Workshop Walkthrough

Prerequisites

- Visual Studio. 2010, 2012 or 2013 should all work with the examples here.
- .NET Framework 4.0
- A working internet connection

That's it. Everything else we're using we'll install via NuGet during the course of the workshop.

The Code on GitHub

The complete code for each section is on GitHub. There are tagged releases corresponding to the material in each section, and they're all available here:

https://github.com/dylanbeattie/HeyStack/releases

I'd recommend you try implementing the features in each section yourself, but if you can't get a particular feature to work or you're running out of time, just grab the Git source when we start the next section and you won't get left behind.

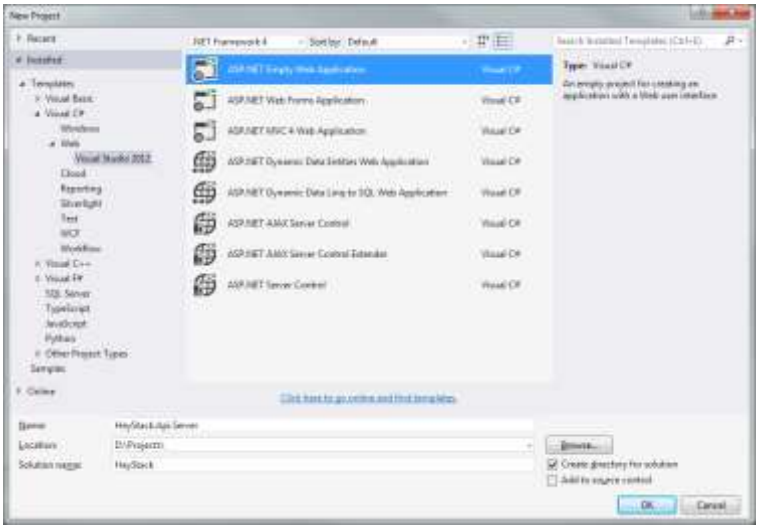
Part 1: Your First ServiceStack Service

During this workshop, we're going to build a really simple movie database. Everyone loves movies, right? But first, we're going to get up and running with a basic ServiceStack architecture.

> As a curious developer, I want to build my first ServiceStack service, so that I can prove the architecture works and begin developing real services using the same pattern.

Create a new Visual Studio project

Pick a name - I've called my sample application HeyStack, so for this step I created a new web application called `HeyStack.Api.Server`, in a new solution I just called `HeyStack`.



Because we'll be using ServiceStack to handle all our routing and HTTP concerns, we just want an **Empty ASP.NET Web Application** – if you're using VS2013 it might be hidden under the "Visual Studio 2012" submenu.[1]

Aside: at this point, I always shut down Visual Studio, create a \src subfolder, move all the source code into that folder, and then re-open it. It'll keep things cleaner if you start adding things like SQL scripts, artwork, documentation, etc. to your project.

Install the ServiceStack NuGet packages

ServiceStack is on NuGet – use the GUI to install the **ServiceStack** package, or run `Install-Package ServiceStack` from the NuGet package manager console.

Register the ServiceStack handler

Before ServiceStack will handle your HTTP requests, you'll need to register the handler in web.config as follows.[2]

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false" />
  <handlers>
    <add path="*" name="ServiceStack.Factory" preCondition="integratedMode"
         type="ServiceStack.HttpHandlerFactory, ServiceStack"
         verb="*" resourceType="Unspecified" allowPathInfo="true" />
  </handlers>
</system.webServer>
```

Add a global application class

Since we've got an empty project, we don't even have a global application class yet – so right-click your project, Add Item, and add the Global Application Class

(Once it's in, open `Global.asax.cs` remove all the methods except `Application_Start` – we won't need them.)

[1] And remember to use C# instead of Visual Basic or you'll get get sore hands from All That Ridiculously Verbose Typing That Visual Basic Loves So Much.
[2] This is for IIS7+; for hosting on IIS6 or Mono check out https://github.com/ServiceStack/ServiceStack/wiki/Create-your-first-webservice

Create your first web service

OK, this is where ServiceStack really gets going. Every service in your ServiceStack app consists of three parts:

1. A request DTO – what's the client asking for?
2. A service implementation, that takes a request DTO and returns…
3. A response DTO – what do we want to send back?

We're going to build a Status service, that clients can use to ping our application and find out what our service status is. (Pretty useful, huh?) In a new folder `\Services` in your application, create the following classes

```
[Route("/status")]
public class GetStatusDto {
    /* this class intentionally left blank */
}

public class StatusResultDto {
    public string Status { get; set; }
}

public class StatusService : Service {
    public StatusResultDto Get(GetStatusDto request) {
        var message = String.Format("{0} at {1} is OK",
            Environment.MachineName, DateTime.Now);
        return new StatusResult { Status = message };
    }
}
```

Wire it all together

There's two things left to do. We need to tell ServiceStack about our services, and then we need to tell ASP.NET about ServiceStack.

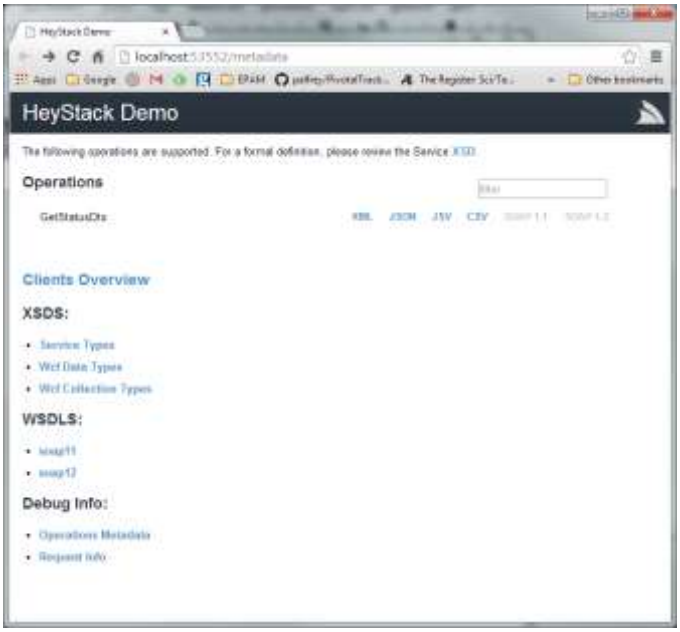
Add a class `AppHost` inside your Global application class (yes, it's a nested class - this is fine). AppHost inherits from ServiceStack's **AppHostBase** class, and is responsible for configuring and initializing our services. And then add a line to `Application_Start` that will spin up a new `AppHost` and call `Init()`.

You'll end up with something like this:

```
public class Global : HttpApplication {

    protected void Application_Start(object sender, EventArgs e) {
        new AppHost().Init();
    }

    public class AppHost : AppHostBase {
        // The name of your app and where to find your services
        public AppHost() : base("HeyStack Demo", typeof(StatusService).Assembly){}

        public override void Configure(Funq.Container container) {
            //register any dependencies your services use, e.g:
            //container.Register<ICacheClient>(new MemoryCacheClient());
        }
    }
}
```

Go Go Gadget Status!

Hit F5, and – if everything's wired up properly – you'll get the ServiceStack service landing page, listing your services and supported encoding formats:



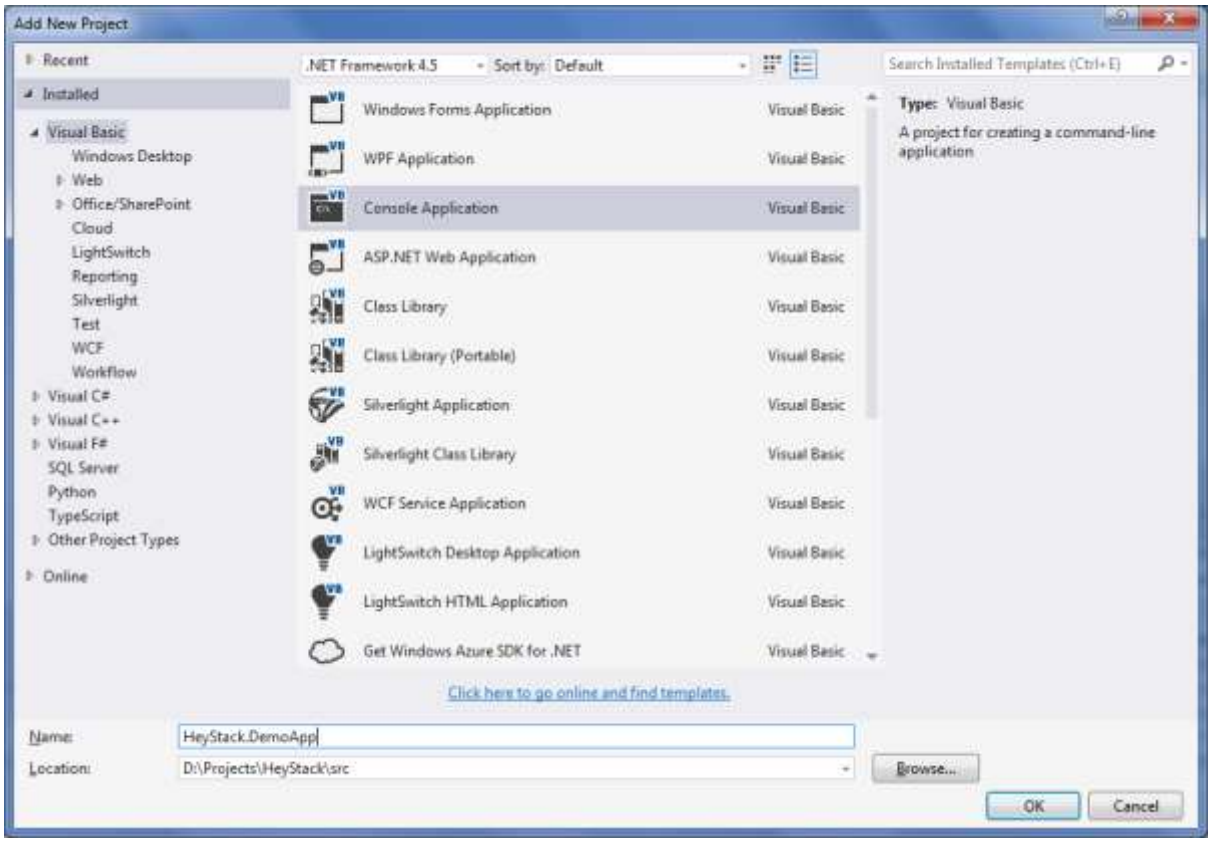
You can now hit your new service endpoint by making an HTTP request to /status. To see your status response in different formats, either pass `?format=json` or `?format=xml` in the query string, or set an HTTP `Accept:` header on your request.

Part 2: An API Client Library

OK, we've now got an API that returns some useful information. Now we want to create a strongly-typed .NET client API, so that anybody who wants access to your data can just install a client library DLL and away they go.

> As an overworked developer, I want to share my data and services via a client API, so that I can get other people to build my apps for me.

To do this – and prove that it works – we're going to build a .NET console application application that uses our ServiceStack server and our new API client. Add a new .NET Console Application to your solution, called HeyStack.DemoApp:



Build it, right-click and "Set as Startup Project" so Visual Studio will run your app (rather than the API server) when we hit F5. Throw a couple of lines into your Program.Main() method just to verify it's working:

```
namespace HeyStack.DemoApp {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hello World");
            Console.ReadKey(false);
        }
    }
}
```

Next, we're going to restructure our project a bit, so we can create an API client and plug it into our new web app. Create a new class library projects – HeyStack.ServiceModel. This will be the shared library where we define our DTOs, interfaces and other features that will be shared by both our client and our API server. For starters, create a /Status folder in your Common project, and move the GetStatusDto and StatusResultDto classes out of the Api.Server project and into this new folder. Remember to fix the namespaces afterwards.

You'll need to add a couple of NuGet package references. Add **ServiceStack.Interfaces** to your **ServiceModel** project, and add **ServiceStack.Client** to your **DemoApp** console application. You'll also need to create project references as follows:

- HeyStack.Api.Server needs a reference to HeyStack.ServiceModel
- HeyStack.DemoApp needs a reference to HeyStack.ServiceModel[3]

Configure your solution so that it'll start multiple projects when you run it – right-click your solution, Set Startup Projects, Multiple Startup Projects, and set it to run your API server and your DemoApp console application.

Now grab the URL that your API server will listen on (Right-click the API.Server project, properties, Web in Visual Studio), and modify your Program.cs in your DemoApp to look like this – replacing **http://localhost:58027/** with the URL of your own API server:

```
using HeyStack.ServiceModel.Status;
using ServiceStack;
using System;

namespace HeyStack.DemoApp {
    class Program {
        private const string ServiceUri = "http://localhost:58027/";
        static void Main(string[] args) {
            var client = new JsonServiceClient(ServiceUri);
            Console.WriteLine("Retrieving status from {0}", ServiceUri);
            var status = client.Get<StatusResultDto>(new GetStatusDto());
            Console.WriteLine(status.Status);
            Console.ReadKey(false);
        }
    }
}
```

There you go. We've got a ReSTful web service, and a .NET console app using a strongly-typed API client to connect to it, request a status, deserialize the result from JSON and print it to the console. Not bad for 20-odd lines of code, huh?

Extra Credit…

- How would you modify your app to use XML serialization instead of JSON?

[3] You could, of course, put HeyStack.ServiceModel in a NuGet package to make life even easier for your users.

Part 3: IOC and Dependency Injection with Funq

> As a QA Manager, I want the developers to make their code easily testable, so I can create automated tests to improve confidence in the quality of our product.

OK, next thing we're going to do is to test our services, but before we can do that, we need to isolate some of our dependencies. ServiceStack includes the Funq IoC container[4], so we're going to configure Funq and wire up our services accordingly.

Our Status service is returning a machine name and the current date/time – both things that'll we can't easily control, so let's isolate these behind a couple of simple service interfaces:

```
public interface IHost {
    /// <summary>Returns the name of the machine that's hosting this code.</summary>
    string MachineName { get; }
}

public interface IClock {
    /// <summary>Returns the current local date and time.</summary>
    DateTime Now { get; }
}
```

We'll need some implementations for these, of course:

```
public class MyClock : IClock {
    public DateTime Now { get { return DateTime.Now; } }
}

public class MyHost : IHost {
    public string MachineName { get { return Environment.MachineName; } }
}
```

and we'll need to modify our StatusService so that it calls our new services:

```
public class StatusService : Service {
    private readonly IHost host;
    private readonly IClock clock;

    public StatusService(IHost host, IClock clock) {
        this.host = host;
        this.clock = clock;
    }

    public StatusResultDto Get(GetStatusDto request) {
        var message = String.Format("{0} at {1} is OK",
            host.MachineName, clock.Now);
        return new StatusResultDto { Status = message };
    }
}
```

[4] Funq is a fast, lightweight container written by Daniel Cazzulino (@kzu), the same person who wrote Moq –if you want to know more, there's a great series of screencasts showing how Funq was created at http://blogs.clariusconsulting.net/kzu/funq-screencast-series-on-how-to-building-a-di-container-using-tdd/

Now we need to wire up our new services. By default, ServiceStack uses the Funq IoC container to resolve service dependencies, so open up your global.asax.cs file and find the Configure method on your AppHost, and register the two services:

```
public class AppHost : AppHostBase {
    // The name of your app and where to find your services
    public AppHost() : base("HeyStack Demo", typeof(StatusService).Assembly) { }

    public override void Configure(Funq.Container container) {
        container.Register<IHost>(c => new MyHost());
        container.Register<IClock>(c => new MyClock());
    }
}
```

That's it. Hit F5, run your project, and you should get… well, exactly what you had a moment ago. But now we've restructured things, we can start putting some tests around our services.

Part 4: Unit Testing your API

> As a developer who's fed up of getting yelled at by the QA Manager, I want to build some automated tests around my services, so that I can be confident I haven't accidentally broken anything that used to work.

Create a new class library project – HeyStack.Api.Server.UnitTests – and add the following NuGet packages to it:

- NUnit
- Moq
- ServiceStack
- Shouldly

Create a new class StatusServiceTests. We're going to mock the service dependencies so we can test our status service logic, so plug in a test method:

```
[Test]
public void Status_Service_Returns_HostName() {
    var mockHost = new Mock<IHost>();
    mockHost.Setup(h => h.MachineName).Returns("TESTHOST");
    var mockClock = new Mock<IClock>();
    var service = new StatusService(mockHost.Object, mockClock.Object);
    var result = service.Get(new GetStatusDto());
    result.Status.ShouldStartWith("TESTHOST at ");
}
```

Fire up your test runner of choice – vanilla NUnit will do just fine if you're not using Resharper or NCrunch – and verify that your test passes.

Create a second test using the same pattern, but this time test for the date/time being returned correctly by the status service. This should highlight some design flaws with our solution so far… shout when you've worked out what they are ☺

Part 5: Integration Tests

> As a QA manager who's tired of finding bugs even when all the unit tests are green, I want to introduce some higher-level test coverage, so I can be confident we've assembled our tested components into a working system.

One of the great advantages of ServiceStack is that it's very easy to host a 'real' API instance within a test suite. Our unit test above verifies the logic of our StatusService, but they wouldn't highlight any problems with serialization, request binding, request and response filters – so we're going to build a second set of tests, that run at a higher level and actually test the whole request-response cycle.

Add another test project – **HeyStack.Api.Server.IntegrationTests** – and, as before, we're going to add **Moq**, **NUnit**, **ServiceStack** and **Shouldly** from NuGet.

Now we're going to spin up two classes. First, we're going to create a **TestableAppHost**.

```
/// <summary>The testable app host will host the same services as our 'real' server
/// but with mocked dependencies injected via a different container config.</summary>
public class TestableAppHost : AppSelfHostBase {
    private readonly Action<Container> configure;

    public TestableAppHost(Action<Container> configure)
        : base("HeyStack Test Service", typeof(StatusService).Assembly) {
        this.configure = configure;
    }

    public override void Configure(Container container) {
        configure(container);
    }
}
```

This is functionally similar to the AppHost inside our GlobalApplication class, but with two key differences. It inherits from **AppSelfHostBase** instead of **AppHostBase**, and we're going to initialise our IoC container by passing in an **Action<Container>** to give us finer-grained control over our service resolution.

Next, we're going to write a test class that uses **TestableAppHost** to run tests against our **StatusService**.

```
public class StatusServiceTests {

    private const string TEST_URL = "http://localhost:1337/";
    private Mock<IHost> mockHost;
    private Mock<IClock> mockClock;

    private TestableAppHost host;

    [TestFixtureSetUp]
    public void TestFixtureSetUp() {
        mockHost = new Mock<IHost>();
        mockClock = new Mock<IClock>();
        host = new TestableAppHost(container => {
            container.Register(mockHost.Object);
            container.Register(mockClock.Object);
        });
        host.Init();
        host.Start(TEST_URL);
    }

    [TestFixtureTearDown]
    public void TestFixtureTearDown() {
        host.Stop();
    }

    [Test]
    public void StatusService_Returns_HostName_Via_Json() {
        mockHost.Setup(mock => mock.MachineName).Returns("TESTHOST");
        var client = new JsonServiceClient(TEST_URL);
        var status = client.Get<StatusResultDto>(new GetStatusDto());
        status.Status.ShouldStartWith("TESTHOST");
    }
}
```

Note here that the TestFixtureTearDown is important – if you don't stop your host, it'll just keep running even when your tests have completed; then the next test run won't be able to bind to port 1337 and all your tests will fail.

Extra Credit

How would you create an integration test to use XML serialization instead of JSON?

If you follow this pattern, you're going to end up with very similar sets of test coverage around your (local) unit tests, your JSON test and your XML tests. How could you refactor your solution to eliminate this repetition?

Part 6: Documenting your API

> As a developer who's using the HeyStack API, I'd like comprehensive and up-to-date API documentation so I know what services and resources are available via the HeyStack API.

ServiceStack supports Swagger, a "simple, open standard for describing ReSTful APIs"[5] To get started, install the NuGet package **ServiceStack.Api.Swagger** into your HeyStack.Api.Server project. You'll need to edit your AppHost (inside Global.asax.cs) to enable the Swagger plug-in:

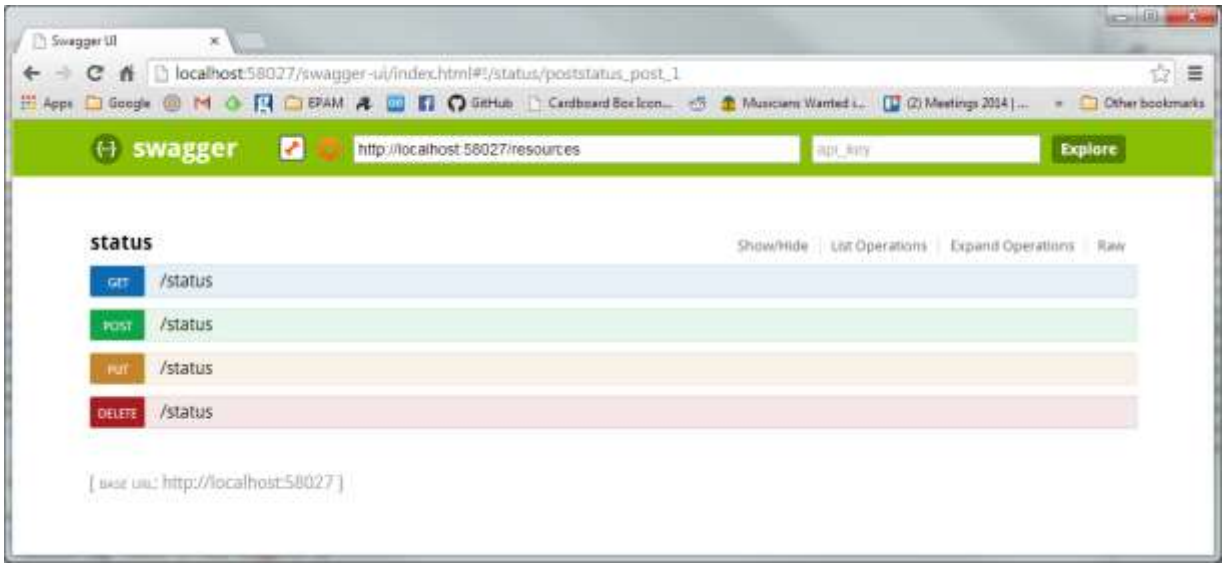
```
public override void Configure(Funq.Container container) {

    Plugins.Add(new SwaggerFeature());

    ...

}
```

Once that's enabled, you should be able to browse to http://localhost:54321/resources and see a JSON representation of your API interface. Next, open up index.html in your new **/swagger-ui** folder and make a couple of changes:

Line 22, change the url: to "/resources"

```
<script type="text/javascript">
    $(function () {
      window.swaggerUi = new SwaggerUi({
      url: "/resources",
      dom_id: "swagger-ui-container",
      ...
</script>
```

Now run your API server project, point your browser at http://localhost:54321/swagger-ui/index.html and you should see something like:



[5] https://github.com/wordnik/swagger-spec

That's very pretty, but it's not actually accurate – it's documented POST, PUT and DELETE methods (which our API doesn't support), and there's no explanation of what the status service is for or why you'd want to use it.

Decorating your Service with Swagger Attributes

To provide richer API documentation, we can decorate our request DTOs with attributes describing their supported verbs, response codes, and a description of the API methods they support.

```
using System.Net;
using ServiceStack;

namespace HeyStack.ServiceModel.Status {
    [Api("ServiceStack Demonstration API")]
    [ApiResponse(HttpStatusCode.OK, "OK")]
    [ApiResponse(HttpStatusCode.InternalServerError,
        "Internal server error - something went wrong.")]
    [Route("/status", "GET",
        Summary = "Return information about the status of the API server",
        Notes = @"This service will report on the status of the server that's
            hosting our API, returning the hostname and the server's local date
            and time."
    )]
    public class GetStatusDto {
        /* this class intentionally left blank */
    }
}
```

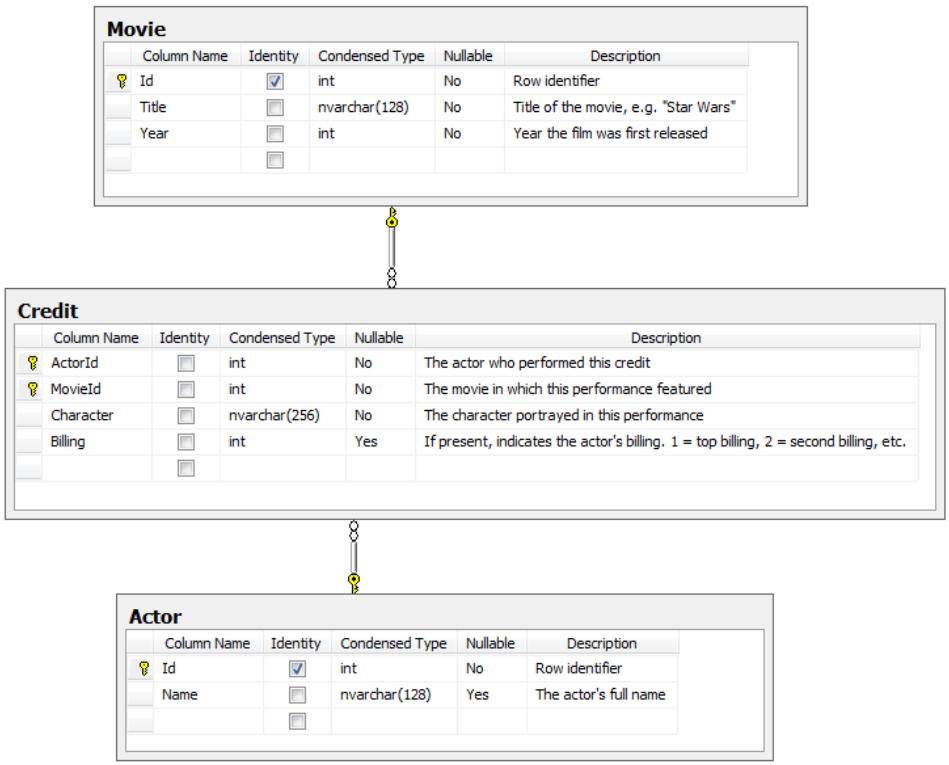
After adding these additional attributes, your Swagger API documentation page will look something like this:

Part 7: Talking to Databases with OrmLite

> As a third-party developer who's getting a bit bored of writing proofs-of-concept based on the 'status' API, I'd love it if HeyStack exposed some real data, so that I can create useful products based on it.

Right, let's get our API serving some real data.

For this workshop, I've set up a hosted SQL Azure database hosting some simple data about movie credits – films, actors, and roles they've played.



To connect to this database, use the following connection string:

```
Server=tcp:notytb9g7e.database.windows.net,1433;
Database=HeyStack;
User ID=heystack@notytb9g7e;
Password= ProgNet2014@SkillsMatter;
Trusted_Connection=False;
Encrypt=True;
Connection Timeout=30;
```

Yes, you're all sharing the same database. Be nice. ☺ (If you'd rather use your own local copy, a SQL file containing the schema and data is in the sql folder of the GitHub project.)

Install the NuGet Package **ServiceStack.OrmLite.SqlServer** into your HeyStack.Api.Server project.

API design – resources and methods

Our first API release is going to expose the following method/resource operations:

Method	URL	Comments
GET	/movies	Returns list of movies
POST	/movies	Add a new movie to the database; return a redirect to the new movie

GET /movies

We're going to need:

- A GetMoviesDto request object
- A MovieService
- A MovieResult
- A MovieListResult, that represents a collection of MovieResults

ReST discussion point – could we just use an IList<MovieResult>? Do we really need a first-class MovieListResult? Why?

- A MovieDatabase – something that wraps basic CRUD operations. With ReST, it's rare that you'll be working with DDD-style aggregates and deep object graphs, so a full repository pattern is probably unnecessary; just a simple stateless wrapper around OrmLite will suffice.
- A DTO for passing Movie data in and out of the our database

POST /movies

We're going to need:

- A PostMovieDto request object
- DB methods to support save (insert/update) a movie object

Show me the code!

There's a lot of new classes in this iteration, so I haven't included code listings here.

To view the complete code for this iteration, go to

https://github.com/dylanbeattie/HeyStack/tree/part_07

To just see the changes required to add OrmLite and SQL database support, see the Git commit at

https://github.com/dylanbeattie/HeyStack/commit/c30753331c2693748aaf46e7f8741b2de6fa5501

Part 8: Adding API methods

OK, now it's time to flesh out our prototype API and add some more resources and methods to it. Here's a proposed API design; you should be able to implement all of the following services using the techniques and libraries we've already covered in this workshop:

Method	Endpoint	Comments
PUT	/movies/{id}	Create or update details of a specific movie
DELETE	/movies/{id}	Remove a movie from the database
GET	/movies/{id}	Returns details of a specific movie
GET	/movies/{id}/credits	Returns a list of actors who appeared in a specific movie, along with details of which part they played.
GET	/actors	Returns a list of actors
GET	/actors/{id}	Returns information about a specific actor
GET	/actors/{id}/credits	Returns a list of movies in which a specific actor has appeared
POST	/actors	Add a new actor to the database
DELETE	/actors/{id}	Remove an actor from the database

Remember, for each API method, you should be looking to implement:

- Appropriate request/response DTOs
- Unit testing, with local service calls against a mocked data layer
- Integration testing, against a locally-hosted service. You might even want to build a fake data layer using in-memory dictionaries or SQLite, so you can run tests with persistent data
- API documentation – supported methods, possible return codes, and comments on what the endpoint actually does

Next Steps

This workshop has hopefully demonstrated some of ServiceStack's capabilities and features, and given you enough running code that you can see how to plug in additional features without too much difficulty. Here's a couple of things we didn't cover that you might want to look at:

Hosting pages with Razor and Markdown

ServiceStack has a fully-featured Razor view engine and a Markdown parser/renderer built in, allowing you to build websites (as opposed to APIs) that can be hosted in a console app, IIS or a Windows service. It's also a great way to add help pages and documentation to your existing APIs.

Redis

ServiceStack includes a native C# client for Redis, a high-performance open-source key-value store. Redis can be used as a cache to improve your API performance, or persisted to disk for longer-term data storage.

Authentication and Authorization

ServiceStack includes authentication providers for most common security mechanisms, including basic authentication, digest authentication, several common OpenID providers and more.

Message Queues

As of version 4, ServiceStack services can be hosted in a message queuing system, using RabbitMQ to deliver request and receive responses instead of HTTP. This has great potential for adapting existing web service based systems into resilient, distributed queue-based architectures.