



# AUTODESK UNIVERSITY 2015

SD9945

## Integrating .NET Code with AutoCAD I/O to Add Design Intelligence to Your Website

Kean Walmsley  
Autodesk

### Learning Objectives

- Get started with the AutoCAD I/O service
- Learn how to take existing AutoCAD .NET applications and create "core" modules
- Learn how to drive AutoCAD I/O with custom .NET application from a website or web service
- Learn how to integrate the results from AutoCAD I/O, displaying them via the web

### Description

Over the last decade or so, software developers have amassed a significant amount of intellectual property harnessing AutoCAD software's .NET API. AutoCAD I/O enables standard AutoCAD software commands, as well as those implemented in .NET, to be executed in the cloud, generating results that you can integrate into your own business-to-business or business-to-customer website. This class will take a concrete example of a .NET application creating custom jigsaw puzzles inside AutoCAD software. During the class we will show how to move the core implementation to AutoCAD I/O via the Core Console, and then make use of this to power a new business-to-customer website. Potential customers will be able to specify custom designs for jigsaw puzzles and visualize the results before finalizing their orders.

### Your AU Experts

*Kean Walmsley is a software architect for AutoCAD software products with Autodesk, Inc. He writes regular posts for his popular development-oriented blog, [Through the Interface](#).*

## Introduction

During the course of this session, we're going to look at the steps required to implement a web-site for turning pictures into jigsaw puzzles. We're not going to go quite as far as creating the puzzles, themselves, but rather the DWG or DWF files that can be used to drive a laser cutter to do the work.

The premise for the application is the following: laser cutters are very good at cutting, but engraving can be very time-consuming, especially for larger projects. It can also be quite complicated to upload images for engraving and have them fit a cutting layer – there's quite a lot of manual effort needed to get the positioning right.

Our site – [jigsawify.com](http://jigsawify.com) – takes a different approach. It performs edge detection (a common Computer Vision algorithm) on the chosen image, extracting the vectors (although in our case these will be encoded as pixels) and adds them on a separate cutting layer to the output file. The code will also create the outline for the puzzle itself, of course, which will be randomly generated based on the specified dimensions and number of pieces.



Now, though, it's time to introduce AutoCAD I/O...

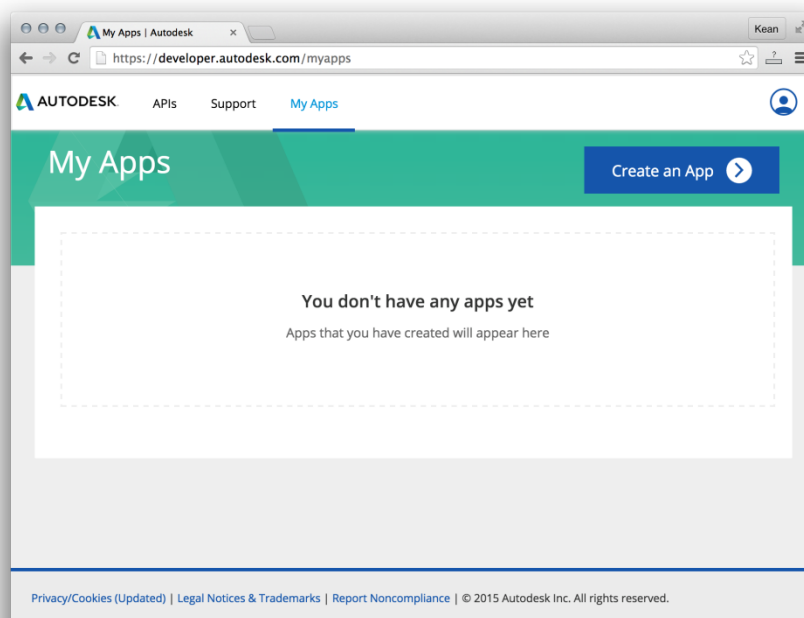


## Getting Started with AutoCAD I/O

The first destination on your journey with AutoCAD I/O is the Autodesk developer portal. It's from here that you'll register your application and get hold of the credentials you'll need to authenticate against and call the AutoCAD I/O web-service.

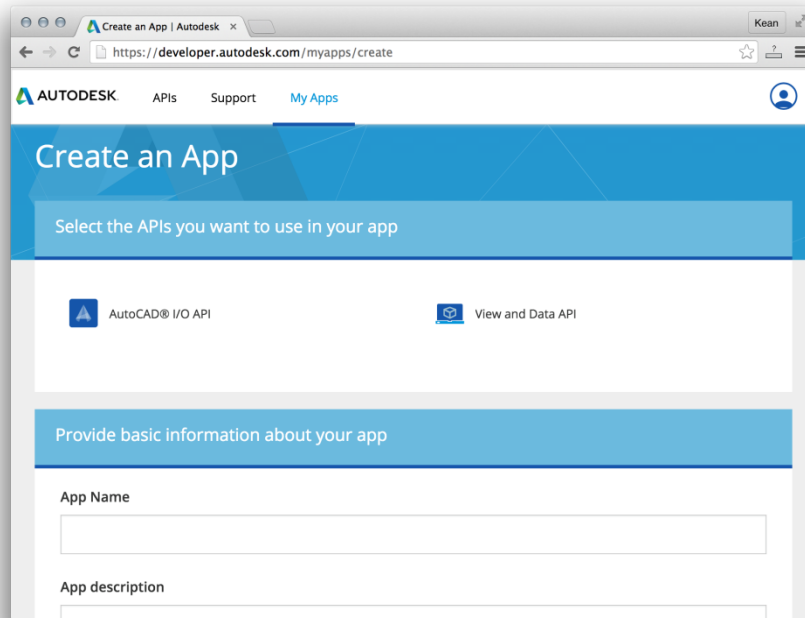
<http://developer.autodesk.com>

Once you've registered and signed in, you'll be able to create your first app via the *My Apps* tab.



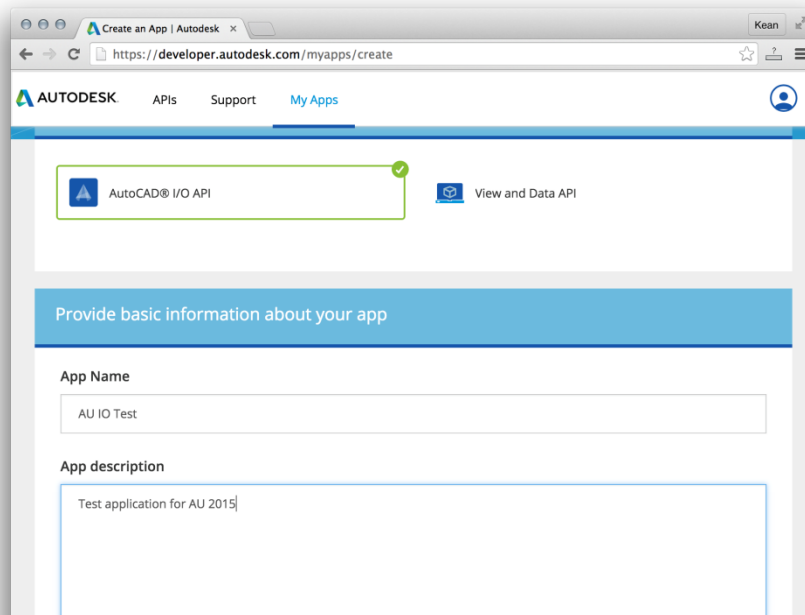
Selecting *Create An App* will take you to a page where you can enter information regarding your app.





The screenshot shows the Autodesk 'Create an App' interface. At the top, there's a navigation bar with 'AUTODESK', 'APIs', 'Support', and 'My Apps'. The main heading is 'Create an App'. Below it, a blue bar says 'Select the APIs you want to use in your app'. Two API options are listed: 'AutoCAD® I/O API' (selected) and 'View and Data API'. Below this, another blue bar says 'Provide basic information about your app'. There are two input fields: 'App Name' and 'App description'.

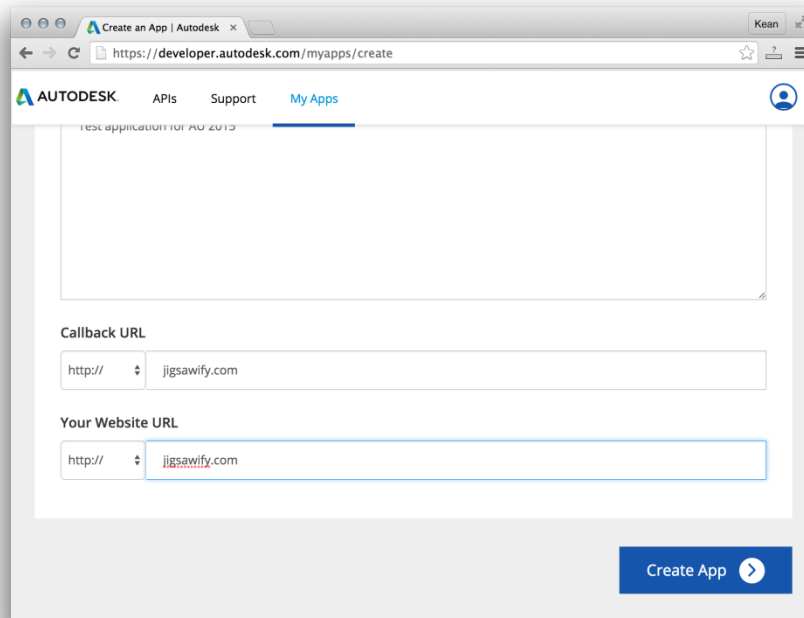
You need to select AutoCAD I/O API and enter a unique app name and a description.



This screenshot shows the same 'Create an App' page, but now the 'AutoCAD® I/O API' is selected, indicated by a green checkmark and a green border. The 'App Name' field now contains the text 'AU IO Test'. The 'App description' field contains the text 'Test application for AU 2015'.

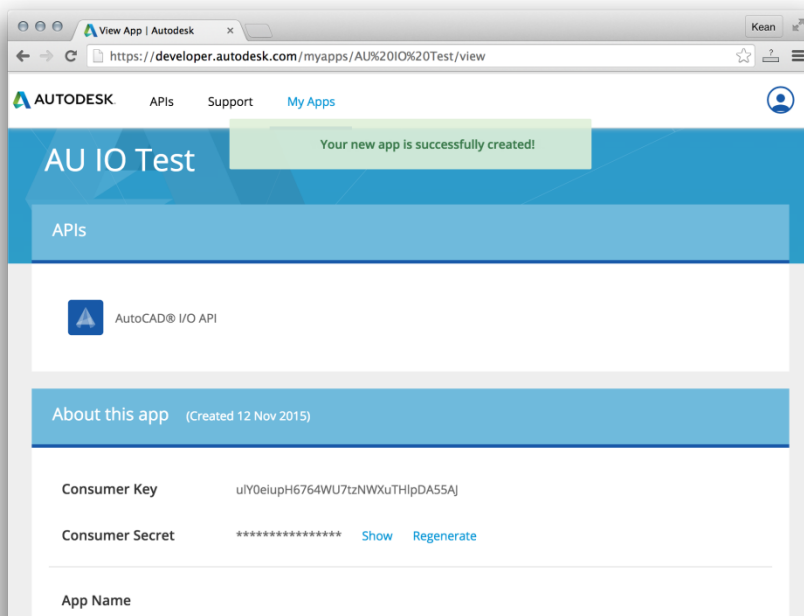


After which you specify a callback URL and an optional URL for your website. If you don't currently have a URL to enter for these, don't worry: it's something you can edit afterwards, in case.



The screenshot shows the 'Create an App' page on the Autodesk Developer Portal. The browser address bar shows the URL `https://developer.autodesk.com/myapps/create`. The page has a navigation bar with 'AUTODESK', 'APIs', 'Support', and 'My Apps'. Below the navigation bar, there is a large text area for a description. Underneath, there are two input fields: 'Callback URL' and 'Your Website URL'. Both fields have a dropdown menu set to 'http://' and the text 'jigsawify.com'. At the bottom right, there is a blue button labeled 'Create App' with a right-pointing arrow.

When you hit *Create App*, you'll get access to your Consumer Key and Secret.



The screenshot shows the 'View App' page on the Autodesk Developer Portal. The browser address bar shows the URL `https://developer.autodesk.com/myapps/AU%20IO%20Test/view`. The page has a navigation bar with 'AUTODESK', 'APIs', 'Support', and 'My Apps'. Below the navigation bar, there is a blue header with the text 'AU IO Test' and a green notification box that says 'Your new app is successfully created!'. Underneath, there is a section titled 'APIs' with a blue button labeled 'AutoCAD® I/O API'. Below that, there is a section titled 'About this app' with the text '(Created 12 Nov 2015)'. Underneath, there is a table with two rows: 'Consumer Key' and 'Consumer Secret'. The 'Consumer Key' row has the value 'uIY0eiupH6764WU7tzNWxuTHlpDA55AJ'. The 'Consumer Secret' row has the value '\*\*\*\*\*' and two links: 'Show' and 'Regenerate'. At the bottom, there is a section titled 'App Name'.



It's this information that needs to be placed in your custom application for it to be able to access the AutoCAD I/O service.

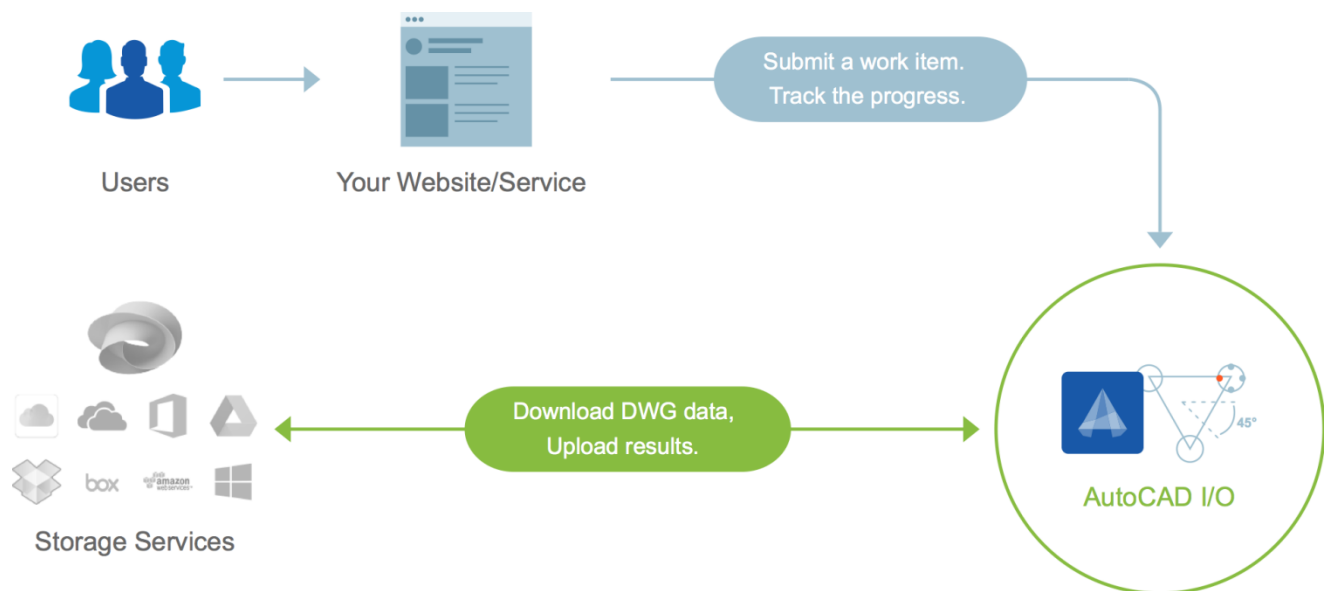
At this stage it's worth talking a bit about the need for care with these credentials: if someone uses them to access AutoCAD I/O – or another Autodesk web service – then any costs incurred (whether against a free quota or actually costing real money) will be from your account. It's therefore very important not to embed this information in any module that gets shipped to your customers: we will certainly use these credentials when administering our AutoCAD I/O activities, for instance, but only from a local .NET application. At no point should that application be distributed externally.

The typical way to manage this is to place the credentials behind a web-service that exposes some kind of authentication function. You can call this from your web-site – or another web-service – to retrieve an authorization token to be used for calls into AutoCAD I/O.

Now that we have our credentials, it's time to use them to call some functionality on AutoCAD I/O. It's worth taking a look at the AutoCAD I/O documentation:

<https://developer.autodesk.com/api/autocadio/v2>

This page gives a nice overview of the capabilities of AutoCAD I/O, including this high-level process diagram:



Our next step is with the AutoCAD I/O samples on GitHub:

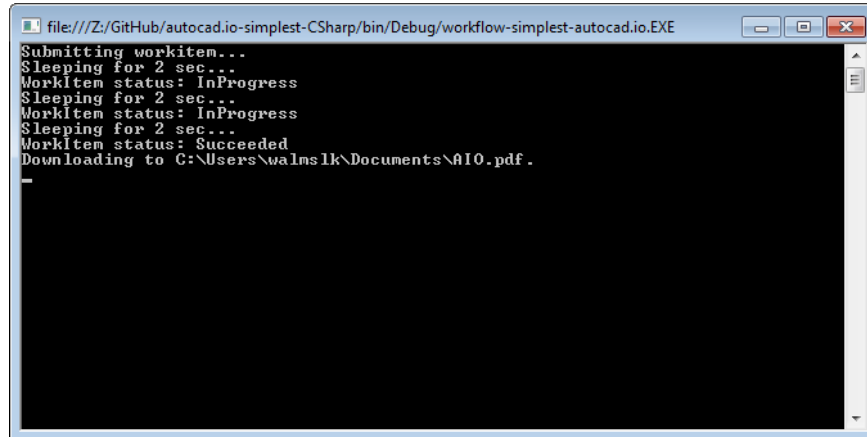
<https://github.com/Developer-Autodesk/AutoCAD.io>



The one we're going to use is the "Simplest C#" sample that makes use of the standard PlotToPDF activity:

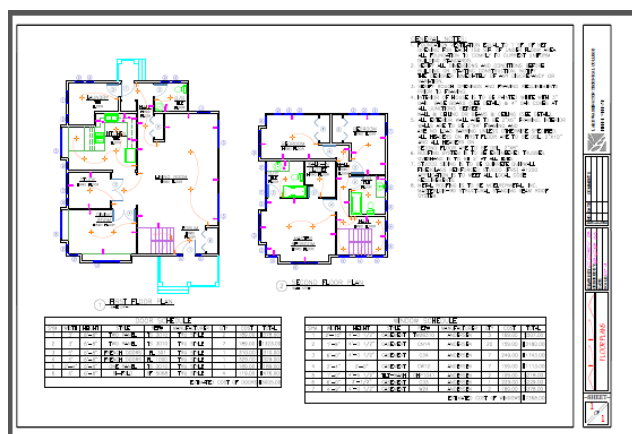
<https://github.com/Developer-Autodesk/autocad.io-simplest-CSharp>

Once we've cloned this sample to our local system, we can plug our credentials into the *Program.cs* file, build the sample in Visual Studio and then run it.



```
file:///Z:/GitHub/autocad.io-simplest-CSharp/bin/Debug/workflow-simplest-autocad.io.EXE
Submitting workitem...
Sleeping for 2 sec...
Workitem status: InProgress
Sleeping for 2 sec...
Workitem status: InProgress
Sleeping for 2 sec...
Workitem status: Succeeded
Downloading to C:\Users\walmslk\Documents\AIO.pdf.
```

If you take a look at My Documents on your system, you should find two files: *AIO-report.txt* and *AIO.pdf*. The first contains the command-line output from the AcCoreConsole.exe process that was used to execute the WorkItem, the second contains the PDF output:



To understand a little more about what's happening, let's take a look at the primary sub-routine:



```

static void Main(string[] args)
{
    //instruct client side library to insert token as Authorization value into each request

    var container =
        new Container(new Uri("https://developer.api.autodesk.com/autocad.io/us-east/v2/"));
    var token = GetToken();
    container.SendingRequest2 +=
        (sender, e) => e.RequestMessage.SetHeader("Authorization", token);

    //create a workitem

    var wi = new WorkItem()
    {
        Id = "", //must be set to empty
        Arguments = new Arguments(),
        ActivityId = "PlotToPDF" //PlotToPDF is a predefined activity
    };

    wi.Arguments.InputArguments.Add(new Argument()
    {
        Name = "HostDwg", // Must match the input parameter in activity
        Resource =
            "http://download.autodesk.com/us/samplefiles/acad/blocks_and_tables_-_imperial.dwg",
        StorageProvider =
            StorageProvider.Generic //Generic HTTP download (as opposed to A360)
    });

    wi.Arguments.OutputArguments.Add(new Argument()
    {
        Name = "Result", //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
        Resource = null //use storage provided by AutoCAD.IO
    });

    container.AddToWorkItems(wi);
    Console.WriteLine("Submitting workitem...");
    container.SaveChanges();

    //polling loop

    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);

    //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();

//Resource property of the output argument "Result" will have the output url

var url = wi.Arguments.OutputArguments.First(a => a.Name == "Result").Resource;
DownloadToDocs(url, "AIO.pdf");

//download the status report

url = wi.StatusDetails.Report;
DownloadToDocs(url, "AIO-report.txt");
}
```

In a nutshell, we're creating a WorkItem with one input argument – a standard sample DWG located on the web – and one output argument – the location at which the resultant PDF should be placed (and this has been left blank, as AutoCAD I/O is able to assign this for us).

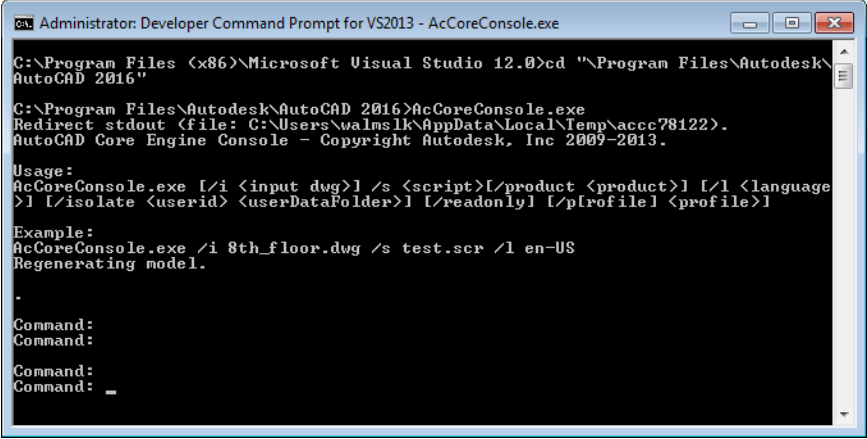
Then we simply wait for the WorkItem to complete, at which point we download both the PDF and the status report.



## Creating core modules for your .NET apps

Our goal is to host custom code – which could be LISP or C++, as well as our own choice of .NET – in AutoCAD I/O. In order to do this, we need to create a module that can be loaded and executed by the Core Console.

To understand the capabilities of the Core Console, the simplest way is to open a command prompt, navigate to AutoCAD's *Program Files* folder, and launch *AcCoreConsole.exe*:



```

Administrator: Developer Command Prompt for VS2013 - AcCoreConsole.exe

C:\Program Files (x86)\Microsoft Visual Studio 12.0>cd "%Program Files\Autodesk\AutoCAD 2016"

C:\Program Files\Autodesk\AutoCAD 2016>AcCoreConsole.exe
Redirect stdout (file: C:\Users\walmslk\AppData\Local\Temp\accc78122).
AutoCAD Core Engine Console - Copyright Autodesk, Inc 2009-2013.

Usage:
AcCoreConsole.exe [/i <input dwg>] /s <script> [/product <product>] [/l <language>]
>] [/isolate <userid> <userDataFolder>] [/readonly] [/profile <profile>]

Example:
AcCoreConsole.exe /i 8th_floor.dwg /s test.scr /l en-US
Regenerating model.

Command:
Command:

Command:
Command: _

```

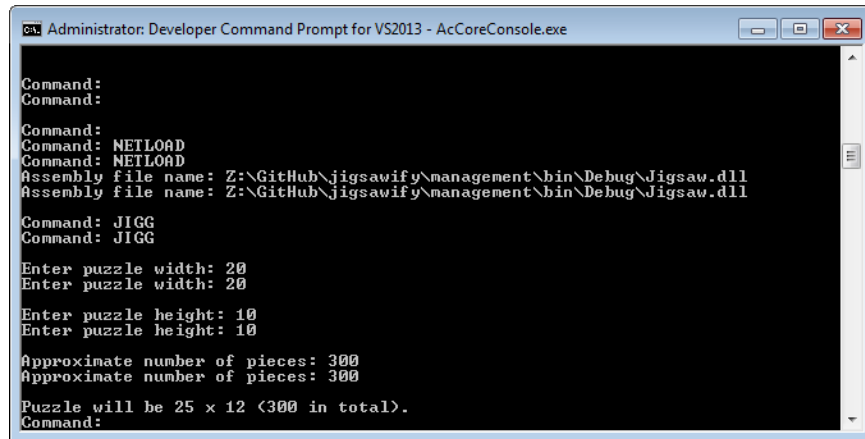
From here you get access to a very large subset of AutoCAD commands and API functionality. For instance, even though it's at the command-line, there's actually a significant amount of graphics-related functionality that's available.

To target the Core Console from a .NET app, you need to make sure you reference only *AcDbMgd.dll* and *AcCoreMgd.dll* (not *AcMgd.dll*, which is specific to the full AutoCAD editor). You can also do this using NuGet by adding the *AutoCAD.NET.Core* package.

This will automatically give you access to the subset of functionality available to applications running in either the Core Console or in AutoCAD I/O. You'll clearly need to provide a command-line interface to your commands: providing a GUI isn't going to work.

At a first step, here's a simple command that we call successfully from the Core Console:





```

Administrator: Developer Command Prompt for VS2013 - AcCoreConsole.exe

Command:
Command:

Command:
Command: NETLOAD
Command: NETLOAD
Assembly file name: Z:\GitHub\jigsawify\management\bin\Debug\Jigsaw.dll
Assembly file name: Z:\GitHub\jigsawify\management\bin\Debug\Jigsaw.dll

Command: JIGG
Command: JIGG

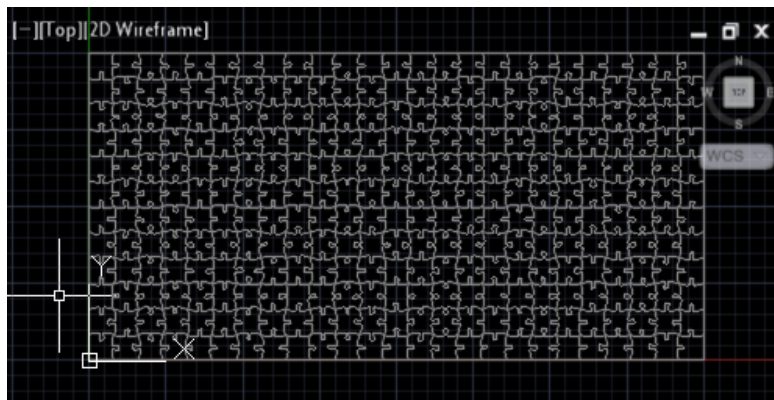
Enter puzzle width: 20
Enter puzzle width: 20

Enter puzzle height: 10
Enter puzzle height: 10

Approximate number of pieces: 300
Approximate number of pieces: 300
Puzzle will be 25 x 12 (300 in total).
Command:

```

If we save the results, we can load the DWG into AutoCAD to make sure it worked:



The thing is, this style of command-line interface – where we use methods such as `Editor.GetString()` to get the values for the individual parameters – can't be used exactly like this for AutoCAD I/O. The parameters for AutoCAD I/O activities are passed as JSON, which gets written to a file on the local instance and read in by the command being executed.

It's this level of indirection that allows AutoCAD I/O to have Workitems with different parameter values, and the effort from a developer's perspective is modest: they simply need to use `Editor.GetString()` to find the path to the JSON parameters file, and then use a component such as *Newtonsoft.Json.dll* to deserialize it into usable settings.

One way to do this is to have a single, shared function that gets called by the various command implementations – in our case JIGG and JIGIO.



## Creating a custom Activity with a .NET AppPackage

Now that we have a working CRX module (which is the term used for modules targeting the AutoCAD Core), we can package it up and create an Activity in AutoCAD I/O that we can (eventually) use from our WorkItems.

The simplest way to do this is to modify an existing project, such as this one:

<https://github.com/Developer-Autodesk/autocad.io-custom-activity-apppackage-CSharp>

This base project has been integrated into the source project for the Jigsawify.com website:

<https://github.com/KeanW/jigsawify>

Once you've updated the CrxApp source to build your custom module – and made some other minor adjustments, elsewhere – you will be able to run the console app to generate both the AppPackage containing your module and the Activity that makes use of it.

```
file:///Z:/GitHub/jigsawify/management/bin/Debug/ActivityManager.EXE
AppPackage 'Adsk_JigsawPackage' already exists. What do you want to do? [Delete/
Recreate/Update/Leave]<Update>:R
Generating autoloader zip...
Creating/Updating AppPackage...
Uploading autoloader zip...
Activity 'Adsk_JigsawActivity' already exists. Do you want to recreate it? [Yes/
No]<No>:Y
Creating/Updating Activity...
Submitting workitem...
Sleeping for 2 sec...
WorkItem status: InProgress
Sleeping for 2 sec...
WorkItem status: InProgress
Sleeping for 2 sec...
WorkItem status: InProgress
Sleeping for 2 sec...
WorkItem status: InProgress
Sleeping for 2 sec...
WorkItem status: Succeeded
Downloading to C:\Users\walmslk\Documents\AI0.zip.
Downloading to C:\Users\walmslk\Documents\AI0-report.txt.
```

As you can see from the above output, it also contains some code to create a test WorkItem, just to make sure the Activity and its AppPackage work properly.



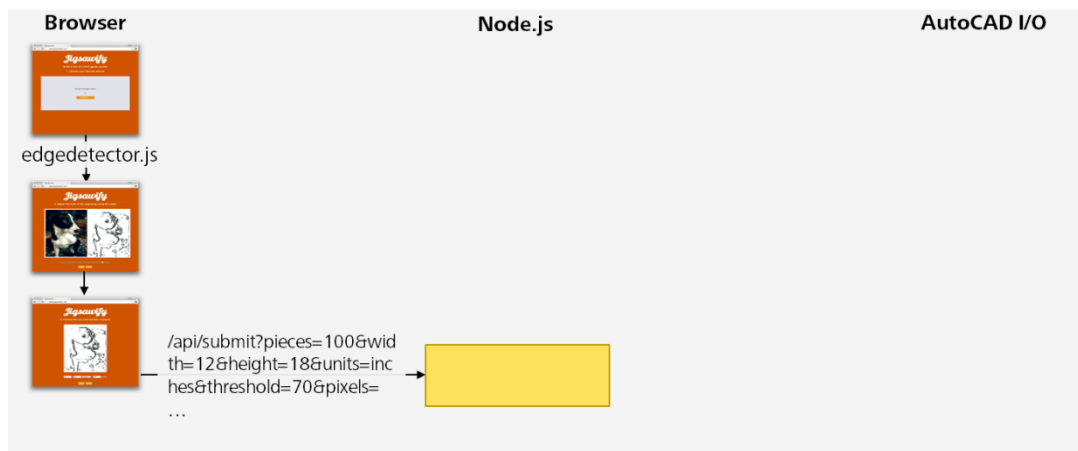
## Calling a custom Activity from your web-site

Now that we've created our custom Activity – and the AppPackage it depends on – we need to have our web-site call it.

In this case we're going to review a few different architectures, to understand what works and what doesn't.

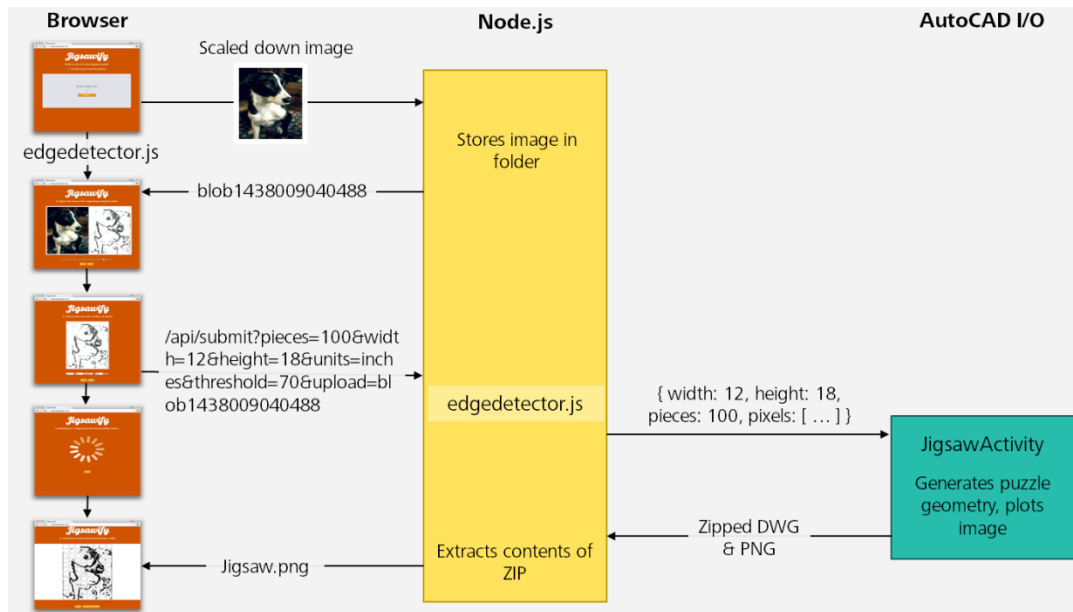
In our application, we want to do some work in the browser – to perform some edge detection on a user-provided image, using the resultant pixels as an overlay for our jigsaw puzzle – and the tricky thing is communicating these pixels from the browser to our AutoCAD I/O Activity.

This is the first approach I took when developing the prototype. It involved encoding all the pixels as part of the URL fired off by the web-site:

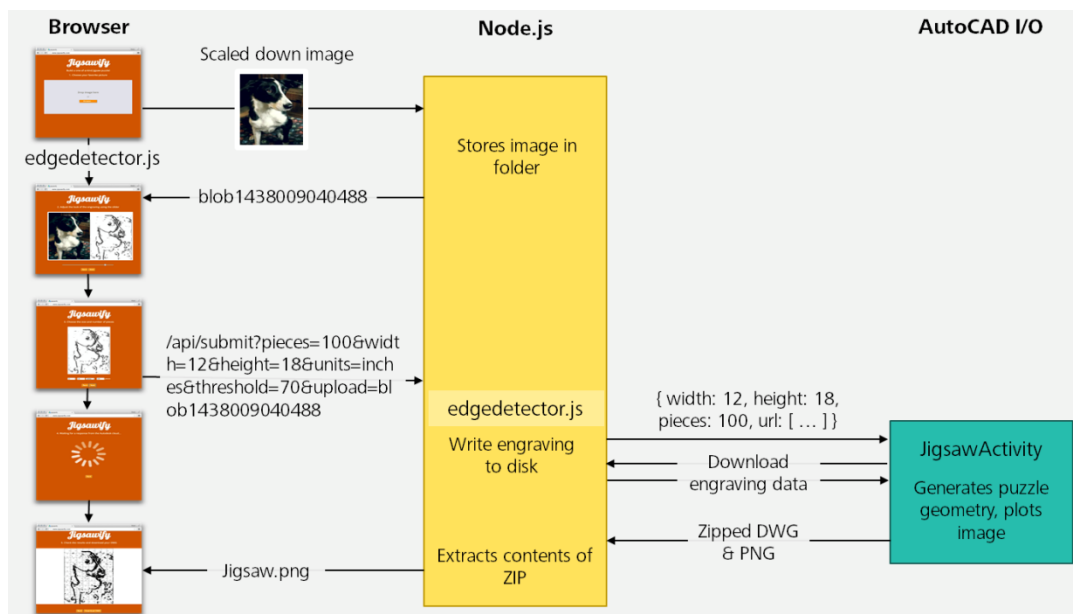


This fails because there is a fairly small limit on the size of a valid URL (it's around 2,000 characters, but that's still not enough for complex engravings). I did try encoding the data in different ways, but generally this only pushed the limit out slightly.

The next approach required some additional complexity on the back-end: I decided to upload an appropriately scaled version of the selected image – returning an identifier to the browser – and then run the same JavaScript edge detection code on the server – this is in many ways the beauty of Node.js – to determine the pixels to send to AutoCAD I/O.



This worked well for small images, but soon caused issues on the server side with larger ones. It turns out that encoding the pixels inside the WorkItem parameters caused an issue when logging the WorkItem contents on the server side: if the size of the parameters passed exceeded 40K, the processing would fail. While arguably a server-side issue, it certainly required a process rethink.

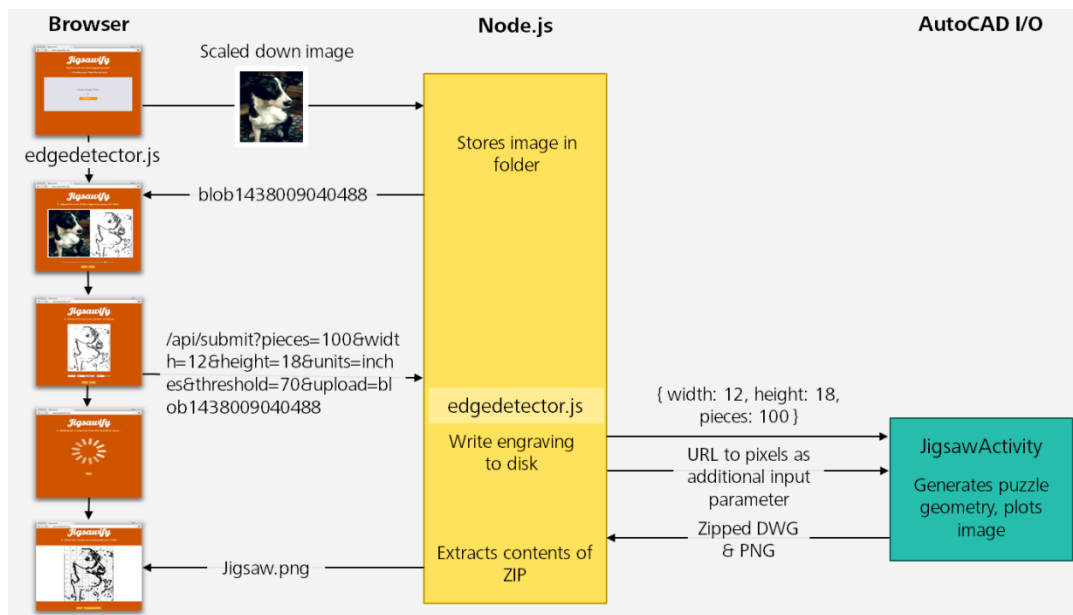


The next version, above, passed a URL as part of the parameters read by our Activity, which then attempts to download the pixel data stored in the custom web-service. This worked, initially, until the

AutoCAD I/O team shored up the server-side security: Activities are not allowed direct access to external network resources, for very good reason. The machine instances that drive AutoCAD I/O run *AcCoreConsole.exe* in a sandbox – with limited user privileges, preventing (among other things) network access.

If you want to have data downloaded and made available to your Activity, you need to specify it as an input parameter: the Core Engine Runner process – part of AutoCAD I/O's execution infrastructure – goes through the various input parameters and downloads their content for local usage.

Here's the approach that finally worked: we have an additional input parameter that specifies the URL to our pixel data. The Core Engine Runner downloads the contents of this JSON file to the local system, from which our command can load and deserialize the data to do something with it.



The system is now working well, and can be used to generate DWG and DXF files to drive a laser cutter. Give it a try at [Jigsawify.com](http://Jigsawify.com) or [check out the source](#) to build your own site that uses AutoCAD I/O with custom .NET code for added design intelligence.

