# Create and Update Workspace Items

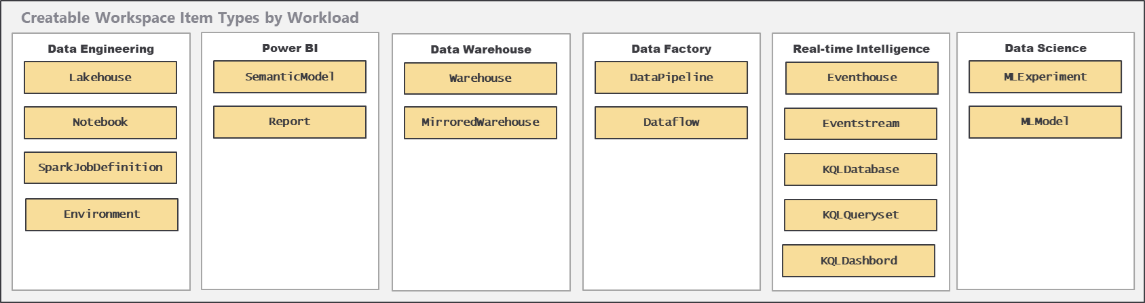
Deploying a Fabric solution typically involves two steps. The first step is creating a new workspace and adding role assignments to control access. The second step involves populating the workspace by creating new workspace items such as lakehouses, notebooks, semantic models and reports. Fortunately, the Fabric platform provides a wide variety of creatable workspace item types which make it possible to design, implement and deploy solutions that acquire, transform, analyze and visualize data.

As a Fabric developer, you must learn a few essential programming techniques used to create workspace items. Furthermore, workspace items are not created in isolation. Therefore, you’ll be required to create workspace items that depend on other workspace items. For example, you must learn how to create a notebook using a technique that binds the notebook to an existing lakehouse. As another example, you might be required to create a Power BI report which gets bound to an existing semantic model. The key point is that you must be able to create a set of workspace items that have been configured to work together in he context of a single Fabric solution.

## Creatable Workspace Item Types

The programming model of the Fabric REST APIs is built upon the generic abstraction of the ***workspace item***. While different types of workspace items will have different properties, there are benefits to creating a common set of properties that exist across every type of workspace item. There are also the benefits to standardizing patterns and best practices for automating the creation and provisioning of all types of workspace items.

The Fabric platform offers a evolving set of creatable workspace item types shown in the following diagram. You can use these creatable types together with the **Create Item** API to create and configure workspace items as part of the Fabric solution deployment process.



This is not an exhaustive list. New createable workspace item types are added to Fabric on a regaular basis.

### Item Definitions

The Fabric REST API programming model introduces the abstraction of the ***item definition*** which used as a mechanism to create and update workspace items. At a high level, an item definition represents a set of system files make up the definition of a workspace item. Each of the system files in an item definition is known as a **part**. While all item definitions are constructed using a common format, each workspace item type defines its own set of parts required to fill out a complete definition.

There are three primary scenarios in which you will program directly with item definitions. First, you can pass an item definition when calling the **Create Item** API. Second, you can retrieve an item definition for existing workspace item by calling the **Get Item Definition** API. Third, you can modify an existing workspace item by passing an item definition when calling the **Update Item Definition** API.

A diagram of a process

Description automatically generated

When calling the **Create Item** API, the item definition is passed in the POST request body as part of the JSON payload. To create a notebook, you can call the **Create Item** API passing an item definition which contains a single system file part name **notebook-content.py**. As shown in the following diagram, an item definition contains a **parts** property collection which contains a set of one or more **part** files.

A screenshot of a computer

Description automatically generated

You should keep in mind that item definitions must be represented in a JSON structure that can be passed across the network in API calls. This leads to an important question. How can you embed the content of a file inside a valid JSON structure? The answer is encode the file contents of item definition parts using Base64 encoding.

In order to create an item definition, you must first convert the contents of each item definition part file into a Base64 encoded format. After that, you can add the encoded file contents into the **payload** property of a part as an ordinary string as shown in the following JSON code listing.

{

"displayName": "notebook1",

"type": "Notebook",

"definition": {

"parts": [

{

"path": "notebook-content.py",

"payload": "{PY\_FILE\_CONTENT\_BASE64\_ENCODED}",

"payloadType": "InlineBase64"

}

]

}

}

Note that each part in an item definition requires three properties which are **path**, **payload** and **payloadType**. Each **part** is added in a **parts** collection which is a property of the **definition**. You have seen that the item definition for a notebook is fairly simple in that it only includes a single file. However, the item definitions for other workspace item types often contain multiple files.

Here’s what can be a bit confusing at first. Some workspace item types support creation and updates using item definitions while others do not. For example, you will use definitions to create some types of workspace items such as notebooks, Spark job definitions, semantic models and reports. However, you will not use an item definition when creating other types of workspace items such as lakehouses and warehouses.

### The Generic Items Endpoints versus Item-specific Endpoints

When you begin to program with Fabric REST APIs, you will find scenarios in which you can choose between two different endpoints used to achieve the same goal. As an example, you can call the **Create Item** API to create different types of workspace items by executing a POST request against the generic **items** endpoint.

https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items

Alternatively, you can create workspace items using one of the item-specific endpoints. For example, you can call the **Create Lakehouse** API to create a lakehouse by executing a POST request against the **lakehouses** endpoint.

https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/lakehouses

Likewise, you can call the **Create Notebook** API by executing a POST request against the **notebooks** endpoint.

https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/notebooks

In many scenarios, it doesn’t really make a difference which of these two endpoints you use as the result will be the same in either case. However, there are a few scenarios in which the item-specific endpoint provides an extra configuration option known as a ***creation payload*** that is not available through the generic **items** endpoint. You will see an example of this ahead in this chapter.

There is another important motivation for preferring item-specific endpoints over the generic **items** endpoint. That is the ability to retrieve item-specific property values. Consider an example of calling an API to discover the property values of a lakehouse. You can call the **Get Item** API to retrieve lakehouse properties by executing a GET request on the following URL.

https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{lakehouseId}

This call to the **Get Item** API will return a JSON result with the properties that exist in across every type of workspace item.

{

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

"type": "Lakehouse",

"displayName": "lakehouse1",

"description": "a sample lakehouse",

"workspaceId": "99999999-9999-9999-9999-9999999999999",

}

As you can see, a call to **Get Item** returns the set of common properties that exist for all workspace item types. A call to **Get Item** doesn’t return any lakehouse-specific properties. Now compare that to a call to **Get Lakehouse** which is executed as a GET request on the **lakehouses** endpoint.

https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/lakehouses/{lakehouseId}

The call to **Get Lakehouse** returns a JSON result with the **properties** property which contains an extended set of lakehouse-specific properties.

{

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

"type": "Lakehouse",

"displayName": "lakehouse1",

"description": " a sample lakehouse",

"workspaceId": "99999999-9999-9999-9999-9999999999999",

"properties": {

"oneLakeTablesPath": "https://onelake.dfs.fabric.microsoft.com/{WORKSPACE\_ID}/{LAKEHOUSE\_ID}/Tables",

"oneLakeFilesPath": "https://onelake.dfs.fabric.microsoft.com/{WORKSPACE\_ID}/{LAKEHOUSE\_ID}/Files",

"sqlEndpointProperties": {

"connectionString": "{LAKEHOUSE\_UNIQUE\_ID}.datawarehouse.fabric.microsoft.com",

"id": "{SQL\_ENDPOINT\_DATABASE\_NAME}",

"provisioningStatus": "Success"

}

}

}

As you can see, these extended lakehouse-specific properties are important as they provide connection information for its SQL endpoint as well as URLs which a developer can use to access OneLake storage associated with the lakehouse using the ADLS Gen2 APIs.

There are common scenarios in Fabric solution deployment in which you will create a lakehouse and then retrieve lakehouse-specific properties. For example, consider a scenario in which you first create a lakehouse and then you need to create a DirectLake semantic model which connects to the lakehouse’s SQL endpoint. In this scenarios, you’ll be required to retrieve the extended lakehouse properties for the SQL endpoint so you can add the SQL connection information into the item definition which you use to create the semantic model.

## Create Workspace Items

Now it’s time to examine programming techniques you can use to create workspace items. In our first example, we’ll examine how to create a new lakehouse using the generic **items** endpoint. Next, we’ll examine how to accomplish the same result using the item-specific **lakehouses** endpoint. This will allow you to see an example of when an item-specific endpoints can provide more functionality than the generic **items** endpoint. After that, we’ll move on to an example creating a notebook which involves an item definition.

### Create Workspace Items using the Generic Items Endpoint

You can call the **Create Item** API by executing a POST request against the generic **items** endpoint.

POST https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items

When calling the **Create Item** API to create a lakehouse, you do not need an item definition. Instead, you just need to pass JSON in the request body which includes two required properties which are **displayName** and **type**. You can optionally pass a third property named **description**.

{

"displayName": "lakehouse1",

"type": "Lakehouse",

"description": "a sample lakehouse"

}

A successful call to **Create Item** will return either **201 Created** or **202 Accepted**. That means some calls to **Create Item** will execute synchronously while others execute asynchronously as long running operations. The details of programming long running operations was discussed earlier in Chapter 3. In the specific case when you are calling **Create Item** to create a lakehouse, a successful call will execute synchronously and return a status code of **201 Created** and a response body with the following JSON result.

{

"id": "11111111-1111-1111-1111-1111111111111",

"type": "Lakehouse",

"displayName": "lakehouse1",

"description": "a sample lakehouse",

"workspaceId": "99999999-9999-9999-9999-9999999999999"

}

Now let’s look at an example of calling the **Create Item** API using the Fabric REST API .NET SDK. First, you must create an object of type **CreateItemRequest** which is used to generate the POST request body. Next, you pass the **CreateItemRequest** object along with the workspace Id in a call to **Core.Items.CreateItem**. It is best to chain the call **CreateItem** together with a call to the **Value** property. The **Value** property abstracts away whether the call to the **Create Item** API is processed synchronous versus asynchronous. The **Value** property returns a strongly-typed **Item** object which makes it possible to retrieve the **Id** property value for the new lakehouse.

CreateItemRequest createLakehouseRequest = new CreateItemRequest("lakehouse1", ItemType.Lakehouse);

Item lakehouse = fabricApiClient.Core.Items.CreateItem(WorkspaceId, createLakehouseRequest).Value;

Guid lakehouseId = lakehouse.Id.Value;

In many deployment scenarios, you will need to record the lakehouse id for later use. For example, you will need to track the lakehouse id if you are deploying a Fabric solution in which you need to create a notebook that is bound to the lakehouse.

### Create Workspace Items using Item-specific Endpoints

You’ve just seen how to create a lakehouse using the **Create Item** API which uses the generic **items** endpoint. Now let’s compare that to creating a workspace item using an item-specific endpoint. In this example, we will call the **Create Lakehouse** API by executing a POST request against the **lakehouses** endpoint.

POST https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/lakehouses

When calling the **Create Lakehouse** API, you must pass JSON in the request body with a required property named **displayName** and an optional property named **description**. Unlike a call to the generic **Create Item** API, you are not required to pass the **type** property because the workspace item type is automatically implied by the **lakehouses** endpoint.

{

"displayName": "lakehouse1",

"description": "a sample lakehouse",

}

Now here is where the **lakehouses** endpoint provides extended functionality not available through the generic **items** endpoint. The lakehouses endpoint support an extended property named **creationPayload** which can be used to create a lakehouse with a featured enabled to support schemas.

{

"displayName": "lakehouse1",

"description": "a sample lakehouse",

"creationPayload": {

"enableSchemas": true

}

}

Currently, there are only a handful of workspace item types that support the **creationPayload** property which allow you to pass item-specific initialization parameters. However, this is expected to change over time as Microsoft continues to evolve item-specific endpoints.

Now let’s look at the C# code used to call the **Create Lakehouse** API using the.NET SDK. First, you must create an object of type **CreateLakehouseRequest**. Next, you pass the **CreateLakehouseRequest** object along with the workspace Id in a call to **Lakehouse.Items.CreateLakehouse**. You can chain the **CreateLakehouse** together with a call to the **Value** property.

CreateLakehouseRequest createRequest = new CreateLakehouseRequest(“lakehouse1”);

createRequest.CreationPayload = new LakehouseCreationPayload(enableSchemas: true);

Item lakehouse = fabricApiClient.Lakehouse.Items.CreateLakehouse(WorkspaceId, createRequest).Value;

Guid lakehouseId = lakehouse.Id.Value;

### Create Workspace Items using an Item Definition

Every type of workspace item that supports item definitions defines what file or files makes up the set of item definition parts.

Now let’s step through creating an item definition to create a notebook.

# METADATA \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# META {

# META "kernel\_info": {

# META "name": "synapse\_pyspark"

# META },

# META "dependencies": {

# META "lakehouse": {

# META "default\_lakehouse": "{LAKEHOUSE\_ID}",

# META "default\_lakehouse\_name": "{LAKEHOUSE\_NAME}",

# META "default\_lakehouse\_workspace\_id": "{WORKSPACE\_ID}",

# META "known\_lakehouses": [

# META {

# META "id": "{LAKEHOUSE\_ID}"

# META }

# META ]

# META }

# META }

# META }

# CELL \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

message = "Hello World"

print(message)

Show base64 encoding of file content

xxx

POST https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/notebooks

Working

{

"displayName": "notebook1",

"type": "Notebook",

"definition": {

"parts": [

{

"path": "notebook-content.py",

"payload": "{PY\_FILE\_CONTENT\_BASE64\_ENCODED}",

"payloadType": "InlineBase64"

}

]

}

}

This chapter covers the essential topic of item definitions. Item

### Construct an Item Definitions using a Non-default Format

Every workspace item type that supports item definitions has a default format. For example, the item definition for notebook has a default format in which notebook content maintained in .py file format. However, you can also create an item definition for a notebook based on the **ipynb** file format where the notebook contents are stored in a JSON file.

When you are creating notebooks, you can choose whether to maintain the template files for notebook contents in either py file format or ipynb file format.

{

"displayName": "notebook1",

"type": "Notebook",

"definition": {

“format”: “ipynb”

"parts": [

{

"path": "notebook-content.py",

"payload": "{IPYNB\_FILE\_CONTENT\_BASE64\_ENCODED}",

"payloadType": "InlineBase64"

}

]

}

}

xxx

## List Workspace Items

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

A screenshot of a computer

Description automatically generated

Call List Items

GET https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items

Returns JSON result with value property containing

{

"value": [

{

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

"displayName": "sales",

"description": "A lakehouse description",

"type": "Lakehouse",

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

},

{

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

"displayName": "Create Lakehouse Tables",

"description": "A notebook description",

"type": "Notebook",

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

}

]

}

Here is C# code using .NET SK

List<Item> items = fabricApiClient.Core.Items.ListItems(WorkspaceId).ToList();

foreach (var item in items) {

Console.WriteLine($"{item.DisplayName} is a {item.Type} with an id of {item.Id}");

}

If the list of items is large enough, the List Items API has the potential to return paginated results. Don’t forget to look for continuation tokens.

### List a Specific Type of Workspace Item

xxx

GET https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items**?type=Lakehouse**

Xxxx

List<Item> notebookItems = fabricApiClient.Core.Items.ListItems(WorkspaceId, type:"Lakehouse").ToList();

foreach (var item in notebookItems) {

Console.WriteLine($"{item.DisplayName} is a notebook with an id of {item.Id}");

}

There is another way to accomplish this by going through the item-specific endpoint.

GET https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/notebooks

X

{

"value": [

{

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

"type": "Lakehouse",

"displayName": "sales",

"description": " A lakehouse description",

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

"properties": {

"oneLakeTablesPath": "https://onelake.dfs.fabric.microsoft.com/{WORKSPACE\_ID}/{LAKEHOUSE\_ID}/Tables",

"oneLakeFilesPath": "https://onelake.dfs.fabric.microsoft.com/{WORKSPACE\_ID}/{LAKEHOUSE\_ID}/Files",

"sqlEndpointProperties": {

"connectionString": "abcdef12345678.datawarehouse.fabric.microsoft.com",

"id": "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",

"provisioningStatus": "Success"

}

}

}

]

}

x

### Query Item-specific Properties

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.

We have just implemented the provisioning flow for an.

1. Create workspace
2. Create lakehouse
3. Create notebook

## 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

## Update Workspace Items

There are two different Fabric REST APIs used to update a workspace item. The **Update Item** API is a simple API which is only used to update the **displayName** property and the **description** property of a workspace item. There is also the **Update Item Definition** API which is more powerful allowing you to update any aspect of a workspace item.

Let start with Update Item.

PATCH https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{itemId}

Cccc

{

"displayName": "Item's New name",

"description": "Item's New description"

}

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

Ssss

POST https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{itemId}/updateDefinition

Ssss

{

"displayName": "notebook1",

"type": "Notebook",

"definition": {

"parts": [

{

"path": "notebook-content.py",

"payload": "{UPDATED\_V2\_PY\_FILE\_CONTENT\_BASE64\_ENCODED}",

"payloadType": "InlineBase64"

}

]

}

}

Note that this JSON body is structured exactly like the JSON body sent to **Create Item**. A key point is that you must include the entire definition.

## 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

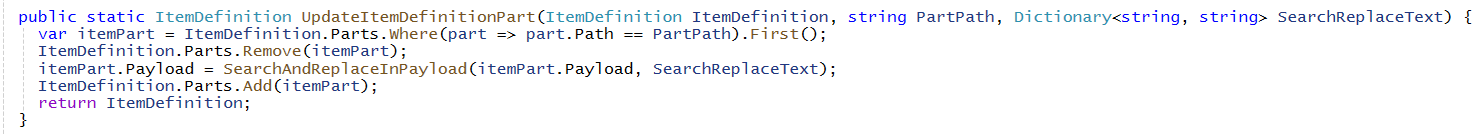
* Create and bind connections
* Refresh import-mode semantic models
* Create dictionary with semantic model Ids to redirect reports from source workspace to clone workspace

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

# Variable Libraries

## Fodder

In earlier chapters, you’ve seen examples of automating the creation of platform items such as workspaces and connection. Platforms items are created and managed as top-level items within the current Entra Id tenant. Creating workspace items is different because they are created exist inside the scope of an existing workspace.

Imagine you have deployed a solution with a workspace which contains workspace items such as a lakehouse and a notebook. When you want to remove the solution, it is not necessary to delete the workspace items individually. Instead, you can just delete the workspace and all the workspace items inside are automatically deleted as well.

Configuring access to workspace items is different than it is for platform items. In the majority of cases, you can configure access at the workspace level by adding workspace roles. Once you add workspace role assignments for a user or service principal, that security principal will have permissions to access all workspace items inside the workspace.