Developing with Power BI Embedding

In the three years since its initial release, Power BI has become widely-adopted by corporations and other organizations as a Software-as-a-Service (SaaS) application. The Power BI Service provides licensed users with the ability to access and interact with Power BI reports and dashboards through the *Power BI portal* which most users access through the URL of https://app.powerbi.com. The Power BI Service also provides licensed users with a first-class mobile experience using the *Power BI Mobile apps* that Microsoft has published for iPhone, Android and Windows 10. Microsoft promotes Power BI along with PowerApps and Flow as the primary services in the Microsoft Business Application Platform which offer customers a *no code, low code* approach to building custom business solutions.

In addition to its SaaS offerings, Power BI also provides developers with Platform-as-a-Service (PaaS) capabilities which make it possible to embed Power BI resources into custom applications. By learning to use a set of APIs created by the Power BI team at Microsoft, a developer can embed Power BI reports, dashboards and dashboard tiles into custom web applications that target the browsers, tablets and mobile devices.

This whitepaper focuses on the details and the developer skills required for developing with Power BI embedding and explains the essential concepts and terminology. As you will see, there are quite a few details you must learn when you get started. However, once you understand the landscape of Power BI embedding, you will be able to effectively extend the custom applications and components you are developing with interactive reports and dashboards.

**Prerequisites:** This whitepaper assumes the reader is already familiar with using the Power BI Service and with building and publishing PBIX files using Power BI Desktop.

# Introduction to Power BI Embedding

Let's begin by examining how Power BI embedding works at a fundamental level. From an architectural perspective, Power BI embedding involves adding an iframe to a web page and configuring the iframe with a URL and a security token to load a Power BI report or dashboard directly from the Power BI Service. Figure 1 depicts a simple web page in a custom web application with an embedded Power BI report.

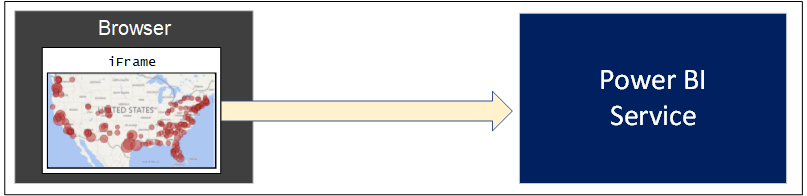


Figure 1: Power BI embedding is based on rendering a Power BI resource inside an iframe element.

The underlying architecture for Power BI embedding has been designed using open web standards including HTML5, CSS, JavaScript, OData, OAuth and OpenID Connect. This means that Power BI embedding techniques can be used by developers using a wide variety of programming languages and development platforms. These open standards also allow custom applications which use Power BI embedding to provide a wide reach to target any type of modern browser as well tablets and mobile devices.

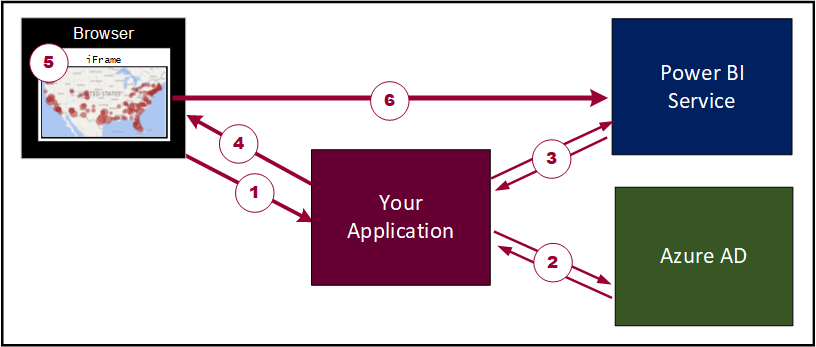


Figure 2: The basic set of steps required to implement Power BI embedding.

The majority of applications that leverage Power BI embedding techniques are developed using a common pattern. The diagram in Figure 2 shows the typical sequence of steps involved when a custom application embeds a Power BI report on a web page. Let's walk through the steps.

1. User launches the application.
2. The application calls to Azure AD to obtain an access token to call the Power BI Service.
3. The application calls the Power BI Service API to retrieve embedding data about a specific Power BI resource.
4. The application passes the embedding data and a security token to client-side code running in the browser.
5. Client-side code in the browser calls the Power BI JavaScript API to embed the Power BI resource.
6. The Power BI JavaScript API dynamically creates the iframe and initializes the embedded resource.

There is an important observation here. Once a Power BI report has been embedded in the iframe, it has a direct connection back to the Power BI Service. This allows the users of your application to interact with the embedded report using familiar Power BI report features such as slicers and bookmarks.

## Embeddable Resources

The PaaS features of the Power BI Service continue to evolve offering support for new types of embedded resources. Currently, the Power BI Service supports embedding for the following types of Power BI resources.

1. Power BI reports
2. Power BI dashboards
3. Power BI dashboard tiles
4. Power BI Q&A experience
5. Power BI report visuals

While you can embed several different types of Power BI resources, the list of supported features for each type is quite different. More specifically, embedded reports support quite a few extra features that are not supported by any of the other types. For example, you can embed a report in edit mode allowing the user to update the report layout and to save these changes back to the Power BI Service. It is also possible to embed a new report on top of an existing Power BI dataset allowing the user to create a report from scratch.

In a simple *embed-it-and-forget-it* scenario, you can just embed a report and rely on the underlying report to supply its own interactive behavior using slicers, highlighting, drillthrough pages and bookmarks. But you can go far beyond that. Once you master the skills of developing with Power BI embedding, you'll be able to extend the interactive behavior of an embedded report by writing client-side code to set custom filters, apply bookmarks and set custom page layouts. Many developers today are using this approach to embed Power BI reports inside a user interface experience with a custom navigation scheme or a custom filtering experience.

Embedded dashboards do not provide the same level of interactive behavior when compared to embedded reports. For example, clicking on a tile inside an embedded dashboard does not redirect the user to a report page as it does when accessing the same dashboard through the Power BI portal. However, embedded dashboards do exhibit a modest amount of interactive behavior because hovering over a tile will still generate popup tooltips.

There is good news if you are working with real-time dashboards which you have built on top of streaming datasets, push datasets or hybrid datasets. You can embed a real-time dashboard in a custom application and it will continue to update itself automatically just as with dashboards that are accessed through the Power BI portal.

When you embed a dashboard, it is an all-or-nothing proposition when it comes to which tiles are displayed. But you also have the option to embed dashboard tiles individually. This can be handy when you want to select which tiles from a dashboard are displayed or you want more flexibility as to where each dashboard tile is displayed. One other thing to keep in mind that you cannot embed a tile from a real-time dashboard and see the updates in real time. When you are working with real-time dashboards, you must embed the entire dashboard and not individual dashboard tile.

The Q&A Experience is the newest arrival to the supported list of embeddable resources. If you have used the Q&A Experience in the Power BI portal, then you can imagine what the experience would look like when isolated by itself on your custom web page. When you embed the Q&A Experience, you must configure it by referencing an underlying dataset. You can also seed questions into the Q&A Experience to provide the user a starting point that will return results and display an initial visualization.

Now let's discuss how to embed a report visual. In truth, you cannot really embed a report visual by itself. Instead, you play a little trick. You embed the report which contains the visual and apply a custom layout to hide every other visual in the report except for the visual you want to display. You can implement a custom layout to reset the position, width and height of the visual so it occupies all the real estate inside the hosting iframe.

Custom report layouts can be applied dynamically after an embedded report has already loaded. This can lead to using custom layouts in creative ways. For example, let's say you embed a report with a custom layout that displays a single visual. You could then extend the user experience to provide some interactive control where the user can move from one visual to the next to iterate through all the visuals in the report. The user interface experience of switching between visuals will be very fast because the entire report has already been loaded into memory.

## First-Party Embedding versus Third-Party Embedding

When initially designing a custom application which will implement Power BI embedding, you must decide between two development models: *first-party embedding* versus *third-party embedding*. Whichever of these two development models you choose will determine which user account will be used to authenticate with Azure AD to obtain the access token needed to call into the Power BI Service.

First-party embedding is known as the *user-owns-data* model. That's because you implement an Azure AD authentication flow to obtain an access token for the current user and you use the current user's identity and permissions to call into the Power BI Service API. First-party embedding is used in scenarios in which all users have an Azure AD user account and a Power BI user license. For example, a company using Office 365 and Power BI might decide to use first-party embedding to develop a custom Intranet-style application which surfaces Power BI reports and dashboards to an audience of users within a single Azure AD tenant.

While first-party embedding is a good fit for organizations using Power BI and Dynamics 365, it is not as useful for independent software vendors (ISVs) and other types of companies that provide SaaS applications to their customers. Third-party embedding is more attractive to ISVs because it can be used in scenarios in which users have neither Power BI licenses nor Azure AD user accounts. Therefore, an ISV can use third-party embedding in commercial applications that use a custom identity provider or some other identity provider service other than Azure AD.

ISVs like third-party embedding because there's no need to authenticate users with Azure AD or to acquire a Power BI license for each user. However, the lack of user licensing with third-party embedding begs an important question. If you are not paying Microsoft for Power BI through user licensing, how do you pay them for using the PaaS capabilities of the Power BI Service? The answer to that question involves dedicated capacities which are introduced in the next section.

There is a very important security aspect of Power BI embedding that differs between applications that use first-party embedding versus third-party embedded. While you must pass a security token to the browser to properly initialize the iframe with the embedded resource, the type of security token used between these two scenarios is different. When you use first-party embedding, you embed resources using an access token you retrieve from Azure AD when authenticating the current user.

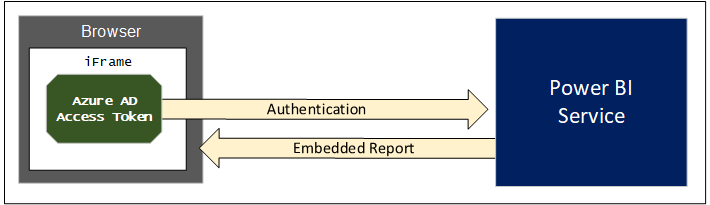


Figure 3: First-party embedding involves embedding a resource using an Azure AD access token.

When using third-party embedding, you do not authenticate the current user with Azure AD, so you need to acquire a security token by some other means. This is accomplished by calling into the Power BI Service API to acquire a different type of security token known as an *embed token*.

There is a key observation here. An embed token is far more constrained than an Azure AD access token. That's because an embed token is specific to a single Power BI resource such as a report or a dashboard, When generating an embed token, you can specify the permissions you are granting such as view, edit and create. When using third-party embedding, you will generate a separate embed token for each Power BI resource that you need to embed on a web page.

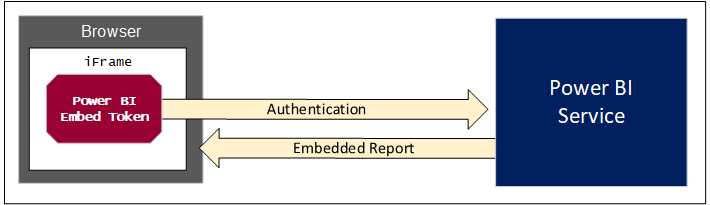


Figure 4: Third-party embedding involves embedding a resource using an Power BI embed token.

## Dedicated Capacities

It has never been a goal of Microsoft to become non-profit organization. Therefore, you should expect to pay licensing fees whenever using Power BI embedding features in a production environment. This is also true when using the PaaS features of the Power BI Service to implement Power BI embedding. For simple first-party embedding scenarios, acquiring a Power BI Pro license for each user might be all you need. For scenarios involving third-party embedding, you cannot pay Microsoft by purchasing user licenses because your users are unknown to Azure AD and the Power BI Service. Instead, you pay for third-party embedding by purchasing a license for a dedicated capacity.

A *dedicated capacity* is an isolated execution environment inside the Power BI cloud. When you purchase a license for a dedicated capacity, you are effectively paying Microsoft for the memory and the processing cycles that are used to serve up your Power BI content. In many (but not all) scenarios, a dedicated capacity runs on its own isolated hardware within a Microsoft data center. Running a dedicated capacity on its own isolated hardware improves performance and scalability because an organization does have to worry about sharing memory and processing cycles with other organizations in what is known as the "noisy neighbor" problem.

To take advantage of a dedicated capacity, you must create associations with Power BI workspaces. This includes both app workspaces and personal workspaces. Once a workspace has been associated with a dedicated capacity, all the content inside that workspace is then served up by the dedicated capacity.

Any workspace that has not been associated with a dedicated capacity runs within the context of a common execution environment known as the *shared capacity*. Since the shared capacity stores and serves content to many organizations at the same time, constraints are placed at the individual level and at the organizational level so that no one organization can monopolized shared capacity resources such as processing cycles and memory.

When you are working with an app workspace running in the shared capacity, there are some important limitations. For example, the datasets you create are limited to 1GB in size in memory. You are also limited in scheduling dataset refreshes to eight times a day. In order to move beyond these limitations, you must acquire a dedicated capacity.

When you need to acquire a dedicated capacity, you must choose between several different choices. First, you can acquire a dedicated capacity though Office 365 by purchasing Power BI Premium. Power BI Premium is available through two families of SKUs known as the P SKUs and EM SKUs. Alternatively, you can provision a dedicated capacity using the Power BI Embedded service in Microsoft Azure. When you create a dedicated capacity in Microsoft azure using the Power BI Embedded service, you set its pricing tier using an A SKU. Therefore, acquiring a dedicated capacity requires you to choose between these three options.

* Power BI Premium P SKU
* Power BI Premium EM SKU
* Power BI Embedded A SKU

## Power BI Premium Capacities

Let's start by examining the P SKUs available through Power BI Premium. Many people refer to P SKUs as the *all-in* SKUs because they offers the greatest number of features. P SKUs are also the most expensive. Table 1 shows the five different P SKUs available for Power BI Premium along with their specifications and price. Note that a dedicated capacity created from a Power BI Premium P SKU is always run on its own dedicated hardware.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Power BI Premium P SKUs | P1 | P2 | P3 | P4 | P5 |
| Virtual cores (aka v-cores) | **8** | **16** | **32** | **64** | **128** |
| Memory (GB) | **25** | **50** | **100** | **200** | **400** |
| Peek renders per hour | **2,400** | **4,800** | **9,600** | **19,200** | **38,400** |
| DirectQuery executions per second | **30** | **60** | **120** | **240** | **480** |
| Dedicated hardware | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** |
| Price per month | **$4,995** | **$9,995** | **$19,995** | **$39,995** | **$79,995** |

Table 1: Power BI Premium offers five P SKUs.

The P SKUs offers valuable features to both the PasS capabilities and the SaaS capabilities of the Power BI Service. When using the PaaS capabilities of the Power BI Service, you can use a P SKU to support either first-party embedding or third-party embedding. If your organization is leveraging the SaaS capabilities of the Power BI Service to serve reports and dashboards to a large number of content consumers, a P SKU can save your organization money. This requires a bit more explanation.

Microsoft sells Power BI using both user-based licensing and capacity-based licensing. In organizations that rely exclusively on user-based licensing, users with the Power BI free license are not allowed to share content with other users nor can they consume content shared by others. This means that all users require a Power BI Pro license regardless of whether they are authoring reports and dashboards or just consuming reports and dashboards created by others.

Capacity-based licensing using Power BI Premium makes it possible for users with the free Power BI license to consume content created by others. When an organization is using Power BI Premium, they still require a Power BI Pro license for each author, but they are able to eliminate all user-based licensing costs for their read-only consumers.

Consider a simple example of an organization which has 20 Power BI content authors and 1000 other users who will be read-only consumers of Power BI content. Without capacity-based licensing, a company would have to purchase a Power BI Pro license for all users which will cost $10,200 per month.

$10,200/month = 1020 Power BI Pro licenses @ $10/month

If an organization purchases the Power BI Premium P1 SKU for $4,995 per month, the total cost of licensing drops down to $5,195 per month.

$4995/month - 1 Power BI Premium P1 license @ $4995/month

$200/month - 20 Power BI Pro licenses @ $10/month

$5195/month - Total cost

In addition to the Power BI Premium P SKUs, Microsoft also offers Power BI Premium EM SKUs which can be purchased at a much lower price point. The key difference between these two SKUs is the P SKUs make it possible for users with the Power BI free license to consume content through the Power BI portal and through the Power BI Mobile apps. The Power BI Premium EM SKUs, on the other hand, do not provide users with the Power BI free license to access content through the Power BI portal or through the Power BI Mobile apps. Therefore, the EM SKUs are primarily valuable in cases where you are embedding reports outside of the user interface experience of the Power BI portal.

Currently. Microsoft offers three Power BI Premium EM SKUs that are shown in table 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Power BI Premium EM SKUs | EM1 | EM2 | EM3 |
| Virtual cores (aka v-cores) | **1** | **2** | **4** |
| Memory (GB) | **2.5** | **5** | **10** |
| Peak renders per hour | **300** | **600** | **1200** |
| DirectQuery executions per second | **3.75** | **7.5** | **15** |
| Dedicated hardware | **No** | **No** | **Yes** |
| Price per month | **$625** | **$1,245** | **$2,495** |

Table 2: Power BI Premium offers three EM SKUs.

The primary use cases for using Power BI Premium EM SKUs typically involve a small number of users. The first use case involves no-code scenarios where business users are embedding Power BI reports in a Microsoft Teams tab or on a page in a SharePoint site using the Power BI webpart provided by Microsoft as a standard webpart in SharePoint Online. The second use cases involves development scenarios with Power BI embedding where you are using first-party embedding to serve up content to users who do not possess Power BI Pro licenses.

While Power BI Embedded EM SKUs are cheaper than the others, they have a few important limitations which you should understand. While EM1 and EM2 are the least inexpensive, they do not run on dedicated hardware and they require a yearly commitment through Microsoft's volume discounting program. A dedicated capacity based on a EM3 can be acquired with a monthly commitment and it will run on dedicated hardware. However, what happens when you compare the EM3 SKU to the P1 SKU?

A EM3 capacity is half the cost of a P1 capacity and it gets half of what the P1 capacity gets in terms of v-cores and memory. The EM3 SKU is half the price of the P1 SKU so you might think it's a good deal. However, you should remember that P SKUs provide free license users with the ability to access to content through the Power BI portal and the Power BI Mobile apps while the EM SKUs do not.

You can purchase a monthly subscription to Power BI Premium through the Purchases Services link in the Office 365 admin Center. The act of purchasing a Power BI Premium subscription will trigger the Power BI Service to provision a new dedicated capacity inside your Office 365 tenant. Once this dedicated capacity has been provisioned, you can view and manage it from the Power BI Admin portal as shown in Figure 5.

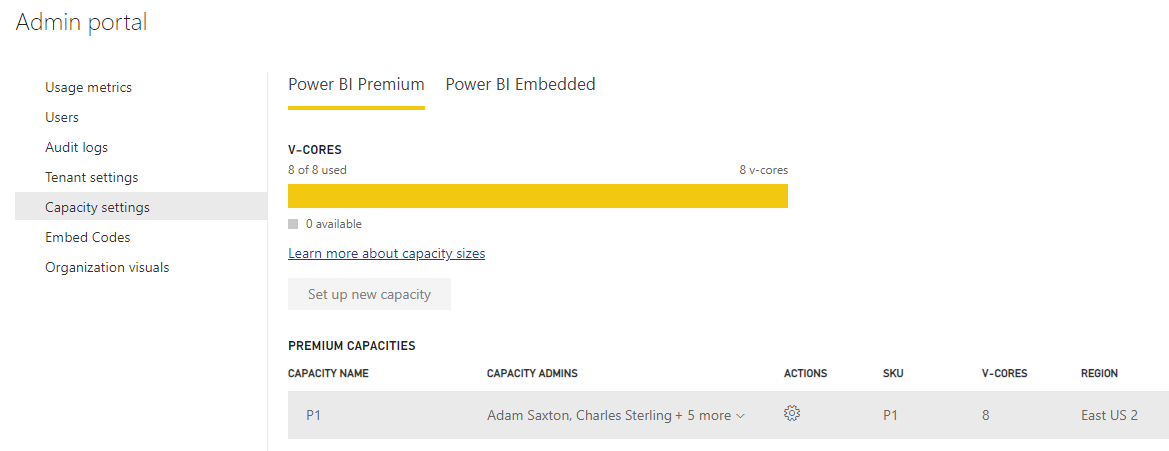


Figure 5: The Power BI Admin portal allows you to view and manage dedicated capacities.

As you can see from the screenshot shown in Figure 5, the Power BI Admin portal provides one tab to display *Power BI Premium* capacities and a second tab to display *Power BI Embedded* capacities. A Power BI Premium capacity and a Power BI Embedded capacity are both a type of dedicated capacity. Both types of dedicated capacities are useful in various scenarios involving software development with Power BI embedding. However, there are also important differences between Power BI Premium capacities and Power BI Embedded capacities which you should understand.

One big difference between a Power BI Premium capacity and a Power BI Embedded capacity has to do with where you go to manage and monitor them. Power BI Premium capacities can be managed and monitored from within the Power BI Admin portal. Power BI Embedded capacities are managed and monitored using the Azure portal or by using PowerShell commands or REST API calls available thought the Azure Resource Manager.

If you drill into the view for a Power BI Premium capacity in the Power BI Admin portal, you can see statistics on its CPU and memory usage, memory thrashing and the number of queries executed against the DirectQuery limit. By monitoring these statistics, you can assess on a day-by-day basis whether a dedicated capacity is being overloaded. You can also see if you are paying more than you need to service your audience of users.

When you purchase a new subscription for a Power BI Premium P SKU, the Power BI Service responds by provisioning a single Power BI Premium capacity using all the resources that come with the SKU. For example, image you purchase a Power BI Premium P3 subscription through the Office 365 Admin center. The Power BI Service will initially create a new Power BI Premium capacity with 32 v-cores. Once the capacity has been created, you can spread its resources across two or more capacities. For example, you could split up the 32 v-cores from a P3 subscription into two capacities with 16 v-cores each. Alternatively, you could split the 32 v-cores by creating one capacity with 24 v-cores for high priority workloads and a second capacity with 8 v-cores for less important workloads.

See the following link for more information on configuring Power BI Premium capacities:  
<https://docs.microsoft.com/en-us/power-bi/service-admin-premium-manage>

Keep in mind that a dedicated capacity doesn't provide any real value until you populate it with content. You add content to a dedicated capacity by creating associations with workspaces. To publish content, you first create a new app workspace and associate it with a dedicated capacity. Then you build out the content in the app workspace by adding datasets, reports and dashboards. After that, all the content in that app workspace will be served from the dedicated capacity making it available to users who do not have a Power BI Pro license.

Each dedicated capacity has a configurable set of capacity administrators. A Power BI Pro user in the role of capacity administrator can associate app workspaces with a dedicated capacity using the Power BI Amin portal or by using Power BI Administrative PowerShell commands. A content author with a Power BI Pro license who has been granted the *Capacity assign* permission can create a new app workspace and associate it with a dedicated capacity using the *Create an app workspace* dialog as shown in Figure 6.

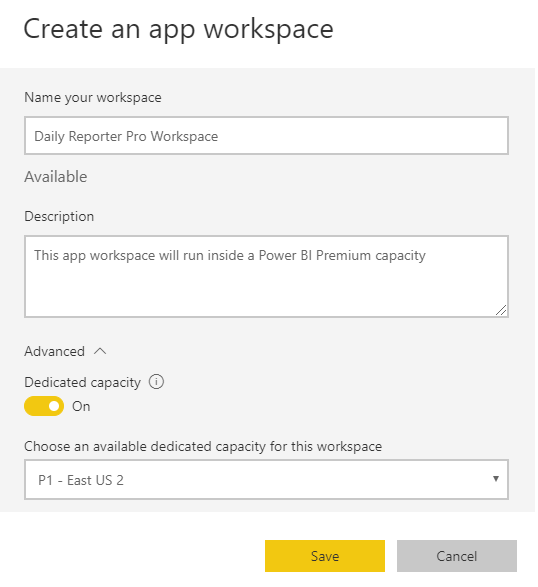


Figure 6: The Create an app workspace dialog allows you to associate a new workspace with a dedicated capacity.

Remember that any workspace that has not been associated with a dedicated capacity will run in the shared capacity and that all users require a Power BI Pro license to consume content from the shared capacity. This is true in scenarios where users are accessing Power BI content through the Power BI portal. It is also true in scenarios where you are developing custom applications that use first-part embedding.

When you are navigating between workspaces in the Power BI portal, it's easy to tell which ones are associated with a dedicated capacity because they are displayed with a diamond icon as shown in figure 7. Any workspace that is not displayed with the diamond icon is running within the shared capacity.

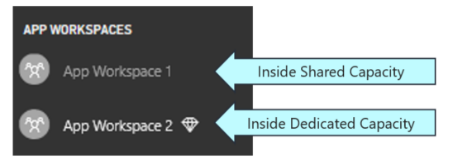


Figure 7: A workspace associated with a dedicated capacity is displayed with a diamond icon.

## Power BI Embedded Capacities

The Power BI Embedded service in Microsoft Azure allows you to create a dedicated capacity as an scalable, on-demand service. This type of dedicated capacity in known as a *Power BI Embedded capacity*. A key point about Power BI Embedded capacities is that they only support third-party embedding and the app-owns-data model. They do not support first-party embedding or the user-owns-data model. For this reason, Power BI Embedded capacities are mainly used by ISVs and not by corporate developers building enterprise applications for a single organization.

You must have an Azure subscription to create a Power BI Embedded capacity. When creating a Power BI Embedded capacity, you must choose a pricing tier which determines its cost and how many resources are allocated to it. The available pricing tier SKUs are A1, A2, A3, A4 A5 and A6 as shown in Table 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Power BI Embedded Capacity Pricing Tiers | A1 | A2 | A3 | A4 | A5 | A6 |
| Virtual cores (aka v-cores) | **1** | **2** | **4** | **8** | **16** | **32** |
| Memory (GB) | **3** | **5** | **10** | **25** | **50** | **100** |
| Peek renders per hour | **300** | **600** | **1,200** | **2,400** | **4,800** | **9.600** |
| DirectQuery executions per second | **3.75** | **7.5** | **15** | **30** | **60** | **120** |
| Dedicated hardware | **No** | **No** | **Yes** | **Yes** | **Yes** | **Yes** |
| Approximate price per month | **$750** | **$1,495** | **$2,995** | **$5,995** | **$11,995** | **$23,995** |
| Approximate price per hour | **$1** | **$2** | **$4** | **$8** | **$16** | **$32** |

Table 3: When creating a Power BI Embedded capacity, you must choose between 6 available pricing tiers.

Power BI Embedded capacities are particularly attractive to ISVs because they provide the following features that are not matched by Power BI Premium capacities.

1. You can automate the provisioning of Power BI Embedded capacities using PowerShell scripts and Azure templates
2. Power BI Embedded capacities do not require a monthly commitment; instead you pay by the hour
3. You can pause and resume a Power BI Embedded capacity at any time to save money
4. You can scale a Power BI Embedded capacity up or down in a matter of minutes

Power BI Embedded capacities will appeal to organizations and developers who have already embraced Microsoft Azure. Just as with other types of on-demand services in Azure such as web apps, storage accounts and virtual machines, you can create a Power BI Embedded capacity by hand in the Azure portal. Alternatively, you can automate the provisioning of Power BI Embedded capacities using Azure templates, PowerShell script or REST-based API calls to the Azure Resource Manager. Figure 8 shows a screenshot of what a new Power BI Embedded capacity looks like once it has been created in the Azure portal.

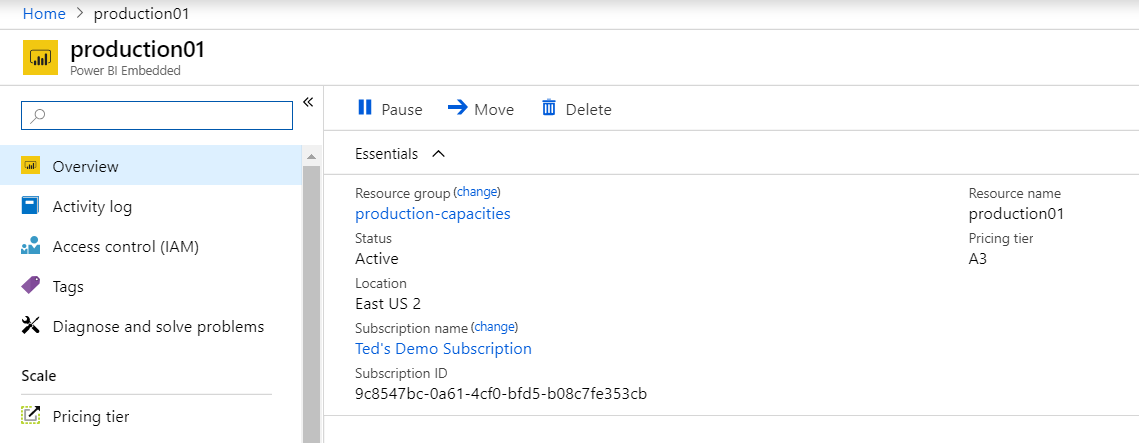


Figure 8: You can create, manage and monitor Power BI Embedded Capacities in the Azure portal.

If you examine the screenshot in figure 8, you will notice that the toolbar for a Power BI Embedded capacity provides a **pause** button. A Power BI Embedded capacity is an Azure service that you can pause and resume at any time. Since there is no monthly commitment with a Power BI Embedded capacity, you are only charged while the service is running. If you are in a scenario where a Power BI Embedded capacity doesn’t have to run 24x7, you can shut it off to save money. It only takes a minute or two to pause or to resume the service.

Power BI Embedded capacities also have a significant advantage over Power BI Premium capacities when it comes to scaling up or down. Consider a scenario where your reporting application experiences a burst of user activity on the first day of each month. You can configure the hosting Power BI Embedded capacity with a less expensive pricing tier (e.g. A2) for standard usage. Then you can scale the Power BI Embedded capacity up to a more powerful pricing tier (e.g. A4) for the day in each month where the spike appears.

The Azure portal also provides monitoring tools so you can determine exactly how much you need to spend to achieve the required level of performance for a given user audience. While you must use the Azure portal and other Azure DevOps tools to create, configure and monitor Power BI Embedded capacities, you can the Power BI Admin portal to view Power BI Embedded capacities and to see their SKU (aka pricing tier), Region and Status as shown in figure 9.

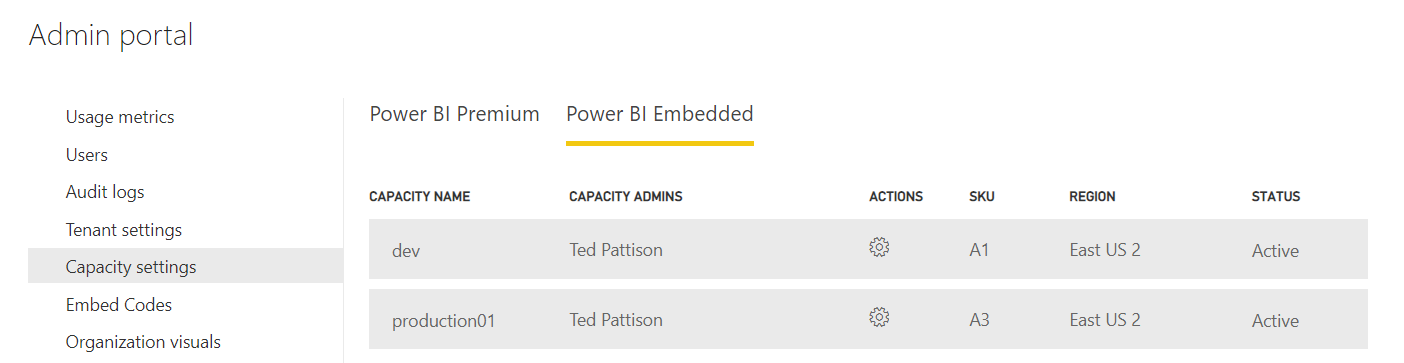


Figure 9: You can view Power BI Embedded capacities and associate workspaces in the Power BI Admin portal.

If you drill into a specific Power BI Embedded capacity in the Power BI Admin portal, you will notice that it does not provide you with the ability to configure the underlying capacity or to assign capacity administrators. Since a Power BI Embedded capacity is an Azure service, you must navigate to the Azure portal to change its pricing tier or assign capacity administrators.

While you cannot configure a Power BI Embedded capacity in the Power BI Admin portal, you can configure its workspace associations just as with a Power BI Premium capacity using the **Assign workspaces** dialog as shown in Figure 10.

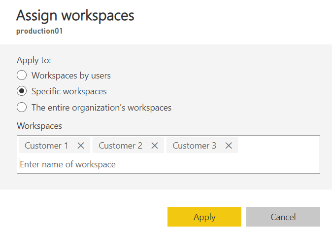


Figure 10: The Assign workspaces dialog for a dedicated capacity allows you to create workspace associations

The Power BI Admin portal page for any type of dedicated capacity contains a **Workloads** section that displays all the current workspace associations as shown in Figure 11. This view provides the capacity administrator with a user experience to view and manage the workspaces running inside.

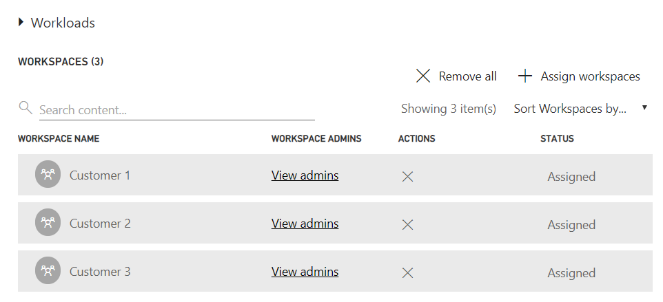


Figure 11: The Power BI Admin portal allows you to view and manage workspaces associations.

## Configuring Production Tenants

Before deploying a custom application that uses Power BI embedding into production, you must make sure that the hosting Azure AD tenant is properly configured with whatever dedicated capacities are required. Over the last few pages, you've learned about three different families of SKUs for creating dedicated capacities including the P SKU, the EM SKU and the A SKU. Table 4 provides a comparison of the three SKU types to assist you with selecting the best type of dedicated capacities for a particular scenario.

|  |  |  |  |
| --- | --- | --- | --- |
|  | P SKU | EM SKU | A SKU |
| Purchased through… | **Office 365** | **Office 365** | **Microsoft Azure** |
| Supports third-party embedding in custom applications | **Yes** | **Yes** | **Yes** |
| Supports first-party embedding in custom applications | **Yes** | **Yes** | **No** |
| Supports free users accessing content in the Power BI portal | **Yes** | **No** | **No** |
| Supports free users accessing content in Power BI Mobile | **Yes** | **No** | **No** |
| Billing cycle | **Monthly** | **Monthly** | **Hourly** |
| Commitment | **Monthly** | **Monthly/Yearly** | **None** |
| Turn it off when you're not using it | **No** | **No** | **Yes** |

Table 4: Each of the three SKUs for purchasing a dedicated capacity offers a unique set of different features.

For any given scenario, you should be able to use the comparison points shown in Table 4 to determine the best type and number of dedicated capacities required. You should also have a plan to monitor the usage statistics of these dedicated capacities over time so you can ensure you have the resources you need to maintain adequate performance and that you are not spending more than you need to. If you can move from a Power BI Premium P5 subscription down to a P4 subscription, it will save your organization $480,000 per year.

Let's look at a few examples. Consider a scenario in which you have developed an application that uses first-party embedding and the application has a small audience of 20-50 users. You cannot use a Power BI Embedded capacity because the A SKUs don't support first-party embedding. Given the small number of users, a Power BI Premium capacity will not be cost effective. The recommended approach would be to assign Power BI Pro users accounts to all users to allow them to consume content from the shared capacity.

Consider a second scenario where an application uses the app-owns-data model and third-party embedding. This represents the classic ISV scenario where the ISV application handles user authentication and the user identities are unknown to Azure AD and the Power BI Service. Here is a scenario that requires a dedicated capacity. You can choose between either a Power BI Premium capacity or a Power Embedded capacity because they both support third-party embedding. As an ISV, you will likely choose to go with Power BI Embedded capacities due to the provisioning and scaling advantages they have over Power BI Premium capacities.

Power BI Premium capacities are most useful to organizations with a large number of Power BI users. A common motivation for purchasing a Power BI Premium capacity is to move a large number of read-only consumers from the Power BI Pro license to the free license. A Power BI Premium P1 subscription costs $5000 per month. Once you exceed the breakeven number of 500 users who are read-only consumers, the Power BI Premium P1 subscription will be less expensive than purchasing a Power BI Pro license for every user.

A second motivation for purchasing a Power BI Premium subscription is to move beyond the limitations and constraints imposed by the shared capacity. You will need to acquire a dedicated capacity if you are working with datasets that are larger than 1GB in memory or if you need to schedule data refresh more than eight times a day. If you want the content to be accessible through the Power BI portal and the Power BI Mobile apps, then the type of dedicated capacity you need is a Power BI Premium capacity created with a P SKU.

A Power BI Premium capacity is flexible because it offers a combination of SaaS features and PaaS features at the same time. Imagine a scenario where you have purchased a Power BI Premium capacity and you are using it to serve up content through the Power BI portal and through the Power BI Mobile apps to an audience of 2,000 employees that all have the Power BI free license. At this point you're only leveraging the SaaS features of the Power BI Premium capacity.

Now let's say you decide to develop a custom application that uses third-party embedding to display Power BI reports and dashboard to an external audience of customers or partners. You can use the same Power BI Premium capacity to serve content to both audiences. Alternatively, you could split the Power BI Premium capacity into two so you can isolate the external audience from the internal audience in its own dedicated capacity.

## Creating a Development Tenant

Now that you have learned about what's required in a production tenant, it's time to discuss how to create a development environment for Power BI embedding. After all the scary pricing information you have seen for dedicated capacities, you'll be happy to know that you can set up a development environment for Power BI embedding in a matter of minutes and it will not cost you a dime. All you really need is an Office 365 development tenant and a Power BI Pro license.

If you do not already have a development tenant, you can create a new Office 365 tenant with a free 30-day trial of the Office 365 E5 subscription for up to 25 users. The Office 365 E5 subscription includes a Power BI Pro subscription which provides the licensing you need to call the Power BI Service API and to implement Power BI embedding in a development scenario.

In any production scenario, your custom application is only permitted to generate embed tokens for content in workspaces associated with a dedicated capacity. Fortunately, these rules are relaxed for development scenarios. That means you can rely on the shared capacity when you are developing with Power BI embedding which eliminates the cost and overhead of provisioning dedicated capacities during the development lifecycle.

Keep in mind that Microsoft is able to monitor the generation of embed tokens across all Power BI customer tenants. This gives Microsoft the ability to detect when embed tokens are being generated from the shared capacity and used to embed Power BI resources. This, in turn, gives Microsoft the ability to assess whether the generation and usage of embed tokens within your tenant falls within reasonable limits for development scenarios.

Microsoft monitors shared capacity statistics including the number of embed tokens generated per hour and the number of IP addresses where embed tokens are being used to embed Power BI resources. If Microsoft sees evidence of production usage of embed tokens generated from the shared capacity, that will raise a flag and you can expect a call from Microsoft support or your Microsoft sales representative.

# Developer Quick Start into Power BI Embedding

This section provides step-by-step instructions to get you up and running with Power BI embedding. Next, you will step through the Power BI Onboarding Experience to create and populate an app workspace in Power BI and to register a new Azure application in your development tenant. After that, you will be ready to create a new ASP.NET web application and write the code required to embed reports, dashboard and the Q&A experience on a custom web page.

## Creating an Office 365 trial tenant

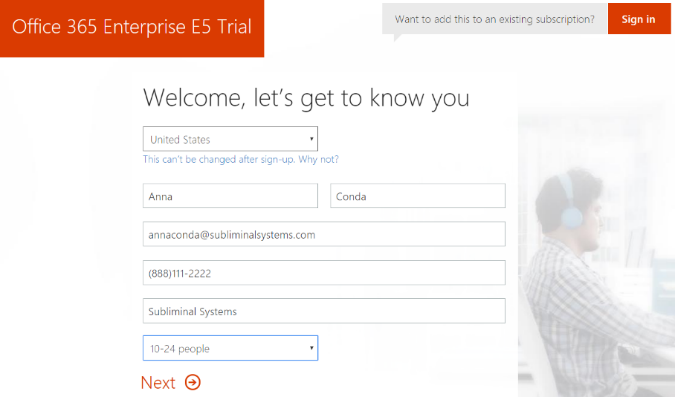
When you follow the steps in this section to create a new Office 365 trial tenant, you will be asked to provide a user name and a password for a user account that will be configured as the Global administrator. You can use this account when developing and testing applications that use Power BI embedding. However, you should also test your applications with standard user accounts that have no administrative permissions. The trial tenant you will create allows you to create up to 25 user accounts with Office 365 E5 subscriptions. Remember that any user with an Office 365 E5 subscription is automatically assigned a Power BI Pro license as well

1. Navigate to the Office 365 trial sign up page using an Incognito browser window.
   1. Launch the Chrome browser.
   2. Using the dropdown menu in the upper right, select the command to open a **New incognito window**.
   3. Copy and paste the following URL into the address bar of the incognito window to navigate to the sign up page.

<https://go.microsoft.com/fwlink/p/?LinkID=698279&culture=en-US&country=US>

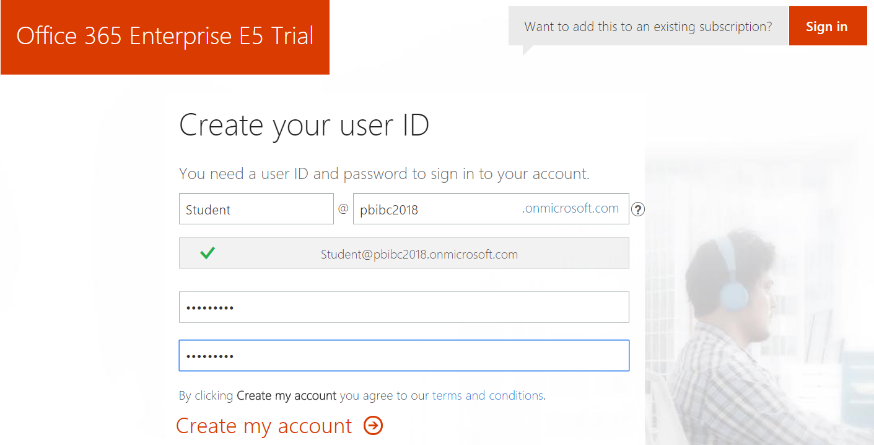
It’s not always necessary to sign up for an Office 365 trial account using an incognito window. However, most errors that occur when attempting to sign up are caused by cached browser settings such as residue from an earlier Office 365 trial account. The solution to overcoming most errors when signing up for a trial account is using an incognito window.

1. Fill out the form with your personal information and click **Next**.



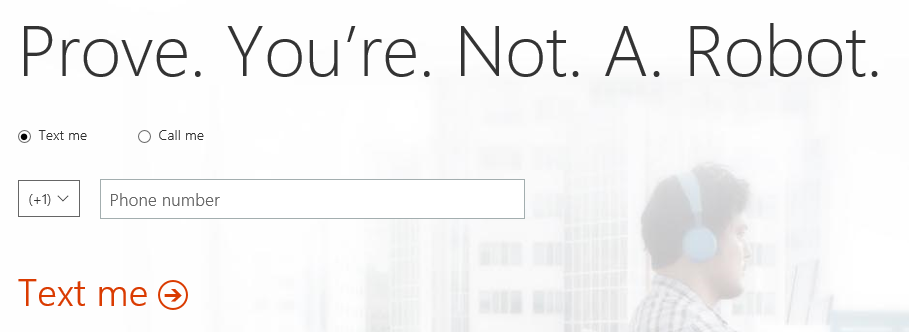
The information you provide on the next page of the signup process will be used to name your new Office 365 tenant.

1. On the **Create your user ID** page…
   1. Enter a user name
   2. Enter a unique company name *(you might have to try a few before you get one that’s unique)*
   3. Enter a password that you will remember.

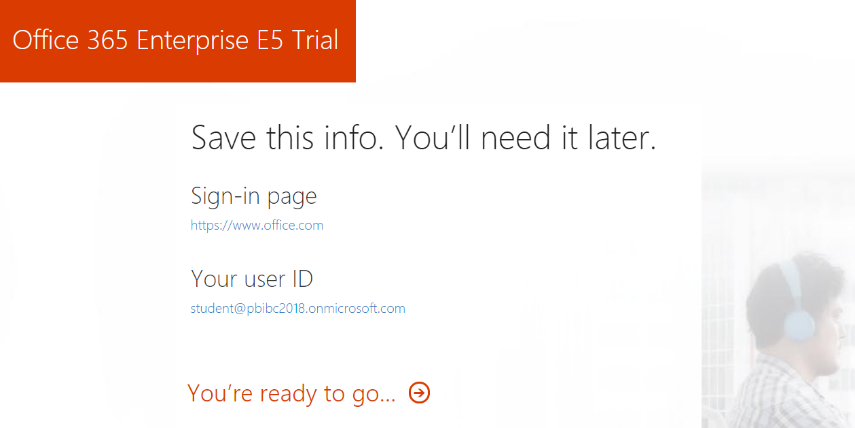


Note that the company name you enter on this page will be used to create the domain name for your new Office 365 trial tenant. For example, if you were to enter a company name of **pbibc2018**, it would result in the creation of a new Office 365 tenant within a domain of **pbibc2018.onMicrosoft.com**. The user name you enter will be used to create the first user account which will be given administrative rights within the Office trial tenant. If you enter a user name of **Student**, then the email address as well as user principal name for this account will be **Student@pbibc2018.onMicrosoft.com**.

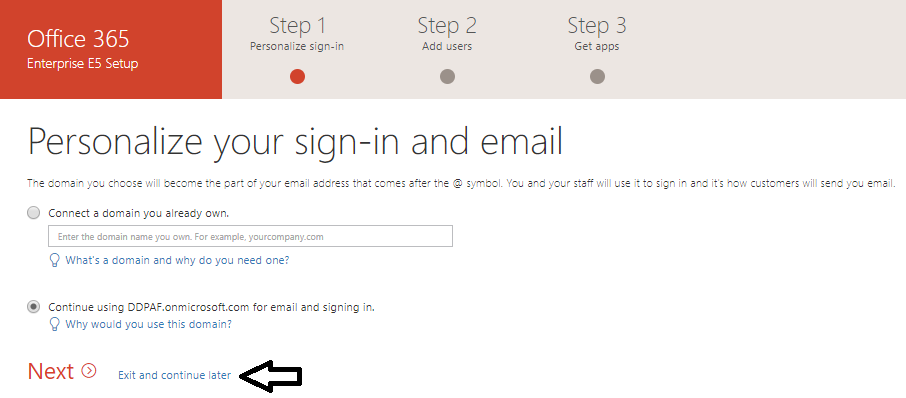
1. Click **Next** to continue to step 3.
2. Complete the validation form in step 3 by proving you are not a robot.
   1. Select the **Text me** option and provide the number of your mobile phone.
   2. When you go through this process, a Microsoft service will send you a text message that contains an access code.
   3. You retrieve the access code form your mobile device and use it to complete the validation process.



1. Once you have completed the validation process, click the **You’re ready to go…** link to navigate to the portal welcome page for your new Office 365 trial tenant. Note that you should already be logged on using the user account that was created during the signup process.



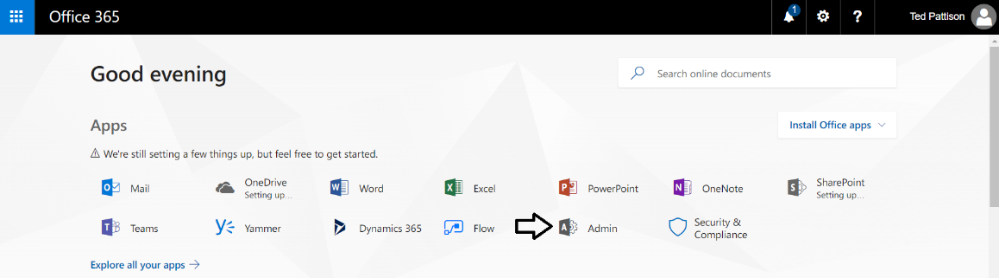
1. If you are prompted with the **Personalize your sign-in and email**, click the **Exit and continue later** link at the bottom of the page.



At this point, you have already created your new Office 365 tenant which can support creating up to 25 user accounts with Office 365 Enterprise E5 trial licenses. Note that some Office 365 services within your new Office 365 tenant such as the Office 365 admin center, PowerApps, Flow and Power BI can be accessed immediately. Other services in your Office 365 tenant such as SharePoint Online, OneDrive for Business and Outlook will not be ready immediately and can take some time to provision.

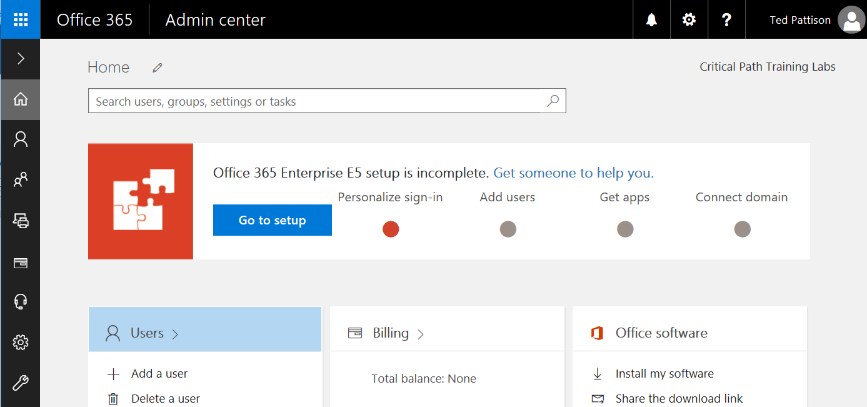
There is no more need to run the browser in incognito mode anymore because it’s only required to get through the signup process. You can now return to using a standard browser window. However, it’s always a good thing to check to see who you are logged in as because sometimes the browser may log you on using a different Office 365 account you have instead of your new trial account.

1. At this point, you should be located on an Office 365 welcome page. Click the **Admin** tile to go to the Office **365 admin center**.

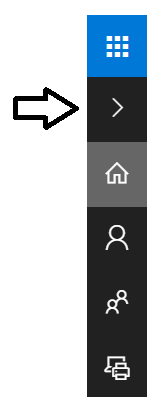


If you are presented with the Office 365 admin center welcome dialog, close it by clicking the **X** menu in the upper right corner.

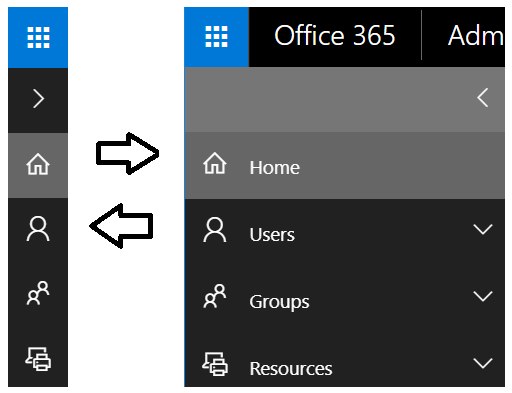
1. Verify that you are able to access the home page of the **Office 365 admin** **center**. as shown in the following screenshot



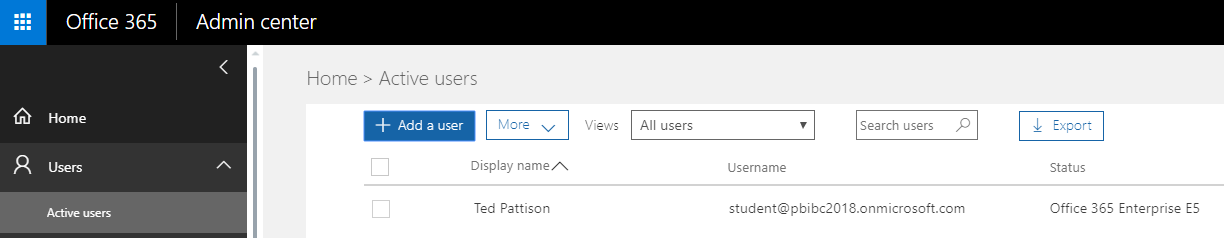
1. Inspect the set of Active Users in the current tenancy.
   1. Locate the top **Menu** button for the left navigation menu. It’s the second button from the top with the arrow icon which sits just beneath the Office 365 App Launcher menu button.



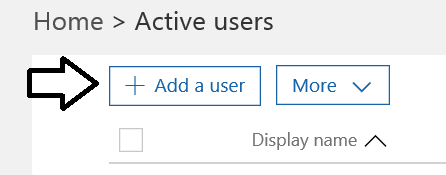
* 1. Click the top **Menu** button several times and see how it toggles the left navigation between a collapsed and expanded mode.



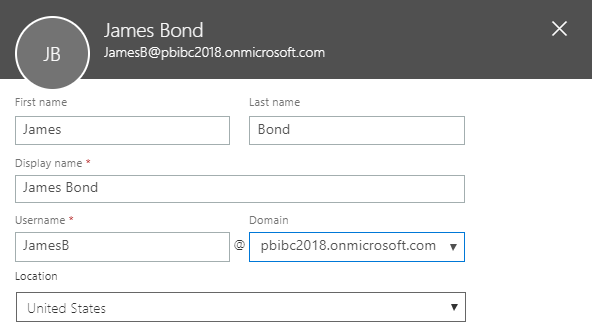
* 1. Once the **Active Users** page is displayed, you should be able to verify that the user account you are currently logged on as is the only user account that exists in the current tenancy. Remember that this account has been set up as a Global Administrator to the tenant because it is the account that was used when creating the tenant.



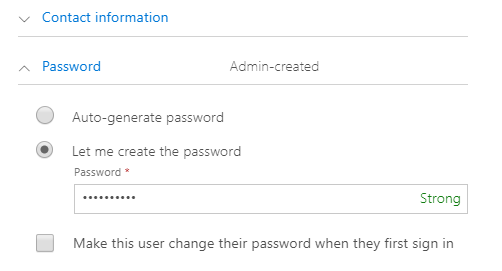
1. Create a new user account.
   1. On the **Active Users** page, click the button **Add a user** button to create a new user account

. 

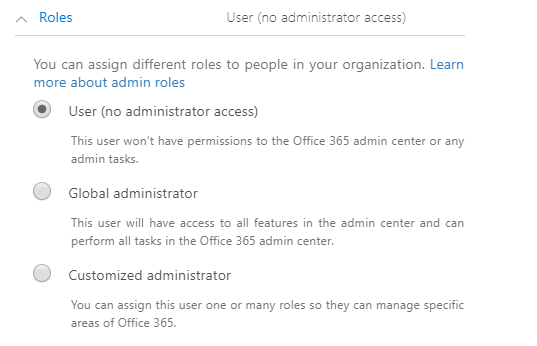
* 1. Fill in the **Create new user account** form with information for a new user account. When creating this account, you can use any name you would like. These lab instructions will demonstrate this by creating a user account for a person named **James Bond** with a user name and email of **JamesB@pbibc2018.onmicrosoft.com**.



* 1. Expand **Password** section under **Contact Information** section.
     1. Select the option for **Let me create the password**.
     2. Enter a password of **pass@word1** into the textboxes labeled **Password** and **Retype** **Password**.
     3. Uncheck the checkbox for the option labeled **Make this user change their password when they first sign in**.

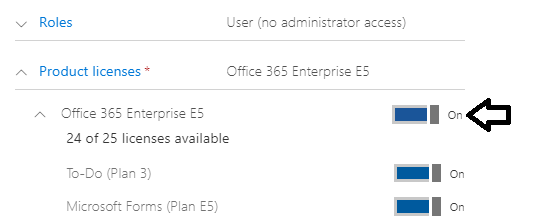


* 1. Expand the roles section. You do not need to change anything in this section, although you should note that this new user account will be created as a standard user account without any administrator access or privileges.

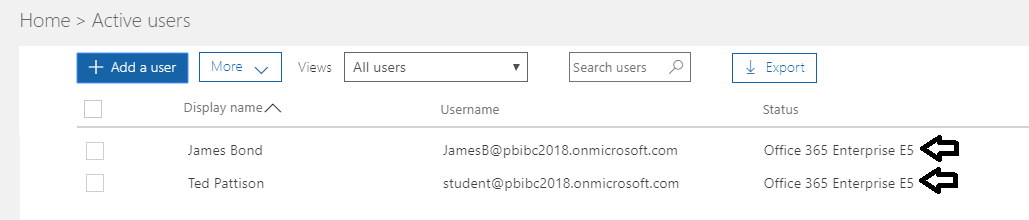


Note that the new account is usually assigned a trial license for **Office 365 Enterprise E5** plan. However, it’s a good practice to check and make sure the new user has been assigned a license for **Office 365 Enterprise E5** which includes the **Power BI Pro** license.

* 1. In the **Product licenses** section, make sure the **Office 365 Enterprise E5** license is set to **On**..



* 1. Click the **Save** button at the bottom of the new user form to create the new user account.
  2. When you see the **User was added** message, click **Send email and close** to dismiss the **Add new user** task pane.
  3. Verify that the new user account has been created and is displayed along with your primary user account.



Now you have a secondary user account that does not have any administrative permissions. It's important that you test applications which use first-party embedding with standard user accounts to ensure your application doesn't require users with special permissions.

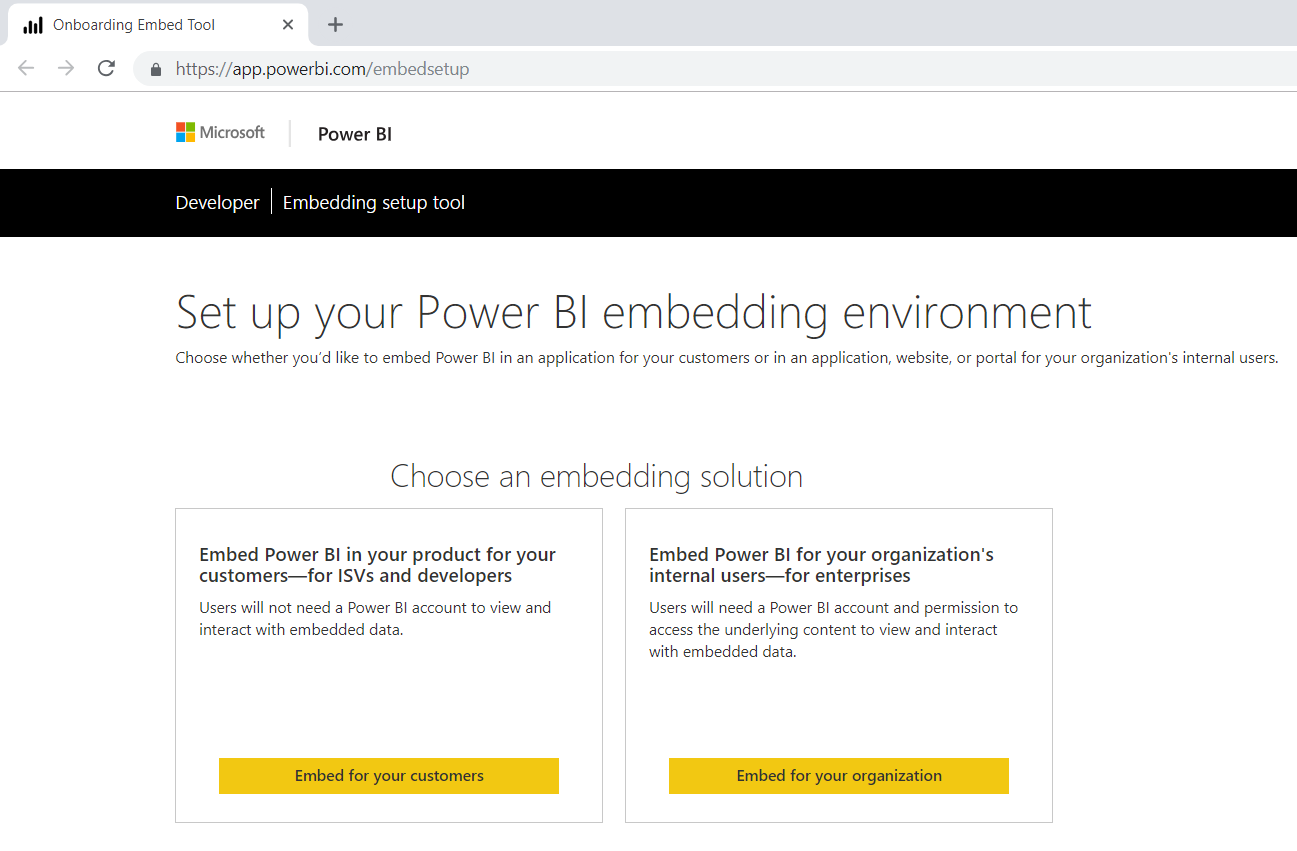
## Using the Power BI Embedded Onboarding Experience

In this section you will use the Power BI Onboard Experience to get up and running with Power BI embedding as soon as possible.

1. Using the browser, navigate to the Power BI Onboard Experience using the following URL

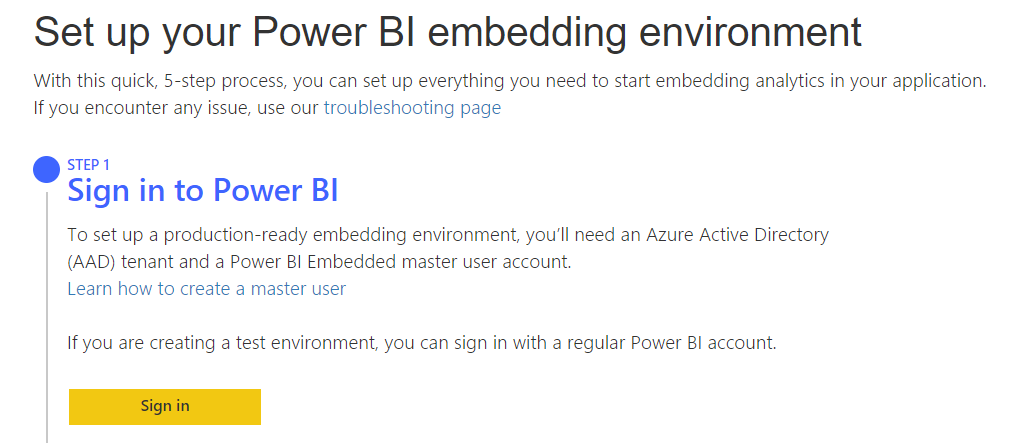
<https://app.powerbi.com/embedsetup>

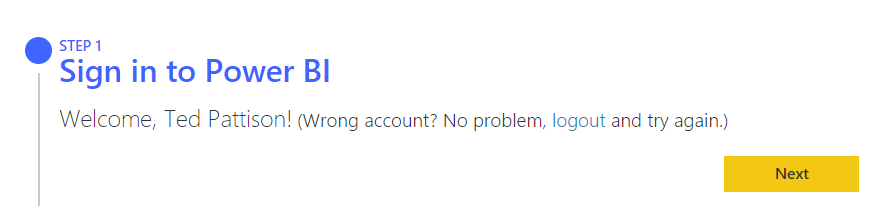
1. You should see this.

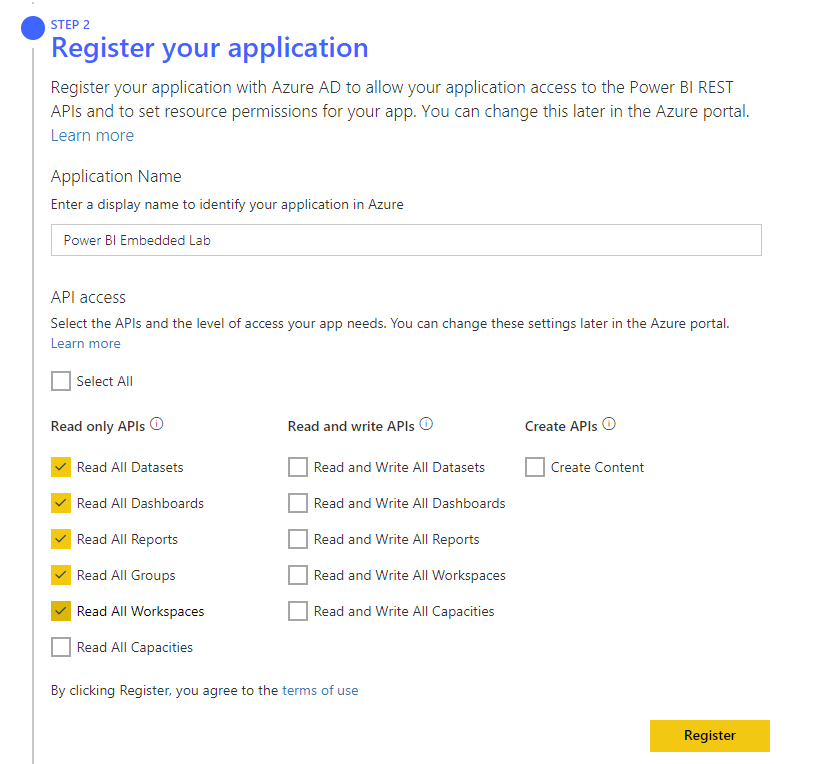


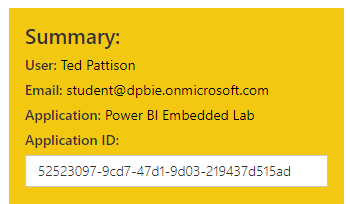
There are two button on the page. The **Embed for your customers** button is used to begin the set up for an application using third-party embedding and the app-owns-data model. The Embed for your organization button is used to begin the set up for an application using first-party embedding and the user-owns-data model.

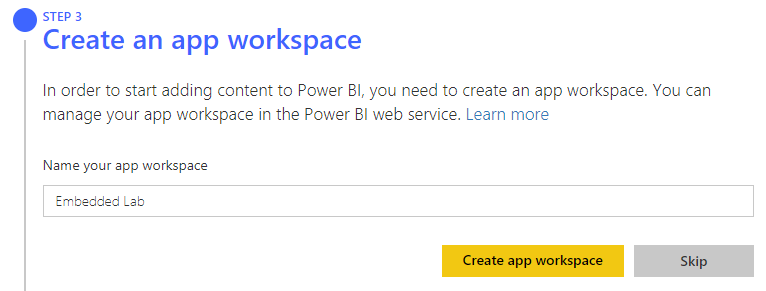
1. Press the **Embed for your customers** button to begin the set up for an application that uses third-party embedding.
2. On the Set up your Power BI embedding environment page, click the **Sign in** button and sign in with your primary Office 365 user account that is in the role of Global administrator.

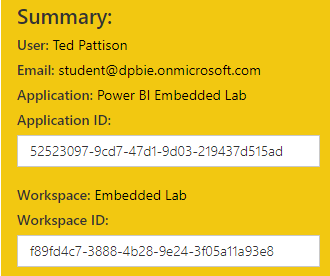


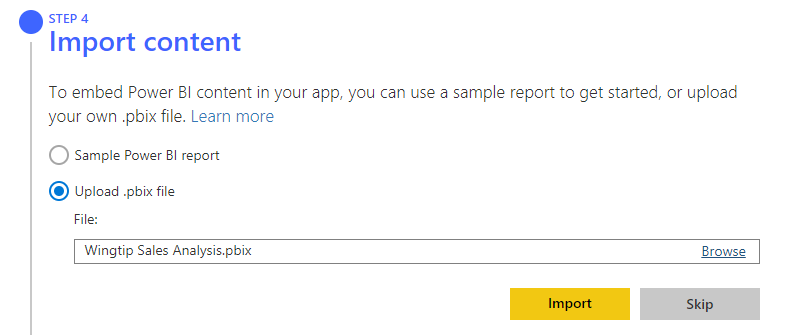


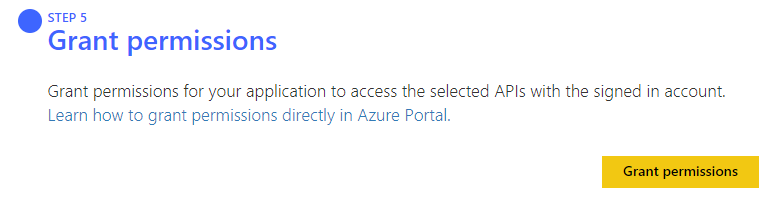


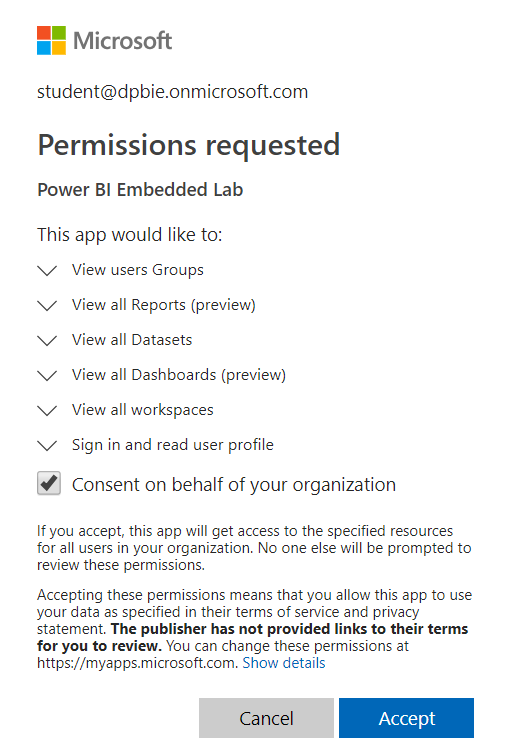




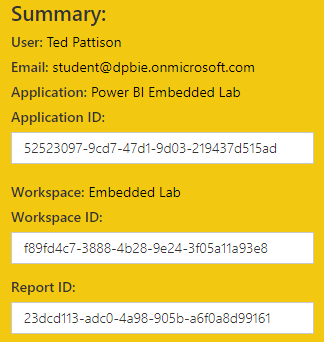




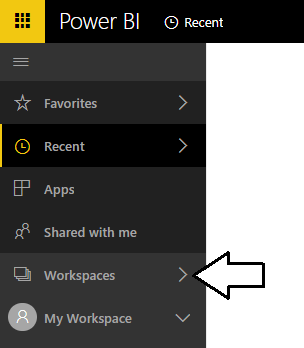


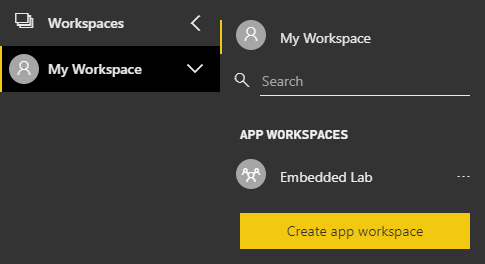


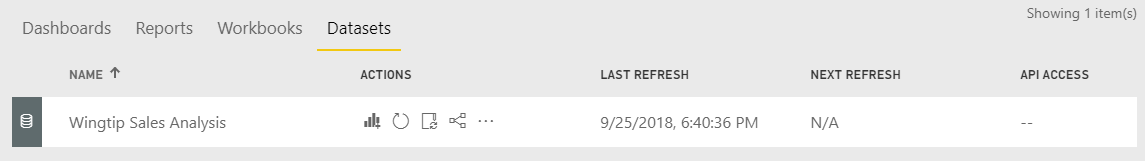


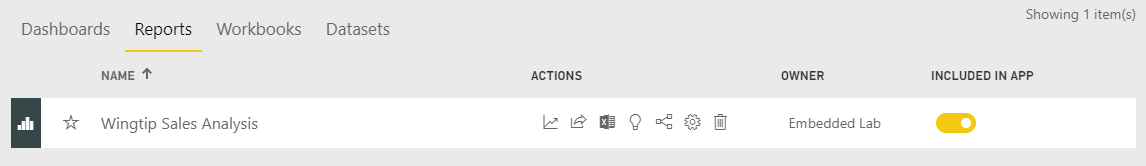


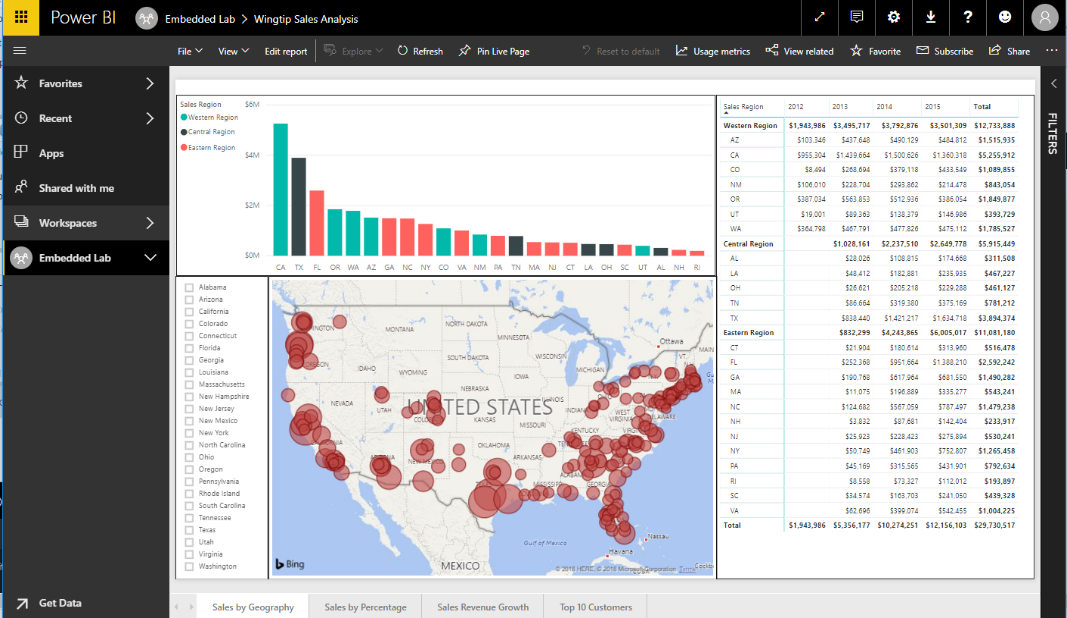
1. Inspect the app workspace named **Embedded Lab** created by the onboarding experience.
   1. Create a new tab in the browser and navigate to the Power BI portal.
   2. Click the **Workspace** flyout menu in the left navigation.











1. Check this out

https://app.powerbi.com/groups/{app-workspace-id}/reports/{report-id}/ReportSection1

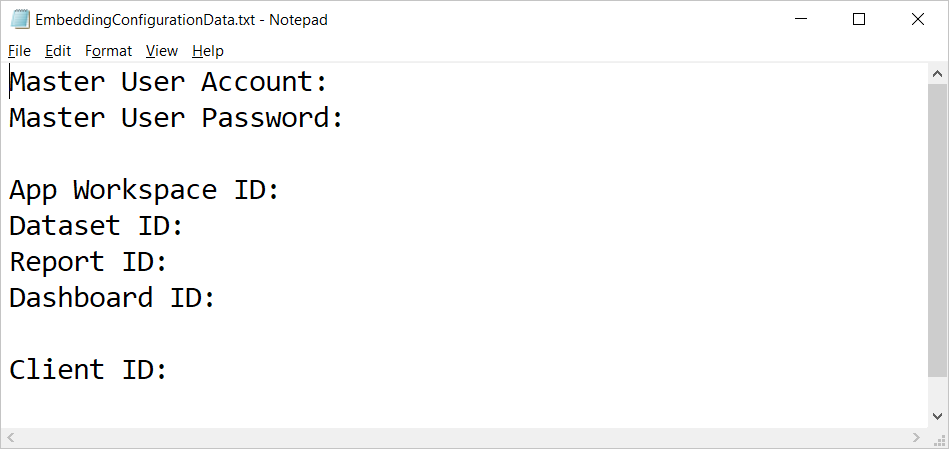
1. x

Now that you have finished preparing the app workspace with the content you will use for Power BI embedding, you must record a few key pieces of configuration data you will need later in this lab. First, you will record which Active Directory user account you will use as the master user account. Next, you will record the identifying GUID for the **Embedded Lab** app workspace and the identifying GUIDs for the dataset, report and dashboard you created inside this app workspace.

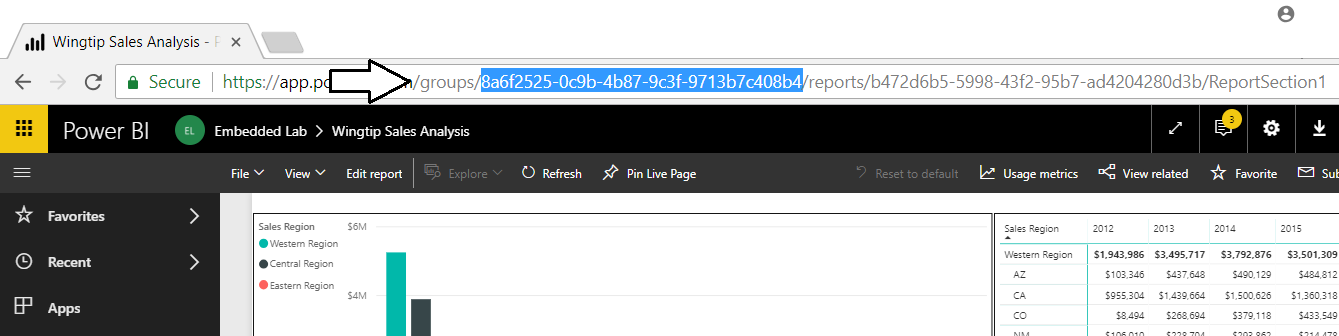
1. Record configuration data that you will need later in this lab.
   1. Locate the configuration file named **EmbeddingConfigurationData.txt** which is located at the following path.

C:\Student\Modules\08\_PBIEmbedded\Lab\EmbeddingConfigurationData.txt

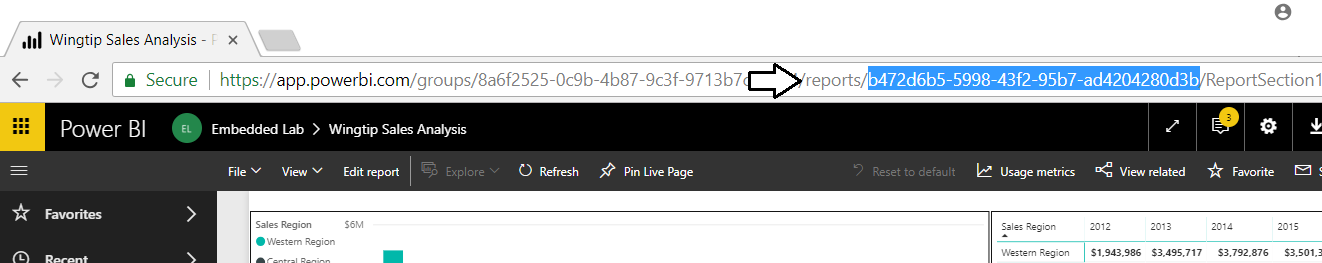
* 1. Open **EmbeddingConfigurationData.txt** using Widows Notepad and inspect it contents. It contains the names of 7 essential pieces of configuration data you will need later in this lab.



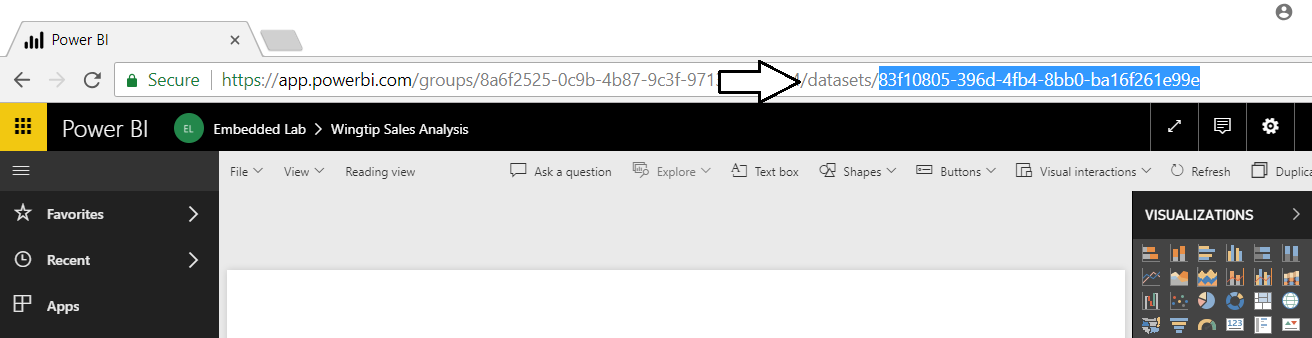
* 1. Complete the top two lines by adding the name of your primary Office 365 account and the password for your account.
  2. Navigate to the **Wingtip Sales Analysis** report inside the **Embedded Lab** app workspace.
  3. Locate and copy the app workspace ID from the URL by copy the GUID that comes after **/groups/**.



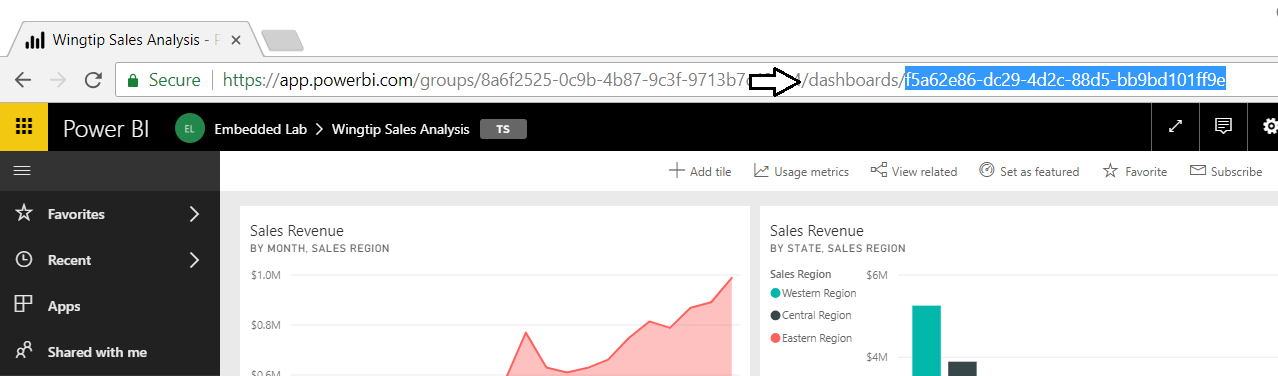
* 1. Copy the app workspace ID into **EmbeddingConfigurationData.txt**.
  2. Navigate back to the **Wingtip Sales Analysis** report inside the **Embedded Lab** app workspace.
  3. Locate and copy the report ID from the URL by copy the GUID that comes after **/reports/**.



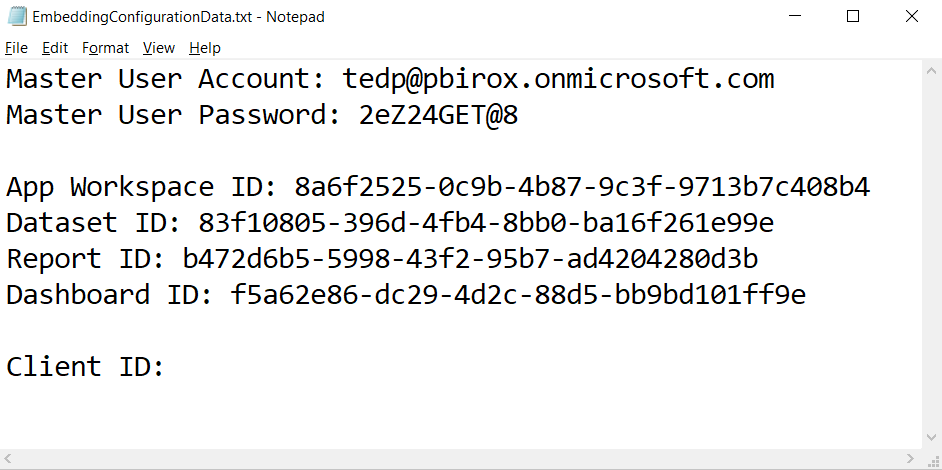
* 1. Copy the report ID into **EmbeddingConfigurationData.txt**.
  2. Navigate to the **Wingtip Sales Analysis** dataset inside the **Embedded Lab** app workspace to create a new report.
  3. Locate and copy the dataset ID from the URL by copy the GUID that comes after **/datasets/**.



* 1. Copy the dataset ID into **EmbeddingConfigurationData.txt**.
  2. Navigate to the **Wingtip Sales Analysis** dashboard inside the **Embedded Lab** app workspace.
  3. Locate and copy the dashboard ID from the URL by copy the GUID that comes after **/dashboards/**.



* 1. Copy the dashboard ID into **EmbeddingConfigurationData.txt**.
  2. You should have now updated **EmbeddingConfigurationData.txt** with all the configuration data you need with the exception of the client ID that you will create in the next exercise.

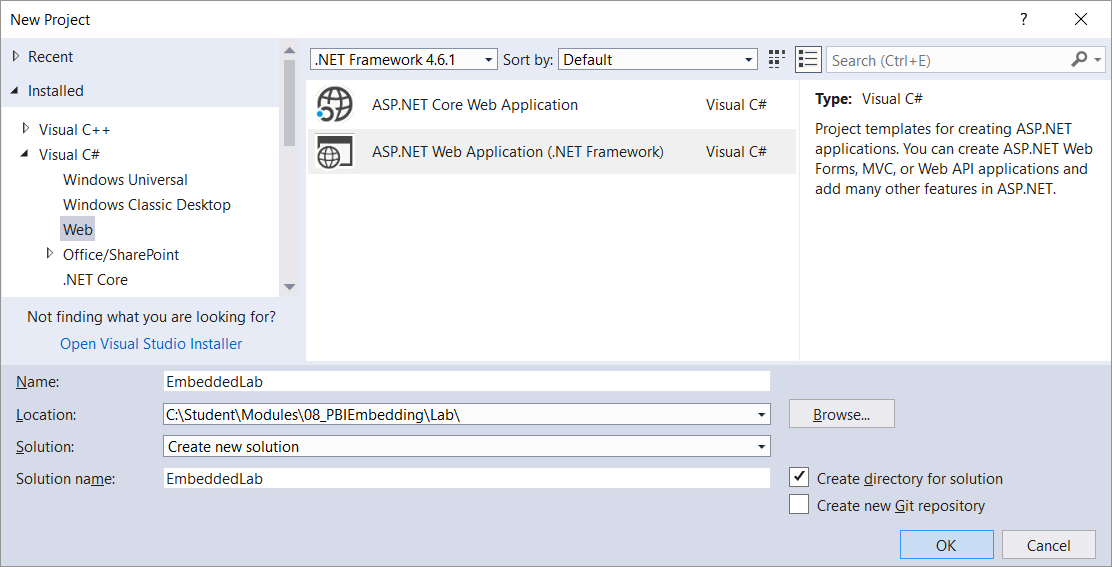


* 1. Save your changes to **EmbeddingConfigurationData.txt**.

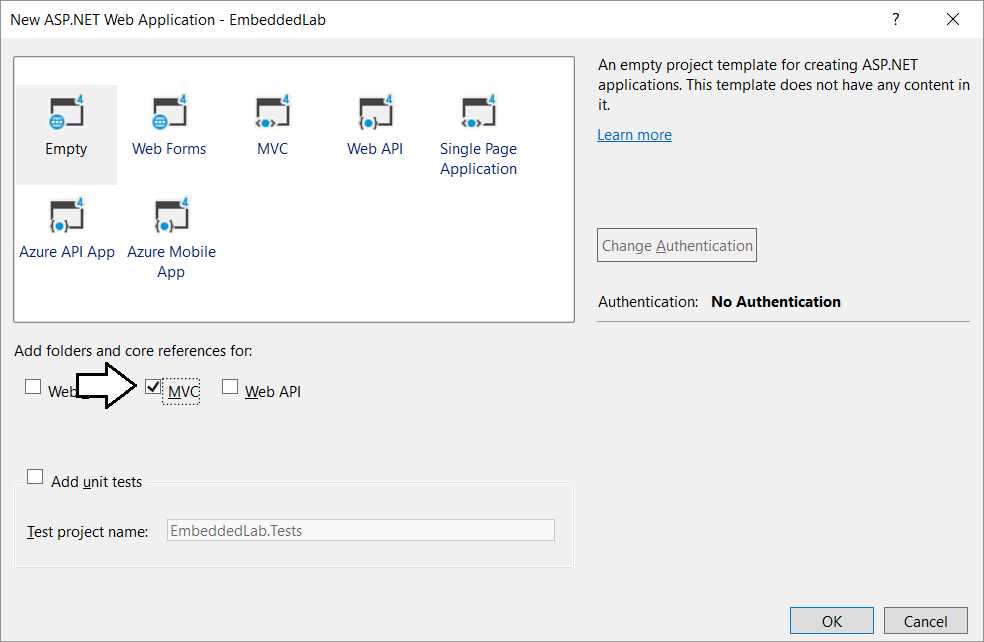
## Embed a Report in an ASP.NET MVC Application

In this exercise you will create a new Web Application project using Visual Studio 2017 and the ASP.NET MVC framework.

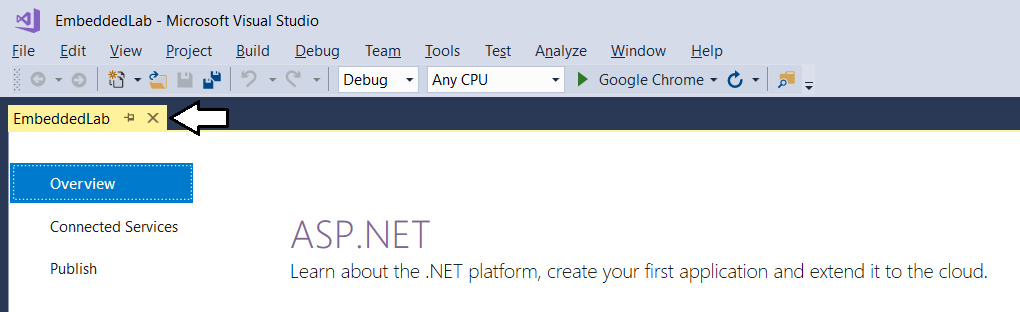
1. Launch **Visual Studio 2017**.
2. Create a new ASP.NET MVC project in Visual Studio 2017.
   1. In Visual Studio select **File > New > Project**.
   2. In the **New Project** dialog:
      1. Select **Installed > Templates > Visual C# > Web**.
      2. Select the **ASP.NET Web Application** project template.
      3. Name the new project **EmbeddedLab**.
      4. Add the new project into the folder at **C:\Student\Modules\08\_PBIEmbedding\Lab**.
      5. Click **OK** to display the **New ASP.Net Web Application** wizard.



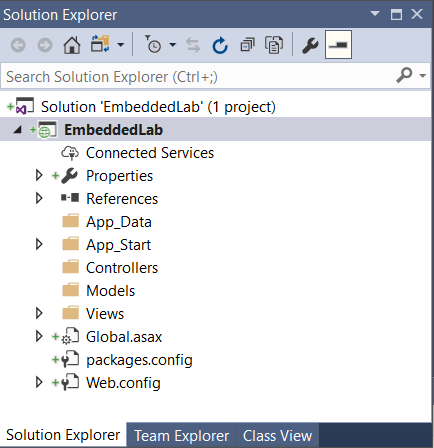
* 1. In the **New ASP.Net Web Application** dialog, select the **Empty** template.
  2. In the section with the caption **Add folders and core references**, make sure the **MVC** checkbox is checked.
  3. Click the **OK** button to create the new project.



* 1. When Visual Studio finishes creating the project, it displays an information page. Close this page by clicking the **x** in the tab.

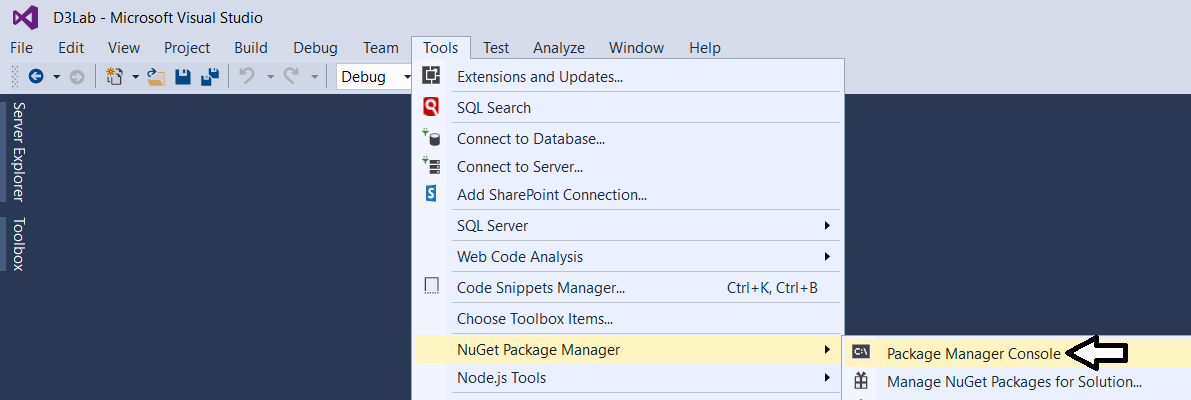


* 1. Take a minute to familiarize yourself with the structure of the project in the **Solution Explorer**.

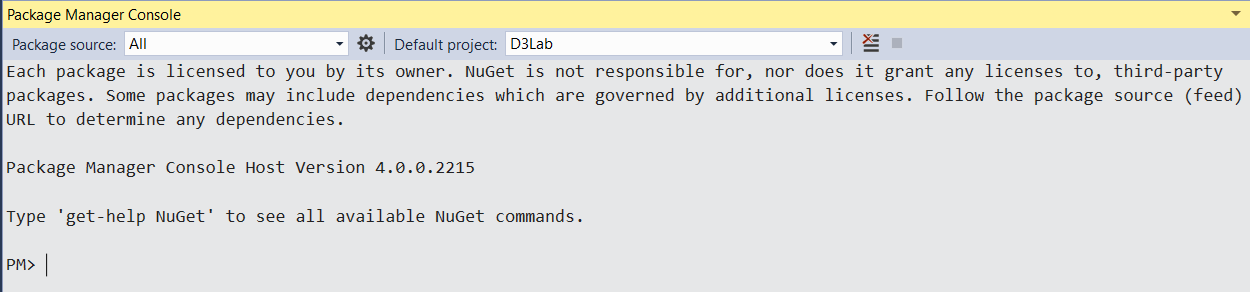


At this point, you have created a new ASP.NET MVC project based on the **Empty** project template. You will need to add an MVC controller and several MVC views before your application provides any type of user interface experience. Before adding a controller or writing any code, you will first update the project’s NuGet packages that were automatically included with your new project. You will also prepare for Power BI embedding by adding the NuGet package for the Azure AD Authentication library (ADAL) and the NuGet packages for the Power BI Service API and the Power BI JavaScript API.

1. Configure the **Embedded Lab** project with the required set of NuGet packages
   1. From the Visual Studio menu, select the command **Tools > NuGet Package Manager > Package Manager Console**.



* 1. You should now see the **Package Manage Console** with a **PM>** command prompt as shown in the following screenshot



* 1. Type in and execute the following command to install the NuGet package for **bootstrap**.

Install-Package bootstrap

* 1. Type in and execute the following command to install the NuGet package for **Azure Active Directory Authentication library**.

Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory

* 1. Type in and execute the following command to install the NuGet package for the **Power BI Service API**.

Install-Package Microsoft.PowerBI.Api

* 1. Type in and execute the following command to install the NuGet package for the **Power BI JavaScript API**.

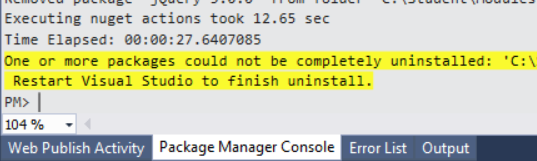
Install-Package Microsoft.PowerBI.JavaScript

Now that you have installed the required NuGet packages for Power BI embedding, you will now run the **Update-Package** cmdlet to make sure all the packages in your project are updated to the latest versions available in the NuGet repository.

* 1. Type in and execute the following command to update all NuGet packages in the project to their most current version.

Update-Package

* 1. The first time you run the **Update-Package** cmdlet, you will be prompted to restart Visual Studio to complete the update.



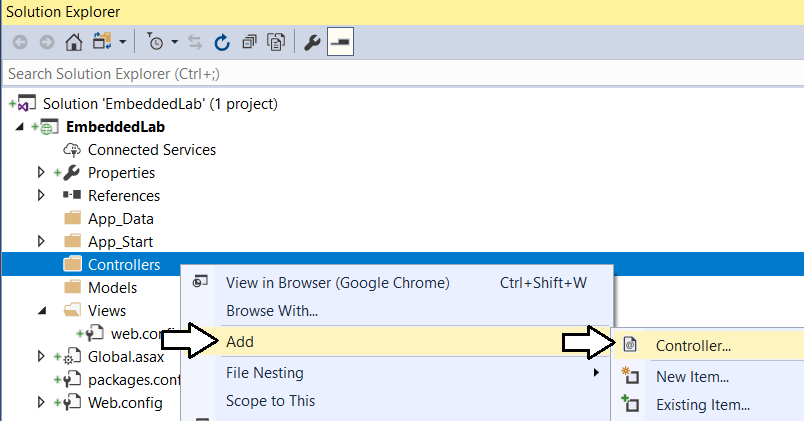
* 1. Restart Visual Studio and open the **EmbeddedLab** project.
  2. Open the Package Manager Console window if it is not already open.
  3. Execute the **Update-Package** cmdlet one more time to ensure all NuGet packages are updated to their most current version.

Update-Package

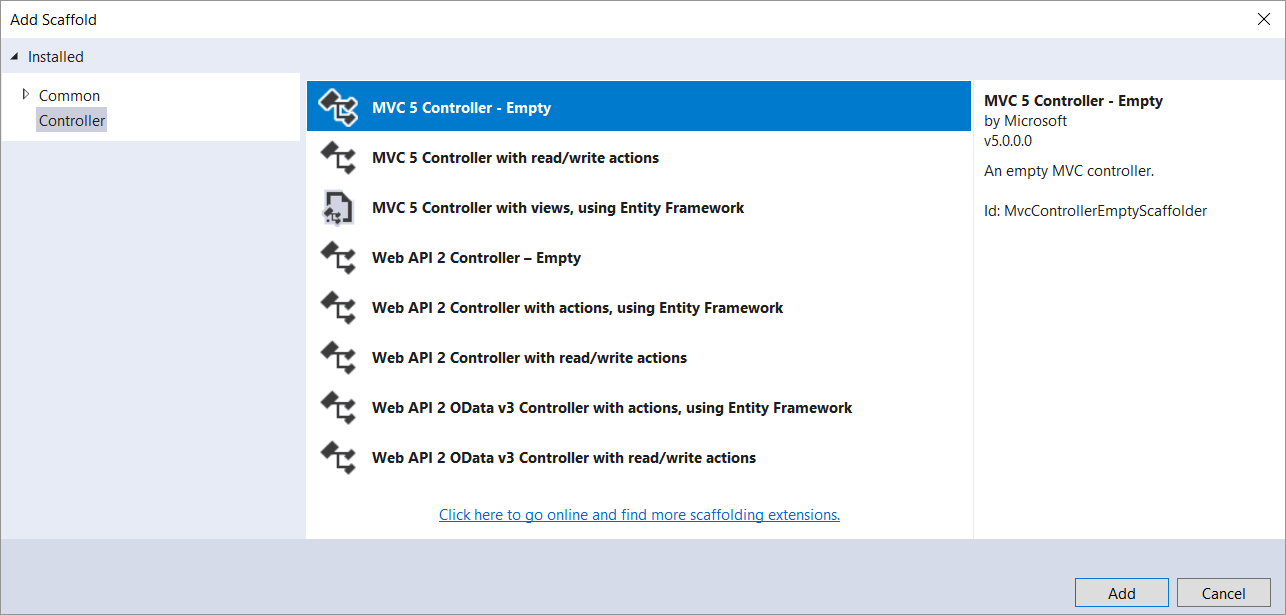
* 1. You should now see an output message in the Package Manager Console indicating “There are no new updates available”.



1. Add the **HomeController** class.
   1. In Solution Explorer, right-click on the **Controllers** folder.



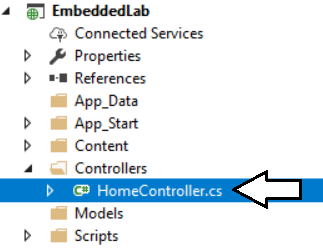
* 1. In the **Add Scaffold** dialog, select the first option **MVC 5 Controller – Empty** and then click **Add**.



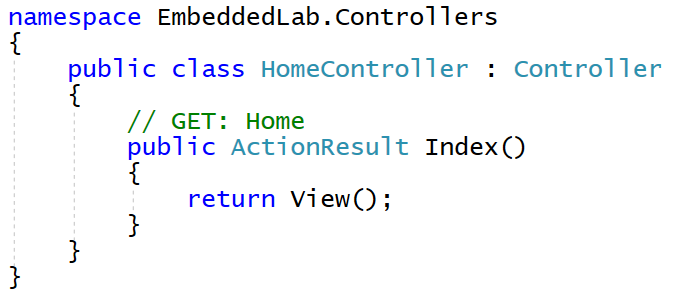
* 1. In the **Add Controller** dialog, enter a **Controller name** of **HomeController** and then click **Add**.



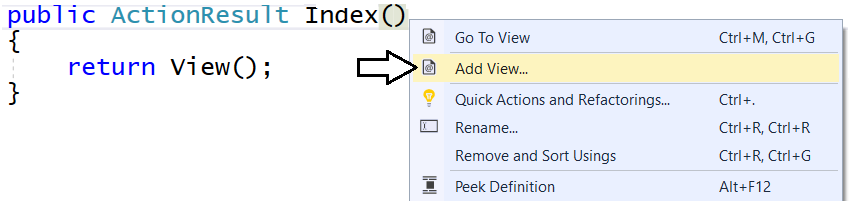
* 1. You should now see a new source file in the **Controllers** folder named **HomeController.cs**.



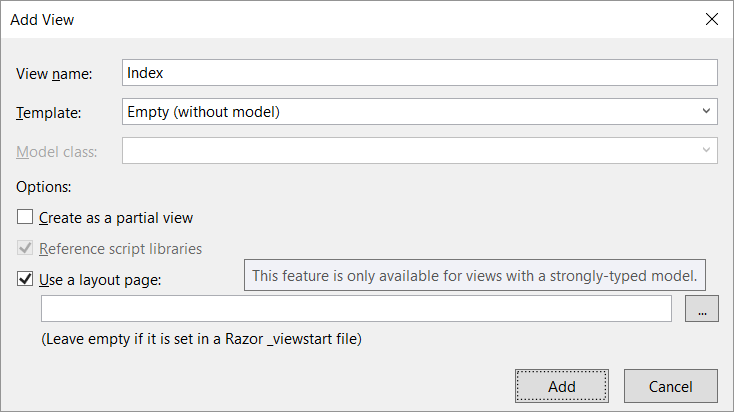
* 1. Inside **HomeController.cs**, you will find the starting point for the **HomeController** class with a single method named **Index**.



1. Add a view for the **Index** action method of the **Home** controller class.
   1. Inside **HomeController.cs**, right-click the **Index** method and select the **Add View…** command.

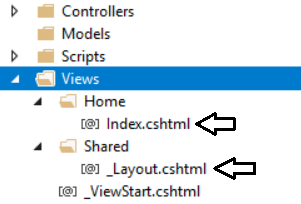


* 1. In the **Add View** dialog, accept all the default setting as shown in the following screenshot and click **Add**.

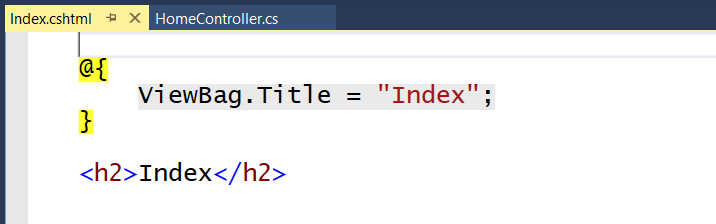


When you create a new view and leave the **Use a layout page** option selected, a new shared layout page named **\_Layouts.cshtml** is automatically added to the project in the **Views/Shared** folder.

* 1. In Solution Explorer, you should be able to verify that your project contains two new files.
     1. Inside the **Views/Home** folder there is a new razor view file named **Index.cshtml**.
     2. Inside the **Views/Shared** folder there is a new shared layout page named **\_Layouts.cshtml**.



* 1. Examine the code that has been added to **Index.cshtml**.



* 1. Delete all the code inside **Index.cshtml** and replace it with the following HTML code.

<div id="homePageContainer" class="container" >

<div class="jumbotron">

<h2>Power BI Embedded Lab</h2>

</div>

</div>

* 1. Save your changes and close **Index.cshtml**.

Over the next few steps, you will add the HTML code for a shared layouts page into **\_Layout.cshtml** in a sequence of several different copy-and-paste operations. If you’d rather copy and paste the all the code for **\_Layout.cshtml** at once, you can find the completed HTML code inside a file named **Layout.cshtml.txt** located in the **C:\Student\Modules\08\_PBIEmbedded\Lab\Snippets** folder.

1. Modify the shared layouts page named **\_Layouts.cshtml**.
   1. In Solution Explorer, expand the **Views** folder and then expand the **Shared** folder.
   2. Double-click on **\_Layouts.cshtml** to open it in an editor window.
   3. Delete the entire contents of **\_Layouts.cshtml** and replace with the following HTML starter page.

<!DOCTYPE html>

<html>

<head>

</head>

<body>

</body>

</html>

* 1. Copy and paste the following HTML code to provide the **head** section

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Embedded Lab</title>

<link href="~/Content/bootstrap.css" rel="stylesheet" />

<link href="~/Content/Site.css" rel="stylesheet" />

<script src="~/Scripts/jquery-3.3.1.js"></script>

</head>

* 1. Make sure your script link to jQuery matches the version number of the jQuery library source file in the **Scripts** folder.
  2. Copy and paste the following HTML code to provide the **body** section of the page.

<body>

<!-- Add Banner with TopNav and Toolbar Here -->

<!-- Add Main Body Content Here -->

<!-- Add JavaScript Code to Resize Page Elements Here -->

</body>

Now you will copy and paste HTML markup code into each of the three sections in the HTML **body** element.

* 1. Copy and paste the following code into the body just below the ***Add Banner with TopNav and Toolbar Here*** comment.

<!-- Add Banner with TopNav and Toolbar Here -->

<div id="banner" class="container">

<nav id="topnav" class="navbar navbar-expand-sm navbar-dark bg-dark">

<ul class="navbar-nav">

<li class="nav-item active">

@Html.ActionLink("Embedded Lab", "Index", "Home",

routeValues: null, htmlAttributes: new { @class = "nav-link navbar-brand" })

</li>

<li class="nav-item">

@Html.ActionLink("Report", "Report", "Home",

routeValues: null, htmlAttributes: new { @class = "nav-link" })

</li>

<li class="nav-item">

@Html.ActionLink("Dashboard", "Dashboard", "Home",

routeValues: null, htmlAttributes: new { @class = "nav-link" })

</li>

<li class="nav-item">

@Html.ActionLink("Q&A", "Qna", "Home",

routeValues: null, htmlAttributes: new { @class = "nav-link" })

</li>

<li class="nav-item">

@Html.ActionLink("New Report", "NewReport", "Home",

routeValues: null, htmlAttributes: new { @class = "nav-link" })

</li>

</ul>

</nav>

@RenderSection("toolbar", required: false)

</div>

* 1. Copy and paste the following code into the body just below the **Add Main Body Content Here** comment

<!-- Add Main Body Content Here -->

<div id="content-box" class="container body-content">

@RenderBody()

</div>

* 1. Copy and paste the following code into the body just below the **Add Main Body Content Here** comment

<!-- Add JavaScript Code to Resize Page Elements -->

<script>

$(function () {

var heightBuffer = 12;

var newHeight = $(window).height() - ($("#banner").height() + heightBuffer);

$("#content-box").height(newHeight);

$("#embedContainer").height(newHeight);

$(window).resize(function () {

var newHeight = $(window).height() - ($("#banner").height() + heightBuffer);

$("#content-box").height(newHeight);

$("#embedContainer").height(newHeight);

});

});

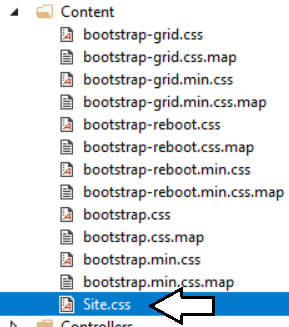
</script>

* 1. Save your changes and close **\_Layouts.cshtml**.

1. Modify the **Sites.css** file with a set of custom CSS styles.
   1. Using Windows Explorer, locate the snippet file named **Site.css.txt** in the **Students** at the following location.

C:\Student\Modules\08\_PBIEmbedded\Lab\Snippets\Site.css.txt

* 1. Double click on **Site.css.txt** to open it in Notepad.
  2. Select all the CSS code inside **Site.css.txt,** copy it to the Windows clipboard and return to Visual Studio.
  3. In Solution Explorer, expand the **Content** folder and then double-click on **Sites.css** open it in an editor window.

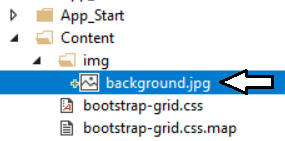


* 1. Delete all the existing content from **Sites.css** and paste in the content from the Windows clipboard.
  2. Save your changes and close **Sites.css**.

1. Add a new image named **background.jpg** to the project to provide a page background.
   1. Using Windows Explorer, locate the file named **background.jpg** in the **Students** folder at the following location.

C:\Student\Modules\08\_PBIEmbedded\Lab\StarterFiles\background.jpg

* 1. In Solution Explorer, create a new folder named **img** inside the **Contents** folder.
  2. Copy the file **background.jpg** into the **img** folder.



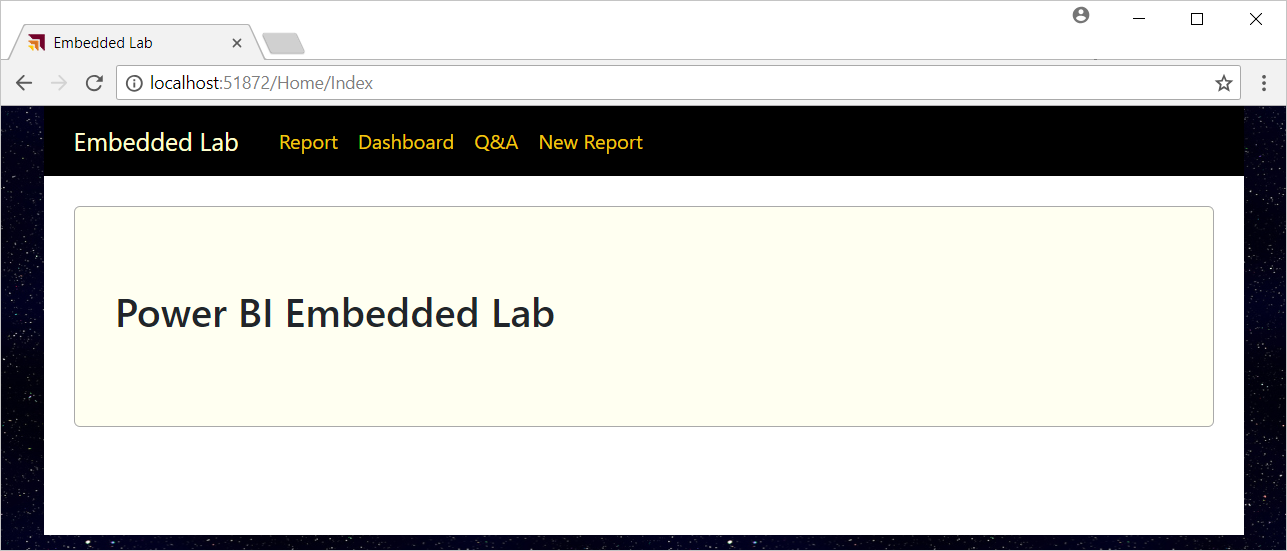
1. Add a **favicon.ico** file to the root folder of the **EmbeddedLab** project.
   1. Using Windows Explorer, locate the file named **favicon.icon** in the **Students** folder at the following location.

C:\Student\Modules\08\_PBIEmbedded\Lab\StarterFiles\favicon.ico

* 1. Copy the file named **favicon.ico** to the root folder of your project.



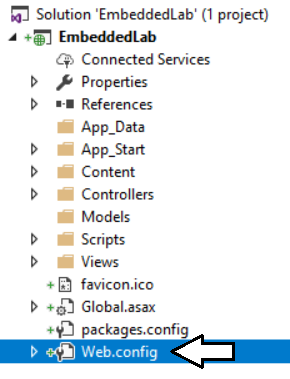
1. Test out the **EmbeddedLab** project using the Visual Studio Debugger
   1. Press the **{F5}** key to start up the project in the Visual Studio debugger.
   2. When the project starts, the home page should load in the browser and match the following screenshot.



* 1. Close the browser, return to Visual Studio and stop the debugger.

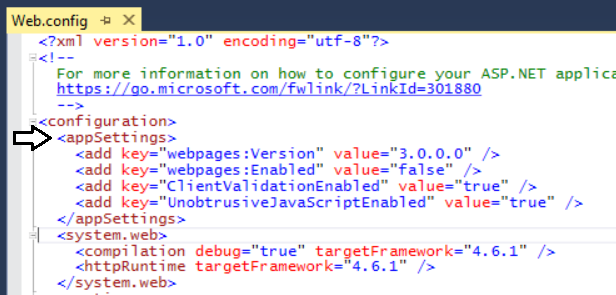
Note that the navigation links on the top navigation menu are not working yet. Over the next few exercises, you will add MVC action methods and razor views to implement Power BI embedding behavior behind each of these navigation links.

1. Modify the project’s **web.config** file to add **appSetting** values for the required configuration data.
   1. Open the **web.config** file located at the root of the **EmbeddedLab** project.

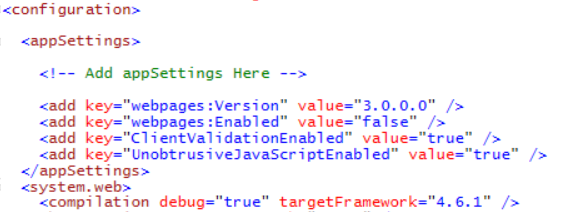


Make sure you open the **web.config** file located at the root of the project and not the **web.config** file inside the **Views** folder.

* 1. Locate the **<appSettings>** element at the top of **web.config**.



* 1. Add a few blank lines just after the **<appSettings>** element opening tag.



* 1. Copy and paste the following XML code into the **web.config** file underneath the **<appSettings>** opening tag.

<add key="aad-account-name" value="" />

<add key="aad-account-password" value=" " />

<add key="app-workspace-id" value="" />

<add key="dataset-id" value="" />

<add key="report-id" value="" />

<add key="dashboard-id" value=" " />

<add key="client-id" value="" />

* 1. Copy configuration values from **EmbeddingConfigurationData.txt** into the new **appSetting** values in **web.config.**
  2. You should be able to supply values for each of the seven appSetting values as shown in the following screenshot.



* 1. Save your changes and close **web.config**.

1. Create classes to provide MVC view models for Power BI Embedding data.
   1. Add a new C# source file named **MvcViewModels.cs** inside the **Models** folder.



* 1. If there is any code inside **MvcViewModels.cs**, delete it and replace it with the following code.

namespace EmbeddedLab.Models {

// data required for embedding a report

public class ReportEmbeddingData {

public string reportId;

public string reportName;

public string embedUrl;

public string accessToken;

}

// data required for embedding a new report

public class NewReportEmbeddingData {

public string workspaceId;

public string datasetId;

public string embedUrl;

public string accessToken;

}

// data required for embedding a dashboard

public class DashboardEmbeddingData {

public string dashboardId;

public string dashboardName;

public string embedUrl;

public string accessToken;

}

// data required for embedding a dashboard

public class QnaEmbeddingData {

public string datasetId;

public string embedUrl;.

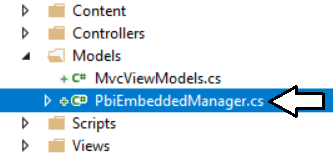
public string accessToken;

}

}

* 1. Save your changes and close **MvcViewModels.cs**.

1. Create the **PbiEmbeddedManager** class.
   1. Add a new class named **PbiEmbeddedManager** inside the **Models** folder.
   2. The **Models** folder should now contain a C# source file named **PbiEmbeddedManager.cs**.



* 1. Delete any code inside **PbiEmbeddedManager.cs** and replace it with the following starter code.

using System;

using System.Configuration;

using System.Threading.Tasks;

using Microsoft.Rest;

using Microsoft.PowerBI.Api.V2;

using Microsoft.PowerBI.Api.V2.Models;

using Microsoft.IdentityModel.Clients.ActiveDirectory;

namespace EmbeddedLab.Models {

public class PbiEmbeddedManager {

}

}

* 1. Modify the **PbiEmbeddedManager** class by adding the following set of static fields.

class PbiEmbeddedManager {

private static string aadAuthorizationEndpoint = "https://login.windows.net/common/oauth2/authorize";

private static string resourceUriPowerBi = "https://analysis.windows.net/powerbi/api";

private static string urlPowerBiRestApiRoot = "https://api.powerbi.com/";

private static string userName = ConfigurationManager.AppSettings["aad-account-name"];

private static string userPassword = ConfigurationManager.AppSettings["aad-account-password"];

private static string workspaceId = ConfigurationManager.AppSettings["app-workspace-id"];

private static string datasetId = ConfigurationManager.AppSettings["dataset-id"];

private static string reportId = ConfigurationManager.AppSettings["report-id"];

private static string dashboardId = ConfigurationManager.AppSettings["dashboard-id"];

private static string clientId = ConfigurationManager.AppSettings["client-id"];

}

In addition to fields for the seven configuration values, there are other fields named **aadAuthorizationEndpoint**, **resourceUriPowerBi** and **urlPowerBiRestApiRoot** which are used when authenticating with Azure AD and calling to the Power BI Service API.

* 1. At the bottom of **PbiEmbeddedManager** class, add a new method named **GetAccessToken** using the following code.

private static string GetAccessToken() {

AuthenticationContext authenticationContext = new AuthenticationContext(aadAuthorizationEndpoint);

AuthenticationResult userAuthnResult =

authenticationContext.AcquireTokenAsync(

resourceUriPowerBi,

clientId,

new UserPasswordCredential(userName, userPassword)).Result;

return userAuthnResult.AccessToken;

}

* 1. Underneath the **GetAccessToken** method, add a new method named **GetPowerBiClient** using the following code.

private static PowerBIClient GetPowerBiClient() {

var tokenCredentials = new TokenCredentials(GetAccessToken(), "Bearer");

return new PowerBIClient(new Uri(urlPowerBiRestApiRoot), tokenCredentials);

}

You have implemented the essential behavior in the **PbiEmbeddedManager** class to authenticate with Azure AD and to create new **PowerBIClient** objects which represents the top-level entry point into the Power BI Service API. Now you are at a point where you can add methods to the **PbiEmbeddedManager** class which call into the Power BI Service API to retrieve embedding data.

1. Add the **GetReportEmbeddingData** method to the **PbiEmbeddedManager** class.
   1. At the bottom of the **PbiEmbeddedManager** class, add a method named **GetReportEmbeddingData** with the following code.

public static async Task<ReportEmbeddingData> GetReportEmbeddingData() {

PowerBIClient pbiClient = GetPowerBiClient();

var report = await pbiClient.Reports.GetReportInGroupAsync(workspaceId, reportId);

var embedUrl = report.EmbedUrl;

var reportName = report.Name;

GenerateTokenRequest generateTokenRequestParameters = new GenerateTokenRequest(accessLevel: "edit");

string embedToken =

(await pbiClient.Reports.GenerateTokenInGroupAsync(workspaceId,

report.Id,

generateTokenRequestParameters)).Token;

return new ReportEmbeddingData {

reportId = reportId,

reportName = reportName,

embedUrl = embedUrl,

accessToken = embedToken

};

}

* 1. Save your changes to **PbiEmbeddedManger.cs**.

Now that you have added the **GetReportEmbeddingData** method, you will create a new action method that calls this method.

1. Add the **Report** action method to the **HomeController** class.
   1. Inside the **Controllers** folder, open the C# source file named **HomeController.cs**.
   2. Update the set of **using** statements at the top of **HomeController.cs** using the following code.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Web;

using System.Web.Mvc;

using EmbeddedLab.Models;

* 1. Underneath the **Index** method, add a new asynchronous action method named **Report** using the following code.

public class HomeController : Controller {

public ActionResult Index() {

return View();

}

public async Task<ActionResult> Report() {

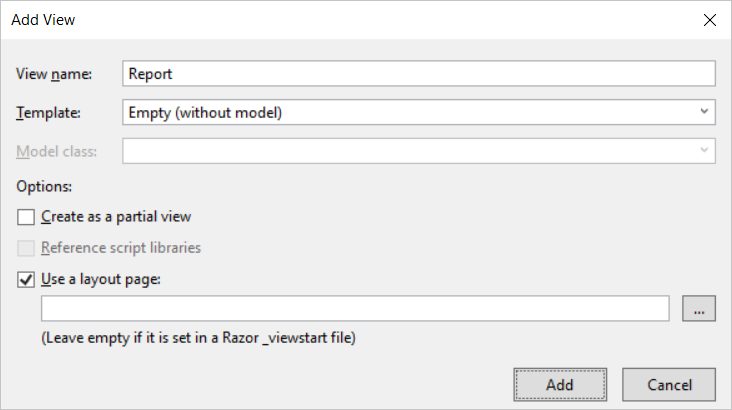
ReportEmbeddingData embeddingData = await PbiEmbeddedManager.GetReportEmbeddingData();

return View(embeddingData);

}

}

* 1. Right-click on the **Report** method and select the **Add View…** command from the context menu.
  2. In the **Add View** dialog, accept all the default settings and click the **Add** button.



* 1. You should be able to verify that a new razor view file named **Report.cshtml** has been created in the **Views/Home** folder.
  2. Delete all existing content from **Report.cshtml** and replace it with the following code.

@model EmbeddedLab.Models.ReportEmbeddingData

<div id="embedContainer" />

<script src="~/Scripts/powerbi.js"></script>

<script>

// data required for embedding Power BI report

var embedReportId = "@Model.reportId";

var embedUrl = "@Model.embedUrl";

var accessToken = "@Model.accessToken";

// Get models object to access enums for embed configuration

var models = window['powerbi-client'].models;

var config = {

type: 'report',

id: embedReportId,

embedUrl: embedUrl,

accessToken: accessToken,

permissions: models.Permissions.All,

tokenType: models.TokenType.Embed,

viewMode: models.ViewMode.View,

settings: {

filterPaneEnabled: false,

navContentPaneEnabled: true,

}

};

// Get a reference to HTML element that will be embed container

var reportContainer = document.getElementById('embedContainer');

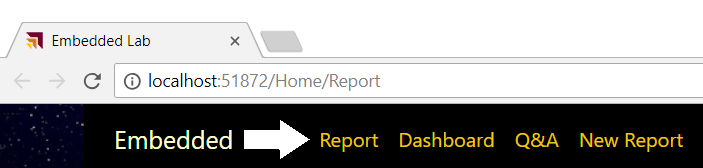
// Embed the report and display it within the div container.

var report = powerbi.embed(reportContainer, config);

</script>

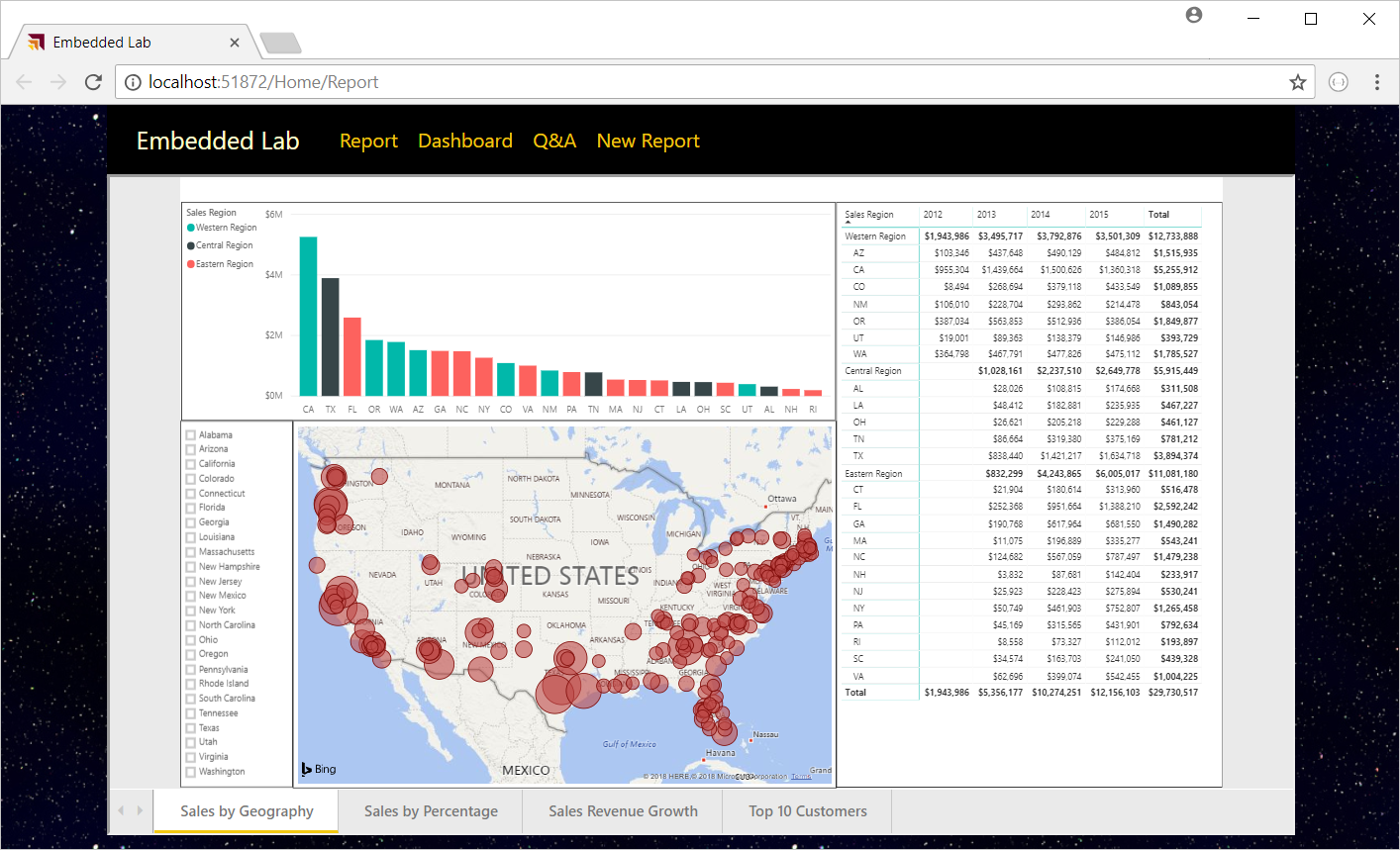
* 1. Save your changes and close **Report.cshtml**.

1. Test out the application by running it in the Visual Studio debugger.
   1. Press the **{F5}** key in Visual Studio to begin a new debugging session.
   2. Click the **Report** link in the top navigation menu.



If the editor window with a razor view such as **Report.cshtml** is the active window when you press the **{F5}**, the Visual Studio debugger will automatically take you to this view at the start of your debugging session.

* 1. You should now see the report has been embedded on the web page.



Try resizing the browser window. You will see that your application responds by dynamically changing the size of the HTML element with the ID of **embedContainer** and the embedded report responds automatically by changing its size to fit the new dimensions.

* 1. Close the browser window and return to Visual Studio and stop the current debugging session.

## Add an Interactive Toolbar for an Embedded Report

In this exercise, you continue to work on **Report.cshtml** by adding a new toolbar with three command buttons. You will also add JavaScript code behind these command buttons to invoke actions on the embedded report.

1. Add the HTML layout code for a toolbar into **Report.cshtml**.
   1. Open **Report.cshtml** if it is not already open.
   2. Copy and paste the following HTML code into **Report.cshtml** just below the **@model** directive at the top.

@section toolbar {

<div id="toolbar" class="btn-toolbar bg-dark" role="toolbar" >

<button type="button" id="toggleEdit" class="btn btn-sm" >Toggle Edit Mode</button>

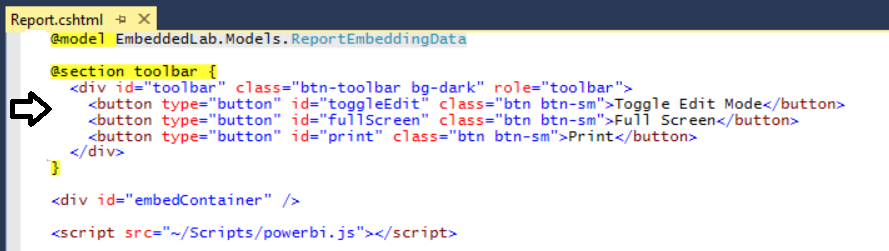
<button type="button" id="fullScreen" class="btn btn-sm" >Full Screen</button>

<button type="button" id="print" class="btn btn-sm" >Print</button>

</div>

}

* 1. The top of **Report.cshtml** should match the following screenshot.



* 1. Inside **Report.cshtml**, move down inside **<script>** block and add a few new lines after the line which calls **powerbi.embed**.
  2. Copy and paste the following JavaScript code at the bottom of the **<script>** block just before the close **</script>** tag.

var viewMode = "view";

$("#toggleEdit").click(function () {

// toggle between view and edit mode

viewMode = (viewMode == "view") ? "edit" : "view";

report.switchMode(viewMode);

// show filter pane when entering edit mode

var showFilterPane = (viewMode == "edit");

report.updateSettings({

"filterPaneEnabled": showFilterPane

});

});

$("#fullScreen").click(function () {

report.fullscreen();

});

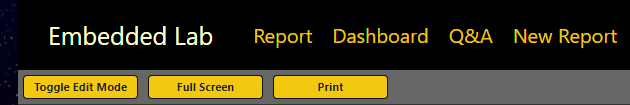
$("#print").click(function () {

report.print();

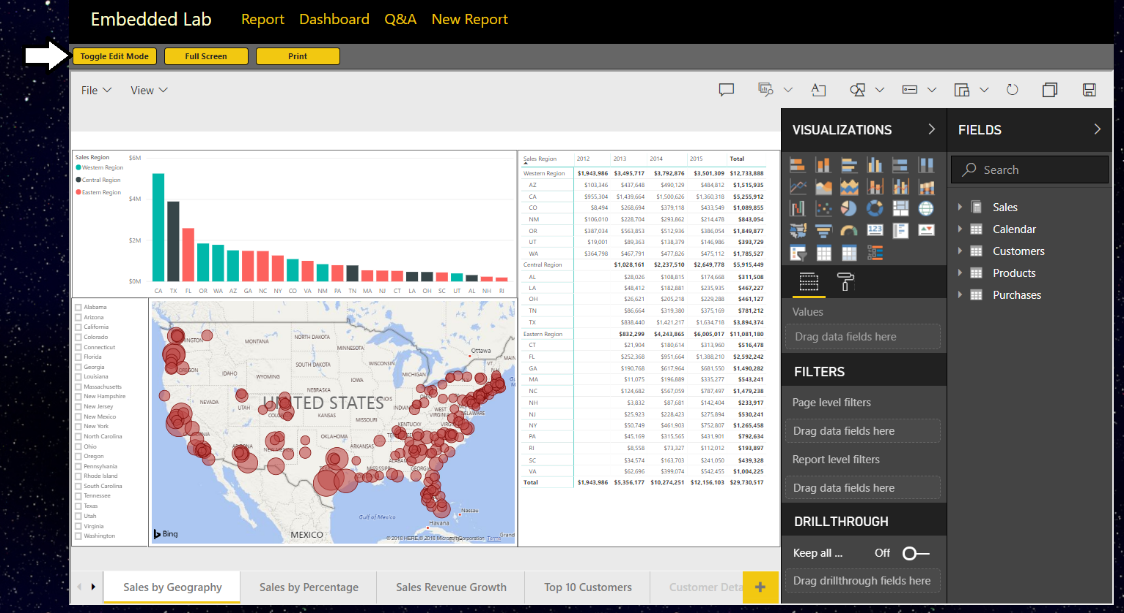
});

* 1. Save your changes to **Report.cshtml**.

1. Test out the application by running it in the Visual Studio debugger.
   1. Press the **{F5}** key in Visual Studio to begin a new debugging session.
   2. Click the **Report** link in the top navigation menu.
   3. You should now see three toolbar buttons with the captions **Toggle Edit Mode**, **Full Screen** and **Print**.



* 1. Click the **Toggle Edit Mode** button several times. The report should toggle back and forth between edit and reader mode.



* 1. Experiment by clicking the **Full Screen** button.
  2. Experiment by clicking the **Print button**.
  3. Close the browser window and return to Visual Studio and stop the current debugging session.

## Embed a Dashboard

In this exercise you will embed a dashboard. As you will see, it’s not very different from the steps you have already implemented to embed a report.

1. Add a new method to the **PbiEmbeddingManger** class named **GetDashboardEmbeddingData**.
   1. Open **PbiEmbeddedManager.cs** in an editor window if it’s not already open.
   2. Navigate to the bottom of the class definition just beneath the **GetReportEmbeddingData** method
   3. Paste in the definition for a new method named **GetDashboardEmbeddingData** using the following code.

public static async Task<DashboardEmbeddingData> GetDashboardEmbeddingData() {

PowerBIClient pbiClient = GetPowerBiClient();

var dashboard = await pbiClient.Dashboards.GetDashboardInGroupAsync(workspaceId, dashboardId);

var embedUrl = dashboard.EmbedUrl;

var dashboardDisplayName = dashboard.DisplayName;

GenerateTokenRequest generateTokenRequestParameters = new GenerateTokenRequest(accessLevel: "view");

string embedToken =

(await pbiClient.Dashboards.GenerateTokenInGroupAsync(workspaceId,

dashboardId,

generateTokenRequestParameters)).Token;

return new DashboardEmbeddingData {

dashboardId = dashboardId,

dashboardName = dashboardDisplayName,

embedUrl = embedUrl,

accessToken = embedToken

};

}

* 1. Save your changes to **PbiEmbeddedManager.cs**.

1. Add a new action method to the **HomeController** class named **Dashboard**.
   1. Open **HomeController.cs** in an editor window if it’s not already open.
   2. Add a new action method named **Dashboard** just beneath the **Report** method using the following code.

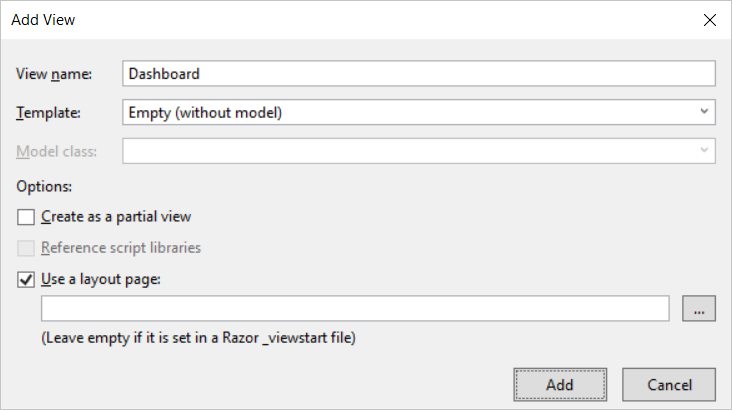
public async Task<ActionResult> Dashboard() {

DashboardEmbeddingData embeddingData = await PbiEmbeddedManager.GetDashboardEmbeddingData();

return View(embeddingData);

}

1. Create a razor view for the **Dashboard** action method.
   1. Right-click on the **Dashboard** action method and select the **Add View…** command from the context menu.
   2. In the **Add View** dialog, accept all the default settings and click the **Add** button.



* 1. You should see that a new razor view file named **Dashboard.cshtml** has been created in the **Views/Home** folder.
  2. Delete any existing code inside **Dashboard.cshtml** and replace it with the following HTML code.

@model EmbeddedLab.Models.DashboardEmbeddingData

<div id="embedContainer" />

<script src="~/Scripts/powerbi.js"></script>

<script>

// data required for embedding Power BI report

var embedDashboardId = "@Model.dashboardId";

var embedUrl = "@Model.embedUrl";

var accessToken = "@Model.accessToken";

// Get models object to access enums for embed configuration

var models = window['powerbi-client'].models;

var config = {

type: 'dashboard',

id: embedDashboardId,

embedUrl: embedUrl,

accessToken: accessToken,

tokenType: models.TokenType.Embed,

pageView: "fitToWidth"

};

// Get a reference to the embedded report HTML element

var embedContainer = document.getElementById('embedContainer');

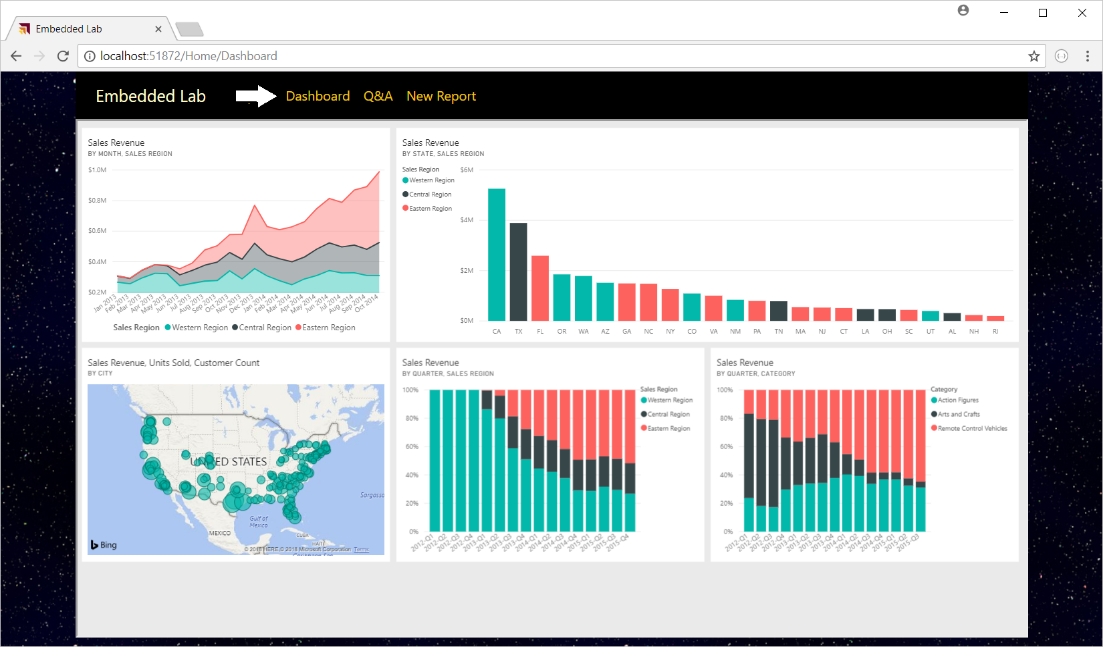
// Embed the report and display it within the div container.

var dashboard = powerbi.embed(embedContainer, config);

</script>

* 1. Save your changes to **Dashboard.cshtml**.

1. Test out the application by running it in the Visual Studio debugger.
   1. Press the **{F5}** key in Visual Studio to begin a new debugging session.
   2. Click the **Dashboard** link in the top navigation menu and you should see the dashboard embedded in the web page.



* 1. Try changing the size of the browser window and see how the application responds by adjusting the size of the dashboard.
  2. Close the browser window and return to Visual Studio and stop the current debugging session.

## Embed the Power BI Q&A Experience

In this exercise you will embed the Power BI Q&A experience. To accomplish this, you will be required to provide the dataset ID associated with the dataset on which you want to execute natural language queries.

1. Add a new method to the **PbiEmbeddingManger** class named **GetQnaEmbeddingData**.
   1. Open **PbiEmbeddedManager.cs** in an editor if it’s not already open.
   2. Navigate to the bottom of the class definition just beneath the **GetDashboardEmbeddingData** method
   3. Add a new method named **GetQnaEmbeddingData** by copying and pasting the following code.

public async static Task<QnaEmbeddingData> GetQnaEmbeddingData() {

PowerBIClient pbiClient = GetPowerBiClient();

var dataset = await pbiClient.Datasets.GetDatasetByIdInGroupAsync(workspaceId, datasetId);

string embedUrl = "https://app.powerbi.com/qnaEmbed?groupId=" + workspaceId;

string datasetID = dataset.Id;

GenerateTokenRequest generateTokenRequestParameters = new GenerateTokenRequest(accessLevel: "view");

string embedToken =

(await pbiClient.Datasets.GenerateTokenInGroupAsync(workspaceId,

dataset.Id,

generateTokenRequestParameters)).Token;

return new QnaEmbeddingData {

datasetId = datasetId,

embedUrl = embedUrl,

accessToken = embedToken

};

}

* 1. Save your changes to Open **PbiEmbeddedManager.cs**.

1. Add a new action method to the **HomeController** class named **Qna**.
   1. Open **HomeController.cs** in an editor window if it’s not already open.
   2. Add a new action method named **Qna** just beneath the **Dashboard** method using the following code.

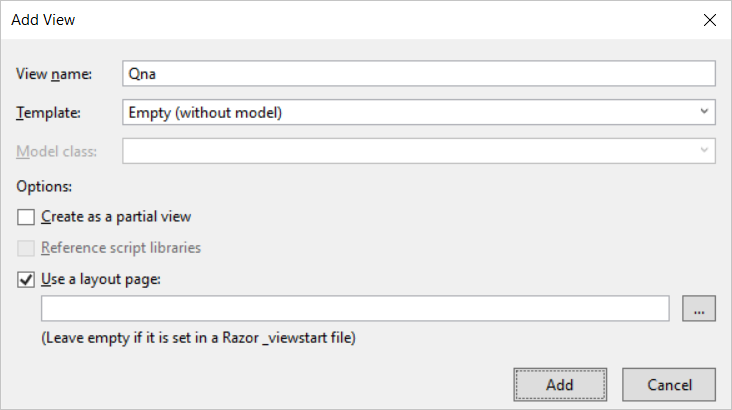
public async Task<ActionResult> Qna() {

QnaEmbeddingData embeddingData = await PbiEmbeddedManager.GetQnaEmbeddingData();

return View(embeddingData);

}

1. Create a razor view for the **Qna** action method.
   1. Right-click on the **Qna** action method and select the **Add View…** command from the context menu.
   2. In the **Add View** dialog, accept all the default settings and click the **Add** button.



* 1. You should see that a new razor view file named **Qna.cshtml** has been created in the **Views/Home** folder.
  2. Delete any existing code inside **Qna.cshtml** and replace it with the following HTML code.

@model EmbeddedLab.Models.QnaEmbeddingData

<div id="embedContainer" />

<script src="~/Scripts/powerbi.js"></script>

<script>

// Get data required for embedding

var datasetId = "@Model.datasetId";

var embedUrl = "@Model.embedUrl";

var accessToken = "@Model.accessToken";

// Get models object to access enums for embed configuration

var models = window['powerbi-client'].models;

var config = {

type: 'qna',

tokenType: models.TokenType.Embed,

accessToken: accessToken,

embedUrl: embedUrl,

datasetIds: [datasetId],

viewMode: models.QnaMode.Interactive,

question: "What is sales revenue by quarter and sales region as stacked area chart"

};

// Get a reference to the embedded report HTML element

var embedContainer = document.getElementById('embedContainer');

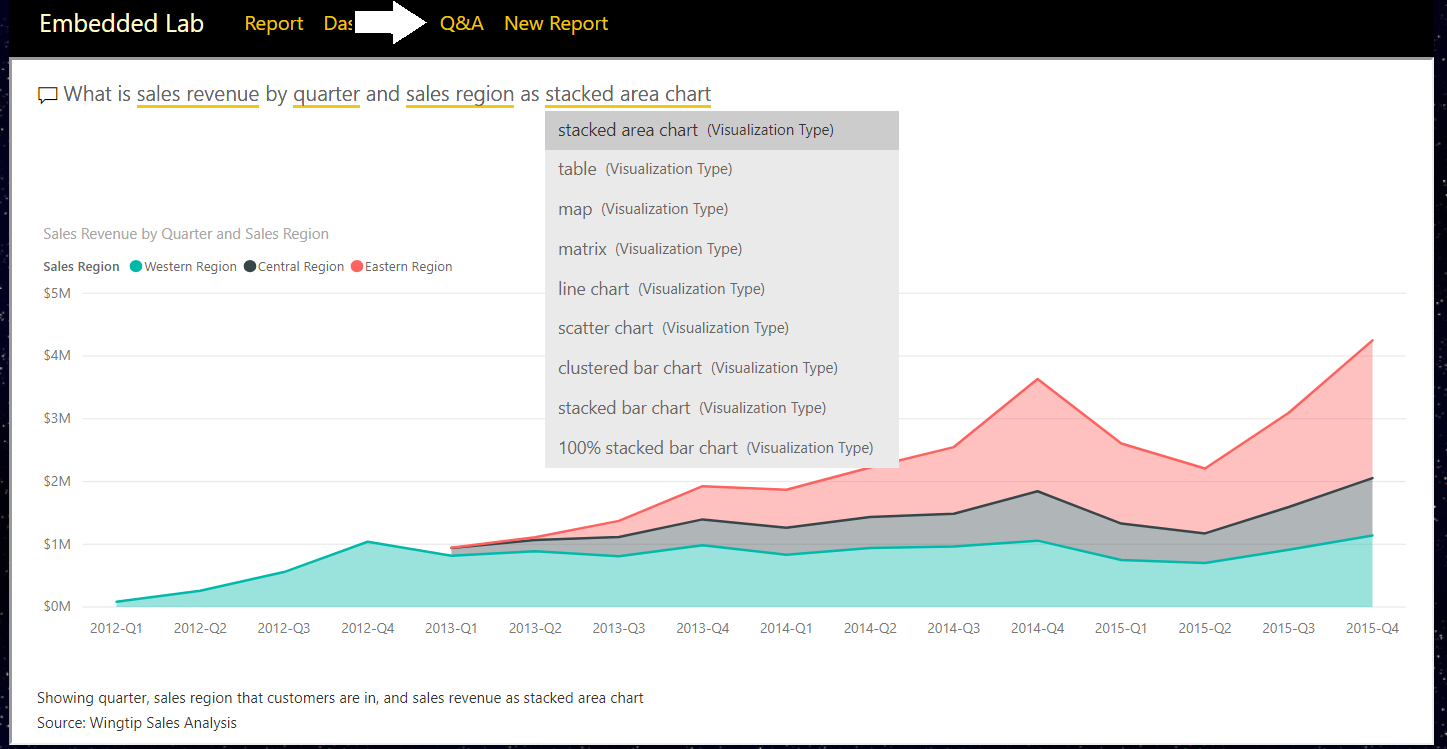
// Embed the report and display it within the div container.

var embeddedObject = powerbi.embed(embedContainer, config);

</script>

* 1. Save your changes to **Qna.cshtml**.

1. Test out the application by running it in the Visual Studio debugger.
   1. Press the **{F5}** key in Visual Studio to begin a new debugging session.
   2. Click the **Q&A** link in the top navigation menu and you should see the Q&A experience embedded in the web page.



* 1. Experiment by typing questions in English and seeing how the Q&A experience responds with data and visualizations.
  2. Close the browser window and return to Visual Studio and stop the current debugging session.

## Embed a New Report

In this exercise you will implement the behavior to embed a new report based on a specific dataset. This exercise will be a bit more complicated than the previous exercises because you must implement a client-side event handler to handle the report “Save As” event in which you will redirect the browser to a new action method named **Reports** passing the new report ID in a query string parameter.

1. Add a new method to the **PbiEmbeddingManger** class named **GetNewReportEmbeddingData**.
   1. Open **PbiEmbeddedManager.cs** in an editor window if it is not already open.
   2. Navigate to the bottom of the class definition just beneath the **GetQnaEmbeddingData** method.
   3. Add a new method named **GetNewReportEmbeddingData** by copying and pasting the following code.

public static async Task<NewReportEmbeddingData> GetNewReportEmbeddingData() {

string embedUrl = "https://app.powerbi.com/reportEmbed?groupId=" + workspaceId;

PowerBIClient pbiClient = GetPowerBiClient();

GenerateTokenRequest generateTokenRequestParameters =

new GenerateTokenRequest(accessLevel: "create", datasetId: datasetId);

string embedToken =

(await pbiClient.Reports.GenerateTokenForCreateInGroupAsync(workspaceId,

generateTokenRequestParameters)).Token;

return new NewReportEmbeddingData {

workspaceId = workspaceId,

datasetId = datasetId,

embedUrl = embedUrl,

accessToken = embedToken

};

}

Notice that you are required to pass a dataset ID when generating an embed token which will be used to embed a new report.

1. Add a new action method to the **HomeController** class named **NewReport**.
   1. Open **HomeController.cs** in an editor window if it’s not already open.
   2. Add a new action method named **NewReport** just beneath the **Qna** method using the following code.

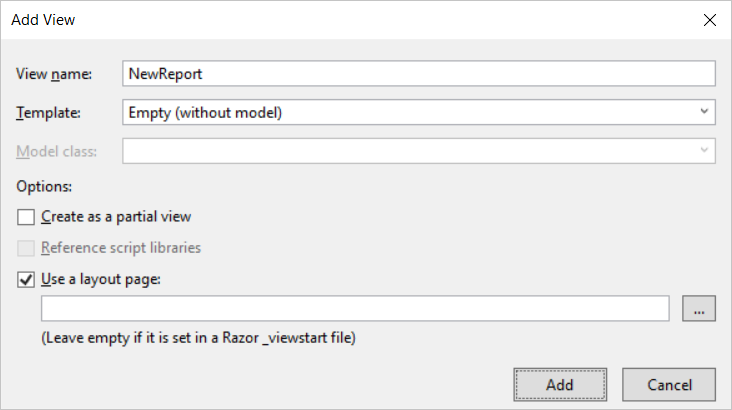
public async Task<ActionResult> NewReport() {

NewReportEmbeddingData embeddingData = await PbiEmbeddedManager.GetNewReportEmbeddingData();

return View(embeddingData);

}

1. Create a razor view for the **NewReport** action method.
   1. Right-click on the **NewReport** action method and select the **Add View…** command from the context menu.
   2. In the **Add View** dialog, accept all the default settings and click the **Add** button.



* 1. You should see that a new razor view file named **NewReport.cshtml** has been created in the **Views/Home** folder.
  2. Delete any existing code inside **NewReport.cshtml** and replace it with the following HTML code.

@model EmbeddedLab.Models.NewReportEmbeddingData

<div id="embedContainer" />

<script src="~/Scripts/powerbi.js"></script>

<script>

// Get data required for embedding

var embedWorkspaceId = "@Model.workspaceId";

var embedDatasetId = "@Model.datasetId";

var embedUrl = "@Model.embedUrl";

var accessToken = "@Model.accessToken";

// Get models object to access enums for embed configuration

var models = window['powerbi-client'].models;

var config = {

datasetId: embedDatasetId,

embedUrl: embedUrl,

accessToken: accessToken,

tokenType: models.TokenType.Embed,

};

// Get a reference to the embedded report HTML element

var embedContainer = document.getElementById('embedContainer');

// Embed the report and display it within the div container.

var report = powerbi.createReport(embedContainer, config);

// add event handler to load existing report afer saving new report

report.on("saved", function (event) {

console.log("saved");

console.log(event.detail);

window.location.href = "/Home/Reports/?reportId=" + event.detail.reportObjectId;

});

</script>

* 1. Save your changes to **NewReport.cshtml**.

You should observe how the code in this script block registers a callback function by calling the **report.on(“Saved”)** method. You should also observe that this event handle is written to redirect the browser to the **Reports** action of the **Home** controller along with a query string parameter named **reportId** which will be used to pass the identifying GUID of the newly created report. Over the next few steps you will create the **Reports** action method in the **Home** controller class to load an existing report that has just been created.

1. Add a new method to the **PbiEmbeddingManger** class named **GetEmbeddingDataForReport**.
   1. In **PbiEmbeddedManager.cs**, add the **GetEmbeddingDataForReport** method by copying and pasting the following code.

public static async Task<ReportEmbeddingData> GetEmbeddingDataForReport(string currentReportId) {

PowerBIClient pbiClient = GetPowerBiClient();

var report = await pbiClient.Reports.GetReportInGroupAsync(workspaceId, currentReportId);

var embedUrl = report.EmbedUrl;

var reportName = report.Name;

GenerateTokenRequest generateTokenRequestParameters = new GenerateTokenRequest(accessLevel: "edit");

string embedToken =

(await pbiClient.Reports.GenerateTokenInGroupAsync(workspaceId,

currentReportId,

generateTokenRequestParameters)).Token;

return new ReportEmbeddingData {

reportId = currentReportId,

reportName = reportName,

embedUrl = embedUrl,

accessToken = embedToken

};

}

1. Add a new action method to the **HomeController** class named **Reports**.
   1. Open **HomeController.cs** in an editor window if it’s not already open.
   2. Add a new action method named **Reports** just beneath the **NewReports** method using the following code.

public async Task<ActionResult> Reports(string reportId) {

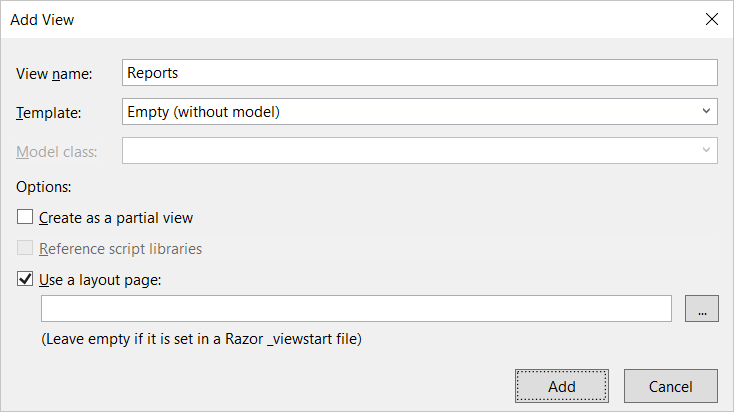
ReportEmbeddingData embeddingData =

await PbiEmbeddedManager.GetEmbeddingDataForSpecificReport(reportId);

return View(embeddingData);

}

1. Create a razor view for the **Reports** action method.
   1. Right-click on the R**eports** action method and select the **Add View…** command from the context menu.
   2. In the **Add View** dialog, accept all the default settings and click the **Add** button.



A page break has been inserted here to prevent the following code section from wrapping across pages.

* 1. You should see that a new razor view file named **Reports.cshtml** has been created in the **Views/Home** folder.
  2. Delete any existing code inside **Reports.cshtml** and replace it with the following HTML code.

@model EmbeddedLab.Models.ReportEmbeddingData

@section toolbar {

<div id="toolbar" class="btn-toolbar bg-dark" role="toolbar">

<button type="button" id="toggleEdit" class="btn btn-sm">Toggle Edit Mode</button>

<button type="button" id="fullScreen" class="btn btn-sm">Full Screen</button>

<button type="button" id="print" class="btn btn-sm">Print</button>

</div>

}

<div id="embedContainer" />

<script src="~/Scripts/powerbi.js"></script>

<script>

// Data required for embedding Power BI report

var embedReportId = "@Model.reportId";

var embedUrl = "@Model.embedUrl";

var accessToken = "@Model.accessToken";

// Get models object to access enums for embed configuration

var models = window['powerbi-client'].models;

var config = {

type: 'report',

id: embedReportId,

embedUrl: embedUrl,

accessToken: accessToken,

tokenType: models.TokenType.Embed,

permissions: models.Permissions.All,

viewMode: models.ViewMode.Edit,

settings: {

filterPaneEnabled: false,

navContentPaneEnabled: true,

}

};

// Get a reference to HTML element that will be embed container

var reportContainer = document.getElementById('embedContainer');

// Embed the report and display it within the div container.

var report = powerbi.embed(reportContainer, config);

var viewMode = "edit";

$("#toggleEdit").click(function () {

// toggle between view and edit mode

viewMode = (viewMode == "view") ? "edit" : "view";

report.switchMode(viewMode);

// show filter pane when entering edit mode

var showFilterPane = (viewMode == "edit");

report.updateSettings({

"filterPaneEnabled": showFilterPane

});

});

$("#fullScreen").click(function () {

report.fullscreen();

});

$("#print").click(function () {

report.print();

});

</script>

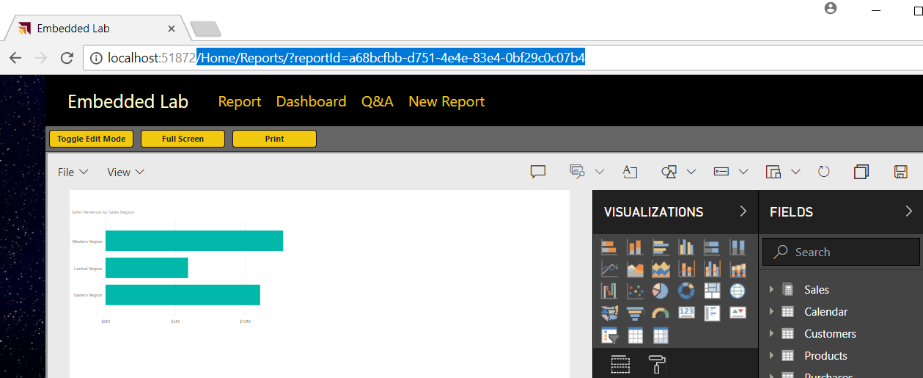
1. Test out the application by running it in the Visual Studio debugger.
   1. Press the **{F5}** key in Visual Studio to begin a new debugging session.
   2. Click the **New Report** link in the top navigation menu and you should see an new empty in design mode.
   3. Add a simple visual to the new report.
   4. Save the new report by dropping down the **File** menu and selecting the **Save as** command.



* 1. In the **Save your report** dialog, give the new report a name such as **My New Report** and click the **Save** button.



* 1. After the report has been saved, the browser should redirect to the **Home/Reports** action method and your application should be able to load in the newly created report using the GUID for its report ID.



* 1. When you are done with your testing, close the browser, return to Visual Studio and stop the current debugging session.

## Download the Sample Projects

This whitepaper contains dozens code listings to illustrate programming techniques with Power BI embedding. The majority of these code listings have been taken from the following set of five sample development projects that are maintained in the same GitHub repository that contains this whitepaper.

* **DailyReporterPro**: an ASP.NET MVC application which demonstrate third-party embedding (Visual Studio 2017)
* **DailyReporterPersonal**: an ASP.NET MVC application which demonstrate first-party embedding (Visual Studio 2017)
* **PowerBiDaySpa**: Client-side SPA created with JavaScript and jQuery (Visual Studio 2017)
* **powerbi-embed-react-demo**: Client-side SPA created with TypeScript and React (Node.js & Visual Studio Code)
* **powerbi-spfx-webparts**: SharePoint Framework Webpart created with TypeScript and React (Node.js & Visual Studio Code)

You are encouraged to download these sample projects and get them running in your own Power BI embedding developer environment. For many developers, there is no substitute for the spontaneous gratification of being able to press the {F5} key and see the code running right before their eyes. The first three projects are based on Visual Studio 2017 and the classic Visual Studio development experience. The other two projects are based on modern development with Node.js and can be opened and run using Visual Studio Code.

# Authenticating with Azure AD

In the Developer Quick Start, you learned to use the Power BI Embedding Onboarding Experience to provide a quick and easy way to set up a Power BI embedding development environment. Behind the scenes, the Onboarding Experience wizard called into Azure AD to create and configure a new Azure AD application on your behalf. While the Onboarding Experience is a convenient way to get started, it will only take you so far. If you want to develop with Power BI embedding for real-world scenarios, you must learn how authentication and authorization works in Azure AD at a fundamental level.

Your motivation for learning to work with Azure AD stems from the Power BI embedding requirement to call the Power BI Service API. To successfully call the Power BI Service API, you must first acquired an access token from Azure AD. Now, here is where things can get complicated because the manner in which you acquire access tokens varies significantly depending upon the scenario for which you are developing. Therefore, this section will begin with a quick primer on how authentication and authorization works in Azure AD from the ground up.

## OAuth 2.0 and OpenID Connect

Azure AD provides authentication and authorization services using two key open standards named OAuth 2.0 and OpenID Connect. *OAuth 2.0* is an open standard for an authorization framework based on distributing access tokens to client applications. *OpenID Connect* is another open standard which layers on top of OAuth 2.0 to fill in a few missing pieces with respect to user authentication and identity.

It's not really as confusing as it sounds. You don't have to think about *OAuth 2.0* and *OpenID Connect* separately. Instead, these two open standards combine to create a single protocol with one set of rules. When you hear developers talk about OAuth with Azure AD, it’s usually implied that they are talking about *OAuth 2.0* combined with *OpenID Connect*.

The authorization framework of OAuth 2.0 defines four roles in the authorization process including the client, the resource owner, the authorization server and resource servers as shown in Figure 3.1. The *client* is the custom application you are developing and the *resource owner* is the user who is using your application. The *resource servers* represent secured endpoints on the Internet which you need to access such as the Microsoft Graph API and the Power BI Service API.

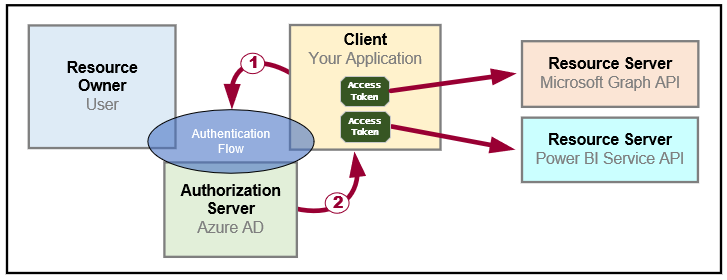


Figure 3.1: OAuth 2.0 provides a framework for distributing access tokens to client applications.

At the heart of OAuth 2.0 lies the process of transferring an access token from the authorization server to the client in a safe and secure manner. This process is known as an *authentication flow*. The client implements an authentication flow to retrieve access tokens from the authorization server. You must choose the type of authentication flow for each application you build depending on upon the details of your development scenario. OAuth 2.0 defines the following types of authentication flows to handle scenarios for web applications, desktop applications, mobile phones, and smart devices.

* **User Password Credential Flow**: used authenticate the user in desktop and mobile applications
* **Authorization Code Flow**: used to authenticate the user in web applications
* **Implicit Flow**: used to authenticate the user in a SPA (i.e. single page application with only client-side code)
* **Client Credentials Flow**: used to authenticate the application with an app-only identity

Once the client has used an authentication flow to acquire an access token, the client then must transmit the access token whenever it executes an HTTP operation against the target resource server. The resource server trusts the authorization server and therefore trusts whatever information it finds inside the access token. This allows the resource server to inspect an access token to discover what permissions have been granted to the client.

A key principle in OAuth 2.0 is that of *delegated access*. This is different than impersonation because your application will not call to a resource server using the identity and permissions of the user. Instead, your application gets its own separate identity which is granted a subset of the user's permissions. When using delegated access, your application calls to a resource server *on behalf of the user* instead of calling *as the user*.

The authorization server generates access tokens using requirements and a format defined by OAuth 2.0. An access token will always contain several IDs including an ID for the client, an ID the authorization server and an ID for the target resource server. In many cases, the access token also includes a user-specific ID for the resource owner. However, this is not always the case because you can also implement an authentication flow to acquire an app-only access token. Since app-only authentication does not involve any user, the resulting access token will not contain an ID for a resource owner.

Keep in mind that every access token is generated for one specific resource. As an example, consider a scenario in which you are developing an application that needs to call both the Power BI Service API and the Microsoft Graph API. You will need acquire a separate access token for each API because they are recognized as separate resources.

When you implement an authentication flow to acquire an access token, you pass a *resource identifier* to the authorization server to indicate the target resource. A resource identifier is really just a unique string but it's formatted as a URI. The resource identifier for the Power BI Service API is *https://analysis.windows.net/powerbi/api*. The resource identifier for the Microsoft Graph API is *https://graph.microsoft.com*.

An access token is often referred to as a *bearer token*. The key point here is that any party that obtains an access token (*i.e. the bearer*) can take advantage of the permissions that have been granted inside. You can make the analogy that an access token is like cash and not like your ATM card which requires you to know a PIN to use it. An attacker who can capture your access token has the ability to use it and to compromise your application's security enforcement policies. Therefore, access tokens should always passed across the network in an encrypted form using SSL and HTTPS and never in clear text with just HTTP.

An access token contains a *duration which* defines the lifetime for which the access token is valid. Access tokens are given a relatively short lifetime to decrease the surface area for attackers. For example, Azure AD generates access tokens with a duration of 65 minutes. Once an access token expires, any attempt to use it with result in an access denied error.

The OAuth 2.0 framework provides *refresh tokens* to deal with the problem of expired access tokens. During certain types of authentication flows, the authorization server passes a refresh token to the client in addition to the access token. While an access token expires in about an hour, a refresh token is valid for 90 days. Once the original access token expires, the client uses the refresh token as its credentials when it calls Azure AD to acquire a new access token. Given the long lifetime of a refresh token, it is common for the client to persist refresh tokens so they can be used across user sessions. The use of refresh tokens makes it possible to reduce the number of times a user is prompted with an interactive login and required to enter a user name and password.

As noted earlier, the OAuth 2.0 framework has a few shortcomings with respect to user authentication and identity. These shortcomings are addressed by OpenID Connect which adds a third type of security token known as an **ID token**. An OD token contains additional data on the authenticated user and makes it possible for the client to validate the user's identity.

There is one last fundamental detail you need to know about how OAuth 2.0 works. The OAuth 2.0 framework requires you to register the client application with the authorization server before it can be used. When an application is registered, it must be assigned an ID, a friendly name and a set of permissions as shown in Figure 3.2. Depending upon the type of application you are creating, you might also need to configure the registered application with reply URLs and credentials.



Figure 3.2 provide a high-level view of the data tracked by azure AD when you register an application.

The process of registering an application is somewhat analogous to creating a new user account. A new user account is created with a login ID and a set of credentials allowing the user to successfully log in and establish an identity. In a similar fashion, a registered application is created with an application ID and an optional set of credentials which allows the client application to authenticate with the authorization server to establish its identity.

Now that you have learned the fundamentals of OAuth 2.0 and OpenID Connect, it's time to put all this theory to work and begin discussing what needs to be done in Azure AD to register and use a custom application. It is important to know that Azure AD currently supports two different implementations of OAuth 2.0 and OpenID Connect. These two different implementation go by the names of the v1.0 endpoint and the v2.0 endpoint.

The Azure AD v1.0 endpoint has been generally available for over 5 years and it is heavily used in production applications. You can register and configure Azure AD applications for the v1.0 endpoint using the Azure portal or a PowerShell script. The v2.0 endpoint and its user experience for registering applications is still in a preview program. Furthermore, the v2.0 endpoint only supports a subset of the features available through the v1.0 endpoint. Therefore, the section will focus on the Azure AD v1.0 endpoint and then briefly discuss the changes you can expect in the future if you decide to move to the v2.0 endpoint.

## Creating Azure AD Applications

The way to register a client application with Azure AD is to create a new Azure AD application. For production scenarios you can create new Azure AD applications using a PowerShell script or by using the Microsoft Graph API. When you are just getting started with Azure AD, the easiest way to create an Azure AD application is to use the Azure portal. If you navigate to the **Azure Active Directory** link in the Azure portal and then click the **App registrations**, you will see a view that allows you to view existing applications in the current Azure AD tenant as shown in figure 3.3.

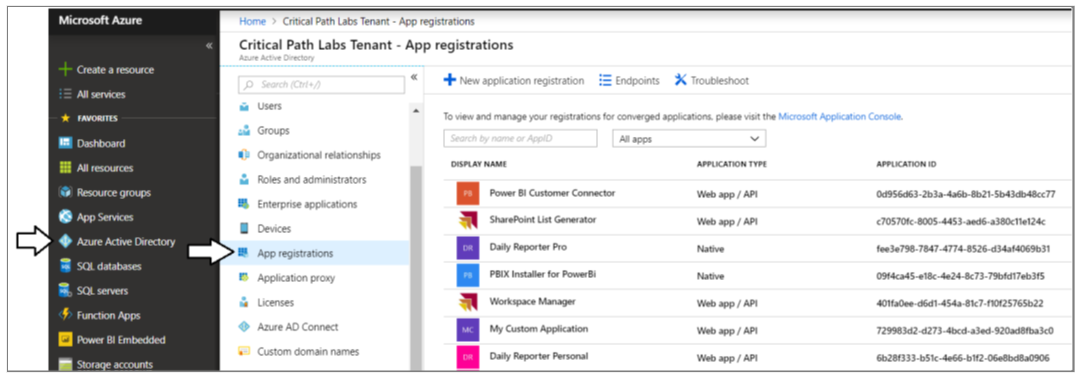


Figure 3.3: The Azure portal makes it possible to view and manage

The *App registrations* view in the Azure portal shown in Figure 3.3 provides a **New application registration** button which you can click to navigate to the **Create** blade where you can create new a new Azure AD application by hand. The screenshot shown in Figure 3.4 demonstrates entering the data required to create a new application. Note that when creating a new application, you must select an Application Type of either *Native* or *Web app / API*. As you will learn, some development scenarios call for you to create a Native application while other call for you to register your application as a Web app / API.

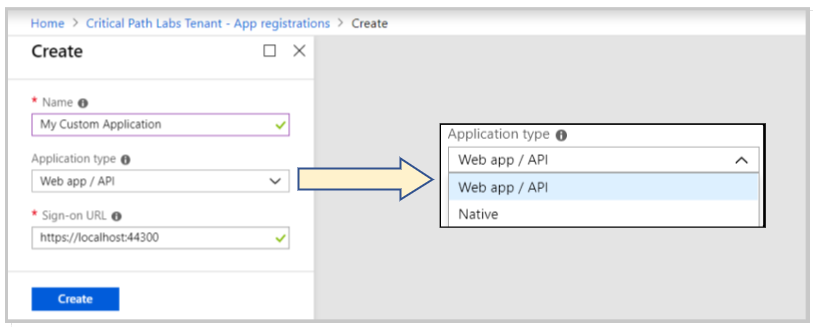


Figure 3.4: When creating an Azure AD application, you must select an Application Type of Native or Web app / API.

When you click the **Create** button to create a new application, Azure AD will generate a new GUID to serve as the Application ID as shown in Figure 3.4. The Application ID is similar to the user principle name for a Azure AD user account in that it identifies a security principal that can be authenticated. Your application will be required to use its Application ID to identify itself when it authenticates with Azure AD.

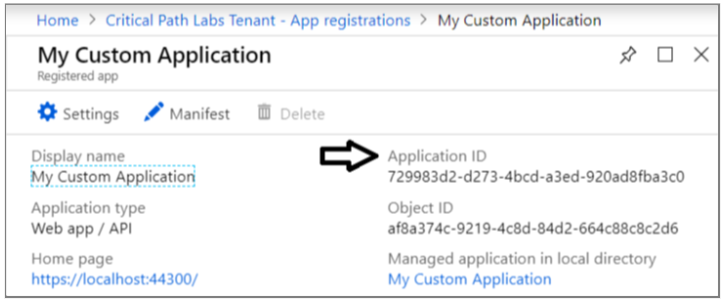


Figure 3.5: Whenever you create a new application, Azure AD will assign it a unique Application ID.

There is potential confusion surrounding the name of the ID for an Azure AD application. In the Azure portal, the ID is referred to as an **Application ID**. However, many other developers and libraries refer it as a **Client ID** instead. It might lead you to the questions "what's the difference between and Application ID and a Client ID?". The answer is there is no difference. They're just two different names for the same thing. The world would be a much better place if everyone agreed to just use one of these, but it's too late for that.

Once you have initially created a new Azure AD application in the Azure portal, there is usually additional configuration that needs to be completed before you can actually use the application. If you click the **Setting** button as shown in Figure 3.6, you can navigate to the blades that make it possible to configure a new Azure AD application with reply URLs, permissions and security keys which act as credentials. You will see several examples of this throughout this section as you begin to implement each of the four authentication flows.



Figure 3.6: In the Azure portal, you can configure application settings such as Reply URLs, Permissions and Keys.

## Configuring and Granting Permissions

An essential aspect of creating the Azure AD application for a custom application is configuring the required permissions. Configuring an Azure AD application with required permissions is what makes it possible for your application to call Azure AD-secured services such as the Power BI Service API and the Microsoft Graph API.

Each resource that is secured by Azure AD defines its own custom set of permissions. This means that the set of permissions for the Power BI Service API will be different from the set of permissions for the Microsoft Graph API. For example, the Power BI Service API defines permissions such as *Dashboard.Read.All*, *Report.ReadWrite.All* and *Content.Create*. The Microsoft Graph API defines different permissions such as *Calendars.Read*, *Contacts.ReadWrite* and *Files.ReadWrite*.

Let's say you want to configure permissions for your new Azure AD application by in the Azure portal. Figure 3.7 shows the *Add API access* blade and the *Enable Access* blade in the Azure portal which make it possible to configure Power BI Service permissions for you application. As you can see in the *Enable Access* blade, there are two different types of permissions which include delegated permissions and application permissions. It's important that you understand the differences between these two different types of permissions.

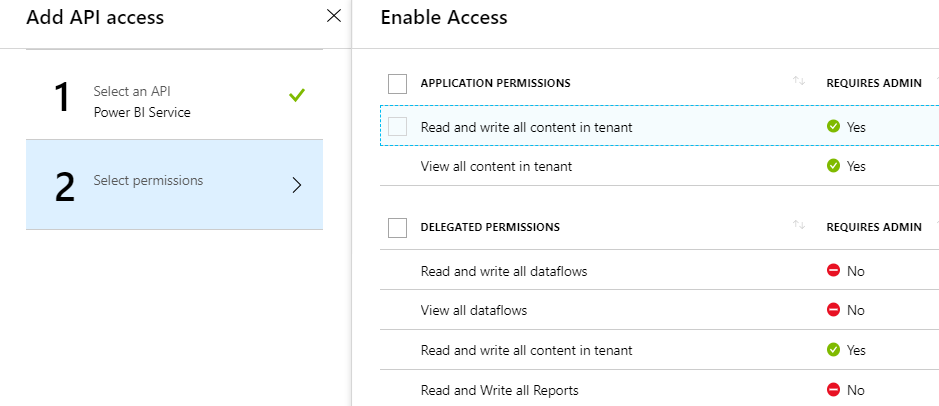


Figure 3.7: The Power BI Service API provides application permissions and delegated permissions.

*Delegated permissions* are used to call into an API with delegated access on behalf of a specific user. Delegated permissions are based on the principle that users can grant an application a subset of their own permissions. Delegated permissions are more restrictive than application permissions because they can never grant a level of permissions greater than the permissions of the current user.

Application permissions are used when your application makes calls to a resource with an app-only identity. An important observation is that application permissions can be far more powerful than delegated permissions. That's because application permissions are never restricted by the permissions of any particular user. Let’s look at an example of delegated permissions and application permissions using the Power BI Service API.

The Power BI Service API provides a delegated permission named *View all reports*. If your application is granted that permission, you can access all the reports that the current user is allowed to view. However, you will not be able to access any report to which the current user does not have access.

Now let's compare this delegated permission to an application permission. The Power BI Service API provides an application permission named *View all content in tenant*. Obviously, this application permission is far more powerful because it allows your application to access any Power BI content in all workspaces across the current Azure AD tenant.

Remember that the type of authentication flow you choose to implement determines the type of permissions you can use. In order to take advantage of application permissions, you must authenticate the application with no user identity using the Client Credentials flow which will generate an app-only access token. The other three types of authentication flows will generate access tokens that contain a user identity in addition to the app identity. When an access token contains the user identity, your code will always rely on delegated permissions instead of application permissions.

As you can see from Figure 3.7, some delegated permissions have their *REQUIRES ADMIN* property set to true. This means that a user requires Power BI administrative permissions in order to use those permissions. It also means that a user requires Power BI administrative permissions just to log into the application. Therefore, it is important to use permissions that require administrative permissions sparingly because they prevent any user with administrative permissions from using the application at all.

An important aspect of using delegated permissions has to do with obtaining user consent. The central idea is that a user needs to approve or grant permissions to an application before that application can make calls on behalf of that user. The act of the user consenting to the permissions required by your application is what actually grants the delegated permissions to your application.

Consider a simple example that illustrates how user consent works. Imagine you are developing a custom application using first party embedding application where users must authenticate using their Azure AD user accounts. Azure AD provides a **common consent framework** which provides built-in interactive behavior when a user logs into an application with delegated permissions for the first time. After each user successfully authenticates for the first time, Azure AD will prompt the user with an interactive *Permissions requested* dialog like the one shown in Figure 3.8.

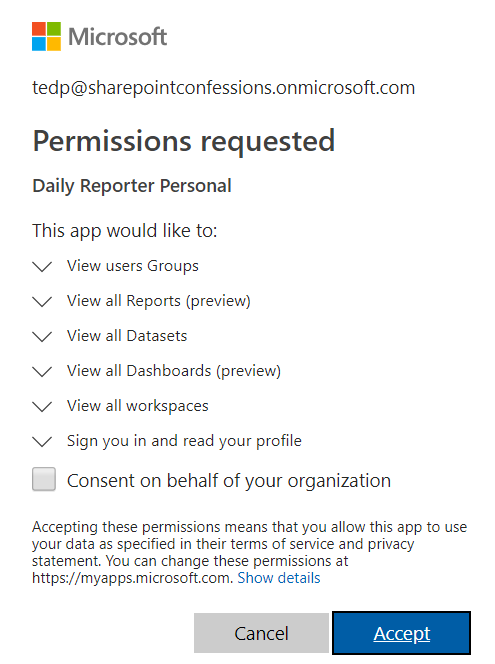


Figure 3.8: Azure AD provides a Common Consent framework which allows user to granted delegated permissions.

The *Permissions requested* dialog shown in Figure 3.8 lists all the delegated permissions required by the application. If the user clicks the *Accept* button, it will effectively grant all delegated permissions to your application for the current user. Once a user clicks the *Accept* button, Azure AD remembers that this user has consented and it does not need to interact with the user in future authentication requests. Azure AD is able to track which users have already consented and which users still need to provide their consent when they first log into the application.

The *Permissions requested* dialog shown in Figure 3.8 displays a checkbox with the caption *Consent on behalf of your organization*. This option is made available to administrators who have the ability to consent for all users in the organization at once making it unnecessary for individuals users to go through the consent process themselves. The *Required permissions* blade in the Azure portal as sown in Figure 3.9 provides the *Grant permissions* button which accomplishes the same goal. When you click the *Grant permissions* button, it automatically grants all delegated permissions to your application for all users at once.

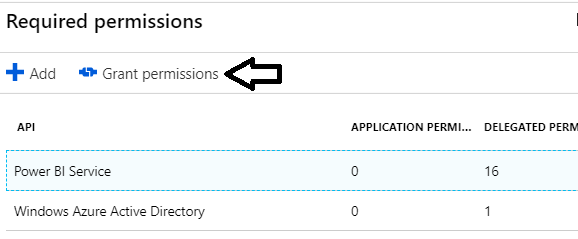


Figure 3.9: You can click the Grant permissions button in the Azure portal to consent for all users at once.

The one last thing to note about user consent with the Azure AD v1.0 endpoint is that it does not support dynamically updating the permissions list over time. Instead, when a user consents to the required permissions, the permission list for that user is created as a static list that cannot be updated.

Consider a scenario in which you have configured an Azure AD application with four requested permissions and you have also deployed the application into production. Each user that logs into the application and consents to the requested permissions will have a permissions list with these four granted permissions.

Now imagine the business requirements for the application change forcing you to add two more required permission to the application. While you can update the Azure AD application by adding these two new permissions, there is no elegant way to propagate these new permissions into the existing permission lists for user who have already consented. The only way to accomplish this goal is to deleted all the granted permissions and to go through the consent process for all users with a fresh start.

## Creating Azure AD Applications using PowerShell

While you can create and configure Azure AD application by hand in the Azure portal, that can become tedious and error prone. As a developer, you should become familiar with PowerShell scripting with the [Azure AD PowerShell module](https://docs.microsoft.com/en-us/powershell/azure/active-directory/install-adv2?view=azureadps-2.0). This module provides administrative cmdlets that allow you to create and configure Azure AD applications. If you are running on Windows 10 you can install this PowerShell module by running the following PowerShell command from an administrative command prompt.

Install-Module AzureAD

Once you have installed the AzureAD PowerShell module, you can login interactively and create a session by executing the cmdlet named *Connect-AzureAD*.

Connect-AzureAD

If you call *Connect-AzureAD* without passing any parameters, you will be prompted with a browser-based dialog to login using your organizational user account and password. Once you have logged in, you can execute other cmdlets in the AzureAD module to create, configure and view Azure AD applications.

Calling *Connect-AzureAD* without passing any parameters creates an interactive login experience which is great when you do not want to hardcode credentials in a PowerShell script. However, having to enter a user name and password can be tedious when you are constantly running a PowerShell script during the authoring and testing phase. If the situation calls for it, you can also hardcode a user name and password into your PowerShell script to avoid having to enter credentials when you are running your scripts.

$userName = "myuseraccount@myorg.onMicrosoft.com"

$password = "pass@word1"

$securePassword = ConvertTo-SecureString –String $password –AsPlainText -Force

$credential = New-Object –TypeName System.Management.Automation.PSCredential `

–ArgumentList $userName, $securePassword

Connect-AzureAD -Credential $credential

You can create a new Azure AD application with PowerShell by executing the cmdlet named *New-AzureADApplication*. Here is a simple example of calling *New-AzureADApplication* with a minimal set of parameters to create a new Azure AD application as a native application.

New-AzureADApplication `

-DisplayName "My First Native App" `

-PublicClient $true `

-AvailableToOtherTenants $false `

-ReplyUrls @("https://localhost/app1234")

There are many different parameters you can pass when calling *New-AzureADApplication*. The example you just saw involved passing a minimal set of four parameters named *DisplayName*, *PublicClient*, *AvailableToOtherTenants* and *ReplyUrls*. Depending on the type of authentication flow you are implementing, you usually need to pass other parameters as well.

The *DisplayName* parameter is used to provide the text for the application's friendly name. The *PublicClient* parameter is used to indicate whether you want to create the application as a Native client versus a Web app / API. You can create a Native client by passing a value of *$true* for the *PublicClient* parameter. You pass a value of *$false* to create a new Azure AD application as a Web app / API.

The *AvailableToOtherTenants* parameter is used to indicate whether you are creating a single-tenant application or a multitenant application. If you pass a value of $false to the *AvailableToOtherTenants* parameter, you will create a single-tenant application that is only accessible to users in the same tenant where the application was created. The use of single-tenant applications is common in enterprise development scenarios where the application only supports users inside a single organization.

If you pass a value of $true to the *AvailableToOtherTenants* parameter, you will create a multitenant application that is accessible to users in other Azure AD tenants. The use of multitenant applications is common among ISVs because they can create a single application that can be used across multiple customers that all have their own Azure AD tenants.

Keep in mind that working with multitenant applications introduces complexity into the way you configure Azure AD applications as well as the way you write the code to authenticate users. Therefore, you should always work with single-tenant applications unless you really need multitenant support.

The *New-AzureADApplication* cmdlet accepts a *ReplyUrls* parameter which allows you to configure a new Azure AD application with one or more reply URLs. When you create a new Azure AD application as a *Web app / API*, you must provide a reply URL that tells Azure AD where your application is running on the Internet. For example, the reply URL for a production application could be *https://myAzureWebApp.azurewebsites.net*. The reply URL for an application you are currently testing and debugging in Visual Studio could be *https://localhost:44300*. Remember that you are not restricted to one reply URL. You can configure an application with more than one reply URL in scenarios where it makes sense.

In the case of a native application, your application might require a reply URL. However, the reply URL for a native application does not have to be a real endpoint on the Internet. Instead, the reply URL for a native application just needs to be a string value formatted as a URI such as *https://localhost/app1234*.

When a native application authenticates using an interactive login, it must pass a reply URL to Azure AD that matches one of the reply URLs that have been configured for the application. Azure AD will return an access denied error if you pass a reply URL that does not match one of the reply URLs that has been registered with the application. Azure AD is also notoriously strict about returning access denied errors in cases where the reply URL matching fails due to case sensitivity or a missing backslash.

## Understanding Service Principals in Azure AD

When you begin to create and work with Azure AD applications, it's important to understand the relationship between the Azure AD application object and another important Azure AD object known as the *service principal*. The service principal object acts as the identity for your application within a specific tenant. This begs the question "why can't the application object be used provide an identity for the application?" To answer this question requires a bit of background information.

Remember that a multitenant application is accessible to users across Azure AD tenants. In other words, a multitenant application can execute within the context of many different tenants. However, an Azure AD application requires a separate identity for each tenant in which it runs. The first time a multitenant application runs in the context of a new tenant, Azure AD automatically creates a new service principle object. While all tenants identify the application itself using a single application ID, each tenant gets its own service principle with a unique object ID. The service principal object has the responsibility of tracking user consent and which delegated permissions have been granted to the application.

While the additional complexity of service principals was added to Azure AD to manage multitenant applications, it is something you still have to deal with when working with a single-tenant application. If you create an Azure AD application in PowerShell without creating a service principal, Azure AD will create the service principal on demand the first time the application is accessed by a user. However, it's a good practice to explicitly create the local service principal after an Azure AD application in your PowerShell scripts.

Note that you cannot pass an application ID when creating a new Azure AD application. Instead, Azure AD will always generate a new GUID for the application ID. When you call the *New-AzureADApplication* cmdlet, it returns an object that represents the new Azure AD application. This application object provides many properties including an *AppId* property which you can read to determine the application ID for a new Azure AD application that you have just created.

After creating an new application with *New-AzureADApplication*, you can create the application's service principal by calling *New-AzureADServicePrincipal*. When you call *New-AzureADServicePrincipal*, you must pass the application ID as shown in the following PowerShell script.

# log in user and capture authentication result

$authResult = Connect-AzureAD

# get more info about the logged in user

$user = Get-AzureADUser -ObjectId $authResult.Account.Id

# create Azure AD Application

$aadApplication = New-AzureADApplication `

-DisplayName "My First Native App" `

-PublicClient $true `

-AvailableToOtherTenants $false `

-ReplyUrls @("https://localhost/app1234")

# create service principal for application

$appId = $aadApplication.AppId

$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId

# assign current user as application owner

Add-AzureADApplicationOwner -ObjectId $aadApplication.ObjectId -RefObjectId $user.ObjectId

Note that the PowerShell script also performs one other common task. It assigned ownership of the application to the logged on user. When you create a new Azure AD application in the Azure portal or by using PowerShell, Azure AD will not assign a default owner. Instead, you must explicitly assign yourself or other users as the application owner.

Once you have created the service principal, you can begin to configure the application's requested permissions. To accomplish this, you can create a *RequiredResourceAccess* object and then set its *ResourceAppId* property to the application ID for the service you want to access. If your application requires permissions to call the Power BI Service API, you can pass its well-known application ID which is *00000009-0000-0000-c000-000000000000*.

$requiredAccess = New-Object -TypeName "Microsoft.Open.AzureAD.Model.RequiredResourceAccess"

$requiredAccess.ResourceAppId = "00000009-0000-0000-c000-000000000000"

The way in which you add a specific permission is by creating a *ResourceAccess* object which needs to be initialized with the GUID that identifies the specific permissions and the Scope parameters which indicated that the permission is a delegated permission as opposed to an application permission. Here is a simple example of creating a *ResourceAccess* object for the *Report.Read.All* permission which is a delegated permission of the Power BI Service API.

$permission1 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `

-ArgumentList "4ae1bf56-f562-4747-b7bc-2fa0874ed46f","Scope"

Once you have created the *ResourceAccess* objects for all the delegated permissions you need, you can assign them to the *RequiredResourceAccess* object and then assign the *RequiredResourceAccess* object to target application using the following PowerShell code.

# configure delegated permisssions for the Power BI Service API

$requiredAccess = New-Object -TypeName "Microsoft.Open.AzureAD.Model.RequiredResourceAccess"

$requiredAccess.ResourceAppId = "00000009-0000-0000-c000-000000000000"

# create first delegated permission - Report.Read.All

$permission1 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `

-ArgumentList "4ae1bf56-f562-4747-b7bc-2fa0874ed46f","Scope"

# create second delegated permission - Dashboards.Read.All

$permission2 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `

-ArgumentList "2448370f-f988-42cd-909c-6528efd67c1a","Scope"

# add permissions to ResourceAccess list

$requiredAccess.ResourceAccess = $permission1, $permission2

# add permissions by updating application with RequiredResourceAccess object

Set-AzureADApplication -ObjectId $aadApplication.ObjectId -RequiredResourceAccess $requiredAccess

As you look at the proceeding PowerShell script, you first thought it likely "so where do I find all these GUIDs that I use to identify specific delegated permissions?" You can answer this question by writing a simple PowerShell script that enumerates through the *Oauth2Permissions* collection property of the service principal object. Here is a simple example.

Connect-AzureAD

$powerBiServiceAppId = "00000009-0000-0000-c000-000000000000"

$powerBiService = Get-AzureADServicePrincipal | Where-Object {$\_.AppId -eq $powerBiServiceAppId}

$powerBiService.Oauth2Permissions | Sort-Object Type, Value | Format-Table Type, Value, Id

When you run this PowerShell script, its output is shown as a table in the following listing. You can use this table to look up the IDs for any of the delegated permissions from the Power BI Service API you need when configuring an application.

Type Value Id

---- ----- --

Admin Tenant.Read.All 01944dba-21df-426f-bb8c-796488be96ad

Admin Tenant.ReadWrite.All d594897b-76e7-4b2b-984b-b4adff35e109

User Capacity.Read.All 76e2ebd5-0dfb-4a5b-93c7-ed89e0362834

User Capacity.ReadWrite.All 4eabc3d1-b762-40ff-9da5-0e18fdf11230

User Content.Create f3076109-ca66-412a-be10-d4ee1be95d47

User Dashboard.Read.All 2448370f-f988-42cd-909c-6528efd67c1a

User Dashboard.ReadWrite.All b271f05e-8329-4b97-baa4-91cf15b99cf1

User Data.Alter\_Any ecc85717-98b0-4465-af6d-1cbba6f9c961

User Datapool.Read.All f9759906-80a4-4f4a-b010-24b832bc6a30

User Datapool.ReadWrite.All ddd37690-e119-40c5-a821-3746ea6125c4

User Dataset.Read.All 7f33e027-4039-419b-938e-2f8ca153e68e

User Dataset.ReadWrite.All 322b68b2-0804-416e-86a5-d772c567b6e6

User Group.Read a65a6bd9-0978-46d6-a261-36b3e6fdd32e

User Group.Read.All 47df08d3-85e6-4bd3-8c77-680fbe28162e

User Metadata.View\_Any ecf4e395-4315-4efa-ba57-a253fe0438b4

User Report.Read.All 4ae1bf56-f562-4747-b7bc-2fa0874ed46f

User Report.ReadWrite.All 7504609f-c495-4c64-8542-686125a5a36f

User Workspace.Read.All b2f1b2fa-f35c-407c-979c-a858a808ba85

User Workspace.ReadWrite.All 445002fb-a6f2-4dc1-a81e-4254a111cd29

## Active Directory Authentication Library

It's possible to implement an authentication flow without any assistance from an external library. After all, an authentication flow is just a standardized sequence of HTTP requests sent between your application and Azure AD. As long as your programming language and development platform support sending HTTP requests and handling HTTP responses, you can write all the code that's required to acquire access tokens from Azure AD. But just because you can doesn't mean you should.

Microsoft provides the *Azure Active Directory Library (ADAL)* to assist developers meet the requirements of implementing authentication flows for the Azure AD v1.0 endpoint. There is one version of ADAL for .NET developers (*ADAL.NET*) which can be used to implement authentication flows with managed languages such as C#. There is a second version of ADAL for JavaScript (ADAL.JS) used to implement implicit authentication flows in single page applications (SPAs) created with JavaScript frameworks such as React.js and AngularJS.

ADAL adds value to the development process by abstracting away many of the low-level details required to implement an authentication flow with Azure AD. When you're programming with ADAL, you don't have to worry about sending HTTP requests to Azure AD or parsing the HTTP response to extract the access code. ADAL does that for you. If this isn't enough for you, ADAL provide even more value in certain scenarios by caching access tokens and refresh tokens.

To add ADAL.NET to a Visual Studio project, install the NuGet package *Microsoft.IdentityModel.Clients.ActiveDirectory*. This NuGet package adds the ADAL.NET library to assist you implement authentication flows and to acquire access tokens. There is also a GitHub repository which contains the source code for ADAL.NET along with a few other valuable developer resources which is accessible through the following URL.

<https://github.com/AzureAD/azure-activedirectory-library-for-dotnet>

To use ADAL.JS you simply need to include a script link to main JavaScript library file named *adal.js*. The adal.js library can be added to a Visual Studio project using a NuGet package. The ADAL.JS library can also be added to a Node.js project using a standard Node.js package named *powerbi-client*. There is also a GitHub repository with the source code, distribution files and documentation for ADAL.JS at the following URL.

<https://github.com/AzureAD/azure-activedirectory-library-for-js>

There is one aspect of using ADAL.JS that can be confusing and frustrating if you are not using the AngularJS framework. The problem is that the standard *adal.js* library has been packaged together with a complimentary library named *adal.angular.js* which contains a custom AngularJS service . The *adal.angular.js* library is great addition which adds a lot of value when you are developing client-side applications with AngularJS. But if you creating a SPA using React.js, it's confusing because Microsoft does not provide a package that will allow you to add *adal.js* to your project without also adding the unneeded library named *adal.angular.js*.

Now that you have learned about Azure AD, creating application with PowerShell and ADAL, it's finally time to start writing some code and learning how to implement Azure AD authentication flows. Over the next few pages, you will learn to implement each of the four types of authentication flows including user password credential flow, authorization code flow, Implicit flow and client credentials flow.

## Calling the Power BI Service API from a Native Application

Let's begin by creating a new native application and going through all the steps required to call the Power BI Service API. A native application is a good starting point when you are first learning how to authenticate with Azure AD and to acquire access tokens. In this example, we will be creating a C# console application in Visual Studio which calls into the Power BI Service API. The first step is to create an new Azure AD application and configure it with the delegated permissions you need to call the Power BI Service API. You can accomplish this step by running the following PowerShell script.

# connect to Azure AD

$authResult = Connect-AzureAD

# get more info about the logged in user

$user = Get-AzureADUser -ObjectId $authResult.Account.Id

# create Azure AD Application

$aadApplication = New-AzureADApplication `

-DisplayName "My First Native App" `

-PublicClient $true `

-AvailableToOtherTenants $false `

-ReplyUrls @("https://localhost/app1234")

# create service principal for application

$appId = $aadApplication.AppId

$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId

# assign current user as application owner

Add-AzureADApplicationOwner -ObjectId $aadApplication.ObjectId -RefObjectId $user.ObjectId

# configure delegated permisssions for the Power BI Service API

$requiredAccess = New-Object -TypeName "Microsoft.Open.AzureAD.Model.RequiredResourceAccess"

$requiredAccess.ResourceAppId = "00000009-0000-0000-c000-000000000000"

# create first delegated permission - Report.Read.All

$permission1 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `

-ArgumentList "4ae1bf56-f562-4747-b7bc-2fa0874ed46f","Scope"

# create second delegated permission - Dashboards.Read.All

$permission2 = New-Object -TypeName "Microsoft.Open.AzureAD.Model.ResourceAccess" `

-ArgumentList "2448370f-f988-42cd-909c-6528efd67c1a","Scope"

# add permissions to ResourceAccess list

$requiredAccess.ResourceAccess = $permission1, $permission2

# add permissions by updating application with RequiredResourceAccess object

Set-AzureADApplication -ObjectId $aadApplication.ObjectId -RequiredResourceAccess $requiredAccess

When you run this script, it will create a new native application in Azure AD with a new GUID for the application ID and a reply URL of *https://localhost/app1234*. Once you have create the Azure AD application, the next step to create a new C# Console application in Visual Studio. Once you have created the new project for the C# console application, you should install the NuGet package for ADAL.NET named Microsoft.IdentityModel.Clients.ActiveDirectory as shown in Figure 3.10. Note that there is a second NuGet package named *Newtonsoft.Json* which will be used convert JSON returned the Power BI Service API into strongly-typed .NET objects making it easier to access content from code written in C#.

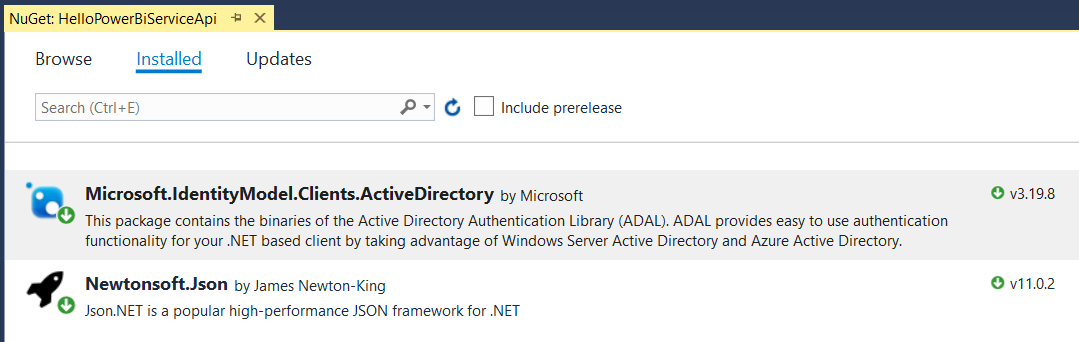


Figure 3.10: ADAL.NET is added by installing the NuGet package named Microsoft.IdentityModel.Clients.ActiveDirectory.

Now it's time to write a complete C# program that calls the Power BI Service API. First, the program must implement an authentication flow to call to Azure AD and obtain an access token for the Power BI Service API. Second, the application must transmit the access token along with any HTTP request sent to the Power BI Service API. Let's start by examining the entire program at once and after that we'll walk through smaller sections of this code.

using System;

using System.Net;

using System.Net.Http;

using Microsoft.IdentityModel.Clients.ActiveDirectory;

using Newtonsoft.Json;

using HelloPowerBiServiceApi.Models;

namespace HelloPowerBiServiceApi {

class Program {

const string aadAuthorizationEndpoint = "https://login.windows.net/common/oauth2/authorize";

const string resourceUriPowerBi = "https://analysis.windows.net/powerbi/api";

const string clientId = "c07dc205-fe75-45dd-93d8-48aa30cac269";

static readonly Uri redirectUri = new Uri("https://localhost/app1234");

static string GetAccessToken() {

var authContext = new AuthenticationContext(aadAuthorizationEndpoint);

var promptBehavior = new PlatformParameters(PromptBehavior.Auto);

AuthenticationResult result =

authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, redirectUri, promptBehavior).Result;

return result.AccessToken;

}

static string ExecuteGetRequest(string restUrl) {

HttpClient client = new HttpClient();

HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, restUrl);

request.Headers.Add("Authorization", "Bearer " + GetAccessToken());

request.Headers.Add("Accept", "application/json;odata.metadata=minimal");

HttpResponseMessage response = client.SendAsync(request).Result;

if (response.StatusCode != HttpStatusCode.OK) {

throw new ApplicationException("Error occured calling the Power BI Servide API");

}

return response.Content.ReadAsStringAsync().Result;

}

static void Main() {

var json = ExecuteGetRequest("https://api.powerbi.com/v1.0/myorg/reports/");

ReportCollection reports = JsonConvert.DeserializeObject<ReportCollection>(json);

foreach (Report report in reports.value) {

Console.WriteLine(report.name);

}

}

}

}

Let's begin with the constant values in the application. First, there is the constant named *aadAuthorizationEndpoint* which tracks the URL which is used to begin the authentication flow.

const string aadAuthorizationEndpoint = "https://login.windows.net/common/oauth2/authorize";

When you use ADAL to implement an authentication flow, the user will be redirected to the authorization endpoint in the browser which will begin an interactive login experience. Note that this example uses the common authorization endpoint.

https://login.windows.net/common/oauth2/authorize

Using the common authorization endpoint is required when you are developing multitenant applications. It's also convenient when working single-tenant applications. For scenarios where you are developing an application that will only authenticate with one specific Azure AD tenant, it is also possible to configure the authorization endpoint using the tenant id.

https://login.microsoftonline.com/37bf5ca4-68cb-4f6a-b915-efd9d1dcb35a/oauth2/authorize

Next, there is a constant named *resourceUriPowerBi* which is the resource identifier for the Power BI Service API. This value must be passed to Azure AD in order to create access tokens which allow you to call the Power BI Service API.

const string resourceUriPowerBi = "https://analysis.windows.net/powerbi/api";

The application also contains a constant named *clientid* and a read-only variable named *redirectUri*. These values must match the values for the Azure AD application you have created. These values are important because they must be passed to Azure AD during an authentication flow in order to acquire an access token.

const string clientId = "c07dc205-fe75-45dd-93d8-48aa30cac269";

static readonly Uri redirectUri = new Uri("https://localhost/app1234");

Now that you understand the purpose of each program constant, it's time to walk through the code inside the *GetAccessToken* function which uses ADAL to implement a interactive authentication flow. The code begins by creating an *AuthenticationContext* object which is initialized using the authorization endpoint.

var authContext = new AuthenticationContext(aadAuthorizationEndpoint);

Next, the code calls *AcquireTokenAsync* passing the resource identifier for the Power BI Service API, the client id, the redirect URI and a parameter to control the interactive prompt behavior. This is the call that begins the authentication flow.

AuthenticationResult authResult =

authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, redirectUri, promptBehavior).Result;

When you call *AcquireTokenAsync* method in a desktop application such as a C# console application, ADAL is able to provide a browser-based login experience by prompting the user with a dialog containing an embedded instance of Internet Explorer. This interactive experience allows a user to enter login credentials and to consent to delegated permissions just as if the user were logging into a browser-based application. You can create a PlatformParameters object which allows you to control whether the user is always prompted for login credentials or whether the program can use cached credentials from a previous login.

var promptBehavior = new PlatformParameters(PromptBehavior.Auto);

Once the user completes the interactive log in experience, the interactive dialog is dismissed and the return value from the *AcquireTokenAsync* method makes it possible to retrieve the access token. While the *AcquireTokenAsync* method exhibits asynchronous behavior, you can assign the *Result* property from the *AcquireTokenAsync* return value to a *AuthenticationResult* variable to simulate calling a synchronous methods to simplify your code. The *AuthenticationResult* object has several useful properties but the one we are interested in here is the AccessToken property.

static string GetAccessToken() {

var authContext = new AuthenticationContext(aadAuthorizationEndpoint);

var promptBehavior = new PlatformParameters(PromptBehavior.Auto);

AuthenticationResult authResult =

authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, redirectUri, promptBehavior).Result;

return authResult.AccessToken;

}

Now let's move on to the *ExecuteGetRequest* function which use the an *HttpClient* object to execute an HTTP GET operation. The main point to see here is that this function adds the Authorization header to each requests and sets it value to a string that combines the word "Bearer" together with a space and an access token returned from the *GetAccessToken* function. As long as the call to *client.SendAsync* returns a successful HTTP status code, the *ExecuteGetRequest* function returns the content from the HTTP GET operation as a string.

static string ExecuteGetRequest(string restUrl) {

HttpClient client = new HttpClient();

HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, restUrl);

request.Headers.Add("Authorization", "Bearer " + GetAccessToken());

request.Headers.Add("Accept", "application/json;odata.metadata=minimal");

HttpResponseMessage response = client.SendAsync(request).Result;

if (response.StatusCode != HttpStatusCode.OK) {

throw new ApplicationException("Error occured calling the Power BI Servide API");

}

return response.Content.ReadAsStringAsync().Result;

}

When can now move on to the last function named *Main*. When the Main function begins to execute, it calls *ExecuteGetRequest* and passes the REST URL required by the Power BI Service API to retrieve the reports in the current users personal workspace.

string restUrl = "https://api.powerbi.com/v1.0/myorg/reports/";

var json = ExecuteGetRequest(restUrl);

The call to *ExecuteGetRequest* triggers a call to *GetAccessToken* which begins the authentication flow. The user should be prompted to log in by enter a user name and password. If delegated permissions have not yet been granted, the user should also be prompted with the consent dialog. Once the user has completed the interactive login, the call to GetAccessToken returns an access token back to the *ExecuteGetRequest* function which then passes the access token in the *Authorization* header when it calls to the Power BI Service. Figure 3.11 shows an example of using the popular Fiddler to inspect a call to the Power BI Service and see the access token that's being transmitted in the *Authorization* header.

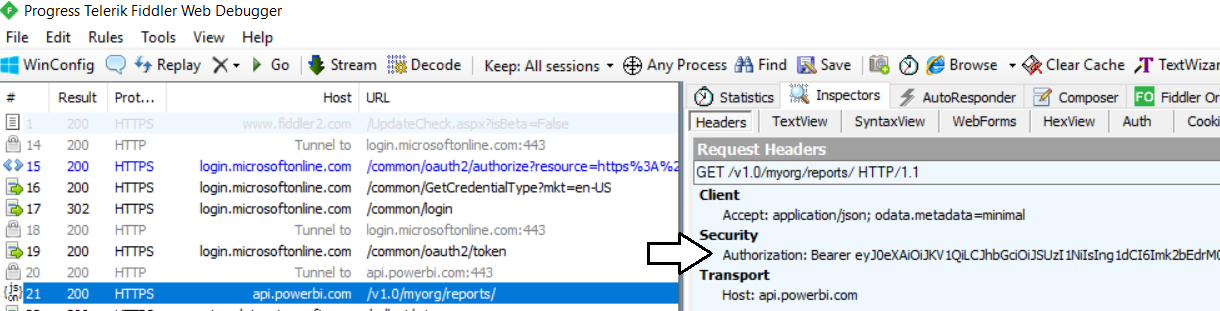


Figure 3.11: All calls to the Power BI Service API must pass an access token in the Authorization header

At this point, the call to *ExecuteGetRequest* returns back to Main with a string value containing the JSON returned from the Power Service API for the reports in the user's personal workspace. This JSON has a standard format that is shown in Figure 3.12.



Figure 3.12: This is an example of the JSON format returned by the Power BI Service API.

The final step now is to convert the JSON string value into strongly-typed objects to simply the code. This can be accomplished by defining the following two classes in C# and use them in a call to *JsonConvert.DeserializeObject*.

public class Report {

public string id { get; set; }

public string name { get; set; }

public string webUrl { get; set; }

public string embedUrl { get; set; }

public bool isOwnedByMe { get; set; }

public string datasetId { get; set; }

}

public class ReportCollection {

public List<Report> value { get; set; }

}

The call to *JsonConvert.DeserializeObject* returns a value object that holds a collection of Report objects that can be enumerated using a C# foreach loop. This makes it possible to display the names of each report to the console window.

static void Main() {

var json = ExecuteGetRequest("https://api.powerbi.com/v1.0/myorg/reports/");

ReportCollection reports = JsonConvert.DeserializeObject<ReportCollection>(json);

foreach (Report report in reports.value) {

Console.WriteLine(report.name);

}

}

You have just seen a complete C# program that acquires an access token from Azure AD and uses it to call the Power BI Service API to retrieve data about the reports in the user's personal workspace. Now that you have seen a complete walkthrough of the code, let's discuss what is really going on behind the scenes. ADAL provided an implementation of the authorization code flow using the following steps.

1. ADAL open a browser and redirected the user to authorization endpoint begin the authorization code flow.
2. Azure AD prompts the user with a standard login page to enter login credentials.
3. The users completes the login process by entering credentials and consenting to delegated permissions.
4. Azure AD returns an authorization code back to your application.
5. ADAL call to the Azure AD token endpoint passing the authorization code to obtain an access token.

While this is an example of the authorization code flow, it's a specializes version for a native client. Later in the section *Programming the Authorization Code Flow in a Web App*, you will learn about the authorization code flow in greater detail. But for now you can observe how much work ADAL does for you behind the scenes. All you were required to do was to call *AcquireTokenAsync*. ADAL does all the work implement an authentication flow that involves interactive behavior and several roundtrips between your application and Azure AD.

## Programming User Password Credential Flow in a Native Application

In the previous example, the application leveraged ADAL functionality to provide interactive behavior which prompts the user to login when the application starts. During development, you might find it helpful to hardcode the user name and password into your code so it runs without prompting. You can accomplish by rewriting the *GeAccessToken* method shown earlier. This technique involves creating a *UserPasswordCredential* object that is initialized with an Azure AD user account login and a password. You then pass the *UserPasswordCredential* object when you call *AcquireTokenAsync*.

static string GetAccessToken() {

var authContext = new AuthenticationContext(aadAuthorizationEndpoint);

var userPasswordCredentials = new UserPasswordCredential("user1@myorg.onMicrosoft.com", "pass@word1");

AuthenticationResult authResult =

authContext.AcquireTokenAsync(resourceUriPowerBi, clientId, userPasswordCredentials).Result;

return authResult.AccessToken;

}

When you run the program with this new implementation of *GetAccessToken*, you will find that the program runs without requiring any interaction on the part of the user. You might have also noticed that you are not required to pass a reply URL when you use the user password credential flow. The way the application interacts with Azure AD is also quite different because ADAL implements the user password credential flow by making a single call to the Azure AD token endpoint which involves passing the user name and password across the network. While the user password credential flow is easy to program, it is considered to be the least secure of the Azure AD authentication flows due to passing a password across the network.

Another issue to be aware of when using the user password credential flow is that it does not provide any ability to provide interactive behavior. If you attempt to acquire an access token using the user password credential flow with a user who has not yet consented to the application's delegated permissions, the call to *AcquireAccessToken* will fail. Remember that you can work around this problem by navigating the *Required permissions* blade for the application in the Azure portal and clicking the *Grant permissions* button.

While the user password credential flow is less secure that other authentication flows, you will find that in certain development scenarios you are required to use it. One noteworthy scenario in which developers have been required to authenticate using the user password credential flow has been when developing with Power BI embedding using third-party embedding and the app-own-data model. Let's examine why.

When the Power BI Service API was first introduced, it did not provide support for application permissions or app-only access tokens. Instead, it only support delegated permissions and user-specific access tokens. With these limitations, the Microsoft recommendation for implementing third-party embedding and the app-owns-data model included the following.

1. Create an Azure AD user account in the same tenant to serve as a master user account
2. Assign a Power BI Pro license to the master user account.
3. Configure the master user account as the administrator of any app workspace it needs to access
4. Authenticate the master user account and acquire access tokens using the user password credential flow

Here is the important takeaway. When you develop with third-party embedding using the security model, your application does not access the Power BI Service API under the identity of the current user nor under the identity of the application itself. Instead, your application accesses the Power BI Service API on behalf of the master user account and it relies on delegated permissions which must have already been granted. Plenty of developers have run into the issue where the user password credential flow fail because it cannot provide interactive behavior for the user to consent the required permissions.

There is good news for companies developing with third-party embedding and the app-owns-data model. Microsoft is introducing new support that allows you to call into the Power BI Service API using app-only access tokens. This means your custom application can use the client credentials flow to establish an app-only identity that doesn't involve any user account which provides two big benefits. First, your application can now rely on application permissions instead of delegated permissions. Second, it eliminates the problem of provisioning and licensing a master user account. We will examine authenticating for third-party embedding in depth in the section titled *Programming the Client Credentials Flow in a Web App*.

## Programming the Authorization Code Flow in a Web App

In an earlier section you saw that ADAL can provide an implementation of the authorization code flow in a native client. When you use this flow in a native client by calling *AcquireTokenAsync*, ADAL prompts the user with a dialog with an embedded browser to provide an interactive login experience. However, the ADAL implementation of the authorization code flow in a native client cuts a few corners and does not meet the requirements of the OAuth 2.0 framework and OpenID connect. In order to implement the authorization code flow the right way, you must create an Azure application as a Web App / API instead of as a native client.

The OAuth 2.0 framework differentiates between confidential clients and public clients. A *confidential client* is an application that contain credentials such as a password or certificate file without exposing this sensitive data to a potential attacker. A *public client* is the opposite because it cannot protect sensitive data. A public client is used in scenarios where an application is running on a client device or within a browser where an attacker can see all the data used by the application.

Another import change is that the application must be running at an HTTPS endpoint that is registered as a reply URL. This adds an important security dimension because Azure AD will only return an access token when it sees tha the application is running at an endpoint that is registered as a reply URL. This cuts down the attack surface.

Here is the high-level overview of the authorization code flow.

1. The application redirects the user to the authorization endpoint.to start flow
2. User enter credentials and if required consents to required permissions
3. Azure AD send POST to application with authorization code.
4. Application passes authorization code and application secret to token endpoint to acquire an access token.

Key point

1. The application never sees the user's password.
2. The authentication flow validate both the user identity and the application identity.
3. Access token is acquired in server-to-server call so never passes through browser or client device.

You need more than just ADAL to implement the authorization code flow. If you are developing with ASP.NET MVC, the most common approach is to combine ADAL together with the OWEN framework and a set of OWEN middleware components provide by Microsoft.

What is OWEN? 1 paragraph.

What does OWEN add

1. It know how to redirect to authorization endpoint.
2. It provide listening mechanism to handle POST callback from Azure AD with authorization code.
3. After the end of the authentication process, OWEN middle populates the ASP.NET principal object
4. Allow you to use Authorization attribute

Here are the NuGet packages

1. Microsoft.Owin
2. Microsoft.Owin.Host.SystemWeb
3. Microsoft.Owin.Security
4. Microsoft.Owin.Security.Cookies
5. Microsoft.Owin.Security.OpenIdConnect

More

public partial class Startup {

private static string commonAuthority = " https://login.microsoftonline.com/common/";

private static string clientId = ConfigurationManager.AppSettings["client-id"];

private static string replyUrl = ConfigurationManager.AppSettings["reply-url"];

public void ConfigureAuth(IAppBuilder app) {

app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);

app.UseCookieAuthentication(new CookieAuthenticationOptions());

app.UseOpenIdConnectAuthentication(

new OpenIdConnectAuthenticationOptions {

ClientId = clientId,

Authority = commonAuthority,

TokenValidationParameters = new TokenValidationParameters { ValidateIssuer = false },

PostLogoutRedirectUri = replyUrl,

Notifications = new OpenIdConnectAuthenticationNotifications() {

AuthorizationCodeReceived = (context) => {

// code to authenticate and acquire access token

}

});

}

}

And now you add a controller class named AccountControl.

more

using System.Web;

using System.Web.Mvc;

using Microsoft.Owin.Security.Cookies;

using Microsoft.Owin.Security.OpenIdConnect;

using Microsoft.Owin.Security;

namespace DailyReporterPersonal.Controllers {

public class AccountController : Controller {

public void SignIn() {

if (!Request.IsAuthenticated) {

HttpContext.GetOwinContext().Authentication.Challenge(

new AuthenticationProperties { RedirectUri = "/" },

OpenIdConnectAuthenticationDefaults.AuthenticationType);

}

}

public void SignOut() {

string callbackUrl = Url.Action("SignOutCallback", "Account",

routeValues: null, protocol: Request.Url.Scheme);

HttpContext.GetOwinContext().Authentication.SignOut(

new AuthenticationProperties { RedirectUri = callbackUrl },

OpenIdConnectAuthenticationDefaults.AuthenticationType,

CookieAuthenticationDefaults.AuthenticationType);

}

public ActionResult SignOutCallback() {

if (Request.IsAuthenticated) {

return RedirectToAction("Index", "Home");

}

return View();

}

[Authorize]

public ActionResult UserProfile() {

return View();

}

}

}

When you first acquire an access token suing ADAL, this library provides built-in code which inserts the access token along with a refresh token into a cache. After that, you can call ADAL methods such as AcquireAccessTokenSilent to retrieve an access token from the cache. If there isn't a valid access token in the cache, ADAL will use the refresh token to acquire a new access token from Azure AD. All this work of caching and refreshing expired access tokens takes place behind the scenes and is transparent to your code.

There is good news here. While you now understand what refresh tokens are and how they work, this is something you only need to understand in theory, but not in practice. When you begin to program with the Azure Active Directory Library (ADAL), you will happily discover that this library abstracts away any need for a developer to write any code that directly deals with refresh tokens. In fact, any code that uses ADAL and is working directly with refresh tokens is likely not using the library as it was intended.

The next two authentication flows are used in a Web app to authenticate the user and to establish user identity. *Authorization Code Grant Flow* is more secure because it requires application to provide a client secret during the authentication process just after requiring the user to provide a secret password. *Implicit Grant Flow* is used by client-side Web applications such as single page applications (SPAs) which run entirely within the browser and cannot keep any hidden secrets. The implicit grant flow authentication is a bit less secure because it does not include a client secret and the access token is passed directly back to the client code running in the browser.

When you create an Azure AD application as a Web app, you can configure it with secret credentials to achieve stronger levels of authentication. In most cases, you will also configure an Azure AD application which as a Web app with one or more Reply URLs. Reply URLs add an extra security dimension because Azure AD can verify that the application is running within a pre-configured DNS domain on the Internet. This can really help to decrease the surface area that is exposed to attackers.

Azure AD also makes it possible to create an Azure AD application as a Native app instead of as a Web app. Native apps are used for specific scenarios such as a .NET application running on the laptop computer or an iOS app running on an iPhone. An important aspect of a Native app is that it is considered to be a *public client*. Unlike a web app which can keep track of server-side secrets, native apps cannot keep secrets such as client credentials. Therefore, Native apps can only authenticate with a user name and password. This means that a native app cannot establish application identity nor can it take advantage of application permissions.

So why am I going into all this detail about native apps? As it turns out, it’s important to 3rd party embedding where you must create the Azure AD application for your custom application as a native app. I will explain why this requirement exists later in this post. For now, I just want you to keep in mind that Native app is more restricted and less secure than a web app in several ways.

## Programming the Implicit Flow in a Single Page Application (SPA)

Azure AD maintains an application manifest for every registered applications.



Figure 3.6: The Azure portal makes it possible to view and, if necessary, edit the application manifest.

## Programming the Client Credentials Flow in a Web App

The *Client Credentials Grant Flow* is used in a web app to authenticate the application itself and to establish an application identity which has no associated user identity. This authentication flow is used when an application needs to take advantage of application permissions. However, the *Client Credentials Grant Flow* isn’t relevant to Power BI embedding because the Power BI Service API does not currently support any application permissions.

## Choosing the Best Authentication Flow for Your Scenario

xxxxxxx

## Understanding the Azure AD v2.0 Endpoint

The v1.0 endpoint allows only work and school accounts to sign in to your application (Azure AD). The v2.0 endpoint allows work and school accounts from Azure AD and personal accounts (MSA) (hotmail.com, outlook.com, msn.com) to sign in.

v2.0 supports incremental and dynamic consent.

v2.0 supports also changes the way you work with resource identiters and scopes. In v2.0 that are combined together ans a scope will contain both a resource ID and a permissions

v2.0 does not automatically pass a refresh token and an id token during an authentication flow. Instead, you application must request permissions using well-known scopes. You application must be granted the offline\_access scope in order to receive refresh tokens. You application must be granted the email scope and the profile scope to get the same information about the current user in the ID token.

MSAL .NET

MSAL.js

# Programming the Power BI Service API

## Calling the Power BI Service API using Direct REST Calls

## Calling the Power BI Service API using the .NET Library

## Calling the Power BI Service API using the SharePoint Framework

## Generating Embed Tokens for Third-party Embedding

An Embed Token determines:

* Which resource can be accessed
* Which access level (view, create, edit)
* How long it can be accessed
* The data can be seen by the user

Embed tokens expire when the access token expires

* The app can be developed to silently refresh the embed token when it expires

## Generating Embed Tokens using Row-level Security

# Programming the Power BI JavaScript API

## Embedding Reports

## Interacting with Embedded Reports

## Embedding Dashboard and Tiles

## Embedding the Q&A Experience