Autodesk University 2012

Moving code to the cloud – it's easier than you think

Kean Walmsley – Autodesk

CP1914 This class will discuss the benefits of the cloud for software developers and look at a concrete example of moving desktop functionality to the cloud. We will look at issues related to the architecture of software destined for the cloud. We will compare commercial cloud offerings such as Amazon Web Services™ and Windows Azure™ and will take an look a several client-side environments for consuming this data, such as AutoCAD® software , Unity 3D, Android™, Apple® iOS, and HTML5.

Learning Objectives

At the end of this class, you will be able to:

- Apply best practices for architecture of cloud-based software components
- Select an appropriate cloud-hosting provider
- Create a simple web-service and deploy it to the cloud
- Consume a cloud-based web-service from a variety of client environments

About the Speaker

Kean has been with Autodesk since 1995, working for most of that time in a variety of roles – and in a number of different countries – for the Autodesk Developer Network organization. Kean's current role is Software Architect for the AutoCAD family of products, and he continues to write regular posts for his popular development-oriented blog, "Through the Interface" (http://blogs.autodesk.com/through-the-interface). Kean currently lives and works in Switzerland.

kean.walmsley@autodesk.com

Why all this talk of the cloud?

The software industry is steadily adopting a model commonly referred to as “cloud computing”. For many software developers, this will not appear to be anything very new: many have worked with centralized computing resources in the past – in the mainframe era – but that’s not to say that this shift isn’t valid.

So let’s step back and look briefly at how the industry has evolved over recent decades. The mainframe era gave way to personal computing: Autodesk was one of many software companies that identified this trend and rode the wave to become a successfully business. The initial releases of AutoCAD were far from being really useable, in many ways, but it was clear that Moore’s Law – which indicated that the number of transistors on chips would double every 18-24 months – would hold long enough for the various pieces of the puzzle to provide the necessary performance from a personal computer before very long.

Nothing lasts forever, especially if it’s an exponential law. Moore’s Law as we know it has hit a bit of a barrier in recent years: we’re no longer seeing CPU clock speed doubling, for instance, as we’re hitting certain physical laws that prevent this from happening. This article from Herb Sutter does a great job of explaining this in detail:

http://www.gotw.ca/publications/concurrency-ddj.htm

That said, technology continues to evolve: rather than chips doubling in speed, we’re seeing the number of cores doubling, and the overall computing power that’s available to us via the cloud continuing to grow, too. For more information on this shift, see Herb’s sequel to the above article:

http://herbsutter.com/welcome-to-the-jungle

Ultimately, for software developers to continue to see performance gains (and cost efficiencies) software is having to be architected to work in a more distributed manner, with much of its processing performed on the cloud. This work distribution is being made possible via improvements in infrastructure – it’s very common for people to have redundant methods of accessing the Internet, for instance, whether wired or wirelessly – and the cost of centralized resources are continuing to drop as performance between major hosting providers turns computing resources into a commodity (and some would say utility: http://www.amazon.com/The-Big-Switch-Rewiring-Edison/dp/0393333949).

At the same time as we’re seeing some kind of plateauing in performance of local, *sequential* software execution (although note the emphasis on sequential: parallelizing code can still bring

performance gains from multi-core systems), we're seeing a rise in the availability of lower-powered, mobile devices. As we enter the post-PC era, it's increasingly the desire to be able to access centralized computing resources from devices that are little more than "dumb" terminals (although the modern smartphone contains more computing power than existed on the planet on the day many of their users was born).

And the world is becoming truly heterogeneous in terms of computing devices: over time software developers will decreasingly target specific operating systems, having core algorithms executing centrally. There may still be some amount of native code targeting various supported devices, but even that is likely to reduced as true cross-platform execution improves (whether via toolkits or HTML5).

Moving code to the cloud can make sense

So why would you move product functionality to the cloud? Here are some reasons:

- Performance
    - If you have a problem you can easily chunk up and parallelize – rendering is a great example of this, as we've seen with Project Neon(which it seems is now known as Autodesk 360 Rendering) – then the cloud can provide significant value. Renting 1 CPU for 10,000 seconds is (more or less) the same cost as buying 10,000 CPUs for 1 second.
- Scalability
    - With cloud services you pay for what you use, which should scale linearly with your company's income (or benefits) from hosting functionality in that way. Dynamic provisioning allows companies to spin up servers to manage usage spikes, too, which allows infrastructure to be made available "just in time" rather than "just in case".
- Reliability
    - You often hear about measurements such as "five nines" uptime, which means 99.999% availability (or about 5 minutes of downtime per year). Some providers are no doubt proving better than others at meeting their availability SLAs, but the fact remains: having a local system or server die generally creates more significant downtime than those suffered by the outages suffered by cloud providers. And that should only get better, over time.
- Low cost

    - As cloud services get increasingly commoditized – and Microsoft, Google and Amazon are competing fiercely in the cloud space, driving costs down further – using the cloud is becoming increasingly cost-effective.

That's a bit about the "why", here's the "when"…

- Computation intensive
    - If you have serious number crunching going on locally in your desktop apps – which either ties up resources that could be used differently or stops your apps running on lower-spec hardware – then the cloud is likely to look attractive. I mentioned we use make the cloud available for rendering, but we're doing the same with simulation and analysis, too.
- Collaboration
    - Imagine implementing the collaboration features of AutoCAD WS without the cloud…
- Frequent change
    - If you have applications that go through rapid release cycles, then update deployment/patching is likely to be a challenge for you. Hosting capabilities on the cloud – appropriately versioned when you make breaking interface changes, of course – can certainly help address this.
- Large data sets
    - The ideal scenario is clearly that data is co-located with the processing capability that needs to access it. Much of this data is currently stored on local systems – which makes harnessing the cloud a challenge – but as data shifts to be hosted there (for lots of very good reasons), this starts to become more compelling.
    - Another example: let's say you have an application that relies on a local database of pricing information. Making sure this database is up-to-date can be a royal pain: it's little surprise, then, that a number of the early adopters of cloud technology in the design space relate to pricing applications.

These were the main benefits presented to ADN members during DevDays. There are few additional benefits that I'd like to add…

- Customer intimacy
    - Delivering software as a service can increase the intimacy you have with customers – and with partners, if you're providing a platform. You have very good knowledge of how your technology is being used – and this has a "Big Brother"

flip-side that people often struggle with, as you clearly have to trust your technology provider – which can allow you to provide better service and even anticipate customer needs.

- Technology abstraction
    - You may have some atypical code that you'd like the user to not have to worry about: let's say you have some legacy product functionality implemented using Fortran or COBOL that you'd rather not have to provide a local runtime to support. Hiding it behind a web service reduces the complexity in deploying the application and can provide a much cleaner installation/deployment/usage capability.
- Device support
    - This is probably obvious (as many of the preceding points will have been to some of you, I expect), but web services are accessible from all modern programming languages on any internet-enabled device. Web services are a great way to more quickly support a variety of usages of your application's capabilities on a variety of devices.

Today's Example

We're going to take an example that hits on a few of these topics: we have a core algorithm – implemented in F#, which some might questionably classify as arcane ;-) – that we want to move behind a cloud-hosted web-service and use from a number of different devices.

The algorithm generates Apollonian Gaskets – a 2D fractal, which places as many circles as it can within the "whitespace" inside a circle – and Apollonian Packings – the 3D equivalent which obvious deals with spheres.

We're going to have fun using this service to generate some interesting 3D visualizations on a variety of platforms.

Choosing a cloud hosting provider

Autodesk is a very heavy user of Amazon Web Services, which might indicate it would be a good, long-term choice for users and developers to adopt (as co-location with data is of benefit, as we've seen).

That said, there are lots of factors that contribute to this kind of decision.

The early popularity of AWS was due in large part to its focus on providing Infrastructure-as-a-Service (IaaS): they made it really easy for companies with their own servers to move them across to be hosted centrally. Many companies shifted from on-premise servers (or perhaps their own data-centers) to centrally hosted and managed servers.

Microsoft's approach has been to deliver highly integrated Platform-as-a-Service (PaaS) offerings: they abstract away the physical machine, focusing on the "roles" that you deploy to the cloud. Microsoft is now starting to deliver via an IaaS model, just as Amazon is providing more by way of PaaS from their side.

In our particular example, we're going to make use of Windows Azure. That's not to say it's better for everyone – it's just what I've chosen to use for this project as the integration with Visual Studio is first-class and I have free hosting provided via my MSDN subscription.

If you're interested in AWS, I recommend looking at some of the guides on ADN's Cloud & Mobile DevBlog (http://adndevblog.typepad.com/cloud_and_mobile).

Another option is Google App Engine, which provides an even higher level of abstraction that Azure. It seems to be an excellent system for highly granular, scalable tasks (without even having the underlying concept of physical machines in the picture). If interested in learning more about Google App Engine, I recommend attending tomorrow's 8am class by my colleague (and manager), Ravi Krishnaswamy:

CP2568 – PaaSt the Desktop: Implementing Cloud-Based Productivity Solutions with the AutoCAD® ObjectARX® API

Architecting for the cloud

There are lots of decisions to be made when considering moving application functionality to the cloud.

Not least of which is "what is your core business logic?" meaning the algorithms that are application- and device-independent. The algorithms that moving away from your core implementation increases your flexibility and platform independence.

You also need to consider the data that needs to be transferred between the client and the cloud: both in terms of the arguments that need to be sent to your cloud-based "function" and the results that need to be brought back down to earth afterwards. Ideally you'd be working with data that's already hosted in the cloud – and this is likely to happen more and more, over time – but that's not necessarily where we are today.

You should also think about whether there are optimizations to be made around repeated data: should you be making use of cloud-hosted database storage (which is very cheap when compared with compute resources) or some kind of caching service? In our case we're going to re-calculate the data, each time, but we could very easily implement a cache of the "unit" results and then multiply those for specifically requested radii.

Another important question is whether offline working needs to be supported: does a local version of the algorithm – or a cache of local data – need to be maintained in order to enable this?

An increasingly easy decision, amongst others that are quite tricky, is how to expose your web-services. These days the commonly accepted approach is to expose RESTful services (http://en.wikipedia.org/wiki/Representational_state_transfer), which means the transport protocol for data is standard HTTP and (most commonly) any results will be encoded in JSON – JavaScript Object Notation (http://en.wikipedia.org/wiki/JSON). JSON isn't actually a required part of REST – it's also possible for RESTful services to return XML – but it has become the de-facto approach that is most favored by API consumers.

The previous "standard" was SOAP – Simple Object Access Protocol (http://en.wikipedia.org/wiki/SOAP). SOAP has gradually ceded ground to REST, as it required more effort to create XML envelopes containing the data to transmit to the web-service, and is generally more verbose and requires more bandwidth.

A common requirement is around authentication: you probably want to be able to monitor and control access to your web-services. This is not going to be covered in this class, but you may want to look at OAuth-compatible toolkits such as DotNetOpenAuth (http://www.dotnetopenauth.net), which comes pre-integrated with AS.NET 4.5.

How you end up exposing your RESTful web-services will depend on your choice of technology stack (although these days it should be simple to do so from which choice you make). The Microsoft stack – which would often involve ASP.NET at some level, irrespective of whether you host on AWS or Azure – certainly abstracts away a lot of the messiness with exposing web-services, but comes with a certain execution overhead. If you really want to get "close to the metal" then you might also want to consider a Linux environment: not only do you end up with lower execution overhead but the cost associated with Linux instances can be very interesting. And as we know, the actual implementation of the web-service should be largely irrelevant to the consumer.

For today's example we're going to go with Microsoft and choose its ASP.NET MVC 4 Web API. This is a great way to expose web-sites with associated web-services, and seems to be the product of choice for people using the Microsoft stack to expose web-services, these days. WCF, the Windows Communication Framework, provides some very interesting capabilities – especially when needing to marshal more complex data-types to and from web-services – but

our requirement is relatively simple and the Web API seems the best fit. The ADN DevBlog mentioned earlier provides some good information on using WCF.

Considering cloud costs

One of the key benefits of the cloud is its ability to scale as you provision more resources to deliver your web-services. The counterpoint is that if you over-estimate the resources required to do this, your costs will be proportionally higher than they need to be.

Companies making heavy use of the cloud tend to invest in tools they can use to scale up and down automatically based on usage. This is not a topic that we'll cover today, but it's worth pointing out that getting this right is important for any significant cloud-based deployment.

There are some general things you can do to keep costs in check: consider looking for ways to reduce your instance sizes – dropping from a small to an extra-small instance size can bring significant costs benefits (adding some caching or online database storage might be a way to enable this, helping reduce the processing load).

Online calculators are available to help you determine up-front costs associated with provisioning resources, here are the calculators for Azure (http://www.windowsazure/pricing) and AWS (http://aws.amazon.com/calculator). Be sure to monitor actual costs (and optimize provisioning based on real usage, ideally), to make sure they are in line with projections.

The Problem

Now let's go back to today's "problem". We want to move our business logic – the core F# algorithm used to generate 2D and 3D Apollonian fractals – behind a cloud-based web-service. The original implementation for the 3D packing algorithm was provided in C++ by a Professor of Mathematics at the prestigious ETH in Zurich, but I chosen to migrate the code to F# to see how it looked (and having the code in a language that isn't necessarily easy to get working on OS X, iOS and Android demonstrates the "technology abstraction" point from an earlier slide nicely.

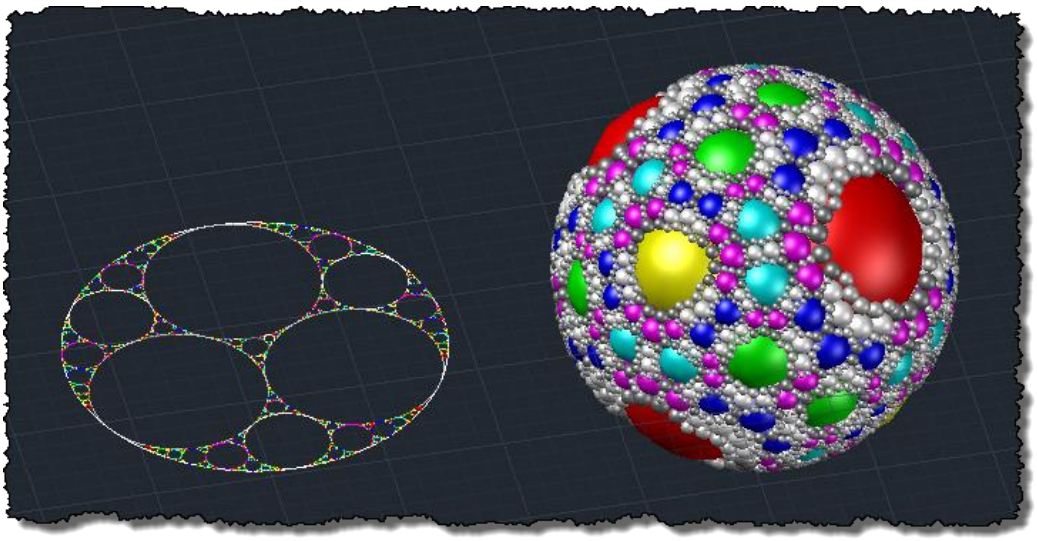
As mentioned earlier, we're not going to worry about authenticating users of our web-service: it's a topic that would probably deserve a class of its own. We're going to expose a simple, unsecured web-service (at least in terms of the need for authentication to make use of it).

Once it's up, you'll be able to query the geometry defining 2D Apollonian Gaskets using the "circles" API:

- http://apollonian.cloudapp.net/api/circles/2/2 (returns circle definitions via JSON)

And 3D Apollonian Packings using the “spheres” API:

- http://apollonian.cloudapp.net/api/spheres/2/2 (returns sphere definitions via JSON)



Building a simple web-service

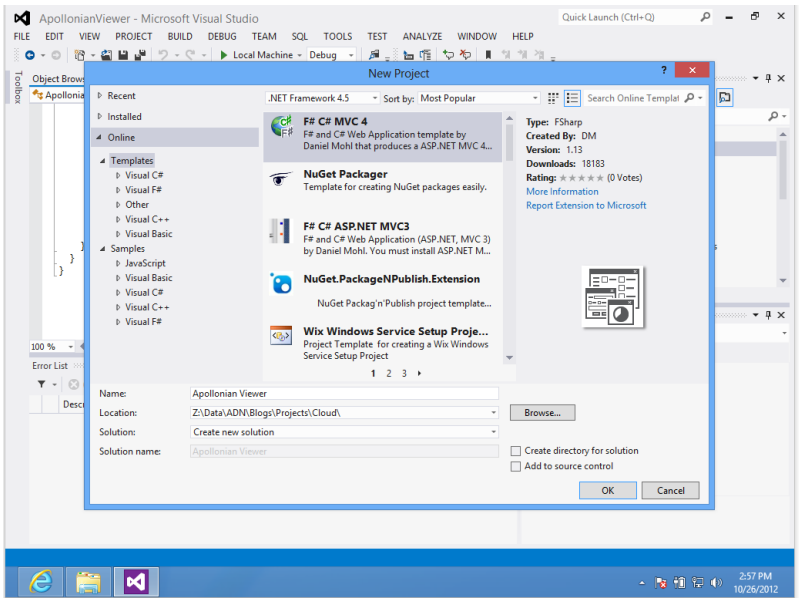
We’re going to use the ASP.NET MVC 4 Web API to expose our web-service. MVC – standing for the common “Model View Controller” architectural pattern, which is used to separate model data from UI and interactions – is Microsoft’s technology of choice for defining and hosting web-sites and -services on top of ASP.NET.

We don’t care a great deal about the web-site – we’re much more interested in the web-services – but we’ll go ahead and create one, anyway.

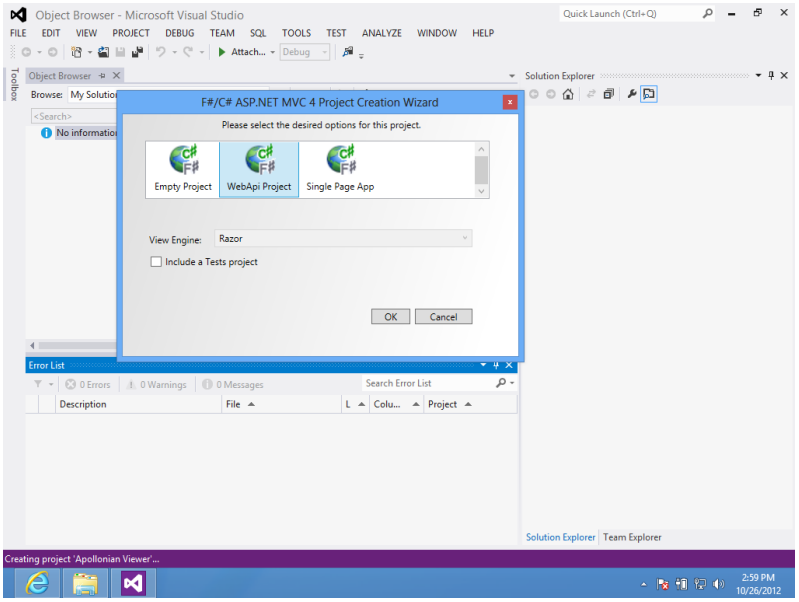
While Windows Azure apparently now supports .NET 4.5, we’re going to stick with .NET 4.0 (at the time of writing this new capability had only just been announced). We’re going to install an F#-aware project template into VS2012 and make use of that to create our Web API project.

Once published to Azure, the code will be hosted and executed on Windows Server 2008 (although this is really a detail – this is not something we should have to worry about at all).

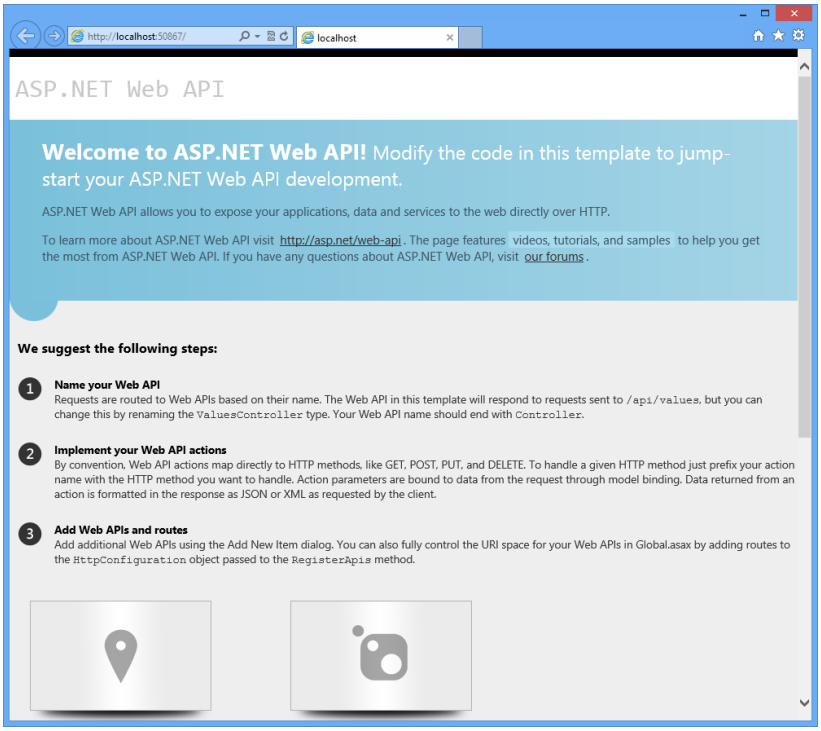
We'll start by getting our project template installed. We can select it via the "Extensions and Updates" manager on the VS 2012 Tools menu, searching for "F# C# MVC 4".



Once installed, we can launch a project of this type and select "WebApi Project":



Visual Studio will go ahead and create our basic project from the template. We can launch the default web-site via the debugger:



To test the default implementation of the web-service, try appending the following suffix to the URL:

/api/values

At this point the browser will ask us whether we want to save or open the results from the web-service. Opening the results in Notepad should show them to be ["value1","value2"].

There are a few changes we'll make to the project to get it working as we want it.

Firstly, we should change the .NET Framework target to 4.0 from 4.5 for both the contained projects.

Then some changes to the web-site project (ApollonianPackingWebApi).

We want to copy across some files from the "ToCopy" folder:

- *Site.css* into the Content folder
- Three images into the Images folder (two of which need adding to the project)
- *Index.cshtml* into the Views -> Home folder
- *crossdomain.xml* into the root folder

Now running the project should look very different (although the .CSS change doesn't always get picked up if running locally – the background of the "Welcome" area should be orange, but often looks blue before it makes it up to Azure).

There are still some changes needed to allow our service to support “cross domain scripting” (which for us primarily means being callable from client-side HTML5/JavaScript code). The first step was to add the *crossdomain.xml* file but we also need to open s and add these elements:

Inside `<configuration><system.web>`:

```
    <customErrors mode="Off"/>
```

Inside `<configuration><system.webServer>`:

```
    <httpProtocol>
      <customHeaders>
        <add name="Access-Control-Allow-Origin" value="*" />
      </customHeaders>
    </httpProtocol>
```

And then we should expand the serialization limit beyond the default to make sure it’s large enough for our largest JSON string:

```
  <system.web.extensions>
    <scripting>
      <webServices>
        <jsonSerialization maxJsonLength="500000"></jsonSerialization>
      </webServices>
    </scripting>
  </system.web.extensions>
```

The changes should allow our code to be called properly from our HTML5 sample.

We can then update the web-services project (ApollonianPackingWebAppApi).

Copy across the various files into the root project folder:

Global.fs will update the existing file. Here are its contents:

```
namespace FsWeb

open System
open System.Web
open System.Web.Mvc
open System.Web.Routing
open System.Web.Http
open System.Data.Entity
open System.Web.Optimization
open System.Linq
open System.Collections.Generic
open Newtonsoft.Json
open Newtonsoft.Json.Serialization

type OrderedContractResolver() =
  inherit DefaultContractResolver()
```

```
  override x.CreateProperties(tp, ms) =
    (base.CreateProperties(tp, ms).OrderBy
      (fun(p) -> p.PropertyName)).ToList() :> IList<JsonProperty>

type BundleConfig() =
  static member RegisterBundles (bundles:BundleCollection) =
    bundles.Add
      (ScriptBundle("~/bundles/jquery").Include
        ("~/Scripts/jquery-1.*"))
    bundles.Add
      (ScriptBundle("~/bundles/jqueryui").Include
        ("~/Scripts/jquery-ui*"))
    bundles.Add(ScriptBundle("~/bundles/jqueryval").Include
      ("~/Scripts/jquery.unobtrusive*",
       "~/Scripts/jquery.validate*"))
    bundles.Add(ScriptBundle("~/bundles/modernizr").Include
      ("~/Scripts/modernizr-*"))
    bundles.Add(StyleBundle("~/Content/css").Include
      ("~/Content/*.css"))
    bundles.Add
      (StyleBundle("~/Content/themes/base/css").Include
        ("~/Content/themes/base/jquery.ui.core.css",
          "~/Content/themes/base/jquery.ui.resizable.css",
          "~/Content/themes/base/jquery.ui.selectable.css",
          "~/Content/themes/base/jquery.ui.accordion.css",
          "~/Content/themes/base/jquery.ui.autocomplete.css",
          "~/Content/themes/base/jquery.ui.button.css",
          "~/Content/themes/base/jquery.ui.dialog.css",
          "~/Content/themes/base/jquery.ui.slider.css",
          "~/Content/themes/base/jquery.ui.tabs.css",
          "~/Content/themes/base/jquery.ui.datepicker.css",
          "~/Content/themes/base/jquery.ui.progressbar.css",
          "~/Content/themes/base/jquery.ui.theme.css"))

type Route =
  { controller : string
    action : string
    rad : UrlParameter
    steps : UrlParameter }

type ApiRoute =
  { rad : obj
    steps : obj }

type Global() =
  inherit System.Web.HttpApplication()

  static member RegisterGlobalFilters
    (filters:GlobalFilterCollection) =
      filters.Add(new HandleErrorAttribute())

  static member RegisterRoutes(routes:RouteCollection) =
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

    routes.MapHttpRoute(
```

```
      "DefaultApi",
      "api/{controller}/{rad}/{steps}",
      { rad = RouteParameter.Optional
        steps = RouteParameter.Optional }) |> ignore

    routes.MapRoute(
      "Default",
      "{controller}/{action}/{rad}/{steps}",
      { controller = "Home"
        action = "Index"
        rad = UrlParameter.Optional
        steps = UrlParameter.Optional } )

  member this.Start() =

    // Only support JSON, not XML

    let cfg = GlobalConfiguration.Configuration
    cfg.Formatters.Remove(cfg.Formatters.XmlFormatter) |> ignore

    // Order the JSON fields alphabetically

    let stg = new JsonSerializerSettings()
    stg.ContractResolver <-
      new OrderedContractResolver() :> IContractResolver
    cfg.Formatters.JsonFormatter.SerializerSettings <- stg

    AreaRegistration.RegisterAllAreas()
    Global.RegisterRoutes RouteTable.Routes |> ignore
    Global.RegisterGlobalFilters GlobalFilters.Filters
    BundleConfig.RegisterBundles BundleTable.Bundles
```

The other four files need to be added to the project. Two of them, *CirclePackingFull.fs* and *SpherePackingInversion.fs*, are copied directly from the previous AutoCAD-hosted version of the application. The other two, *CirclesController.fs* and *SpheresController.fs*, implement the logic to pass the API requests through to our core algorithm implementations.

Here's *CirclesController.cs*:

```
namespace FsWeb.Controllers

open System
open System.Web.Http

type Circle (X, Y, C, L) =
  member this.X = X
  member this.Y = Y
  member this.C = C
  member this.L = L

type CirclesController() =
  inherit ApiController()

  // GET /api/values/rad/steps
```

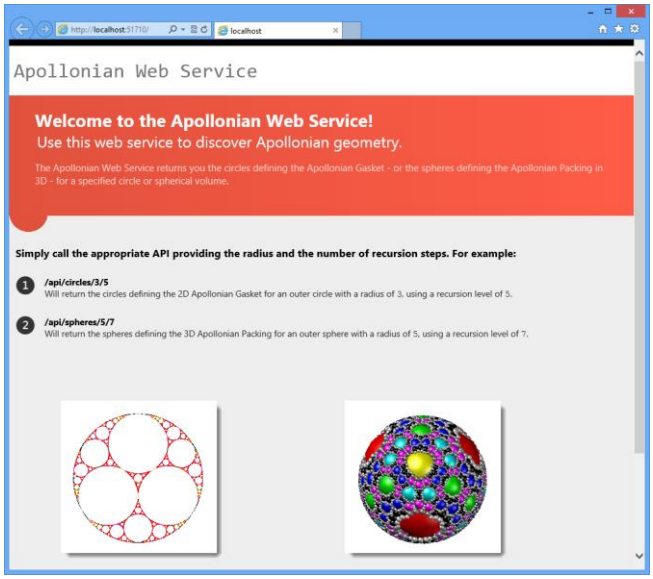
```
  member x.Get(rad:double, steps:int) =
    CirclePackingFullFs.Packer.ApollonianGasket rad steps |>
      List.map
        (fun ((a,b,c),d) ->
          new Circle
            (Math.Round(a, 4),
             Math.Round(b, 4),
             Math.Round(c, 4), d)) |>
      List.toSeq
```

And here's *SpheresController.cs*:

```
namespace FsWeb.Controllers

open System
open System.Web.Http

type Sphere (X, Y, Z, R, L) =
  member this.X = X
  member this.Y = Y
  member this.Z = Z
  member this.R = R
  member this.L = L

type SpheresController() =
  inherit ApiController()

  // GET /api/values/rad/steps

  member x.Get(rad:double, steps:int) =
    SpherePackingInversionFs.Packer.ApollonianGasket
      steps 0.01 false |>
      List.map
        (fun ((a,b,c,d),e) ->
          new Sphere
            (Math.Round(a * rad, 4),
             Math.Round(b * rad, 4),
             Math.Round(c * rad, 4),
             Math.Round(d * rad, 4),
             e))
         |> List.toSeq
```

You can safely remove *ValuesController.fs* from the project (deleting it from disk, should you so wish).

To test our web-site and -service, we first want to launch it in a browser (most easily via the debugger):



With the web-site loaded, we can then add these URL suffices into the browser to test the two APIs:

- /api/circles/2/2
- /api/spheres/2/2

The first number in each of these URLs specifies the desired radius of the outer circle/sphere to be packed while the second tells the recursion level: how “deep” the fractal should go.

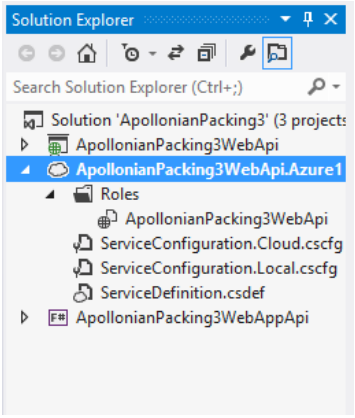
Here are the results of the first of these calls. It should be easy enough to see how the JSON file contains a list of circle definitions, each with X, Y, Curvature and Level values:


```
[{"C":1.0774,"L":2,"X":2,"Y":3.0718},{"C":1.0774,"L":2,"X":2.9282,"Y":
1.4641},{"C":1.0774,"L":2,"X":1.0718,"Y":1.4641},{"C":6.9641,"L":1,"X"
:2,"Y":2},{"C":17.1603,"L":0,"X":2.1748,"Y":2.1009},{"C":17.1603,"L":0
,"X":2,"Y":1.7981},{"C":17.1603,"L":0,"X":1.8252,"Y":2.1009},{"C":2.23
21,"L":0,"X":3.3441,"Y":2.776},{"C":2.2321,"L":0,"X":2,"Y":0.448},{"C"
:2.2321,"L":0,"X":0.6559,"Y":2.776}]
```


Assuming both web-service calls return valid JSON files, we are now ready to publish to Azure.

Preparing to publish to Azure

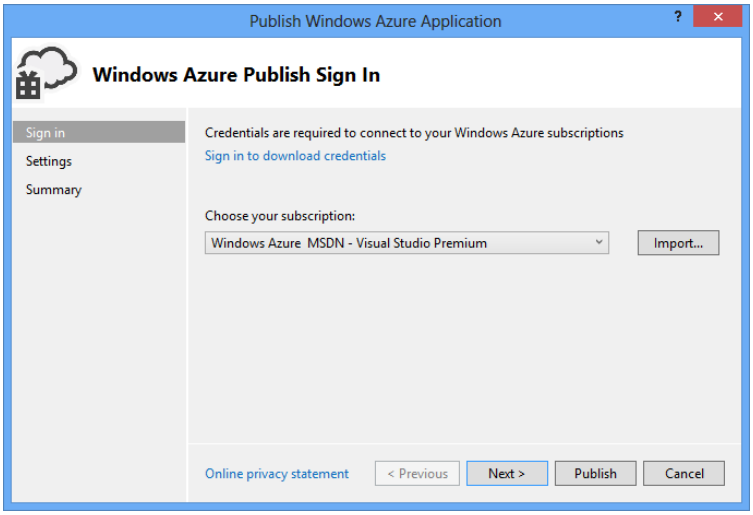
Now that we're ready to publish to Azure, we need to add a deployment project to our solution. Right-click "ApollonianPackingWebApi" in the Solution Explorer and select "Add Windows Azure Cloud Service Project". This will add a new project into our solution:



We can now double-click the entry under the "Roles" folder in order to adjust the parameters for that role.

It's here that we can adjust the number and of size of the instances to deploy to Azure, as well as more advanced settings related to Virtual Networks and Caching.

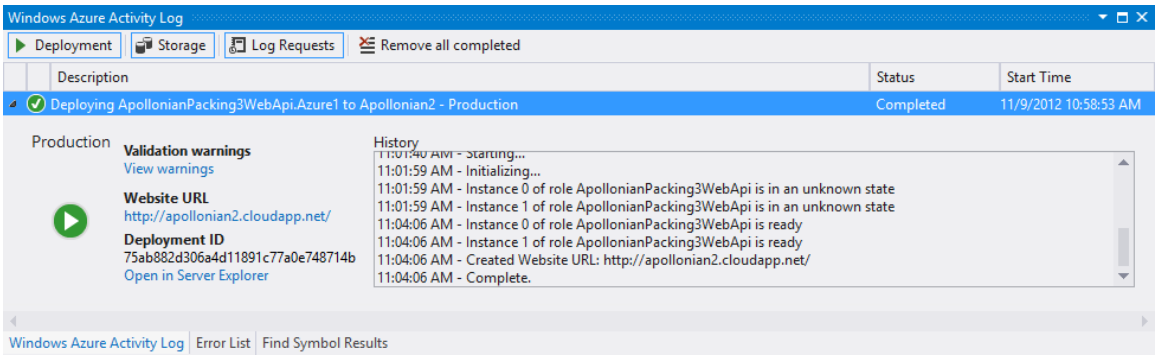
Then we can right-click on the newly added project and select "Publish…".



We need to sign in to MSDN in order to get our credentials – these get downloaded to your local system in a .publish file – and after selecting "Next" we can add a new cloud service in our preferred location, as well as choosing whether to post to staging or production (we'll be lazy

and go straight to production) and specifying remote desktop settings in case we want to connect to the VM instance hosting our role (sometimes needed in case of debugging).

The actual deployment process can take some time (~5 minutes or so), at which point we should see a “completed” message inside Visual Studio:



Now the site will be ready for testing at the URL you assigned to your cloud service, e.g.:

http://apollonian2.cloudapp.net

And, of course, the web-services, too:

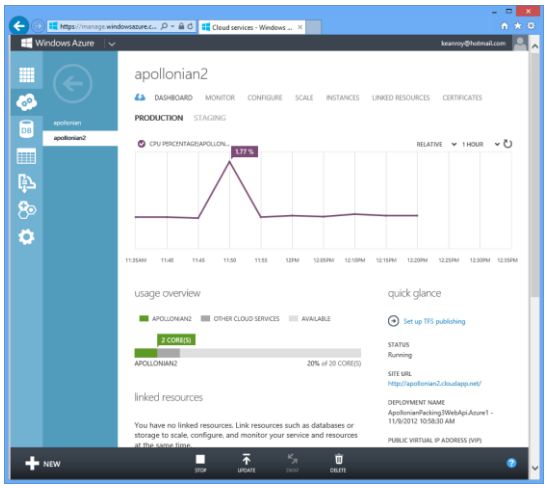
http://apollonian2.cloudapp.net/api/circles/2/2

http://apollonian2.cloudapp.net/api/spheres/2/2

So that’s all there is to it – we now have a functioning, cloud-based web-site and -service.

To get information on the web-service’s status – including its usage, cost & billing information – you can log into the Windows Azure Management Console:

http://windows.azure.com



Calling our web-service from anywhere*

* AutoCAD, Android, iOS, WinRT, HTML5 & Unity3D.

In this section, we'll take a whirlwind tour of some different client environments. Let's start by revisiting AutoCAD, looking at some C# code that calls into our web-service rather than the local F# code.

At the core of this implementation we need some HTTP-related code to call the web-service and some JSON-related to code to parse the results.

Here's the function we'll use to call the web-service:

```
private static dynamic ApollonianPackingWs(
  Editor ed, double p, int numSteps, bool circles
)
{
  string json = null;

  // Call our web-service synchronously (this isn't ideal, as
  // it blocks the UI thread)

  HttpWebRequest request =
    WebRequest.Create(
      "http://apollonian.cloudapp.net/api/" +
      (circles ? "circles" : "spheres") +
      "/" + p.ToString() +
      "/" + numSteps.ToString()
    ) as HttpWebRequest;

  // Get the response

  try
  {
    using (
      HttpWebResponse response =
        request.GetResponse() as HttpWebResponse
      )
    {
      // Get the response stream

      StreamReader reader =
        new StreamReader(response.GetResponseStream());

      // Extract our JSON results

      json = reader.ReadToEnd();
    }
  }
```

```
    catch (System.Exception ex)
    {
      ed.WriteMessage(
        "\nCannot access web-service: {0}", ex.Message
      );
    }

    if (!String.IsNullOrEmpty(json))
    {
      // Use our dynamic JSON converter to populate/return
      // our list of results

      var serializer = new JavaScriptSerializer();
      serializer.RegisterConverters(
        new[] { new DynamicJsonConverter() }
      );

      // We need to make sure we have enough space for our JSON,
      // as the default limit may well be exceeded

      serializer.MaxJsonLength = 50000000;

      return serializer.Deserialize(json, typeof(List<object>));
    }
    return null;
  }
```

There's really not a great deal to it, although there's a little work going on to deserialize the JSON returned. If we were targeting .NET 4.5 rather than 4.0, we could make use of some new capabilities in the .NET Framework to parse JSON, but this version makes use of a 3rd party JSON serializer (from here: http://www.drowningintechnicaldebt.com/ShawnWeisfeld/archive/2010/08/22/using-c-4.0-and-dynamic-to-parse-json.aspx). One thing I liked about this particular implementation was its use of .NET 4.0's dynamic keyword to simplify parsing the JSON. This capability has apparently now been added to the Json.NET (http://json.codeplex.com) library, so if I was starting again I might possibly choose that, instead (I've used it successfully on other projects calling web-services).

Our main implementation – excluding the code requesting data from the user – now becomes:

```
    Transaction tr =
      doc.TransactionManager.StartTransaction();
    using (tr)
    {
      // Start by creating layers for each step/level

      Utils.CreateLayers(db, tr);

      // We created our Apollonian gasket in the current space,
      // for our 3D version we'll make sure it's in modelspace

      BlockTable bt =
        (BlockTable)tr.GetObject(
```

```
    db.BlockTableId, OpenMode.ForRead
  );
BlockTableRecord btr =
  (BlockTableRecord)tr.GetObject(
    bt[BlockTableRecord.ModelSpace], OpenMode.ForWrite
  );

// Let's time the WS operation

Stopwatch sw = Stopwatch.StartNew();
dynamic res = ApollonianPackingWs(ed, radius, steps, false);
sw.Stop();

if (res == null)
  return;

ed.WriteMessage(
   "\nWeb service call took {0} seconds.",
   sw.Elapsed.TotalSeconds
 );

// Go through our "dynamic" list, accessing each property
// dynamically

foreach (dynamic tup in res)
{
  double rad = System.Math.Abs((double)tup.R);
  if (rad > 0.0)
  {
    Solid3d s = new Solid3d();
    s.CreateSphere(rad);
    Point3d cen =
      new Point3d(
        (double)tup.X, (double)tup.Y, (double)tup.Z
      );
    Vector3d disp = cen - Point3d.Origin;
    s.TransformBy(Matrix3d.Displacement(disp + offset));

    // The Layer (and therefore the colour) will be based
    // on the "level" of each sphere

    s.Layer = tup.L.ToString();

    btr.AppendEntity(s);
    tr.AddNewlyCreatedDBObject(s, true);
  }
}
tr.Commit();

ed.WriteMessage(
   "\nCreated {0} spheres.", res.Count
 );
```

As you'd expect, the code works in a very similar fashion to the previous implementation (it's the same code doing the work, just in a geographically different location :-).

Supporting multiple platforms

We'll now take a look at some options for creating viewers of our 3D data on different platforms.

We'll start by looking at some native clients for Android, iOS and Windows 8, before looking at one cross-platform toolkit (Unity3D) and then HTML5 (via WebGL).

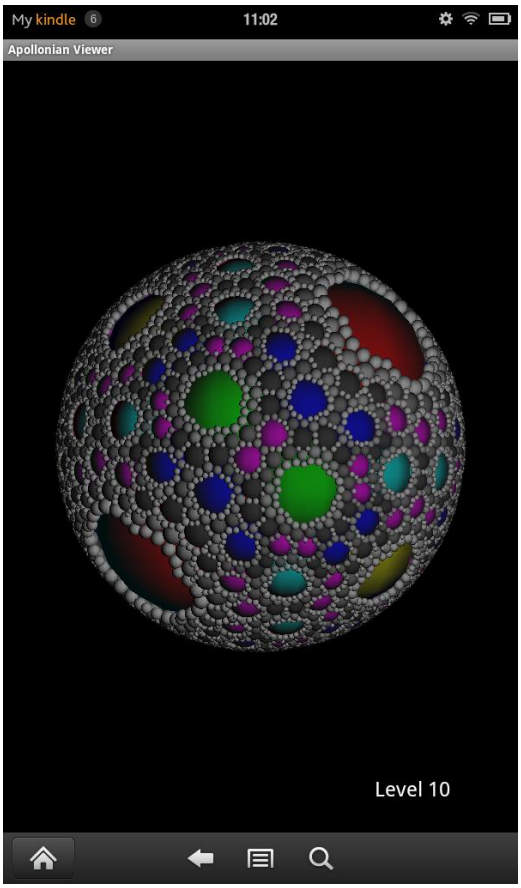
If you're interested in continuing the discussion, there's a conveniently timed round-table session following on after this session (hosted by me):

- CP4342-R – Cloud and mobile developer round-table)

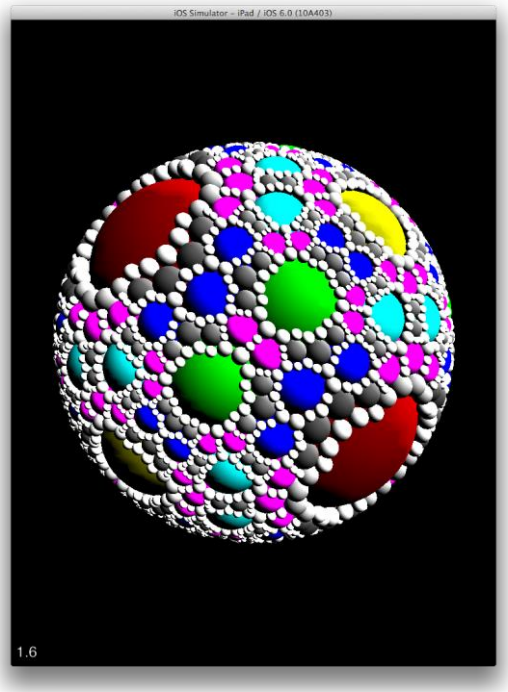
Apollonian Viewer for Android

The Android stack is Java-based, which made it a very familiar environment (at least with respect to the code created) for a C# developer such as myself. The tooling was a bit different – the IDE I used was Eclipse – but I was surprised to find out how much I ended up enjoying it: there are a few quirks, but there are also capabilities I'd really like to see in Visual Studio, such as warnings regarding unused namespaces.

The 3D object library – to avoid making low-level calls into OpenGL ES 2.0 – was named Rajawali. An open source toolkit developed by Dennis Ippel in the UK, who provided me with some great support (and even custom features) during the implementation. And the results were very impressive.



Apollonian Viewer for iOS



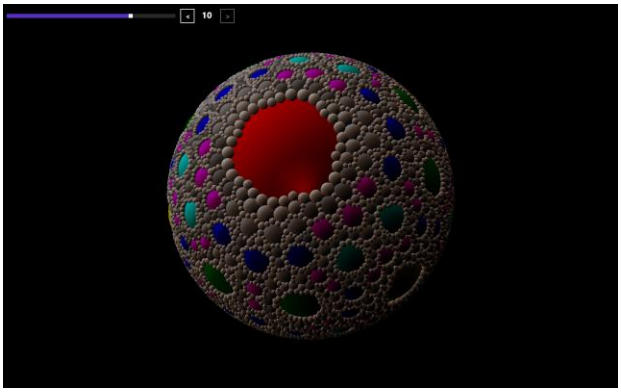
Now I fully admit I have trouble with Objective-C. I understand its syntax is dictated largely by its origins (it was apparently inspired by Smalltalk, back in the day), but I have become so very used to a traditional structure for calling methods (object.method(arg1,arg2)) that I find Objective-C very difficult to adjust to.

That said, it does appear to be a very powerful, capable and well-loved language. It's just not something In enjoyed working with closely, myself.

The development environment was Xcode, which I found to be convoluted and unstable, but I assume this is something you get used to.

The 3D object library was iSGL3D – I also looked into a few others but found this to be the best fit, overall. Once again it was based on OpenGL ES 2.0, but the similarities to Rajawali ended there: I found it difficult to get good results (perhaps because I didn't have direct help from the developer and struggled with Objective-C, in general).

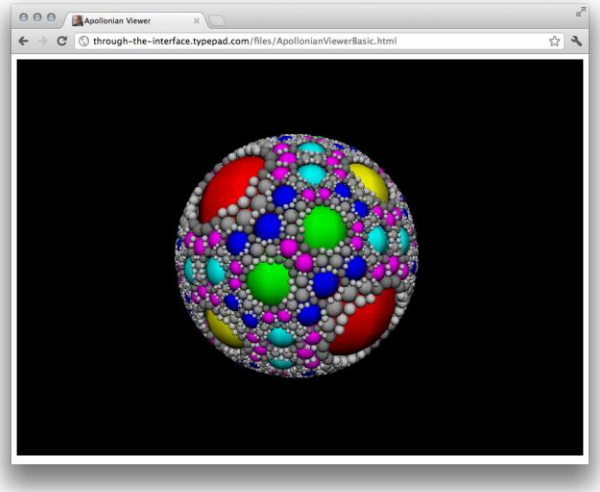
Apollonian Viewer for WinRT



Developing a WinRT-focused client – one that works a Windows Store application on Windows 8 – was interesting. I had already done some work on WinRT, but writing a 3D viewer meant diving into DirectX. In theory you're supposed to call this from C++ but I ended up wimping out and making use of SharpDX to bridge to it from C#.

Development was done in VS2012 (which was just fine), but because – at least at the time of writing – I couldn't find a decent 3D object library, I had to get down and dirty and write my own pixel and vertex shaders. A pretty painful experience – I had to learn a great deal about rendering pipelines that I've since forgotten – but the results were pretty impressive.

Apollonian Viewer for HTML5



In order to understand the possibilities around HTML5 – which many people believe to be the future for cross-platform development – I decided to dive in and create a WebGL-based viewer for our data.

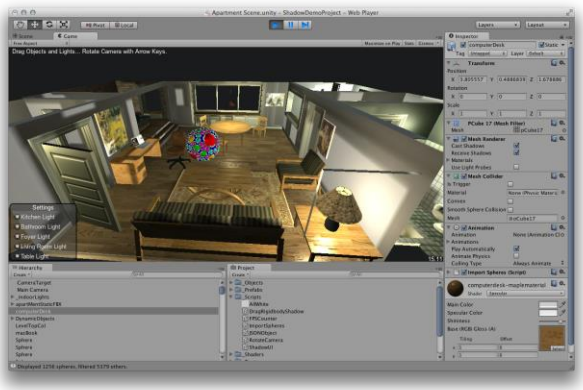
HTML5 means JavaScript – which I like better than Objective-C, although it's far from being my favorite language – but I did find the tools available to me had evolved considerably since I last used it in earnest: modern browsers have pretty advance debugging capabilities and even Visual Studio does a pretty good job with the language.

I chose Three.js as the 3D object library, and found it excellent: the results were impressive. It worked well from most browsers – although had to fall back to "canvas" rendering in IE, which was a shame. WebGL is hardware accelerated, so the performance was great.

Viewing an apollonian packing in a Unity3D scene

And finally, let's take a look at Unity3D. I had really wanted to try out this toolkit: I'd heard great things about from a number of people and also felt it wouldn't be fair to only focus on native apps and HTML5 without taking a look at it.

I haven't written the title of this section as if I had written a new viewer, although some people have used Unity3D to do just this: I merely wanted to make use of a standard Unity3D scene and add the results of our web-service call into it.



I used Unity3D on OS X (and also had a play with the Windows version) and was able to target a number of environments for free (desktop environments and the web). You can pay to be able to target additional platforms – the list is impressive.

From a programming perspective it was easy: I used C# - Unity3D uses Mono to make this work when not on Windows – but might also have chosen JavaScript. The development environment – when I needed to work on code outside the Unity scene editor – was MonoDevelop: a fairly decent IDE.

Summary

To summarize what we've seen in this session…

- We extracted some F# code from an existing AutoCAD application
- We placed it behind a web-service implemented using ASP.NET MVC4
- We published the web-service to Windows Azure
- We then modified the AutoCAD client to call the web-service
- We then saw how we could also use the data from…
    - Native apps: Android, iOS, WinRT
    - Web apps: HTML5 & WebGL
    - Cross-platform apps: Unity3D

Overall the experience of creating the web-service was straightforward, although admittedly we kept things really simple: if we'd chosen to implement authentication life would have been at least marginally more interesting. Posting to Azure and managing the deployment was also made very easy by the integrated and standalone tools.

It was fun to do some native development, to understand what's involved (although I enjoyed Android, WinRT and iOS in that order, I would say). On balance, though, I expect HTML5 to come into its own – even on mobile devices – over the coming year, and if you need to support multiple platforms it's well worth investigating the cross-platform tools that meet your requirements.

Want to continue the discussion? Come along to the round-table in 30 minutes time!

Blog References

Cloud & mobile series summary

Circle packing in AutoCAD: creating an Apollonian gasket using F# – Part 1

Circle packing in AutoCAD: creating an Apollonian gasket using F# – Part 2

Sphere packing in AutoCAD: creating an Apollonian packing using F# – Part 1

Sphere packing in AutoCAD: creating an Apollonian packing using F# – Part 2

Moving to the Cloud

Exposing a RESTful web service for use inside AutoCAD using the ASP.NET Web API – Part 1

Exposing a RESTful web service for use inside AutoCAD using the ASP.NET Web API – Part 2

Architecting for the Cloud

Consuming data from a RESTful web-service inside AutoCAD using .NET

Hosting our ASP.NET Web API project on Windows Azure – Part 1

Hosting our ASP.NET Web API project on Windows Azure – Part 2

Using Windows Azure Caching with our ASP.NET Web API project

Calling a cloud-based web-service from AutoCAD using .NET

Calling a web-service from a Unity3D scene

Creating a 3D viewer for our Apollonian service using Android – Part 1

Creating a 3D viewer for our Apollonian service using Android – Part 2

Creating a 3D viewer for our Apollonian service using Android – Part 3

Creating a 3D viewer for our Apollonian service using iOS – Part 1

Creating a 3D viewer for our Apollonian service using iOS – Part 2

Creating a 3D viewer for our Apollonian service using iOS – Part 3

Creating a 3D viewer for our Apollonian service using HTML5 – Part 1

Creating a 3D viewer for our Apollonian service using HTML5 – Part 2

Creating a 3D viewer for our Apollonian service using HTML5 – Part 3

Creating a 3D viewer for our Apollonian service using WinRT – Part 1

Creating a 3D viewer for our Apollonian service using WinRT – Part 2