# Chapter 03 - 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 access tokens must be transmitted to the Fabric RESI APIs in 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 will also introduce 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 as well as how to deal with throttling errors in your code.

## 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 API call to the **List Workspace** API succeeds. The next step is to read the JSON content from the body of the response using 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 returned by the **List Workspaces** API which is structured as 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 added the serialization classes to your project, you can deserialize the JSON result returned by the **List Workspaces** API by calling **JsonSerializer.Deserialize**. Note that when calling **JsonSerializer.Deserialize**, you should include the **ListWorkspaceResponse** class in angled brackets to indicate it is the .NET type into which to serialize the JSON result.

// 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 workspace objects.

// 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 a few low-level details just to execute a Fabric REST API call. You have to acquire an access token and include it in the **Authorization** request header. You are required to inspect the HTTP status codes returned by calls to Fabric REST APIs such as **List Workspaces**. Checking the status code allows you to determine whether the call was successful or not. You need some programming technique to read property values from a JSON result. You’ve seen how this can be accomplished by adding custom serialization classes to your project and then using **JsonSerializer.Deserialize** to convert the JSON content into .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**. The .NET 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 have already been discussed 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 you add the **Microsoft.Fabric.API** NuGet package to your project, you will see the Fabric REST API .NET SDK adds a large number of public classes to your project. You will find that quite a few of these public classes have names that end with the word **Client**. The public client classes act as wrappers to Fabric REST API endpoints.

A screenshot of a computer

Description automatically generated

Now that you have seen 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 class that provides access to individual client classes that wrap specific endpoints of the Fabric REST APIs. Here is an example of using the **Workspaces** client to call **List Workspaces**. The object returned from the call to **List Workspaces** has a **ToList** method which returns a standard .NET list.

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. There are other conveniences as well. For example, 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. That’s because the .NET SDK defines its own set of serialization class designed to convert between JSON and strongly-typed .NET objects. For example, the .NET SDK includes a serialization class named **CreateWorkspaceRequest**. You use this class to create a .NET object which can be serialized into JSON which can be sent in the body of a POST request to the **Create Workspace** API.

The .NET SDK defines other serialization classes used to deserialize JSON results returned from 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 defined by the .NET SDK.

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**. Converting between these upper-case and lower-case formats is something the .NET SDK does automically as it serailizes and deserialized JSON.

You already seen several good reasons why you should leverage the .NET SDK when developing with the Fabric REST APIs. However, there are a few more good reasons. The first is that the .NET SDK provides automatic support dealing with paginated results and continuation tokens. Second, the .NET SDK provides automatic support for managing Fabric REST API calls which support the long running operations pattern. Third, the .NET SDK provides built-in support to deal with throttling errors. You will learn about all three topics before the end of this chapter.

## 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 inspecting the JSON returned from 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 caller must make a third call an so on. The sequence of follow-up calls must continue 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. The **ListWorkspaceResponse** serialization class was initially defined with a single property name **value**. Now we need to extend this serialization class by adding two additional 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. Each new paginated result must be appended to the aggregated list named **workspaces**. Once a call returns where **continuationUri** is null, you can assume all 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);

}

Wait a second. Isn’t that the same code as shown earlier? Yes, it is. A .NET SK call to **ListWorkspaces().ToList** 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 about the long running operations pattern which includes monitoring the progress of an asynchronous API call in progress. You must also learn how to retrieve the result of a long running operation after it’s completed. Let’s start with a review which compares synchronous API calls to 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.

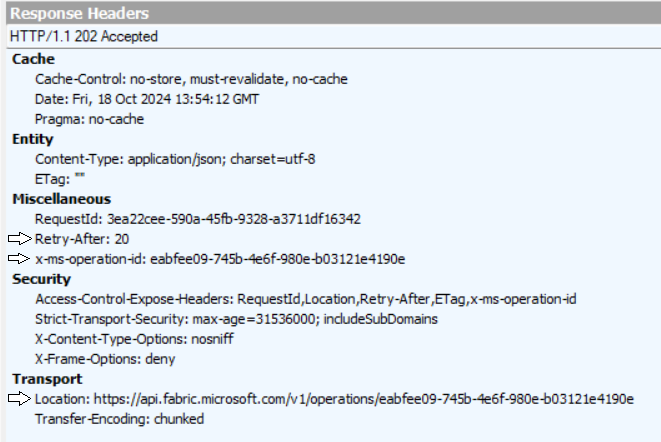
The **Create Item** API is an example of an API that support the long running operations pattern. If you examine the documentation for this API, you will see the response supports returning either **201 Created** or **202 Accepted**. That means the **Create Item** API sometimes runs synchronous returning **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 the **Create Item** API to create a lakehouse might execute synchronously while calling the **Create Item** API 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, an API call that returns **202 Accepted** indicating a long running operation also includes a special set of response headers that provide the essential information you need to monitor the progress and completion of the 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 whether the operator hasn’t started or is still in 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 a JSON result with a structure shown in the following code listing.

{

"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 be set with one of 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 have determined that a long running operation has succeeded, you still have one more task to complete. You must call another API named **Get Operation Result** to retrieve metadata that represents the operation result. For example, after calling the **Create Item** API to create a semantic model, you need to call **Get Operation Result** to get workspace item properties such as the semantic model id.

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**. Examine the following code which uses **HttpClient** to execute a POST request with JSON in the request body.

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 inspect 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 whether 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 more complicated when 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 when handling a **202 Accepted** response 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("x-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 returned in 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 can 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 repeatedly whenever it sees a **status** value other than **Succeeded** or **Failed.** Inside the **while** loop, there is code to wait for a period of time set by the **retryAfter** interval and then to call the **Get Operation Status** API.

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 repeatedly 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 already 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 new workspace item. However, there are also cases where a long running operation does not have a result. Therefore, you must write conditional logic based on whether 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 the operation has an associated result. If the **Location** header isn’t present, 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 the details required for executing API calls which start long running operations, take a moment to review **ExecutePostRequest** with a complete implementation of 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 implement the long running operations pattern without the.NET SDK. If you take this approach, it is recommended you create a generic, reusable function such as **ExecutePostRequest** in your project to eliminate having to reimplement the pattern for long running operation 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 might be surprised at how much easier it is. The code from the previous listing which did not use the .NET SDK can be reduced to a single line of code. The following code listing shows the simplest way to handle the long running operation when calling **CreateItem** and then chaining a call to **Value**.

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;

}

This code listing demonstrates how the.NET SDK is able to build in automatic support for handling long running operations. A .NET SDK call to **Items.CreateItem** includes logic to call the **Create Item** API and then to inspect the HTTP status code returned to determine whether it is **201 Created** or **202 Accepted**.

If a call to the **Create Item** API returns a status code of **202 Accepted**, that indicates the call started a long running operation. The code added by .NET SDK includes conditional logic to pause for the retry-after interval and then to call **Get Operation State** in a loop. There is also logic after the loop completes to call **Get Operation Result** to retrieve and 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 block for as long as the long running operation takes to complete. However, you can add a timeout value when calling **CreateItem** by adding the **timeoutInMinutes** parameter.

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 not completed by the number of minutes specified in the timeout.

## Fabric REST API Throttling

The Fabric REST APIs enforce a throttling scheme to maintain optimal performance and reliability. This throttling enforcement 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 such as the **List Workspaces** API has a throttling limit of 50 API calls per minute. If a single caller attempts to execute more than 50 API calls in a minute, the 51st API call will be rejecting by returning 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** indicating a **Too many requests** error, it will also include the **Retry-After** response header. The value of the **Retry-After** header tells the caller how many seconds to wait before the current throttling time windows ends and the call can be resubmitted.

A screenshot of a computer

Description automatically generated

### Handling Throttling Errors without the .NET SDK

The following code example demonstrates how to deal with a throttling error. When a Fabric REST API call returns a status code of **429** indicating a **Too many requests** error, this code responds by resubmitting the call after waiting for a period of time set by the **Retry-After** response header.

public static string ExecuteGetRequest(string RestUri) {

// 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");

// execute GET request

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

switch (response.StatusCode) {

// handle case where call succeeds

case HttpStatusCode.OK:

return response.Content.ReadAsStringAsync().Result;

// handle case when throttling error occurs

case HttpStatusCode.TooManyRequests:

// pause for retry-interval to wait until throttling window ends

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

Thread.Sleep(retryAfterInteral \* 1000);

// reattempt call the same API

response = client.GetAsync(RestUri).Result;

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

// return JSON result to caller if API call completes successfully

return response.Content.ReadAsStringAsync().Result;

}

else {

// throw error if reattempt call fails

throw new Exception("Error executing GET request");

}

// handle case where initial API call fails

default:

throw new Exception("Error executing GET request");

}

Now we will compare this code listing to code which uses the .NET SDK to handle throttling errors.

### Handling Throttling Errors with the ,NET SDK

As you would expect, the .NET SDK helps by handling throttling errors behind the scenes. Consider the following example of calling the **ListWorkspaces** method using method the **Workspaces** client.

Var workspaces = fabricApiClient.Core.Workspaces.ListWorkspaces().ToList();

The key takeaway is that the .NET SDK automatically adds code similar to what you’ve seen in the previous example. If the .NET SDK call to **ListWorkspaces** experiences a **429** throttling error, there is built-in logic to resubmit the API call a second time after waiting for a time interval set by the **Retry-After** response header.