Chapter 7: Querying Digital Twins

From reading this book, you should now have mastered managing digital twin models and their relationships. In this chapter, we will explain the query language and how to query the data from these models and their relationships. We will start by setting up a usable set demo graph in an Azure Digital Twins instance. After that, we will execute several different types of queries using Azure Digital Twins Explorer and discuss the results. Finally, we will examine several examples of using the .NET SDK to query using C# code.

In this chapter, we will learn how to query and filter models and relationships of an Azure Digital Twins instance using Azure Digital Twins Explorer, and through .NET code.

The following sections can be found in this chapter:

- Setting up a demo graph
- Basic querying
- Querying by model
- Querying relationships
- Filtering results
- Querying using code
- Querying asynchronous calls using code

Technical requirements

We will continue in the .NET console application using the .NET SDK to further build up Azure Digital Twins instances using models. Azure Digital Twins Explorer will be used to view the results of the .NET calls made by the console application.

Setting up a demo graph

We require an example graph of several digital twins with relationships to understand how we can query digital twins. Therefore, we start by setting up a demo graph to support further steps.

Go to the following GitHub URL: **https://github.com/PacktPublishing/Hands-on-Azure-Digital-Twins**. Then, copy the contents of `/SmartBuildingConsoleApp/Models/chapter7` into your project. This contains several model files and an Excel file. The result should be similar to the following figure:

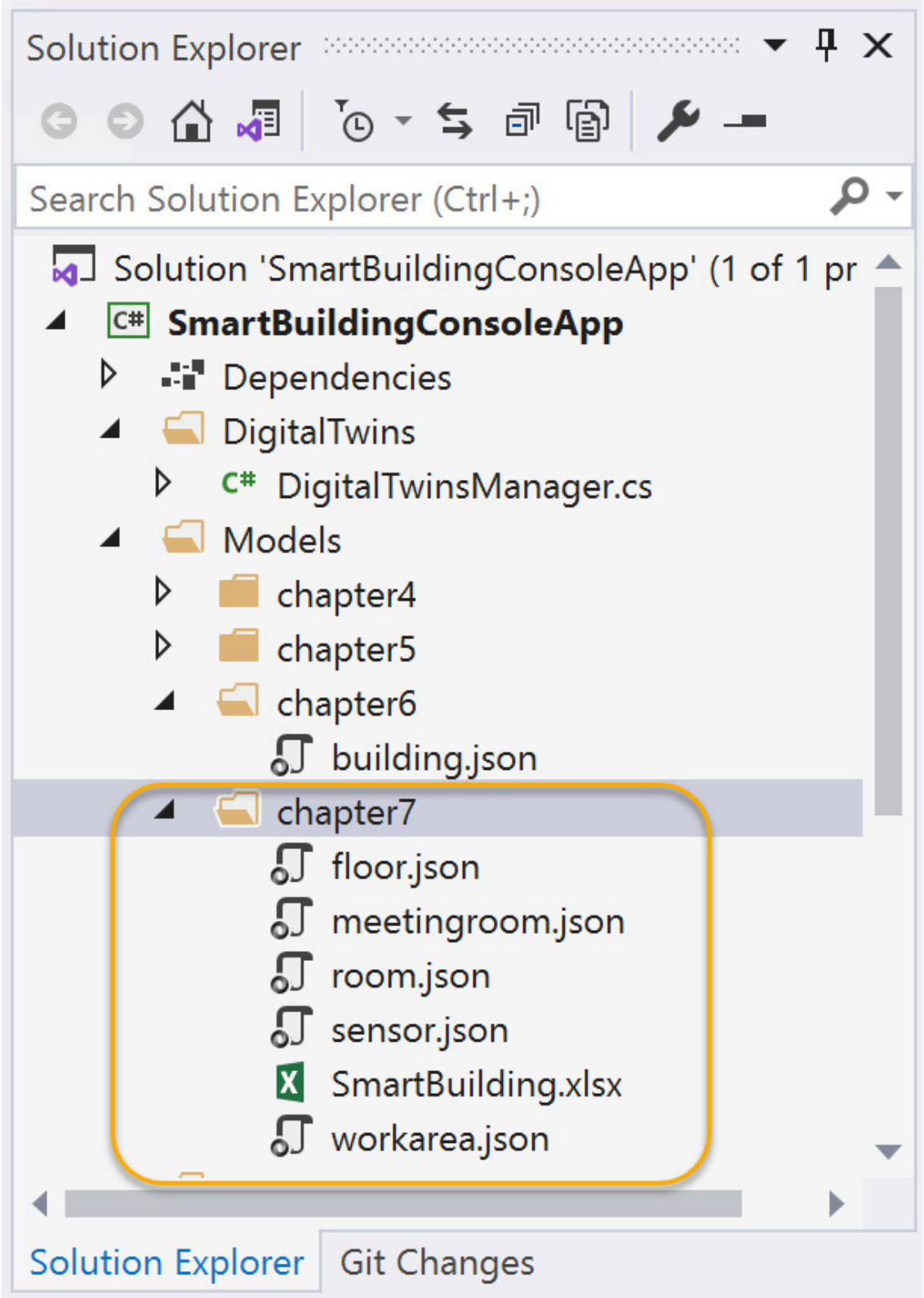

Figure 7.1 – The result after copying the contents of the chapter7 folder

We need to modify several existing models with additional properties in order to build our demo graph. In ***Chapter 4***, *Understanding Models*, you learned how to manage models. Models can be removed and added via Azure Digital Twins Explorer. Execute the following steps to replace the models:

1. Open **Azure Digital Twins Explorer**.

2. In the model view of Azure Digital Twins Explorer, delete the models in the following order: `Meetingroom`, `Workarea`, `Room`, `Sensor`, `Floor`. The `Meetingroom` and `Workarea` models must be removed first before the `Room` model is removed since they are both derived from `Room`.
3. Add the models in the following order via the model view of Azure Digital Twins Explorer: `Floor`, `Sensor`, `Room`, `Workarea`, `Meetingroom`. Use the corresponding JSON files from the `chapter7` folder to do this.

The next step is importing the demo graph into our Azure Digital Twins instance, using Azure Digital Twins Explorer. This is shown in *Figure 7.2*. Execute the following steps:

1. Click on the **Import Graph** button.
2. Select the Excel `SmartBuilding.xlsx` file.
3. Click the **Open** button to open the demo graph:

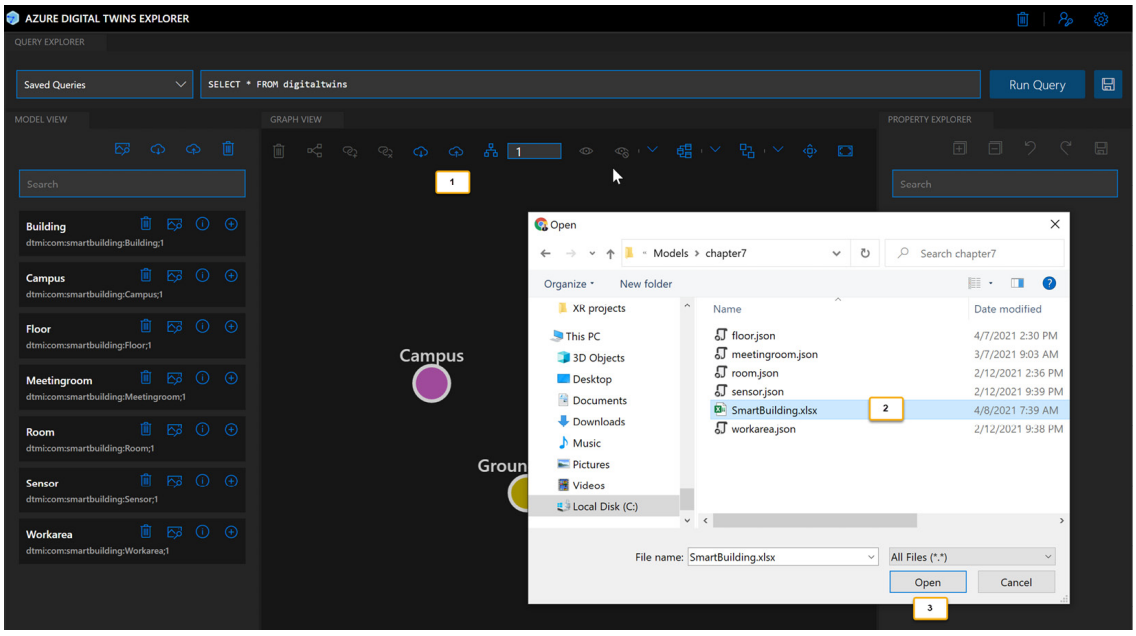

Figure 7.2 – Importing our demo graph data

The demo graph is loaded into the graph view as a preview, as shown in *Figure 7.3*. Click the **Save** button to store the demo graph into the Azure Digital Twins instance:

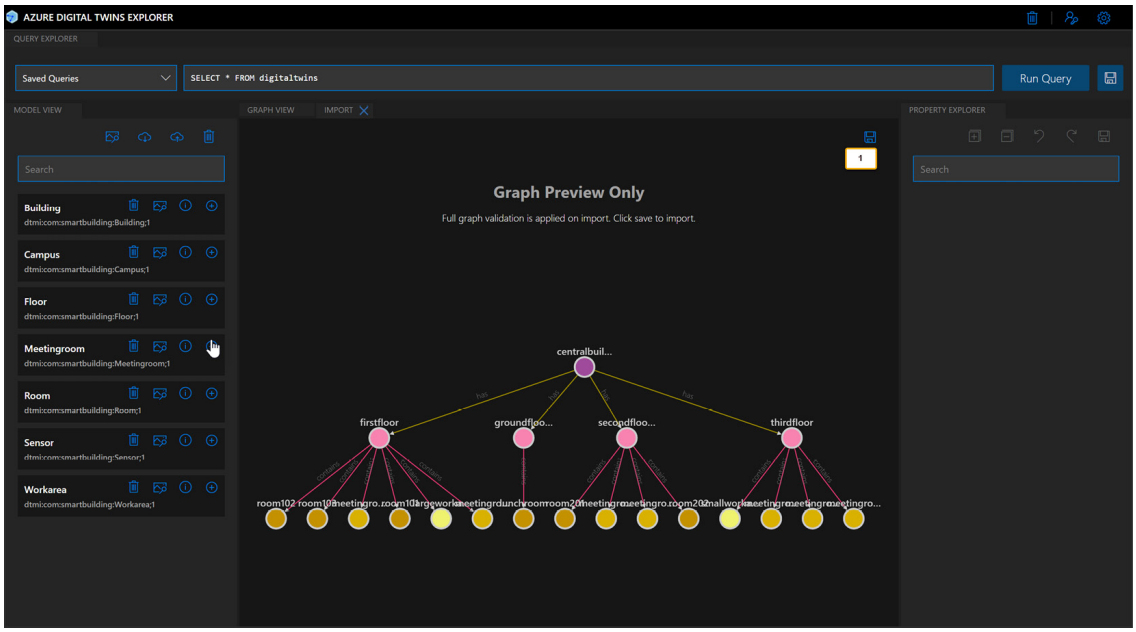

Figure 7.3 – Graph preview of the demo graph

Azure Digital Twins Explorer will show an **Import Successful** dialog after the demo graph is imported, as shown in *Figure 7.4*. It will also show how many digital twins and relationships are imported. Click the **Close** button to continue, as shown in the following screenshot:

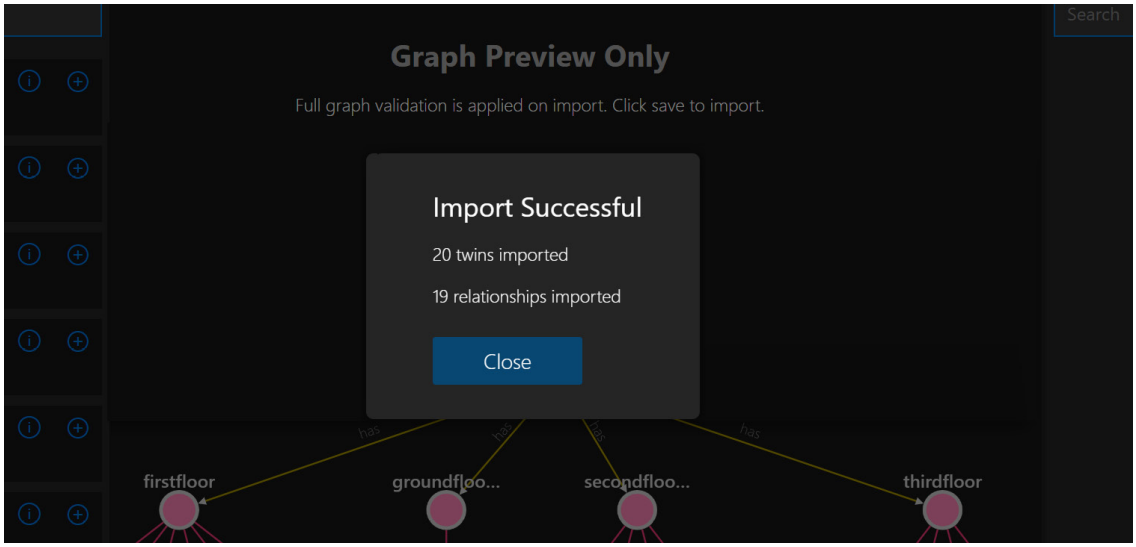

Figure 7.4 – The Import Successful dialog shows the result of the import

Azure Digital Twins Explorer still shows the result of our work from the previous chapters. An import does not replace the contents of the Azure Digital Twins instance. If we were to run the import again, we would get an additional structure with the same names, but with different underlying IDs:

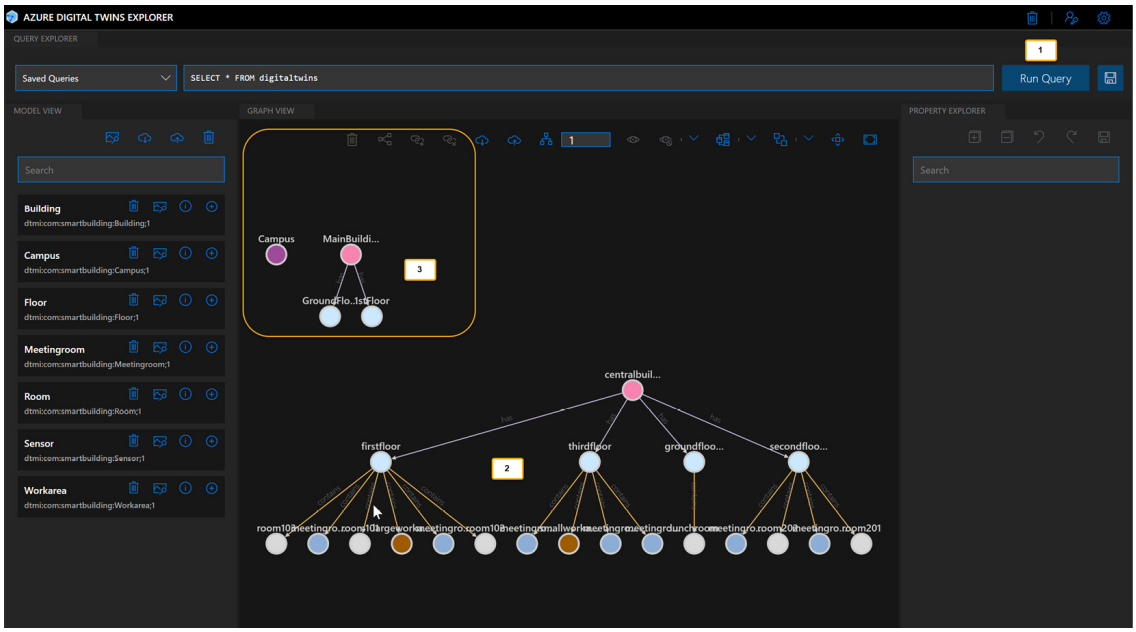

Figure 7.5 – Removing the structure of the work from the previous chapters

Let's clean up the structure of the previous chapter, as shown in *Figure 7.5*. Execute the following steps:

1. Click the **Run Query** button to refresh the graph view.
2. View the imported demo graph.
3. Remove the structure of the previous chapters. Select each relationship one by one and click the **Delete Relationship** button. The digital twins can be removed all at once by multi-selecting them using the *Shift* button. Click the **Delete Selected Twins** button to remove them all at once. The following screenshot shows the results:

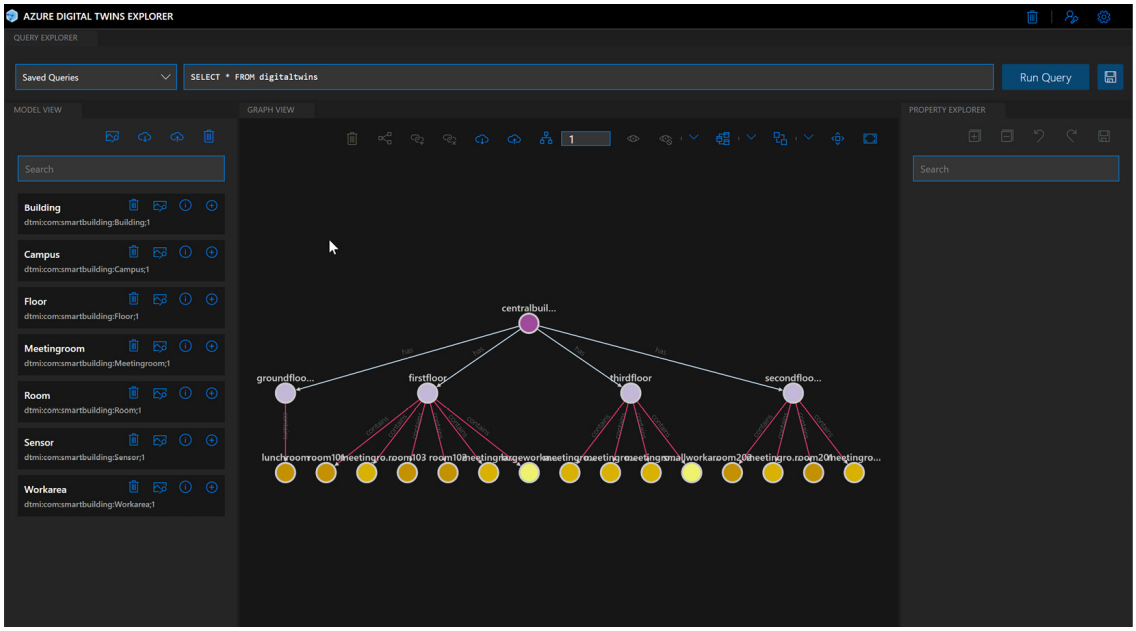

Figure 7.6 – The result after cleaning up the structure

The Azure Digital Twins instance now only contains the imported model.

In this section, we have set up a demo graph containing a large structure of several types of digital twins. In the next section, we will begin to execute several basic queries on this structure, in order to understand how they work.

Basic querying

Let's start with some basic queries that you can perform. We will be using Azure Digital Twins Explorer to execute these queries. The most basic query is the one that we have already been using. Enter the following query in the **Query** field and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS
```

The result of the query can be viewed in *Figure 7.7*. We have set **Run Layout** to **fCoSE** to make it more readable. **fCoSE**, or **fast Compound Spring Embedder**, is a faster version of the CoSE algorithm. The algorithm combines several techniques to produce a more aesthetic version of a force-directed graph. It tries to create a visual model of all digital twins and their underlying relationships, where the relationships are drawn as short as possible while maintaining a readable layout:

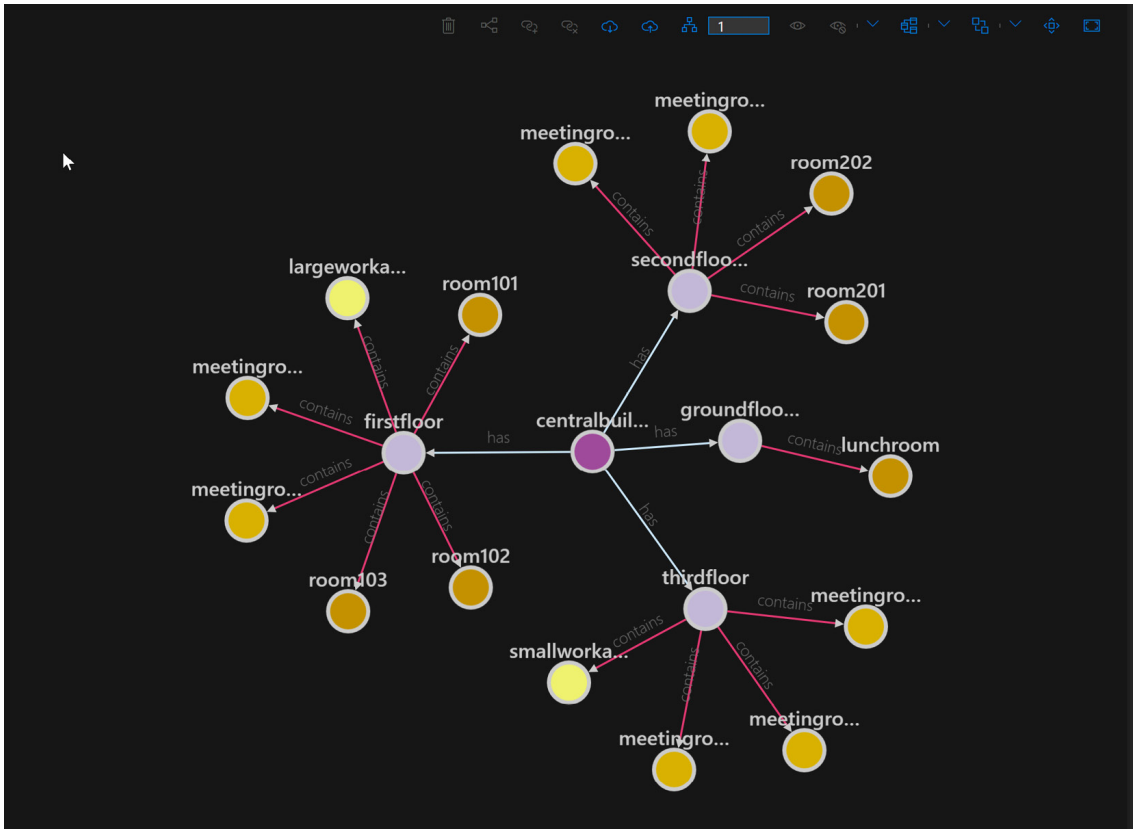

Figure 7.7 – The result of querying all digital twins

In the next query, we will add a filter. Enter the following query in the **Query** field and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS DT WHERE DT.lightson = TRUE
```

The filter in the query says to only look for digital twins that have a property called `lightson`, and that have the property set to the `TRUE` value. The result of the query is seen in *Figure 7.8*:

Figure 7.8 – Querying digital twins with a lights-on filter

We can see that only the digital twins that are based on the **Floor** model, and that also have the `lightson` property set to `TRUE`, are shown. All other related objects are not returned. To do that will require us to perform a `JOIN`. This will be explained later in this chapter.

The next query will be checking for a temperature between two values. Enter the following query in the **Query** field and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS DT WHERE
DT.temperaturevalue>20.0 AND DT.temperaturevalue<21.0
```

The result of the query is seen in *Figure 7.9*:

Figure 7.9 – A query returning rooms that have a temperature value between 20 and 21 degrees

Just like the previous query, only digital twins with the `temperaturevalue` property, and the value of that property set between both values, are returned.

IMPORTANT NOTE

A query with a property in the `WHERE` *clause will return all digital twins that contain that property. This could mean that digital twins from different model types will be returned. We will later use* `IS_OF_MODEL` *to control the query result based on the model type.*

The following query uses the `IN` operator. Querying using the `IN` operator can reduce the number of queries by specifying multiple outcome results. Enter the following query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS DT WHERE DT.$dtId IN
['centralbuilding','thirdfloor','meetingroom301',
```

```
    'meetingroom302']
```

This query checks if the ID of the digital twin is equal to one of the IDs named in the collection between the brackets after the `IN` operator. The result of the query is seen in *Figure 7.10*:

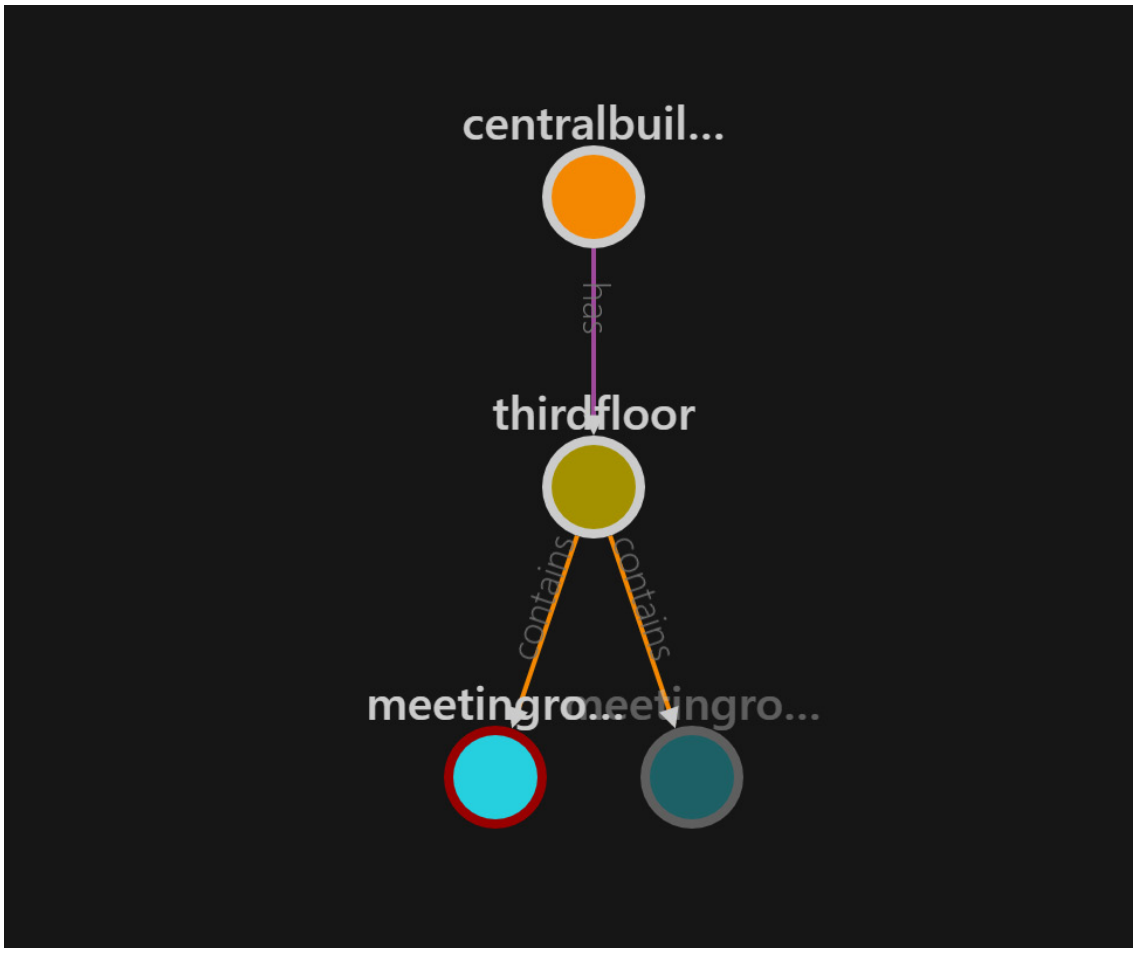

Figure 7.10 – Query result of an IN operator example

This query returns digital twins that have a relationship with each other. Azure Digital Twins Explorer is showing those relationships. While a digital twin contains the reference to another digital twin, the result does not contain the actual relationship object itself. Only digital twins are returned by the query, since the query output only specifies digital twins.

It is also possible to identify digital twins that have a certain property defined. Enter the following query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS WHERE IS_DEFINED(occupied)
```

This query returns only digital twins that have a property called `occupied`. The result is shown in *Figure 7.11*:

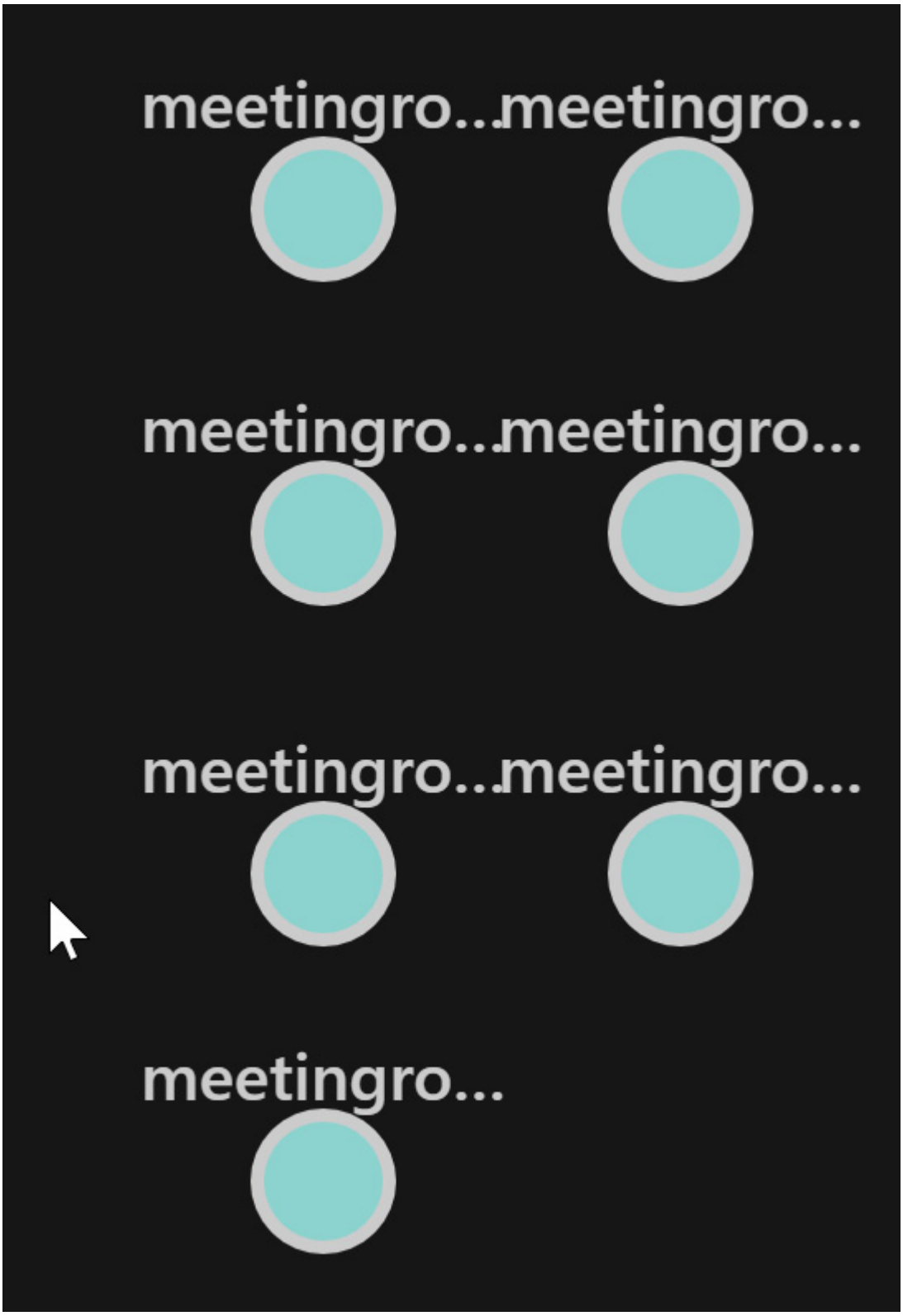

Figure 7.11 – A query returning only digital twins that have the occupied property

It is also possible to check if a property is of the NUMBER type. Enter the following query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS DT WHERE
IS_NUMBER(DT.temperaturevalue)
```

This will return the same result as the previous query, since the same digital twins contain a property called `temperaturevalue` of the `NUMBER` type.

The next query allows us to get a certain amount of top results back. Enter the following query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT TOP(9) FROM digitaltwins
```

The query returns the top nine digital twins returned by the query. The result is shown in *Figure 7.12*:

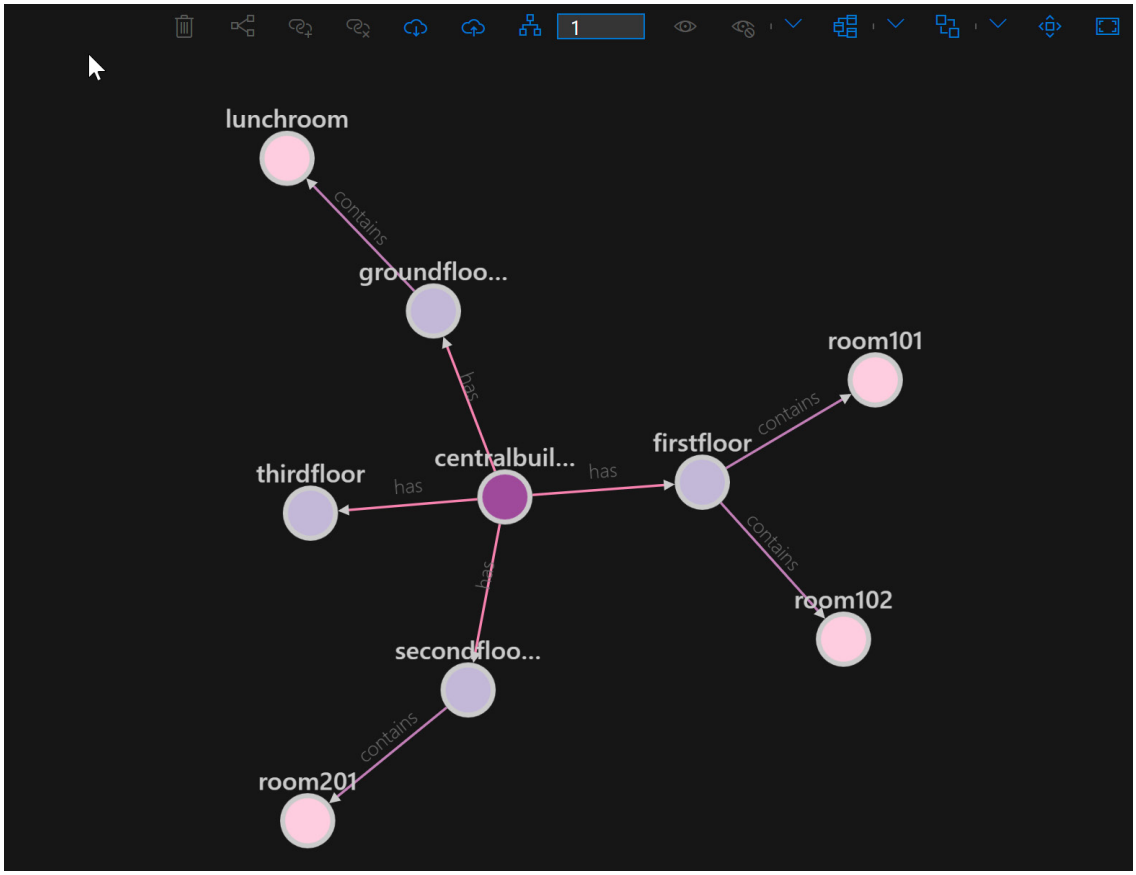

Figure 7.12 – The top nine results of a query

The order of digital twins is unspecified, causing this method to return the top rows based on the underlying database. Furthermore, this differs from the order in which digital twins and their relationships were created.

The last query helped us to determine the number of digital twins returned. Enter the following query in the **Query** field and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT COUNT() FROM DIGITALTWINS
```

This query will not show any results in Azure Digital Twins Explorer, since it does not return any digital twins and/or relationships. It only returns the number of digital twins. It allows you to count the result set, and it can be used on queries with a `WHERE` clause, and even with the `JOIN` operator.

In this section, we have learned some of the basic ways to query digital twins. We have also learned about the `IS_NUMBER`, `TOP`, and `COUNT` operators. In the next section, we will learn how to query by model type.

Querying by model

In this section, we will explain how you can query by model. Each digital twin is derived from a model. The Query API allows us to use the `IS_OF_MODEL` operator. The definition of such a query is shown in the following:

```
SELECT * FROM DIGITALTWINS <Collection> WHERE IS_OF_MODEL(
<Collection>, <ModelId>, <Exact>)
```

The `IS_OF_MODEL` operator can have up to three parameters. Each of the parameters is explained here:

Parameters	Result set
`<ModelId>`	The result set contains all created digital twins that are based on this specific model and all models derived from this model. Only digital twins that have the same or higher version number as the model will be part of the result set.
`<Collection>`	The result set is based on the collection specified. This is mostly used when joining another result set in the query by using the `JOIN` keyword. This will be explained in a later section of this chapter.
`<Exact>`	The result set is based on the exact `<ModelId>` and version of the model by using the `exact` keyword.

We will explain the results of using these parameters in several examples using the `IS_OF_MODEL` operator. We start by defining the `<ModelId>`.

Enter the following query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS WHERE
IS_OF_MODEL('dtmi:com:smartbuilding:Room;1')
```

This query specifies to return all digital twins that are based on the `Room` model. The result is shown in *Figure 7.13*:

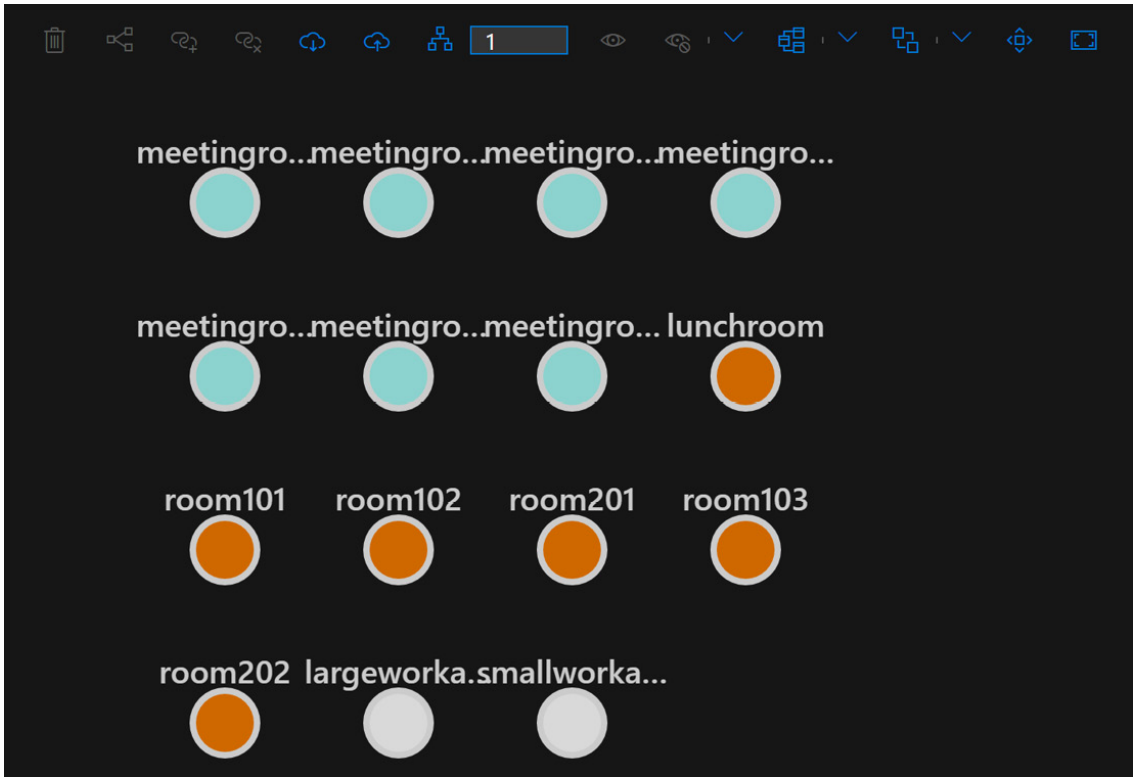

Figure 7.13 – All digital twins that are based on the Room model

We can see that it also returns the digital twins that are based on derived models, in this case, `Meetingroom` and `ActivityArea`.

We will add the `exact` keyword to the query. Enter the following query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT * FROM DIGITALTWINS WHERE
IS_OF_MODEL('dtmi:com:smartbuilding:Room;1', exact)
```

The `exact` keyword will make sure that only digital twins that are based on the exact model are returned. The result is shown in *Figure 7.14*:

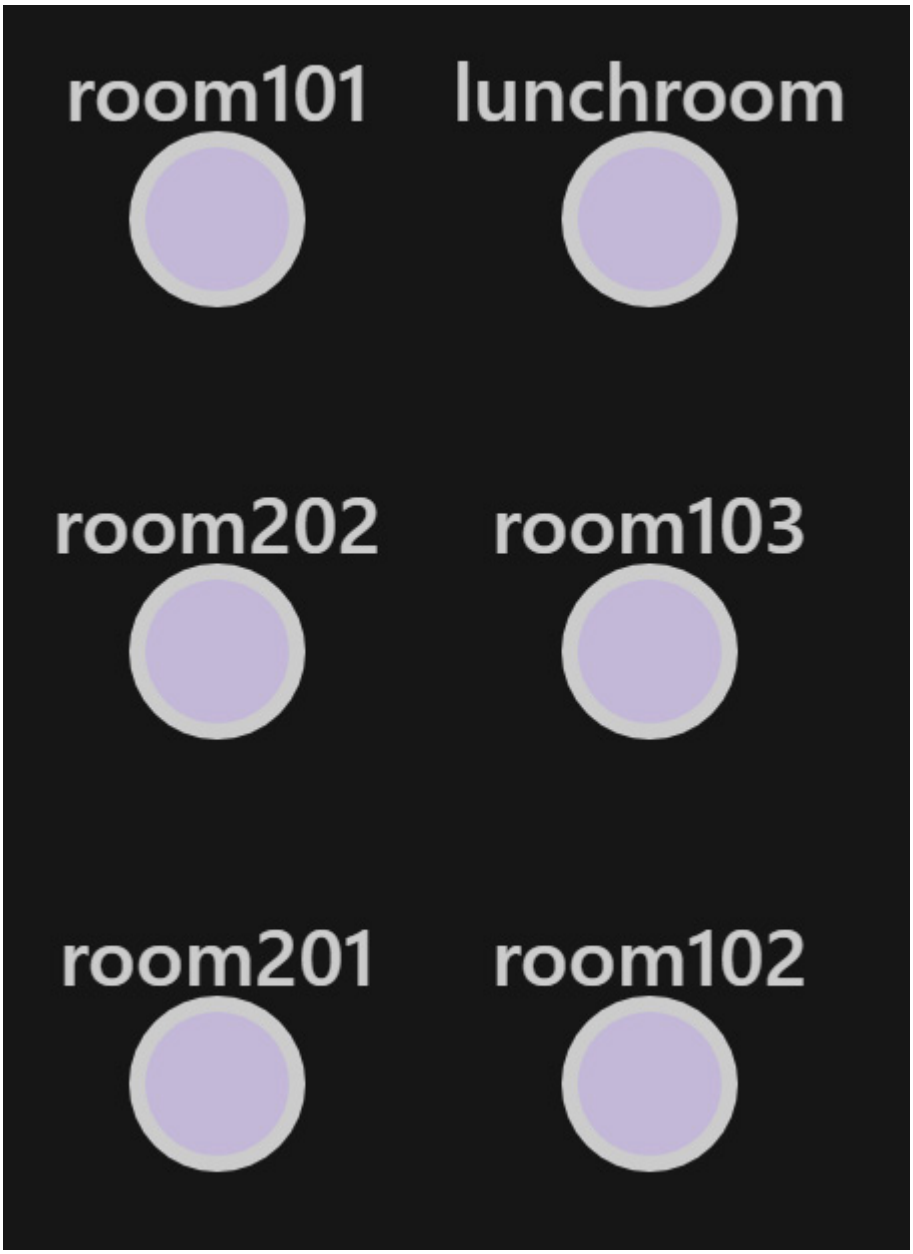

Figure 7.14 – All digital twins based on the Room model with the exact keyword specified

This section explained how to query based on models. This allows us to filter our results even more. In the next section, we will learn about querying relationships using the `JOIN` operator.

Querying relationships

This section will explain how we can query relationships by using the `JOIN` operator. The `JOIN` operator allows us to join (or combine, as we say) different result sets of digital twins. You can add up to five `JOIN` operators in one query.

There are some rules that we have to apply when using a `JOIN` operator. These rules are as follows:

- We need to specify a name for a result set. That means that we need to replace `FROM DIGITALTWINS` with, for example, `FROM DIGITALTWINS DT`. `DT`, is, in this case, the name of the result set. The name can be anything we want.
- The query is required to have a `$dtId` value in the `WHERE` clause.
- The `SELECT *` needs to be replaced with, for example, `SELECT DT, RT`. `DT` and `RT`, in this case, are both result sets. At least one result set or a specific field in a result set is required.

The following query does a `JOIN` operation between digital twins with a specified relationship called `contains`. In our current Azure Digital Twins instance, a `contains` relationship is only defined between `Floor` and `Room`. Enter the following query in the **Query** field and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT DT,RT FROM DIGITALTWINS DT JOIN RT RELATED
DT.contains WHERE DT.$dtId = 'secondfloor'
```

The query contains a `$dtId` value in the `WHERE` clause. It is required when using the `JOIN` operator. Since we want to have all the rooms belonging to a specific floor, the `$dtId` value is referencing the ID of the floor called `secondfloor`. We will return `DT,RT` as the named result sets.

The result is shown in *Figure 7.15*:

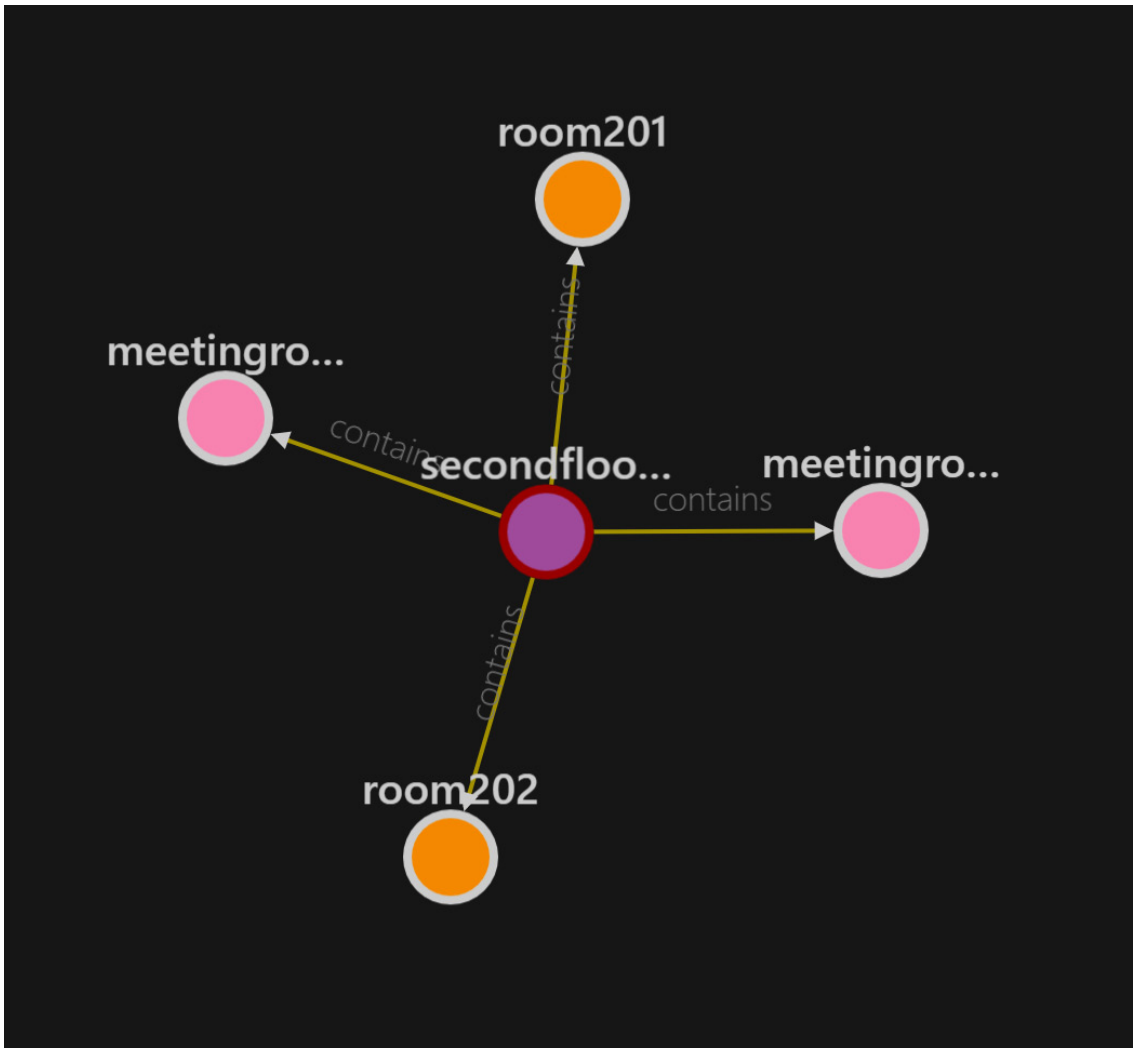

Figure 7.15 – The result of a query using a JOIN operator

The result contains all rooms available for the floor with the `secondfloor` name.

We want to have the building, a specific floor called `firstfloor`, and its meeting rooms returned. This query contains multiple `JOIN` operators. To explain the process of building such a query, we will specify each step. We start by getting the floors for buildings:

```
SELECT BU,FL FROM DIGITALTWINS BU JOIN FL RELATED BU.has
```

This query would return all buildings. Since we have a single building created in our Azure Digital Twins instance, only one building and its floors are returned.

We will extend the query by specifying a specific floor, as follows:

```
SELECT BU,FL FROM DIGITALTWINS BU JOIN FL RELATED BU.has
WHERE FL.$dtId='firstfloor'
```

We now extend the query by getting all the rooms of that floor, as follows:

```
SELECT BU,FL,RO FROM DIGITALTWINS BU JOIN FL RELATED
BU.has JOIN RO RELATED FL.contains WHERE
FL.$dtId='firstfloor'
```

Finally, we want to specify that we only want to get meeting rooms returned. Enter this query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT BU,FL,RO FROM DIGITALTWINS BU JOIN FL RELATED
BU.has JOIN RO RELATED FL.contains WHERE
IS_OF_MODEL(RO,'dtmi:com:smartbuilding:Meetingroom;1') AND
FL.$dtId='firstfloor'
```

The result is shown in *Figure 7.16*:

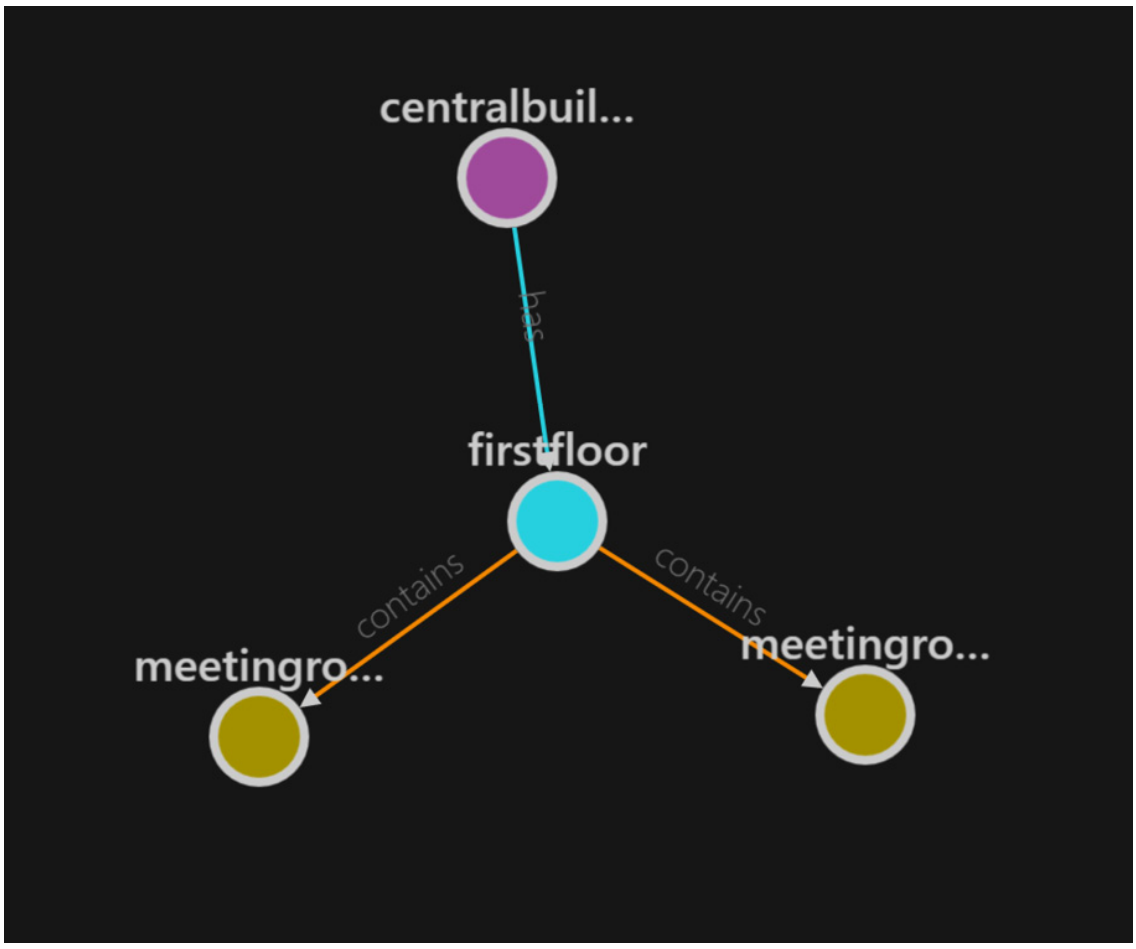

Figure 7.16 – A query returning digital twins using multiple JOIN operators and a WHERE clause

We have learned how to query relationships by using the `JOIN` operator. In the next section, you will learn how to filter the results of a query.

Filtering results

This section will explain how to filter your results more specifically by including relationships and single properties into the result of digital twins. The first example returns not only the digital twins, but also the relationships between the digital twins, by using something we call **projection**.

Projection means that we define a name for the relationship, and in case of the query, the `JOIN` operation. A small example will explain this more clearly. Normally, a `JOIN` operation would be specified as follows:

```
SELECT BU,FL FROM DIGITALTWINS BU JOIN FL RELATED BU.has
WHERE BU.$dtId='centralbuilding'
```

Now, we use projection to also return the relationship. Enter this query in the **Query** field, and click the **Run Query** button in Azure Digital Twins Explorer:

```
SELECT BU,FL,BF FROM DIGITALTWINS BU JOIN FL RELATED
BU.has BF WHERE BU.$dtId='centralbuilding'
```

The result is shown in *Figure 7.17*:

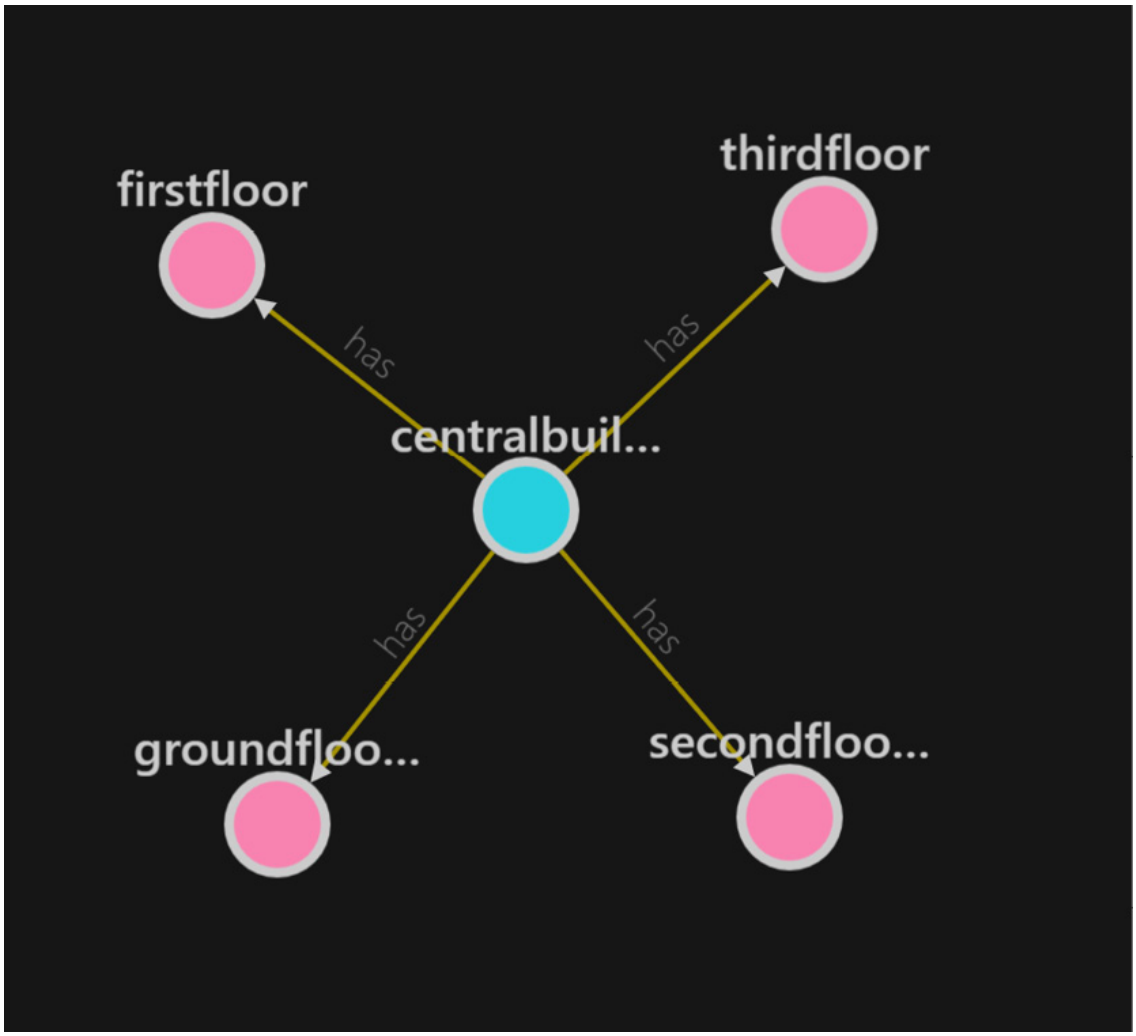

Figure 7.17 – A query result for getting digital twins and relationships

There is no difference in Azure Digital Twins Explorer with or without the relationships. The viewer will always show the result. However, the result of the query with **projection** will also contain the relationships next to the digital twins. This can be helpful if we have properties specified at a relationship.

In the following examples, we will show how to return properties of a digital twin or a relationship in the result of a query. When using properties, a primitive check is required. The primitive check can be done by adding the `IS_PRIMITVE(<property>)` to the `WHERE` clause of the query. This is shown in the following snippet:

```
SELECT temperaturevalue FROM DIGITALTWINS WHERE
IS_OF_MODEL('dtmi:com:smartbuilding:Meetingroom;1') AND
IS_PRIMITIVE(temperaturevalue)
```

It is of no use to execute the query in Azure Digital Twins Explorer, since it will not return any digital twins. The query only returns the `tempera-`

`turevalue` of all meeting rooms in the Azure Digital Twins instance.

The second example will return the `lightson` and `level` properties of different digital twins by using a `JOIN` operator in the query. All properties require a primitive check. Refer to the following code:

```
SELECT FL.lightson,BF.level FROM DIGITALTWINS BU JOIN FL
RELATED BU.has BF WHERE BU.$dtId='centralbuilding' AND
IS_PRIMITIVE(FL.lightson) AND IS_PRIMITIVE(BF.level)
```

No results can be shown in Azure Digital Twins Explorer, since it does not return digital twins.

In this section, you have learned about filtering using **projection** and the properties of digital twins. In the next section, you will learn how to execute all these queries using code.

Querying using code

Querying can be done through a call with the Query API using the .NET SDK. The result, however, can differ from the query request. It is mainly determined by the `SELECT` specification. Here, you will see several different return scenarios:

Type of query	Result item
`SELECT * FROM`	`BasicDigitalTwin`
`SELECT Building, Floor, Meetingroom FROM`	`Dictionary<string, BasicDigitalTwin>` where each result (`Building`, `Floor`, and `Meetingroom`) is a key in the dictionary
`SELECT Floor.level`	Integer

Since the result can differ per query, it is somewhat difficult to write a method that can be reused. It completely depends on what result you want the query to give back. However, it also allows you to write your own classes, supporting the serialization of results backward and forward.

A result set is always based on the `Pageable<>` class. It is a collection of values that support iteration over multiple service requests.

We will start by creating a method supporting the `SELECT *` type of query. Open the `DigitalTwinsManager` class and add the following code:

```
public Pageable<BasicDigitalTwin> QueryDigitalTwins(string
query)
{
    Pageable<BasicDigitalTwin> result = null;
    try
    {
        result = client.Query<BasicDigitalTwin>(query);
    }
    catch (RequestFailedException)
    {
    }
    return result;
}
```

Open the `Main` method in the `Program.cs` file and replace the code with the following:

```
Pageable<BasicDigitalTwin> result =
dtHelper.QueryDigitalTwins("SELECT * FROM DIGITALTWINS");
foreach (BasicDigitalTwin item in result)
{
    Console.WriteLine(item.Id);
}
```

Run the application. The result is shown in *Figure 7.18*. The code outputs the IDs of each digital twin returned in the result of the query:

Figure 7.18 – The result of executing a query to get all digital twins

The following example shows how to handle the result of a query where several digital twin names are specified in the `SELECT` statement. Open the `DigitalTwinsManager` class and add the following code:

```
public Pageable<Dictionary<string, BasicDigitalTwin>>
Query(string query)
{
    Pageable<Dictionary<string, BasicDigitalTwin>> result
= null;
    try
    {
        result = client.Query<Dictionary<string,
BasicDigitalTwin>>(query);
    }
    catch(RequestFailedException)
    {
    }
    return result;
}
```

This code does not differ greatly from the previous example. However, instead of returning a pageable collection of `BasicDigitalTwin` objects, it returns a pageable collection of `Dictionary<string, BasicDigitalTwin>` objects.

Open the `Main` method in the `Program.cs` file and replace the code with the following:

```
Pageable<Dictionary<string, BasicDigitalTwin>> result =
dtHelper.Query("SELECT BU,FL FROM DIGITALTWINS BU JOIN FL
RELATED BU.has WHERE BU.$dtId='centralbuilding'");
foreach (Dictionary<string, BasicDigitalTwin> item in
result)
{
    BasicDigitalTwin BU = item["BU"] as BasicDigitalTwin;
    BasicDigitalTwin FL = item["FL"] as BasicDigitalTwin;
    Console.WriteLine(string.Format("{0}-{1}", BU.Id,
FL.Id));
}
```

Each result of the pageable collection contains a dictionary. Since we have used `SELECT BU,FL` as a statement, the dictionary contains two keys that are similar to the specified digital twin sets in the `SELECT` statement. To get a part of the returned digital twins from the result item, we use the following call:

```
BasicDigitalTwin BU = item[<digital twin set >] as
BasicDigitalTwin
```

Run the application. The result is shown in *Figure 7.19*:

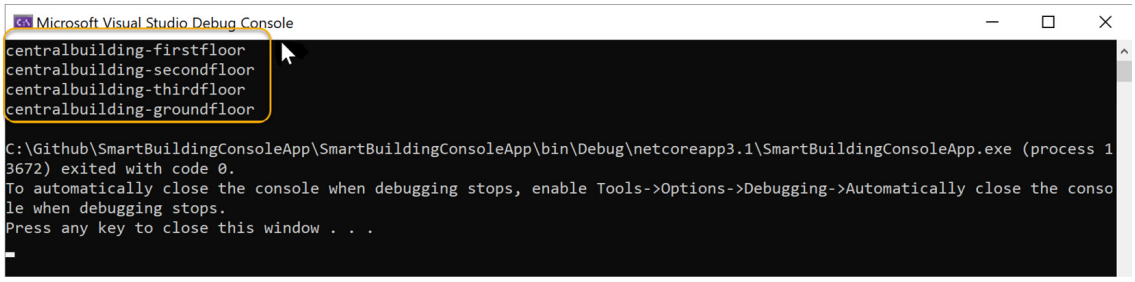

Figure 7.19 – Handling multiple result sets using a query

You have learned how to query code in order to return different forms of result sets. In the next section, you will learn to query asynchronous calls using code.

Querying asynchronous calls using code

The query calls of the Query API from Azure Digital Twins support paging. However, this requires another approach to be used in calling the methods. Until now, we have been calling methods synchronously. Since a call always depends on the availability of the service or the network connectivity, building production solutions requires using asynchronous calls.

Asynchronous calls also allow us to perform paging with the result of a query. Paging does immediately think about a fix number of results by page. In the case of executing queries, we talk more about handling each result from the query. The following example calls a method for each result in the query.

Open the `DigitalTwinsManager` class and add the following code inside the class definition, directly under the class properties:

```
public delegate void QueryResult(BasicDigitalTwin dt);
```

This code contains the definition of a delegate function. A **delegate function** describes the format of a function that could be, for example, given as a parameter to another method.

And that is exactly what we will do. Add the following two methods to the `DigitalTwinsManager` class:

```
public void QueryDigitalTwins(string query, QueryResult
onQueryResult)
{
    System.Threading.Tasks.Task task =
System.Threading.Tasks.Task.Run(
        () => QueryDigitalTwinsAsync(query,
onQueryResult));
}
public async void QueryDigitalTwinsAsync(string query,
QueryResult onQueryResult)
{
    AsyncPageable<BasicDigitalTwin> result =
client.QueryAsync<BasicDigitalTwin>(query);
    try
```

```
        {
            await foreach (BasicDigitalTwin dt in result)
            {
                onQueryResult(dt);
            }
        }
        catch (RequestFailedException)
        {
        }
    }
```

The `QueryDigitalTwinsAsync` method contains an asynchronous call to the Query API. For each result of the query, the method specified by the `onQueryResult` parameter is called.

TIP

It is possible to replace the `onQueryResult` *call with a call that has an array of a fixed number of results that allows you to specify a page size.*

We want to have the `QueryDigitalTwinsAsync` method be executed in a separate thread. Therefore, we have the `QueryDigitalTwins` method that uses the `System.Threading.Tasks.Task` class to create a separate thread that executes the query.

Open the `program.cs` file and add the following code to the file:

```
    public static void OnQueryResult(BasicDigitalTwin dt)
    {
        Console.WriteLine(dt.Id);
    }
```

This method will be called every time a search result is retrieved during the asynchronous call. Replace the contents of the `Main` method with the following code:

```
        DigitalTwinsManager dtHelper = new
    DigitalTwinsManager();
        dtHelper.QueryDigitalTwins("SELECT * FROM
    DIGITALTWINS", OnQueryResult);
```

```
        while(true) { }
```

This method will call the `QueryDigitalTwins` method and give a reference to the static `OnQueryResult` method. The `while(true) {}` loop is required, since otherwise, the console application would already be closed before the second thread can start feeding back the results.

Run the application. The result of the application can be seen in *Figure 7.20*:

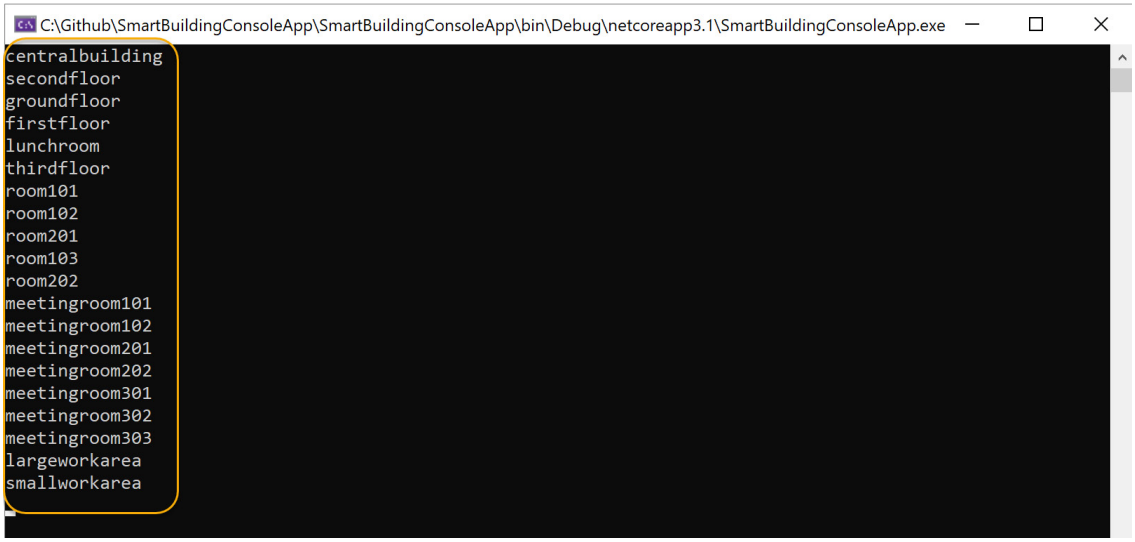

Figure 7.20 – Results are returned via an asynchronous call

Asynchronous calls will be used more often in the upcoming chapters.

Summary

In this chapter, you have learned how the query language provided by the Query API allows us to query digital twins and their relationships. We have learned how these queries are built up and how they return their results. We have also learned how to use the .NET SDK to execute these queries using C# code. In this chapter, we have mastered how to execute queries using Azure Digital Twins Explorer and through .NET code. Acquiring data from an Azure Digital Twins instance is done by executing queries almost every time.

In the next chapter, we will learn in several steps what is required to build our first model. Based on several scenarios and types of data, we will explain the best way to create your Azure Digital Twins instances.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which class is used when a result set is returned from a query?
   a. `Paging<> class`
   b. `Collection<> class`
   c. `Pageable<> class`
2. Which keyword is required in `IS_OF_MODEL` to make sure that the query only returns the specified model?
   a. No keyword is required.
   b. `exact`.
   c. The ID of the model.
3. What is the advantage of querying asynchronously?
   a. It is just another way of writing code.
   b. It allows you to implement paging and is more suitable for production.

Further reading

If you want to learn more on the subject, check out these resources:

- Query the Azure Digital Twins twin graph, available at the following URL: **https://docs.microsoft.com/en-us/azure/digital-twins/how-to-query-graph**.