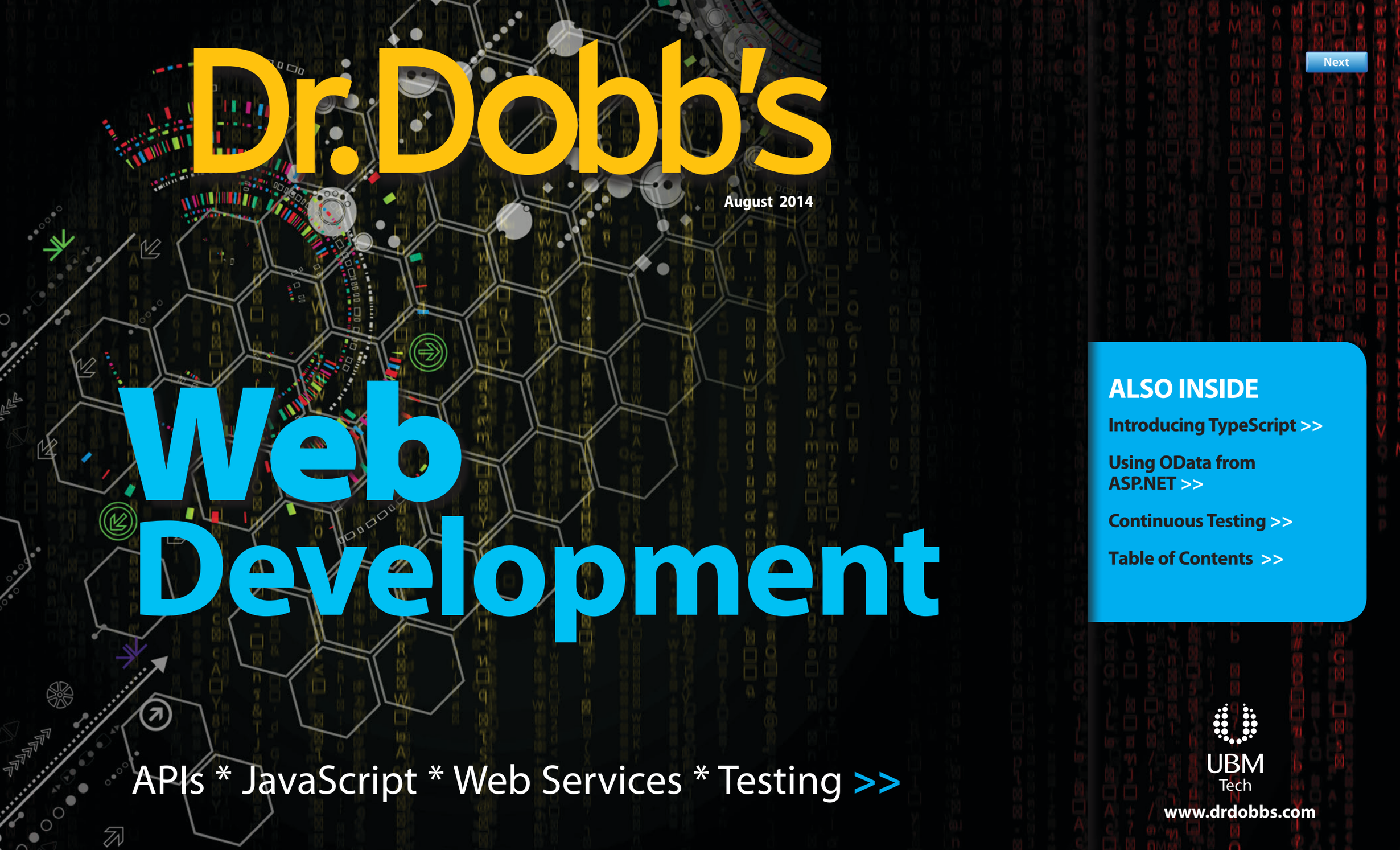
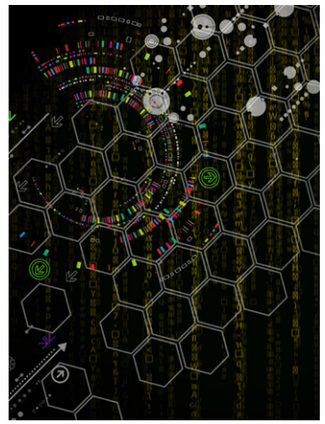

Dr. Dobb's
August 2014
Web Development
APIs * JavaScript * Web Services * Testing >>
Next
ALSO INSIDE
Introducing TypeScript >>
Using OData from ASP.NET >>
Continuous Testing >>
Table of Contents >>
UBM Tech
www.drdobbs.com

Dr.Dobb's

CONTENTS

August 2014



FEATURES

More on DrDobbs.com

Memory Leaks in iOS 7

iOS Networking APIs are leaking memory. The good news: You can stop them.

http://www.drdobbs.com//240168600

What Makes a Good Check-in?

F ew organizations have strong opinions and articulated policies on what a check-in should consist of. As long as the check-in is more or less usable in a code review, it's generally considered good enough. We can do better than this.

http://www.drdobbs.com/240168601

Got Google?

Google services are everywhere. But knowing which ones to include in new apps is difficult due to the company's perceived lack of long-term commitment to products and APIs.

http://www.drdobbs.com/240168592

Why Build Your Java Projects with Gradle Rather than Ant or Maven?

The default build tool for Android (and the new star of build tools on the JVM) is designed to ease scripting of complex, multi-language builds.

http://www.drdobbs.com/240168608

How To Use Reverse Iterators Without Getting Confused

A reverse iterator is a standard-library template that traverses the same range of elements as the corresponding iterator..

http://www.drdobbs.com/240168652

Introduction to TypeScript

Microsoft's alternative to JavaScript adds modularity, generics, and type information while maintaining JS source compatibility.

By Jonathan Turner and Amanda Silver @jntrnr

Used to develop applications for browsers, servers, phones, and the desktop, JavaScript has grown to be one of the most ubiquitous programming languages. For all its popularity, the language was not originally designed for the scale and complexity of today's use cases. One of the key features of programming languages designed for this kind of use is a type system that can enable developers to use modern tooling to refactor, navigate, and detect errors in the code as it grows and changes. TypeScript helps to fill this gap for JavaScript applications. As a superset of JavaScript, TypeScript allows programmers to use their existing JavaScript code and frameworks. TypeScript provides a flexible type system that layers well on top of existing JavaScript, which enables developers to scale their codebases more quickly and with more confidence than with vanilla JavaScript.

TypeScript Starts and Ends with JavaScript

The problem with many errors in JavaScript is that they're silent. For example, it's perfectly valid JavaScript to access an object with an arbitrary member name. If the member doesn't already exist, it will be created. The only way to detect errors like this is to come across them at runtime or by looking for them with unit tests. If we're unlucky, the error lies dormant, waiting for an unsuspecting user to find it.

```
function changeDirection(s) {
    if (Math.random() > 0.5) {
        s.goLet = true; // <-- Silent error
    }
    else {
        s.goRight = true;
    }
```

```
    return s;
}

var s = { goLeft: false, goRight: false };
s = changeDirection(s);
```

TypeScript allows you to annotate JavaScript with type information, and then to run this newly annotated code through a compiler. These types are completely optional, and there's no requirement that the user fully specify all the type information in a program before starting to see benefits.

To help catch the silent error mentioned earlier, we can add an annotation that describes the shape of the object `changeDirection` expects.

```
function changeDirection(s: { goLeft: boolean; goRight: boolean }) {
    if (Math.random() > 0.5) {
        s.goLet = true; // <-- The property 'goLet' does not exist
                        // on value
                        // of type
                        // '{ goLeft: boolean; goRight: boolean; }'.
    }
    else {
        s.goRight = true;
    }

    return s;
}

var s = { goLeft: false, goRight: false };
s = changeDirection(s);
```

We can also refactor this code to use interfaces. Interfaces in TypeScript work differently than interfaces in languages like C# and Java because they are implicitly satisfied by the shape an object has, rather than explicitly requiring the developer to specify that the interface is implemented.

```
interface Direction {
    goLeft: boolean;
    goRight: boolean;
}

function changeDirection(s: Direction) {
    if (Math.random() > 0.5) {
        s.goLeft = true;
    }
    else {
        s.goRight = true;
    }

    return s;
}

var s = { goLeft: false, goRight: false };
s = changeDirection(s);
```

These type annotations exist only during design time and are compiled away, leaving only clean JavaScript in the output with no additional overhead. These annotations allow you to build up checkable documentation about your program. As your application scales, your team can scale with it because having checkable documentation helps enforce a basic consistency in your codebase.

TypeScript Does Generics

For a type system to be beneficial, it needs to be expressive enough to describe common shapes and relationships. TypeScript's type system has been tuned to work with the patterns that are common in JavaScript code. Because of JavaScript's dynamic nature, a flexible generics type system is a key piece of supporting JavaScript programming.

To show how generics work in TypeScript, let's first start with this non-generic example:

```
function filterSmiths(arr: { lastName: string }[]):
        { lastName: string }[] {
    var result = [];
    for (var i in arr) {
        if (arr[i].lastName == "Smith") {
            result.push(arr[i]);
        }
    }
    return result;
}
```

When we call the `filterSmiths` function, we'd like to be able to update the fields and return the same type that was passed in. If we pass in an object that has more than a `.lastName` property, we don't want to lose those extra properties in the resulting type.

```
var result = filterSmiths([
    { firstName: "Bob", lastName: "Smith" },
    { firstName: "Sam", lastName: "Jones" }
]);  //lost .firstName in the result type
```

With generics, we can maintain the type information. If we naively switch our type to a generic type, though, we immediately see an error:

```
function filterSmiths<T>(arr: T[]): T[] {
    var result = [];
    for (var i in arr) {
        if (arr[i].lastName == "Smith") {
            result.push(arr[i]);  // error: The property 'lastName'
                                  // does not exist on value of type 'T'.
        }
    }
    return result;
}
```

The compiler doesn't have sufficient information about `T` to know that we can access the `.lastName` property. To provide this, we need to create a constraint — a description of the minimal shape the generic type `T` has to provide.

```
interface HasLastName {
    lastName: string;
}


function filterSmiths<T extends HasLastName>(arr: T[]): T[] {
    var result = [];
    for (var i in arr) {
        if (arr[i].lastName == "Smith") {
            result.push(arr[i]);
        }
    }
    return result;
}
```

TypeScript Plays Well With Others

TypeScript's key benefit is that it's able to work with existing JavaScript, including both JavaScript that's part of your project and JavaScript from other libraries that your application depends on. Being able to do so fluidly means not rewriting your entire codebase to bring it into a TypeScript application. Instead, TypeScript allows you to describe objects that are visible to your app but may have been loaded outside your script. This includes JavaScript libraries like jQuery, Backbone, and AngularJS that provide utility functionality crucial to your application.

Let's take jQuery as an example. The jQuery library makes a `$` symbol available at runtime that lets developers access much of the jQuery functionality. We could describe the full jQuery API to the compiler, but as a first step, all we need to do is tell the compiler that this symbol will be visible at runtime: `declare var $: any;`

This tells the compiler that the $ symbol is not being created by our application, but rather by an external script or library being run before our script is run. It also says that the type of this variable is any. With this, the compiler will allow you to access any member you wish on this variable without complaint. This enables you to get up and running quickly. While this is effective to get started, it doesn't let the compiler give us the errors and auto-completion, since the compiler lacks any type information about the $ symbol.

Modules and namespaces allow programmers to untangle the mess and create components that can be separately maintained, extended, and even replaced.

To get proper type-checking, we need to have the API of jQuery documented for the compiler. Luckily for us, volunteers have already been hard at work documenting the APIs of many JavaScript libraries, including jQuery. You can reach this repository at http://is.gd/tWuldR on GitHub. To use these API documentation files, called .d.ts files, you include them with your project files or alongside the source files you pass to the compiler. Here's an example of the .d.ts file for jQuery:

```
// The jQuery instance members

interface JQuery {
  // AJAX
  ajaxComplete(handler: any): JQuery;
  ajaxError(handler: (evt: any, xhr: any, opts: any) => any): JQuery;
  ajaxSend(handler: (evt: any, xhr: any, opts: any) => any): JQuery;
  ajaxStart(handler: () => any): JQuery;
  ajaxStop(handler: () => any): JQuery;
  // ...
}

declare var $: JQueryStatic;
```

One way to think of .d.ts files is as the equivalent to headers in a C-based language. They act to describe the API and are a companion to the library they're describing. Similarly, in TypeScript, you use the .d.ts file to inform your tooling and load the corresponding library at runtime.

Modularity in TypeScript

As applications grow larger, it becomes ever more important to have clean separation between components. Without this separation, components morph into a tangled mess of global definitions that become increasingly fragile and more difficult to maintain and extend. Modules and namespaces allow programmers to untangle the mess and create components that can be separately maintained, extended, and even replaced — fortified with the knowledge that such changes won't affect the rest of the system.

TypeScript has two kinds of modules. The first is an internal module. Internal modules help you organize your code behind an extensible namespace, moving it out of the global namespace. This example shows the earlier `changeDirection` example refactored to use an internal module:

```
export interface Direction {
    goLeft: boolean;
    goRight: boolean;
}

export function changeDirection(s: Direction) {
    if (Math.random() > 0.5) {
        s.goLeft = true;
    }
    else {
        s.goRight = true;
    }
```

```
        return s;
    }
}

var s = { goLeft: false, goRight: false };
s = RoadMap.changeDirection(s);
```

These module loaders do the additional service of removing the need to manually order your JavaScript files. Instead, module loaders automatically handle modules by loading a module's dependencies first, before loading the module.

The second kind of module is an external module. External modules let you treat entire files as modules. The added advantage of external modules is that they can be loaded using one of the popular JavaScript module loaders. These module loaders do the additional service of removing the need to manually order your JavaScript files. Instead, module loaders automatically handle modules by loading a module's dependencies first, before loading the module. The end result is a set of modules with clean declarations of their dependencies and a compiler-enforced separation between modules.

Here is the previous example, this time refactored as two external modules:

```
//RoadMap.ts
export interface Direction {
    goLeft: boolean;
    goRight: boolean;
}

export function changeDirection(s: Direction) {
    if (Math.random() > 0.5) {
        s.goLeft = true;
    }
    else {
        s.goRight = true;
    }
    return s;
}

//Main.ts
import RoadMap = require("RoadMap");
var s = { goLeft: false, goRight: false };
s = RoadMap.changeDirection(s);
```

Notice that we now have two separate files, each importing or exporting directly from the file. The RoadMap.ts file has become a single external module denoted by the filename. In the Main.ts file, we load the RoadMap.ts using an import call. This is how we describe the dependency between these two modules. Once imported, we can interact with the module just as before.

To compile external modules, we also have to tell the compiler what kind of module loader we will be using. Currently, the compiler supports two styles: AMD/RequireJS and Node/CommonJS. To compile for AMD, we pass AMD as the module type to the compiler:

```
> tscRoadMap.tsMain.ts--moduleAMD
```

The resulting JavaScript files will then be specialized for the AMD-style module loaders like RequireJS. For example, compiling the Main.ts from our last sample outputs:

```
define(["require", "exports", "RoadMap"],
    function (require, exports, RoadMap) {
        var s = { goLeft: false, goRight: false };
        s = RoadMap.changeDirection(s);
    });
```

You can see where the import call has become part of the list of dependencies being tracked by the module loader using the define call. TypeScript allows us to manage our files as separate external modules,

with all the benefits of using module loaders, while also getting all of the type-checking benefits we expect from working in TypeScript.

Type Inference in TypeScript

Another technique that TypeScript uses to focus types on usability is type inference. Type inference has moved from its functional programming roots to being part of most programming languages today. It is a powerful tool to focus types on utility, rather than being boilerplate. In Typescript, type inference helps to infer types in some of the common JavaScript coding patterns.

The first of these patterns is to infer types during variable declaration from an initializer, a technique common with many programming languages.

```
var x = 3;  // x has type number
```

The next example infers types in the opposite direction as the previous code, by inferring the type left-to-right. If the variable has a declared type, we can infer information about the type of the initializing expression. In this example, the parameter `x` in the function on the right-hand side has its type inferred to number based on the function type provided.

```
var f: (x: number) => string =
       function (x) { return x.toString(); }
```

The next example shows how the context of expression can also help infer its type. Here, the type of the function expression is inferred because the call in which it is created can be resolved to the function declaration, allowing inference to use the type of the declared parameter.

```
function f(g: (x: number) => string) { return g(3); }
f(function (x) { return x.toString(); })
```

We can rewrite the previous examples using lambdas to better understand how the contextual type helps maintain code readability by reducing code cruft.

```
function f(g: (x: number) => string) { return g(3); }
f(x => x.toString());
```

Because of the heavy use of patterns like callbacks in JavaScript, contextual type inference helps keep code simple without sacrificing the power of having the type information available.

Conclusion

TypeScript offers a lightweight, flexible way of working with standards-based JavaScript while enjoying the power that static type information provides. TypeScript's type system focuses on compatibility with existing JavaScript and is designed to require less effort to use than many statically typed languages. If you'd like to learn more about TypeScript, read up on it at the TypeScript homepage at http://www.typescriptlang.org.

— Jonathan Turner is the Program Manager for the TypeScript team at Microsoft. Amanda Silver is the Principal Director Program Manager for Client Platform Tools at Microsoft.

Comment

Using OData from ASP.NET

Take advantage of your knowledge of both ASP.NET MVC and the ASP.NET Web API to easily create OData endpoints.

By Gaston Hillar @GastonHillar

OData (Open Data Protocol) is a standardized protocol for creating and consuming data APIs through regular HTTP requests and REST. It is the "ODBC of the Web" and provides a standard solution for very common Web API patterns. For example, OData (http://www.odata.org/) includes support for queries, paging, and batching, among many other typical requirements.

Creating an OData Endpoint

Microsoft ASP.NET Web API OData adds most of the necessary things to create OData endpoints and easily provide OData query syntax using the ASP.NET Web API (initially introduced in .NET Framework 4.5). ASP.NET Web API 2.2 added many improvements that make it easier to work with OData, including support for OData 4.0. In this article, I provide an example that uses the features provided in Microsoft ASP.NET Web API 2.2 OData.

ASP.NET Web API 2 was included in Visual Studio 2013 and provided support for OData 3.0. The recently launched ASP.NET Web API 2.2 added support for the newest version of the standardized protocol: OData 4.0. In addition, the newest ASP.NET Web API version includes many interesting improvements related to OData, such as attribute routing, model aliasing, support for Enums, ETags, the `$format` query option, and the ability to specify which properties can be sorted, expanded, filtered, or navigated.

I want to work with ASP.NET Web API 2.2 in Visual Studio 2013, so I have to run a few NuGet Package Manager Console commands. I'll explain the steps to create a project that generates an OData endpoint and allows you to run OData queries on a simple `Games` entity with support provided by Entity Framework, but you don't have to use Entity Framework to take advantage of ASP.NET Web API OData. You can use a similar structure with other data sources.

Create a new ASP.NET Web Application project in Visual Studio 2013. I'll use `Games` for both the project and solution names.

Select the `Empty` template and check the Web API checkbox in the New ASP.NET Project dialog box. Then, click OK. Visual Studio 2013 will add the folders and core references for the ASP.NET Web API.

Execute the following commands in the NuGet Package Manager Console to access the latest ASP.NET Web API 2.2. The package follows the Semantic Versioning specification, so it has version 5.2.0. After executing this command, the project will be working with ASP.NET Web API 2.2:

```
Install-Package Microsoft.AspNet.WebApi -Version 5.2.0
Install-Package Microsoft.AspNet.WebApi.OData
```

I'll use a simple `Game` entity model to represent the data for the OData service. Add a `Game` class to the Models folder. The following lines show the code for this new class. Notice that I use an `int` for the key (`GameId`). I don't use a `Guid` to keep the HTTP requests easier to understand.

```
namespace Games.Models
{
    public class Game
    {
        public int GameId { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public int ReleaseYear { get; set; }
    }
}
```

Now, it is necessary to build the project because I can take advantage of the Visual Studio 2013 scaffolding, which will use reflection to gather information about the `Game` class.

I want to add a Web API 2 OData controller with CRUD actions to allow me to create, read, update, delete, and list entities from an Entity Framework data context. Visual Studio 2013 will generate the class that handles HTTP requests to perform these actions. Right-click on the Controllers folder within Solution Explorer and select Add | Controller…. Select Web API 2 OData Controller with actions, using Entity Framework, and click Add. Enter `GamesController` for the controller name and check the Use Async Controller Actions checkbox in the Add Controller dialog box. Select Game (`Games.Models`) in the Model class dropdown list, and click on the New Data Context… button. Visual Studio will specify `Games.Models.GamesContext` as the default new data context type name in a new dialog box. Leave the default name and click Add. Finally, click Add when you are back in the Add Controller dialog box.

Visual Studio will create the following `GamesContext` class in Models/GamesContext.cs:

```
using System.Data.Entity;

namespace Games.Models
{
    public class GamesContext : DbContext
    {
        public GamesContext() : base("name=GamesContext")
        {
        }

        public System.Data.Entity.DbSet<Games.Models.Game>
            Games { get; set; }

    }
}
```

The following lines show the generated code for the `GamesController` class in Controllers/GamesController.cs. `GamesController` implements the abstract class `System.Web.Http.OData.ODataController`. `ODataController` is the base class for OData controllers that supports writing and reading data using the OData formats. I've replaced the deprecated `[Queryable]` attribute with `[EnableQuery]`. Sadly, the scaffolding still works with the

previous ASP.NET Web API OData version and generates code that includes the deprecated attribute. In fact, in some cases, you might have problems with scaffolding and the newest ASP.NET Web API OData-related libraries. If you notice mixed versions in the packages, you can copy and paste the code to your project instead of following the steps I mentioned before.

The `[EnableQuery]` attribute enables a controller action to support OData query parameters. The controller uses the async modifier to define asynchronous methods for the POST, PUT, PATCH/MERGE, and DELETE verbs. In addition, you will notice the use of the `await` keyword and many calls to methods that end with the `Async` suffix. Microsoft .NET Framework 4.5 introduced asynchronous methods. If you haven't worked with the `async` modified in ASP.NET MVC controllers, you can read Using Asynchronous Methods in ASP.NET 4.5 and in MVC 4 (http://www.drdobbs.com/240008768) to get a feel for it.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.Http.ModelBinding;
using System.Web.Http.OData;
using System.Web.Http.OData.Routing;
using Games.Models;

namespace Games.Controllers
{
    public class GamesController : ODataController
    {
        private GamesContext db = new GamesContext();

        // GET odata/Games
        [EnableQuery]
        public IQueryable<Game> GetGames()
        {
            return db.Games;
        }

        // GET odata/Games(5)
        [EnableQuery]
        public SingleResult<Game> GetGame([FromODataUri] int key)
        {
            return SingleResult.Create(db.Games.Where(game =>
                game.GameId == key));
        }

        // PUT odata/Games(5)
        public async Task<IHttpActionResult> Put([FromODataUri]
            int key, Game game)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            if (key != game.GameId)
            {
                return BadRequest();
            }

            db.Entry(game).State = EntityState.Modified;

            try
            {
                await db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!GameExists(key))
                {
                    return NotFound();
                }
                else
                {
                    throw;
```

```
        }
    }

    return Updated(game);
}

// POST odata/Games
public async Task<IHttpActionResult> Post(Game game)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    db.Games.Add(game);
    await db.SaveChangesAsync();

    return Created(game);
}

// PATCH odata/Games(5)
[AcceptVerbs("PATCH", "MERGE")]
public async Task<IHttpActionResult> Patch([FromODataUri]
    int key, Delta<Game> patch)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    Game game = await db.Games.FindAsync(key);
    if (game == null)
    {
        return NotFound();
    }

    patch.Patch(game);

    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!GameExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return Updated(game);
}

// DELETE odata/Games(5)
public async Task<IHttpActionResult> Delete([FromODataUri]
    int key)
{
    Game game = await db.Games.FindAsync(key);
    if (game == null)
    {
        return NotFound();
    }

    db.Games.Remove(game);
    await db.SaveChangesAsync();

    return StatusCode(HttpStatusCode.NoContent);
}

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}

private bool GameExists(int key)
{
    return db.Games.Count(e => e.GameId == key) > 0;
}
}
}
```

Notice the use of the `System.Web.Http.AcceptVerbs` attribute in the `Patch` method to make the `action` method respond to both the PATCH and MERGE HTTP verbs.

```
[AcceptVerbs("PATCH", "MERGE")]
public async Task<IHttpActionResult> Patch([FromODataUri]
    int key, Delta<Game> patch)
```

The parameterless `GetGames` method returns the entire `Games` collection. The `GetGame` method with a key parameter looks up a Game by its key (the `GameId` property). Both methods respond to HTTP GET; and the final method call will depend on the OData query. I'll discuss the effects of the `[EnableQuery]` attribute later.

The rest of the supported HTTP verbs use a single method for each verb: PUT (`Put`), POST (`Post`), and DELETE (`Delete`). Thus, you can query the entity set (`GET`), add a new game to the entity set (`POST`), perform a partial update to a game (`PATCH/MERGE`), replace an entire game (`PUT`), and delete a game (`DELETE`).

The generated controller code includes commented code that allows you to add a route for the controller. Change the code in App_Start/WebApiConfig.cs with the following lines that create an entity data model, also known as EDM, for the OData endpoint and add a route to this OData endpoint.

```
using System.Web.Http;
using System.Web.Http.OData.Builder;
using Games.Models;

namespace Games
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            ODataConventionModelBuilder builder =
                new ODataConventionModelBuilder();
            builder.EntitySet<Game>("Games");
            config.Routes.MapODataRoute("odata", "odata",
                builder.GetEdmModel());
        }
    }
}
```

The `System.Web.Http.OData.Builder.ODataConventionModelBuilder` class allows you to automatically map CLR classes to an EDM model based on a set of default naming conventions. You can have more control over the generating process by calling `ODataModelBuilder` methods to create the EDM. The call to the `EntitySet` method registers an entity set as part of the model. In this case, the name of the entity set is specified as `Games` in the single argument, and the conventions assume the controller is named `GamesController`. You can make additional calls to `EntitySet` to create the EMD model for each entity set you want to make available in a single endpoint.

Finally, the call to `config.Routes.MapODataRoute` maps the specified OData route and adds it to the OData endpoint. The first parameter specifies a friendly name for the route, and the second parameter sets the URI prefix for the endpoint. Thus, the URI for the `Games` entity set is `/odata/Games`. Obviously, it is necessary to add the hostname and port to the URI. For example, if the project URL is http://localhost:50723/, the URI for the `Games` entity set will be http://localhost:50723/odata/Games. The third parameter is the pre-built `Microsoft.Data.Edm.IEdmModel` that is retrieved from the builder with a call to the `GetEdmModel` method. In order to check the project URL, right-click on the project name (`Games`) in Solution Explorer, select Properties, click on Web, and read the configuration in the Project URL textbox.

Working with the OData Endpoint

You can press F5 to start debugging, and the OData endpoint will be ready to process requests. You can use either Telerik Fiddler (http://www.telerik.com/fiddler) or the curl utility (http://curl.haxx.se/) to compose and send requests to the endpoint with different headers and check the results returned by the OData endpoint. Telerik Fiddler is a free Web debugging proxy with a GUI. The curl utility is a free and open source command line tool to transfer data to/from a server and it supports a large number of protocols. You can easily install curl in any Windows version from the Cygwin package installation option (http://www.cygwin.com/), and execute it from the Cygwin Terminal. I'll provide examples for both Fiddler and curl. I'll always use http://localhost:50723/ as my project URL in the samples (don't forget to replace it with your project URL in your own code).

If you send an HTTP GET request to http://localhost:50723/odata, you will receive the service document with the list of entity sets for the OData endpoint. In this case, the only entity set is `Games`. You can execute the following line in a Cygwin terminal to retrieve the service document with curl.

```
curl -X GET "http://localhost:50723/odata/"
```

The following lines show the raw HTTP response:

```
<?xml version="1.0" encoding="utf-8"?>
<service xml:base="http://localhost:50723/odata" xmlns="http://www.
w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title type="text">Default</atom:title>
    <collection href="Games">
      <atom:title type="text">Games</atom:title>
    </collection>
  </workspace>
</service>
```

In Fiddler, click Composer or press F9, select GET in the dropdown menu in the Parsed tab, and enter `http://localhost:50723/odata/` in the textbox at the right-hand side of the dropdown (see Figure 1). Then, click Execute and double click on the 200 result that appears on the capture log. If you want to see the raw response, just click on the Raw button below the Request Headers panel (see Figure 2).

If you add `$metadata` to the previous URI, the OData endpoint will return the service metadata document that describes the data

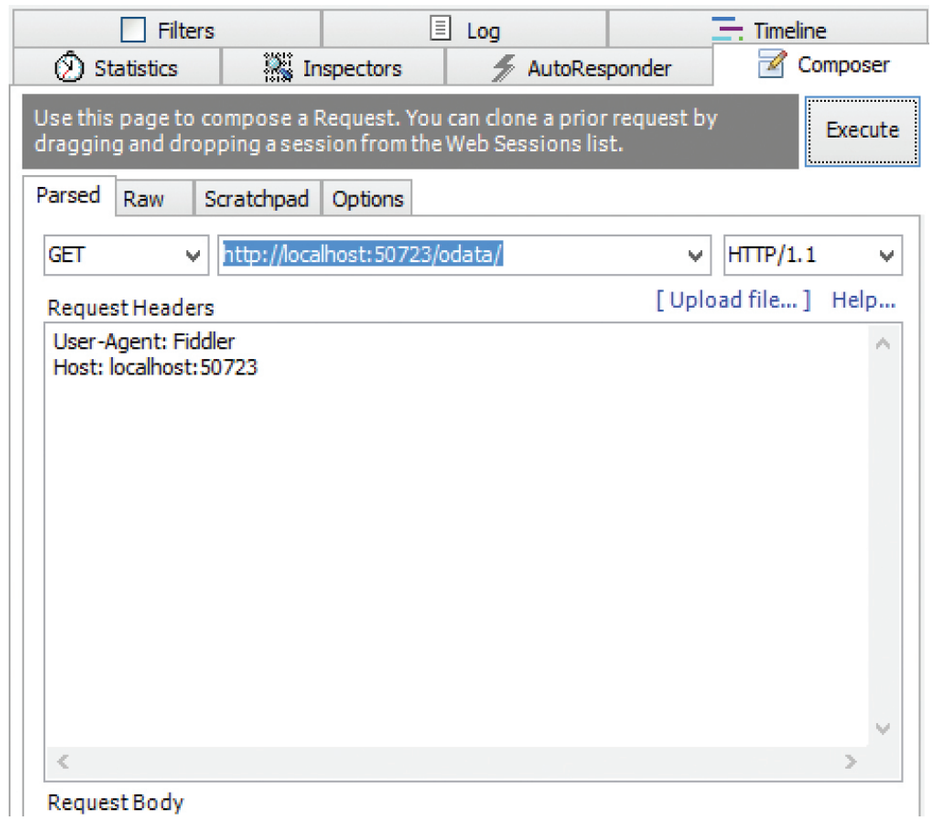


Figure 1: Composing a request in Fiddler.

model of the service with the Conceptual Schema Definition Language (CSDL) XML language. You can execute the following line in a Cygwin terminal to retrieve the service metadata document with curl. In Fiddler, you don't need to add a backslash (\) before the dollar sign ($); you can enter the raw URI: `http://localhost:50723/odata/$metadata`.

```
curl -X GET "http://localhost:50723/odata/\$metadata"
```

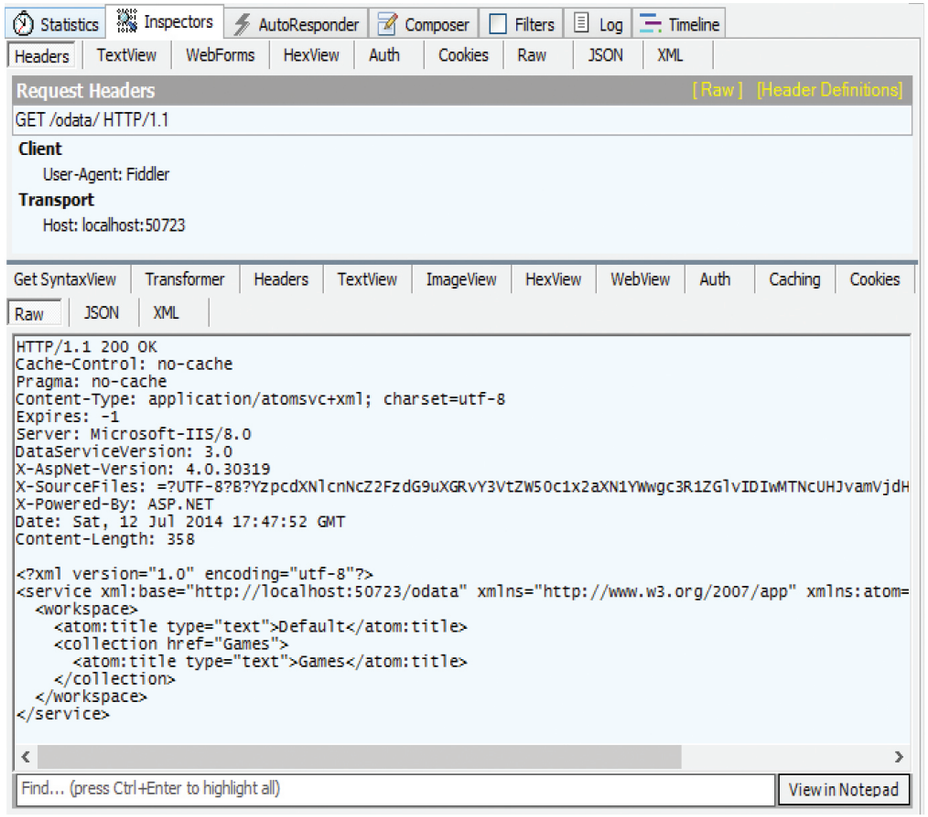


Figure 2: Reading the raw response for the request in Fiddler.

The following lines show the raw HTTP response:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0" xmlns:edmx=
    "http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices m:DataServiceVersion=
    "3.0" m:MaxDataServiceVersion="3.0" xmlns:m=
    "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <Schema Namespace="Games.Models" xmlns=
        "http://schemas.microsoft.com/ado/2009/11/edm">
      <EntityType Name="Game">
        <Key>
          <PropertyRef Name="GameId" />
        </Key>
        <Property Name="GameId" Type="Edm.Int32" Nullable="false" />
        <Property Name="Name" Type="Edm.String" />
        <Property Name="Category" Type="Edm.String" />
        <Property Name="ReleaseYear" Type="Edm.Int32" Nullable="false" />
      </EntityType>
    </Schema>
    <Schema Namespace="Default" xmlns=
    "http://schemas.microsoft.com/ado/2009/11/edm">
      <EntityContainer Name="Container" m:IsDefaultEntityContainer=
        "true">
        <EntitySet Name="Games" EntityType="Games.Models.Game" />
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

Establish a breakpoint at the first line of the `Post` method within the `GamesController` class to see all the work done under the hood by ASP.NET Web API OData. Then, compose and execute an HTTP `POST` request to http://localhost:50723/odata/Games in Fiddler with the following values in the request headers and request body:

```
Request headers: Content-Type: application/json
Request body: { "Name":"Tetris 2014","Category":"Puzzle",
    "ReleaseYear":2014 }
```

You can achieve the same effect with the following curl command:

```
curl -H "Content-Type: application/json" -d '{
    "Name":"Tetris 2013","Category":"Puzzle","ReleaseYear":2013 }'
    -X POST "http://localhost:50723/odata/Games"
```

The request specifies that the body will use JSON and the `Post` method in C# receives a `Game` instance. This method works the same way as an ASP.NET MVC controller `Post` method, in which you check the `ModelState.IsValid` property value.

```
public async Task<IHttpActionResult> Post(Game game)
```

In this case, the model is valid, and when you continue with the execution, either Fiddler or curl will display the following HTTP 201 Created response with the URI for the recently generated element: `http://localhost:50723/odata/Games(1)`:

```
HTTP/1.1 201 Created
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Location: http://localhost:50723/odata/Games(1)
Server: Microsoft-IIS/8.0
DataServiceVersion: 3.0
X-AspNet-Version: 4.0.30319
X-SourceFiles: =?UTF-8?B?YzpcdXNlcnNcZ2FzdG9uXGRvY3VtZW50c1x2aXN1YW
    wgc3R1ZGlvIDIwMTNcUHJvamVjdHNcR2FtZXNcR2FtZXNcb2RhdGFcR2FtZXM=?=
X-Powered-By: ASP.NET
Date: Sat, 12 Jul 2014 04:09:44 GMT
Content-Length: 151

{
  "odata.metadata":
    "http://localhost:50723/odata/$metadata#Games/@Element","GameId":
    1,"Name":"Tetris 2014","Category":"Puzzle","ReleaseYear":2014
}
```

If you send an HTTP `GET` request to http://localhost:50723/odata/Games(1), you will receive the details for the recently saved element in the default JSON light serialization format. OData 3.0 introduced the JSON light serialization format to reduce footprint. The following two curl commands will produce the same result and will end in a call to the GetGame method with the key parameter that retrieves the Game from the database based on the received key value. (In Fiddler, you just need to add `Accept: application/json` to `Request` headers.)

```
curl -X GET "http://localhost:50723/odata/Games(1)"
curl -H "Accept: application/json" -X GET
    "http://localhost:50723/odata/Games(5)"
```

With or without the Accept: application/json line added to the request headers, the result will use the JSON light serialization format:

```
{
  "odata.metadata":"http://localhost:50723/odata/$metadata#Games/
    @Element","GameId":1,"Name":"Tetris 2014","Category":"Puzzle",
    "ReleaseYear":2014
}
```

If you want to receive the response with the older OData v2 JSON "verbose" serialization format, you just need to add the `Accept: application/json;odata=verbose` line to the request headers. For example, the following curl command retrieves the same game with the verbose JSON format:

```
url -H "Accept: application/json;odata=verbose" -X GET
    "http://localhost:50723/odata/Games(1)"
```

The third option is to use the Atom Pub XML serialization format. If you want this format, you simply need to add the `Accept:`

`application/atom+xml` line to the request headers. The following curl command retrieves the same game with the Atom Pub XML format:

```
curl -H "Accept: application/json;odata=verbose" -X GET
    "http://localhost:50723/odata/Games(1)"
```

The `GetGame` method is decorated with the `[EnableQuery]` attribute; hence, you can use OData query parameters. For example, the following HTTP `GET` request uses the `$select` option to specify that it just wants the `Name` property to be included in the response body: `http://localhost:50723/odata/Games(1)?$select=Name`. In curl, you need to add a backslash (\) before the dollar sign (`$`):

```
curl -X GET "http://localhost:50723/odata/Games(1)?\$select=Name"
```

The following lines show the raw HTTP response that only includes the requested property value:

```
{
  "odata.metadata":"http://localhost:50723/odata/$metadata#Games/@Elem
ent&$select=Name","Name":"Tetris 2014"
}
```

The `GetGames` method is also decorated with the `[EnableQuery]` attribute. Thus, you can use the `$filter` query option to select the games that satisfy a specific predicate expression. For example, the following HTTP `GET` request uses the `$filter` option with the `eq` (equal) operator to retrieve the game whose name is equal to Tetris 2014:

```
http://localhost:50723/odata/Games?$filter=Name%20eq%20'Tetris%202014'
```

In curl, you need to add a backslash (\) before the dollar sign (`$`):

```
curl -X GET "http://localhost:50723/odata/Games?\$filter=
    Name%20eq%20'Tetris%202014'"
```

The following lines show the raw HTTP response:

```
{
  "odata.metadata":"http://localhost:50723/odata/$metadata#Games",
      "value":[
    {
      "GameId":1,"Name":"Tetris 2014","Category":"Puzzle",
          "ReleaseYear":2014
    }
  ]
}
```

The code for the `GetGames` method simply includes the `[EnableQuery]` attribute and is just one line that returns an `IQueryable<Game>`. As you can see, ASP.NET Web API OData performs a lot of work under the hood to parse and translate the OData query options and operators to a Linq query.

Conclusion

As you have learned from this simple example, ASP.NET Web API OData allows you to take advantage of your existing knowledge of both APS.NET MVC and the ASP.NET Web API to easily create OData endpoints. The good news is that you can customize all the components and easily add features, such as complex server-side behavior modification via the addition of OData actions. Enjoy!

— Gastón Hillar is a senior contributing editor at Dr. Dobb's.

Comment

Tired of Waiting for QA? Embrace Continuous Testing

Automating late-stage tests — such as performance and mobile testing — early in the development pipeline gets rid of long waits and efficiently reduces release timelines.

By Brad Johnson @bradjohnsonsv

Automating late-stage tests such as performance and mobile testing early in the development pipeline gets rid of long waits and efficiently reduces release timelines.

Nobody likes waiting. It is something we all try to avoid and it is surely one of the principal reasons developers hate testing. Whether it's writing scripts to automate unit tests, or waiting for QA clearance, testing holds back releases and the opportunity to build fun new things. In some organizations, this delay has led to a genuine dislike and distrust of the people and teams that cause the waiting…the testers.

Continuous testing brings developers and testers closer together with process and technologies that result in better products that ultimately make end users wait less. Eliminating end-user wait is the most important goal of all for developers.

Continuous Integration (CI), as Kohsuke Kawaguchi, the architect of the most popular CI server, Jenkins (http://jenkins-ci.org/), once told me, is fundamentally about continuous, complete, automated testing. Much of the development world considers running unit tests and other code build tasks as the primary role of CI. However, as the lines between traditional QA and development continue to

blur, the types of tests that CI runs have become more sophisticated, extending now to load testing and to functional testing of mobile apps on real devices. In this article, I examine how these latter forms of testing can be done with CI and how a model of continuous testing enables compression of the software development pipeline.

Running performance testing early (that is, as soon as code is relatively stable) is a novel concept to many. However, development managers are beginning to realize the value of having developers run small load tests as early as possible.

Performance Testing: Pursuing the End of "Wait"

Running performance testing early (that is, as soon as code is relatively stable) is a novel concept to many. However, development managers are beginning to realize the value of having developers run small load tests as early as possible. These component-level load tests can identify scalability issues and help developers tune front-end performance, isolate memory leaks, and eliminate simple performance killers early in the development cycle. Additionally, when these tests are launched with CI and run as part of automated regression testing, development managers can begin tracking performance trends very early — a cutting-edge advantage that IT organizations are only beginning to appreciate.

Figure 1 shows the results of performance tests when automated as part of a Jenkins-driven build.

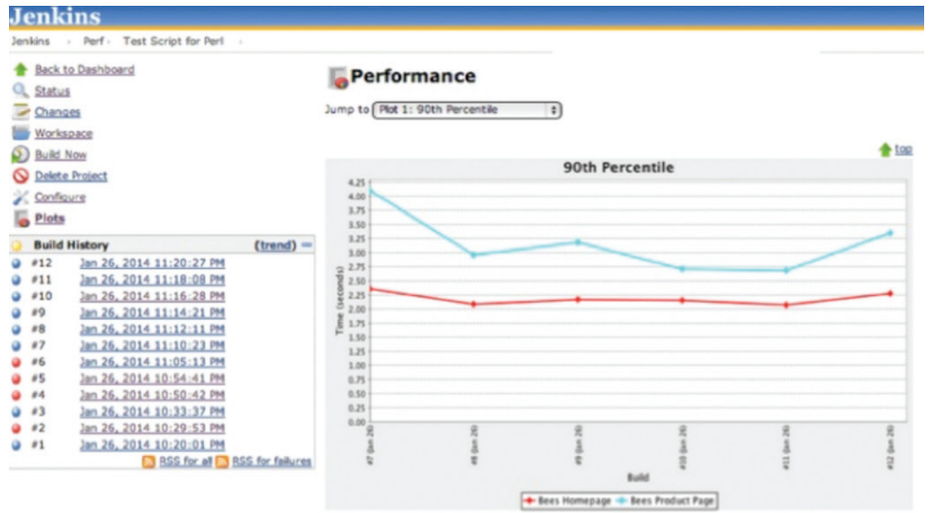


Figure 1: Performance trending in Jenkins.

Continuous Testing

In pursuing the benefits of early automation of performance testing, you might encounter these obstacles:

- *Getting developers to care, and write more tests:* Performance testing has traditionally been an elite art, and it still is. To get it done correctly, developers will want to engage performance engineering experts on your team, or work with a third party, to choose a suitable load testing tool and help define simple, effective load tests. The by-product of this effort is that much of the low-level testing that consumes performance experts will now be covered earlier by tests run by developers. This frees up performance experts to run more complex, end-to-end performance engineering and production tuning.

- *Setting up a test system to handle load:* Most early-stage development takes place on a developer's desktop. At this phase, performance tests should be small and limited. But as soon as builds can be run on a larger system, a team member should be assigned to assure the same configuration and scale of system is provided for the entire team. Big or small, the system should be consistent in order to provide a good baseline for tests. The cloud is a great choice for test systems. In fact, more and more companies are defining a task in their CI system to provision a test environment in a cloud, run a set of automated load tests, report the results, and then tear down the test environment.

Seek out mobile automation that makes test creation fast and setting test validations easy. Your team doesn't want to be coding more for automation than for app dev, so choose wisely.

Mobile Testing: Running App Tests on Real Devices

Mobile app development, by nature, moves fast and demands automated testing. However, it's widely believed that CI can't apply to mobile app testing because loading apps to devices is always a manual process. Add flaky test automation and mobile OS/device fragmentation to this, and the problem is certainly challenging. These were definitely serious obstacles in the recent past, but with plugins to CI frameworks like Jenkins, and great advances in mobile test automation, these barriers no longer remain.

Search the Jenkins site for "mobile testing" and you'll find good solutions to get your apps to your devices. Whether you are using real devices, emulators or cloud-based options, you can automate the process that takes code from a CI solution and loads the app to the devices for testing. You want to ensure the device is clean, rebooted and ready. This is a necessary step, and fortunately available software can facilitate it.

Mobile test automation is an area with few great options, but it's getting better rapidly. The bane of automation, any automation, is development and maintenance expense that exceeds the benefits of automating. Seek out mobile automation that makes test creation fast and setting test validations easy. Your team doesn't want to be coding more for automation than for app dev, so choose wisely. Also, assure that the automation works over and over, even when mobile OS updates occur (assuming your app handles those, too).

The good news about mobile devices is that relatively speaking, they're cheap. Pick an automation framework that doesn't require special or modified devices and then building your device lab is just a matter of choosing which devices to start with. This can seem daunting, but use the Pareto Principle (http://is.gd/yzohkD), choose a few of the most popular devices on the leading mobile operating systems, and you are on your way. Whether you provide the devices yourself or choose a "device cloud" service, you will want to know that the devices your app is being tested on are exactly like those your end users are buying off the shelf. Not jail-broken or rooted, not sick and dying, not overloaded with apps and activity (unless, of course, that's a test case of yours).

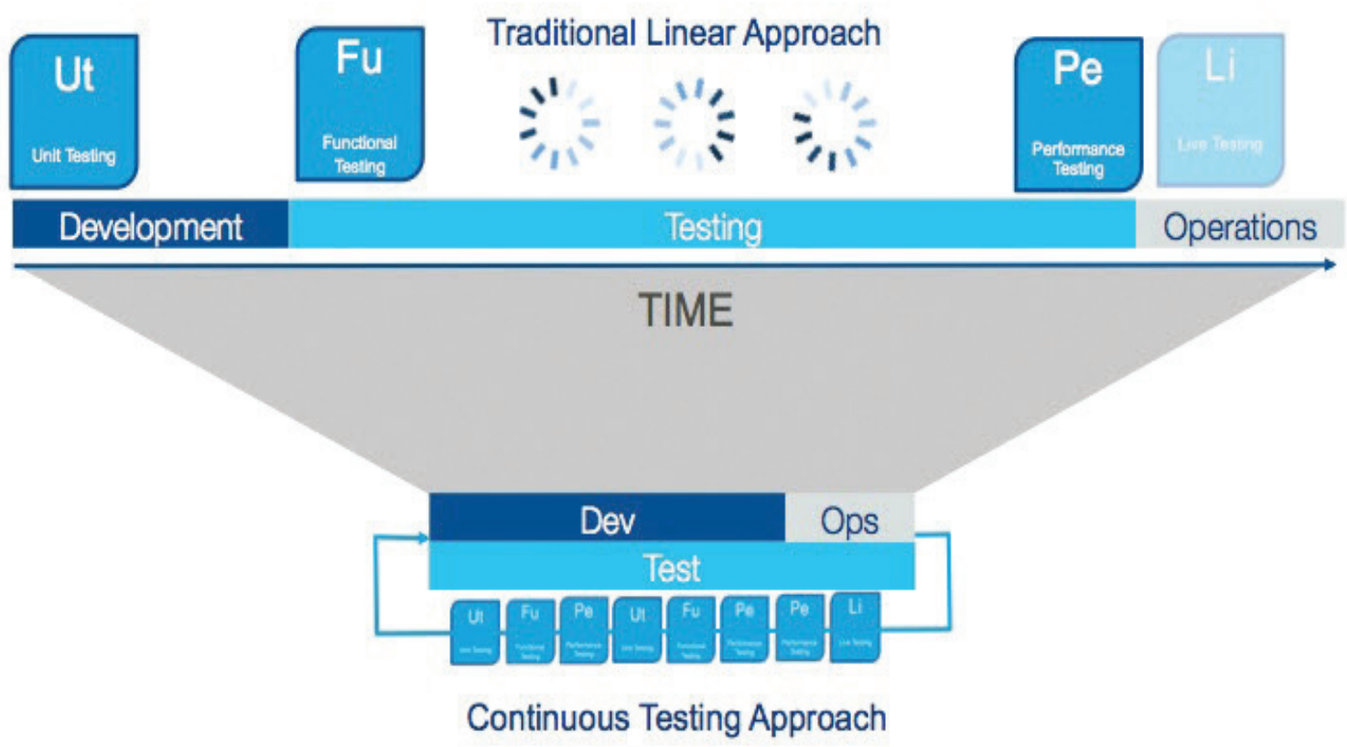


Figure 2: Continuous testing compresses time (Copyright SOASTA).

With good automation and good devices at your disposal, you're now ready to establish your mobile testing framework. Any popular CI solution can be set up to push apps to the devices and run automated tests. This setup is proven to eliminate months of waiting for many companies who only tested manually in the past.

Early Performance Testing Creates Process Compression

When teams learn to move past hating the testing wait and embrace the opportunity to embed more types of testing early and often in the development cycle, time shrinks (Figure 2).

Performance teams can then focus on more tests, with more-complex scenarios as minor performance issues have been eliminated on the developers' desktops.

This frees up teams to run tests, even of production environments, in the time that's been freed by the compressing of testing tasks.

Development teams equipped with functional automation tools, or, test teams who work collaboratively with developers to implement test automation early, thereby eliminate the QA Delay that holds back so many great releases.

Waiting is bad. Join the many companies that now shrink their wait times by collapsing the old dev-test-release cycle with continuous testing.

— Brad Johnson is the vice president of product marketing at SOASTA, a company that specializes in mobile and Web testing with emphasis on cloud-based solutions. He previously worked at Compuware, Mercury Interactive, and Borland.

Comment

From the Vault

Building Successful Web APIs

The syntax of APIs matter much less than their discoverability and the ease with which developers can figure out how to use them effectively, as this great guide from 2012 shows.

By Ronnie Mitra @mitraman

As the demand for connectivity between applications, devices, and platforms grows, so too does the need for Web-based interfaces to connect them together. Not too long ago, these interfaces were called "Web services" and they were often the key component of a Service Oriented Architecture (SOA) initiative. But, with increasing frequency, the terms "Web API," "API," and "API program" are now being used instead.

So, what is the difference? For many in the technology field, a Web *service* implies the usage of the SOAP messaging format, while an *API* implies that any format other than SOAP will be used for messaging. Further to this, some believe that SOA programs live behind the firewall and support internal connectivity, while API programs are focused on outwards facing integration. The colloquial definitions that have evolved for these terms are interesting to note, but it is important to realize that Web APIs still align with the broader, commonly held definition of an SOA. Ultimately, APIs enable the SOA goals of business and technical alignment, increased component reuse, and a higher level of connectedness.

The key difference between the SOA world and the API domain is philosophical. Architects who talk about SOA are often concerned primarily with the exposure and cataloging of service libraries over the network. Whereas architects who talk about APIs will often emphasize a developer-centric view of integration. It is this focus on the usability of an interface that has radically changed the way Web interfaces are designed and exposed.

Why Does Usability Matter?

The various API business drivers can be reduced into two important

goals: the pursuit of growth in API usage and the pursuit of cost reduction in application development. Interest in usability-oriented API design has been fueled by a key realization: Interfaces that are easier to use will help businesses achieve both aims.

Designing for usability is the art of designing products that help users accomplish their own tasks by minimizing the level of user effort required. When given a choice, users prefer products that are easier to use; and in turn, easy to use products improve productivity.

The Nuts and Bolts of Web API Usability

There is a wealth of literature and research available on the design of more-usable interfaces. But the bulk of this material is targeted towards usability design for websites, mobile applications, and physical products; in other words, the products that the average person spends the majority of their time interfacing with. Designing an API for a developer-user who must, in turn, create a software application that can interact with the runtime interface of an API adds a special and important nuance to the usability discussion.

The good news is that even the specific niche of API usability has been well served with guidance from experts. For example, Joshua Bloch's illuminating talk on API design (http://is.gd/pHxtyd) is required viewing and within this publication you can find excellent insight into usability measurement for APIs from Steven Clarke (http://is.gd/si1lt4). Even then, the information is not a perfect fit — most of it is geared towards the design of traditional, locally installed APIs. The nature of Web-based interactions are inherently different; nonetheless, this foundation of usability research and existing design advice provides an ideal starting point.

In general, interaction designers must determine how to extend their users' knowledge to use the interface correctly, how to provide the appropriate controls to operate the interface, and how to provide feedback on the result of the interactions. Interaction design pioneer Bill Verplank (http://is.gd/1GCo2J) describes this process as answering the three questions: "How do you do? How do you feel? How do you know?"

If we examine how developers learn, use, and get feedback from Web APIs, we can break down the key interactions into four major categories: discovery, learning, invocation, and analysis.

Discovery

The first touchpoint for a developer is the discovery of an API that supports a task they want to accomplish.

The method for aiding the discovery of an API often depends on the accessibility of the interface. When an API is created for business partners or for internal use, discovery often occurs through direct and targeted communication from the interface provider. For example, a listing of the APIs that are available and their function may be provided in an email or as part of a contract between two organizations. In some cases, a corporate registry of services might exist to facilitate the internal discovery of an organization's APIs

For public-facing APIs, publishers must provide a means to make developers aware that a programmatic interface exists without having an existing relationship in place. A simple method of advertising the API is to provide links within the organization's primary website to a developer-focused microsite. (In addition, publishers can employ traditional marketing techniques such as email advertisement, conference sponsorship, and hackathon events to increase the likelihood of a discovery interaction and word-of-mouth marketing.)

In some cases, a developer may discover the API before an application function is actually required, possibly leading to the invention of a new application idea based solely on the existence of the API, its underlying function, and the tasks it can facilitate. This type of serendipitous discovery is particularly important when an API has been built to foster innovation.

User-focused interaction gives prospective developers an immediate understanding of how the API can help them.

The key to a successful discovery interaction is to go beyond the minimum requirement of simply communicating the API's existence by helping the prospective developer perceive the value of the product. Good APIs provide straightforward communication of how the interface will provide utility to the developer and support their tasks. For example, users who visit a telephony API page that immediately presents the story of an application using the interface to make a telephone call are likely to quickly grasp the value of the product. This user-focused interaction gives prospective developers an immediate understanding of how the API can help them; and it should lead to higher adoption rates.

Learning

Once developers discover an API that helps them achieve their tasks, they must learn the technical details required to actually use it. They need to gain an understanding of the API's architectural style, the names and purposes of entities, and the parameters available to fine-tune results and responses.

In other words, the developer is asking the question "How do I get this API to support the task I am trying to accomplish?" The goal of a highly usable API should be to provide the means to answer this question easily and effectively with appropriate documentation and a suitable interface.

API documentation can exist in many forms and may range from static document-based reference material for passive reading to a fully immersive and interactive learning experience. Neither is inherently better than the other and each may be better suited to different types of learners.

In addition, an API publisher may provide documentation that is targeted at machines instead of human users. The WSDL description format for SOAP services is an excellent example of this type of machine readable interface documentation. Machine documentation is useful when appropriate tooling exists (as is the case for SOAP), but outside of the SOAP world, Web API client-side tooling is neither widespread nor is it standardized. API designers exposing these kinds of interfaces need to avoid making a machine readable document the sole mechanism for learning because it presents a high barrier for human comprehension.

The key when choosing a documentation strategy is to consider the consumption of the material from the user's perspective. For example, users who are encountering an API for the first time will prioritize simple tutorials and examples that reduce the time it takes to make the first successful API call. On the other hand, developers who have reached an intermediate or expert level will be more interested in an indexed list of resources with more-complex examples.

Beyond the immersion level of the learner, a designer should consider the characteristics of their primary developer targets. This in-

cludes determining their platforms, programming languages, the organizations that they belong to, and even the goals of the end-users they are building applications for. This understanding should ultimately shape the vocabulary, examples, and descriptions of the API to suit the mental model of the target user base.

Of course, the design decisions that take the most time and get the most attention are the ones related to the Web interface itself. API designers must create an interface that provides the core level of functionality required for their users to do what they need, but in a way that requires as little learning as possible and prevents the system from acting in an undesired or unexpected manner.

Developers who have experience using REST or HTTP-based APIs expect that some form of documentation will be provided to aid them.

This includes big decisions regarding the protocols, message formats, and architectural style, as well as a multitude of smaller decisions such as the names and number of parameters, the size of responses, and the relationship structures for entities. Whenever possible, design choices should be informed by the particular developer-user perspective being targeted.

Within the Web API world, human-readable documentation has been established as the conventional way to provide a view of the interface itself.

Developers who have experience using REST or HTTP-based APIs expect that some form of documentation will be provided to aid them in the learning stage

Invocation

Invocation is the first chance for developers to interact with the runtime of an API that has been created rather than simply extending their base of knowledge. Invocation usually involves writing and running client code to test a function. It provides an opportunity to get feedback from the interface and validate assumptions about how the API actually works.

It turns out that this feedback mechanism is essential to learning, but having to write client code presents a barrier to receiving immediate feedback. As a result, usability-focused APIs will often offer shortcut mechanisms to make it easier for their users to push, pull, and prod the API. For example, an API provider may offer its developers a Web-based "API explorer" tool that allows users to dynamically generate interface requests, invoke the API, and view the results — all without writing any code.

Feedback from API invocation is essential, because it is the primary way that developers are able to "sense" the interface they are using. But, feedback is only useful if it is both meaningful and helpful to the user — nonsensical error codes and silent responses do not improve usability. Instead, an API invocation should result in a response that informs both the application client of the runtime result and the human developer who coded the client.

API designers may choose to provide usability aides to make the development of application clients easier. For example, sample code and sample applications make it much easier for prospective developers to write their own applications. Going even further, API owners may choose to provide entire client libraries or SDKs to simplify the act of invocation as much as possible for their primary users. From a developer perspective, an SDK shifts the API experience from a network or

HTTP-based syntax and vocabulary to a local programmatic interface with an application-specific vocabulary.

For example, developers making calls to a typical HTTP object-based API will need to learn the names of resources, the HTTP methods allowed for the API, and then determine how to make the call over the protocol and serialize objects into requests and responses. However, when given an SDK, they can focus simply on the local invocation required to accomplish their goal.

Usability aides and tools provide great usability uplift, but there is a development and maintenance cost associated with each of them. This cost can be particularly taxing in the Web API space because multiple platforms and programming languages often need to be supported. As a result, it is wise to limit development of usability aids to a small number of high-priority user categories and provide a more generic HTTP-based API for the remainder.

Analysis, Troubleshooting, and Support

Learning and invocation-based interactions enable developers to build client applications; however, an additional level of interaction arises after construction is complete. How does the developer determine the level of activity and usage for the application? How does he get a sense of the invocation performance or the pattern of calls that his application has made? Whom does he contact if the API itself is underperforming or is no longer available?

These interactions should be supported with the same usability focus as the rest of the API; otherwise, an organization risks losing active users and increasing operational risk. Self-service analytics tools that permit developers to view and visualize their apps' invocation data are important. It is also useful to provide data about the API's operational statistics and be transparent about the availability and health of the interface. APIs that provide insight into uptime, performance metrics, and outages build a sense of trust within their developer community.

In Practice

Formally adopting usability design practices for Web APIs can be expensive and labor-intensive, and many API designers may struggle to acquire the necessary time and resources for proper user testing and surveying. But even the simple acknowledgment that usability is a feature can have a positive impact on the end result if it permeates the interface design culture of an organization.

Developer-centric interfaces that offer high quality documentation, interactive components, SDK aides, and analytics are winning over developers and reducing the time to market for applications.

As APIs become as ubiquitous as electricity and telephone systems for businesses, the focus on growth and productivity goals will increase. Organizations that embrace a user-centric approach to interface design will reap benefits, while organizations that lack empathy for their API users may be left behind.

— Ronnie Mitra is the director of API design at Layer 7 Technology's API Academy.

Comment

Dr. Dobb's

DR. DOBB'S
EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000