BUILDER™
API Guide

Version 2022

September 29, 2022

Table of Contents

INTRODUCTION

About BUILDER API

The BUILDER API allows communication with computerized maintenance management systems (CMMSs) and other software through the use of service calls that create, transmit, modify, and delete BUILDER data objects.

This Guide to the BUILDER API contains:

- Information for getting started with the API
- Topical introduction to the API calls
- A code example for each call
- Sample use cases

About Permissions

The permissions you need in the BUILDER API to execute a given call will parallel those needed in the BUILDER Web interface to perform the same actions.

The permissions you have for operating via the BUILDER API are governed by the account (user name and password) that you use to set your credentials when configuring the API.

What's New in BUILDER API

With the 2022 version, the BUILDER API now supports:

- Attachments (these can be associated with assets or with inspections)
- Inventory cost modifiers
- Cost modifier libraries and lists
- Selected calls for funding and funding sources
- Cost books and cost records
- Getting additional data sets (inflation books, RSL books)
- Work items
- Scenarios

- Getting Facility/Building attributes using customized calls such as GetFacilityConstructionTypes
- Getting selected work configuration information (policies, policy sequences, prioritization schemes)
- Getting distress severity values and distress density values

GETTING STARTED WITH BUILDER API

Prerequisites

To get started with BUILDER API you need:

1. A specific BUILDER instance you want to connect to; know the web service URL for this.
2. A user account for that BUILDER instance, with user name and password.

Configure the API

If you are not using a BUILDER API instance that has already been configured, see below for the steps needed to configure the API for use.

A basic introductory tutorial is provided next. Use cases with sample code are available in the References section.

Step 1: Add a New Service Reference

Visual Studio

To add a new service reference to the project in Visual Studio,

1. In the Solution Explorer panel, right-click on the Service Reference folder.
2. Select "Add service reference" from the dropdown list. *The "Add Service Reference" popup window will appear*.
3. Enter the following information in the "Add Service Reference" window:
    a. For Address, insert the web service URL for your BUILDER instance identified in Step 1 of the prerequisites.
    b. Ignore the Services area of the "Add Service Reference" window.
    c. Make up a name for the service reference, and enter that name in the text box below the label "Namespace".
4. Activate the **OK** button to add the service reference to the project.

Other (WSDL)

If you are not using Visual Studio to add a new service reference, you will need to know the location of the WSDL. Generally, this will be

http: (or https:) //<BuilderURL>/WebService/Builder.svc?wsdl

For example, the link to the BUILDER WSDL for USAF when using Microsoft's svcutil.exe is:

https://builder.cecer.army.mil/USAF/WebService/Builder.svc?wsdl

Or, for other uses:

https://builder.cecer.army.mil/USAF/WebService/Builder.svc?singleWsdl

Step 2 (Optional): Enlarge MaxReceivedMessageSize

If errors indicate that MaxReceivedMessageSize has been exceeded, you can do the following in Visual Studio to enlarge it:

1. Go into App.config (This should be one of the tabs across the top of the Visual Studio display).
2. Find the system.servicemodel element in the XML.
3. There should be an element below system.servicemodel called "binding".
4. Add MaxReceivedMessageSize as an attribute to binding, and set its value equal to a large number.
5. Or, if MaxReceivedMessageSize is already present, increase its value to a large number.

Step 3: Create a New Instance of the BuilderClient Class

The BuilderClient class is already in the Namespace created in Step 1 above. To create a new instance of it, select one of the following methods depending on whether you have multiple types of endpoints (for example, development vs. production) or only one.

Option A: One Endpoint Type

At the top of the body of your main program or class, insert the following:

```
var service = new BuilderClient( );
```

where `service` is an example variable name; you may substitute your own variable name.

Option B: Multiple Endpoint Types

At the top of the body of your main program or class, insert the following:

```
var service = new BuilderClient("<binding name from App.config>");
```

where `service` is an example variable name; you may substitute your own variable name.

Step 4: Set the Credentials

Two or three credentials need to be set for the new BuilderClient class instance created above. After the `var.service` line defined in Step 3, insert the following.

1. **User name (Required).** Enter your user name for the BUILDER instance you want to connect to, as follows:

```
service.ClientCredentials.Username.Username = "<your BUILDER user name>";
```

2. **Password (Required).** On the next line, enter the corresponding password:

```
service.ClientCredentials.Username.Password= "<your BUILDER password>";
```

3. **Endpoint Address (Optional).** If you want to use a different endpoint address than the one you entered into the "Address" field of the service reference in Step 1, add the following line:

```
service.Endpoint.Address = new Endpoint.Address("<URL of the web service>");
```

where `service` is an example variable name used when creating the new instance of the BuilderClient class. Your variable name may differ.

Step 5: Set a Global Variable

In this last step, you will choose a global variable name to represent the instance of BuilderClient that you are using.

```
client = service;
```

where `client` is the global variable and `service` is the parameter for the constructor.

> **CAUTION:** When you create a new instance of a DTO object, you can't use the `client` variable; you will need to use the service reference name.

Introductory Tutorial

This topic provides instructions for doing some basic tasks using the API, to familiarize you with getting and displaying information using BUILDER API.

Exercise 1: Check Version Numbers

Run the code below to request the version number of the Web service being used. Version and build number will be output to the console.

No substitutions or replacements are needed in this code to run it.

```
public void getServiceFullVersion()
{
  Console.WriteLine("service version is " +
     client.GetVersionService().Data.Major.ToString() + "." +
     client.GetVersionService().Data.Minor.ToString() + "." +
     client.GetVersionService().Data.Revision.ToString() + " build " +
     client.GetVersionService().Data.Build.ToString() );
}
//the GetVersionService() call returns DTOVersion
```

If all you need is the build number, you can use this shorter piece of code:

```
public void getServiceBuildVersion()
{
  Console.WriteLine(client.GetVersionService().Data.Build.ToString());
}
```

where `client` is the global variable name representing the instance of BuilderClient that you are using.

Output: the BUILDER version you are using should appear on the first line of the console, followed by the version of the Web service on the next line of the console.

Exercise 2: List Facilities in a Complex

The functions shown in this section below show how to generate either (a) a list or (b) an array of GUIDs representing all Facilities in a Complex. For the first input parameter in each function, you need the GUID of the desired parent Complex (represented by CPX_ID in the examples).

The GUID of an asset (inventory item) is its `ID` property. This GUID will be the same for all of the example functions shown in this exercise.

Basic Form

The basic form of the service call is

```
GetFacilitiesByParentID(Guid id, int skip, int take)
```

where `count` is the number of records (Facilities), and `skip` and `take` are used for paging: `skip` is the number of records to skip before returning records; `take` is the number of records to be returned.

How Many Facilities?

The following code uses the `count` property of the service response and displays to the console the number of Facility records:

```
public void facCount()
{
   Console.WriteLine(client.GetFacilitiesByParentID(Guid.Parse("CPX_ID"), 0, 0).Count.ToString());
}
```

Output: On the console, you should see the total number of Facilities that are in the Complex specified by the GUID given in the first parameter.

Display on Console

This code shows setting `skip` and `take` to display on the console the GUIDs of the first three records:

```
public void getFacilitiesByParentID_A_()
{
   // option A. output to console to see a list of the first 3 Facilities under the parent Complex
   foreach (DTOFacility item in client.GetFacilitiesByParentID(Guid.Parse("CPX_ID"),0,3).Data)
   {
     Console.WriteLine(item.ID.ToString() + "\n");
   }
}
```

This code shows setting `skip` and `take` to display on the console the GUIDs of the next three records:

```
public void getFacilitiesByParentID_A_()
{
   // option A. output to console to see a list of the next 3 Facilities under the parent Complex
   foreach (DTOFacility item in client.GetFacilitiesByParentID(Guid.Parse("CPX_ID"),3,3).Data)
   {
     Console.WriteLine(item.ID.ToString() + "\n");
   }
 }
```

Store in Array

As an alternative to showing the list of Facility GUIDs on the console, you can store them in an array:

```
public void getFacilitiesByParentID_B_()
{
   // option B. store in an array
   DTOFacility[] theFacilities = client.GetFacilitiesByParentID(Guid.Parse("CPX_ID"), 0, 100).Data;
}
```

Additional Exercises

Additional examples can be found at Sample Use Cases in the Reference section.

GLOBAL INFORMATION AND SETTINGS

This section of the documentation contains API calls that pertain to BUILDER as a whole rather than to one aspect of it, such as inventory or inspections.

"BUILDER and Service Versions" below outlines the calls to see what BUILDER version number your BUILDER instance is running, and what service version number the instance is running.

BUILDER and Service Versions

Starting with the 2022 version of BUILDER API, the BUILDER service version and BUILDER version can be obtained together with one simple call, "GetVersions( ) " below.

Code examples below illustrate API calls for obtaining version information about the BUILDER service, information about what BUILDER version is being run, or both.

Code Examples: BUILDER and Service Versions

Code examples are in C#.

GetVersions( )

To request the version number of the Web service and the version number of the SMS application (BUILDER) being used, use the code below.

No substitutions or replacements are needed in this code to run it.

```
public void getVersions()
{
  var response = client.GetVersions();

  // console output (optional):
  Console.WriteLine("BUILDER version is "
      + response.Data.BuilderVersion.Major.ToString() + "."
      + response.Data.BuilderVersion.Minor.ToString() + "."
      + response.Data.BuilderVersion.Revision.ToString()
      + " build " + response.Data.BuilderVersion.Build.ToString());

  Console.WriteLine ("Service version is "
     + response.Data.ServiceVersion.Major.ToString() + "."
      + response.Data.ServiceVersion.Minor.ToString() + "."
      + response.Data.ServiceVersion.Revision.ToString()
      + " build " + response.Data.ServiceVersion.Build.ToString());
}
// the GetVersions() call returns DTOBuilderServiceVersionPair
```

GetVersionBuilder( )

Run the code below to request the version number of the SMS application (BUILDER) being used. Version and build number will be output to the console.

No substitutions or replacements are needed in this code to run it.

```
public void getBuilderFullVersion()
{
  Console.WriteLine("BUILDER version is " +
     client.GetVersionBuilder().Data.Major.ToString() + "." +
     client.GetVersionBuilder().Data.Minor.ToString() + "." +
     client.GetVersionBuilder().Data.Revision.ToString() + " build " +
     client.GetVersionBuilder().Data.Build.ToString() );
}
//the GetVersionBuilder() call returns DTOBuilder
```

If all you need is the build number, you can use this shorter piece of code:

```
public void getBuilderBuildVersion()
{
  Console.WriteLine(client.GetVersionBuilder().Data.Build.ToString());
}
```

GetVersionService( )

Run the code below to request the version number of the Web service being used. Version and build number will be output to the console.

No substitutions or replacements are needed in this code to run it.

```
public void getServiceFullVersion()
{
  Console.WriteLine("service version is " +
     client.GetVersionService().Data.Major.ToString() + "." +
     client.GetVersionService().Data.Minor.ToString() + "." +
     client.GetVersionService().Data.Revision.ToString() + " build " +
     client.GetVersionService().Data.Build.ToString() );
}
//the GetVersionService() call returns DTOVersion
```

If all you need is the build number, you can use this shorter piece of code:

```
public void getServiceBuildVersion()
{
  Console.WriteLine(client.GetVersionService().Data.Build.ToString());
}
```

INVENTORY

About Inventory Service Calls

Using BUILDER API, you can do the following at each inventory level:

- Get inventory (Get by GUID, or Get by Alternate ID, or Get by Parent ID, or "Search for Assets by Name" on page 31)
- Create inventory
- Update inventory
- Get performance records (metrics); not available at Section Details level
- Delete inventory

  Note About Performance Records: If you will be creating inspections, you may want to delay getting performance records until after you have submitted the inspection information (either in the BUILDER web app or via the API) and performed a Site or Facility rollup. This way, the performance metrics reported will reflect your most recent information.

  For most assets, performance records–such as condition index–can't be directly viewed as a property of the asset using the API. However, the API service call "GetPerformanceRecords(Guid ownerlink, PerformanceRecordType metric, int year)" on page 94 can be used at all levels of inventory except Section Detail.

Additional actions related to inventory are:

- *Attachments*. You can associate ("add") one or more attachments to any inventory item, get the attachments, or delete them. See the chapter on Attachments.
- *Rollup*. You can perform global rollup, Site rollup, or Facility rollup. See "Inventory Rollup" on page 50.
- *Get rollup status*. See "Inventory Rollup" on page 50.
- *Get, create, assign, update or delete cost modifiers* that add to the cost of assets or multiply the cost by a fixed factor to account for regional cost variations or other special factors. See the chapter on Inventory Cost Modifiers.
- *Cost modifier libraries*. Design the scope of a cost modifier by assigning it to a cost modifier "library" that is available for a particular level and area of inventory. Get, update or delete such library assignments. See "Cost Modifier Libraries " on page 101 in the chapter on data libraries.

About Inventory Sample Use Cases

Sample use cases show code for the following functions related to inventory:

- "Use Case 1: Get Inventory" on page 140
- "Use Case 2: Add a Facility to the Inventory Tree" on page 144
- "Use Case 3: Add Inventory to a Facility" on page 146
- "Use Case 4: Update Inventory Data" on page 149

Get Assets by GUID

You can get assets from Organization down through Section Details using the BUILDER API. This topic covers selecting one asset to get by providing its GUID (`ID` property). This approach works at all inventory levels. The topic also discusses how to display or store desired properties.

If you want to use a different method to get inventory, you can select it from this list of the top-level options:

- Enter the GUID (`ID` property) of the asset. **Scope:** Works at all inventory levels, and for Section Details.
- Enter the Alternate ID of the asset. **Scope:** Facilities/Buildings or Sections only.
- Enter the GUID (`ID` property) of the parent asset. This approach will return all assets having that parent. **Scope:** Works at all inventory levels except the root of the inventory tree.
- Search by Name or name fragment. **Scope:** Organization and Site only.

This topic explains the first alternative.

About Selecting by GUID

Scope: Works at all inventory levels.

To specify an asset by entering its GUID, you will need to enter the value in its `ID` property as the argument to one of the following methods:

- GetOrganization
- GetSite
- GetComplex

- GetFacility
- GetSystem
- GetComponent
- GetSection
- GetSectionDetail

These service calls will get the asset specified by the GUID. See the section "Display or Store Information" on page 18 for how to create code to display or store an asset's properties.

Converting a String to GUID Format

Using `Guid.Parse`

The code examples shown below take the string representation of a GUID and convert it to GUID format simultaneously as you call the service call. The result is that a (Guid id) is passed into the call via the parameter. In the examples, this is done using

```
(Guid.Parse("<guid>"));
```

where you insert the asset's `ID` property in string format (in double quotation marks) where `<guid>` is shown in the model above.

Using `new Guid`

An alternate option, used in the use cases but not shown in the shorter code examples, is using the following as the call parameter:

```
(new Guid ("<guid>") );
```

where you likewise insert the asset's `ID` property in string format (in double quotation marks) where `<guid>` is shown above.

Code Examples: Get Inventory by GUID

A C# code example for each level of inventory is provided below. The code examples incorporate the approach of converting a string to GUID format, as explained in "Converting a String to GUID Format" above.

General Model

The examples provided below are in the format

```
var item = client.Get<Inventory level>(Guid.Parse("<guid>"));
```

where <Inventory level> is replaced by the word Organization or Site or another inventory level, with initial uppercase letter; item will be an instance of the corresponding DTO object (such as DTOSite); and "<guid>" is the string representation of the GUID of the asset you wish to get.

GetOrganization(Guid id)

Replace ORG_ID with the string representation of an Organization's GUID to get an Organization and store its properties in an instance of DTOOrganization named "theOrganization" :

```
public void getOrganization()
{
    var theOrganization = client.GetOrganization(Guid.Parse("ORG_ID"));
}
// returns DTOOrganization
```

GetSite(Guid id)

Replace SITE_ID with the string representation of a Site's GUID to get the Site and store its properties in an instance of DTOSite named "theSite" :

```
public void getSite()
{
    var theSite = client.GetSite(Guid.Parse("SITE_ID"));
}
// returns DTOSite
```

GetComplex(Guid id)

Replace CPX_ID with the string representation of a Complex's GUID to get the Complex and store its properties in an instance of DTOComplex named "theComplex" :

```
public void getComplex()
{
    var theComplex = client.GetComplex(Guid.Parse("CPX_ID"));
}
// returns DTOComplex
```

GetFacility(Guid id)

Replace FAC_ID with the string representation of a Facility's GUID to get a Facility and store its properties in an instance of DTOFacility named "theFacility" :

```
public void getFacility()
{
    var theFacility = client.GetFacility(Guid.Parse("FAC_ID"));
}
// returns DTOFacility
```

GetSystem(Guid id)

Replace SYS_ID with the string representation of a System's GUID to get the System and store its properties in an instance of DTOSystem named "theSystem" :

```
public void getSystem()
{
    var theSystem = client.GetSystem(Guid.Parse("SYS_ID"));
}
// returns DTOSystem
```

GetComponent(Guid id)

Replace COMP_ID with the string representation of a Component's GUID to get the Component and store its properties in an instance of DTOComponent named "theComponent" :

```
public void getComponent()
{
    var theComponent = client.GetComponent(Guid.Parse("COMP_ID"));
}
// returns DTOComponent
```

GetSection(Guid id)

Replace SEC_ ID with the string representation of a Section's GUID to get the Section and store its properties in an instance of DTOSection named "theSection" :

```
public void getSection()
{
    var theSection = client.GetSection(Guid.Parse("SEC_ID"));
}
// returns DTOSection
```

GetSectionDetail(Guid id)

Replace SEC_DETAIL_ ID with the string representation of a Section Detail's GUID to get the Section Detail and store its properties in an instance of DTOSectionDetail named "theSectionDetail" :

```
public void getSectionDetail()
{
    var theSectionDetail = client.GetSectionDetail(Guid.Parse("SEC_DETAIL_ID"));
}
// returns DTOSectionDetail
```

Display or Store Information

Viewing information contained in the selected asset will require being aware of properties associated with that asset, and doing some coding.

Output to Console

To output a property to the console for viewing, use this code if the property is a string:

```
Console.WriteLine(item.<property>);
```

where `item` is the instance of the DTO object.

Alternatively, use the code below if the property is not a string:

```
Console.WriteLine(item.<property>.ToString() );
```

Store a Single Property in a Variable

To store an individual property in a variable, use this code:

```
var my<property> = item.<property>;
```

For example,

```
var myCI = item.CI;
```

Get Assets by Alternate ID

This topic covers selecting by alternate ID, which applies only to Facilities and Sections. It will also discuss how to display or store desired properties.

If you want to use a different method to get inventory, you can select it from this list of the top-level options:

- Enter the GUID (`ID` property) of the asset. **Scope:** Works at all inventory levels, and for Section Details.
- Enter the Alternate ID of the asset. **Scope:** Facilities/Buildings or Sections only.
- Enter the GUID (`ID` property) of the parent asset. This approach will return all assets having that parent. **Scope:** Works at all inventory levels except the root of the inventory tree.
- Search by Name or name fragment. **Scope:** Organization and Site only.

This topic explains the second alternative.

About Selection by Alternate ID

Scope: Facility or Section.

> **Note:** When RPUIDs are being used at a Site or installation, an RPUID will typically be the alternate ID for a Facility.

If an asset has an alternate ID, the service calls listed below provide an additional way to specify a Facility/Building or a Section:

- GetFacilityByAlternateID (string AlternateID)
- GetSectionByAlternateID (string AlternateID)

These service calls will get the asset specified by the GUID. See the section "Display or Store Information" on the facing page for how to create code to display an asset's properties.

> **WARNING:** It is up to the client to ensure uniqueness across alternate IDs. Getting by alternate ID will typically return one DTO object, but could return multiple objects.

Code Examples: Get Inventory by Alternate ID

A C# code example is provided below for each level of inventory accommodated.

GetFacilityByAlternateID(string AlternateID)

To get a Facility using this approach, replace ALT_ID with the string representing a real alternate ID such as an RPUID:

```
public void getFacilityByAlternateID()
{
    var idFacilities = client.GetFacilityByAlternateID("ALT_ID").Data;

    //Optional: display the ID of the Facilities to the console
    foreach (DTOFacility item in idFacilities)
        Console.WriteLine(item.ID.ToString());
    Console.Read();
}
// returns (service response) array of DTOFacility
```

GetSectionByAlternateID(string AlternateID)

To get a Section using this approach, replace ALT_ID with the string representating a real alternate ID:

```
public void getSectionByAlternateID()
{
    var idSections =  client.GetSectionByAlternateID("ALT_ID").Data;

    //Optional: display  the ID of the Section(s) to the console
    foreach(DTOSection item in idSections)
        Console.WriteLine(item.ID.ToString());
    Console.Read();
}
// returns (service response) array of DTOSection
```

Display or Store Information

Viewing information contained in the selected asset will require being aware of properties associated with that asset, and doing some coding.

The code examples above show how to enter the alternate ID as a parameter and see the corresponding BUILDER ID(s) on the console. To display different properties, replace `item.ID.ToString()` with `item.<property>`, using `ToString()` as needed.

Full-line examples are provided in Output to Console, below.

Output to Console

To output a property to the console for viewing, use this code if the property is a string:

```
Console.WriteLine(item.<property>);
```

where `item` is the instance of the DTO object.

Alternatively, use the code below if the property is not a string:

```
Console.WriteLine(item.<property>.ToString() );
```

Store a Single Property in a Variable

To store an individual property in a variable, use this code:

```
var my<property> = item.<property>;
```

For example,

```
var myCI = item.CI;
```

Get Assets by Parent ID

You can use Parent ID to get assets from Organization down through Section Details. This topic will show code examples for getting assets that have a common parent ID. It will also discuss how to display or store desired properties.

If you want to use a different method to get inventory, you can select it from this list of the top-level options:

- Enter the GUID (`ID` property) of the asset. **Scope:** Works at all inventory levels, and for Section Details.
- Enter the Alternate ID of the asset. **Scope:** Facilities/Buildings or Sections only.
- Enter the GUID (`ID` property) of the parent asset. This approach will return all assets having that parent. **Scope:** Works at all inventory levels except the root of the inventory tree.
- Search by Name or name fragment. **Scope:** Organization and Site only.

This topic explains the third alternative.

About Selecting by Parent ID

Scope: All inventory levels except the root of the inventory tree (root Organization or root Site).

This approach can be used to store multiple assets in an array, to either be worked on sequentially or selected from. Alternatively, you can get a list of items returned and then make use of a different technique, such as getting by GUID, to see or store individual properties.

To get assets found directly below a given parent, you will need to enter the parent's GUID (`ID` property) as the argument to one of the following methods:

- GetOrganizationsByParentID. (Won't work at topmost Organization level)
- GetSitesByParentID
- GetComplexesByParentID
- GetFacilitiesByParentID
- GetSystemsByParentID
- GetComponentsByParentID
- GetSectionsByParentID
- GetSectionDetailsByParentID

Specifying an asset this way will usually return more than one result. See "Display or Store Information" on page 28 for how to display information associated with the assets returned.

> **Note about property name variations in the Parent ID property:** Although the phrase "ByParentID" is used in the service call at all inventory levels, the corresponding property is not always named ParentID in the data object. This can be helpful to know if you are looking for the ParentID property in a child object in order to use that GUID in a call.
>
> For details about how the property name that refers to the parent ID changes at different inventory levels, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

Code Examples: Get Inventory by Parent ID

A C# code example for each level of inventory is provided below.

GetOrganizationsByParentID(Guid id)

The sample code below facilitates identifying one or a small number of desired Organizations out of a list by getting each qualifying Organization, then displaying its number/name and its `ID` (GUID).

> **Note:** If you have a large number of Organizations, you may prefer to use the call "GetOrganizationsByParentIDPaged(Guid id, int skip, int take)" on the next page.

To use this code, replace PARENT_ID with the string representation of the appropriate GUID (the `ID` property of an Organization, or the `ParentID` property of a Site).

```
public void getOrganizationsByParentID()
{
  var theOrganizations = client.GetOrganizationsByParentID(Guid.Parse("PARENT_ID")).Data;

        // loop for output to display if desired
        foreach(var item in theOrganizations)
        {
            Console.WriteLine(item.Number + "-" + item.Name);
            Console.WriteLine(item.ID.ToString() + "\n");
        }
}
// returns array of DTOOrganization
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetOrganizationsByParentIDPaged(Guid id, int skip, int take)

The following code does the same thing as GetOrganizationsByParentID, except that it delivers results in separate pages (a paged collection) rather than an array.

To use this code,

- replace PARENT_ID with the string representation of the appropriate GUID (the `ID` property of an Organization, or the `ParentID` property of one of the Sites)
- set the integer indicating the starting point (`skip`; currently zero)
- set the integer defining page length (`take`; currently 100).

```
public void getOrganizationsByParentIDPaged()
{
   var theOrgs = client.GetOrganizationsByParentIDPaged(Guid.Parse("PARENT_ID"), 0, 100);

        // output to console to see a page of Organizations under the parent Org
        foreach(var item in theOrgs)
        {
            Console.WriteLine(item.Number + "-" + item.Name);
            Console.WriteLine(item.ID.ToString() + "\n");
        }
}
// returns paged collection of DTOOrganization
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetSitesByParentID(Guid id)

The example code below facilitates identifying one or a small number of desired Sites out of a list by getting each qualifying Site, then displaying its number/name and its `ID` (GUID).

To use this code, replace PARENT_ID with the string representation of the appropriate GUID (the `ID` property of an Organization, or the `ParentID` property of one of the Sites).

```
public void getSitesByParentID()
{
    var theSites = client.GetSitesByParentID(Guid.Parse("PARENT_ID")).Data;

    // loop for output to display if desired
    foreach (var item in theSites)
    {
        Console.WriteLine(item.Number + "-" + item.Name);
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOSite
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetComplexesByParentID(Guid id)

The example code below facilitates identifying one or a small number of desired Complexes out of a list by getting each qualifying Complex, then displaying its number/name and its `ID` (GUID).

To use this code, replace PARENT_ID with the string representation of a real parent GUID, which is the `ID` property of a Site.

```
public void getComplexesByParentID()
{
    var theComplexes = client.GetComplexesByParentID(Guid.Parse("PARENT_ID")).Data;

    // loop for output to display if desired
    foreach (var item in theComplexes)
    {
        Console.WriteLine(item.Number + "-" + item.Name);
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOComplex
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetFacilitiesByParentID(Guid id)

The sample code below facilitates identifying a number of desired Facilities out of a list by getting qualifying Facilities (in this example, up to 100 Facilities that all have the same Parent ID), then displaying each Facility's number/name and its `ID` (GUID).

To use this code, you can replace PARENT_ID with the string representation of the `ID` property of a Complex, of a Site, or of an Organization.

```
public void getFacilitiesByParentID()
{
  var theFacilities = client.GetFacilitiesByParentID(Guid.Parse("PARENT_ID"), 0, 100).Data;

    // loop for output to display if desired
    foreach (var item in theFacilities)
    {
        Console.WriteLine(item.Number + "-" + item.Name);
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOFacility
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetSystemsByParentID(Guid id)

The example code below facilitates identifying one or a small number of desired Systems out of a list by getting each qualifying System, then displaying its name and `ID` (GUID).

In this code, replace PARENT_ID with a real parent GUID (`ID` property), which is the `FacilityID` property of a System.

```
public void getSystemsByParentID()
{
    var theSystems = client.GetSystemsByParentID(Guid.Parse("PARENT_ID")).Data;

    // loop for output to display if desired
    foreach (var item in theSystems)
    {
        Console.WriteLine(item.Name);
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOSystem
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetComponentsByParentID(Guid id)

The example code below facilitates identifying one or a small number of desired Components out of a list by getting each qualifying Component, then displaying its name and `ID` (GUID).

In this code, replace PARENT_ID with the string representation of a real parent GUID (`ID` property), which is the `SystemID` property of a Component.

```
public void getComponentsByParentID()
{
    var theComponents = client.GetComponentsByParentID(Guid.Parse("PARENT_ID")).Data;

    // loop for output to display if desired
    foreach (var item in theComponents)
    {
        Console.WriteLine(item.Name);
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOComponent
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetSectionsByParentID(Guid id)

The example code below facilitates identifying one or a small number of desired Sections out of a list by getting each qualifying Section, then displaying its name and `ID` (GUID).

In this code, replace PARENT_ID with the string representation of a real parent GUID (`ID` property), which is the `ComponentID` property of a Section.

```
public void getSectionsByParentID()
{
    var theSections = client.GetSectionsByParentID(Guid.Parse("PARENT_ID")).Data;

    // loop for output to display if desired
    foreach (var item in theSections)
    {
        Console.WriteLine(item.Name);
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOSection
```

For more detail about Parent ID properties, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.

GetSectionDetailsByParentID(Guid id)

The example code below facilitates identifying one or a small number of desired Section Details out of a list by getting each qualifying Section Detail, then displaying its name and `ID` (GUID).

In this code, replace PARENT_ID with the string representation of a real parent GUID (`ID` property), which is the `SectionID` property of a Section Detail.

```
public void getSectionDetailsByParentID()
{
    var theDetails = client.GetSectionDetailsByParentID(Guid.Parse("PARENT_ID")).Data;

    // loop for output to display if desired
    foreach (var item in theDetails)
    {
        Console.WriteLine(item.ID.ToString() + "\n");
    }
}
// returns array of DTOSectionDetail
```

Special Case: Getting All of the Sections in a Site

In addition to getting the Sections present in a parent Component, you can also have the option to get all Sections present in a given Site.

GetSectionsBySiteID(Guid ID, int skip, int take)

The example code below facilitates identifying Sections contained at a given Site, and can be used to view a page of Sections on the console. Alternatively (not shown), you can build a loop to accumulate all of the Sections into an array.

To use this code, replace SITE_ID with the string representation of a Site's GUID (`ID` property). Adjust the value of the integer showing the starting point (`skip`) if you iterate the code.

```
public void getSectionsBySiteID()
{
    var theSections = client.GetSectionsBySiteID(Guid.Parse("SITE_ID"), 0, 100).Data;

        // output to console to see name and ID of a page of Sections:
        foreach (var item in theSections)
        {
            Console.WriteLine(item.Name);
            Console.WriteLine(item.ID.ToString() + "\n");
        }
}
// returns array of DTOSection
```

Display or Store Information

Output to Console

The code to output a property of an inventory item to the console for viewing will be in this pattern:

```
Console.WriteLine(array[x].<property>);
```

where `array` is the array of DTO objects returned by the call `Get<Inventory level>sByParentID("PARENT_ID")`, `x` is the index number in the array for the desired inventory object, and <property> is the desired property of the inventory item that you want shown, such as `Name`.

Alternatively, use the code pattern below if the property being output is not a string:

```
Console.WriteLine(array[x].<property>.ToString() );
```

Store a Single Property in a Variable

To store an individual property in a variable, use this code pattern:

```
var my<property> = array[x].<property>;
```

where `array` is the array of DTO objects returned by `Get<Inventory level>sByParentID("PARENT_ID")`, and `x` is the index number in the array for the desired inventory object.

For example,

```
var myCI = array[x].CI;
```

In Depth:
Parent ID Properties at All Inventory Levels

This topic provides, for each inventory level, detailed information about the `ParentID` property or its equivalent.

The information is provided here especially to help users specify GUID parameters for the following methods:

- GetSystemsByParentID(Guid id)
- GetComponentsByParentID(Guid id)
- GetSectionsByParentID(Guid id)
- GetSectionDetailsByParentID(Guid id)

From the System level down through Section Detail, the property name corresponding to `ParentID` is different for assets at each level. From Organization through Facility, the property name is `ParentID`.

The sections below give detail for each inventory level.

Organization

At this inventory level, the property name is ParentID. The parent of an Organization will be one of the following:

a. An Organization.
b. Null. If the Organization's `ParentID` is null, it means the Organization is at the top of the inventory tree (i.e., is the "root node"). It has no parent, and no sibling.

Site

At this inventory level, the property name is `ParentID`. The parent of a Site will be of the following:

a. An Organization.
b. Null. If the Site's `ParentID` is null, it means the Site is at the top of the inventory tree (also called "root node"); it has no parent, and no sibling.

Complex

At this inventory level, the property name is `ParentID`. The parent of a Complex will always be a Site.

Facility

At this inventory level, the property name is `ParentID`. The parent of a Facility will be one of the following:

a. A Complex created and named by a user.
b. The "Unassigned" Complex. A Facility that is not placed in a user-created Complex can be found via the BUILDER API in the Complex named "Unassigned". In the BUILDER Web interface, Complexes in the "Unassigned" Complex are displayed directly under the Site.

System

At this inventory level, the property name is `FacilityID`. The parent of a System will always be a Facility/Building.

Component

At this inventory level, the property name is `SystemID`. The parent of a Component will always be a System.

Section

At this inventory level, the property name is `ComponentID`. The parent of a Section will always be a Component.

SectionDetail

At this inventory level, the property name is `SectionID`. The parent of a Section Detail will always be a Section.

Search for Assets by Name

This topic covers searching for an asset by its name, or by name fragment, which works only for Organizations, Sites, and Complexes. The topic also discusses how to display or store desired properties.

If you want to use a different method to get inventory, you can select it from this list of the top-level options:

- Enter the GUID (`ID` property) of the asset. **Scope:** Works at all inventory levels, and for Section Details.
- Enter the Alternate ID of the asset. **Scope:** Facilities/Buildings or Sections only.
- Enter the GUID (`ID` property) of the parent asset. This approach will return all assets having that parent. **Scope:** Works at all inventory levels except the root of the inventory tree.
- Search by Name or name fragment. **Scope:** Organization and Site only.

This topic explains the last alternative.

About Searching by Name

Scope: Organization, Site, or Complex only.

To search by name, you will need to enter a name or name fragment as the argument. The string search is *not* case sensitive.

> **Note:** This way of specifying an asset may return more than one result.

When used with a valid name or name fragment, search by name will get one or more Organizations if any are found whose name includes the search fragment.

Code Examples: Search by Name

A C# code example is provided below for each level of inventory accommodated.

Short Example

An example of a complete service call is

```
client.SearchOrganizationNames("aspen");
```

This will get all Organizations containing the word or word fragment "Aspen" or "aspen".

Sample code immediately below includes code to display `ID` (GUID) and the full Organization name to the console. Also, the section "Display or Store Information" on the facing page explains how to display or store information associated with asset.

SearchOrganizationNames(string nameFragment)

The code below gets each qualifying Organization, stores it in the array `orgs`, then displays the `ID` property (GUID) and name. (GUID is placed first in the console output because presumably the GUID will always be the same length.)

To use this code, replace NAME_FRAGMENT with your desired search string. Qualifying Organizations will be those whose `Name` property contains your desired search string. The search is not case sensitive.

```
public void getOrgsByName()
{
    var orgs = client.SearchOrganizationNames("NAME_FRAGMENT").Data;

    // Optional: display the name and ID of the Org(s) with a colon as the separator
    foreach (var org in orgs)
    {
        Console.WriteLine(org.ID.ToString() + ":" + org.Name );
    }
    Console.Read();
}
// returns array of DTOOrganization
```

SearchOrganizationNamesPaged(string nameFragment, int skip, int take)

This call returns a page of DTO objects from an array of DTOOrganization. Replace NAME_FRAGMENT with your desired search string. (The search is not case sensitive.)

The code sample shown below returns the first 100 results that have a Name property containing your desired search string (or all of the results, if there are fewer than 100 matches).

```
public void getPagedOrgsByName()
{
   var orgs = client.SearchOrganizationNamesPaged("NAME_FRAGMENT", 0, 100);

   // Optional: display the name and ID of the Org(s) with a colon as the separator
    foreach (var org in orgs)
    {
        Console.WriteLine(org.ID.ToString() + ":" + org.Name );
    }
    Console.Read();
}
// returns paged collection of DTOOrganization
```

SearchSiteNames(string nameFragment)

The code below gets each qualifying Site, stores it in the array `sites`, then displays the `ID` property (GUID) and name. (GUID is placed first in the console output because presumably the GUID will always be the same length.)

To use the code, replace NAME_FRAGMENT with your desired search string. Qualifying Sites will be those whose `Name` property contains your desired search string. The search is not case sensitive.

```
public void getSitesByName()
{
    var sites = client.SearchSiteNames("NAME_FRAGMENT").Data;

    // Optional: display the name and ID of the Site(s) with a colon as the separator
    foreach (var site in sites)
    {
        Console.WriteLine(site.ID.ToString() + ":" + site.Name );
    }
    Console.Read();
}
// returns array of DTOSite
```

SearchComplexNames(string nameFragment)

The code below gets each qualifying Complex, stores it in the array `complexes`, then displays the `ID` property (GUID) and name. (GUID is placed first in the console output because presumably the GUID will always be the same length.)

To use this code, replace NAME_FRAGMENT with your desired search string. Qualifying Complexes will be those whose `Name` property contains your desired search string. The search is not case sensitive.

```
public void getComplexesByName()
{
    var complexes = client.SearchComplexNames("NAME_FRAGMENT").Data;

    // Optional: display name & ID of Complex(es) with a colon as separator
    foreach (var complex in complexes)
    {
        Console.WriteLine(complex.ID.ToString() + ":" + complex.Name );
    }
    Console.Read();
}
// returns array of DTOComplex
```

Display or Store Information

Viewing information contained in the selected asset will require being aware of properties associated with that asset, and doing some coding.

In the code examples below, `item` represents the DTO instance you are getting.

Output to Console

To output a property to the console for viewing, use this code if the property is a string:

```
Console.WriteLine(item.<property>);
```

where `item` is the instance of the DTO object.

Alternatively, use the code below if the property is not a string:

```
Console.WriteLine(item.<property>.ToString() );
```

Store a Single Property in a Variable

To store an individual property in a variable, use this code:

```
var my<property> = item.<property>;
```

For example,

```
var myCI = item.CI;
```

Asset Properties: Building/Facility

This topic presents calls relating to Building properties.

Code Examples: Get Selected Facility Properties

GetFacilityBuildingTypes(int skip, int take)

The values for `skip` and `take` in the code below suffice to cover available standard Building Types provided in BUILDER.

```
public void getFacilityBuildingTypes()
{
    var response = client.GetFacilityBuildingTypes(0, 25);
}
// returns array of DTOFacilityBuildingType
```

GetFacilityConstructionTypes(int skip, int take)

The values for `skip` and `take` in the code below suffice to cover available standard Construction Types provided in BUILDER.

```
public void getFacilityConstructionTypes()
{
    var response = client.GetFacilityConstructionTypes(0, 25);
}
// returns array of DTOFacilityConstructionType
```

GetFacilityUseTypes(int skip, int take)

The values for `skip` and `take` in the code below suffice to cover available standard types of Building Use provided in BUILDER.

```
public void getFacilityUseTypes()
{
    var response = client.GetFacilityUseTypes(0, 25);
}
// returns array of DTOFacilityUseType
```

GetFacilityBuildingStatuses( )

You can use this call to see the options available for Building Status. Each of the elements returned in the array will have an ID, and that ID corresponds to the value of that status used for the BuildingStatus enum. A description of the BuildingStatus enum can be found at the entry for "CreateFacility(DTOFacility facility)" on page 43.

The call takes no parameter.

```
public void getFacilityBuildingStatuses()
{
    var response = client.GetFacilityBuildingStatuses();
}
// returns array of DTOBuildingFacilityStatus
```

Asset Properties: Other than Facility

This topic presents calls relating to properties of BUILDER assets other than at the Facility level. The calls are of two types:

- Generally, the calls shown below that do not have an ID parameter input serve as reference resources for lists of asset properties. You can use these calls to show lists of options for certain asset properties such as Paint Type.
- Other calls relate to retrieving detailed information about a specific property. For example, GetCMCType is such a call. It receives a parameter indicating which CMCType you want details about and returns that CMCType object.

Also shown is the call to get a unit of measure (UOM).

Calls you can use to access non-facility asset properties are as follows:

- GetUnitOfMeasure
- GetSystemTypes
- GetComponentTypes
- GetComponentMaterialCategories
- GetMaterialCategoryCMCs
- GetSubComponents
- GetUniformatSection
- GetCMCType
- GetPaintTypes
- GetPaintType

Code Examples: Get Selected Non-Facility Properties

Code examples are in C#.

GetUnitOfMeasure(int measureId)

When a unit of measure (UOM) is specified as a property of an asset, it is the ID of the unit of measure that is listed. To verify what the UOM corresponding to the ID is, you can look it up with the GetUnitOfMeasure call.

To use the code below, replace the digit in the parameter with the ID of the UOM you want to identify.

```
public void getUnitOfMeasure()
{
    var uom = client.GetUnitOfMeasure(105);
}
// returns DTOUnitOfMeasure for the integer provided in the parameter
```

A full list of UOM IDs is provided in Appendix A.

GetSystemTypes( )

This call takes no parameter. The information returned can be used to identify one or more values to use as parameter for the GetComponentTypes call.

> **Note:** System types are also listed in the first column of Appendix B.

```
public void getSystemTypes()
{
    var response = client.GetSystemTypes();
}
// returns array of DTOSystemType
```

GetComponentTypes(int SystemTypeID)

Use this call to get the Component types associated with any given System type.

The information returned can be used to identify one or more ComponentTypeID values to use as parameter for the GetComponentMaterialCategories call, or as the first parameter for the GetMaterialCategoryCMCs call.

To use the code below, replace SYS_TYPE_ID with the `ID` of the System type (DTOSystemType) you want to see Component types for. Alternatively, you can use the `SystemTypeID` property of a given System. If needed, you can get System types using the call GetSystemTypes.

Note: Component types are also listed in the first column of Appendix C .

```
public void getComponentTypes()
{
    var response = client.GetComponentTypes("SYS_TYPE_ID");
}
// returns array of DTOComponentType
```

GetComponentMaterialCategories(int ComponentTypeID)

Use this call to get the material categories associated with any given Component type.

Information returned from this call can be used to supply the second parameter of the call GetMaterialCategoryCMCs.

To use the code below, replace COMP_TYPE_ID with the `ID` of the Component type (DTOComponentType) you want material categories for. Alternatively, you can use the `ComponentTypeID` property of a given Component. If needed, you can get the Component types for any given System type using the call GetComponentTypes.

```
public void getComponentMaterialCategories()
{
    var response = client.GetComponentMaterialCategories("COMP_TYPE_ID");
}
// returns array of DTOSectionMaterialCategory
```

GetMaterialCategoryCMCs(int ComponentTypeID, int MaterialCategoryID)

This call requires two parameters. The DTOCMCTypes returned in the array will have properties such as `MaterialCategory`, `UnitOfMeasure`, and `ComponentSubType`.

To use the code shown below,

- Replace COMP_TYPE_ID with the `ID` of the Component type (DTOComponentType) you want material category CMCs for. Alternatively, you can use the

`ComponentTypeID` property of a given Component. If needed, you can get the Component types for any given System type using the call GetComponentTypes.

- Replace MATERIAL_CATEGORY_ID with the `ID` property of a Section material category (DTOSectionMaterialCategory). If needed, you can get Section material categories using the call GetComponentMaterialCategories.

```
public void getMaterialCategoryCMCs()
{
    var response = client.GetMaterialCategoryCMCs("COMP_TYPE_ID", "MATERIAL_CATEGORY_ID");
}
// returns array of DTOCMCType
```

GetSubComponents(int CMC)

The parameter needed for this call is called CMC in the signature, but what you need to supply is a CMCID.

A CMC type will have a five-digit int CMCID, and there are over 1000 possibilities. Arrays of CMCID's can be found as properties of DTOSectionTypes.

To use the call given below, replace cmcInt with a valid CMCID.

```
public void getSubComponents()
{
    var response = client.GetSubComponents(cmcInt);
}
// returns array of DTOSubcomponent
```

GetUniformatSection(string UniformatID)

A Uniformat section will have a four-digit ID, and there are over 600 possibilities. To obtain a list of Uniformat ID's, see Appendix D. The numbers you want to select from are the "Component ID" numbers in the first column.

> **IMPORTANT:** For this call, you need to input the UniformatID parameter as a string.

The code below shows an example UniformatID string; you will need to replace it with the ID of the Uniformat Section you want to get.

```
public void getUniformatSection()
{
    var response = client.GetUniformatSection("2011");
}
// returns DTOSectionType
```

GetCMCType(int CMCID)

A CMC type will have a five-digit int CMCID, and there are over 1000 possibilities. For the parameter, you can find CMCIDs by examining Sections because each Section has a `CMCID` property.

To use the sample code, replace CMCID with an actual integer CMCID.

```
public void getCMCType()
{
    var response = client.GetCMCType(CMCID);
}
// returns DTOCMCType
```

GetPaintTypes( )

This is a reference list call that returns an array of all paint types. The call takes no parameter.

The output can be used to locate a paint type ID, which is a required parameter for the call "GetPaintType(int id)" below. Alternatively, there is also an Appendix to this document that lists paint type ID numbers.

```
public void getPaintTypes()
{
    var response = client.GetPaintTypes();
}
// returns array of DTOPaintType
```

GetPaintType(int id)

To get details about a single paint type, you can use this call, supplying the PaintType ID in the parameter. This ID will be a two- or three-digit integer.

The identifier needed for the parameter for this call can be found in Appendix C, or obtained from the array returned by"GetPaintTypes( )" above.

In the code below, the PaintTypeID of 10 is supplied as an example. The PaintTypeIDs range from 10 to 420 and are all multiples of 10.

```
public void getPaintType()
{
    var response = client.GetPaintType(10);
}
// returns DTOPaintType
```

Lock and Unlock Inventory

Using the BUILDER API, you can lock and unlock individual Systems.

An example of when you might do this is when one or more Systems are checked out and back in to BuilderRED or another remote device designed to assist with taking BUILDER inventory or performing inspections.

Code Examples: Lock and Unlock Systems

Code examples are in C#.

LockSystem(Guid id)

This call locks a System so that it can not be edited in BUILDER until after the System is unlocked again.

To use the code shown here, replace SYS_ID with the string representation of a System's GUID.

```
public void lockSystem()
{
    var response = client.LockSystem(Guid.Parse("SYS_ID"));
}
// returns Boolean: true = unlocked system successfully locked
```

A return of false from the LockSystem call can mean that an unlocked system has not been successfully locked, or it can mean that the system was already locked (and still is).

UnlockSystem(Guid id)

This call unlocks a System, such as when it has been locked against editing using the LockSystem call.

To use the code shown here, replace SYS_ID with the string representation of a System's GUID.

```
public void unlockSystem()
{
    var response = client.UnlockSystem(Guid.Parse("SYS_ID"));
}
// returns Boolean: true = system successfully unlocked
```

A return of false from the UnlockSystem call can mean that a locked system has not been successfully unlocked, or it can mean that the system was already unlocked (and still is).

Create Inventory

Subject to your BUILDER user permission level and scope, you can add assets from Organization down through Section Details using BUILDER API.

After you create an asset, you have the option to associate one or more attachments with it. For this, see "Add Attachment" on page 88.

The "Create" Pattern

The pattern for creating a new instance of an object in the API involves these steps:

1. Declare a new instance of the DTO object, using var.
2. Assign values to the object's properties.
3. Call the create function, where the parameter you pass in is the DTO object instance that you assigned in Step 1.

Code Examples: Create Inventory

A C# code example for each level of inventory is provided below.

> **IMPORTANT:** In the Create calls, do NOT use a variable (such as "client", which is used in the majority of the code examples provided) as a substitute for the service reference name.
>
> Because of the possibility of multiple BUILDER instances, spell out a service reference name used by your organization when creating a new asset.

CreateOrganization(DTOOrganization org)

To add an Organization to the inventory tree, you will need to write a program to create an instance of an Organization and assign values to its properties.

> **Note:** If the instance is root of the inventory tree, its `ParentID` property is null.

```
public void createOrganization()
{
    // Step 1. Declare new instance of the DTO object
    // Actual service reference name is required--no variable
    var theOrganization = new <service reference name>.DTOOrganization();
```

```
    // Step 2. Set properties. Name and number combo must be unique in inventory tree
    theOrganization.Name = "new Org";
    theOrganization.Number = "01";
    theOrganization.ParentID = Guid.Parse("PARENT_ID"); //parent Org's GUID, or null if root

    // Step 3. Make the create API call
    //   where parameter is name of the DTOOrganization instance you created in Step 1
    Guid orgGuid = client.CreateOrganization(theOrganization).Data;
}
// returns guid (ID property) of the new organization
```

CreateSite(DTOSite site)

To add a Site to the inventory tree, you will need to write a program to create an instance of a Site and assign values to its properties.

> **Note:** If the Site is root of the inventory tree, its `ParentID` property is null.

```
public void createSite()
{
   // Step 1. Declare new instance of the DTO object
   // Actual service reference name required--no variable
    var theSite = new <service reference name>.DTOSite();

    // Step 2. Set properties. Name and number combo must be unique in the Org
    theSite.Name = "new site";
    theSite.Number = "01";
    theSite.ParentID = Guid.Parse("PARENT_ID"); // REQUIRED: GUID of parent Org,
       //or null if Site is root

    // Step 3. Call the create service call
    //   where the parameter is the name of the instance you created in Step 1.
    Guid facGuid = client.CreateSite(theSite).Data
}
// returns guid (ID property) of the new site
```

CreateComplex(DTOComplex complex)

To add a Complex to the inventory tree, you will need to write a program to create an instance of DTOComplex and assign values to its properties.

```
public void createComplex()
{
    // Step 1. Declare new instance of the DTO object
    var theComplex = new <service reference name>.DTOComplex();

    // Step 2. Set desired properties
    theComplex.Name = "new comp";   // name and number combo must be unique in the Site
    theComplex.Number = "01";
    theComplex.ParentID = Guid.Parse("PARENT_ID");  // REQUIRED: GUID of parent Site

    // Step 3. Call the create service call
    Guid cpxGuid = client.CreateComplex(theComplex).Data;
}
// returns guid (ID property) of the new complex
```

CreateFacility(DTOFacility facility)

> **Note:** If the Facility has not been placed into a named Complex, it is in the Complex "Unassigned".

To add a Facility to the inventory tree, you will need to write a program to create an instance of DTOFacility and assign values to its properties.

The value to use for the required `ComplexID` property will depend on whether the Facility has been placed into a named Complex:

- If the Facility has been placed into a named Complex, then the value needs to be the GUID of that Complex.
- If it has not, then you need to use the value for the "Unassigned" Complex. One way (but not the only way) to find this value is to run the call GetComplexesByParentID. For the parameter, use the GUID of the Site that will contain the new Facility. In the results, look for the Complex with the value "Unassigned" for the `Name` property.

The value to use for the required property `BuildingStatus` is governed by the BuildingStatus enum, which maps values to status choices as follows:

1 = Active

2 = Vacant

3 = ToBeTransferred

4 = ToBeDemolished

5 = ToBeAcquired

6 = ToBeBuilt

7 = Transferred

8 = Demolished

9 = Non-Functional

10 = Semi-Active

11 = Excess

12 = Caretaker

13 = Closed

14 = Outgranted

15 = Surplus

16 = Environmental Hold

17 = Disposed

If no definition is found, the status of this property will be null.

```
public void createFacility()
{
    // Step 1. Declare new instance of the DTO object
    var theFacility = new <service reference name>.DTOFacility();

    // Step 2. Set desired properties
    theFacility.Name = "new fac";  //name and number combo must be unique in the Site
    theFacility.Number = "01";
    theFacility.ComplexID = Guid.Parse("CPX_ID"); // REQUIRED: GUID of parent Complex

    // Step 3. Call the create service call
    Guid facGuid = client.CreateFacility(theFacility).Data;
}
// returns guid (ID property) of the new facility
```

CreateSystem(DTOSystem system)

To add a System to the inventory tree, you will need to write a program to create an instance of a System and assign values to its properties.

```
public void createSystem()
{
    // Step 1. Declare new instance of the DTO object
    var theSystem = new <service reference name>.DTOSystem();

    // for the above, actual service reference name required--no variable

    // Step 2. Set desired properties
    theSystem.Name = "new system";
    theSystem.FacilityID = Guid.Parse("FAC_ID");  // REQUIRED: GUID of parent Facility

    // Step 3. Call the create service call
    Guid sysGuid = client.CreateSystem(theSystem).Data;
}
// returns guid (ID property) of the new system
```

CreateComponent(DTOComponent component)

To add a Component to the inventory tree, you will first need to write a program to create an instance of a Component and assign values to its properties.

```
public void createComponent()
{
    // Step 1. Declare new instance of the DTO object
    // -->must use actual service reference name, not variable
    var theComponent = new <service reference name>.DTOComponent();
```

```
    // Step 2. Set desired properties
    theComponent.Name = "new comp";              // name must be unique in the System
    theComponent.SystemID = Guid.Parse("SYS_ID");  // REQUIRED: GUID of parent System
    // Step 3. Call the create service call
    Guid CompGuid = client.CreateComponent(theComponent).Data
}
// returns guid (ID property) of the new component
```

CreateSection(DTOSection section)

To add a Section to the inventory tree, you will need to write a program to create an instance of a Section and assign values to its properties.

```
public void createSection()
{
    // Step 1. Declare new instance of the DTO object
    var theSection = new <service reference name>.DTOSection();

    // for the above, actual service reference name required--no variable

    // Step 2. Set desired properties
    theSection.Name = "new sec";  //name must be unique in the Component
    theSection.ComponentID = Guid.Parse("COMP_ID"); // REQUIRED: GUID of parent Component

    // Step 3. Call the create service call
    Guid secGuid = client.CreateSection(theSection).Data;
}
// returns guid (ID property) of the new section
```

CreateSectionDetail(DTOSectionDetail detail)

To add a Section Detail to the inventory tree, you will need to write a program to create an instance of a Section Detail and assign values to its properties.

```
public void createSectionDetail()
{
    // Step 1. Declare new instance of  DTO object
    var SectionDetail = new <service reference name>.DTOSectionDetail();

    // for the above, actual service reference name required--no variable

    // Step 2. Set desired properties
    SectionDetail.SectionID = (Guid.Parse("SEC_ID")); // REQUIRED: GUID of parent Section

    // Step 3. Call the create service call
    Guid orgGuid = client.CreateSectionDetail(theSectionDetail).Data;
}
// returns guid (ID property) of the new section detail
```

Update Inventory

Subject to your BUILDER user permission level and scope, you can update inventory from Organization down through Section Details using BUILDER API.

The "Update" Pattern

The pattern for updating an asset or an instance of another object in the API involves these steps:

1. Declare a new instance of the DTO object, using var.
2. Call the get function, using the GUID of the thing you want to update, and set the new instance equal to what the get function returns. This populates the new instance with the data currently in the thing to be updated.
3. Assign new values to properties you want to change (or add additional properties).
4. Call the update function with the variable name of the new DTO object instance (as declared in Step 1) as the parameter.

An example of Step 4 is

```
var response = client.UpdateFacility(theFacility);
```

There are ways to make the update code more elegant or compact–for example, the first two steps can be combined into one complex call–but the four steps above are used for the inventory code examples for simplicity and clarity.

Code Examples: Update Inventory

A C# code example for each level of inventory is provided below.

UpdateOrganization(DTOOrganization org)

To use the code below, replace ORG_ID with the string representation of an Organization's GUID and set appropriate properties.

```
public void updateOrganization()
{
    //Step 1. Declare new instance of DTO object
    // Actual service reference name is required here--no variable
    var theOrganization = new <service reference name>.DTOOrganization();
```

```
    //Step 2. Populate it using the get function
    theOrganization = client.GetOrganization(Guid.Parse("ORG_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theOrganization.Name = "<new org name>";

    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateOrganization(theOrganization);

    //Optional error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateSite(DTOSite site)

To use the code below, replace SITE_ID with the string representation of a Site's GUID and set appropriate properties:

```
public void updateSite()
{
    //Step 1. Declare new instance of DTO object
    var theSite = new <service reference name>.DTOSite();

    //Step 2. Populate it using the get function
    theSite = client.GetSite(Guid.Parse("SITE_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theSite.Name = "<new site name>";

    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateSite(theSite);

    //error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateComplex(DTOComplex complex)

To use the code below, replace CPX_ID with the string representation of a Complex's GUID and set appropriate properties.

```
public void updateComplex()
{
    //Step 1. Declare new instance of DTO object
    var theComplex = new <service reference name>.DTOComplex();

    //Step 2. Populate it using the get function
    theComplex = client.GetComplex(Guid.Parse("CPX_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theComplex.Name = "<new complex name>";
```

```
    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateComplex(theComplex);

    //Optional error message code
    if (response.Success == false)
       Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateFacility(DTOFacility facility)

To use the code below, replace FAC_ID with the string representation of a Facility's GUID and set appropriate properties.

```
public void updateFacility()
{
    //Step 1. Declare new instance of DTO object
    var theFacility = new <service reference name>.DTOFacility();

    //Step 2. Populate it using the get function
    theFacility = client.GetFacility(Guid.Parse("FAC_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theFacility.Name = "<new facility name>";

    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateFacility(theFacility);

    //Optional error message code
    if (response.Success == false)
       Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateFacility(DTOFacility facility, bool preserveExistingRecords)

Note: Sometimes referred to as UpdateFacilityPreserveExistingRecords.

Currently this call is coded to work the same as the standard UpdateFacility(DTOFacility facility) call. If you do decide to use this version of the UpdateFacility call, the Facility will need to be in a named Complex (not the "Unassigned" Complex) for this call to work.

UpdateSystem(DTOSystem system)

To use the code below, replace SYS_ID with the string representation of a System's GUID and set appropriate properties.

```
public void updateSystem()
{
    //Step 1. Declare new instance of DTO object
    // Actual service reference name is required here--no variable
    var theSystem = new <service reference name>.DTOSystem();
```

```
    //Step 2. Populate it using the get function
    theSystem = client.GetSystem(Guid.Parse("SYS_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theSystem.Name = "<new system name>";

    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateSystem(theSystem);

    //Optional error message code
    if (response.Success == false)
       Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateComponent(DTOComponent component)

To use the code below, replace COMP_ID with the string representation of a Component's GUID and set appropriate properties.

```
public void updateComponent()
{
    //Step 1. Declare new instance of DTO object
    var theComponent = new <service reference name>.DTOComponent();

    //Step 2. Populate it using the get function
    theComponent = client.GetComponent(Guid.Parse("COMP_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theComponent.Name = "<new component name>";

    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateComponent(theComponent);

    //Optional error message code
    if (response.Success == false)
       Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateSection(DTOSection section)

To use the code below, replace SEC_ID with the string representation of a Section's GUID and set appropriate properties.

```
public void updateSection()
{
     //Step 1. Declare new instance of DTO object
    var theSection = new <service reference name>.DTOSection();

    //Step 2. Populate it using the get function
    theSection = client.GetSection((Guid.Parse("SEC_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theSection.Name = "<new section name>";
```

```
    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateSection(theSection);

    //Optional error message code
    if (response.Success == false)
       Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

UpdateSectionDetail(DTOSectionDetail detail)

To use the code below, replace SEC_DETAIL_ID with the string representation of a Section Detail's GUID and set appropriate properties:

```
public void updateSectionDetail()
{
    //Step 1. Declare new instance of DTO object
    var theSectionDetail = new <service reference name> DTOSectionDetail();

    //Step 2. Populate it using the get function
    theSectionDetail = client.GetSectionDetail(Guid.Parse("SEC_DETAIL_ID")).Data;

    //Step 3. Assign new values to properties that you want to change, for EXAMPLE:
    theSection.Name = "<new section detail name>";

    //Step 4. Call update function with the new DTO instance as the parameter
    var response = client.UpdateSectionDetail(theSectionDetail);

    //Optional error message code
    if (response.Success == false)
       Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

Updating Attachments

To update an asset's attachment image(s), you will need to delete the old attachment and add the new attachment. For deleting an attachment, see "DeleteAttachment(Guid attachmentId)" on page 92. For adding attachments, see AddAttachment (if the attachment is a .bmp file) or AddAttachmentWithFileExtension.

Inventory Rollup

Rolling up inventory will accumulate upwards the aggregate amount of Replacement Cost. Rollup will also aggregate the performance metrics upwards through the tree.

Scope: Inventory rollup can be done at three different levels: Global (the entire inventory tree); Site; or Facility.

Rollups assign a GUID to the rollup performed, and this is what is returned by an Initiate<Level>Rollup call, where <Level> is Global, Site, or Facility. The Guid can then be fed into the GetRollupStatus call.

Code Examples: Rollup Options

Code examples are in C#.

> **Note:** In each of the rollups described below (Global Rollup, Site Rollup, and Facility Rollup), the rollup examples already include code that implements the GetRollupStatus call to display the rollup state on the console.

InitiateGlobalRollup()

> **Permissions Note:** Global rollup can be performed only by a BUILDER Administrator.

Here is the code to perform a global rollup. This call has no parameters.

```
public void initiateGlobalRollup()
{
    // use InitiateGlobalRollup to get the guid for the rollup
    Guid rollupGuid = client.InitiateGlobalRollup().Data;

    // display rollup status on console
    Console.WriteLine(client.GetRollupStatus(rollupGuid).Data);
    Console.Read();
}
```

InitiateSiteRollup(Guid id)

To use the code below to roll up a Site, replace SITE_ID with the string representation of the Site's GUID.

```
public void initiateSiteRollup()
{
    // use InitiateSiteRollup to get the guid for the rollup
    Guid rollupGuid = client.InitiateSiteRollup(Guid.Parse("SITE_ID"));

    // display rollup status on console
    Console.WriteLine(client.GetRollupStatus(rollupGuid).Data);
    Console.Read();
}
```

InitiateFacilityRollup(Guid id)

To use the code below to roll up a Facility, replace FAC_ID with the string representation of a Facility's GUID.

```
public void initiateFacilityRollup()
{
    // use InitiateFacilityRollup to get the guid of the rollup
    Guid rollupGuid = client.InitiateFacilityRollup(Guid.Parse("FAC_ID"));

    // display rollup status on console
    Console.WriteLine(client.GetRollupStatus(rollupGuid).Data);
    Console.Read();
}
```

Code Example: Get Rollup Status

Code examples are in C#.

> **Note:** In each of the rollups described above (Global Rollup, Site Rollup, and Facility Rollup), the rollup examples already include code that implements the GetRollupStatus call to display the rollup state on the console.

GetRollupStatus(Guid id)

The call for getting the status (state) of a rollup is:

```
var status = client.GetRollupStatus(<GUID of rollup>);
```

The call returns the rollup state, which can be "Running", "Completed", "Failed", or "NotFound".

Fuller code examples with console output can be found as part of the code examples for initiating rollups.

Delete Inventory

Subject to your BUILDER user permission level and scope, you can delete assets from Organization down through Section Details using BUILDER API.

Precautions

> **WARNING:** Deleting an asset deletes all inventory contained in that asset. For example, deleting a Site will delete all inventory (Complexes, Buildings,

Systems, Components, and Sections) in that Site. **Deletion is a significant step to take and should only be done when certain that you wish to clear the entire inventory of the selected asset.**

Best Practice Recommendation: Where a Building/Facility is involved, you have the alternative to change the Facility's Building Status property instead of deleting the Facility (see details at "DeleteFacility(Guid id)" on the next page). For all asset levels, making frequent backups of the inventory database will protect you from significant data losses if an unintended deletion is performed.

Code Examples: Delete Inventory

A C# code example for each level of inventory is provided below.

Each of these calls returns a Boolean indicating whether the asset was deleted (true), or either not deleted or not found (false). Another way to confirm that the asset is deleted or not present is to follow the Delete call with a GetOrganization, GetSite, GetComplex, GetFacility, GetSystem, GetComponent, GetSection, or GetSectionDetail API call–you should receive an error message as a property of the service response.

DeleteOrganization(Guid id)

WARNING: Deleting an Organization will also delete all inventory (Complexes, Buildings, Systems, Components, and Sections) in that Organization. This is a significant step to take and should only be done when you are certain that you wish to clear the entire inventory of the Organization you have selected.

Replace ORG_ID with the string representation of an Organization's GUID to delete the Organization.

```
public void deleteOrganization()
{
    var response = client.DeleteOrganization(Guid.Parse("ORG_ID"));
}
// returns Boolean: true = organization deleted
```

DeleteSite(Guid id)

WARNING: Deleting a Site deletes all inventory (Complexes, Buildings, Systems, Components, and Sections) in that Site. This is a significant step to take and should only be done when certain that you wish to clear the entire inventory of the selected Site.

Replace SITE_ID with the string representation of a Site's GUID to delete the Site.

```
public void deleteSite()
{
    var response = client.DeleteSite(Guid.Parse("SITE_ID"));
}
// returns Boolean: true = site deleted
```

DeleteComplex(Guid id)

WARNING: Deleting a Complex deletes all inventory (Buildings, Systems, Components, and Sections) in the Complex. This is a significant step to take and should only be done when you are certain that you wish to clear the entire inventory of the Complex you have selected.

Replace CPX_ID with the string representation of a Complex's GUID to delete the Complex.

```
public void deleteComplex()
{
    var response = client.DeleteComplex(Guid.Parse("CPX_ID"));
}
// returns Boolean: true = complex deleted
```

DeleteFacility(Guid id)

WARNING: Deleting a Building deletes all the inventory (Systems, Components, and Sections) in the Building. This is a significant step to take and should only be done when you are certain that you wish to clear the entire inventory of the Building you have selected.

Best Practice Recommendation: Instead of deleting a Building, you can use the Current Status property to mark Buildings that have been demolished or transferred to another owner. With this method, the Building's records remain in the database, but the Building is ignored in processes that should only consider current buildings. To change the Building Status, you can first use the call "GetFacilityBuildingStatuses( )" on page 35 to see the Building Status options available, then use "UpdateFacility(DTOFacility facility)" on page 48 to set the Facility's `BuildingStatus` property to the desired value.

Replace FAC_ID with the string representation of a Facility's GUID to delete the Facility.

```
public void deleteFacility()
{
    var response = client.DeleteFacility(Guid.Parse("FAC_ID"));
}
// returns Boolean: true = facility deleted
```

DeleteSystem(Guid id)

WARNING: Deleting a System will delete all inventory (Components and Sections) in the System. This is a significant step to take and should only be done when you are certain that you wish to clear the entire inventory of the System you have selected.

Replace SYS_ID with the string representation of a System's GUID to delete the System.

```
public void deleteSystem()
{
    var response = client.DeleteSystem(Guid.Parse("SYS_ID"));
}
// returns Boolean: true = system deleted
```

DeleteComponent(Guid id)

WARNING: Deleting a Component will delete all Sections in the Component. Be sure that you wish to take this action. This is a significant step to take and should only be done when you are certain that you wish to clear the entire inventory of the Component you have selected.

Replace COMP_ID with the string representation of a Component's GUID to delete the Component.

```
public void deleteComponent()
{
    var response = client.DeleteComponent(Guid.Parse("COMP_ID"));
}
// returns Boolean: true = component deleted
```

DeleteSection(Guid id)

Replace SEC_ID with the string representation of a Section's GUID to delete the Section.

```
public void deleteSection()
{
    var response = client.DeleteSection(Guid.Parse("SEC_ID"));
}
// returns Boolean: true = section deleted
```

DeleteSectionDetail(Guid id)

Replace SEC_DETAIL_ID with a Section Detail's GUID to delete the Section Detail.

```
public void deleteSectionDetail()
{
    var response = client.DeleteSectionDetail(Guid.Parse("SEC_DETAIL_ID"));
}
// returns Boolean: true = section detail deleted
```

INVENTORY COST MODIFIERS

Cost modifiers allow bulk modification of expected inventory cost to reflect local or special situations such as remote location, high security site, or material shortages.

Subject to your BUILDER user permission level and scope, you can create cost modifiers and specify cost modifier assignments at most levels of inventory. This chapter covers manipulating inventory cost modifiers and inventory cost modifier assignments through the BUILDER API.

About Inventory Cost Modifiers

Cost modifiers allow bulk modification of expected inventory cost to reflect local or special situations such as remote location, high security site, or material shortages.

There are two types of cost modifers: cost adders, and cost multipliers.

- A *cost adder* will add a fixed cost to each inventory item at the specified inventory level, within the scope specified
- A *cost multiplier* will multiply the cost of each inventory item within the scope by a specified amount, such as by 1.2

To be used, a cost modifier needs to be created, and then assigned to the appropriate inventory.

About Inventory Cost Modifier Assignments

After a cost modifier has been created, it can be assigned to appropriate inventory.

Cost Modifier Service Calls

Using BUILDER API, you can perform the following functions related to inventory cost modifiers:

- Get available modifiers
- Create a modifier
- Update a modifier
- Delete a modifier
- Get modifier assignments
- Create a modifier assignment

- Update a modifier assignment
- Delete a modifier assignment

Other actions related to modifiers have to do with the "libraries" of available modifiers. These actions, which typically require more advanced BUILDER permissions, are:

- Get a modifier library
- Create a modifier library
- Update a modifier library
- Delete a modifier library

These other actions are described in "Cost Modifier Libraries " on page 101.

Get Available Cost Modifiers

Using the BUILDER API, you can get the inventory cost modifiers that are available for any given asset at a level from Site down through Section. This topic lists the service calls for getting these available cost modifiers in the scope you provide, the scope being defined by the GUID of an asset.

To get the cost modifier that have already been assigned to a given asset, see "Get Cost Modifier Assignments" on page 64.

Scope: Works at inventory levels Site through Section.

To get the cost modifiers available for an asset, you will first need to enter the value contained in the asset's `ID` property/GUID as the argument to one of the following methods:

- GetAvailableModifiersBySite
- GetAvailableModifiersByComplex
- GetAvailableModifiersByFacility
- GetAvailableModifiersBySystem
- GetAvailableModifiersByComponent
- GetAvailableModifiersBySection

These service calls will get the cost modifiers available to be associated with the GUID provided in the parameter. See the section "Get Available Cost Modifiers " above below for how to create code to display the available modifiers.

The availability of cost modifiers is governed by the cost modifier libraries. Users with sufficient permissions can get, add, update, or delete cost modifier libraries as described in "Cost Modifier Libraries " on page 101.

Code Examples: Get Available Cost Modifiers

A C# code example is provided below for each level of inventory accommodated.

General Model

The call examples provided below are in the format

```
var item = client.getAvailableModifiersBy<Inventory level>(Guid.Parse("<Inventory level>_ID"));
```

GetAvailableModifiersBySite(Guid id)

This code gets an array of all available inventory cost modifiers for a given Site and stores it in an instance of DTOCostModifier[ ] named "siteModifiers".

To use this code, replace SITE_ID with the string representation of a Site's GUID:

```
public void getAvailableModifiersBySite()
{
    var siteModifiers = client.getAvailableModifiersBySite(Guid.Parse("SITE_ID"));
}
// returns (service response) array of DTOCostModifier
```

GetAvailableModifiersByComplex(Guid id)

This code gets the array of available inventory cost modifiers for a given Complex and stores it in an instance of DTOCostModifier[ ] named "complexModifiers".

To use this code, replace CPX_ID with the string representation of a Complex's GUID:

```
public void getAvailableModifiersByComplex()
{
    var complexModifiers = client.GetAvailableModifiersByComplex(Guid.Parse("CPX_ID"));
}
// returns (service response) array of DTOCostModifier
```

GetAvailableModifiersByFacility(Guid id)

This code gets the array of available inventory cost modifiers for a given Facility and stores it in an instance of DTOCostModifier[ ] named "facilityModifiers".

To use this code, replace FAC_ID with a Facility's GUID:

```
public void getAvailableModifiersByFacility()
{
    var facilityModifiers = client.GetAvailableModifiersByFacility(Guid.Parse("FAC_ID"));
}
// returns (service response) array of DTOCostModifier
```

GetAvailableModifiersBySystem(Guid id)

This code gets the array of available inventory cost modifiers for a given System and stores it in an instance of DTOCostModifier[ ] named "systemModifiers".

To use this code, replace SYS_ID with the string representation of a System's GUID:

```
public void getAvailableModifiersBySystem()
{
    var systemModifiers = client.GetAvailableModifiersBySystem(Guid.Parse("SYS_ID"));
}
// returns (service response) array of DTOCostModifier
```

GetAvailableModifiersByComponent(Guid id)

This code gets the array of available inventory cost modifiers for a given Component and stores it in an instance of DTOCostModifier[ ] named "componentModifiers".

To use this code, replace COMP_ID with the string representation of a Component's GUID:

```
public void getAvailableModifiersByComponent()
{
  var componentModifiers = client.GetAvailableModifiersByComponent(Guid.Parse("COMP_ID"));
}
// returns (service response) array of DTOCostModifier
```

GetAvailableModifiersBySection(Guid id)

This code gets the array of available inventory cost modifiers for a given Section and stores it in an instance of DTOCostModifier[ ] named "sectionModifiers".

To use this code, replace SEC_ID with the string representation of a Section's GUID:

```
public void getAvailableModifiersBySection()
{
    var sectionModifiers = client.GetAvailableModifiersBySection(Guid.Parse("SEC_ID"));
}
// returns (service response) array of DTOCostModifier
```

Create Cost Modifier

Subject to your BUILDER user permission level and scope, you can update cost modifiers and their assignments at the Organization level and at the Facility level down through Section level using BUILDER API.

Scope: Works at inventory levels Organization, Facility, System, Component, and Section. The call for creating cost modifiers is available for use in the same scope as the set of calls that collectively get available modifiers (as described in "Get Available Cost Modifiers " on page 57): all inventory levels from Site through Section.

Instead of using a different API call for each applicable level of inventory, to create a cost modifier you will (1) pass in a name for the modifier, then (2) specify at what inventory level it applies and (3) specify whether it is an Adder or Multiplier.

With respect to the inventory level, a cost modifier that is at the System level can only be assigned to a System, and so forth. (see "Create Cost Modifier Assignment" on page 66.) The action of the modifier will be applied to every Section below the asset to which it is assigned.

The two types of cost modifier are *Adder* and *Multiplier*. A sample Adder might have a ModifierValue property of 1, which would add a dollar to the cost of each asset. A sample Multiplier might have a ModifierValue property of 1.2, which would increase the cost of each asset by twenty percent.

Code Example: Create Cost Modifier

A C# code example is provided below for how to create any level of cost modifier.

CreateModifier(DTOCostModifier costModifier)

When you create a cost modifier using the CreateModifier call, you will specify the desired inventory level by setting one of the new modifier's properties rather than by using different versions of the call for different inventory levels.

> **Note:** Additional cost modifier properties are shown at the description of the UpdateModifier call.

Verbose version

This copy of the process to create a cost modifier is fully commented, and includes sample property strings.

```
public void createModifier()
{
    //declare new instance
    var newModifier = new <service reference name>.DTOCostModifier();

    //set desired properties;
    //the first three shown here are the absolute minimum for BUILDER API validation
    newModifier.ModifierName = "add1dollar";
    newModifier.CostModLevelName = "Section";   // CostModLevelName should be
      // "Section" "Component" "System" "Building" "Complex" or "Site"
    newModifier.CostModTypeName = "Adder";     // Should be "Adder" or "Multiplier"
    newModifier.ModifierValue = 1;
    newModifier.ModifierDescription = "DESCRIPTION TEXT";
    newModifier.CustomFlag = false; //true or false
    newModifier.IsActive = true;    //true or false

    var response = client.CreateModifier(newModifier).Data;
}
// returns guid of the new modifier
```

Lean version

This version is the code without most of the comments, and without sample property names.

```
public void createModifier()
{
    var newModifier = new <service reference name>.DTOCostModifier();

    newModifier.ModifierName = "ENTER NAME HERE";
    newModifier.CostModLevelName = "ENTER VALUE HERE"; // CostModLevelName should be
        // "Section" "Component" "System" "Building" "Complex" or "Site"
    newModifier.CostModTypeName = "ENTER VALUE HERE";  // "Adder" or "Multiplier"
    newModifier.ModifierValue = 1;

    var response = client.CreateModifier(newModifier);
}
```

Update Cost Modifier

Subject to your BUILDER user permission level and scope, you can update cost modifiers and their assignments at the Organization level and at the Facility level down through Section level using BUILDER API. For updating a cost modifier assignment, see "Update Cost Modifier Assignment" on page 68.

Scope: Works at inventory levels Organization, Facility, System, Component, and Section.

Use this call to change the values of properties in an instance of DTOCostModifier.

Code Example: Update Cost Modifier

A C# code example is provided below for how to update any level of cost modifier.

UpdateModifier(DTOCostModifier costModifier)

Use this call to make additions or modifications to an existing cost modifier.

If you need to change the inventory asset level that the cost modifier is associated with (assuming that a cost modifier of that name is available at the new level in the cost modifier library), here are the integers that can be used as values for the `CostModLevelId` property:

1 = Section

2 = Component

3 = System

4 = Building

5 = Complex

6 = Site

Note: When you create a cost modifier, you enter the inventory level that the modifier applies to using a string name such as "Section" or "Complex" for the `CostModLevelName` property. By contrast, when you update a cost modifier, you enter an integer value for the `CostModLevelId` property. (`CostModTypeName` remains the same for both calls.)

There are multiple ways to identify the GUID of the modifier to be updated. The code below uses the approach of getting a modifier list and selecting one of the modifiers in that list.

To use the code, replace MOD_LIST_ID with the GUID of the modifier list containing the modifier you want to update.

```
public void updateModifier()
{
    //currently (2022) API has no GetModifier call, so get the modifier list
    var theModifiers = client.GetModifierList(Guid.Parse("MOD_LIST_ID"));

    //create new instance
    var updateMod = new <service reference name>.DTOCostModifier();

    //before using next line, decide which modifier in list to populate the new instance with
    updateMod = theModifiers.Data[0];   // using the first modifier here as example
```

```
    //add properties or change property values as needed; for example:
    updateMod.ModifierValue = "UPDATE VALUE HERE";
    updateMod.CostModLevelId = "INT VALUE"; //int value 1-6
    updateMod.ModifierDescription = "NEW DESCRIPTION";

    var response = client.UpdateModifier(updateMod);

    //error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
// returns Boolean: true = update successful
```

Delete Cost Modifier

Subject to your BUILDER user permission level and scope, you can delete cost modifiers and their assignments at the Organization level and at the Facility level down through Section level using BUILDER API.

Code Example: Delete Cost Modifier

A C# code example is provided below for how to delete any level of cost modifier.

DeleteModifier(Guid id)

To use the code below, replace COST_MOD_ID with the string representation of a cost modifier's GUID to delete a modifier.

After the call succeeds, the deleted cost modifier will no longer be available for assignment within the scope specified in the parameter.

> **Note:** Depending on your needs, a potential alternative to deleting is to use the "UpdateModifier(DTOCostModifier costModifier)" on the previous page call to set the modifier's `IsActive` property to false.

```
public void deleteModifier()
{
    var response = client.DeleteModifier(Guid.Parse("COST_MOD_ID"));
}
// returns Boolean: true = modifier deleted
```

Get Cost Modifier Assignments

Using the BUILDER API, you can get the cost modifiers that have been applied to any asset from Site down through Section. This topic lists the service calls for getting the assignments applicable to the scope you provide, the scope being the GUID of an asset.

To instead get the cost modifiers that are available at a given inventory level, see "Get Available Cost Modifiers " on page 57.

Scope: Works at the Organization level, and at inventory levels Facility through Section.

To get the cost modifiers that have been assigned to a given asset (inventory item), you will first need to enter the value in the asset's `ID` property as the argument to one of the following methods:

- GetModifierAssignmentsByOrg
- GetModifierAssignmentsByFacility
- GetModifierAssignmentsBySystem
- GetModifierAssignmentsByComponent
- GetModifierAssignmentsBySection

These service calls will get the cost modifier assignments associated with the GUID provided in the parameter. See the section Display or Store Information below for how to create code to display the assignments.

Code Examples: Get Cost Modifier Assignments

A C# code example is provided below for each level of inventory accommodated.

General Model

The code examples provided below are in the format

```
var item = client.getModifierAssignmentsBy<Inventory level>(Guid.Parse("<Inventory level>_ID"));
```

GetModifierAssignmentsByOrg(Guid id)

This code gets the cost modifier assignments for a given Organization and stores them in an array named "orgModAssignments".

To use this code, replace ORG_ID with the string representation of a real Organization's GUID.

```
public void getModifierAssignmentsByOrg()
{
    var orgModAssignments = client.GetModifierAssignmentsByOrg(Guid.Parse("ORG_ID"));
}
// returns (service response) array of DTOCostModifierAssignment
```

GetModifierAssignmentsByFacility(Guid id)

This code gets the cost modifier assignments for a given Facility and stores them in an array named "facModAssignments".

To use this code, replace FAC_ID with the string representation of a Facility's GUID.

```
public void getModifierAssignmentsByFacility()
{
   var facModAssignments = client.GetModifierAssignmentsByFacility(Guid.Parse("FAC_ID"));
}
// returns (service response) array of DTOCostModifierAssignment
```

GetModifierAssignmentsBySystem(Guid id)

This code gets the inventory cost modifiers for a given System and stores them in an array named "sysModAssignments".

To use this code, replace SYS_ID with the string representation of a System's GUID.

```
public void getModifierAssignmentsBySystem()
{
    var sysModAssignments = client.GetModifierAssignmentsBySystem(Guid.Parse("SYS_ID"));
}
// returns (service response) array of DTOCostModifierAssignment
```

GetModifierAssignmentsByComponent(Guid id)

This code gets the cost modifier assignments for a given Component and stores them in an array named "compModAssignments".

To use this code, replace COMP_ID with the string representation of a Component's GUID.

```
public void getModifierAssignmentsByComponent()
{
  var compModAssignments=client.GetModifierAssignmentsByComponent(Guid.Parse("COMP_ID"));
}
// returns (service response) array of DTOCostModifierAssignment
```

GetModifierAssignmentsBySection(Guid id)

This code gets the cost modifier assignments for a given Section and stores them in an array named "secModAssignments".

To use this code, replace SEC_ID with the string representation of a Section's GUID.

```
public void getModifierAssignmentsBySection()
{
    var secModAssignments = client.GetModifierAssignmentsBySection(Guid.Parse("SEC_ID"));
}
// returns (service response) array of DTOCostModifierAssignment
```

Create Cost Modifier Assignment

Scope: Works at inventory levels Organization, Facility, System, Component, and Section.

To assign a cost modifier to an asset, you will specify the asset's ID and the cost modifier's ID.

The action of the modifier will be applied to every Section below the asset to which it is assigned.

You can only assign a System level cost modifier to a System, and so forth. Therefore, you need to be aware of the cost modifier's level, whether it is "Section" "Component" "System" "Building" "Complex" or "Site", and use the call for the appropriate level:

- CreateModifierAssignmentByOrg
- CreateModifierAssignmentByFacility
- CreateModifierAssignmentBySystem
- CreateModifierAssignmentByComponent
- CreateModifierAssignmentBySection

Code Examples: Create Cost Modifier Assignments

A C# code example is provided below for each level of inventory accommodated.

General Model

The pattern for creating a cost modifier assignment in the API is to

1. Create a [Guid] variable to hold the ID of the Organization.
2. Create a [Guid] variable to hold the ID of the modifier.
3. (Optional) If a comment is desired, create a variable to hold the comment string.
4. Make the API call.

CreateModifierAssignmentByOrg(Guid orgId, Guid modId, string comment)

Pair an Organization's ID with a cost modifier ID using this call.

```
public void CreateOrgModAssignment()
{
    // create variables to hold ORG_ID; MOD_ID; and, optionally, a comment
    var orgId = Guid.Parse("ORG_ID");
    var modId = Guid.Parse("MOD_ID");
    var comment = "OPTIONAL COMMENT TEXT";

    var response = client.CreateModifierAssignmentByOrg(orgId, modId, comment);
}
// returns guid of the assignment
```

CreateModifierAssignmentByFacility(Guid facId, Guid modId, string comment)

Pair a Facility's ID with a cost modifier ID using this call.

```
public void CreateFacModAssignment()
{
    // create variables to hold FAC_ID, MOD_ID, and, optionally, a comment
    var facId = Guid.Parse("FAC_ID");
    var modId = Guid.Parse("MOD_ID");
    var comment = "OPTIONAL COMMENT TEXT";

    var response = client.CreateModifierAssignmentByFacility(facId, modId, comment);
}
// returns guid of the assignment
```

CreateModifierAssignmentBySystem(Guid sysId, Guid modId, string comment)

Pair a System's ID with a cost modifier ID using this call.

```
public void createSystemModAssignment()
{
    // create variables to hold SYS_ID, MOD_ID, and, optionally, a comment
    var sysId = Guid.Parse("SYS_ID");
    var modId = Guid.Parse("MOD_ID");
    var comment = "OPTIONAL COMMENT TEXT";

    var response = client.CreateModifierAssignmentBySystem(sysId, modId, comment);
}
// returns guid of the assignment
```

CreateModifierAssignmentByComponent(Guid compId, Guid modId, string comment)

Pair a Component's ID with a cost modifier ID using this call.

```
public void CreateComponentModAssignment()
{
    // create variables to hold COMP_ID, MOD_ID, and, optionally, a comment
    var compId = Guid.Parse("COMP_ID");
    var modId = Guid.Parse("MOD_ID");
    var comment = "OPTIONAL COMMENT TEXT";

    var response = client.CreateModifierAssignmentByComponent(compId, modId, comment);
}
// returns guid of the assignment
```

CreateModifierAssignmentBySection(Guid secId, Guid modId, string comment)

Pair a Section's ID with a cost modifier ID using this call.

```
public void CreateSectionModAssignment()
{
    // create variables to hold SEC_ID, MOD_ID, and, optionally, a comment
    var secId = Guid.Parse("SEC_ID");
    var modId = Guid.Parse("MOD_ID");
    var comment = "OPTIONAL COMMENT TEXT";

    var response = client.CreateModifierAssignmentBySection(secId, modId, comment);
}
// returns guid of the assignment
```

Update Cost Modifier Assignment

Scope: Works at inventory levels Organization, Facility, System, Component, and Section.

These calls allow you to change the comment associated with a cost modifier assignment. If you want to change either the asset ID or the cost modifier ID in the relationship established by the assignment, you will need to delete the assignment and create a new one.

To update the comment for a cost modifier assignment you will need to provide (1) the assignment's `ID` property and (2) the updated comment, using one of the following methods:

- UpdateModifierAssignmentByOrg
- UpdateModifierAssignmentByFacility
- UpdateModifierAssignmentBySystem
- UpdateModifierAssignmentByComponent
- UpdateModifierAssignmentBySection

Code Examples: Update Cost Modifier Assignment

A C# code example is provided below for each level of inventory accommodated.

UpdateModifierAssignmentByOrg(Guid assignmentId, string comment)

This code allows you to change the comment associated with a cost modifier assignment.

To use, replace ASSIGN_ID with the GUID of the assignment you want to update:

```
public void updateModifierAssignmentByOrg()
{
    //provide ID and updated comment to be used as parameters
    var orgAssignId = Guid.Parse("ASSIGN_ID");
    var newComment = "NEW COMMENT";

    //call the update function
    var response = client.UpdateModifierAssignmentByOrg(orgAssignID, newComment);
}
// returns Boolean: true = successful update
```

UpdateModifierAssignmentByFacility(Guid assignmentId, string comment)

This code allows you to change the comment associated with a cost modifier assignment.

To use, replace ASSIGN_ID with the GUID of the assignment you want to update:

```
public void updateModifierAssignmentByFacility()
{
    //provide ID and updated comment to be used as parameters
    var facAssignId = Guid.Parse("ASSIGN_ID");
    var newComment = "NEW COMMENT";

    //call the update function
    var response = client.UpdateModifierAssignmentByFacility(facAssignId, newComment);
}
// returns Boolean: true = successful update
```

UpdateModifierAssignmentBySystem(Guid assignmentId, string comment)

This code allows you to change the comment associated with a cost modifier assignment.

To use, replace ASSIGN_ID with the GUID of the assignment you want to update:

```
public void updateModifierAssignmentBySystem()
{
    //provide ID and updated comment to be used as parameters
    var sysAssignId = Guid.Parse("ASSIGN_ID");
    var newComment = "NEW COMMENT";
```

```
    //call the update function
    var response = client.UpdateModifierAssignmentBySystem(sysAssignId, newComment);
}
// returns Boolean: true = successful update
```

UpdateModifierAssignmentByComponent(Guid assignmentId, string comment)

This code allows you to change the comment associated with a cost modifier assignment.

To use, replace ASSIGN_ID with the GUID of the assignment you want to update:

```
public void updateModifierAssignmentByComponent()
{
    //provide ID and updated comment to be used as parameters
    var compAssignId = Guid.Parse("ASSIGN_ID");
    var newComment = "NEW COMMENT";

    //call the update function
    var response = client.UpdateModifierAssignmentByComponent(compAssignId, newComment);
}
// returns Boolean: true = successful update
```

UpdateModifierAssignmentBySection(Guid assignmentId, string comment)

The code below allows you to change the comment associated with a cost modifier assignment.

To use, replace ASSIGN_ID with the GUID of the assignment you want to update.

```
public void updateModifierAssignmentBySection()
{
    //provide ID and updated comment to be used as parameters
    var secAssignId = Guid.Parse("ASSIGN_ID");
    var newComment = "NEW COMMENT";

    var response = client.UpdateModifierAssignmentBySection(secAssignId, newComment);
}
// returns Boolean: true = successful update
```

Delete Cost Modifier Assignment

Subject to your BUILDER user permission level and scope, you can delete cost modifiers and their assignments at the Organization level and at the Facility level down through Section level using BUILDER API.

Scope: Works at inventory levels Organization, Facility, System, Component, and Section.

These calls will delete all modifier assignments created for the asset specified in the call by its GUID. It will not affect the modifiers themselves.

Code Examples: Delete Cost Modifier Assignment

A C# code example is provided below for each level of inventory accommodated.

DeleteModifierAssignmentByOrg(Guid id)

Replace ORG_ID with the string representation of a real Organization's GUID to delete all modifier assignments associated with that Organization.

```
public void deleteModifierAssignmentByOrg()
{
    var response = client.DeleteModifierAssignmentByOrg(Guid.Parse("ORG_ID"));
}
// returns Boolean: true = modifier assignment deleted
```

DeleteModifierAssignmentByFacility(Guid id)

Replace FAC_ID with the string representation of a Facility's GUID to delete all modifier assignments associated with that Facility.

```
public void deleteModifierAssignmentByFacility()
{
    var response = client.DeleteModifierAssignmentByFacility(Guid.Parse("FAC_ID"));
}
// returns Boolean: true = modifier assignment deleted
```

DeleteModifierAssignmentBySystem(Guid id)

Replace SYS_ID with the string representation of a System's GUID to delete all modifier assignments associated with that System.

```
public void deleteModifierAssignmentBySystem()
{
    var response = client.DeleteModifierAssignmentBySystem(Guid.Parse("SYS_ID"));
}
// returns Boolean: true = modifier assignment deleted
```

DeleteModifierAssignmentByComponent(Guid id)

Replace COMP_ID with the string representation of a Component's GUID to delete all modifier assignments associated with that Component.

```
public void deleteModifierAssignmentByComponent()
{
    var response = client.DeleteModifierAssignmentByComponent(Guid.Parse("COMP_ID"));
}
// returns Boolean: true = modifier assignment deleted
```

DeleteModifierAssignmentBySection(Guid id)

Replace SEC_ID with the string representation of a Section's GUID to delete all modifier assignments associated with that Section.

```
public void deleteModifierAssignmentBySection()
{
    var response = client.DeleteModifierAssignmentBySection(Guid.Parse("SEC_ID"));
}
// returns Boolean: true = modifier assignment deleted
```

Get Modifier Lists

The calls described below will get an array of the modifiers that are included in a modifier library ("GetModifierList(Guid id)" below) or an array of the modifiers in the scope of a given Organization ("GetModifierListByOrg(Guid id)" below).

Example Code: Cost Modifier Lists

Code examples are in C#.

GetModifierList(Guid id)

The calls GetModifierLibrary and GetModifierList both use the GUID of a modifier library as the parameter. The difference between the two calls is that GetModifierList returns a list of all the modifiers in the library, whereas GetModifierLibrary returns information about the library itself.

To use the code below, replace MOD_LIB_ID with the string representation of a cost modifier library's GUID.

```
public void getModifierList()
{
    var response = client.GetModifierList(Guid.Parse("MOD_LIB_ID"));
}
// returns array of DTOCostModifier
```

GetModifierListByOrg(Guid id)

The code below returns all the cost modifiers associated with the specified Organization.

To use the code, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getModifierListByOrg()
{
   var response = client.GetModifierListByOrg(Guid.Parse("ORG_ID"));
}
// returns array of DTOCostModifier
```

INSPECTIONS

Inspection Service Calls

Using BUILDER API, you can do the following for both direct rating inspections and distress inventory inspections:

- Get Inspection
- Create Inspection
- Update Inspection
- Delete Inspection

Inspection Sample Use Case

The following sample use case shows creating a direct rating inspection:

- "Use Case 5: Create a Direct Rating Inspection" on page 151

Color vs. Numeric Condition Rating

In the BUILDER Web interface, color selections (Green+, Green, Green-, etc.) are used for a direct rating on a Section. In BUILDER API a numeric condition rating is used instead.

The table below shows the condition rating equivalent to what BUILDER assigns to each color selection, as well as the range of condition ratings covered by each color selection.

Condition Rating Equivalents for Direct Inspection Color Selections

Color Selection	Cond. Rating Equivalent	Range
Green +	100	100 - 100
Green	95	93 - 99
Green -	88	86 - 92
Amber +	80	75 - 85
Amber	71	65 - 74
Amber -	61	56 - 64
Red +	50	37 - 55
Red	30	11 - 36
Red -	10	0 - 10

Get Inspection

The GetInspection call works for both a direct inspection or a distress survey. To get a direct inspection, focus on those inspections having a value of 2 for the `Type` property. To get a distress survey, you will want to focus on those inspections having a value of 1 for the `Type` property.

There are additional calls below that are exclusively to get data specific to distress surveys.

Code Examples: Get Inspections

There are two alternative ways to get an inspection:

a. Use the GUID of a Section to get all inspections performed on that Section, then select one.
b. Use the GUID of the inspection itself to get that inspection.

Each of these is illustrated below. Code examples are in C#.

GetSectionInspections(Guid id)

For this call, the GUID input parameter you need to provide is that of a Section.

The sample code below takes the `ID` property (GUID) of the inspected Section as a parameter, and returns an array of the inspections performed on that Section (stored in the variable secInsps). The array will not be in any particular order.

In the code example, the `ID` property and `InspectionType` property (1 = distress survey, 2 = direct inspection) of each inspection is output to the console, along with the inspection date. If the desired inspection can be identified by its date, then the corresponding `ID` can be used as the parameter to get just that particular inspection, using the call GetInspections.

To use the code, replace SEC_ID with the string representation of a Section's GUID.

> **Note:** The comment instructions for using the code are set up to display only direct inspections or only distress surveys, but the array returned by the call contains inspections of all types performed on the designated Section.

```
public void getSectionInspections()
{
    var secInsps = client.GetSectionInspections(Guid.Parse("SEC_ID")).Data;

    foreach (var item in secInsps)
    {
       // for distress surveys only:
       // use the next two lines and comment out direct ratings code
       if (item.Type == 1)
          Console.WriteLine(item.ID.ToString() + ": " + item.Date.ToString());

       // for direct ratings only:
       // use the next two lines and comment out distress survey code
       if (item.Type == 2)
          Console.WriteLine(item.ID.ToString() + ": " + item.Date.ToString());
    }

    Console.Read();
}
//returns array of DTOInspection
```

GetInspection(Guid id)

This call returns an instance of an inspection object. The input parameter that you need to provide is the GUID of an inspection. This can be a direct inspection or a distress survey.

> **Tip:** One way to come up with the GUID of an inspection is to run the call "GetSectionInspections(Guid id)" on the previous page.

In the code example below, the first line declares an instance of DTOInspection, and the following (optional) line outputs the inspection's GUID, inspection type, and the date of inspection.

To use the code, replace INSP_GUID with the string representation of an inspection's GUID.

```
public void getInspection()
{
    var insp = client.GetInspection(Guid.Parse("INSP_ID")).Data;

    Console.WriteLine(insp.SectionID.ToString() + ":" + insp.Date.ToString());
}
// returns DTOInspection
```

Code Examples: Get Data Specific to Distress Surveys

Code examples are in C#.

GetDistressSeverityValues(int[ ] IDs)

This call facilitates retrieving an array showing the ID numbers for severity values associated with distress surveys. The input parameter that you need to provide is the size of the array you want.

The code below shows how to construct the array of integers to be passed to the call as its parameter.

```
public void getDistressSeverityValues()
{
    // create an integer array for severity values
    var insp = severityIDs = new int[2];

    // then populate the array
    severityIDs[0] = 1;
    severityIDs[2] = 2;

    // after setting up the array, make the call
    var response = client.GetDistressSeverityValues(severityIDs);
}
// returns array of DTOSeverityValue
```

GetDistressDensityValues(int[ ] IDs)

This call facilitates retrieving an array showing the ID numbers for density values associated with distress surveys. The input parameter that you need to provide is the size of the array you want.

The code below shows how to construct the array of integers to be passed to the call as its parameter.

```
public void getDistressDensityValues()
{
    // create an integer array for density values
    var densityIDs = new int[2];

    // then populate the array
    densityIDs[0] = 1;
    densityIDs[2] = 2;

    // after setting up the array, make the call
    var response = client.GetDistressDensityValues(densityIDs);
}
// returns array of DTODensityValue
```

Lock and Unlock Inventory

Using the BUILDER API, you can lock and unlock individual Systems.

An example of when you might do this is when one or more Systems are checked out and back in to BuilderRED or another remote device designed to assist with taking BUILDER inventory or performing inspections.

Code Examples: Lock and Unlock Systems

Code examples are in C#.

LockSystem(Guid id)

This call locks a System so that it can not be edited in BUILDER until after the System is unlocked again.

To use the code shown here, replace SYS_ID with the string representation of a System's GUID.

```
public void lockSystem()
{
    var response = client.LockSystem(Guid.Parse("SYS_ID"));
}
// returns Boolean: true = unlocked system successfully locked
```

A return of false from the LockSystem call can mean that an unlocked system has not been successfully locked, or it can mean that the system was already locked (and still is).

UnlockSystem(Guid id)

This call unlocks a System, such as when it has been locked against editing using the LockSystem call.

To use the code shown here, replace SYS_ID with the string representation of a System's GUID.

```
public void unlockSystem()
{
    var response = client.UnlockSystem(Guid.Parse("SYS_ID"));
}
// returns Boolean: true = system successfully unlocked
```

A return of false from the UnlockSystem call can mean that a locked system has not been successfully unlocked, or it can mean that the system was already unlocked (and still is).

Create Inspection

For creating an inspection, you need to know that the value of the `Type` property of an inspection is 1 for a distress survey, 2 for a direct rating.

It helps to be familiar with the DTOInspection class, and also with the DTOSample class if the inspections you are handling use sampling.

Enums for Creating Inspection

`InspectionType` Enum

The value to use for the required property `Type` is governed by the InspectionType enum, which maps values to inspection type choices as follows:

1 = DistressSurvey

2 = DirectRating

3 = DistressWithQuantity

4 = RooferInspection

`InspectionSource` Enum

The value to use for the required property `InspectionSource` is governed by the InspectionSource enum, which maps values to inspection source choices as follows:

0 = InstallDate (initial inspection generated by BUILDER with CI = 100)

1 = Inspection

2 = WorkComp

3 = RapidInspection

4 = Roofer

5 = RooferRepairProject

Code Example: Create Inspection

Code examples are in C#.

CreateInspection(DTOinspection inspection)

To create a direct rating inspection,

1. Declare a new instance of an inspection and assign values to the non-array properties.
2. Enter sample information in one of two alternate ways:
   a. For a direct rating inspection where `inspectionName.IsSampling` is False, the entire Section is the sample. Assign to `inspectionName.Sample[0].ConditionRating` the value corresponding to the desired direct rating, as shown in the table in "Numerical Values for Direct Rating Selections."
   b. For a direct rating inspection where `inspectionName.IsSampling` is True, you will need to create individual samples, indicating the sample location for each (the sample location property isn't needed when `isSampling` is False).
3. Call `CreateInspection(<name of instance created in Step 1>)`. and store the returned GUID in a variable with data format type of guid.

Below is sample code for creating a direct rating inspection without sampling.

```
public void createInspectionDirect()
{
    //creates non sampling direct rating inspection
    var insp = new <service reference name>.DTOInspection();
    insp.Type = 2;              // Type 2 is direct rating
    insp.Date = DateTime.Now;
    insp.IsSampling = false;
    insp.Source = "Inspection";

    // enter information of samples.
    // If isSampling false, there is just one sample that comprises the whole section

    var samp = new <service reference name>.DTOSample();
    samp.IsPaint = false;
    //when isSampling = false, sample qty should equal section qty
    samp.Quantity = 3;
    samp.ConditionRating = 70;

    //make array for sample properties
    <service reference name>.DTOSample[] smpArray = { samp };

    insp.Samples = smpArray;
    Guid theGuid = client.CreateInspection(insp).Data;
    Console.WriteLine(theGuid.ToString());
    Console.Read();
}
// returns guid of the new inspection
```

Update Inspection

Code Example: Update Inspection

Code examples are in C#.

UpdateInspection(DTOinspection inspection)

This code example populates the new instance of the inspection directly with a Get call without using a separate variable to store the results of the Get call.

To use the code, replace INSP_ID with the string representation of the GUID (`ID` property) of the inspection to be updated.

```
public void updateInspection()
{
    //declare new instance and populate
    var updateInspection = new <service reference name>.DTOInspection();
    updateInspection = client.GetInspection(Guid.Parse("INSP_ID")).Data;

    //add properties or change property values as needed; for example:
    insp.Samples[0].IsPaint = true;
    insp.Samples[0].PaintRating = 100;

    var response = client.UpdateInspection(updateInspection);

    //error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

Delete Inspection

Code Example: Delete Inspection

The call for deleting a distress survey and the call for deleting a direct inspection are the same.

Code examples are in C#.

DeleteInspection(Guid id)

To use this code, replace INSP_ID with the string representation of the GUID (`ID` property) of the inspection or distress survey to be deleted.

```
public void deleteInspection()
{
    client.DeleteInspection(Guid.Parse("INSP_ID"));
}
// returns Boolean: true = inspection deleted
```

KNOWLEDGE-BASED INSPECTIONS

Knowledge-based inspections (KBIs) are those that have been performed in response to recommendations from a KBI Schedule generated in BUILDER.

The current capability in the API with respect to KBIs is that you can get KBIs that have been performed.

Get Knowledge-Based Inspection

Scope: Site, Complex, Facility, Section.

Using the BUILDER API, you can get knowledge-based inspections (KBIs) at different inventory levels. These will always be presented with paged results.

Code Examples: Get KBIs, with Paged Results

Code examples are in C#.

GetSiteKBIs(Guid id, int skip, int take)

To use the code shown, replace SITE_ID with the string representation of a Site's GUID.

The sample below returns the first 100 results (or all of them, if the array has fewer than 100 elements) in an array of DTOKnowledgeBasedInspection.

```
public void getSiteKBIs()
{
    var response = client.GetSiteKBIs(Guid.Parse("SITE_ID"), 0, 100);

    // Optional: Output the number of KBIs in the array to the console
    Console.WriteLine("The number of KBIs found by this call is "
    + response.Data.Length.ToString() );
}
// returns DTOKnowledgeBasedInspection
```

GetComplexKBIs(Guid id, int skip, int take)

The sample below returns the first 100 results (or all of them, if the array has fewer than 100 elements) in an array of DTOKnowledgeBasedInspection.

To use the code shown, replace CPX_ID with the string representation of a Complex's GUID.

```
public void getComplexKBIs()
{
    var response = client.GetComplexKBIs(Guid.Parse("CPX_ID"), 0, 100);

    // Optional: Output the number of KBIs in the array to the console
    Console.WriteLine("The number of KBIs found by this call is "
    + response.Data.Length.ToString() );
}
// returns array of DTOKnowledgeBasedInspection
```

GetFacilityKBIs(Guid id, int skip, int take)

The sample below returns the first 100 results (or all of them, if the array has fewer than 100 elements) in an array of DTOKnowledgeBasedInspection.

To use the code shown, replace FAC_ID with the string representation of a Facility's GUID.

```
public void getFacilityKBIs()
{
    var response = client.GetFacilityKBIs(Guid.Parse("FAC_ID"), 0, 100);

    // Optional: Output the number of KBIs in the array to the console
    Console.WriteLine("The number of KBIs found by this call is "
    + response.Data.Length.ToString() );
}
// returns array of DTOKnowledgeBasedInspection
```

GetSectionKBIs(Guid id, int skip, int take)

The sample below returns the first 100 results (or all of them, if the array has fewer than 100 elements) in an array of DTOKnowledgeBasedInspection.

To use the code shown, replace SEC_ID with the string representation of a Section's GUID.

```
public void getSectionKBIs()
{
    var response = client.GetSectionKBIs(Guid.Parse("SEC_ID"), 0, 100);

    // Optional: Output the number of KBIs in the array to the console
    Console.WriteLine("The number of KBIs found by this call is "
    + response.Data.Length.ToString() );
}
// returns array of DTOKnowledgeBasedInspection
```

ATTACHMENTS

Using the BUILDER API, you can add an attachment to an inventory asset, or to an inspection. The asset or inspection is called the "owner" of the attachment.

You can fetch attachments, add attachments, and delete attachments. To change an attachment, you will need to delete the old one and create a new one.

BUILDER API has these calls related to attachments:

- AddAttachment
- AddAttachmentWithFileExtension
- FetchAttachmentsByOwner
- FetchAttachmentsWithDetailsByOwner
- FetchAttachmentDetailsWithHashesByOwner
- FetchAttachment
- FetchAttachmentWithDetails
- DeleteAttachment

Fetch Attachment

BUILDER API provides a rich variety of ways to define the attachments you want to fetch.

Code Examples: Fetch Attachment

Code examples are in C#.

FetchAttachment(Guid attachmentId)

This call returns an attachment identified by its GUID (`ImageID` property). It does not return any of the meta information such as description and other readable properties. To receive those details along with the attachment itself, you need to use the call "FetchAttachmentWithDetails(Guid attachmentId)" on the next page.

If you don't know the GUID of the attachment, you may need to use another call, such as "FetchAttachmentsByOwner(Guid ownerId, string level)" on the next page, in order to get the Guid to use.

To use the code, replace ATTACHMENT_ID with the string representation of a valid attachment's `ImageID` property. If you don't already have the ID of the attachment, the call FetchAttachmentsWithDetailsByOwner can help you find it.

```
public void fetchAttachment()
{
    var response = client.FetchAttachment(Guid.Parse("ATTACHMENT_ID"));
}
// returns byte data of the attachment
```

FetchAttachmentWithDetails(Guid attachmentId)

This call returns an attachment identified by its GUID (`ImageID` property). This returns the image along with meta information such as description and other readable properties.

If you don't know the GUID of the attachment, you may need to use another call, such as FetchAttachmentsByOwner(Guid ownerId, string level), in order to get the Guid to use.

To use the code, replace ATTACHMENT_ID with the string representation of a valid attachment's `ImageID` property. If you don't already have the ID of the attachment, the call FetchAttachmentsWithDetailsByOwner can help you find it.

```
public void fetchAttachmentWithDetails()
{
    var response = client.FetchAttachmentWithDetails(Guid.Parse("ATTACHMENT_ID"));
}
// returns DTOAttachment
```

FetchAttachmentsByOwner(Guid ownerId, string level)

This call returns a collection of attachments based on the parameters supplied. It does not return any of the meta information such as description and other readable properties. To receive those details along with the attachments themselves, you need to use the call FetchAttachmentsWithDetailsByOwner.

To use the code, replace OWNER_ID with the string representation of a valid asset or inspection's `ID` property, and replace "LEVEL" with one of the following:

- "Section_Detail"
- "Section"
- "Component_Section"
- "Component"
- "System_Component"
- "System"

- "Building_System"
- "Building"
- "Complex"
- "Site"
- "Inspection"
- "Inspection_Data"

```
public void fetchAttachmentsByOwner()
{
    var response = client.FetchAttachmentsByOwner(Guid.Parse("OWNER_ID"),
      "LEVEL");
}
// returns byte data of the attachment
```

FetchAttachmentsWithDetailsByOwner(Guid ownerId, string level)

This call returns an array of attachments based on the parameters supplied. This returns the image along with meta information such as description and other readable properties.

To use the code, replace OWNER_ID with the string representation of a valid asset or inspection's `ID` property, and replace "OWNER_LEVEL" with one of the following:

- "Section_Detail"
- "Section"
- "Component_Section"
- "Component"
- "System_Component"
- "System"
- "Building_System"
- "Building"
- "Complex"
- "Site"
- "Inspection"
- "Inspection_Data"

```
public void fetchAttchsWithDetailsByOwner()
{
    var response = client.FetchAttachmentsWithDetailsByOwner(Guid.Parse("OWNER_ID"), "LEVEL");
}
// returns array of DTOAttachment
```

FetchAttachmentDetailsWithHashesByOwner(Guid ownerId, string level)

This call returns an array of attachments based on the parameters supplied, including the same metadata as described in FetchAttachmentsWithDetailsByOwner, and then hashes it.

To use the code, replace OWNER_ID with the string representation of a valid asset or inspection's GUID (`ID` property), and replace "OWNER_LEVEL" with one of the following:

- "Section_Detail"
- "Section"
- "Component_Section"
- "Component"
- "System_Component"
- "System"
- "Building_System"
- "Building"
- "Complex"
- "Site"
- "Inspection"
- "Inspection_Data"

```
public void fetchAttchDetailsWithHashesByOwner()
{
   var response = client.FetchAttachmentDetailsWithHashesByOwner(Guid.Parse("OWNER_ID"),
      "LEVEL");
}
// returns array of DTOAttachment
```

Add Attachment

You can associate one or more attachments with an inspection or with any level of asset except Organization. If you connect multiple attachments to the same asset or inspection, each attachment will need a separate call.

Code Examples: Add Attachment

Code examples are in C#.

AddAttachment(Guid ownerId, string level, string imgTitle, string imgDesc, byte[ ] attachment)

> **IMPORTANT:** The only kind of file you can attach with this call is a file with .bmp file extension.

The five parameters for this call are:

1. **Guid ownerId**. This is the GUID of the attachment's "owner", which is the inventory asset or inspection that the attachment is to be associated with.
2. **string level**. This specifies the asset level or other type of owner (such as inspection). The options that can be used for "level" are:
    - "Section_Detail"
    - "Section"
    - "Component_Section"
    - "Component"
    - "System_Component"
    - "System"
    - "Building_System"
    - "Building"
    - "Complex"
    - "Site"
    - "Inspection"
    - "Inspection_Data"
3. **string imgTitle**. Title for the attachment.
4. **string imgDesc**. Description of what the attachment represents.
5. **byte[ ] attachment**. The array containing the information for the image attachment. This is built and populated in the code, rather than a value being directly provided in a parameter.

The first parameter, represented in the code sample by OWNER_ID, is the GUID of the entity that the attachment belongs with. This entity can be an inspection, or it can be a Site, Complex, Building/Facility, System, Component, or Section.

This code will transfer one byte at a time of a .bmp image into an attachment. Before running it, make the following replacements:

- Replace FULL FILE DIRECTORY PATH with the path of the image to be attached, including the file name of the image and its extension.
- Replace OWNER_ID with the string representation of the appropriate GUID.

- Replace "OWNER_LEVEL" with one of the following: "Section_Detail" "Section" "Component_Section" "Component" "System_Component" "System" "Building_System" "Building" "Complex" "Site" "Inspection" "Inspection_Data" "Section_Detail"
- Replace IMG_TITLE and IMG_DESCR with strings you enter for the title and description of the image, respectively.

```
public void addAttachment();
{
byte[] imageByteArray = null;
string imagePath = @"FULL FILE DIRECTORY PATH";
FileStream fileStream = new FileStream(imagePath, FileMode.Open, FileAccess.Read);

using (BinaryReader reader = new BinaryReader(fileStream))
   {
       imageByteArray = new byte[reader.BaseStream.Length];
       for (int i = 0; i < reader.BaseStream.Length; i++)
       {
               imageByteArray[i] = reader.ReadByte();
       }
   }
var response = client.AddAttachment(Guid.Parse("OWNER_ID"), "OWNER_LEVEL",
     "IMG_TITLE", "IMG_DESC", imageByteArray);
}
//returns guid of the image attachment
```

AddAttachmentWithFileExtension(Guid ownerId, string level, string imgTitle, string imgDesc, byte[ ] attachment, string fileExtension)

You can use this call to add an attachment having any of the following extensions,

.accdb, .bmp, .doc, .docx, .jpeg, .jpg, .pdf, .png, .xksx, .zip

The six parameters for this call are:

1. **Guid ownerId**. This is the GUID of the attachment's "owner", which is the inventory asset or inspection that the attachment is to be associated with.
2. **string level**. This specifies the asset level or other type of owner (such as inspection). The options that can be used for "level" are:
    - "Section_Detail"
    - "Section"
    - "Component_Section"
    - "Component"
    - "System_Component"
    - "System"
    - "Building_System"
    - "Building"

- "Complex"
- "Site"
- "Inspection"
- "Inspection_Data"

3. **string imgTitle**. Title for the attachment.
4. **string imgDesc**. Description of what the attachment represents.
5. **byte[ ] attachment**. The array containing the information for the attachment. This is declared and populated in the code, rather than a value being directly provided in a parameter.
6. **string fileExtension**. accdb, bmp, doc, docx, jpeg, jpg, pdf, png, xlsx, or zip

This code will transfer one byte at a time of a file or image into an attachment. Before running it, make the following replacements:

- Replace FULL FILE DIRECTORY PATH with the path of the image to be attached, including the file name of the image/attachment and its extension.
- Replace OWNER_ID with the string representation of the appropriate GUID.
- Replace "OWNER_LEVEL" with one of the following: "Section_Detail" "Section" "Component_Section" "Component" "System_Component" "System" "Building_ System" "Building" "Complex" "Site" "Inspection" "Inspection_Data"
- Replace IMG_TITLE and IMG_DESCR with strings you enter for the title and description of the image, respectively.
- Replace "ATTACHMENT_EXT" with "accdb", "bmp", "doc", "docx", "jpeg", "jpg", "pdf", "png", "xlsx", or "zip"

```
public void addAttachmentWithFileExtension()
{
byte[] attachmentByteArray = null;
string attachmentPath = @"FULL FILE DIRECTORY PATH";
FileStream fileStream = new FileStream(attachmentPath, FileMode.Open, FileAccess.Read);

using (BinaryReader reader = new BinaryReader(fileStream))
    {
       attachmentByteArray = new byte[reader.BaseStream.Length];
       for (int i = 0; i < reader.BaseStream.Length; i++)
       {
               attachmentByteArray[i] = reader.ReadByte();
       }
    }

// in the API call, acceptable values for ATTACHMENT_EXT are:
// accdb, bmp, doc, docx, jpeg, jpg, pdf, png, xlsx, or zip

var response = client.AddAttachmentWithFileExtension(Guid.Parse("OWNER_ID"),
   "OWNER_LEVEL", "IMG_TITLE", "IMG_DESC", attachmentByteArray, "ATTACHMENT_EXT");
}
//returns guid of the attachment
```

Delete Attachment

Code Example: Delete Attachment

DeleteAttachment(Guid attachmentId)

You can use this call to delete one attachment at a time, by GUID.

To use the code, replace ATTACHMENT_ID with the string representation of a valid attachment's GUID (`ID` property).

```
public void deleteAttachment()
{
    var response = client.DeleteAttachment(Guid.Parse("ATTACHMENT_ID"));
}
// returns Boolean: true = attachment deleted
```

PERFORMANCE METRICS

About Performance Metrics

Performance metrics (such as condition index) can not be directly manipulated using BUILDER API. However, you can access the information by using the service call

```
GetPerformanceRecords(Guid ownerlink, PerformanceRecordType metric, int year)
```

using the GUID of the inventory item ("ownerlink"), stating the desired record type to obtain ("metric"), and specifying the year ("year").

Scope: All inventory levels except Section Detail.

This service call can be used at all inventory levels from Section through Organization, by providing the GUID of the inventory object as the parameter. The format is provided in "GetPerformanceRecords(Guid ownerlink, PerformanceRecordType metric, int year)" on the next page.

> **Best Practice:** Best practice is to run a rollup before getting performance records, to ensure that the desired metric(s) are obtained from the most current data available in the system.

Available Metric Types

The value to use for the parameter "metric" is governed by the `PerformanceRecordType` enum. Either the integer value or the enum should work equally well. The second parameter in the GetPerformanceRecords code example shows how to use an enum as a PerformanceRecordType property.

0 = CI

1 = FI

2 = RCI (ROOFER Condition Index)

3 = FCI (Facility/Financial Condition Index)

4 = PI

5 = All types listed above.

Year

Only a single year can be specified for the parameter "year".

Get Performance Metrics

Code Example: Get Performance Records

Code examples are in C#.

GetPerformanceRecords(Guid ownerlink, PerformanceRecordType metric, int year)

> **Note:** The format of the call to get performance records for an asset does not depend on the inventory level of the associated asset.
>
> **Best Practice:** Best practice is to run a rollup before getting performance records, to ensure that the desired metric(s) are obtained from the most current data available in the system.

The GUID called for in the signature of this call is the GUID of the asset or inspection to get performance records for.

To use the code,

- Replace OWNER_ID with the string representation of an inspection's or an inventory asset's GUID.
- Select a metric type. CI is shown here as an example of how to use the enum (rather than the integer value).
- Specify a year, only one year. 2019 is used in the example.

```
public void getPerformanceRecords()
{
    var PerformanceRecords = client.GetPerformanceRecords(Guid.Parse("OWNER_ID"),
       PerformanceRecordType.CI, 2019).Data;

    foreach(var record in PerformanceRecords)
    {
        Console.WriteLine("CI: " + record.Value.ToString());
    }
    Console.Read();
}
// returns array of DTOPerformanceRecord
```

WORK CONFIGURATION

Scope: The currently available work configuration calls operate at the Organization level only.

BUILDER work configuration is an advanced level task. It involves designing business rules for work generation, and also involves working with data sets.

This chapter shows the API calls related to work configuration that are currently available.

> **CAUTION:** In BUILDER API Version 2022, selected calls related to work configuration have been included to help with obtaining information requisite for creating scenarios. They are not sufficient for performing work configuration.

Standards

There are no active API calls related to standards at present.

Policies

There are no active API calls related to standards at present.

Get Policy Sequences

Code Example: Policy Sequences

Code examples are in C#.

Scope: The currently available work configuration calls operate at the Organization level only.

GetOrgPolicySequences(Guid id)

This call returns all policy sequences associated with the specified Organization.

To use this code, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getOrgPolicySequences()
{
    var response = client.GetOrgPolicySequences(Guid.Parse("ORG_ID"));
}
// returns array of DTOPolicySequence
```

Get FCI Policies

Code Example: FCI Policies

Code examples are in C#.

Scope: The currently available work configuration calls operate at the Organization level only.

GetOrgFCIPolicies(Guid id)

This call returns all FCI policies associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID. The call returns an array of DTOFCIPoliicy, and in each FCI policy, the FCIPolicyOwnerID property is the GUID of the Organization.

```
public void getOrgFCIPolicies()
{
    var response = client.GetOrgFCIPolicies(Guid.Parse("ORG_ID"));
}
// returns array of DTOFCIPolicy
```

Get Prioritization Schemes

Code Example: Prioritization Schemes

Code examples are in C#.

Scope: The currently available work configuration calls operate at the Organization level only.

GetOrgPrioritizationSchemes(Guid id)

This call returns all prioritization schemes associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID. The call returns an array of DTOPrioritizationScheme, and in each prioritization scheme, the PrioritizationSchemeOwnerID property is the GUID of the Organization.

```
public void getOrgPrioritizationSchemes()
{
    var response = client.GetOrgPrioritizationSchemes(Guid.Parse("ORG_ID"));
}
// returns array of DTOPrioritizationScheme
```

DATA LIBRARIES

Work configuration can also involve choosing between default and customized data libraries. Data libraries (aka "books" or "data sets") are used for the following:

- Cost data libraries (cost books): Costs used for assets in the inventory.
- Cost modifier libraries, used with Inventory Cost Modifiers.
- Inflation data libraries: Inflation factors that are applied each year to the costs being used for assets.
- Remaining Service Life (RSL) data libraries: Figures used for average life span of various assets. When combined with the age of an asset, this provides an estimated remaining service life for the asset.
- Component Importance Index (CII) data libraries. (Not included in the BUILDER API at this time.)

Working with data libraries requires an advanced level of permissions. However, working with cost records might not require as elevated a permissions level.

Asset costs can be adjusted using the Cost and Inflation data sets, and by using Inventory Cost Modifiers.

Cost Data Libraries (Cost Books)

This topic explains how to get, create, or update a cost book, also known as a CostMOA. The 2022 BUILDER API does not currently contain a call for deleting a cost book.

You can get a single cost book directly by its GUID; you can get all the cost books associated with a given Organization; and you can get all the cost books associated with a given source.

Code Examples: Cost Data Sets

Code examples are in C#.

GetCostMOA(Guid id)

This call allows you to get a single cost book/cost MOA.

To use the example code given below, replace MOA_ID with the string representation of a cost book's GUID (its `MOA_ID` property).

Tip: To find the GUID of a cost book, you can start with the GUID of an Organization the cost book is employed for and use the call GetCostMOAForOrganizationId.

```
public void getCostMOA()
{
    var response = client.GetCostMOA(Guid.Parse("MOA_ID"));
}
// returns DTOCostMOA
```

GetCostMOAForOrganizationId(Guid guid)

This call returns the cost books associated with the specified Organization.

To use the code below, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getCostMOAForOrganizationId()
{
    var response = client.GetCostMOAForOrganizationId(Guid.Parse("ORG_ID"));
}
// returns array of DTOCostMOA
```

GetCostMOAForSource(string costbook)

This call will get the cost book associated with a given source.

To use this code, replace SOURCE_VALUE with a Source identifier, which will be a string.

```
public void getCostMOAForSource()
{
    var response = client.GetCostMOAForSource("SOURCE_VALUE");
}
// returns DTOCostMOA
```

CreateCostMOA(DTOCostMOA costbook)

When you invoke this call, the BUILDER API will initiate the new cost book by copying over the cost records from the Reference cost book into your newly created cost book. Namely, the Reference cost book is the starting point, after which you can use the call "UpdateCostRecord(DTOCostMOA costMOA)" on page 106 to change records in your new cost book. No other cost book aside from Reference can be the starting point when using BUILDER API to create a new cost book.

To use the sample code, replace OWNER_ID with the string representation of the GUID of the cost book owner (This can be an Organization or a Site), and replace NOT DUPL with a name for the new cost book.

Note about the Owner ID property: Data sets can be assigned both at the Organization level and at the Site level, and the data set designated at the Site level has priority.

The properties listed in the code below are only those that are required by BUILDER for validation of the object; they might not be everything needed for a cost book that is usable.

```
public void createCostMOA()
{
    var newCostBook = new <service reference name>.DTOCostMOA();

    //set properties here
    newCostBook.Owner_ID = "OWNER_ID";
    newCostBook.MOA_Name = "NOT DUPL"; // cannot be a duplicate name
    newCostBook.Min_Cost = 0;
    newCostBook.Min_Paint_Cost = .01;

    var response = client.CreateCostMOA(newCostBook);
}
//returns guid of the new cost book
```

UpdateCostMOA(DTOCostMOA costMOA)

Use this call to update information about a cost book.

Note: If what you want to do instead (or additionally) is to update the records in the cost book, use the call "UpdateCostRecord(DTOCostMOA costMOA)" on page 106. For example, if you have a newly created cost book, it will be populated with the records from the Reference cost book, and you may want to change some of those records in the new cost book.

In the code below, replace "COST_BOOK_ID" with the text representation of the GUID of a cost book.

```
public void updateCostMOA()
{
    //get the cost book to be updated
    var costBook = client.GetCostMOA(Guid.Parse("COST_BOOK_ID"));

    //declare a new instance and populate
    var updateBook = new DTOCostMOA();
    updateBook = costBook.Data;

    // Insert lines below to add or change property values as needed

    var response = client.UpdateCostMOA(updateBook);

    //error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
//returns Boolean: true = update successful
```

Cost Modifier Libraries

If you have the appropriate advanced permissions in BUILDER, you can set up one or more cost modifier libraries for other users to take advantage of when adding and assigning cost modifiers. This topic explains how to use the BUILDER API to get all of the available libraries by organization and how to get, create, update, and delete cost modifier libraries.

> **Note:** A cost modifier library specifies properties of the library. To get a list of the modifiers in a library does not require as high permissions as working with libraries themselves. Calls related to modifier lists can be found at "Get Modifier Lists" on page 72.

Example Code: Get Cost Modifier Libraries

Code examples are in C#.

GetAvailableLibrariesByOrg(Guid id)

This call will get all of the cost modifier libraries available for the Organization specified in the parameter by its GUID. A cost modifier library contains modifiers of potentially different scopes, so when getting available libraries by Organization, no inventory level needs to be specified as it does, for example, when getting modifiers themselves.

> **Note:** An alternative call is "GetModifierLibraryByOrg(Guid id)" on the next page, which returns just one cost modifier library, namely the first available one for the Organization specified by GUID in the parameter.

The code below gets an array of all available cost modifier libraries for a given Organization and stores it in an instance of DTOCostModifierLibrary[ ] named "orgModLibraries".

To use the code, replace ORG_ID with the string representation of an Organization's GUID:

```
public void getAvailableModifierLibrariesByOrg()
{
    var orgModLibraries = client.GetAvailableLibrariesByOrg(Guid.Parse("ORG_ID"));
}
// returns array of DTOCostModifierLibrary
```

GetModifierLibrary(Guid id)

The code below gets a cost modifier library by its GUID and stores it in an instance of DTOCostModifierLibrary named "modLibrary".

To use the code, replace MOD_ID with the string representation of a cost modifier library's GUID.

```
public void getModifierLibrary()
{
    var modLibrary = client.GetModifierLibrary(Guid.Parse("MOD_LIB_ID"));
}
// returns DTOCostModifierLibrary
```

GetModifierLibraryByOrg(Guid id)

In contrast to the call GetAvailableLibrariesByOrg, this call will get just the first of the cost modifier libraries available for the Organization specified in the parameter by its GUID.

The code below gets the first available cost modifier library for a given Organization and stores it in an instance of DTOCostModifierLibrary named "orgModLibrary1".

To use the code, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getModifierLibraryByOrg()
{
   var orgModLibrary1 = client.GetModifierLibraryByOrg(Guid.Parse("ORG_ID"));
}
// returns DTOCostModifierLibrary
```

Example Code: Create Cost Modifier Library

CreateModifierLibrary(DTOCostModifierLibrary modLibrary)

The code below shows the minimum code (what the BUILDER API validates) for creating a new modifier library.

To use the code, you will need to know the GUID of the Organization you want the library associated with. Enter a name for the cost modifier library where prompted, and replace ORG_ID with the string representation of the Organization's GUID.

```
public void createModifierLibrary()
{
    var modLibrary = new <service reference name>.DTOCostModifierLibrary();

    modLibrary.CostModLibraryName = "ENTER VALUE HERE";
    modLibrary.OrganizationId = Guid.Parse("ORG_ID"); // Assign the library to an Org

    // Add more properties as needed

    var response = client.CreateModifierLibrary(modLibrary);
}
// returns guid of the new modifier library
```

Example Code: Update Cost Modifier Library

UpdateModifierLibrary(DTOCostModifierLibrary modLibrary)

To use the code below, replace MOD_ID with the string representation of a cost modifier library's GUID.

```
public void updateModLibrary()
{
    //get the cost modifier library to be updated
    var modLibrary = client.GetModifierLibrary(Guid.Parse("MOD_ID"));

    //declare a new instance and populate
    var updateModLibrary = new <service reference name>.DTOCostModifierLibrary();
    updateModLibrary = modLibrary.Data;

    //add properties or change property values as needed, for example
    updateModLibrary.CostModLibraryName = "MOD_LIB_NAME";

    var response = client.UpdateModifierLibrary(updateModLibrary);

    //error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
// returns Boolean: true = update successful
```

Example Code: Delete Cost Modifier Library

DeleteModifierLibrary(Guid id)

If you no longer need to use a particular collection of modifiers as a library any more, you can use this call to delete the library.

To use the code below, replace MOD_LIB_ID with the string representation of a cost modifier library's GUID.

```
public void deleteModifierLibrary()
{
    var response = client.DeleteModifierLibrary(Guid.Parse("MOD_LIB_ID"));
}
// returns Boolean: true = modifier library deleted
```

Other Data Libraries

Code Examples: Other Data Sets

Code examples are in C#.

GetOrgInflationSets(Guid id)

This call returns all inflation sets associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID.

```
public void getOrgInflationSets()
{
    var response = client.GetOrgInflationSets(Guid.Parse("ORG_ID"));
}
//returns array of DTOInflationSet
```

GetOrgRSLSets(Guid id)

This call returns all RSL (remaining service) life sets associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID.

```
public void getOrgRSLSets()
{
    var response = client.GetOrgRSLSets(Guid.Parse("ORG_ID"));
}
//returns array of DTORSLSet
```

COST RECORDS

Cost records help with estimation of costs by storing the expected cost for each type of Component-Section. A cost book (DTOCostMOA) is a collection of cost records. In the cost record properties, the property MOA_Link records the GUID of the cost book it belongs to.

Using the BUILDER API, you can get and update cost records. It is especially important to be able to update cost records, because when a new cost book is formed using the BUILDER API, it is automatically initiated with the records found in the Reference cost book. There is no call to delete a cost record.

Get Cost Records

Example Code: Get Cost Records

Code examples are in C#.

GetCostRecord(Guid guid)

This call returns a single cost record, specified by its GUID.

To use the call given below, replace COST_RECORD_ID with the text representation of a real cost record GUID.

```
public void getCostRecord()
{
    var response = client.GetCostRecord(Guid.Parse("COST_RECORD_ID"));
}
//returns DTOCostRecord
```

GetCostRecordBySourceId(string id)

This call returns an array of the cost records associated with the specified "source", which can be the name of the cost records' source or another alternate ID.

> **IMPORTANT:** For this call, the input parameter `id` needs to be in string format.

To use the call given below, replace STRING_SOURCE_ID with the name of the source you want to get the cost records from.

```
public void getCostRecordBySourceId()
{
    var response = client.GetCostRecordBySourceId("STRING_SOURCE_ID");
}
//returns array of DTOCostRecord
```

GetRecordsByCostBookId(Guid guid)

This call returns an array of the cost records in the specified cost book.

To use the call given below, replace COST_BOOK_ID with the text representation of a real cost book GUID.

```
public void getRecordsByCostBookId()
{
    var response = client.GetRecordsByCostBookId(Guid.Parse("COST_BOOK_ID"));
}
//returns array of DTOCostRecord
```

Update Cost Record

Use the call shown here to edit properties in an existing cost record.

Code Example: Update Cost Record

Code examples are in C#.

UpdateCostRecord(DTOCostMOA costMOA)

To update the records in a cost book, use this call. For example, if you have a newly created cost book, it will be populated with the records from the Reference cost book, and you may want to change some of those records in the new cost book.

> **Note:** If what you want to do instead (or additionally) is to update information about the cost book the records are contained in, use the call "UpdateCostMOA(DTOCostMOA costMOA)" on page 100 .
>
> **CAUTION:** This call returns the GUID of the cost record instead of a Boolean indicating success or failure, so the response variable is not an indicator of call success.

To use the code below, replace "COST_RECORD_ID" with the text representation of the GUID of a cost record.

```
public void updateCostRecord()
{
    //get the cost record to be updated
    var costRecord = client.GetCostRecord(Guid.Parse("COST_RECORD_ID"));

    //declare a new instance and populate
    var uRecord = new DTOCostRecord();
    uRecord = costRecord.Data;

    // Insert lines below to add properties or change property values as needed

    var response = client.UpdateCostRecord(uRecord);

    //INCLUDE error message code because response variable is GUID, not Boolean
    if (response.Success == false)
        Console.WriteLine("update failed");
}
//returns GUID of the cost record
```

FUNDING

Using the BUILDER API, you can get funding sources associated with a given Organization, fund funding sources, and update funding records.

Get Funding

Calls listed here allow you to find out where funding is coming from for an Organzation, and to get details about funding within that source by examining associated DTOFunding objects. The DTOFunding object contains properties such as `FundingID`, `FundingAmount`, and `FundingYear`.

Code Examples: Get Funding

Code examples are in C#.

GetFundingSourcesForOrganizationId(Guid guid)

This call returns an array of the sources of work funding streams for the specified Organization.

To use the code below, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getFundingSourcesForOrganizationId()
{
    var response = client.GetFundingSourcesForOrganizationId(Guid.Parse("ORG_ID"));
}
// returns (service response) array of DTOFundingSource
```

GetFundingForFundingSource(Guid guid)

Once a funding source has been identified by its GUID (the funding source's `FSID` property), you can examine the DTOFunding objects associated with it.

> **Tip:** To find the GUID of a funding source, you can use the call "Get Funding" above.

To use the call given below, replace FUND_SOURCE_ID with the string representation of a funding source's GUID.

```
public void getFundingForFundingSource()
{
    var response = client.GetFundingForFundingSource(Guid.Parse("FUND_SOURCE_ID"));
}
// returns array of DTOFunding
```

Update Funding Record

Code Example: Update Funding Record

UpdateFundingRecord(DTOFunding fund)

The code to update funding contains two steps to get the funding record to be updated:

1. Retrieve array of funding records by funding source. This makes the properties of the pre-updated funding records available.
2. Identify the desired funding record item in the array. In the sample code below, this happens after the variable `updateFund` is declared to contain the new (updated) funding record.

To use the code given below, replace FUND_SOURCE_ID with the string representation of a funding source's GUID (its `FSID` property).

```
public void updateFundingRecord()
{
    //get the array containing the funding record to be updated
    var funds = client.GetFundingForFundingSource(Guid.Parse("FUND_SOURCE_ID"));

    //declare new instance, populate with the first funding record in the array AS EXAMPLE
    var updateFund = new <service reference name>.DTOFunding();
    var updateFund = funds.Data[0];
       // now updateFund has all the properties of the first record

    //add properties or change property values as needed; for example:
    updateFund.FundingAmount = 180000.00;

    var response = client.UpdateFundingRecord(updateFund);

    //optional error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
// returns Boolean: true = successful update
```

WORK GENERATION (Work Items)

Work generation in BUILDER involves generating work items, work projects (optional), and work plans. However, current API functionality with respect to work generation is limited to work items.

BUILDER API allows you to implement the following list of basic work item capabilities via the API.

- Get a work item
- Get the activity setting for a work item
- Create a work item
- Update a work item
- Delete a work item
- Retrieve a collection of all work items for a Site
- Retrieve a collection of all work items for a Complex
- Retrieve a collection of all work items for a Facility

Get Work Item

Scope: Site, Complex, Facility; or an individual work item.

The calls described here are for official BUILDER work items. To examine work items generated by a scenario, refer to the call "GetScenarioWorkItems(Guid id, int skip, int take)" on page 117.

The options for getting work items are:

- GetWorkItem
- GetSiteWorkItems
- GetComplexWorkItems
- GetFacilityWorkItems

There is also a call for getting the activity option selected for a work item:

- GetActivity

Code Examples: Get Work Item Information

Code examples are in C#.

GetWorkItem(Guid id)

When you want to examine a single work item and know its GUID, use this call.

To use the code given below, replace ID with the string representation of a work item's GUID.

```
public void getWorkItem()
{
    var response = client.GetWorkItem(Guid.Parse("ID"));
}
// returns one DTOWorkItem
```

GetSiteWorkItems(Guid id, int skip, int take)

This call returns all work items for the specified Site.

To use the code given below, replace SITE_ID with the string representation of a Site's GUID. Adjust `skip` and `take` as needed.

```
public void getSiteWorkItems()
{
    var response = clientGetSiteWorkItems(Guid.Parse("SITE_ID"),0, 100);
}
// returns paged collection of DTOWorkItem
```

GetComplexWorkItems(Guid id, int skip, int take)

This call returns all work items for the specified Complex.

To use the code given below, replace CPX_ID with the string representation of a Complex's GUID. Adjust `skip` and `take` as needed.

```
public void getComplexWorkItems()
{
    var response = clientGetComplexWorkItems(Guid.Parse("CPX_ID"), 0, 100);
}
// returns paged collection of DTOWorkItem
```

GetFacilityWorkItems(Guid id, int skip, int take)

This call returns all work items for the specified Facility.

To use the code given below, replace FAC_ID with the string representation of a Facility's GUID. Adjust `skip` and `take` as needed.

```
public void getFacilityWorkItems()
{
    var response = clientGetFacilityWorkItems(Guid.Parse("FAC_ID"), 0, 100);
}
// returns paged collection of DTOWorkItem
```

GetActivity(Guid id, int skip, int take

This call will return a work item activity.

To use the code, replace ACTIVITY_ID with the string representation of an Activity's GUID. You can find the GUID as the `ActivityID` property of a work item.

```
public void getActivity()
{
    var response = client.GetActivity(Guid.Parse("ACTIVITY_ID"));
}
// returns one DTOActivity
```

Create Work Item

In BUILDER, a work plan is what will define, guide, and track progress on sustainment, modernization, or demolition work that needs to be done. Creating the work plan is the goal of the inventory, assessment, data reference, and work configuration efforts that precede it.

In the BUILDER software, a work plan is an aggregate of work items. The call below shows how to create a work item in the BUILDER API.

Code Example: Create Work Item

Code examples are in C#.

CreateWorkItem(DTOWorkItem workItem)

The code sample given here does not accomplish creation of a complete work item; rather, it shows the minimum necessary to be validated by BUILDER.

To meet this minimum, you will need to know the Section ID and the cost activity ID. You will also need to know the work item type, which is an enum. Although there are four options for the `WorkItemType` enum, only 1 and 2 are functional in the BUILDER API.

The `WorkItemType` enum maps values to status choices as follows:

1 = ComponentSectionWorkItem

2 = FacilityWorkItem

To use the code below,

- Replace SEC_ID with the string representation of the GUID of the Section the work item is associated with (in this code example, the work item type is a Section work item).
- Replace COST_ACTIVITY_ID with the string representation of the GUID of the cost activity.

```
public void createWorkItem()
{
    var csWorkItem = new BuilderPreview.DTOWorkItem();

    // Set properties. WorkItemType is an enum
    csWorkItem.Type = <service reference name>.WorkItemType.ComponentSectionWorkItem;
    csWorkItem.SectionID = Guid.Parse("SEC_ID");
    csWorkItem.FiscalYear = 2022;
    csWorkItem.ActivityID = Guid.Parse("COST_ACTIVITY_ID");
    csWorkItem.EstimateCostOverwrite = false;
    //add additional work item property values

    var response = client.CreateWorkItem(csWorkItem);
}
//returns guid of new work item
```

Update Work Item

Subject to your BUILDER user permission level and scope, you can update a work item using BUILDER API. Example code for the basics of an update is given below, followed by further explanation of the update pattern.

Code Example: Update Work Item

Code examples are in C#.

UpdateWorkItem(DTOWorkItem workItem)

To use this code, replace ID with the string representation of the work item to be updated.

```
public void updateWorkItem()
{
    //get the work item to be updated
    var wItem = client.GetWorkItem(Guid.Parse("ID"));

    //declare a new dto instance and populate
    var uWorkItem = new <service reference name>.DTOWorkItem();
    var uWorkItem = wItem.Data;

    //set or change properties here
    uWorkItem.ID = wItem.Data.ID;   // populate the new instance
    uWorkItem.Type = wItem.Data.Type;   // this will probably not change
    uWorkItem.FiscalYear = wItem.Data.FiscalYear;
```

```
    // Insert lines below to add or change property values as needed

    //call update function with the new dto object as the parameter
    var response = client.UpdateWorkItem(uWorkItem);

    //error message code
    if (response.Success == false)
        Console.WriteLine("update failed");
}
//returns Boolean: true = update successful, and the mapping is correct
```

Delete Work Item

Code Example: Delete Work item

Code examples are in C#.

DeleteWorkItem(Guid id)

Replace WORK_ITEM_ID with the string representation of a real work item's GUID to delete the work item.

```
public void deleteWorkItem()
{
    var response = client.DeleteWorkItem(Guid.Parse("WORK_ITEM_ID"));
}
// returns Boolean: true = delete successful
```

SCENARIOS

Scenarios allow the user to experiment, to apply different funding options and levels and see the outcome without changing their actual BUILDER data.

Something to keep in mind while working with the Scenarios API calls is that there is a separate object type, DTOScenarioOrganization, that is different from DTOOrganization. If you call CreateScenarioOrganizationsForScenario to specify some but not all of a set of sibling Organzations to use with a Scenario, DTOScenarioOrganization is what you make use of to specify the Organizations you want to select.

Run Scenario

Code Example

InitiateScenario(Guid id)

This call will run the scenario specified in the parameter. To use the code shown, replace SCENARIO_ID with the string representation of a scenario's GUID.

```
public void initiateScenario()
{
    var inQ = client.InitiateScenario(Guid.Parse("SCENARIO_ID"));
}
// returns Boolean: true = scenario put in queue to be run
```

Get Scenario Information

The code examples here are for examining scenarios and getting other information about scenarios, such as an array of work items generated by a particular scenario.

Code Examples: Get Scenario Information

Code examples are in C#.

GetOrgScenarios(Guid id)

This call returns an array of all scenarios in the Organization specified in the parameter. The call is also useful for tracking down the GUID of a scenario in order to use it as a parameter in another call such as GetScenarioStatus.

To use this code, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getOrgScenarios()
{
    var response = client.GetOrgScenarios(Guid.Parse("ORG_ID"));
}
// returns array of DTOScenario
```

GetScenario(Guid id)

This call returns (but does not automatically display) a scenario. The input parameter that you need to provide is the GUID of a scenario.

> **Tip:** One way to come up with the GUID of a scenario is to run the call "GetOrgScenarios(Guid id)" on the previous page.

To use the code, replace SCENARIO_ID with the string representation of a scenario's GUID.

```
public void getScenario()
{
    var response = client.GetScenario(Guid.Parse("SCENARIO_ID"));
}
// returns DTOScenario
```

GetScenarioStatus(Guid id)

This call gets the status of the scenario specified by its GUID. To use the call given below, replace SCENARIO_ID with the string representation of a scenario's GUID.

```
public void getScenarioStatus()
{
    var response = client.GetScenarioStatus(Guid.Parse("SCENARIO_ID"));
}
// Returns an enum that indicates status of the scenario
```

The value of the enum returned shows the status of the scenario, as follows:

0	=	Stopped
1	=	Start
2	=	Running
3	=	Queued
100	=	Complete
-1	=	NeverRun
-2	=	ErrorOccurred
-3	=	BeingAborted
-4	=	Aborted

GetScenarioWorkItemCount(Guid id)

This call returns a count of the work items generated by the scenario specified in the parameter.

To use the code shown, replace ID with the string representation of a scenario's GUID, which is the `ID` field of the scenario.

```
public void getScenarioWorkItemCount()
{
    var response = client.GetScenarioWorkItemCount(Guid.Parse("SCENARIO_ID"));
}
// returns integer that is count of the scenario work items
```

GetScenarioWorkItems(Guid id, int skip, int take)

This call returns an array of the work items generated by the specified scenario.

> **Caution:** The work items returned are in the work item format. But keep in mind that these are not actual work items created and stored in BUILDER. They are "scenario work items" only; that is, they are hypothetical and exist only in the framework of the scenario.

To use this code, replace SCENARIO_ID with the with the string representation of a scenario's GUID, which is the `ID` field of the scenario.

```
public void getScenarioWorkItems()
{
    var response = client.GetScenarioWorkItems(Guid.Parse("SCENARIO_ID"), 0, 100);
}
// returns array of DTOWorkItem
```

GetAvailableScenarioSystems(Guid ID)

This call returns all Systems in the scope of the scenario specified in the parameter.

To use this code, replace SCENARIO_ID with the with the string representation of a scenario's GUID, which is the `ID` field of the scenario.

```
public void getAvailableScenarioSystems()
{
    var response = client.GetAvailableScenarioSystems(Guid.Parse("SCENARIO_ID"));
}
// returns array of DTOScenarioSystem
```

Create Scenario

When you create a scenario, many properties will need to be configured. This section will guide you through the process. It covers

- Creating a scenario with the minimum required properties
- Creating scenario Organizations and Systems (two separate calls)
- Changing funding from unconstrained to constrained for a scenario
- Retrieving Organization information needed for the minimum required properties

Code Examples: Create and Configure Scenario

Code examples are in C#.

CreateScenario(new DTOScenario dtoScenario)

The code example below shows the minimum required properties for a scenario, with sample values. Each required property, except `newScenario.Name`, is explained after the end of the code example.

```
public void createScenario();
{
    var newScenario = new <service reference name>.DTOScenario();

    //Set the minimum properties validated by BUILDER

    newScenario.Name = "<scenario name string>";

    newScenario.CopyFirstYear = false;
    newScenario.CreateSampleWorkItems = false;
    newScenario.FundingConstrained = false;   // if true, must define add'l properties
    newScenario.InflationConstrained = false;
    newScenario.PreviousFYWorkCompleted = false;  // if true, define add'l properties
    newScenario.FCI_Funding = false;
    newScenario.UseAllSystems = true;
    newScenario.SimulationYears = 6;

    newScenario.Org_ID = Guid.Parse("ORG_ID");
    newScenario.CostBookID = Guid.Parse("COST_BOOK_ID");
    newScenario.FCIPolicyID = Guid.Parse("FCI_POLICY_ID");
    newScenario.InflationBookID = Guid.Parse("INFLATION_BOOK_ID");
    newScenario.PolicySequenceID = Guid.Parse("POLICY_SEQ_ID");
    newScenario.PrioritizationSchemeID = Guid.Parse("PRIORITIZATION_SCHEME_ID");
    newScenario.RSLBookID = Guid.Parse("RSL_BOOK_ID");

    //Make the call
    var response = client.CreateScenario(newScenario);
}
//returns ID of the newly created scenario if successful
```

newScenario.CopyFirstYear

(In the BUILDER web application, this is a checkbox under the **Scope** tab.)

- If false, scenario will start from scratch to create first year's work items
- If true, will copy in current work items for first year instead of creating new

newScenario.CreateSampleWorkItems

(In the BUILDER web application, this is a checkbox under the **Scope** tab.)

If you set this property to true, you don't need to create sample work items yourself; this property is simply a setting saying whether you want to allow or not allow sample work items to be generated in the scenario (based on sample inspections).

- If false, disregard any sampling
- If true, sample work items may be generated by the scenario (if there are sample inspections)

newScenario.FundingConstrained

- In the example code, this has been set to false
- If true, you will need to define additional properties (see the call "CreateConstrainedFundingForScenario(DTOFundingSource dtoFundSource, Guid id)" on page 124

newScenario.InflationConstrained

- If false, no inflation will be applied
- If true, will apply inflation configuration set forth in the active inflation book (see "newScenario.InflationBookID" on page 121)

newScenario.PreviousFYWorkCompleted

- If false, BUILDER assumes that NONE of the previous years' work has been completed
- If true, all previous FY work items will be treated as having been completed for the years specified in the scenario's `PreviousFY` property

> **CAUTION:** Before you build the string to be used as the `PreviousFY` property, make sure that each of the years you select actually contains work items. If you try to "trust" (that work items have been completed) a year in which there are no work items, the call might not work properly.

In the BUILDER Web interface, the list of fiscal years to choose from has been populated with only the years that contain work items. But the API does not validate for presence of work items in the fiscal years selected.

newScenario.FCI_Funding

(This property corresponds to the "Use FCI Objectives to Generate Funding Profile" checkbox at the **Funding** tab in the BUILDER web application.)

- If false, FCI objectives will not be used to generate the funding profile
- If true, FCI objectives will be used to generate the funding profile

newScenario.UseAllSystems

(When true, this property corresponds to marking the "Select All" checkbox at the **Scope** tab when creating a new scenario in the BUILDER web application.)

If this property is not officially validated by the BUILDER API, if it is not set to either true or false, and thus Null, the API will probably throw an error.

- If true (the default), all available Systems will be used when running the scenario
- If false, you need to configure what Systems are to be used by running the call "CreateScenarioSystemsForScenario(Guid id, DTOScenarioSystem[ ] systems, bool useAllSystems)" on page 123.

Available systems can be determined by running the call "GetAvailableScenarioSystems (Guid ID)" on page 117

newScenario.SimulationYears

Enter the number of years you want to simulate, as an integer.

newScenario.Org_ID

Replace ORG_ID with the string representation of the GUID of the Organization the scenario is associated with.

newScenario.CostBookID

Replace COST_BOOK_ID with the string representation of the GUID of the cost book to be used. One way to find this is to run the call "GetCostMOAForOrganizationId(Guid guid)" on page 126.

newScenario.FCIPolicyID

Replace FCI_POLICY_ID with the string representation of the GUID of the FCI Policy to be used. One way to find this is to run the call "GetOrgFCIPolicies(Guid guid)" on page 126.

newScenario.InflationBookID

Replace INFLATION_BOOK_ID with the string representation of the GUID of the inflation book to be used. One way to find this is to run the call "GetOrgInflationSets(Guid guid)" on page 127.

newScenario.PolicySequenceID

Replace POLICY_SEQ_ID with the string representation of the GUID of the policy sequence to be used. One way to find this is to run the call "GetOrgPolicySequences (Guid guid)" on page 127.

newScenario.PrioritizationSchemeID

Replace PRIORITIZATION_SCHEME_ID with the string representation of the GUID of the prioritization scheme to be used. One way to find this information is to run the call "GetOrgPrioritizationSchemes(Guid guid)" on page 127.

newScenario.RSLBookID

Replace RSL_BOOK_ID with the string representation of the GUID of the RSL (remaining service life) book to be used. One way to find this is to run the call "GetOrgRSLSets(Guid guid)" on page 127.

Code Examples: Create Scenario Organizations and Systems

Creating Scenario Organizations and specifying scenario Systems to be used for the scenario are optional process of selecting specific Organizations in the tree and/or specific Systems to be involved in the scenario. See the description of each call to determine whether you need to use either (or both) of them.

Code examples are in C#.

CreateScenarioOrganizationsForScenario(DTOScenarioOrganization[ ] dtoScenarioOrgs, Guid id)

When this call is needed

An example of when you would use this call is when you have a collection of Organizations at the same level in the tree, and you want to run a scenario on some of these Organizations but not all of them. This is common when trying to create a scenario high up in the inventory tree.

If either of the following is true, you will NOT need this call:

- You want your scenario to include all the suborganizations under the highest level Organization that applies to the scenario, or
- The Organization you want to run the scenario on has no suborganizations

Code preview

The code below shows how to build an array of DTOScenarioOrganization based on the Organizations you want to use if you need to select some but not all of equal-level Organizations. The call itself will need, as its parameters, (1) the array of Organizations you have pre-built, and (2) the ID of the scenario.

> **Best Practice Tip:** Unless the source group of suborganizations from which you will select is small, you should use GetOrganizationsByParentIDPaged to get the Organizations under a parent (and then iterate), because if you simply use GetOrganizationsByParentID, that call could time out.

Ultimately you need to have your array (scenarioOrgs in the code example) completely constructed before making the CreateScenarioOrganizationsForScenario call.

> **Note:** The DTO objects you will be getting and selecting from are of the type DTOOrganization, but the array you will create is populated with DTO objects of the type DTOScenarioOrganization. Hence the manual population of the array. (Only the Organization's ID property needs to be transferred into the new array from the source Organization.)

Prerequisite

The scenario associated with this call must be or have been created before invoking CreateScenarioOrganizationsForScenario because the scenario's ID is one of the parameters.

Example code

```
public void createScenarioOrganizations();
{
    //scenario's Org_ID property is the GUID of the parent Org you'll select Orgs from

    var subOrgs = client.GetOrganizationsByParentIDPaged(Guid.Parse("Scenario.Org_ID"),
       0, 50);
    //Add code to iterate and store in subOrgs

    //Next, create new array for holding the Orgs you want. Note transition from
    // DTOOrganization[ ] in first line(s) of code to DTOScenarioOrganization[ ]
    //
    var scenarioOrgs = new <service reference name>.DTOScenarioOrganization[2];

    //Set up the first item for the array
    var firstOrg = new <service reference name>.DTOScenarioOrganization();
    firstOrg.ScenarioOrgID = Guid.NewGuid();  // each item needs its own GUID assigned
    firstOrg.ScenarioID = Guid.Parse("ScenarioID"); // will be same for all items

    firstOrg.OrganizationID = subOrgs.Data[0].ID;   // want just the ID from the
       //relevant suborganization

    //Pass it into the first slot of the array
    scenarioOrgs[0] = firstOrg;

    //Set up the next item for the array, in this case the last entry in subOrgs
    var lastOrg = new <service reference name>.DTOScenarioOrganization();
    lastOrg.ScenarioOrgID = Guid.NewGuid();        // new guid
    lastOrg.ScenarioID = Guid.Parse("ScenarioID"); // same for all items
    lastOrg.OrganizationID = subOrgs.Data[subOrgs.Data.Length - 1].ID;

    //Pass it into second slot of array
    scenarioOrgs[1] = lastOrg;

    //Make the call
    client.CreateScenarioOrganizationsForScenario(scenarioOrgs, Guid.Parse("ScenarioID"));
}
//returns Boolean: true = success
```

CreateScenarioSystemsForScenario(Guid id, DTOScenarioSystem[ ] systems, bool useAllSystems)

The most common case for a scenario is that all available Systems are selected for the scenario to act on, and that is the default.

However, if you want to select only some Systems to use for the scenario's Organizations, use this call. (The call corresponds to selecting from the System checkboxes below the **Select All** checkbox when creating or editing a scenario in the web application.)

Prerequisites

- The scenario associated with this call must be or have been created before invoking CreateScenarioSystemsForScenario because the scenario's ID is one of the

call's parameters.

- The `.UseAllSystems` property of that scenariodoes *not* need to have been been set to false, because that will be taken care of by the third parameter of this call.
- If applicable to your situation, you need to have already successfully picked selected suborganizations. (See "When this Call Is Needed" at the description of the call CreateScenarioOrganizationsForScenario.)
- You need to know which Systems you want used in the scenario. The call "GetAvailableScenarioSystems(Guid ID)" on page 117 will show what Systems are available.

Undoing the effects of this call

If you use this call to limit which Systems are used for the scenario, and then the situation changes such that you want to use all Systems, the best way to accomplish this is to run the UpdateScenario call with the following line of code, replacing updateScenario with the name of your scenario:

```
updateScenario.UseAllSystems = true
```

Example code

```
public void createScenarioSystems();
{
    var availableSystems = client.GetAvailableScenarioSystems(Guid.Parse("ScenarioID"));
    //The above returns a list of everything available to be selected in the Org
    // associated with the scenario you have supplied the GUID for. It doesn't matter
    // whether the Organization actually contains any Systems of that type yet.

    //Next, create new array to hold the Systems you want the scenario to use.
    var scenarioSystems = new DTOScenarioSystem[2];  // adding 2 systems to scope

    //For this example, add first and last available scenario System to scenario scope
    scenarioSystems[0] = availableSystems.Data[0];
    scenarioSystems[1] = availableSystems.Data[availableSystems.Data.Length-1];


    //When making the call, important to supply false for the last parameter
    //(false means that it is false that you want all systems)
    var response2 = client.CreateScenarioSystemsForScenario(Guid.Parse("ScenarioID"),
        scenarioSystems, false);
}
//returns Boolean: true = success
```

CreateConstrainedFundingForScenario(DTOFundingSource dtoFundSource, Guid id)

Use this call when you want to change a scenario's funding source from unconstrained (the default) to constrained.

If funding is not constrained, funds are divided so that the same amount of money is allocated to each year. If funding is constrained, you select a funding source that predefines how much is available for each year.

When the call succeeds, it changes the scenario's `.Constrained` property to true.

Prerequisites

- ID of the Organization associated with the scenario (the scenario's `Org_ID` property). This will be used as the parameter to "GetFundingSourcesForOrganizationId (Guid guid)" on page 108 in the code example, and is called SCENARIO_ORG_ID in this example.
- ID of the scenario. This is called SCENARIO_ID in the code example. To find the ID of the scenario, you can use the call "GetOrgScenarios(Guid id)" on page 115, passing in the Organization ID described above, and selecting the desired scenario from the array of DTOScenario returned.

You will also need to select which funding source you want to constrain.

Code with higher level of commenting

```
public void createConstrainedFundingForScenario()
{
    // Step 1. Find available funding sources: Step 1 returns
    var funds = client.GetFundingSourcesForOrganizationId(Guid.Parse("SCENARIO_ORG_ID"));

    // Step 2. (NOT PROVIDED)
    // Use logic, or some other approach, to decide which funding source to constrain
    // This code uses as example the first item in the array returned by Step 1.

    // Step 3. Create new instance of DTO object and set properties
    DTOFundingSource constFunds = new <service reference name>.DTOFundingSource()
       //where constFunds is the source to constrain
    constFunds.FSID = funds.Data[0].FSID;
    constFunds.FSName = funds.Data[0].FSName;
    constFunds.Order = funds.Data[0].Order;
    constFunds.Org_ID = funds.Data[0].Org_ID;

    // Step 4. Make the API call, passing in
    // (1) the funding source to be constrained and
    // (2) string representation of the ID of the associated Scenario
    var response = client.CreateConstrainedFundingForScenario(constFunds,
       Guid.Parse("SCENARIO_ID"));
}
// returns Boolean: true = selected source of funds is now constrained
```

Code with limited comments

```
public void createConstrainedFundingForScenario()
{
   var funds = client.GetFundingSourcesForOrganizationId(Guid.Parse("SCENARIO_ORG_ID"));

   // Use logic, or some other approach, to decide which funding source to constrain
```

```
   // set comments
   DTOFundingSource constFunds = new <service reference name>.DTOFundingSource()
      // where constFunds represents the source to constrain
   constFunds.FSID = funds.Data[0].FSID;      // first funding source used as example
   constFunds.FSName = funds.Data[0].FSName;
   constFunds.Order = funds.Data[0].Order;
   constFunds.Org_ID = funds.Data[0].Org_ID;

   var response = client.CreateConstrainedFundingForScenario(constFunds,
      Guid.Parse("SCENARIO_ID"));
}
// returns Boolean: true = selected source of funds is now constrained
```

Code Examples: Get Scenario-Related Organization Information

When you create a scenario using the call CreateScenario, there is a collection of properties that you will need to provide values for. The calls below, duplicated from other locations in the documentation, allow you to get these values.

Code examples are in C#.

GetCostMOAForOrganizationId(Guid guid)

This call returns the cost books associated with the specified Organization.

To use the code below, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getCostMOAForOrganizationId()
{
    var response = client.GetCostMOAForOrganizationId(Guid.Parse("ORG_ID"));
}
// returns array of DTOCostMOA
```

GetOrgFCIPolicies(Guid guid)

This call returns all FCI policies associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID. The call returns an array of DTOFCIPoliicy, and in each FCI policy, the FCIPolicyOwnerID property is the GUID of the Organization.

```
public void getOrgFCIPolicies()
{
    var response = client.GetOrgFCIPolicies(Guid.Parse("ORG_ID"));
}
// returns array of DTOFCIPolicy
```

GetOrgInflationSets(Guid guid)

This call returns all inflation sets associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID.

```
public void getOrgInflationSets()
{
    var response = client.GetOrgInflationSets(Guid.Parse("ORG_ID"));
}
//returns array of DTOInflationSet
```

GetOrgPolicySequences(Guid guid)

This call returns all policy sequences associated with the specified Organization.

To use this code, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getOrgPolicySequences()
{
    var response = client.GetOrgPolicySequences(Guid.Parse("ORG_ID"));
}
// returns array of DTOPolicySequence
```

GetOrgPrioritizationSchemes(Guid guid)

This call returns all prioritization schemes associated with the specified Organization.

To use this code, replace ORG_ID with the string representation of an Organization's GUID.

```
public void getOrgPrioritizationSchemes()
{
    var response = client.GetOrgPrioritizationSchemes(Guid.Parse("ORG_ID"));
}
// returns array of DTOPrioritizationScheme
```

GetOrgRSLSets(Guid guid)

This call returns all RSL (remaining service) life sets associated with the specified Organization.

To use this code, replace ORG_ID with the text representation of a real Organization GUID.

```
public void getOrgRSLSets()
{
    var response = client.GetOrgRSLSets(Guid.Parse("ORG_ID"));
}
//returns array of DTORSLSet
```

Update Scenario

Use the call shown here to add or edit properties in an existing scenario.

Code Example: Update Scenario

Code examples are in C#.

UpdateScenario(DTOScenario dtoScenario)

In Step 3 of this call, you will need to set properties. For each scenario property that you want to change or add, use the format

```
    updatedScenario.<property> = <value>;
```

> **WARNING:** Any scenario properties that are not set, either using `client.GetScenario(Guid.Parse("SCENARIO_ID"))` (Step 2), or explicitly set(Step 3), will be SET TO NULL.

To use the code below, replace SCENARIO_ID with the GUID of the scenario you are updating.

```
public void updateScenario()
{
    //Step 1. Declare new instance of DTO object
    var updatedScenario = new <service reference name>.DTOScenario();

    //Step 2. Populate it using the get function
     updatedScenario = client.GetScenario(Guid.Parse("SCENARIO_ID")).Data;

    //Step 3. Add new properties or assign new values to existing ones, for EXAMPLE:
    updatedScenario.Name = "<new scenario name>";

    //Step 4. Call update function with the new DTO object instance as the parameter
    client.UpdateScenario(updatedScenario);
}
// returns Boolean: true = update successful
```

Below is example code without comments.

```
public void updateScenario()
{
    var updatedScenario = new DTOScenario();

    var myScen = client.GetScenario(Guid.Parse("SCENARIO_ID"));
    updatedScenario = myScen.Data;
    updatedScenario.Name = "<new scenario name>";

    var response = client.UpdateScenario(updatedScenario);
}
```

Compare Scenario Variations

The code example in this topic shows how to make a copy of a scenario to compare between previous FY work trusted to be completed vs not. The same concept can be adapted to compare the effects of different changes as well.

UpdateScenario vs. CreateScenario (general to all types of scenario comparisons)

You can run a scenario comparison in one of two ways:

a. Use the "UpdateScenario(DTOScenario dtoScenario) " on the previous page call to make a new variation of the scenario if it's sufficient to compare the new scenario variation to a previously generated report of the original condition (because the original scenario will be overwritten by the changes).
b. Alternatively, use "CreateScenario(new DTOScenario dtoScenario)" on page 118 to create a new scenario, populate it with the original scenario's data, and then update the new scenario with the changed condition(s) while retaining the original scenario.

How to specify fiscal years to trust

In the example code, the procedure for trusting completion of work in previous fiscal years has two steps:

- Construct a string specifying the years
- Set the property .PreviousFYWorkCompleted = true

These steps can actually be completed in either order.

> **CAUTION:** Before you build the string to be used as the `PreviousFY` property, make sure that each of the years you select actually contains work items. If you try to "trust" (that work items have

been completed) a year in which there are no work items, the call might not work properly.

In the BUILDER Web interface, the list of fiscal years to choose from has been populated with only the years that contain work items. But the API does not validate for presence of work items in the fiscal years selected.

Running the example code

If you know the GUID of the scenario, you can replace the two lines of code following the first comment in the example code with the following one line of code, passing in the scenario's GUID itself (not a text representation) in place of SCENARIO_ID.

```
var scenarioCopy = client.GetScenario(SCENARIO_ID).Data;
```

However, if you don't already know the GUID of the scenario, you will want to use the GetOrgScenarios call (the first line of code below) or another method to find it.

To use the example code, replace ORG_ID with the string representation of the GUID of the Organization associated with the scenario.

```
public void compareScenarioFYTrusted();
{
    //First, get array of the scenarios associated with the Organization, and select a scenario
    var orgScenarios = client.GetOrgScenarios(Guid.Parse("ORG_ID"));
    var scenarioCopy = orgScenarios.Data[0];     // selecting first scenario as the example

    //Next, start the procedure for trusting that work in previous FY's has been completed
    scenarioCopy.PreviousFYWorkCompleted = true;  // doesn't automatically make it true for all years

    var prevFY = "";

    for (int x=0;x < scenarioCopy.SimulationYears; x++)
    {
        if (x == 0 || x == 1)
        {
            //build each year onto the string, separated by commas, and using NO SPACES
            prevFY += (DateTime.Now.Year - 1).ToString() + ",";
        }
    }

    scenarioCopy.PreviousFY = prevFY;

    //Call updates the scenario to trust completion of the work items for the previous years you selected

    var response = client.UpdateScenario(scenarioCopy);
}
//returns Boolean: true = update successful
```

Delete Scenario

DeleteScenario(Guid id)

Replace SCENARIO_ID with the string representation of a real scenario GUID to delete the scenario.

```
public void deleteScenario()
{
    var response = client.DeleteScenario(Guid.Parse("SCENARIO_ID");
}
// returns Boolean: true = scenario deleted
```

BUILDER API ADMINISTRATION

This chapter describes API calls that are reserved for BUILDER administrators. Currently the only reserved call is

```
SetCurrentUser(string commonName)
```

This call enables the "Proxy User" below capability.

Proxy User

The API service call described below can be performed only by users with BUILDER Administrator privilege.

Sample Code: Proxy User Capability

SetCurrentUser(string commonName)

If you have administrative privilege in BUILDER, you can set up one or more special-purpose user accounts and then switch to them as desired by using the proxy user capability in the BUILDER API. An example of why a proxy user account is useful is that you can run tests employing user accounts at different privilege levels.

The service call that enables the proxy user capability is SetCurrentUser. The parameter you need to pass in is the user name for the desired account (called commonName in the call signature). It represents the certificate common name of the user to switch to.

```
SetCurrentUser(string commonName);

// returns Boolean: true = user has been set to the name provided
```

Note: Security will be validated at the server.

REFERENCES

Source of Appendixes:

BUILDER™ Remote Entry Database (BRED) Data Dictionary Version 3.3.7, 2015, Appendix A.

APPENDIX A: UNITS OF MEASURE

This table provides for each unit of measure (UOM):
(1) the UOM_ID; (2) the BUILDER™ storage unit ("metric") abbreviation; (3) the English unit abbreviation;
(4) the conversion factor; and (5) a definition.
Source: BUILDER tm Remote Entry Database (BRED) Data Dictionary Version 3.3.7, 2015, Appendix A.

UOM_ID	UOM_MET_UNIT_ABBR	UOM_ENG_UNIT_ABBR	UOM_CONV	Definition
1	EA	EA	1	Each
2	LM	LF	3.28083399	Linear meter / Linear foot
3	SM	SF	10.76386871	Square meter / Square foot
4	CM	IN	0.39370079	Centimeter / inch
104	SM	SY	1.1959911	Square meter / Square yard
105	CM	CY	1.30795074	Cubic meter / Cubic yard
106	FLT	FLT	1	Flight (stairs)
107	STP	STP	1	Stop (elevator)
108	MBH	MBH	1	Thousands of BTU per Hour
109	TON	TON	1	Tons (English tons only)
110	AMP	AMP	1	Ampere
111	KVA	KVA	1	Kilovolt ampere
112	XX	XX	1	UOM not defined
113	AC	AC	1	Acre
114	BL	BL	1	Barrel
115	FA	FA	1	Family Units
116	FB	FB	1	Linear Feet of Berthing
117	FP	FP	1	Firing Points
118	GA	GA	1	U.S. Gallon
119	GM	GM	1	U.S. Gallons per minute
120	KG	KG	1	Thousands U.S. Gallons/Day
121	TH	TH	1	Tons per hours
122	TR	TR	1	Ton, Refrigeration
123	CF	CF	1	Cubic foot
124	KV	KV	1	Kilovolt
125	KW	KW	1	Kilowatt
126	LN	LN	1	Firing Lane
127	MB	MB	1	Millions of BTU per hour
128	MG	MG	1	Millions of U.S. Gallons
129	MI	MI	1	Kilometer/Mile
130	MW	MW	1	Megawatt
131	NA	NA	1	Not Applicable
132	OL	OL	1	Number of Outlets
133	LBS	LBS	1	Pounds
134	WATT	WATT	1	Watt

APPENDIX B: SYSTEM IDENTIFIER

SYS_ID (BLDG_SYS_LINK)	SYS_DESC	UII_CODE	IS_UII
201	A10 FOUNDATIONS	A10	TRUE
202	A20 BASEMENT CONSTRUCTION	A20	TRUE
203	B10 SUPERSTRUCTURE	B10	TRUE
204	B20 EXTERIOR ENCLOSURE	B20	TRUE
205	B30 ROOFING	B30	TRUE
206	C10 INTERIOR CONSTRUCTION	C10	TRUE
207	C20 STAIRS	C20	TRUE
208	C30 INTERIOR FINISHES	C30	TRUE
209	D10 CONVEYING	D10	TRUE
210	D20 PLUMBING	D20	TRUE
211	D30 HVAC	D30	TRUE
212	D40 FIRE PROTECTION	D40	TRUE
213	D50 ELECTRICAL	D50	TRUE
214	E10 EQUIPMENT	E10	TRUE
215	E20 FURNISHINGS	E20	TRUE
216	F10 SPECIAL CONSTRUCTION	F10	TRUE
217	F20 SELECTIVE BUILDING DEMOLITION	F20	TRUE
218	G10 SITE PREPARATIONS	G10	TRUE
219	G20 SITE IMPROVEMENTS	G20	TRUE
220	G30 SITE CIVIL/MECHANICAL UTILITIES	G30	TRUE
221	G40 SITE ELECTRICAL UTILITIES	G40	TRUE
222	G90 OTHER SITE CONSTRUCTION	G90	TRUE
223	H10 WATERFRONT STRUCTURES	H10	TRUE
224	H20 GRAVING DRYDOCKS	H20	TRUE
225	H30 COASTAL PROTECTION	H30	TRUE
226	H40 NAV DREDGING / RECLAMATION	H40	TRUE
227	H50 WATERFRONT UTILITIES	H50	TRUE
228	H60 WATERFRONT DEMOLITION	H60	TRUE
229	H70 WATERFRONT ATFP	H70	TRUE

APPENDIX C: PAINT TYPE IDENTIFIER

PAINT_TYPE_ID (SEC_PAINT_LINK)	PAINT_TYPE_DESC
10	Alkyd Gloss Enamel
20	Alkyd Glss Enl-Low in VOC
30	Alkyd High Gloss Enamel
40	Alkyd Modified Oil Paint
50	Alkyd Paint
60	Alkyd Primer-Enl-Undrcoat
70	Alkyd Resin Paint
80	Alkyd Semigloss Enamel
90	Alkyd-Resin Varnish
100	Alumin Hear Resist 1200 F
110	Aluminum Paint
120	Asphalt Varnish
130	Chlorin Rubber Intermedia
140	Enamel: Floor and Deck
150	Gen Purpose Wax, Solvent
160	Gloss & Semigloss Latex
170	Heat-Resist 400 degF Enml
180	Iron-Oxide Oil Paint
190	Latex Acrylic Emulsion
200	Latex High Traffic
210	Latex Paint
220	Latex Primer Coating
230	Latex Stain
240	Latex Surface Sealer
250	Low Sheen Oil Varnish
260	Moist Curing Polyurethane
270	Oil Stain
280	Phenolic-Resin Spar Varni
290	Iron/ZincOx,Lnsd Oil/Alkd
300	Rubber Paint (Swim Pools
310	Rubber-Base Paint
320	Rubbing Oil Varnish
330	Semi-Transparen Oil Stain
340	Semigloss Enamel
350	Silicone Alkyd Paint
360	Textured Coating
370	Two-Part Epoxy Coating
380	Varnish Surface Sealer
390	Water-Emulsion Floor Wax
400	Water-Resist Spar Varnish
410	Zinc Rich Phenolic Varnis
420	Zinc-Molybdate Alkyd Prim

APPENDIX D: COMPONENT IDENTIFIER

COMP_ID (SYS_COMP_COMP_LINK)	COMP_ SYS_LINK	COMP_DESC	COMP_IS_ EQUIP
2011	201	A1010 STANDARD FOUNDATIONS	0
2012	201	A1020 SPECIAL FOUNDATIONS	0
2013	201	A1030 SLAB ON GRADE	0
2021	202	A2010 BASEMENT EXCAVATION	0
2022	202	A2020 BASEMENT WALLS	0
2031	203	B1010 FLOOR CONSTRUCTION	0
2032	203	B1020 ROOF CONSTRUCTION	0
2041	204	B2010 EXTERIOR WALLS	0
2042	204	B2020 EXTERIOR WINDOWS	0
2043	204	B2030 EXTERIOR DOORS	0
2051	205	B3010 ROOF COVERINGS	0
2052	205	B3020 ROOF OPENINGS	0
2061	206	C1010 PARTITIONS	0
2062	206	C1020 INTERIOR DOORS	0
2063	206	C1030 SPECIALTIES	0
2071	207	C2010 STAIR CONSTRUCTION	0
2072	207	C2020 STAIR FINISHES	0
2081	208	C3010 WALL FINISHES	0
2082	208	C3020 FLOOR FINISHES	0
2083	208	C3030 CEILING FINISHES	0
2084	208	C3040 INT COATINGS / SPECIAL FINISHES	0
2091	209	D1010 ELEVATORS AND LIFTS	1
2092	209	D1030 ESCALATORS AND MOVING WALKS	1
2093	209	D1020 WEIGHT HANDLING EQUIPMENT	1
2099	209	D1090 OTHER CONVEYING SYSTEMS	1
2101	210	D2010 PLUMBING FIXTURES	1
2102	210	D2020 DOMESTIC WATER DISTRIBUTION	1
2103	210	D2030 SANITARY WASTE	1
2104	210	D2040 RAIN WATER DRAINAGE	1
2105	210	D2090 OTHER PLUMBING SYSTEMS	1
2111	211	D3010 ENERGY SUPPLY	1
2112	211	D3020 HEAT GENERATING SYSTEMS	1
2113	211	D3030 COOLING GENERATING SYSTEMS	1
2114	211	D3040 DISTRIBUTION SYSTEMS	1
2115	211	D3050 TERMINAL & PACKAGE UNITS	1
2116	211	D3060 CONTROLS & INSTRUMENTATION	1
2117	211	D3070 SYSTEMS TESTING & BALANCING	1
2119	211	D3090 OTHER HVAC SYSTEMS AND EQUIPMENT	1
2121	212	D4040 SPRINKLERS	1
2122	212	D4030 STANDPIPE SYSTEMS	1
2123	212	D4050 FIRE PROTECTION SPECIALTIES	1

COMP_ID (SYS_COMP_COMP_LINK)	COMP_ SYS_LINK	COMP_DESC	COMP_IS_ EQUIP
2124	212	D4010 FIRE ALARM AND DETECTION SYSTEMS	1
2125	212	D4020 FIRE SUPP WATER SUPPLY / EQUIP	1
2129	212	D4090 OTHER FIRE PROTECTION SYSTEMS	1
2131	213	D5010 ELECTRICAL SERVICE & DISTRIBUTION	1
2132	213	D5020 LIGHTING & BRANCH WIRING	1
2133	213	D5030 COMMUNICATIONS & SECURITY	1
2139	213	D5090 OTHER ELECTRICAL SERVICES	1
2141	214	E1010 COMMERCIAL EQUIPMENT	1
2142	214	E1020 INSTITUTIONAL EQUIPMENT	1
2143	214	E1030 VEHICULAR EQUIPMENT	1
2144	214	E1040 GOVERNMENT FURNISHED EQUIPMENT	1
2149	214	E1090 OTHER EQUIPMENT	1
2151	215	E2010 FIXED FURNISHINGS	1
2152	215	E2020 MOVEABLE FURNISHINGS	1
2161	216	F1010 SPECIAL STRUCTURES	0
2162	216	F1020 INTEGRATED CONSTRUCTION	0
2163	216	F1030 SPECIAL CONSTRUCTION SYSTEMS	0
2164	216	F1040 SPECIAL FACILITIES	0
2165	216	F1050 SPECIAL CTRLS / INSTRUMENTATION	0
2171	217	F2010 BUILDING ELEMENTS DEMOLITION	0
2172	217	F2020 HAZARDOUS COMPONENTS ABATEMENT	0
2181	218	G1010 SITE CLEARING	0
2182	218	G1020 SITE DEMOLITION & RELOCATIONS	0
2183	218	G1030 SITE EARTHWORK	0
2184	218	G1040 HAZARDOUS WASTE REMEDIATION	0
2191	219	G2010 ROADWAYS	0
2192	219	G2020 PARKING LOTS	0
2193	219	G2030 PEDESTRIAN PAVING	0
2194	219	G2040 SITE DEVELOPMENT	0
2195	219	G2050 LANDSCPAING	0
2196	219	G2060 AIRFIELD PAVING	0
2201	220	G3010 WATER SUPPLY	0
2202	220	G3020 SANITARY SEWER	0
2203	220	G3030 STORM SEWER	0
2204	220	G3040 HEATING DISTRIBUTION	0
2205	220	G3050 COOLING DISTRIBUTION	0
2206	220	G3060 FUEL DISTRIBUTION	0
2209	220	G3090 OTHER SITE MECHANICAL UTILITIES	0
2211	221	G4010 ELECTRICAL DISTRIBUTION	0
2212	221	G4020 SITE LIGHTING	0
2213	221	G4030 SITE COMMUNICATION AND SECURITY	0
2219	221	G4090 OTHER SITE ELECTRICAL UTILITIES	0
2221	222	G9010 SERVICE AND PEDESTRIAN TUNNELS	0

COMP_ID (SYS_COMP_COMP_LINK)	COMP_ SYS_LINK	COMP_DESC	COMP_IS_ EQUIP
2222	222	G9090 OTHER SITE CONSTRUCTION	0
2231	223	H1010 SUBSTRUCTURE	0
2232	223	H1020 SUPERSTRUCTURE	0
2233	223	H1030 DECK	0
2234	223	H1040 MOORING AND BERTHING SYSTEM	0
2235	223	H1050 REPAIR AND REHABILITATION	0
2236	223	H1060 APPURTENANCES	0
2251	225	H3010 WAVE PROTECTION	0
2252	225	H3020 SLOPE PROTECTION	0
2261	226	H4010 DREDGING	0
2262	226	H4020 DREDGING DISPOSAL	0
2263	226	H4030 RECLAMATION	0
2271	227	H5010 CIVIL/MECHANICAL UTILITIES	0
2272	227	H5020 ELECTRICAL UTILITIES	0
2273	227	H5030 FIRE PROTECTION AND SUPPRESSION	0
2281	228	H6010 IN OR OVER-WATER DEMOLITION	0
2282	228	H6020 NON IN OR OVER-WATER DEMOLITION	0
2283	228	H6030 HAZARDOUS COMPONENTS ABATEMENT	0
2291	229	H7010 WATERSIDE ATFP	0
2292	229	H7020 LANDSIDE ATFP	0

SAMPLE USE CASES

This chapter provides some sample use cases that can be studied and/or copied and modified:

- Get inventory tree data down to the Facility level
- Add a Facility to the inventory tree
- Add inventory to a Facility
- Update inventory (change quantity of a Section)
- Add a Direct Rating inspection

Use Case 1: Get Inventory

This use case shows how to get assets in the inventory tree down to the Facility level. It is intended to show how to follow one branch down the inventory tree, selecting at each hierarchical level of the tree the next desired inventory item to investigate. It is not intended to cover the entire tree.

The functions in the sample code display `name` and/or `number` (inventory items Facility or above must have at least one or the other, and may have both), plus the `ID` property at each inventory level. As desired, you can substitute different properties to be displayed.

> **Note:** To get a performance metric such as CI, FI, or PI, you will need to use the service call "Get Performance Metrics" on page 94. An exception is at the Facility level, where you can access `CI`, `FI`, `PI`, and/or `FCI` as a property of the Facility.

How to Use the Code Example

To use the code example provided below,

- Run the code one function at a time.
- Substitute actual `ID`s (which are GUIDs) for <ID of parent>. For more information about parent IDs at different inventory levels, see "In Depth: Parent ID Properties at All Inventory Levels" on page 29.
- The function `getOrgs()` may need to be used multiple times, depending on how many Organization levels there are in your inventory tree.

Code Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;
using BuilderService;

namespace web_service_test
{
    class UseCase01
    {
        BuilderService.BuilderClient client;
        DTOOrganization[] rootOrgs;
        DTOSite[] rootSites;
        public UseCase01()
        {
            var service = new BuilderClient(); // no parameter needed unless multiple bindings
            service.ClientCredentials.UserName.UserName = "<BUILDER user name>";
            service.ClientCredentials.UserName.Password = "<BUILDER password>";
            // line below is not needed if using endpoint that was preset during configuration
            service.Endpoint.Address = new EndpointAddress("<URL of web service>");
            client = service;

        }

        // Identify the root org or site by its having null ParentID
        // skip to COMPLEX section if root is a site (i.e., no data in rootOrgs)
        public void setRoots()
        {
            rootOrgs = client.GetOrganizationsByParentID(null).Data;
            rootSites = client.GetSitesByParentID(null).Data;
        }

// ORGANIZATION

        public void getOrgsInRoot()  // list orgs under root org; if none, skip to SITE section

        {
            foreach(DTOOrganization org in rootOrgs)
            {
                var orgs = client.GetOrganizationsByParentID(ORG_ID).Data;

                foreach (var o in orgs)
                {
                    Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
                    Console.WriteLine("    " + o.ID + "\n");
                }
            }
        }

        public void getOrgs()  // list orgs under selected org. Repeat as needed for sub-orgs
        {
            var orgs = client.GetOrganizationsByParentID(new Guid("<ID of parent>")).Data;

            foreach(var o in orgs)
```

```
            {
                Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
                Console.WriteLine("    " + o.ID + "\n");
            }
        }

// SITE

        // Option A: special case of a site directly under root org
        public void getSitesInRoot()
        {
            foreach(DTOOrganization org in rootOrgs)
            {
                var sites = client.GetSitesByParentID(ORG.ID).Data;

                foreach (var o in sites)
                {
                    Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
                    Console.WriteLine("    " + o.ID + "\n");
                }
            }
        }

        // Option B: list sites under selected org (not root org)
        public void getSites()
        {
            var sites = client.GetSitesByParentID(new Guid("<ID of parent>")).Data;

            foreach(var o in sites)
            {
                Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
                Console.WriteLine("    " + o.ID + "\n");
            }
        }

// COMPLEX

       // Option A: special case where root is a Site
        public void getComplexesInRootSite()
        {
            foreach(DTOSite site in rootSites)
            {
                var cpxs = client.GetComplexesByParentID(site.ID).Data;

                foreach (var o in cpxs)
                {
                    Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
                    Console.WriteLine("    " + o.ID + "\n");
                }
            }
        }

        // Option B: list complexes under selected site (not root site)
        // Note that one complex will be named "Unassigned"
        public void getComplexes()
        {
            var cpxs = client.GetComplexesByParentID(new Guid("<ID of parent>")).Data;

            foreach(var o in cpxs)
```

```
            {
                Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
                Console.WriteLine("    " + o.ID + "\n");
            }
        }

// FACILITY

        public void getFacilities()  // list facilities under selected complex
        // If desired facility appears directly under a site in the BUILDER web interface,
        // use the ID of complex named "Unassigned"
        {
            var facs = client.GetFacilitiesByParentID(new Guid("<ID of parent complex>"), 0, 500).Data;

            foreach(var o in facs)
            {
               Console.WriteLine("Num: " + o.Number + "    Name: " + o.Name);
            }
        }
    }
}
```

Use Case 2: Add a Facility to the Inventory Tree

This use case steps through the code needed to add a Facility to the inventory tree. It will also explain the different types of properties that belong to the DTOFacility class.

> **Note:** You need BUILDER user permission level of Inspector Supervisor or above to create a Facility.

How to Use the Code Example

To use the code example provided below, substitute an actual `ID` property (which is a GUID) for each instance of <guid> or sample GUID.

Code Example

```
using System;
using System.Net;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;
using BuilderService;

namespace web_service_test
{
    class UseCase02Fac
    {
        BuilderService.BuilderClient client;
        DTOOrganization rootOrg;
        DTOSite rootSite;
        public UseCase02Fac()
        {
            var service = new BuilderClient(); //no parameter needed unless multiple bindings
            service.ClientCredentials.UserName.UserName = "<user name>";
            service.ClientCredentials.UserName.Password = "<password>";
            // line below is not needed if using endpoint that was preset during configuration
            service.Endpoint.Address = new EndpointAddress("<URL of web service>");
            client = service;
        }

        public void createFac()
        {
            var theFac = new DTOFacility();

            // set required properties
            theFac.Name = "Meadow Lane"; //name *OR* number required; ok to have both
            theFac.Number = "0005";
            theFac.ComplexID = new Guid("4604ef49-5a11-47de-8d25-e03c28f8c3fe");
            theFac.YearConstructed = 2016;
            theFac.StatusID = 1;         // GetBuildingStatuses call returns options
            theFac.UseTypeID = 23;       // use lookup; see useTypeLookup() below
            theFac.ConstructionTypeID = 2; // use lookup; see constructionTypeLookup()
            theFac.NumberFloors = 1;
            theFac.Quantity = 7000;      // this is square feet of area if UM = English;
                                         // square meters if UM = metric
```

```
            // end of required properties

            // optional properties
            theFac.BuildingTypeID = 3;   // use lookup; see FacBuildingTypeLookup()

            // output to console
            Console.WriteLine(client.CreateFacility(theFac).Data.ToString());
            Console.Read();
        }

        public void useTypeLookup()  //display output of GetFacilityUseTypes lookup
        {
            foreach (var item in client.GetFacilityUseTypes(0, 30).Data)
                Console.WriteLine(item.Description + ": " + item.ID.ToString());
            Console.Read();
        }

        public void constructionTypeLookup() //display output of GetFacilityConstructionTypes lookup
        {
            foreach(var item in client.GetFacilityConstructionTypes(0,30).Data)
                Console.WriteLine(item.Description + ": " + item.ID.ToString());
            Console.Read();
        }

        public void FacBuildingTypeLookup()  //display output of GetFacilityBuildingTypes lookup
        {
            foreach(var item in client.GetFacilityBuildingTypes(0, 30).Data)
                Console.WriteLine(item.Description + ": " + item.ID.ToString());
            Console.Read();
        }

    }
}
```

Use Case 3: Add Inventory to a Facility

This use case steps through the code needed to add inventory to a Facility, including determining the CMCID (Component Material Category ID).

How to Use the Code Example

To use the code example provided below,

- Know either the GUID (`ID` property) or the `AlternateID` property for the Facility you wish to add inventory to. Do one of the following:
  a. Substitute the GUID for <guid> in the function `dispGetFac()`, or
  b. Substitute the `AlternateID` for `Tiger` in `dispGetFacAlt()`.
- For each instance of <guid> or sample GUID, substitute an actual `ID` (which is a GUID); more detail at next bullet point.
- Run the functions `clubPlumbing()`, `fixtures()`, and `Wsinks()` one at a time, filling in the GUID returned from creating a higher level inventory item as the parent ID for the inventory item the next level down. The parent ID property may be `FacilityID`, `SystemID`, or `ComponentID`, depending on the hiearchical level of the inventory item being created.

Code Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;
using BuilderService;

namespace web_service_test
{
    class UseCase03Fill
    {
        BuilderService.BuilderClient client;
        public UseCase03Fill()
        {
            var service = new BuilderClient(); // no parameter needed unless multiple bindings
            service.ClientCredentials.UserName.UserName = "<user name>";
            service.ClientCredentials.UserName.Password = "<password>";
            // line below is not needed if using endpoint that was preset during configuration
            service.Endpoint.Address = new EndpointAddress("<URL of web service>");
            client = service;

        }
```

```
// Get Facility, Option A: get by GUID
public void dispGetFac()  // get the Facility you want to add inventory to
{
    var fac = client.GetFacility(new Guid("5d6262c0-4a91-44ec-9af5-e2e5afb3b55b")).Data;

    Console.Write("Name:" + fac.Name + "\nNumber" + fac.Number);
}

// Get Facility, Option B: get by AlternateID
public void dispGetFacAlt()  // alternatively, get Facility by alternate ID
{
    var fac = client.GetFacilityByAlternateID("Tiger").Data;

    foreach(DTOFacility item in fac)  // Alternate ID is potentially not unique
    {
        Console.WriteLine("CI:" + item.CI.ToString());
    }
}


public void clubPlumbing()  // create a System in the Facility
{
    var sys = new DTOSystem();

    // identify the parent Facility
    sys.FacilityID = new Guid("e7b69a3f-8c91-4364-bac4-87311bb2da0d");

    // to get SystemTypeID, use function getSysTypeExp() below
    sys.SystemTypeID = 210;
    sys.Name = "clubhouse plumbing";
    Console.WriteLine(client.CreateSystem(sys).Data.ToString());
}


public void fixtures()  // Create a Component in the System created above
{
    var comp = new DTOComponent();

    // identify the parent System
    comp.SystemID = new Guid("fcfca311-542f-460b-942f-a8ef826f0cac");

    // To get ComponentTypeID, use function getCompTypeExp() below
    comp.ComponentTypeID = 2101;

    Console.WriteLine(client.CreateComponent(comp).Data.ToString());
}


// Create a Section in Component in the Component created above
public void Wsinks()
{
    var sec = new DTOSection();

    // identify the parent Component
    sec.ComponentID = new Guid("90a18f4f-0753-4bc6-bd9f-b446fb9a6082");

    sec.Quantity = 2;
```

```
            sec.YearBuilt = 2017;

            sec.IsPainted = false;

            // To get CMCID (Component Material Category ID), use getSecTypeExp() below
            sec.CMCID = 2165;
            Console.WriteLine(client.CreateSection(sec).Data.ToString());

        }

    // Functions below output System/Component/Section types

        // Use function getSysTypeExp() to show System types
        // Then select number 210 for plumbing system
               public void getSysTypeExp()
        {
            foreach(var item in client.GetSystemTypes().Data)
            {
                Console.WriteLine(item.ID.ToString() + ": " + item.Description);

            }
        }

        // show Component types available for a plumbing system (plumbing=#210)
        // Then select Component #2101, plumbing fixtures
        public void getCompTypeExp()
        {
            foreach(var item in client.GetComponentTypes(210).Data)
            {
                Console.WriteLine(item.ID.ToString() + ": " + item.Description);
            }
               }

        // show Section types available for Component Type #2101 (plumbing fixtures)
        // Then select Section Type #2165, the CMCID for sinks
        public void getSecTypeExp()
        {
            foreach (var item in client.GetComponentMaterialCategories(2101).Data)
                Console.WriteLine(item.ID + ": " + item.Description);
        }

    }
}
```

Use Case 4: Update Inventory Data

This use case shows updating a Facility. The key function is provided both with extensive comments ("Verbose") and without.

How to Use the Code Example

To use the code example provided below, substitute the `ID` property of the inventory item you are updating where <guid> appears in the example.

Code Example

```
using System;
using System.Net;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;
using BuilderService;

namespace web_service_test
{
    class UseCase04Update
    {
        BuilderService.BuilderClient client;

        public UseCase04Update()
        {
            var service = new BuilderClient( ); //no parameter needed unless multiple bindings
            service.ClientCredentials.UserName.UserName = "<username>";
            service.ClientCredentials.UserName.Password = "<password>";
            // line below is not needed if using endpoint that was preset during configuration
            service.Endpoint.Address = new EndpointAddress("<URL of web service>");
            client = service;
        }

        public void updateFacilityVerbose() // use this function OR updateFacility() below
        {
            // step 1a. create new instance of dto object
            var theFacility = new DTOFacility();

            // step 1b. set it equal to returned value from get function
            theFacility = client.GetFacility(new Guid("<guid>")).Data;

            //step 2.
            // (a) assign new values to properties that you want to change,
            //     AND/OR
            // (b) add one or more new properties.
            //The statement format is the same either way

            // (a) property change, for example the Facility has been added on to, is now larger:
            theFacility.Quantity = 4500;

            // (b) add a new property, for example an alternate ID:
            theFacility.AlternateID = "01467"
```

```
            // step 3. call the update function with the new dto object as the parameter
            client.UpdateFacility(theFacility);
        }

           // this is the same function, just without the comments
        public void updateFacility()
        {
            var theFacility = new DTOFacility();
            theFacility = client.GetFacility(new Guid("<guid>")).Data; //sub real ID for <guid>

            theFacility.Quantity = 4500;
            theFacility.AlternateID = "01467"

            client.UpdateFacility(theFacility);
        }

    }
}
```

Use Case 5: Create a Direct Rating Inspection

This use case creates some inventory to be inspected, and shows creating a direct rating inspection.

How to Use the Code Example

To use the code example provided below,

- Substitute an actual `ID` (which is a GUID) for each placefiller <guid> in the use case.
- Start by running the function `clubPlumbing()`, using as `sys.FacilityID` the guid of a Facility that does not already contain a Plumbing system. This will run `clubPlumbing()`, `fixtures()`, `Lsinks()`, and `Wsinks()` to create inventory to be inspected.
- Next, run `createInsp()`, `getRollupInit()`, and `getWsinksCI()`.
- Optionally, this can be followed by running `updateSinkInspection()`, `getRollupInit()`, and `getWsinksCI()` to see the effect on the CI of the inspected section. The code example changes the sample condition rating to 95, but this value can be set as desired.
- See Direct Rating Inspections: Color vs. Condition Rating for the sample condition rating settings equivalent to each direct rating color selection (Green, Amber, Red).

Code Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;
using BuilderService;

namespace web_service_test
{
    class UseCase05Direct
    {
        BuilderService.BuilderClient client;
        DTOOrganization rootOrg;
        DTOSite rootSite;


        public UseCase05Direct()
        {
            var service = new BuilderClient(); //no parameter needed unless multiple bindings
            service.ClientCredentials.UserName.UserName = "<username>";
            service.ClientCredentials.UserName.Password = "<password>";
```

```
        // line below is not needed if using endpoint that was preset during configuration
        service.Endpoint.Address = new EndpointAddress("<URL of web service>");
        client = service;
    }


    public void createInsp()
    {
        var insp = new DTOInspection();
        insp.SectionID = new Guid("<guid>"); // enter guid of Wsinks
        insp.Date = DateTime.Now;
        insp.IsSampling = false;
        insp.Type = 2;  // type=Direct Rating, from InspectionType enum; var type=short
        insp.Source = "Inspection";  // source=Inspection, from InspectionSource enum;
                                     // var type=string

        // enter information of samples.
        // If isSampling is false, there is just one sample that comprises the whole section

        var samp = new DTOSample();
        samp.IsPaint = false;

        //when isSampling = false, sample qty should equal section qty
        samp.Quantity = 3;
        samp.ConditionRating = 61;  // enter a condition rating here (Amber-)

        //make array for sample properties
        DTOSample[] smpArray = { samp };

        insp.Samples = smpArray;
        Guid theGuid = client.CreateInspection(insp).Data; //store GUID of created inspection
        Console.WriteLine(theGuid.ToString());
        Console.WriteLine(client.GetInspection(theGuid).Data.CI.ToString());
        Console.WriteLine(client.GetSection(new Guid("<guid of Wsinks>")).Data.CI.ToString());
        Console.Read();
    }

    public void updateSinkInspection()
    {
        // In the line below, enter guid of inspection to be updated
        var insp = client.GetInspection(new Guid("<guid>")).Data;

        //insp.Samples[0].ConditionRating = 95; // change condition rating to Green
        Console.WriteLine(client.UpdateInspection(insp).Data.ToString());
        Console.Read();
    }


    public void getRollupInit()
    {
        // use InitiateFacilityRollup to roll up, and get the guid for the rollup.
        // In the line below, enter guid of the facility to be rolled up
        Guid rollupGuid = client.InitiateFacilityRollup(new Guid("<guid>")).Data;

        // use guid for the rollup to check status of the rollup
        Console.WriteLine(client.GetRollupStatus(rollupGuid).Data);
        Console.Read();
    }
```

```
public void getWsinksCI()
{
    // enter guid of Wsinks section in the line below
    foreach(var item in client.GetPerformanceRecords(new Guid("<guid>"),
       PerformanceRecordType.CI, 2017).Data)
    {
        Console.WriteLine("Value: " + item.Value.ToString());
        Console.WriteLine("Type: " + item.Type.ToString());
    }
    Console.Read();
}


//create inventory to inspected; just provide Facility ID

// this function will create System AND incorporate lower levels of inventory
public void clubPlumbing()
{
    var sys = new DTOSystem();

    // on the next line, enter guid of a facility with no Plumbing system entered in BUILDER
    sys.FacilityID = new Guid("<guid>");

    sys.SystemTypeID = 210;
    sys.Name = "clubhouse plumbing";
    fixtures(client.CreateSystem(sys).Data);
    //Console.WriteLine(toReturn.ToString());
}

public void fixtures(Guid parentID)
{
    var comp = new DTOComponent();

    comp.SystemID = parentID;

    comp.ComponentTypeID = 2101;
    Guid compGuid = client.CreateComponent(comp).Data;
    Lsink(compGuid);
    Wsinks(compGuid);
    Console.WriteLine(client.CreateComponent(comp).Data.ToString());
}

public void Lsink(Guid ParentID ) // Laundry sink
{
    var sec = new DTOSection();

    sec.ComponentID = ParentID
    sec.Quantity = 1;
    sec.YearBuilt = 2015;
    sec.IsPainted = false;
    sec.Name = "Laundry sink";
    sec.CMCID = 21476;

    Console.WriteLine(client.CreateSection(sec).Data.ToString());
}

public void Wsinks(Guid ParentID )  // Women's room sinks
{
    var sec = new DTOSection();
```

```
            sec.ComponentID = ParentID;
            sec.Quantity = 3;
            sec.YearBuilt = 2017;
            sec.IsPainted = false;
            sec.Name = "Ladies room sinks";
            sec.CMCID = 41166;

            Console.WriteLine(client.CreateSection(sec).Data.ToString());
        }

    }
}
```

Index