Revit API Intro Labs

Lab 6 – Extensible Storage

Created by A. Nagy, May 2011
Updated by DevTech AEC WG
Last modified: 11/3/2023

<C#>C# Version**</C#>**

Objective: In this lab, we will learn how to use Extensible Storage mechanism to add custom data to a Revit element. During the course of building a command, we'll also cover Selection Filter and Transaction. We'll learn how to:

- Define a storage data "Schema" and attach an instance of it to an Revit element
- Use an ISelectionFilter to limit the selection to Walls
- Use manual Transaction mode

Tasks: We'll write a command that creates WallSocketLocation Schema with two Fields, SocketLocation and SocketNumber, and attach its Entity (or instance) to a wall.

1. Ask the user to pick a wall
2. Create our Schema with the two Fields
3. Create an Entity of the schema and set its field values
4. Assign the entity to the wall using SetEntity()

The following is the breakdown of step by step instructions in this lab:

1. Define a New External Command with Manual Transaction Mode
2. Create a Selection Filter
3. Create and Attach a Schema Entity to a Wall
4. Summary

1. Define a New External Command with Manual Transaction Mode

We'll add another external command to the current project.

1.1 Add a new file and define another external command to your project. Let's name them as follows:

- File name: **6_ExtensibleStorage.vb (or .cs)**
- Command class name: **ExtensibleStorage**

Required Namespaces:

In addition to the name spaces you have used, add the name space:

- Autodesk.Revit.DB.ExtensibleStorage

Below is the starting point of our new command.

<C#>
```
//  Model Creation - learn how to create elements

[Transaction(TransactionMode.Manual)]
public class ExtensibleStorage : IExternalCommand
{
    public Result Execute(
        ExternalCommandData commandData,
        ref string message,
        ElementSet elements)
    {
        //  Get the access to the top most objects.
        UIDocument uiDoc = commandData.Application.ActiveUIDocument;
        Document doc = uiDoc.Document;

        Transaction trans = new Transaction(doc, "Extensible Storage");
        trans.Start();

        //  ...

        trans.Commit();

        return Result.Succeeded;
    }
}
```
</C#>

2. Create a Selection Filter

Though this topic will be part of the UI Labs and that's where its components will be explained, we'll use PickObject() and ISelectionFilter in this lab as well. We simply need to create a class that implements the ISelectionFilter interface and its two functions: AllowElement() and AllowReference(). Let's create it inside the ExtensibleStorage class:

<C#>
```
class WallSelectionFilter : ISelectionFilter
{
  public bool AllowElement( Element e )
  {
    return e is Wall;
  }

  public bool AllowReference( Reference r, XYZ p )
  {
    return true;
  }
}
</C#>
```

Now we'll be able to use an instance of this class when asking the user to pick an element in the user interface to restrict the selection to only walls.

Let's ask the user to pick a wall, to which we want to add data, and then get back the wall from the returned Reference:

```
<C#>
        // Pick a wall

        Reference r = uiDoc.Selection.PickObject(
            ObjectType.Element, new WallSelectionFilter());

        Wall wall = doc.GetElement(r) as Wall;
</C#>
```

3. Create a Schema and Attach an Entity to a Wall

To create a Schema, we use the SchemaBuilder object, which requires a GUID that will identify the schema later on. Let's declare this GUID in our ExtensibleStorage class:

```
<C#>
Guid _guid = new Guid("87aaad89-6f1b-45e1-9397-2985e1560a02");
</C#>
```

Note: you could generate and use another GUID as well, using Visual Studio's 'Tools >> Generate GUID' function.

Let's start building our schema by creating a new SchemaBuilder instance and set its properties. We'll set the Read and Write access levels to public. If we set it to Vendor or Application, then we would also need to specify the vendor id for the schema using SetVendorId(). The vendor id is the Registered Developer Symbol (RDS) that can be created on this site:
http://www.autodesk.com/symbreg
Since vendor ids are not case sensitive, the string will be converted to upper case before it is stored in the schema.

```
<C#>
      // Create a schema builder

      SchemaBuilder builder = new SchemaBuilder(_guid);

      // Set read and write access levels
      builder.SetReadAccessLevel(AccessLevel.Public);
      builder.SetWriteAccessLevel(AccessLevel.Public);

      // Note: if this was set as vendor or application access,
      // we would have been additionally required to use SetVendorId

      // Set name to this schema builder

      builder.SetSchemaName("WallSocketLocation");
      builder.SetDocumentation("Data store for socket info in a wall");
</C#>
```

Add two fields to the schema. One will contain a location property and will be of type XYZ, and the other one will contain the id number of our socket, and will be of type string.
Once these are added we can call Finish() to create the Schema object.

```
<C#>
      // Create field1

      FieldBuilder fieldBuilder1 =
        builder.AddSimpleField("SocketLocation", typeof(XYZ));

      // Set unit type

      fieldBuilder1.SetSpec(SpecTypeId.Length);

      // Add documentation (optional)

      // Create field2

      FieldBuilder fieldBuilder2 =
        builder.AddSimpleField("SocketNumber", typeof(string));

      //fieldBuilder2.SetUnitType(UnitType.UT_Custom);

      // Register the schema object

      Schema schema = builder.Finish();
</C#>
```

Now we can create two Entities based on our Schema and assign them to the selected wall.

```
<C#>
      // Create an entity (object) for this schema (class)

      Entity ent = new Entity(schema);

      Field socketLocation = schema.GetField("SocketLocation");
      ent.Set<XYZ>(socketLocation, new XYZ(2, 0, 0),
          UnitTypeId.METERS);

      Field socketNumber = schema.GetField("SocketNumber");
      ent.Set<string>(socketNumber, "200");

      wall.SetEntity(ent);

      // Now create another entity (object) for this schema (class)
      // (This simply replaces the ent1 above. Just for testing.
      //  You may comment out for now.)

      Entity ent2 = new Entity(schema);
      Field socketNumber1 = schema.GetField("SocketNumber");
      ent2.Set<String>(socketNumber1, "400");
      wall.SetEntity(ent2);
```
</C#>

We could also list all the available Schema's and the Fields available in our schema just to see if everything went the way we wanted it.

<C#>
```
      // List all schemas in the document

      string s = string.Empty;
      IList<Schema> schemas = Schema.ListSchemas();
      foreach( Schema sch in schemas )
      {
        s += "\r\nSchema Name: " + sch.SchemaName;
      }
      TaskDialog.Show( "Schema details", s );

      // List all Fields for our schema

      s = string.Empty;
      Schema ourSchema = Schema.Lookup( _guid );
      IList<Field> fields = ourSchema.ListFields();
      foreach( Field fld in fields )
      {
        s += "\r\nField Name: " + fld.FieldName;
      }
      TaskDialog.Show( "Field details", s );
```
</C#>

Now let's check if the Entity's we assigned to the selected wall can be accessed all right.

<C#>
```
      // Extract the value for the field we created

      Entity wallSchemaEnt = wall.GetEntity( Schema.Lookup( _guid ) );

      XYZ wallSocketPos = wallSchemaEnt.Get<XYZ>(
        Schema.Lookup( _guid ).GetField( "SocketLocation" ),
        UnitTypeId.METERS );

      s = "SocketLocation: " + PointToString( wallSocketPos );
      string wallSocketNumber = wallSchemaEnt.Get<String>(
        Schema.Lookup( _guid ).GetField( "SocketNumber" ) );

      s += "\r\nSocketNumber: " + wallSocketNumber;

      TaskDialog.Show( "Field values", s );
</C#>
```

Where PointToString() is a simple helper function to convert a point as a string for display. You may reuse the one you have written in Lab2, e.g.:

```
<C#>
        // Helper Function: returns XYZ in a string form.
        //
        public static string PointToString(XYZ pt)
        {
            if (pt == null)
            {
                return "";
            }

            return string.Format("({0},{1},{2})",
                pt.X.ToString("F2"), pt.Y.ToString("F2"), pt.Z.ToString("F2"));
        }

</C#>
```

4. Summary

In this lab, we learned how to define and attach a custom data to a Revit element, using Extensible Storage mechanism in the API. We have learned how to:

- Create and access extensible storage entities of Revit elements.