# Introduction to the Fabric REST APIs

Microsoft Fabric is a platform that provides the ability to design, implement and deploy data-centric solutions. Fabric makes it possible to build end-to-end solutions by offering services for data movement, processing, ingestion, transformation, real-time event routing, semantic modeling and report building. Fabric services are exposed through a set of **Fabric** **workloads** which include Data Factory, Data Engineering, Data Warehouse, Data Science, Real-Time Analytics, Power BI and Data Activator.

A blue logo with green text

Description automatically generated

Microsoft promotes Fabric as a SaaS platform that simplifies life for users and administrators alike. Fabric has also been designed to offer a full-fledged development platform to professional developers and independent software venders (ISVs). The Fabric REST APIs extend the Fabric platform by giving developers the ability to automate the deployment and management of Fabric solutions in a production environment. Fabric also provides ALM and CI/CD features which leverage the platform’s GIT integration to provide organizations with a robust and reliable way to update solutions after they have been deployed into production.

## Design Solutions based on Workspaces and Workspace Items

As you begin developing solutions for Microsoft Fabric, you should design your solutions in terms of workspaces and workspace items. As you learn more about the different Fabric workloads and the types of workspace items they offer, you will become more experienced in architecting end-to-end solutions.

Let’s start with an example of designing a solution with workspace items from the **Power BI workload**. Using the Fabric REST APIs, you can automate the creation of a Fabric workspace followed by the creation of a semantic model which consumes data using import-mode and a Power BI report connected to that semantic model.

A screenshot of a computer

Description automatically generated

You can leverage workspace items from the **Data Engineering workload** by creating a workspace with a lakehouse and notebooks containing Python code written to ingest data files and to generate a schema of tables inside the lakehouse. As you will learn, the Fabric REST APIs make it possible to automate running notebooks on demand as part of the solution deployment process.

A close-up of a computer

Description automatically generated

You can also automate the creation of a custom Spark environment for executing the code inside notebooks. Creating a custom Spark environment is valuable if you need to load Spark libraries or you need to control the number and size of the nodes in the Spark cluster which processes the execution of code in running notebooks.

Keep in mind you can always mix and match workspace items from multiple Fabric workloads. For example, you can use Data Engineering workspace items to design a solution with a lakehouse and notebooks containing ETL logic used to populate lakehouse tables. Next, you can extend the solution by creating a semantic model in DirectLake mode that consumes data from the lakehouse table schema. Then you can complete the solution by creating one or more Power BI reports that consume data from the DirectLake semantic model.

A close-up of a computer

Description automatically generated

What if you need to migrate a data-centric solution from another data platform that is based on SQL. That’s where the **Data Warehouse workload** comes in by allowing you to design SQL-based solutions using a warehouse which is created as a workspace item. You can then extend a Data Warehouse solution with complementary workspace items such as a lakehouse for staging data and data pipelines to ingest data files and to execute SQL statements against the warehouse’s SQL endpoint to create and populate tables in the warehouse.

A close-up of a logo

Description automatically generated

If you need to design a solution which can handle a large volume of incoming data, you can leverage workspace items offered by the **Real-time Intelligence workload**. You can design a Real-time Intelligence solution by creating an eventhouse with an eventstream configured to write its output into a KQL database. You can further extend the solution by adding KQL querysets and KQL dashboards.

A screenshot of a computer

Description automatically generated

Once you’ve designed a solution using workspace items from the Real-time Intelligence workload, you can further extend it with a Reflex item from the **Data Activator workload**. This makes it possible to create reactionary triggers which fire to initiate some type of action in response to certain conditions being met as data is ingested.

What’s the key takeaway? Workspace items are the fundemtnal building blocks used to create Fabric solutions.

Once you’ve designed a Fabric solution using a composition of workspace items, you can move forward to test out your theory by implementing the solution in a Proof of Concept (PoC). The PoC is important because it allows you to verify that the solution scales to the anticipated volume of data and that the solution does what it’s supposed to.

Over the next few chapters you will learn how to automate the creation of workspaces and workspace items using the Fabric REST APIs. As you will soon learn, workspace items are created using the **Create Item** API together with a special type of creatable item template known as an **item definition**. Learning how to program with item definitions will be one of the most essential skills you need to learn as a Fabric platform developer.

It usually makes sense to implement the PoC by hand at first using Fabric’s browser-based UI experience. You can start by creating a workspace. After that, you can create and configure a set of workspace items from whichever Fabric workloads you need. Once you have implemented the PoC by hand, you can test out your solution to verify that it scales as required and that it behaves the way you expect it to.

Here’s some good news. Once you implement a Fabric solution by hand, you can reverse engineer item definitions from the existing workspace items. You accomplish this calling the **Get Item Definition** API and storing the response as a set of item definition files. This technique will allow you to acquire the resources you need to generate item defintions you can use to call the **Create Item** API and the **Update Item Definition** API. This guidance document will revisit this essential topic in the **Create and Update Workspace Items** chapter.

## Develop Multitenant Applications on the Fabric Platform

**Multitenant application development** is a software architecture that allows an ISV to serve multiple customers using a single instance of an application. In a multitenant architecture, each customer is considered to be a separate **tenant.** You can think of an analogy with a large apartment building where each tenant has their own apartment. A requirement of multitenancy is that each tenant is created in isolation from all other tenants.

If you have worked with Entra Id (formerly Azure AD), the word **"tenant"** might make you think of an Entra Id tenant. However, the concept of a tenant is different when designing a multitenant application for Fabric. In this context, each tenant represents a customer with one or more users. With the proper planning, you can build a multitenant environment with Fabric which scales to 100s or 1000’s of customer tenants scope inside a single Entra Id tenant.

When developing multitenant applications for Fabric, it’s a best practice to create a separate workspace for each customer tenant. By provisioning each customer tenant using a separate workspace, you can provide a base level of isolation. In a more complicated solution design, it might make sense to create multiple workspaces for each customer tenant. However, a design based on a single workspace per customer tenant is a good place to start.

A diagram of a workflow

Description automatically generated

When developing a multitenant application, it’s essential that you learn to fully automate the process of provisioning new customer tenants. This provisioning process typically involves creating a new workspace and then creating and configuring a set of workspace items inside. If parts of the tenant provisioning process require manual intervention, that can limit your ability to scale up to a large number of customer tenants.

A critical aspect of designing multitenant applications has to do with achieving the required level of isolation. The last thing you want is for a user from one customer tenant to see data from another customer tenant. Creating each new customer tenant in its own Fabric workspace provides the first level of isolation.

A second level of isolation can be achieved by using a different service principal to create and populate each customer workspace. A multitenant design which requires a one-to-one mapping between customer tenants and service principals prevents any Entra Id identity from having access to more than just a single tenant.

## Fabric as a PaaS Platform for Developers

Microsoft actively promotes Fabric as a **Software as a Service (SaaS)** platform. However, Fabric can also been seen as a powerful **Platform as a Service (PaaS)** platform from the perspective of an ISV. For example, an ISV can build a multitenant application that leverage Fabric’s PaaS features while at the same time making their own application available to their customer base as a SaaS offering.

A close-up of a square

Description automatically generated

By leveraging Fabric as a PaaS platform, an ISV can extend their reach to users and organizations that do not use Entra Id. An ISV can authenticate the users of their SaaS application using whatever identity provider they’d like. For example, some ISVs may prefer to authenticate their users with an identity provider other than Entra Id such as Okta, Auth0 or OneLogin. Once the ISV application authenticate the user, it can validate the user’s identity and enforce access control policies. However, the ISV application does not propagate the user’s identity to Fabric when it calls the Fabric REST APIs. Instead, the ISV uses a service principal to call the Fabric REST APIs.

## Fabric REST API Architecture

If you’re an experienced software developer, it’s likely that you’ve learned to work with quite a few different APIs before. If you’ve programmed with other Microsoft APIs such as the Microsoft Graph API or the Power BI REST API, you will find that programming with the Fabric REST API will be familiar.

The Fabric REST APIs are designed to be accessible to any developer on any development platform. To this end, Fabric REST API architecture has been designed using the principles of REST and open security standards which include OAuth2 and Open ID Connect. You can develop a custom application that executes API calls by submitting HTTP requests against Fabric REST API endpoints.

A close-up of a computer screen

Description automatically generated

If you are working within the Microsoft public cloud, Fabric REST API endpoints are accessible through the base URL of **https://api.fabric.microsoft.com/v1**. If you are working in a different cloud such as a sovereign cloud or a government cloud, the base URL will be slightly different.

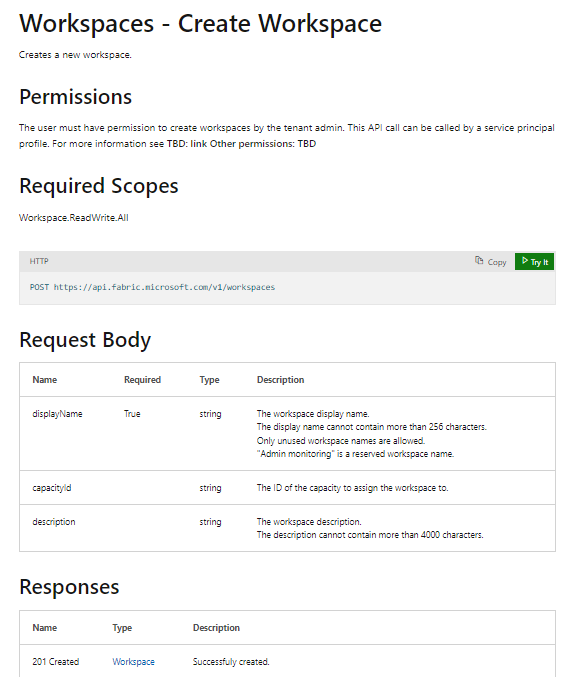
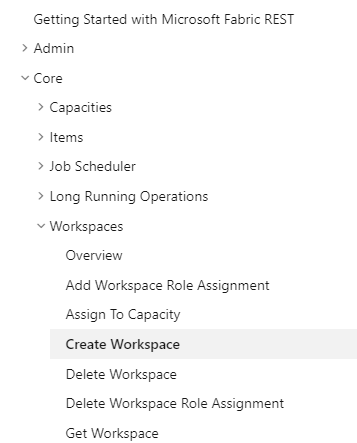
The Fabric REST API is secured by the Entra Id Service. Before calling the Fabric REST APIs, your application must first acquire an access token from the Entra Id Service by executing an authentication flow. After that, your application must transmit the access token each time it calls to the Fabric REST APIs. This is accomplished by passing the access token in the **Authorization** request header.

## Fabric REST API Documentation

If you plan on developing solutions with Fabric, you should become familiar the Fabric REST API documentation. This is an essential developer resource which is available at the following URL.

* <https://learn.microsoft.com/rest/api/fabric>

The Fabric REST API documentation provides essential details for developers such as the requirements for constructing REST URLs and structuring the JSON that goes into a request body. This documentation also tells you what you can expect in the response in terms of HTTP status codes and the structure of JSON in the response body.



## The Fabric ISV Playbook Developer Sample

This guidance document is accompanied by a developer sample project named **FabricIsvPlaybook**. The code in this project demonstrates how to deploy Fabric solutions using the Fabric REST APIs by creating and configuring workspace items. The **FabricIsvPlaybook** project is a simple .NET console application built using .NET 8, C# and the Fabric REST API .NET SDK. Source code for the **FabricIsvPlaybook** project is available in a public GitHub repository at the following URL.

* <https://github.com/PowerBiDevCamp/Fabric-ISV-Playbook>

You can also download all the project files for **FabricIsvPlaybook** project as a single ZIP archive using [**this link**](https://github.com/PowerBiDevCamp/Fabric-ISV-Playbook/archive/refs/heads/main.zip).

Once you have downloaded the **FabricIsvPlaybook** project source code to a local folder, you can open it using Visual Studio 2022. The high-level project structure of the **FabricIsvPlaybook** is shown in the following screenshot.

A screenshot of a computer

Description automatically generated

You can open the **FabricIsvPlaybook** project using any version of Visual Studio 2022 including the community version which is free. If you need to download a version of Visual Studio 2022, you can visit [**this link**](https://visualstudio.microsoft.com/downloads/).

In the root folder of the **FabricIsvPlaybook** project, there are source files named **Program.cs** and **AppSettings.cs**. These are the two primary source files you will update as you experiment with this developer sample. You need to edit the values for configuration settings in **AppSettings.cs** for various demos to work correctly. You will also be commenting and uncommenting lines of code in **Program.cs** to switch between different demos.

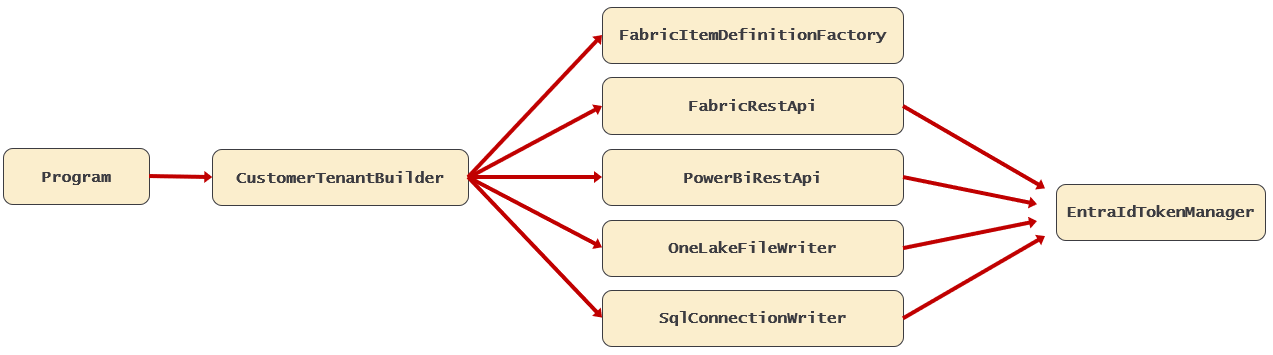
The **FabricIsvPlaybook** project contains a top-level folder name **Services** with the source files for a set of C# classes, each of which has been designed to encapsulate logic to accomplish a specific set of tasks.

A screenshot of a computer

Description automatically generated

The code in the **FabricIsvPlaybook** project has been written as a developer sample to demonstrate deploying Fabric solutions using the Fabric REST APIs. By examining and testing out the code in this project, you will build your understanding of how to deploy solutions by creating workspaces and workspace items.

The follow diagram shows how the classes in the **FabricIsvPlaybook** project interact with one another.



Here is a quick summary of each class and its purpose in the developer sample.

* **EntraIdTokenManager** contains code to authenticate with Entra Id and acquire access tokens.
* **FabricRestAPI** contains code that executes API calls using the Fabric REST API.
* **FabricItemDefinitionFactory** contains code which creates item definitions used to call Create Item.
* **PowerBiRestAPi** contains code that executes API calls using the Power BI REST API.
* **OneLakeFileWriter** contains code using ADLS GEN2 APIs to upload files to OneLake storage in lakehouse.
* **SqlConnectionWriter** contains code to execute SQL statements against SQL endpoint of warehouse.
* **CustomerTenantBuilder** contains top-level logic for deploying several examples of Fabric solutions.
* **Program** contains entry point to application. You command and uncomment code to run different tests.
* **AppLogger** contains code to write formatted output to the console window so you can see what’s going on.

The **FabricIsvPlaybook** project is a developer sample which has been created as a learning resource. The design of this project is based on several assumptions and simplifciations which make things easier to understand. Some of the code in this project needs to be hardened and/optimized before it’s used in a real-world application.

## Learning Path for Fabric Developers

Now that you’ve reached the end of the first chapter, you’ve started your journey to become a productive Fabric developer. Over the next few chapters, you’ll continue on this journey by examining sample code and learning to programming with Fabric REST APIs. However, you can’t really do anything with the Fabric REST APIs until you learn how to authenticate with the Entra Id Service to acquire access tokens. That’s why the second chapter dives into security.

The **Authentication and Authorization** chapter examines security and discusses your options for executing Fabric REST API calls as a service principal versus as a user. The chapter also discusses how using managed identities in Microsoft Azure can provide a more secure approach for executing API calls as a service principal.

The **Fabric REST API Fundamentals** chapter introduces essential concepts and programming patterns used with the Fabric REST APIs. The chapter also introduces the Fabric REST API .NET SDK which boosts developer productivity by hiding tedious and low-level details associated with directly calling Fabric REST API endpoints.

The **Program Capacities and Workspaces** chapter covers using the Fabric REST APIs to create workspaces and to associate workspaces with premium capacities. The chapter also examines how to configure access to workspace content by adding workspace role assignments for users, groups and service principals.

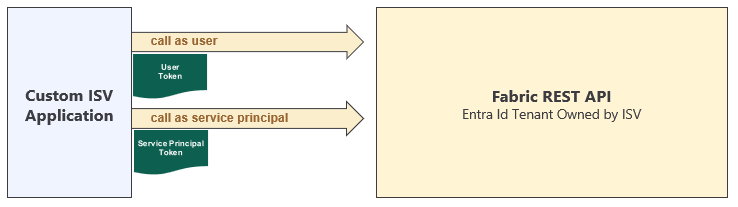
The **Programming Connections** chapter covers how to create connections with the Fabric REST APIs. This is an important topic because you will be required to create connections in order to connect workspace items such as shortcuts, data pipelines and semantic models to external datasources. The chapter will examine creating several different types of connections include Shareable Cloud Connections (SCCs), On-Premises Gateway connections and Virtual Gateway connections.

The **Create and Update Workspace Items** chapter covers creating workspace items using the **Create Item** API. Along the way, you’ll learn about item definitions and how they are used to create and configure workspace items. You will also learn how to update workspace items using the **Update Item Definition** as well as how to reverse engineer the item definition for a workspace item by calling the **Get Item Definition** API.

These first six chapters are the foundational chapters of this guidance document because they introduce a sequence of concepts and topics that continue to build on one another. After you have completed the first six chapters, you should feel free to skip ahead to any of the workload-specific deployment chapters that are most interesting to you.

# Authentication and Authorization

Every call to the Fabric REST APIs executes under the identity of a specific security principal. The Fabric REST APIs support two types of security principal identity which are **user principals** and **service principals**. When you interact with the Entra Id Service to acquire an access token, the way in which you structure your request will determine whether the service returns a user token or a service principal token. The identity of the security principal found inside the access token is what actually determines the identity used to execute an API call.



When calling the Fabric REST APIs, you must decide whether to execute API calls as a user or as a service principal. In general, it is a better choice to execute API calls as a service principal. This is especially true when developing multitenant applications. That’s because there are several common problems when authenticating and executing API calls as a user.

The first problem with user authentication is that you need an actual human being to be present to interact with the Entra Id sign in process. Think about a scenario in which your application needs to complete a task at night with unattended execution when all the users are asleep. In network environments where administrators have enabled multifactor authentication (MFA), your code will often fail when attempting to acquire a user token.

While issues with MFA impacting user authentication are the most common, there are several other issues that can create problems as well. Here are some other things to keep in mind.

* When attempting to acquire a user token without interaction, developers often resort to using an insecure authentication flow where the user name and password are passed across the network.
* A user token can never elevate your code to run with a greater level of permissions than the current user.
* There is always the possibility that a user will leave the organization. This often leads to problems when that user is the owner or lone administrator of workspaces, connections and workspace items.

In a perfect world, you would be able to execute every API call as a service principal. But in the imperfect world in which we live, that is not always the case. To execute API calls as a service principal, you or your organization’s IT department must either create a custom Entra Id application or create a managed identity in Microsoft Azure. Furthermore, you need a user with Fabric administrator permissions who can access the Fabric Admin portal to configure a service principal with access to the Fabric REST APIs. If you cannot meet the requirements to configure a service principal, your only option will be to call Fabric REST API as a user.

There will also be edge cases in which a specific Fabric API endpoint does not support service principals. This is another case where executing API calls as a user is your only option.

As of October 2024, Microsoft’s efforts to add service principal support across all the Fabric REST APIs is still a work in progress. While many endpoints already support execution by a service principal, there are others that do not. Full support in the Fabric REST APIs for executing API calls as a service principal is expected to be completed by the end of 2024.

## Fabric REST API Permission Scopes

When you call the Entra Id service to acquire an access token, the request must include one or more permission scopes. Therefore, you should have a general understanding of how these permission scopes are created. Let’s start with an examination of how permission scopes are created specifically for the Fabric REST APIs.

A permission scope is a string value which begins with a **resource URI**. The role of the resource URI is to identify the target resource or API. In the Microsoft public cloud, the Fabric REST APIs have the following resource URI.

* **https://api.fabric.microsoft.com/**

Note that the resource URI for the Fabric REST API will change slightly when you are developing for Fabric environments in sovereign clouds and government clouds.

To create a Fabric permission scope, you parse the resource URI together with a permission name. Here are a few examples of permission scopes used with the Fabric REST APIs.

* **https://api.fabric.microsoft.com/.default**
* **https://api.fabric.microsoft.com/Workspace.ReadWrite.All**
* **https://api.fabric.microsoft.com/Item.ReadWrite.All**

Now that you’ve learned what permission scopes are, the next step is learning how to put them to use when acquiring access tokens in an authentication flow. The choice of which Fabric REST API permission scopes to use depends on whether you are acquiring an access token for a user or a service principal.

## Authenticate and Acquire Access Tokens for a Service Principal

There are two ways to set up a custom application to execute Fabric REST API calls as a service principal. The first way is to create a custom application in the Entra Id Service. When you do this, the Entra Id service will automatically create a service principal for you. The second way is using a Microsoft Azure subscription to create a managed identity. Creating a managed identity also results in the automatic creation of a service principal.

So what’s the difference between creating a custom application in the Entra Id Service versus creating a managed identity in Microsoft Azure? That’s an important question you should be able to answer. Let’s examine the requirements of each approach to illustrate the benefits and drawbacks.

Let’s start by covering the requirements of using an Entra Id application to authenticate as a service principal. First, you must create custom application in Entra Id service and configured it with either a client secret or a client certificate. The client secret or client certificate must be made available to the application at runtime because it must be passed to the Entra Id Service to authenticate when acquiring an access token. If you meet these requirements, the Entra Id Service will return an access token which includes the service principal object Id.

Now let’s contrast creating an Entra Id application to creating a managed identity. First, you will need an Azure subscription to create a managed identity. The managed identity must also be associated with Azure resource such as a Web App, a Function App or a virtual machine (VM). Once the managed identity has been created and properly configured, it can use used to acquire an access token for a service principal. The most significant benefit is that it eliminates the need for your application deal with a client secret or a client certificate.

Using a custom Entra Id application requires your application to deal with secret credentials, while a managed identity does not. Microsoft recommends using managed identities over custom Entra Id applications in production because it eliminates the need for creating and protecting secret authentication credentials.

### Acquire Access Tokens with a Custom Entra Id Application

You use **Client Credentials Flow** to acquire an access token for a service principal with an Entra Id application. This is the authentication flow which requires you to pass a client secret or a client certificate to authenticate. When using Client Credentials Flow, you should always use the default permission scope of the Fabric REST APIs which is **https://api.fabric.microsoft.com/.default**.

In C#, you can use a string array to create a set of one or more permission scopes.

public static readonly string[] Default = new string[] {

"https://api.fabric.microsoft.com/.default"

};

While you can write code that directly interacts with the Entra Id Service to acquire access tokens, Microsoft recommends instead that you use the Microsoft Authentication Libraries (MSAL). By integrating one of the MSAL libraries into your development projects, it will make your code easier to write. It will also make your code more secure because MSAL libraries are fully tested and frequently updated to deal with emerging security threats.

Let’s look at an example that uses the .NET version of MSAL named **Microsoft.Identity.Client** which can be added to a .NET project as a NuGet package. The **GetAccessTokenForServicePrincipal** method shown in the following code listing implements Client Credentials Flow to acquire an access token for a service principal.

private static string GetAccessTokenForServicePrincipal() {

string clientId = AppSettings.ServicePrincipalAuthClientId;

string clientSecret = AppSettings.servicePrincipalAuthClientSecret;

string tenantId = AppSettings.ServicePrincipalAuthTenantId;

string tenantSpecificAuthority = "https://login.microsoftonline.com/" + tenantId;

string[] scopes = new string[] { "https://api.fabric.microsoft.com/.default" };

var appConfidential = ConfidentialClientApplicationBuilder.Create(clientId)

.WithClientSecret(clientSecret)

.WithAuthority(tenantSpecificAuthority)

.Build();

return appConfidential.AcquireTokenForClient(scopes).ExecuteAsync().Result.AccessToken;

}

The **GetAccessTokenForServicePrincipal** method retrieves application configuration settings for the client Id, the client secret and the tenant Id for an Entra Id application. These three application settings are then passed to the Entra Id service in the call to **AcquireTokenForClient** when acquiring access tokens using Client Credentials Flow.

If you are not developing with C# and.NET, you should be aware that Microsoft provides other versions of MSAL for developers using other programming languages, platforms and frameworks. You can read more about the full scope of MSAL library support at the following URL.

* [**https://learn.microsoft.com/en-us/entra/identity-platform/msal-overview**](https://learn.microsoft.com/en-us/entra/identity-platform/msal-overview)

In the past, the biggest security concern in large organizations has been to ensure that user credentials are never compromised. However, things have changed over the last few years as user authentication has been strengthened through MFA. Microsoft now sees that compromised security credentials for a service principal poses a greater potential threat.

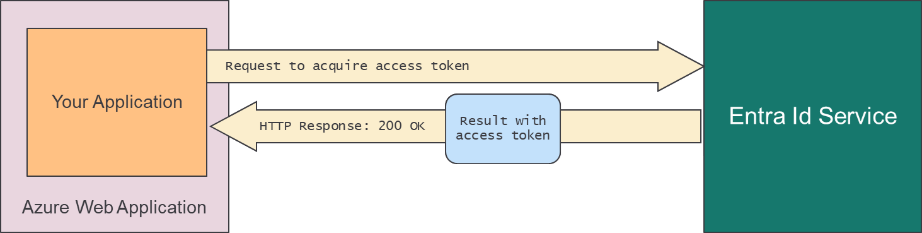
Crafty attackers have been able to discover and steal client secrets in places where a client secret should never be found. The obvious example of this is a forgetful developer who mistakenly published source code with a client secret to a public GitHub repository. The threat of compromised service principal credentials has led many companies to create defensive security policies such as one where source files checked into source control are scanned to ensure the code doesn’t contain any client secrets.

The issue discussed here leads to an important question. Does your organization really want to be responsible for creating and protecting service principal credentials such as client secrets or client certificates? If your answer is NO, then you’ll want to read on into the next section where you will learn how using managed identities can eliminate the requirements of your application to work with client secrets or client certificates.

### Acquire Access Tokens with a Managed Identity

The primary advantage of using a managed identity to execute API calls as a service principal is that your application doesn’t need to deal with a client secret or a client certificate to authenticate. Instead, you leave it up to the Entra Id Service to authenticate a service principal using other means. Let’s step through the fundamentals of how authentication works with a managed identity.

A managed identity is associated with an Azure resource such as a Web app, a Function app or a virtual machine. When your application runs on an Azure resource such as a Web app, it can authenticate with the Entra Id service using any managed identity that’s been associated with that Azure resource.



When your application authenticates using a managed identity, the Entra Id Service is able to verify where the token request originated. If it can verify that the token request originated from an Azure resource with an associated managed identity, authentication succeeds and an access token is returned to the caller.

Microsoft Azure supports both System-assigned Managed Identities and User-assigned Managed Identities. While both types of managed identity can be used to acquire access tokens for the Fabric REST APIs, you should understand some of the fundamental differences between them.

**System-assigned Managed Identities (SAMIs)** are easier to use because there is always a one-to-one relationship between the SAMI and an Azure resource. For example, an Azure Web application can only have one SAMI. You can create a SAMI with just a few clicks from the **Settings > Identity** page of an Azure resource in the Azure portal. Once it is created, it is ready to use.

A screenshot of a computer

Description automatically generated

When you authenticate using a SAMI, you don’t have to pass a client Id to identity the managed identity. That’s because there can only be one SAMI associated with the hosting Azure resource. Developers working on the .NET platform can leverage **Azure.Identity** library to implement the authentication flow required for a managed identity. Here is a code sample which uses a SAMI to authenticate and acquire an access token for a service principal.

private string GetAccessTokeForSami() {

ManagedIdentityCredential credential = new Azure.Identity.ManagedIdentityCredential();

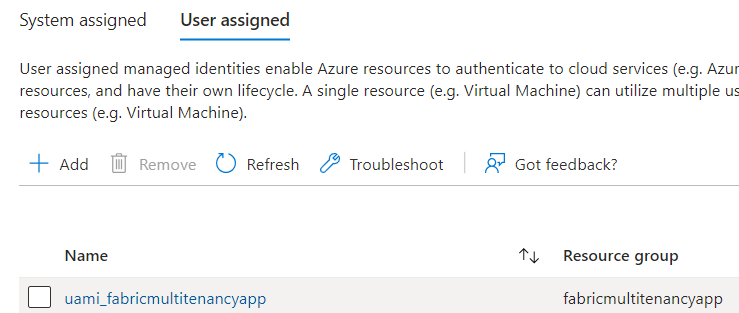
Azure.Core.AccessToken token = credential.GetToken(new Azure.Core.TokenRequestContext(scopes));

return token.Token;

}

**User-assigned Managed Identities (UAMIs)** provide more flexibility because they allow for many-to-many associations to Azure resources. For this reason, Microsoft generally recommends using UAMI instead of SAMIs. With UAMIs, you can create a set of managed identities and associate them all with a single Azure resource. Likewise, you can associate a single UAMI with multiple Azure resources.

Working with UAMIs requires an extra step. First, you create the UAMI from the **Managed Identity** page in the Azure portal. After creating the UAMI, you must then navigate in the Azure portal to the **Settings > Identity** page of an Azure resource. From there you can click the **Add** button to create the association between the UAMI and that Azure resource.



When using a UAMI, you must pass a Client Id to authenticate and acquire an access token. That’s because there isn’t a default one-to-one mapping between the hosting Azure resource and an associated UAMI.

private string GetAccessTokeForUAMI(string[] Scopes) {

string clientId = AppSettings.managedIdentityClientId;

string[] scopes = new string[] { "https://api.fabric.microsoft.com/.default" };

ManagedIdentityCredential credential = new Azure.Identity.ManagedIdentityCredential(clientId);

Azure.Core.AccessToken token = credential.GetToken(new Azure.Core.TokenRequestContext(Scopes));

return token.Token;

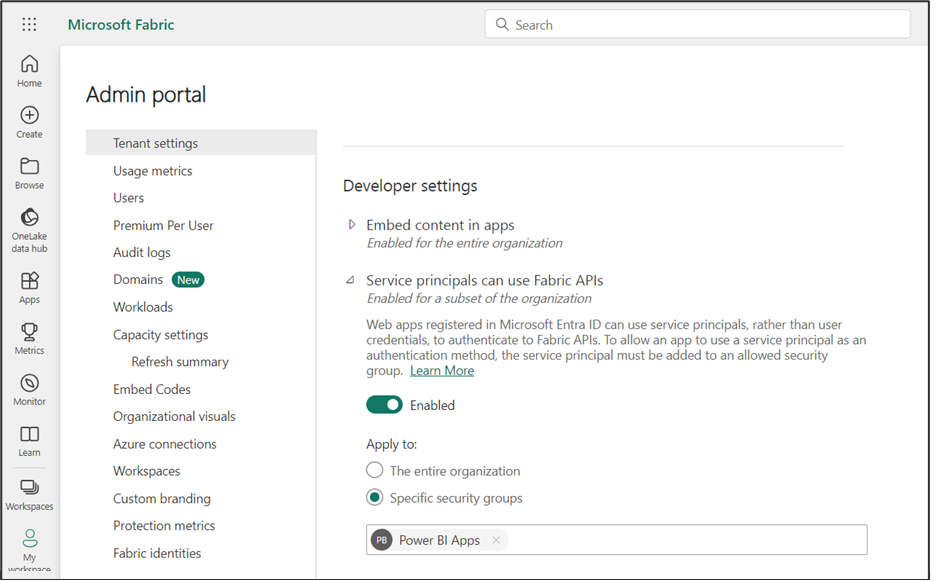
}

You have just seen that authenticating with a UAMI requires a Client Id while a SAMI does not. However, the more significant aspect is that neither SAMIs nor UAMIs require you to pass a client secret or client certificate to authenticate. Both SAMIs and UAMIs eliminate your application’s need to track secret credentials.

### Configure Service Principal Access to the Fabric REST API

Up to this point, you’ve learned how to authenticate and acquire access tokens for a service principal. You can accomplish this using either a custom Entra Id application or a managed identity. However, there still remains an extra configuration step that needs accomplished before a service principal has the permissions it needs to call the Fabric REST APIs. If you miss this step, calling to the Fabric REST APIs under the identity of the service principal will fail with an Access Denied error.

By default, a service principal does not possess the required permissions to call the Fabric REST APIs. Instead, a service principal must be configured in the Fabric admin portal using the tenant-level setting **Service Principals can use Fabric REST API**.



One complication with the **Service Principals can use Fabric REST API** setting is that you cannot directly add a service principal in the Fabric Admin portal. Instead, you can only configure this setting with users and Entra Id groups. The way to configure permissions for a service principal is by following these steps.

* Create an Entra Id group
* Add this Entra Id group to the **Service Principals can use Fabric REST API** setting
* Add service principals as members of the Entra Id group.

A close-up of a sign

Description automatically generated

Once you have configured a service principal with the **Service Principals can use Fabric REST API** setting, it will have the permissions it needs to call the Fabric REST APIs. But this does not mean that the service principal will automatically have access to any workspaces or workspace items. That’s because a service principal has no default access to any workspaces. In this sense, a service principal is treated just like a user.

If you want to enable a service principal with access to a specific workspace, you must assign the service principal to a workspace role. You can add a service principal into a workspace role such as Admin, Member, Contributor or Visitor. Once you add a service principal to a workspace in a role such as Admin or Member, the service principal will have the permissions it needs to begin creating workspace items.

In multitenant application development, it’s a best practice to automate the creation of workspaces using service principals. When you create a workspace using a service principal, it’s automatically assigned to the Admin role which provides full read-write access to the workspace and the workspace items inside.

### Configure Service Principal with Admin Access

The Fabric REST APIs provide a special set of Admin APIs that are made available to a smaller set of privileged users and service principals. The Fabric Admin APIs provide the caller with the ability to discover and inspect every workspace and every connection across the current Entra Id tenant. This is far more powerful than the non-Admin APIs because the caller can access every workspace without the requirement of being assigned workspace roles.

A screenshot of a computer

Description automatically generated

Many organization use the Fabric Admin APIs to build custom administrative applications that manage and monitor their Fabric environment. The Fabric Admin APIs can be used to discover workspaces and to inspect the workspace items inside. The Fabric Admin APIs can also inspect every connection across the tenant and alert the IT staff when users are creating connections that violate company-specific policies.

Now let’s discuss what is required to call the Fabric Admin APIs. That depends on whether the caller is a user or a service principal. To call the Fabric Admin API as a user, the user must first be configured in Entra Id in the role assignment of **Fabric Administrator**. This topic will be covered in greater detail later in this chapter.

Things are a bit different when configuring Admin API access for a service principal. Configuring Admin API access for service principal is accomplished in the **Admin API settings** section of the Fabric Admin portal using another tenant-wide setting named **Service principals can access read-only admins APIs**.

A screenshot of a computer

Description automatically generated

You can separate the Fabric REST APIs into two categories. There are the Admin APIs and the non-Admin APIs. As it turns out, the non-Admin APIs have their own name which is the **Fabric User APIs**. This can be confusing at first because the name suggests the User API for users not service principals. That’s not true. The Fabric User APIs are meant for both users and service principal.

The key to remember is that the Fabric Admin APIs allow you to discover and access every workspace and every connection on a tenant-wide basis. This is quite different than with the Fabric User APIs where caller can only access workspaces and connections in which they’ve been configured with access through role assignments.

## Authenticate and Acquire Access Tokens for a User

You have just learned how to authenticate and acquire access tokens to execute API calls as a service principal. Now it’s time to examine how things change when authenticating and acquiring access tokens to execute API calls as a user. Authenticating as a user is more complex because it relies on authentication flows which require interactive behavior on the part of the user. For example, users are often forced to interactively respond to MFA challenges by typing an access code into their mobile phone.

Writing code to acquire user tokens also introduces a second complication that doesn’t apply to service principals. Any request to acquire a user token must include a granular set of delegated permission scopes. Now it’s time to discuss how delegated permission scopes fit into the delegated access model which is part of Entra Id.

### Delegated Permission Scopes

The user authorization model in Entra Id is based upon the concept of **delegated access**. With delegated access, API calls are not executed with the same identity and the same permissions as the user. Instead, API calls executed with a user token are said to execute **on behalf of the user** which is very different than **by the user**.

When you execute an API call with a user token, the call executes with an application identity which is distinct from the user identity. Furthermore, the API call doesn’t execute with the full set of permissions that have been granted to the user. Instead, the API call executes with a subset of the user’s permissions as defined by the set of delegated permission scopes found inside the access token.

Let’s start with an example. Consider the following set of delegated permission scopes.

public static readonly string[] TenantProvisioning = new string[] {

"https://api.fabric.microsoft.com/Workspace.ReadWrite.All",

"https://api.fabric.microsoft.com/Connection.ReadWrite.All",

"https://api.fabric.microsoft.com/Item.ReadWrite.All",

"https://api.fabric.microsoft.com/Item.Execute.All",

“https://api.fabric.microsoft.com/Item.Reshare.All”

};

As you can see, this set of delegated permission scopes includes the **Workspace.ReadWrite.All** permission. This delegated permission scope grants your code read-write access to every workspace in which the user has been assigned permissions with read-write access. But the **Workspace.ReadWrite.All** permission does not give your code access to any workspaces that are not accessible to the current user.

Think about the scenario where the user has been granted read-only access to a specific workspace. In this case, your code will be limited to read-only access despite the presence of the **Workspace.ReadWrite.All** permission in the access token. The key point is that delegated permission scopes will never provide your code with a level of permissions greater than the current user.

The previous example includes **item-generic permission scopes** such as **Item.ReadWrite.All**. Item-generic permission scopes are used to grant permissions to every type of workspace item including lakehouses, notebooks, data pipelines, semantic models and reports.

public static readonly string[] GeneralProvisioningScopes = new string[] {

"https://api.fabric.microsoft.com/Item.ReadWrite.All",

"https://api.fabric.microsoft.com/Item.Execute.All",

“https://api.fabric.microsoft.com/Item.Reshare.All”

};

You can use **item-specific permission scopes** if you’re concerned with the security principle of least privilege and you would rather configure delegated permissions in a more granular fashion. Think about developing a solution where your code needs access to semantic models and reports but not for any other type of workspace items. You could use the following set of delegated permission scopes in a token acquisition request to achieve that end.

public static readonly string[] ItemSpecificProvisioningScopes = new string[] {

"https://api.fabric.microsoft.com/SemanticModel.ReadWrite.All",

"https://api.fabric.microsoft.com/SemanticModel.Execute.All",

“https://api.fabric.microsoft.com/SemanticModel.Reshare.All”,

"https://api.fabric.microsoft.com/Report.ReadWrite.All",

"https://api.fabric.microsoft.com/Report.Execute.All",

“https://api.fabric.microsoft.com/Report.Reshare.All”

}

Imagine you are writing code to acquire user tokens. Wouldn’t it be helpful for you to see a complete reference which lists all the delegated permission scopes supported by the Fabric REST APIs? Here is a handy little developer trick. You can run the following PowerShell script which uses the Microsoft Graph PowerShell SDK to generate a complete list of delegated permission scope support by the Fabric REST API.

if (!(Get-MgContext)) { Connect-MgGraph }

$outputFile = "$PSScriptRoot\FabricServicePermissions.txt"

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

$filter = "appId eq '" + $fabricServiceAppId + "'"

$fabricService = Get-MgServicePrincipal -Filter $filter -Property Oauth2PermissionScopes

$fabricServicePerms = $fabricService | Select-Object -ExpandProperty Oauth2PermissionScopes

"--- Fabric Service API Delegated Permission Scopes ---" | Out-File -FilePath $outputFile

$permList = $fabricServicePerms | Sort-Object Type, Value | Format-Table Value, Type, Id

$permList | Out-File -FilePath $outputFile -Append

Notepad $outputFile

The first time you run the script, you’ll be prompted to sign in to establish a connection to the Microsoft Graph API. Once connected, this script calls **Get-MgServicePrincipal** to obtain a reference to the service principal for the Fabric Service. After that, the script queries the service principal’s **Oauth2PermissionScopes** property which returns a list of delegated permission scopes which is then displayed as a simple text file in Notepad.

A screenshot of a computer

Description automatically generated

There are over 100 different delegated permission scopes defined by the Fabric REST APIs. If your application is designed to execute API calls using user tokens, you must ensure your application acquires access token which contain all the delegated permission scopes that required by your code. If you miss a delegated permission scope required by a particular Fabric API endpoint, calls to that endpoint will be rejected with an Access Denied error.

Remember, you can alwayts eliminate the complexity of determining which delegated permission scopes to include by designing your application to exeute API calls using a service principal.

### Implement User Authentication Flows using MSAL

The way to implement user authentication flows changes significantly depending upon the type of project you are developing. For example, there are versions of MSAL to use when you’re developing a web application with server-side code. There are other versions of MSAL to use when you’re developing a single page application (SPA) with a framework such as React.js where all the programming logic is written in client-side code. The **FabricIsvPlaybook** developer sample uses a version of MSAL that is mainly used in desktop application and mobile applications.

Earlier you saw how to acquire tokens for a service principal using the .NET version of MSAL named **Microsoft.Identity.Client**. Now we will use the same library to implement a user authentication flow to acquire user tokens. The **GetAccessTokenForUser** method shown in the following listing demonstrates how to implement an interactive authentication flow to acquire a user token.

private const string tenantCommonAuthority = "https://login.microsoftonline.com/organizations";

private static string GetAccessTokenForUser(string[] scopes) {

string clientId = AppSettings.UserAuthClientId;

string redirectUri = AppSettings.UserAuthRedirectUri;

// create new public client application

var appPublic = PublicClientApplicationBuilder.Create(clientId)

.WithAuthority(tenantCommonAuthority)

.WithRedirectUri(redirectUri)

.Build();

// redirect user to browser with Entra Id Login experience

var authResult = appPublic.AcquireTokenInteractive(scopes).ExecuteAsync().Result;

// return access token to caller

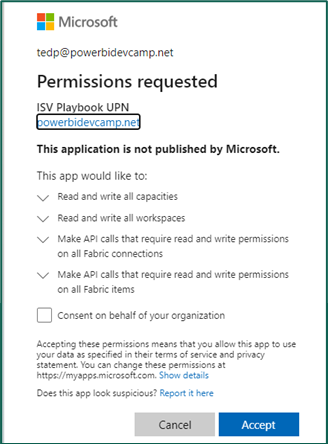
return authResult.AccessToken;

}

The code in the **GetAccessTokenForUser** method starts by creating a **PublicClientApplication** object which is initialized using the Client Id and Redirect URI of a custom Entra Id application. The code then calls **AcquireTokenInteractive** which starts the user authentication flow into motion. MSAL.NET responds by redirecting the user in the browser to a login page in the Entra Id Service where the user is prompted to sign in. Once the user completes the interactive sign in process, the call to **AcquireTokenInteractive** returns a response to the caller which contains an access token.

### Obtain User Consent for Delegate Permissions

Another important issue that can potentially impact an application’s ability to acquire user tokens is user consent. The fundamental idea is that the delegated access model requires that the user gets to approve of any delegated permission scope requested by an application. As a consequence, the Entra Id Service prompts the user with the **Permissions requested** consent dialog the first time a delegated permission scope is requested by a specific user.



The Entra Id Service remembers the list of delegated permission scopes to which the user has consented. That usually means that a user only see the **Permissions requested** consent dialog the first time they log into an application. However, this is not always the case.

An application can start by requesting a minimal set of delegated permission scopes in the initial request for a user token. If a user then invokes a command which require a greater set of permissions, the application can request an access token with additional delegated permissions scopes to which the user has not yet consented. Entra Id will then prompt the user with the **Permissions requested** consent dialog again to ensure the user has consented to each and every delegated permission scope that is added to an access token acquired by the client application.

### Admin-restricted permissions

Earlier you learned the Fabric REST APIs provide a special set of Admin APIs made available to a set of privileged users and service principals. These Admin APIs allow the caller with the ability to discover and inspect every workspace and connection with a tenant-wide scope.

There are two important requirements for executing Fabric Amin APIs with a user token. First, the user account must be assigned to the **Fabric Administrator** role in Entra Id as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

The second requirement is that the user tokens contains one of the Admin delegated permission scopes which are **Tenant.Read.All** and **Tenant.ReadWrite.All**.

public static readonly string[] FabricAdminScopes = new string[] {

"https://api.fabric.microsoft.com/Tenant.ReadWrite.All"

}

Note that any attempt to acquire an access token with the delegated permission scope of **Tenant.Read.All** or **Tenant.ReadWrite.All** will fail if the user hasn’t been configured as a Fabric administrator in the Entra Id Service.

### Authenticate Users with Microsoft’s Azure PowerShell Application

Image you’re in a situation in which you want to start experimenting with the Fabric REST APIs, but you’re working in a Fabric environment in which you cannot create your own Entra Id applications. This could be the case if you don’t have the required level of permissions in your organization’s Entra Id directory. Or maybe there’s just too much bureaucratic red tape involved with getting the IT department in your organization to create a custom Entra Id application on your behalf.

Fortunately, Microsoft provides a built-in Entra Id application which you can use to authenticate users and to acquire access tokens for the Fabric REST API. This Entra Id application is the **Azure PowerShell application** which is automatically installed and available in every Entra Id tenant. You can acquire user tokens by using the Azure PowerShell application which has a Client Id of **1950a258-227b-4e31-a9cf-717495945fc2** and a configured Redirect Uri of **http://localhost**.

The first benefit of authenticating users with Azure PowerShell application is that it eliminates your need for creating a custom Entra Id application. This can be valuable for developers in environments in which they cannot create their own Entra Id applications. A second benefit is that acquiring access tokens with the Azure PowerShell application simplifies dealing with delegated permission scopes.

When acquiring user tokens with the Azure PowerShell application, you only need to request a single delegated permission scope named **user\_impersonation**. The important thing to understand is that **user\_impersonation** is the all-powerful delegated permission scope. In other words, it automatically executes your API calls with the same level of permissions that have been granted to the current user.

The **GetAccessTokenForAzurePowershell** method shown in the following code listing illustrates using the **Azure PowerShell application** to acquire user tokens with the **user\_impersonation** delegated permission scope.

private static string GetAccessTokenForAzurePowershell() {

// Azure PowerShell application uses the same client Id across all Entra Id tenants

const string azurePowershellClientId = "1950a258-227b-4e31-a9cf-717495945fc2";

const string azurePowershellRedirectUri = "http://localhost";

string[] scopes = { "https://api.fabric.microsoft.com/user\_impersonation" };

// create new public client application

var appPublic = PublicClientApplicationBuilder.Create(azurePowershellClientId)

.WithRedirectUri(azurePowershellRedirectUri)

.WithAuthority(tenantCommonAuthority)

.Build();

// redirect user to browser with Entra Id Login experience

var authResult = appPublic.AcquireTokenInteractive(scopes).ExecuteAsync().Result;

// return access token result to caller

return authResult.AccessToken;

}

Keep in mind that the Azure PowerShell application is something that is best used in development environments. It should mainly be used in scenarios in which you’re creating a POC or just experimenting with the Fabric REST APIs. The Azure PowerShell application is not meant to be used when you deploy a custom application to production.

Remember you can only use the Azure PowerShell application in desktop applications because it’s configured with a Redirect Uri of **http://localhost**. When developing a web application, you cannot authenticate users with the Azure PowerShell application because you are not able to configure a custom Redirect Uri.

## The EntraIdTokenManager class

The **FabricIsvPlaybook** developer sample contains a class named **EntraIdTokenManager** which encapsulates all the application’s code to authenticate and acquire access tokens from the Entra Id Service. Other classes in the project such as **FabricRestApi** can acquire an access token by calling the **GetFabricAccessToken** method.

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

The **EntraIdTokenManager** class supports three different authentication modes.

1. Authenticate as a service principal using a confidential Entra Id application
2. Authenticate as a user using a public Entra Id application
3. Authenticate as a user using the built-in Azure PowerShell application

The **FabricIsvPlaybook** project makes it easy to switch between authentication modes. Inside the source file named **EntraIdTokenManager,** there is a C# enumeration named **AppAuthenticationMode** which contains three named constants for the three possible authentication modes.

public enum AppAuthenticationMode {

ServicePrincipalAuth,

UserAuth,

UserAuthWithAzurePowershell

}

In the **AppSettings** class, there’s a static property named **AuthenticationMode** which is based on enum named **AppAuthenticationMode**. The default value is set to **UserAuthWithAzurePowershell**.

public static AppAuthenticationMode AuthenticationMode = AppAuthenticationMode.UserAuthWithAzurePowershell;

The default authentication mode setting of **UserAuthWithAzurePowershell** has been chosen because it’s the only one that works automatically without requiring you to create a custom application in Entra Id. The other two possible authentication mode settings require you to create a custom Entra Id application and to update constants in the **AppSettings** class with metadata such as the application’s Client Id.

### Create a Custom Entra Id Application for User Authentication

The enable user authentication, you must create a new application in the Entra Id Service. You should create this application as a public application by adding a **Redirect URI** using the **Public client/native (mobile or desktop)** setting and a Redirect URI value of **http://localhost** as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

Once you’ve created the new Entra Id application, you need to copy and paste the Client Id and Redirect Uri into the two constants named **UserAuthClientId** and **UserAuthRedirectUri** in the **AppSettings** class.

// Public client application created in Entra Id Service for user auth

public const string UserAuthClientId = "22222222-2222-2222-2222-222222222222";

public const string UserAuthRedirectUri = "http://localhost";

Once you have configured **UserAuthClientId** and **UserAuthRedirectUri** in the **AppSettings** class, you can then set the **AuthenticationMode** to **UserAuth** before acquiring an access token.

// set app auth mode

AppSettings.AuthenticationMode = AppAuthenticationMode.UserAuth;

// acquire access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

When **AuthenticationMode** is set to **UserAuth,** a call to **GetFabricAccessToken** will use your custom Entra Id application to authenticate the user. The first time you authenticate using this application, you will be prompted by the **Requested permission** dialog due to the Entra Id policy of user consent with delegated access. Once you click the **Accept** button to consent, Entra Id will return an access token at the tail end of the authentication flow.

### Create a Custom Entra Id Application for Service Principal Authentication

The enable service principal authentication, you must create a new Entra Id application and configure it with a client secret. When creating an Entra Id application for a service principal, there is no need to add a Redirect Uri. That’s because Redirect Uris are only used with user authentication.

Once you have created an Entra Id application, you must then update three constants in the **AppSettings** class named **ServicePrincipalAuthTenantId**, **ServicePrincipalAuthClientId** and **ServicePrincipalAuthClientSecret**.

// Condifential client application created in Entra Id Service for service principal auth

public const string ServicePrincipalAuthTenantId = "33333333-3333-3333-3333-333333333333";

public const string ServicePrincipalAuthClientId = "44444444-4444-4444-4444-444444444444";

public const string ServicePrincipalAuthClientSecret = "ADD\_CLIENT\_SECRET\_HERE";

Make sure you configure your service principal with access to the Fabric REST APIs as discussed earlier this chapter in the section titled **Configure Service Principal Access to the Fabric REST API**. If you forget this step, you will still be able to acquire an access token for the service principal. However, all your API calls to the Fabric REST APIs will fail with an Access Denied error.

Once you have configured constants in the **AppSettings** class, you can then set the **AuthenticationMode** to **ServicePrincipalAuth** before acquiring an access token. After that, calling **GetFabricAccessToken** will return an access token with the identity of the service principal.

// set app auth mode

AppSettings.AuthenticationMode = AppAuthenticationMode.ServicePrincipalAuth;

// acquire access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

This guidance document will not dive further into the code of the **EntraIdTokenManager** class. That is left as an exercise for any reader interested in learning more about how to implement authentication flows with MSAL.

The **EntraIdTokenManager** class leverages MSAL token caching support where access tokens and refresh tokens for users are cached locally. This token caching support provides the development-time convenience of not having to interactively sign in each time you run a test with the **FabricIsvPlaybook** application. The **FabricIsvPlaybook** application can silently acquire access tokens either from the local token cache or by using a refresh token from the local cache to acquire a new access token from the Entra Id Service without requiring user interaction.

## Transmit an Access Token in a Fabric REST API Call

This chapter has already covered authenticating and acquiring access tokens for the Fabric REST APIs. The chapter will now conclude by showing how to include the access token when executing Fabric REST API calls. Examine the following code which demonstrates a simple “Hello World” example of calling the **List Workspaces** API.

// get access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

// parse together Authorization header value

string authorizationHeader = "Bearer " + accessToken;

// create HttpClient and set request headers

HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Add("Authorization", authorizationHeader);

client.DefaultRequestHeaders.Add("Accept", "application/json");

// create REST URI to call List Workspaces API

string restUri = "https://api.fabric.microsoft.com/v1/workspaces";

// execute GET request to call List Workspaces API

HttpResponseMessage response = client.GetAsync(restUri).Result;

Note the string for the **Authorization** header is parsed together using the word **Bearer** followed by a space and then the access token as shown in the following screenshot.



# Fabric REST API Fundamentals

The prior chapter ended with a code listing showing how to execute a secure API call to the **List Workspaces** API. You learned that the access token is transmitted by adding it to the **Authorization** request header. Now it’s time to discuss other important aspects of executing Fabric REST API calls such as detecting whether the call completed successfully and writing the code to consume JSON content returned inside the response body.

This chapter introduces the **Fabric REST API .NET SDK**. As you will see, this SDK provides a library of wrapper classes that can boost developer productivity by hiding many of the more tedious and low-level aspects of executing API calls using the Fabric REST APIs.

This chapter examines three other fundamental concepts you must understand to effectively program using the Fabric REST APIs. You will learn how to use continuation tokens when calling API endpoints that return paginated results. You will also learn how program Fabric REST API endpoints that support an asynchronous processing model known as long-running operations. The chapter concludes with a discussion of how the Fabric REST APIs enforce throttling to ensure an application cannot overload the service by calling too many API in a small window.

## Hello World with the Fabric REST API using C#

Let’s revisit the “Hello World” code from last chapter that calls the **List Workspaces** API. We’ll start by extending the prior code listing with an **if** statement with conditional logic that depends on whether the call succeeded.

// get access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

// parse together Authorization header value

string authorizationHeader = "Bearer " + accessToken;

// create HttpClient and set request headers

HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Add("Authorization", authorizationHeader);

client.DefaultRequestHeaders.Add("Accept", "application/json");

// create varaible with REST URI for List Workspaces API

string restUri = "https://api.fabric.microsoft.com/v1/workspaces";

// execute GET request to call List Workspaces API

HttpResponseMessage response = client.GetAsync(restUri).Result;

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

// handle case where API call was successful

string jsonWorkspacesList = response.Content.ReadAsStringAsync().Result;

}

else {

// handle case where API call was not successful

}

The first thing to notice is the **if** statement which inspects the HTTP status code returned by the call to the **List Workspaces** API. If you look at the documentation for the **List Workspaces** API, it tells you that that this API returns a status code of **200 OK** when to completes successfully. That allows you to add conditional logic that branches one way when the call succeeds and another way if the code fails.

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

// handle case where List Workspaces API call was successful

string jsonResponse = response.Content.ReadAsStringAsync().Result;

}

else {

// handle case where API call was not successful

}

Now let’s discuss what to do when the call to the **List Workspace** API succeeds. The next step is to read the JSON content from the body of the response with a call to **response.Content.ReadAsStringAsync().Result**.

// handle case where API call was successful

string jsonResponse = response.Content.ReadAsStringAsync().Result;

This code demonstrates retrieving the JSON content from the response body as a text-based value However, you still you need a way to read discrete property values from the JSON content which is structured in a manner shown in the following code listing.

{

"value": [

{

"id": "11111111-1111-1111-1111-111111111111",

"displayName": "Workspace 1",

"description": "",

"type": "Workspace",

"capacityId": "99999999-9999-9999-9999-999999999999"

},

{

"id": "22222222-2222-2222-2222-222222222222",

"displayName": "Workspace 2",

"description": "",

"type": "Workspace",

"capacityId": "99999999-9999-9999-9999-999999999999"

}

]

}

One popular technique used by .NET developers to read properties from JSON content is to create C# serialization classes to assist with converting back and forth between JSON and strongly-typed .NET objects. You can use the **JsonSerializer** class that is built into .NET to serialize C# objects into JSON. Likewise, you can deserialize JSON into one or more .NET objects.

Let’s examine how create serialization classes to deserialize the JSON content returned a call to **List Workspaces**. The **FabricWorkspace** class is used to deserialize workspace objects and the **ListWorkspaceResponse** class is used to deserialize the top-level response which contains a **value** property referencing a list of workspace objects.

class FabricWorkspace {

public string id { get; set; }

public string displayName { get; set; }

public string description { get; set; }

public string type { get; set; }

public string capacityId { get; set; }

}

class ListWorkspaceResponse {

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

}

Once you’ve created the serialization classes, you can deserialize the JSON returned from **List Workspaces** by calling **JsonSerializer.Deserialize**. Note that when calling **JsonSerializer.Deserialize**, you should include the **ListWorkspaceResponse** class in angled brackets to indicate the is the type to serialize the JSON content into.

// handle case where List Workspaces API call was successful

string jsonResponse = response.Content.ReadAsStringAsync().Result;

// deserailize JSON to strongly-typed ListWorkspaceResponse object

var listWorkspaceResponse = JsonSerializer.Deserialize<ListWorkspaceResponse>(jsonResponse);

Once you have deserialized the JSON into a **ListWorkspaceResponse** object, you can access the list of FabricWorkspace objects by accessing the **value** property. This makes it relatively easy to enumerate through the collection of workspaces.

// handle case where List Workspaces API call was successful

string jsonResponse = response.Content.ReadAsStringAsync().Result;

// deserailize JSON to strongly-typed ListWorkspaceResponse object

var listWorkspaceResponse = JsonSerializer.Deserialize<ListWorkspaceResponse>(jsonResponse);

// get list of workspace objects

List<FabricWorkspace> workspaces = listWorkspaceResponse.value;

// ennumerate through workspace objects

foreach (var workspace in workspaces) {

Console.WriteLine(workspace.displayName);

}

Now that you have seen how all the pieces fit together, let’s review all the code required to call **List Workspace**.

// get access token

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

// parse together Authorization header value

string authorizationHeader = "Bearer " + accessToken;

// create HttpClient and set request headers

HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Add("Authorization", authorizationHeader);

client.DefaultRequestHeaders.Add("Accept", "application/json");

// create varaible with REST URI for List Workspaces API

string restUri = "https://api.fabric.microsoft.com/v1/workspaces";

// execute GET request to call List Workspaces API

HttpResponseMessage response = client.GetAsync(restUri).Result;

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

// handle case where List Workspaces API call was successful

string jsonResponse = response.Content.ReadAsStringAsync().Result;

// deserailize JSON to strongly-typed ListWorkspaceResponse object

var listWorkspaceResponse = JsonSerializer.Deserialize<ListWorkspaceResponse>(jsonResponse);

// get list of workspace objects

List<FabricWorkspace> workspaces = listWorkspaceResponse.value;

// enuemrate through workspace objects

foreach (var workspace in workspaces) {

Console.WriteLine(workspace.displayName);

}

}

else {

// handle case where API call was not successful

}

What should be clear is that you must deal with quite few low-level details just to execute a Fabric REST API call. You are required to acquire an access token and include it in the **Authorization** request header. You are also required to inspect the HTTP status code returned by to call to **List Workspaces**. This allows you to determine whether the call was successful or not. Finally, you need some technique to read property values from a JSON result. As you have seen, this can be accomplished by adding custom serialization classes to your project and then using **JsonSerializer.Deserialize** to convert the JSON content into strongly-typed .NET objects.

## Develop with the Fabric REST API .NET SDK

Developers using the .NET platform to build Fabric solutions will really appreciate the **Fabric REST API .NET SDK**. This SDK provides a significant productivity boost to developers by hiding many of the tedious and low-level details associated with programming the Fabric REST APIs that were discussed earlier in this chapter.

You can add the **Fabric REST API .NET SDK** to a .NET project in Visual Studio by adding the NuGet package named **Microsoft.Fabric.API** as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

Once added the **Microsoft.Fabric.API** NuGet package, you will see that the Fabric REST API .NET SDK adds a large number of public classes to your project. You will Public class which end with the name Client act as class wrappers to endpoints exposed by the Fabric REST APIs.

A screenshot of a computer

Description automatically generated

Now that you know how to add the Fabric REST API .NET SDK to a project, let’s move ahead and discuss how to write the equivalent “Hello World” code to call the **List Workspaces** API. The easiest way to get started is to create a new **FabricClient** object which is initialize by passing an access token to the constructor.

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

FabricClient fabricApiClient = new FabricClient(accessToken);

The **FabricClient** object is a top-level object that provides access to individual client objects that wrap endpoints of the Fabric REST APIs. Here is an example of using the **Core.Workspaces** client to call the **List Workspaces** API. The call to **List Workspaces** returns an object which exposes a **ToList** method to return a standard .NET collection.

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

FabricClient fabricApiClient = new FabricClient(accessToken);

List<Workspace> workspaces = fabricApiClient.Core.Workspaces.ListWorkspaces().ToList();

foreach (var workspace in workspaces) {

Console.WriteLine(workspace.DisplayName);

}

At this point you have seen a comparison of the code required to call the **List Workspaces** API both with and without using the Fabric REST API .NET SDK. It should be clear that the.NET SDK makes programming much easier because it abstracts away executing HTTP requests with **HttpClient** and it automatically adds the access token to the **Authorization** header behind the scenes. There’s another convenience. You don’t have to inspect the HTTP status code to test for success. If the call to **ListWorkspaces** doesn’t throw an exception, the call succeeded.

Another important observation is that the Fabric REST API .NET SDK eliminates the need to create your own custom serialization classes. Instead, the .NET SDK defines its own set of serialization class which convert between JSON and strongly-typed .NET objects behind the scenes. The .NET SDK includes serialization class used to convert C# objects into a JSON payload used in the body of a POST requests such as a call to **Create Workspace**. Likewise, there are other serialization classes used to deserialize JSON content from the response body when calling Fabric REST APIs such as the **List Workspaces** API.

In the previous example, the variable named **workspaces** is based on the type of **List<Workspace>**. While the **List** type is defined in a standard .NET library, the **Workspace** type is a serialization class is defined by the .NET SDK as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

You will notice that the JSON returned by the the Fabric REST APIs use a lower case formatting with property names such as **id**, **displayName** and **capacityId**. However, the serialization types in the .NET SDK use an upper case formatting with names such as **Id**, **DisplayName** and **CapacityId**. Convertting between these format is something the .NET SDK does transparently behind the scenes as it serailizes and deserialized JSON.

You already seen several good reasons to leverage the .NET SDK when developing with the Fabric REST APIs. However, there are two more good reasons. The first reason is that the .NET SDK provides automatic support dealing with paginated results and continuation tokens. The second reason is that the .NET SDK provides automatic support for managing Fabric REST API calls which support the long running operations pattern. Over the next few sections, this chapter will introduce these topics and show the code required both with and without using the .NET SDK

## Paginated Results and Continuation Tokens

There are Fabric REST API endpoints that implement a pattern known as paginated results. The motivation for paginated results is the need to avoid passing too much data across the network at once. For example, an API call might request a list that is too large to pass back to the caller in a single response body. The paginated results pattern allows an API endpoint to pass data to the caller in smaller chunks (i.e. pages). The use of paginated results improves the performance and efficiency of API calls, especially when dealing with a large amount of data.

The Fabric REST APIs implement the paginated results pattern using **continuation tokens**. When an API calls returns with a continuation token, that tells the caller that there is additional data to retrieve. Here is an example of what a paginated results looks like when inspects the JSON returned in a call to the **List Workspaces** API.

{

"continuationToken": "ABCsMTAwMDAwLDA%3D",

"continuationUri": "https://api.fabric.microsoft.com/v1/workspaces?continuationToken=ABCsMTAwMDAwLDA%3D",

"value": [

{

"id": "11111111-1111-1111-1111-111111111111",

"displayName": "Workspace 1",

"description": "",

"type": "Workspace",

"capacityId": "99999999-9999-9999-9999-999999999999"

}

]

}

The presence of the **continuationToken** and **continuationUri** properties in the JSON response tells the caller that there is additional data to retrieve. With a paginated result, the caller is expected to execute a follow up GET request to the same Uri as the original GET request with a **continuationToken** query string value.

https://api.fabric.microsoft.com/v1/workspaces?continuationToken=ABCsMTAwMDAwLDA%3D

If the second call returns with the **continuationToken** and **continuationUri** properties, the call must make a third call an so on until there is a response which does not contain the **continuationToken** and **continuationUri** properties. Once a call returns without the **continuationToken** and **continuationUri** properties, the caller can assume it has retrieved all the data associated with the initial request.

### Program Paginated Results without the .NET SDK

Let’s revisit the code for calling the **List Workspaces** API shown earlier. The code that does not use the .NET SDK needs to be updated to deal with the paginated result pattern. Currently, the **ListWorkspaceResponse** serialization class is defined with a single property name **value**. We need to extend this serialization class by adding properties named **continuationToken** and **continuationUri**.

class ListWorkspaceResponse {

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

public string continuationToken { get; set; }

public string continuationUri { get; set; }

}

Now let’s update the C# code shown earlier to properly retrieve list data when dealing with paginated results.

// handle case where List Workspaces API call was successful

string jsonResponse = response.Content.ReadAsStringAsync().Result;

// deserailize JSON to strongly-typed ListWorkspaceResponse object

var listWorkspaceResponse = JsonSerializer.Deserialize<ListWorkspaceResponse>(jsonResponse);

// get list of workspace objects

List<FabricWorkspace> workspaces = listWorkspaceResponse.value;

// check to see of there are additional paginated results

while (listWorkspaceResponse.continuationUri != null) {

response = client.GetAsync(listWorkspaceResponse.continuationUri).Result;

jsonResponse = response.Content.ReadAsStringAsync().Result;

var paginatedResult = JsonSerializer.Deserialize<ListWorkspaceResponse>(jsonResponse);

workspaces.AddRange(paginatedResult.value);

}

// enumerate through workspace objects

foreach (var workspace in workspaces) {

Console.WriteLine(workspace.displayName);

}

Drilling down into the **while** loop, the test to check if **continuationUri** is not null is what drives the continued calls to retrieve additional paginated results. Once **continuationUri** is null, all the data has been retrieved and the flow of execution can exit the **while** loop.

// check to see of there are additional paginated results

while (listWorkspaceResponse.continuationUri != null) {

response = client.GetAsync(listWorkspaceResponse.continuationUri).Result;

jsonResponse = response.Content.ReadAsStringAsync().Result;

var paginatedResult = JsonSerializer.Deserialize<ListWorkspaceResponse>(jsonResponse);

workspaces.AddRange(paginatedResult.value);

}

Now you’ve seen how to retrieve paginated results when not using the Fabric REST API .NET SDK. Let’s now see how the .NET SDK makes it easier to deal with paginated results.

### Program Paginated Results with the .NET SDK

Now let’s examine code that uses the Fabric REST API .NET SDK and see how it compares to the code you’ve just seen that implements the paginated results pattern without using the .NET SDK. Here is an example is a call to **ListWorkspaces** with the .NET SDK which correctly deals with paginated results.

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

FabricClient fabricApiClient = new FabricClient(accessToken);

List<Workspace> workspaces = fabricApiClient.Core.Workspaces.ListWorkspaces().ToList();

foreach (var workspace in workspaces) {

Console.WriteLine(workspace.DisplayName);

}

You might notice that this code hasn’t changed from earlier. It’s the same code. What you can see if that a call to **ListWorkspaces().ToList** automatically deals with paginated results and continuation tokens behind the scenes. This obviously makes you code much cleaner and much easier to read, write and maintain.

## Long Running Operations

There are Fabric REST API endpoints that have been designed to execute synchronously. That means a call to this type of API completes its work before it returns to the caller. However, there are other Fabric REST API endpoints that support an asynchronous execution pattern known as **long-running operations**.

As a developer getting up to speed on the Fabric REST APIs, you must learn the fundamentals of of the long running operation pattern which includes monitoring the progress of an asynchronous API call in progress. You must also learn to retrieve the result of a long running operation after it has completed. Let’s start with a general discussion which compares synchronous API calls to an asynchronous API calls.

The **Create Workspace** API is an example of a synchronous API. A call to **Create Workspace** will complete its work before it returns to the caller. If a call to **Create Workspace** succeeds, it returns with an HTTP status code of **201 CREATED** and the response body contains JSON content with metadata such as the new workspace Id. If a call to **Create Workspace** fails, it returns with an HTTP status code other than **201 CREATED** and a response body contains JSON content with error information that (hopefully) describes went wrong.

APIs that are synchronous return HTTP status codes such as **200 OK** and **201 Created**. Asynchronous API calls are different because they return an HTTP status code of **202 Accepted**. When an API calls returns **202 Accepted**, it indicates that a job (i.e operation ) to be completed is queued up and will run sometime in the future. The caller who receives a **202 Accepted** response then must take additional steps to monitor the operation’s progress.

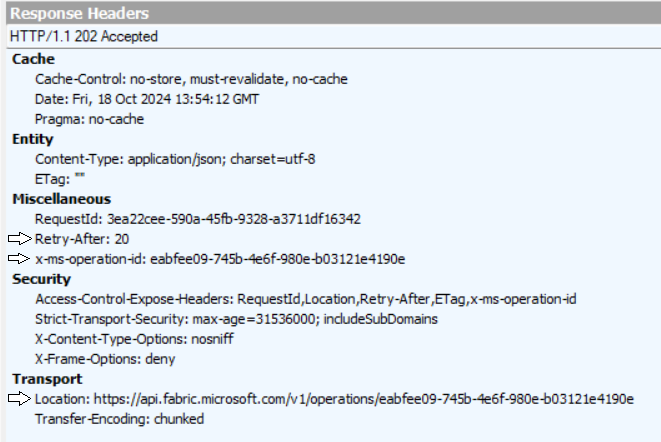
The **Create Item** API is an example of an API that support the long running operations pattern. If you examine the documentation for this API, you will see the response supports returning either **201 Created** or **202 Accepted**. That means the **Create Item** API sometimes runs synchronous returning **201 Created** and at other times it runs asynchronously returning **202 Accept**.

It turns out that the type of item being created determines whether a call to Create Item runs synchronously or asynchronously. For example, calling **Create Item** to create a lakehouse might execute synchronously while calling **Create Item** to create a notebook executes asynchronously as a long running operation.

### Response Headers for Long Running Operations

When a call to **Create Item** returns **202 Accepted**, it indicates the call will be executed asynchronously as a long running operation. That means you must find a way to monitor the operation’s progress. Fortunately, a call that returns **202 Accepted** to indicate that it’s a long running operation also includes a set of response headers that provide the essential information you need to monitor the progress and completion of any long running operation.

A Fabric REST API call that starts a long running operation will return with HTTP status code of **202 Accepted** along with three essential response headers named **x-ms-operation-id, Location** and **Retry-After**. The following screenshot shows how you can examine these there response headers using the Fiddler utility.



When a call to a Fabric REST API endpoint starts a long running operation, it generates a GUID-based Id for the operation which is passed to the caller in the response header named **x-ms-operation-id**. The **Location** response header contains a REST URI which can be used to call the **Get Operation State** API.

https://api.fabric.microsoft.com/v1/operations/{OPERATION\_ID}

After a long running operation has started, you need to call the **Get Operation State** API to determine whether the operation has completed or if it is still I progress.

The third response header is named **Retry-After**. This response header contains an integer value that tells the caller how long to wait before calling the **Get Operation State** API. For example, the **Retry-After** response header shown in the previous screenshot has a value of 20. This tells to the caller to wait 20 seconds before calling **Get Operation State** API to check on the status of the long running operation.

A call to **Get Operation State** returns with a JSON result in the response body that looks like this.

{

"status": "Succeeded",

"createdTimeUtc": "2024-10-18T13:55:35.6618747",

"lastUpdatedTimeUtc": "2024-10-18T13:55:46.708761",

"percentComplete": 100,

"error": null

}

As you can see, the response contains a **status** property that indicates the progress of the long running operation. The **status** property will always be set to on the following values

* **NotStarted**: The operation hasn’t started yet
* **Running**; The operation is in-progress
* **Succeeded**: The operation completed successfully
* **Failed**: The operation has failed

If you check the **status** property of a long running operation and see a value of **NotStarted** or **Running**, you know you have to wait for some period of time and then continue monitoring the operation until it completes. However, a **status** property value of **Succeeded** tells you the operation has completed successfully. If you see a **status** property value of **Failed**, it also means the operation has completed but not in a successful manner.

After you monitor a long running operation until it succeeds, you still have one more task to complete. That is you must call another API named **Get Operation Result** to retrieve information about the item that was created by the long running operation. For example, you need to call the **Get Operation Result** API after a long running operation to create a workspace item such as a semantic model. This allows you to read the metadata such as the item Id for the item that created by the operation.

A call to **Get Operation Result** will have a REST URI very similar to the REST URI for calling **Get Operation State**. The only difference is that a call **Get Operation Result** has a **/result** at the end

https://api.fabric.microsoft.com/v1/operations/{OPERATION\_ID}/result

The structure of the JSON returned from a call to **Get Operation Result** will vary depending on the Fabric REST API endpoint which started the long running operation. In the case where you started a long running operation by calling the **Create Item** API, a call to **Get Operation Result** will return JSON with properties for the workspace item which was created by the operation.

{

"id": "11111111-1111-1111-1111-111111111111",

"type": "Notebook",

"displayName": "Create Lakehouse Tables",

"description": "",

"workspaceId": "99999999-9999-9999-9999-999999999999"

}

**Program Long Running Operations without the .NET SDK**

First, let’s examine how to program a long running operation without using the .NET SDK. We will start by writing a generic function named **ExecutePostRequest** which defines string parameters for **restUri** and **postBody**.

private static string ExecutePostRequest(string restUri, string postBody = "") {

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Add("Authorization", "Bearer " + accessToken);

client.DefaultRequestHeaders.Add("Accept", "application/json");

HttpContent body = new StringContent(postBody);

body.Headers.ContentType = new MediaTypeWithQualityHeaderValue("application/json");

HttpResponseMessage response = client.PostAsync(restUri, body).Result;

// more to come soon

}

The call to **PostAsync.Result** returns a **HttpResponseMessage** object that exposes the **StatusCode** property. The **StatusCode** property lets you determine the HTTP status code returned by the API call. You need to inspect the HTTP status code to determine whether a call completed synchronously of if it started a long running operation.

HttpResponseMessage response = client.PostAsync(restUri, body).Result;

switch (response.StatusCode) {

// handle case where synchronous call succeeds with OK 200 or CREATED 201

case HttpStatusCode.OK:

case HttpStatusCode.Created:

// handle case where call started long running operation with ACCEPTED 202

case HttpStatusCode.Accepted:

}

It’s easy to deal with a synchronous API call which returns a status code of either **200 OK** or **201 Created**. You can just retrieve the result of the API call by reading the text-based JSON from the response body.

switch (response.StatusCode) {

// handle case where synchronous call succeeds with OK (200) or CREATED (201)

case HttpStatusCode.OK:

case HttpStatusCode.Created:

// return result to caller

return response.Content.ReadAsStringAsync().Result;

Things are far more complicated when you determine that a call returns with a status code of **202 Accepted**. That means the API call being processed as a long running operation. This first thing you should do is to inspect the three response headers named **x-ms-operation-id, Location** and **Retry-After**.

// handle case where call started long running operation with ACCEPTED 202

case HttpStatusCode.Accepted:

// get long running operation response headers

string operationId = response.Headers.GetValues("•\tx-ms-operation-id").First()

string uriOperationState = response.Headers.GetValues("Location").First();

int retryAfter = int.Parse(response.Headers.GetValues("Retry-After").First());

You can use the **Location** response header value to find the REST URI used to call the **Get Operation Status** API. You should also use the **Retry-After** response header value to determine how many seconds to wait before calling **Get Operation Status** as shown in the following code listing.

// handle case where call started long running operation with ACCEPTED 202

case HttpStatusCode.Accepted:

// get headers in response with URL for operation status and retry interval

string operationId = response.Headers.GetValues("x-ms-operation-id").First()

string uriOperationState = response.Headers.GetValues("Location").First();

int retryAfter = int.Parse(response.Headers.GetValues("Retry-After").First());

// wait for retry interval before calling Get Operation State

Thread.Sleep(retryAfter \* 1000);

// call Get Operation State

response = client.GetAsync(uriOperationState).Result;

At this point, you need new serialization class to deserialize the JSON returned by the **Get Operation Status** API. In this example, we will create a custom serialization class named **FabricOperation** to convert the JSON result from a call to **Get Operation Status** into a strongly-typed .NET object.

public class FabricOperation {

public string status { get; set; }

public DateTime createdTimeUtc { get; set; }

public DateTime lastUpdatedTimeUtc { get; set; }

public object percentComplete { get; set; }

public FabricErrorResponse error { get; set; }

}

The **FabricOperation** class allows you to convert the JSON result for an operation with **JsonSerializer.Deserialize**. You can now use the **operation** variable which references a **FabricOperation** object to inspect operation status.

// call Get Operation State

response = client.GetAsync(uriOperationState).Result;

string jsonOperation = response.Content.ReadAsStringAsync().Result;

// convert operation JSON result into .NET object

FabricOperation operation = JsonSerializer.Deserialize<FabricOperation>(jsonOperation);

Now that you have found a way to inspect **status** property, you can write the canonical **while** loop used to monitor the progress and completion of a long running operation. The **while** loop continues to execute whenever it sees a **status** value other than **Succeeded** or **Failed.** Inside the while loop, there is code to delays for a period of time set by the **retryAfter** interval and then it calls **Get Operation Status**.

while (operation.status != "Succeeded" && operation.status != "Failed") {

Thread.Sleep(retryAfter \* 1000); // wait for retry interval

Console.Write(".");

response = client.GetAsync(uriOperationState).Result;

jsonOperation = response.Content.ReadAsStringAsync().Result;

operation = JsonSerializer.Deserialize<FabricOperation>(jsonOperation);

}

This **while** loop executes continuously until it sees an operation status of either **Succeeded** or **Failed**. If the operation has a **status** value of **Succeeded**, it means the long operation has completed successfully. Once a long running operation completes successfully, there is still one remaining task. You need to call another API named **Get Operation Result** to retrieve a JSON result for the operation.

In most cases, a long running operation will have an associated result. For example, a long running operation which creates a workspace item will have a JSON result with the properties for the workspace item that was created. However, there are also cases where a long running operation does not any return. Therefore, you write condition code that determine if the long running operation has an associated result.

When a call to **Get Operation Status** returns with an **status** property of **Succeeded**, you can check to see if the response also contains the **Location** header. If the **Location** header is present in the response, it indicates that there is an operation result. If the **Location** header is present, it means the long running operation has no result.

The following listing demonstrates how to write condition logic to handle returning the result of a long running operation. In the case, where the operation **status** is **Succeeded** but the response does not contain the **Location** header, you can return an empty string because you know the operation has no associated result.

if (operation.status != "Succeeded") {

if (!response.Headers.Contains("Location")) {

// (1) handle case where operation has no associated result

return string.Empty;

}

else {

// (2) handle case where operation has associated result by retrieving it

string uriOperationResult = response.Headers.GetValues("Location").First();

response = client.GetAsync(uriOperationResult).Result;

return response.Content.ReadAsStringAsync().Result;

}

}

In the case where the operation **status** is **Succeeded** and the response contains the **Location** header, you can use the **Location** header value as the REST URI to call the **Get Operation Result** API. The call to **Get Operation Result** returns the JSON content for the operation result.

// get REST URI for calling Get Operation Result

string uriOperationResult = response.Headers.GetValues("Location").First();

// Call Get Operation Result

response = client.GetAsync(uriOperationResult).Result;

// return JSON content for operation result to caller

return response.Content.ReadAsStringAsync().Result;

Now that you have seen a walk though of all the details for programming a long running operation, take a moment to review the full implementation of **ExecutePostRequest** which implements the long running operations pattern.

private static string ExecutePostRequest(string restUri, string postBody = "") {

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

HttpContent body = new StringContent(postBody);

body.Headers.ContentType = new MediaTypeWithQualityHeaderValue("application/json");

HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Add("Accept", "application/json");

client.DefaultRequestHeaders.Add("Authorization", "Bearer " + accessToken);

HttpResponseMessage response = client.PostAsync(restUri, body).Result;

switch (response.StatusCode) {

// handle case where synchronous call succeeds with status code of OK (200) or CREATED (201)

case HttpStatusCode.OK:

case HttpStatusCode.Created:

return response.Content.ReadAsStringAsync().Result;

// handle case where call starts long running operation with status code of ACCEPTED (202)

case HttpStatusCode.Accepted:

// get headers in response with URL for operation status and retry interval

string operationId = response.Headers.GetValues("x-ms-operation-id").First();

string uriOperationState = response.Headers.GetValues("Location").First();

int retryAfter = int.Parse(response.Headers.GetValues("Retry-After").First());

// wait for retry interval before calling Get Operation State

Thread.Sleep(retryAfter \* 1000);

// call Get Operation State

response = client.GetAsync(uriOperationState).Result;

string jsonOperation = response.Content.ReadAsStringAsync().Result;

FabricOperation operation = JsonSerializer.Deserialize<FabricOperation>(jsonOperation);

while (operation.status != "Succeeded" && operation.status != "Failed") {

Thread.Sleep(retryAfter \* 1000); // wait for retry interval

response = client.GetAsync(uriOperationState).Result;

jsonOperation = response.Content.ReadAsStringAsync().Result;

operation = JsonSerializer.Deserialize<FabricOperation>(jsonOperation);

}

if (operation.status != "Succeeded") {

if (!response.Headers.Contains("Location")) {

// handle case where long running operation has no result

return string.Empty;

}

else {

// handle case where long running operation has result

string uriOperationResult = response.Headers.GetValues("Location").First();

response = client.GetAsync(uriOperationResult).Result;

return response.Content.ReadAsStringAsync().Result;

}

}

else {

// handle case where operation has status of Failed

}

default: // handle exeception where HTTP status code indicates failure

throw new ApplicationException("ERROR executing HTTP POST request " + response.StatusCode);

}

}

Now you’ve seen what it takes to program long running operations in a project where you are not using the Fabric REST API .NET SDK. If you take this approach, it is recommended you create a generic, reusable function such as **ExecutePostRequest** to eliminate having to implement the same long running operation pattern multiple times.

**Program Long Running Operations with the .NET SDK**

As you might expect, the Fabric REST API .NET SDK makes it much easier to program API calls that support long running operations. However, you still might be surprised at how much easier it is. You have seen all the code from the previous listing which is required to support an API call that support long running operations. All that code can be reduced to a single line of code. The following code listing shows the simplest way to handle the long running operation by calling **CreateItem** from the **Items** client and then chaining a second call to the **Value** property.

public static Item CreateItem(Guid WorkspaceId, CreateItemRequest CreateRequest) {

// call Create Item API with built-in support to handle long running operations

var newItem = fabricApiClient.Core.Items.CreateItem(WorkspaceId, CreateRequest).Value;

// return new item object to caller

return newItem;

}

As you can see, the.NET SDK support is able to build in automatic support for handling long running operations. A call to **CreateItem** with the **Items** client executes logic executes the call to the **Create Item** API across the network and then determines whether the call returned with a status code of either **201 Created** or **202 Accepted**.

If a call to the **Create Item** API returns a status code of **202 Accepted** to indicate the start of a long running operation, the .NET SDK includes conditional logic to wait for the retry after interval and then to call **Get Operation State** in a loop. There is also logic to call **Get Operation Result** after the loop completes to cache the operation result which you can access as a strongly-typed object using the **Value** property.

When you call **CreateItem**, the .NET SDK automatically provides the code to monitor the long running operation and to automatically return once the long running operation completes. Therefore, a call to **CreateItem** will patiently wait for as long as it the long running operation to complete. However, you can also add a timeout when calling **CreateItem** by adding the **timeoutInMinutes** parameter as shown in the following code listing.

var newItem = fabricApiClient.Core.Items.CreateItem(WorkspaceId, CreateRequest, **timeoutInMinutes:10**).Value;

When you pass the **timeoutInMinutes** parameter, the call to **CreateItem** will throw an exception if the long running operation has completed by the number of minutes specified in the timeout.

## Fabric REST API Throttling

The Fabric REST APIs use throttling to maintain optimal performance and reliability. Throttling limits the number of API calls which can be executed by a single caller within a specific time window. Throttling is enforced the same way regardless of whether the caller is a user or a service principal.

Consider a simple example where a Fabric REST API endpoint has a limit of 50 API calls per minute. If a single caller attempt to execute more than 50 API calls in a minute, the 51st API call will return with HTTP status code of **429** which indicates a **Too many requests** error.

When a Fabric REST API call returns with an HTTP status code of 429 (Too many requests), it will also include the **Retry-After** response header. The value of the **Retry-After** header indicates to the caller how many seconds to wait before reattempting the request.

A screenshot of a computer

Description automatically generated

# Program Capacities and Workspaces

Fabric REST API provides support for platform types and workspace item types. Platform item type instances created within scope of the current Entra Id tenant. Workspace item type instances created within scope of a Fabric workspace.

A screenshot of a computer

Description automatically generated

## Inspect Capacities

var capacities = fabricApiClient.Core.Capacities.ListCapacities().ToList();

foreach (var capacity in capacities) {

Console.WriteLine($"[{capacity.Sku}] {capacity.DisplayName} (ID={capacity.Id})");

}

## Create Workspaces

Here is code

public static Workspace CreateWorkspace(string WorkspaceName, string Description = null) {

var createRequest = new CreateWorkspaceRequest(WorkspaceName);

createRequest.Description = Description;

// associate workspace with Premium capacity

createRequest.CapacityId = new Guid(AppSettings.PremiumCapacityId);

// call Crete Workspace API

var workspace = fabricApiClient.Core.Workspaces.CreateWorkspace(createRequest).Value;

// determine Id of new workspace

Guid workspaceId = workspace.Id;

// return workspace object to caller

return workspace;

}

## Assign Workspace Roles to User, Groups and Service Principals

Adding users

public static void AddUserAsWorkspaceMemeber(Guid WorkspaceId, Guid UserId, WorkspaceRole RoleAssignment) {

var user = new Principal(UserId, PrincipalType.User);

var roleAssignment = new AddWorkspaceRoleAssignmentRequest(user, RoleAssignment);

fabricApiClient.Core.Workspaces.AddWorkspaceRoleAssignment(WorkspaceId, roleAssignment);

}

Ssss

public static void AddGroupAsWorkspaceMemeber(Guid WorkspaceId, Guid GroupId, WorkspaceRole RoleAssignment) {

var group = new Principal(GroupId, PrincipalType.Group);

var roleAssignment = new AddWorkspaceRoleAssignmentRequest(group, RoleAssignment);

fabricApiClient.Core.Workspaces.AddWorkspaceRoleAssignment(WorkspaceId, roleAssignment);

}

Ssss

public static void AddServicePrincipalAsWorkspaceMemeber(Guid WorkspaceId, Guid ServicePrincipalObjectId, WorkspaceRole RoleAssignment) {

var user = new Principal(ServicePrincipalObjectId, PrincipalType.ServicePrincipal);

var roleAssignment = new AddWorkspaceRoleAssignmentRequest(user, RoleAssignment);

fabricApiClient.Core.Workspaces.AddWorkspaceRoleAssignment(WorkspaceId, roleAssignment);

}

Xxx

var workspace = FabricRestApi.CreateWorkspace(WorkspaceName, "test workspace");

Guid TestUser1Id = new Guid(AppSettings.TestUser1Id);

Guid TestUser2Id = new Guid(AppSettings.TestUser2Id);

Guid TestADGroup1 = new Guid(AppSettings.TestADGroup1);

Guid TestServicePrincipal = new Guid(AppSettings.ServicePrincipalObjectId);

FabricRestApi.AddUserAsWorkspaceMemeber(workspace.Id, TestUser1Id, WorkspaceRole.Admin);

FabricRestApi.AddUserAsWorkspaceMemeber(workspace.Id, TestUser2Id, WorkspaceRole.Viewer);

FabricRestApi.AddGroupAsWorkspaceMemeber(workspace.Id, TestADGroup1, WorkspaceRole.Member);

FabricRestApi.AddServicePrincipalAsWorkspaceMemeber(workspace.Id, TestServicePrincipal, WorkspaceRole.Admin);

## Provision Workspace Identity

xxxxxx

# Programming Connections

Creating connections programmatically is required in scenarios which involve OneLake Shortcuts, Data Pipelines and Semantic Models.

Before we dive into a discussion of connections, let’s take a moment to distinguish between **Inbound Security** versus **Outbound Security**. Inbound security is involved when a custom application executes API calls on Fabric REST API endpoints. A key point is that the custom application runs outside the Fabric environment. Before the application can call to the Fabric REST API, it must first authenticate with the Entra Id Service in order to acquire access tokens. It must then transmit an access token in each and every API call to Fabric REST API endpoints. This is a topic that has already been covered earlier in this guidance document.

Outbound security is different because it involves a scenario where you’ve created some type of workspace item inside the Fabric environment which must connect to an external datasource. For example, you can create a semantic model which connects to an Azure SQL database. In another example, you can create a OneLake shortcut which connects to an ADLS storage container. With outbound security, you can create and bind connections using the Fabric REST API as shown in the following diagram.

A close-up of a screen

Description automatically generated

Here is an important factor to keep in mind. When you create a connection with code, you are not really establishing a connection across the network at that point in time. Instead, you are really just creating a persistent Fabric object with metadata for creating a connection at some point in the future. This metadata includes the datasource type and path as well as security credentials. It is not until the connection is actually used by something in Fabric such as a semantic model or a OneLake shortcut when the Fabric Service reads this metadata and uses it to establish a connection to the datasource across the network.

Connections in Fabric are scoped at level of Entra Id tenant. That means that Fabric connection can be shared across workspaces. Of course, just because you can doesn’t mean that you should.

You can see all the connections you have access to using the Manage Connections and Gateways page in the Fabric Service.

A screenshot of a computer

Description automatically generated

Fabric connections support four different connectivity types.

1. Personal Cloud Connections (PCCs)
2. Sharable Cloud Connections (SCCs)
3. On-prem Gateway Connections
4. Virtual Gateway Connections

Personal Cloud Connections (PCCs) have been used in Power BI for years, However, they are limited because they cannot be shared. Each PCC is owned and exlussively used by a single user or service principal. There is no way to share a PCC.

Sharable Cloud Connections (SCCs) are new and serve as the strategic replacement for PCCs. Once a SCC has been created, it can be shared with other users or service principals. The creator of a Connection is automatically configured with connection Role Assignment of Owner. Other users and service principals can be added to the SCC membership with a Role Assignments of User, UserWithReshare or Owner.

## Create an Anonymous Web Connection

Content to come

## Create a Azure SQL Connectiong using Basic Credentials

Content to come

## Create an Azure Storage Connection using Service Principal Credentials

Content to come

# Create and Update Workspace Items

Fabric solutions built using workspace items. Developers must learn to discover, create and manage workspace items inside scope of a workspace.

A screenshot of a computer screen

Description automatically generated

Discover what items exist in specific workspace by calling List Items API

A screenshot of a computer

Description automatically generated

## Item Definitions in Fabric

Fabric items can be created and updated using item definitions. You can pass item definition when calling Create Item API. You can retrieve item definition for existing Fabric item by calling Get Item Definition. You can modify existing workspace item by calling Update Item Definition passing item definition.

A diagram of a process

Description automatically generated

When calling Create Item, you pass displayName, type and optionally an item definition.

Some type of Fabric Items are created using an item definition while others are not.

So, what exactly is an item definition? An item definition includes array of parts where each part is item-specific file. For example, you can create an item definition for a semantic model using two files which are **definition.pbism** and **model.bim**. Likewise, you can create an item definition for a report using three files which are **definition.pbir**, **report.json** and a report theme file.

A screenshot of a computer

Description automatically generated

File content for parts converted to/from inline base64 format when transmitted across network. Item definition for each item type requires unique set of parts.

## Create a Semantic Model using the CreateItem API

Constructing Item Definitions for the CreateItem API. FabricItemDefinitionFactory class used to build item definitions for calls to CreateItem. Examples provided for semantic models, reports, notebooks and data pipelines. Template files for item definitions maintained in ItemDefinitions folder as embedded resources. The utility method CreateInlineBase64Part converts standard string into Base64 string for inline part.

The method GetImportedSalesModelCreateRequest constructs item definition to create semantic model.

public static CreateItemRequest GetImportedSalesModelCreateRequest(string DisplayName) {

string part1FileContent = FabricIsvPlaybook.Properties.Resources.definition\_pbism;

string part2FileContent = FabricIsvPlaybook.Properties.Resources.sales\_model\_import\_bim;

var createRequest = new CreateItemRequest(DisplayName, ItemType.SemanticModel);

createRequest.Definition =

new ItemDefinition(new List<ItemDefinitionPart>() {

CreateInlineBase64Part("definition.pbism", part1FileContent),

CreateInlineBase64Part("model.bim", part2FileContent)

});

return createRequest;

}

Let’s focus on this code.

createRequest.Definition =

new ItemDefinition(new List<ItemDefinitionPart>() {

CreateInlineBase64Part("definition.pbism", part1FileContent),

CreateInlineBase64Part("model.bim", part2FileContent)

});

N

private static ItemDefinitionPart CreateInlineBase64Part(string Path, string Payload) {

string base64Payload = Convert.ToBase64String(Encoding.UTF8.GetBytes(Payload));

return new ItemDefinitionPart(Path, base64Payload, PayloadType.InlineBase64);

}

This code calls new on **new** on the **ItemDefinition** class of the .NET SDK and initializes with a list of item definition parts. In this case the parts are for a semantic model and they include definition.pbism and model.bim.

var workspace = FabricRestApi.CreateWorkspace(WorkspaceName);

var modelCreateRequest = FabricItemDefinitionFactory.GetImportedSalesModelCreateRequest(ImportedModelName);

var model = FabricRestApi.CreateItem(workspace.Id, modelCreateRequest);

Here is code

public static Item CreateItem(Guid WorkspaceId, CreateItemRequest CreateRequest) {

// call CreateItem API to create new item

var newItem = fabricApiClient.Core.Items.CreateItemAsync(WorkspaceId, CreateRequest).Result.Value;

// return object for new new workspace item to caller

return newItem;

}

This is e URL with the post.

https://api.fabric.microsoft.com/v1/workspaces/{WORKSPACE\_ID}/items

This is what gets sent across the network.

The call

A screenshot of a computer

Description automatically generated

When you call the CreateItem API to create a new semantic model, the call is processed as a long running operation (LRO). The POST request

* Location: https://api.fabric.microsoft.com/v1/operations/{OPERATION\_ID}
* Retry-After : 20

The developer needs to write code that waits 20 seconds and then sends a GET request to the URL in the Location to determine the operation status. The returned result has a status property that must be set to true before the result is available.

A screenshot of a computer

Description automatically generated

Then you can get the result of the LRO which is the metadata for the semantic model that has just been created.

A close-up of a product

Description automatically generated

What nice is that Result.Value, the SDK hides the complexity of having to deal with this call as a LRO.

// call CreateItem API to create new item

var newItem = fabricApiClient.Core.Items.CreateItemAsync(WorkspaceId, CreateRequest).Result.Value;

## Prepare the Semantic Model After Creation

Import-mode semantic model needs to be refreshed after it has been created from item definition. The key takeaway is that call to the CreateItem API creates a semantic model instance but it does not automatically populate semantic model with data.

var workspace = FabricRestApi.CreateWorkspace(WorkspaceName);

// create seantic model

var modelCreateRequest = FabricItemDefinitionFactory.GetImportedSalesModelCreateRequest(ImportedModelName);

var model = FabricRestApi.CreateItem(workspace.Id, modelCreateRequest);

// create new connection for semantic model

var url = PowerBiRestApi.GetWebDatasourceUrl(workspace.Id, model.Id.Value);

var connection = FabricConnectionsApi.CreateAnonymousWebConnection(url);

// bind connection to semantic model using Power BI REST API

PowerBiRestApi.BindSemanticModelToConnection(workspace.Id, model.Id.Value, new Guid(connection.id));

// refresh semantic model using Power BI REST API

PowerBiRestApi.RefreshDataset(workspace.Id, model.Id.Value);

Currently, Power BI REST API required to bind connection and refresh semantic model. Developer binds semantic model to connection using BindToGatewayInGroup API.

public static void BindSemanticModelToConnection(Guid WorkspaceId, Guid SemanticModelId, Guid ConnectionId) {

BindToGatewayRequest bindRequest = new BindToGatewayRequest { DatasourceObjectIds = new List<Guid?>() };

bindRequest.DatasourceObjectIds.Add(ConnectionId);

pbiClient.Datasets.BindToGatewayInGroup(WorkspaceId, SemanticModelId.ToString(), bindRequest);

}

In the fullness of time, this binding and refresh functionality will be added to the Fabric REST API.

## Create Reports using the Create Item API

When creating report from item definition, you must bind report to semantic model. Report definition is maintained in report.json while definition.pbir binds report to specific semantic model.

When you create a new report, you need to bind it to a target semantic model. When you create a new semantic model, you need to capture item Id returned from Create Item. When creating the item definition for the report, you use item Id for semantic model for binding.

Let’s examine a template file for definition.pbir.

{

"version": "1.0",

"datasetReference": {

"byPath": null,

"byConnection": {

"connectionString": null,

"pbiServiceModelId": null,

"pbiModelVirtualServerName": "sobe\_wowvirtualserver",

"pbiModelDatabaseName": "**{SEMANTIC\_MODEL\_ID}**",

"name": "EntityDataSource",

"connectionType": "pbiServiceXmlaStyleLive"

}

}

}

Ssssssss

**// substitute target SemanticModelId into item part named definition.pbir**

string part1FileTemplate = FabricIsvPlaybook.Properties.Resources.definition\_pbir;

string part1FileContent = part1FileTemplate.Replace("**{SEMANTIC\_MODEL\_ID}**", SemanticModelId.ToString());

xxx

public static CreateItemRequest GetSalesReportCreateRequest(Guid SemanticModelId, string DisplayName) {

// substitute target SemanticModelId into item part named definition.pbir

string part1FileTemplate = FabricIsvPlaybook.Properties.Resources.definition\_pbir;

string part1FileContent = part1FileTemplate.Replace("**{SEMANTIC\_MODEL\_ID}**", SemanticModelId.ToString());

string part2FileContent = FabricIsvPlaybook.Properties.Resources.sales\_report\_json;

string part3FileContent = FabricIsvPlaybook.Properties.Resources.CY24SU02\_json;

var createRequest = new CreateItemRequest(DisplayName, ItemType.Report);

createRequest.Definition =

new ItemDefinition(new List<ItemDefinitionPart>() {

CreateInlineBase64Part("definition.pbir", part1FileContent),

CreateInlineBase64Part("report.json", part2FileContent),

CreateInlineBase64Part("StaticResources/SharedResources/BaseThemes/CY24SU02.json", part3FileContent),

});

return createRequest;

}

Dddddd

var workspace = FabricRestApi.CreateWorkspace(WorkspaceName);

// create semantic model

var modelCreateRequest = FabricItemDefinitionFactory.GetImportedSalesModelCreateRequest(ImportedModelName);

var model = FabricRestApi.CreateItem(workspace.Id, modelCreateRequest);

// prepare semantic model

var url = PowerBiRestApi.GetWebDatasourceUrl(workspace.Id, model.Id.Value);

var connection = FabricConnectionsApi.CreateAnonymousWebConnection(url);

PowerBiRestApi.RefreshDataset(workspace.Id, model.Id.Value);

// create CreateItemRequest object with target semantic model Id

var createRequestReport =

FabricItemDefinitionFactory.GetSalesReportCreateRequest(model.Id.Value, ImportedModelName);

// create report which is bound to semantic model created earlier

var report = FabricRestApi.CreateItem(workspace.Id, createRequestReport);

We have just implemented the classic Power BI provisioning flow for an import-mode semantic model and an associated report.

1. Create workspace
2. Create semantic model
3. Create connection
4. Bind connection to semantic model
5. Refresh semantic model
6. Create report bound to semantic model

Report definition has dependencies on names of tables, columns and measures in semantic model. Report has no dependencies on the type of underlying semantic model. A single report definitions can be used across different types of semantic models. Shared report definition can be bound to import-model semantic models and DirectLake semantic models.

## Export Existing Items using Get Item Definition

Get Item Definition API allows you retrieve item definition for existing items. Call Get Items to discover set of item Ids in workspace then call Get Item Definition once for each item Id. Make it possible to create a folder on local file system with set of files for each item definition part

A close-up of a sign

Description automatically generated

POST request to the follow URL.

https://api.fabric.microsoft.com/v1/workspaces/{WORKSPACE\_ID}/items/{ITEM\_ID}/getDefinition

This API is processed as a long running operation (LRO).

https://api.fabric.microsoft.com/v1/workspaces/{WORKSPACE\_ID}/items/{ITEM\_ID}/getDefinition?format=TMSL

This is what you get

A screenshot of a computer

Description automatically generated

Dump to local file system

A close-up of a white background

Description automatically generated

Xxxx

A screenshot of a computer program

Description automatically generated

Xxx

A screenshot of a computer

Description automatically generated

## Update Workspace Items using Update Item

This API is used to update two generic properties that are common across all workspace items. These two properties include display name and description.

public static Item UpdateItem(Guid WorkspaceId, Guid ItemId, string DisplayName, string Description = null) {

var updateRequest = new UpdateItemRequest {

DisplayName = DisplayName,

Description = Description

};

return fabricApiClient.Core.Items.UpdateItem(WorkspaceId, ItemId, updateRequest).Value;

}

### Update Workspace Items using Update Item Definition

You cannot use the Update Item API when you need to update the underlying item definition for a workspace item such as a semantic model or a report. Instead, you must call the Update Item Definition API and pass an item definition that includes the required changes.

A close-up of a sign

Description automatically generated

public static UpdateItemDefinitionRequest GetImportedSalesModelUpdateRequest(string DisplayName) {

string part1FileContent = FabricIsvPlaybook.Properties.Resources.definition\_pbism;

string part2FileContent = FabricIsvPlaybook.Properties.Resources.sales\_model\_import\_v2\_bim;

return new UpdateItemDefinitionRequest(

new ItemDefinition(new List<ItemDefinitionPart>() {

CreateInlineBase64Part("definition.pbism", part1FileContent),

CreateInlineBase64Part("model.bim", part2FileContent)

}));

}

Just like when constructing an item definition for a Create Item, you prepare the exact same type of item definition when you call Update Item Definition.

Note you must include every item definition part when calling the Update Item Definition API. You cannot just pass an item definition with the parts that have changed.

Here is some code.

// get item definition with updated verion 2 file for model.bim

var updateModelRequest = FabricItemDefinitionFactory.GetImportedSalesModelUpdateRequest(ImportedModelName);

// call

fabricApiClient.Core.Items.UpdateItemDefinition(WorkspaceId, ItemId, updateModelRequest);

Send POST request to this URL.

https://api.fabric.microsoft.com/v1/workspaces/{WORKSPACE\_ID}/items/{ITEM\_ID}/updateDefinition

Use Update Item Definition API to update semantic models and reports. You can update semantic models and reports completely independent of one another

A computer screen shot of a computer program

Description automatically generated

Now update the report. Demo with Version 2 of report has updated layout and uses new built-in theme. Updated report content for layout maintained in sales\_report\_v2\_json resource. Report update includes adding new built-in theme named NewExecutive.json.

public static UpdateItemDefinitionRequest GetSalesReportUpdateRequest(Guid SemanticModelId, string DisplayName) {

string part1FileContent = FabricIsvPlaybook.Properties.Resources.definition\_pbir.Replace("{SEMANTIC\_MODEL\_ID}", SemanticModelId.ToString());

string part2FileContent = FabricIsvPlaybook.Properties.Resources.sales\_report\_v2\_json;

string part3FileContent = FabricIsvPlaybook.Properties.Resources.CY24SU02\_json;

string part4FileContent = FabricIsvPlaybook.Properties.Resources.NewExecutive\_json;

return new UpdateItemDefinitionRequest(

new ItemDefinition(new List<ItemDefinitionPart>() {

CreateInlineBase64Part("definition.pbir", part1FileContent),

CreateInlineBase64Part("report.json", part2FileContent),

CreateInlineBase64Part("StaticResources/SharedResources/BaseThemes/CY24SU02.json", part3FileContent),

CreateInlineBase64Part("StaticResources/SharedResources/BuiltInThemes/NewExecutive.json", part4FileContent)

}));

}

Sss

A screenshot of a computer

Description automatically generated

### Clone Workspace Items to a New Workspace

Fabric REST API makes it possible to clone workspaces. Enumerate through all items in source workspace using ListItems. Read item definitions from source workspace using GetItemDefinition. When required, update item definition of source item.

A diagram of a process

Description automatically generated with medium confidence

Steps to cloning semantic models

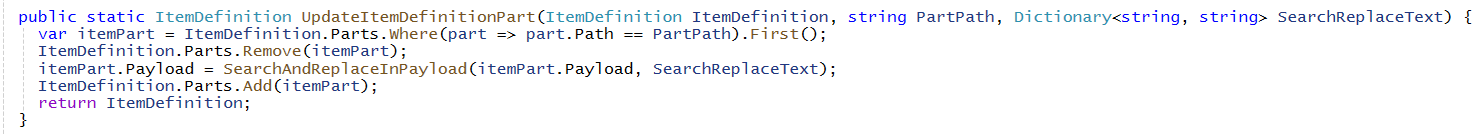
* Create and bind connections
* Refresh import-mode semantic models
* Create dictionary with semantic model Ids to redirect reports from source workspace to clone workspace

Report item definitions require update. You must substitute semantic model Ids to reference semantic models in clone workspace

A screen shot of a computer code

Description automatically generated

X



Now this.

A screen shot of a computer code

Description automatically generated

# Deploy Power BI Solutions with Semantic Models and Reports

Power BI Desktop supports Power BI Desktop Developer Mode. Power BI Desktop has traditionally allowed saving projects in PBIX file format. Power BI Desktop now allows saving projects in Power BI Project (PBIP) format.

* Power BI Desktop saves semantic model definition using model.bim and definition.pbism
* Power BI Desktop saves report definition using report.json and definition.pbir

A screenshot of a computer

Description automatically generated

.pbi folder used to store local resources that should not be saved into source control system.

To use PBIP formats, you must enable **Power BI Project (.pbip) save option** in Power BI Desktop Options dialog.

A screenshot of a computer program

Description automatically generated

Power BI Desktop offers a feature to save the item definitions for semantic models and report in more-modern and advanced formats. Optionally configure semantic models and reports to use newest definition formats.

A white background with black text

Description automatically generated

## TMSL versus TMDL - Semantic Model Formats

Semantic Model Definitions can be saved in TMSL format or TMDL format

* TMSL is simple - a single model.bim file contains entire semantic model definition
* TMDL is more granular – better for source control when multiple developers are working on same model

A screenshot of a computer

Description automatically generated

## PBIR-Legacy versus PBIR - Report Definition Formats

Report Definitions can be saved in PBIR-Legacy format or PBIR format. PBIR-Legacy is original Power BI report definition format with entire definition stored in report.json file. PBIR is modern, more granular format which allows for editing/creating definitions using JSON schema

A screenshot of a computer

Description automatically generated

# Deploy Solutions with Lakehouses and Notebooks

Let’s start with Fabric lakehouse fundamentals. Lakehouse used to store structured data and unstructured data in a single location. Lakehouse data transparently stored in OneLake using Azure Data Lake Storage Gen2 (ADLS Gen2). Lakehouse data accessible across workloads like data engineering, data warehouse, data science & BI.

Lakehouse enables storing and refining data using medallion architecture (bronze>silver>gold)

Lakehouse has a specific structure. The **Tables** section used to store structure data in tables in Delta format. **Files** section used to store unstructured data as files.

A screenshot of a computer

Description automatically generated

Ingesting Data into Lakehouse.

* Running Spark jobs
* Running Data Pipelines
* Creating Shortcuts
* Using ADLS Gen2 APIs
* Running Fabric Gen2 Dataflows

For running Spark jobs, you can run them as notebooks or as Spark Job Definitions

For running Data Pipelines. You can build a data pipeline with activities to copy data from external sources

Creating Shortcuts allows you to create references to make files from an external source appear to be in lakehouse storage

You can use ADLS Gen2 APIs to copy files into the File section of a lakehouse. This is implemented by client application which uploads files into lakehouse storage using push approach

Running Fabric Gen2 Dataflows. Running a dataflow which imports data and saves it delta tables in lakehouse storage

### The Apache Spark Runtime

Fabric Provides Apache Spark Runtime. Spark is a unified engine for large-scale data analytics. Spark code can be executed through Spark Job Definition. Spark code can be executed using interactive Fabric Notebook.

A diagram of a spark session

Description automatically generated

Apache Spark provides APIs to read and write data to/from Lakehouse in OneLake

Steps to provisioning flow which builds out medallion architecture

* **Create bronze layer** by coping CSV files into **Files** section
* **Create silver layer** by loading CSV data into dataframes and saving them as delta tables with OLTP schema
* **Create gold layer** by transforming data from silver tables into delta tables with star schema

A screenshot of a computer

Description automatically generated

You can leverage Fabric support for Spark to create and populate lakehouse tables

* Create lakehouse (or multiple lakehouses for greater isolation)
* Create Spark Environment as workspace item for running Spark jobs (when Environments API becomes available)
* Create Notebook (or Spark Job Definition) containing ETL logic to ingest data files and to create tables
* Run Notebook using Job Scheduler – Run Job On-demand API

### Create a Lakehouse using the Create Item API

Lakehouse created without item definition – you only pass displayName and type.

public static Item CreateLakehouse(Guid WorkspaceId, string LakehouseName) {

// Item create request for lakehouse des not include item definition

var createRequest = new CreateItemRequest(LakehouseName, ItemType.Lakehouse);

// create lakehouse

return CreateItem(WorkspaceId, createRequest);

}

xx

A screenshot of a computer

Description automatically generated

Response from Create Item includes lakehouse id which is required to create notebooks

A screenshot of a computer code

Description automatically generated

### Create a Notebook using the Create Item API

Notebook created calling **CreateItem** with notebook part **CreateLakehouseTables.ipynb**

Notebook code and all other content stored in single JSON file with **ipynb** extension

Fabric also supports calling **CreateItem** with standard Python file with **py** extension

### Create Spark Environments using Create Item

ooeoe

### Configure Workspace Spark Settings

Code used in developer sample application to create lakehouse

### Create a DirectLake Semantic Models on a Lakehouse

# Deploy Solutions using Warehouses and Data Pipelines

# Deploy Solutions using Real-time Intelligence