# Program Workspaces and Capacities

Whether you’re deploying a one-off solution in Fabric or you’re developing a large-scale multitenant application, you need the skills to automate the creation and management of workspaces. This chapter explains how to create and configure workspaces using the Fabric REST APIs. Just as a journey of a thousand miles begins with a single step, the deployment of a Fabric solution begins with the creation of a single workspace.

The chapter begins by reviewing the fundamental role that capacities play in the Fabric environment. The chapter also explains the **why** and **how** of assigning workspaces to capacities during the workspace provisioning process. You’ll learn how to configure authorized access to a workspace by adding workspaces roles for users, groups and service principals. The chapter concludes by examining how to automate the provisioning of workspace identity.

## Assign Workspaces to Fabric Capacities

As a Fabric solution developer, it’s essential you understand the role that capacities play in a Fabric environment. Capacities are used by Fabric as a mechanism to allocate a distinct pool of resources for computation. The size of the capacity determines the amount of computation power available. The size of the capacity can also determine which Fabric features are supported as well as what additional user licensing might be required.

In a Fabric environment, all top-level items such as workspaces and capacities are created inside the scope of an Entra Id tenant. You configure resource allocation in a Fabric environment by assigning workspaces to capacities as part of the workspace provisioning process as shown in the following diagram. Keep in mind you cannot assign a workspace from one Entra Id tenant to a capacity in a different Entra Id tenant.

A diagram of a work space

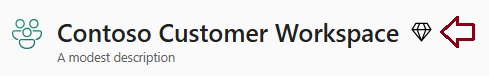
Description automatically generated

Microsoft recommends creating new capacities using Fabric F SKUs. However, it’s also possible to work with capacities that have been created using Power BI P SKUs. Capacities created using F SKU licensing (e.g. F16, F64) are known as **Fabric capacities**. Capacities created through a Power BI P SKU licensing (e.g. P1, P2) are referred to as **premium capacities**. Either type of capacity can be used to develop and deploy Fabric solutions.

There is a one-to-many relationship between capacities and workspaces. A workspace can only be assigned to a single capacity. However, it’s possible to assign multiple workspaces to one capacity. At one extreme, you can assign all your workspaces to one large capacity. At the other extreme, you can assign each workspace to its own dedicated capacity. Most organizations end up somewhere in the middle. With a Fabric environment, you can decide on whatever set of capacities best fits your need to provide the necessary resources to a set of workspaces.

Consider the lonely plight of **Workspace 8** in the prior diagram. This workspace hasn’t been explicitly assigned to either a Fabric capacity or a premium capacity. As it turns out, this type of workspace is automatically assigned to a hidden capacity which is known as the **shared capacity**.

Workspaces which run out of the shared capacity are very limited in terms of functionality and scale. This makes it impractical to develop and test Fabric solutions in your Fabric environment if you don’t have access to a Fabric capacity or premium capacity. In some cases, you can start a Fabric trial which provides a free Fabric capacity for a time period of 60 days. In all case, you want to ensure the workspaces you create are assigned to a Fabric capacity or premium capacities as indicated by the diamond icon to the right of the workspace display name.



Visit the following link to get additional information on Fabric SKUs and Fabric capacity licensing.

* <https://learn.microsoft.com/en-us/fabric/enterprise/licenses>

## Platform Item Types versus Workspace Item Types

The Fabric REST APIs provide support for programming many different types of items. You can separate these programmable item types into two categories which are platform items and workspace items. **Platform items** are created in the scope of the hosting Entra Id tenant. **Workspace items** are created in the scope of a workspace.

A screenshot of a computer

Description automatically generated

Platform items exist as top-level objects within a Fabric environment. Each platform item has its own type of access control list (ACL) which makes it possible to configure access for users, groups and service principals. For example, the ACL for a workspace is a list of **workspace roles**. You configure access to a workspace by adding workspace roles for users, groups and service principals.

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 a workspace roles for a user or service principal, that security principal will have permissions to access all workspace items inside the workspace. For scenarios which require a more granular security configuration, it is also possible to set item-level permissions.

The level of access within a workspace is determined by the type of workspace role. Fabric defines four different types of workspace roles include **Admin**, **Member**, **Contributor** and **Viewer**. You can visit the following link to get additional information and see a table which lists the permissions included with each type of workspace role.

* <https://learn.microsoft.com/en-us/fabric/get-started/roles-workspaces#-workspace-roles>

## Program Capacities

This section on programming capacities is pretty short. That’s because there are really only two things you can do to when programming capacities with the Fabric REST APIs. First, you can call the **List Capacities** API to retrieve the list of the capacities to which the caller has access. Second, you can use the Fabric REST APIs to assign a workspace to a capacity. However, the Fabric REST APIs do not support creating new capacities. However, you can use the Azure Microsoft Fabric REST APIs to automate the creation and management of Fabric capacities.

### List Capacities

You call the **List Capacities** API by executing a GET request to a URI with **/capacities** appended to the base URI.

GET https://api.fabric.microsoft.com/v1/capacities

In Microsoft’s public cloud, the Fabric REST APIs have a base URI of **https://api.fabric.microsoft.com/v1**. This is the base URL that will be used for code samples in this guidance document. However, you should understand that the base URI for the Fabric REST APIs will be slightly different in sovereign clouds and government clouds.

A successful call to the **List Capacities** API returns an HTTP status code of **200 OK** and a JSON result which contains the list of capacities in the current Entra Id tenant accessible to the caller as seen in the following listing.

{

"value": [

{

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

"displayName": "My Little Capacity",

"sku": "F4",

"region": "West Central US",

"state": "Active"

},

{

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

"displayName": "My Big Capacity",

"sku": "F64",

"region": "West Central US",

"state": " Active"

}

]

}

Now let’s look at a simple example of calling the **List Capacities** API using the .NET SDK. The following code listing demonstrates how to chain a call to **ToList** after the call to **ListCapacities**. This makes it possible to retrieve the capacities list as a standard .NET collection which you can enumerate through using a **foreach** loop.

// create .NET SDK Client object

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

FabricClient fabricApiClient = new FabricClient(accessToken);

// Call List Capacities API

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

// enumerate through capacities list

foreach (var capacity in capacities) {

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

}

Remember Fabric REST APIs such as **List Capacities** have the potential to return paginated results if the requested list is large enough. This isn’t an issue when you’re using the .NET SDK which automatically handles paginated results for you. If you are calling the **List Capacities** API without using the .NET SDK, you should add extra logic to inspect results for the presence of continuation tokens and to implement the pattern for paginated results as discussed earlier in the **Fabric REST API Fundamentals** chapter.

It’s possible that a call to **List Capacities** will succeed but still return an empty list of capacities. That means your code is running under an identity of a user or service principal which has no access to any premium capacities. This is a problem within your Fabric environment you will need to resolve. That’s because your code needs access to a Fabric capacity or premium capacity to which it can assign workspaces as you begin to develop and test Fabric solution deployment.

### Create Fabric Capacities using Code

The Fabric REST API support for programming capacities is limited to the **List Capacities** API and a few other APIs used to assign workspaces to capacities. Assigning workspaces to capacities using the Fabric REST APIs will be covered later in this chapter. However, the Fabric REST APIs do not provide support for creating capacities.

You can use the **Azure Microsoft Fabric REST API** to automate the creation, configuration and management of Fabric capacities. This is the same REST API used to createand manageother types of Azure resources such as Managed Identities, ADLS Gen2 Storage containers, Key Vaults and Azure SQL Databases.

The Azure Microsoft Fabric RESTAPIs provide endpoints to list, create and manage Fabric capacities. The following table lists the Azure Microsoft Fabric RESTAPI operations for programing Fabric capacities.

|  |  |  |
| --- | --- | --- |
| Check Name Availability | List By Resource Group | Resume |
| Create Or Update | List By Subscription | Suspend |
| Delete | List Skus | Update |
| Get | List Skus For Capacity |  |

This guidance document will not examine creating and managing capacities using Azure Microsoft Fabric REST API. Reader interested in automating the creation of Fabric capacities can visit the following URL for more information.

* [Fabric Capacities - REST API (Azure Fabric) | Microsoft Learn](https://learn.microsoft.com/en-us/rest/api/microsoftfabric/fabric-capacities?view=rest-microsoftfabric-2023-11-01)

## Program Workspaces

Now it’s time to examine how to program workspaces using the Fabric REST APIs. The section will begin with a review of the **List Workspaces** API and then move onto to other workspace APIs which support CRUD operations. In addition to learning to use APIs for creating, updating and deleting workspaces, you will see how to use other APIs to use to assign a workspace to a Fabric capacity, to add workspace roles and to provision workspace identity.

### List Workspaces

You call the **List Workspaces** API by executing a GET request to a URI parsed together with the following format.

GET https://api.fabric.microsoft.com/v1/workspaces

A successful call to the **List Workspaces** API returns an HTTP status code of **200 OK** and a JSON result with a value property containing the list of workspaces accessible to the caller in the current Entra Id tenant.

{

"value": [

{

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

"displayName": "Workspace 1",

"description": "",

"type": "Workspace",

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

}

]

}

If the requested list of workspaces is large enough, a call to the **List Workspaces** API will returned paginated results. If you are not using the .NET SDK, your code should handle continuation tokens by implemenitng the pagainted results pattern.

Now let’s look at a simple example of calling the **List Workspaces** API using the .NET SDK. The following code listing demonstrates how to chain a call to **ToList** after the call to **ListWorkspaces**. This makes it possible to retrieve the workspaces list as a standard .NET collection which you can enumerate through using a **foreach** loop.

// create .NET SDK Client object

string accessToken = EntraIdTokenManager.GetFabricAccessToken();

FabricClient fabricApiClient = new FabricClient(accessToken);

// call the List Workspaces API

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

// enumerate through list of workspaces

foreach (var workspace in workspaces) {

Console.WriteLine(workspace.DisplayName);

}

### Create a Workspace

You call the **Create Workspace** API to create new workspaces by executing a POST to a URI to the base URI and **/workspaces**.

POST https://api.fabric.microsoft.com/v1/workspaces

You must include JSON in the request body with the **displayName** property. In addition to the **displayName** property which is required, you can optionally add the **description** property and the **capacityId** property.

{

"displayName": "Contoso Customer Workspace",

"description": "A modest description",

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

}

A successful call to the **Create Workspace** API runs synchronously and returns a **201 Created** HTTP status code. A successful response will also include JSON in the response body with several properties including the **id** property for the new workspace.

{

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

"displayName": "Contoso Customer Workspace",

"description": "A modest description",

"type": "Workspace",

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

}

Now let’s look at an example of calling the **Create Workspace** API using the .NET SDK. The first step is to create an object of type **CreateWorkspaceRequest** and initialize this object with property values to populate the POST request body. Next, you call the **CreateWorkspace** method exposed by the **Workspaces** client passing the **CreateWorkspaceRequest** object.

// acquire data for create workspace request

string workspaceName = "Contoso Customer Workspace";

string workspaceDescription = "Workspace for Contoso Customer Tenant";

Guid capacityId = new Guid("99999999-9999-9999-9999-999999999999");

// prepare create workspace request

var createRequest = new CreateWorkspaceRequest(workspaceName);

createRequest.Description = workspaceDescription;

createRequest.CapacityId = capacityId;

// call Create Workspace API

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

// capture Id of new workspace

Guid workspaceId = workspace.Id;

In the previous code listing, you can observe there is a chained call to the **Value** property after the call to **CreateWorkspace**. The **Value** property returns a strongly-typed .NET object with properties for the new workspace that has just been created. This makes it possible to capture the workspace **Id** property which is an essential step as you begin to create workspace items inside a new workspace.

There is an important restriction in Fabric that prevents two workspaces from having the same display name within the scope of an Entra Id tenant. If you attempt to create a workspace using the same display name as another existing workspace, the **Create Workspace** API will return a **409 Conflict** error with a message.

If you pass an invalid capacity Id value in a call to Create Workspace, the call will fail with a **404 Not Found** error. This can occur if you pass a capacity Id value that does not reference an existing Fabric capacity in the same Entra Id tenant. This can also occur if the capacity Id references a valid capacity but the caller has not be granted the permissions required to assign workspaces to the capacity.

### Assign a Workspace to a Capacity

You’ve seen it’s possible to assign a new workspace to a Fabric capacity when calling the **Create Workspace** API. This is the best practice in most scenarios as capacity assignment becomes part of the workspace provisioning process. However, there are other scenarios where you need to assign a workspace to a capacity after the workspaces has already been created.

Consider a scenario where a workspace was assigned to one capacity when it was created and now you’d like to reassign the workspace to a different capacity. This can be accomplished by calling the **Workspaces -** **Assign To Capacity** API. To call the **Assign To Capacity** API, you execute a POST request to a URI parsed together with the following URI format.

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

When calling the **Assign To Capacity** API, you must pass JSON in the body of the POST request with the **capacityId** property.

{

"capacityId": "88888888-8888-8888-8888-888888888888"

}

A successful call to the **Assign to Capacity** API returns **202 Accepted** indicating the call will be processed as a long running operation. The following C# code demonstrates using the .NET SDK to call the **Assign To Capacity** API. As shown in the following code listing, you must first initialize an **AssignWorkspaceToCapacityRequest** object with the capacity Id. Then you pass that object as a parameter in the call to **AssignToCapacity** along with another parameter with the workspace Id.

var assignmentRequest = new AssignWorkspaceToCapacityRequest(“88888888-8888-8888-8888-888888888888”);

fabricApiClient.Core.Workspaces.AssignToCapacity(workspaceId, assignmentRequest);

### Unassign a Workspace from a Capacity

There could also be a scenario when you want to unassign a workspace from a capacity. The **Unassign From Capacity** API can be used to accomplish this task by you executing a POST request to a URI parsed together with the following URI format.

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

A successful call to the **Unassign From Capacity** API returns **202 Accepted** indicating the call will be processed as a long running operation. The following C# code demonstrates using the .NET SDK to call the **Unassign From Capacity** API.

fabricApiClient.Core.Workspaces.UnassignFromCapacity(workspaceId);

Remember that a successful call to the **Unassign From Capacity** API will assign the workspace to the shared capacity which severely limits the functionality available when deploying Fabric solutions.

### Get Workspace Information

While a call to the **List Workspace** API retrieves information about multiple workspaces at once, the **Get Workspace** API to retrieve more detailed information about a single workspace. To call the **Get Workspace** API, you execute a GET request to a URI parsed together with the following URI format.

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

A successful call to **Get Workspace** returns a **200 OK** status code and an JSON result with extended workspace metadata like the JSON shown in the following listing.

{

"id": "12341234-1234-1234-1234-123412341234”,

"displayName": "Contoso Customer Workspace",

"description": "Workspace for Contoso Customer Tenant",

"type": "Workspace",

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

"capacityAssignmentProgress": "Completed",

"capacityRegion": "East US",

"workspaceIdentity": {

"applicationId": " aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa",

"servicePrincipalId": "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb"

},

"oneLakeEndpoints": {

"blobEndpoint": "https://eastus-onelake.blob.fabric.microsoft.com",

"dfsEndpoint": "https://eastus-onelake.dfs.fabric.microsoft.com"

}

}

You can observe that the JSON result returned by **Get Workspace** includes additional properties not returned by **List Workspaces**. For example, there is information about workspace identity, OneLake endpoints and the geographical region of the capacity to which the workspace has been assigned. The following C# example demonstrates how to call the **Get Workspace** API using the .NET SDK and capturing information about the capacity’s region.

WorkspaceInfo workspaceInfo = fabricApiClient.Core.Workspaces.GetWorkspace(workspaceId);

string capacityRegion = workspaceInfo.CapacityRegion.Value.ToString();

You can observe that a call to **GetWorkspace** returns a **WorkspaceInfo** object which provides strongly-typed access to workspace properties returned by the **Get Workspace** API such as the capacity region.

### Update a Workspace

You can update the display name and description of an existing workspace using the **Update Workspace** API. You call the **Update Workspace** API by executing a PATCH request to a URI parsed together with the following URI format.

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

You must include JSON in the request body with the **displayName** property and/or the **description** property.

{

"displayName": "Workspace Display Name V2",

"description": "A much better description"

}

A successful call to **Update Workspace** returns **200 OK** and the following JSON result.

{

"id": "33bae707-5fe7-4352-89bd-061a1318b60a",

"displayName": "Workspace Display Name V2",

"description": "A much better description",

"type": "Workspace"

}

You call **Update Workspace** using the .NET SDK by passing an **UpdateWorkspaceRequest** to the **UpdateWorkspace** method.

var updateReqiest = new UpdateWorkspaceRequest {

DisplayName = "Workspace display name V2",

Description = "A much better description"

};

fabricApiClient.Core.Workspaces.UpdateWorkspace(workspaceId, updateReqiest);

### Delete a Workspace

You can call the **Delete Workspace** API by executing a DELETE request to a URI parsed together with the following URI format.

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

If the call to **Delete Workspace** succeeds, it will return a HTTP status code of **OK 200**.

The following code demonstrates calling the **Delete Workspace** API using the .NET SDK.

fabricApiClient.Core.Workspaces.DeleteWorkspace(workspaceId);

### Build a Workflow for Testing Solution Deployment

Now that you have seen how the Fabric REST APIs provide standard CRUD functionality for workspaces, it’s time to combine these APIs into a useful workflow process for testing solution deployment. Consider a scenario where you’re testing the deployment of a Fabric solution composed of workspace items. Each time you conduct a deployment test, you should create a new workspace. That means your code must delete the workspace from the previous test before you can create another workspace with the same display name.

Examine the following code listing which demonstrates a workflow used to test Fabric solution deployment. The workflow begins by calling the **List Workspace** API so it can compare the display name for the new workspace against the display names of all existing workspaces. If the workflow finds an existing workspace with the same display name, it deletes that workspace before moving on to create a new workspace for the current test. Each time this workflow runs, it deletes the workspace from the previous test so it can then create a new workspace with the same display name.

string workspaceName = "Contoso Customer Workspace";

string workspaceDescription = "Contoso Customer Workspace";

Guid premiumCapcityId = new Guid("99999999-9999-9999-9999-999999999999");

var createRequest = new CreateWorkspaceRequest(workspaceName);

createRequest.Description = workspaceDescription;

createRequest.CapacityId = premiumCapcityId;

// check to see if another workspace already exists wth same display name

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

foreach (var workspace in workspaces) {

if (workspace.DisplayName.Equals(workspaceDisplayName)) {

// if workspace exists, delete it and break out of foreach loop

fabricApiClient.Core.Workspaces.DeleteWorkspace(workspace.Id);

break;

}

}

// create workspace for testing

var createRequest = new CreateWorkspaceRequest(workspaceDisplayName);

Workspace newWorkspace = fabricApiClient.Core.Workspaces.CreateWorkspace(createRequest);

## Add Workspace Role Assignments

An essential aspect of the workspace provisioning process is configuring permissions for users, groups and service principals. This is accomplished by adding **workspace role assignments**. When you add a workspace role assignment, you can choose between four possible role types making it possible to configure different access levels from read-only access all the way up to admin-level access.

You add workspace role assignments using the **Add Workspace Role Assignment** API. To call this API, you execute a POST request to a URI parsed together with the following URI format which includes the workspace id followed by **/roleAssignments**.

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

The body of the POST request must contain JSON with the **principal** property and the **role** property. The **principal** property must include a **type** property which can be set to **User**, **Group** or **ServicePrincipal**. The **principal** has also has an **id** property which references an existing object Id from Entra Id for a user, group or service principal. The **role** property can be set to **Admin**, **Member**, **Contributor** or **Viewer**.

{

"principal": {

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

"type": "User"

},

"role": "Member"

}

One thing to keep in mind is that you cannot use a user’s email address when adding a workspace role assignments. You must use object Id for the user’s account as it’s defined in Entra Id.

A successful call to the **Add Workspace Role Assignment** returns **201 Accepted** and a JSON result as shown in following listing.

{

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

"principal": {

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

"type": "User"

},

"role": "Member"

}

You can see from the JSON response in the previosu listing that the newly-added workspace role assignment has a top-level **id** property. However, Fabric does not create a new GUID-based id to track the workspace role assignment. Instead, the top-level **id** property has the same value as the **id** property inside the **principal** property. They key point is that Fabric uses object Id that references the user, group or service principal from Entra Id.

The fact that Fabric uses the Entra object id as the id for each workspace role assignment has an important implication. That is you can only create one workspace role assignment for each user, group or service principal per workspace. Imagine a scenario where you’ve already added a **Viewer** role assignment for a specific user. After that, you try to add a second role assignment for the same user based on the **Member** role type. The second API call will fail with an error message indicating the provided principal already has a role assigned in the workspace.

The key point is that you cannot update an existing workspace role assignment. Instead, you must delete a workspace role assignment and then recreate it to change a user, group or service principal from one type of role assignment to another.

If you examine the documentartion for the **Add Workspace Role Assignment** API, you can see that **ServicePrincipalProfile** is allowed as a forth type of security principal. You should understand the **principal** type of **ServicePrincipalProfile** is included to support backwards compatibility with the Power BI REST API. If you haven’t heard of service principal profiles, don’t worry abou them as they are not part of the Fabric REST API programming model.

### Add a Workspace Role Assignment for a User

Let’s create a generic method which uses the Fabric REST API .NET SDK to add a workspace role assignment for a user.

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);

}

Now you can test the **AddUserAsWorkspaceMemeber** method by calling it to configure access for two users.

Guid TestUser1Id = new Guid(AppSettings.TestUser1Id);

Guid TestUser2Id = new Guid(AppSettings.TestUser2Id);

AddUserAsWorkspaceMemeber(workspaceId, TestUser1Id, WorkspaceRole.Admin);

AddUserAsWorkspaceMemeber(workspaceId, TestUser2Id, WorkspaceRole.Viewer);

### Add a Workspace Role Assignment for a Group

Now let’s create a second generic method which uses the.NET SDK to add a workspace role assignment for a Entra Id group.

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);

}

Now you can test the **AddGroupAsWorkspaceMemeber** method by calling it to configure access for an Entra Id group..

Guid TestADGroup1 = new Guid(AppSettings.TestADGroup1);

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

### Add a Workspace Role Assignment for a Service Principal

Now let’s create a third generic method to add a workspace role assignment for a service principal.

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);

}

Now you can test the **AddServicePrincipalAsWorkspaceMemeber** method by calling it to configure access for a service principal.

Guid TestServicePrincipal = new Guid(AppSettings.ServicePrincipalObjectId);

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

If you are testing Fabric solution deploy using a service principal, it can be very helpful to add an **Admin** workspace role assignment for your own Entra Id user account. This makes it possible for you to inspect workspaces in the Fabric User Experience. This can be very helpful as you are troubleshooting and debugging.

### List Workspace Role Assignments

If you want to inspect the role assignments that have been added to an existing workspace, you can use the **List Workspace Role Assignments** API. You can call this API by executing a GET request to a URI parsed together with the following URI format.

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

A successful call to **List Workspace Role Assignments** returns a status code of **200 OK** and a JSON result like the one shown in the following listing.

{

"value": [

{

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

"principal": {

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

"displayName": "John Dough",

"type": "User",

"userDetails": { "userPrincipalName": "john@company1.com" }

},

"role": "Admin"

},

{

"id": "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb",

"principal": {

"id": "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb",

"displayName": "Report Customizers",

"type": "Group",

"groupDetails": {

"groupType": "SecurityGroup",

"email": null

}

},

"role": "Member"

},

{

"id": "cccccccc-cccc-cccc-cccc-cccccccccccc",

"principal": {

"id": "cccccccc-cccc-cccc-cccc-cccccccccccc",

"displayName": "Fabric ISV Playbook Application",

"type": "ServicePrincipal",

"servicePrincipalDetails": { "aadAppId": "11111111-1111-1111-1111-111111111111" }

},

"role": "Admin"

}

]

}

If you examine the JSON returned by the **List Workspace Role Assignments** API, you can see it provides different information depending on the type of principal. For example, the JSON for a principal with **type** of **User** also includes a **userDetails** property with an inner **userPrincipalName** property.

"principal": {

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

"displayName": "John Dough",

"type": "User",

"userDetails": {

"userPrincipalName": "john@company1.com"

}

}

The JSON for a principal with **type** of **Group** also includes a **groupDetails** property which contains a **groupType** property and **email** property.

"principal": {

"id": "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb",

"displayName": "Report Customizers",

"type": "Group",

"groupDetails": {

"groupType": "SecurityGroup",

"email": null

}

}

The JSON for a principal with **type** of **ServicePrincipal** also includes a **servicePrincipalDetails** property which contains a **aadAppId** property.

"principal": {

"id": "cccccccc-cccc-cccc-cccc-cccccccccccc",

"displayName": "Fabric ISV Playbook Application",

"type": "ServicePrincipal",

"servicePrincipalDetails": {

"aadAppId": "11111111-1111-1111-1111-111111111111"

}

}

The following C# code demonstrates calling the **List Workspace Role Assignments** API using the .NET SDK and enumerating through the list of role assignments for a specific workspace.

var roleAssignments = fabricApiClient.Core.Workspaces.ListWorkspaceRoleAssignments(WorkspaceId);

foreach (var roleAssignment in roleAssignments) {

Console.WriteLine(roleAssignment.Principal.DisplayName);

}

### Delete a Workspace Role Assignment

You can call the **Delete Workspace Role Assignment** API by executing a DELETE request to a URI parsed together with the following URI format.

DELETE https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/roleAssignments/{workspaceRoleAssignmentId}

If the call to **Delete Workspace Role Assignment** succeeds, it will return a HTTP status code of **OK 200**.

The following code demonstrates calling the **Delete Workspace Role Assignment** API using the .NET SDK.

Guid WorkspaceId = new Guid("11111111-1111-1111-1111-111111111111");

Guid RoleAssignmentId = new Guid("bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb");

fabricApiClient.Core.Workspaces.DeleteWorkspaceRoleAssignment(WorkspaceId, RoleAssignmentId);

## Program Workspace Identity

Fabric supports a feature know as **workspace identity** that offers significant value to professional developers. When the workspace identity feature is enabled for a specific workspace, the Fabric environment automatically creates a service principal behind the scenes in the Entra Id Service. This service principal is created with the same display name as the workspace itself. Since the service principal is created as an Entra Id identity, you can configure access for the workspace identity just like you’d configure access to any other user or service principal within the scope of an Entra Id tenant.

The workspace identity feature allows a developer to create outbound connections used by workspace items such as OneLake shortcuts, data pipelines and semantic models. When a connection is established, the Fabric environment handles authentication for the service principal and access token acquisition behind the scenes. The workspace identity feature also follows security best practice by eliminating any need for a custom application to manage secret credentials for a service principal such as a client secret or a client certificate.

The workspace identity feature supports creating connections to datasources secured by Entra Id such as ADLS Gen2 storage containers and Azure SQL. The workspace identity feature also provides the foundation for another security feature known as **trusted workspace access**. The following chapter titled **Program Connections** will go into the details of creating connections based in workspace identity. This chapter will just focus on provisioning workspace identity as part of the workspace provisioning process.

If you have never experimented with workspace identity feature, you can configure it by hand in the **Workspace identity** tab of the **Workspace settings** pane by clicking the **+ Workspace identity** button. If you click this button, it starts the provisioning process for the workspace identity by creating a new service principal in the Entra Id Service.

A screenshot of a computer

Description automatically generated

Note that workspace identity requires a Fabric capacity (F SKU) or a Fabric trial capacity (FT SKU). You cannot enable workspace identity in a workspace associated with the shared capacity or a Premium capacity created with a P SKU such as a P1.

Once the workspace identity provisioning process is completed, the **Workspace identity** tab of the **Workspace settings** pane displays an ID property which is the Entra object Id for the service principal. Note that the workspace identity provisioning process automatically configures the service principal with access to the workspace by adding a **Contributor** workspace role assignment for it.

A screenshot of a computer

Description automatically generated

Now that you seen how to provision workspace identity by hand, let’s move on to discuss how to accomplish the same goal through the Fabric REST APIs.

### Provision Workspace Identity

The Fabric REST APIs provide the **Provision Identity** API which can be used to enable the workspace identity feature for a specific workspace**.** To call the **Provision Identity** API, you execute a POST request to a URI parsed together with the following URI format.

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

A successful call to the **Provision Identity** API returns an HTTP status code of **202 Accepted** indicating that the API call will be processed as a long running operation. If you are not using the Fabric REST API .NET SDK, you need to write extra code to monitor the progress of the provisioning process as discussed in the **Fabric REST API Fundamentals** chapter. However, the .NET SDK makes it simple to provision workspace identity. When you call the **ProvisionIdentity** method on the **Workspaces** client, that call will block until the provisioning process is completed.

Guid workspaceId = new Guid(“11111111-1111-1111-1111-111111111111”);

fabricApiClient.Core.Workspaces.ProvisionIdentity(workspaceId);

When deploying Fabric solutions which leverage the workspace identity feature, it makes sense to call the **Provision Identity** API at the tail end of the workspace provisioning process as demonstrated in the following code listing.

// acquire data for create workspace request

string workspaceName = "Contoso Customer Workspace";

Guid capacityId = new Guid("99999999-9999-9999-9999-999999999999");

// prepare create workspace request

var createRequest = new CreateWorkspaceRequest(workspaceName);

createRequest.CapacityId = capacityId;

// create Workspace

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

// provision workspace identity

fabricApiClient.Core.Workspaces.ProvisionIdentity(workspace.Id);

// get workspace info

WorkspaceInfo workspaceInfo = fabricApiClient.Core.Workspaces.GetWorkspace(workspace.Id);

// get service principal object Id for workspace identity

string servicePrincipalIdForWorkspace = workspaceInfo.WorkspaceIdentity.ServicePrincipalId;

### Deprovision Workspace Identity

You can call the **Deprovision Workspace** API to disable the workspace identity feature for specific workspace. Note that a call to **Deprovision Workspace** will also have the effect of deleting the service principal from Entra Id. To call the **Deprovision Workspace** API, you execute a POST request to a URI parsed together with the following URI format.

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

When you call the API to deprovision workspace identity, the request is processed as a long running operation.

A successful call to the **Deprovision Identity** API returns an HTTP status code of **202 Accepted** indicating that the API call will be processed as a long running operation. Here is a simple example of calling the **Deprovision Identity** API using the .NET SDK.

Guid workspaceId = new Guid(“11111111-1111-1111-1111-111111111111”);

fabricApiClient.Core.Workspaces.DeprovisionIdentity(workspaceId);