# Salesforce App-Owns-Data Embedding Sample

**[SalesforceAppOwnsDataEmbedding](https://github.com/PowerBiDevCamp/SalesforceAppOwnsDataEmbedding/tree/main/SalesforceAppOwnsDataEmbedding/force-app/main/default/" \t "_blank)** is a sample project which demonstrates how to implement App-Owns-Data embedding with Power BI reports. This project has been created using the [**Salesforce Developer Experience (SFDX)**](https://developer.salesforce.com/developer-centers/developer-experience/) and the [**Salesforce CLI**](https://developer.salesforce.com/tools/sfdxcli). The goal of this sample project is to provide guidance and demonstrate best practices to developers who need to implement Power BI embedding in a Salesforce environment.

App-Owns-Data embedding has a big advantage over User-Owns-Data embedding when developing for Salesforce. More specifically, App-Owns-Data embedding does not require each user to have an Azure AD organizational account and a Power BI license. When developing with the App-Owns-Data embedding model, your Salesforce users can remain unknown to Power BI and your code has the flexibility to embed reports for any users you want.

## Project Architecture

The architecture of this solution is built on top of an Apex class named [**PowerBiEmbedManager**](https://github.com/PowerBiDevCamp/SalesforceAppOwnsDataEmbedding/blob/main/SalesforceAppOwnsDataEmbedding/force-app/main/default/classes/PowerBiEmbedManager.cls) which is programmed to interact with both Azure AD and the [**Power BI REST API**](https://docs.microsoft.com/en-us/rest/api/power-bi/) as shown in the following diagram.



**PowerBiEmbedManager** implements [**Client Credentials Flow**](https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-client-creds-grant-flow) when it interacts with Azure AD to acquire an app-only access token. App-only access tokens are important because they makes it possible to call the Power BI REST API under the identity of a service principal instead of calling under the identity of a user. Making calls to the Power BI REST API as service principal is a best practice for developing with App-Owns-Data embedding.

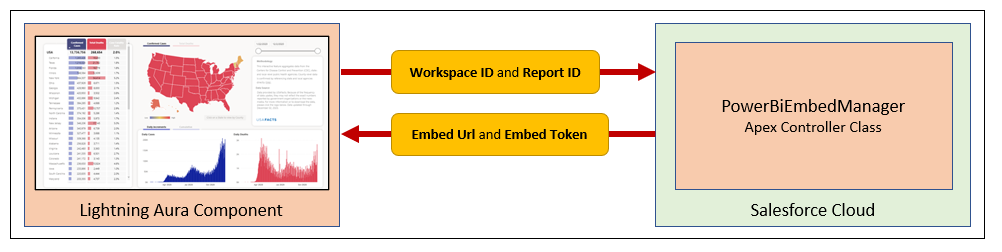
**PowerBiEmbedManager** must call the Power BI REST API for two different reasons. First, it much acquire embedding data such as the Embed Url associated with a specific report ID. Second, **PowerBiEmbedManager** must call the Power BI REST API to generate embed tokens which are required with App-Owns-Data embedding.

**PowerBiEmbedManager** has been designed as a controller class by exposing a public **getEmbeddingDataForReport** method which has been marked with the **AuraEnabled** annotation making it accessible to Lighting Aura components and to Lightning web components. A client-side component can call **getEmbeddingDataForReport** to retrieve the embedding data and the embed token for a specific report.

The **SalesforceAppOwnsDataEmbedding** project contains a Lighting Aura component named [**powerBiReportAura**](https://github.com/PowerBiDevCamp/SalesforceAppOwnsDataEmbedding/tree/main/SalesforceAppOwnsDataEmbedding/force-app/main/default/aura/powerBiReportAura). When you add an instance of the **powerBiReportAura** component to a Lightning application page, you must configure it with the **Workspace ID** and the **Report ID** for a specific report in a Power BI workspace. This design makes it possible to add multiple instances of the **powerBiReportAura** component and configure each instance to embed a different Power BI report.



Once you have configured a **powerBiReportAura** component instance with a **Workspace ID** and **Report ID**, these two configuration values will be passed as parameters when the component calls **getEmbeddingDataForReport**. The **PowerBiEmbedManager** class responds to this remote function call by returning the embed URL and the embed token which will be used to embed a report in the browser.



Once the **powerBiReportAura** component has successfully called **getEmbeddingDataForReport**, it has acquired the embedding data and the embed token it needs to embed a report on the hosting web page. In a final step, the **powerBiReportAura** component executes JavaScript code in the browser and uses the [**Power BI JavaScript API**](https://docs.microsoft.com/en-us/javascript/api/overview/powerbi/overview) to implement the report embedding process.



When a Power BI report is embedded on a Web page such as a Lightning application page, it establishes a direct connection back to the Power BI Service. Once the report has loaded, the user can begin to interact with the report by setting filters and navigating between pages. As users interact with the report, these interactions are handled by direct communications between the report and the Power BI Service.



## Setting Up the Test Environment

In order to set up and test this sample project, you'll need a Power BI report in a Microsoft 365 tenant in which you can create a new Azure AD application. If you need to create a free trial Microsoft 365 environment for testing with Azure AD and Power BI, you can do so using this [Microsoft 365 trial sign-up page](https://github.com/PowerBiDevCamp/Camp-Sessions/raw/master/Create%20Power%20BI%20Development%20Environment.pdf). You'll also need a Salesforce development environment. If you don't already have a Salesforce development environment, you can sign up for one for free using the [Salesforce lightning platform signup page](https://developer.salesforce.com/signup).

Once you have a Salesforce organization for testing, you will need to complete the following setup tasks to configure the environment for the **SalesforceAppOwnsDataEmbedding** project.

* Add remote site settings for Azure AD and Power BI REST API
* Create an Azure AD application to call the Power BI REST API
* Create a custom metadata type to store client credentials for the Azure AD application
* Create a static resource by uploading a copy of the Power BI JavaScript API (powerbi.js)

### Add Remote Site Settings

Start by navigating to the Salesforce organization **Setup page** and searching for the **Remote Site Settings** page. Use the Remote Site Setting to add remote site settings for **https://login.microsoftonline.com** and **https://api.powerbi.com**. Once again, these configuration settings are required so that code in the Apex class can execute HTTP requests to Azure AD and the Power BI REST API.

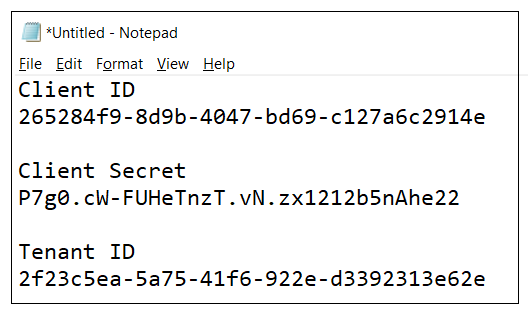
Graphical user interface, text, application, email

Description automatically generated

### Create the Azure AD Application

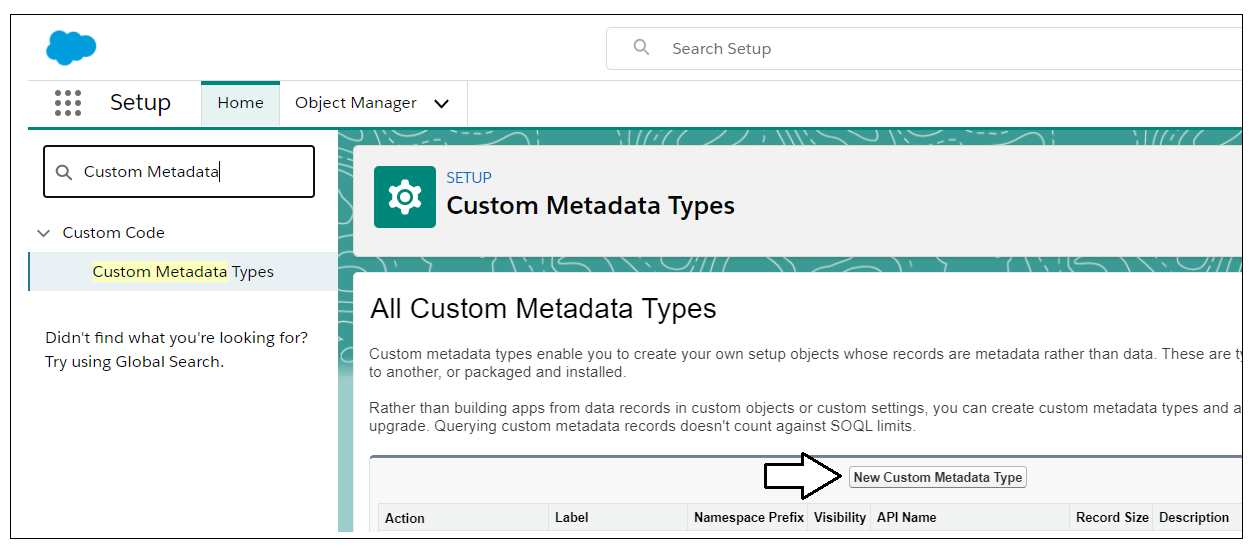
After adding the remote site settings, the next step is to create and configure a new Azure AD application to support calling the Power BI REST API as a service principal. You can create the required Azure AD application by following the steps in [**Create an Azure AD Application for App-Owns-Data Embedding**](https://github.com/PowerBiDevCamp/SalesforceAppOwnsDataEmbedding/blob/main/Create%20an%20Azure%20AD%20Application%20for%20App-Owns-Data%20Embedding.md).

Once you have completed the steps to create and configure the new Azure AD application, you should have a **Client ID**, **Client Secret** and **Tenant ID** that you will need in the next step where you will create a custom metadata type to store these client credentials.



### Create the Custom Metadata Type

The next step is to create a custom metadata type that will be used to track the client credentials. More specifically, the custom metadata type will be designed to track three values including **Client ID**, **Client Secret** and **Tenant ID** that are required to authenticate with Azure AD as a service principal. Start by navigating to the Salesforce organization **Setup page** and searching for the **Custom Metadata Types** page. Once you get to the **Custom Metadata Types** page, click the **New Custom Metadata Type** button to open the **New Custom Metadata Type** page.



In the **New Custom Metadata Type** page, enter the following values.

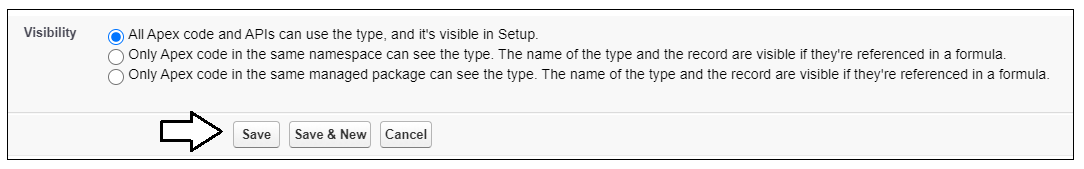
* **Label** = **Power BI Auth Setting**
* **Plural Label** = **Power BI Auth Settings**
* **Object Name** = **Power\_BI\_Auth\_Setting**

The **New Custom Metadata Type** page should now match the following screenshot.

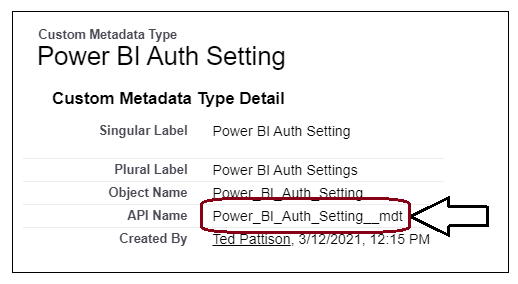
Graphical user interface, text, application, email

Description automatically generated

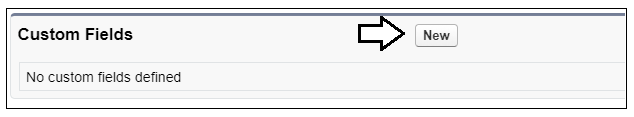
Below in the **New Custom Metadata Type** dialog there is a **Visibility** setting. Leave the **Visibility** setting at it's default value of **All Apex code and APIs can use the type and it’s visible in Setup**. This is the setting you want when testing a POC application in a development environment. However, it's common to change this setting for a custom metadata type in a production environment to store sensitive data like a Client Secret in a more secure and encrypted fashion. Click the **Save** button to create the new custom metadata type.



The best thing about creating a custom metadata type is that it automatically becomes an actual type in the Apex programming language. You can determine what the type name to use in your Apex code by looking at the **API Name** property which in this scenario is **Power\_BI\_Auth\_Setting\_\_mdt**. As you can see, the **API Name** is generated by taking the **Object Name** you provide and appending **\_\_mdt** at the end.



After you have initially created the new custom metadata type, you must add custom fields for the Azure AD application's **Client ID**, **Client Secret** the **Tenant I**D. Click the **New** button in the **Custom Fields** section to begin adding fields.



Create the following three new custom fields.

* Create a new **Text** field named **ClientId** with unique values and a max of 36 characters
* Create a new **Text** field named **ClientSecret** with max of 255 characters
* Create a new **Text** field named **TenantId** with max of 36 characters

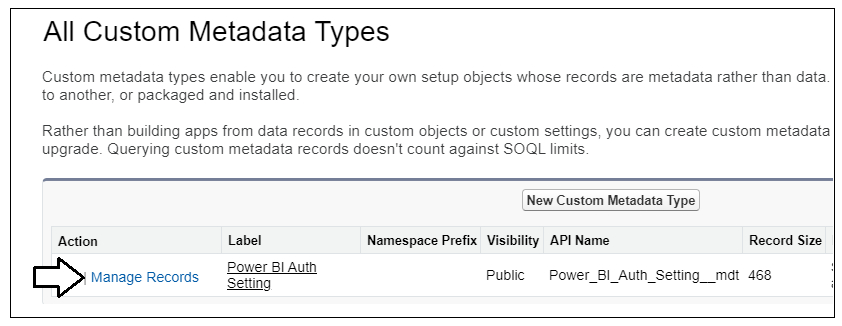
Note that when you add custom fields that they will be automatically added as public fields to the custom metadata type. The **API Name** values created for these fields will be **ClientId\_\_c**, **ClientSecret\_\_c** and **TenantId\_\_c**.

Graphical user interface, text, application

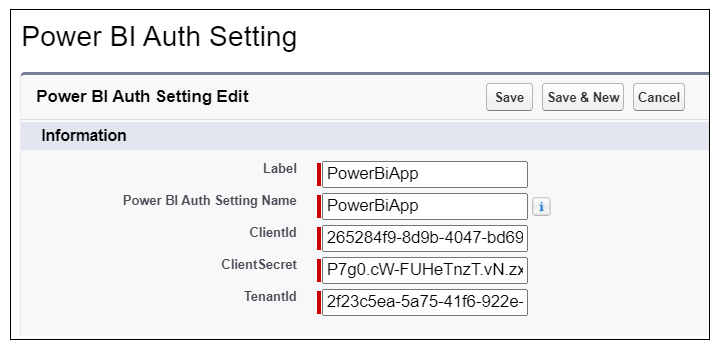
Description automatically generated

At this point you have created the custom metadata type. Next, you are going to create a new record using the custom metadata type to track a single set of client credentials for testing.

Return back to the **Custom Metadata Types** page. You should see a new entry for the new custom metadata type with a Label value of **Power\_BI\_AUTH\_SETTING**. Click on the **Manage Records** button to create a new record.



You should now see an entry form where you can enter values for the **ClientId**, **ClientSecret** and **TenantId**. Enter a value of **PowerBiApp** for both the **Label** and the **Power BI Auth Setting Name**. Enter the values you collected earlier for the **ClientId**, **ClientSecret** and **TenantId**. When you are done, click **Save** to create the new record with the client credentials.



When you return back to the **Power BI Auth Settings** page, you can see that the new record has a **Power BI Auth Settings Name** of **PowerBiApp**. You will need this value in the Apex code that needs to retrieve these field values from this record.

Graphical user interface, application

Description automatically generated

Now that you have create a custom metadata type and an associated record to store client credentials, you should understand the benefits. It's very easy to access the record of a custom metadata type when programming in Apex. You can now write the following Apex code to retrieve the client credentials when they are needed for authentication.

// get auth settings from custom metadata type record

Power\_BI\_Auth\_Setting\_\_mdt authSetting = Power\_BI\_Auth\_Setting\_\_mdt.getInstance('PowerBiApp');

string TenantId = authSetting.TenantId\_\_c;

string ClientId = authSetting.ClientId\_\_c;

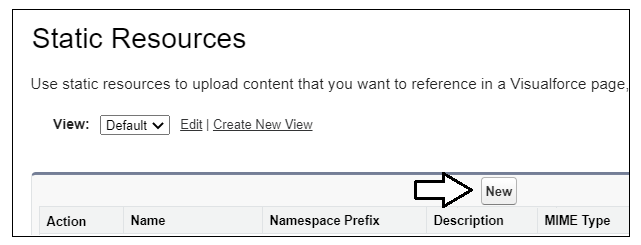
string ClientSecret = authSetting.ClientSecret\_\_c;

Now there is just one more task remaining in the set up process. This is you must upload the source file for the Power BI JavaScript API (powerbi.js) into your Salesforce organization as a static resource.

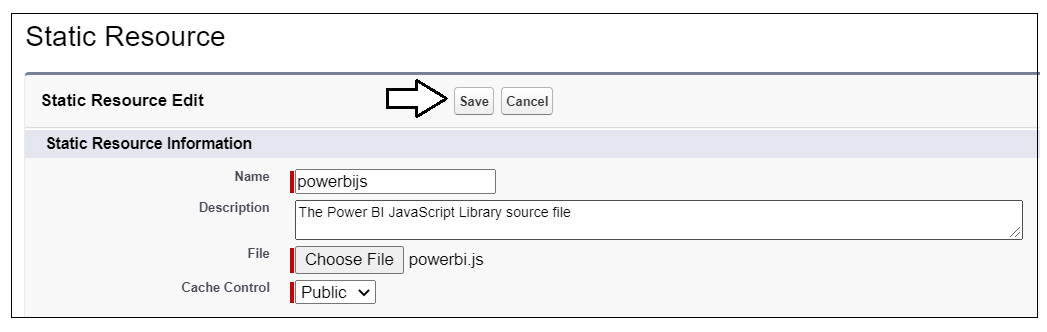
### Upload powerbi.js as a Static Resource

First, you must download a local copy of **powerbi.js** which is the main JavaScript source file for the Power BI JavaScript API. This can be done using [this ink](https://raw.githubusercontent.com/microsoft/PowerBI-JavaScript/master/dist/powerbi.js). Once you have downloaded a local copy, you can then upload it to your Salesforce organization as a static resource.

Start by navigating to the Salesforce organization **Setup page** and searching for the **Static Resources** page. While it's easy to miss, there is a **New** button on this page. Click the **New** button to create a new static resource.



Give the new static resource a **Name** of **powerbijs**. Upload the source file named **powerbi.js** by clicking the **Choose File** button and then locating the file in the **File Open** dialog. Set the **Cache Control** to **Public** to enable sharing and faster load times for this JavaScript library file in the browser. When you have entered all the data, click **Save** to upload **powerbi.js** and make it available as a new static resource named **powerbijs**.



When you return to the **Static Resources** page, you should be able to verify that the static resource has been created and that it's name is **powerbijs**. This name will be important because it must be used by any Lightning component that needs to load this library.

Graphical user interface, text, application, email

Description automatically generated

## Setting Up the Developer Workstation

In order to setup and run this sample, you need to install the following software.

* [Install Node.JS](https://nodejs.org/en/download/)
* [Install Visual Studio Code](https://code.visualstudio.com/Download)

When you have installed Visual Studio Code, you must install a Visual Studio Code extension the Salesforce Expansion Pack.



References

* [Salesforce Extensions Documentation](https://developer.salesforce.com/tools/vscode/)
* [Salesforce CLI Setup Guide](https://developer.salesforce.com/docs/atlas.en-us.sfdx_setup.meta/sfdx_setup/sfdx_setup_intro.htm)
* [Salesforce DX Developer Guide](https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_intro.htm)
* [Salesforce CLI Command Reference](https://developer.salesforce.com/docs/atlas.en-us.sfdx_cli_reference.meta/sfdx_cli_reference/cli_reference.htm)

### Download the Project Source Code

Here is the GitHub repo with the sample code discussed in this article. This code is provided in an SFDX project. This is not an introduction to Salesforce development. It is expect the reader either knows the fundamentals or is willing to learn the fundamentals. Salesforce has done a great job at providing developer material at places such as trailhead.

Here are the Salesforce features

* Apex controller class
* Custom Metadata Type
* Remote Site Settings
* Lightning Aura component
* Lightning Web Component

### Project Structure

What's inside



### Apex Class Code Walkthrough

xxxxxx

### Lightning Aura Component Walkthrough

xxxxxx

### Lightning Web Component Walkthrough

Xxxxxxeb

### The Tip of an Iceberg

xxx