THE EXPERT'S VOICE® IN ASP.NET

SECOND EDITION

ASP.NET Web API 2

Building a REST Service from Start to Finish

HARNESS THE CAPABILITIES OF ASP.NET WEB API 2

Jamie Kurtz and Brian Wortman

Apress®

Contents at a Glance

Introduction

With the introduction of services technology over a decade ago, Microsoft made it relatively easy to build and support web services with the .NET Framework. Starting with XML Web Services, and then adding the broadly capable Windows Communication Foundation (WCF) several years later, Microsoft gave .NET developers many options for building SOAP-based services. With some basic configuration changes, you could support a wide array of communication protocols, authentication schemes, message formats, and WS-* standards with WCF. But as the world of connected devices evolved, the need arose within the community for a simple HTTP-only services framework—without all of the capabilities (and complexity) of WCF. Developers realized that most of their newer services did not require federated authentication or message encryption, nor did they require transactions or Web Services Description Language (WSDL)–based discovery. And the services really only needed to communicate over HTTP, not named pipes or MSMQ.

In short, the demand for mobile-to-service communication and browser-based, single-page applications started increasing exponentially. It was no longer just large enterprise services talking SOAP/RPC to each other. Now a developer needed to be able to whip up a JavaScript application, or 99-cent mobile app, in a matter of days—and those applications needed a simple HTTP-only, JSON-compatible back end. The Internet needs of these applications looked more and more like Roy Fielding's vision of connected systems (i.e., REST).

And so Microsoft responded by creating the ASP.NET Web API, a super-simple yet very powerful framework for building HTTP-only, JSON-by-default web services without all the fuss of WCF. Model binding works out of the box, and returning Plain Old CLR Objects is drop-dead easy. Configuration is available (though almost completely unnecessary), and you can spin up a RESTful service in a matter of minutes.

The previous edition of this book, *ASP.NET MVC 4 and the Web API: Building a REST Service from Start to Finish*, spent a couple chapters describing REST and then dove into building a sample service with the first version of ASP.NET Web API. In a little over a hundred pages, you were guided through the process of implementing a working service. But based on reader feedback, I discovered that a better job needed to be done in two major areas: fewer opinions about patterns and best practices and various open source libraries, and more details on the ASP.NET Web API itself. So when the second version of the ASP.NET Web API was released, Brian Wortman and I decided it was time to release a version 2 of the book. Brian wanted to help me correct some glaring "bugs," and also incorporate some great new features found in ASP.NET Web API 2. And so this book was born.

In this second edition, we will cover all major features and capabilities of the ASP.NET Web API (version 2). We also show you how to support API versioning, input validation, non-resource APIs, legacy/SOAP clients (this is super cool!), partial updates with PATCH, adding hypermedia links to responses, and securing your service with OAuth-compatible JSON Web Tokens. Improving upon the book's first edition, we continue to evolve the message and techniques around REST principles, controller activation, dependency injection, database connection and transaction management, and error handling. While we continue to leverage certain open source NuGet packages, we have eliminated the chatter and opinions around those choices. We also spend more time on the code—lots of code. Unit tests and all. And in the end, we build a simple KnockoutJS-based Single Page Application (SPA) that demonstrates both JSON Web Token authentication and use of our new service.

We have also made improvements in response to your feedback regarding the source code that accompanied the first book. Therefore, on GitHub you will find a git repository containing all of the code for the task-management service we will build together (`https://github.com/jamiekurtz/WebAPI2Book`). The repository contains one branch per chapter, with multiple check-ins (or "commits") per branch to help guide you step-by-step through the implementation. The repository also includes a branch containing the completed task-management service, with additional code to help reinforce the concepts that we cover in the book. Of course, feel free to use any of this code in your own projects.

I am very excited about this book. Both Brian and I are firm believers in the "Agile way," which at its heart is all about feedback. So we carefully triaged each and every comment I received from the first book and did our best to make associated improvements. And I'm really excited about all the new features in the second version of ASP.NET Web API. So many capabilities have been added, but Microsoft has managed to maintain the framework's simplicity and ease of use.

We hope you not only find this book useful in your daily developer lives, but also find it a pleasure to read. As always, please share any feedback you have. We not only love to hear it, but your feedback is key in making continuous improvements.

Cheers,
—Jamie Kurtz
(Brian Wortman)

CHAPTER 1

ASP.NET as a Service Framework

In the years since the first release of the .NET Framework, Microsoft has provided a variety of approaches for building service-oriented applications. Starting back in 2002 with the original release of .NET, a developer could fairly easily create an ASP.NET ASMX-based XML web service that allowed other .NET and non-.NET clients to call it. Those web services implemented various versions of SOAP, but they were available for use only over HTTP.

In addition to support for web services, the 1.0 release of .NET provided support for Remoting. This allowed developers to write services that weren't necessarily tied to the HTTP protocol. Similar to ASMX-based web services, .NET Remoting essentially provides object activation and session context for client-initiated method calls. The caller uses a proxy object to invoke methods, and the .NET runtime handles the serialization and marshaling of data between the client's proxy object and the server's activated service object.

Towards the end of 2006, Microsoft released .NET 3.0, which included the Windows Communication Foundation (WCF). WCF not only replaced ASMX web services and .NET Remoting, but also took a giant step forward in the way of flexibility, configurability, extensibility, and support for more recent security and other SOAP standards. For example, with WCF, a developer can write a non-HTTP service that supports authentication with SAML tokens and host it in a custom-built Windows service. These and other capabilities greatly broaden the scenarios under which .NET can be utilized to build a service-oriented application.

MORE ON WCF

If you're interested in learning more about WCF, I recommend reading either *Programming WCF Services* by Juval Lowy (O'Reilly Media, 2007) or *Essential Windows Communication Foundation (WCF): For .NET Framework 3.5* by Steve Resnick, Richard Crane, and Chris Bowen (Addison-Wesley Professional, 2008). Both of these books are appropriate for WCF novices and veterans alike, as they cover the spectrum from basic to advanced WCF topics. There is also an excellent introduction to WCF in *Pro C# 5.0 and the .NET 4.5 Framework, Sixth Edition* by Andrew Troelsen (Apress, 2012).

If you need to set up communication between two applications, whether they are co-located or separated by thousands of miles, rest assured WCF can do it. And if its out-of-the-box features don't suffice, WCF's tremendous extensibility model provides ample opportunity for plugging in just about anything you can think of.

And this is where we will take a bit of a left turn, off the evolutionary path of ever greater capability and flexibility and towards something simpler and more targeted at a small set of specific scenarios. As this books is about building RESTful services with the ASP.NET Web API, we want to start looking at the need for such services (in contrast to SOAP/RPC style services), and also what types of features and capabilities they provide.

In the Land of JavaScript and Mobile Devices

During much of the growth of the Internet over the past two-plus decades, web sites and pages have relied on server-side code for anything but basic HTML manipulation. But more recently, various AJAX-related tools and frameworks—including (but not limited to) JavaScript, jQuery, HTML5, and some tricks with CSS—have given rise to the need for services that are less about complex enterprise applications talking to each other and more about web pages needing to get and push small amounts of data. One significant example of these types of applications is the Single Page Application (SPA). You can think of these as browser-hosted "fat client" applications, where JavaScript code is connecting from your browser to a service back end. In cases such as these, communicating with a service over HTTP is pretty much a given, since the web sites themselves are HTTP applications. Further, security requirements of browser-based applications tend to be simpler than those of distributed out-of-browser applications, and thus support for all of the various security-related SOAP standards is not required of the service.

In addition to simpler protocol and security needs, web pages typically communicate with other applications and services using text-based messages rather than binary-formatted messages. As such, a service needs only to support XML or JSON serialization.

Beyond web applications, today's smartphones and tablets have created a huge demand for services in support of small, smart-client mobile applications. These services are very similar in nature to those that support AJAX-enabled web sites. For example, they typically communicate via HTTP; they send and receive small amounts of text-based data; and their security models tend to take a minimalist approach in order to provide a better user experience (i.e., they strive for less configuration and fewer headaches for users). Also, the implementation of these services encourages more reuse across the different mobile platforms.

In short, there is a recent and growing desire for a service framework that, out of the box, provides exactly what is needed for these simple, text-based HTTP services. While WCF can be used to create such services, it is definitely not configured that way by default. Unfortunately, the added flexibility and configurability of WCF make it all too easy to mess something up.

And this is where the ASP.NET Web API comes into the picture.

Advantages of Using the ASP.NET Web API

Once you know that you don't need the extended capabilities of WCF, you can start considering a smaller, more targeted framework like ASP.NET Web API. And now on its second version, the ASP.NET Web API provides even more capabilities out of the box, without sacrificing simplicity or its focus on the basics of HTTP service communication. In this section, you'll look at a few of these.

Configuration

As is the case when building a web site, there isn't much to configure to get an ASP.NET Web API-based service up and running. The concept of endpoints doesn't exist (as it does with WCF), and neither do contracts. As you'll see later, an ASP.NET Web API-based service is pretty loose in comparison to a WCF service. You pretty much just need a REST URL, a set of inbound arguments, and a response JSON or XML message.

REST by Default

Speaking of REST, building services with the ASP.NET Web API provides most of the nuts and bolts of what you need to adhere to the constraints of the REST architecture. This is largely due to the URL routing feature provided by the framework. Unlike WCF, where a service is an address to a physical file (i.e., an address that maps directly to a service class or `.svc` file), service addresses with the ASP.NET Web API are RESTful routes that map to controller methods. (We'll talk more about the basics of the REST architectural style in the next chapter.) As such, the paths lend themselves very nicely to REST-style API specifications.

This concept of routing is critical to understanding how the ASP.NET Web API can be used for building services, so let's look at an example. In this book, you will learn how to develop a simple task-management service. You can imagine having a SOAP-based service method to fetch a single task. This method would take a task's TaskId and return that task. Implemented in WCF, the method might look like this:

```
[ServiceContract]
public interface ITaskService
{
    [OperationContract]
    Task GetTask(long taskId);
}

public class TaskService : ITaskService
{
    private readonly IRepository _repository;

    public TaskService(IRepository repository)
    {
        _repository = repository;
    }

    public Task GetTask(long taskId)
    {
        return _repository.Get<Task>(taskId);
    }
}
```

With an appropriately configured .svc file and corresponding endpoint, you would have a URL that looks similar to this:

```
http://MyServer/TaskService.svc
```

The caller would then post a SOAP request with the SOAP action set to GetTask, passing in the TaskId argument. Of course, when building a .NET client, much of the underlying SOAP gunk is taken care of for you. But making SOAP calls from JavaScript or a mobile application can be a bit more challenging.

WHAT DO WE MEAN BY "TASK"?

We understand that "task" is an overloaded word, and the fact that the .NET Framework includes a Task class only complicates matters. Therefore, what we mean by the word "task" is based on the context in which it appears. The Task classes we will implement in the task-management service (there are three of them, at different layers in the application) map to the problem domain. Please take care to avoid confusing them with the .NET Framework's Task class.

This same example under the ASP.NET Web API would involve creating a controller instead of a WCF service class. The method for fetching a Task object exists on the controller, but it is no longer defined by a contract, as it is in WCF. The controller might look like this:

```
public class TasksController : ApiController
{
    private readonly IRepository _repository;
    public TasksController(IRepository repository)
    {
        _repository = repository;
    }

    public Task Get(long taskId)
    {
        return Json(_repository.Get<Task>(taskId));
    }
}
```

If you've built any RESTful services using the ASP.NET MVC Framework (as opposed to the ASP.NET Web API), one of the biggest differences you'll notice is the base class being used, `ApiController`. This base class was built specifically for enabling RESTful services, and you simply return the object (or objects in a collection) of the data being requested. Contrast this with the required use of `ActionResult` in an MVC-based REST controller method.

The URL for obtaining a specific Task from the preceding controller would be this:

`http://MyServer/Tasks/123`

Unlike with the MVC Framework, the URL doesn't need to include the controller's method name. This is because, with the ASP.NET Web API, HTTP verbs (e.g., GET, POST, PUT) are automatically mapped to corresponding controller methods. As you'll see in the next chapter, this helps you create an API that adheres more closely with the tenets of the REST architecture.

For now, the important thing to realize is that the entirety of this service call is contained in the URL itself; there is no SOAP message to go along with the address. And this is one of the key tenets of REST: resources are accessible via unique URIs.

A QUICK OVERVIEW OF REST

Created by Roy Fielding, one of the primary authors of the HTTP specification, REST is meant to take better advantage of standards and technologies within HTTP than SOAP does today. For example, rather than creating arbitrary SOAP methods, developers of REST APIs are encouraged to use only HTTP verbs:

- GET
- POST
- PUT
- DELETE

REST is also resource-centric; that is, RESTful APIs use HTTP verbs to act on or fetch information about resources. These would be the nouns in REST parlance (e.g., Tasks, Users, Customers, and Orders). Thus, you have verbs acting on nouns. Another way of saying this is that you perform actions against a resource.

Additionally, REST takes advantage of other aspects of HTTP systems, such as the following:

- Caching
- Security
- Statelessness
- Network layering (with various firewalls and gateways in between client and server)

This book will cover REST principles sufficiently for you to build services using the ASP.NET Web API. However, if you're interested, you can find several good books that cover the full breadth of the REST architecture. You might also find it interesting to read Chapter 5 of Fielding's doctoral dissertation, where the idea of REST was first conceived. You can find that chapter here:

`http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`

Before moving on, let's quickly address a point that some may be thinking about: you can indeed create REST services with WCF. Looking around the Internet, you can certainly find arguments on both sides of the ASP.NET Web API versus WCF debate (for building RESTful services). Since this is a book on how to build services with the ASP.NET Web API, let's skip that debate altogether.

Abstraction with Routes

Somewhat similar to service interfaces and their implementations in WCF, routes give the ASP.NET Web API service developer a layer of abstraction between what the callers see and the underlying implementation. In other words, you can map any URL to any controller method. When the API signature (i.e., the REST URL) isn't hard-wired to a particular interface, class, or `.svc` file, you are free to update your implementation of that API method, as long as the URL specification for that method remains valid.

One classic example of using URLs to handle changing implementations is in the case of service versioning. By creating a new route with a "v2" (or similar) embedded in the URL, you can create an arbitrary mapping between an implementation and a versioning scheme or set of versions that doesn't exist until sometime later. Thus, you can take a set of controllers and decide a year from now that they will be part of the v2 API. Later on in this book, you learn about a few different options for versioning your ASP.NET Web API service.

Controller Activation Is, Well, Very Nice

Whether the subject is the older XML Web Services (a.k.a. ASMX services), WCF, services with ASP.NET MVC or with the ASP.NET Web API, the concept of *service activation* is present. Essentially, since by-and-large all calls to a service are new requests, the ASP.NET or WCF runtime activates a new instance of the service class for each request. This is similar to object instantiation in OO-speak. Note that service activation is a little more involved than simply having the application code create a new object; this book will touch on this topic in more depth in later chapters. Understanding activation and dependency resolution is very important if you want to have a solid grasp of any service application, including the ASP.NET Web API.

Simpler Extensible Processing Pipeline

ASP.NET Web API provides a highly-extensible, yet much simpler, processing pipeline. We will cover several examples of this in this book. For example, *delegating handlers* (a.k.a. "handlers") and *filters* are mechanisms providing pre- and post-processing capabilities.

Handlers allow you to execute custom code prior to any controller being activated within the application. In fact, handlers can be configured to handle routes that have no corresponding controller.

Filters are essentially classes that contain a few methods allowing you to run some code before and after specific controller methods are invoked. These come in a few different flavors: action filters, authorization filters, and exception filters. These filters take the form of attributes, and they are either decorated on specific controller methods, decorated on the controllers themselves, or configured globally for all methods.

It's a bit tough to describe, but once you write and debug a few controllers — along with a delegating handler and some action filters—you will start noticing how clean and easy Microsoft has made this arrangement. Nothing is hidden from you, making it simple to understand and step through an entire service call in the debugger.