AUTODESK UNIVERSITY

PD21274-R

Inventor.io: Building Web Applications with Inventor on Forge Platform

Andy Akenson
Autodesk

Learning Objectives

- Learn how Inventor.io fits into the Forge Platform
- Discover and understand Inventor.io
- Learn about building apps with Inventor.io
- Get takeaways on how you can use these building blocks to automate your workflows

Description

We'll demonstrate some example apps in the Forge Platform using Inventor.IO software and we'll use this to create an interactive discussion on what kind of things you'd like to build on the Forge Platform using Inventor.IO. This session features Forge, Design Automation API and Inventor Professional.

Your AU Expert

Andy is a Software Architect at Autodesk. He has spent the last 9 years working on the Inventor product in areas such as User Interface, Visualization/Materials and most recently connecting Inventor to Autodesk's cloud strategy. He brings 20 years of software engineering experience integrating multiple technologies into complex systems as well as programming experience in JavaScript, C#, C++, Java and various scripting languages.

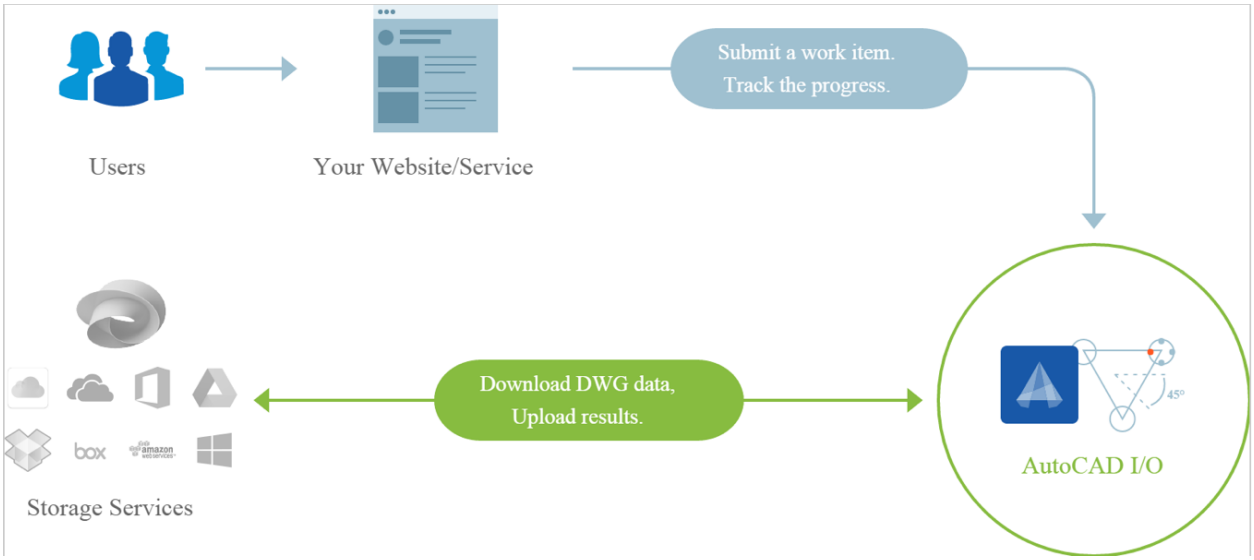
andrew.akenson@autodesk.com

How Inventor fits into the Forge Platform

With Forge Autodesk is creating a set of web service APIs that power the future of making things. Forge takes components from Autodesk's powerful library of software and delivers them as cloud-based building blocks for companies to create their own new solutions.

Design Automation API

The Design Automation API was first introduced into Forge providing AutoCAD scripts in the cloud.



AUTOCAD DESIGN AUTOMATION API FLOW

Design Automation API Terminology

The Inventor addition to the Design Automation API works with all of the same basic components that already exist today

Activity	an action that can be executed within the engine
AppPackage	A module referenced by an Activity in order to perform specific functions
module file	An AppPackage entity
WorkItem	a job that is submitted to and executed by the AutoCAD core engine *Note: Once a WorkItem is created, it cannot be modified.*
Engine	the actual processing engine that runs the WorkItem job and processes the Activity

Adding Inventor into the Design Automation API

We've kept the API for the Design Automation API intact and added an Inventor "engine". This allows the same API for both AutoCAD and Inventor while providing domain specific work done in the AppPackage.

Design Automation API Inventor Overview.

While the basic components of the API are exactly the same for AutoCAD and Inventor the actual contents and usage may vary. For Inventor this looks something like:

Activity	For example, this could be changing parameter values on the model
AppPackage	Read-only Inventor module file For example, this might be an Inventor plug-in that changes parameters values in a Part
module file	Inventor plug-in, read-only Inventor files, spreadsheets, templates, iLogic rules
WorkItem	For example, define the actual values for the parameter change and get the result file
Engine	Currently supporting Inventor 2017

Building Apps with Design Automation API and Inventor

The Design Automation API is exactly the same for Inventor and AutoCAD but for the moment they are accessed with a different endpoint:

AutoCAD: `https://developer.api.autodesk.com/autocad.io/us-east/v2/`

Inventor: `https://developer.api.autodesk.com/inventor.io/us-east/v2/`

All of the examples show here are leveraging Microsoft's OData in C# for the client app but the Design Automation API Documentation contains all of the information so it can be built in any technology.

AppPackages and Inventor plug-ins

An AppPackage can contain an Inventor plug-in. This is a sub-set of the API that is available from Inventor add-ins. It's basically an Inventor add-in without any UI. The AppPackage contents will need to be setup as follows:

```
\---samplePlugin.bundle
   |   PackageContents.xml
   |
   \---Contents
         samplePlugin.dll
         samplePlugin.Inventor.addin
```

PackageContents.xml

The PackageContents.xml is the manifest for providing meta-data for the AppPackage

```
<?xml version="1.0" encoding="utf-8" ?>
<ApplicationPackage SchedmaVersion="1.0" Version="1.0" ProductCode="{FAE1868B-18EE-4090-A676-535335C4D41A}"
Name="samplePlugin" Description="sample Plugin" Author="Inventor IO Dev">
    <CompanyDetails Name="Autodesk, Inc" Phone="415.555.5555" Url="www.autodesk.com"
Email="noreply@autodesk.com" />
    <Components>
        <!-- For Inventor Engine, "Platform" attribute must be "Inventor" -->
        <RuntimeRequirements  SeriesMax="R21.17" SeriesMin="R21.17" OS="Win64" Platform="Inventor" />
        <!-- For Inventor Plug-in, the "Module" attribute must point to the .addin manifest file. -->
        <ComponentEntry LoadOnAutoCADStartup="False" LoadOnCommandInvocation="False"
            AppDescription="samplePlugin App Package."
            ModuleName="./Contents/samplePlugin.Inventor.addin" AppName="samplePlugin"/>
    </Components>
    <EnvironmentVariables>
    </EnvironmentVariables>
</ApplicationPackage>
```

PACKAGECONTENTS.XML

Most of the content in the PackageContent.xml is boiler plate and came from the AutoCAD implementation. The two import parts that you'll need to change are the Platform and ModuleName elements as highlighted above.

Contents Directory

The Contents directory contains the Inventor plug-in dll as well as the typical Inventor .addin file with the Type set to “Plugin”. The name of the dll and addin file must match what is specified in the ModuleName elmenet from PackageContents.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Addin Type="Plugin">
  <ClassId>{FAE1868B-18EE-4090-A676-535335C4D41A}</ClassId>
  <ClientId>{FAE1868B-18EE-4090-A676-535335C4D41A}</ClientId>
  <DisplayName>sample plugin</DisplayName>
  <Description>a sample plugin</Description>
  <Assembly>samplePlugin.dll</Assembly>
</Addin>
```

SAMPLEPLUGIN.INVENTOR.ADDIN

Inventor plug-in

The Inventor plug-in can be written in any Inventor add-in supported language (.NET or unmanaged C++). For these examples I’ll use C#. The entry point into the plug-in will be one of the two methods **`Run`** or **`RunWithArguments`**.

```
public void Run(Document doc)

public void RunWithArguments(Document doc, NameValueMap map)
```

The Document object is defined as part of the WorkItem and is opened and passed into one of these two methods. If CommandLineArgs are specified in the activity these are passed in as the **map**. The map values are stored in the key values “_1”, “_2” and so on for each command line argument.

Any other inputs specified in either the AppPackage or WorkItem can be loaded from the current working directory and any file saved is available as an output in the Result of a WorkItem.

The following example shows how to get the parameters to change and output file from the client, change the parameters and save out the file as the specified output file. The trace statements are logged and the log file is obtained from the WorkItem.

```
public void RunWithArguments(Document doc, NameValueMap map)
{
    StringBuilder traceInfo = new StringBuilder("RunWithArguments called with ");
    traceInfo.Append(doc.DisplayName);
    Trace.TraceInformation(map.Count.ToString());

    // values in map are keyed on _1, _2, etc
    for (int i = 0; i < map.Count; i++)
    {
        traceInfo.Append(" and ");
        traceInfo.Append(map.Value["_" + (i + 1)]);
    }

    Trace.TraceInformation(traceInfo.ToString());

    Trace.TraceInformation("Modifying param _1");
    // First param "_1" should be the filename of the JSON file containing the parameters and values
    string paramFile = map.Value["_1"];
    Trace.TraceInformation("paramFile: " + paramFile);
    string json = System.IO.File.ReadAllText(paramFile);
    Trace.TraceInformation("changeParameters file: \"" + json + "\"");

    PartDocument partDoc = (PartDocument)doc;
    ChangeParameters(json, partDoc);

    // Second param "_2" should be the output filename
    string outputFile = map.Value["_2"];

    string dirPath = System.IO.Path.GetDirectoryName(doc.FullDocumentName);
    Trace.TraceInformation("writing file: " + dirPath + "\\" + outputFile);
    doc.SaveAs(dirPath + "\\" + outputFile, false);
    Trace.TraceInformation("output saved");
}

public void ChangeParameters(string json, PartDocument doc)
{
    PartComponentDefinition partDef = doc.ComponentDefinition;

    Dictionary<string, string> parameters =
        JsonConvert.DeserializeObject<Dictionary<string, string>>(json);
    foreach (KeyValuePair<string, string> entry in parameters)
    {
        // entry.key = param name
        // entry.value = param value string
        Trace.TraceInformation("param to change: {0}:{1}", entry.Key, entry.Value);
        Parameter param1 = partDef.Parameters[entry.Key];
        param1.Expression = entry.Value;

     }
     doc.Update();
     Trace.TraceInformation("doc updated.");
}
```

SAMPLEPLUGIN C# CODE

Besides the Document object available in the run method, the constructor of the plug-in also has a reference to the InventorServer object to provide full access to the InventorServer API which is basically the Inventor API without any User Interface.

```
Inventor.InventorServer m_inventorServer;

public SampleAutomation(Inventor.InventorServer inventorServer)
{
        m_inventorServer = inventorServer;
}
```

Design Automation Client

The Design Automation API is typically setup as follows:

1. Authenticate with your Forge Client Key and Secret and get a token back. It will have the form “Bearer <token>”:

```
string AuthUrl = "https://developer-dev.api.autodesk.com/authentication/v1/authenticate";
string ConsumerKey = “<your forge key>”;
string ConsumerSecret = “<your forge secret>”;

using (var client = new HttpClient())
{
    var values = new List<KeyValuePair<string, string>>();
    values.Add(new KeyValuePair<string, string>("client_id", Credentials.ConsumerKey));
    values.Add(new KeyValuePair<string, string>("client_secret", Credentials.ConsumerSecret));
    values.Add(new KeyValuePair<string, string>("grant_type", "client_credentials"));
    values.Add(new KeyValuePair<string, string>("scope", "code:all"));
    var requestContent = new FormUrlEncodedContent(values);
    var response = client.PostAsync(AuthUrl, requestContent).Result;
    var responseContent = response.Content.ReadAsStringAsync().Result;
    var resValues = JsonConvert.DeserializeObject<Dictionary<string, string>>(responseContent);
    return resValues["token_type"] + " " + resValues["access_token"];
}
```

2. Create an AppPackage object and upload an AppPackage. This is typically just done one time or when when you need to update an existing AppPackage. Once uploaded it behaves like a re-usable library. The UploadObject method uploads the zip file containing an AppPackage structure as shown above to any cloud storage that is accessible to the Engine.

```
// Container is base OAuth Container object
builder.Path += "AppPackages/Operations.GetUploadUrl";
var url = container.Execute<string>(builder.Uri, "GET", true, null /*new
BodyOperationParameter("FileName","foo.zip")*/).First();

var response = UploadObject(url, AppProperties.LocalAppPackage);
if (response.StatusCode != HttpStatusCode.OK)
    return null;

var package = new AppPackage();
package.Version = 1;
package.RequiredEngineVersion = “21.17”; // Inventor version
package.Id = “samplePlugin”;
package.Resource = url;
container.AddToAppPackages(package);

Console.Write("Submitting AppPackage...");
container.SaveChanges();
```

3. Next define an Activity that provided the interface for the WorkItem into the AppPackage. This is typically also done once.

```
Activity activity = new Activity();

// Setup the required activity properties
activity.Id = “samplePlugin”;
activity.AppPackages.Add(AppProperties.AppPackageId);
activity.RequiredEngineVersion = “21.17”;
activity.Instruction = new Instruction()
{
        Script = "",
       CommandLineParameters = AppProperties.ParamFile + " " + AppProperties.OutputFile
};

// Setup the input and output parameters for the activity
activity.Parameters = new Parameters();
Parameter inputReqParam = new Parameter()
{
        Name = AppProperties.”HostDwg”, // Required by framework for now
       LocalFileName = “SamplePart.ipt”
};
activity.Parameters.InputParameters.Add(inputReqParam);

Parameter inputChangeParam = new Parameter()
{
       Name = "ChangeParameters",
       LocalFileName = “changeParams.json”
};
activity.Parameters.InputParameters.Add(inputChangeParam);

Parameter outputParam = new Parameter()
{
        Name = “Result”,
        LocalFileName = “SampleBoxOutput.ipt”
};
activity.Parameters.OutputParameters.Add(outputParam);

// Created a new activity
container.AddToActivities(activity);
```

4. While the AppPackage and the Activity are usually setup once the WorkItem is created to actually run the Activity and gather the results and is not persisted. Here's how to create a simple WorkItem to change parameters via a JSON object

```
Activity activity = …
var wi = new WorkItem()
{
     Id = "", //must be set to empty
     Arguments = new Arguments(),
     ActivityId = activity.Id /*AppProperties.ActivityId*/
};

// The DesignAutomation API current requires this InputArgument.
wi.Arguments.InputArguments.Add(new Argument()
{
    Name = "HostDwg",// Must match the input parameter in activity
    Resource = <CloudStorage InputFile>,
    StorageProvider = StorageProvider.Generic //Generic HTTP download (as opposed to A360)
});

// This shows passing parameters and values into the plug-in
wi.Arguments.InputArguments.Add(new Argument()
{
    Name = "ChangeParameters",
    Resource = "data:application/json,{\"d2\":\"0.5 in\", \"d3\":\"0.2 in\"}",
    StorageProvider = StorageProvider.Generic,
    ResourceKind = ResourceKind.Embedded
});

wi.Arguments.OutputArguments.Add(new Argument()
{
    Name = AppProperties.OutputArgName, //must match the output parameter in activity
    StorageProvider = StorageProvider.Generic, //Generic HTTP upload (as opposed to A360)
    HttpVerb = HttpVerbType.POST, //use HTTP POST when delivering result
});

container.AddToWorkItems(wi);
```

5. Now that the WorkItem is created submit it by calling SaveChanges on the Container object and poll for the results

```
Console.WriteLine("Submitting workitem...");
container.SaveChanges();

//polling loop - wait for result
do
{
    Console.WriteLine("Sleeping for 2 sec...");
    System.Threading.Thread.Sleep(2000);
    container.LoadProperty(wi, "Status"); //http request is made here
    Console.WriteLine("WorkItem status: {0}", wi.Status);
}
while (wi.Status == ExecutionStatus.Pending || wi.Status == ExecutionStatus.InProgress);
```

6. Once the results are ready the report and any resulting files are accessible from the WorkItem and urls can be obtained for the files.

```
//re-query the service so that we can look at the details provided by the service
container.MergeOption = Microsoft.OData.Client.MergeOption.OverwriteChanges;
wi = container.WorkItems.ByKey(wi.Id).GetValue();

// get the url for the status report (output from Engine and plug-in)
var reportUrl = wi.StatusDetails.Report;

// Resource property of the output argument "Result" will have the output url
var resultUrl = wi.Arguments.OutputArguments.First(a => a.Name == "Result").Resource;
```

Design Automation Building Blocks

During the Roundtable we'll have an interactive conversation around what workflows the Design Automation API can enable. We're providing the blocks but you get the create new and innovative solutions. Here are a few problems statements we'll use to seed the conversation and brainstorming:

- Automation of variations of an existing product
- Transforming a large number of files