

SD473689

Tips & Tricks: what I learnt while supporting Design Automation for Inventor

Adam Nagy
Autodesk

Learning Objectives

- Explain what Design Automation is for, and show some example workflows
- Know how to estimate service costs
- Optimize your Design Automation based services
- Start automation processes multiple ways

Description

Forge is a set of web services provided by Autodesk. Design Automation is one of its components that added support for Inventor about one year ago. In this class I will provide an introduction to the Design Automation API and how it can be used to automate Inventor processes. I'll show how you can estimate and optimize the costs of using this service, and the various ways you can run and speed up processes on the Design Automation service. I will also cover some of the specific tips I learned while helping customers with specific workflows. For example how to get modelling error details from Inventor, how to obtain a list of supported files types, and other useful ideas. I will demonstrate the coding ideas using the .NET Core development environment.

Speaker(s)

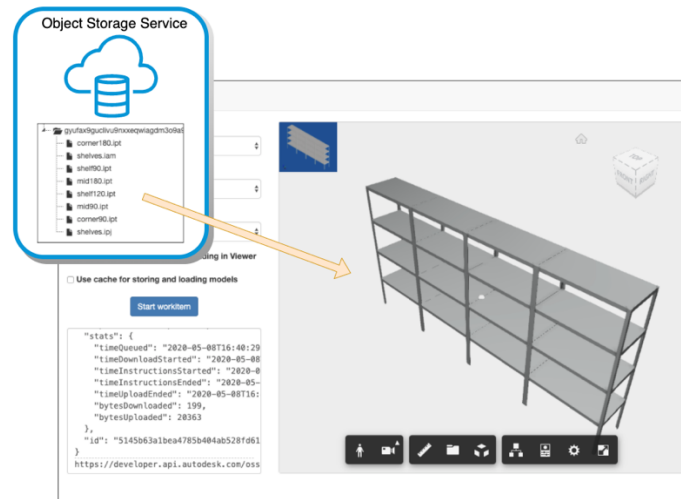
Adam Nagy joined Autodesk back in 2005, and he has been providing programming support, consulting, training, and evangelism to external developers. He started his career in Budapest working for a civil engineering CAD software company. He then worked for Autodesk in Prague for 3 years, and he now lives in South England, United Kingdom. Adam focuses on supporting Forge and the API's of our manufacturing products, Inventor and Fusion 360.
Twitter @AdamTheNagy

What is Design Automation?

Design Automation enables you to run a headless version of our desktop products, like **Inventor**, directly on our servers to help you automate certain tasks.

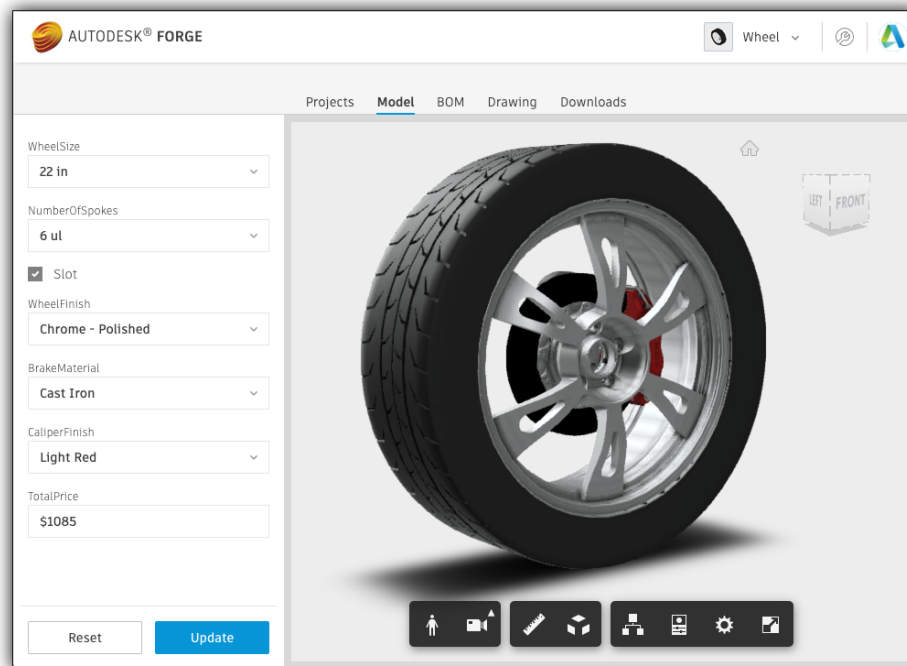
Configurators

Currently, most users are creating different kinds of configurators using this technology – e.g. things like the [Shelf Configurator](#) ([source code](#)) shown below.



SHELF CONFIGURATOR

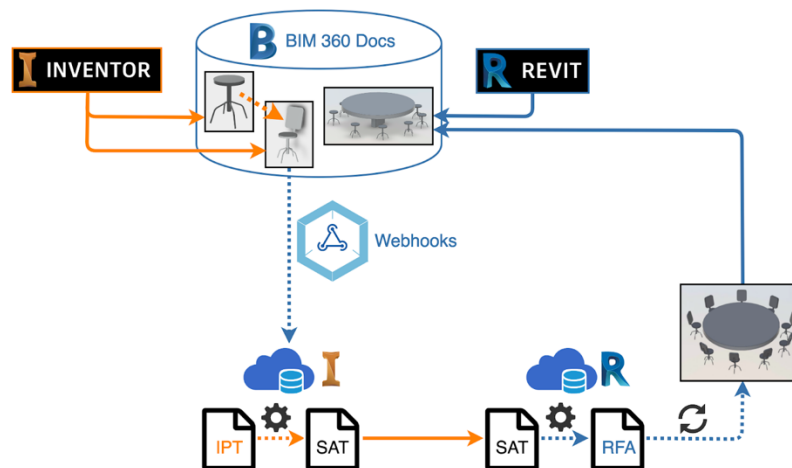
In order to help current **Configurator 360** users create their own similar solutions based on **Design Automation API for Inventor**, our engineering team created this [extensive sample](#) ([source code](#)) with fully public source code that anyone can take and use. It even has user management and model caching implemented.



SAMPLE FOR CONFIGURATOR 360 USERS

Automation

Configurators are not the only use case for this technology. You could also use it for example to streamline workflows. Perhaps migrating other file formats to **Inventor** documents (see <https://design-migration.azurewebsites.net/>), or use it directly from inside **Vault** to validate or update files (see https://github.com/sajith-subramanian/Inventor_Design_Automation_with_Vault), or something more complex that would take advantage of other **Forge** services as well (see <https://github.com/Autodesk-Forge/forge-update-revitfamily-from-inventorpart>).

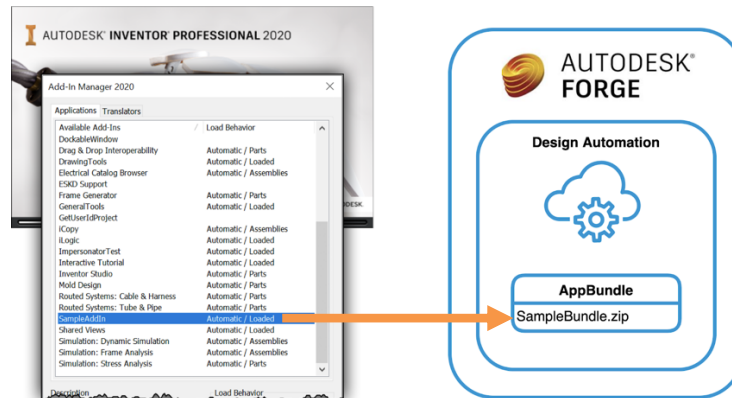


SAMPLE SHOWCASING DESIGN AUTOMATION (INVENTOR AND REVIT) AND WEBHOOKS

The above shown sample app monitors files on **BIM 360 Docs** and as a new version of an **Inventor** model gets uploaded, it gets notified through the **Forge webhooks** system, and can

automatically update the **Revit** family from the **Inventor** model and can also update the **Revit** project which is using that **Revit** family.

How does it work?



ADD-IN ON THE DESKTOP VS APPBUNDLE ON THE CLOUD

On the desktop you would create **add-ins** to customise the functionality of **Inventor** and in case of **Design Automation** you can use the same **Inventor API** that you used on the desktop and can create so called **App Bundles** to achieve the same customization.

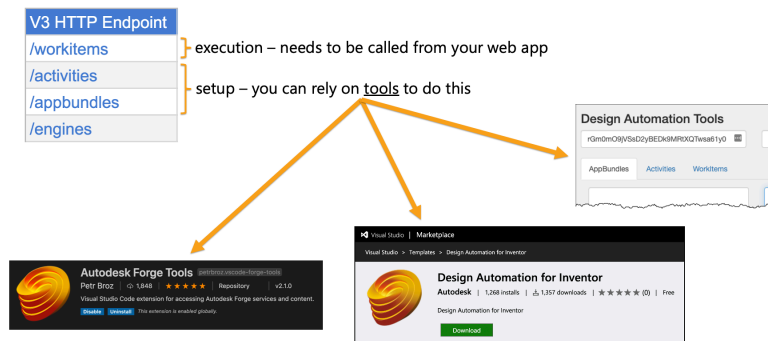
<https://forge.autodesk.com/blog/migrate-inventor-add-design-automation>

V3 HTTP Endpoint	Programming concept	Product concept
/workitems	Function call	Product execution, session
/activities	Function definition	Script file command line parameters
/appbundles	Shared library	Plugin
/engines	Instruction set	Product (Revit, AutoCAD, Inventor, etc) to use

The **/appbundles** endpoint of the **Design Automation API** will enable you to upload your code that will be using the **Inventor API** and work with the input documents you provide in order to produce the resulting documents or other files you need.

The **/activities** endpoint will enable you to create so called activities that will specify which **App Bundle** you want to work with, what kind of input you'll provide for the **App Bundle** and what outputs it should produce.

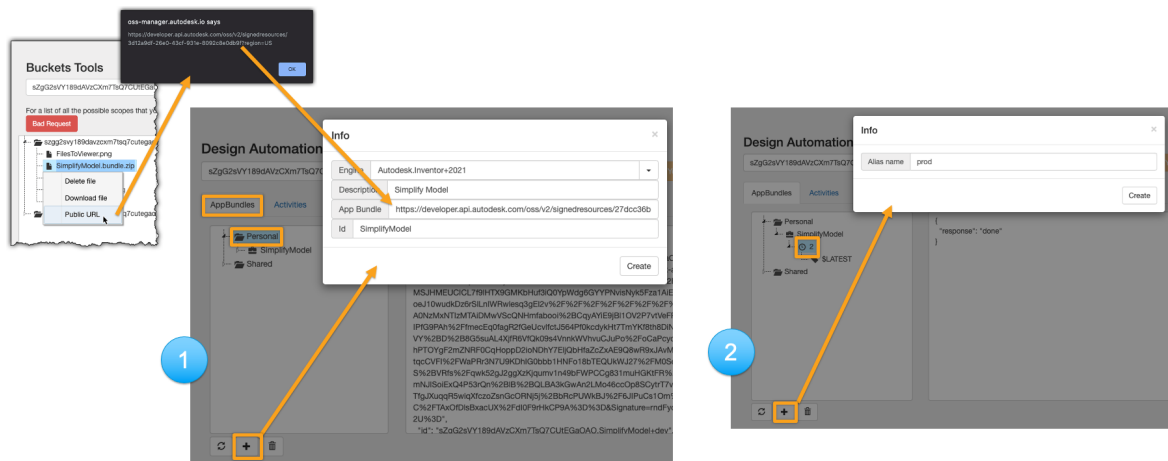
And the **/workitems** endpoint allows you to create jobs based on the specification of an activity.



MAIN DESIGN AUTOMATION API ENDPOINTS

I consider the creation of the **App Bundle** and **Activity** part of the **setup**, and **Work Item** creation is the **execution**.

You don't have to write any code to do the setup, you can rely on tools like the **VS Code** extension, the **Design Automation Tools** website or the **Interaction** project that is part of the solution created using the **Visual Studio** template

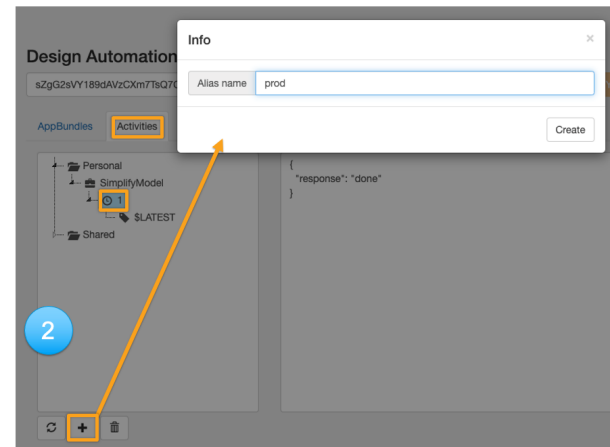
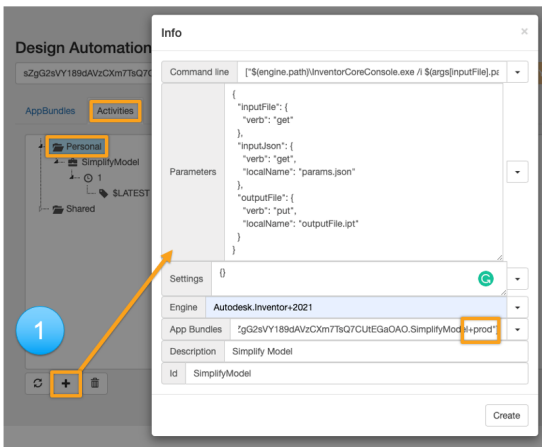


APP BUNDLE AND ALIAS CREATION

When creating the **App Bundle**, you have to upload the **zip** file with your code in it to some place where **Design Automation** can access it.

That's what we are doing in this example as well, we are using the **Buckets Tools** web app to upload the **zip** file into an **OSS** bucket and then generate a **signed URL** for it that we can use in the body of our request.

When creating an **App Bundle** or **Activity** you also have to create an **Alias** for the uploaded version in order to use it. In this case we are creating an alias called "**prod**" that we are going to use when creating the activity.



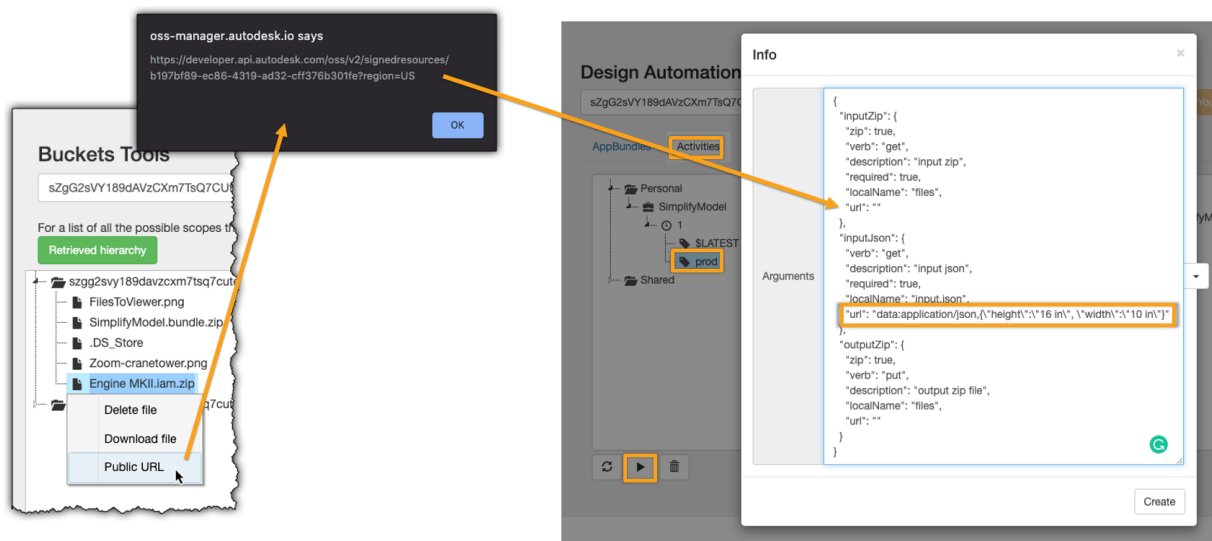
ACTIVITY AND ALIAS CREATION

When creating the **Activity** we'll have to provide the **Command Line** which will specify how exactly **Inventor Server** will be run, what parameters will be passed to it. Most **activities** tend to use the **/al** and **/i** command line arguments that will tell **Inventor Server** which **App Bundle** to load and which document to open.

You also need to provide the **input** and **output parameters** (or files) that the **activity** will need – in our case we'll have an **inputFile**, an **inputJson** and an **outputFile**.

When specifying which **App Bundle** the activity should be using you can see that we are providing the alias to point out the version of the **App Bundle** we want to use – which in this case is called "**prod**".

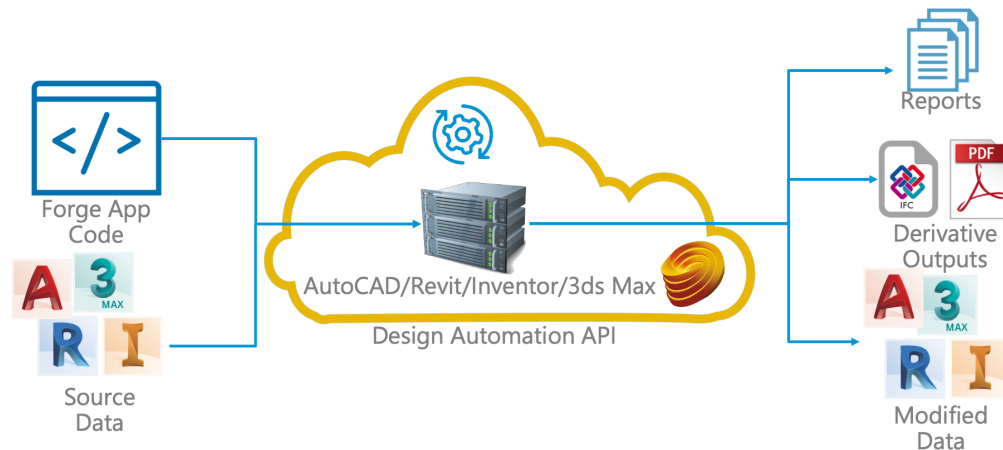
After creating the **Activity** we need to assign an **Alias** to this new version so that we will be able to reference it when starting a **Work Item**



WORK ITEM CREATION

In order to start a **Work Item** we need to specify the **Activity** we want to use and the **URLs** for all the **input** and **output** files.

Here again we can use the **Buckets Tools** to upload those files to a bucket on **OSS** (Object Storage Service) and create a **pre-signed URL** for them that we can pass to the **Work Item**. As you can see, in case of having **json** as input we can also pass its content directly as part of the **URL** instead of uploading such a file to **OSS** and then create a **pre-signed URL** for it that we can then pass to the **Work Item**. So it's really easy to work with them.



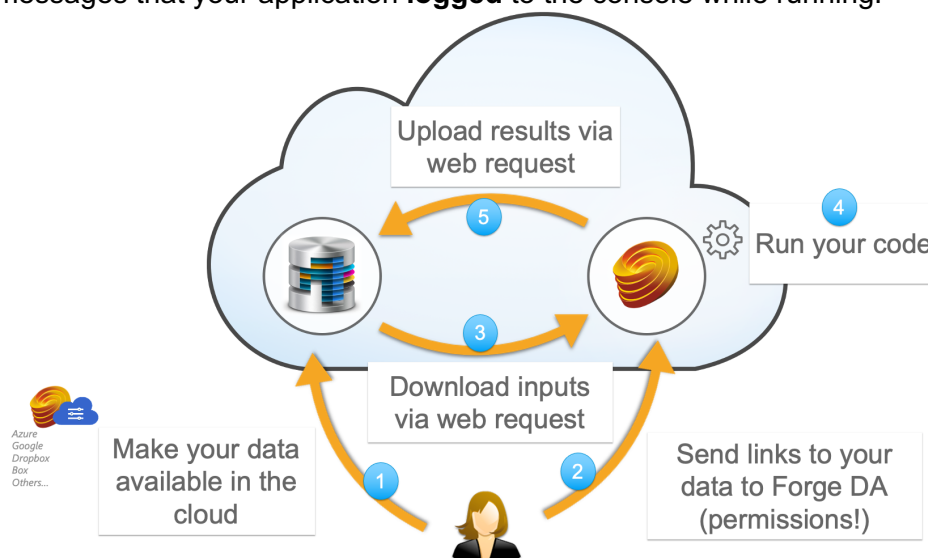
USING DESIGN AUTOMATION

Here is an overview of the steps I previously showed.

You need to provide your **App Bundle** for **Design Automation** along with the **input** files that you want to work with.

The code in your **App Bundle** will be able to **create** new documents, **modify** existing ones or **export** to various other file formats.

Each time your code is run on **Design Automation** a **report** file will be produced that will also include all messages that your application **logged** to the console while running.



RUNNING A WORK ITEM

And here are the steps of running your **Work Item**.

First you need to make the **input files** available for **Design Automation**. You can upload them to any online file storage including of course **Forge** storage services as well like **OSS** – you could use e.g. the **Buckets Tools** website for that, as I showed you previously.

Then in the second step you kick off your **Work Item** whose input parameters need to include the **URLs** to the **input** and **output** files as well

As the third step, **Design Automation** will download the input files

Then in the 4th step it will run the code of your **App Bundle** to produce the results

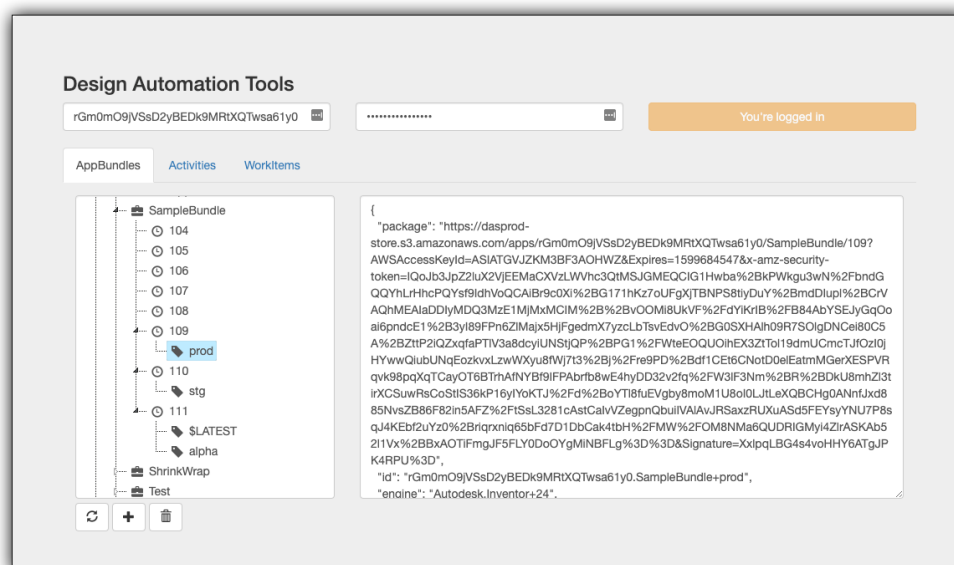
The final step then is to upload the **resulting files** to the location you previously specified

Things I learnt

Aliases

In the beginning I was wondering, why we need them, why we couldn't just simply upload new versions of **App Bundles** and **Activities** and reference them based on their version number. It helped me to think of them as **tags** or **labels** that I can just attach to a specific version – and that's how I decided to show them in the **Design Automation Tools** sample as well.

Since you always rely on these tags, therefore you can just freely upload new versions and delete existing ones (that have no tags) without affecting the rest of your system.



DESIGN AUTOMATION TOOLS SAMPLE

E.g. you could have a **"prod"** alias (as shown in the picture) for the **production** ready version and **"stg"** for the **staging** version that is being tested right now before being made public.

Since all the production ready systems will use the **App Bundle** and **Activity** version with the **"prod"** alias therefore once the version currently with the **"stg"** alias passed all the tests, you can just simply add the **"prod"** alias to them and all production systems will start using those versions without making any other changes.



CORECT WAY TO REFERENCE A SPECIFIC VERSION

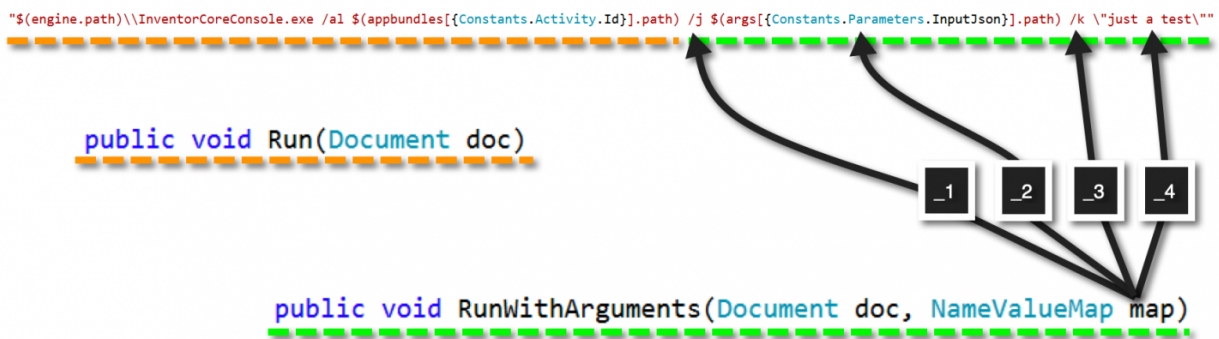
The alias named **\$LATEST** is just for internal use and you cannot delete it, move it, or use it to reference a specific version of a resource you want to use.

You also cannot use the version number to reference a specific version of an **App Bundle** or **Activity**.

You always need to use the **alias**.

Run vs RunWithArguments

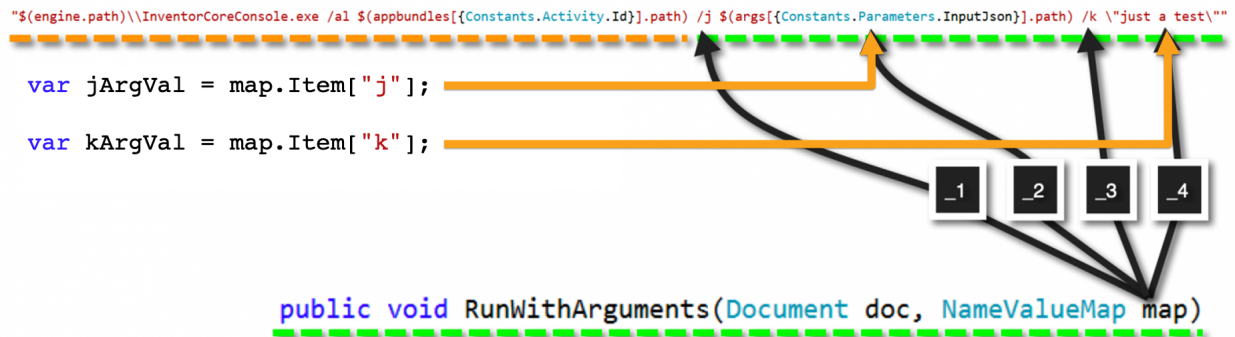
Why are there two entry points in case of **App Bundles** for **Inventor**: **Run** and **RunWithArguments**?



RUN VS RUNWITHARGUMENTS

The **commandLine** parameter of an **Activity** specifies how **Inventor** server will be called, what parameters will be passed to it. If all the arguments used are something that inventor server supports like **/al** (which specifies which appbundle to load) **/i** (which specifies which document to open) then the **Run()** entry point will be called with only the opened document passed to it. However, if you also add any custom arguments to the commandLine like **/j** or **/k** (as in the above example) then **RunWithArguments()** will be called with an additional object called "**map**" - this will contain the list of additional arguments that were passed through the command line.

The keys for them will be named with an **underscore plus index**, so **_1**, **_2**, etc, which enables you to access them directly without iterating through the list.



ACCESS COMMANDLINE ARGUMENT VALUES

Recently there were some improvements added to help you access your **commandLine** arguments. Now you can also retrieve their values based on the **name of the argument** instead of relying on its **position number**.

You can find more details on this in the **online API reference**:

<https://forge.autodesk.com/en/docs/design-automation/v3/reference/cmdLine/cmdLine-inventor/>

See <https://forge.autodesk.com/blog/run-vs-runwitharguments>

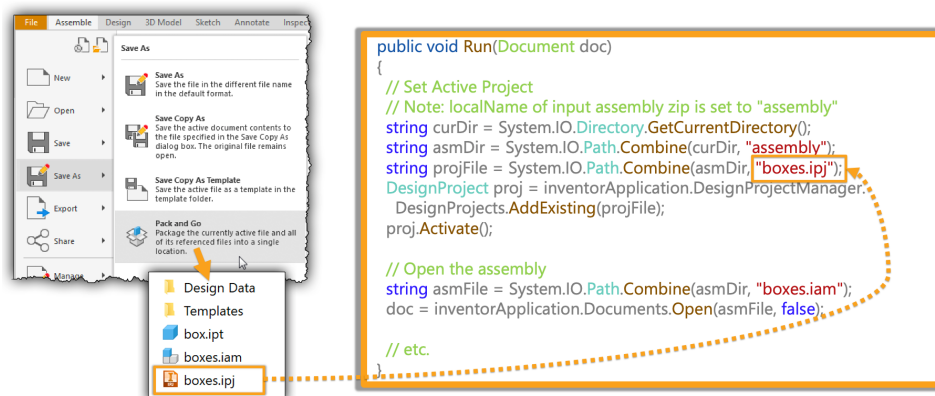
Resolve file references

Project file from /p

By default on **Design Automation** the project named **"Default"** is used (just like on the desktop), which is not in the **CurrentDirectory** of your **Work Item** and you cannot set its **Workspace** folder either. So it's not helping with file resolution at all.

Therefore, we introduced a new command line argument to help with that: **/p** for project. This will create a new **Project** in the **CurrentDirectory** of the **Work Item** and activate that before opening the document specified in the commandLine

Project file from Pack and Go



USING PROJECT FILE FROM PACK AND GO

Many people use **Pack and Go** to gather all the files needed when sharing their assemblies with others. This will generate a **project** file as well and you could activate that on **Design Automation** before opening the **assembly**. In case of our **assembly** the **Pack and Go** function generated a **project** file called “**boxes.ipj**” and that will be part of the input assembly **zip** that we can use on the **Design Automation** server.

Custom file resolution

Instead of relying on projects, you could also implement your own file resolution mechanism – if that's what you prefer.

You can do that by handling the **OnFileResolution** event which will get the file name and location where Inventor is looking for a referenced file, and lets **you** provide the correct location of that file.

```
// Make sure you keep a reference to events objects otherwise the events won't fire
FileAccessEvents fae = inventorApplication.FileAccessEvents;
fae.OnFileResolution += Fae_OnFileResolution;

// etc

private void Fae_OnFileResolution(
    string RelativeFileName,
    string LibraryName,
    ref byte[] CustomLogicalName,
    EventTimingEnum BeforeOrAfter,
    NameValueMap Context,
    out string FullFileName,
    out HandlingCodeEnum HandlingCode)
{
    // It's best practice to first say we didn't handle the event
    // and then change it later on if needed
    HandlingCode = HandlingCodeEnum.kEventNotHandled;

    // Let's say all my parts will be in this specific folder
    string partsFolder = System.IO.Path.Combine(g_bundlePath, "SampleBundlePlugin.bundle");

    // Get the file name without the path
    string fileName = System.IO.Path.GetFileName(RelativeFileName);

    // Combine it with our folder
    FullFileName = System.IO.Path.Combine(partsFolder, fileName);

    if (System.IO.File.Exists(FullFileName))
    {
        HandlingCode = HandlingCodeEnum.kEventHandled;
    }
}
```

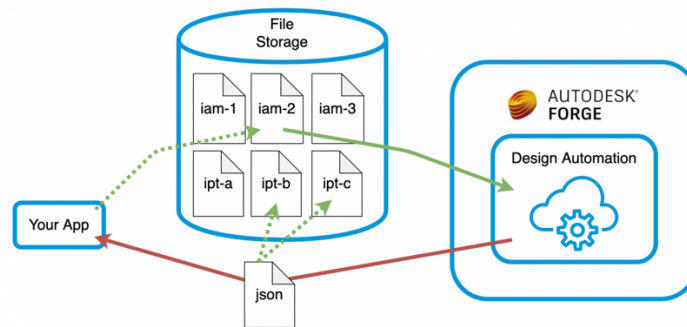
USE ONFILERESOLUTION EVENT

See <https://forge.autodesk.com/blog/resolving-referenced-inventor-files>

Find references

Another issue could be that you do not even know which files you'll need to provide for **Design Automation**, what other files the **assembly** you are trying to update requires?

This is not only true for **Design Automation**, but e.g. **Model Derivative API** as well. You cannot translate an **Inventor** file to **SVF** (which is the **Forge Viewer** format) if the referenced documents are missing.



GET LIST OF REFERENCED FILES

You can create an **App Bundle** that retrieves the **direct references** of a given **assembly** document using the **Inventor API's ReferencedFileDescriptors** property.

Once you have this information, you could provide that for the client in e.g. a **json** file.

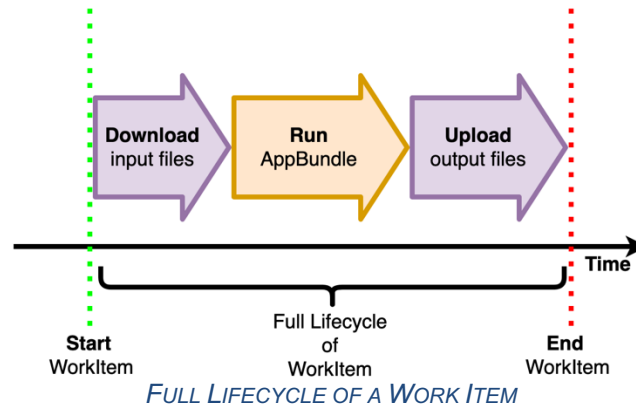
In case of a **multi level assembly** things get more complicated, since there you would have to **repeat** the process for the **subassemblies**, and their **subassemblies**, and so on.

See <https://forge.autodesk.com/blog/get-list-referenced-files>

Estimate costs

You are charged for the **full lifecycle** of a job (or **Work Item**) running on our server.

That includes the time spent **downloading** the input files to our server, the **running** time of your code inside **Inventor Server**, and then the time it takes to **upload** the resulting files to some place from our server.



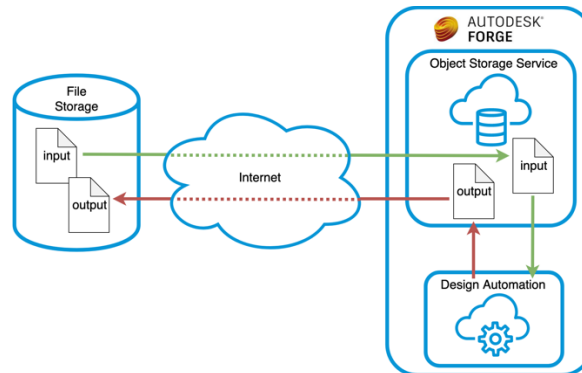
Estimate running time

The running time of your code can be estimated by doing the same automation locally on your desktop. If it takes **Inventor** 2 minutes to update the model after the necessary parameter change on your computer, then I would expect it to take a similar amount of time or even less on our servers.

Estimate download/upload time

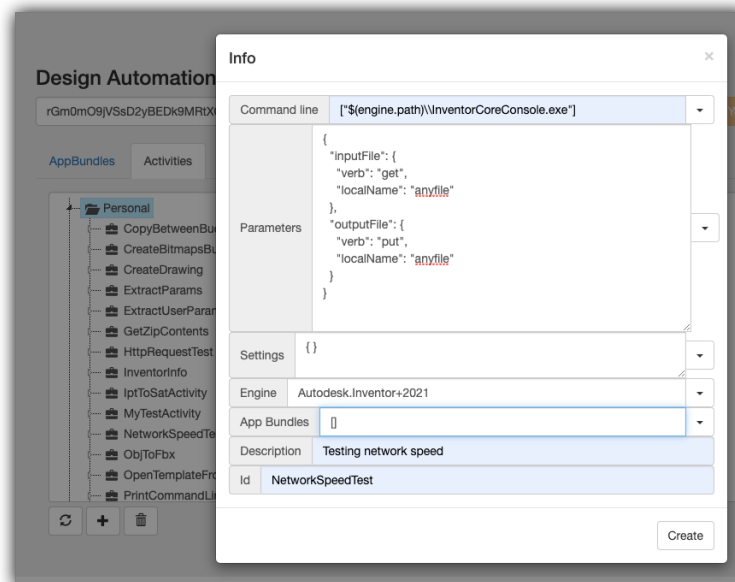
The **upload/download** time will depend on the internet speed between the server where the files are stored and our server. So obviously the fastest is if the files are on our servers already (at least temporarily, while running the **Work Item**)

However, some servers can provide the same or similar speed too.



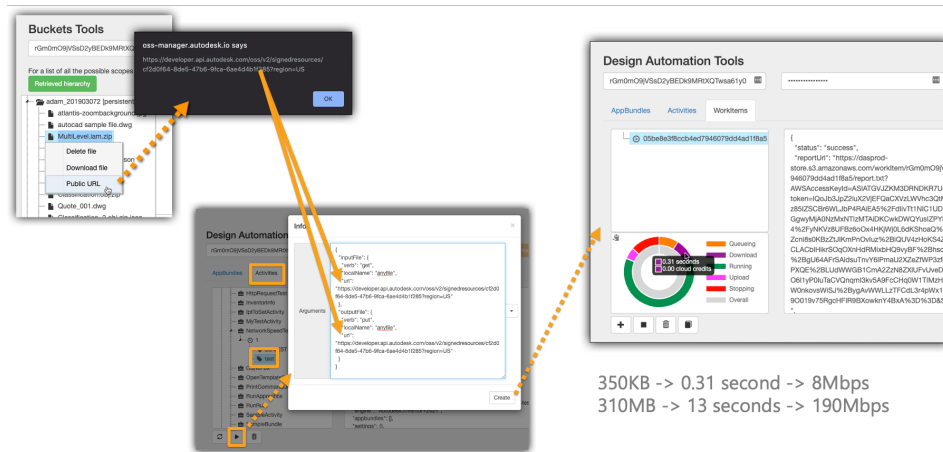
CACHING FILES IN OSS BUCKET

One simple way to test the network speed would be to use any **Activity** that requires an **input** and **output** file. Even if the job itself fails, you'll still get information at least about the time it took to **download** the files.



ACTIVITY FOR TESTING UPLOAD/DOWNLOAD SPEED

We could just create the following simple **Activity**, that does not even rely on any **App Bundles** to use for this. We can set it up using the **Design Automation Tools** sample. Just log in with your **Forge App's credentials**, and create an **Activity** with these parameters and settings.



TESTING NETWORK SPEED WITH ACTIVITY

Once it's done, you can then run a **Work Item** based on this **Activity**. You'll just need a **presigned** read/write **URL** that **Design Automation** can use to download the file and then upload it back up, thereby testing both the download and upload speed from the **Design Automation** server.

We can use the [Buckets Tools](#) to generate such **URL**'s and use them for the **Work Item**. Once the workitem finished, the **Design Automation Tools** will show a breakdown of the time spent on various activities, including **download** and **upload**. In our case you can see that the download of the zip file which is **350KB**, took a **3rd of a second**. That gives a speed of **8Mbps**

But a **310MB** file took **13 seconds**, giving us a speed of **190 Mbps**.

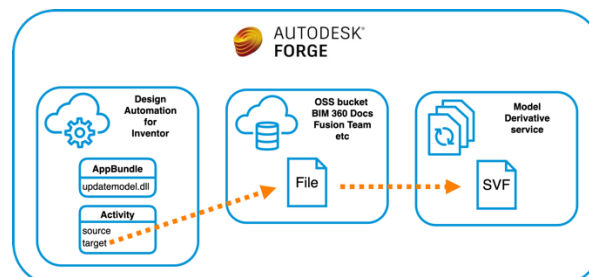
I guess the big difference in speed is caused by the **overhead** of setting up the download and upload, which in case of a small file is a much bigger portion.

See <https://forge.autodesk.com/blog/estimate-design-automation-costs> & <https://forge.autodesk.com/blog/optimize-design-automation-process>

Generate viewables

In case of **Inventor**, you can generate the viewables needed to show your model in the **Forge Viewer**, directly on the **Design Automation** server as well.

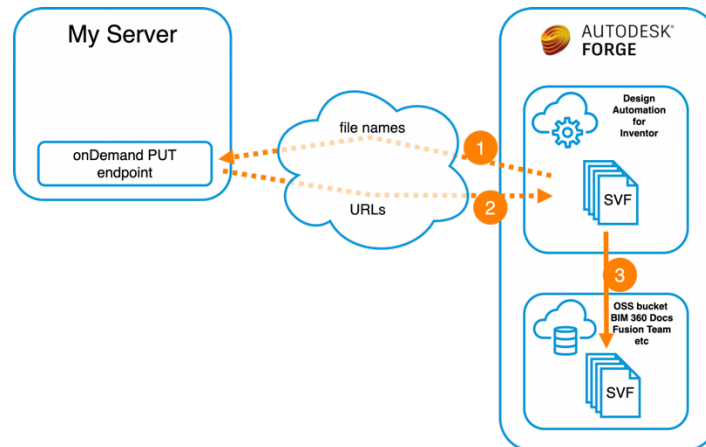
This can help you **reduce the waiting time** for your customers to see the updated 3d model.



GENERATE VIEWABLES WITH MODEL DERIVATIVE API

Usually you generate the viewables (it's called the **SVF** bubble, which is a bunch of files) using the **Model Derivative** service. This also takes care of storing those viewables for the given file and makes it really easy to load them into the **Viewer**.

If you decide to generate the **SVF** on the **Design Automation** server, then **you** will need to take care of storing those files and make them accessible for the **Viewer**. You could either store them directly on your server, or place them in some kind of storage, e.g. an **OSS** bucket.

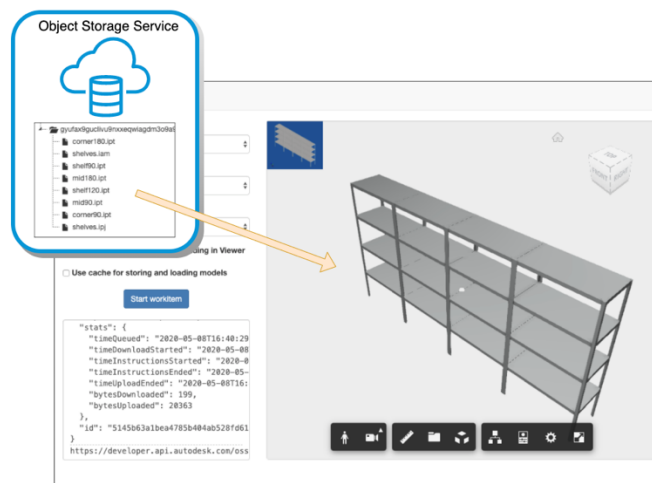


STORING THE FILES IN AN OSS BUCKET

See <https://forge.autodesk.com/blog/speed-viewable-generation-when-using-design-automation-inventor>

Skip generating viewables

In certain cases you might not even need to generate viewables for each configuration. Such scenario is if you're working with existing components with predefined dimensions, where the configuration only specifies **which** components should be used, and **where** they need to be placed inside the model. It does not allow the user to specify arbitrary dimensions for the components.



SHELF CONFIGURATOR SAMPLE

You might have various sizes of shelves, mid- and corner posts that the user can choose from. In this case you could generate the viewables for the various components in advance using the **Model Derivative** service, and only use **Design Automation** to find out which parts should be used and where they should be placed.

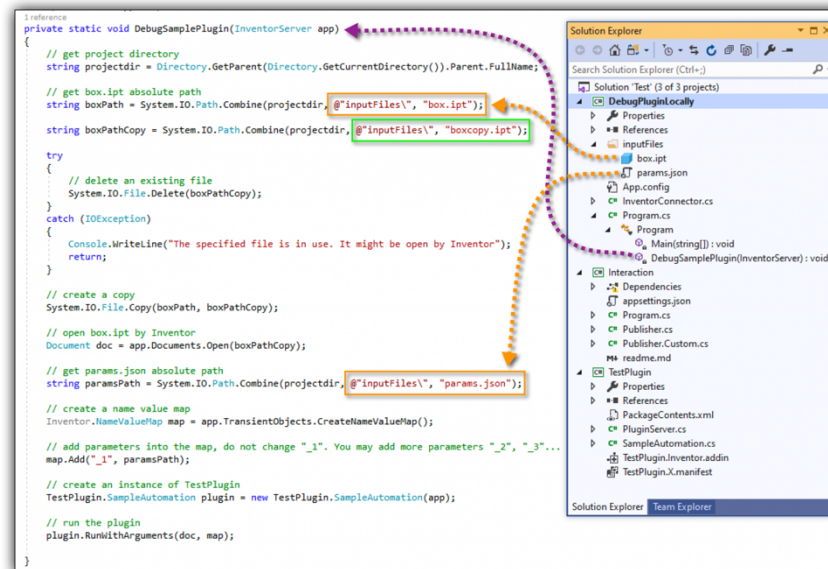
This way you can avoid waiting for the **SVF** to be generated and uploaded somewhere that the **Viewer** can access, and instead, can simply load all the components into the **Viewer** and specify their **location** and **transformation** that **Inventor** calculated.

See <https://forge.autodesk.com/blog/faster-configuration-results>

Debugging

Some things you can debug locally.

You can do that by creating **App Bundles** using the **Visual Studio** project template for **Design Automation for Inventor**.



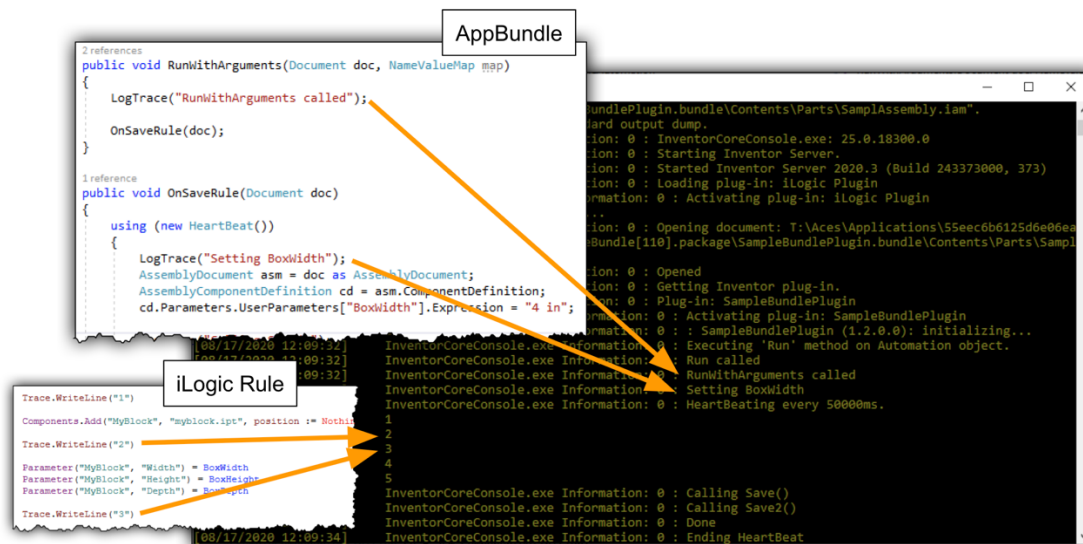
VS PROJECT CREATED USING THE DESIGN AUTOMATION TEMPLATE

This will also create a project called **DebugPluginLocally** that helps you with debugging your project on your desktop.

The code that sets things up in a way that imitates what happens on the **Design Automation** server is inside the **DebugSamplePlugin** function.

It will start up a local instance of **Inventor** and enable you to step through your code just like in case of an **Inventor add-in**.

See <https://forge.autodesk.com/blog/design-automation-inventor-vs-template-local-debug>

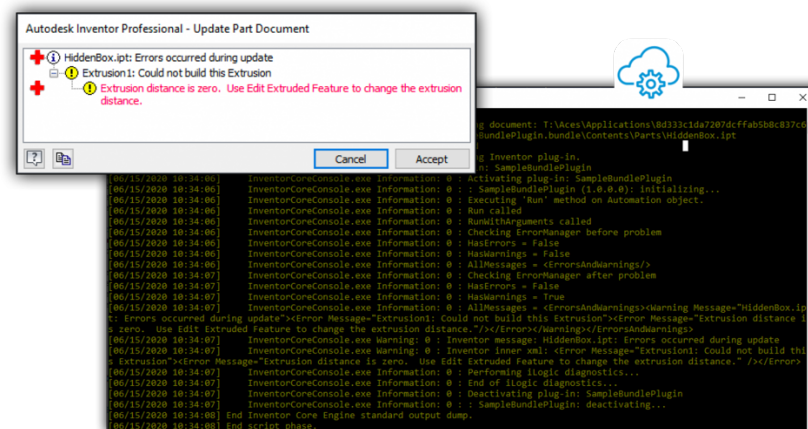


LOGGING MESSAGES FROM BOTH APP BUNDLE AND ILOGIC RULES

Since you don't have direct access to our servers, therefore you'll have to rely on **logs** to identify where things went wrong.

As shown in the picture, you can log messages to the **Work Item's** report file from both your **App Bundle** and the **iLogic Rules**

See <https://forge.autodesk.com/blog/design-automation-inventor-vs-template-run-workitem>



ERROR MANAGER

When using **Inventor** on the **desktop** there is the **error manager** window which shows you what kind of errors the model ran into when **Inventor** tried to update things in it.

You have access to this on **Design Automation** as well through the **Inventor API**.

```
public void RunWithArguments(Document doc, NameValueMap map)
{
    var em = inventorApplication.ErrorManager;
    LogTrace($"Checking ErrorManager before problem");
    LogTrace($"HasErrors = {em.HasErrors}");
    LogTrace($"HasWarnings = {em.HasWarnings}");
    LogTrace($"AllMessages = {em.AllMessages}");

    PartDocument pd = doc as PartDocument;
    pd.ComponentDefinition.Parameters["height"].Expression = "0";
    pd.Update2();

    LogTrace($"Checking ErrorManager after problem");
    LogTrace($"HasErrors = {em.HasErrors}");
    LogTrace($"HasWarnings = {em.HasWarnings}");
    LogTrace($"AllMessages = {em.AllMessages}");
}
```

```
[06/15/2020 10:34:07] InventorCoreConsole.exe Information: 0 :
Checking ErrorManager after problem [06/15/2020 10:34:07]
InventorCoreConsole.exe Information: 0 : HasErrors = False
[06/15/2020 10:34:07] InventorCoreConsole.exe Information: 0 :
HasWarnings = True [06/15/2020 10:34:07] InventorCoreConsole.exe
Information: 0 : AllMessages = <ErrorsAndWarnings> <Warning
Message="HiddenBox.ipt: Errors occurred during update"> <Error
Message="Extrusion1: Could not build this Extrusion"> <Error
Message="Extrusion distance is zero. Use Edit Extruded Feature to
change the extrusion
distance."/> </Error> </Warning> </ErrorsAndWarnings> [06/15/2020
10:34:07] InventorCoreConsole.exe Warning: 0 : Inventor message:
HiddenBox.ipt: Errors occurred during update [06/15/2020 10:34:07]
InventorCoreConsole.exe Warning: 0 : Inventor inner xml: <Error
Message="Extrusion1: Could not build this Extrusion"> <Error
Message="Extrusion distance is zero. Use Edit Extruded Feature to
change the extrusion distance." /> </Error>
```

MESSAGES FROM THE ERROR MANAGER

You can use the **ErrorManager** object to retrieve the same messages and pass it back to the user whose model you are updating using **Design Automation**.

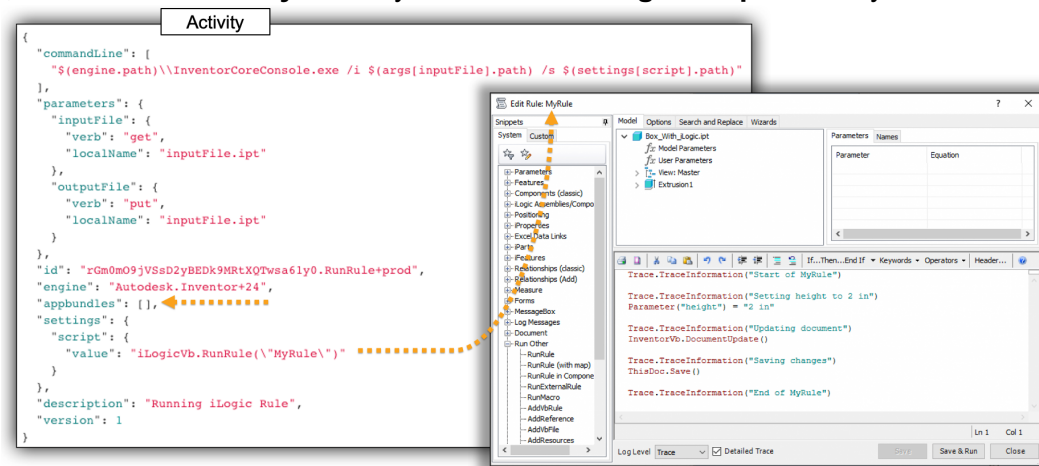
So you will not only be able to tell the user that a given modification failed, but also the reason for it.

As you can see we have access to the exact same messages that were shown on the previous slide, telling us that the parameter values we provided resulted in **Extrusion1** having a zero distance and that causes problems.

See <https://forge.autodesk.com/blog/get-modeling-error-details-inventor>

No App Bundle

If you want to do something very simple, then you might not even need to create an **App Bundle**, because the **Activity** allows you to execute **iLogic scripts** directly.



```
{
  "commandLine": {
    "$engine.path\\InventorCoreConsole.exe /i $(args[inputFile].path) /s $(settings[script].path)"
  },
  "parameters": {
    "inputFile": {
      "verb": "get",
      "localName": "inputFile.ipt"
    },
    "outputFile": {
      "verb": "put",
      "localName": "inputFile.ipt"
    }
  },
  "id": "rGm0m09jVSsD2yBEDk9MRtXQTwsa6ly0.RunRule+prod",
  "engine": "Autodesk.Inventor+24",
  "appbundles": [],
  "settings": {
    "script": {
      "value": "iLogicVb.RunRule(\"MyRule\")"
    }
  },
  "description": "Running iLogic Rule",
  "version": 1
}
```

RUNNING ILOGIC RULE DIRECTLY FROM ACTIVITY

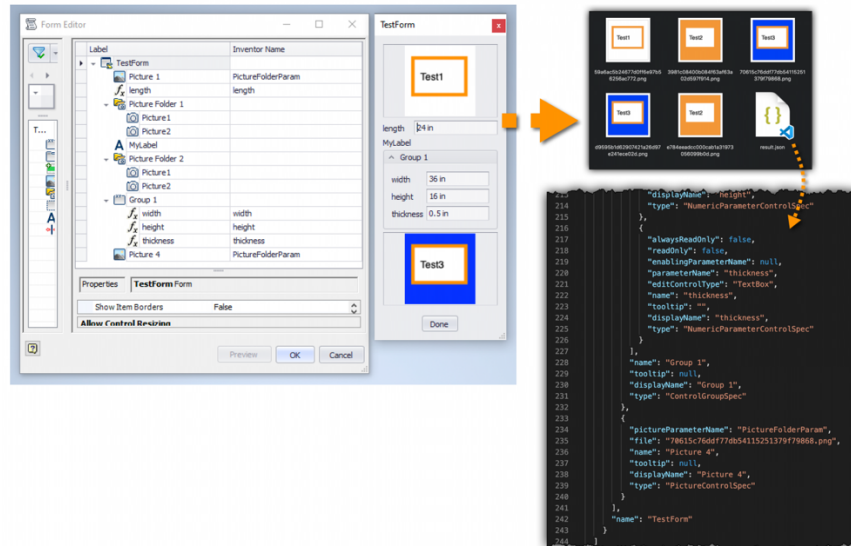
As you can see, here we have an **Activity** that is using no **App Bundles**, but instead has a **script** parameter which includes the **iLogic** code we want to run.

In this case we are simply running the **iLogic Rule** called **MyRule** that is available in the **Inventor** document that we use as input.

See <https://forge.autodesk.com/blog/run-illogic-rule-without-appbundle>

Get iLogic Form

Often Inventor documents that use **iLogic Rules**, also include one or more **iLogic Forms** that help the user identify which parameters should be modified to correctly configure the model.



INFORMATION EXTRACTED FROM ILOGIC FORMS

Unfortunately, there is no mechanism to expose these forms **directly** to the user on your website. However, using **iLogic libraries** you can extract all the information from the **Forms**: which **parameters** are listed, what **Rule** the “Done” button runs, what **pictures** are displayed on the form, etc

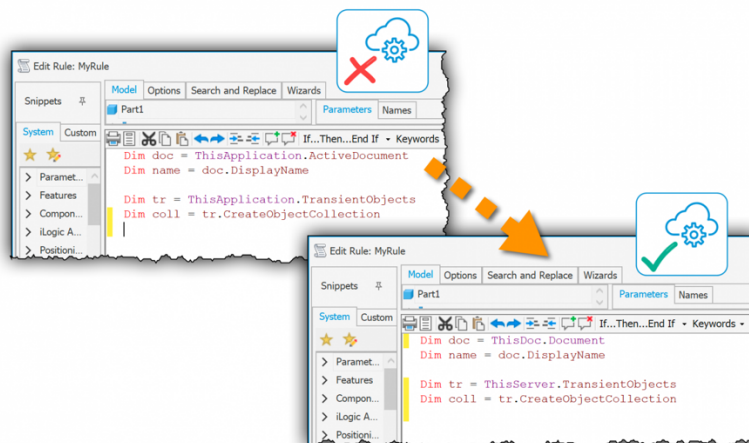
Based on this information you could automatically generate a similar user interface on your website for your clients.

That is what the sample created by the engineering team is doing as well.

See <https://forge.autodesk.com/blog/get-ilogic-form-information-inventor-documents>

Prepare iLogic Rules for Design Automation

When using **Design Automation API for Inventor**, you might have to change some of the code in the **iLogic Rules** of your documents.

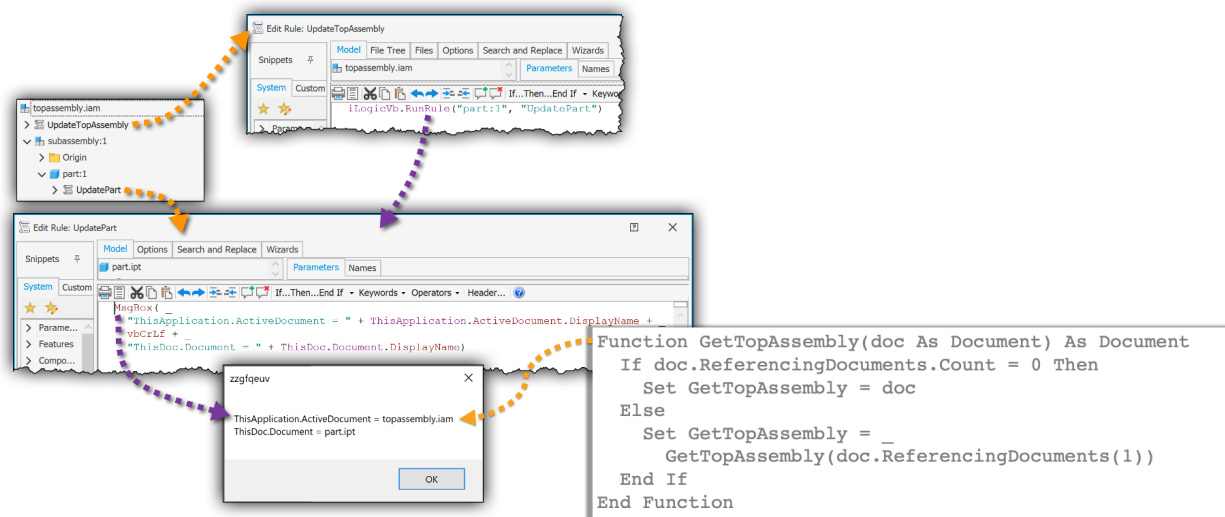


PREPARE CODE FOR DESIGN AUTOMATION

On the desktop inside **Inventor Application**, a user can open and activate documents without your add-in's permission, and so when one of its commands is executing, it might need to know which document is currently active - that's what the **ActiveDocument** property of the **Application** class will tell us.

On the cloud, using **Inventor Server**, your application (i.e. the code inside your **App Bundle**) has full control of things. So, there is not much point in keeping track of which document is active - your application should know what document it opened and needs to work with. And that's why there is no **InventorServer.ActiveDocument**.

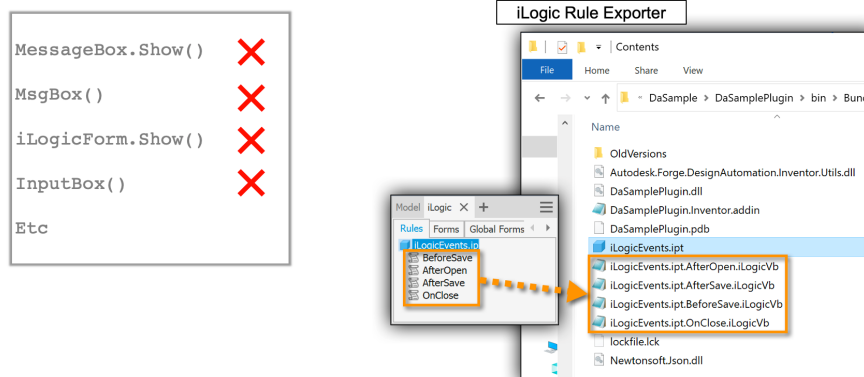
ThisApplication itself is not supported either. You could use **ThisServer** instead which is also supported on the desktop. This will only contain properties and functions that **Inventor Server** supports, so you won't find e.g. **ActiveDocument** property on it.



ACTIVEDOCUMENT VS THISDOC.DOCUMENT

Most of the time you could simply replace **ThisApplication.ActiveDocument** in your **iLogic Rules** with **ThisDoc.Document** (which returns the document whose **Rule** is running) - it depends on the **Rule's** intent.

However, in certain cases **ThisApplication.ActiveDocument** might have been a shortcut to the **top assembly** containing the **part** whose **Rule** (in our case "**UpdatePart**") is running: In that case you would need a function to get to the top assembly. Something like what the **GetTopAssembly** function shown above does.



ILOGIC RULE EXPORTER

Since there is no user to interact with, you'll have to remove all the message boxes and other components that the user would have to react to - i.e. calls to `MessageBox.Show()`, `MsgBox()`, `iLogicForm.Show()`, `InputBox()`, etc

There is a very useful tool called "**iLogic Rule Exporter**" that can help you find where such functions are used in your **iLogic Rules**.

It enables you to extract all the **Rules** from any **Inventor** document and its referenced documents into **iLogicVb** files

This then makes it much easier to search their source code for certain things: e.g. where a particular parameter is modified, where specific **Inventor API**'s are used, and so on.

That should help you find where **UI** components are used and work around them.

See <https://forge.autodesk.com/blog/prepare-ilogic-rules-design-automation>

Additional Resources

I provided links to relevant articles in each section. Most of them will be referencing articles on the **Forge Blog**: <https://forge.autodesk.com/blog>

In order to get started developing something with **Forge**, the best place to visit is the **Learn Forge** website: <https://learnforge.autodesk.io/#/>

There is also the **Forge Online Documentation**:
<https://forge.autodesk.com/developer/documentation>