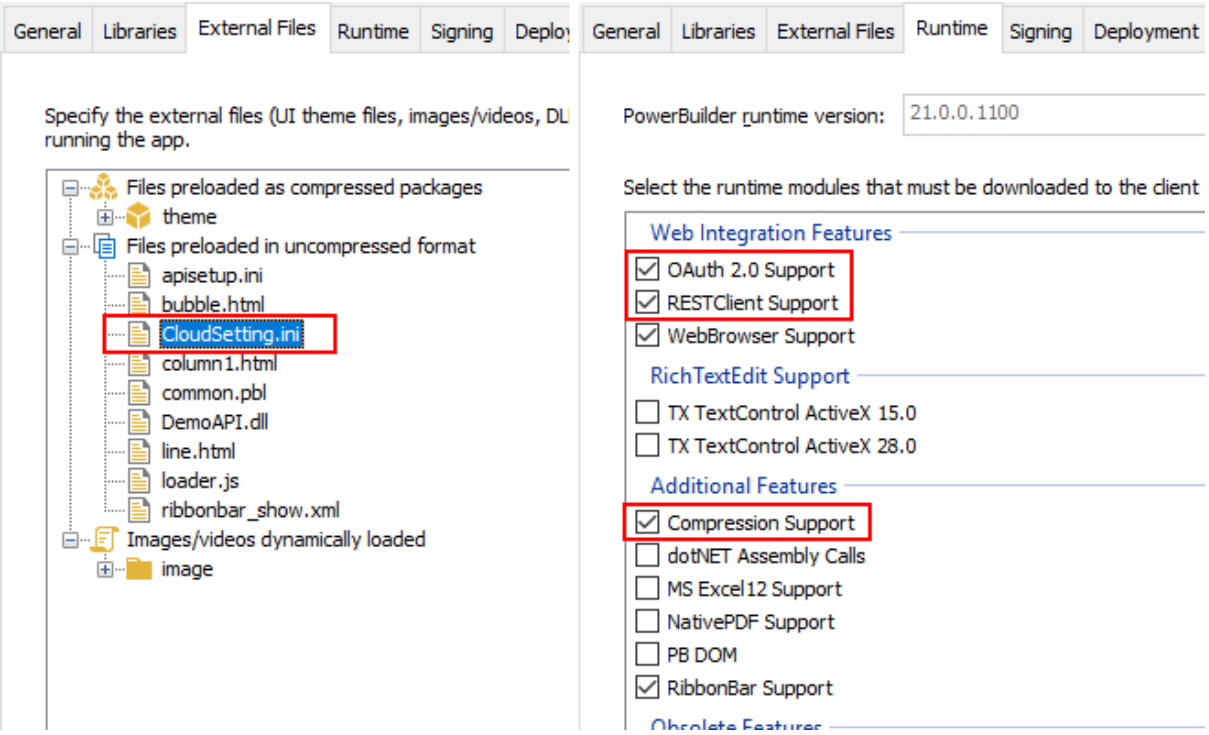


6.5.1.3.5 Modify and re-deploy the PowerServer project

Step 1: Add the INI file **CloudSetting.ini** to the **Files preloaded in uncompressed format** section under the **External Files** tab.

Step 2: Select **OAuth 2.0 Support**, **RESTClient Support** and **Compression Support** under the **Runtime** tab.

Figure 6.28:



Step 3: Double check the URL of the PowerServer Web APIs in the **Web APIs** tab. Make sure the port number is not occupied by any other program.

Tip: You can execute the command "netstat -ano | findstr *portnumber*" to check if the port number is occupied by any other program.

Step 4: Double check that **Use external auth service** is selected from the **Auth Template** list box in the **Web APIs** tab.

Step 5: Save the changes and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) so that the above settings can take effect in the installable cloud app.

6.5.1.4 Modifying the authentication template

The Azure AD server address must be provided so that the PowerServer Web APIs can use it to validate the token passed from the client. And if validation is successful, it can get data from the database.

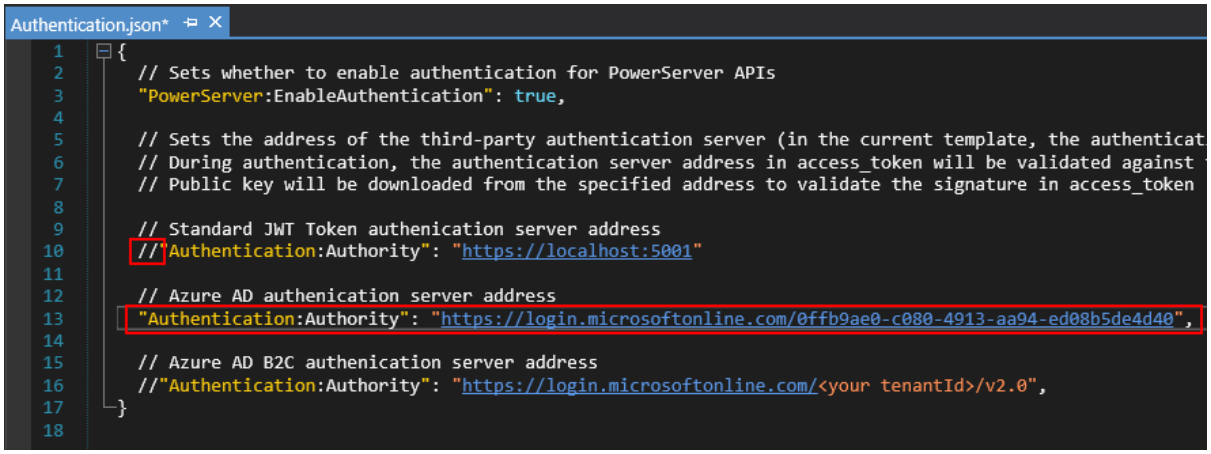
Note

The authentication template will be restored if the "**Auth Template**" option is changed and the PowerServer C# solution is re-built from the PowerBuilder IDE. Therefore, do not change the "**Auth Template**" option if you have made changes to the template in the solution.

Open the **Authentication.json** file, comment the server address for standard JWT, uncomment the server address for Azure AD, and specify the Azure AD tenant ID that will be used to validate the token passed from the client.

```
// Azure AD authentication server address
"Authentication:Authority": "https://login.microsoftonline.com/0ffb9ae0-c080-4913-
aa94-ed08b5de4d40",
```

Figure 6.29:



6.5.2 Azure Active Directory (AD) B2C

6.5.2.1 Preparations

Before making changes to the PowerBuilder client app, let's follow the steps below to make sure 1) the PowerBuilder application can run successfully, 2) the app has been deployed as an installable cloud app successfully, and 3) the PowerServer C# solution has been successfully generated.

In this tutorial, we will take Sales Demo as an example.

Step 1: Select Windows **Start | Appeon PowerBuilder 2021**, and then right-click **Example Sales App** and select **More | Run as administrator**.

Step 2: When the SalesDemo workspace is loaded in the PowerBuilder IDE, click the **Run** button in the PowerBuilder toolbar.

Step 3: When the application main window is opened, click the **Address** icon in the application ribbon bar and make sure data can be successfully retrieved.

Step 4: Create and configure a PowerServer project for the Sales Demo app (detailed instructions are provided in the Quick Start guide).

IMPORTANT: In the **Web APIs** tab, select **Use external auth service** from the **Auth Template** list box.

Step 5: Deploy the application as an installable cloud app and make sure the installable cloud app can run successfully and the PowerServer C# solution is generated.

The PowerServer C# solution provides templates for configuring the address of the authentication server like Azure AD or Azure AD B2C.

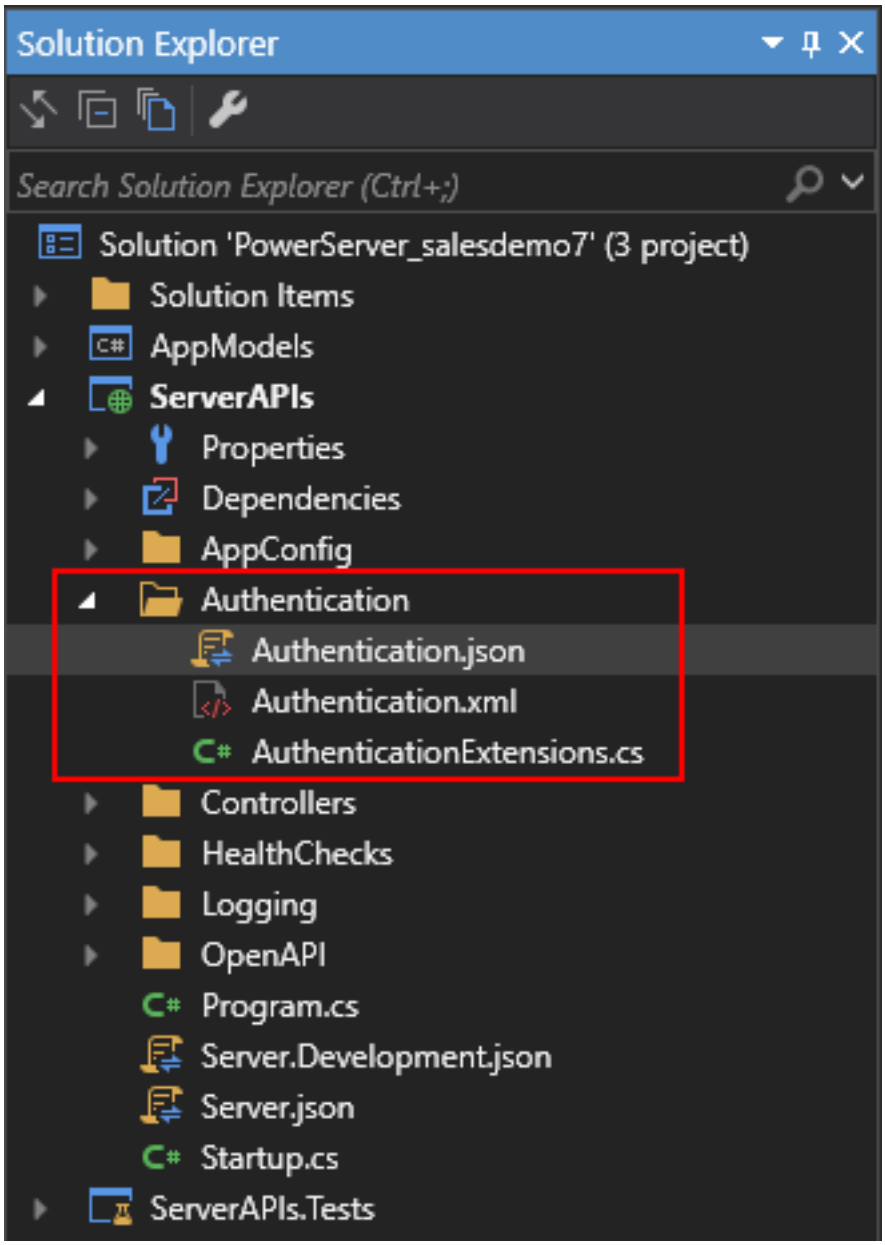
- **Authentication.json** contains the settings for enabling the authentication feature ("PowerServer:EnableAuthentication") and specifying the address of the authentication server ("Authentication:Authority"). The PowerServer Web APIs will validate the token

against the authentication server; and if validation is successful, data will be obtained from the database.

The "PowerServer:EnableAuthentication" setting is set to **true** by default. Setting it to **false** will turn off the authentication feature.

The "Authentication:Authority" setting is set for JWT by default; you can set the address of Azure AD and Azure AD B2C.

Figure 6.30:



6.5.2.2 Creating an Azure AD B2C tenant

The following outlines the key steps for setting up an Azure AD B2C tenant. For complete and detailed instructions, please refer to Tutorial: Create an Azure Active Directory B2C tenant.

During the process of creating the tenant, gather the following information:

- **Tenant ID**: for example, ed7837a1-96e2-4243-8ac8-172bc467f42c
- **Primary domain**: for example, powerserverb2c.onmicrosoft.com
- **Application (client) ID**: for example, ddaf52bf-1039-4f7a-ab85-51a219c1d4d7
- **Client secret**: for example, VgJo8X8qu4nCW.gf.FRxe.lhBZfE9F6.MA
- **Application ID URI**: for example, https://powerserverb2c.onmicrosoft.com/ddaf52bf-1039-4f7a-ab85-51a219c1d4d7
- **Scope**: for example, https://powerserverb2c.onmicrosoft.com/ddaf52bf-1039-4f7a-ab85-51a219c1d4d7/.default

The above information will be used later.

6.5.2.3 Modifying the PowerBuilder client app

6.5.2.3.1 Purpose

In this section, we will modify the PowerBuilder application source code and the PowerServer project settings to achieve the following results:

- Gets the user credential from the application login window, then authenticates it with the Azure AD B2C tenant and gets a token.
- Uses the token to access data from the PowerServer Web API.
- Refreshes the token when necessary.

6.5.2.3.2 Add scripts

Step 1: Declare the following global variables.

```
//Token expiresin
Long gl_Expiresin
//Refresh token clockskew
Long gl_ClockSkew = 3
```

Step 2: Define a global function and name it **f_Authorization()**.

Select from menu **File** > **New**; in the **New** dialog, select the **PB Object** tab and then select **Function** and click **OK** to add a global function.

This global function uses the HTTP Post method to send the user credentials to the authorization server and then gets the identity token from the HTTP Authorization header.

Add scripts to the **f_Authorization()** function to implement the following scenario:

- Scenario 1: Supports Client Credentials (GrantType="client_credentials") and gets the client ID and secret from the application.
- Scenario 2: Supports Resource Owner Password (GrantType="password") and gets the username and password from a login window.

Scripts for scenario 1:

When the application starts, the application uses the client ID and secret stored in the application to get the token from Azure AD B2C, and when the token expires, it automatically refreshes the token.

```
//Integer f_Authorization() for client_credentials
//UserName & Password from login window
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String  ls_url, ls_UserName, ls_UserPass
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")
//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "ddaf52bf-1039-4f7a-ab85-51a219c1d4d7"
ltr_Request.clientsecret = "VgJo8X8qu4nCW.gf.FRxe.lhBZfE9F6.MA"
ltr_Request.granttype = "client_credentials"
ltr_Request.Scope = "https://powerserverb2c.onmicrosoft.com/ddaf52bf-1039-4f7a-
ab85-51a219c1d4d7/.default"

loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " " +
 ls_AccessToken, true)
 //Set Global Variables
 gl_Expiresin = ltr_Response.getexpiresin( )

 li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Scripts for scenario 2:

When the application starts, the client ID and secret stored in the application as well as the username and password from the login window will be sent to Azure AD B2C to get the token, and when the token expires, the login window displays for the user to input the username and password again.

The following scripts hard code the username and password instead of getting them from the login window. You can change the scripts to use the login window after you implement the login window and return the username and password to the **f_Authorization()** function.

```
//Integer f_Authorization() for password
//UserName & Password from login window
```

```
OAuthClient    loac_Client
TokenRequest   ltr_Request
TokenResponse  ltr_Response
String  ls_url, ls_UserName, ls_UserPass
String  ls_TokenType, ls_AccessToken
String  ls_type, ls_description, ls_uri, ls_state
Integer li_Return, li_rtn

li_rtn = -1
ls_url = profilestring("CloudSetting.ini","setup","TokenURL","")

//TokenRequest
ltr_Request.tokenlocation = ls_url
ltr_Request.Method = "POST"
ltr_Request.clientid = "ddaf52bf-1039-4f7a-ab85-51a219c1d4d7"
ltr_Request.clientsecret = "VgJo8X8qu4nCW.gf.FRxe.lhBZfE9F6.MA"
ltr_Request.scope = "https://powerserverb2c.onmicrosoft.com/ddaf52bf-1039-4f7a-
ab85-51a219c1d4d7/.default"
ltr_Request.granttype = "password"

//login window can be implemented to return username & password according to actual
 needs
//Open(w_login)
//Return UserName & Password

ls_UserName = "appeontest"
ls_UserPass = "Test2008aa"

If IsNull ( ls_UserName ) Or Len ( ls_UserName ) = 0 Then
 MessageBox( "Tips", "UserName is empty!" )
 Return li_rtn
End If
If IsNull ( ls_UserPass ) Or Len ( ls_UserPass ) = 0 Then
 MessageBox( "Tips", "Password is empty!" )
 Return li_rtn
End If

ltr_Request.UserName = ls_UserName
ltr_Request.Password = ls_UserPass

loac_Client = Create OAuthClient
li_Return = loac_Client.AccessToken( ltr_Request, ltr_Response )
If li_Return = 1 and ltr_Response.GetStatusCode () = 200 Then
 ls_TokenType = ltr_Response.gettokentype( )
 ls_AccessToken = ltr_Response.GetAccessToken()
 //Application Set Authorization Header
 Getapplication().SetHttpRequesTheader("Authorization", ls_TokenType + " "
 +ls_AccessToken, true)
 //Set Global Variables
 gl_Expiresin = ltr_Response.getexpiresin( )

 li_rtn = 1
Else
 li_Return = ltr_Response.GetTokenError(ls_type, ls_description, ls_uri, ls_state)
 MessageBox( "AccessToken Falied", "Return :" + String ( li_Return ) + "~r~n" +
 ls_description )
End If

If IsValid ( loac_Client ) Then DesTroy ( loac_Client )

Return li_rtn
```

Step 3: Insert a timing object (**timing_1**) to the application and add the following scripts to the **Timer** event of **timing_1**.

1) Open the application object and then select from menu **Insert** > **Object** > **Timing** to add a timing object to the application.

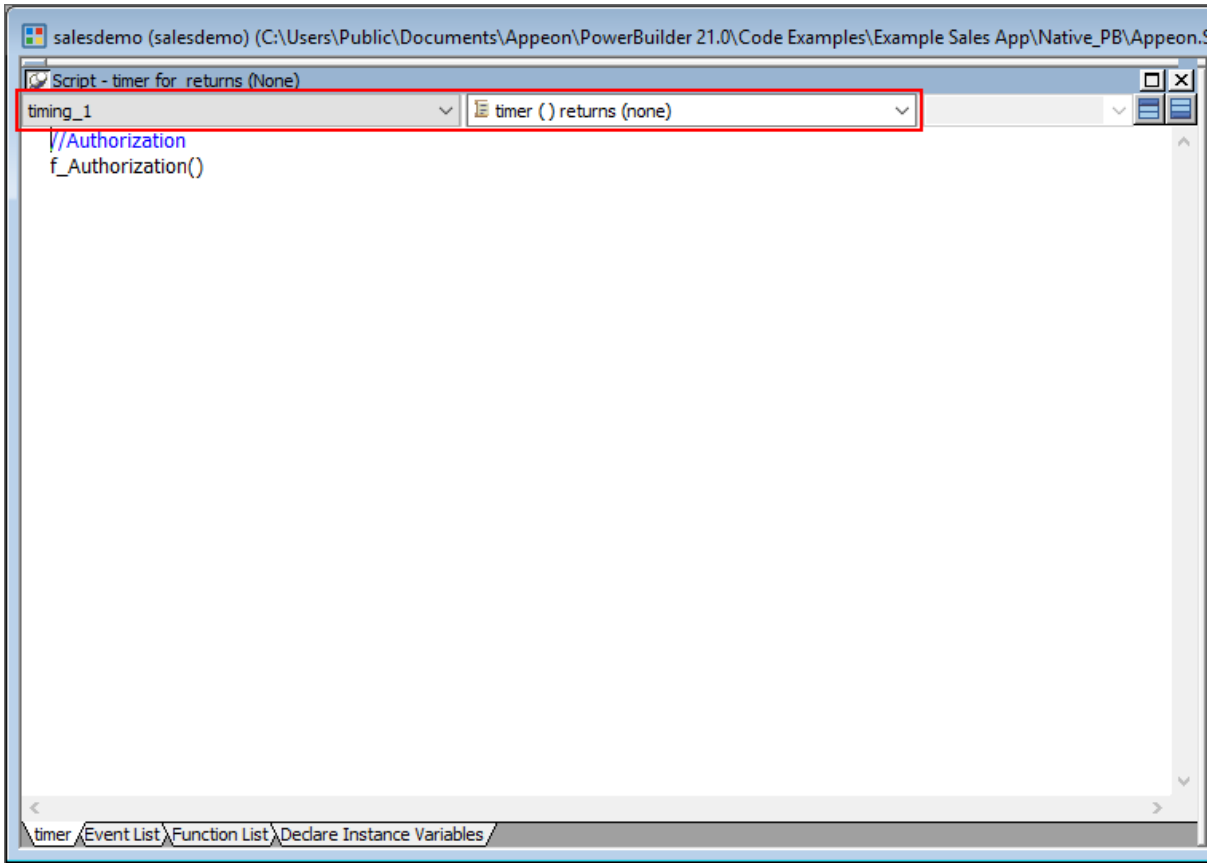
2) Add the following scripts to the **Timer** event of **timing_1**.

```
//Authenticates the user
f_Authorization()
```

When displayed in the source editor, the Timer event looks like this:

```
event timer;//Authenticates the user
f_Authorization()
end event
```

Figure 6.31:



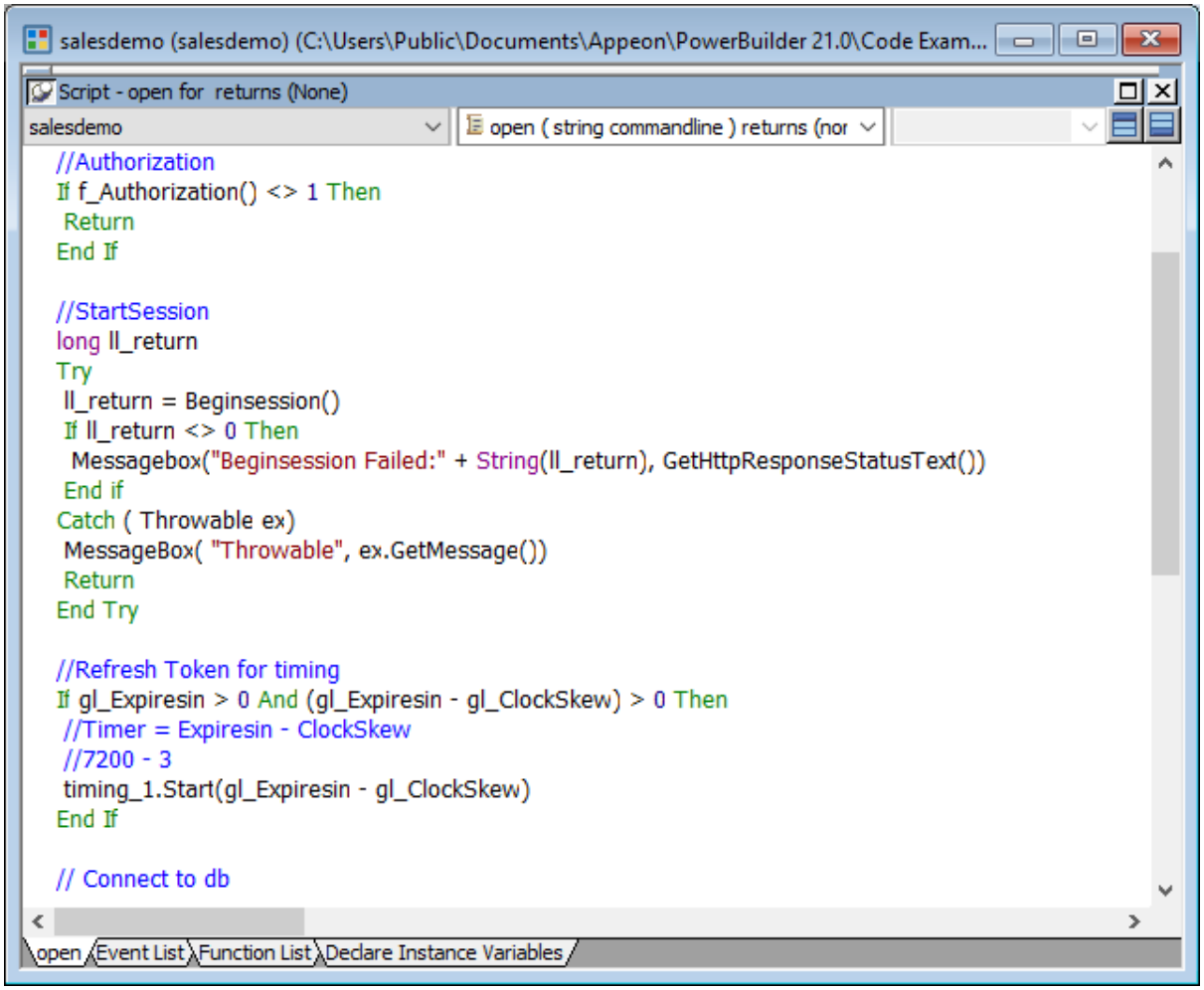
Step 4: Add the following scripts to the application **Open** event.

Place the scripts before the database connection is established. The scripts get the token from Azure AD B2C and then start the user session (using the **BeginSession** function) to include the token information in the session.

```
//Authenticates the user and returns the token
If f_Authorization() <> 1 Then
 Return
End If

//Starts the session
long ll_return
Try
 ll_return = Beginsession()
```

```
 If ll_return <> 0 Then
  Messagebox("Beginsession Failed:" + String(ll_return),
 GetHttpResponseStatusText())
 End if
Catch ( Throwable ex)
 MessageBox( "Throwable", ex.GetMessage())
 Return
End Try

//Refreshes the token for timing
If gl_Expiresin > 0 And (gl_Expiresin - gl_ClockSkew) > 0 Then
 //Timer = Expiresin - ClockSkew
 //7200 - 3
 timing_1.Start(gl_Expiresin - gl_ClockSkew)
End If

//Connects to db
```

Figure 6.32:



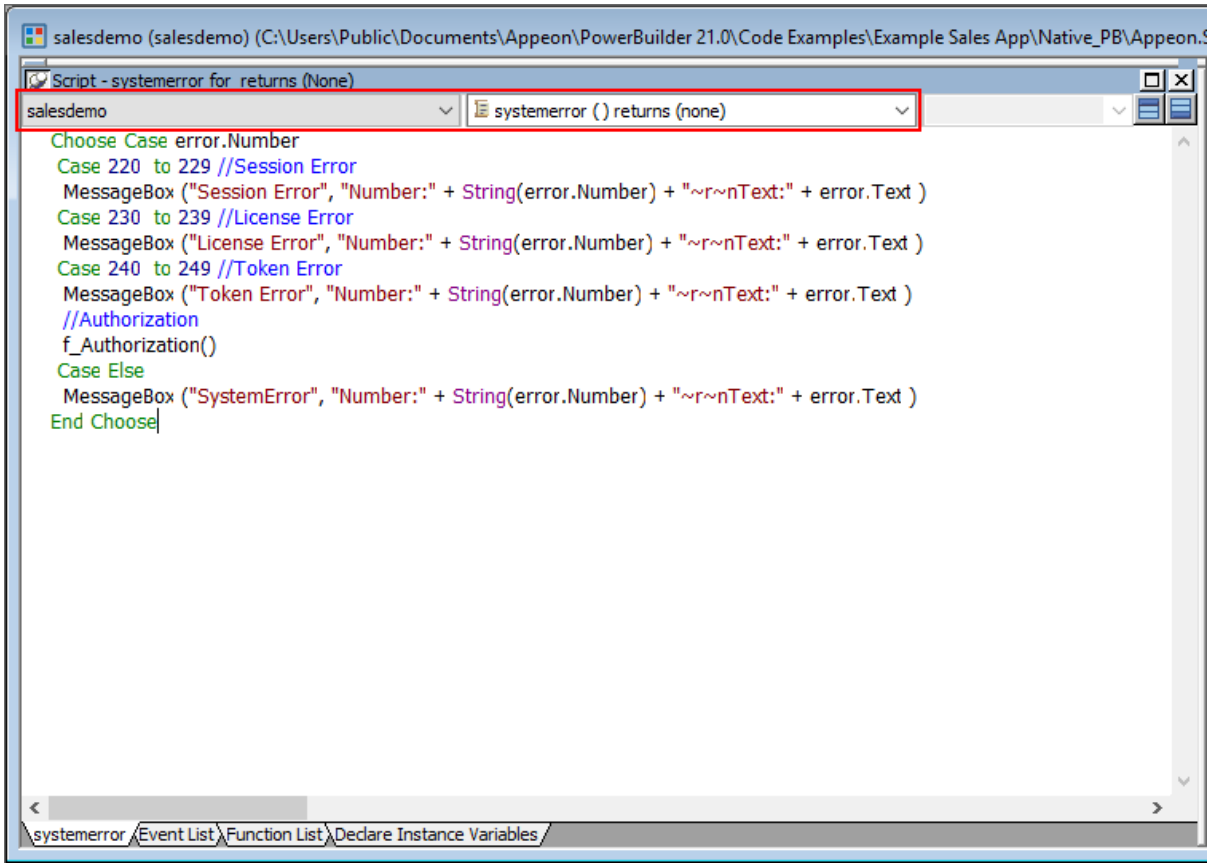
Step 5: Add the following scripts to the **SystemError** event.

The scripts will trigger the **SystemError** event when the session or license encounters an error; and if the token is invalid or expires, the scripts will call the **f_Authorization** function to get the token again.

```
Choose Case error.Number
 Case 220  to 229 //Session Error
```

```
 MessageBox ("Session Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 230  to 239 //License Error
  MessageBox ("License Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
 Case 240  to 249 //Token Error
  MessageBox ("Token Error", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
  //Authorization
  f_Authorization()
 Case Else
  MessageBox ("SystemError", "Number:" + String(error.Number) + "~r~nText:" +
 error.Text )
End Choose
```

Figure 6.33:



6.5.2.3.3 Add an INI file

Create an INI file in the same location as the PBT file and name it **CloudSetting.ini**.

The INI file specifies the URL for requesting the token from Azure AD B2C.

```
[Setup]
TokenURL=https://login.microsoftonline.com/powerserverb2c.onmicrosoft.com/oauth2/
v2.0/token
```

6.5.2.3.4 Start session manually by code

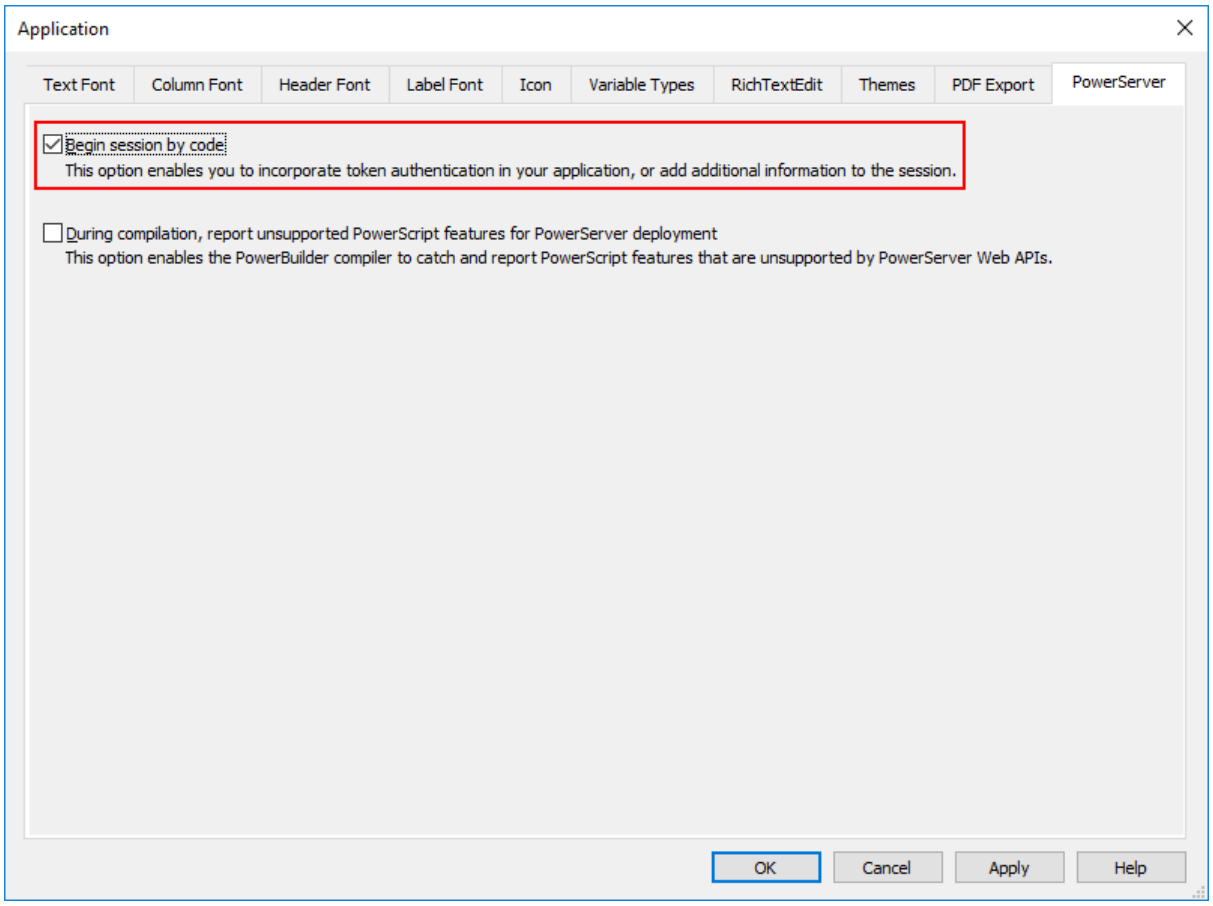
By default, the user session is automatically created when the application starts; and the session includes no token. For the session to include the token, the session must be started manually by code instead of automatically.

To start the session manually by code,

Step 1: Enable "**Begin session by code**" in the PowerBuilder IDE. (Steps: Open the application object painter, click **Additional Properties** in the application's **Properties** dialog; in the **Application** dialog, select the **PowerServer** tab and then select the **Begin session by code** option, and click **Apply**.)

After this option is enabled, when the **BeginSession** function in the application **Open** event is called, it will create a session that includes the token information (See scripts in step 4 in "Add scripts").

Figure 6.34:

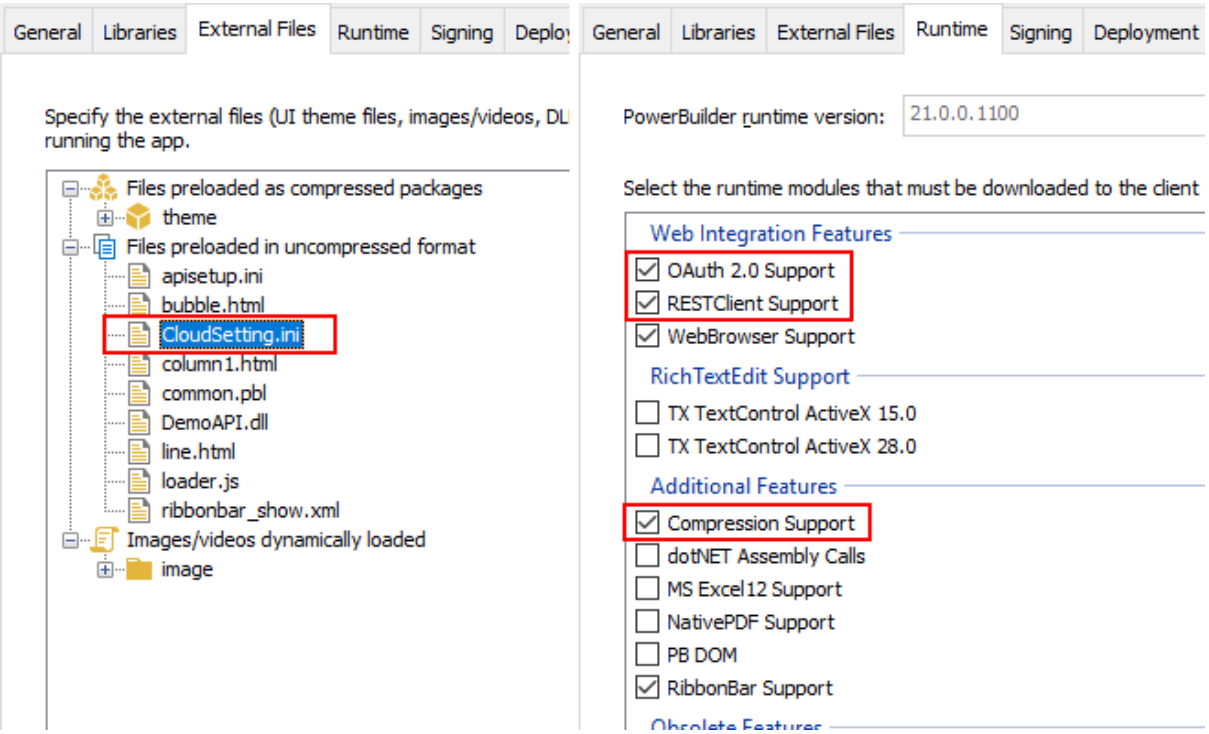


6.5.2.3.5 Modify and re-deploy the PowerServer project

Step 1: Add the INI file **CloudSetting.ini** to the **Files preloaded in uncompressed format** section under the **External Files** tab.

Step 2: Select **OAuth 2.0 Support**, **RESTClient Support** and **Compression Support** under the **Runtime** tab.

Figure 6.35:



Step 3: Double check the URL of the PowerServer Web APIs in the **Web APIs** tab. Make sure the port number is not occupied by any other program.

Tip: You can execute the command "netstat -ano | findstr *portnumber*" to check if the port number is occupied by any other program.

Step 4: Double check that **Use external auth service** is selected from the **Auth Template** list box in the **Web APIs** tab.

Step 5: Save the changes and deploy the PowerServer project (using the "Build & Deploy PowerServer Project" option) so that the above settings can take effect in the installable cloud app.

6.5.2.4 Modifying the authentication template

The Azure AD B2C server address must be provided so that the PowerServer Web APIs can use it to validate the token passed from the client. And if validation is successful, it can get data from the database.

Note

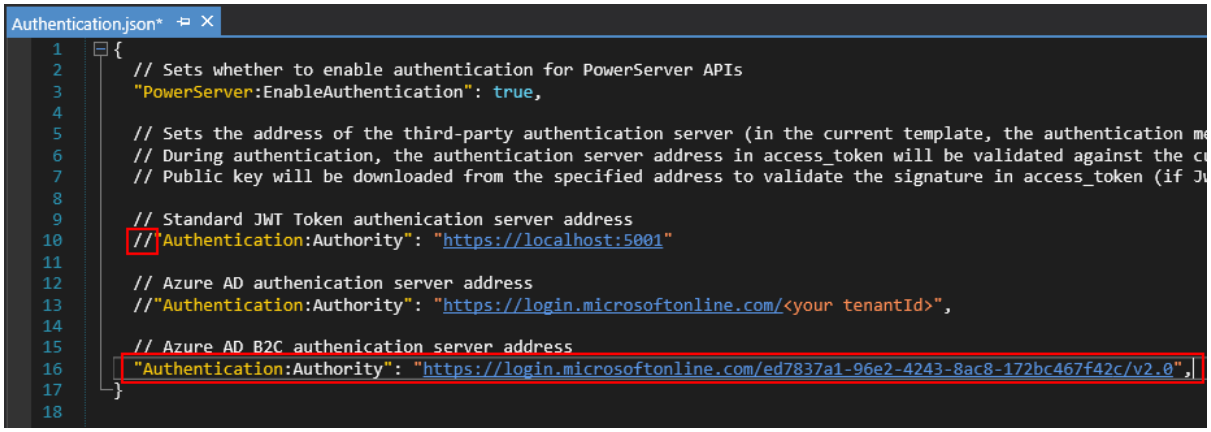
The authentication template will be restored if the "**Auth Template**" option is changed and the PowerServer C# solution is re-built from the PowerBuilder IDE. Therefore, do not change the "**Auth Template**" option if you have made changes to the template in the solution.

Open the **Authentication.json** file, comment the server address for standard JWT, uncomment the server address for Azure AD B2C, and specify the Azure AD B2C tenant ID that will be used to validate the token passed from the client.

```
// Azure AD B2C authentication server address
```

```
"Authentication:Authority": "https://login.microsoftonline.com/
ed7837a1-96e2-4243-8ac8-172bc467f42c/v2.0",
```

Figure 6.36:



7 Tutorial 7: Building your PowerServer project with commands

Besides building and deploying your PowerServer project in the PowerBuilder IDE, you can also build and deploy your PowerServer project with commands (**PBAutoBuild210.exe**).

7.1 Task 1: Preparing the environment

Step 1: Install the following software on the same machine:

- Windows 10 (64-bit)
- PowerBuilder IDE 2021
- PowerBuilder Runtime 2021
- PowerBuilder Compiler 2021
- PowerServer Toolkit 2021

Step 2: Prepare the database driver if the **MySQL**, **Oracle**, or **Informix** database connection is required.

If the **MySQL**, **Oracle**, or **Informix** database connection is used and if the corresponding database driver has not been downloaded yet, you will need to manually download the database driver from the NuGet website.

- For MySQL, download MySql.Data 8.0.25 and the license file, and unzip the files to %USERPROFILE%\.sd\19.0\dbDrives\MySql.Data\8.0.25.
- For Oracle, download Oracle.ManagedDataAccess.Core 2.19.110 and the license file, and unzip the files to %USERPROFILE%\.sd\19.0\dbDrives\Oracle.ManagedDataAccess.Core\2.19.110.
- For Informix, download IBM.Data.DB2.Core 2.2.0.100 and the license file, and unzip the files to %USERPROFILE%\.sd\19.0\dbDrives\IBM.Data.DB2.Core\2.2.0.100.

They will be automatically loaded by **PBAutoBuild210.exe** when creating the database connection.

7.2 Task 2: Exporting the build file

This tutorial will assume that you have already followed the Quick Start guide to

1. create a PowerServer project for the **Example Sales Demo**;
2. build and deploy the PowerServer project (using the **Build & Deploy PowerServer Project** option in the PowerBuilder IDE) successfully; and
3. run the installable cloud app successfully.

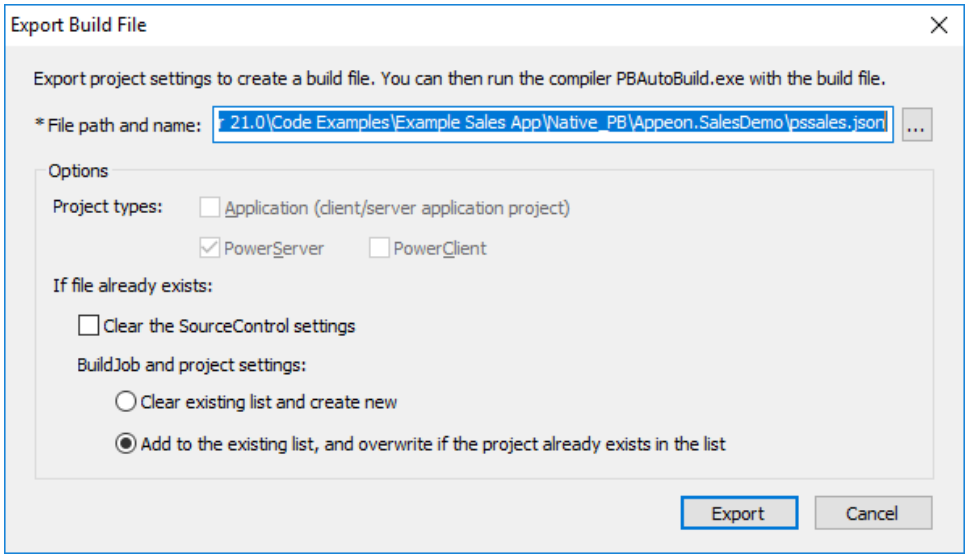
Now let's export the PowerServer project settings of the **Example Sales Demo** to a JSON file:

Step 1: Open the workspace for the **Example Sales App** in the PowerBuilder IDE, and then double-click the PowerServer project object to open the PowerServer project painter.

Step 2: When the PowerServer project painter is opened, click the **Export PowerServer Build File** button ( ) in the toolbar.

Step 3: In the **Export Build File** dialog box, write down the path and filename to be exported, and then click **Export**.

Figure 7.1:



7.3 Task 3 (Optional): Configuring the build file

The exported build file contains all the settings required for building and deploying the PBL files of the **Example Sales App**. It also contains some advanced settings that allow you to:

- Get and merge source code from SVN, Git, or VSS;
- Execute additional commands during the build process.

7.3.1 Getting source code from SVN, Git, or VSS

You can configure the exported build file to download source code from SVN, Git, or VSS before the build process starts.

Step 1: Make a copy of the exported build file and place it to a location near your PowerBuilder application target, so that you could manage the file path (especially the relative file path) easily.

Note: the relative path will be relative to the build file.

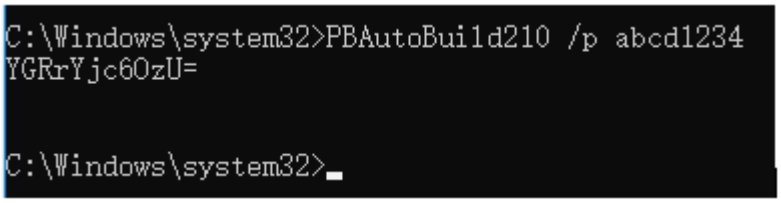
Step 2: Open the build file in a text editor, locate the "BuildPlan" block and then configure the corresponding part, for example, "Git" as shown below.

If the computer connects to Internet through a proxy server, make sure to configure the proxy server settings in the "**Proxy**" part.

```
    "BuildPlan": {
        "SourceControl": {
            "PreCommand": "",
            "SVN": [
                {"SrcPath": "","User": "","Password": "","DestPath": "","Proxy":
 {"Ip": "","Port": 0,"Username": "","Password": ""}}
            ],
            "Git": [
                {"SrcPath": "https://github.com/Appeon/PowerBuilder-AutoBuild-
Sales-SourceCode", "User": "tester@appeon.com", "Password": "YGRrYjc6OzU=",
 "DestPath": ".\\Build", "Proxy": {"Ip": "","Port": 0,"Username": "","Password":
 ""}}
            ],
```

Note: The password must be an encrypted value which is generated from the original password by executing "PBAutoBuild210.exe /p", as shown below.

Figure 7.2:



Step 3: If the source code downloaded from SVN, Git, or VSS is not the PBL file but objects in ws_objects, then you will need to merge the objects to the PBL file. Locate the "Merging" block in the build file and then configure as below:

Setting "RefreshPbl" to true if you want to refresh the PBL files by deleting and then generating the PBL files again.

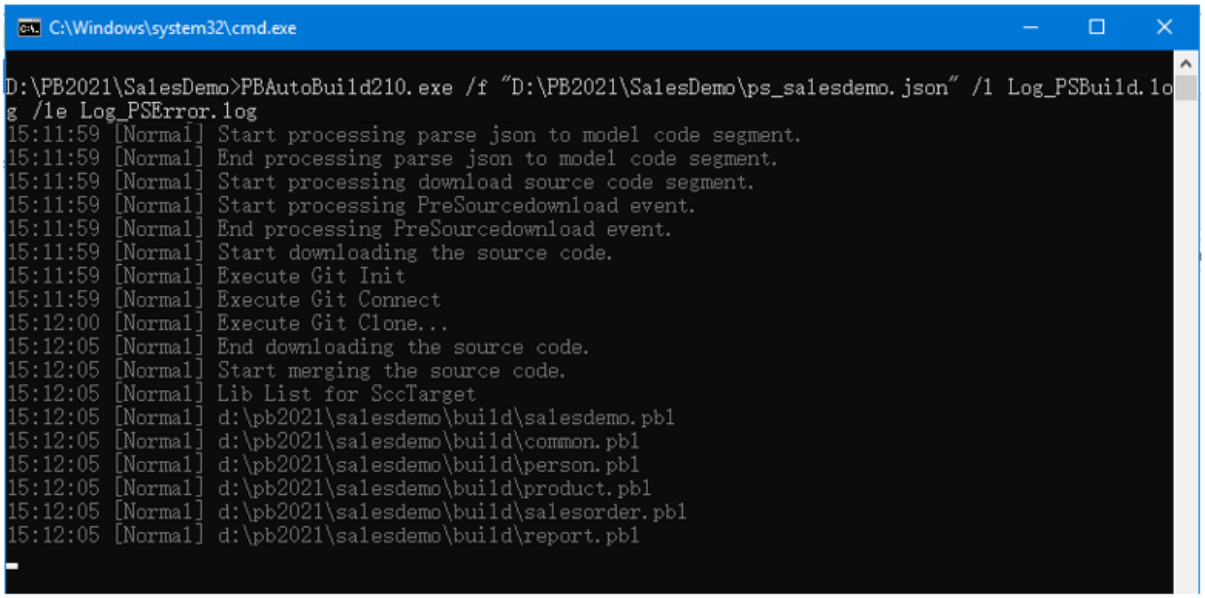
```
            "Merging": [
                {"Target": ".\\Build\\salesdemo.pbt", "LocalProjectPath": ".\
\Build", "RefreshPbl": false}
            ],
```

At the same time, make sure to double check the target location is set correctly in the "Projects" block, for example,

```
        "BuildJob": {
            "PreCommand": "",
            "Projects": [
                {"Target": ".\\Build\\salesdemo.pbt", "Name": "ps_salesdemo"}
            ],
            "PostCommand": ""
        }
```

When the PBAutoBuild210.exe command is executed later, it will first download the source code from the Git server and then merge the source code, as shown below.

Figure 7.3:



7.3.2 Executing additional commands

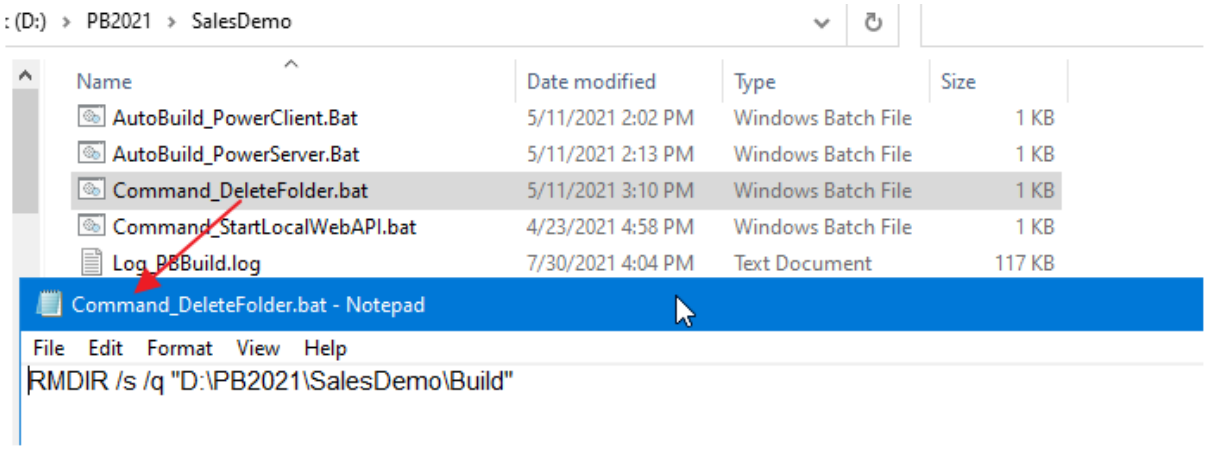
The entire build & deploy process is made up of several steps, and additional commands can be executed before and/or after some particular steps such as the "SourceControl" and "BuildJob" steps.

Example 1: to add commands to remove the "build" folder before downloading the source code.

Step 1: Create a bat file which contains the following command, and save the bat file as **Command_DeleteFolder.bat**.

```
RMDIR /s /q "D:\PB2021\SalesDemo\Build"
```

Figure 7.4:



Step 2: In the build file, locate the "BuildPlan" block and then the "PreCommand" setting; and add the file path and name of **Command_DeleteFolder.bat**.

```
    "BuildPlan": {
        "SourceControl": {
            "PreCommand": "Command_DeleteFolder.bat",
            "SVN": [
```

```
                ...
            ],
```

When the **PBAutoBuild210.exe** command is executed later, it will execute the commands in **Command_DeleteFolder.bat** before it downloads the source code.

Besides the "PreCommand" setting, there is also a "PostCommand" setting for the "SourceControl" and "BuildJob" steps, which allows you to execute commands after that particular step.

Example 2: to add commands to start the PowerServer Web APIs after building the PowerServer project in the PowerBuilder IDE.

Step 1: Create a bat file which contains the following command, and save the bat file as **startwebapi.bat**.

```
dotnet.exe run --no -build --project C:\Users\appeon\source\repos
\PowerServer_salesdemo\ServerAPIs\ServerAPIs.csproj
```

Step 2: In the build file, locate the "BuildJob" block and then the "PostCommand" setting; and add the file path and name of **startwebapi.bat**.

```
        "BuildJob": {
            "PreCommand": "",
            "Projects": [
                ...
            ],
            "PostCommand": "startwebapi.bat /show / sync"
        }
```

When the **PBAutoBuild210.exe** command is executed later, it will execute the commands in **startwebapi.bat** after it finishes building the PowerServer project.

Example 3: to publish the PowerServer Web API after building the PowerServer project in the PowerBuilder IDE.

Step 1: Create a bat file which contains the following command, and save the bat file as **publish.bat**.

```
dotnet.exe publish C:\Users\appeon\source\repos\PowerServer_salesdemo\ServerAPIs
\ServerAPIs.csproj -c release -o C:\Publish
```

Step 2: In the exported build file, locate the "BuildJob" block and then the "PostCommand" setting; and add the file path and name of **publish.bat**.

```
        "BuildJob": {
            "PreCommand": "",
            "Projects": [
                ...
            ],
            "PostCommand": "publish.bat /show / sync"
        }
```

When the **PBAutoBuild210.exe** command is executed later, it will execute the commands in **publish.bat** after it finishes building the PowerServer project.

Note

The **dotnet** commands can also be integrated with Jenkins. See Task 5 for more details.

7.4 Task 4: Running the PBAutoBuild210.exe command

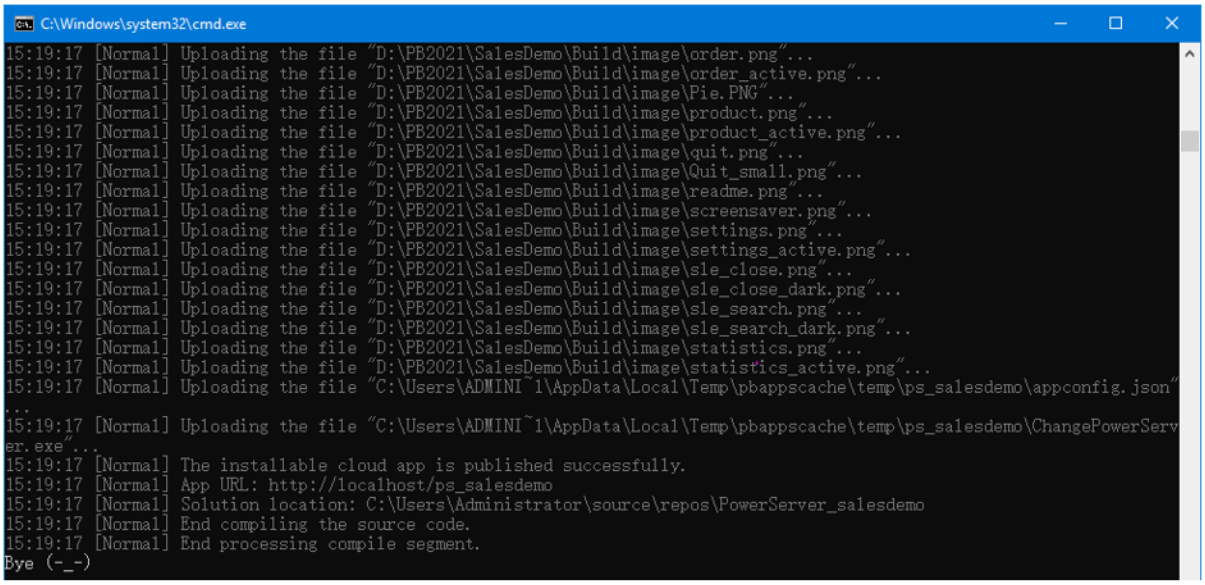
Step 1: In the command line window, execute the **PBAutoBuild210.exe** file and the build file. For example,

```
PBAutoBuild210.exe /f "D:\PB2021\SalesDemo\ps_salesdemo.json" /l Log_PSBuild.log /
le Log_PSError.log /lu Log_PSUnsupport.log
```

The **PBAutoBuild210.exe** file can be running with several parameters. For a complete list, refer to Build & deploy using commands.

Step 2: Carefully check the information in the command line window to make sure the build and deploy process is successful.

Figure 7.5:



The build file and commands used in this tutorial can be downloaded from https://github.com/Appeon/PowerBuilder-AutoBuild-Sales-Example. After you download these files to D:\PB2021\SalesDemo\, you can follow instructions in the readme file.

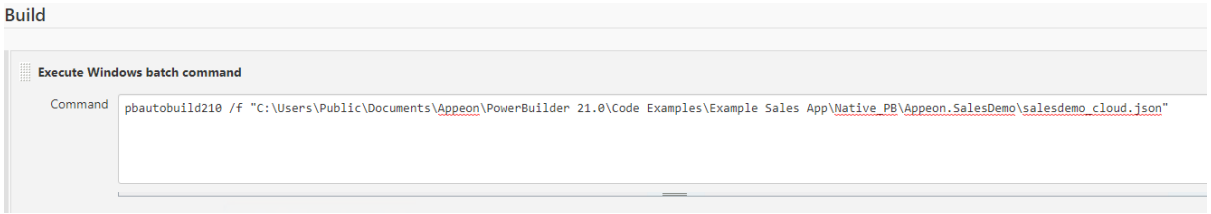
7.5 Task 5: Integrating with Jenkins

The **PBAutoBuild210** command can integrate with Jenkins to automate the build and deployment process for PowerServer projects. Refer to the Jenkins user documentations for how to use Jenkins.

Following gives a few examples on how to integrate the **PBAutoBuild210** and **dotnet** commands with Jenkins.

Example 1: to execute the PBAutoBuild210 command and the build file.

Figure 7.6:



Example 2: to download source code from SVN, Git, or VSS, and then execute the PBAutoBuild210 command and the build file.

Double check that the PBT location is the same one in all required areas.

Figure 7.7:

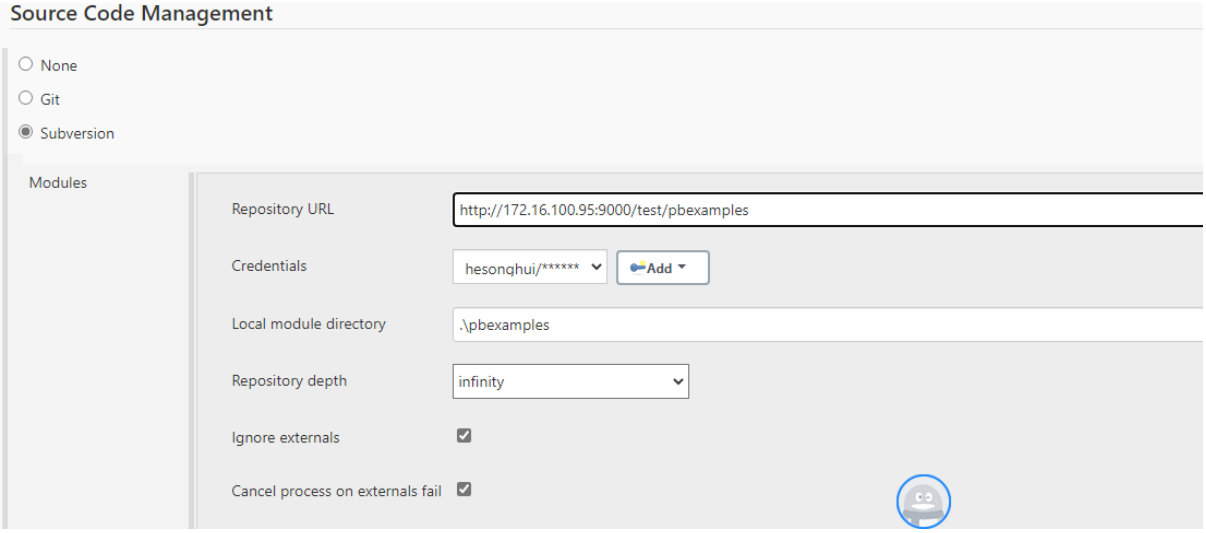
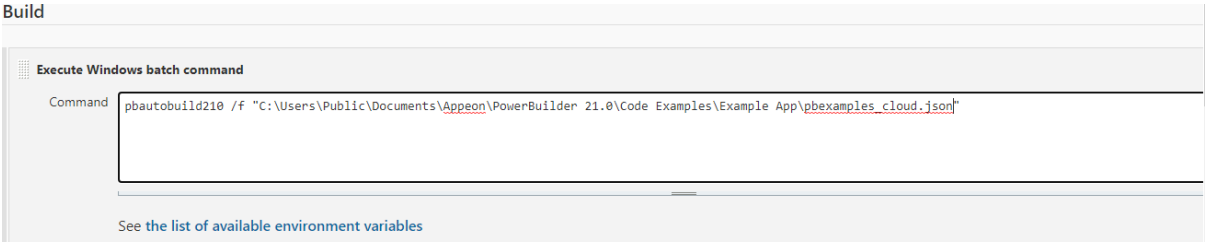


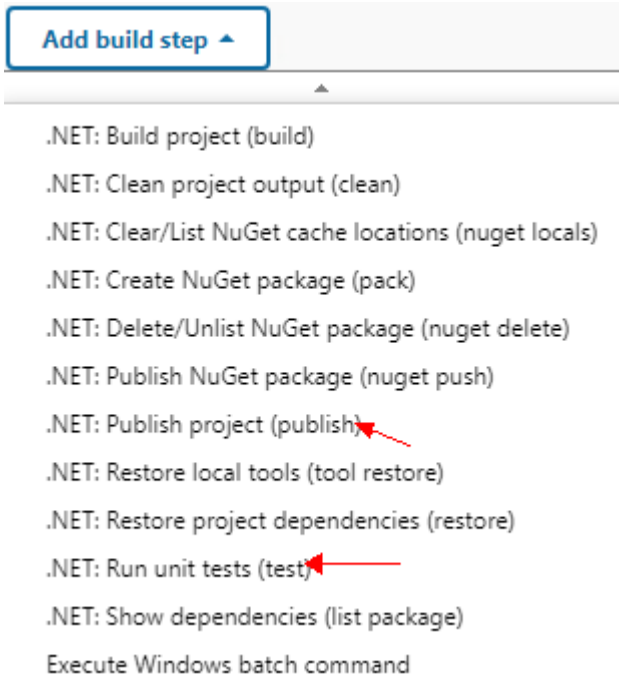
Figure 7.8:



Example 3: to publish or run the PowerServer Web APIs

You can integrate **dotnet** commands with Jenkins. After you install the .NET SDK Support plugin for Jenkins, the **dotnet** commands (for example, **dotnet publish**, **dotnet run** etc.) are available as shown below. Refer to https://www.jenkins.io/doc/pipeline/steps/dotnet-sdk/ for more details.

Figure 7.9:


Add build step
.NET: Build project (build)
.NET: Clean project output (clean)
.NET: Clear/List NuGet cache locations (nuget locals)
.NET: Create NuGet package (pack)
.NET: Delete/Unlist NuGet package (nuget delete)
.NET: Publish NuGet package (nuget push)
.NET: Publish project (publish)
.NET: Restore local tools (tool restore)
.NET: Restore project dependencies (restore)
.NET: Run unit tests (test)
.NET: Show dependencies (list package)
Execute Windows batch command

8 Tutorial 8: Creating a standalone installable package

Each PowerBuilder installable cloud application is composed of two parts:

- the application client-side which resides on the Web server
- the PowerServer Web APIs which resides on the .NET server

Therefore, to create a standalone installable package for the PowerBuilder installable cloud application, you need to create two packages:

- An executable installer or zipped file of the application client-side
- A distributable package of the PowerServer Web APIs

After that, you will need to set the Web API URL correctly to the application client-side, so that the application knows where to access the PowerServer Web APIs at runtime.

8.1 Packaging the client app

When deploying the PowerServer project as an installable cloud app, you can choose to package the client-side as an executable installer or a zipped file, and then install the client to the Web servers.

To package the client app:

1. Go to the **Client Deployment** tab of the PowerServer project painter, and then click **Package the compiled app and manually deploy later**.

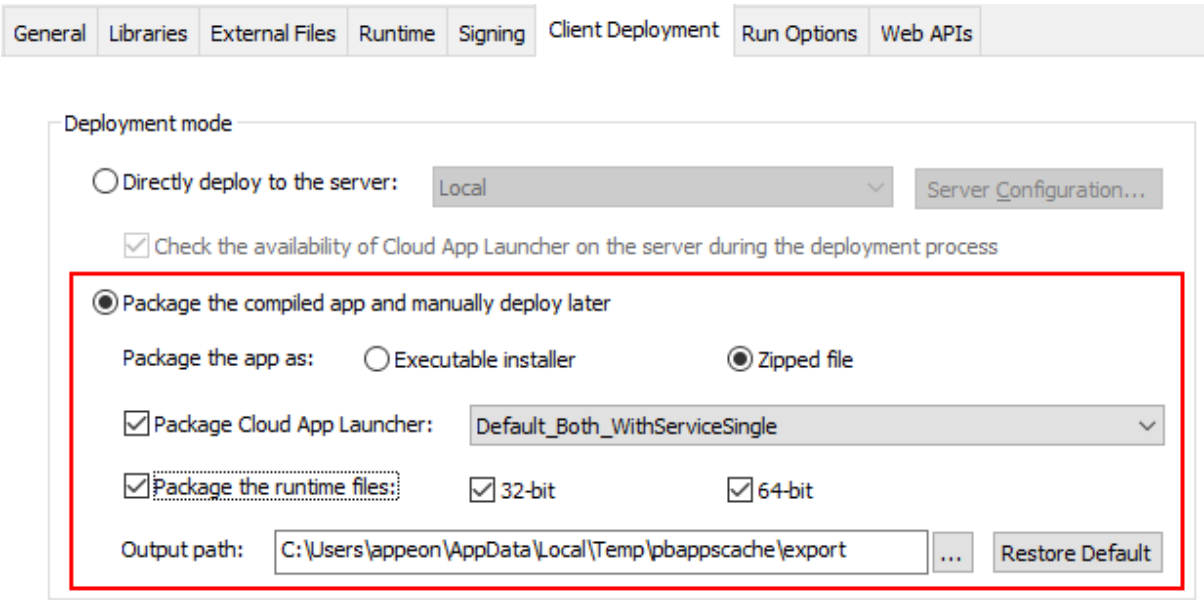
2. Specify to generate the package as an executable installer or a compressed zip file, and select whether to package the cloud app launcher and the PowerBuilder Runtime files.

   If you select **Zipped file**, an *appname*_Installer.zip file is generated in the specified path. You can copy the zip file to the server and then decompress it to the Web root.

   If you select **Executable installer**, an *appname*_Installer.exe file is generated in the specified path. You can copy the executable file to the server and then run it to install the application to the Web root.

3. Specify the location where the package will be generated.

Figure 8.1:



4. Save the project settings and then click the **Build & Deploy PowerServer Project** button ( ) or **Deploy PowerServer Project** ( ) button in the toolbar to generate the package.

Note

Do not manually change the name of the installed or de-compressed application folder on the server, otherwise the application uninstall program will fail to run.

8.2 Packaging the PowerServer Web APIs

For easier distributions, the PowerServer Web APIs can be published to a local folder.

Step 1: On the development machine, open the PowerServer C# solution in SnapDevelop. Log in to SnapDevelop if required.

Click the **Open C# Solution in SnapDevelop** button ( ) in the toolbar to launch the PowerServer C# solution in SnapDevelop. Or go to the location where the PowerServer C# solution is generated; and double click **PowerServer_[appname].sln** to launch the solution in SnapDevelop.

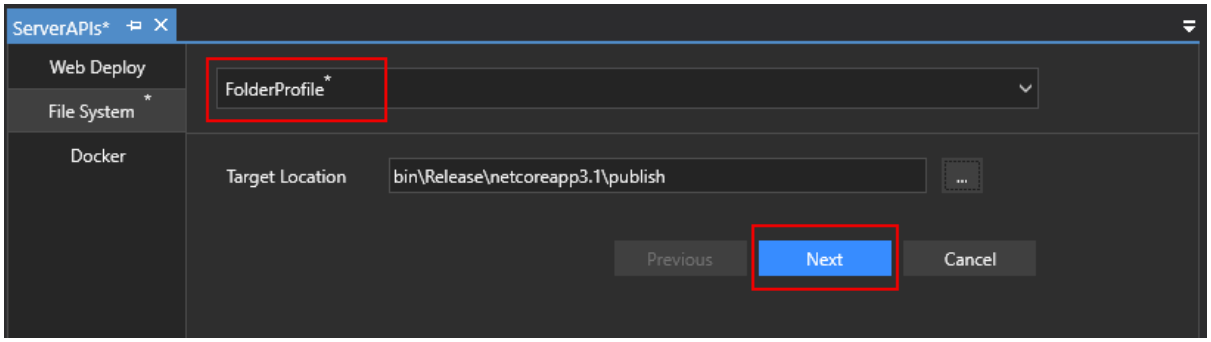
At startup, the solution will install/update the dependencies. Wait until the **Dependencies** folder completes the install/update. (Make sure the machine can connect to the NuGet site: https://www.nuget.org in order to successfully install PowerServer NuGet packages).

Step 2: In the **Solution Explorer**, right click on the **ServerAPIs** project node, and select **Publish**.

Step 3: In the window that appears, select **File System**, and click **Start**.

Step 4: Specify a name for the profile and specify the destination folder where the files will be published, and click **Next**.

Figure 8.2:

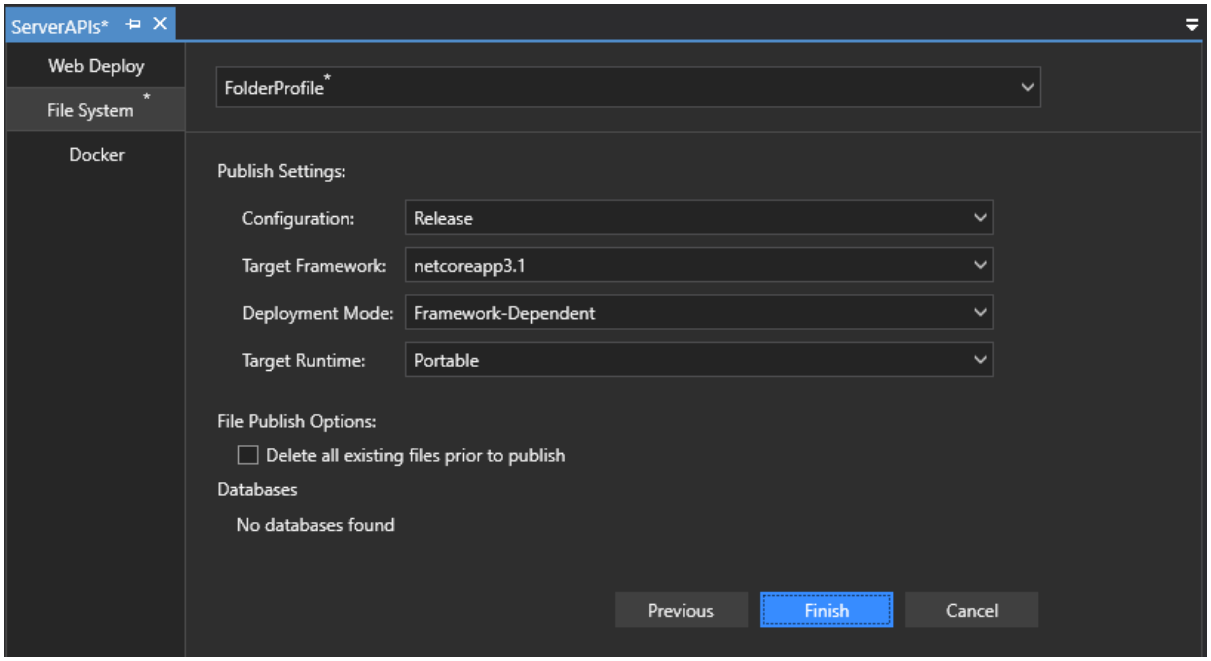


Step 5: Specify the publish settings or use the default settings and then click **Finish**.

If published as a ***Framework-Dependent*** package, the package will only include the project itself and its dependencies. Users have to install the .NET Core runtime in order to run the project.

If published as a ***Self-Contained*** package, the package will not only include the project itself and its dependencies, but also include the .NET Core runtime and libraries. Users can run it on a machine that has no .NET Core runtime installed.

Figure 8.3:

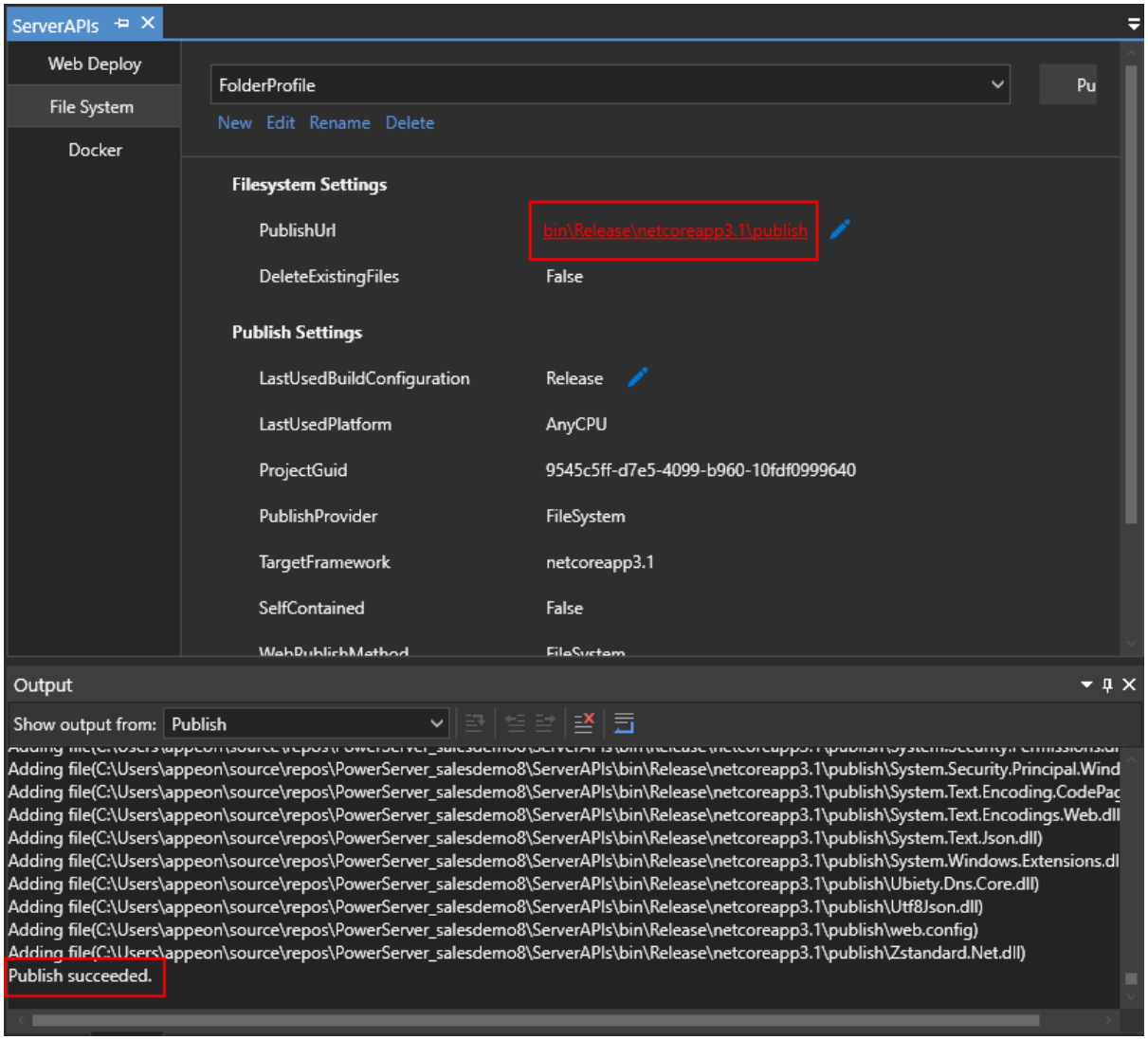


Publishing begins automatically. If any error or failure is reported in the **Output** window, click the link provided at the end to view more details and possible solutions.

Step 6: Make sure publishing was successful.

Click the **PublishURL** to open the folder that contains the published files.

Figure 8.4:



8.3 Telling client app where PowerServer Web APIs is

The client app needs to know where to access the PowerServer Web APIs when it starts to run. You can tell a client app where to access the PowerServer Web APIs before and/or after the PowerServer project deployment in the PowerBuilder IDE.

Before the PowerServer project deployment in the PowerBuilder IDE, you can specify the **Web API URL** field on the **Web APIs** tab in the PowerServer project painter. Then, the specified Web API URL will be included in the application client-side when it is deployed or installed to the Web server. It is highly recommended that you specify an HTTPS URL for the production environment.

Figure 8.5:

After the application is deployed or installed to the Web server, you can change the Web API URL that is included in the application without needing to re-deploying or re-installing the application. To do that, you can run the following commands in the "1.01" sub-folder of the application folder on the Web server.

To get the current URL:

```
dotnet CustomizeDeploy.dll -url
```

To change the URL:

```
dotnet CustomizeDeploy.dll -url=<URL>
```

For example

Figure 8.6:

```
C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -url
Web API URL: https://172.16.100.71:5009

C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -url=https://www.appeon.com:6666
Web API URL changed successfully.

C:\inetpub\wwwroot\pssales\1.01>dotnet CustomizeDeploy.dll -url
Web API URL: https://www.appeon.com:6666
```

For more about how to change the Web API URL using commands, see How-to Guides > Customize the deployed app using commands.

9 Tutorial 9: Load testing installable cloud apps

9.1 Load testing installable cloud apps with LoadRunner

LoadRunner is an automated performance and testing product from Micro Focus. It can simulate hundreds or thousands of concurrent users, to put the application through the real-life user loads, and examine the behavior and performance.

When you load test installable cloud apps with LoadRunner, after recording a script, you are required to correlate a few dynamic values and also parameterize static values in the script. Otherwise, the script will fail to replay. In this tutorial, we will explain the relevant techniques in detail and with examples that are specifically required for load testing installable cloud apps with LoadRunner. For the common LoadRunner functions, please refer to the relevant LoadRunner documentation.

9.1.1 Dynamic Values in the Recorded Script

The main dynamic values in the recorded script for installable cloud apps are “sessionid” and “transactionid”. Both values are dynamic and can only stay valid for a short time, therefore, it is necessary to capture them using the function “web_reg_save_param” and then save them into parameters in the script.

Specially, about “sessionid”: Because all the requests and responses between the client application and PowerServer are tracked by “sessionid”, “sessionid” is encrypted in every request/response for security reasons in production environment. It is necessary to set “sessionid” to plain text with the technique explained at Running the Application in Test Mode before Recording the Script.

9.1.2 Enclosing Parameters in Angle Brackets “<>”

LoadRunner scripts typically enclose parameters in curly braces “{}”. However, because the scripts recorded for installable cloud apps contain many JSON strings, which contain a lot of formatting curly braces “{}”, it is recommended that you enclose parameters in angle brackets when you edit the script recorded for installable cloud apps.

9.1.3 Running the Application in Test Mode before Recording the Script

As explained above in Dynamic Values in the Recorded Script, “sessionid” is encrypted in production environment. It causes difficulty to correlate the value in the script. To work around the problem, the PowerServer Web APIs has two modes:

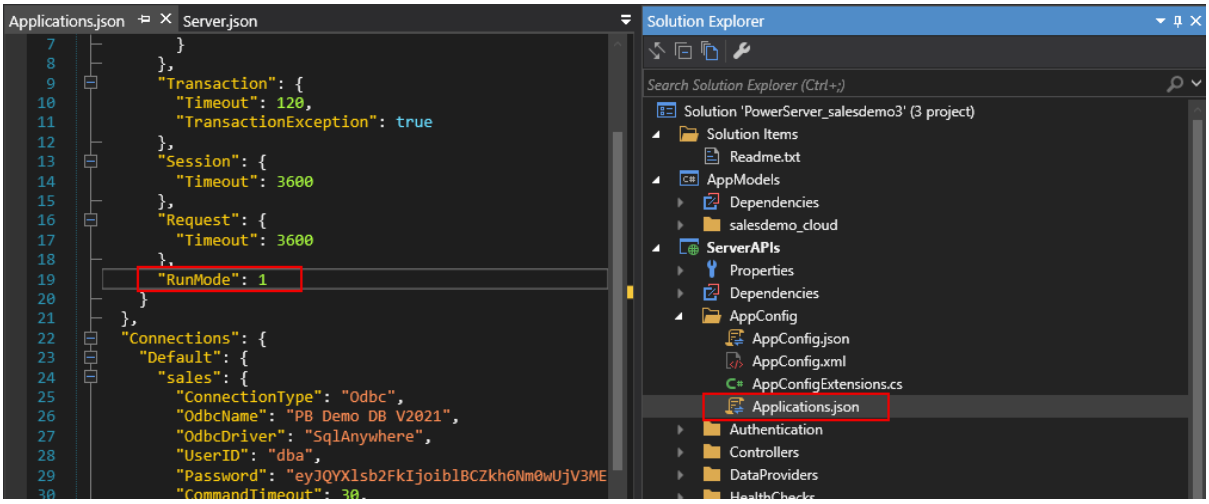
- “0”- normal mode
- “1”- test mode

Under the test mode, the “sessionid” included in the requests and responses is in plain text. The security is compromised but it shall be sufficient for the test environment.

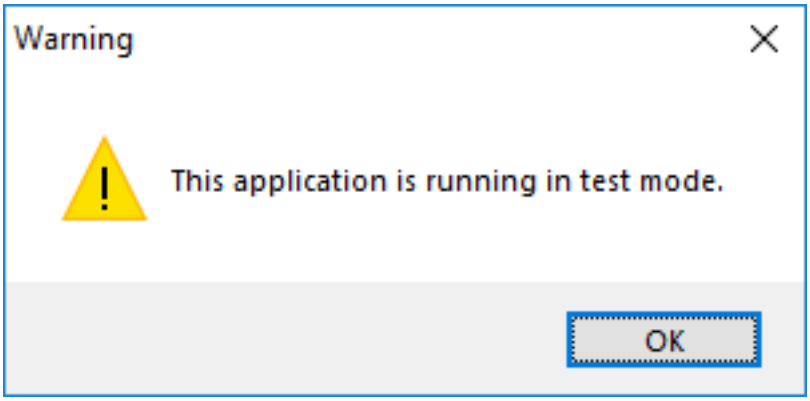
9.1.3.1 How to switch to the test mode

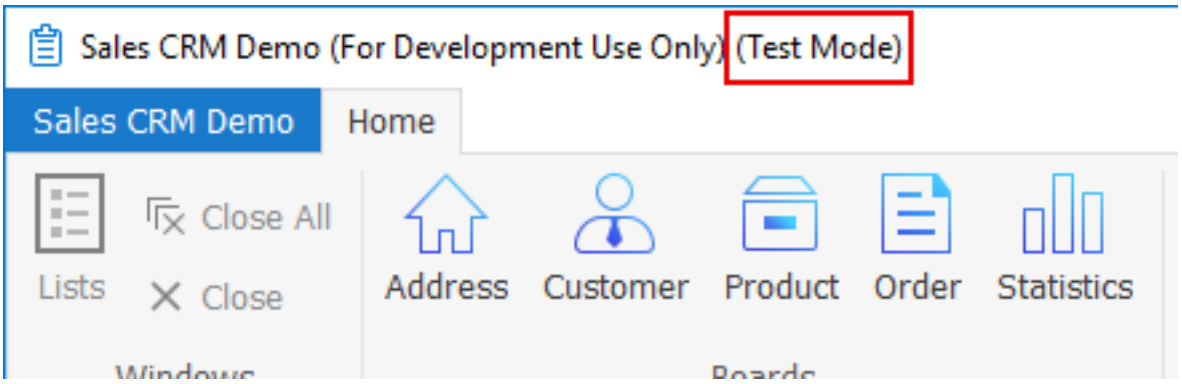
1. Open the file AppConfig | applications.json in the ServerAPIs project of the PowerServer solution.

2. Change the “RunMode” attribute value from “0” to “1”.



With the “RunMode” set to 1, when the installable cloud app is started, the app will prompt the following warning, and all window titles in the installable cloud app will show “Test Mode”.





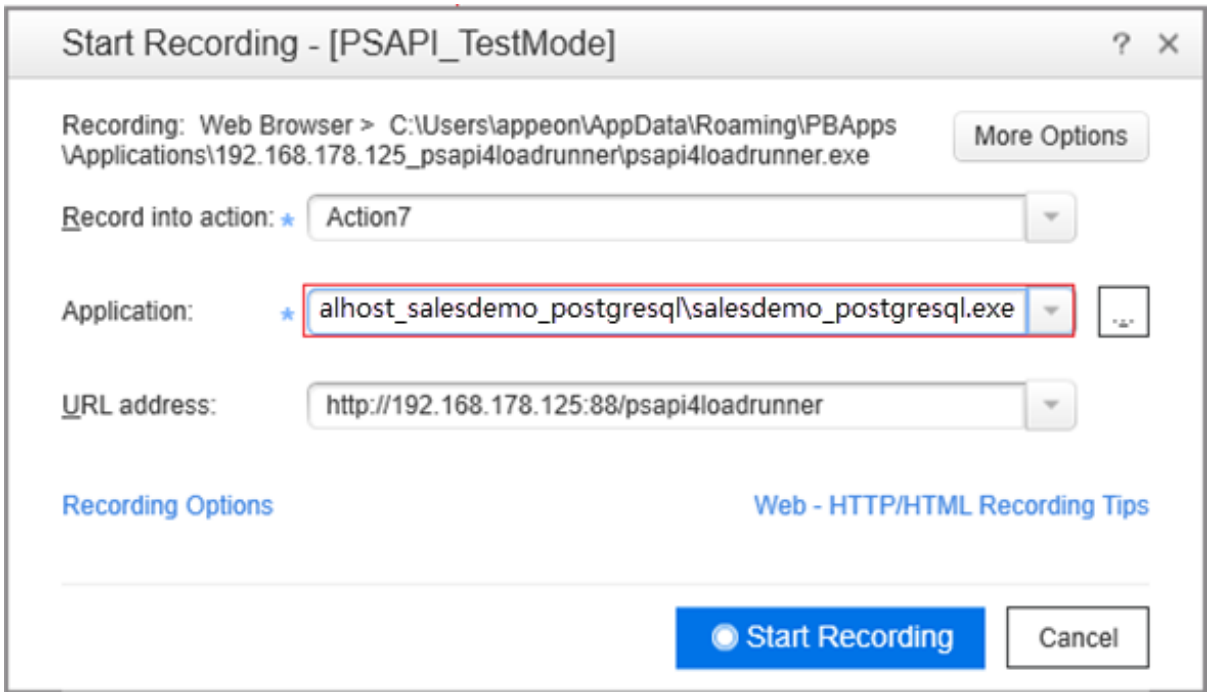
Note: Instructions in the document assume that the installable cloud app is running in the test mode.

9.1.4 Recording

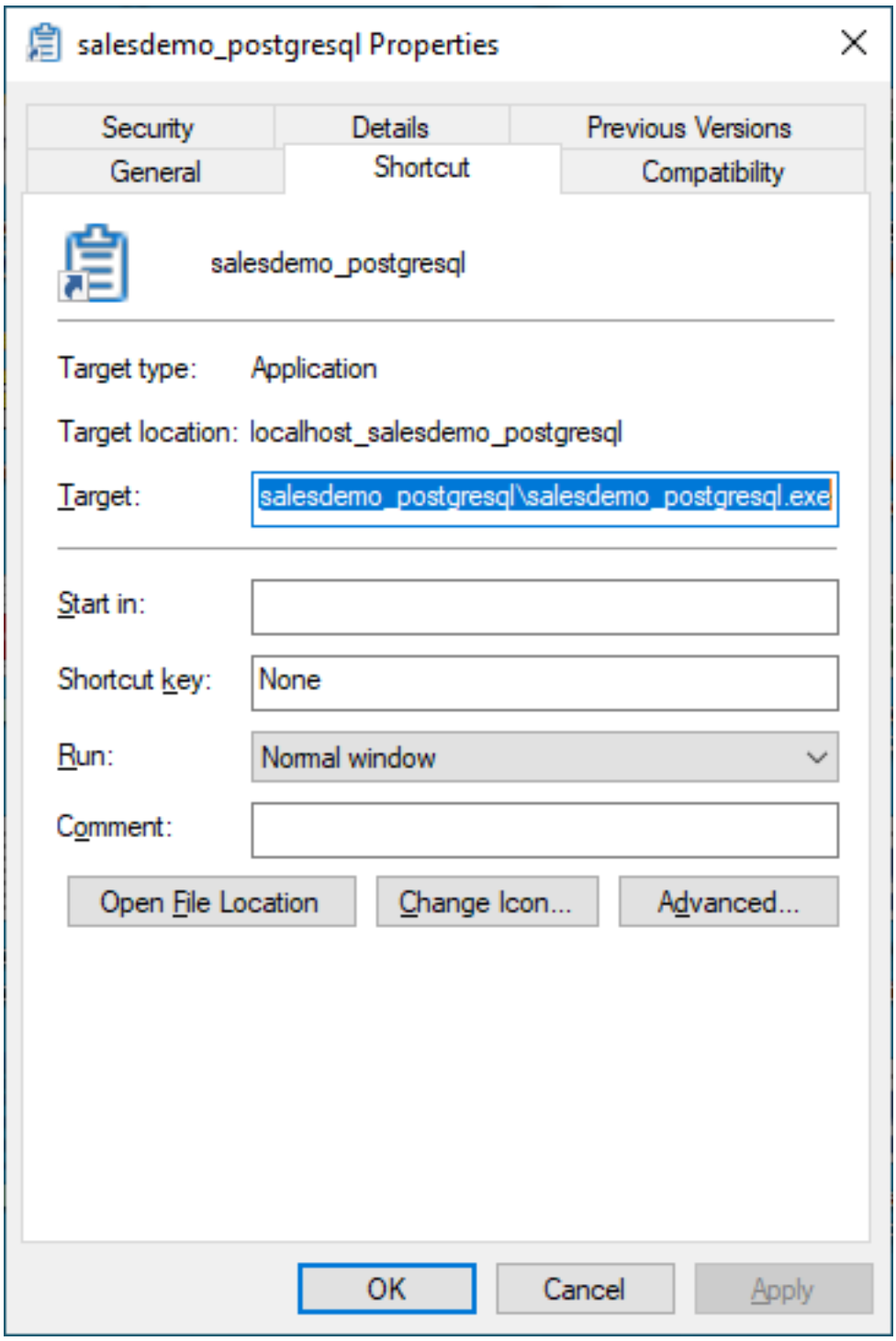
9.1.4.1 Specifying the app .exe file as the Application

For recording, please fill up the Start Recording dialog similar to the screenshot A provided below. Specifically, for the Application to run the URL, please specify the .exe file of the client app, which you can get by right clicking the desktop shortcut of the app and then copying from the properties > Target field.

Screenshot A: The Application field value is the .exe file of the client app

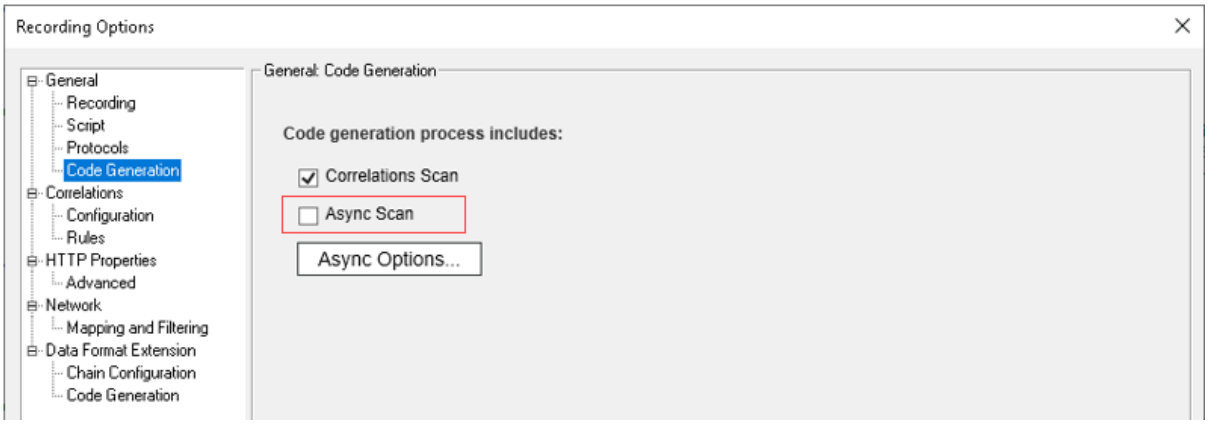


Screenshot B: The Application field value read from the Target field of the app properties



9.1.4.2 Disabling the async scan

The “Async Scan” option under Code Generation shall be disabled. If the option is on, LoadRunner will automatically generate asynchronous callback scripts. It is hard to parameterize the transaction IDs properly in the asynchronous callback scripts.



9.1.5 Correlating the Session ID

Session ID ("sessionid") is different every time you run an installable cloud app. Obviously, since the execution of the recorded script depends upon the session ID returned by the server, it is necessary to catch the proper "sessionid" from the CreateSession server response and then attach it to the subsequent requests. That is, correlating the session ID in the script for successful replay.

9.1.5.1 How to correlate the session ID in the recorded script

1. Add scripts in the CreateSession for capturing the session ID and assign it to a parameter.

   A CreateSession request looks like the following:

```
web_custom_request("CreateSession",
      "URL=http://192.168.178.125:5001/api/ServerApi/CreateSession",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t60.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"52C33A54-6687-40ef-
ACA8-4FC34B8066CE\",\"appname\":
            \"psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",\"session
\":null,\"type\":31,\
            "transaction\":null,\"content\":{\"createsession\":{\"securestring
\":\""

 "eyJ0aW1lc3RhbXAiOjE2MjQyNjk4MjYsInBheWxvYWQiOiJuQStOK1ZKb1pqRkxFRzBsT0QzMnAzZmtsVlc2Qk

 xydGZuQ2FqcUJvcTNTY0FORDdYeDZmTDFnMVFGUFhNdk9EVGpWTjgyVERLOTdSMHhHVEhSMmxXZz09Iiwic2lnbm
            F0dXJlIjoiOUpicHc2OUFwWFYxcEg4UTEraDRodHh4SHlGVnptS2lhWmdNZmJQS1pP

 UmdBVW9JcHdoTzNTY2krbnltV2NTZ2lidzUyZHhsYjluQlpjQXgyUmd4S2c9PSJ9\"}}}",
      LAST);
```

   You need to add the following code above the CreateSession request:

   - Add a web_set_max_html_param_len function to be the first line in the script file. It would set the maximum length of the HTML string which LoadRunner can retrieve:

```
web_set_max_html_param_len("262144");
```

- Add a web_reg_save_param function above CreateRequest to capture the session ID and assign it to the parameter “gs_SessionID”.

```
web_reg_save_param("gs_SessionID","LB=\"sessionid\":\"","RB=\"},
\"graceperiod","Search=Body",LAST);
```

2. Replace the session ID with the parameter every time it occurs.

- Identify the session ID that need to be correlated. You may do a global search in the script for “session” which is followed by the session ID.

- Replace every occurrence of the session ID with the parameter <gs_SessionID>.

Taking the ConnectAndCreateTransaction request as an example. The following is the original request. You see the session ID is the string following the “session”.

```
web_custom_request("ConnectAndCreateTransaction",
      "URL=http://192.168.178.125:5001/api/ServerApi/
ConnectAndCreateTransaction",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t65.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"D024C3D8-FE54-490e-
BAB6-23F8CAB3D8F3\",\"appname\":\"psapi4loadrunner\",
            \"namespace\":\"Psapi4loadrunner\",\"session\":
\"""eyJ0aW1lc3RhbXAiOjE2MjQyNjk4MjgsInBheWxvYWQiOiJ3bmFMV0FuZ2

 14S2RnQlhqbFVqQlRNNmswRVUxN2gwaFhpb3pEbm9SakkyOUtkYk9kWlMwWGgxeWtPdTlYRFI5QTFITXZKTVcvRW9IbEp

 rZlA0Y3YvOEVjMWF5OVNxTzlSUFpEYjJPYTIrektQUmRPSXV2L25yVXYwdDhGNmpzRUY0TzVqUTROMmVhc1piY0c5NkpJ

 cTNiZ1BxWTZtL2dodlhBYW84U3AiLCJzaWduYXR1cmUiOiJQTEo3WDJaRnBVV1FyOXcvK1dwaG4rWnRWK3BPc0taRUhwd
            WRuUW52QVh5VlgvUUpDTlZFZUVFR0twcGtIOUs0c3FQTDB3SVZqQT09In0=\",\"type
\":7,\"transaction\":null,\"content\":
            {\"connect""\":{\"cachegroup\":\"\",\"cachename\":\"\",
\"transactionname\":\"sqlca\",\"params\":\"""eyJ0aW1lc

 3RhbXAiOjE2MjQyNjk4MjgsInBheWxvYWQiOiJQK1kvMk5yWmVTUWY0dElIempGMy9vdlk0VFhWY0NWNlFrd2UxVGJkSE

 dtNDN2WlB6dFFCLzk0Ulk4dVFOYkNISnZqblltQ3RoTUZCRFJaS21YZmtDWExPRkdjbWt2WXFkSXBEcDRhN0NLUDBVK1g

 3aWp2S0FkZEpNOFU3QmxXbEI5MGx4ZEp5OVVTNU00NzZhaGxZMlRSYlF1NTlWcksxdER0VngxbERKRU80cGFvRVVkM0cx

 MDFPNDZ3UmNvVVp6MlB1TXkrUG1VNVlvWmF0eUhxRjN3ZExyRlZLIiwic2lnbmF0dXJlIjoiVGJoR2dEeEtLWFg2V3V6b

 TNvUlgxSTkzeDJiS1lncjd0MmFXcmxaUkp0TTh6dEZpVUc4R1Y3VTgrdk""xZL2pQK29lZGxTdW0yWWc9PSJ9\"}}}",
      LAST);
```

After replacing the occurrence of the session ID with the parameter <gs_SessionID>, the script will look like the following:

```
web_custom_request("ConnectAndCreateTransaction",
      "URL=http://192.168.178.125:5001/api/ServerApi/
ConnectAndCreateTransaction",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
```

```
      "Referer=",
      "Snapshot=t65.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"D024C3D8-FE54-490e-
BAB6-23F8CAB3D8F3\",
            \"appname\":\"psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",
            \"session\":\"""<gs_SessionID>\",\"type\":7,\"transaction\":null,
\"content\"
            :{\"connect""\":{\"cachegroup\":\"\",\"cachename\":\"\",
\"transactionname\":\"sqlca\",\"params\":\

 """eyJ0aW1lc3RhbXAiOjE2MjQyNjk4MjgsInBheWxvYWQiOiJQK1kvMk5yWmVTUWY0dElIempGMy9vdlk0VFhWY0NWNl

 xVGJkSENHTFNnc2R0allOZUdtNDN2WlB6dFFCLzk0Ulk4dVFOYkNISnZqblltQ3RoTUZCRFJaS21YZmtDW

 ExPRkdjbWt2WXFkSXBEcDRhN0NLUDBVK1gxOGhQQVhiNjJoSXA3aWp2S0FkZEpNOFU3QmxXbEI5MGx4ZEp5OVVTNU00Nz

 ZMlRSYlF1NTlWcksxdER0VngxbERKRU80cGFvRVVkM0cxQk5RbUtrcVBqaTdaMDFPNDZ3UmNvVVp6MlB1T

 XkrUG1VNVlvWmF0eUhxRjN3ZExyRlZLIiwic2lnbmF0dXJlIjoiVGJoR2dEeEtLWFg2V3V6bTIwei94WXdKb2FFWTNvUl

 zeDJiS1lncjd0MmFXcmxaUkp0TTh6dEZpVUc4R1Y3VTgrdk""xZL2pQK29lZGxTdW0yWWc9PSJ9\"}}}",
      LAST);
```

9.1.6 Correlating the Transaction ID

Installable cloud operations perform all database operations in transactions. Each transaction is assigned with a unique transaction ID. An existing transaction ID will become invalid if any of the following cases occur:

- When the relevant session ID becomes invalid;
- At the execution of Connect, Disconnect, Commit, or Rollback.

When the Connect, Disconnect, Commit or Rollback is executed, the response body will contain the new transaction ID. When the Commit or Rollback is executed but has failed, the response body will still contain the old transaction ID, which means that the old transaction is still valid.

It is necessary to catch the proper “transactionid” from the server response on Connect/Disconnect/Commit/Rollback, and then attach it to the subsequent requests. That is, correlating the transaction ID in the script for successful replay.

9.1.6.1 How to correlate the transaction ID in case of single transaction

The following steps take the Connect request as an example, and the session ID has already been correlated.

1. Add scripts in the Connect for capturing the transaction ID and assign it to a parameter.

   A ConnectAndCreateTransaction request looks like the following:

```
web_custom_request("ConnectAndCreateTransaction",
      "URL=http://192.168.178.125:5001/api/ServerApi/
ConnectAndCreateTransaction",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
```

```
      "Snapshot=t275.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"667FF1FE-77F8-40b6-869E-
EA90D466B504\",
            \"appname\":\"psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",
\"session\":\""
            "<gs_SessionID>\",\"type\":7,\"transaction\":null,\"content\":
{\"connect"
            "\":{\"cachegroup\":\"\",\"cachename\":\"\",\"transactionname\":
\"sqlca\",\"params\":\""

 "eyJ0aW1lc3RhbXAiOjE2MjUyMDM0OTMsInBheWxvYWQiOiJjNjJLb3BST1pMVVBuZVc4NXk5bUgzb1RvOXI1NSs

 rR05UTC90S2wyR2lubExleE1CdCtVaGttRWJtd2N2Um5vVTlBejBCbDV3Mlh0aW5zYi80SU9jQW5uc2hqNXdSbUt

 0ZGx0UlJkeVllSCtEU1JTc1ZNVjd5SmttTnBTdHpoZnZHK3Z0RnhJWE1JczZHRzh5QVJsd2U1WlJ3VEFUVmIxeFF

 rZnl3MXdOc1ZUeFNGMDNsb2UwMlZXVU1JOUU5MzhuTHhoaXRxMElmTTZjaVhST21la2xMaEh6ZkREcm1tc3RWYVU

 4OWhuY2ZZNU5oaXdtMHFnYkowY2Fsdmp6OENOcjhmVHpZR1BmQ0UxZW1YZ3lkNG8vUT09Iiwic2lnbmF0dXJlIjo
            idFkwVDlURUxYY0tOTEhBdmZzQUNOQ0tuRWtlRUtKNEFMSnBsamJ1aFRtYk92dU"
            "FhSnBrNXlLMTBhMEtiMzQ1dkc4Vm9tRTJZaG9Kb0FnbUlOaHF1cWc9PSJ9\"}}}",
      LAST);
```

You need to add the following code above the ConnectAndCreateTransaction request:

- Add a web_reg_save_param function above the request to capture the transaction ID and assign it to the parameter “gs_TransactionID”.

```
web_reg_save_param("gs_TransactionID","LB=\"transactionid\":\"","RB=\"},
\"content","Search=Body",LAST);
```

2. Replace the transaction ID with the parameter every time it occurs.

- Identify the transaction ID that need to be correlated. You may do a global search in the script for “transactionid” which is followed by the transaction ID. Ensure that there is no confusion between the current transaction ID and the other ones.

- Replace every occurrence of the transaction ID with the parameter <gs_TransactionID>.

```
web_custom_request("RetrieveWithParm",
      "URL=http://192.168.178.125:5001/api/ServerApi/RetrieveWithParm",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t279.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":
\"014DD22B-11AB-4238-88D1-7892060396AD\",\"appname\":
            \"psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",\"session\":
\"""<gs_SessionID>\",
            \"type\":1,\"transaction\":{\"transactionid\":\"""<gs_TransactionID>
\",\"transactionname\
            ":\"sqlca\"},\"content\":{\"retrieves\":[{\"retrieveid\":
\"014DD22B-11AB-4238-88D1-789206
            0396AD\",\"parent\":\"\",\"dataobject\":\"d_customers\",
\"parentcolumn\":\"\",\"isreport\":
            false,\"isdynamic\":false,\"dwsyntax\":\"\",\"sql\":\"\",
\"processing\":1,\"arguments\":[]}]}}",
```

```
      LAST);
```

9.1.6.2 How to correlate the transaction ID in case of multiple transactions

If your application has multiple transactions, each transaction has its unique transaction ID. The transactions can be differentiated by their transaction names, and their transaction IDs shall be assigned with different parameters, so that each parameter will correlate with its own transaction.

1. Add scripts in the Connect for capturing the transaction ID and assign it to a parameter.

   A ConnectAndCreateTransaction request in a transaction named as “itr_dynamiccon” looks like the following:

```
web_custom_request("ConnectAndCreateTransaction_2",
      "URL=http://192.168.178.125:5001/api/ServerApi/
ConnectAndCreateTransaction",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t337.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"5D86818C-CAF5-49fe-
B78A-4AC7AC550F88\",\"appname\"
            :\"psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",\"session
\":\"""<gs_SessionID>\"
            ,\"type\":7,\"transaction\":null,\"content\":{\"connect""\":
{\"cachegroup\":\"\",\"cachename\
            ":\"PostgreSQL\",\"transactionname\":\"itr_dynamiccon\",\"params\":
\"""eyJ0aW1lc3RhbXAiOjE2Mj

 UyMDg5MTEsInBheWxvYWQiOiJna21qUzVZY01RM294ZnJLVFNQQ1NyRFJuUENqTmxPTHZEeENGUlJ0MngrNW1WV3d2WFJ

 NK3BCalRiMEFKY05EZm5iMFU5VVQ0ckhjdnRRQ0tubkV1Q29adlhoOExlRXZvaFhsSGJlMmQ1cFdRZzc2VnhLOGYwZHZl

 SDBGMVRBRTB6YU9zaEhuM2lKMGZEQkJSak1scXl1MThyUnM0OFlOY3dlMGY2ZGpHbWVvUGxmVU40RzM0MTcwMCtzbUh3U

 ng5ZUlIcnUvR2pRS0hkYmFFRWJyMlAxR2tMZVg4UE1wVlVtZTh5ektJWlRZZUVkSFBtd3crekpLNlJFeE9QMFFKVVdIVW

 4wajFKemh0MURYcW95VjlKMjJyam5CNVk3U1dBYy91UnRlQkxmUT0iLCJzaWduYXR1cmUiOiIyT2t2SGRIaWtoWGZzeXQ

 vSHNhSUk4R3VzQj""M5OXJmbER1YzdTWEpkNjg1bDlxUjJJbkpzRGVVbTcxdVl6WHh0UGZFeTFGTFUzcDRjRWNWK2p1YW
            F0UT09In0=\"}}}",
      LAST);
```

   You need to add the following code above the ConnectAndCreateTransaction request:

   - Add a web_reg_save_param function above the request to capture the transaction ID and assign it to the parameter “gs_TransactionID_Dycache”. Here the parameter name has a suffix “_Dycache” to identify the transaction “itr_dynamiccon”.

```
web_reg_save_param("gs_TransactionID_Dycache","LB=\"transactionid\":\"","RB=
\"},\"content","Search=Body",LAST);
```

2. Replace the transaction ID with the parameter every time it occurs.

   - Identify the transaction ID that need to be correlated. You may do a global search in the script for the transaction name “itr_dynamiccon” which follows the transaction ID.

- Replace every occurrence of the transaction ID with the parameter <gs_TransactionID_Dycache>.

```
web_custom_request("RetrieveWithParm",
      "URL=http://192.168.178.125:5001/api/ServerApi/RetrieveWithParm",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t338.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"81EDB9E2-CC47-4b68-
B70A-09B46DD88261\",\"appname\":\
            "psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",\"session\":
\"""<gs_SessionID>\",\
            "type\":1,\"transaction\":{\"transactionid\":
\"""<gs_TransactionID_Dycache>\",\"transactionname\"
            :\"itr_dynamiccon\"},\"content\":{\"retrieves\":[{\"retrieveid\":
\"81EDB9E2-CC47-4b68-B70A-09B46
            DD88261\",\"parent\":\"\",\"dataobject\":\"d_customers\",
\"parentcolumn\":\"\",\"isreport\":false,
            \"isdynamic\":false,\"dwsyntax\":\"\",\"sql\":\"\",\"processing\":1,
\"arguments\":[]}]}}",
      LAST);
```

9.1.7 Parameterizing Static Values in SQLs

When recording an installable cloud app to create the script, you would use static values for SQL statements (Retrieve, Select, etc.). It is not realistic to use the same value for all replays. Therefore, it is necessary to parameterize the static values.

9.1.7.1 How to parameterize static values in Retrieve

1. Find the static value in the Retrieve request body. You shall get it in the “arguments” node.

2. Select the value, right click and select “Replace with Parameter” -> “Create new parameter”.

3. Enter the parameter name.

   In the example script below, the “customer_id” static value has been parameterized with the parameter <customerID>:

```
web_custom_request("RetrieveWithParm",
      "URL=http://192.168.178.125:5001/api/ServerApi/RetrieveWithParm",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t144.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"50CA839C-E8DF-4303-
A146-33EBB30BEB45\",\"appname\":\
            "psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",\"session\":
\"""<gs_SessionID>\",\
```

```
            "type\":1,\"transaction\":{\"transactionid\":\"""<gs_TransactionID>
\"},\"content\":{\"retrieves\":
            [{\"retrieveid\":\"50CA839C-E8DF-4303-A146-33EBB30BEB45\",\"parent
\":\"\",\"dataobject\":
            \"d_customer_pro\",\""""parentcolumn\":\"\",\"isreport\":false,
\"isdynamic\":false,\"dwsyntax\":\"\",
            \"sql\":\"\",\"processing\":1,\"arguments\":[{\"category\":1,\"name
\":\"customer_id\",\""""type\":
            \"number\",\"value\":<customerID>}]}]}}",
      LAST);
```

9.1.7.2 How to parameterize static values in Select

1. Find the static value in the Select request body. You shall get it in the “parameters” node.
2. Select a value in the list of parameters, right click and select “Replace with Parameter” -> “Create new parameter”.
3. Enter the parameter name.
4. Repeat step 2 to 3 until all values are parameterized

   In the example script below, the Select request has been parameterized with the parameters <customerID> and <NotName>:

```
web_custom_request("SelectWithParm",
      "URL=http://192.168.178.125:5001/api/ServerApi/SelectWithParm",
      "Method=POST",
      "Resource=0",
      "RecContentType=application/json",
      "Referer=",
      "Snapshot=t358.inf",
      "Mode=HTTP",
      "EncType=application/json;charset=UTF-8",
      "Body={\"version\":\"1.0\",\"requestid\":\"7C362898-230B-4858-
AB15-2241DC3FD982\",\"appname\":
            \"psapi4loadrunner\",\"namespace\":\"Psapi4loadrunner\",\"session\":
\""""<gs_SessionID>\",
            \"type\":11,\"transaction\":{\"transactionid\":
\""""<gs_TransactionID>\",\"transactionname\":
            \"sqlca\"},\"content\":{\"esqlselect\":{\"modulename\":\"\",\"sqlid
\":\"sqlHandle_01_2763E2FE\",
            \"parameters\""":[{\"category\":1,\"name\":\"name\",\"type\":\"int
\",\"value\":<customerID>},
            {\"category\":1,\"name\":\"name\",\"type\":\"string\",\"value\":
\"<NotName>\"}]}}}",
      LAST);
```

9.1.8 Replaying

If you replay the script on a different machine from the one you recorded the script, to make sure that the Application field value from Recording is still valid during replay, you must manually run the application before the replay.

9.2 Load testing installable cloud apps with JMeter

9.2.1 Overview

This tutorial assumes a basic understanding of the following concepts:

- JMeter basis. To familiarize yourself with JMeter basics, go through https://jmeter.apache.org/usermanual/index.html.
- PowerServer and installable cloud app basis. To quickly get started with PowerServer and deploy the Sales Demo application, go through Quick Start.

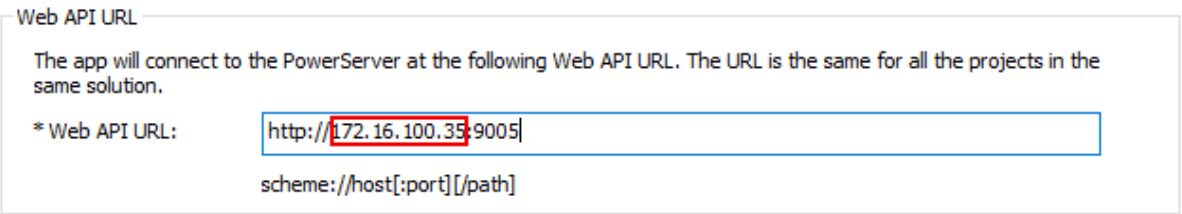
9.2.2 Preparing the installable cloud application

9.2.2.1 Configuring and deploying the application

This tutorial will take the deployed Sales Demo application as an example to walk through the test script recording and parameterizing.

You can follow Quick Start to deploy the Sales Demo application.

Note that you must use the actual IP address (instead of "localhost") for the Web API URL. (Reason is in the Section 9.2.3.1, “Recording scripts automatically (using Recorder)” and Section 9.2.3.2, “Recording scripts manually (using Fiddler + JMeter)” sections, you will use the JMeter proxy server or Fiddler (or any other web debugging proxy tool) both of which will bypass "localhost".)



9.2.2.2 Switching the application to test mode

9.2.2.2.1 Why test mode is required

The main dynamic values in the recorded script for installable cloud apps are “sessionid” and “transactionid”. Both values are dynamic and can only stay valid for a short time, therefore, it is necessary to capture them and save them into variables in the script.

Specially, about “sessionid”: Because all the requests and responses between the client application and PowerServer are tracked by “sessionid”, “sessionid” is encrypted in every request/response for security reasons in production environment, which makes it difficult to correlate the value in the script. To work around the problem, the PowerServer Web APIs has two modes:

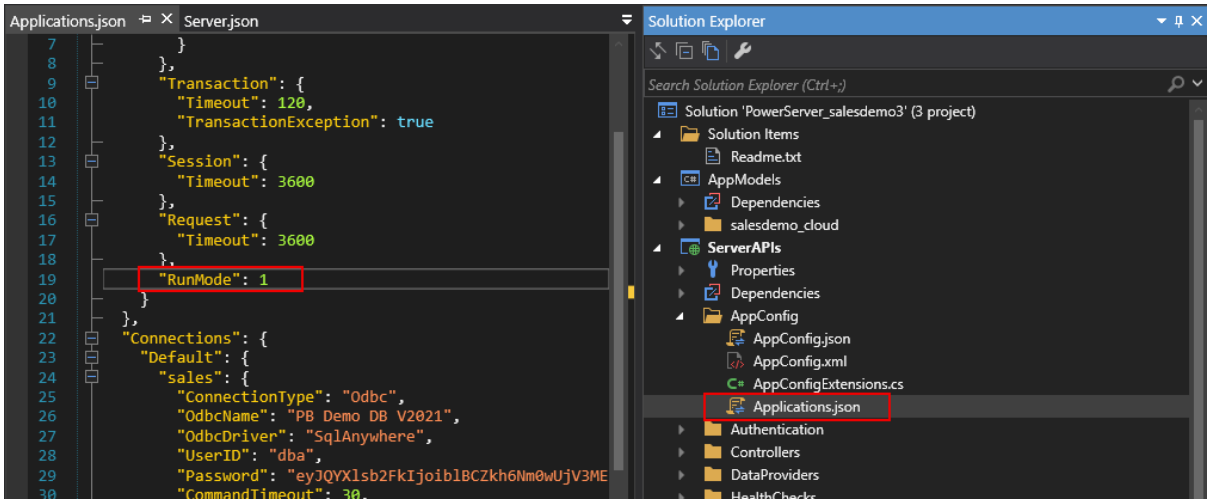
- “0”- normal mode
- “1”- test mode

Under the test mode, the “sessionid” included in the requests and responses is in plain text. The security is compromised but it shall be sufficient for the test environment.

9.2.2.2.2 How to switch to the test mode

Step 1: Open the file AppConfig | applications.json in the ServerAPIs project of the PowerServer solution.

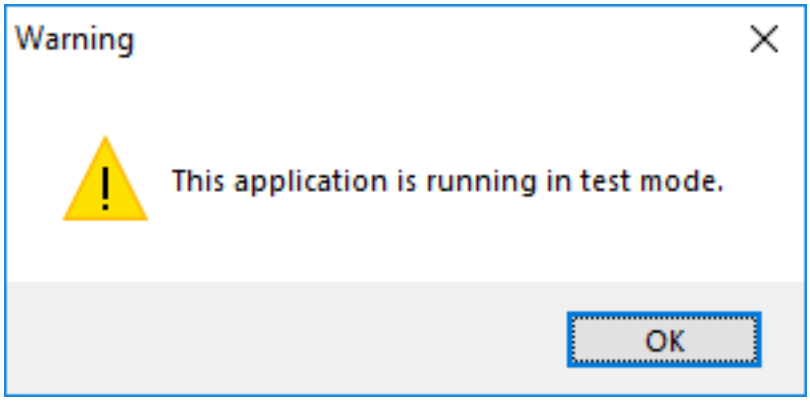
Step 2: Change the “RunMode” attribute value from “0” to “1”.

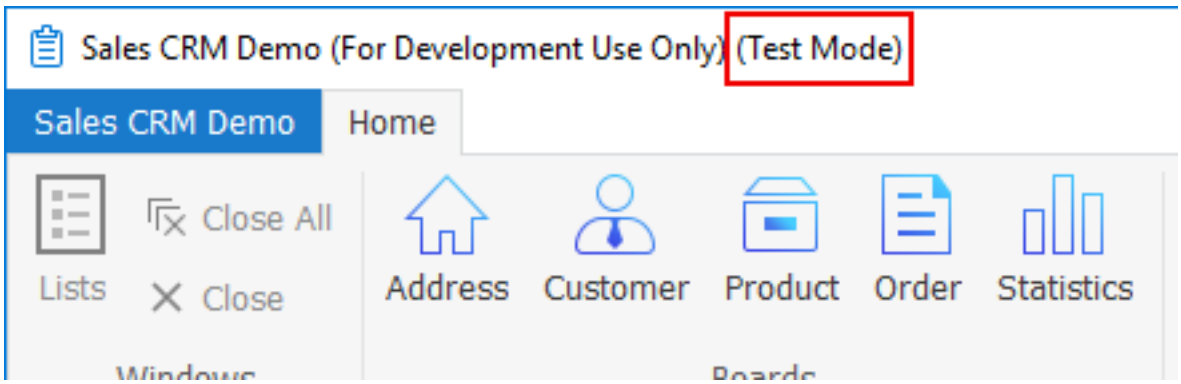


Step 3: Run PowerServer Web APIs.

Step 4: Run the installable cloud app.

With the “RunMode” set to 1 (test mode), the app will prompt the following warning, and all window titles in the installable cloud app will show “Test Mode”.





9.2.2.3 Running PowerServer Web APIs and then JMeter recorder or Fiddler

Keep PowerServer Web APIs running, and close the installable cloud app after you verify that the installable cloud app runs successfully; and then proceed to the next step to record the test scripts.

Note

You must run PowerServer Web APIs before you start JMeter recorder (HTTP(S) Test Script Recorder) or Fiddler (or any other Web debugging proxy tool). Otherwise, the PowerServer Web APIs will fail to start.

Reason is JMeter recorder and Fiddler (as well as any other Web debugging proxy tool) work by adding itself as a proxy instead of using your current proxy settings; therefore if the PowerServer Web APIs connects with the NuGet site and Appeon site through a proxy server, it may fail to start.

9.2.3 Recording JMeter scripts

You will need to create a test plan to record the JMeter scripts. You can create a test plan either by

- Using the JMeter HTTP(S) Test Script Recorder to automatically record the HTTP requests, or
- Using Fiddler (or any other web debugging proxy tool) to get the HTTP requests and then manually adding the HTTP requests to JMeter

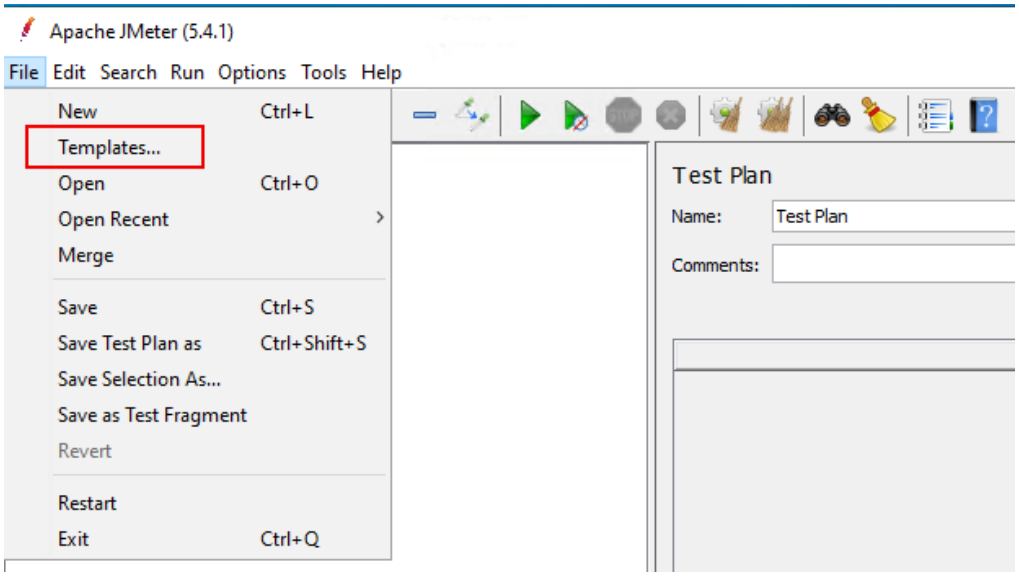
9.2.3.1 Recording scripts automatically (using Recorder)

One easy way to create a test plan is to use the JMeter HTTP(S) Test Script Recorder. With Recorder, you can just browse on the application and do the actions and everything (including HTTP requests) will get recorded automatically.

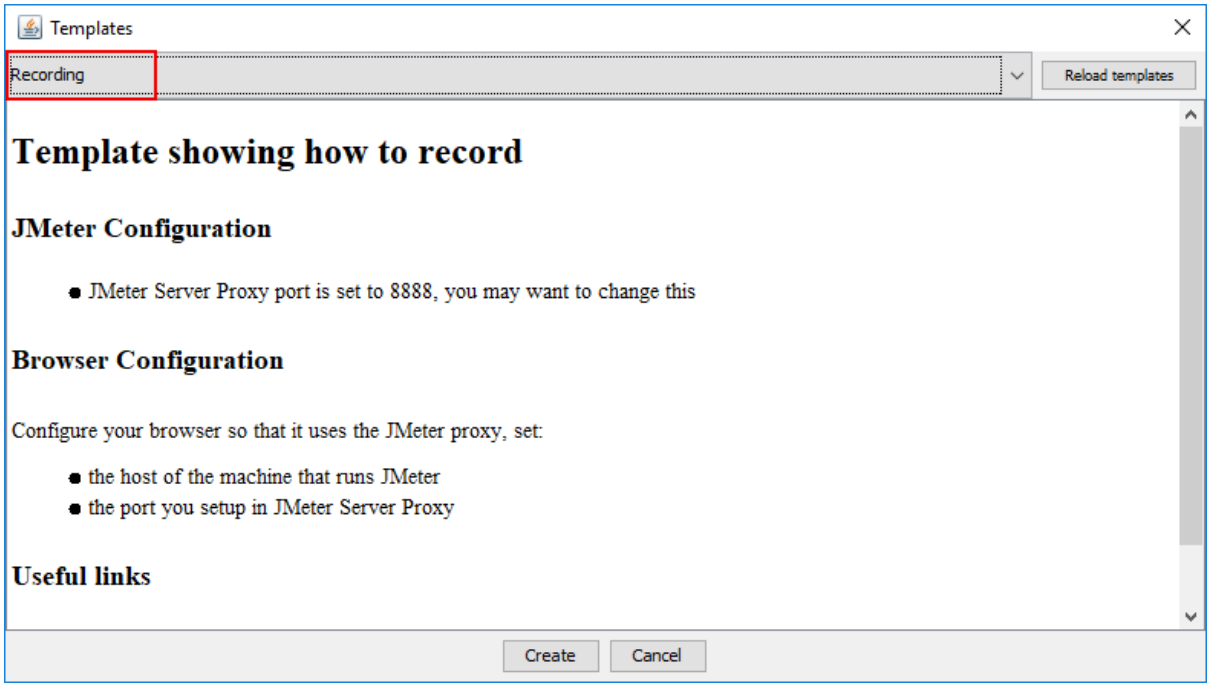
This section provides step-by-step instructions for recording scripts on the Sales Demo application. You can also follow the JMeter documentation: https://jmeter.apache.org/usermanual/jmeter_proxy_step_by_step.html.

9.2.3.1.1 Creating a test plan from templates

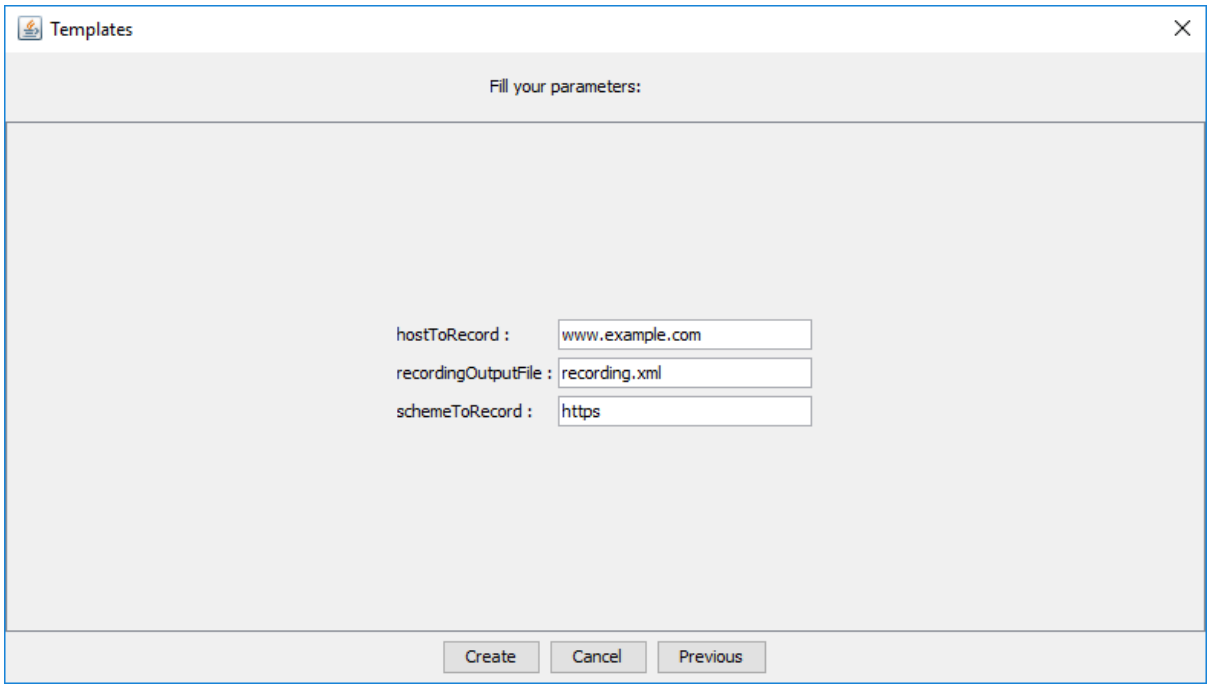
Step 1: Select **Templates** from the **File** menu.



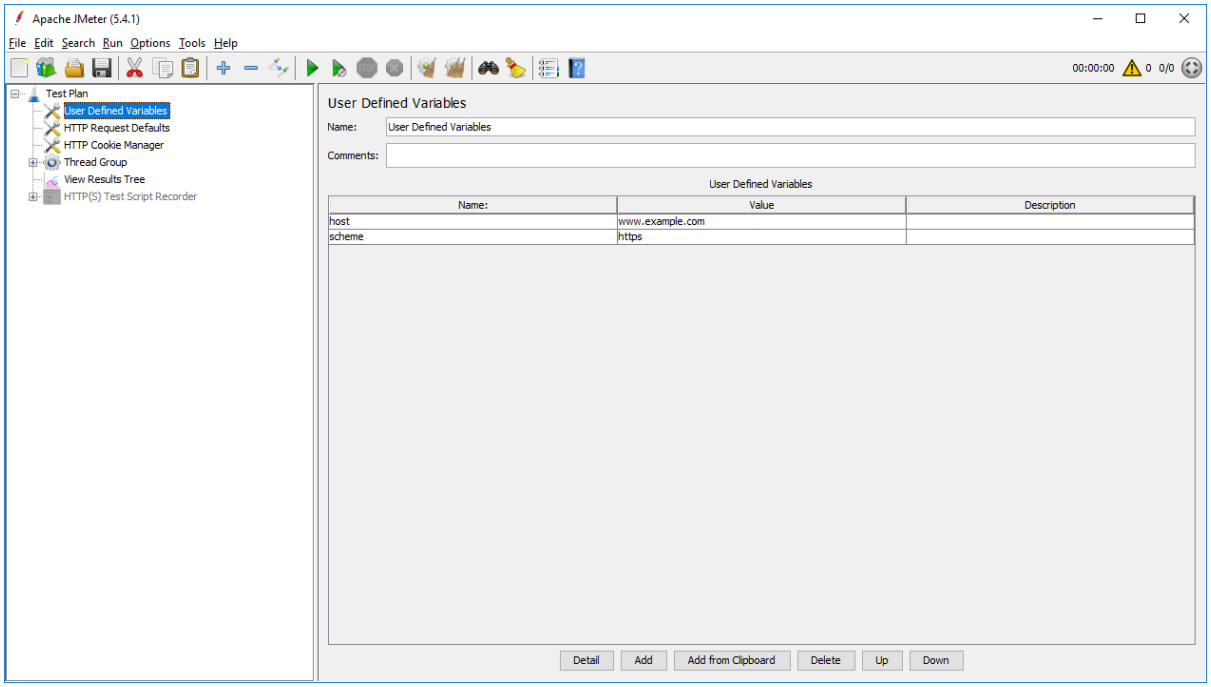
Step 2: Select **Recording** and then click **Create**.



Step 3: Use the default values or modify them according to your needs, and then click **Create**.



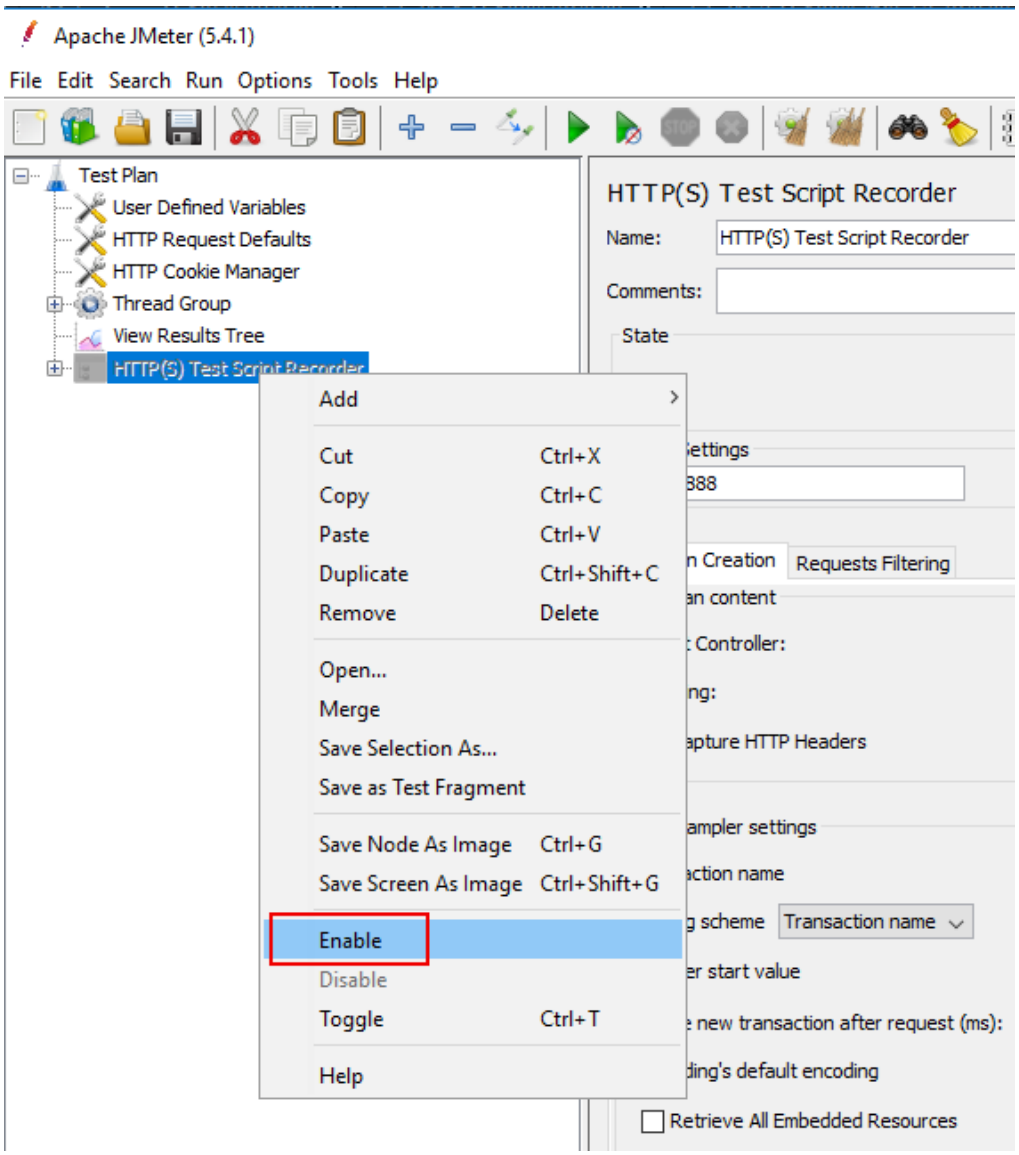
A complete Test Plan is generated successfully.



9.2.3.1.2 Enabling recorder

In Test Plan, the **HTTP(S) Test Script Recorder** is disabled by default.

Right click on it and then select **Enable** to enable it.



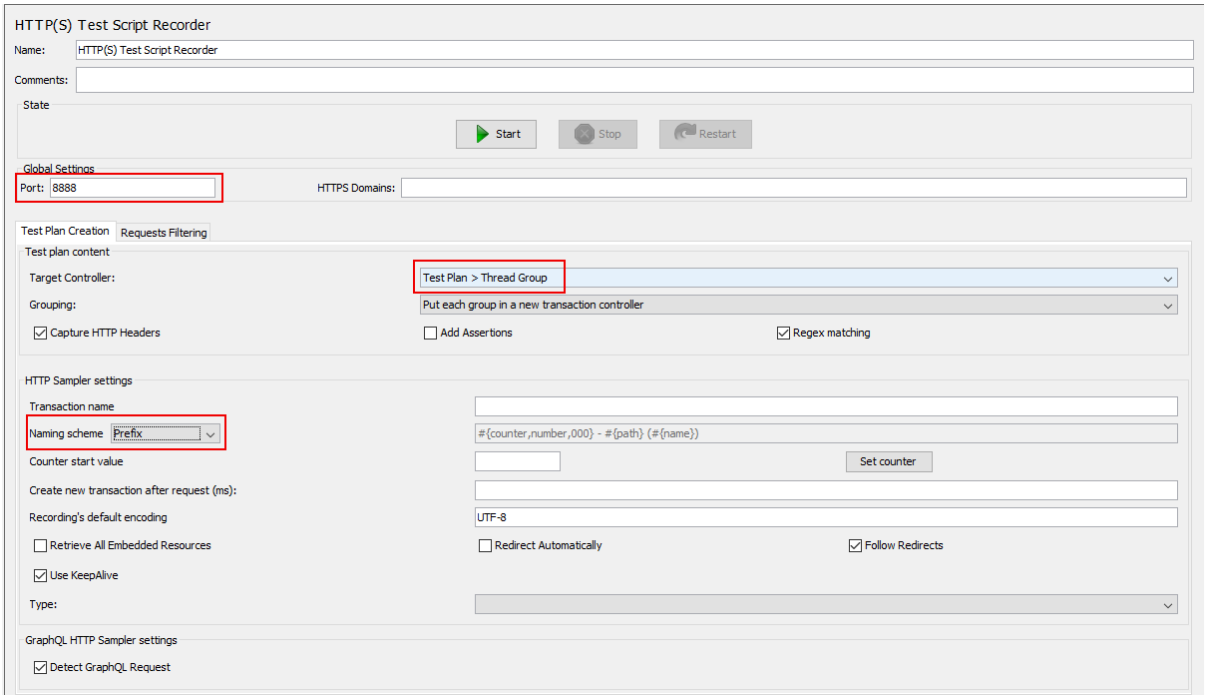
9.2.3.1.3 Configuring recorder

In the **HTTP(S) Test Script Recorder** window, specify the following settings:

1. Use the default port number 8888 or input a new one. Make sure the port is not occupied by any other program. Make sure the browser proxy is set to the same port later.

2. Modify **Target Controller** to **Test Plan > Thread Group**.

3. Modify **Naming scheme** to **Prefix**.

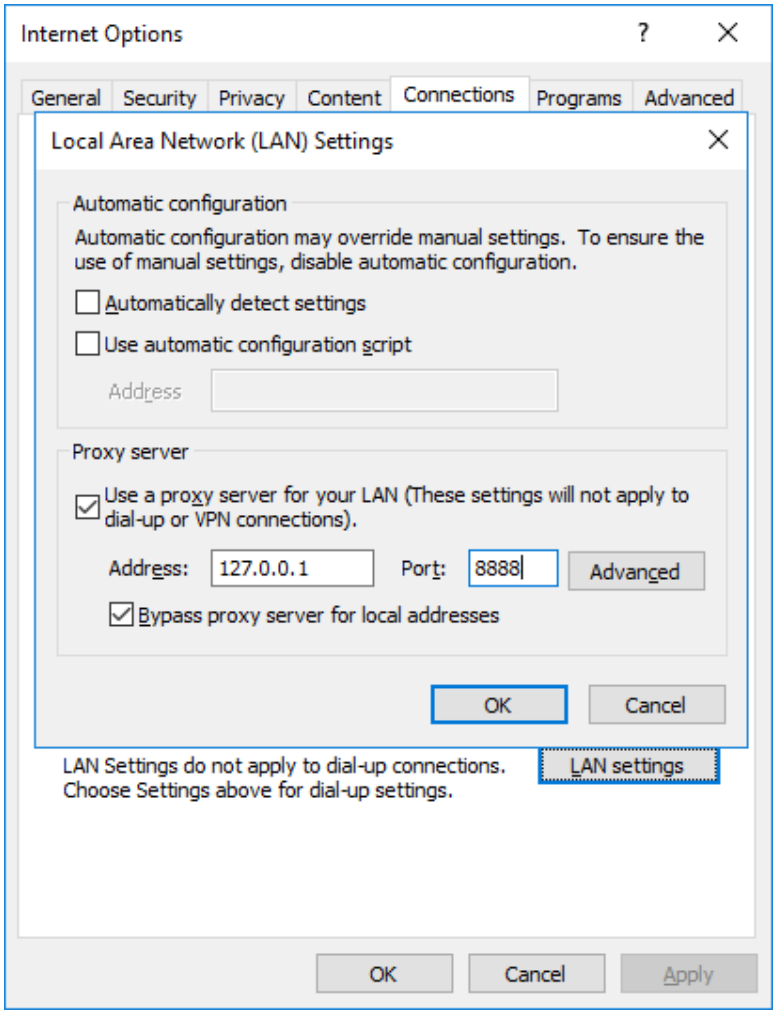


9.2.3.1.4 Configuring your browser to use the JMeter Proxy

Step 1: Open the Web browser. Take Internet Explorer as an example.

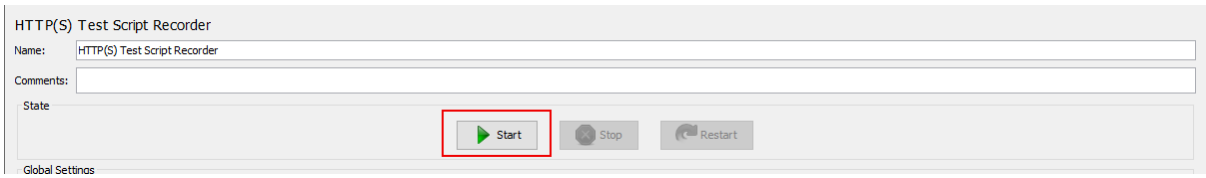
Step 2: Select menu **Tools** > **Internet options**. Select the **Connections** tab and then click the **LAN settings** button.

Step 3: Enter 127.0.0.1 and port 8888 (and make sure the Web API URL is not in the exception list).



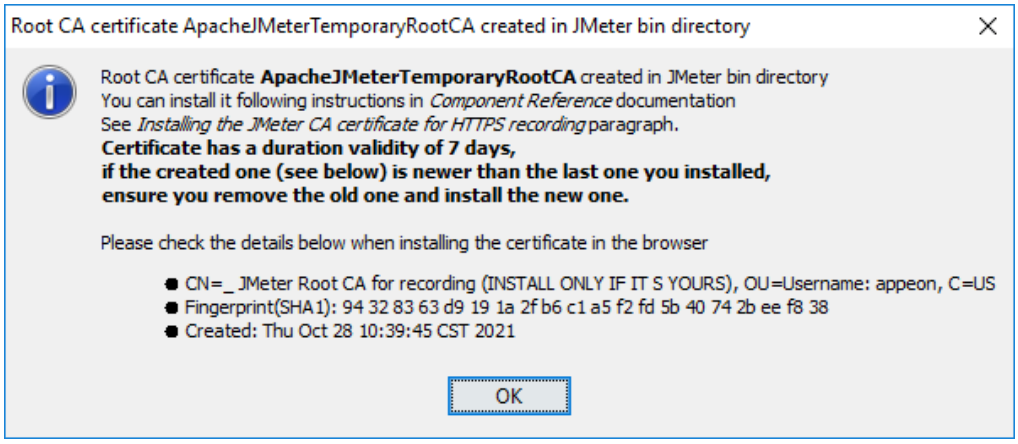
9.2.3.1.5 Recording the scripts

Step 1: Return to **HTTP(S) Test Script Recorder**, and click **Start**.

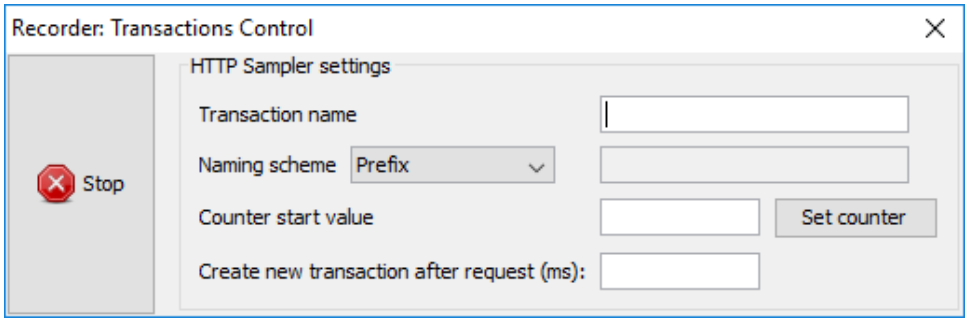


This will start the JMeter proxy server which is used to intercept the browser requests.

Step 2: Click **OK** when prompted to install the certificate as shown below.



Step 3: Keep the "**Recorder: Transactions Control**" window open during recording.



Step 4: Now run the Sales Demo installable cloud app in the Web browser. (Make sure PowerServer Web APIs started before JMeter started. See Running PowerServer Web APIs and then JMeter recorder and Fiddler for why.)

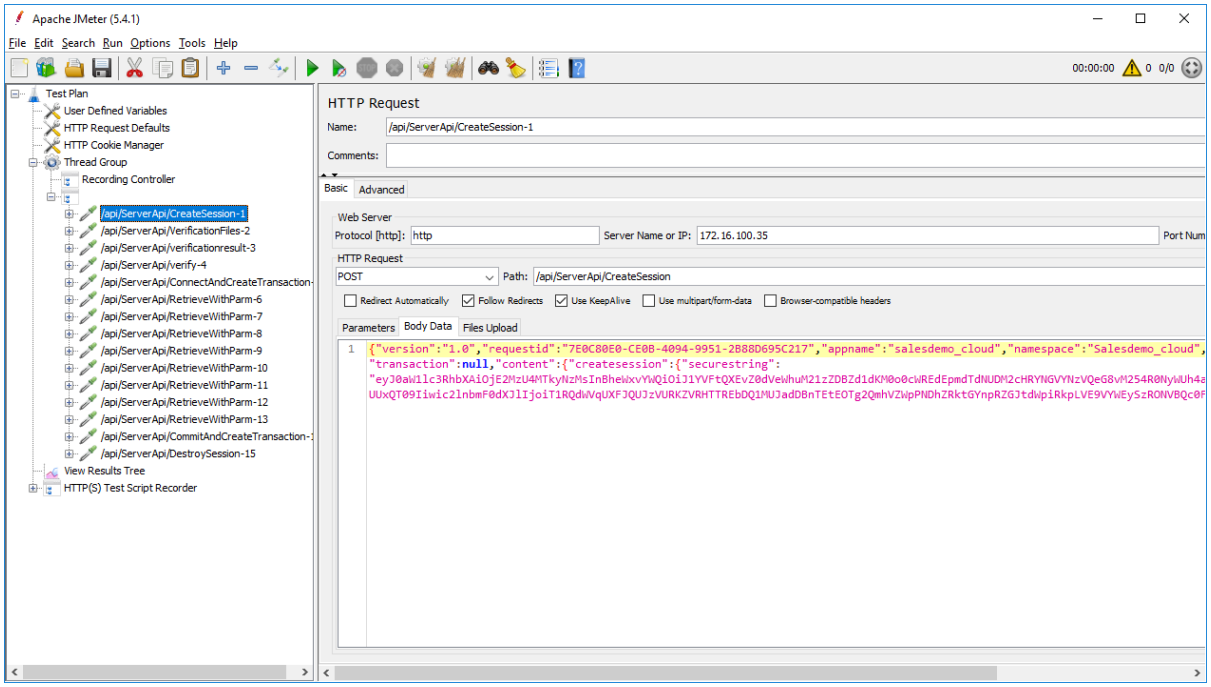
Step 5: Click a few buttons such as Address, Customer etc. in the window and then exit from the application. These HTTP requests will be automatically captured by the script recorder.

Step 6: Close the Web browser and return to the JMeter window.

Step 7: Click the **Stop** button in the "**Recorder: Transactions Control**" window to stop the recording.

9.2.3.1.6 Viewing the recorded scripts

The recorded HTTP requests will be listed in the tree on the left panel. You can manually remove any HTTP requests that are not needed.



9.2.3.1.7 Parameterizing the scripts

To use the dynamic values of the access token, session ID, transaction ID etc., you need to parameterize them in the scripts. See Section 9.2.4, “Parameterization and correlation” for detailed instructions.

9.2.3.2 Recording scripts manually (using Fiddler + JMeter)

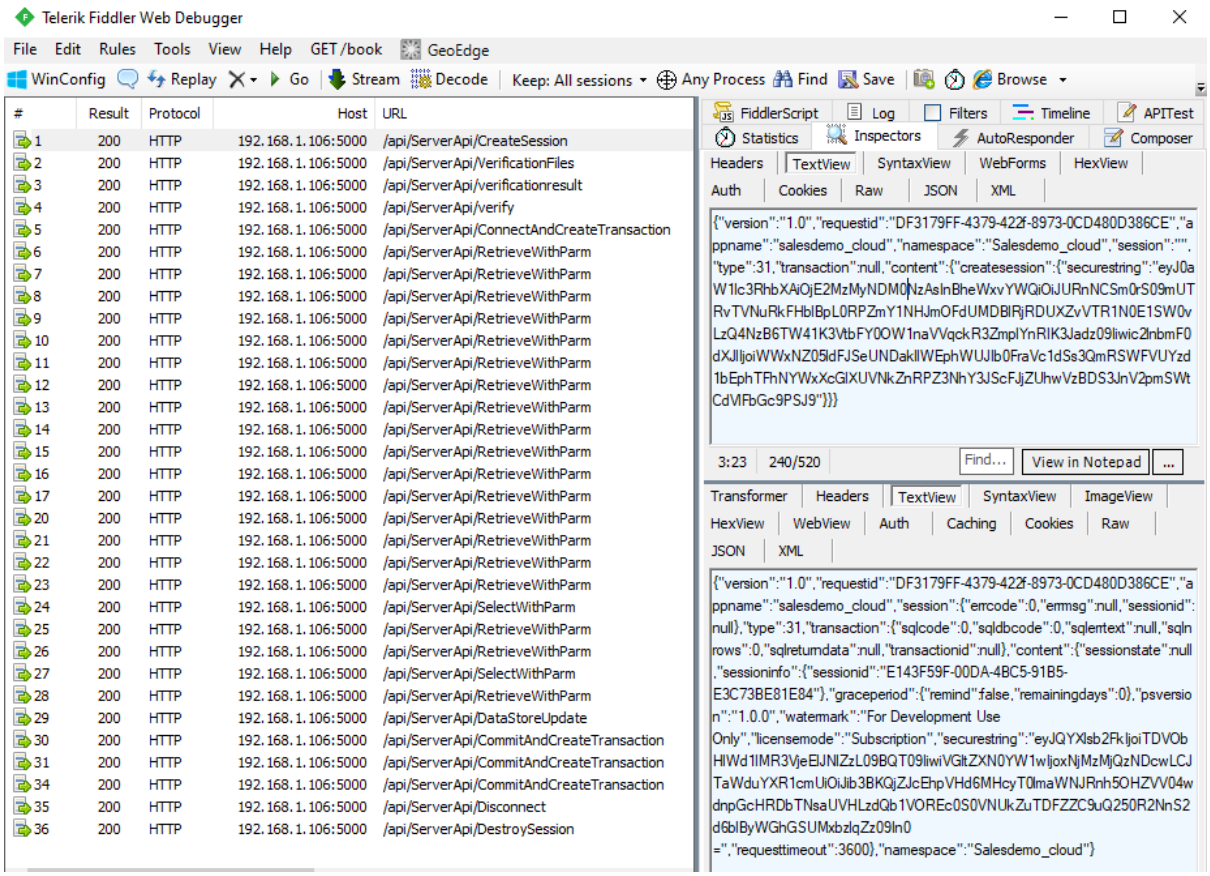
First, you need to use Fiddler (or any other web debugging proxy tool) to get the HTTP requests. For how to use Fiddler, see Debugging with Fiddler.

Then you need to manually add the HTTP requests to JMeter.

9.2.3.2.1 Obtaining HTTP requests

You can view the HTTP requests using Fiddler (or any other web debugging proxy tool).

Fiddler can capture every detail of the HTTP request and the header.

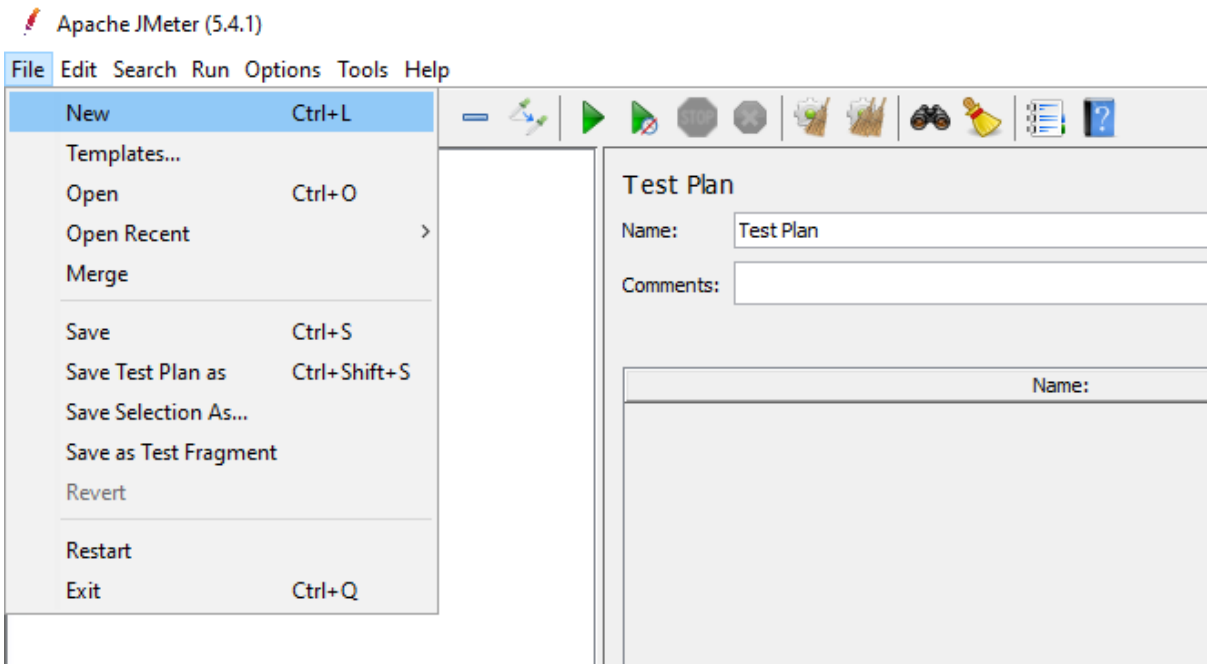


9.2.3.2.2 Creating a test plan and adding HTTP requests

Creating a Test Plan

To create a new test plan, select **File** > **New**. Input a name for the test plan.

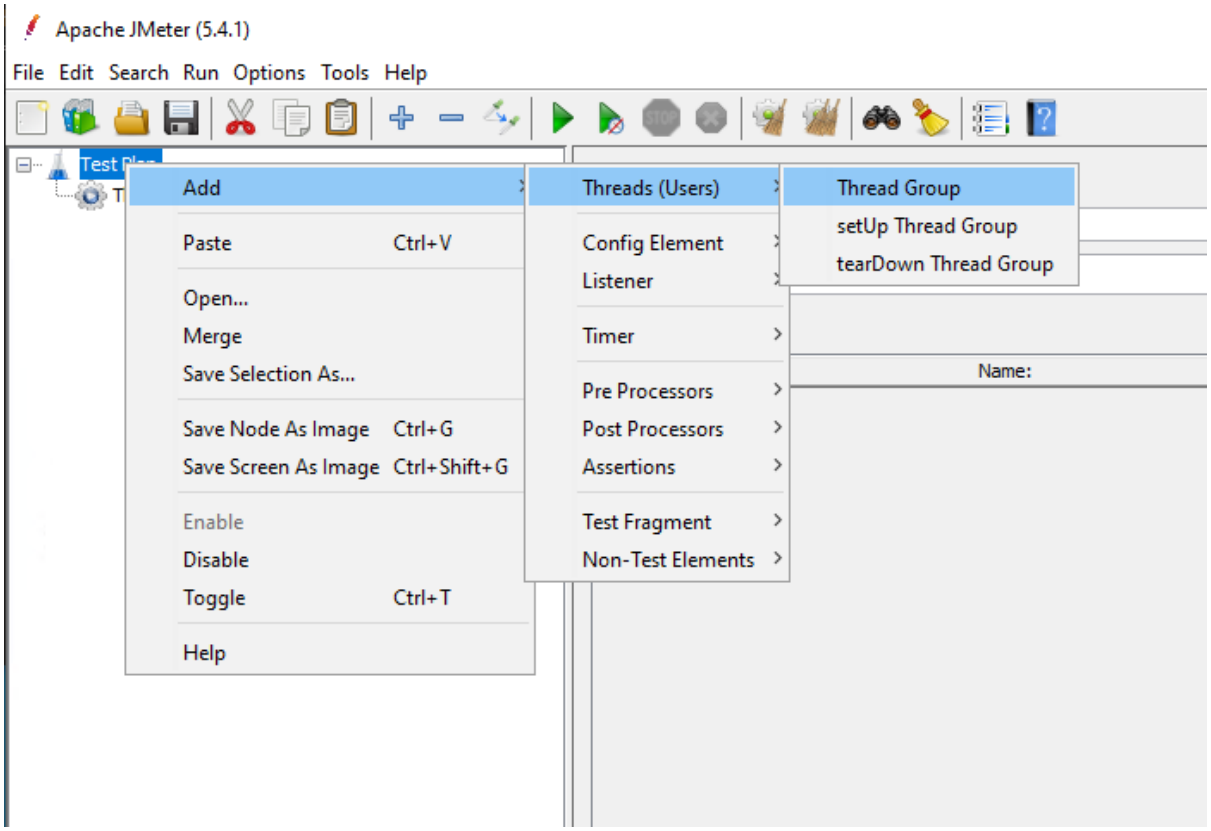
When the test plan is created, it is added to the tree on the left panel. All subsequent elements will be added to this tree in a hierarchical structure.



Adding a Thread Group

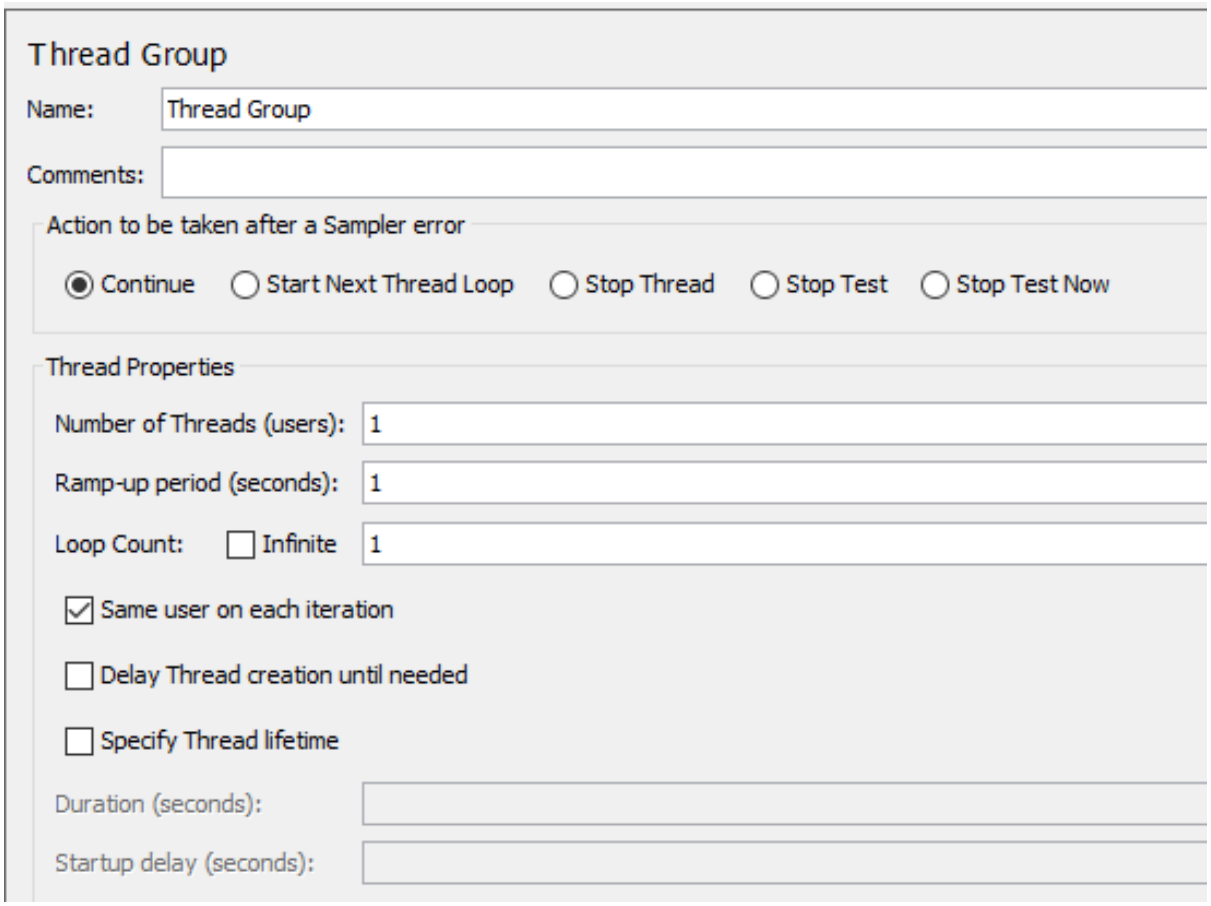
A Test Plan must have at least one thread group. The Thread Group tells JMeter the number of users (threads) you want to simulate, how often users should send requests and how many requests they should send.

To add a Thread Group to the test plan, right click on the test plan that you added just now, and then select **Add** > **Threads (Users)** > **Thread Group**.



You will need to configure the following properties:

- **Number of threads (users)**: how many concurrent users will be accessing the PowerServer Web APIs.
- **Ramp-up period (seconds)**: how long to take to start all users. For example, if set to zero, all users will start immediately. If set the number of users to 100, and ramp-up periods to 50, that means in every second, 2 users will be started.
- **Loop count**: how many times the test should repeat.

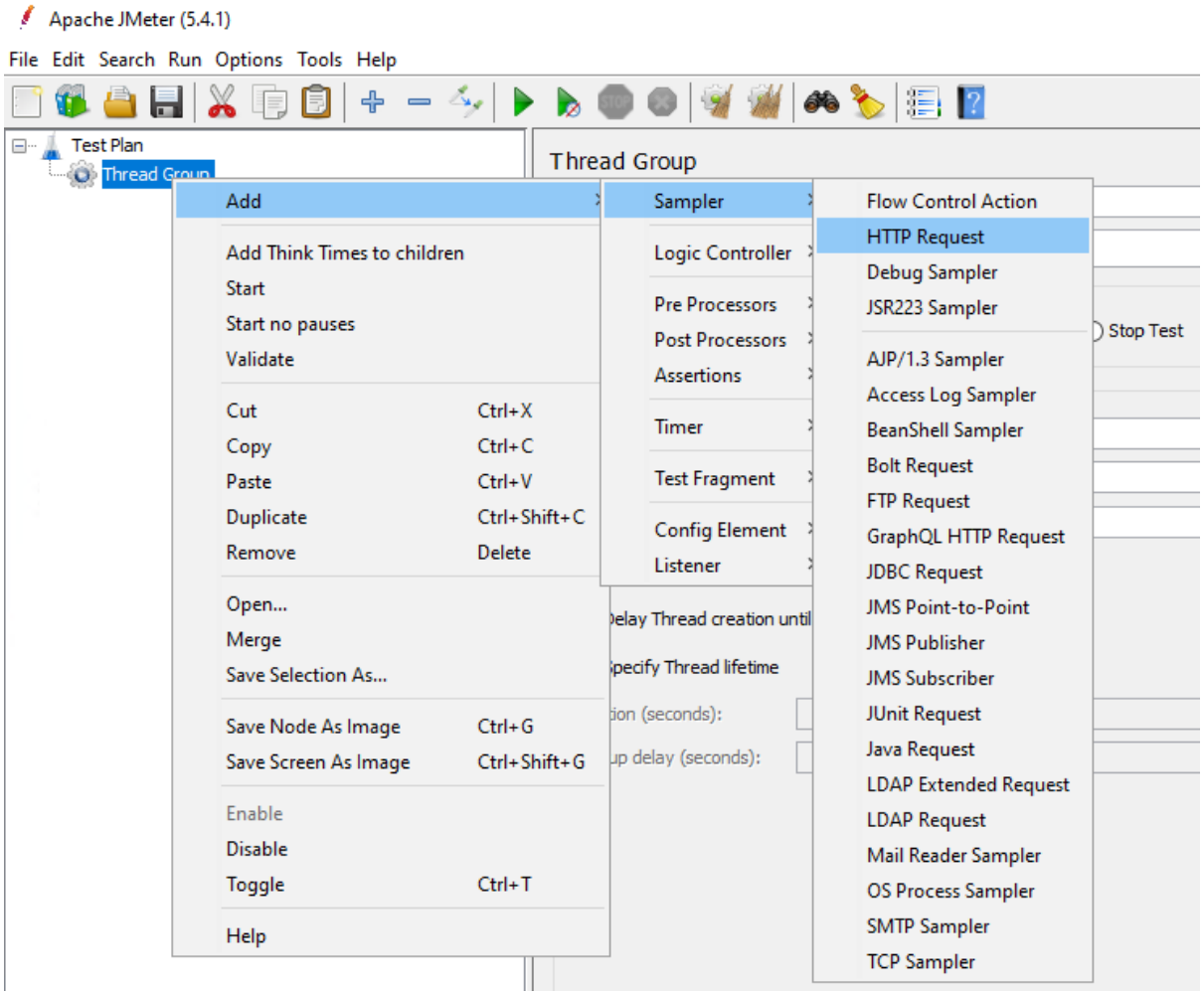


Adding HTTP requests

After you created the test plan and a thread group, you can determine which type of requests to make (such as Web (HTTP/HTTPS), FTP, JDBC, Java etc.)

In this test, you need to make the HTTP request to the PowerServer Web APIs.

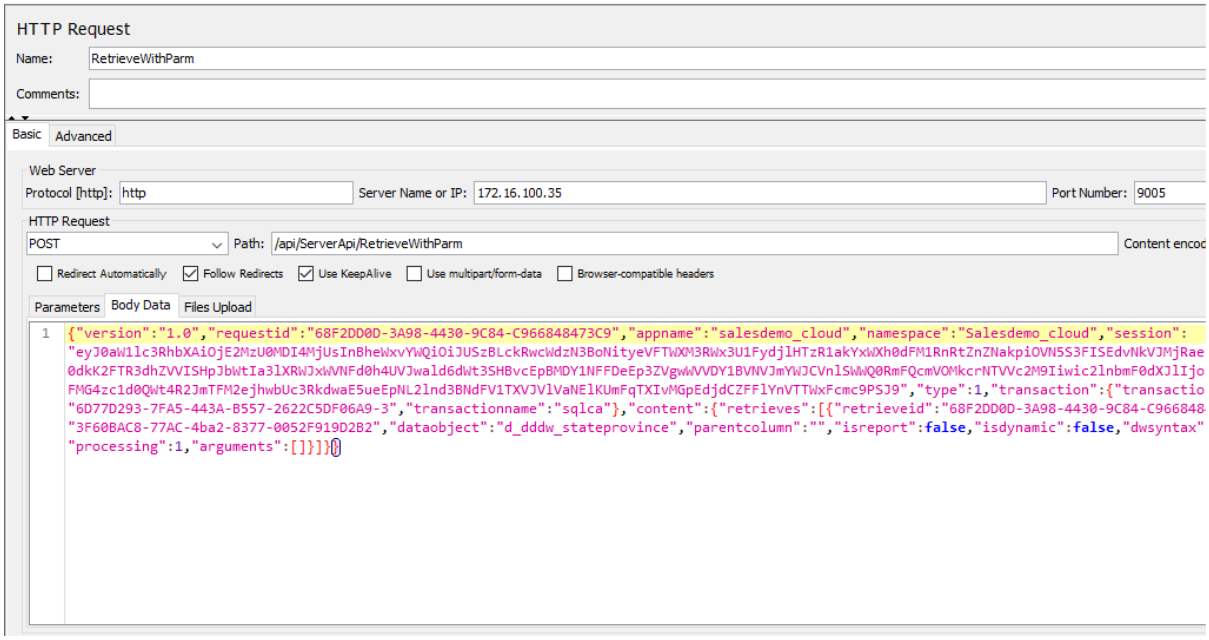
To add an HTTP request, right click on the thread group that you added just now, and then select **Add** > **Sampler** > **HTTP Request**.



When you specify an HTTP Request, you can make use of the information obtained by Fiddler, such as the protocol, server IP, port, HTTP method, path, body data etc.

For example, you can add an HTTP POST request that access the **RetrieveWithParm** Web API, and input the JSON request body to the **Body Data** tab.

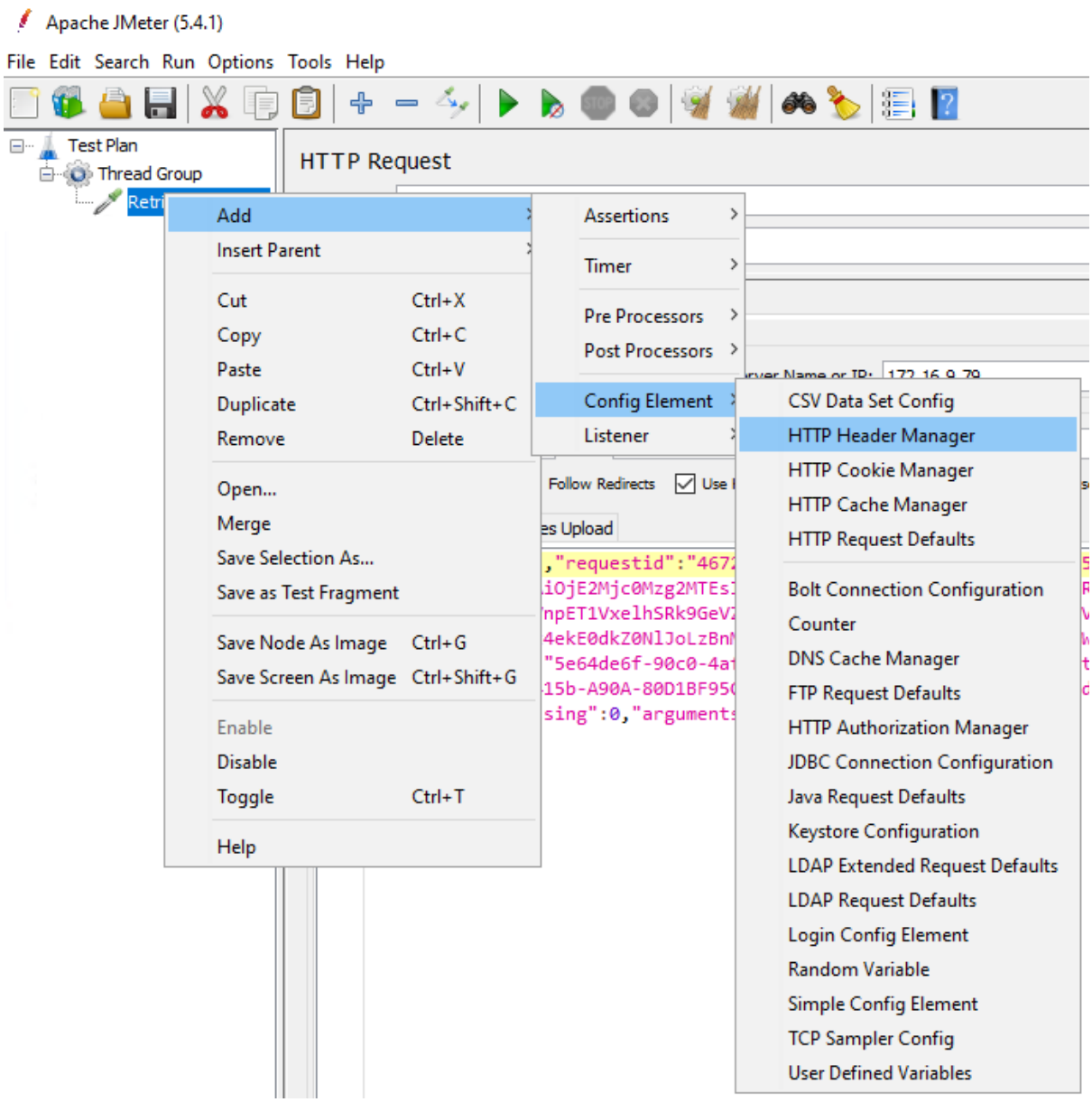
In the same way, you can add requests like GET, POST, PUT, and DELETE.



Adding an HTTP header manager

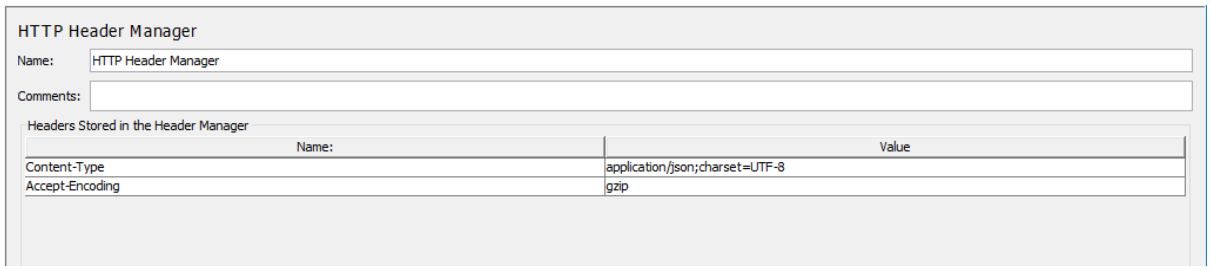
In case you have any specific headers that should be part of the HTTP request, you can add an HTTP header manager. The HTTP header manager lets you add or override HTTP request headers.

To add an HTTP header manager, right click on the HTTP request that you added just now, and then select **Add** > **Config Element** > **HTTP Header Manager**.



When you specify the HTTP header, you can make use of the information obtained by Fiddler.

For example, you can add the **Content-Type** and **Accept-Encoding** to the HTTP header manager.



If all requests will use the same header information, you can use one HTTP Header Manager for all requests (or even for all thread groups), instead of each request having its own HTTP

Header Manager. You can adjust the hierarchical level of the HTTP Header Manager (by drag & drop) in Thread Group.

Adding listeners

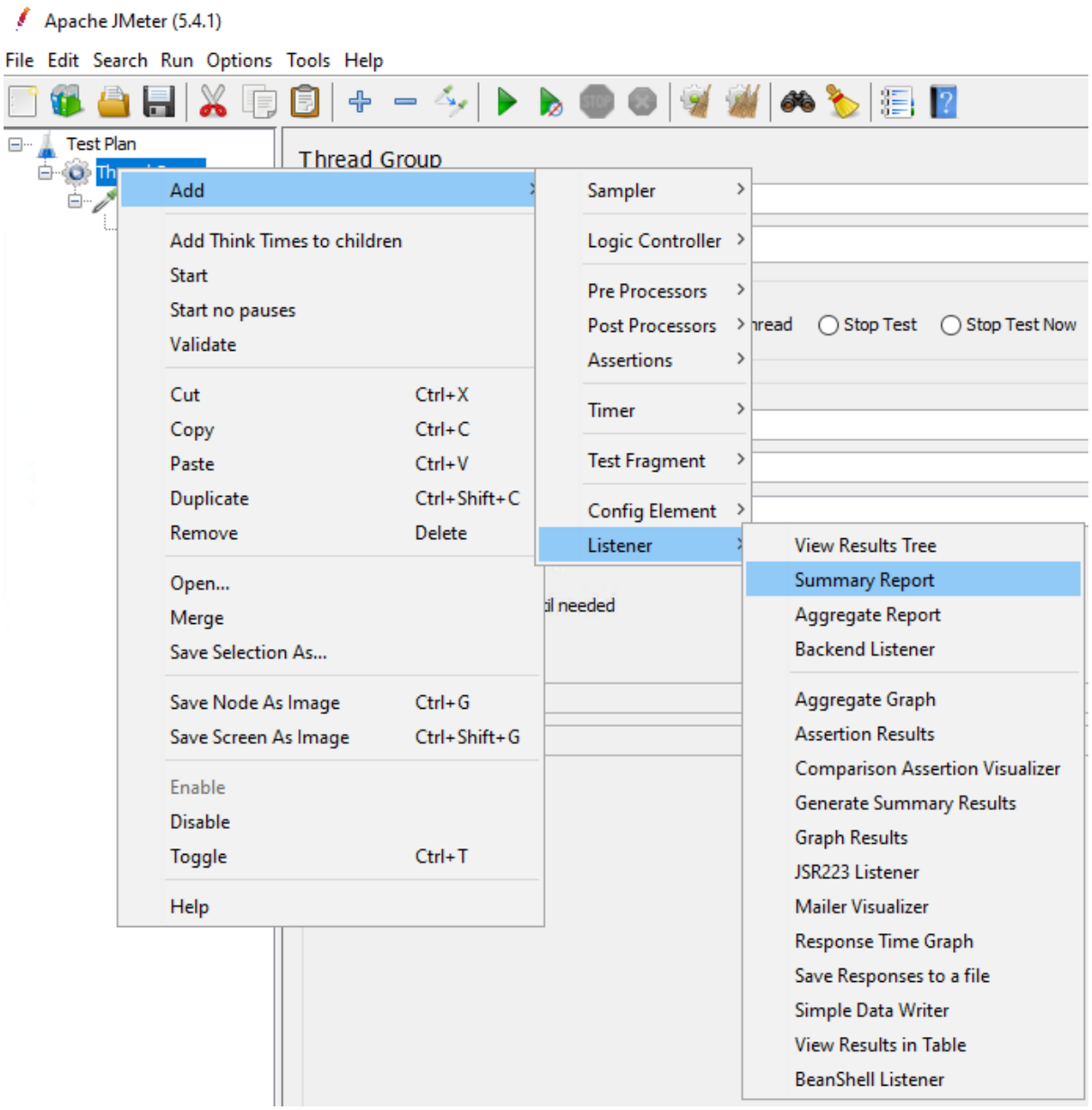
The above is basically everything you need as a minimum setup of an HTTP request suite.

However, in order for you to view the results and statistics of the test, you need to add Listeners. There are several types of listeners such as view results tree, summary report, graph results etc.

- **Summary Report**: you can easily get the performance matrices of each request, such as the number of samples processed, the average response time, throughput, error rate etc.
- **View Results Tree**: you can see all the details related to the request as well as HTTP headers, body size, response code etc. In case any request failed, you can get useful information from this listener for troubleshooting a specific error.
- **Graph Results**: you can see a graphical representation of the throughput vs. the deviation of the tests.
- There are a couple more listeners which you can take some time to explore.

You can add one or more listeners according to your needs.

To add a listener, right click on the thread group, select **Add** > **Listener**, and then choose the listener.



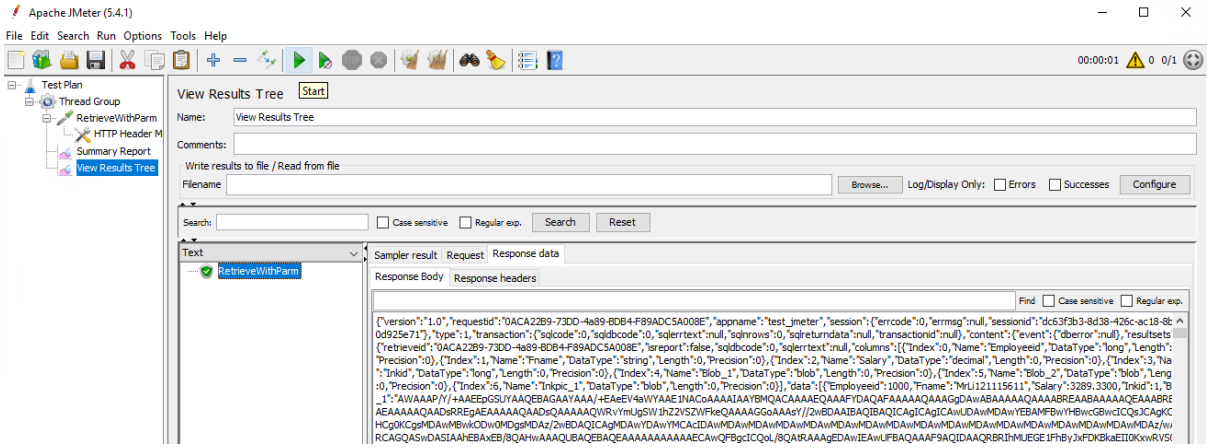
Running tests and viewing results

It is recommended that you save the Test Plan to a file before running it.

To save the Test Plan, select **Save** or **Save Test Plan As…** from the **File** menu. The test scripts will be saved in a JMX file. You can then add this file to your project repository, and other members of your team can load it on their own JMeter tools as well.

Now you can start the test by clicking the **Start** button on the toolbar. This will start the thread group and the results will be captured by the listener.

To run the test again, clean up the previous result by clicking the **Clear All** button on the toolbar.



9.2.3.3 Parameterizing the Retrieve test

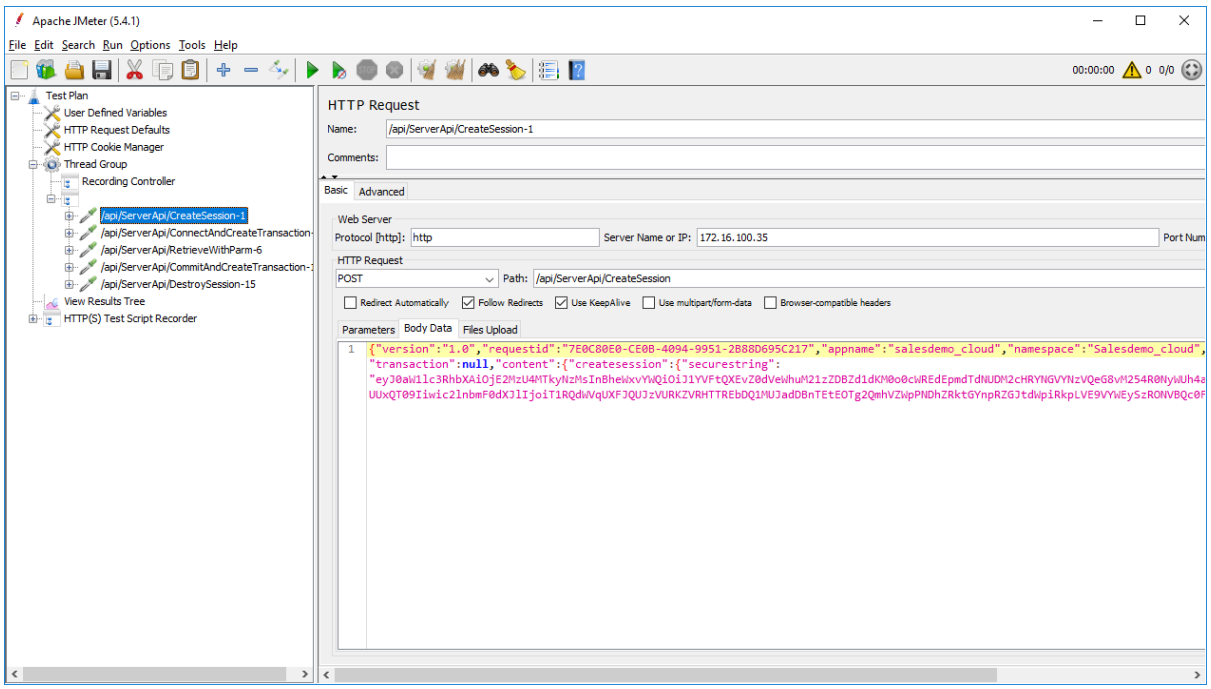
You can keep running the same test without parameterizing the session ID, because the session timeout value is 3600 seconds by default, and if you keep repeating the test within 3600 seconds, the session will stay valid until PowerServer Web APIs is restarted or the application is closed.

Of course, you can also parameterize the session ID and transaction ID so that they are always correlated with the dynamic values instead of static ones.

In this section, you will learn how to parameterize the session ID and transaction ID for the Retrieve test. You will learn more about parameterization and correlation for the access token, retrieval arguments, and ESQL parameters in Section 9.2.4, “Parameterization and correlation”.

This section will reuse the test plan and thread group that was just recorded in Section 9.2.3.1, “Recording scripts automatically (using Recorder)”.

Clean up the recorded test plan by removing any duplicated and unnecessary requests. Suppose the test plan looks like this after cleanup:



Now you will add two other thread groups:

- **setUp Thread Group**: contains the pre-test actions such as creating the session, connecting to the database, starting the transaction etc.
- **tearDown Thread Group**: contains the post-test actions such as committing a transaction, disconnecting from the database, destroying the session etc.

9.2.3.3.1 Adding a setUp Thread Group

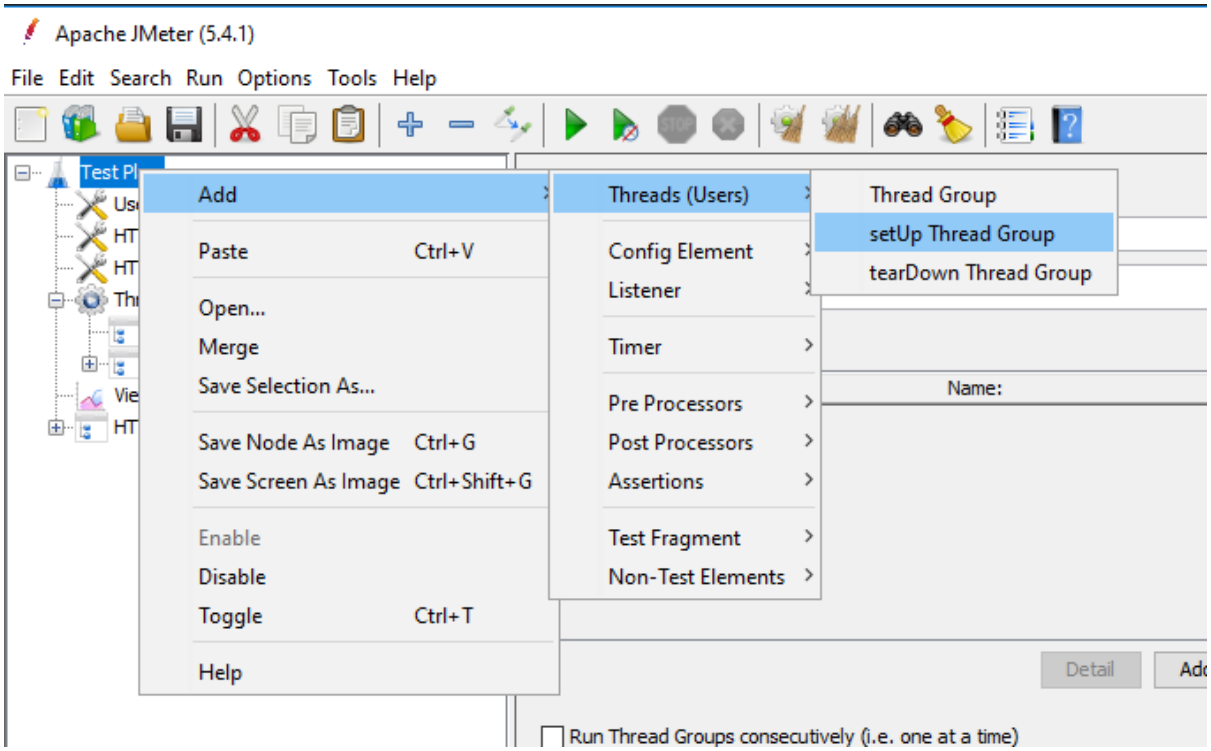
The setUp Thread Group is used when you need to run initial actions to prepare the testing environment, prior to starting your main test. These actions should be configured within the setUp Thread Group and not within the regular Thread Group that you will use for running your load test.

In this tutorial, you will run the following pre-test actions via setUp Thread Group:

- Create the user session
- Connect to the database
- Start the transaction

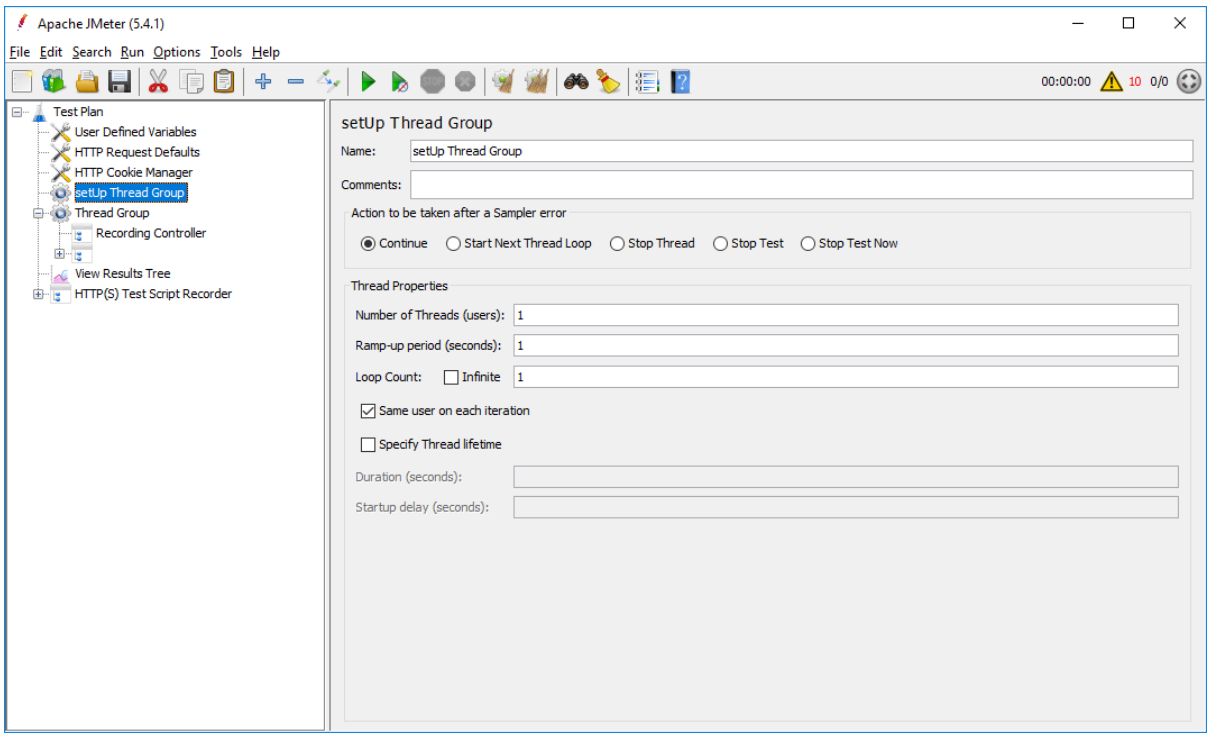
Adding a setUp Thread Group

To add a setUp Thread Group to the test plan, right click on the test plan that you added just now, and then select **Add** > **Threads (Users)** > **setUp Thread Group**.



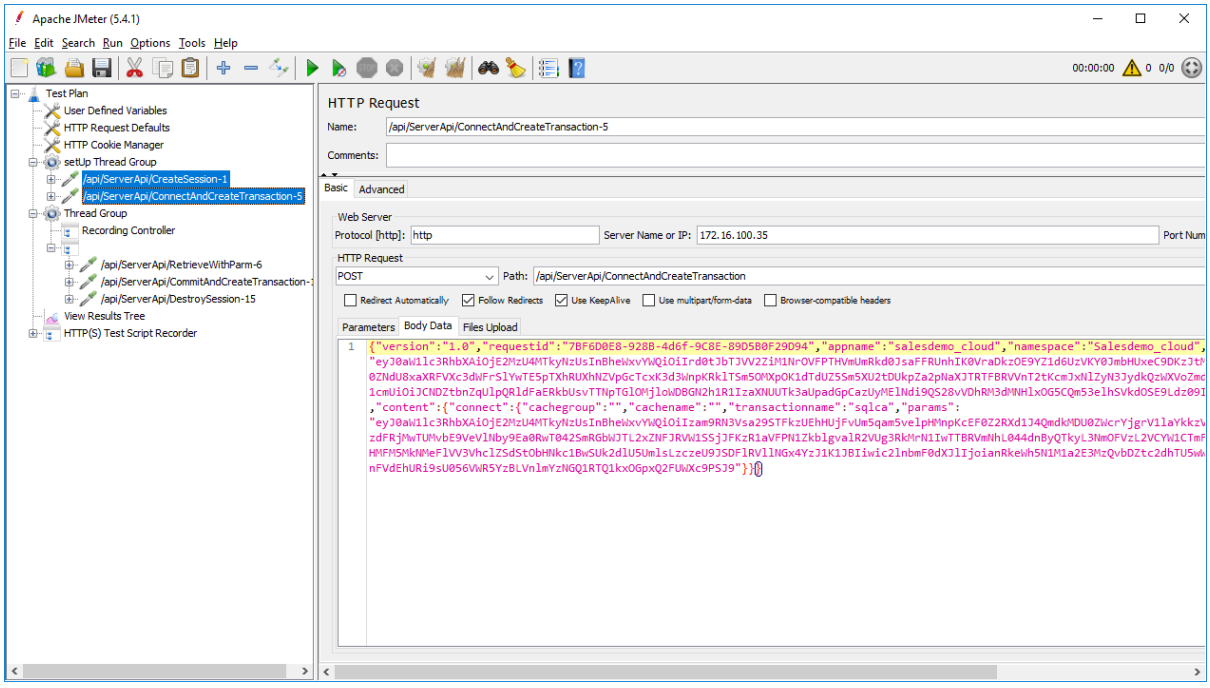
Make sure this setUp Thread Group is listed as the first thread group under the test plan, so it starts before the other thread groups. You can drag the item in the tree to adjust their level and order.

You can use the default settings for the setUp Thread Group.



Adding HTTP requests

Now you can move (by drag & drop) the following HTTP requests from Thread Group to the setUp Thread Group.



Parameterizing the session ID

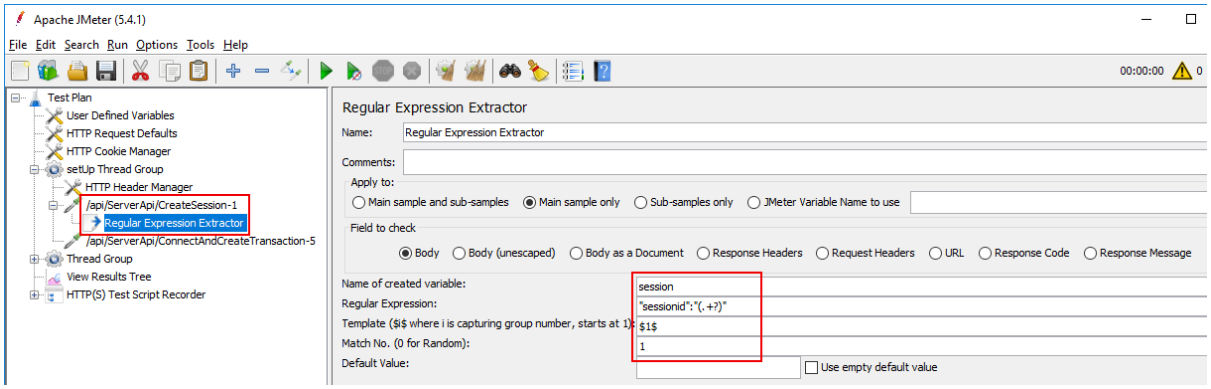
To parameterize the session ID, you can first add a **Regular Expression Extractor** to save the session ID to a local variable; and then add a **BeanShell PostProcessor** to set the local variable as a global property, so that it can be shared in all thread groups.

To add a **Regular Expression Extractor**, right click on the **CreateSession** request and then select **Add** > **Post Processors** > **Regular Expression Extractor**.

Specify the **Regular Expression Extractor** like this.

- **Name of created variable**: “session” or any name you prefer
- **Regular Expression**: "sessionid":"(.+?)"
- **Template**: 1
- **Match No.**: 1

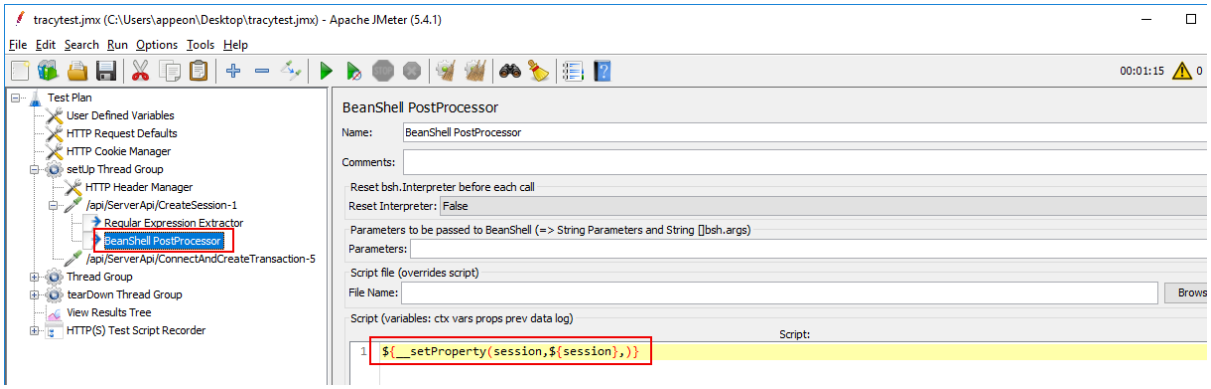
The session ID will be saved to the “`session`” variable.



To add a **BeanShell PostProcessor**, right click on the **CreateSession** request, and then select **Add** > **Post Processors** > **BeanShell PostProcessor**.

Input the following script: `${__setProperty(session,${session},)}`

The “`session`” variable becomes a JMeter global property.



Use the **Search** menu to search for all occurrences of “session” in all thread groups, and then replace the static value of session ID with the global property `${__property(session,,)}`.

Parameterizing the transaction ID

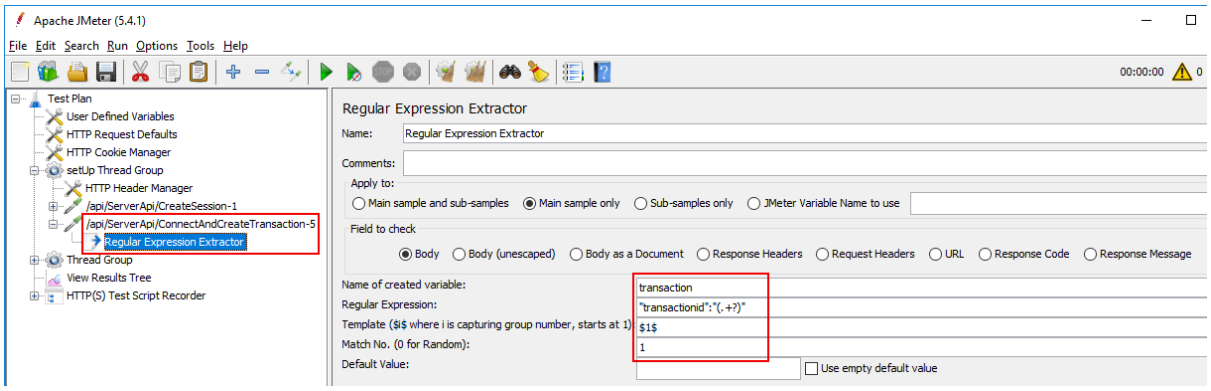
Similar to the session ID, you can first add a **Regular Expression Extractor** to save the transaction ID into a local variable; and then add a **BeanShell PostProcessor** to set the local variable as a global property, so that it can be shared in all thread groups.

To add a **Regular Expression Extractor**, right click on the **ConnectAndCreateTransaction** request and then select **Add** > **Post Processors** > **Regular Expression Extractor**.

Specify the **Regular Expression Extractor** like this:

- **Name of created variable**: “transaction” or any other name you prefer
- **Regular Expression**: "transactionid":"(.+?)"
- **Template**: 1
- **Match No.**: 1

The transaction ID will be saved to the “`transaction`” variable.



To add a **BeanShell PostProcessor**, right click on the **ConnectAndCreateTransaction** request and then select **Add** > **Post Processors** > **BeanShell PostProcessor**.

Input the following script: `${__setProperty(transaction, ${transaction},)}`

The “`transaction`” variable becomes a JMeter global property.

Use the **Search** menu to search for all occurrences of “transactionid” in all thread groups, and then replace the static value of transaction ID with the global property `${__property(transaction,,)}`.

9.2.3.3.2 Adding a tearDown Thread Group

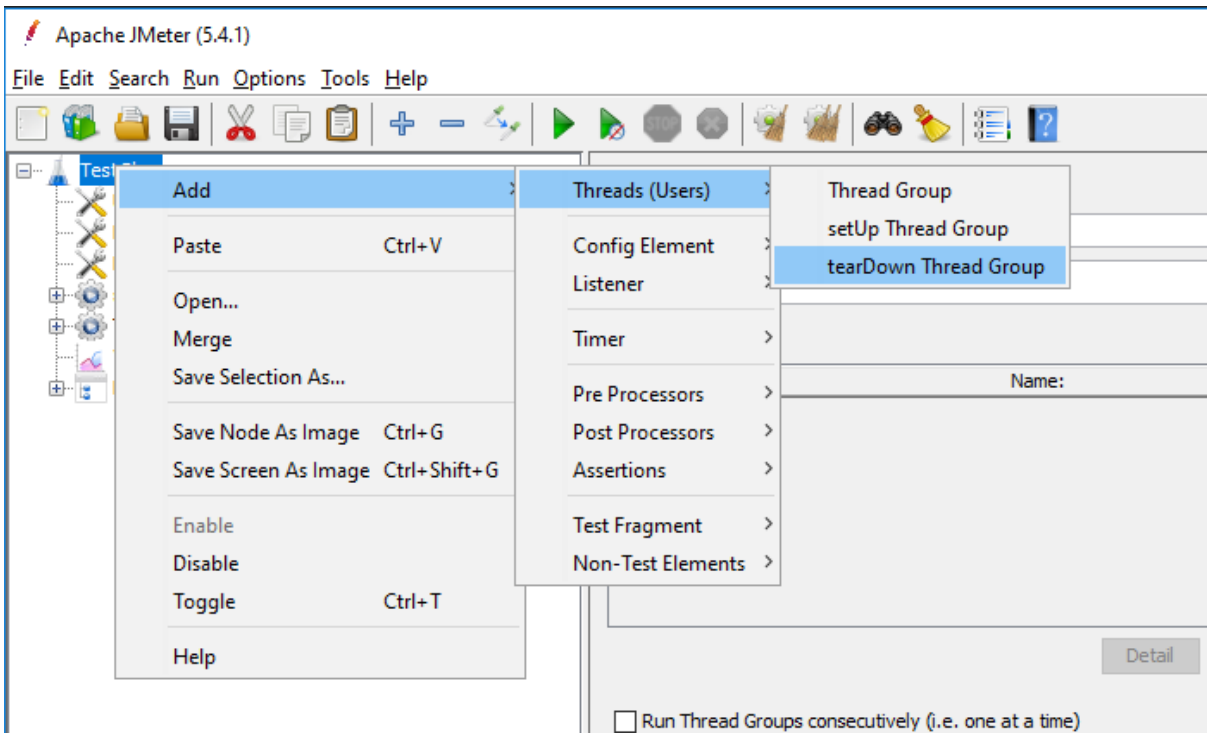
The tearDown Thread Group is used to perform post-test actions. These actions should be configured within the tearDown Thread Group and not within the regular Thread Group that you will use for running your load test.

In this tutorial, you will run the following post-test actions via tearDown Thread Group:

- Commit the transaction
- Destroy the user session

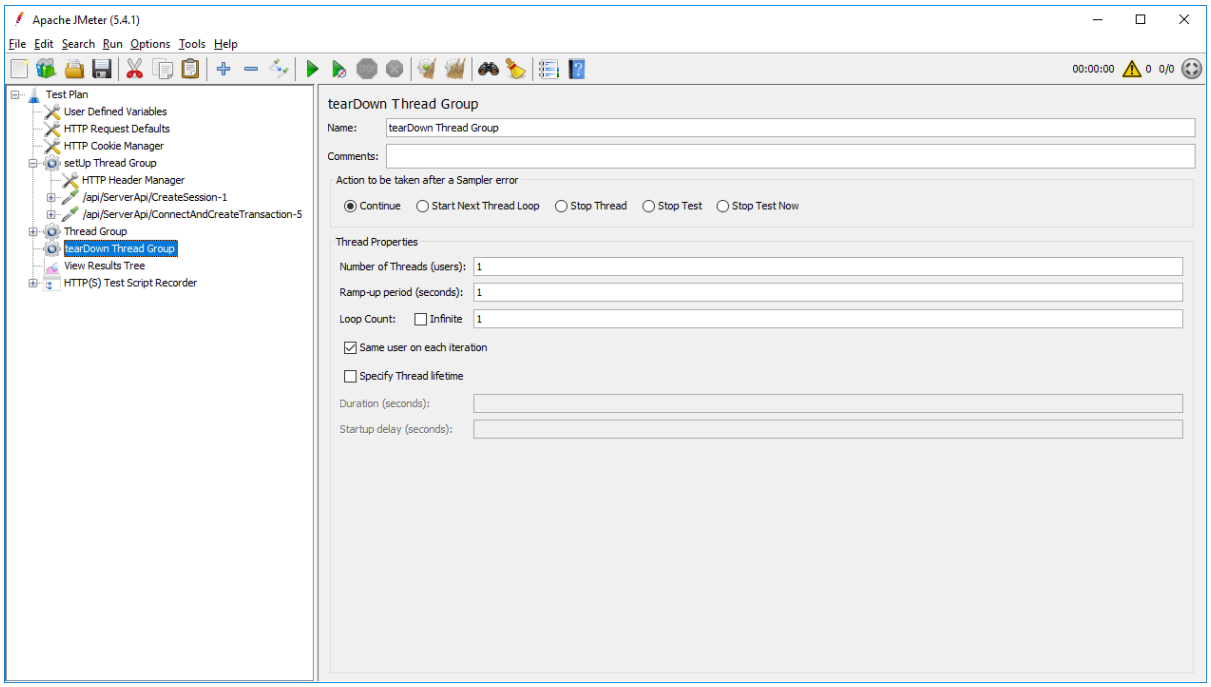
Adding a tearDown Thread Group

To add a tearDown Thread Group to the test plan, right click on the test plan that you added just now, and then select **Add** > **Threads (Users)** > **tearDown Thread Group**.



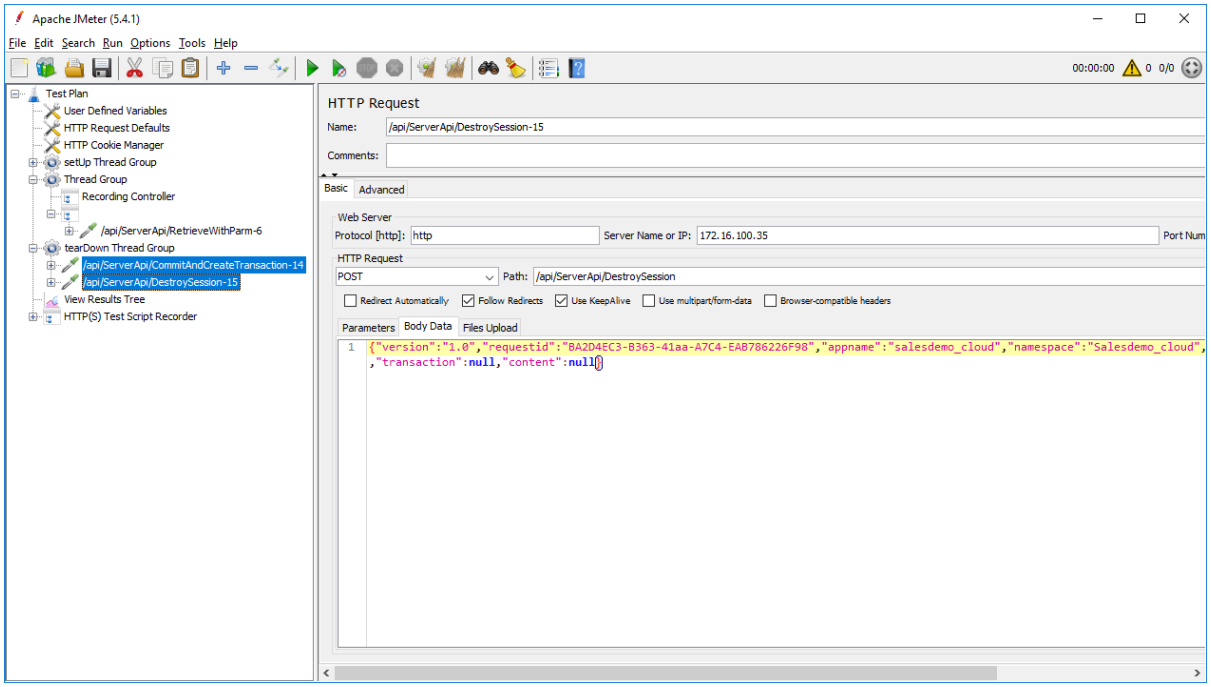
Make sure this tearDown Thread Group is listed after Thread Group, so it starts when Thread Group has finished.

You can use the default settings for the tearDown Thread Group. Normally the tearDown Thread Group runs only once.



Adding HTTP requests

Now you can move (by drag & drop) the following HTTP requests from Thread Group to the setUp Thread Group.



Parameterizing the transaction ID

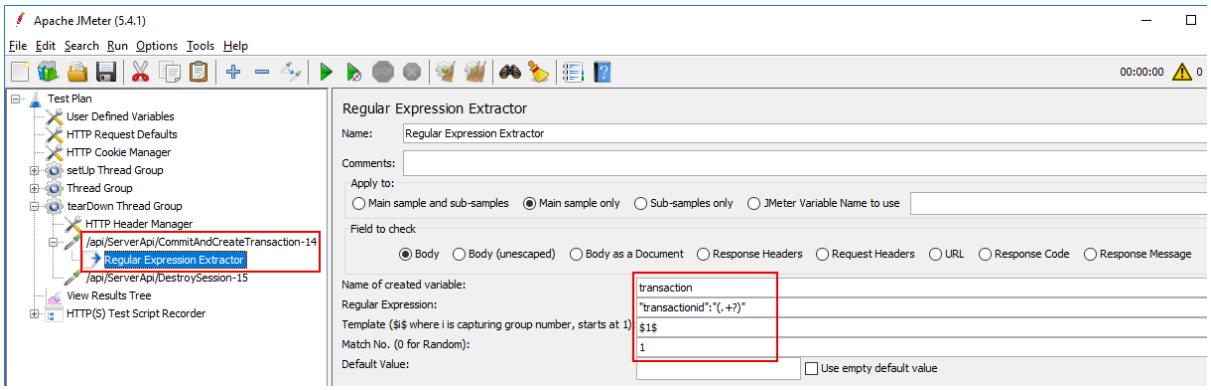
You can first add a **Regular Expression Extractor** to save the transaction ID into a local variable; and then add a **BeanShell PostProcessor** to set the local variable as a global property, so that it can be shared in all thread groups.

To add a **Regular Expression Extractor**, right click on the **CommitAndCreateTransaction** request and then select **Add** > **Post Processors** > **Regular Expression Extractor**.

Specify the **Regular Expression Extractor** like this:

- **Name of created variable**: “transaction” or any other name you prefer
- **Regular Expression**: "transactionid":"(.+?)"
- **Template**: 1
- **Match No.**: 1

The transaction ID will be saved to the “`transaction`” variable.



To add a **BeanShell PostProcessor**, right click on the **CommitAndCreateTransaction** request and then select **Add** > **Post Processors** > **BeanShell PostProcessor**.

Input the following script: `${__setProperty(transaction, ${transaction},)}`

The "`transaction`" variable becomes a JMeter global property.

Use the **Search** menu to search for all occurrences of "transactionid" in all thread groups, and then replace the static value of transaction ID with the global property `${__property(transaction,,)}`.

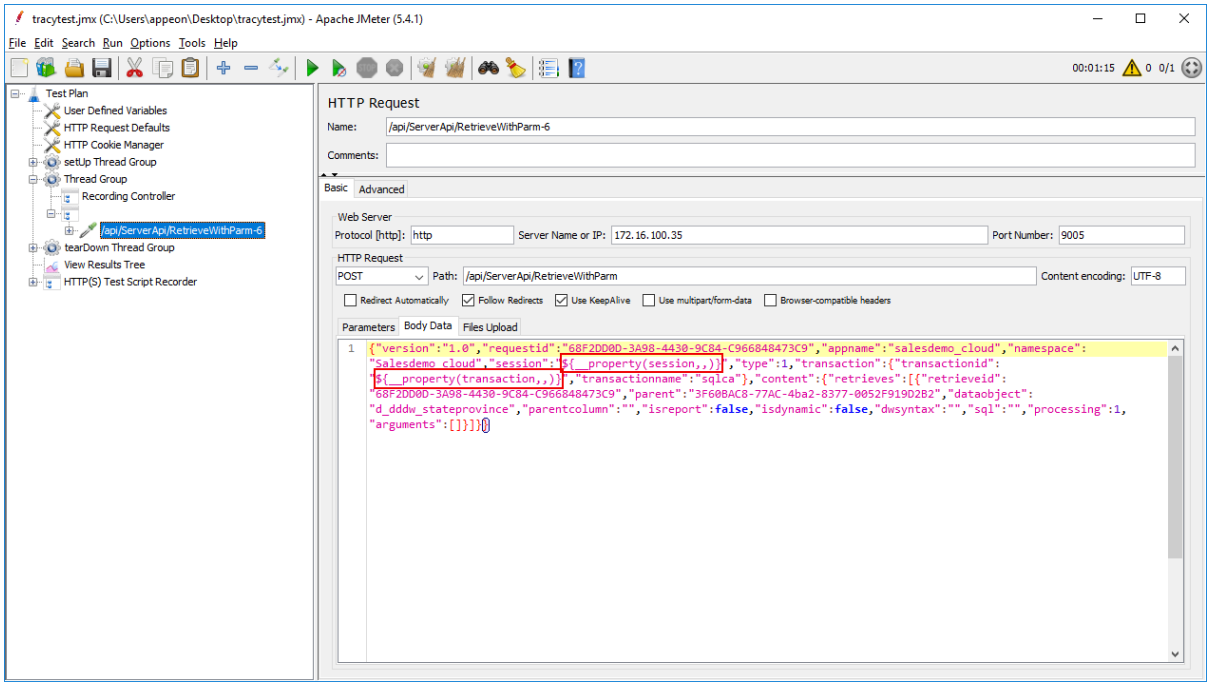
9.2.3.3.3 Configuring Thread Group

After moving requests to the setUp Thread Group and the tearDown Thread Group, now the regular Thread Group contains only one **RetrieveWithParm** request.

Go to the **Body Data** of the **RetrieveWithParm** request, and make sure the session ID and the transaction ID are replaced with the global property:

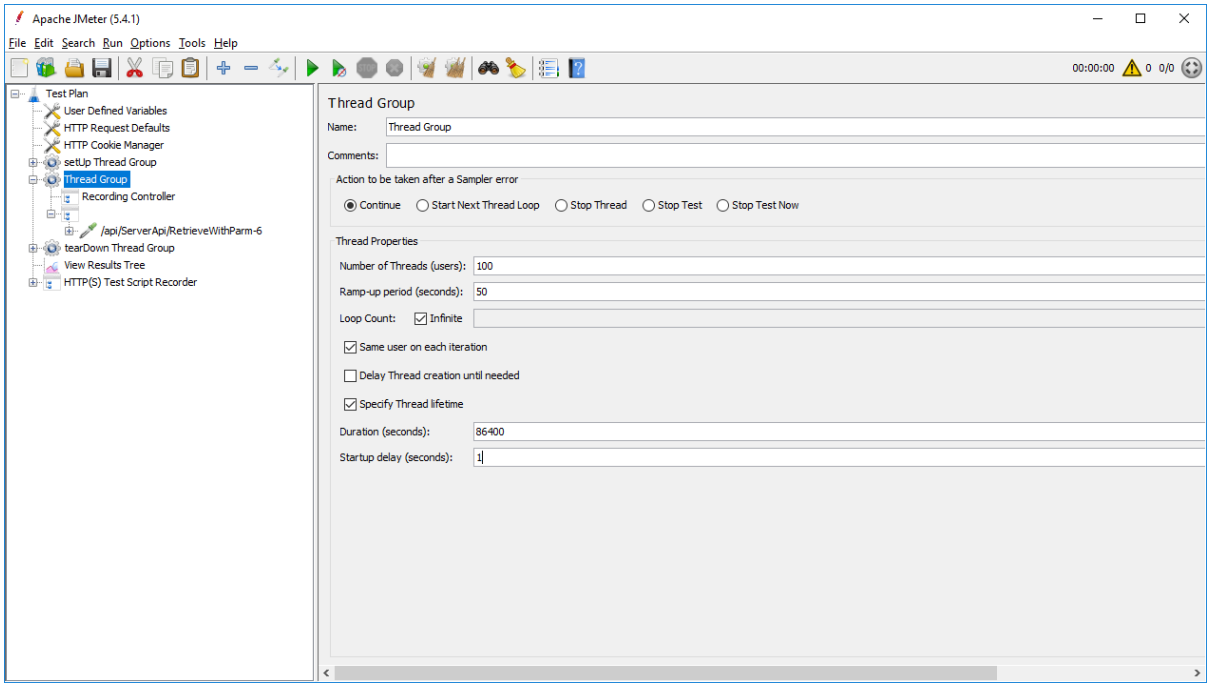
`${__property(session,,)}`

`${__property(transaction,,)}`



Configure the Thread Group according to your needs.

For example, set the number of users to 100, ramp-up period to 50 seconds, which means 2 requests are made in every second (set ramp-up to 0 will start all 100 users at one time). Set loop count to infinite and duration to 86400 seconds, which means the test will be run repeatedly in 24 hours.



9.2.4 Parameterization and correlation

9.2.4.1 Why parameterization and correlation are required

Parameterization and correlation are required for unique/dynamic values that are generated by the server. In the case of PowerServer, the access token, session ID, and transaction ID are all unique/dynamic values generated by PowerServer at runtime. If you re-play the scripts without first changing the value recorded, the scripts will fail, because the dynamic value generated by PowerServer does not match with the value recorded.

Therefore, after the scripts are recorded, you need to find out all occurrences of the access token, session ID, and transaction ID in the script and replace them with variables.

In some cases, dynamic values also refer to the retrieval arguments, ESQL parameters etc.

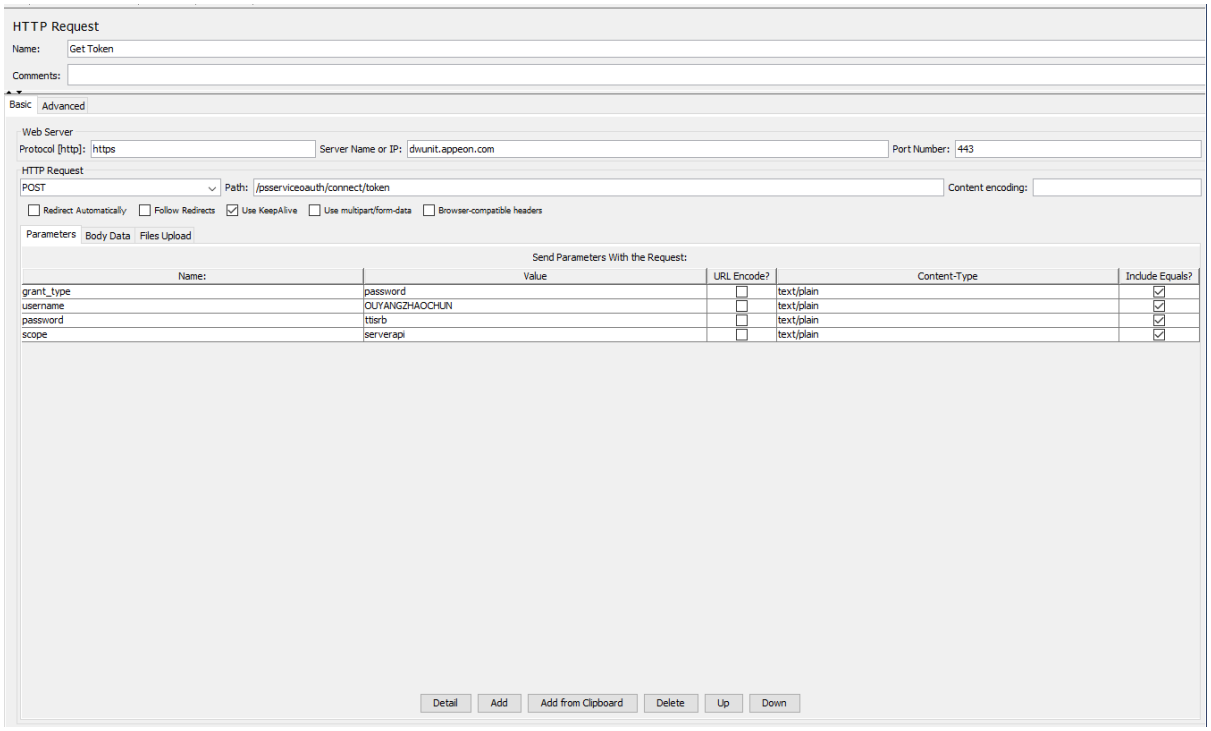
9.2.4.2 Parameterizing the access token

To parameterize the access token, you can use the following

- a **Regular Expression Extractor** that saves the access token into a local variable
- a **BeanShell Sampler** that calls the “setProperty” function to set the local variable as a global property, so that it can be shared in all thread groups

In the case of PowerServer, the **GetToken** request gets the access token, therefore, you add a **Regular Expression Extractor** to the **GetToken** request to get and save the token into a local variable.

A **GetToken** request will look like this:

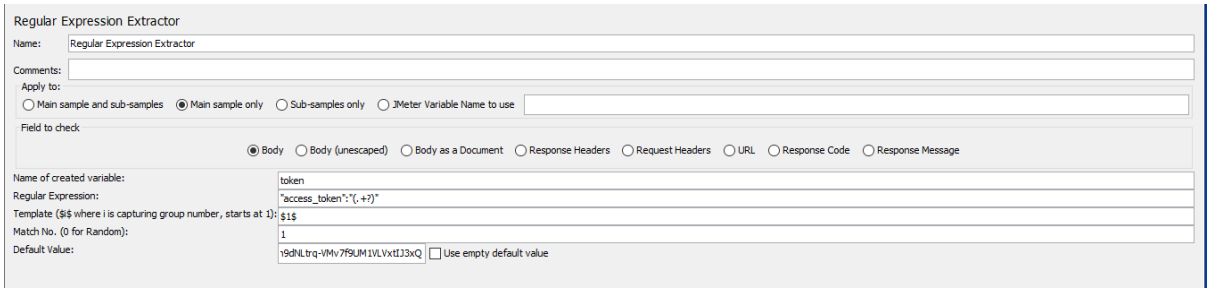


To add a **Regular Expression Extractor**, right click on the **GetToken** request and then select **Add** > **Post Processors** > **Regular Expression Extractor**.

Specify the **Regular Expression Extractor** like this.

- **Name of created variable**: “token” or any name you prefer
- **Regular Expression**: "access_token":"(.+?)"
- **Template**: 1
- **Match No.**: 1

The access token will be saved to the “`token`” variable. You can invoke the local variable by typing `${token}` in the requests (bodies and headers).



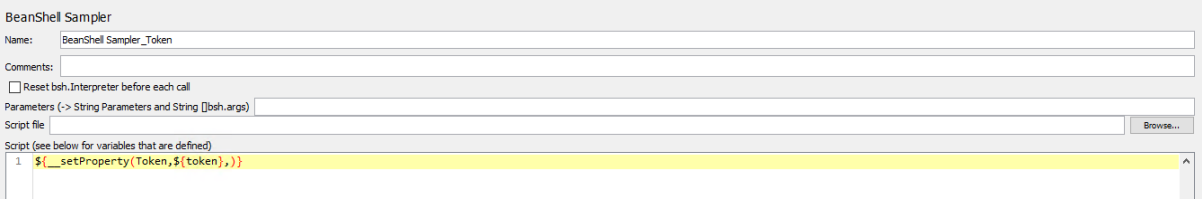
If you want to make the “`token`” variable a global property that can be accessed by all threads and thread groups, you can add a **BeanShell Sampler** or **BeanShell PostProcessor** to call the JMeter “setProperty” function. The “setProperty” function can set the “`token`” variable as a global property. Section 9.2.3.3, “Parameterizing the Retrieve test” has instructions for how to add a **BeanShell PostProcessor**. This section will show how to add a **BeanShell Sampler**.

To add a **BeanShell Sampler**, right click on setUp Thread Group, and then select **Add** > **Sampler** > **BeanShell Sampler**.

- Use the default name or input any name you prefer.
- Input the following script:

  ```
  ${__setProperty(token,${token},)}
  ```

The "`token`" variable becomes a JMeter global property.



Use the **Search** menu to search for all occurrences of "access_token" in all thread groups, and replace the static value of token with the global property `${__property(token,,)}`.

9.2.4.3 Parameterizing the session ID

To parameterize the session ID, you can use the following

- a **Regular Expression Extractor** that saves the session ID into a local variable
- a **BeanShell Sampler** that calls the "setProperty" function to set the local variable as a global property, so that it can be shared in all thread groups

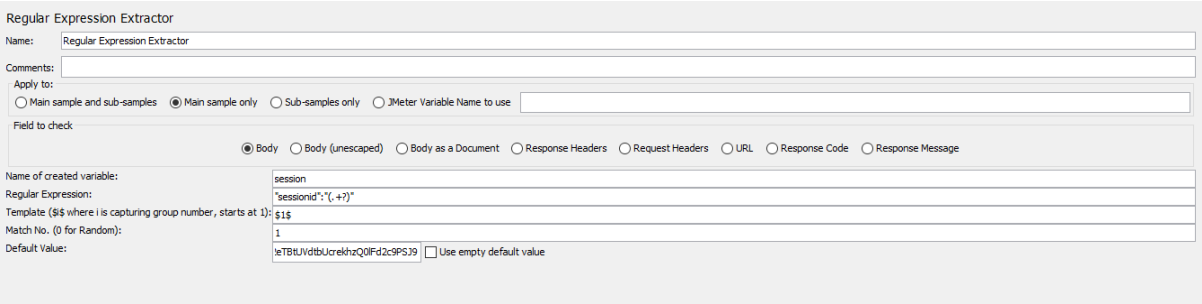
In the case of PowerServer, the **CreateSession** request creates the session ID, therefore, you add a **Regular Expression Extractor** to the **CreateSession** request to get and save the session ID into a local variable.

To add a **Regular Expression Extractor**, right click on the **CreateSession** request and then select **Add** > **Post Processors** > **Regular Expression Extractor**.

Specify the **Regular Expression Extractor** like this.

- **Name of created variable**: "session" or any name you prefer
- **Regular Expression**: "sessionid":"(.+?)"
- **Template**: 1
- **Match No.**: 1

The session ID will be saved to the "`session`" variable. You can invoke the local variable by typing `${session}` in the requests (bodies and headers).



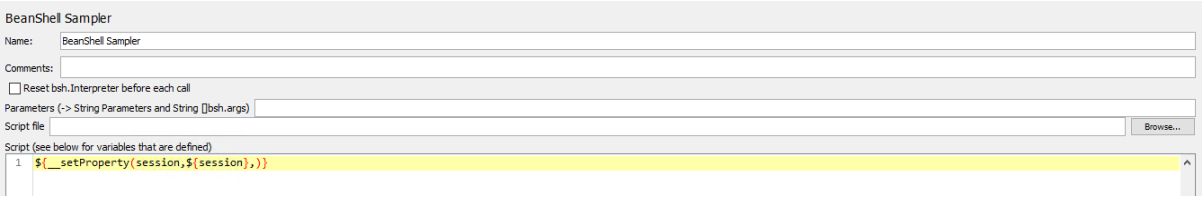
If you want to make the "`session`" variable a global property that can be accessed by all threads and thread groups, you can add a **BeanShell Sampler** or **BeanShell PostProcessor** to call the JMeter "setProperty" function. The "setProperty" function will set the "`session`" variable as a global property. Section 9.2.3.3, "Parameterizing the Retrieve test" has instructions for how to add a **BeanShell PostProcessor**. This section will show how to add a **BeanShell Sampler**.

To add a **BeanShell Sampler**, right click on setUp Thread Group, and then select **Add** > **Sampler** > **BeanShell Sampler**.

- Use the default name or input any name you prefer.
- Input the following script:

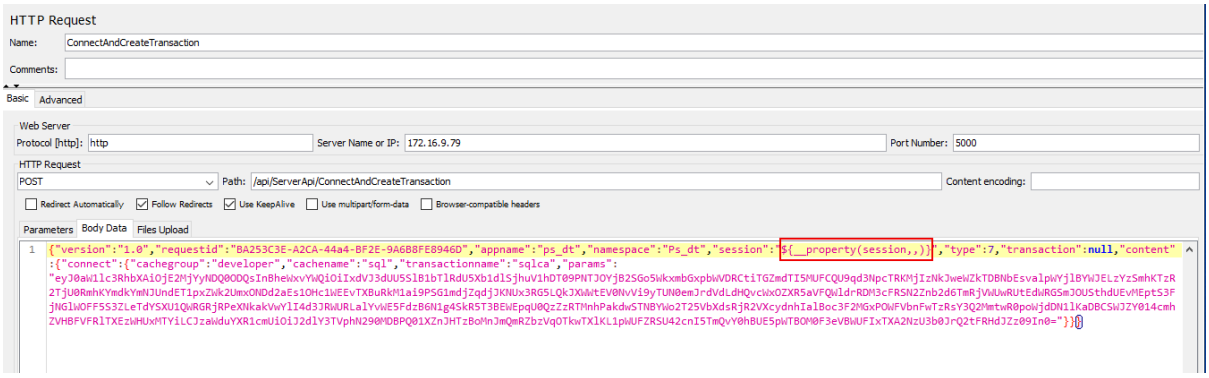
  ```
  ${__setProperty(session,${session},)}
  ```

The "`session`" variable becomes a JMeter global property.



Use the **Search** menu to search for all occurrences of "session" in all thread groups, and replace the static value of session ID with the global property `${__property(session,,)}`.

For example,



9.2.4.4 Parameterizing the transaction ID

Similar to the session ID, you can use the following to parameterize the transaction ID:

- a **Regular Expression Extractor** that saves the transaction ID to a local variable
- a **BeanShell Sampler** that calls the “setProperty” function to set the local variable as a global property, so that it can be shared in all thread groups

In the case of PowerServer, the following requests will update/change the transaction ID and contain the new ID in the response body:

1. Connect
2. DisConnect
3. Commit
4. Rollback

When Commit or Rollback failed, the response body will still contain the old transaction ID, which means the old transaction ID is still valid.

9.2.4.4.1 In single transaction

In the case of PowerServer, the **Connect**, **DisConnect**, **Commit**, and **Rollback** request will update/change the transaction ID, therefore, you need to add a **Regular Expression Extractor** to each of these requests to get and save the transaction ID into a variable.

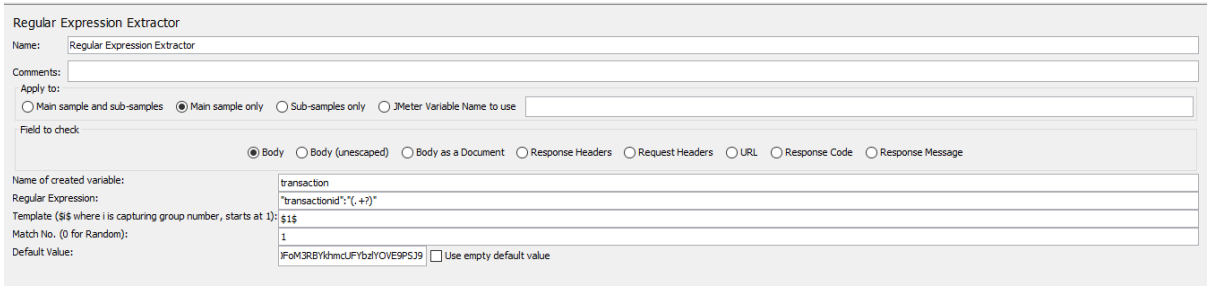
Let’s take the **ConnectAndCreateTransaction** request as an example.

To add a **Regular Expression Extractor**, right click on the **ConnectAndCreateTransaction** request and then select **Add** > **Post Processors** > **Regular Expression Extractor**.

Specify the **Regular Expression Extractor** like this:

- **Name of created variable**: “transaction” or any other name you prefer
- **Regular Expression**: "transactionid":"(.+?)"
- **Template**: 1
- **Match No.**: 1

The transaction ID will be saved to the “`transaction`” variable. You can invoke the local variable by typing `${transaction}` in the requests (bodies and headers).



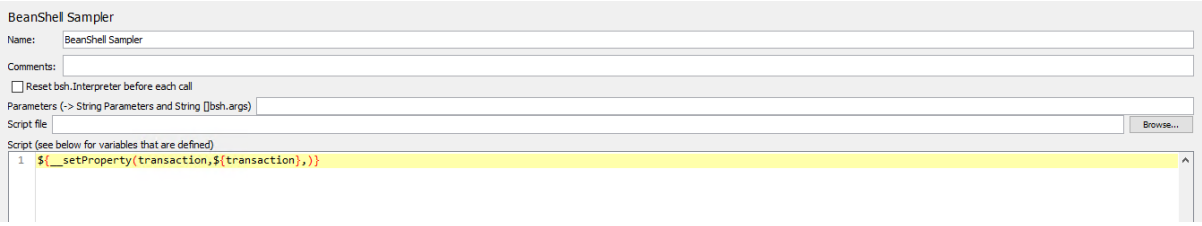
If you want to make the "`transaction`" variable a global property that can be accessed by all threads and thread groups, you can add a **BeanShell Sampler** or **BeanShell PostProcessor** to call the JMeter "setProperty" function. The "setProperty" function will set the "transaction" variable as a global property. Section 9.2.3.3, "Parameterizing the Retrieve test" has instructions for how to add a **BeanShell PostProcessor**. This section will show how to add a **BeanShell Sampler**.

To add a **BeanShell Sampler**, right click on setUp Thread Group, and then select **Add** > **Sampler** > **BeanShell Sampler**.

- Use the default name or input any name you prefer.
- Input the following script:

  ```
  ${__setProperty(transaction,${transaction},)}
  ```

The "`transaction`" variable becomes a JMeter global property.



Use the **Search** menu to search for all occurrences of "transactionid" in all thread groups, and replace the static value of transaction ID with `${__property(transaction,,)}`.

9.2.4.4.2 In multiple transactions

If your application uses multiple transactions, then each transaction will have its unique transaction ID. The transactions can be differentiated by their transaction names, and their transaction IDs shall be assigned with different variables, so that each variable will correlate with its own transaction.

Suppose your application has two transactions: SQLCA, and lstr_trans1. You will need to define two variables to store the ID of each transaction.

For transaction name "SQLCA"

- Add a **Regular Expression Extractor** to store the ID of the "SQLCA" transaction to a variable. Suppose the variable name is "`transaction`".
- Add a **BeanShell Sampler** to call the JMeter "setProperty" function to set the variable as a global property: `${__setProperty(transaction,${transaction},)}`

For transaction name "lstr_trans1"

- Add a **Regular Expression Extractor** to store the ID of the "lstr_trans1" transaction to a variable. Suppose the variable name is "`trans`".
- Add a **BeanShell Sampler** to call the JMeter "setProperty" function to set the variable as a global property: `${__setProperty(trans,${trans},)}`

Make sure to replace the transaction ID with the appropriate variable according to the transaction name. (You can use the **Search** menu to search for the transaction name and then replace its ID with the corresponding variable.)

After execution, you can view the **View Results Tree** to double check the transaction ID.

For example, in the **Connect** and **Disconnect** request pair, the transaction ID should be the same.

9.2.4.5 Parameterizing the retrieval argument

There are many ways for JMeter to parameterize the script. In this section, you will learn how to parameterize the retrieval argument using a CSV file.

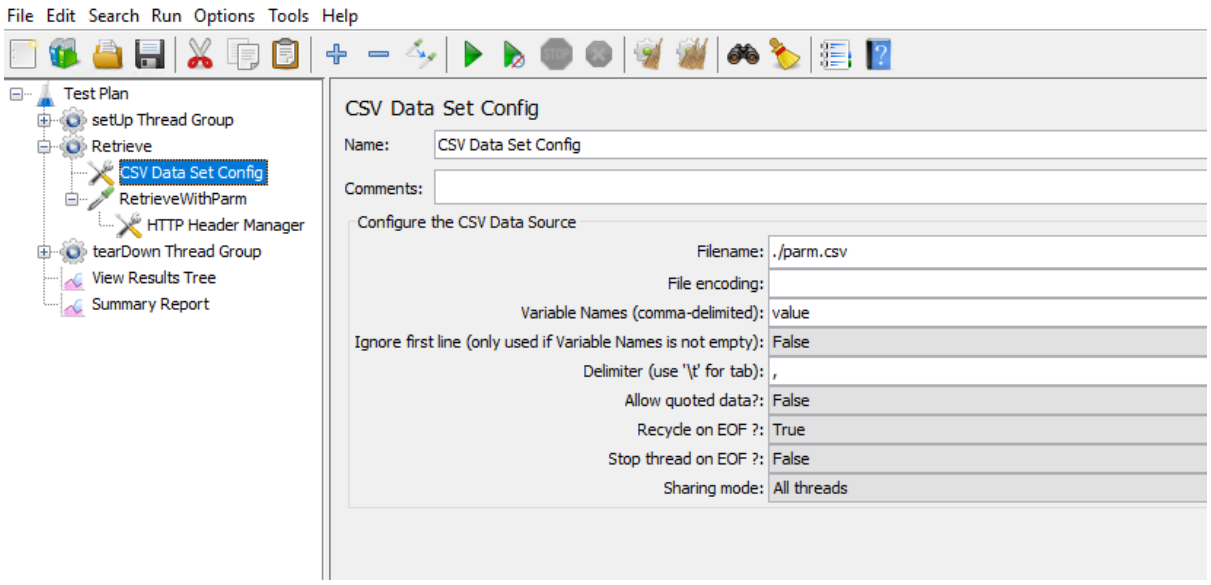
In this section, you will first need to prepare a CSV file that contains test data for the retrieval argument. Suppose there is only one retrieval argument (so there will be only one column in the CSV file).

Then, you will need to add a CSV Data Set Config element to read the data value from the CSV file:

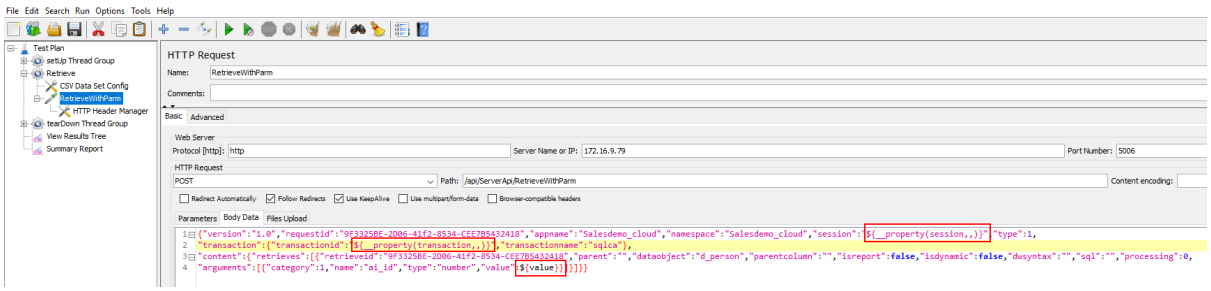
To add a CSV Data Set Config element, right click on Thread Group, and then select **Add** > **Config Element** > **CSV Data Set Config**.

Specify the CSV Data Set Config like this:

- **Filename**: File name and path of the CSV file (if the file is in the bin folder, then enter the filename, or use the full path of the file).
- **Variable name**: “value” or input any name you prefer (if there are multiple columns in the CSV file, define multiple variables and separate them with commas “,”)



Now, you can replace the initial value by typing `${value}` in the request.



9.2.4.6 Parameterizing the ESQL parameter

Another common way to parameterize the script is to use the user defined variables and user parameters.

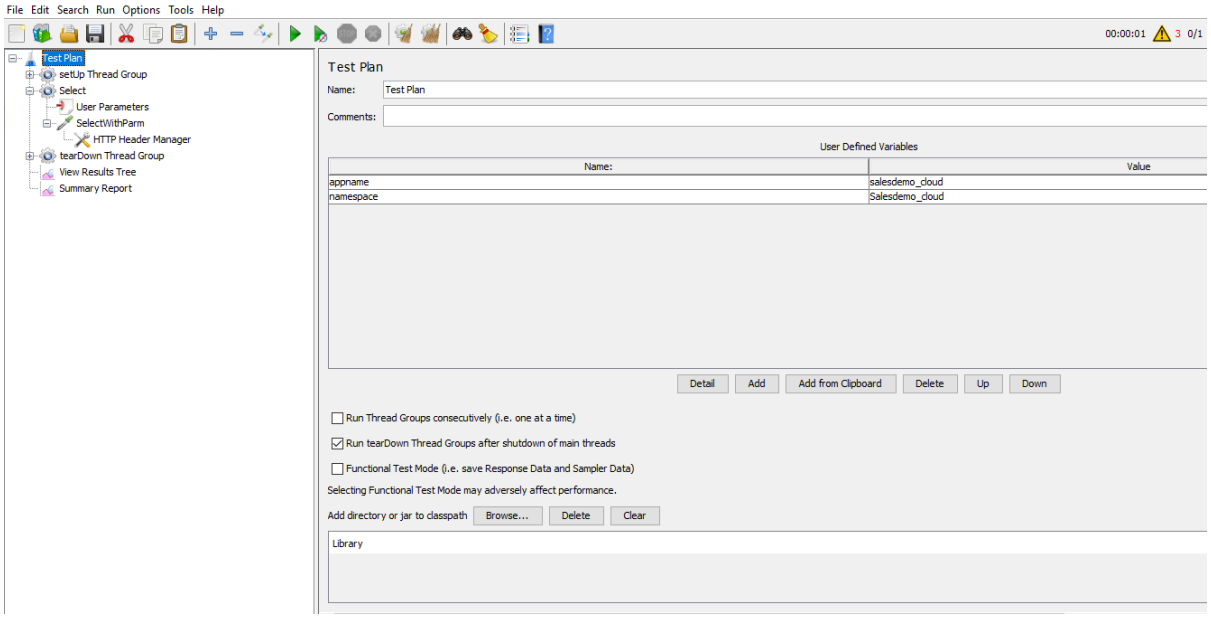
Unlike CSV Data Set Config which can access an external file, user defined variables and user parameters are used when you have less number of test data, because you need to manually insert the test data.

In user defined variables, only one value can be defined for a variable; in user parameters multiple values can be defined for a variable.

In this section, you will learn how to define the user defined variables and user parameters.

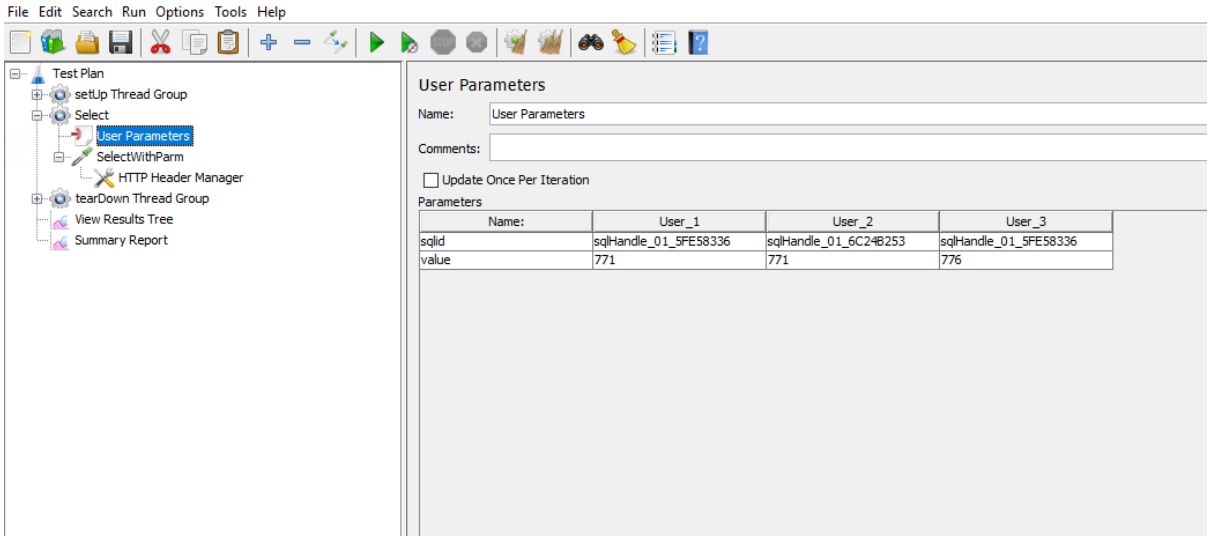
You can define variables in the User Define Variables for Test Plan (then the scope is global) or Thread Group or Sampler (then the scope is local).

For example, select Test Plan, and then input the name and value of the variable.



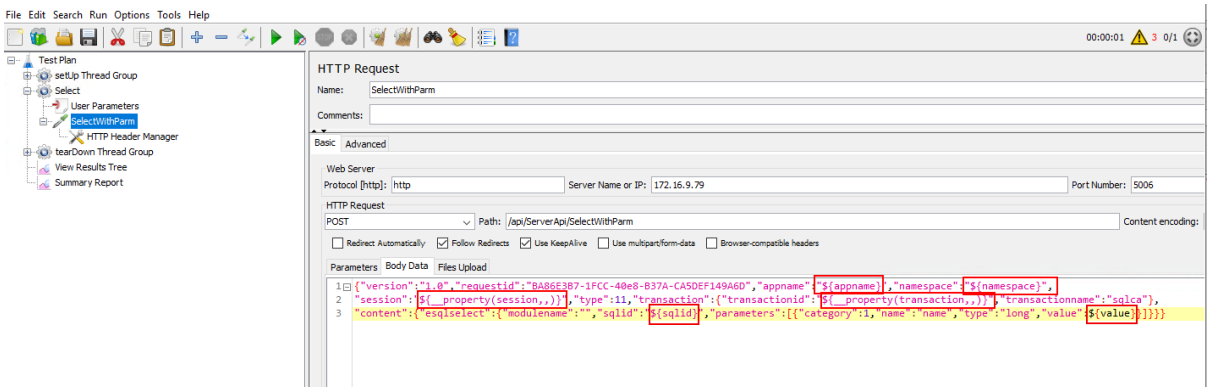
To add the User Parameters, right click on the Thread Group, and then select **Add** > **Pre Processors** > **User Parameters**.

Suppose you have set the number of users to 3. You can define different test data for the 3 users like this.



Now you can replace the initial value with the user defined variables and user parameters in the request.

Take the Select request as an example.


File Edit Search Run Options Tools Help
00:00:01 3 0/1
Test Plan
setUp Thread Group
Select
User Parameters
SelectWithParm
HTTP Header Manager
tearDown Thread Group
View Results Tree
Summary Report
HTTP Request
Name: SelectWithParm
Comments:
Basic Advanced
Web Server
Protocol [http]: http
Server Name or IP: 172.16.9.79
Port Number: 5006
HTTP Request
POST
Path: /api/ServerApi/SelectWithParm
Content encoding:
Redirect Automatically
Follow Redirects
Use KeepAlive
Use multipart/form-data
Browser-compatible headers
Parameters Body Data Files Upload
1 {"version":"1.0","requestid":"BA86E3B7-1FCC-40e8-B37A-CA5DEF149A6D","appname":"${appname}","namespace":"${namespace}",
2 "session":${__property(session,,)},"type":11,"transaction":{"transactionid":${__property(transaction,,)},"transactionname":"sqlca"},
3 "content":{"esqlselect":{"modulename":"","sqlid":${sqlid},"parameters":[{"category":1,"name":"name","type":"long","value":${value}}]}}}

10 Tutorial 10: Setting up a Web server

10.1 Overview

You can choose one of the following Web servers to host the client-side of the installable cloud app:

- Windows IIS
- Windows/Linux Apache
- Windows/Linux Nginx

This tutorial provides detailed instructions on how to set up a Web server for this purpose.

10.2 Setting up IIS

10.2.1 Preparations

In this tutorial, we will set up a Web server and an FTP server running on the same IIS instance.

Step 1: Set up the Web server with the following OS and software:

- Windows Server 2019 (64-bit)
- Microsoft IIS

  The next section Installing Web Server (IIS) has detailed installation instructions.

Step 2: Configure Windows Defender Firewall on the Web server to allow the FTP port (21 in this tutorial). The section "Configuring Windows Defender Firewall" has detailed instructions.

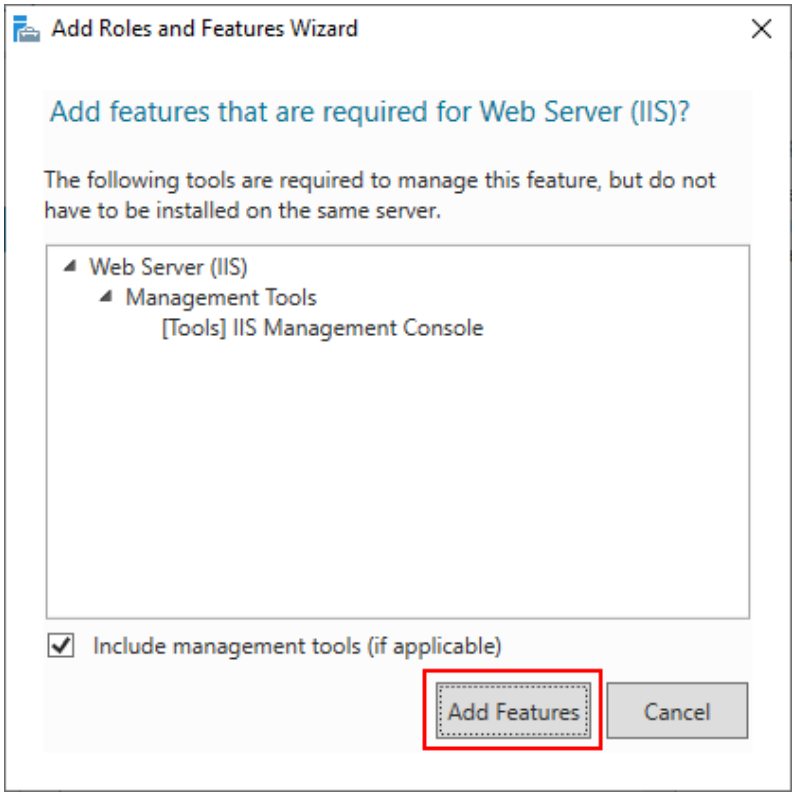
10.2.2 Installing Web Server (IIS)

Step 1: In Windows Server 2019, open **Server Manager**, and then select **Add roles and features**.

Step 2: In the **Add Roles and Features Wizard**, click **Next** several times until the **Server Roles** section displays.

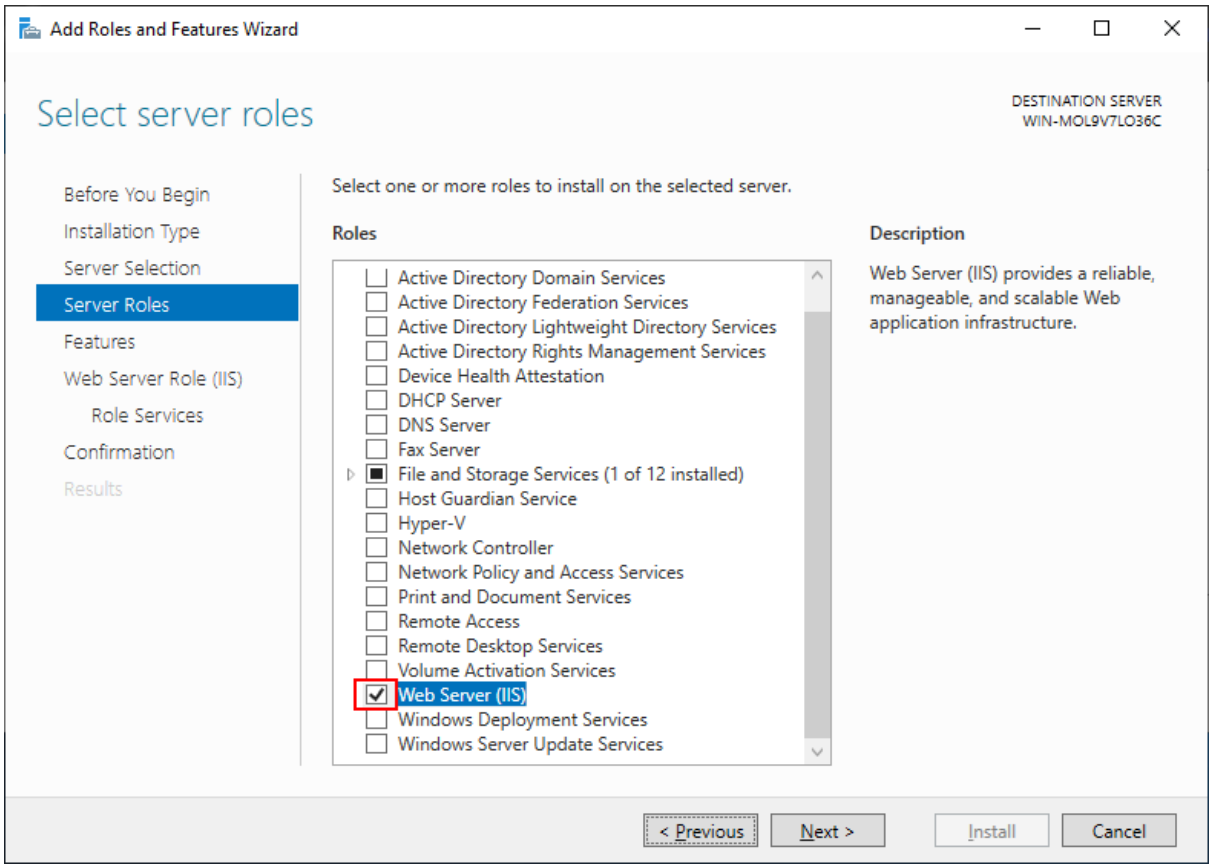
Step 3: Click the check box of **Web Server (IIS)**; and then click **Add Features** when asked whether to add features required for Web server.

Figure 10.1:



Step 4: Make sure the check box of **Web Server (IIS)** is selected.

Figure 10.2:

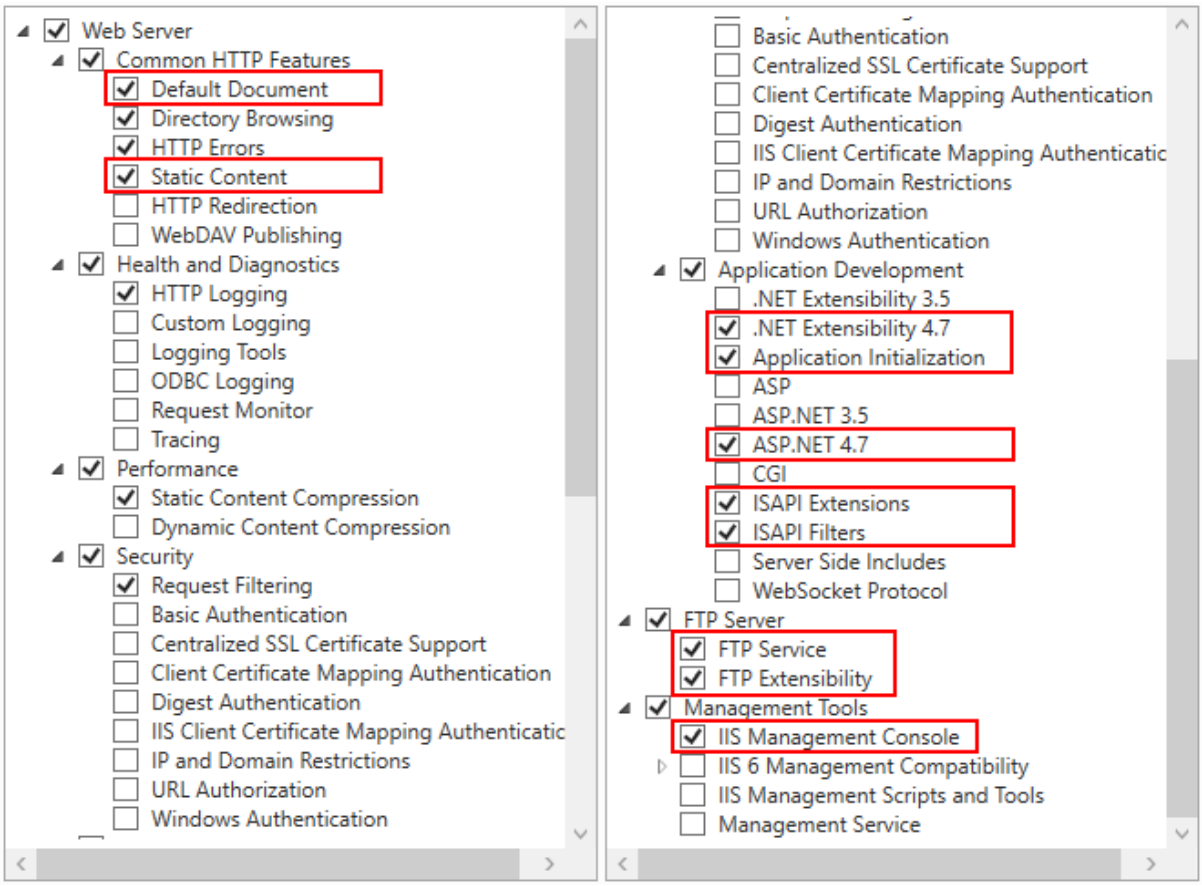


Step 5: Click **Next** until the **Role Services** section displays. Make sure the following role services are selected.

- Default Document
- Static Content
- .NET Extensibility 4.7
- Application Initialization
- ASP.NET 4.7
- ISAPI Extensions
- ISAPI Filters
- IIS Management Console
- **FTP Service**
- **FTP Extensibility**

  FTP Service & **FTP Extensibility** must be enabled if you want to create an IIS FTP site for transferring files from a remote development machine to the Web server.

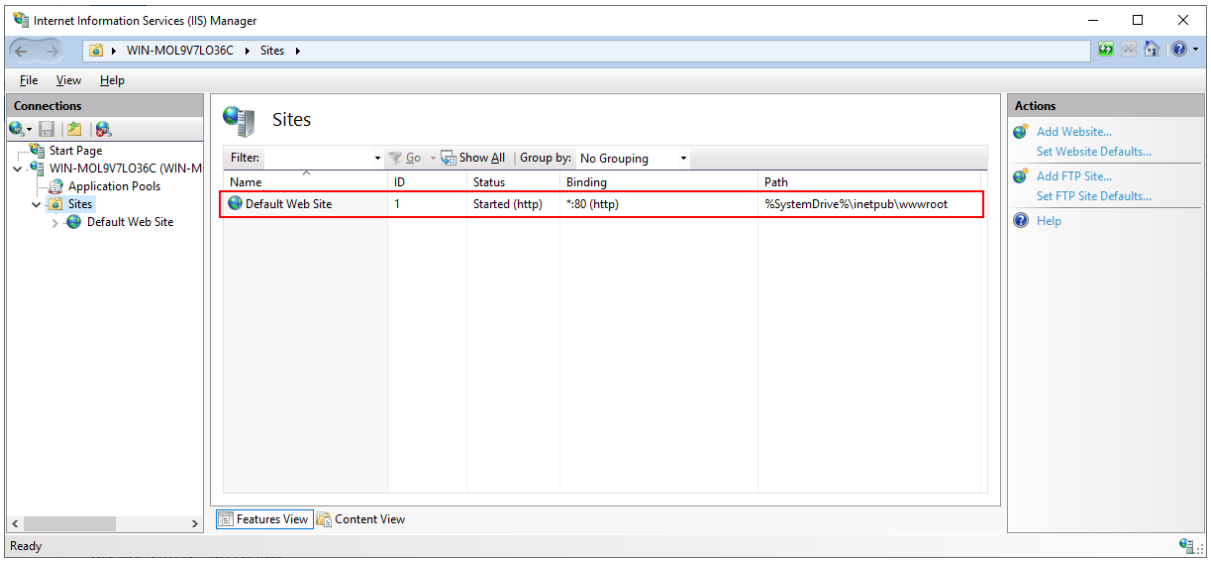
Figure 10.3:



Step 6: Click **Next** and then click **Install**.

After IIS is installed, a **Default Web Site** (with port 80) is automatically created (you could also create new websites with different port numbers).

Figure 10.4:



Step 7: Open a Web browser and run the following URLs to access the **Default Web Site**.

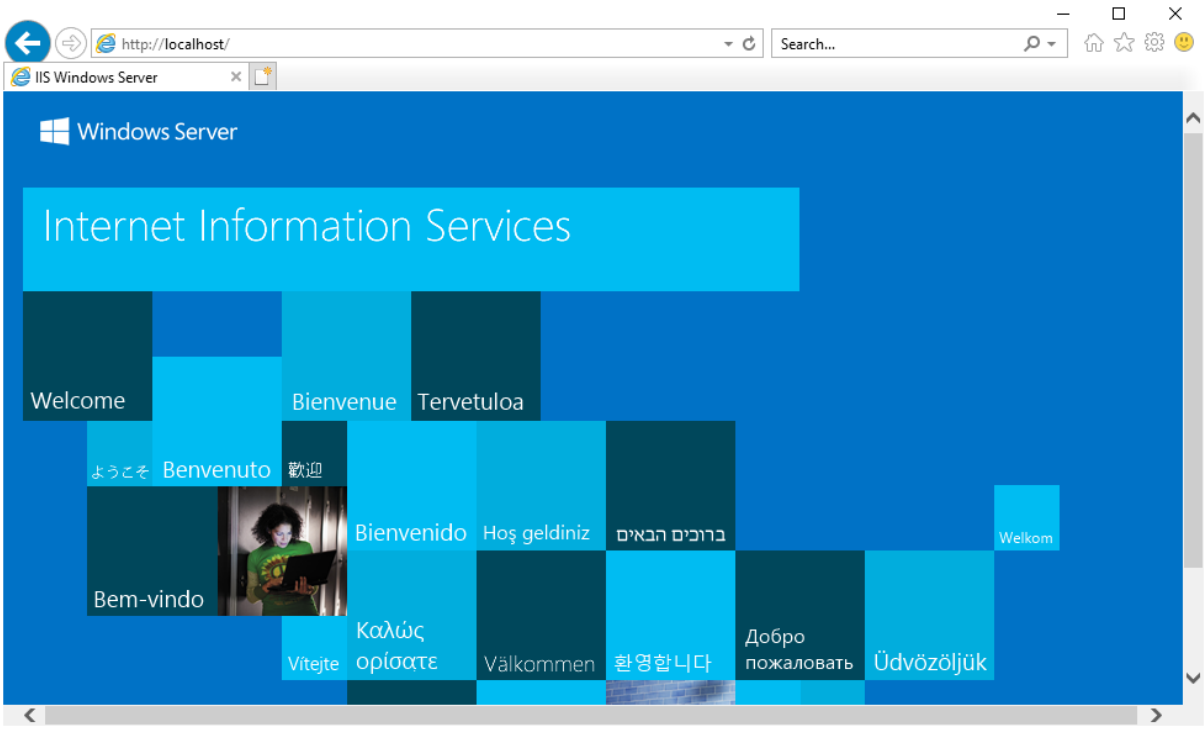
http://localhost:80/

http://*your_server_ip*:80/

TIP: You can use "localhost" or the IP address to access the IIS website on the local computer. To obtain the IP address, open a command prompt window and then type ipconfig<Enter>. Write down the IP address as it is needed when you configure the Web server profile in PowerBuilder.

If the IIS welcome screen displays, the IIS website is working properly.

Figure 10.5:



Also remember the physical path for Default Web Site which is **C:\inetpub\wwwroot** by default (or any other path you have changed to). This is where the client app will be deployed, or the FTP site will point to.

10.2.3 Configuring SSL on IIS

It is highly recommended that you configure Secure Sockets Layer (SSL) for the Web server, so that HTTPS can be used to secure the connections between the client and the Web server.

For how to configure SSL on IIS, refer to https://docs.microsoft.com/en-us/iis/manage/configuring-security/how-to-set-up-ssl-on-iis.

10.2.4 Creating an IIS FTP site

Note

To deploy the client app from the development PC to the remote Web server, you can choose:

- Method 1: Deploy the client app to the remote server through the FTP protocol. This requires that

  1) An FTP server is set up on the Web server (the FTP server's physical path must point to the Web root of the Web server).

  This section will walk you through how to set up an FTP server on the Web server.

  2) The client app is deployed to the remote Web server through the FTP server.

  Tutorial 1 > "Task 4: Setting up the development PC" has detailed instructions.

- Method 2: Package the client app and then install (or copy) it to the remote Web server.

  Follow the instructions in Packaging and copying the client app to package the client app and then install (or copy) it to the Web server Web root.

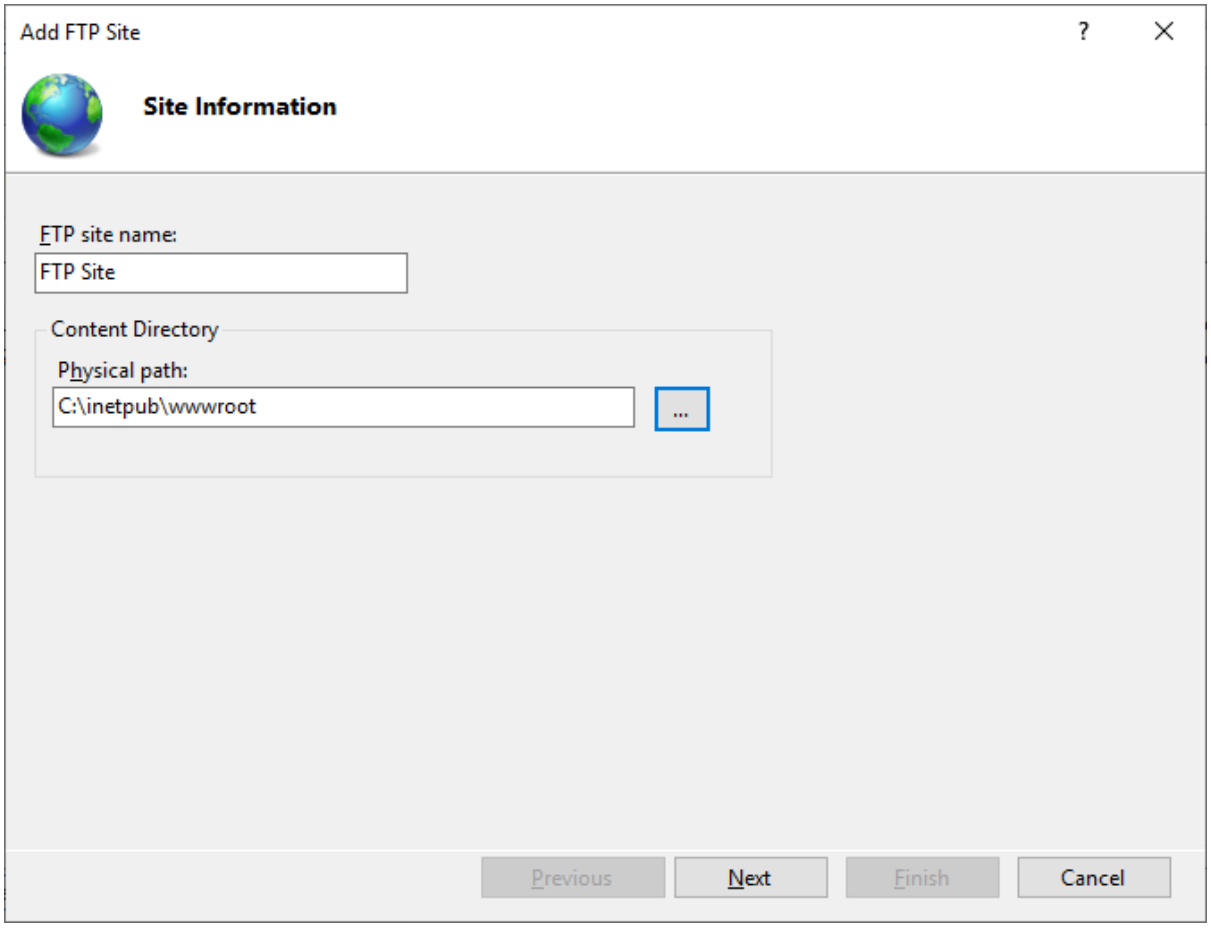
The following steps will walk you through setting up an IIS FTP site on the Web server, so that PowerBuilder can deploy files to the remote Web server through the FTP protocol.

In the previous section, if you have selected to enable **FTP Service** & **FTP Extensibility**, you can create an IIS FTP site to be used by the remote deployment.

Step 1: In the IIS Manager, right click **Sites**, select **Add FTP Site**.

Step 2: Specify a name for the FTP site, and set the physical path to the Web root of the IIS Web server (**C:\inetpub\wwwroot** in this tutorial). Click **Next**.

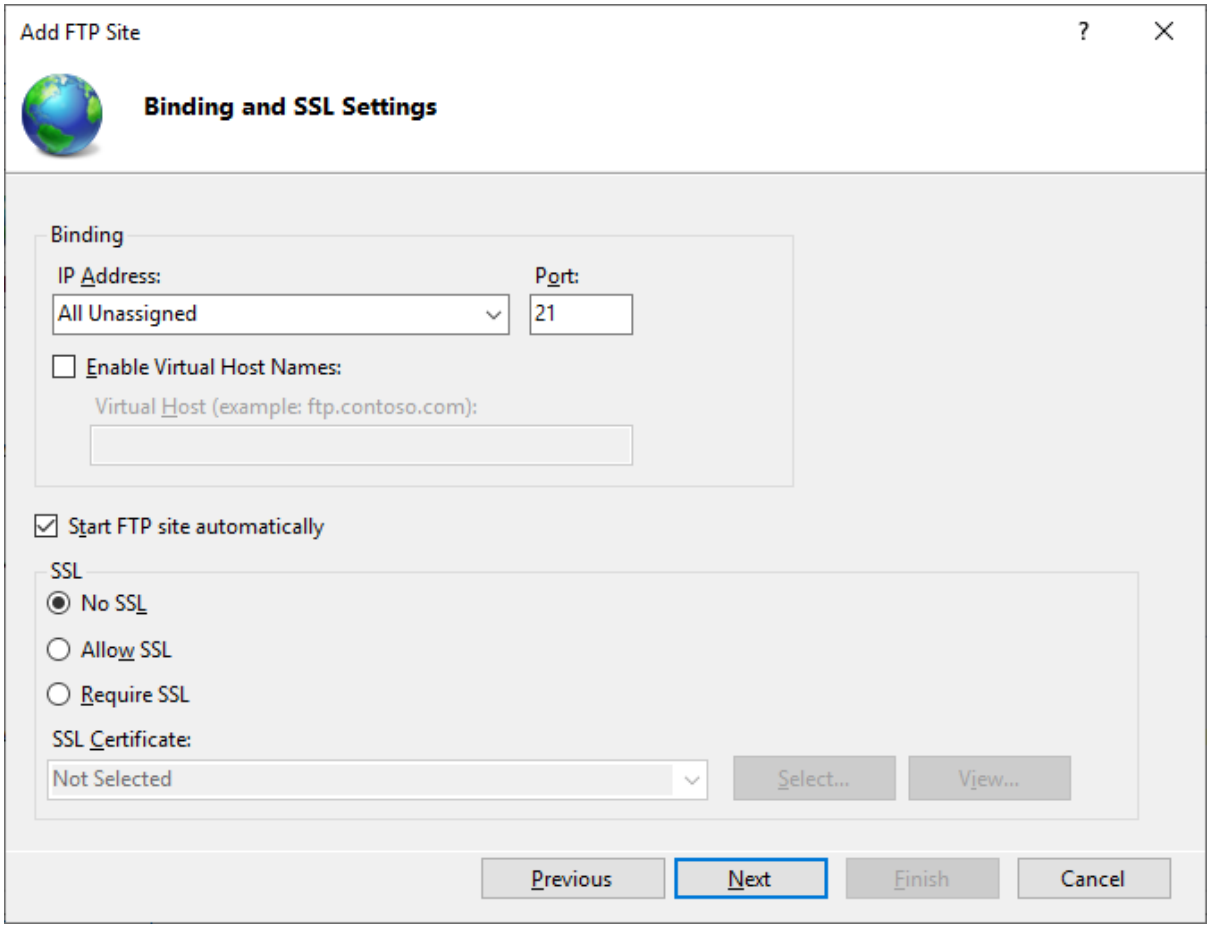
Figure 10.6:



Step 3: Use the default port 21 (or specify a different port if you like). If no certificate is available, you can select **No SSL**. Use the default values for the other settings. Click **Next**.

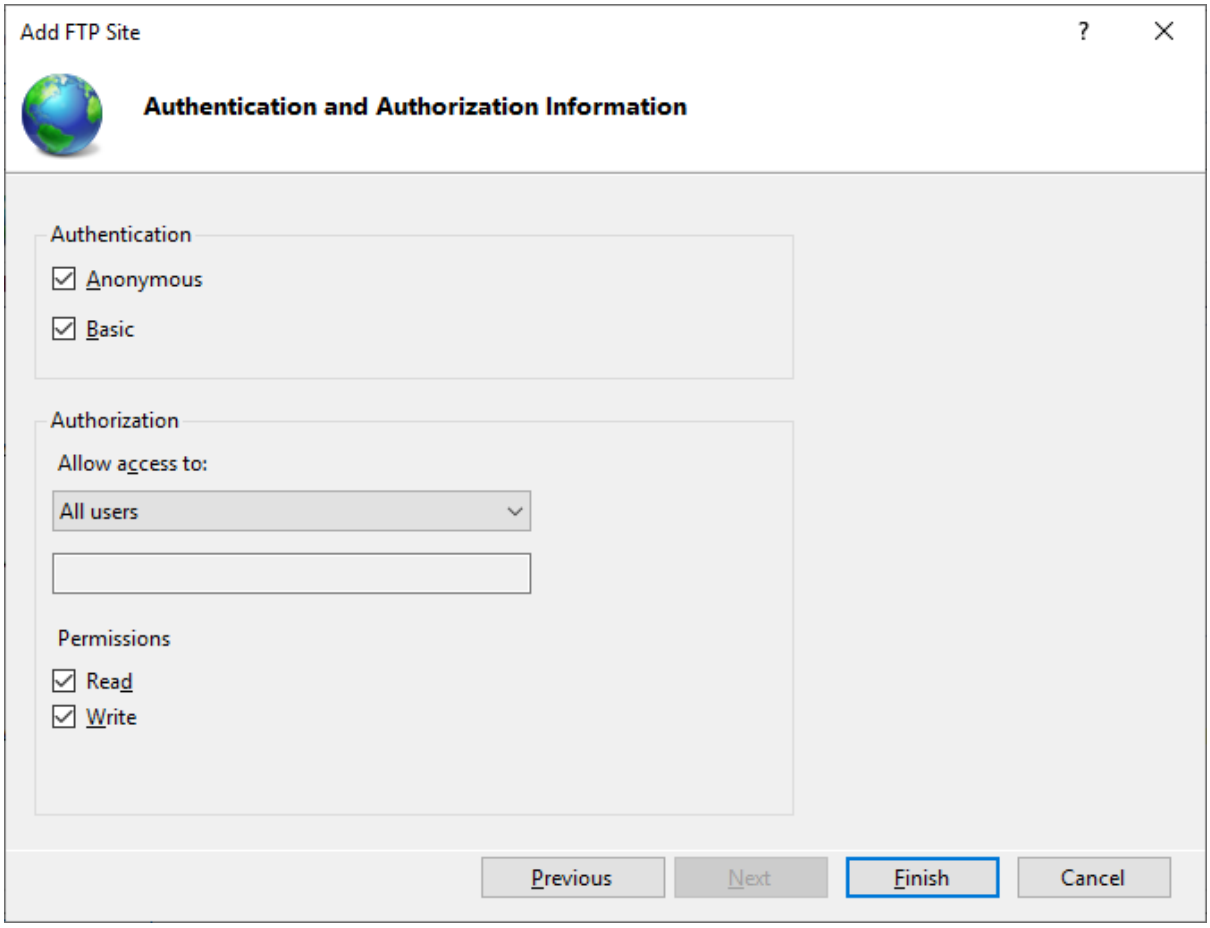
For how to configure SSL on an IIS FTP site, refer to Configure an SSL-based FTP server.

Figure 10.7:



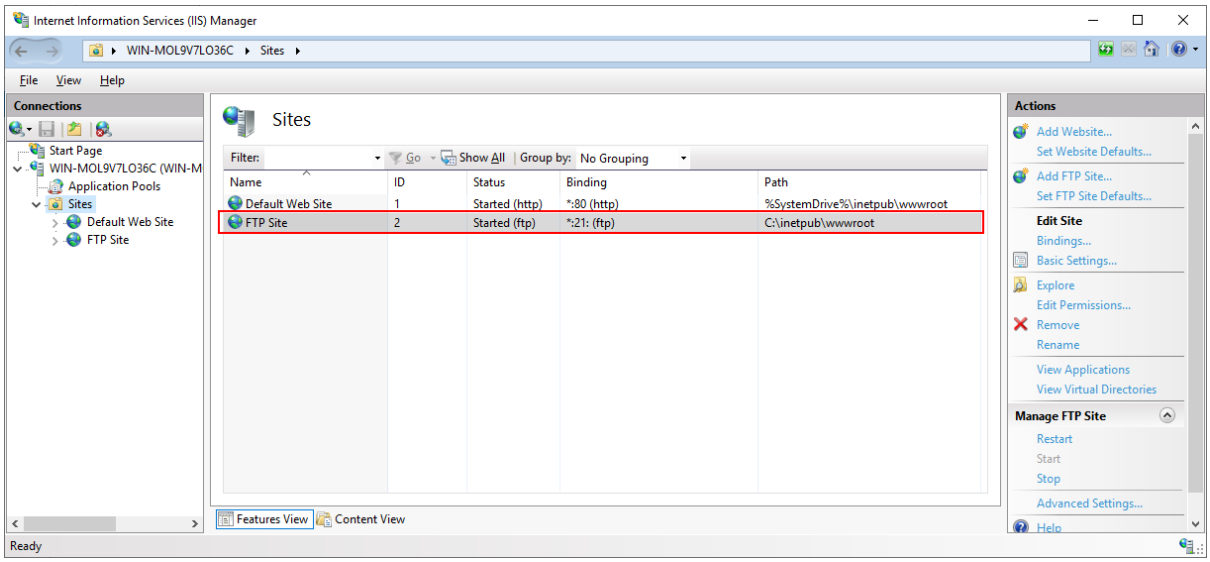
Step 4: Select **Anonymous** and **Basic** authentication. Select **All users** or specify the users that are allowed to access the FTP site, and then select the **Read** and **Write** permissions. Click **Finish**.

Figure 10.8:



The FTP site is created.

Figure 10.9:



Step 5: Open a Web browser and run the following URL to access the FTP site.

ftp://*your_server_ip*:21/

If the FTP root displays, then the FTP site is working properly.

Figure 10.10:

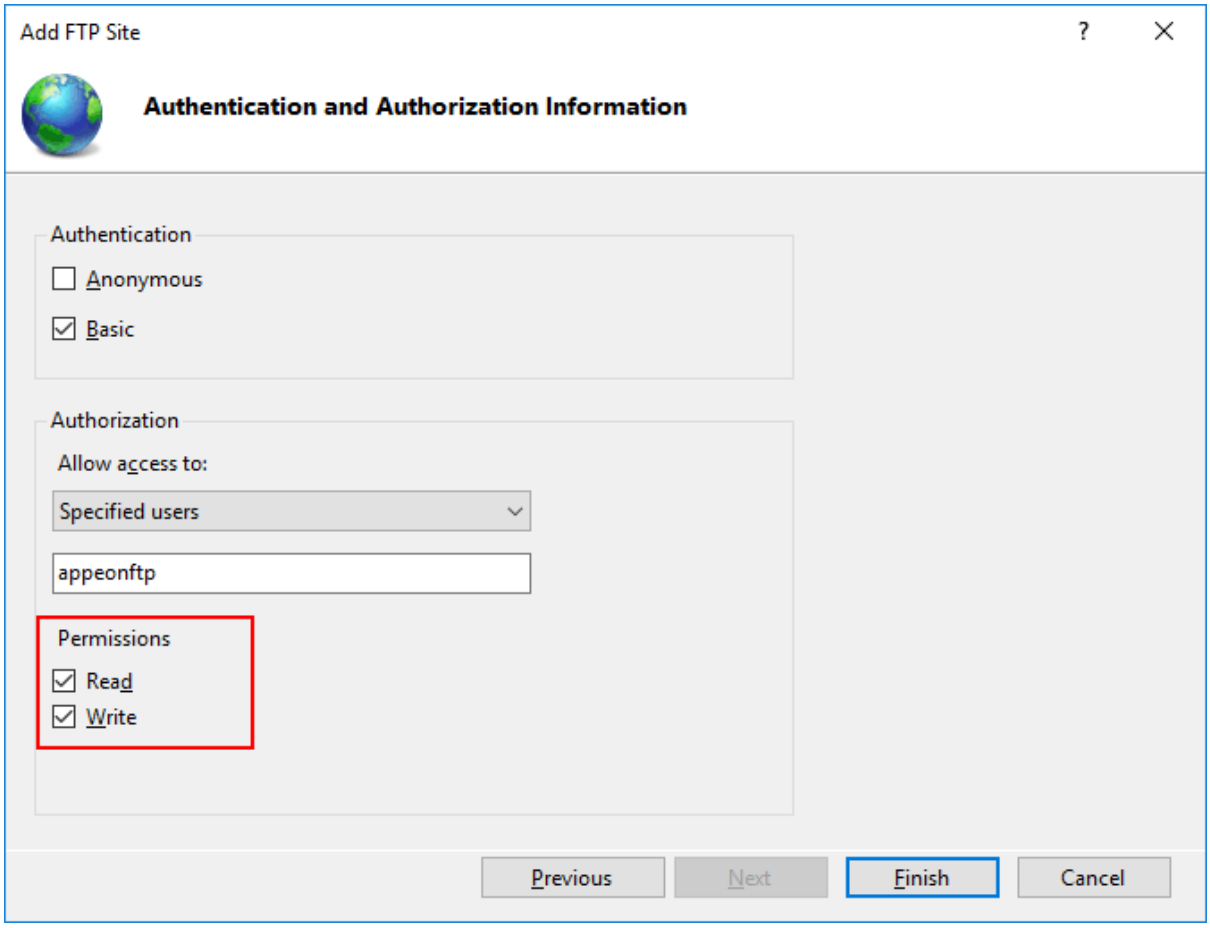
10.2.5 Configuring SSL on FTP server

To configure the FTP server with Secure Sockets Layer (SSL), you can follow instructions in https://docs.microsoft.com/en-us/iis/publish/using-the-ftp-service/using-ftp-over-ssl-in-iis-7.

The following highlights the important settings for configuring SSL on an FTP site:

- The **Physical path** must be the full path to the Web server Web root.
- The FTP site must be set to **Require SSL** or **Allow SSL**.
- An SSL certificate must be selected.
- The **Read** and **Write** permissions must be enabled.

Figure 10.11: FTP site properties



When you configure the Web Server profile in PowerBuilder that connects with an SSL Web server, you should input the HTTPS listener and port number for the Web server.

10.3 Setting up Apache on Windows

10.3.1 Preparations

In this tutorial, we will set up a Web server running on Apache HTTP Server on Windows.

Step 1: Set up the Web server with the following OS and software:

- Windows Server 2019 (64-bit)
- Visual C++ Redistributable
- Apache HTTP Server 2.4.47

  The next section Installing Apache HTTP Server has detailed installation instructions.

Step 2: Configure Windows Defender Firewall on the Web server to allow the port (the HTTP port is 80 and the FTP port is 21 in this tutorial) to go through. The section "Configuring Windows Defender Firewall" has detailed instructions.

10.3.2 Installing Apache HTTP Server

Step 1: Select a binary package provider for Apache for Windows from https://httpd.apache.org/docs/current/platform/windows.html#down.

Step 2: In this tutorial, select **Apache Lounge**, and then download the following packages from https://www.apachelounge.com/download/.

- Visual C++ Redistributable for Visual Studio 2015 - 2019: https://aka.ms/vs/16/release/VC_redist.x64.exe
- Apache 2.4.47 Win64: https://www.apachelounge.com/download/VS16/binaries/httpd-2.4.47-win64-VS16.zip

Step 3: Double click **VC_redist.x64.exe** to install the Visual C++ Redistributable first.

Step 4: Unzip the **httpd-2.4.47-win64-VS16.zip** file and place the **Apache24** folder under the C drive ("C:\Apache24" is the default ServerRoot in **conf\httpd.conf**; and the default folder for web files is DocumentRoot "C:\Apache24\htdocs"). If you place the **Apache24** folder to another location, change the following setting accordingly.

```
Define SRVROOT "c:/Apache24"
```

> **Note**
>
> Paths in **httpd.conf** and other configuration files must be specified using forward slashes ("/") instead of back slashes ("\").

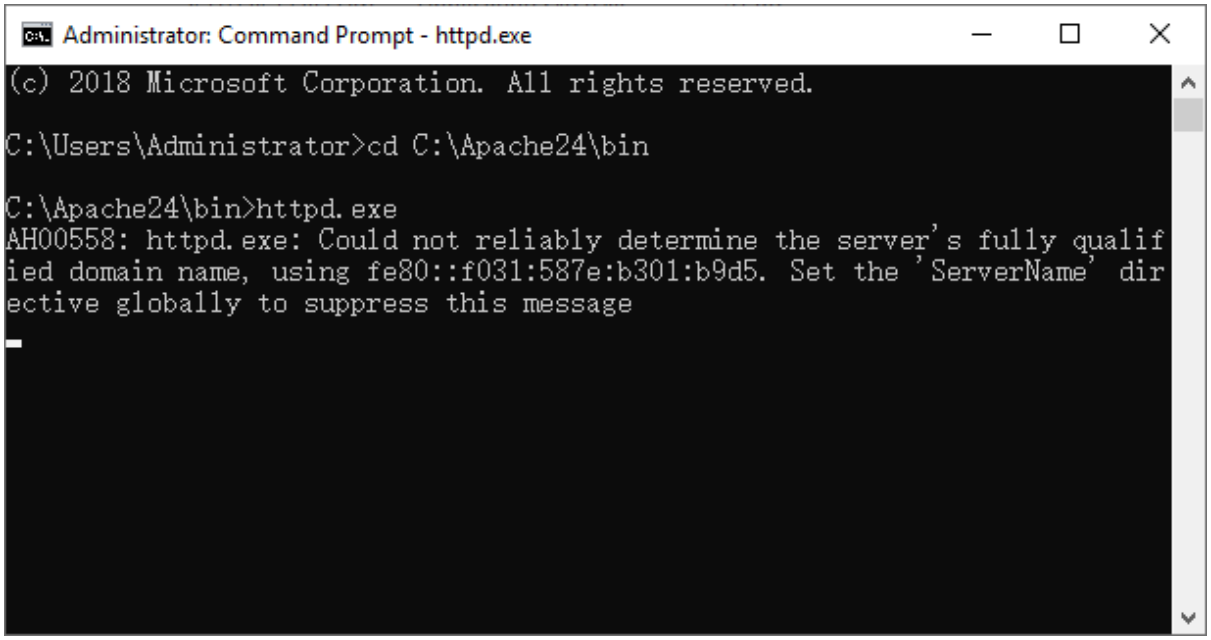
You could also change the IP address, port number, server name etc. in **httpd.conf** rather than using the default values.

Tip: In Windows, you can execute the command "netstat -ano | findstr *portnumber*" to check if the port number is occupied by any other program.

Step 5: Open the command prompt window, go to the C:\Apache24\bin folder, and run the Apache HTTP server.

```
cd C:\Apache24\bin
httpd.exe
```

Figure 10.12:

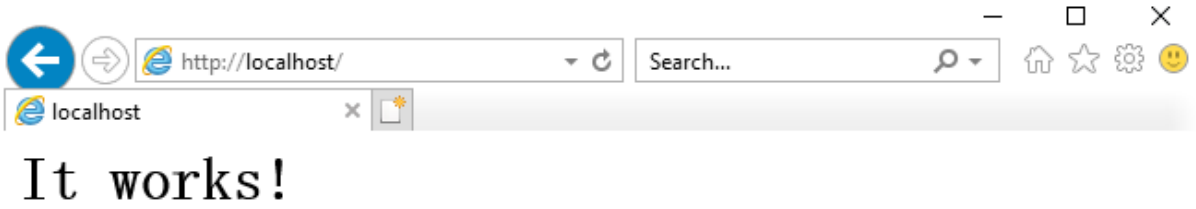


Step 6: Test the Apache HTTP server by opening up a Web browser and typing in the address: http://localhost.

The following message indicates the Apache HTTP server is working properly.

You can further test from the development PC by typing http://*your_server_ip* in a browser.

Figure 10.13:



The Using Apache HTTP Server on Microsoft Windows page has more detailed documentation about using Apache on Windows.

10.3.3 Configuring SSL on Apache

It is highly recommended that you configure Secure Sockets Layer (SSL) for the Web server, so that HTTPS can be used to secure the connections between the client and the Web server.

For how to configure SSL on Apache, refer to https://httpd.apache.org/docs/2.4/ssl/.

10.3.4 Installing FTP server

Note

To deploy the client app from the development PC to the remote Web server, you can choose:

- Method 1: Deploy the client app to the remote server through the FTP protocol. This requires that

  1) An FTP server is set up on the Web server (the FTP server's physical path must point to the Web root of the Web server).

  This section will walk you through how to set up an FTP server on the Web server.

  2) The client app is deployed to the remote Web server through the FTP server.

  Tutorial 1 > "Task 4: Setting up the development PC" has detailed instructions.

- Method 2: Package the client app and then install (or copy) it to the remote Web server.

  Follow the instructions in Packaging and copying the client app to package the client app and then install (or copy) it to the Web server Web root.

The following steps will walk you through setting up an FTP server on the Web server, so that PowerBuilder can deploy files to the remote server through the FTP protocol.

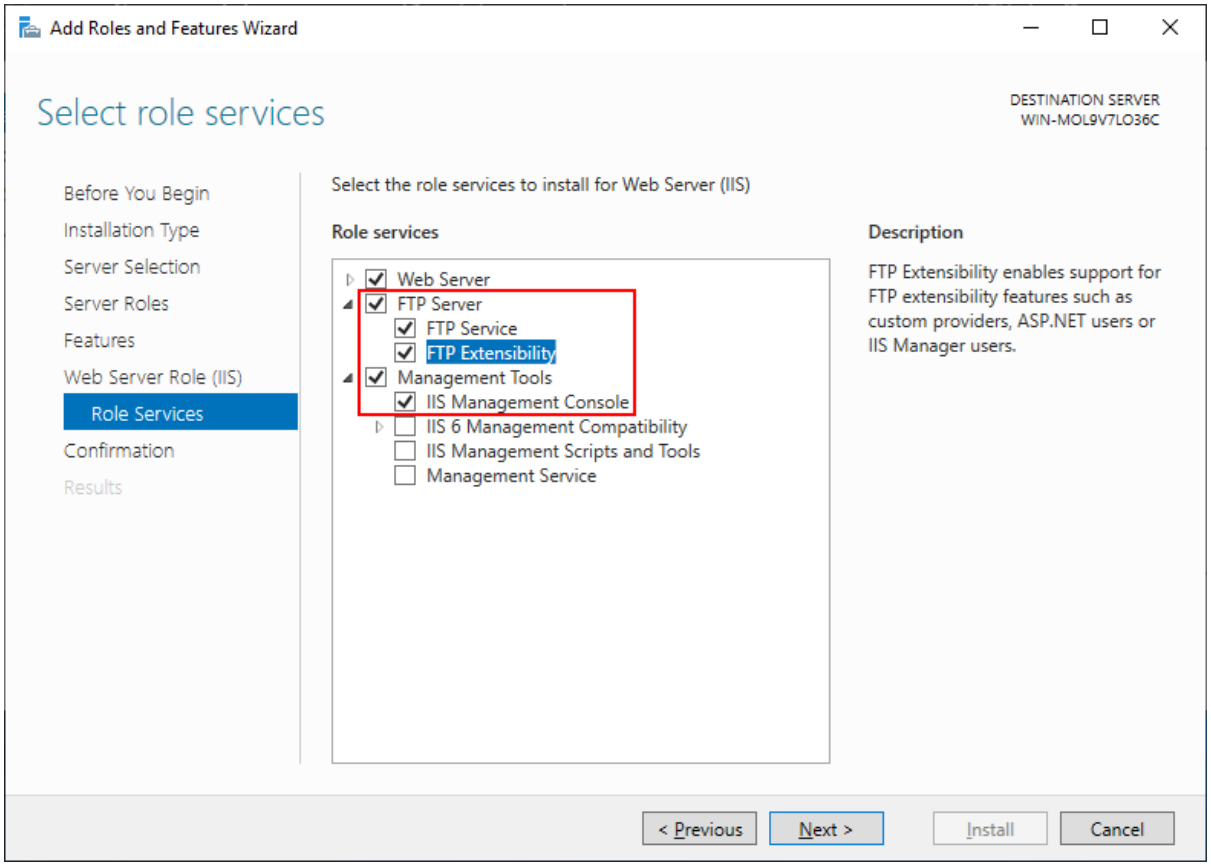
In this tutorial, we set up an IIS FTP server.

To enable the IIS FTP service and create an IIS FTP site,

Step 1: Follow the instructions in Installing Web Server (IIS) until the **Role Services** section displays; and make sure the following role services are selected and installed.

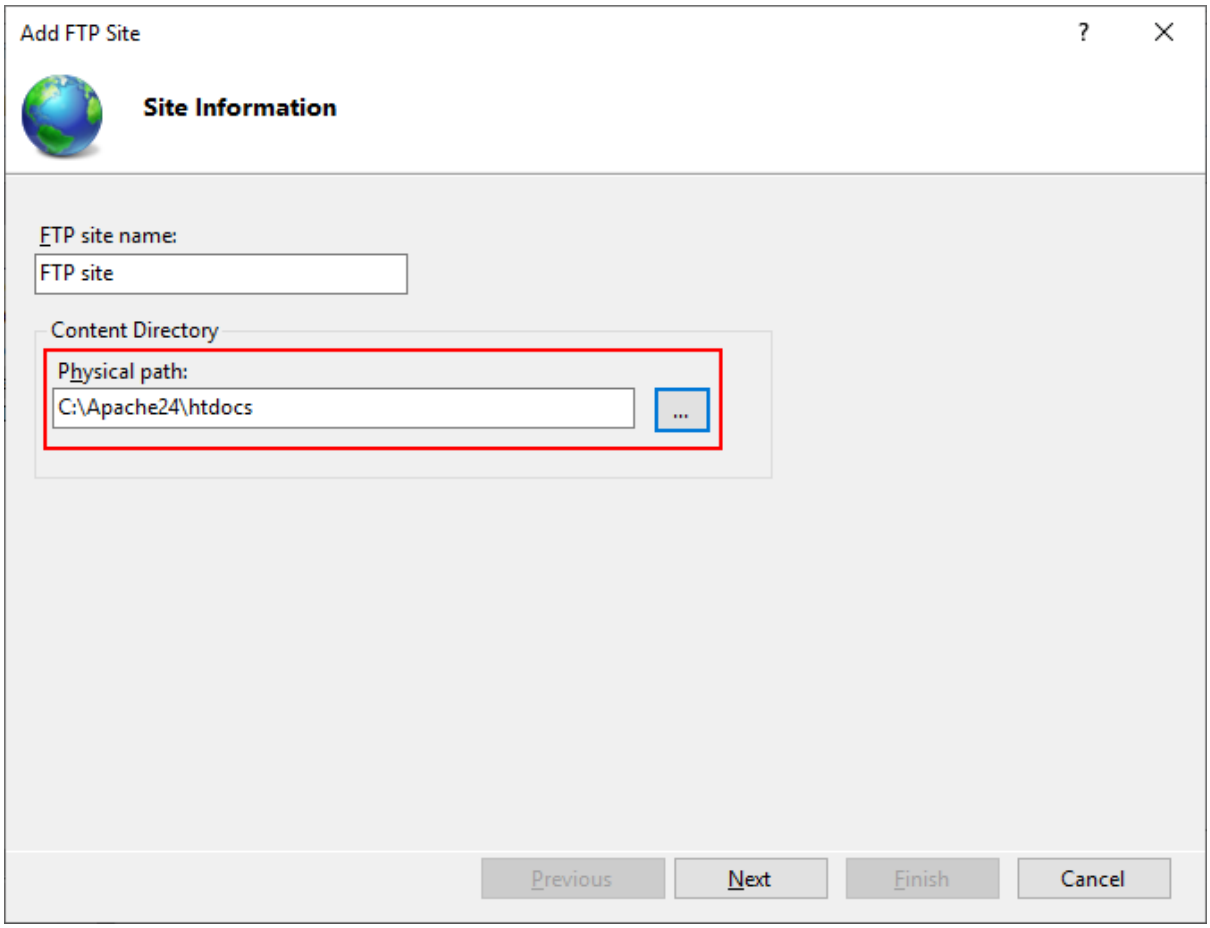
- FTP Server
  - FTP Service
  - FTP Extensibility
- Management Tools
  - IIS Management Console

Figure 10.14:



Step 2: Follow the instructions in Creating an IIS FTP site to create an FTP site and set the physical path to the document root of the Apache HTTP server which is C:\Apache24\htdocs by default.

Figure 10.15:

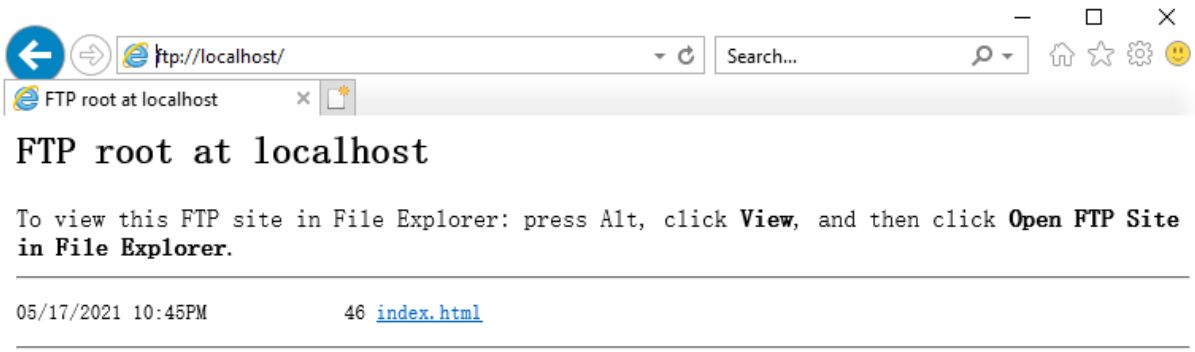


Step 3: Test the FTP site by opening up a Web browser and typing in the address: ftp://localhost.

The following message indicates the FTP site is working properly.

You can further test from the development PC by typing ftp://*your_server_ip* in a browser. (If access failed, check that if the firewall has blocked the FTP port; you can try to turn off the firewall on the server.)

Figure 10.16:



10.4 Setting up Apache on Linux

10.4.1 Preparations

In this tutorial, we will set up a Web server running on Apache HTTP Server on Linux.

Step 1: Set up a server with the following OS and software:

- CentOS 8 (64-bit)
- Apache HTTP Server

  The next section Installing Apache HTTP Server has detailed installation instructions.

Step 2: Configure the CentOS user account: you can either use the **root** account or create a new account with administrative privileges.

Step 3: Set up a firewall on the server and make sure the firewall allows the port (the HTTP port is 80 in this tutorial) to go through.

Step 4: Make sure the server can connect to Internet during the installation of Apache HTTP Server.

10.4.2 Installing Apache HTTP Server

Step 1: Download and install Apache HTTP Server from the CentOS's default software repositories. Make sure the machine can connect to Internet during the download and installation process.

```
$ sudo dnf install httpd
```

During the download and installation process, you might be prompted to enter the password for your user account, or enter y to confirm that you want to install Apache.

Step 2: Start Apache HTTP Server.

```
$ sudo systemctl start httpd
```

Step 3: Verify that the HTTP Server service is running.

```
$ sudo systemctl status httpd
```

Figure 10.17:

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset: d>
   Active: active (running) since Wed 2021-05-26 13:07:42 EDT; 50s ago
     Docs: man:httpd.service(8)
 Main PID: 124346 (httpd)
   Status: "Running, listening on: port 443, port 80"
    Tasks: 213 (limit: 11155)
   Memory: 24.2M
   CGroup: /system.slice/httpd.service
           ├─124346 /usr/sbin/httpd -DFOREGROUND
           ├─124351 /usr/sbin/httpd -DFOREGROUND
           ├─124352 /usr/sbin/httpd -DFOREGROUND
           ├─124353 /usr/sbin/httpd -DFOREGROUND
           └─124354 /usr/sbin/httpd -DFOREGROUND

May 26 13:07:41 localhost.localdomain systemd[1]: Starting The Apache HTTP Server...
May 26 13:07:42 localhost.localdomain httpd[124346]: AH00558: httpd: Could not relia>
May 26 13:07:42 localhost.localdomain systemd[1]: Started The Apache HTTP Server.
May 26 13:07:42 localhost.localdomain httpd[124346]: Server configured, listening on>
~
lines 1-19/19 (END)
```

Step 4: If you have set up a firewall on the server, run the following command to permanently enable HTTP service and port 80:

```
# sudo firewall-cmd --permanent --zone=public --add-service=http
```

```
# sudo firewall-cmd --permanent --zone=public --add-port=80/tcp
```

To apply the changes, reload the firewall service using the following command:

```
# sudo firewall-cmd --reload
```

To verify that the http service and port 80 were added successfully, you can run:

```
# sudo firewall-cmd --permanent --list-all
```

Figure 10.18:

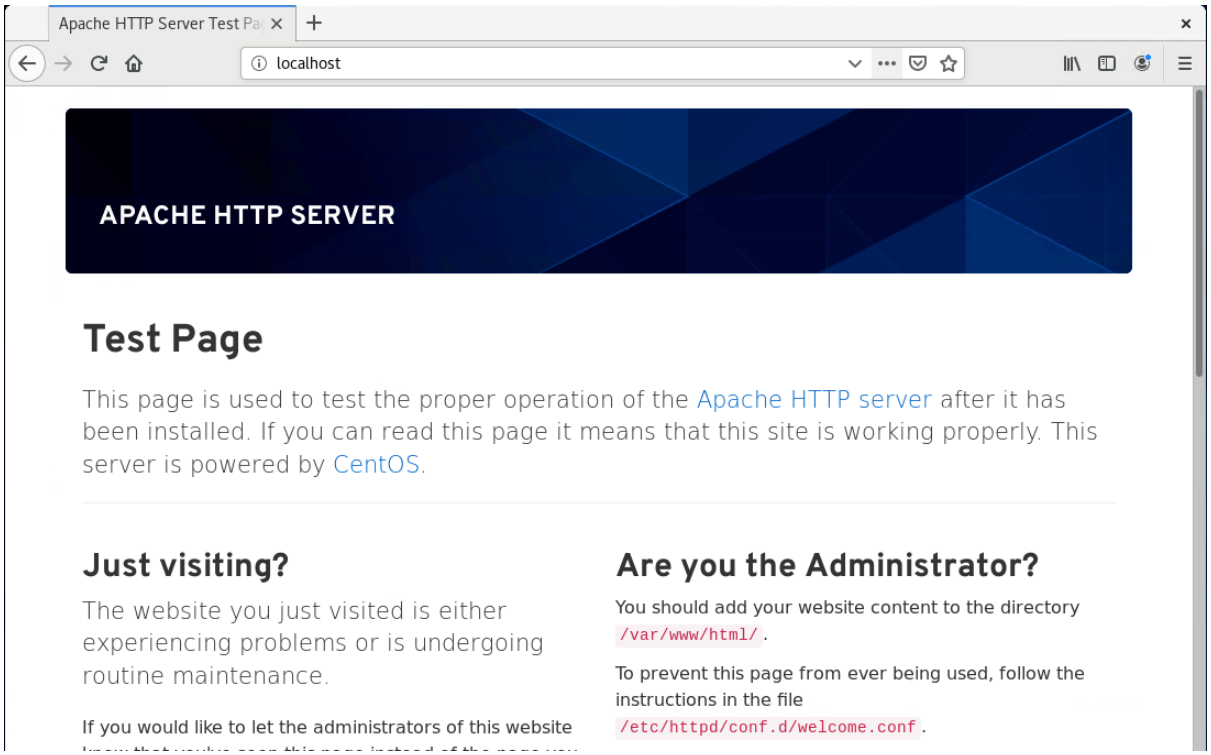
```
[appeon@localhost ~]$ sudo firewall-cmd --permanent --list-all
[sudo] password for appeon:
public
  target: default
  icmp-block-inversion: no
  interfaces:
  sources:
  services: cockpit dhcpv6-client http ssh
  ports: 80/tcp
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

Step 5: Test the Apache HTTP server by opening up a Web browser and typing in the address: http://localhost or http://*your_server_ip*.

The following page indicates the Apache HTTP server is installed and started successfully.

You can further test from the development PC by typing http://*your_server_ip* in a browser.

Figure 10.19:



10.4.3 Configuring SSL on Apache

It is highly recommended that you configure Secure Sockets Layer (SSL) for the Web server, so that HTTPS can be used to secure the connections between the client and the Web server.

For how to configure SSL on Apache, refer to https://httpd.apache.org/docs/2.4/ssl/.

10.4.4 Configuring Apache to be case-insensitive

As PowerBuilder is designed to be case-insensitive and always uses lower cases to access the deployed folders/files, therefore, in a case-sensitive file system like Linux, folder/file names (such as theme files, images etc.) containing upper cases may not be found or loaded. To avoid such issues, you should always use lower cases in folder/file names for your application, or add the following configuration to Apache in Linux to ignore the case:

1. Go to the /etc/httpd/conf folder, and open httpd.conf in a text editor.

2. Search "loadmodule" and add the following lines.

   Pay special attention to the words "**speling**_module" and "mod_**speling**" (not **spelling**).

```
LoadModule speling_module modules/mod_speling.so
CheckSpelling on
```

Figure 10.20:

```
#
# Dynamic Shared Object (DSO) Support
#
# To be able to use the functionality of a module which was built as a DSO you
# have to place corresponding `LoadModule' lines at this location so the
# directives contained in it are actually available _before_ they are used.
# Statically compiled modules (those listed by `httpd -l') do not need
# to be loaded here.
#
# Example:
# LoadModule foo_module modules/mod_foo.so
#
LoadModule speling_module modules/mod_speling.so
CheckSpelling on
Include conf.modules.d/*.conf
```

3. Check if any syntax error in httpd.conf.

```
$ sudo apachectl configtest
```

4. Restart Apache.

```
$ sudo systemctl restart httpd
```

If Apache failed to start, go to the /var/log/httpd folder and view the error_log.log and access_log.log files to read the detailed error information.

10.4.5 Packaging and copying the client app

Note

To deploy the client app from the development PC to the remote Web server, you can choose:

- Method 1: Deploy the client app to the remote server through the FTP protocol. This requires that

  1) An FTP server is set up on the Web server (the FTP server's physical path must point to the Web root of the Apache HTTP Server: /var/www/html/).

  2) The client app is deployed to the remote Web server through the FTP server.

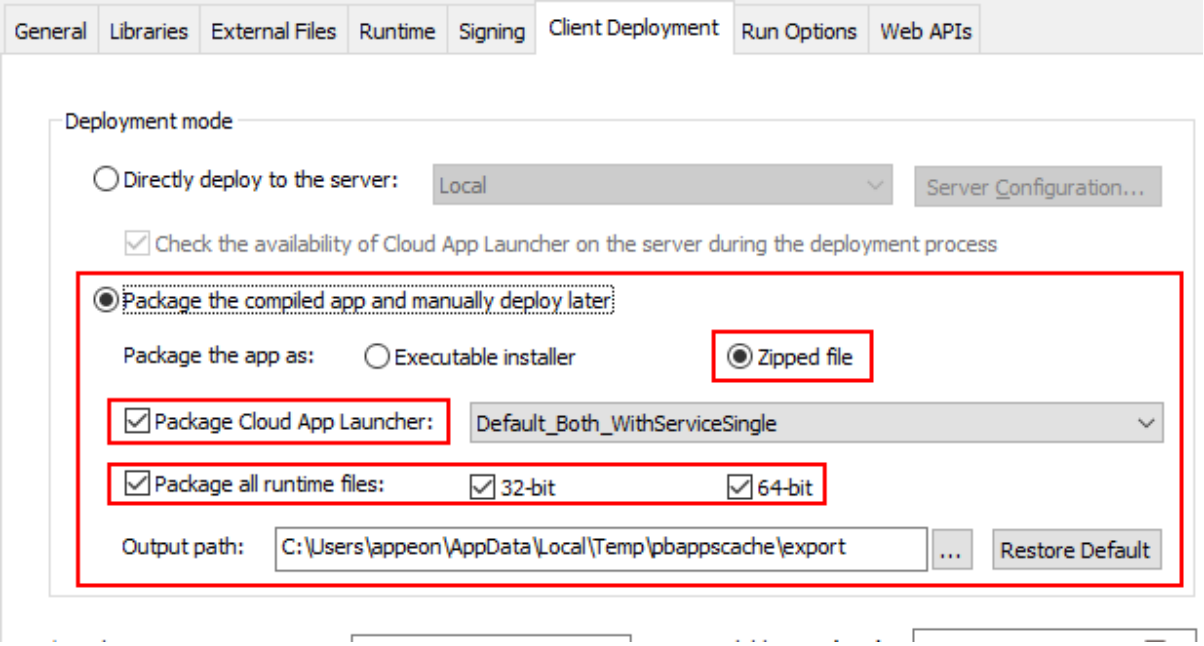
- Method 2: Package the client app and then install (or copy) it to the remote Web server.

  This section will walk you through packaging and copying the client app to the Web root of the Apache HTTP Server: /var/www/html/.

Before you take the steps below to package the client app, make sure you have built the application successfully by following instructions in Tutorial 1 > "Task 4: Setting up the development PC".

Step 1: In the PowerServer project painter, select the **Client Deployment** tab, then select **Package the compiled app and manually deploy later**, and then select **Zipped file**, **Package Cloud App Launcher**, and **Package all runtime files**.

Figure 10.21:



Step 2: Save the project settings and then click the **Build & Deploy PowerServer Project** or **Deploy PowerServer Project** button in the toolbar to generate the package.

When the packaging process is completed, the folder that contains the generated file will be displayed.

Step 3: Copy and extract the generated zipped file to the Web root of the Apache HTTP Server: /var/www/html/.

10.5 Setting up Nginx on Windows

10.5.1 Preparations

In this tutorial, we will set up a Web server running on Nginx.

Step 1: Set up the Web server with the following OS and software:

- Windows Server 2019 (64-bit)
- Nginx 1.19.10

  The next section Installing Nginx has detailed installation instructions.

Step 2: Configure Windows Defender Firewall on the Web server to allow the port (the HTTP port is 80 and the FTP port is 21 in this tutorial). The section "Configuring Windows Defender Firewall" has detailed instructions.

10.5.2 Installing Nginx

Step 1: Download Nginx/Windows-1.19.10 from http://nginx.org/en/download.html.

Step 2: Unzip the downloaded **nginx-1.19.10.zip** file and place the **nginx-1.19.10** folder under the C drive or any location you like.

Step 3: Open the command prompt window, go to the **nginx-1.19.10** folder, and run Nginx.

```
cd C:\nginx-1.19.10
start nginx
```

You could also change the IP address, port number etc. in **conf\nginx.conf** rather than using the default values.

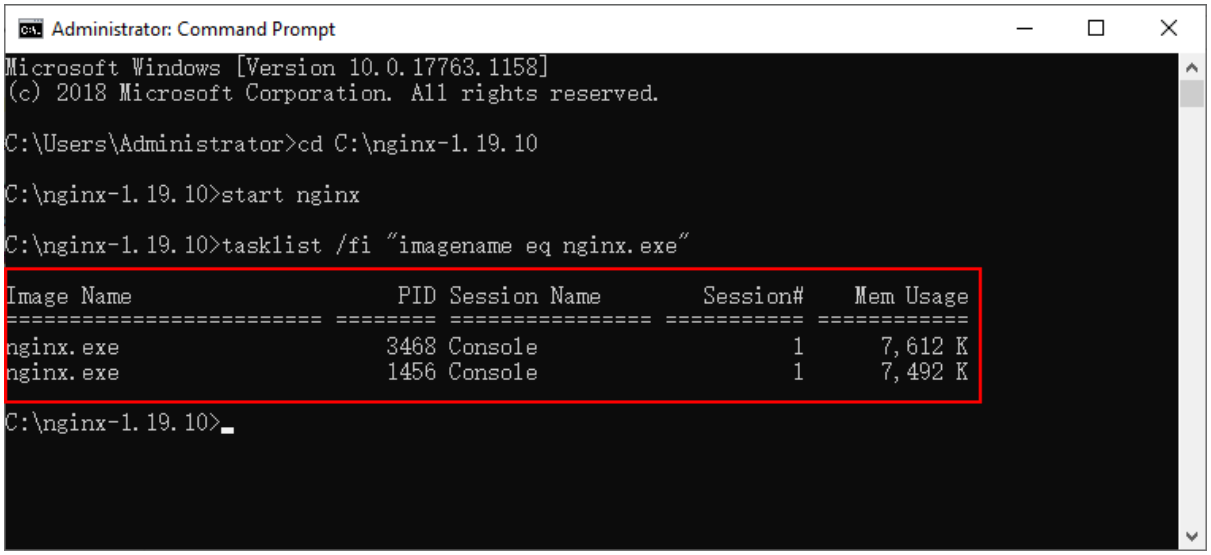
Note

Paths in **nginx.conf** and other configuration files must be specified using forward slashes ("/") instead of back slashes ("\").

Step 4: Run the *tasklist* command to see if the Nginx processes are running.

```
tasklist /fi "imagename eq nginx.exe"
```

Figure 10.22:

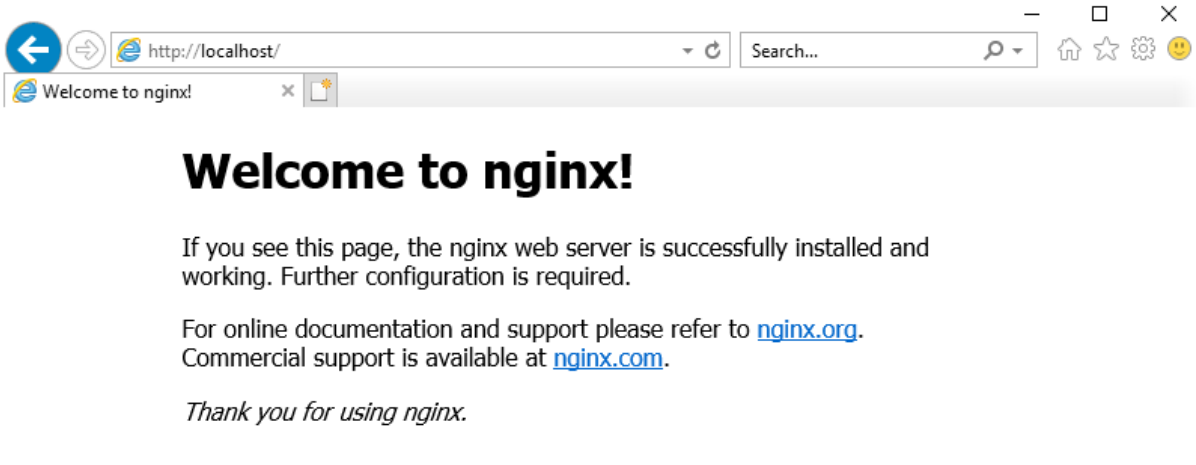


Step 6: Test the Nginx web server by opening up a Web browser and typing in the address: http://localhost.

The following page indicates the Nginx web server is working successfully.

You can further test from the development PC by typing http://*your_server_ip* in a browser.

Figure 10.23:



The Nginx for Windows page has more detailed documentation on using Nginx on Windows.

10.5.3 Configuring SSL on Nginx

It is highly recommended that you configure Secure Sockets Layer (SSL) for the Web server, so that HTTPS can be used to secure the connections between the client and the Web server.

For how to configure SSL on Nginx, refer to http://nginx.org/cn/docs/http/configuring_https_servers.html.

10.5.4 Installing FTP server

Note

To deploy the client app from the development PC to the remote Web server, you can choose:

- Method 1: Deploy the client app to the remote server through the FTP protocol. This requires that

  1) An FTP server is set up on the Web server (the FTP server's physical path must point to the Web root of the Web server).

  This section will walk you through how to set up an FTP server on the Web server.

  2) The client app is deployed to the remote Web server through the FTP server.

  Tutorial 1 > "Task 4: Setting up the development PC" has detailed instructions.

- Method 2: Package the client app and then install (or copy) it to the remote Web server.

  Follow the instructions in Packaging and copying the client app to package the client app and then install (or copy) it to the Web server Web root.

The following steps will walk you through setting up an FTP server on the Web server, so that PowerBuilder can deploy files to the remote server through the FTP protocol.

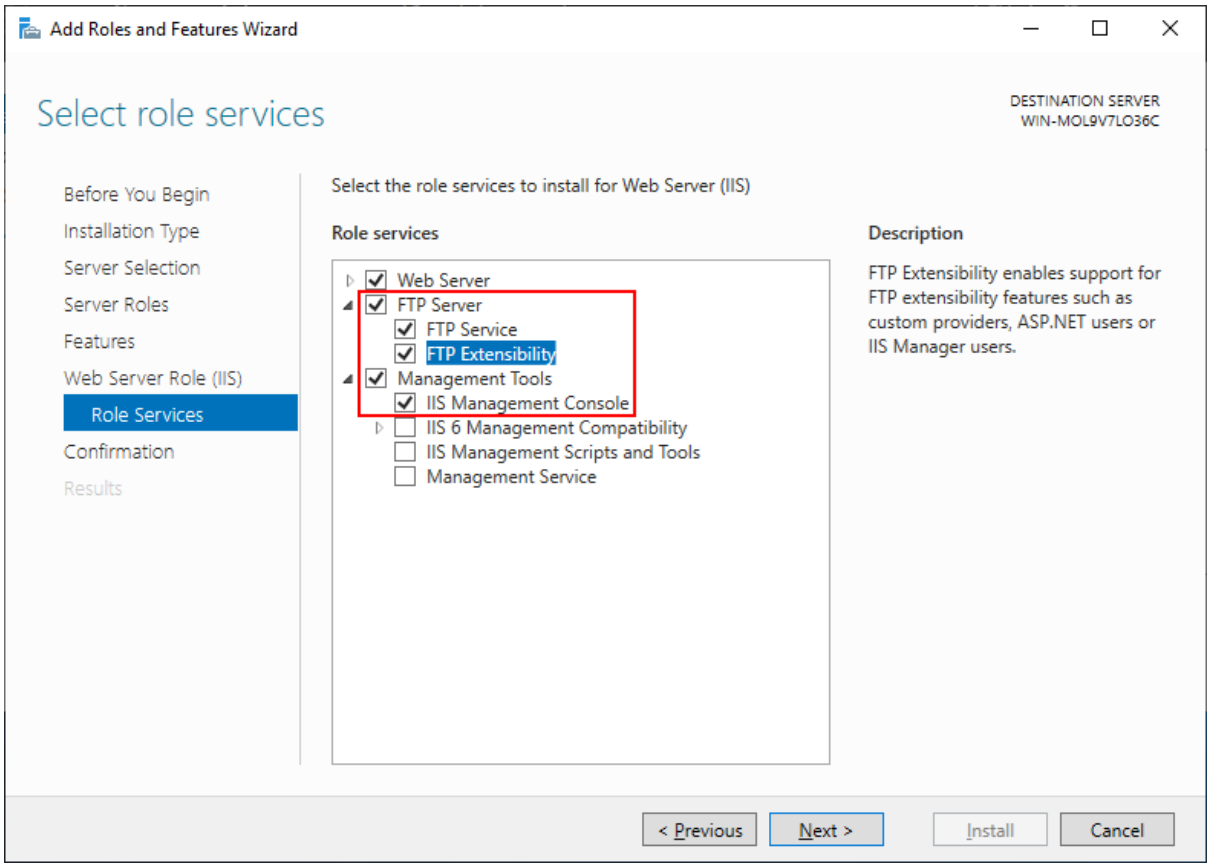
In this tutorial, we set up an IIS FTP server.

To enable the IIS FTP service and create an IIS FTP site,

Step 1: Follow the instructions in Installing Web Server (IIS) until the **Role Services** section displays; and make sure the following role services are selected and installed.

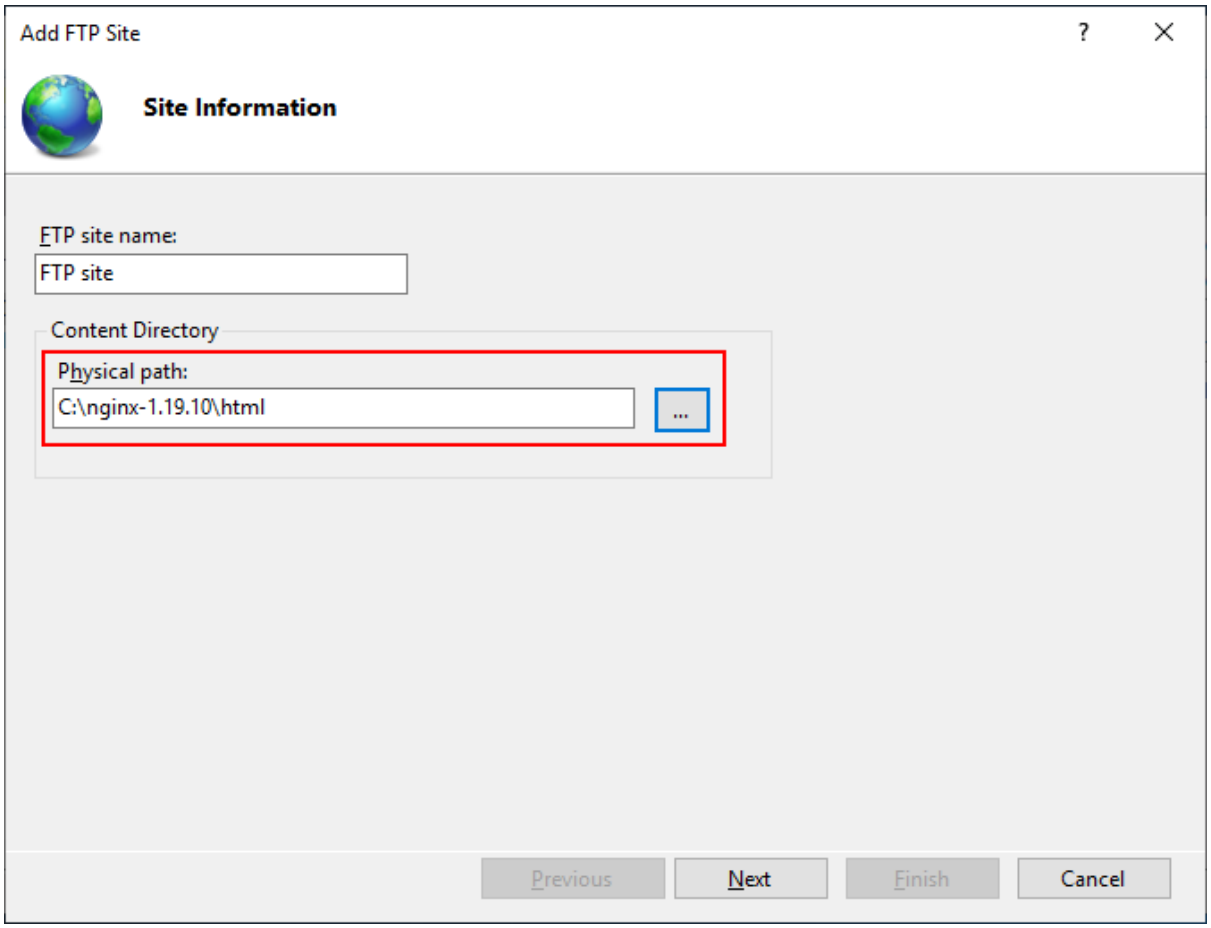
- FTP Server
  - FTP Service
  - FTP Extensibility
- Management Tools
  - IIS Management Console

Figure 10.24:



Step 2: Follow the instructions in Creating an IIS FTP site to create an FTP site and set the physical path to the server root of Nginx which is **nginx-1.19.10\html** by default.

Figure 10.25:



Step 3: Test the FTP site by opening up a Web browser and typing in the address: ftp://localhost.

The following message indicates the FTP site is working successfully.

You can further test from the development PC by typing ftp://*your_server_ip* in a browser. (If access failed, check that if the firewall has blocked the FTP port; you can try to turn off the firewall on the server.)

Figure 10.26:

10.6 Setting up Nginx on Linux

10.6.1 Preparations

In this tutorial, we will set up a Web server running on Nginx on Linux.

Step 1: Set up a Web server with the following OS and software:

- CentOS 8 (64-bit)
- Nginx

  The next section Installing Nginx has detailed installation instructions.

Step 2: Configure the CentOS user account: you can either use the **root** account or create a new account with administrative privileges.

Step 3: Set up a firewall on the server and make sure the firewall allows the port (the HTTP port is 80 in this tutorial) to go through.

Step 4: Make sure the server can connect to Internet during the installation of Nginx.

10.6.2 Installing Nginx

Step 1: Download and install Nginx from the CentOS's default software repositories. Make sure the machine can connect to Internet during the download and installation process.

```
$ sudo dnf install nginx
```

During the download and installation process, you might be prompted to enter the password for your user account, or enter y to confirm that you want to install Nginx.

Step 2: Enable and start the Nginx HTTP server when the installation is completed.

```
$ sudo systemctl enable nginx
```

```
$ sudo systemctl start nginx
```

Step 3: Verify that the Nginx HTTP server service is running.

```
$ sudo systemctl status nginx.service
```

Figure 10.27:

```
appeon@localhost:~
File  Edit  View  Search  Terminal  Help
● nginx.service - The nginx HTTP and reverse proxy server
   Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; vendor preset: disabled)
   Active: active (running) since Sun 2021-05-30 21:56:00 EDT; 4min 58s ago
  Process: 66710 ExecStart=/usr/sbin/nginx (code=exited, status=0/SUCCESS)
  Process: 66708 ExecStartPre=/usr/sbin/nginx -t (code=exited, status=0/SUCCESS)
  Process: 66707 ExecStartPre=/usr/bin/rm -f /run/nginx.pid (code=exited, status=0/SUCCESS)
 Main PID: 66712 (nginx)
    Tasks: 2 (limit: 11155)
   Memory: 8.3M
   CGroup: /system.slice/nginx.service
           ├─66712 nginx: master process /usr/sbin/nginx
           └─66713 nginx: worker process

May 30 21:56:00 localhost.localdomain systemd[1]: Starting The nginx HTTP and reverse proxy server...
May 30 21:56:00 localhost.localdomain nginx[66708]: nginx: the configuration file /etc/nginx/nginx.conf sy>
May 30 21:56:00 localhost.localdomain nginx[66708]: nginx: configuration file /etc/nginx/nginx.conf test i>
May 30 21:56:00 localhost.localdomain systemd[1]: nginx.service: Failed to parse PID from file /run/nginx.>
May 30 21:56:00 localhost.localdomain systemd[1]: Started The nginx HTTP and reverse proxy server.
~
~
~
~
~
lines 1-18/18 (END)
```

Step 4: If you have set up a firewall on the server, run the following command to permanently enable the HTTP service and port 80:

```
# sudo firewall-cmd --permanent --zone=public --add-service=http
```

```
# sudo firewall-cmd --permanent --zone=public --add-port=80/tcp
```

To apply the changes, reload the firewall service using the following command:

```
# sudo firewall-cmd --reload
```

To verify that the http service and port 80 were added successfully, you can run:

```
# sudo firewall-cmd --permanent --list-all
```

Figure 10.28:

```
[appeon@localhost ~]$ sudo firewall-cmd --permanent --list-all
[sudo] password for appeon:
public
  target: default
  icmp-block-inversion: no
  interfaces:
  sources:
  services: cockpit dhcpv6-client http ssh
  ports: 80/tcp
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

Step 5: Test the Nginx HTTP server by opening up a Web browser and typing in the address: http://localhost.

The following page indicates the Nginx HTTP server is installed and started successfully.

You can further test from the development PC by typing http://*your_server_ip* in a browser.

Figure 10.29:

10.6.3 Configuring SSL on Nginx

It is highly recommended that you configure Secure Sockets Layer (SSL) for the Web server, so that HTTPS can be used to secure the connections between the client and the Web server.

For how to configure SSL on Nginx, refer to http://nginx.org/cn/docs/http/configuring_https_servers.html.

10.6.4 Configuring Nginx to be case-insensitive

As PowerBuilder is designed to be case-insensitive and always uses lower cases to access the deployed folders/files, therefore, in a case-sensitive file system like Linux, folder/file names (such as images etc.) containing upper cases may not be found or loaded.

To avoid such issues, you are recommended to

- Change the folder/file names (such as theme, images etc.) to use all lower cases; or
- Configure Nginx in Linux to be case-insensitive.

To configure Nginx in Linux to be case-insensitive,

1. Download the ngx_http_lower_upper_case module and Nginx source code.

   Suppose ngx_http_lower_upper_case is de-compressed to the folder: /src/case/, and Nginx is de-decompressed to the folder: /src/ngx/.

2. Go to the Nginx folder and load the ngx_http_lower_upper_case module.

```
cd /src/ngx/nginx-1.21.3
```

```
./configure --prefix=/nginx  \
--add-module=/src/case/ngx_http_lower_upper_case
```

3. Compile the Nginx source code.

```
make
```

4. If Nginx is already installed, stop Nginx, copy the files from /src/ngx/nginx-1.21.3/objs/nginx to replace the existing ones, and then restart Nginx.

   If Nginx is not yet installed, execute the following command to install Nginx:

```
make install
```

5. Go to the Nginx installation folder and open the nginx.conf file in a text editor.

   Locate the "location" block and modify it like below:

```
location ~[A-Z+] {
          lower $caseurl $request_uri;
          rewrite  ^(.*)  $caseurl  last;
}
```

6. Reload the Nginx configuration.

```
nginx –s reload
```

10.6.5 Packaging and copying the client app

To deploy the client app from the development PC to the remote Web server, you can choose:

- Method 1: Deploy the client app to the remote server through the FTP protocol. This requires that

  1) An FTP server is set up on the Web server (the FTP server's physical path must point to the Web root of the Nginx HTTP server which is /usr/share/nginx/html by default).

  2) The client app is deployed to the remote Web server through the FTP server.

- Method 2: Package the client app and then install (or copy) it to the remote Web server.

  Follow the instructions in Packaging and copying the client app to package and copy the client app to the Web root of the Nginx HTTP server: /usr/share/nginx/html.

11 Tutorial 11: Deploying installable cloud apps to Kubernetes

11.1 Overview

You can deploy the PowerBuilder installable cloud applications (including the client app, the PowerServer Web APIs, and the database) to Kubernetes.

In this tutorial, we will take Azure Kubernetes Service (AKS) as an example to show you how to create a Kubernetes cluster in AKS and then deploy the PowerBuilder installable cloud application to it.

This tutorial assumes a basic understanding of the following concepts:

- Kubernetes concepts. For more information, see Kubernetes core concepts for Azure Kubernetes Service (AKS).
- Docker concepts. For more information, see Docker overview.

Generally speaking, this tutorial accomplish the following major tasks:

- Creating a Kubernetes cluster
- Containerizing your application
- Deploying the containerized application to the Kubernetes cluster

11.2 Before you begin

Prepare a local machine that can connect with the Kubernetes cluster and deploy the Docker container image to the cluster.

You should install the following OS and software to the machine:

- Windows 10
- Docker Desktop

  Docker Desktop includes Docker Engine, Docker CLI client, Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

  After installing Docker Desktop, you need to enable Kubernetes support. To do that, go to **Preferences** > **Kubernetes** and then click **Enable Kubernetes**.
- Azure CLI

  This tutorial requires that you are running the Azure CLI version 2.0.64 or later. Run **az --version** to find the version. If you need to install or upgrade, see Install Azure CLI.
- kubectl

  You will need to use kubectl to connect to the Kubernetes cluster from your local computer and create deployments for the application.

kubectl is already installed if you use the Azure Cloud Shell. You can also install it locally using the **az aks install-cli** command.

- Helm

  To install the Nginx ingress controller, you use Helm. Make sure you are using the latest release of Helm and have access to the ingress-nginx Helm repository.

There are several ways to accomplish a task in Azure, for example, you can create an AKS cluster using the Azure portal, a PowerShell script, an Azure CLI script etc. In this tutorial, we will take priority in using the Azure portal whenever possible.

11.3 Configuring Azure Kubernetes Service

11.3.1 Creating a Kubernetes cluster in AKS

You can create a Kubernetes cluster in Azure Kubernetes Service (AKS) using either of the following methods:

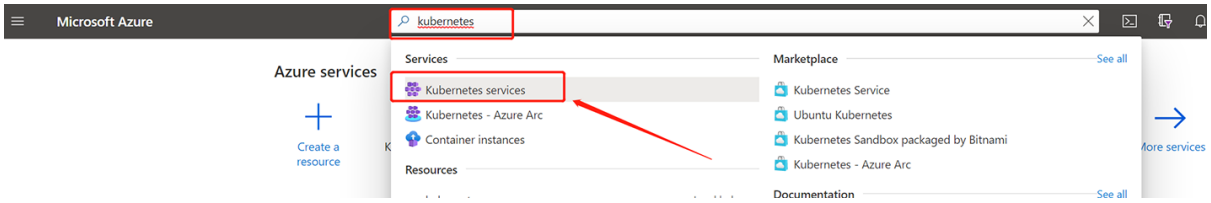
- The Azure CLI
- The Azure portal
- Azure PowerShell
- Using template-driven deployment options, like Azure Resource Manager templates and Terraform

This tutorial will show you how to create the cluster using the Azure portal.

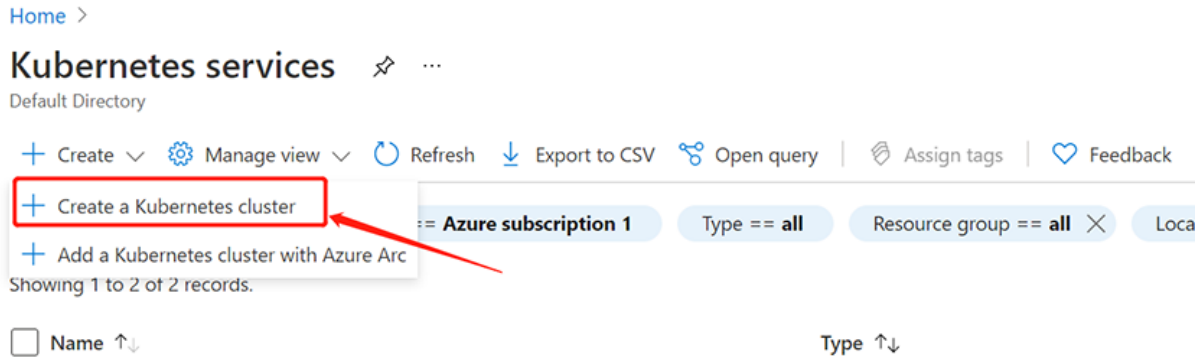
Step 1: Sign in to the Azure portal at https://portal.azure.com.

Step 2: In the search box at the top of the portal, enter **Kubernetes**.

Step 3: In the search results, select **Kubernetes services**.



Step 4: Select **Create a Kubernetes cluster**.



Step 5: On the **Basics** page, configure the following options:

- **Project details**:
  - Select an Azure **Subscription**.
  - Select or create an Azure **Resource group**, such as *pscloudapp*.
- **Cluster details**:
  - Ensure the **Preset configuration** is *Standard*. For more details on preset configurations, see Cluster configuration presets in the Azure portal.
  - Enter a **Kubernetes cluster name**, such as *pscloudapp*.
  - Select a **Region** and **Kubernetes version** for the AKS cluster.
- **Primary node pool**:
  - Leave the default values selected.

Create Kubernetes cluster ...

Basics | Node pools | Authentication | Networking | Integrations | Tags | Review + create

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. Learn more about Azure Kubernetes Service

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Azure subscription 1

Resource group * : (New) Resource group

Create new

Cluster details

Preset configuration: **Standard**
Quickly customize your cluster by choosing the preset configuration applicable to your scenario. Depending on the selection, values of certain fields might change in different tabs. You can modify these values at any time.
View all preset configurations

Kubernetes cluster name * : pscloudapp

Region * : (US) West US 2

Availability zones : Zones 1,2,3

High availability is recommended for standard configuration.

Kubernetes version * : 1.20.9 (default)

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. Learn more about node pools in Azure Kubernetes Service

Node size * : **Standard DS2 v2**
Standard DS2_v2 is recommended for standard configuration.
Change size

Scale method * : Manual / Autoscale
Autoscaling is recommended for standard configuration.

Node count range * : 1 — 5

Review + create | < Previous | Next : Node pools >

Step 6: Select **Next: Node pools** at the bottom of the screen.

Step 7: On the **Node pools** page, keep the default options. Click **Next: Authentication** at the bottom of the screen.

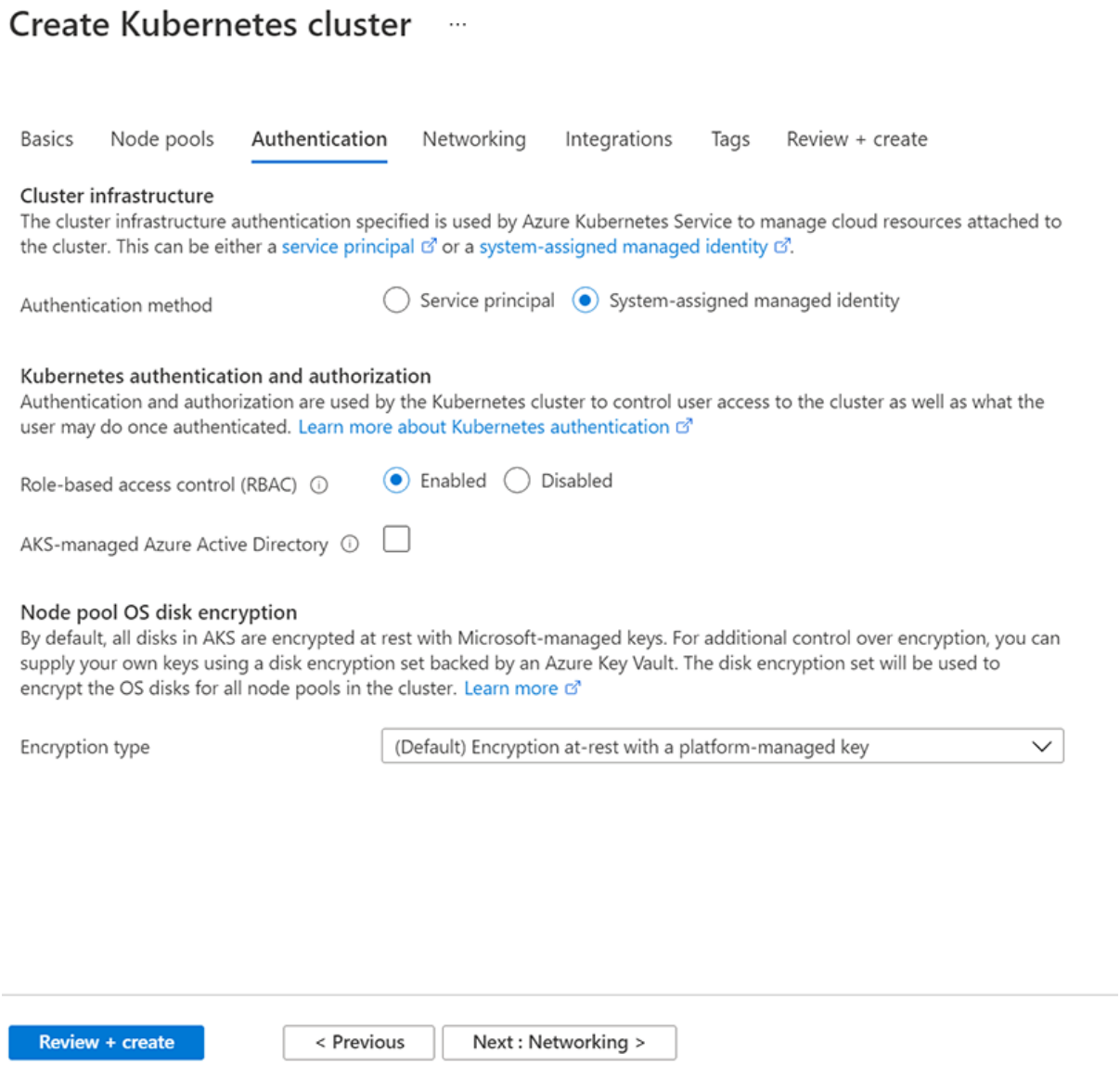
Step 8: On the **Authentication** page, configure the following options:

- Leave the **Authentication method** field with **System-assigned managed identity**.

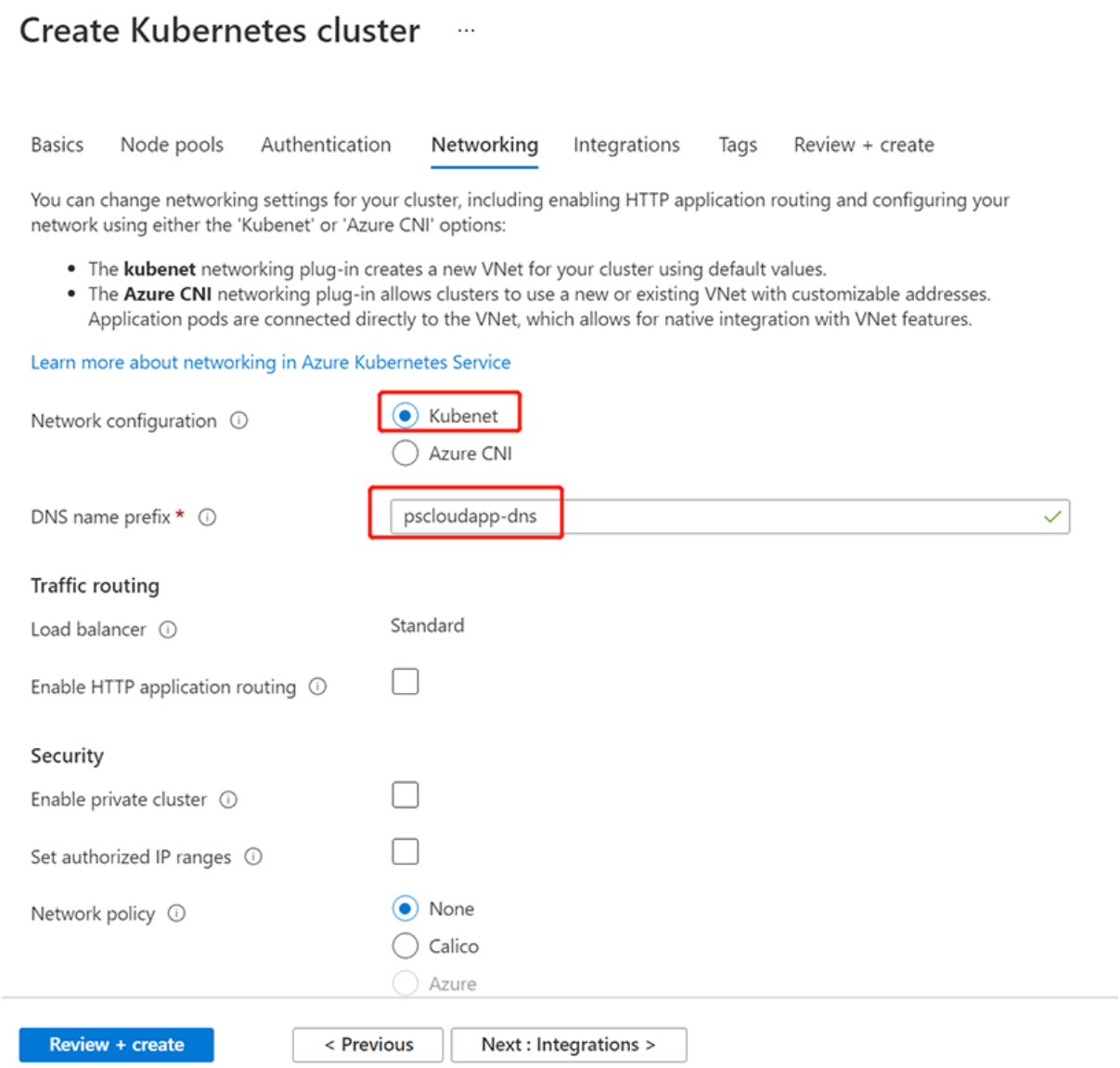
  To avoid needing an **Owner** or **Azure account administrator** role, you can also manually configure a service principal to pull images from ACR. For more information, see ACR authentication with service principals or Authenticate from Kubernetes with a pull secret.

- Enable the Kubernetes role-based access control (Kubernetes RBAC) option to provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

Step 9: Click **Next: Networking** at the bottom of the screen.



Step 10: On the **Networking** page, select **Kubenet**. Click **Next: Integrations** at the bottom of the screen.

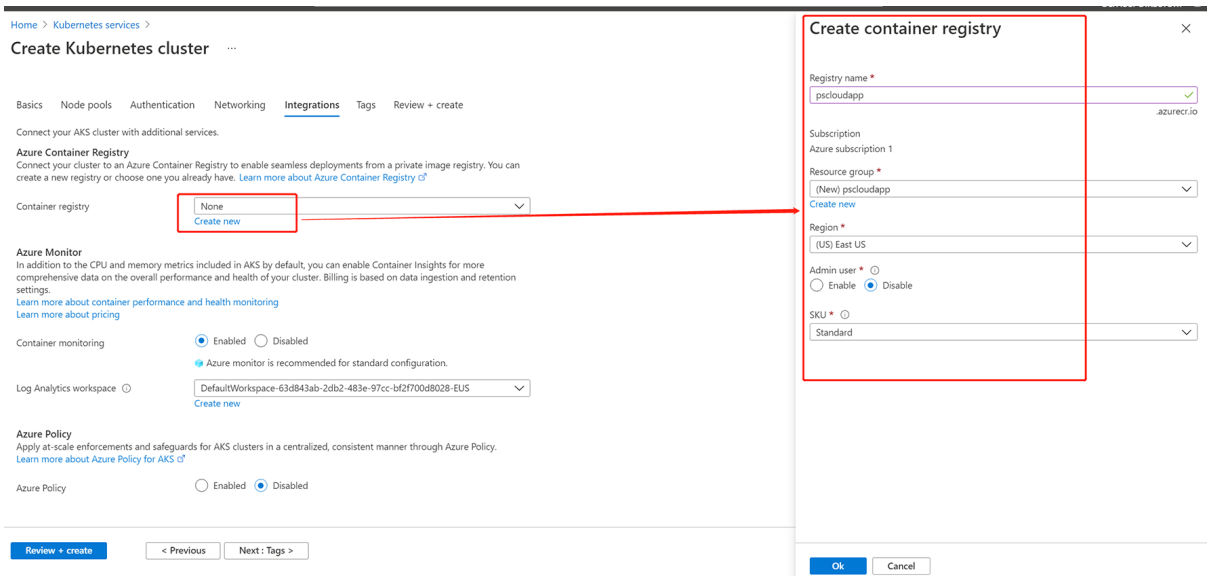


Step 11: On the **Integrations** page, configure the following options:

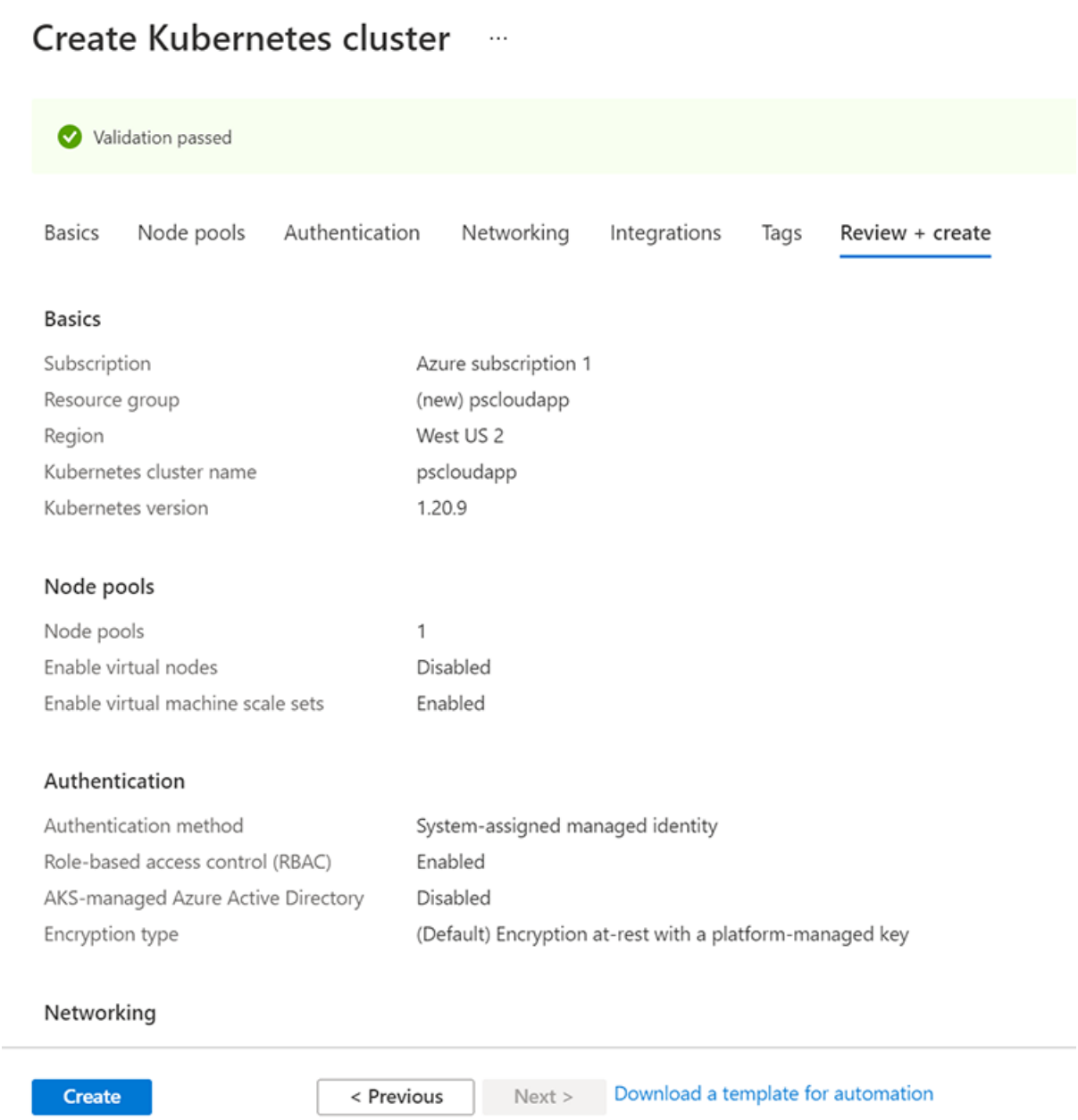
- In the **Container registry** section, select **Create new** to create a new container registry.

  If you selected **Service principal** authentication method, you can only select **None** in the **Container registry** section.

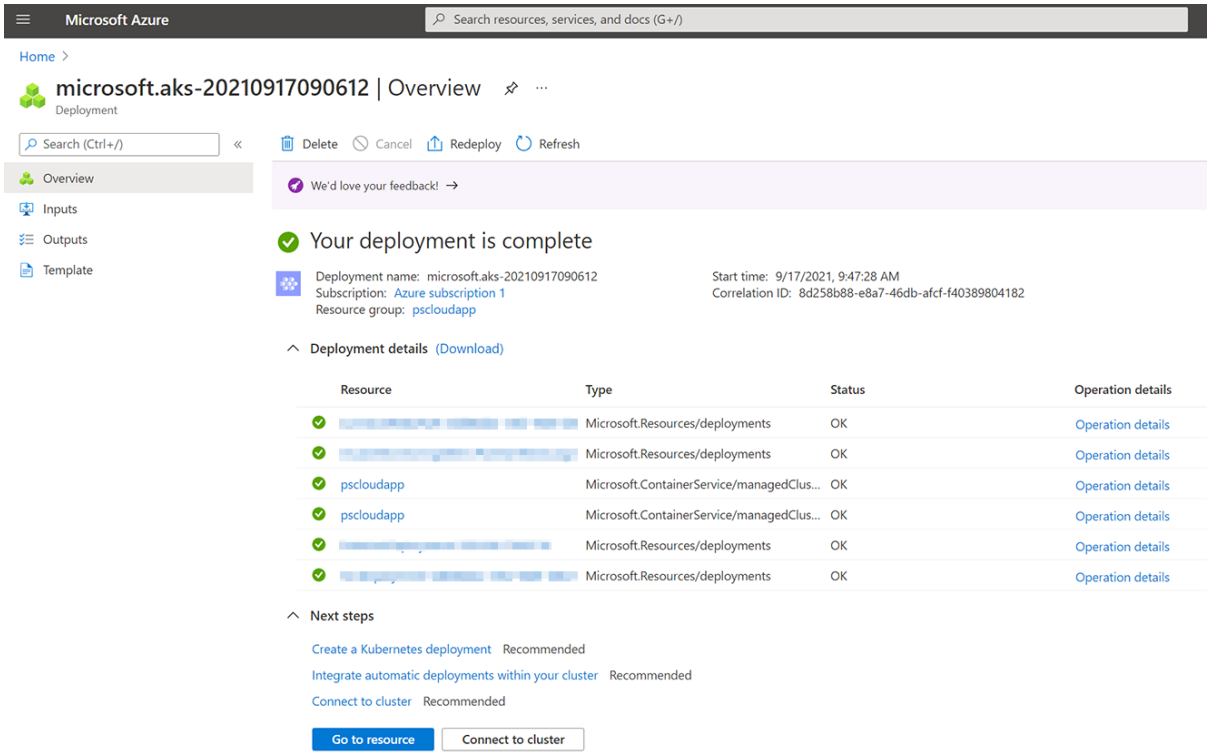
Step 12: Click **Review + create** at the bottom of the screen.



Step 13: When validation completes, click **Create**.



Step 14: When deployment completes, view the details.



11.3.2 Connecting to the Kubernetes cluster

Step 1: Get the authentication code for logging into Azure.

```
Az login --use-device-code
```

```
C:\cloudappdemo>az login --use-device-code
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code HKUEW8893 to authenticate.
```

Step 2: Follow the instructions in the output to log in to Azure.

When login is successful, the following information will display.

```
C:\cloudappdemo>
C:\cloudappdemo>az login --use-device-code
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code HKUEW8893 to authenticate.
The following tenants don't contain accessible subscriptions. Use 'az login --allow-no-subscriptions' to have tenant level access.
293e2fe0-27d1-43ae-8f50-4de65f125941 'PS B2C'
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "[redacted]",
    "id": "[redacted]",
    "isDefault": true,
    "managedByTenants": [],
    "name": "[redacted]",
    "state": "Enabled",
    "tenantId": "[redacted]",
    "user": {
      "name": "[redacted]",
      "type": "user"
    }
  },
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "[redacted]",
    "id": "[redacted]",
    "isDefault": false,
    "managedByTenants": [],
    "name": "[redacted]",
    "state": "Enabled",
    "tenantId": "[redacted]",
    "user": {
      "name": "[redacted]",
      "type": "user"
    }
  }
]

C:\cloudappdemo>
```

Step 3: Configure the kubectl to connect to your Kubernetes cluster using the **az aks get-credentials** command. For example,

```
az aks get-credentials -g pscloudapp -n pscloudapp
```

The command downloads credentials and configures the Kubernetes command-line tool to use them.

```
C:\cloudappdemo>az aks get-credentials -g pscloudapp -n pscloudapp
Merged "pscloudapp" as current context in C:\Users\appeon\.kube\config
```

Step 4: View the connection to your cluster using the following command.

```
Kubectl get nodes
```

The output returns a list of the cluster nodes, make sure the node status is ready.

11.3.3 Installing ingress controller

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. For more, refer to https://docs.microsoft.com/en-us/azure/aks/ingress-basic. You can choose from a number of ingress controllers.

This tutorial shows you how to install the Nginx ingress controller in the AKS cluster.

11.3.3.1 Creating public IP address

By default, an Nginx ingress controller is created with a new public IP address assignment. This public IP address is only static for the life-span of the ingress controller, and is lost if the controller is deleted and re-created. A common configuration requirement is to provide

the Nginx ingress controller an existing static public IP address. The static public IP address remains if the ingress controller is deleted. This approach allows you to use existing DNS records and network configurations in a consistent manner throughout the life cycle of your applications.

There are several ways to create a static public IP address, as described here. In this tutorial, you will create a static public IP address using the Azure portal.

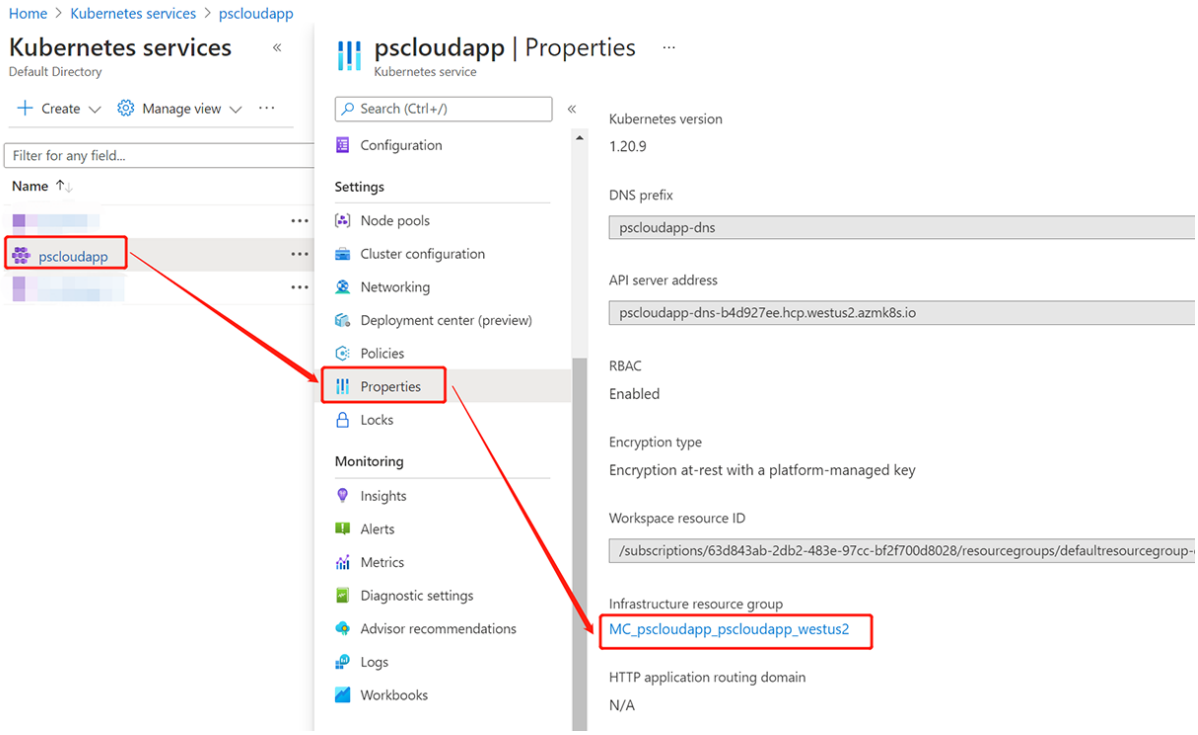
Step 1: Get the resource group name of the AKS cluster before you create the static public IP address:

1. In the Azure portal, select the resource group.

2. Select the **Properties** page.

3. Make a note of the **Infrastructure resource group**.

   For example, the infrastructure resource group for *pscloudapp* is *MC_pscloudapp_pscloudapp_westus2*.



Step 2: Create a static public IP address.

The following example creates a static public IP address named *pscloudapp* in the AKS cluster resource group obtained in the previous step:

1. In the search box at the top of the portal, enter **Public IP**.

2. In the search results, select **Public IP addresses**.

3. Select **+ Create**.

4. In **Create public IP address**, enter, or select the following information:

Select **Standard** from SKU.

Enter a name for the IP address.

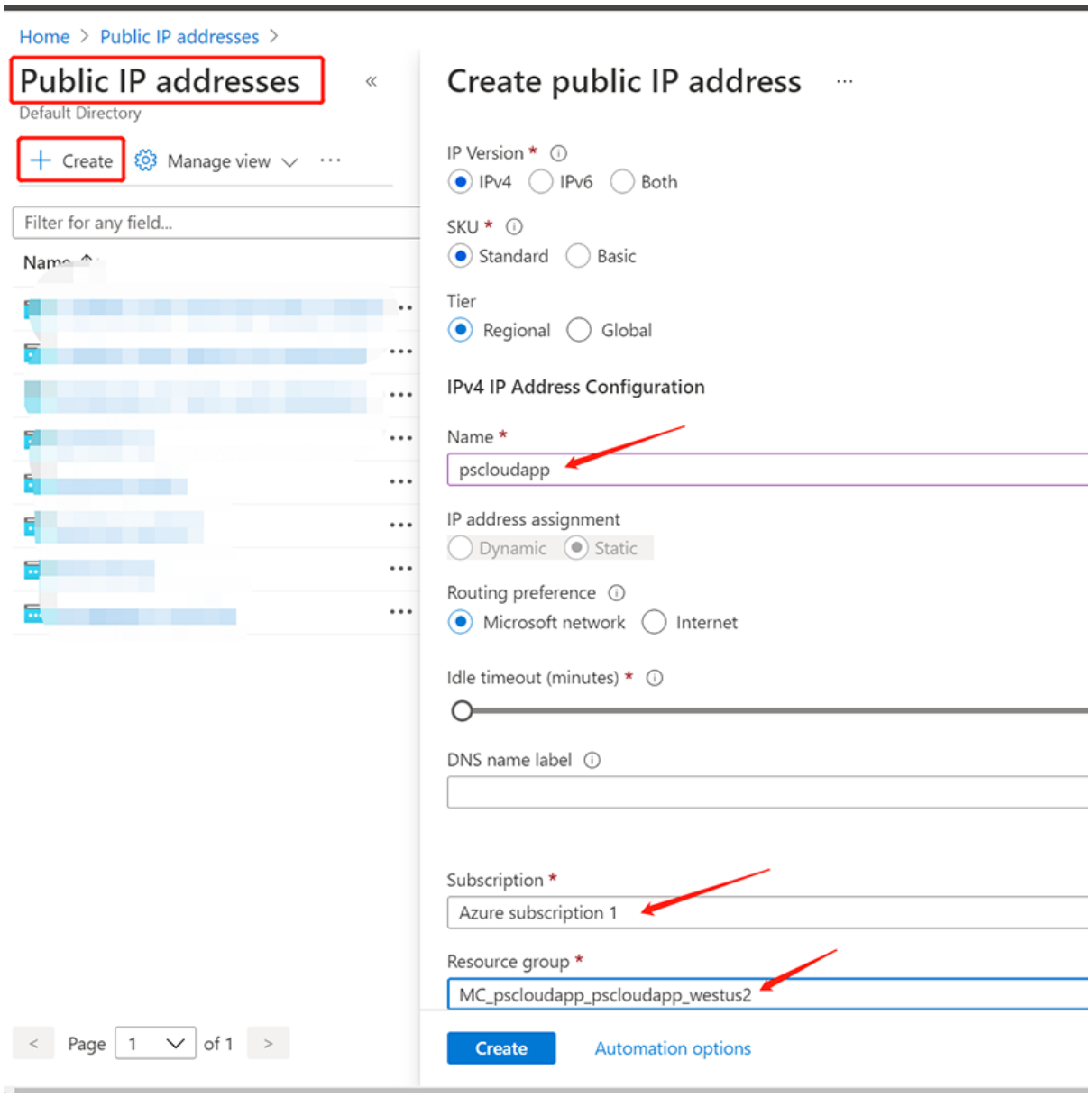
Select your subscription.

Select *MC_pscloudapp_pscloudapp_westus2* from **Resource group**.

Select the same location as the cluster.

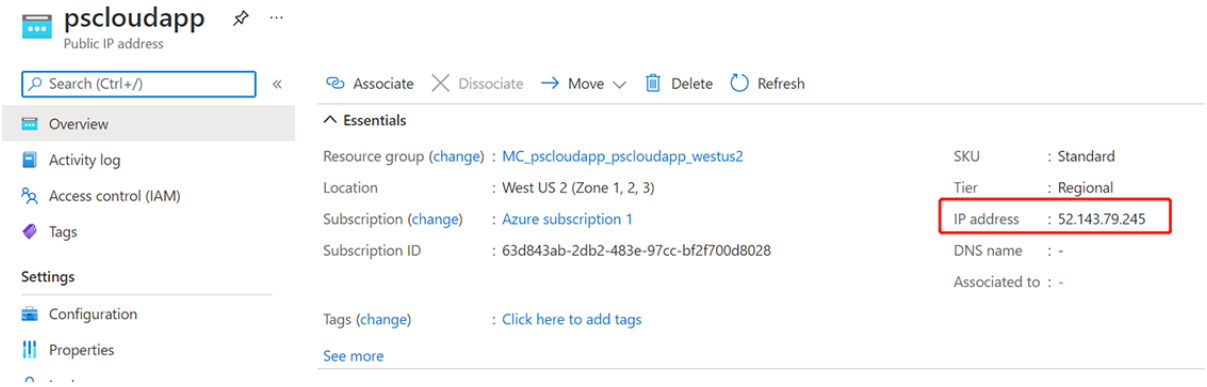
Leave the others as default.

Finally, click **Create**.



When the IP address is created successfully, make a note of the IP address, for example, 52.143.79.245.

You will associate this static public IP address with the Nginx ingress controller in the next section, and you may use it to access the application later.



11.3.3.2 Creating a Kubernetes namespace

Before installing Ingress-Nginx, you are recommended to create a new Kubernetes namespace for the ingress resources.

For example, execute the following command to create a new namespace: **ingress-basic-pscloudapp**.

```
kubectl create namespace ingress-basic-pscloudapp
```

```
C:\cloudappdemo>kubectl create namespace ingress-basic-pscloudapp
namespace/ingress-basic-pscloudapp created

C:\cloudappdemo>kubectl get namespace
NAME                      STATUS   AGE
default                   Active   13d
ingress-basic-pscloudapp  Active   10s
kube-node-lease           Active   13d
kube-public               Active   13d
kube-system               Active   13d
```

11.3.3.3 Installing Ingress-Nginx

Step 1: Add the repo of the ingress-nginx repository to your helm config:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

Step 2: Install the Nginx ingress controller in the **ingress-basic-pscloudapp** namespace created in the previous step.

- The static public IP address created in the earlier step will be assigned to the ingress controller using the **--set controller.service.loadBalancerIP** parameter.
- For added redundancy, two replicas of the Nginx ingress controllers are deployed with the **--set controller.replicaCount** parameter.
- The ingress controller also needs to be scheduled on a Linux node. Windows Server nodes shouldn't run the ingress controller. A node selector is specified using the **--**

set nodeSelector parameter to tell the Kubernetes scheduler to run the Nginx ingress controller on a Linux-based node.

```
helm install nginx-ingress ingress-nginx/ingress-nginx \
        --namespace ingress-basic-pscloudapp \
        --set controller.replicaCount=2 \
        --set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
        --set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux \
        --set controller.admissionWebhooks.patch.nodeSelector."beta\.kubernetes
\.io/os"=linux \
        --set controller.service.loadBalancerIP="52.143.79.245"
```

```
C:\Users\appeon>helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
"ingress-nginx" has been added to your repositories
```

```
C:\Users\appeon>helm install nginx-ingress ingress-nginx/ingress-nginx --namespace ingress-basic-pscloudapp --set controller.replicaCount=2 --set controller.nodeSelector."beta\.kubernetes\.io/os"=linux --set defaultBackend.nodeSel
ector."beta\.kubernetes\.io/os"=linux --set controller.admissionWebhooks.patch.nodeSelector."beta\.kubernetes\.io/os"=linux --set controller.service.loadBalancerIP="52.143.79.245"
NAME: nginx-ingress
LAST DEPLOYED: Thu Sep 30 14:52:32 2021
NAMESPACE: ingress-basic-pscloudapp
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The ingress-nginx controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running 'kubectl --namespace ingress-basic-pscloudapp get services -o wide -w nginx-ingress-ingress-nginx-controller'

An example Ingress that makes use of the controller:

  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    annotations:
      kubernetes.io/ingress.class: nginx
    name: example
    namespace: foo
  spec:
    rules:
      - host: www.example.com
        http:
          paths:
            - backend:
                serviceName: exampleService
                servicePort: 80
              path: /
    # This section is only required if TLS is to be enabled for the Ingress
    tls:
        - hosts:
            - www.example.com
          secretName: example-tls

If TLS is enabled for the Ingress, a Secret containing the certificate and key must also be provided:

  apiVersion: v1
  kind: Secret
  metadata:
    name: example-tls
    namespace: foo
  data:
    tls.crt: <base64 encoded cert>
    tls.key: <base64 encoded key>
  type: kubernetes.io/tls

C:\Users\appeon>
```

Step 3: View the installed Nginx ingress controller.

```
kubectl --namespace ingress-basic-pscloudapp get services -o wide -w nginx-ingress-
ingress-nginx-controller
```

```
C:\Users\appeon>kubectl --namespace ingress-basic-pscloudapp get services -o wide -w nginx-ingress-ingress-nginx-controller
NAME                                       TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)                      AGE     SELECTOR
nginx-ingress-ingress-nginx-controller     LoadBalancer   10.0.157.20   52.143.79.245   80:32365/TCP,443:30416/TCP   8m41s   app.kubernetes.io
/component=controller,app.kubernetes.io/instance=nginx-ingress,app.kubernetes.io/name=ingress-nginx
```

A Kubernetes load balancer service is created for the Nginx ingress controller; and the static public IP address is assigned.

11.3.3.4 Using your own TLS certificates in AKS

You can generate your own certificates, and create a Kubernetes secret for use with the ingress route.

For testing purpose, you can generate a self-signed certificate with openssl. For production use, you should request a trusted, signed certificate through a provider or your own certificate authority (CA).

In this tutorial, we assume that you have already generated a TLS certificate and a private key.

Now, you will generate a Kubernetes TLS secret using the TLS certificate and the private key.

11.3.3.4.1 Creating Kubernetes secret for the TLS certificate

To allow Kubernetes to use the TLS certificate and private key for the ingress controller, you create and use a Secret.

The secret is defined once, and will be referenced later when you define ingress routes.

Step 1: Copy the certificate and the private key to the local publish directory, for example, C:\cloudappdemo\publish.

Step 2: Create a secret. For example, the following command creates a secret named **aks-ingress-tls-appeon.com**.

```
kubectl create secret tls aks-ingress-tls-appeon.com \
          --key server.key \
          --cert server_appeon.com_ssl.cer
```

```
C:\cloudappdemo\publish>kubectl create secret tls aks-ingress-tls-appeon.com --key server.key --cert server_appeon.com_ssl.cer
secret/aks-ingress-tls-appeon.com created
```

11.3.3.4.2 (Optional) Adding the default certificate

You can also add a default certificate, so that it displays no matter when the IP address or domain name is accessed.

Step 1: Edit the Nginx-Ingress deployment configuration file. For example,

```
kubectl edit deployment nginx-ingress-ingress-nginx-controller -o yaml -n ingress-basic-pscloudapp
```

```
C:\cloudappdemo\publish>kubectl edit deployment nginx-ingress-ingress-nginx-controller -o yaml -n ingress-basic-pscloudapp
```

Step 2: Add the following parameter to the configuration.

```
- --default-ssl-certificate=default/aks-ingress-tls-appeon.com
```

```
  spec:
    containers:
    - args:
      - /nginx-ingress-controller
      - --publish-service=$(POD_NAMESPACE)/nginx-ingress-ingress-nginx-controller
      - --election-id=ingress-controller-leader
      - --controller-class=k8s.io/ingress-nginx
      - --configmap=$(POD_NAMESPACE)/nginx-ingress-ingress-nginx-controller
      - --validating-webhook=:8443
      - --validating-webhook-certificate=/usr/local/certificates/cert
      - --validating-webhook-key=/usr/local/certificates/key
      - --default-ssl-certificate=default/aks-ingress-tls-appeon.com
      env:
      - name: POD_NAME
        valueFrom:
          fieldRef:
            apiVersion: v1
```

11.3.4 Logging into Azure container registry

During the process of creating the Kubernetes cluster in AKS, you should have already created an Azure container registry. If not, create one in the Azure portal (by selecting **Create** in the **Home** > **Container registries** page) or using the **az acr create** command.

The Azure container registry is your private Docker registry in Azure. Later, you will push the Docker container images (running the installable cloud app) to the Azure container registry.

You must log in to the Azure container registry before pushing images to it. Take note of the username, password, and login server name of the container registry. You will need this information later.

Step 1: Get the username and password for the container registry.

1. In the Azure portal, select the container registry > **Access keys**.

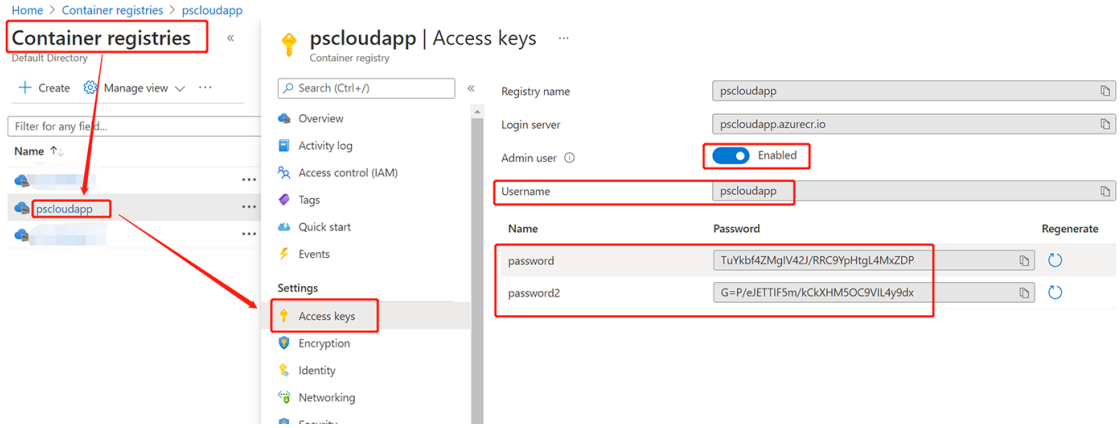
2. Set **Admin user** to **Enabled**.

3. Make a note of the username and password, for example,

   Username: pscloudapp

   Password: TuYkbf4ZMgIV42J/RRC9YpHtgL4MxZDP

   Password2: G=P/eJETTIF5m/kCkXHM5OC9VIL4y9dx



Step 2: Integrate the container registry with the AKS cluster. For example,

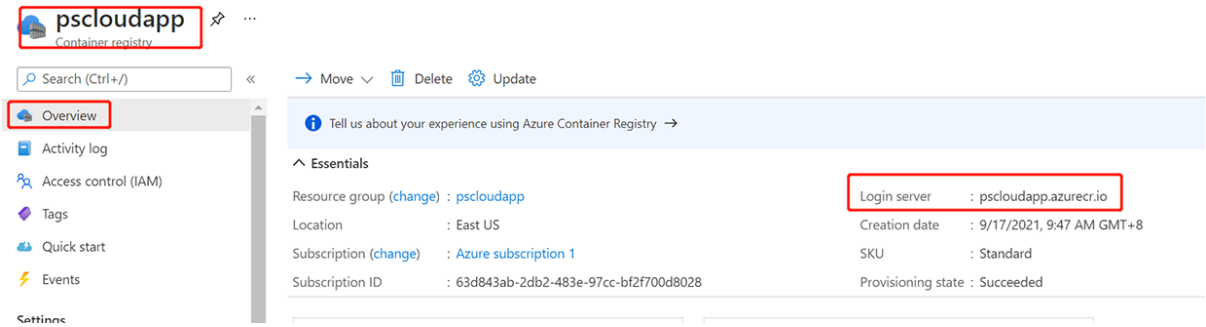
```
az aks update -n pscloudapp -g pscloudapp --attach-acr pscloudapp
```

Step 3: Get the full login server name of the container registry.

1. In the Azure portal, select the container registry > **Overview**.

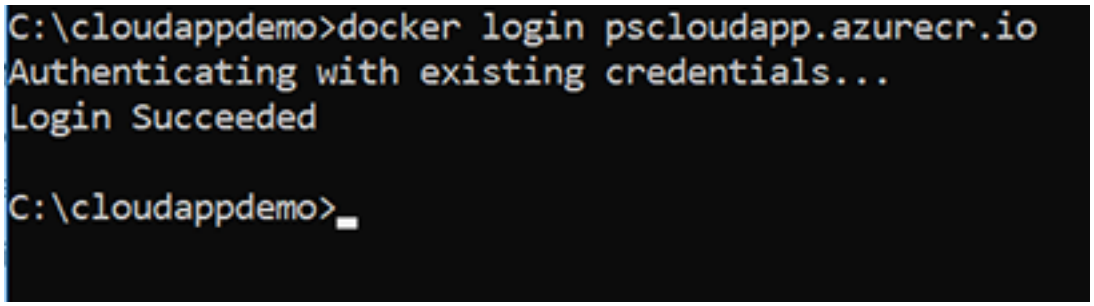
2. Make a note of the login server name.

   For example, pscloudapp.azurecr.io. It will be used to log into the container registry as well as push the images to the registry.



Step 4: Log into the container registry using the full login server name. For example,

```
docker login pscloudapp.azurecr.io
```



11.3.5 Creating a database

For optimal performance, it is highly recommended that the application database is also running in the same Azure instance.

You can create a database in Azure using the Azure portal, a PowerShell script, or an Azure CLI script.

In this tutorial, you will create a database using the Azure portal; and you will create an Azure SQL Database.

Azure SQL Database is based on the latest stable version of the Microsoft SQL Server database engine.

Step 1: In the search box at the top of the portal, enter **SQL database**.

Step 2: In the search results, select **SQL database**.

Step 3: Select **Create**.

Step 4: On the **Basics** page, configure the following options:

1. Select the subscription.

2. Select the resource group.

3. Enter any text as the database name.

4. For **Server**, select **Create New** to create a new server and specify the following:

   a. Server name: Enter a unique name as the server name, for example, pscloudapp (so the full name is pscloudapp.database.windows.net)

   b. Server admin login: Enter any text as the administrator user name, for example, appeon

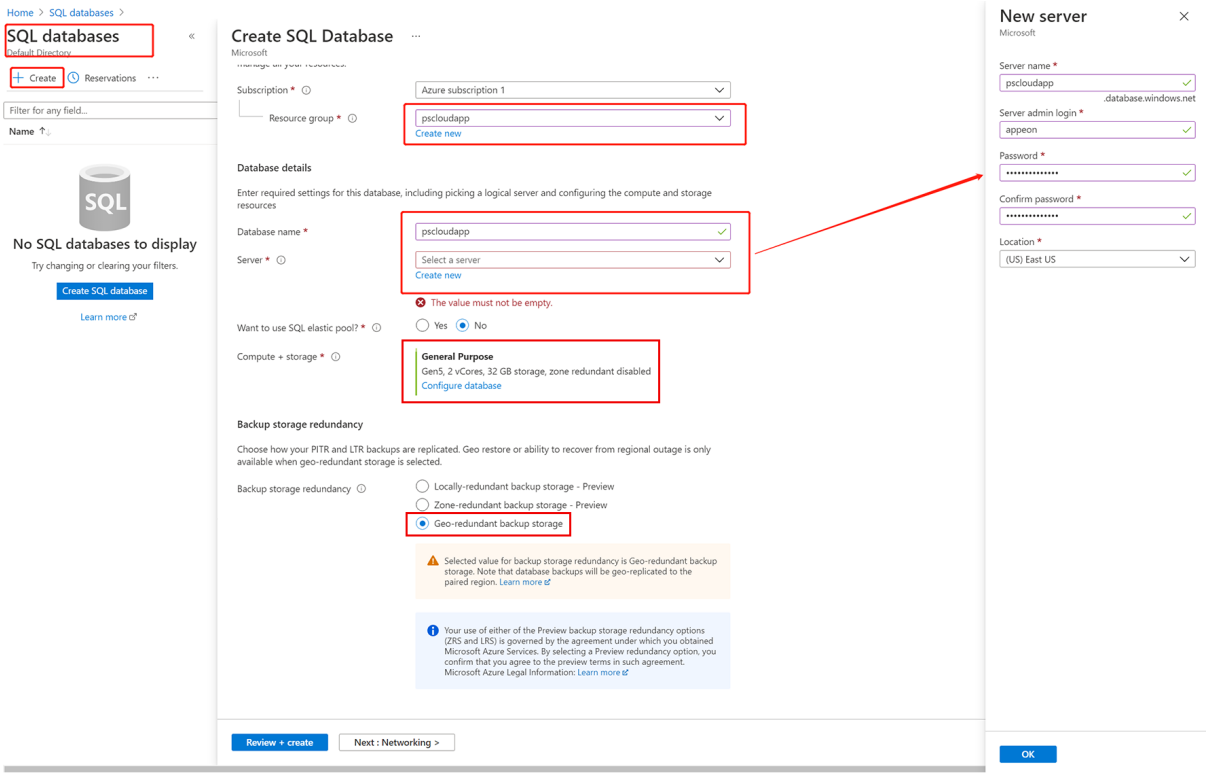
   c. Password: Enter a password that meets the requirement, for example, pwdsqlserver8*

5. Leave **Want to use SQL elastic pool** set to **No**.

6. Under **Compute + storage**, select **Configure database**, and then select the appropriate options and click **Apply**.

7. For **Backup storage redundancy**, select **Geo-redundant backup storage**.

8. Select **Next: Networking** at the bottom of the page.

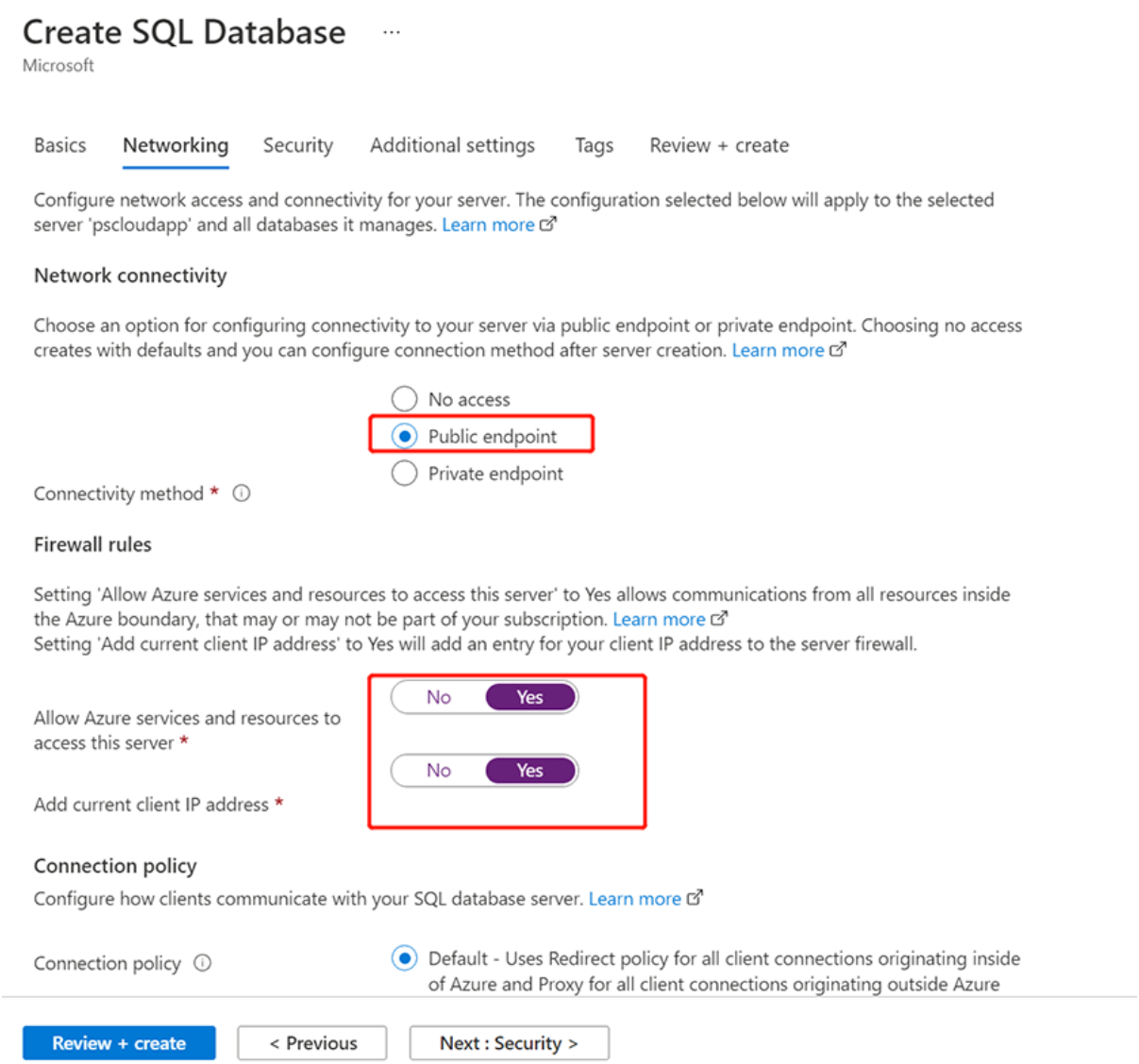


Step 5: On the **Networking** page, configure the following options:

1. For **Network connectivity**, select **Public endpoint**.

2. Set both of **Allow Azure services and resources to access this server** and **Add current client IP address** to **Yes**.

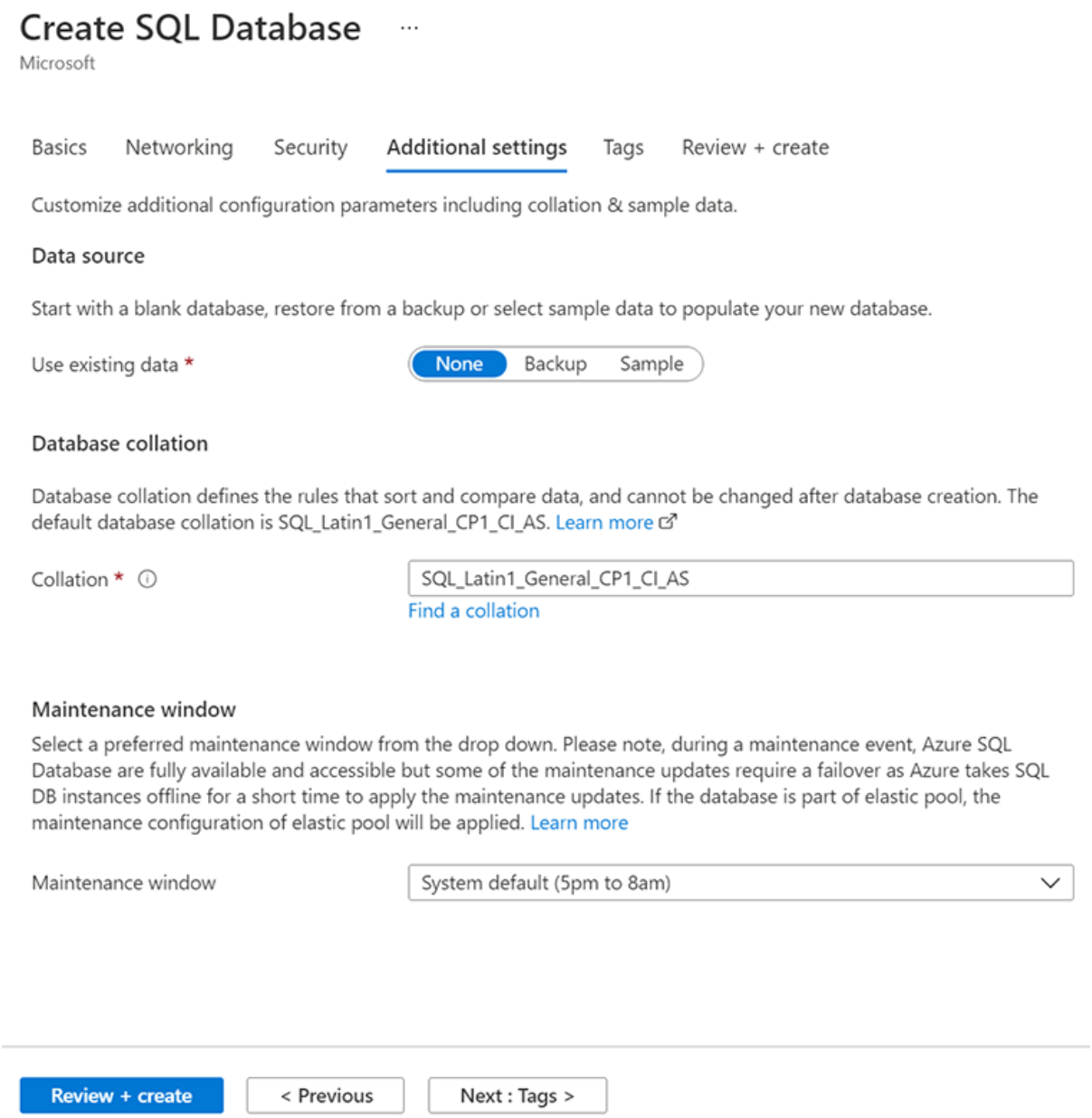
3. Select **Next: Security** at the bottom of the page.



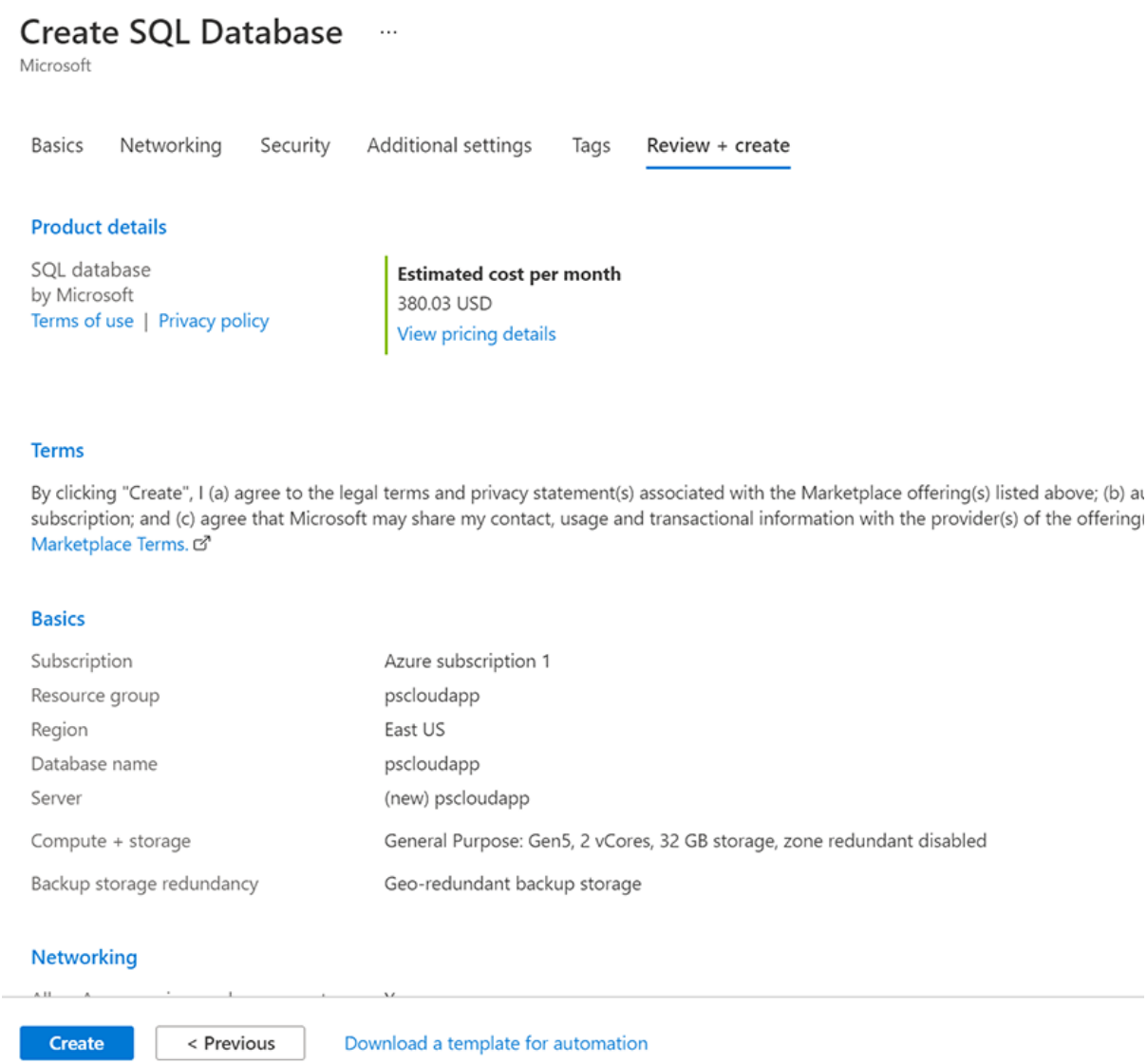
Step 6: Keep the default **Security** options. Click **Next: Additional settings** at the bottom of the screen.

Step 7: On the **Additional settings** page, in the **Data source** section, select whether to restore from a backup or select sample data or start with a blank database.

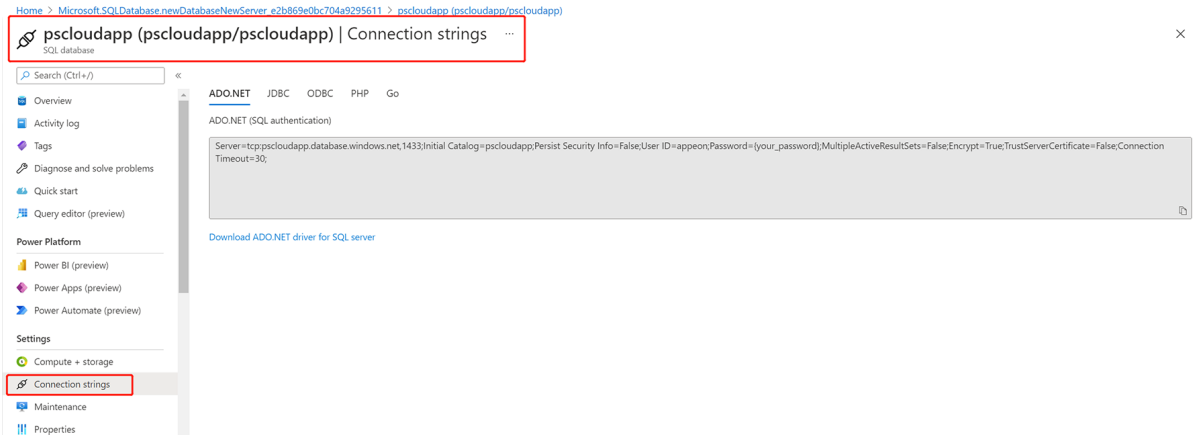
Step 8: Select **Review + create** at the bottom of the page.



Step 9: When validation completes, select **Create**.



After the database is created, you can view the connection strings.



11.4 Containerizing the installable cloud app

11.4.1 Preparing the application

The following modifications are made to the existing PowerServer project. If you have not created a PowerServer project yet, please follow the instructions in the Quick Start guide to create one.

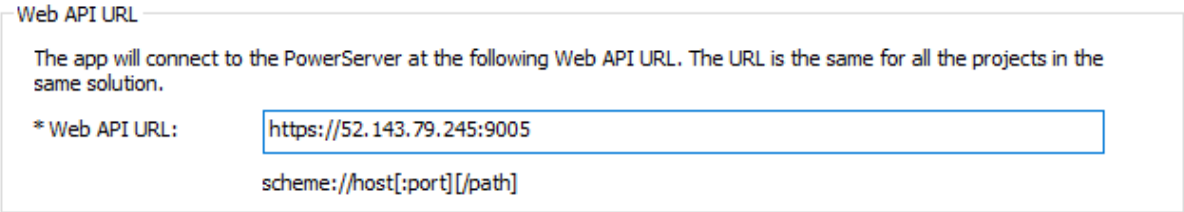
11.4.1.1 Modifying the Web API URL

You need to make sure the client app knows where to call the PowerServer Web APIs before containerizing the application (as the CustomizeDeploy.dll tool cannot be executed after containerized).

Go to the **Web APIs** tab of the PowerServer project painter, specify the URL of the PowerServer Web APIs, for example, **https**://demok8s.appeon.com:9005, or **https**://52.143.79.245:9005.

If you input the domain name (in this tutorial, demok8s.appeon.com) here, make sure the domain name is associated with the IP address. The IP address should be the Azure static public IP address (in this tutorial, 52.143.79.245) created in Creating public IP address.

The port number should be the same one specified later in the YAML manifest file that defines the Kubernetes pod for running the Web API docker image (in this tutorial, the pre-defined port number is 9005).



Notes:

1. Make sure the Kubernetes pod for the Web API will be run at the same domain name/IP address and port number later.

2. If the domain name/IP address and port number are changed later, you will need to modify the settings here and build the PowerServer project again in the PowerBuilder IDE.

11.4.1.2 Modifying the database connection

In Creating a database, you have already created a SQL Server database in Azure.

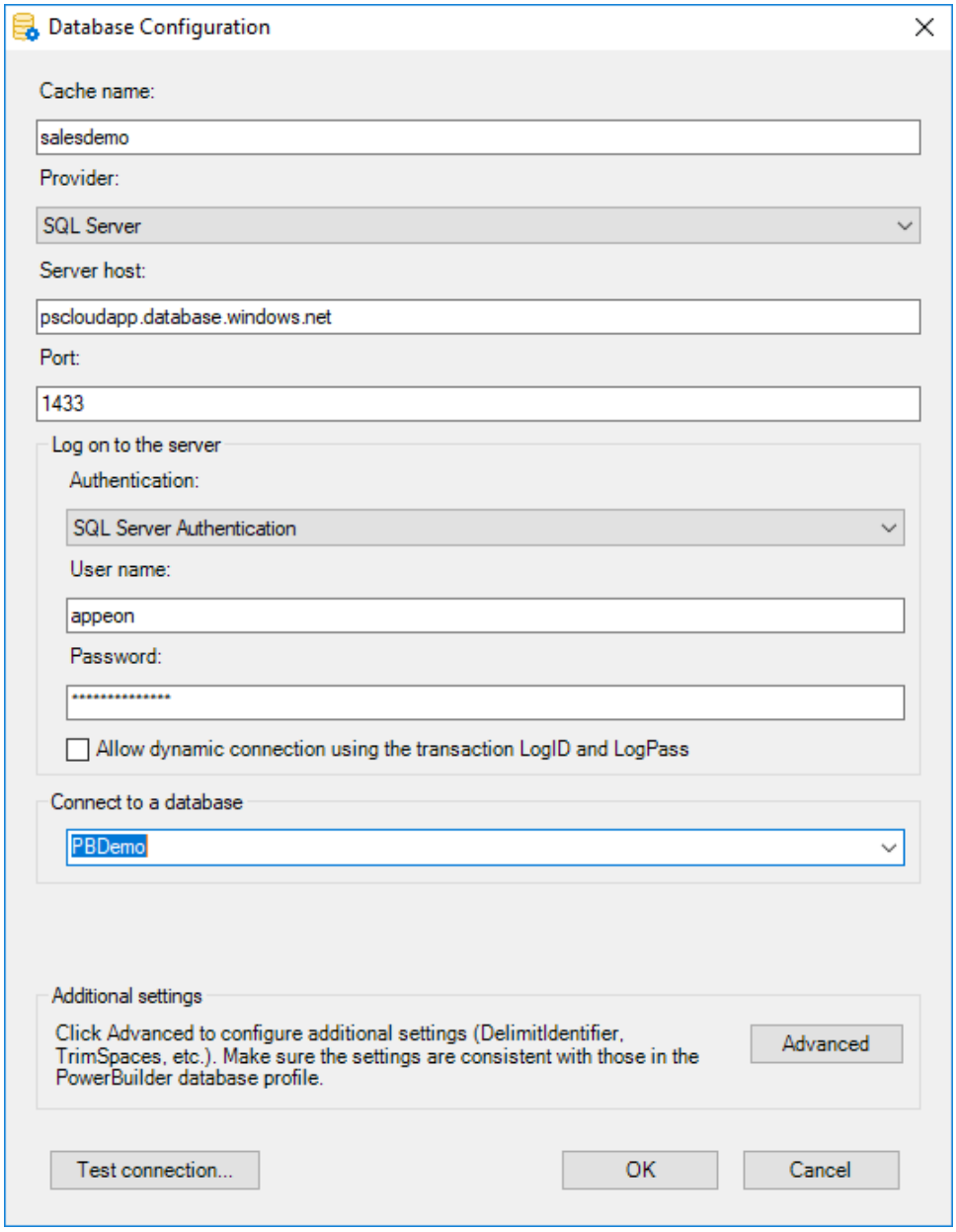
Now you will need to modify the database connection cache to point to this database created in Azure.

Step 1: At the bottom of the **Web APIs** tab of the PowerServer project painter, click the **Database Configuration** button.

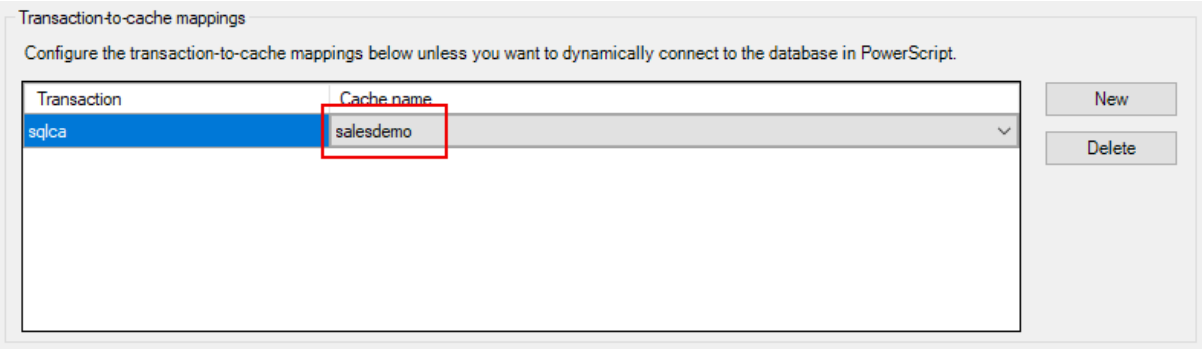
Step 2: In the **Database Configuration** window, click **DB Drivers** in the upper part to make sure the SQL Server driver and the option "I have read and agree to the license ..." both are selected.

Step 3: In the **Database Configuration** window, click **New** in the upper part to create a new connection cache.

Step 4: In the dialog box that displays, specify the database connection settings. For example, you specify the settings as shown below to connect to the Azure SQL database in Creating a database.



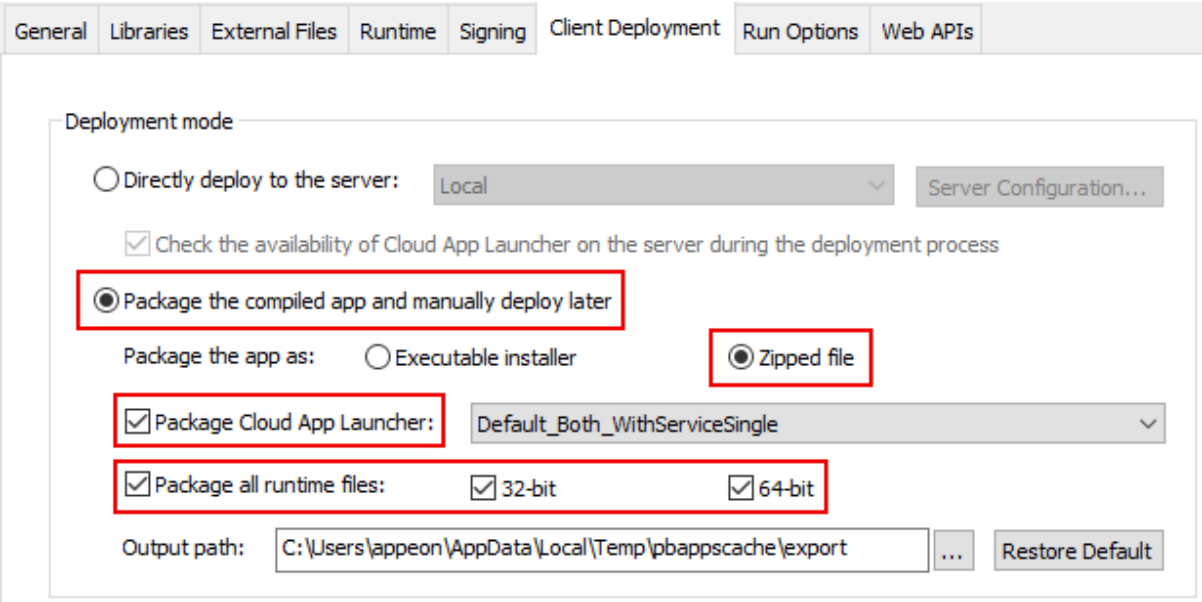
Step 5: When the cache is created successfully, make sure to select this new cache to map with the transaction object in the lower part of the **Database Configuration** dialog.



11.4.1.3 Packaging the client app as a zipped file

To deploy the client app to a Web server which runs as a docker container image, you will have to package the client app as a zipped file first and then manually deploy it to the image.

Go to the **Client Deployment** tab of the PowerServer project painter, and then click **Package the compiled app and manually deploy later**. Specify to generate the package as a compressed zip file, and select to package the cloud app launcher and the PowerBuilder Runtime files.



When the project is built in the next step, a zipped file of the client app will be generated.

11.4.1.4 Building the PowerServer project

After you made changes to the PowerServer project settings, save the project settings and then click the **Build & Deploy PowerServer Project** button in the toolbar.

When the build process completes, the following will be generated:

- a zipped file of the client app
- a C# solution of PowerServer Web APIs

They will be used to create the docker container images in the next step.

11.4.2 Creating the container images

You will need to create two container images: one contains the Web server and the client app, and the other contains the PowerServer Web APIs.

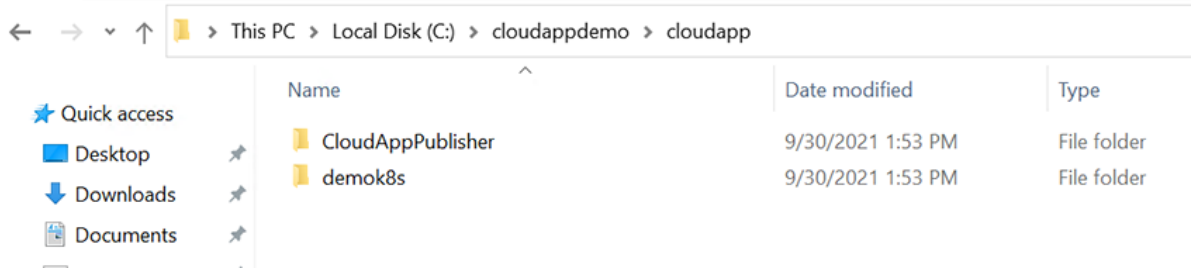
11.4.2.1 Creating an image for the client app

This is to create a Docker container image that contains the Web server and the client app.

This tutorial will show you how to create a container image using the Dockerfile.

Step 1: Extract the zipped file of the client app to the local directory, for example, C:\cloudappdemo\cloudapp.

(In the following screenshot, `demok8s` is the application name. Make a note of the application name, as it will be used in the application URL to access the application later.)



Step 2: Create a Dockerfile and input the following commands.

Example 1: the following commands get an Apache HTTP server image from the public repository and then add the client app to the web root of the Apache HTTP server.

```
FROM httpd:latest
COPY --chown=daemon:daemon "cloudapp/" "/usr/local/apache2/htdocs/"
```

Example 2: the following commands get an Nginx Web server image from the public repository and then add the client app to the web root of the Nginx Web server.

```
FROM nginx:latest
COPY --chown=nginx:nginx "cloudapp/" "/usr/share/nginx/html/"
```

Step 3: Place the Dockerfile to the local directory, for example, C:\cloudappdemo.

Step 4: Use the **docker build** command to create the image and tag it as **powerservercloudapp:001**.

The dot (.) in the middle of the command sets the location of the Dockerfile (in this case, the current directory).

```
cd C:\cloudappdemo
docker build . -t powerservercloudapp:001
```

```
C:\WINDOWS\system32>cd C:\cloudappdemo

C:\cloudappdemo>docker build . -t powerservercloudapp:001
[+] Building 3.2s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                                                    0.1s
 => => transferring dockerfile: 31B                                                                     0.0s
 => [internal] load .dockerignore                                                                       0.0s
 => => transferring context: 2B                                                                         0.0s
 => [internal] load metadata for docker.io/library/httpd:latest                                         2.9s
 => [auth] library/httpd:pull token for registry-1.docker.io                                            0.0s
 => [internal] load build context                                                                       0.1s
 => => transferring context: 15.06kB                                                                    0.0s
 => [1/2] FROM docker.io/library/httpd:latest@sha256:79f4b352524204e67ae00985b62ecc530a956b8700a8f31452eade457325  0.0s
 => CACHED [2/2] COPY --chown=daemon:daemon cloudapp/ /usr/local/apache2/htdocs/                        0.0s
 => exporting to image                                                                                  0.0s
 => => exporting layers                                                                                 0.0s
 => => writing image sha256:7882a56a83ea3b04a432edb14872b46378ca966b97ab15ce5ccd4bb9bd5649c8            0.0s
 => => naming to docker.io/library/powerservercloudapp:001                                              0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

C:\cloudappdemo>_
```

Step 5: After the image is created, use the **docker images** command to see the images.

```
C:\cloudappdemo>docker images
REPOSITORY                          TAG                                       IMAGE ID       CREATED          SIZE
powerservercloudapp                 001                                       7882a56a83ea   16 minutes ago   369MB
```

11.4.2.2 Creating an image for the Web API

This is to create a Docker container image that contains the PowerServer Web APIs.

This tutorial will show you how to build and publish the PowerServer Web APIs as a Docker container image in the SnapDevelop IDE.

Step 1: Open the PowerServer C# solution in SnapDevelop.

Click the **Open C# Solution in SnapDevelop** button in the toolbar to launch the PowerServer C# solution in SnapDevelop. Or go to the location where the PowerServer C# solution is generated; and double click **PowerServer_[appname].sln** to launch the solution in SnapDevelop.

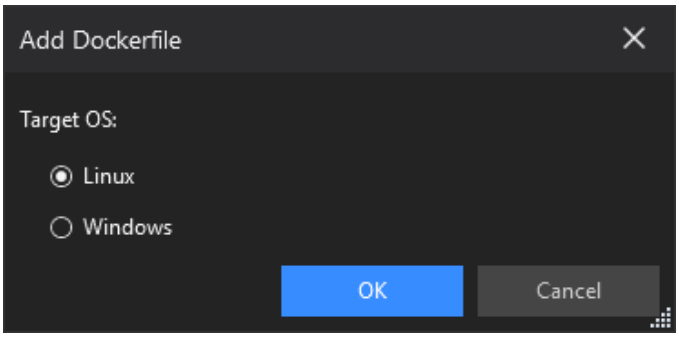
Step 2: Add docker support to the **ServerAPIs** project.

1. In the Solution Explorer, right click on the **ServerAPIs** project node, and select **Add** > **Docker Support**.

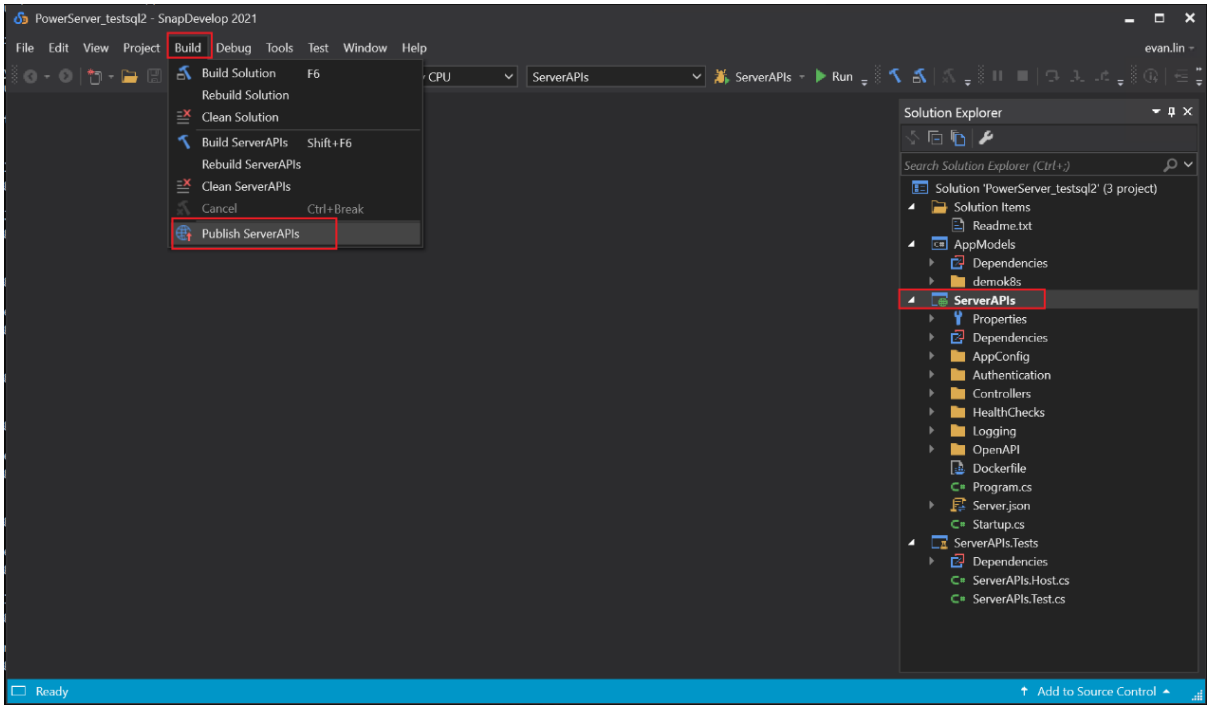
2. In the **Add Dockerfile** dialog, select the target OS: **Linux** or **Windows**, and click **OK**. The target OS indicates the platform where Docker Engine and Docker Container are running.

   A file named **Dockerfile** is automatically created according to the selected OS and added under the **ServerAPIs** project. This file contains all the commands required for building a docker image appropriate for the selected OS.

Figure 11.1:



Step 3: In the Solution Explorer, select the **ServerAPIs** project, and then select menu **Build** > **Publish ServerAPIs**.



Step 4: In the window that appears, select **Docker**, and then click **Start** to configure for publish.

a. Keep **Publish to Personal Repository** checked if you are connecting to your own repository. If the repository is owned by an organization, clear the checkbox, and enter the organization name.

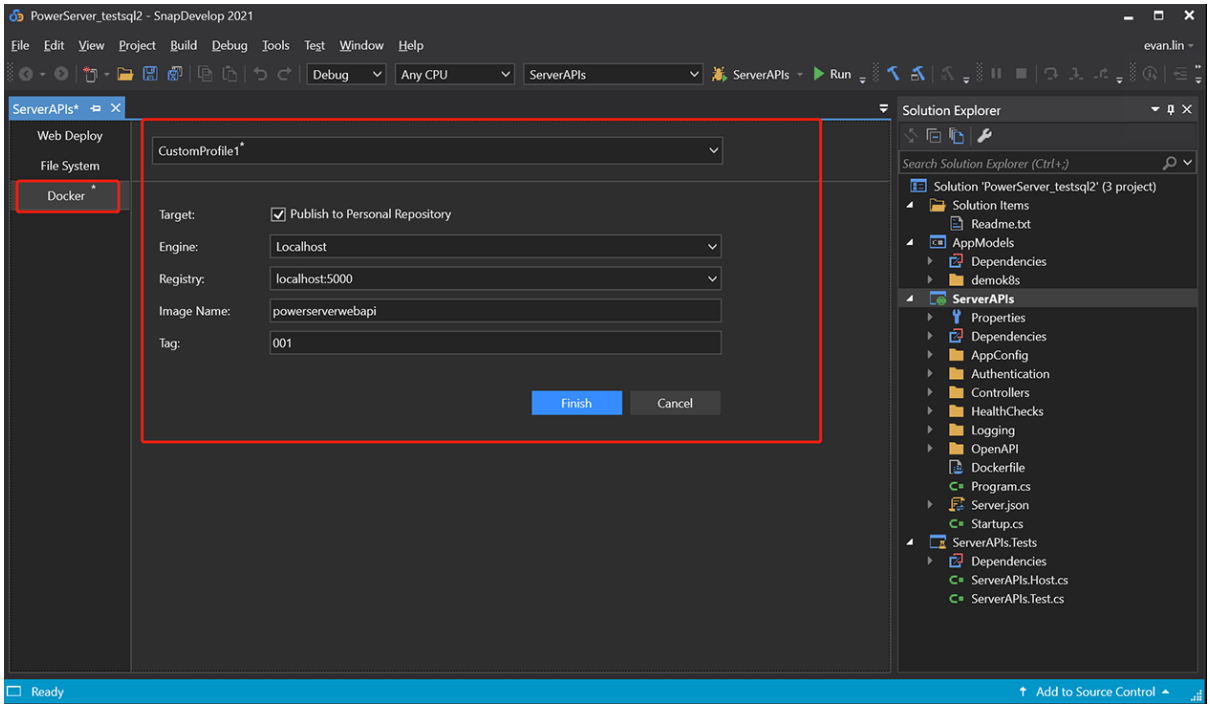
b. In the **Engine** field, select **localhost**.

c. In the **Registry** field, specify to store the container image in the local repository.

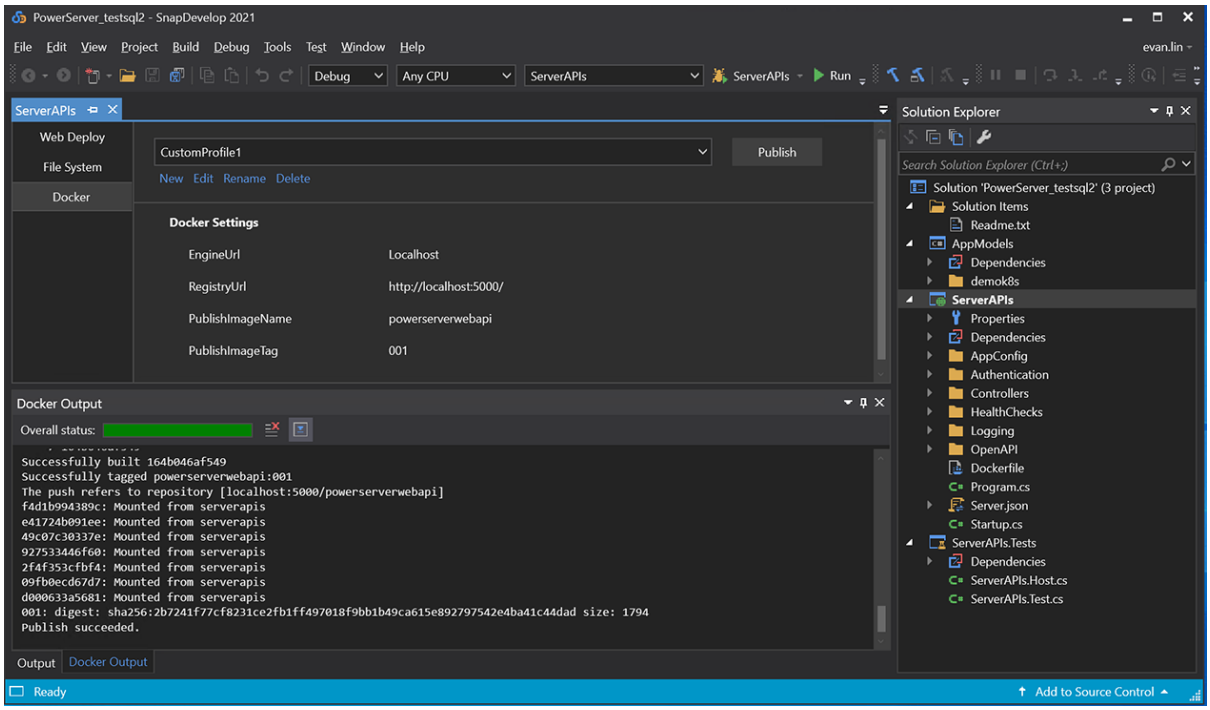
d. In the **Image Name** field, enter a name for the container image you want to create.

e. In the **Tag** field, enter a tag, for example, enter 001 to indicate the image version.

f. Click **Finish** to start building the project as an image and publishing the image to the specified Docker Engine and docker registry.



Step 5: Check the **Docker Output** window and make sure the publish is successful.



11.4.3 Pushing images to Azure container registry

To push container images to Azure container registry, you must first tag the image with the full name of the registry's login name. If you have not written down the login server name of the Azure container registry, follow instructions in Logging into Azure container registry to get it.

To push the image that contains the client app, run the following commands to tag the image with the registry's login name first and then push the image to the Azure container registry.

You can also add **:001** to the end of the image name to indicate the image version.

```
docker tag powerservercloudapp:001 pscloudapp.azurecr.io/powerservercloudapp:001
```

```
docker push pscloudapp.azurecr.io/powerservercloudapp:001
```

To push the image that contains the PowerServer Web APIs, run the following commands to tag the image with the registry's login name first and then push the image to the Azure container registry.

You can also add **:001** to the end of the image name to indicate the image version.

```
docker tag powerserverwebapi:001 pscloudapp.azurecr.io/powerserverwebapi:001
```

```
docker push pscloudapp.azurecr.io/powerserverwebapi:001
```

```
C:\cloudappdemo\publish>docker tag powerservercloudapp:001 pscloudapp.azurecr.io/powerservercloudapp:001

C:\cloudappdemo\publish>docker push pscloudapp.azurecr.io/powerservercloudapp:001
The push refers to repository [pscloudapp.azurecr.io/powerservercloudapp]
bfd9c7c8ee5e: Pushed
5a28409590bc: Pushed
07691779c08b: Pushed
5a4dfe5de0fd: Pushed
f78c692f3e8a: Pushed
476baebdfbf7: Pushed
001: digest: sha256:f474940477c912f34f852946d26d0877361d8e9a6f2db2dbbe565fdb3568f7b7 size: 1579

C:\cloudappdemo\publish>docker tag powerserverwebapi:001 pscloudapp.azurecr.io/powerserverwebapi:001

C:\cloudappdemo\publish>docker push pscloudapp.azurecr.io/powerserverwebapi:001
The push refers to repository [pscloudapp.azurecr.io/powerserverwebapi]
f4d1b994389c: Layer already exists
e41724b091ee: Layer already exists
49c07c30337e: Layer already exists
927533446f60: Layer already exists
2f4f353cfbf4: Layer already exists
09fb0ecd67d7: Layer already exists
d000633a5681: Layer already exists
001: digest: sha256:2b7241f77cf8231ce2fb1ff497018f9bb1b49ca615e892797542e4ba41c44dad size: 1794

C:\cloudappdemo\publish>_
```

11.5 Deploying the application to the Kubernetes cluster

Now that you have already containerized your PowerBuilder installable cloud application (the images that contain the client app and the PowerServer Web APIs have been created and pushed to the Azure container registry), you can deploy them to the Kubernetes cluster. The deployments tell Kubernetes how to create and update instances of your application. Once you have created a deployment, the Kubernetes control plane schedules the application instances included in that deployment to run on individual nodes in the cluster.

11.5.1 Creating the YAML manifest files

You can create a deployment by defining a manifest file in the YAML format. The manifest file defines a cluster's desired state, like which container images to run.

To create all the necessary pods, ingress, and services for running a PowerBuilder installable cloud app, you will need the following manifest files:

- deployment-pscloudapp.yml: This file defines a deployment of the pod that runs the client app.

- deployment-pswebapi.yml: This file defines a deployment of the pod that runs the PowerServer Web APIs.
- ingress-pscloudapp-appeon.com.yml: This file defines an ingress that sends the HTTP/HTTPS requests of the client app to the service.
- ingress-pswebapi-appeon.com.yml: This file defines an ingress that sends the HTTP/HTTPS requests of the PowerServer Web APIs to the service.
- service-pscloudapp.yml: This file exposes the pod running the client app as a Kubernetes service.
- service-pswebapi.yml: This file exposes the pod running the PowerServer Web APIs as a Kubernetes service.
- secret-env-connectstrings.yml: This file defines a secret that contains the sensitive data, environment variables etc. that can be used by the deployments.

You can use Visual Studio Code or a text editor to create and edit the YAML file.

The following sample files only provide the minimal required settings; you can modify the files according to your needs. You can change the file name as you like but keep the file extension as yaml or yml.

Create a manifest file named `deployment-pscloudapp.yml` and copy in the following example YAML:

- It defines a pod named `deployment-pscloudapp.`
- The pod runs the container of the client app and it pulls the container image from the Azure container registry: `pscloudapp.azurecr.io/powerservercloudapp:001.`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-pscloudapp
spec:
  selector:
    matchLabels:
      app: pscloudapp
  template:
    metadata:
      labels:
        app: pscloudapp
    spec:
      containers:
      - name: pscloudapp
        image: pscloudapp.azurecr.io/powerservercloudapp:001
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

Create a manifest file named `deployment-pswebapi.yml` and copy in the following example YAML:

- It defines a pod named `deployment-pswebapi.`
- The pod runs the container for the PowerServer Web APIs and it pulls the container image from the Azure container registry: `pscloudapp.azurecr.io/powerserverwebapi:001.`
- It uses the PowerServer license key and code from the secret `secret-env-connectionstrings`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-pswebapi
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pswebapi
  template:
    metadata:
      labels:
        app: pswebapi
    spec:
      containers:
      - name: pswebapi
        image: pscloudapp.azurecr.io/powerserverwebapi:001
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 9005
        env:
         - name: PowerServer__LicenseKey
           valueFrom:
             secretKeyRef:
               key: PowerServer__LicenseKey
               name: secret-env-connectionstrings
         - name: PowerServer__LicenseCode
           valueFrom:
             secretKeyRef:
               key: PowerServer__LicenseCode
               name: secret-env-connectionstrings
```

Create a manifest file named `ingress-pscloudapp-appeon.com.yml` and copy in the following example YAML:

- It defines an ingress named `ingress-pscloudapp-appeon.com` and a list of rules that match against the incoming requests and route the requests to the service.

  In the following example, requests to the host `pbexam.appeon.com` is routed to the service named `service-pscloudapp` (listening on port 80).

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-pscloudapp-appeon.com
  labels:
    name: ingress-pscloudapp-appeon.com
  annotations:
    kubernetes.io/ingress.class: nginx
```

```
spec:
  rules:
  - host: pbexam.appeon.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service-pscloudapp
            port:
              number: 80
```

Create a manifest file named `ingress-pswebapi-appeon.com.yml` and copy in the following example YAML:

- It defines an ingress named `ingress-pswebapi-appeon.com` and a list of rules that match against the incoming requests and route the requests to the service.

  In the following example, requests to the host `demok8s.appeon.com` is routed to the service named `service-pswebapi` (listening on port 9005).

- The port must be the same one that you specified in the PowerServer project settings > Web API URL in the PowerBuilder IDE (in this tutorial, 9005).

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-pswebapi-appeon.com
  labels:
    name: ingress-pswebapi-appeon.com
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: demok8s.appeon.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service-pswebapi
            port:
              number: 9005
```

Create a manifest file named `service-pscloudapp.yml` and copy in the following example YAML:

- It exposes the pod running the client app as a service so that it can be accessible from the public internet.

```
apiVersion: v1
kind: Service
metadata:
  name: service-pscloudapp
spec:
  selector:
    app: pscloudapp
  ports:
  - port: 80
    targetPort: 80
```

Create a manifest file named `service-pswebapi.yml` and copy in the following example YAML:

- It exposes the pod running the PowerServer Web APIs as a service so that it can be accessible from the public internet.

```
apiVersion: v1
kind: Service
metadata:
  name: service-pswebapi
spec:
  selector:
    app: pswebapi
  ports:
  - port: 9005
    targetPort: 9005
```

Create a manifest file named `secret-env-connectstrings.yml` and copy in the following example YAML:

- It defines the environment variables for the PowerServer license key and code, which makes it possible for you to update the PowerServer license key and code whenever necessary.

```
apiVersion: v1
stringData:
  PowerServer__LicenseKey: $YOURLICENSEKEY
  PowerServer__LicenseCode: $YOURLICENSECODE
kind: Secret
metadata:
  name: secret-env-connectionstrings
  namespace: default
type: Opaque
```

11.5.2 Deploying the application

Step 1: Place the YAML manifest files to the local deployment directory, for example, C:\cloudappdemo\deploy.

```
C:\cloudappdemo\deploy>dir
 Volume in drive C has no label.
 Volume Serial Number is BAC3-3B24

 Directory of C:\cloudappdemo\deploy

09/30/2021  01:34 PM    <DIR>          .
09/30/2021  01:34 PM    <DIR>          ..
09/30/2021  10:53 AM               511 deployment-pscloudapp.yml
09/30/2021  01:17 PM             2,779 deployment-pswebapi.yml
09/18/2021  04:39 PM               447 ingress-pscloudapp-appeon.com.yml
09/18/2021  02:51 PM               442 ingress-pswebapi-appeon.com.yml
09/30/2021  01:20 PM             1,977 secret-env-connectstrings.yml
09/18/2021  02:06 PM               157 service-pscloudapp.yml
09/18/2021  02:07 PM               153 service-pswebapi.yml
               7 File(s)          6,466 bytes
               2 Dir(s)  134,355,349,504 bytes free
```

Step 2: Deploy the installable cloud application through the manifest file.

```
kubectl apply -f .
```

This command parses the manifest files existing in the current directory and creates the Kubernetes pods, services, ingress, and secret.

Make sure the output shows that the resources are created successfully in the AKS cluster.

```
C:\cloudappdemo\deploy>kubectl apply -f .
deployment.apps/deployment-pscloudapp created
deployment.apps/deployment-pswebapi created
ingress.networking.k8s.io/ingress-pscloudapp-appeon.com created
ingress.networking.k8s.io/ingress-pswebapi-appeon.com created
secret/secret-env-connectionstrings created
service/service-pscloudapp created
service/service-pswebapi created

C:\cloudappdemo\deploy>_
```

Step 3: View the status of your containers.

```
kubectl get pods
```

```
C:\cloudappdemo\deploy>kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
deployment-pscloudapp-5695c865fd-ss9qg   1/1     Running   0          11m
deployment-pswebapi-9db55965b-bvqdn      1/1     Running   0          11m
```

11.5.3 Configuring the domain name

If you use the domain name in the YAML manifest file in the previous step, you will need to associate the domain names with the Azure static public IP address, for example,

pbexam.appeon.com 52.143.79.245

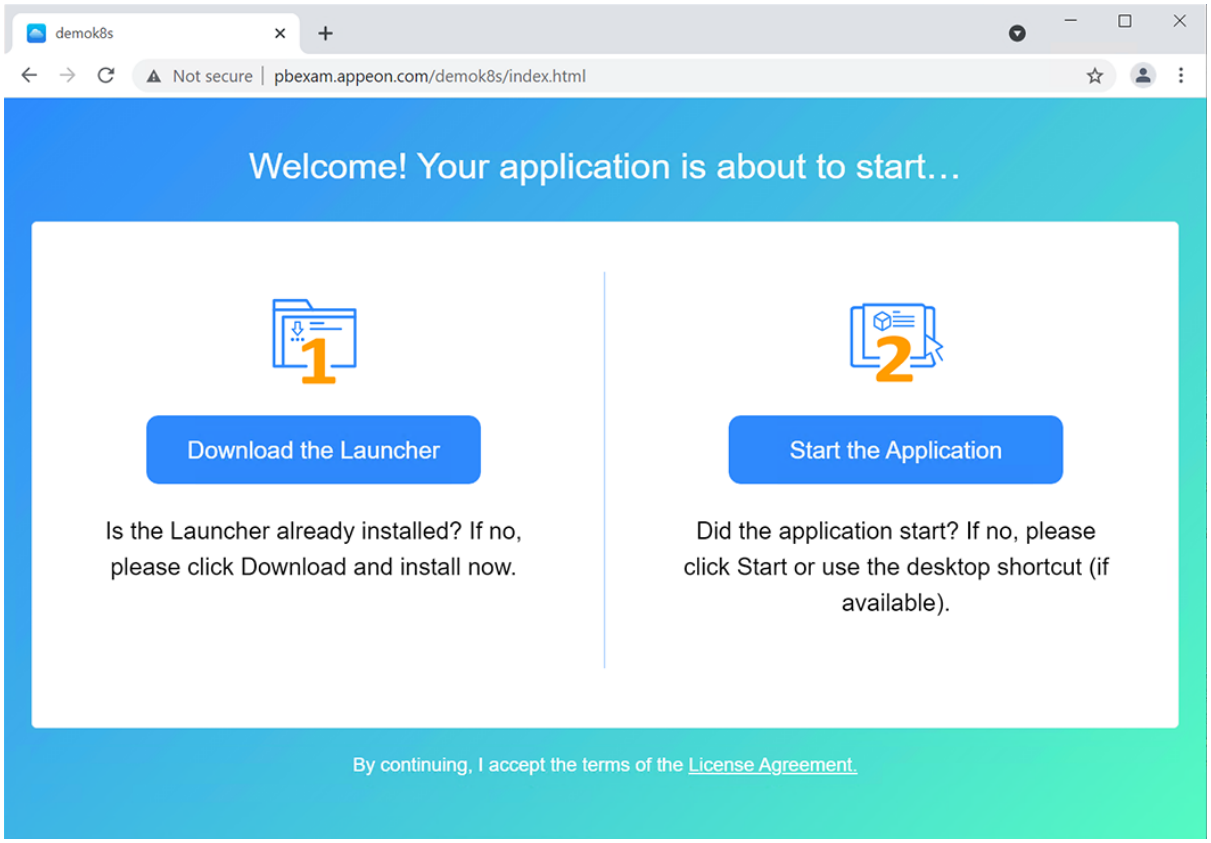
demok8s.appeon.com 52.143.79.245

11.5.4 Testing the application

Test the application by accessing the application URL in the Web browser, for example,

https://pbexam.appeon.com/demok8s

If the download does not start automatically, click **Download the Launcher** to download and install the cloud app launcher first, and then click **Start the Application** to download, install, and start the application.


demok8s
Not secure | pbexam.appeon.com/demok8s/index.html
Welcome! Your application is about to start…
Download the Launcher
Is the Launcher already installed? If no, please click Download and install now.
Start the Application
Did the application start? If no, please click Start or use the desktop shortcut (if available).
By continuing, I accept the terms of the License Agreement.

Working with Database Connections

Contents

1 Overview

1.1 Supported database connection options

A developer has several options for establishing database connections for installable cloud apps.

Table 1.1: Possible options to establish database connections

Option	Connection required for project compilation	Connection required for the app runtime
#1	Create the caches in the project settings > **Database Configuration** window. For more information, refer to Configuring database caches.	Dynamic database connection --- Specify which cache will be used by the transaction object in PowerScript. For more information, refer to Dynamically mapping transaction object with cache.
#2	Create the caches in the project settings > **Database Configuration** window. For more information, refer to Configuring database caches.	Static database connection --- Map the transaction object with the cache in the project settings > **Database Configuration** window. For more information, refer to Setting up static database connection for the app runtime.
#3	Create the caches in the project settings > **Database Configuration** window. For more information, refer to Configuring database caches.	Static database connection --- Directly specify the database connection information in PowerScript. For more information, refer to Making dynamic database connections from the app client.

With either of the options, you need to create the cache in the **Database Configuration** window, because it is required for project compilation: the cache information is necessary for converting the DataWindows to C# models.

If you have configured more than one options, the priority order to take the options is: #1 -> #2 -> #3. Option #3 is not recommended for the production environment because of security concerns. Different from the other two options (which stores the database connection information in PowerServer), #3 stores the database connection information at the app client and has higher risk of exposing the sensitive connection information.

As you can tell from the above table, during the app runtime, there are two ways to connect with the database: static database connection, or dynamic database connection.

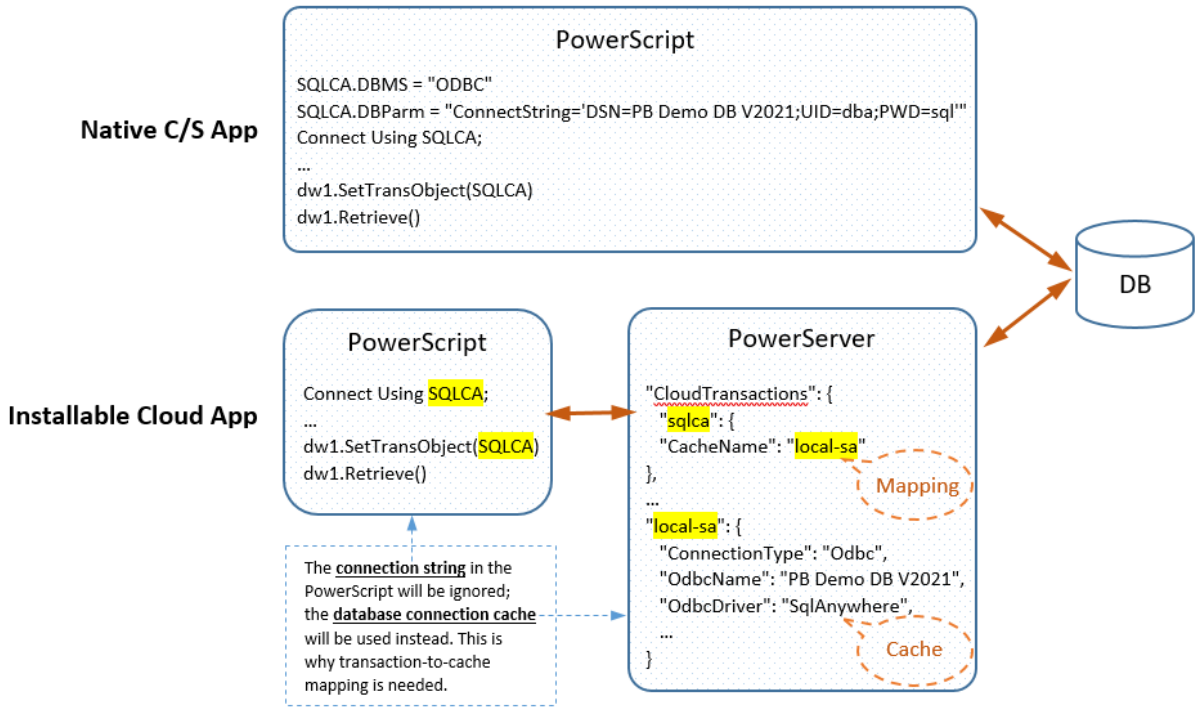
- "Static database connection" (option #2) means the connection configuration (including connection cache settings, and transaction-to-cache mappings) is created before the app is run. The connection configuration is initially created in the **Database Configuration** window and gets deployed to the PowerServer C# solution. Although you may further update the connection configuration in the solution, the configuration from the solution will be used during the app runtime.

- "Dynamic database connection" means the mapping is created when the app is run. A dynamic mapping can be created in the application scripts using the **DBParm CacheName** property (option #1) or using the transaction connections (option #3).

1.2 Comparing the runtime database connections between c/s app and installable cloud app

The following diagram shows you the comparison of runtime database connections between native c/s app and installable cloud app (using option #2):

Figure 1.1:



1.3 Techniques for supporting various connection scenarios

A few settings are available in the PowerServer project settings and also in the PowerServer solution to assist you to handle various connection scenarios. Specifically:

1. If as the app developer, you want to switch between different development, testing and even production environment for running the app.

   Technique: Configuring different DB connection profiles in the PowerServer project settings. If you define the database connections in multiple profiles, the PowerServer project will be compiled against all the configured profiles. When you run the PowerServer Web APIs, you can select the actual connection profile with which the Web APIs will run.

2. If as the app developer, you want to assign different database access, to different app users.

   Technique: Enabling the "Allow dynamic connection using the transaction LogID and LogPass" option in the cache settings. The LogID and LogPass may be unique to each app user. If you enable this option, the database connection will be set up according to the access permission associated with the LogID and LogPass.

3. If as the app distributor (or independent software vendor), you want to have the deployed app working in customer-specific database environment.

Technique: Configuring different cache groups (DBParm CacheGroup property)

You can define multiple database connection scenarios in the cache groups, through changing the PowerServer C# solution or dynamically calling the PowerServer APIs in PowerScript. Then, you can dynamically specify the CacheGroup value in DBParm, so that the deployed application will work in different database connection scenarios for different use cases.

4. If as the app administrator, you have updated the database environment and want to enable the Web APIs to work with the updated databases.

   Technique: Directly updating the connection settings in the Applications.json file

   The Applications.json file can be edited even after the Web APIs have been compiled. If the changes are minor, you can directly update the Applications.json file as the temporary solution.

2 Supported database types

You can create database connection caches for the following databases in the **Database Configuration** window or in the PowerServer C# solution:

- Adaptive Server Enterprise (ODBC) 16.0

  ASE databases can only be connected using the ODBC driver in the PowerServer runtime environment. This is different from the PowerBuilder runtime environment where the ASE database is connected using the native driver. See the next section for the differences caused by this driver change.

- Informix 12.x or 14 (Beta feature) *

  PowerBuilder and/or PowerServer will automatically download the required driver (IBM.Data.DB2.Core 2.2.0.100) from https://www.nuget.org, or you will be asked to specify the location of the driver if https://www.nuget.org cannot be connected.

  * Beta means the feature has not been fully tested, has known bugs, and does not receive standard technical support. We will collect reported bugs and try to address in a future version.

- MySQL 5.6, 5.7, or 8.0

  PowerBuilder and/or PowerServer will automatically download the required driver (MySql.Data 8.0.25) from https://www.nuget.org, or you will be asked to specify the location of the driver if https://www.nuget.org cannot be connected.

- Oracle 12c, 18c, or 19c

  PowerBuilder and/or PowerServer will automatically download the required driver (Oracle.ManagedDataAccess.Core 2.19.110) from https://www.nuget.org, or you will be asked to specify the location of the driver if https://www.nuget.org cannot be connected.

- PostgreSQL 11.3, 12, or 13

- SQL Anywhere (ODBC) 16 (16.0.0.2043 or later) or 17

  If SQL Anywhere is on a different machine from PowerBuilder, make sure to enable the connection pooling setting in the ODBC driver. Connection pooling is enabled by default if SQL Anywhere is on the same machine as PowerBuilder.

- SQL Server 2016, 2017, or 2019

SQL Anywhere and ASE databases can be connected using the ODBC driver only. The other databases are connected using the native database driver.

2.1 ASE database

If your application uses the ASE database, please notice that the drivers used in PowerBuilder and PowerServer are different. In PowerBuilder, the ASE native driver is used, while in PowerServer, the ODBC driver is used. Due to the driver difference, we have observed the following differences when running an installable cloud application against PowerServer:

- The ASE stored procedure might return different values because the default value of the "Set ANSI Null" option is different in these two drivers. (Read more)
- The data values of the SelectBlob variable are truncated in the installable cloud app, because the default value of the "Text size" option in the ODBC driver is 32KB. (Read more)
- Garbage letters display in the installable cloud app when retrieving multibyte data from the ASE database because DBParm does not support the "charset" parameter when using the ODBC driver. (Read more)

There might be other differences we haven't noticed yet. Please carefully examine the build and deploy process for any warnings or errors and fully test your application to make sure the data is correct.

3 Configuring database caches

3.1 Creating database caches in the project settings

It is required to create database caches in the project settings > **Database Configuration** window. The caches contain the database connection information for project compilation (PowerServer Toolkit will connect to the target database when converting the DataWindows to C# models), and will be deployed to the PowerServer solution and then be used for the database connection for the app runtime.

To create a database cache in the Database Configuration window:

1. Click the **Database Configuration** button at the bottom of the **Web APIs** tab.

2. In the **Database Configuration** window, you must select the required database driver and agree to the driver license terms as the driver must be downloaded from the NuGet site to the PowerServer C# solution.

   To select the required database driver:

   - Click **DB Drivers** in the **Database Configuration** window.

   - In the **Required Database Drivers** window, select the driver and the option "I have read and agree to the license ..."; and then click **OK**.

   Figure 3.1:

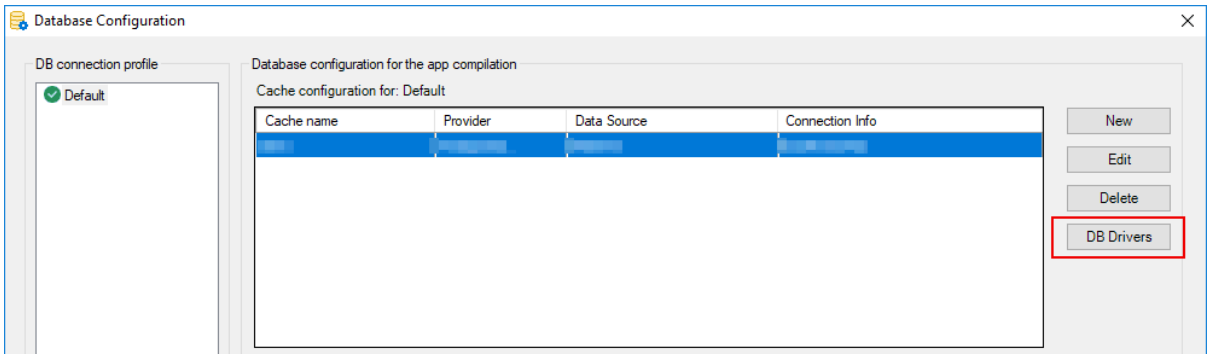
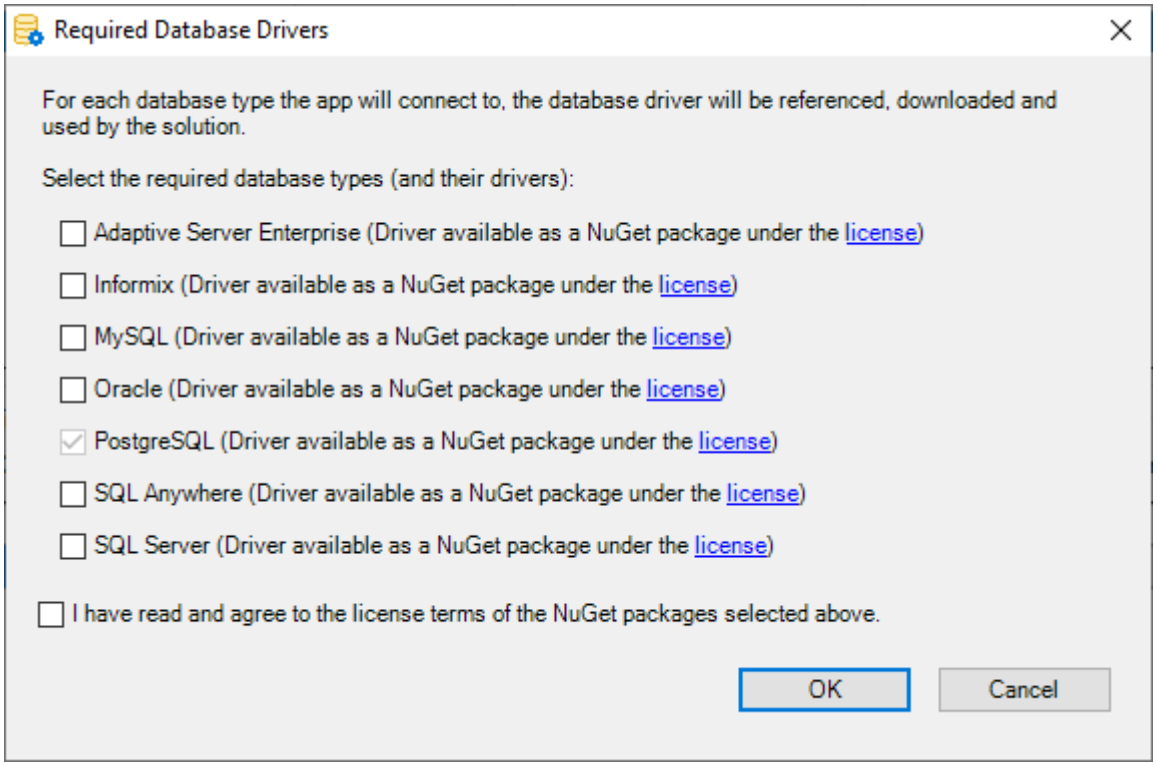


Figure 3.2:



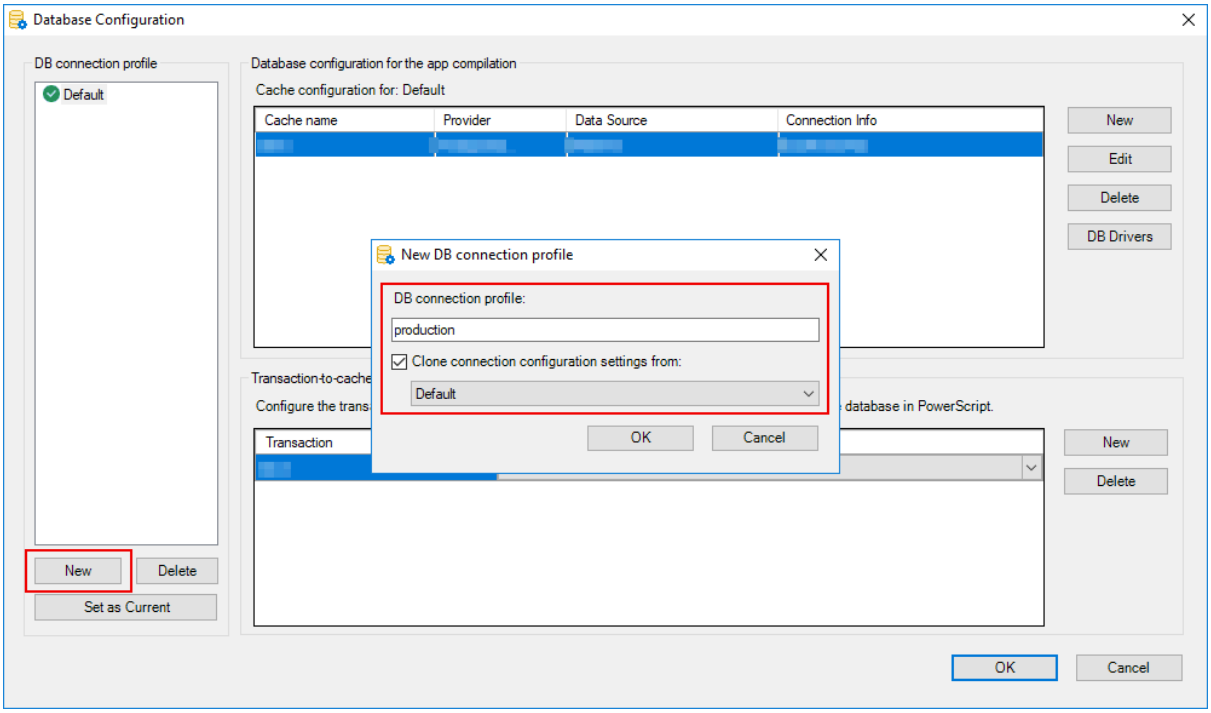
3. In the **Database Configuration** window, you can create different DB connection profiles to be used in different scenarios, for example, create different database connection profiles for the development environment, testing environment, production environment, etc.

   To create a new DB connection profile:

   - Click **New** in the **DB connection profile** group.
   - In the **New DB connection profile** dialog box, specify a name for the DB connection profile, for example, *production*.

     It is more efficient to create the new profile based on the settings of an existing one. You can select the clone option below and then select an existing profile to clone from.

Figure 3.3:



You can then decide which profile to be used in the application by selecting the DB connection profile and clicking the **Set as Current** button.

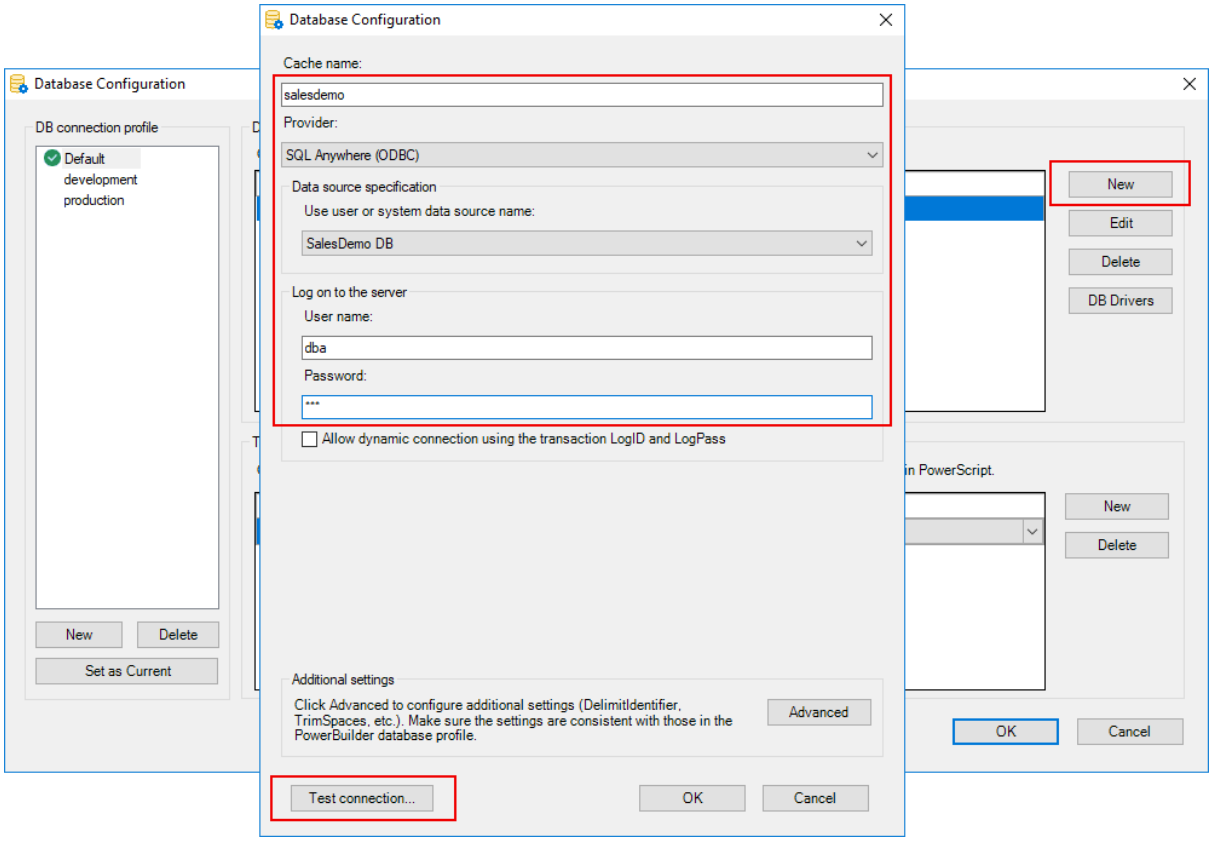
4. In the **Database Configuration** window, you can create the connection cache that connects with the database.

   For example, you can establish a connection with the SQL Anywhere database for the PowerBuilder demo using the following settings:

   - Click **New** in the upper part of the window.
   - In the dialog box that displays, specify any text as the cache name.
   - Select the database provider.
   - Select the data source.
   - Specify the user name (for example, dba) and password (for example, sql).
   - Click **Test Connection** to make sure the database can be connected successfully.

   The **Advanced** button contains additional important settings for the database driver such as DelimitIdentifier, TrimSpaces, etc. If your database has such settings, make sure to click the **Advanced** button to configure those settings.

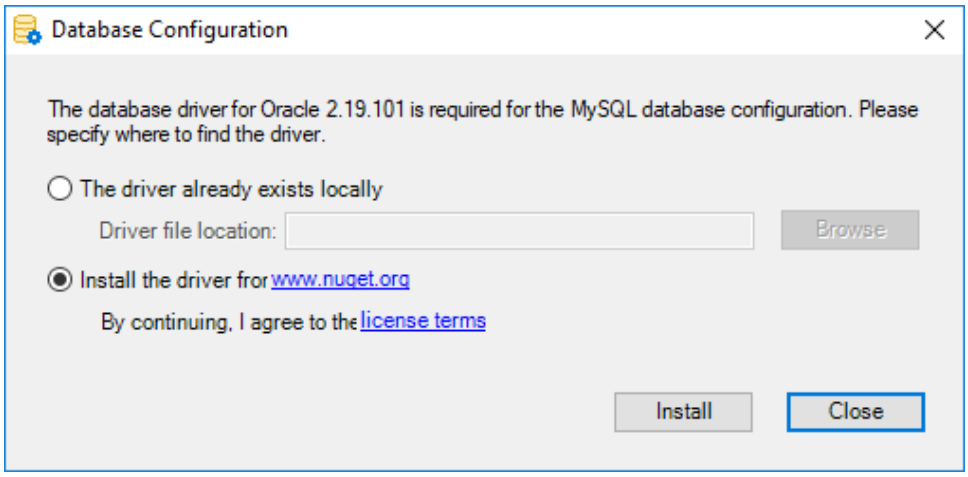
Figure 3.4:



If you select **MySQL**, **Oracle**, or **Informix** from the **Provider** listbox, you will be asked to specify a location for the required driver (MySql.Data 8.0.25, Oracle.ManagedDataAccess.Core 2.19.110, or IBM.Data.DB2.Core 2.2.0.100) or allow PowerBuilder to download and install the required driver from the NuGet website.

The packages downloaded from the NuGet website will be stored to %USERPROFILE %\.nuget\packages and cached in %USERPROFILE%\.sd\19.0\dbDrives\, so they can be automatically loaded when the MySQL or Oracle database connection is created.

Figure 3.5:



3.2 Managing database caches in the PowerServer solution

When the PowerServer project is built and deployed in the PowerBuilder IDE, the cache settings (including database server host/port, database name, login ID, password, advanced settings etc.) configured in the **Database Configuration** window will be deployed and stored in the PowerServer C# solution. You can manually change these settings in the PowerServer C# solution.

To manage database caches in the ServerAPIs project of the PowerServer solution:

1. Open the PowerServer C# solution > **ServerAPIs** project > **AppConfig** > **Applications.json** file.

   The Applications.json file contains the configuration of the "Default" DB connection profile. If you have another connection profile, the profile name is added in the middle of the file name. For example, Applications.Development.json file contains the configuration of the "Development" DB connection profile.

2. In the **Applications.json** file, locate the "Connections" block. This is where the cache(s) is stored.

   In the following example, there are two caches "local-sa" and "local-postgresql" under the "Default" cache group; and each cache contains the database connection information that are configured and deployed from the Database Configuration window. You can modify the existing cache, or create a new cache by making a copy of the existing one.

```
...
  "Connections": {
    "Default": {
      "local-sa": {
        "ConnectionType": "Odbc",
        "OdbcName": "PB Demo DB V2021",
        "OdbcDriver": "SqlAnywhere",
        "UserID": "dba",
        "Password":
 "eyJQYXlsb2FkIjoiYlx1MDAyQkxocTNiMUtWSzhBY1FCbVltU0FBPT0iLCJUaW1lc3RhbXAiOjE2MjU2NDYwNDcsIlN
        "CommandTimeout": 30,
        "OtherOptions": "",
        "DynamicConnection": false
      },
      "local-postgresql": {
        "ConnectionType": "PostgreSql",
        ...
      }
    }
  }
```

   Note: (1) The PowerServer C# solution will be updated every time when the PowerServer project is built and deployed in the PowerBuilder IDE. If you manually modify the settings in **Applications.json**, and want to keep these changes, you should use the "Overwrite server settings (DB connection, Web API port, and license)" option properly. For more information, refer to What settings will be deployed to the solution. (2) If you want to change the database driver, you must make changes in the project settings and then re-deploy the project from the PowerBuilder IDE. Changing the driver directly in the PowerServer solution would cause failure in the running of the installable cloud app.

4 Setting up static database connection for the app runtime

Static database connection means the connection configuration (including connection cache settings, and transaction-to-cache mappings) is created before the app is run. The connection configuration is initially created in the **Database Configuration** window and gets deployed to the PowerServer C# solution. Although you may further update the connection configuration in the solution, the configuration from the solution will be used during the app runtime.

4.1 Creating transaction-to-cache mappings in the project settings

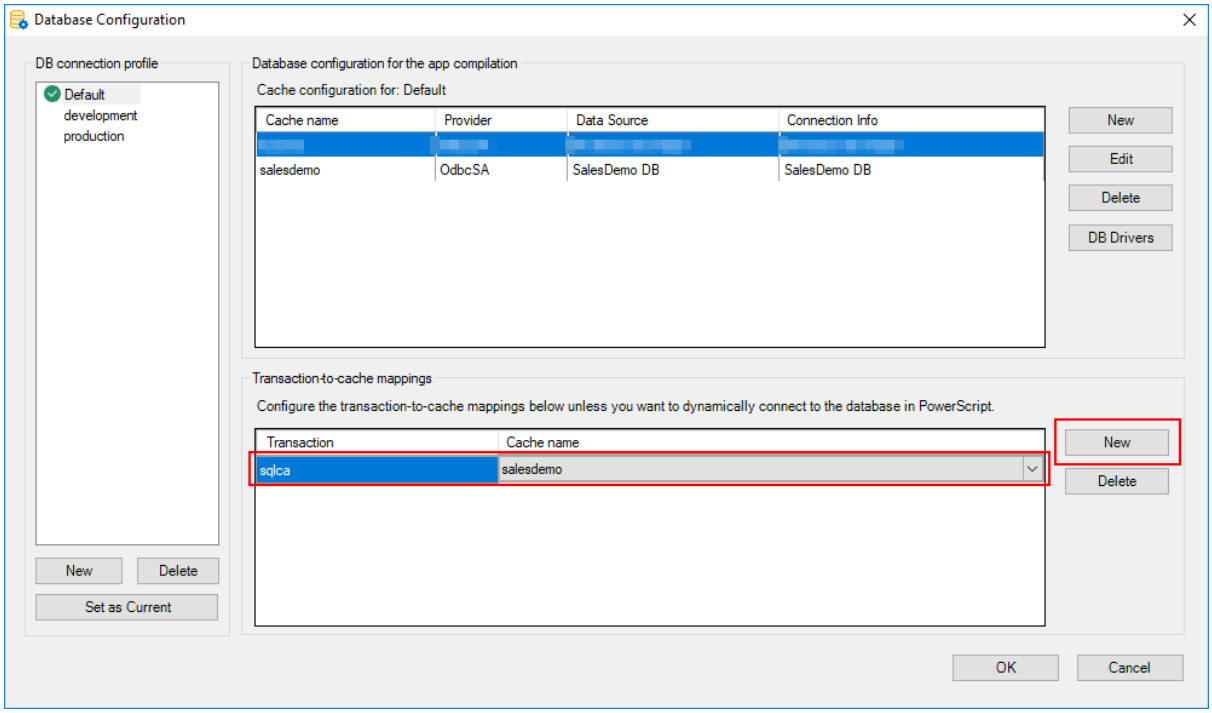
After the database cache is created in the **Database Configuration** dialog, for each transaction object that already exists in the application, map it with the cache in the **transaction-to-cache mappings** section. The mapping will be deployed to the PowerServer solution, and then be used to set up static database connection for the app runtime. Note that you only need to map the transaction objects that already exist in the PowerBuilder application.

To configure the mapping of the transaction object with the cache:

1. Click the **Database Configuration** button at the bottom of the **Web APIs** tab.

2. In the Transaction-to-cache mappings section, you can input the transaction object name (for example "sqlca") and then select one of the configured caches to map with.

Figure 4.1:



4.2 Managing transaction-to-cache mappings in the PowerServer solution

When the PowerServer project is built and deployed in the PowerBuilder IDE, the transaction-to-cache mappings configured in the Database Configuration window will be deployed and stored in PowerServer. You can manually change these settings in the PowerServer C# solution.

To manage the transaction-to-cache mappings in the ServerAPIs project:

1. Open the PowerServer C# solution > **ServerAPIs** project > **AppConfig** > **Applications.json** file.

   The Applications.json file contains the configuration of the "Default" DB connection profile. If you have another connection profile, the profile name is added in the middle of the file name. For example, Applications.Development.json file contains the configuration of the "Development" DB connection profile.

2. In the **Applications.json** file, locate the "Applications" block > [application name] > "CloudTransactions". This is where the transaction-to-cache mapping(s) is stored.

   In the following example, the "sqlca" transaction object is mapped to the "salesdemo" database cache. You can modify the existing mapping, or create a new mapping by making a copy of the existing one.

   ```
   "Applications": {
     "pssales": {
       "CloudTransactions": {
         "sqlca": {
           "CacheName": "salesdemo"
         }
       },
       ...
   ```

   Note: The PowerServer C# solution will be updated every time when the PowerServer project is built and deployed in the PowerBuilder IDE. If you manually modify the settings in **Applications.json**, and want to keep these changes, you should use the "Overwrite server settings (DB connection, Web API port, and license)" option properly. For more information, refer to What settings will be deployed to the solution.

4.3 Using LogID and LogPass properties

In case of static database connection, if the "Allow dynamic connection using the transaction LogID and LogPass" option (equivalent to the "DynamicConnection" setting) in the database cache is enabled, the application will use the LogID and LogPass property values (as shown in the example below) of the Transaction object to log in to the database server (instead of using the values in the User name and Password fields of the cache). Then the installable cloud app can connect to the database based on the user credentials provided at runtime.

```
Transaction.LogId = "sa"
Transaction.LogPass = "Appeon123!@#"
```

5 Setting up dynamic database connection for the app runtime

5.1 Dynamically mapping transaction object with cache using DBParm

Besides statically mapping the transaction object with the database cache in the **Database Configuration** window, for each transaction object that already exists in the application, you can also dynamically map it with the database cache using the **DBParm** **CacheName** property. Such dynamic mapping with DBParm has priority over the static mapping if both exist.

For example,

```
Sqlca.dbparm="cachename='Test'"
```

With the possibility of dynamically mapping a transaction object with the cache in the application scripts, you can create multiple caches which connect to the database with different privileges. When a user logs in, the application decides which cache should be used by the transaction object for establishing the database connection.

5.1.1 Using CacheGroup property in DBParm

The **DBParm** **CacheGroup** property is added for specifying the cache group to be used by the installable cloud app. You can define multiple database connection scenarios in the cache groups, and then dynamically specify the CacheGroup value in DBParm, so that the deployed application will work in different database connection scenarios for different use cases.

The database caches you create in the project settings all belong to the "default" cache group. You can create new cache groups. There are two ways to do it:

1. Add the new cache group in the PowerServer C# solution (in the **ServerAPIs** project > **AppConfig** > **Applications.json** file). You can create a new group by making a copy of the "default" group and then modify or add the cache in the new group. The cache group you created will be preserved every time when you build and deploy the PowerServer project (only the "default' cache group may be updated by deployment).

2. Add the new cache group dynamically in PowerScript by calling the relevant PowerServer APIs (refer to Managing database connections using PowerServer APIs.)

The example below shows you how to add cache groups in the PowerServer C# solution:

```
 "Connections": {
    "default": {
    ...
    },
    "cachegroup1": {
      "dbcache1": {
        "ConnectionType": "Odbc",
        "OdbcName": "sa-db1",
        "OdbcDriver": "SqlAnywhere",
        "UserID": "dba",
        "Password": "...",
```

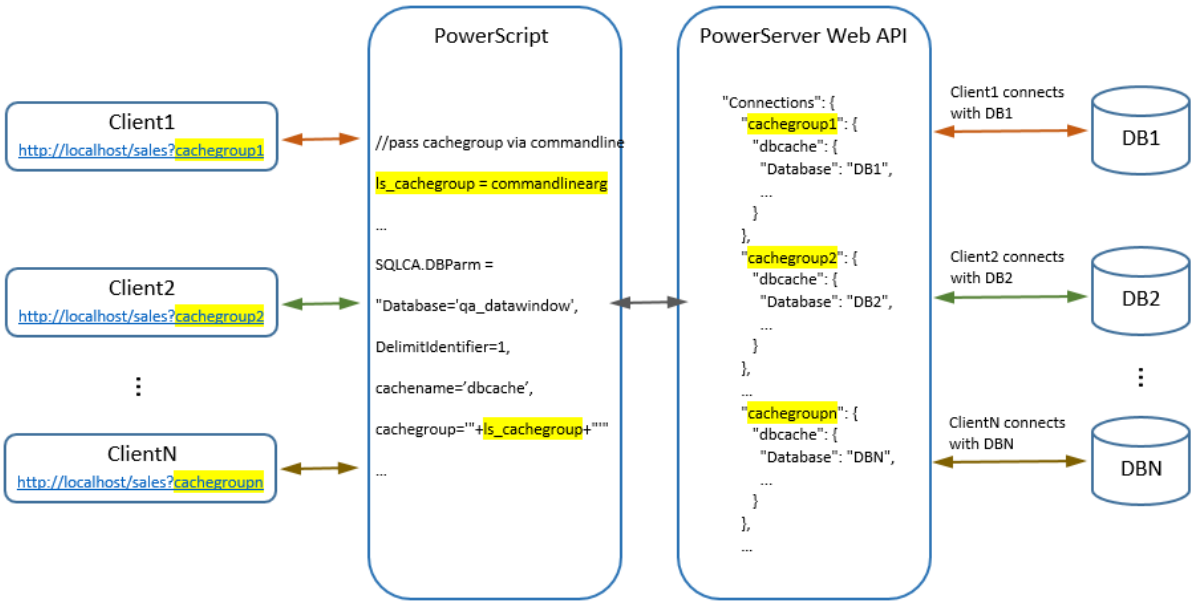
```
        ...
      },
      "dbcache2": {
        "ConnectionType": "PostgreSql",
        "Database": "pgs-db1",
        "Host": "172.16.100.33",
        "Port": 5432,
        "UserID": "postgres",
        "Password": "...",
        ...
      },
      ...
    },
    "cachegroup2": {
      "dbcache1": {
        "ConnectionType": "Odbc",
        "OdbcName": "sa-db2",
        "OdbcDriver": "SqlAnywhere",
        "UserID": "dba",
        "Password": "...",
        ...
      },
      "dbcache2": {
        "ConnectionType": "PostgreSql",
        "Database": "pgs-db2",
        "Host": "172.16.100.89",
        "Port": 5432,
        "UserID": "postgres",
        "Password": "...",
        ...
      },
      ...
    },
    ...
  }
```

Then you can modify the application scripts to use the CacheGroup property:

```
//pass cachegroup from commandline
ls_cachegroup = commandlinearg
if len(ls_cachegroup) = 0 then
   ls_cachegroup = "default"
end if

//db connection info used by PowerBuilder native c/s app
SQLCA.DBMS = "SNC SQL Native Client(OLE DB)"
SQLCA.ServerName = "localhost"
SQLCA.AutoCommit = true

//if "DynamicConnection" is true in the cache, the LogID and LogPass property
 values will be used to log in to
//the database server, instead of using the values specified in the cache.
SQLCA.LogPass = "mypass"
SQLCA.LogId = "mylog"

//cache and cachegroup used by PowerServer
SQLCA.DBParm =
 "Database='qa_datawindow',DelimitIdentifier=1,cachename='dbcache',cachegroup='"+ls_cachegroup
+"'"
connect;

if sqlca.sqlcode <> 0 then
   messagebox("Database Error",sqlca.sqlerrtext)
   return
```

```
end if
```

Figure 5.1:



5.1.2 Using LogID and LogPass properties

In case of dynamic database connection using DBParm, if the "Allow dynamic connection using the transaction LogID and LogPass" option (equivalent to the "DynamicConnection" setting) in the database cache is enabled, the application will use the LogID and LogPass property values (as shown in the example below) of the Transaction object to log in to the database server (instead of using the values in the User name and Password fields of the cache). Then the installable cloud app can connect to the database based on the user credentials provided at runtime.

```
Transaction.LogId = "sa"
Transaction.LogPass = "Appeon123!@#"
```

5.2 Making dynamic database connections from the app client

If there is no transaction-to-cache mapping configured for the app (either statically or dynamically, as explained in Setting up static database connection for the app runtime and Dynamically mapping transaction object with cache), you can make direct connections with the following databases from the application client.

Instead of storing the connection settings in the PowerServer solution, the connection info is stored at the client side, in the script, or in the application INI files. Saving sensitive information at the client is not recommended because of security concerns. Therefore, this approach is not recommended.

- MS SQL Server (through Native Client, OLE DB, ADO.NET)
- Oracle
- SQL Anywhere (through ODBC)
- Adaptive Server Enterprise (through ODBC)

MS SQL Server through Native Client:

```
SQLCA.DBMS = "SNC SQL Native Client(OLE DB)"
SQLCA.LogPass = "Appeon123!@#"
SQLCA.ServerName = "172.16.3.243"
SQLCA.LogId = "sa"
SQLCA.AutoCommit = False
SQLCA.DBParm = "Database='qa_datawindow'"
```

MS SQL Server through OLE DB:

```
SQLCA.DBMS = "OLE DB"
SQLCA.LogPass = "Appeon123!@#"
SQLCA.LogId = "sa"
SQLCA.AutoCommit = False
SQLCA.DBParm =
 "PROVIDER='SQLOLEDB',DATASOURCE='172.16.3.243',PROVIDERSTRING='database=qa_datawindow'"
```

MS SQL Server through ADO.NET:

```
SQLCA.DBMS = "ADO.Net"
SQLCA.LogPass = "Appeon123!@#"
SQLCA.LogId = "sa"
SQLCA.AutoCommit = False
SQLCA.DBParm =
 "Namespace='System.Data.SqlClient',DataSource='172.16.3.243',Database='qa_datawindow'"
```

Oracle:

```
SQLCA.DBMS = "ORA Oracle"
SQLCA.LogPass = "appeon"
SQLCA.ServerName = "172.16.3.98/pdborcl" //servername must point to a remote
 instance; cannot be local.
SQLCA.LogId = "DBO"
SQLCA.AutoCommit = False
SQLCA.DBParm = "DisableBind=1"
SQLCA.DBParm = "TableCriteria='DBO',DisableBind=1"
```

SQL Anywhere through ODBC:

```
SQLCA.DBMS = "ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm = "ConnectString='DSN=PB Demo DB
 V2021;UID=dba;PWD=sql',driver='SqlAnywhere'"
```

Adaptive Server Enterprise through ODBC:

```
SQLCA.DBMS = "ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm =
 "ConnectString='DSN=en_ase1253;UID=en_ase1253;PWD=en_ase1253',driver='ase'"
```

6 Managing database connections using PowerServer APIs

There are a number of connection-related PowerServer APIs for you to manage the database connections or view the connection status during runtime. You can find the list of APIs in the Controllers > ConnectionController.cs file in the ServerAPIs project:

- api/connection/loadone: Loads the configuration of a given connection;
- api/connection/loadgroup: Loads the configuration of a given CacheGroup;
- api/connection/loadall: Loads all the connection configuration;
- api/connection/addone: Adds a connection configuration;
- api/connection/addrange: Adds a group of connection configuration;
- api/connection/addgroup: Adds a group of empty connection configuration, and copies connection configuration list from the specified CacheGroup;
- api/connection/edit: Edits a connection configuration;
- api/connection/removeone: Removes a connection configuration;
- api/connection/removegroup: Removes a CacheGroup and the connection configuration in it.

For details on how each of the APIs is defined, please check the ConnectionController.cs file. For documentations, refer to View the API documentation.The following example shows you how to call the api/connection/loadall API in PowerScript to get all the current connection configuration:

```
//-------------------------------------
loadall----------------------------------------
httpclient  lhc_client
string ls_url
string ls_json

lhc_client = create httpclient

//Load all connection
ls_url = "http://localhost:5000/api/connection/loadall"
//This URL should be replaced with the actual IP address and port number of
 PowerServer Web APIs
//If there are multiple .NET servers, obtain one by one
//lhc_client.SetRequestHeader("Authorization", $token, true)  //If authorization is
 enabled
lhc_client.sendrequest("Get",ls_url)

if lhc_client.getresponsestatuscode() = 200 then
                lhc_client.getresponsebody(ls_json)
                //parse the json
                wf_getsessions(ls_json)
end if

//-----------------------------------------------------------------------------------------
```

And the response is like below:


```
[{"cachegroup":"Default","items":[{"cachename":"ora","configuration":
{"connectiontype":2,"host":"172.16.3.98","port":1521,"odbcname":"","odbcdriver":"","userid":"dbo"
{"cachename":"Test classes","configuration":
{"connectiontype":5,"host":"172.16.9.52","port":5432,"odbcname":"","odbcdriver":"","userid":"post
{"cachename":"Test123","configuration":
{"connectiontype":0,"host":"172.16.9.52","port":5432,"odbcname":"","odbcdriver":"","userid":"post
{"cachegroup":"Developer","items":[{"cachename":"sql","configuration":
{"connectiontype":0,"host":"172.16.3.243","port":1433,"odbcname":"","odbcdriver":"","userid":"sa"
@#","database":"Qa_datawindow","enablepooling":true,"minpoolsize":0,"maxpoolsize":100,"connection
```

Unsupported Features & Workarounds Guide

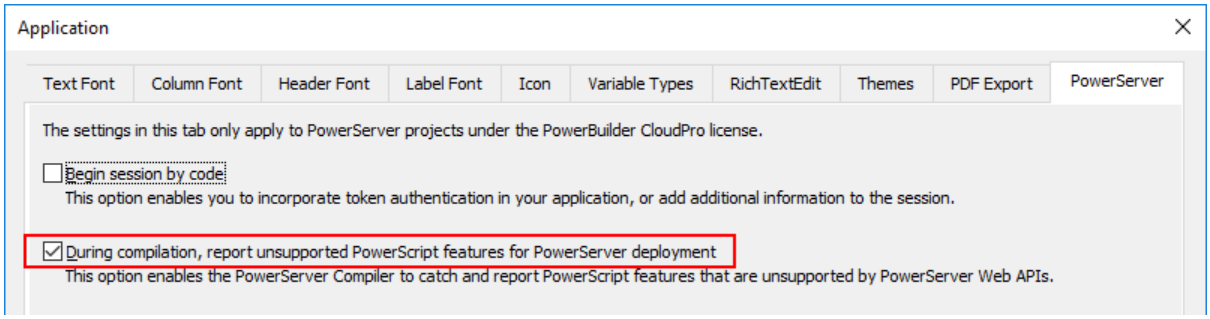
Contents

1 How to detect unsupported features

The PowerScript features will be analyzed during the build & deploy process; and if any feature is detected to be unsupported by the PowerServer Web APIs, it will be reported as an unsupported feature in the Output window. Please note that not every unsupported feature can be detected and listed, therefore, it is strongly recommended that you go through the unsupported features/scenarios and discrepancies documented in this guide and make sure they do not exist in your application.

The unsupported feature analysis option is disabled by default. To enable this option, open the Application painter's **Properties** view, and select the option "During compilation, report unsupported PowerScript features for PowerServer deployment" in the **PowerServer** tab page.

Figure 1.1:



With this option selected, the scripts will be analyzed for unsupported features

- when the **Build & Deploy PowerServer Project** option is selected.

  If any unsupported feature is detected, it will be displayed in the Output window | Unsupported (DWs) tab or Unsupported (PowerServer) tab.

  You can double-click the line or select Edit or Edit Source from the pop-up menu to open the object in the painter or source editor.

  Figure 1.2:

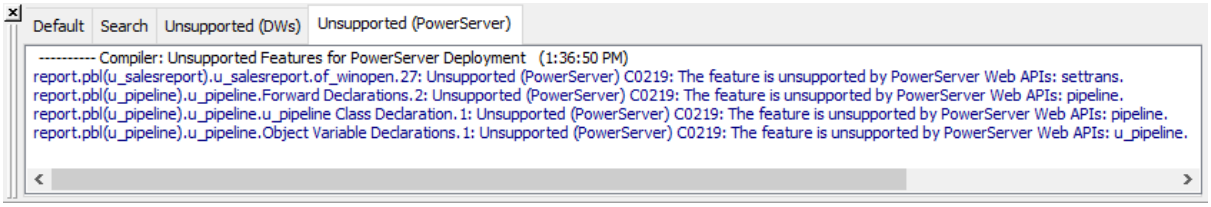
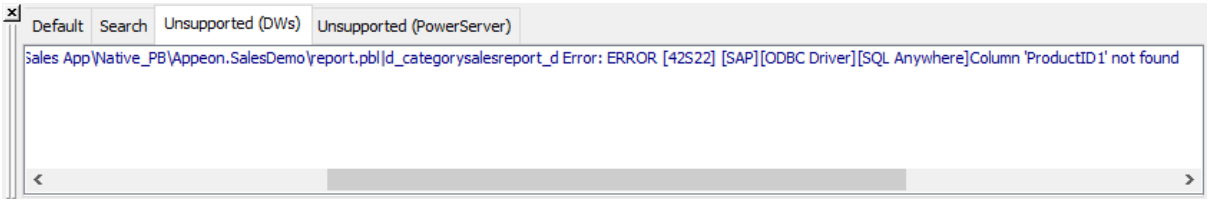


  Figure 1.3:

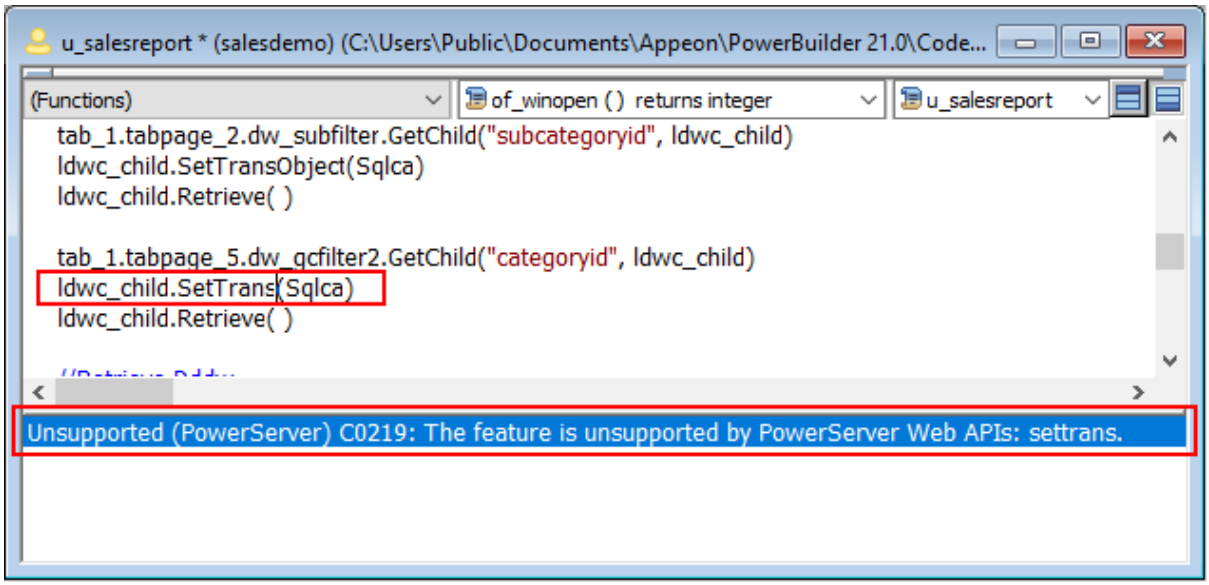


- when the object is saved.

PowerBuilder compiles each time when PowerScript is saved and displays the unsupported feature for PowerServer, regardless of the project type.

If any unsupported feature is detected, it will be displayed as compilation warnings, as shown below.

Figure 1.4:



Make sure to modify the unsupported feature according to the suggested workarounds in this guide. Even if there is no unsupported features reported, it is still recommended that you go through the following sections to find out what features/descrepancies cannot be detected and why they are not working as expected in the installable cloud app.

- Unsupported features & workarounds

  Unsupported features that can be detected -- lists the unsupported features that **can be detected** by the deployment tool and provides possible workarounds.

  Unsupported features that cannot be detected -- lists the unsupported features that **cannot be detected** by the deployment tool and provides possible workarounds.

- Discrepancies & workarounds

  Discrepancies that cannot be detected -- lists the programming or behavior differences between PowerBuilder and PowerServer that **cannot be detected** by the deployment tool and provides possible workarounds.

Tips:

If you want the analysis tool to ignore a piece of code, you can place the code within these labels: *#begin_disable_ufa* and *#end_disable_ufa*.

For example, the *SetTrans* function in the following example will not be reported as an unsupported feature.

```
#begin_disable_ufa

if ispowerserverapp () = true then
```

```
        dw_1.SetTransObject(sqlca)
else
        dw_1.settrans(sqlca)
end if

#end_disable_ufa

dw_1.retrieve()
```

2 Unsupported features & workarounds

2.1 Unsupported features that can be detected

This section lists the unsupported features that **can be detected** by the deployment tool, and provides possible workarounds.

2.1.1 SetTrans

Unsupported feature

SetTrans function is not supported. For example,

```
integer dwcontrol.SetTrans ( transaction transaction )
```

Workaround

Use **SetTransObject** to replace the **SetTrans** function.

2.1.2 Data pipeline

Unsupported feature

Data pipeline (Pipeline object) is not supported.

Workaround

You can consider making data pipeline an independent application and then calling this application:

Step 1: Extract the data pipeline into a separate PowerBuilder target and deploy the target as an executable application.

Step 2: Modify the scripts to call the pipeline executable application and open the window through the commandline parameter.

For example, change the *of_run_window* function of the **w_main** window in the **Example App** (**PB Examples**)

from

```
li_Resp = Open(lw_Ex, as_Window)
```

to

```
if ispowerserverapp () = true then
  if as_Window = 'w_pipeline' or as_Window = 'w_pipeline_blob' or as_Window =
 'w_pipeline_sp' then
     //run the application and pass the commandline parameter to open the window
     run( GetCurrentDirectory ( ) + "\unsupportfeature\unsupport.exe "+as_Window)
  else
     li_Resp = Open(lw_Ex, as_Window)
  end if
  else
  li_Resp = Open(lw_Ex, as_Window)
end if
```

Step 3: In the PowerServer project painter > **External Files** tab > **Files preloaded as compressed packages** section, add the pipeline executable application, including the EXE, PBD, and the runtime files (especially runtime for database interface).

2.1.3 MobiLink

Unsupported feature

MobiLink (MLSync object, MLSynchronization object, and SyncParm object) is not supported.

Workaround

Please consider the workaround used by data pipeline.

2.1.4 Oracle RPC arrays

Unsupported feature

Oracle RPC does not support arrays.

Workaround

Convert the array to a string and use the string instead of the array.

For example, the following stored procedure has an array IN parameter and an array OUT parameter. As a workaround, use varchar to replace both the IN and OUT parameters.

Assembly the IN parameter in the PowerScript:

```
ls_outparam = space(30)
ls_inparam = '123;456'
```

The INI parameter will be split in the stored procedure. Do the same for the OUT parameter.

```
PROCEDURE pro_arrNum(
                in_arrParamNum in usertype_number,
                out_arrParamNum out usertype_number
                )
AS
BEGIN
    out_arrParamNum(1) :=  in_arrParamNum(1) ;
    out_arrParamNum(2) :=  in_arrParamNum(2);

END pro_arrNum;

procedure pro_arrNumTostr(
                in_varchar in varchar,
                out_varchar out varchar
                )
AS
ls_resstr varchar(30);
ls_desstr varchar(30);
li_pos int;
BEGIN
    li_pos := 1;
    ls_resstr := in_varchar;
    while li_pos > 0 and length(ls_resstr) > 0 loop
        li_pos := instr(in_varchar, ';', 1);
        if li_pos > 0 then
            ls_desstr := ls_desstr || ';' || substr(ls_resstr, 1, li_pos - 1) ;
            ls_resstr := substr(ls_resstr, li_pos + 1);
        end if;
    end loop;
    out_varchar := ls_desstr;
    ls_desstr := ls_desstr || ';' || '999';
END pro_arrNumTostr;
```

PowerScript:

```
int li_pos, li_index
string  ls_inparam, ls_outparam, ls_return, ls_value
long ll_outarrparam[]

ls_outparam = space(30)
ls_inparam = '123;456'

gtr_trans.pro_arrNumTostr(ls_inparam, ref ls_outparam)

li_pos = 1
mle_1.text = ''

ll_outarrparam = wf_formatstring(ls_outparam)


public function any wf_formatstring (string as_value);int li_pos, li_index
string  ls_value, ls_return
long ll_foramtvalue[]
li_pos = 1

do while len(as_value)  > 0 and li_pos >  0
        li_pos = pos(as_value,  ";")
        ls_value = mid(as_value,1,  li_pos -1)
        if IsNumber (ls_value)  then
                li_index++
                ll_foramtvalue[li_index] = Integer(ls_value)
        end if
        as_value =  mid(as_value, li_pos+1)
loop

if IsNumber (as_value)  then ll_foramtvalue[li_index+1] = Integer(as_value)

return ll_foramtvalue
end function
```

2.1.5 SQLPreview

Unsupported feature

The **sqltype** argument of the SQLPreview event only supports the PreviewSelect type, and does not support the PreviewInsert, PreviewDelete, and PreviewUpdate types. For the PreviewSelect type, PowerServer does not return the SELECT statement, the SELECT statement generated on the client side will be different from the SELECT statement executed in the database. The SELECT statement on the client side is generated according to the PowerBuilder logic, while the SELECT statement executed in the database is generated by the PowerServer runtime.

The **request** argument of the **SQLPreview** event only supports PreviewFunctionRetrieve, and does not support PreviewFunctionReselectRow and PreviewFunctionUpdate.

Workaround

This feature will be supported in the next release.

2.1.6 SQLReturnData property

Unsupported feature

The **SQLReturnData** property of the **Transaction** object is unsupported.

Workaround

N/A

2.2 Unsupported features that cannot be detected

This section lists the unsupported features that **cannot be detected** by the deployment tool, and provides possible workarounds.

2.2.1 Transaction trace

Unsupported feature

The transaction trace is not supported. For example,

```
SQLCA.DBMS = "TRACE MSOLEDBSQL SQL Server"
```

Workaround

It can be partially worked around using server logs under the debug mode.

2.2.2 Unsupported use cases in Embedded SQLs

Embedded SQLs are supported, but there are a few unsupported use cases.

Unsupported use case #1

When executing a procedure in the cursor, only single result set is supported; the output parameter, return value, and multiple result sets are all unsupported.

```
declare lcs_test1 cursor for execute hr.synonyms_package.get_emp;
open  lcs_test1;
fetch lcs_test1 into :ls_name;
close lcs_test1;
```

Workaround

The code example below processes the result sets, return value and output parameter one at a time.

```
int li_intParam, li_retValue, li_bitResult
string ls_outVarParam, ls_outNvarParam, ls_varResult, ls_ncharResult

li_intParam = 1

declare lp_procName01 procedure for  @li_retValue = get_muiltResultset
 @in_intParam = :li_intParam,  @out_varParam = :ls_outVarParam  output,
 @out_nvarParam = :ls_outNvarParam  output;
execute  lp_procName01;

//Handles the first result set
fetch lp_procName01 into :ls_varResult, :ls_ncharResult, :li_bitResult;
do while sqlca.sqlcode = 0
                fetch lp_procName01
 into :ls_varResult, :ls_ncharResult, :li_bitResult;
loop

//Handles the second result set
fetch lp_procName01 into :ls_varResult, :ls_ncharResult;
do while sqlca.sqlcode = 0
                fetch lp_procName01 into :ls_varResult, :ls_ncharResult;
```

```
loop

//Handles the return value and output parameter
fetch lp_procName01 into :li_retValue, :ls_outVarParam, :ls_outNvarParam;
close lp_procName01;
```

Unsupported use case #2

Different transactions work on the same temp table. For example, after the first transaction is committed, the second transaction still accesses the temp table that is created in the first transaction. In this case, an "invalid object name #TEMPTABLE" error may occur.

Workaround

Make sure that all the operations related with one temp table are performed in the same transaction.

Unsupported use case #3

The DataWindows and/or embedded SQLs included in the PBD file cannot be parsed to the C# models (and the **SetLibraryList** and **AddToLibraryList** functions will not work properly with such PBD files as well).

Workaround

Manually convert the DataWindows and embedded SQLs from the corresponding PBL file through DataWindow Converter.

2.2.3 Retrieve As Needed and Rows to Disk

Unsupported feature

Retrieve As Needed and **Rows to Disk** options are not supported (which means all data will be retrieved).

Workaround

For retrieving a large amount of data, you can consider adding the WHERE clause to retrieve only the data needed.

2.2.4 SyntaxFromSQL

Unsupported feature

SyntaxFromSQL does not support stored procedures and functions if they use the temporary table; it will be supported in later versions.

Workaround

Use a statically created DataWindow (instead of a DataWindow dynamically created by SyntaxFromSQL) to call stored procedures and functions which use the temporary table.

2.2.5 Database synonyms

Unsupported feature

Database synonyms are unsupported. Synonyms of different owners in the same database only supports the SELECT statement.

Workaround

Call database synonyms in the C# assembly or REST APIs and then modify PowerScript to call the C# assembly or REST APIs.

2.2.6 Commit or Rollback Transaction using Dynamic SQL

Unsupported feature

Transactions that are dynamically committed are unsupported.

SQLs that are dynamically committed or rolled back are unsupported.

Example 1:

```
execute immediate "commit";
```

```
string ls_sql
ls_sql = "Rollback"
Execute immediate :ls_sql;
```

Workaround

Call the Commit or Rollback SQL statement directly. For example,

```
Commit {USING TransactionObject};
```

```
Rollback {USING TransactionObject};
```

Example 2:

```
ls_exec = 'SAVE TRANSACTION ' + as_savepointname
execute immediate :ls_exec using sqlca;
```

```
//NOTE this is a rollback of a savepoint, not a rollback of the entire transaction:
ls_exec = 'ROLLBACK TRANSACTION ' + as_savepointname
execute immediate :ls_exec using sqlca;
```

Workaround

Move the related business logic to the procedure and implement the transaction savepoint in the procedure.

2.2.7 Data retrieval and SQL operations in the RetrieveRow event

Unsupported feature

Data retrieval and SQL operations in the **RetrieveRow** event are not supported.

Workarounds

N/A

3 Discrepancies & workarounds

3.1 Discrepancies that cannot be detected

PowerBuilder and PowerServer have discrepancies in dealing with features such as the database connection, alias name etc. These discrepancies **cannot be detected** by the deployment tool.

Due to these discrepancies they might have different behaviors at runtime.

3.1.1 DB connection

In traditional client/server applications, one application just uses one database connection.

In applications deployed from PowerServer, each transaction uses a database connection, and when the transaction is completed, the database connection is ended.

3.1.2 Alias name

For dynamic DataWindow objects that are created by SyntaxFromSQL, if the alias name is the same as the column name, you will need to carefully check if the correct column name is used in the scripts.

Take the following as an example. PowerBuilder will use "t_dwstyle_grid_employ_empid" as the column name, while PowerServer will use "empid" as the column name.

```
select t_dwstyle_grid_employ.empid as empid, t_dwstyle_grid_employ.empname as
 empname, t_dwstyle_dept.deptname as deptname
from t_dwstyle_grid_employ, t_dwstyle_dept
where t_dwstyle_grid_employ.deptid = t_dwstyle_dept.deptid and
 t_dwstyle_grid_employ.empid < 500
```

Thus, the following script will cause a runtime error in the installable cloud app.

```
getitem (row, "t_dwstyle_grid_employ_empid")
```

3.1.3 Data type mismatch

If data type is corrected while SRD is not re-generated to reflect the change (as shown below, data type is still mismatched), PowerBuilder will throw an error when trying to retrieve data, while PowerServer will retrieve data successfully.

```
column=(type=long update=yes updatewhereclause=yes name=starttime
 dbname="t_dwstyle_grid_employ.starttime" )
```

3.1.4 rowsupdated value

When the column is set to not updatable (as shown below), modifying the column data and then performing an update will cause the **rowsupdated** argument of the **UpdateEnd** event to:

- Return 1 in PowerBuilder, but actually no update statement is generated at all.
- Return 0 in PowerServer, indicating that no rows are updated.

```
dw_control.Object.columnname.Update = "No"
```

3.1.5 DisableBind parameter

Suppose the database column **updatetime** is defined as below.

```
updatetime  datetime  not null default  getdate()
```

If the application user inserts a new row, but does not enter a value for this column, then when **DisableBind** is set to 0,

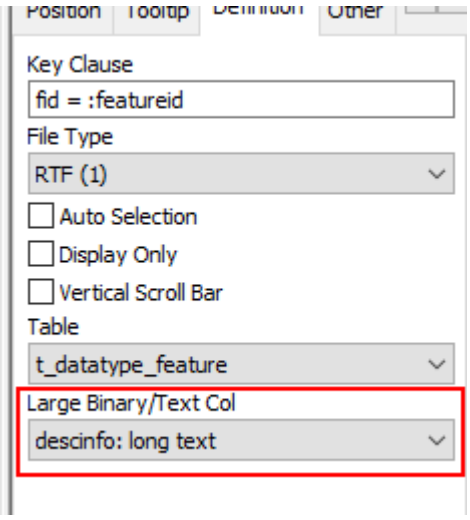
- PowerBuilder throws an error indicating that the **updatetime** column cannot be null.
- PowerServer inserts the data row to the database successfully (the **updatetime** column takes the default value from database).

  In PowerServer, **DisableBind** always takes value 0 for the DataWindow UPDATE statement and may ignore the null value according to the sqldefault attribute; while takes value 0 or 1 for the DataWindow SELECT statement and ESQLs.

3.1.6 TableBlob retrieval

If the TableBlob control selects a text field, PowerBuilder retrieves data for all columns except for this blob; while PowerServer retrieves no data (thus DataWindow will have no data at all).

Figure 3.1:



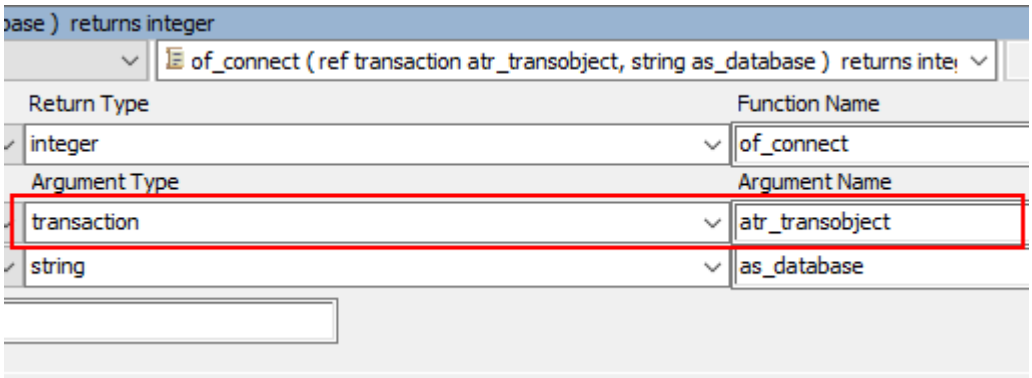
3.1.7 Dynamic DataWindow

When a computed field has no alias, PowerBuilder will use the computed expression as the column title, while PowerServer will automatically give a name (such as "Compute2") as the column title.

3.1.8 TransactionName

PowerServer will use the transaction object name to map with the database cache name. But when the transaction object is defined as an argument of a function, as shown below, PowerServer will get different transaction name from PowerBuilder.

Figure 3.2:




```
transaction ltr_tmp
ltr_tmp = create transaction
gnv_manager.of_connect(ltr_tmp,"qa_datawindow")
```

- PowerBuilder will get *ltr_tmp* as the transaction object name.

  For example,

  ```
  dw_1.settransobject(ltr_tmp)
  dw_1.retrieve()
  ```

- PowerServer will get *atr_transobject* as the transaction object name.

Defining the transaction object as an instance variable or global variable can ensure PowerBuilder and PowerServer get the same transaction object name.

3.1.9 Data type in Dynamic SQL Format 4

In PowerBuilder, the Oracle database may return the numeric data as the decimal type.

```
Row=1, Column=1, type=Decimal, value=1
Row=2, Column=1, type=Decimal, value=2
```

While in PowerServer, the Oracle database may return the numeric data as the longlong type.

```
Row=1, Column=1, type=LongLong, value=1
Row=2, Column=1, type=LongLong, value=2
```

It is recommended that the developer use "choose case" to support the longlong-type numeric data.

```
CHOOSE CASE SQLDA.OutParmType[n]
case TypeLongLong!
   ls_DataType = 'LongLong'
   ls_Value = String(adda_parm.GetDynamicDecimal(li_Idx))
```

3.1.10 Decimal data type in static SQL or DataWindow

PowerServer will return numeric data with a fixed decimal point length according to the decimal precision of the column. If the decimal place is insufficient, zero will be automatically filled; while PowerBuilder will not fill zero after the decimal point of decimal data. For example, the money-type data may display as an integer (for example, 40) in PowerBuilder; while display as a floating point number with 4 decimal places (for example, 40.0000) in PowerServer.

3.1.11 Timing of transaction rollback

If the SELECT statement is executed after the UPDATE statement (like below) and if the execution of UPDATE is successful while the execution of SELECT is not, PowerServer will immediately roll back the transaction (and roll back UPDATE), while PowerBuilder will not. This may cause that data to be retrieved later will be different between PowerServer and PowerBuilder.

```
update dbparm_fortest set name_char = :ls_tmp1,name_varch=:ls_tmp1 where id
 = :li_id using tran01;
select "name_char",name_varch into :ls_char,:ls_varchar from dbparm_fortest where
 "id"= :li_id using tran01;
ls_return += "ls_char=" + ls_char+" "+string(len(ls_char)) + is_newline
ls_return += "ls_varchar=" + ls_varchar+" "+string(len(ls_varchar)) + is_newline
select count(1) into :ll_count from dbparm_fortest where name_char = :ls_name using
 tran01;
```

3.1.12 Oracle AutoCommit and Lock

The Oracle AutoCommit and Lock properties take effect in PowerServer, while take no effect in PowerBuilder.

If an unsupported isolation level is set, for example, lock= "RU", PowerServer will throw an error executing the SQL.

3.1.13 Stored procedure parameter

If the output parameter of stored procedure has default values, even if PowerBuilder did not pass the output parameter, the server can still use the default value of the stored procedure to successfully get data. However, in PowerServer, if PowerBuilder did not pass the output parameter, PowerServer will use null as the default value. This will cause the result set different between PowerBuilder and PowerServer.

If the parameter name or number does not match, PowerBuilder will display an error indicating that the parameter does not exist; while PowerServer will check the schema of stored procedure or function and automatically match the corresponding parameter type and position, therefore PowerServer may be able to execute the stored procedure without errors.

3.1.14 Transaction commit

If the commit statement is executed after Transaction.autocommit = true, PowerServer will display the error: Database connection or transaction is not opened, commit is invalid. To avoid this error, do not execute commit/rollback after Transaction.autocommit = true; or setting the "**TransactionException**" property to *false* in the **ServerAPIs** project > **AppConfig** > **Applications.json** to suppress the error message.

If the commit statement is executed after the select statement, PowerServer will display an error indicating that the database connection failed due to the unknown logic error. To avoid the error, remove the commit statement after the select statement.

3.1.15 Use Describe in Dynamic SQL Format 4

You can no longer use **Describe** in Dynamic SQL Format 4 to check the SQL syntax. For example,

```
describe sqlsa into sqlda;
if not this.of_checktrans(atr ) then
        return string(atr.sqlcode)+":"+atr.sqlerrtext
end if
```

However, this discrepancy can be ignored as the SQL syntax will be checked when opened or executed on the server.

3.1.16 Bit data field

When the SNC SQL Native Client database interface is used, the Bit data field returns -1 when the data value is 1.

It is recommended to use IsPowerServerApp to determine the scripts to execute for the application deployed via PowerServer.

3.1.17 SelectBlob/UpdateBlob supports UTF8 only

SelectBlob and UpdateBlob in PowerServer can only handle the value with UTF8 character encoding (EncodingUTF8!).

When UpdateBlob updates a UTF8 value in PowerServer, or when UpdateBlob updates a UTF16LE value in PowerBuilder, SelectBlob in PowerServer can correctly display the value using the UTF8 or ANSI encoding while SelectBlob in PowerBuilder can correctly display the value using the default UTF16LE encoding.

You can write scripts below to minimize the impact of this discrepancy:

```
if ispowerserverapp () = true then

        Lblob = Blob("Any Text", EncodingUTF8!)
        Updateblob caseresfile  set filestring = :lblob  where filename
 = :as_filename;
        SELECTBLOB filestring into :lblob  from caseresfile where filename
 = :as_filename;
        ls_return = String(lblob, encodingutf8!)

else
        Lblob = Blob("Any Text")
        Updateblob caseresfile  set filestring = :lblob  where filename
 = :as_filename;
        SELECTBLOB filestring into :lblob  from caseresfile where filename
 = :as_filename;
        ls_return = String(lblob)
end if
```

3.1.18 SQLNRows property (with Cursor)

When executing a cursor, PowerServer fetches all data rows at a time therefore SQLNRows returns the total amount of all fetched rows, while PowerBuilder fetches one data row at a time therefore SQLNRows returns 1.

3.1.19 SQLCode property (with SP)

If a stored procedure returns a result set which contains 0 data row, SQLCode property returns -1 in PowerServer while returns 100 in PowerBuilder.


```
"content":{"returnvalue":null,"outparams":
[{"datatype":"long","value":7}],"resultsets":[]}
```

3.1.20 Column name from view

If the table is from a view, PowerBuilder and PowerServer will generate different columns names.

For example,

```
//v_cust_dept is from view cust_dept
select  v_cust_dept.v_id from v_cust_dept;
```

PowerBuilder will generate the column name as v_cust_dept_v_id; while PowerServer will generate the column name as cust_dept_v_id.

If using dw_1.getitem(1,"v_cust_dept_v_id") to get data, runtime error will occur. Use dw_1.getitem(1,index) instead.

4 Incompatible coding styles

Nonstandard PowerScript coding practices might cause problems when converting DataWindow to C# models; and special C# coding conventions might also prevent scripts running as expected in the .NET server. Following are bad or unrecommended practices that are commonly seen in PowerScript and must be avoided or corrected before deploying the application with PowerServer.

4.1 PBLs contain DataWindows with the same name

It is not a recommended practice for multiple PBLs in the same application containing DataWindows with the same name. When converting to the C# models, only the first DataWindow will be converted (the other duplicates will be ignored), because all converted C# models are placed in the same ASP.NET project, and a single ASP.NET project cannot have two models with the same name.

Please avoid having DataWindows with the same name in the application.

4.2 Object name using C# reversed words

C# reversed words (such as "abstract", "base", and "delegate") cannot be used as the PowerBuilder object name, otherwise, the object cannot be executed as expected.

Please avoid using the C# reversed words as object name.

4.3 DataWindow name containing special characters

The C# naming convention does not allow using special characters (such as dash (-), dollar sign ($), number sign (#), and percent sign (%)) in the DataWindow name. Such special characters in the DataWindow name will be replaced with underscores when converted to the C# model; thus the DataWindow will not be found after conversion as the name has changed. For example, the DataWindow name "d_sp_who_with-dash" will be converted to "D_Sp_Who_With_Dash.cs"; and the following error may occur when retrieving data: "Select Error: DataWindow 'd_sp_who_with-dash' was not found".

Please double check that the DataWindow name contains no dash (-), dollar sign ($), number sign (#), or percent sign (%).

4.4 Editing SQL

It is not a PowerBuilder recommended practice to type (instead of select) the logical keyword in the SQL Select painter.

If you type the logical keyword (such as "and") to the value in the SQL Select painter, the model conversion and data retrieval in PowerServer will have the error: Incorrect syntax near ')'. To resolve this error, you will need to select the logical operator, instead of manually typing it. Or go to the Source Editor, and change **AND~"** to **~" LOGIC = ~"and~"**.

Figure 4.1:

Column	Operator	Value	Logical
(vacaciones_empleado.compania	=	:compania)	And
(vacaciones_empleado.codigo_empleado	=	:empleado)	And
regimen_vacaciones.prioridad	=	acaciones_empleado.codigo_empleado = :empleado)))AND	
vacaciones_empleado.dias_disponibles	<=	0	

4.5 Column order in data source and Column Specification

It is not a PowerBuilder best practice for the columns in the data source to be in a different order from the columns in the Column Specification.

If the columns listed in the data source and in the Column Specification are not in the same order, the data retrieval will fail in the .NET server.

Suppose the data source is

```
SELECT "employee"."emp_id","employee"."emp_fname","employee"."start_date" FROM
 "employee"
```

If the column specification is changed from

```
table(column=(type=long update=yes updatewhereclause=yes key=yes name=emp_id
 dbname="employee.emp_id" )
column=(type=char(20) update=yes updatewhereclause=yes name=emp_fname
 dbname="employee.emp_fname" )
column=(type=date update=yes updatewhereclause=yes name=start_date
 dbname="employee.start_date" )
```

to

```
table(column=(type=long update=yes updatewhereclause=yes key=yes name=emp_id
 dbname="employee.emp_id" )
column=(type=date update=yes updatewhereclause=yes name=start_date
 dbname="employee.start_date" )
column=(type=char(20) update=yes updatewhereclause=yes name=emp_fname
 dbname="employee.emp_fname" )
```

then the DataWindow will be converted incorrectly to the model.

```
[Key]
[DwColumn("employee", "emp_id")]
public int? Emp_Id { get; set; }
[ConcurrencyCheck]
[DwColumn("employee", "emp_fname", TypeName = "char")]
public string Start_Date { get; set; }
[ConcurrencyCheck]
[DwColumn("employee", "start_date")]
public DateTime? Emp_Fname { get; set; }
```

And the following error will occur when retrieving data in the application:

```
sqlerrtext=Invalid object name 'employee'.
```

4.6 One compute expression containing multiple computed columns

It is not a PowerBuilder best practice to define multiple computed columns in one compute expression, as shown below.

Figure 4.2:

Computed Columns
0 AS display, 0 AS sort_order, '' AS order_type, 0 AS sort_seq

When converted to the C# models, the computed columns cannot be split into separate ones correctly.

To avoid any problems, please specify one computed column in one compute expression, as shown below.

Figure 4.3:

Computed Columns
0 AS display
0 AS sort_order
'' AS order_type
0 AS sort_seq

4.7 Cursor syntax

When declaring a cursor, it is not a recommended practice to add the *into* keyword. PowerBuilder will consider it redundant and ignore it; while PowerServer will consider it (ls_result in the following example) as a parameter and will display the error: The number of parameters does not match.

```
string ls_sql, ls_result
ls_vid = '100002'
ls_vname = ''
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
ls_sql = 'SELECT v_name into :ls_result  FROM employee WHERE v_id = ?'
PREPARE SQLSA FROM :ls_sql ;
OPEN DYNAMIC my_cursor using :ls_vid ;
FETCH my_cursor INTO :ls_vname ;
CLOSE my_cursor ;
```

4.8 Syntax after UNION

PowerBuilder will not check the syntax after the UNION keyword. If the syntax after UNION is invalid, for example, a space is missing between "**SELECT**" and "**:as_test**" as shown below, PowerServer will fail to convert the DataWindow to a model.

```
SELECT :as_test test1, Dept.ID, Dept.Name FROM Dept
union
  SELECT:as_test test1, Dept.ID, Dept.Name FROM Dept
```

Troubleshooting Guide

Contents

1 Configuring and deploying PowerServer projects

1.1 Permission errors when configuring the Web server profile

Permission errors occur when testing the file path in the Web server profile or trying to connect to the Web server during the deployment process.

Solution:

1. Restart PowerBuilder IDE using the "Run as administrator" option.

2. Check if the Windows user has read and write permissions to the Web root.

1.2 Error during the build process

The following error occurs when building the PowerServer project:

```
Error: ERROR [42000] [Sybase][ODBC Driver][SQL Anywhere]Syntax error near ###'…
```

Solution:

When configuring the database cache in the **Database Configuration**, please click the **Advanced Settings**, and make sure the **DelimitIdentifier** option is set to be consistent with runtime. If this option is changed dynamically at runtime, you will need to create two caches in order to make sure the model conversion is successful.

1.3 Error in the Unsupported (DWs) window

When converting the DataWindow to the model, the following error occurs in the **Unsupported (DWs)** window:

```
Error: Incorrect syntax near the keyword 'user'.
```

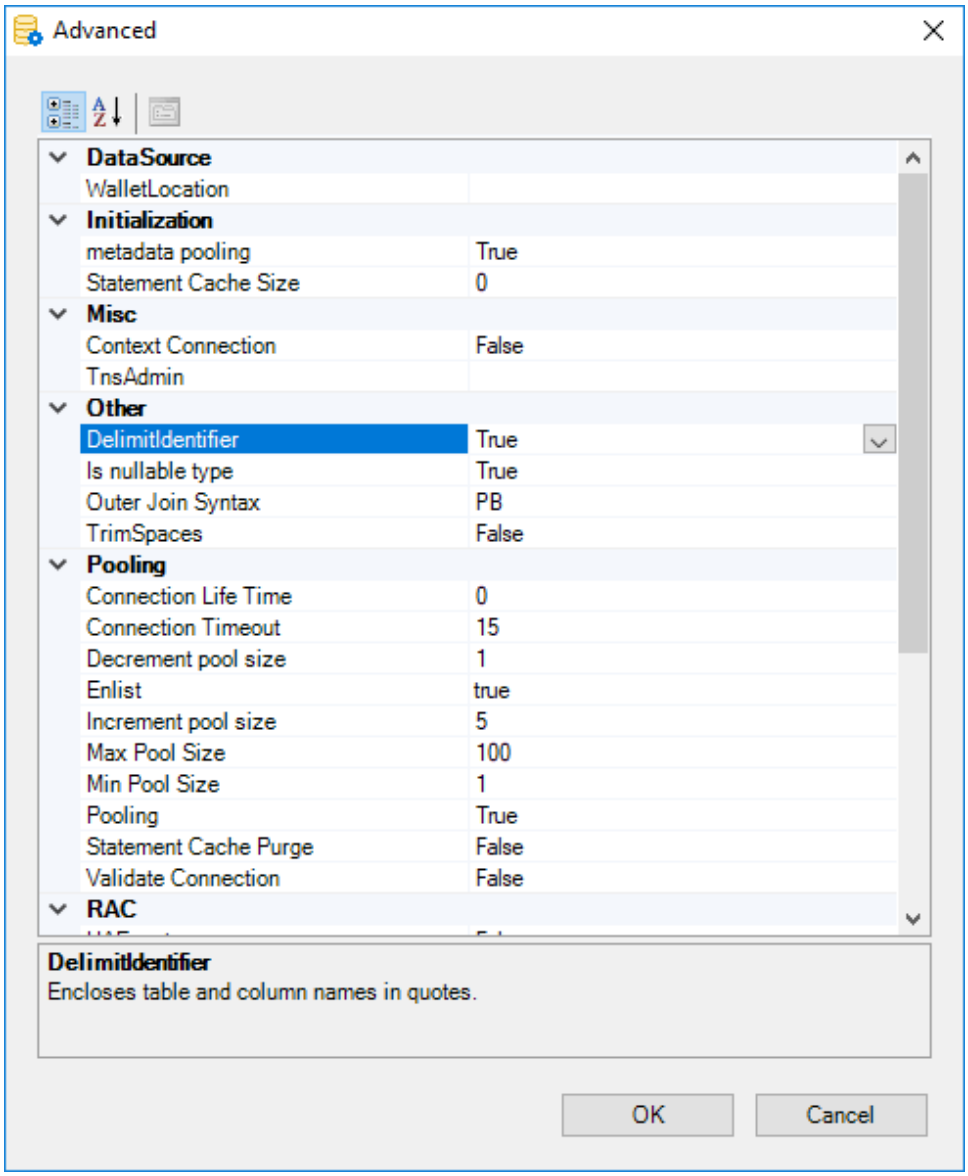
Cause:

The database table or data field contains keywords, and the **DelimitIdentifier** property is specified in the DataWindow DBParm.

Solution:

When configuring the Database Configuration in the PowerServer project, make sure to click the **Advanced** button and then set **DelimitIdentifier** to **True**.

Figure 1.1:



1.4 Failed to generate the PowerServer Web APIs project

The following error is reported in the Output window during the build & deploy process:

```
Failed to delete the file “C:\Users\appeon\source\repos\PowerServer_pssales
\ServerAPIs\bin\Debug\netcoreapp3.1\xxxx.dll”
because it may be occupied by another program. Error code: 5.
Failed to generate the PowerServer Web APIs project. Error code: 0.
```

Or

```
Failed to delete the folder “C:\Users\appeon\source\repos\PowerServer_pssales
\ServerAPIs”
because it may be occupied by another program. Error code: 32.
Failed to generate the PowerServer Web APIs project. Error code: 0.
```

Solution:

1. Close the PowerServer C# solution (PowerServer_pssales in the above example) if it is currently opened in any C# editor.

2. Close the Web APIs if it is currently running.

3. Build and deploy the project again.

1.5 Error uploading application files to FTP

The following error might occur while deploying the client app files to an FTP server:

```
Failed to connect to the server, it does not have permission to upload files.
Failed to publish the installable cloud app.
```

Cause:

The FTP user has no write permission.

Solution:

Make sure the FTP user you set has write permissions, especially when you use an FTP server in the cloud.

Alternatively, you can package and distribute the application files to the server manually. To package the application files, right click on the PowerServer project in the system tree, and select **Package PowerServer Project…**; or see Packaging the client app for more information.

1.6 Changed PBL list

The following message displays when you deployed a project that has already been deployed successfully before.

```
The current library list is different from the library list contained within this
 project. The project library list has been updated.
Refresh build options for the changed list and try again.
```

Cause:

The PBL list was updated (for example, with new or deleted PBLs) after it was first successfully deployed.

It is a known issue that it takes a project (especially a large project) a long time to display this prompt. This issue will be addressed in the later release.

Solution:

Build and deploy the project again.

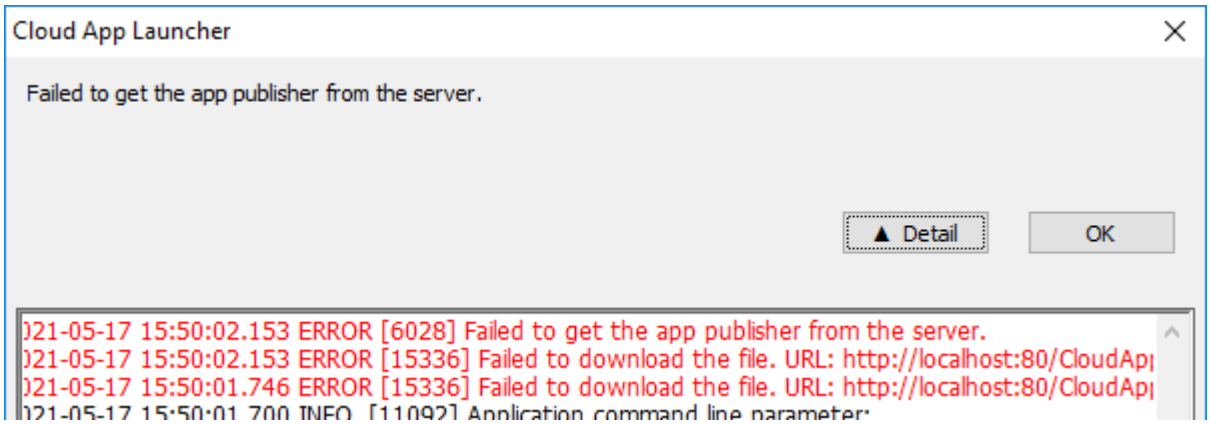
2 Running installable cloud apps

2.1 Cloud app launcher and application executable

2.1.1 Failed to get the app publisher from the server

The following error occurs when trying to run the installable cloud app for the first time.

Figure 2.1:



Cause:

The cloud app launcher has not been uploaded to the server.

Solution:

Upload the app launcher and runtime files according to the instructions in Upload the cloud app launcher and the runtime files; and then run the application again.

2.1.2 Cannot start cloud app launcher

The Cloud App Launcher failed to start even if the launcher has already been installed.

Cause:

The client machine has third-party firewall tool, such as Sophos. The process of Sophos such as *swi_filter.exe* and *swi_service.exe* block the access to the application URL.

Solution:

Configure the firewall to allow access to the following IP address and port numbers:

http://127.0.0.1:26568

http://127.0.0.1:26569

2.1.3 Application executable disappeared suddenly

The application executable file disappeared suddenly after being run successfully for a few times. The application's desktop shortcut disappeared too.

Cause:

The anti-virus software such as McAfee may incorrectly identify the application executable as malicious and block it from running.

Solution:

Try the following to resolve this issue:

1. Add the application executable file (as well as the cloud app launcher) to the exception list of the anti-virus software.

2. Sign the application executable (as well as the cloud app launcher).

3. Update the anti-virus software to the latest version and the latest virus definitions.

4. Report this false positive to the anti-virus software.

2.1.4 Window is slow to open

Under some circumstances, for example, a large application that contains lots of PBLs and the window uses user-defined images, the window may be slow to open in the application (deployed via PowerClient or PowerServer).

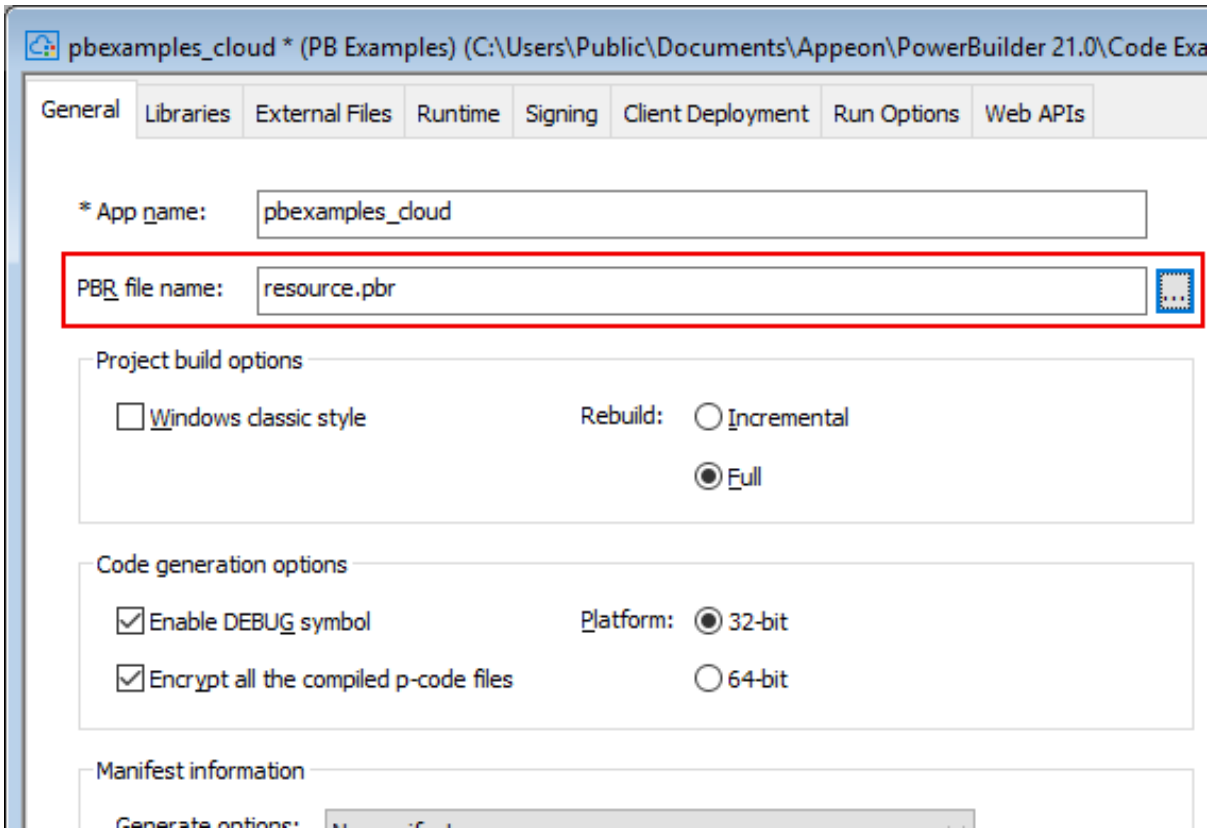
Cause:

The application will first search through all PBD files and then the application directory to find the user-defined images. In the PowerClient/PowerServer deployment, all PBD files are broken down very granularly into each individual object/definition file. When the application searches through a PBD folder, it first opens the _indexes.idx and _files.idx files in the PBD folder and then searches through all individual objects/definition files according to the _files.idx file. If there are many PBD folders, the elapsed time will be much longer.

Solution:

Use a PowerBuilder resource file (PBR) to list the user-defined images. When the PBR file is deployed with the application, the application can find the image very quickly through the PBR file.

Figure 2.2:



2.2 Models and controls

2.2.1 Cannot retrieve data when data includes null values

The following error occur when retrieving data: The property does not allow null value: Object_Ref.

Cause & Solution:

When converting the DataWindow to the model, the nullable property is not correctly set. You can search for the problematic object (for example, Object_Ref) in the exported models in the PowerServer C# solution, and modify it to allow null values. For example,

Change

```
[Key]
        [DwColumn("disp", "object_ref")]
        public long Object_Ref { get; set; }
```

To

```
[Key]
        [DwColumn("disp", "object_ref")]
        public long? Object_Ref { get; set; }
```

2.2.2 PBSELECT DataWindow error

The DataWindow created with PBSELECT crashed or GetSQLSelect returns an error when retrieving data.

Or the DataWindow created with PBSELECT cannot be converted to the model successfully.

Cause & Solution:

There are syntax errors when PowerServer converts PBSELECT to SELECT. It is recommended that you convert PBSELECT to SELECT in PowerBuilder first and then deploy the application with PowerServer again.

2.2.3 RibbonBar control displays blank

In the installable cloud app, the RibbonBar control displays blank.

Cause:

The script file (XML/JSON) that is used to create the RibbonBar control is not selected and deployed with the project.

Solution:

1. Open the PowerServer project object, go to the **External Files** tab, select **Files preloaded in uncompressed format** and then click **Add Files** to add the RibbonBar script file (XML/JSON).

2. Deploy the PowerServer project again to make the change effective.

2.3 Server

2.3.1 Cannot connect to the server when creating the session

The following error might occur when you run an installable cloud app: Cannot connect to the server when creating the session.

Cause A:

The .NET server might have its IP address changed, for example, it is set to obtain IP address automatically.

Solution A:

Set a static IP address or a domain name for the .NET server.

Cause B:

The .NET server might have set up a firewall and the firewall might not allow the specified port number to go through.

Solution B:

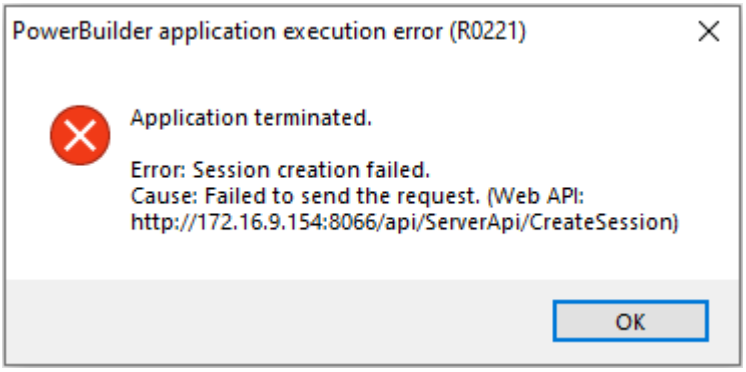
Configure the firewall on the server to allow the specified port number to go through.

2.3.2 Session creation failed

The following error might occur when you run an installable cloud app: Session creation failed.

Error 1:

Figure 2.3:



Cause:

If the host server connects to Internet via a proxy server, then PowerServer Web APIs has to be configured with the proxy server as well.

Solution:

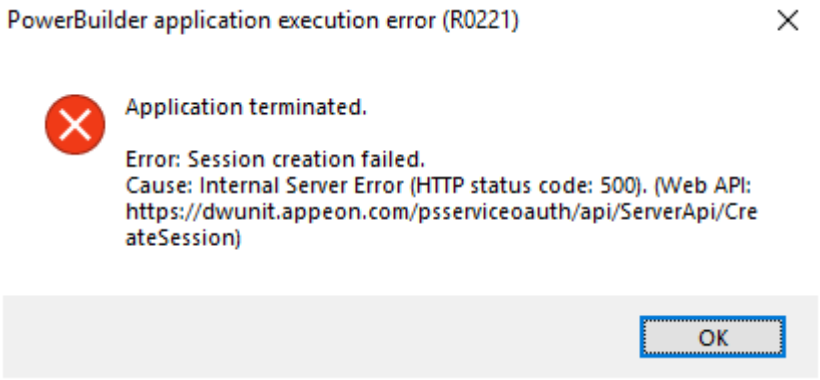
Open the PowerServer C# solution > the **ServerAPIs** project > the **Server.json** file, and configure the proxy server settings in the "**ProxyOptions**" block.

```
    . . .
```

```
  "ProxyOptions": {
    "Server": "",
    "Username": "",
    "Password": ""
  },
  ...
```

Error 2:

Figure 2.4:



Cause:

Two applications under the same IIS website cannot use the same application pool.

Solution:

Step 1: Configure the two applications to use different application pools.

Step 2: Restart the website.

2.3.3 App requires login again

The app might require the user to log in or run again on a daily basis.

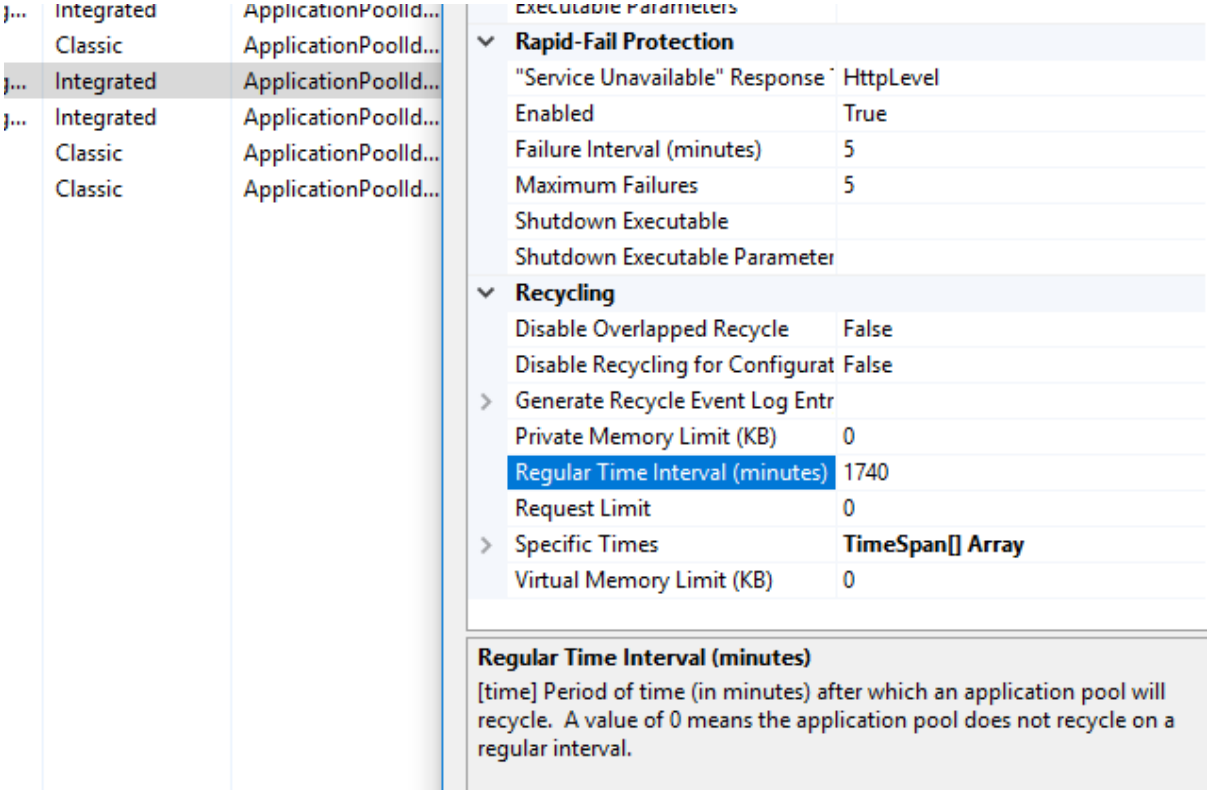
Cause:

In IIS, application pools are recycled every 1,740 minutes by default. The session in a running app will become invalid after the recycle period is reached, so the user has to log in or run the app again.

Solution:

Set the **Regular Time Interval** to 0 to stop recycle, or set to other duration.

Figure 2.5:



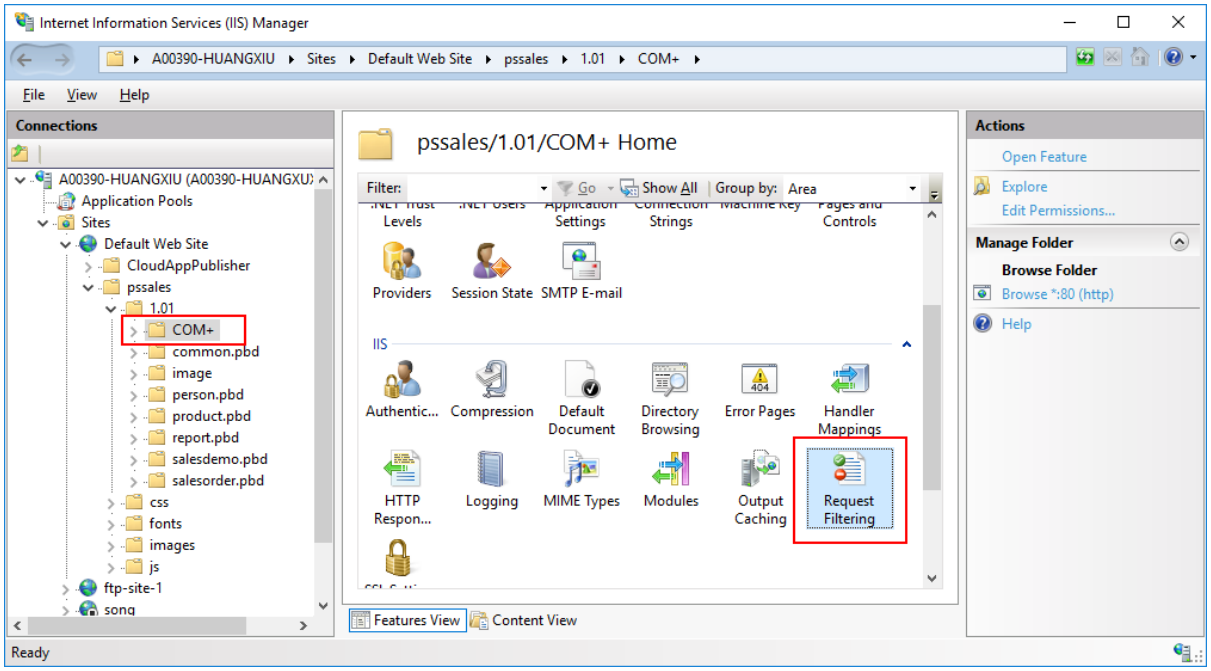
2.3.4 File name containing character + cannot be downloaded

If the file name contains the character "+", an 404 error will occur when the file is downloaded from the IIS Web server.

Cause & Solution:

In IIS Manager, select the folder which contains the character "+" in the folder or file name, and then double click **Request Filtering** on the **Features View**.

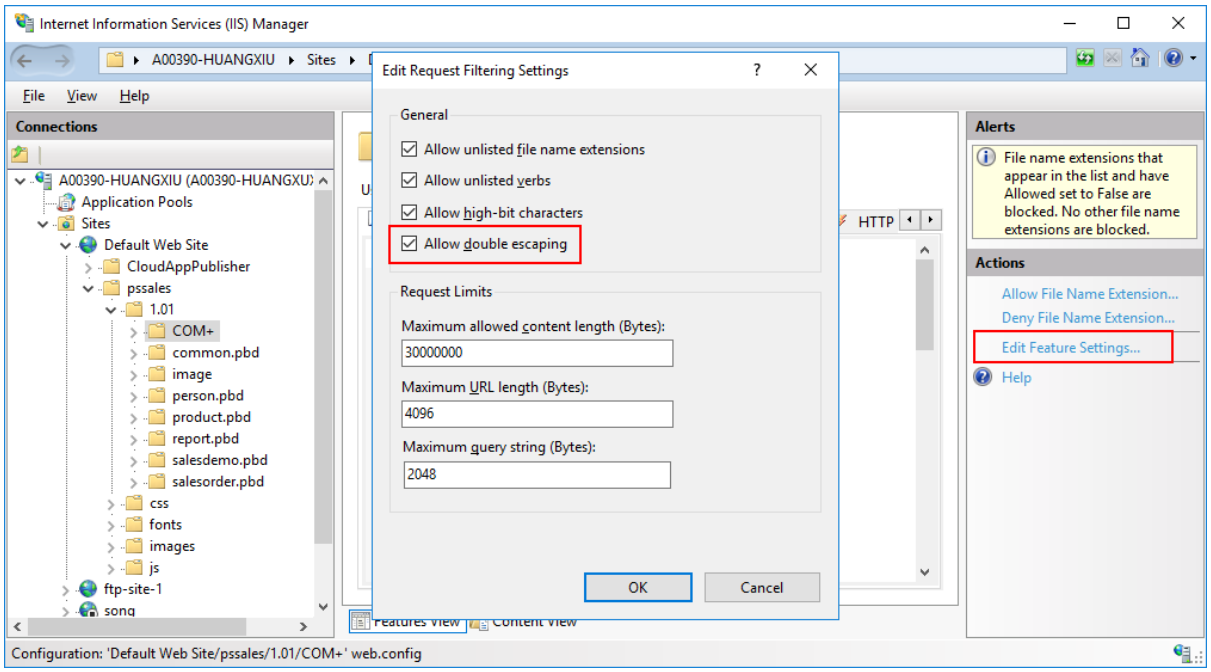
Figure 2.6:



Click **Edit Feature Settings** in the **Actions** pane.

In the **Edit Request Filtering Settings** dialog box, select **Allow double escaping**.

Figure 2.7:



2.3.5 "HTTP Error 404.2 - Not Found" error when running the app

When you run the application which is hosted in the IIS Web server, you get the following error:

```
HTTP Error 404.2 - Not Found
```

```
The page you are requesting cannot be served because of the ISAPI and CGI
 Restriction list settings on the Web server.
```

Cause:

The IIS server settings block the download of the CloudAppLauncher_Installer.exe file.

Solution:

In the IIS Manager, expand the server's node and then the **Sites** node in the **Connections** panel, select the website where the application is hosted, and then double click **Handler Mappings** on the **Features View**, and set **CGI-exe** to **Disabled**.

2.4 Database

2.4.1 Different results returned from an ASE stored procedure

For an ASE stored procedure, for example the following one, the results returned from the installable cloud app and the PowerBuilder C/S app might be different.

```
CREATE PROCEDURE g_qaQuestionSelect
AS
DECLARE @quest_seq tinyint, @ErrorMessage varchar(255)
SELECT @quest_seq = NULL
 IF @quest_seq = NULL   /* here is the error checking @quest_seq = NULL ,
 currently fixed it using  IsNull(@quest_seq,0) = 0 */
      SELECT @ErrorMessage = 'ERROR.'
 ELSE
      SELECT @ErrorMessage = 'DONE.'
```

Cause:

The PowerServer installable cloud application connects to ASE through the ODBC driver, while the PowerBuilder C/S app connects to ASE through the native driver. The default values of **Set ANSI Null** option in these two drivers are different.

Solution:

De-select the **Set ANSI Null** option in the **ODBC Data Source Administrator**.

Figure 2.8:

2.4.2 SelectBlob data truncated

The data values of the SelectBlob variable are truncated in the installable cloud app.

Cause & Solution:

The PowerServer installable cloud application connects to ASE through the ODBC driver, while the PowerBuilder C/S app connects to ASE through the native driver. The **Text size** option in the ODBC driver is 32KB by default.

You can increase the text size value using the Control Panel

1. Select Control Panel | Administrative Tools | Data Sources (ODBC), then select the data source for Adaptive Server Enterprise in the User DSN or System DSN tab.

2. Select Configure to display the ODBC Adaptive Server Enterprise Setup window, then select Advanced.

3. Change the value for Text Size to a larger value (the default value is 32KB). The Adaptive Server ODBC drive truncates any data value that is larger than the value you set here.

2.4.3 Garbage letters display when retrieving multibyte data

When retrieving multibyte data from the ASE database, garbage data displays in the installable cloud app.

Cause:

The PowerServer installable cloud application connects to ASE through the ODBC driver, while the PowerBuilder C/S app connects to ASE through the native driver. When using

the native driver, DBParm supports the **charset** parameter (for example, charset='roman8'); however when using the ODBC driver, DBParm does not support this parameter.

Solution:

Set the charset setting in the ODBC data source configuration page, for example,

cp852 -- PC Eastern Europe

cp1250 -- Microsoft Windows 3.1 Eastern European

cp869 -- IBM PC Greek

cp1253 -- MS Windows Greek

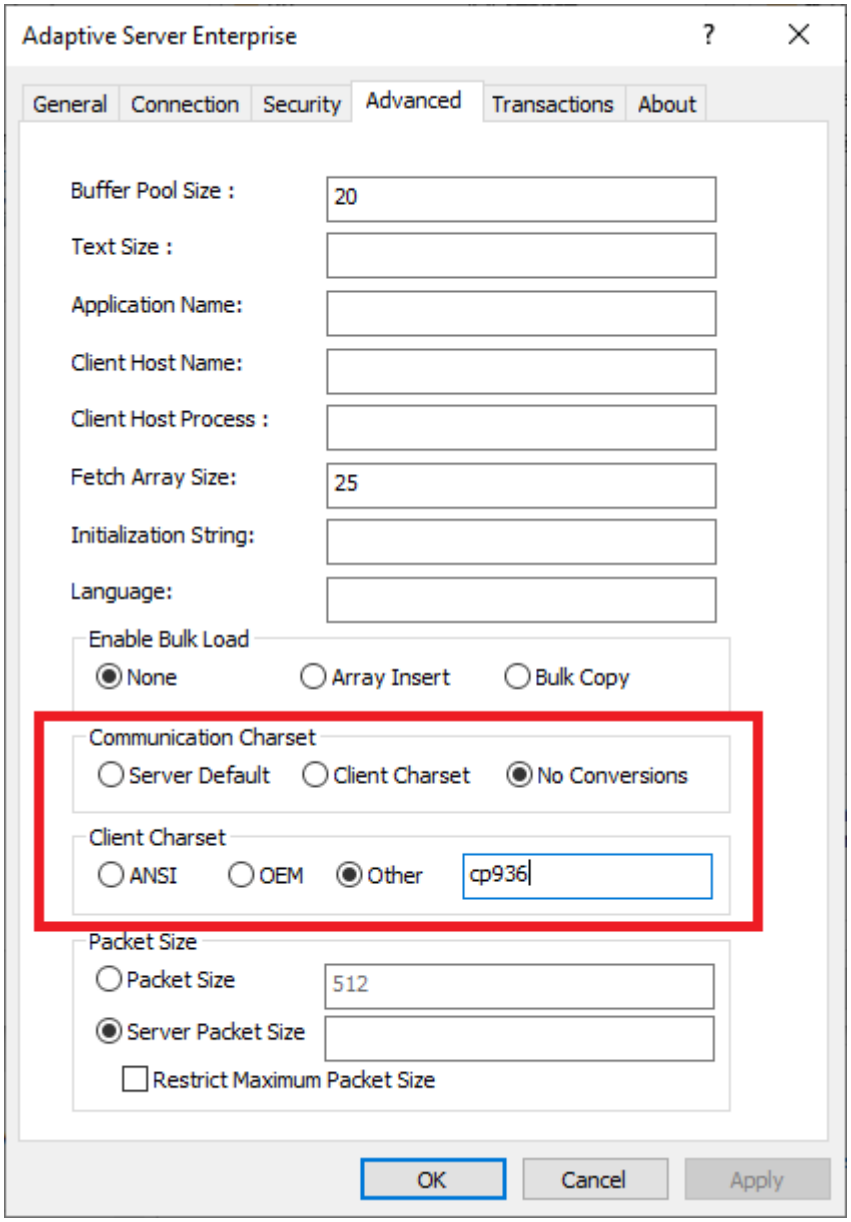
cp932 -- IBM J-DBCS:CP897 + CP301 (Shift-JIS)

sjis -- Shift-JIS (no extensions)

eucksc -- EUC KSC Korean encoding = CP949

cp936 -- Microsoft Simplified Chinese character sets

Figure 2.9:



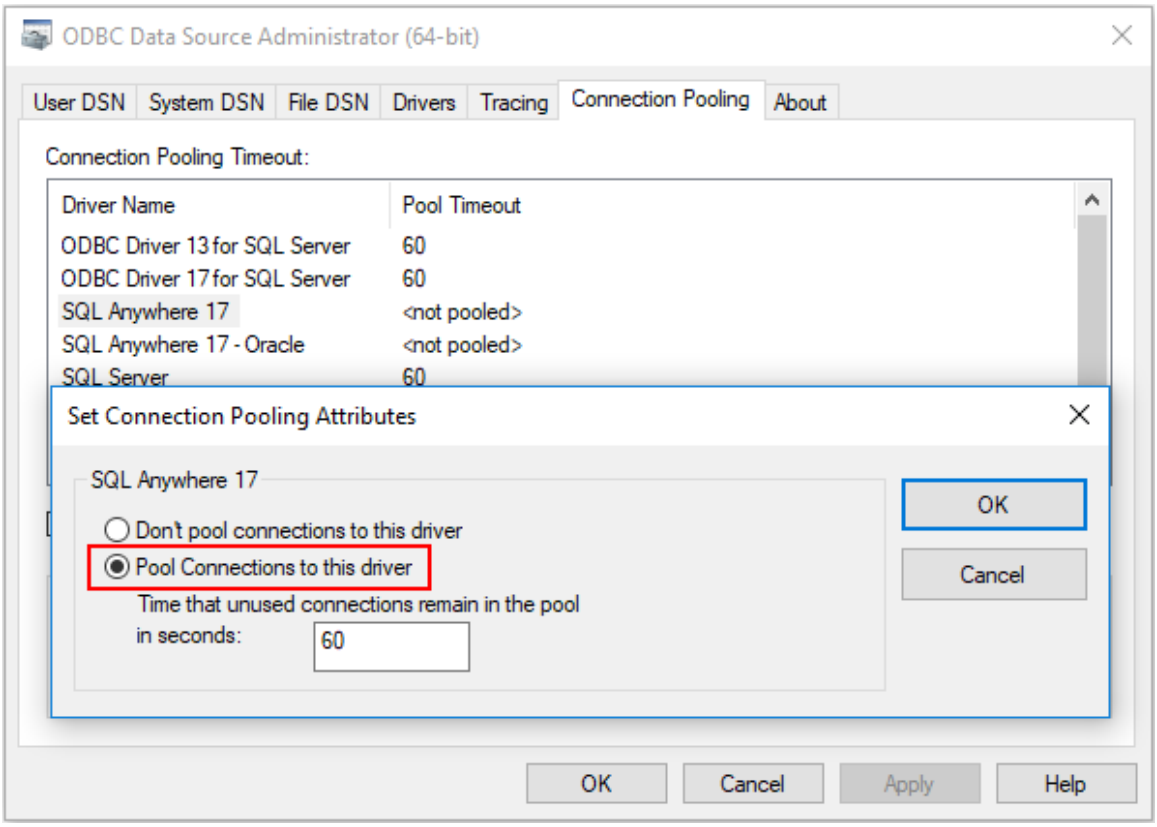
2.4.4 Slow app performance with SQL Anywhere

The app runs slowly when working with the SQL Anywhere database.

Solution:

Select the **Pool Connections to this driver** option in the **ODBC Data Source Administrator** to speed up the performance.

Figure 2.10:



2.4.5 64-bit database cannot be connected from IIS

When connecting with a 64-bit database (such as informix, ASE etc.) from the PowerServer Web APIs that runs in the same process as its IIS worker process (in-process hosting), the following error occurs:

```
ERROR [IM014] [Microsoft][ODBC Driver Manager] The specified DSN contains an
 architecture mismatch between the Driver and Application
```

Cause:

A 32-bit (x86) self-contained deployment published with a 32-bit SDK that uses the in-process hosting model requires that the Application Pool is enabled for 32-bit; while a 64-bit (x64) self-contained deployment that uses the in-process hosting model requires that the Application Pool is disabled for 32-bit.

By default, the Application Pool is enabled for 32-bit.

Solution:

When using a 64-bit database, disable the Application Pool for 32-bit.

To do so, in IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's **Application Pool**. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to **False**.

3 License errors

3.1 Failed to call the license server API

When running the installable cloud app, the Web API console displays the following error:

```
2021-01-27 01:28:32,094 ERROR PowerServer.Client.PowerServerClient.LogMessage [0] -
 MESSAGE: Failed to call the license server Api.
(Error connecting to https://apipsoatest.appeon.com/.well-known/openid-
configuration. A connection attempt failed because the
connected party did not properly respond after a period of time, or established
 connection failed because connected host has failed
to respond..)
```

Cause & Solution:

First, please note that it is possible that the PowerServer console may output the following error information but the application is still running properly. That is because the system allows a grace period in cases when PowerServer fails to validate the license. After the grace period, PowerServer will stop responding the requests from the application.

The error here indicates that PowerServer cannot connect to https://apipsoatest.appeon.com. However, according to the latest information in PB Help, PowerServer is required to connect to https://apips.appeon.com and https://apipsoa.appeon.com, or https://apips.appeon.net and https://apipsoa.appeon.net. The cause of the error must be, the PowerServer Runtime version is too old. You shall manually update the **PowerServer.Core** and **PowerServer.Api** NuGet packages to the latest version.

3.2 Failed to login the license server

When running the installable cloud app, the Web API console displays the following error:

```
2021-01-27 01:28:32,240 ERROR PowerServer.Client.PowerServerClient.LogMessage [0] -
 MESSAGE: License Exception: Failed to login the
license server. (Invalid_client) at
 PowerServer.SessionFacade.CreateSessionIdAsync(String appName, String
 clientEncryptString,
CancellationToken cancellationToken))
```

Cause & Solution:

The "invalid-client" error occurs because the license code is invalid or the license code cached in the system has expired. Please try the following:

1. Check the license code included in the PowerServer project matches the one you obtained from the Appeon website;

2. Clear the PowerServer cache and then build and run the Web APIs again;

   To clear the PowerServer cache, go to %SystemDrive%\Users\[username]\.nuget \packages, and delete the folders starting with "dwnet", "powerserver", "snapobjects", and "powerscript".

3. Make sure that PowerServer is connecting to the correct license servers: https://apips.appeon.com and https://apipsoa.appeon.com, or https://apips.appeon.net and https://

apipsoa.appeon.net. If not, update the **PowerServer.Core** and **PowerServer.Api** NuGet packages to the latest version.

3.3 Cannot access License.json

When the application is deployed to a subfolder under the IIS Web root, the first access to the application always failed while the subsequent access is successful.

Cause:

When tracking the request using Fiddler, the CreateSession request failed at the first access to the application, and the following error message is returned: errmsg=Access to the path 'C:\inetpub\wwwroot\App\AppConfig\License.json' is denied.

Solution 1:

Grant Internet Guest Account and IIS Process Account proper rights to manipulate the Web Root folder. Below are the detailed steps:

1. Right-click on the C:\Inetpub\wwwroot folder. Select the Properties item and select the Security tab page;
2. Add IIS_ISERS(or NETWORK SERVICE) if it is not listed in the box "Group or usernames";
3. Grant Full Control permission to the IIS_ISERS (or NETWORK SERVICE).
4. Restart the IIS server (iisreset.exe).
5. If the issue persists, please try granting "everyone" user full control permission on the C:\inetpub\wwwroot\[appname] folder.

Solution 2:

Refer to the article below to set an account in the administrator group to the Identity property for DefaultAppPool: https://campuslogicinc.freshdesk.com/support/solutions/articles/5000713210-changing-identity-user-for-iis-application-pool.

Restart the IIS server (iisreset.exe).

4 Others

4.1 Failed to update NuGet packages in PowerServer C# solution

The PowerServer C# solution failed to update the NuGet packages; or the PowerServer C# solution failed to build because the dependent NuGet packages were not updated.

Solution:

1) Make sure the computer can connect to the NuGet site (https://www.nuget.org).

2) Clean and then rebuild the PowerServer C# solution.

Performance Guide

Contents

1 Introduction

PowerServer deployments are different from traditional PowerBuilder client/server application deployments in the following ways:

During the PowerServer project compilation and deployments:

- All DataWindows/DataStores are automatically converted to .NET models, and then automatically exposed via REST/JSON APIs;
- All embedded SQLs will be deployed to the server side, and then automatically exposed via REST/JSON APIs;
- All PBD files are broken down very granularly into each individual object/definition file.

When an end user starts an installable cloud app for the first time:

- Each client must download and install a supporting program, Cloud App Launcher, and also download the supporting runtime files;
- Each client will download the app files from the web server. There are two possible ways: Download the app files as necessary, or download all the app files at app startup.
- It is possible that some preload event (e.g., commands for environment detection or control registration) shall be executed before the app starts.

When an installable cloud app starts to run:

- The app has no dependency on a web browser (type, version, or settings), and will run and update itself as needed over the Internet;
- The app runs in a web or cloud environment instead of the previous on-premise environment. It is powered by REST APIs that interface with the data sources, and such REST APIs is hosted in PowerServer in a public or private cloud.

Due to the above differences, it can be well expected that the relevant performance behavior will be different from the client/server applications. This document will provide you some common performance tuning techniques to get maximum performance out of the PowerServer project development and implementation.

2 Performance suggestions on project compilation and deployment

It is normal behavior that the PowerServer Toolkit would take 3 to 4 times as long as a normal compile. There is much additional work to do, including:

- Obtaining database schema for the conversion;
- Converting DataWindows to C# models and static SQLs to C# properties;
- Breaking down all PBD files granularly into each individual object/definition file;
- Encrypting compiled files;
- Deploying the files to the web server, or packaging the files, etc.

There are a few tips for you to speed up the process:

- Use the local machine as the development environment. This means that you set up the Web server, PowerServer, and database server on the local machine, and the database server type and version shall be the same as the one to be used in the production environment.
- Make sure that the option "During compilation, report unsupported PowerScript features for PowerServer deployment" is selected in the application additional properties. The unsupported features report can help you quickly locate and fix problems related with PowerServer deployment.
- You need not sit and wait for the whole process to finish. It is possible to build the PowerServer projects through scripts (see Tutorial 7: Building your PowerServer project with commands).

3 Performance suggestions on loading installable cloud apps for the first time

When an end user starts an installable cloud app for the first time, a number of files will be downloaded from the web server to the client machine, such as Cloud App Launcher, runtime files, the app files, and image files. The downloading performance depends on the network status for sure. Besides, you need to plan the download timing carefully through the relevant PowerServer project settings. Specifically:

- Consider whether to enable or disable the security-strengthening options. These options will add some time marginal time. If judging from the nature of your application, security is not a major concern, you may disable them.
  - In the project settings | General tab, the option "Encrypt all the compiled p-code files"
  - In the project settings | Run Options, the option "Validate the application integrity before the app runs"
- Consider whether to download the files as necessary, or at the app startup. The total time is no different, but you may want to shorten the initial waiting time for the users. The relevant options are:
  - In the project settings | External Files tab, the option to add images and videos in the "Images/videos dynamically loaded" section;
  - In the project settings | Client Deployment tab, the options "Download the app files as necessary" and "Download all the app files at app startup".

    If you select the "Download the app files as necessary", the following files will be downloaded before the app runs: 1) The PowerBuilder Runtime files; 2) The application executable; and 3) The files you selected to be preloaded in the External Files settings. The other files are downloaded only when they are called by the app.

    If you select the "Download all the app files at app startup", the runtime files, app executable, the application files, and external files are all downloaded at the startup, except for the image files that are set as “dynamically-loaded” in the External Files settings.

    Note that usually, if you have already run certain function of an application, when you run it again, there is no need to download any additional files.
- Consider to transfer external files as compressed packaged or in uncompressed format. If an external file/file folder will not change after downloaded to the client, add them to the "Files preloaded as compressed packages" section; if some external file will change often such as config files (XML, INI, etc.), add them to the "Files preloaded in uncompressed format" section.
- Minimize the files that shall be downloaded to the client. For example, in the project settings | Runtime, make sure that only the required runtime modules are selected.

4 Performance suggestions on running installable cloud apps

There are two hypotheses about the performance of running installable cloud apps:

1. The PB application does not have performance problems; but the installable cloud app has.

   In this case, the performance problem may be caused by the connection network.

   And the possible reasons are:

   - The networks connection is slow or unstable;
   - The data package is too large or the SQL syntaxes are not efficient that result in long communication time in a single communication;
   - The same functionality frequently communicates with the server that results in repeated connection performance expense, etc.

   In the case, you should: 1) First consider to reduce the communication times between the client and the server so to reduce the connection performance expense; 2) Secondly, consider to optimize the efficiency of each communication, for example, by retrieving only the necessary data and using the optimal relational calculus in the SQL syntaxes, etc.3) Continue reading the suggestions provided in this chapter and take the suggestions applicable to your application.

2. Both the PB application and the installable cloud app have performance problems:

   If the PB application has performance problems, the deployed installable cloud app will definitely have performance problems as well.

   In this case, you should:

   1) First consider to optimize the performance of the PB application and the database by using all kinds of available system tools. For example, you can use the transaction track analyzer provided by the database provider to analyze and optimize the database performance. Usually, popular database providers provide performance analysis and optimization tools with their databases, you can use these provided tools to optimize the databases.

   2) Secondly, after you make sure that the PB application does not have performance problems, use the hypothesis 1 to analyze the installable cloud app.

4.1 Debugging the performance

If you want to find out what factors/operations cause the performance issue, we suggest that you download Fiddler and use it to track the web traffic between the application and the server, and then locate the web page/operation that is running slowly. For more instructions on how to use Fiddler, please check here: Debugging with Fiddler. Be sure to run the PowerServer Web APIs before you start Fiddler (or any other Web debugging proxy tool). Otherwise, the PowerServer Web APIs will fail to start.

In addition, when the Web APIs is running, you can check the health status of Web APIs by running https://[Web-API-URL]/health-ui in a Web browser, for example, http://

localhost:5009/health-ui/. The health check report contains checking items such as SQL execution performance, the status of local network. It can help you identify the configuration issues or network connection failures affecting the performance.

4.2 Working against the impact of Internet and slow networks on runtime performance

Network chatter and network-intensive code really highlight the weakness of a poor network connection. Any code that results in a server call when executed multiple times sequentially has potential to create network chatter. Here are several common examples of the code that will result in server calls:

- Embedded SQL (Select, Insert, Delete, Update, Cursor);
- Invoking stored procedures or database functions;
- DataWindow/DataStore functions (Retrieve, Update, ReselectRow, ShareData);
- DataWindow/DataStore events (SQLPreview, RetrieveRow);
- Transaction functions (SyntaxFromSQL)
- Invoking a Web Service.

Each of the above statements (except SQLPreview and RetrieveRow) will generate one call to the server. If any of the above statements are contained in a loop or recursive function, well depending on the number of loops, even though it is just one statement it would be executed multiple times generating multiple server calls. Needless to say, loops and recursive functions are some of the most dangerous from a performance perspective.

The reason it is important to minimize server calls is because it can take 100 or even 1,000 times longer to transmit one packet of data over the Internet compared to a LAN. Imagine an event handler is triggered, for example handling an "onClick" event, whose execution will result in 80 synchronous server calls over a LAN with latency of 2 milliseconds (ms). In such scenario the slow-down attributed to network latency would be 0.16 seconds (80 x 2 ms). Now imagine this same event handler running over a WAN with latency of 300 ms. The slow-down attributed to the network latency would be a whopping 24 seconds (80 x 300 ms)! And depending on the amount of data transmitted there could be additional slow-down due the bandwidth bottlenecks.

It is imperative for the developer to be conscious that PowerBuilder applications deployed to the cloud may not be running in a LAN environment, and as such there will be some degree of performance degradation. How much depends on how the code is written, but in most cases the performance degradation still falls within acceptable limits without much performance optimization.

Should you find that certain operations in your application are unacceptably slow, the good news is there are numerous things that you can do as PowerBuilder developers to ensure your PowerBuilder applications perform well in a cloud environment or on slower networks. At a high-level, your code needs to be written such that the server calls and other performance intensive code is minimized or relocated to the middle-tier or back-end.

4.3 Hosting Web APIs and database on the same LAN

Same as any other web applications, for installable cloud apps, the PowerServer Web APIs must be published to a server that locates on the same LAN as the database server. If the database is not on same network as the Web APIs, every request has to go a long way from PowerServer to the database, it is highly possible that there will be performance problem.

4.4 Web API publishing method

To ensure the Web API execution performance, it is strongly recommended that you publish the Web APIs to IIS or Docker or Kestrel. The performance would be much affected if you just use the Compile & Run Web APIs (that is, running the Web APIs from the SnapDevelop IDE). See Tutorials for step-by-step instructions on how to publish the Web APIs.

4.5 Optimizing database server performance

Setting appropriate values for the database parameters based on the actual needs can reduce the occurrence of database deadlock and block hence can improve the concurrency and stability of the Web application.

Common database optimization techniques include: optimizing the table structure, using proper index, and optimizing SQL statement. Additionally, check the following database server settings:

4.5.1 Connection Pooling

Instead of opening and closing connections for every request, connection pooling uses a cache of database connections that can be reused when future requests to the database are required. For example, right now the connection pooling is enabled by default for SQL Anywhere if the database server is on the same machine as PowerBuilder. If SQL Anywhere is on a different machine, you need manually enable the connection pooling.

4.5.2 Command Timeout

Setting appropriate timeout period for commands based on the actual needs can reduce the occurrence of database deadlock and block. You may set the timeout values for transaction, session, and request in the PowerServer C# solution > **ServerAPIs** project > **AppConfig** > **Applications.json** file.

4.6 Tuning excessive server calls

4.6.1 Overview

Excessive server calls in a given operation can create performance issues for that operation on slow and high-latency networks. If you are not familiar with the concept of "server calls", please refer to Impact of the Internet and slow networks and then proceed with this section.

This section will provide three different techniques including code examples to minimize server calls and thereby optimize the performance of your PowerBuilder application for the cloud.

1. Partition transactions utilizing stored procedures
2. Partition non-visual logic utilizing server-side REST APIs
3. Eliminating recursive embedded SQL
4. Eliminating DataWindow computed fields calling user functions that have embedded SQLs

4.6.2 Technique #1: partitioning transactions via stored procedures

Imagine your PowerBuilder client contains the following code:

```
long ll_rows, i
decimal ldec_price, ldec_qty, ldec_amount

ll_rows = dw_1.retrieve(arg_orderid)
for i = 1 to ll_rows
    dw_1.SetItem(i, "price", dw_1.GetItemDecimal(i, "price")*1.2)
next

if dw_1.update() < 0 then
        rollback;
        return
end if

for i = 1 to ll_rows
 ldec_price = dw_1.GetItemDecimal(i, "price")
 ldec_qty = dw_1.GetItemDecimal(i, "qty")

 if ldec_price >= 100 then
  ldec_amount = ldec_amount + ldec_price*ldec_qty
 end if
Next

ll_rows = dw_2.Retrieve(arg_orderid)
dw_2.SetItem(dw_2.GetRow(), "amount", ldec_amount)

If dw_2.update() = 1 then
 Commit;
else
 rollback;
end if
```

This is not only problematic from a runtime performance perspective since there would be numerous server calls over the WAN, but also it could result in a "long transaction" that would tie up the database resulting in poor database scalability.

The business logic and the data access logic (for saving data) are intermingled. When the first "Update( )" is submitted to the database, the related table in the database will be locked until the entire transaction is ended by the "Commit( )". The longer a transaction is the longer other clients must wait, resulting in fewer transactions per unit of time.

To improve the performance and scalability of the application, the above code can be partitioned in two steps:

1. First, move the business logic (or as much possible) outside of the transaction. In other words, the business logic should appear either before all Updates of the transaction or after

Commit of the transaction. This way the transaction is not tied up while the business logic is executing.

2. Second, partition the transaction whereby all the Updates are moved into a stored procedure. The stored procedure will be executed on the database side and only return the final result. This would eliminate the multiple server calls from the multiple updates to just one server call over the WAN for saving all the data in one shot.

It is generally best to actually divide the original transaction into three segments or procedures: "Retrieve Data", "Calculate" (time-consuming logic), and "Save Data". The "Retrieve Data" procedure retrieves all required data for the calculation. This data usually would be cached in a DataWindow(s) or a DataStore(s). In the "Calculate" procedure, the data cached in DataStore will be used to perform the calculation instead of retrieving data directly from the database. The calculation result would be cached back to a DataStore and then saved to the database by the "Save Data" procedure.

Example of the new PB client code partitioned into three segments and invoking a stored procedure to perform the Updates:

```
long ll_rows, i
decimal ldec_price, ldec_qty, ldec_amount
//Retrieve data
dw_2.Retrieve(arg_orderid)
ll_rows = dw_1.retrieve(arg_orderid)
//Calculate (time-consuming logic)
for i = 1 to ll_rows
    dw_1.SetItem(i, "price", dw_1.GetItemDecimal(i, "price")*1.2)
next

for i = 1 to ll_rows
 ldec_price = dw_1.GetItemDecimal(i, "price")
 ldec_qty = dw_1.GetItemDecimal(i, "qty")

 if ldec_price >= 100 then
  ldec_amount = ldec_amount + ldec_price*ldec_qty
 end if
Next

dw_2.SetItem(dw_2.GetRow(), "amount", ldec_amount)
//Save data
declare UpdateOrder procedure for up_UpdateOrder @OrderID = :arg_orderid,
@amount = :ldec_amount;
execute UpdateOrder;
```

Example of code for the stored procedure to Update the database:

```
create procedure up_UpdateOrder(
@orderid integer,
@amount decimal(18, 2)
)
as
begin
update order_detail set price = price*1.2
where ordered = @orderid

if @@error <> 0
begin
    rollback
        return dba.uf_raiseerror()
end
```

```
update orders set amount = @amount
where ordered = @orderid

if @@error <> 0
begin
    rollback
        return dba.uf_raiseerror()
end

commit
end
```

In summary, with the above performance optimization technique, the performance and scalability is improved since the transaction is shorter. The server call-inducing Updates are all implemented on the server-side rather than the client-side, improving the response time. Secondly, moving the business logic out of the transaction further shortens the transaction. If the business logic cannot be moved out of the transaction, one may want to consider implementing the business logic together with the transaction as a stored procedure. In summary, shorter transactions equals better scalability and faster performance.

4.6.3 Technique #2: partitioning non-visual logic via server-side REST APIs

Partitioning non-visual logic and encapsulating it within server-side REST APIs means rewriting the logics in C#, deploying them as REST APIs, and then invoking them from PowerScript. With this technique we have reduced those numerous server calls of the database transaction to just one single call to the REST API, and at the same time created a re-usable component that can be shared by other modules in our PowerBuilder application or shared by other applications.

4.6.4 Technique #3: eliminating recursive embedded SQL

It is actually quite common to find embedded SQL in a loop, especially Select and Insert statements. As explained previously, server calls that are recursive in nature are quite dangerous, potentially generating tremendous number of server calls. If your application requires loops or recursive functions, it would be best to replace any code resulting in server calls with code that does not.

For this technique, we will assume we have Select and Insert SQL statements in a loop. The general idea is to first create a DataWindow/DataStore using the SQL. Then replace the SQL statements contained in the loop with PowerScript modifying the DataWindow/DataStore, which does not result in server calls. If the SQL statement contained in the loop is an Insert statement, we would want to replace that with PowerScript that would insert data into the DataWindow/DataStore. Once all the data has been inserted, then in one shot we would update the DataWindow/DataStore to the database (outside the loop), resulting in only one server call. If the SQL statement contained in the loop is a Select statement, we would retrieve data into a DataWindow/DataStore before executing the loop, and then write PowerScript in the loop to select the desired data from the DataWindow/DataStore.

The following is a code example that increases the price of a specific order by 20%, where embedded SQL is used to update the change row-by-row (hence the loop), and then save those changes to the database:

```
long ll_id
```

```
declare order_detail cursor for
select id from order_detail where orderid = :arg_orderid;
open order_detail;
fetch order_detail into :ll_id;

do while sqlca.sqlcode = 0
 update order_detail set price = price*1.2
 where orderid = :arg_orderid and id = :ll_id;

 if sqlca.sqlcode < 0 then
  rollback;
  return
 end if

 fetch order_detail into :ll_id;
loop
close order_detail;
commit;
```

Now we will replace the embedded SQL with a DataWindow. Specifically, we will cache the data in a DataWindow and update the database with a single DataWindow Update, resulting in just once server call:

```
long ll_rows, i

ll_rows = dw_1.retrieve(arg_orderid)
for i = 1 to ll_rows
    dw_1.SetItem(i, "price", dw_1.GetItemDecimal(i, "price")*1.2)
next

if dw_1.update() = 1 then
commit;
else
rollback;
end if
```

With this technique we have just eliminated server calls from inside the loop, reduced the number of server calls to just one, and created a data caching mechanism at the client-side that can be used to feed data to other controls of the PowerBuilder client.

4.6.5 Technique #4: eliminating DW computed fields calling user functions that have ESQL

If the computed fields in the DataWindow call user functions that have embedded SQLs, for each DataWindow record, the application will need to do a complete round-trip to execute those embedded SQLs. For the PowerBuilder native client/server application, a complete round-trip is "from app to DB"; while for the PowerServer installable cloud application, a complete round-trip is "from app to PowerServer then to DB". Due to the three-tier architecture of the PowerServer installable cloud application, if there are lots of DataWindow records, the performance impact may become significant and noticeable.

To avoid any potential impact, please try the workarounds below:

- Workaround 1: Change the DataWindow SQL to join related tables to get data directly.
- Workaround 2: Retrieve all required data to a DataStore and modify the user function to get data from the DataStore instead of from the database.

4.7 Minimizing large data transmissions

4.7.1 Overview

Suppose you have worked hard to make an application Web-ready using Appeon, and, using your test data, it seemed to perform acceptably. Then, when your users provide "live" test data in realistic volumes, you discover that the application takes a long time to load, and worse, a long time to respond to your user's input. What to do?

Well first you should confirm that your issue is not caused by excessive server calls (see Tuning: Excessive Server Calls). The reason is that majority of the time, PowerBuilder applications are coded such that as additional rows of data are retrieved logic is executed to validate, manipulate, or otherwise handle the data, which can result in server calls. As such, the more rows of data are retrieved the more server calls are made.

Once you are certain the slow-down is not caused by excessive server calls then you can consider reducing the size of data transmission. At a high-level there are several techniques you can employ:

- The first and most popular is staging the data retrieval into manageable increments. For example, you can expose a Next button, and have the application respond to this button click by getting the next logical segment of the result set just like typical Websites or Web applications. Technique #1: retrieving data incrementally gives you instructions on how to achieve this.

- Another technique is to create multiple smaller "specific" views rather than one larger "general" view. Consider adding SQL WHERE clauses based on more search criteria, thus retrieving only the amount of data that is absolutely necessary for a particular view of interest.

- If you have a choice between reducing the number of rows retrieved, and reducing the number of columns, note that a small reduction in columns (described below in Technique #2: minimizing excessive number of columns) can improve performance to an even greater extent than a reduction in rows. This is because most of the time, loops, whether in the application code or in the virtual machine, visit columns first and then rows.

Anything you do to reduce the size of the result set in one way or another can only improve performance and possibly improve usability of your application as well.

4.7.2 Technique #1: retrieving data incrementally

4.7.2.1 For Oracle database server

Oracle includes a pseudo-column called ROWNUM which allows you to generate a list of sequential numbers based on ordinal row. If your application uses Oracle database, apply your Oracle skills and ROWNUM to limit the number of returned rows. For example, this query selects the 10 rows from a table:

```
SELECT *
FROM   (SELECT rownum r, t_dwstyle_grid_employ.empid FROM t_dwstyle_grid_employ)
WHERE  r BETWEEN 10 AND 20;
```

You can impose a NEXT button to the DataWindow. In the Clicked event of the NEXT button, the query changes with ROWNUM increments by 10. Therefore, when the NEXT button is clicked, the DataWindow displays next 10 rows.

4.7.2.2 For all other database servers

If your application uses a non-Oracle database (for example, Microsoft SQL server) you can use the following SQL syntax to limit the number of returned rows to the DataWindow:

```
SELECT TOP 10 *
FROM my_table
WHERE Table.primary_key > = :bottom
ORDER BY Table.primary_key;
```

Before retrieving the first page of data, "bottom" should be set to a value smaller than any primary key value in the table.

Based on this SQL statement, you can implement Next and Previous buttons for the DataWindow. Their Clicked events increment or decrement the bottom variable so that its value matches the primary key value in the first row you want to retrieve then execute the above SQL statement.

4.7.3 Technique #2: minimizing excessive number of columns

As the number of rows in the result set increased, the number of columns will cause greater degradation on performance, especially for nested loops in your application which process rows in the outer loop, and columns in the inner loop. Sometimes the excessive number of columns is intentional and other times it is unintentional.

A sign of unintentionally excessive columns would be the SQL syntax Select * From: consider modifying this syntax to Select fieldList From, where fieldList is the comma-separated list of all, and only, those fields your application will actually need. The performance of the SQL syntax using asterisk will be automatically degraded any time your database administrator modifies the database design by adding columns.

A sign of intentionally excessive columns is simply a long list of columns in your SQL Select statement. Consider analyzing your actual needs to make certain all columns are necessary. It may be possible to request certain columns (needed only in exceptional circumstances) in a separate SQL operation. Please keep in mind if the Visible property of a column is set to zero (the control is not visible), even though the Column cannot be seen, it is still impacting performance.

Debugging Guide

Contents

1 Overview

PowerServer projects have the typical web application structure, consisting of client and server sides. The client-side contains the functionality that a user interacts with, and the server-side deals with the database operations. You need to first decide what operations and functions you will debug, and then choose the right debugging technique and tool. Specifically:

- For client-side operations that have no database interactions, you can continue using the debugging functionality in the PowerBuilder IDE.

  For more instructions, please check the existing documentation: Debugging an application in PowerBuilder User Guide.

- For client-side operations that interact with PowerServer or the database, consider to use a web debugging proxy tool to capture HTTP/HTTPS traffic between the client and the server, to find out where the issue is rooted: the client, the server, or REST API services.

  For more instructions, please refer to: Debugging with Fiddler.

- The unsupported features report and logs during the deployment and running of PowerServer projects can be helpful for locating the causes of the errors occurred during the process.

  For the list of logs and unsupported features report available and their locations, please check Logs and unsupported features report.

Then, there are a few debugging case studies targeting to showcase how to handle real-world issues. Please check the Debugging case studies for the cases that apply certain debugging techniques and tools, and the Troubleshooting Guide document for common errors and their possible solution.

2 Debugging with Fiddler

You may use any web debugging proxy tool that you are familiar with to inspect the traffic between the client-side of an installable cloud application and PowerServer. Telerik Fiddler is one of the options. This section uses Fiddler as the example to explain the relevant techniques.

2.1 Installing Fiddler

Please install Fiddler on the computer that you plan to run and test the installable cloud app deployed from a PowerServer project.

1. In the web browser, navigate to: https://www.telerik.com/download/fiddler

2. Fill the form, accept the license, download and install.

Alternatively, you can download directly from here too:

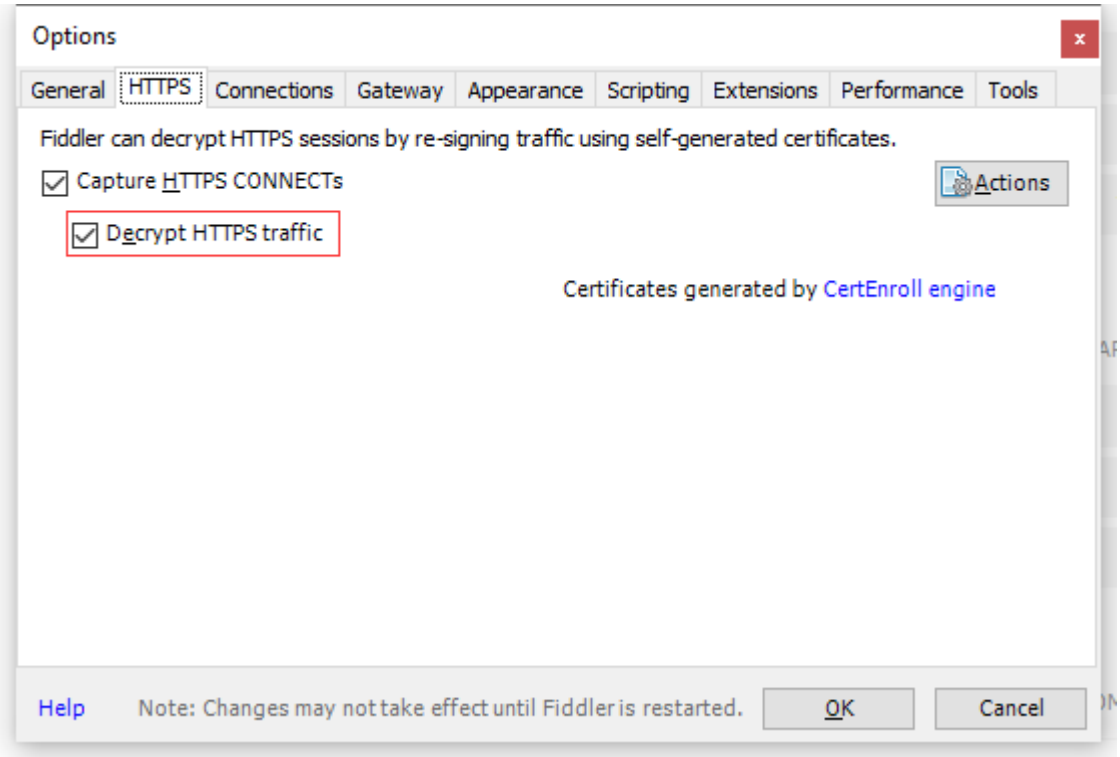
https://telerik-fiddler.s3.amazonaws.com/fiddler/FiddlerSetup.exe

2.2 Configuring Fiddler

The first time you run Fiddler, make sure to enable logging for HTTPS traffic with the following steps:

1. Click Tools > Fiddler Options > HTTPS.

2. Click the Decrypt HTTPS Traffic box.



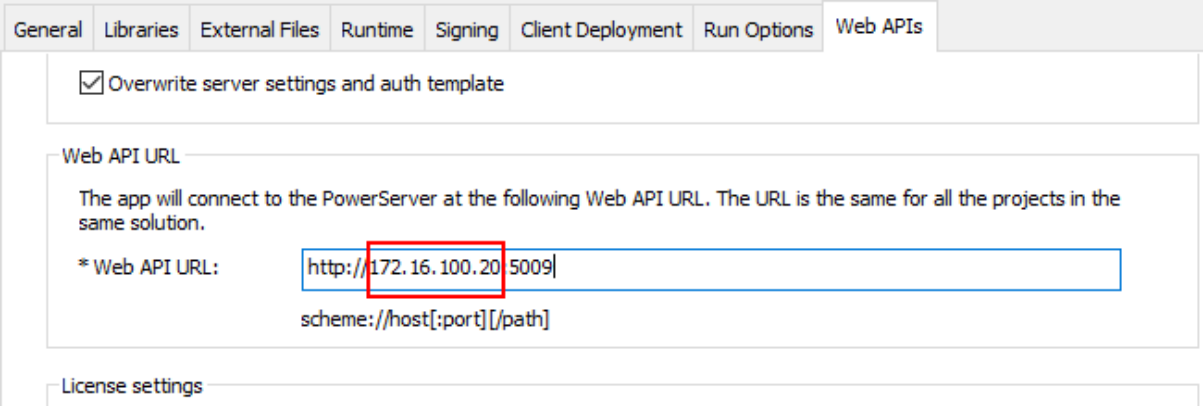
3. In the popup dialog that asks you whether you trust the Fiddler Root certificate, click Yes.

By default, Fiddler does not capture and decrypt secure HTTPS traffic. To capture data sent through HTTPS, the HTTPS traffic decryption must be enabled.

For more configuration settings on Fiddler, you may refer to: https://docs.telerik.com/fiddler/.

2.3 Configuring the PowerServer project

To enable that Fiddler can successfully capture the traffic, make sure that the Web API URL setting of the PowerServer project uses the actual IP address, not "localhost".



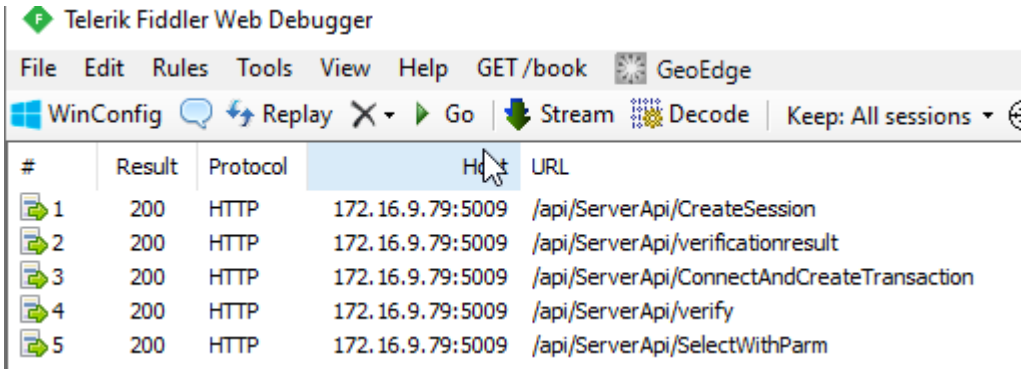
2.4 Running the PowerServer Web APIs and then Fiddler

Be sure to run the PowerServer Web APIs before you start Fiddler (or any other Web debugging proxy tool). Otherwise, the PowerServer Web APIs will fail to start.

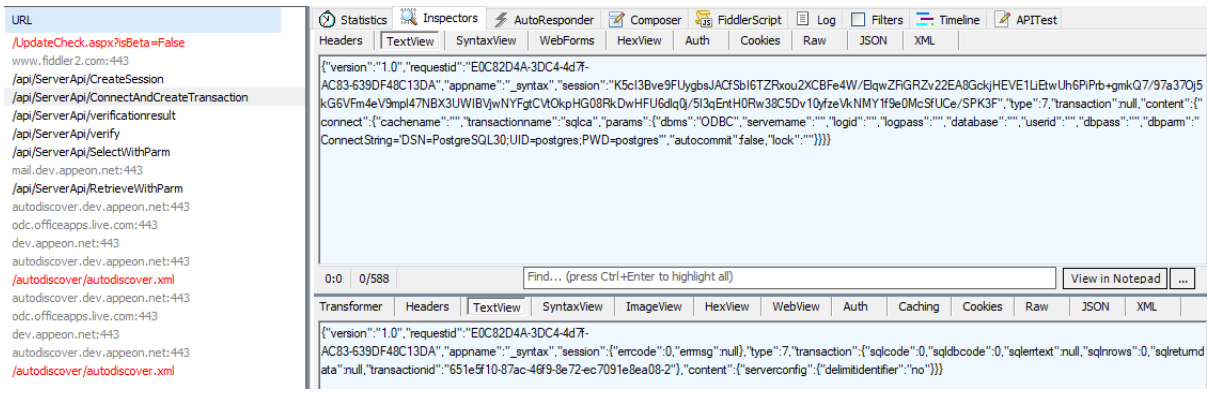
Reason is Fiddler (as well as any other Web debugging proxy tool) works by adding itself as a proxy instead of using your current proxy settings; therefore it will change your proxy settings on startup and reverts them back to what they were when Fiddler is closed. If the PowerServer Web APIs connects with the NuGet site and Appeon site through a proxy server, it may fail to start.

2.5 Capture HTTP(S) with Fiddler

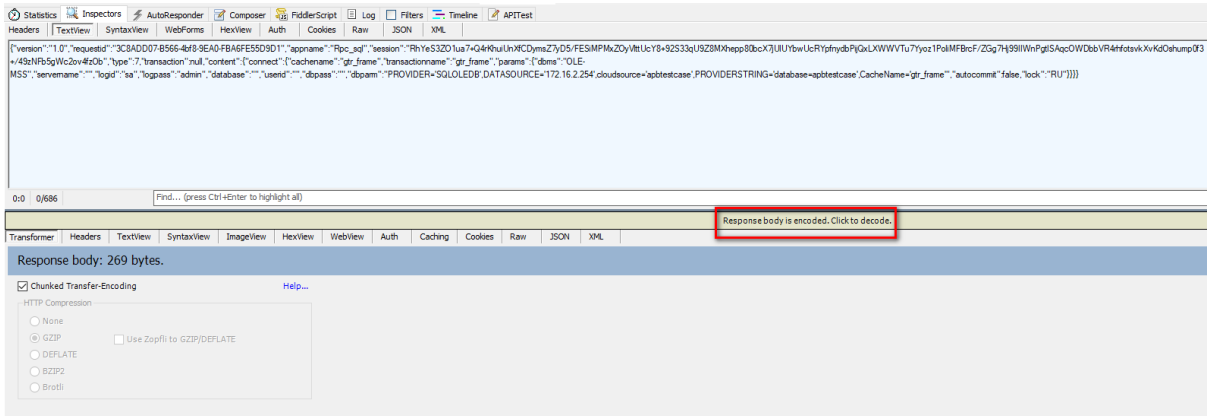
Open up your favorite browser, and simply navigate to the URL of the installable cloud app. You will see the requests sent to PowerServer in the section containing list of sessions, (the left pane)



After selecting one of those sessions, click on Inspectors tab, then the TextView tabs to view the request sent to the server and also the response returned from the server.

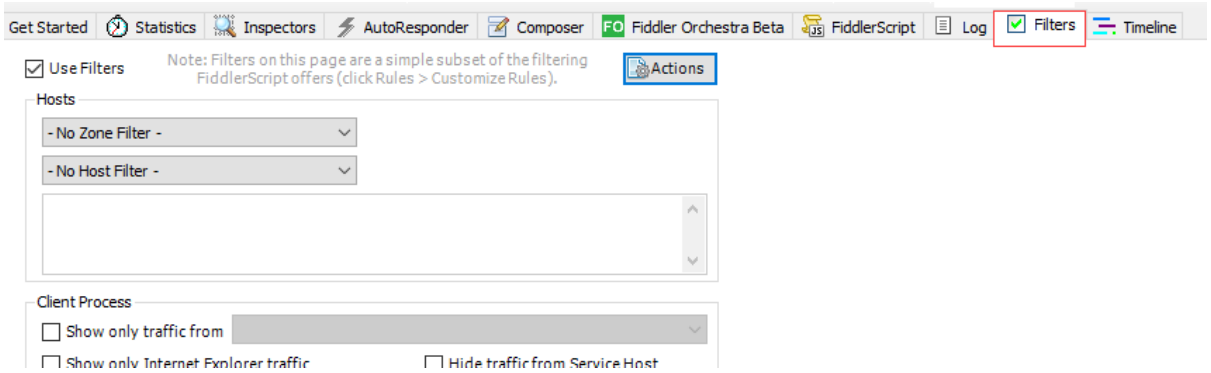


If the response body is encoded, click to decode:



2.6 Filtering the results

It is daunting task to check through hundreds of requests, therefore you may use filters to filter the results.



For example, you can:

- Hide success (2xx) — This rule will remove all of the successful web requests. (An HTTP status code of 200 means success). Usually you do not want to have to look through all of the successes to find the missing files, content expiration intervals that are not properly set, etc.

- Hide Image Requests — There is rarely debugging that can be done on how images are downloaded, you can hide the image requests.

2.7 Inspecting the results

Based on what is reported in Fiddler, you will get what is the next step to take to debug failures in the application.

1. Result: 502

   Result “502” means that Fiddler's request for a web page was blocked (or request delayed) by the site's web server or firewall or load balancer, causing the request to timeout. When it happens, please check whether the connection from the current computer to PowerServer can be successful or not.

2. Result: 404

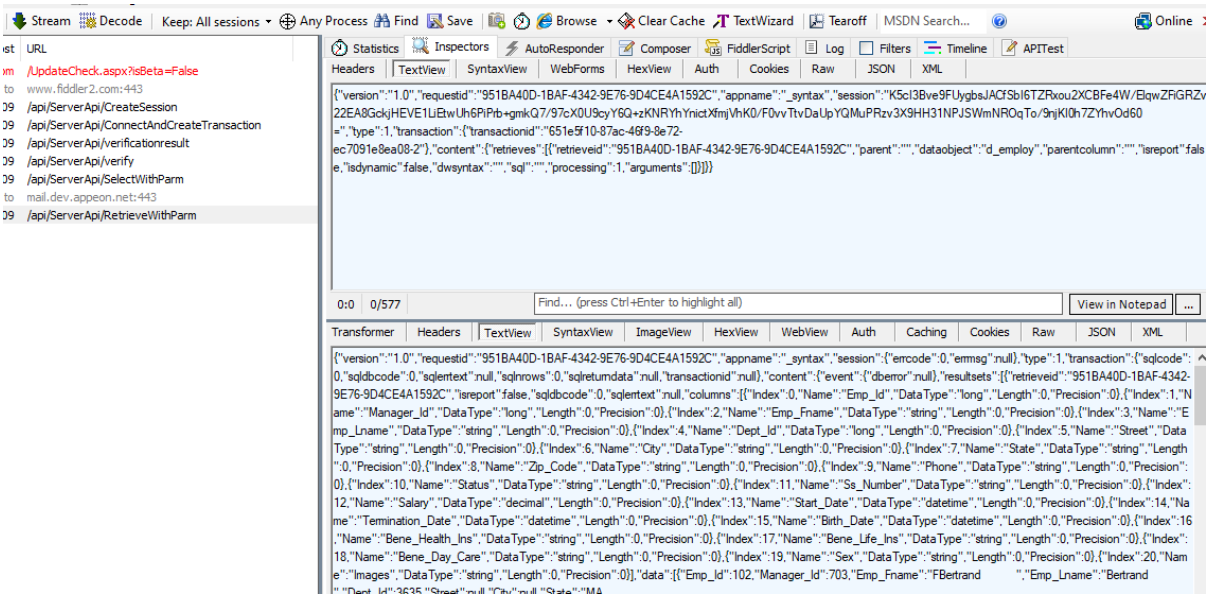
   Result “404” means that the requested item is not found. If it occurs, please check whether the file exists on the web server.

3. Database related SqlCode and SqlErrorText

   In the TextView of the requests, if you find an error with SQLCode or SqlErrorText, it must be something wrong with the database operations. In this case, please further check the relevant code or .cs file behind the request, to see if there is something wrong with the code or the .NET DataStore model (converted from the PowerBuilder DataWindow), or the SnapObjects Runtime.

4. Data retrieval

   You can view the composition of the DataWindow that is performing the data retrieval, and check into possible retrieval errors.



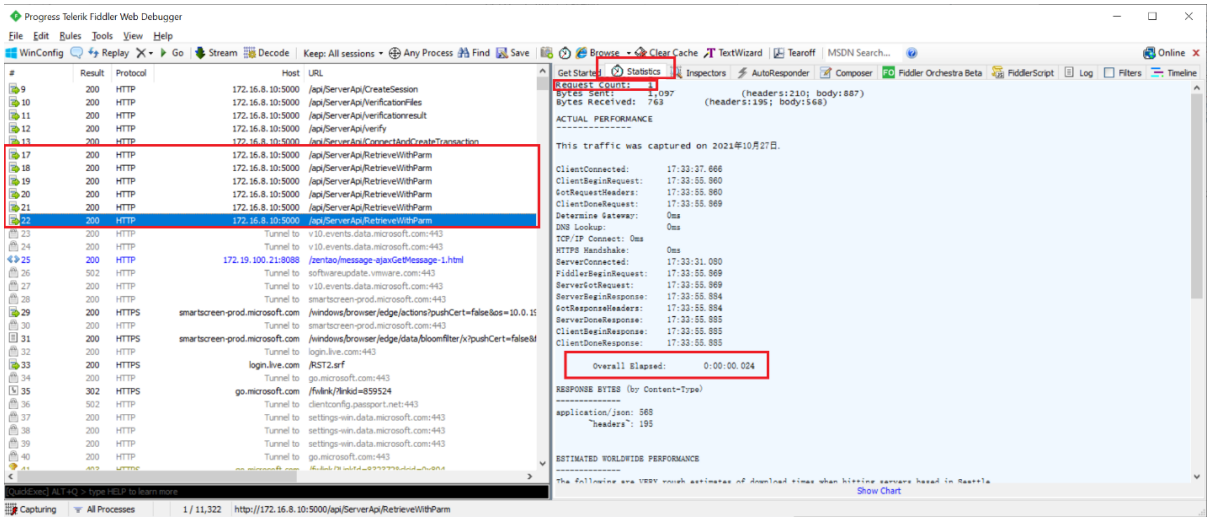
5. Data type mappings

   Pay attention to the data types used in the deployment application or returned by PowerServer when executing SyntaxFromSQL. If the data type is different from what is specified in Data type mapping tables, you may need to make the necessary adjustments to the model .cs file, or possibly in the original SQL.

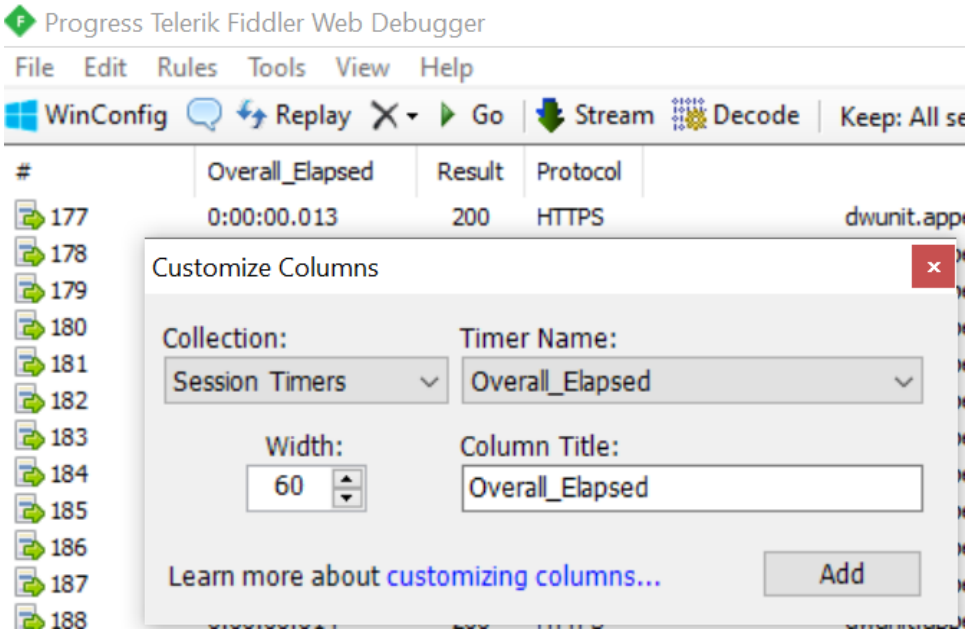
2.8 Analyzing the performance

You can analyze the performance of a module by understanding the information in the Fiddler's Statistics tab, especially:

- Request count
- Overall elapsed



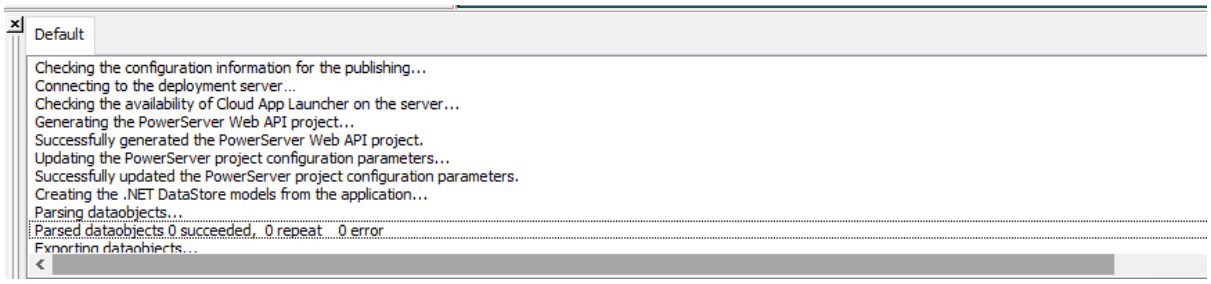
You can also add Overall Elapsed as a custom column, for the convenience of view.



3 Logs and unsupported features report

3.1 Deployment log

When you deploy an application, the output panel shows all the build and deployment actions occurred during the process.

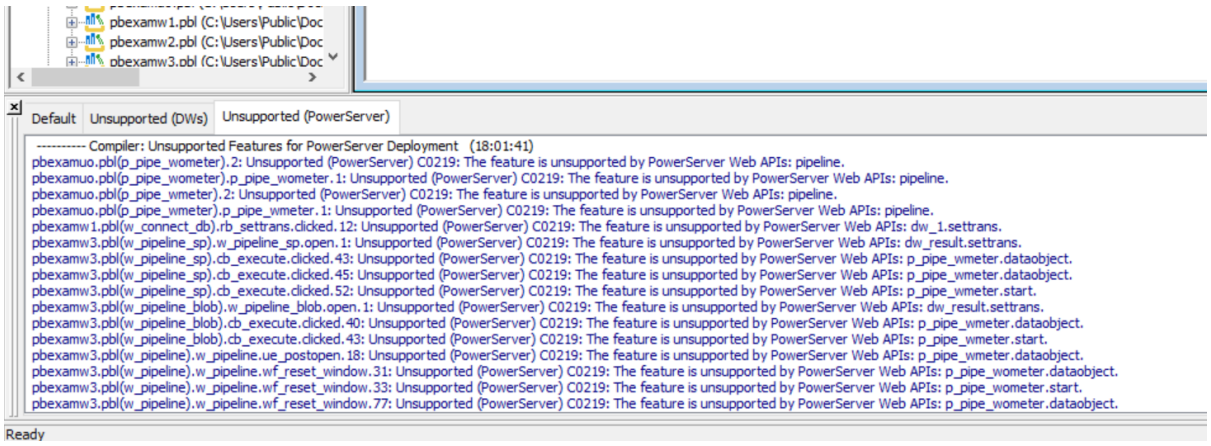


You may also find the log file at \Program Files (x86)\Appeon\PowerBuilder 21.0\log.

3.2 Unsupported features report

If you enable the “During compilation, report unsupported PowerScript features for PowerServer deployment”, PowerServer Toolkit will catch and report PowerScript features that are currently unsupported by PowerServer. You can make changes into PowerScript accordingly.

View the unsupported PowerScript features in Output > Unsupported (DWs) tab or Unsupported (PowerServer) tab:



3.3 Web file download log

When you launch an installable cloud app at the client side, Cloud App Launcher is first downloaded and installed, and then the app files. The download log of all the web files can be found at the client side, the Cloud App Launcher:

For Cloud App Launcher without background service, the log file CloudAppShell.log is at %LocalAppData%\Launcher\log;

For Cloud App Launcher with background service, the log file CloudAppShell.log is at %LocalAppData%\LauncherWithService\log;

The app file download logs are at %AppData%\PBApps\Applications\[appname]\log.

3.4 Web API request log

You can enable logging for the PowerServer Web API requests at the client, by adding the following setting to the PB.INI file (this INI needs to be deployed to the server):

```
[PowerServer]
Api_log = 1
```

The log file (such as api202106241530.log) will be generated at %AppData%\PBApps \Applications\[appname]\log.

3.5 Debugging log in SnapDevelop

If you select Start Debugging in SnapDevelop to start ServerAPIs.exe, when an installable cloud app calls PowerServer services, the Output panel in SnapDevelop will show the relevant debugging information.

Tip: You can look for "SQL:" keyword in the output for the SQL syntax that PowerServer sends to the database for execution.

3.6 PowerServer logs

3.6.1 Log4net logging

The PowerServer Web APIs adopts the Log4net logging framework for logging. The PowerServer log files are stored in the Web API bin folder (make sure you have permissions to write into the folder), for example, \ServerAPIs\bin\Debug\netcoreapp3.1\log. The **ServerAPIs** project > **Logging** > **log4net.config** file controls which folder under \bin\Debug \netcoreapp3.1\log to save the log files in and the log file name. For example:

```
<file value="Logging/logs/powerserver.log" />
//The log file will be rolled based on a size constraint (maximumFileSize)
<maximumFileSize value="100KB" />
```

3.6.2 Logging with the settings in Logging.json

The **ServerAPIs** project > **Logging** > **Logging.json** or **Logging.Development.json** file contains 1) the log level of PowerServer; 2) the logs to display in the console window; 3) the logging of SQLs, sessions and transactions.

Settings in **Logging.json** file will take effect in the production environment (for example, when Web APIs is published and running in IIS, docker etc.). The default log level is warning.

Settings in **Logging.Development.json** file will take effect in the development environment (for example, when Web APIs is running from the SnapDevelop IDE or the PowerBuilder IDE). The default log level is information.

The event levels (in the order of severity) include Trace, Debug, Information, Warning, Error, Critical, and None. The level change can be made at runtime (no need to restart the server).

For example, if you do not want to output logs at the Information level (more detailed level), you can change the log level in the file from "Information" to "Error".

The following is a sample log:

```
2021-03-05 09:09:09,080 ERROR PowerServerApi.ServerApiController.LogMessage [0] -
 MESSAGE: RequestId: CE1A2E41-0C93-4ff9-80FB-F98B096D4176,
ErrorMessage: The INSERT statement conflicted with the CHECK constraint
 "check_age". The conflict occurred in database "Qa_datawindow", table
"dbo.t_update_forcheck", column 'age'.
The statement has been terminated.
```

4 Debugging case studies

4.1 DataWindow related errors

4.1.1 DataWindow retrieve error

An error has occurred when executing the following SQL statement via DataWindow retrieve.

SQL syntax:

```
select  id typeid,
    "Fname"||' - '||"Lname" as FullName,
    diner + interval '1 day' dinneAdd1,
    costs* 0.85 *(case when id%2 = 0 then 1 else -1 end ) zk,
    birthday + interval '10 day' ,
    salary * 1.1 sales,
    to_char(cast(mobilephone as int),'###-####-###')
    from t_dwstyle_alltype a
    where /*DATEDIFF(d,birthday,'1000-02-01') between 8 and 10*/
    EXTRACT(day from birthday - to_timestamp('1000-02-01','YYYY-MM-DD')) > 8 and
 EXTRACT(day from birthday - to_timestamp('1000-02-01','YYYY-MM-DD')) < 10
    or birthday is null
    order by typeid
```

When the DataWindow retrieves data using the SQL, an error message pops up: The property does not allow null value: Dinneadd1.

Debugging technique:

The error means that the model generated from the DataWindow has set non-nullable attribute to the column. However, the Dinneadd1 column is a computed column, so it shall be nullable. It is necessary to check the .cs file of the generated model, to find the column and set nullable attribute to it.

For example, the column defined in the model is:

```
[DwColumn("dinneadd1")]
        public TimeSpan Dinneadd1 { get; set; }
```

Add the nullable attribute to the column:

```
[DwColumn("dinneadd1")]
        public TimeSpan? Dinneadd1 { get; set; }
```

Note: If the SQL syntax contains left join, or union, it may possibly induce similar error.

4.1.2 SyntaxFromSQL execution error

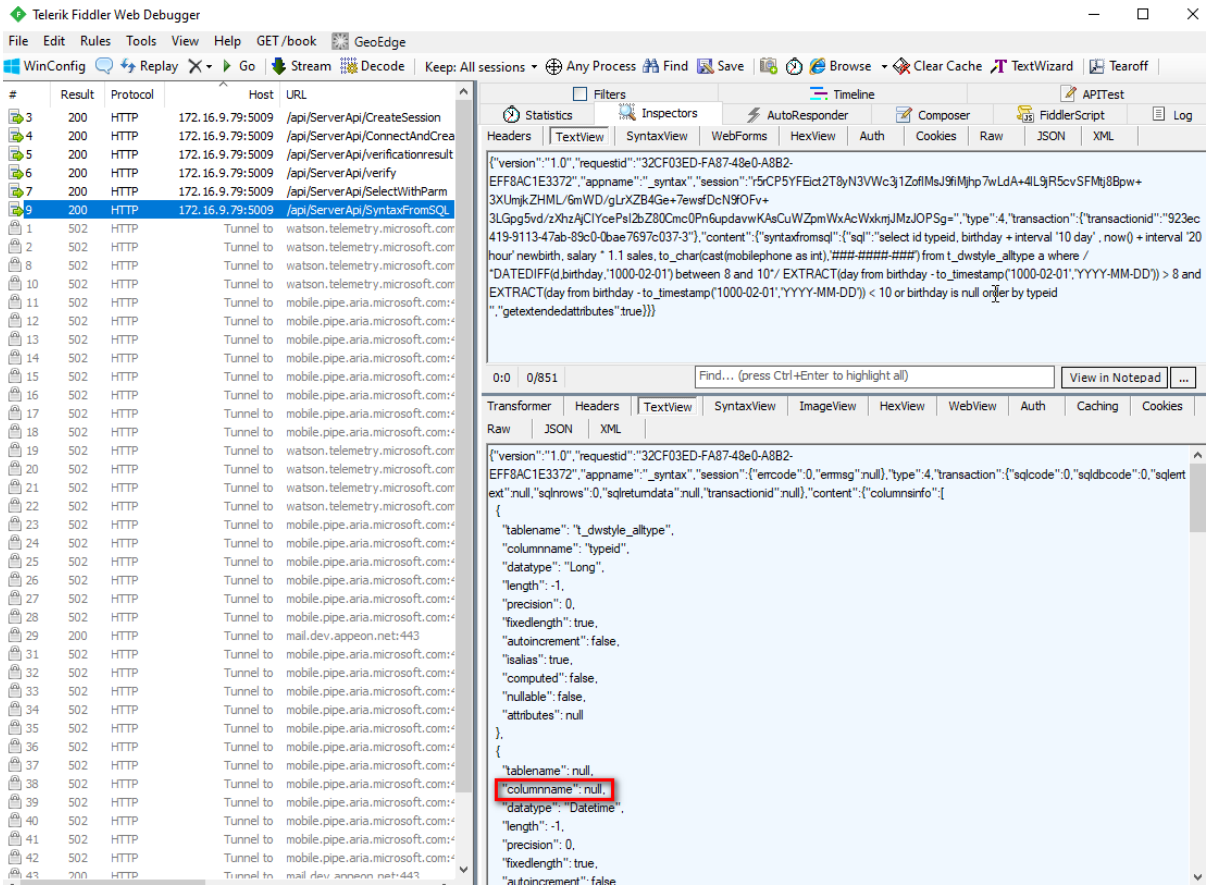
The application crashes when executing the same SQL statement via SyntaxFromSQL.

Debugging technique:

We shall first check the web debugging proxy tool, such as Fiddler, to locate at what operation the error occurred (at the execution of SyntaxFromSQL), and also find out the error message captured. Check in the Inspector TextView for the possible causes:

- Is there SqlErrorText?

- Is any column name empty? --- Check the columnname;
- Any improper date type? --- Check the datatype;
- Issue with the column length or precision?



Then you will see there is a null column name. After setting an alias to the null column, SyntaxFromSQL can then be executed successfully:

Column in the SQL:	birthday + interval '10 day'
Add an alias to the column:	birthday + interval '10 day' rq

4.1.3 Different execution results in different databases

Supposing we are executing the following CREATE TABLE syntax in SQL Server and Oracle.

```
CREATE TABLE appeon_test (id integer NOT NULL,testname varchar(40) NOT NULL ,
 testdate date  , testnumber decimal(12,3)  , PRIMARY KEY (id))
```

With Oracle, the table created is as below:

APPEON_TEST

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(38,0)	No	(null)	1	(null)
2	TESTNAME	VARCHAR2(40 BYTE)	No	(null)	2	(null)
3	TESTDATE	DATE	Yes	(null)	3	(null)
4	TESTNUMBER	NUMBER(12,3)	Yes	(null)	4	(null)

With SQL Server, the table created is as below:

SERVER2254.appeont...- dbo.appeon_test

Column Name	Data Type	Allow Nulls
id	int	☐
testname	varchar(40)	☐
testdate	datetime	☑
testnumber	decimal(12, 3)	☑
		☐

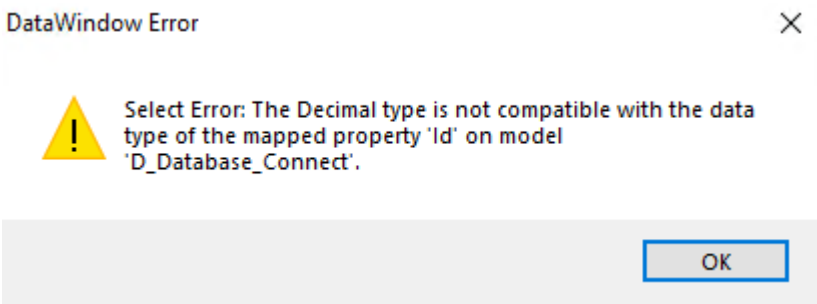
In the PowerBuilder DataWindow SRD, the datatype is long, which can work well in both databases.

```
table(column=(type=long update=yes updatewhereclause=yes key=yes name=id
 dbname="appeon_test.id" )
```

When converting the DataWindow to C# model, with SQL Server, the Id column is of int type:

```
[DwColumn("appeon_test", "id")]
        public int Id { get; set; }
```

Because the Id column of the table is Number in Oracle, when the same model tries to retrieve data from the Oracle database, an error occurs:



Therefore, if using the Oracle database, the model Id shall be changed to the decimal data type.

```
[DwColumn("appeon_test", "id")]
        public decimal Id { get; set; }
```

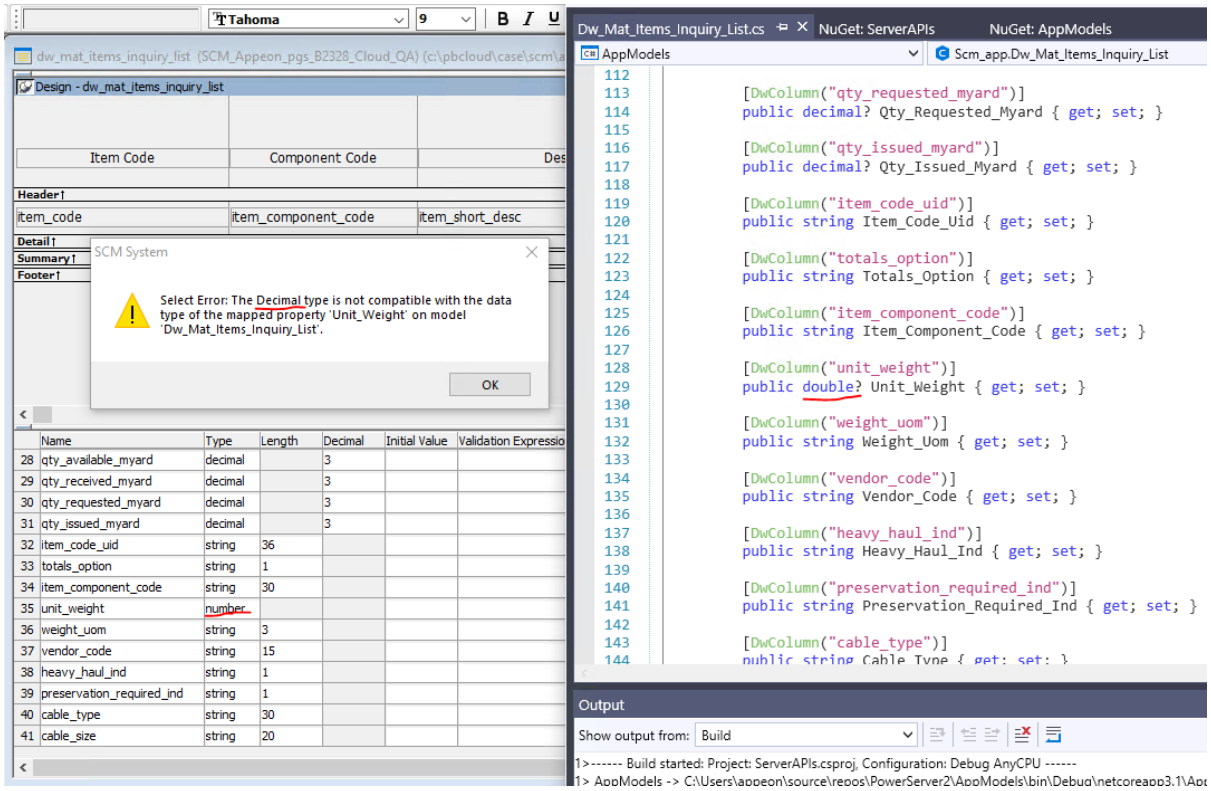
If you hope to run the same model against different databases, it is necessary to add ValueConverter too the model column in the .cs file by:

```
[ValueConverter(typeof(DefaultValueConverter))]
        [DwColumn("appeon_test", "id")]
        public int Id { get; set; }
```

4.1.4 Incompatible data type

With PostgreSQL, when retrieving data into a DataWindow that uses stored procedure as its data source, an error occurred:

Select Error: The Decimal type is not compatible with the data type of the mapped property 'Unit_Weight' on model 'Dw_Mat_Items_Inquiry_List'.



Debugging technique:

Search for the model ”Dw_Mat_Items_Inquiry_List” in SnapDevelop and then “Unit_Weight” in the model .cs file. In the stored procedure of the DataWindow, the data type of Unit_Weight is NUMBER (12,3). According to the Data type mapping tables, the Number data type is mapped to decimal. Therefore, the data type of Unit_Weight shall be changed to decimal in the .cs file:

```
Public decimal? Unit_Weight { get; set; }
```

4.1.5 PBSELECT retrieve error

Sample PBSELECT script:

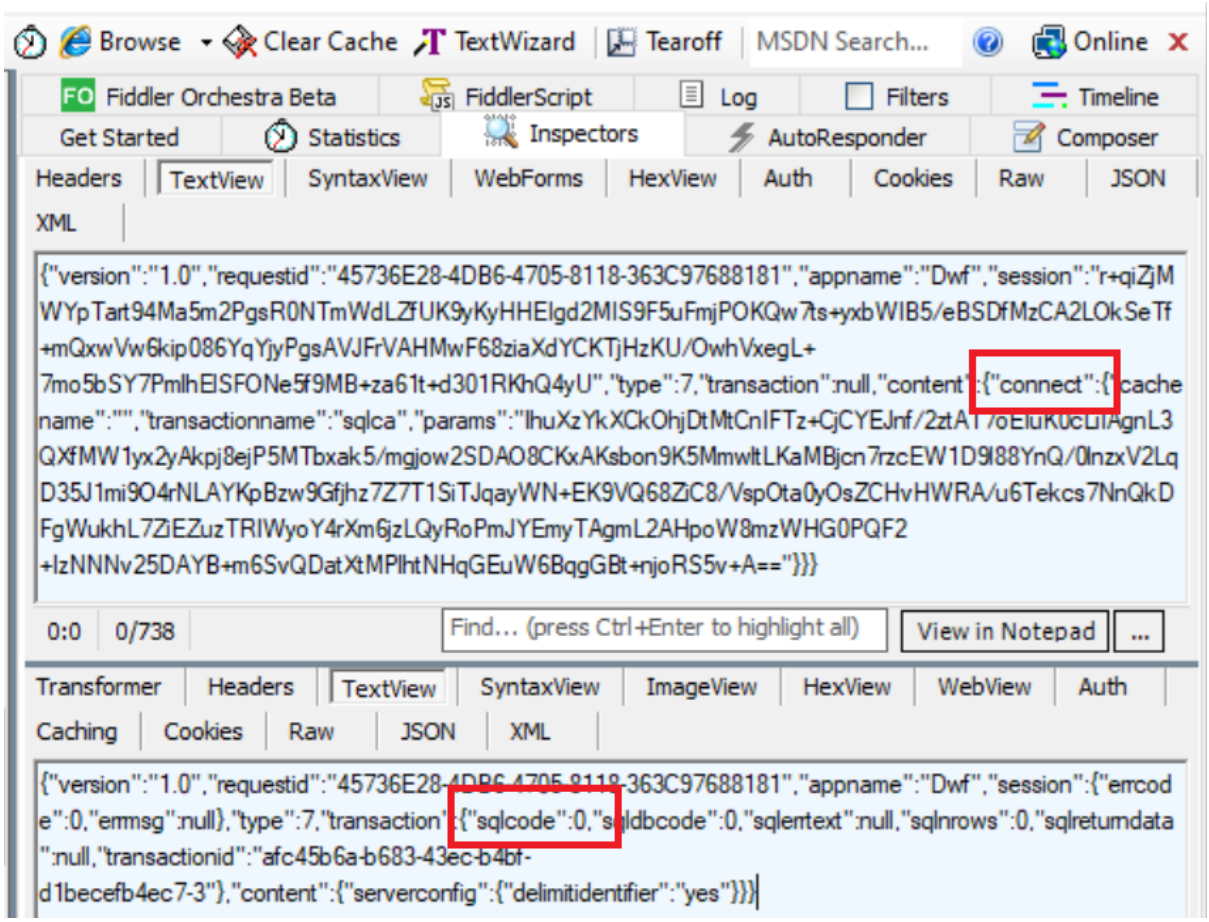
```
retrieve="PBSELECT( VERSION(400) TABLE(NAME=~"dec_emp~" )
```

PBSELECT may easily cause errors. When the SQLPreview event (executed at the client side) converts the PBSELECT to the SELECT syntax, the event may cause the client crash or arrive at incorrect SQL syntax (the syntax relies on the DisableBind, DelimitIdentifier settings in dbparm).

If PBSELECT contains outer joins of multiple tables, there may be unknown error when it is converted to SELECT, and such error cannot be identified by Fiddler because there is no communication from the client to the server.

Debugging technique:

Check in Fiddler whether a connect has occurred, and whether the connection is successful.



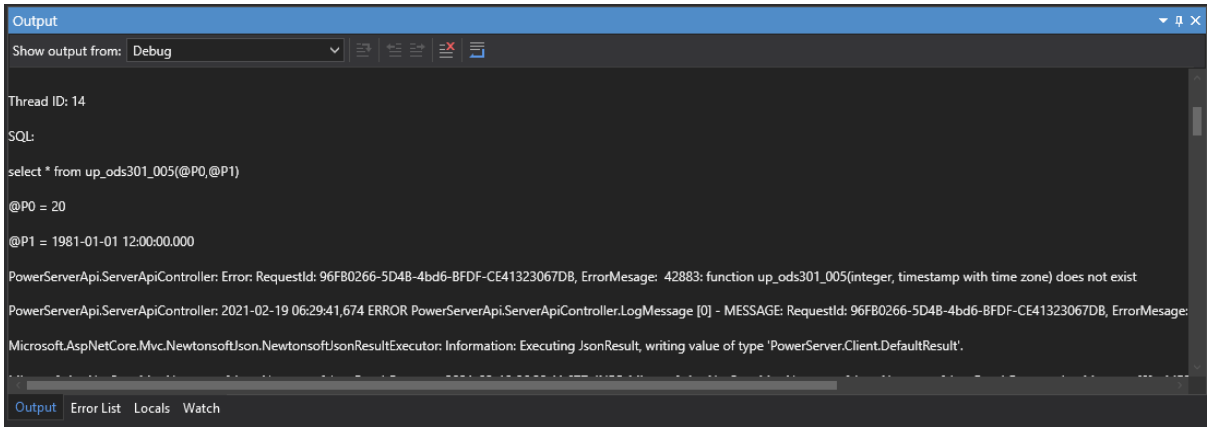
If there is no connection and the client app has crashed, the issue may be caused by PBSELECT. To avoid the problem, better change PBSELECT to a SELECT statement in the application source code, and then deploy the application again.

4.2 Embedded SQL related errors

Possible error when executing an embedded SQL: ErrorMesage: 42883: function up_ods301_005(integer, timestamp with time zone) does not exit

Debugging technique:

When an embedded SQL reports error, the recommended way is to run the server Web APIs in debug mode, and check the SQL statement in the Output panel.



For example, the original embedded SQL is:

```
select * from up_ods301_005(20,'1981-01-01');
```

And the actual statement shown in the Output is:

```
select * from up_ods301_005(@P0,@P1)
@P0 = 20
@P1 = 1981-01-01 12:00:00.000
```

You can then notice that the data type of the second parameter does not match in the two statements (the first one is data, and the second one is timestamp with time zone) (for more accurate data type information, use the web debugging proxy tool).

5 Data type mapping tables

This section provides the mapping rules from the original data type to the C# data type, or the data type returned by PowerServer on executing SyntaxFromSQL. If the C# data type or the type returned by PowerServer is different from what is listed in the tables, you would need to make necessary changes to the C# models or the SQL syntax, to avoid possible errors.

5.1 SQL server data type mappings

Data type in SQL Server	Data type in C# models	Data type returned by PowerServer on SyntaxFromSQL
bigint	long	decimal
binary	blob	blob
bit	bool	number
char	string	char
date	datetime	date
datetime	datetime	datetime
datetime2(7)	datetime	datetime
datetimeoffset(7)	DateTimeOffset	char
decimal(18, 2)	decimal	decimal
float	double	number
geography	blob	blob
geometry	blob	blob
hierarchyid	blob	blob
image		Blob
int	int	long
money	decimal	decimal
nchar	string	char
ntext	string	char
numeric(18, 2)	decimal	decimal
nvarchar	string	char
real	Single	real
smalldatetime	DateTime	datetime
smallint	short	long
smallmoney	decimal	decimal
sql_variant	object	char
text	string	char(32766)
time(7)	TimeSpan	time

timestamp	byte[]	timestamp
tinyint	byte	long
uniqueidentifier	Guid	char
varbinary	blob	blob
varchar	string	char
xml	string	char(32766)

5.2 ASE server data type mappings

Data type in ASE	Data type in C# models	Data type returned by PowerServer on SyntaxFromSQL
bigdatetime	DateTime	
bigint	decimal	Decimal(0)
bigtime	DateTime	datetime
binary	byte[]	char
bit	bool	number(1)
char	string	char
date	DateTime	date
datetime	DateTime	datetime
decimal	decimal	decimal
float	double	number
image		blob
int	int	long
longsysname	string	char
money	decimal	decimal
nchar	string	char
numberic	decimal	decimal
nvarchar	string	char
real	float	real
smalldatetime	DateTime	datetime
smallint	short	long
smallmoney	decimal	decimal
sysname	string	char
text	string(32000)	char
time	TimeSpan	time
timestamp	byte[]	timestamp
tinyint	byte	long

unichar	string	char
unitext	string	char
univarchar	string	char
unsigned bigint	decimal	decimal
usigned int	long	ulong
unsigned smallint	int	ulong
varbinary	byte	char
varchar	string	char

5.3 SQL Anywhere server data type mappings

Data type in SQL Anywhere	Data type in C# models	Data type returned by PowerServer on SyntaxFromSQL
bigint	long	decimal(0)
binary		blob
bit	bool	number
char	string	char
date	datetime	date
datetime	datetime	datetime
datetimeoffset	string	char
decimal(18,2)	decimal	decimal(2)
double	double	number
float	single	real
image		blob
integer	int	long
long binary		blob
long nvarchar	string	char
long varbit	string	char
long varchar	string	char
money	decimal	decimal
nchar	string	char
ntext	string	char
numeric(18,2)	decimal	decimal(2)
nvarchar	string	char
real	single	real
smalldatetime	datetime	datetime
smallint	short	long

smallmoney	decimal	decimal
text	string	char
time	timespan	time
timestamp	datetime	datetime
timestamp with time zone	string	char
tinyint	byte	long
uniqueidentifier	guid	char
uniqueidentifierstr	string	char
unsignedbigint	decimal	decimal
unsignedint	long	UnsingedLong
unsignedsmallint	int	UnsingedLong
varbinary(50)		blob
varbit(50)	string	char
varchar(50)	string	char
xml	string	char
sysname	string	char

5.4 Oracle server data type mappings

Data type in Oracle	Data type in C# models	Data type returned by PowerServer on SyntaxFromSQL
BINARY_DOUBLE	double	BinaryDouble
BINARY_FLOAT	single	BinaryFloat
BLOB		Blob
CLOB	string	Char
CHAR	string	Char
DATE	datetime	Datetime
INTERVAL DAY(2) TO SECOND(6)		
INTERVAL YEAR(2) TO MONTH		
LONG	string	Char
NCLOB	string	Char
NVARCHAR2	string	Char
RAW		Blob
TIMESTAMP(6)	datetime	Datetime

TIMESTAMP(6) WITH LOCAL TIME ZONE	datetime	Datetime
TIMESTAMP(6) WITH TIME ZONE	datetime	Datetime
VARCHAR2	string	char
NUMBER(2)	short	int16
number		Decimal
NUMBER(5,2)	single	Single
number(10)		Int64
number(10,2)		double
number(7)		int32
number(15)		int64

5.5 PostgreSQL data type mappings

Data type in PostgreSQL	Data type in C# models	Data type returned by PowerServer on SyntaxFromSQL
bigint	long	decimal(0)
bigserial	long	decimal(0)
bit	bool	char
bit varying		char
boolean		char
Box		char
bytea		blob
character	string	char
character varying	string	char
cid	Uint32	char
cidr		char
circle		char
date	datetime	date
daterange		char
double precision	double	number
gtsvector		char
inet	ipaddress	char
int2vector		char
int4range		char
integer	int	long

interval		char
json	string	char
line		char
macaddr	physicaladdress	char
money	decimal	number
name		char
numeric(10,1)	decimal	decimal(1)
numrange		char
oid		char
path		char
point		char
polygon		char
real	float	real
refcursor		char
serial	int	long
regdictionary		char
smallint	short	long
smallserial	short	long
Text	string	char
tid		char
time with time zone(6)	datetimeoffset	char
time without time zone(6)	timespan	time
timestamp with time zone(6)	datespan	datetime
timestamp without time zone(6)	datespan	timestamp
tsquery		char
tsrange		char
uuid	guid	char
xid		char
xml	string	char