Building Web Services with Microsoft Azure

There's a plethora of development tools and frameworks available today. But Microsoft Azure provides a scalable and manageable platform for customers to easily deploy, monitor, and troubleshoot their cloud-based applications. Its seamless integration and inherent support for open source software make it an obvious choice for building cloud-based applications and services.

This book will take you through a full application development architecture. You'll start by creating, customizing, and extending HTTP-based web APIs and move on to host your web API with ASP.NET. Then, using a range of different features, you'll learn how to connect your applications with Microsoft Azure Service Bus. You'll learn more advanced topics such as creating Hybrid Services, Azure Storage data services, and the Entity Framework; you'll learn to create and manipulate data services, too. Finally, you'll leverage the full power of SQL database administration in Azure and gain more insights into managing your data.

Who this book is written for

If you are a .NET developer who wants to develop end-to-end RESTful applications in the cloud, then this book is for you. A working knowledge of C# will help you get the most out of this book.

What you will learn from this book

- Build RESTful services using the ASP.NET Web API and Microsoft Azure
- Host and monitor applications in Azure Websites and Azure Mobile Services
- Manage Web APIs using Azure API Management
- Utilize Azure Service Bus to provide elasticity to your applications as well as publish and subscribe features
- Utilize the Microsoft Azure Platform as a Service (PaaS) component in your custom solutions
- Get to grips with the basic characteristics of distributed systems
- Use Entity Framework as the data model
- Leverage your cloud-based storage and discover how to access and manipulate data in the cloud
- Explore the NoSQL options available in Microsoft Azure

Quickly develop scalable, REST-based applications or services and learn how to manage them using Microsoft Azure

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Getting Started with the ASP.NET Web API'
- A synopsis of the book’s content
- More information on Building Web Services with Microsoft Azure

About the Authors

Alex Belotserkovskiy is a technical evangelist for Microsoft Russia and lives in Moscow. He specializes in cloud, Internet of Things, and high performance computing topics. Alex is actively engaged in both local and international speaking activities, and works with top customers and partners to provide professional technical and technological support for their cloud projects.

Alex was the first Russian Windows Azure Most Valuable Professional, in 2012, and is a Microsoft certified developer and enterprise administrator. He is an experienced Microsoft technologies instructor.

Stephen Kaufman works for Microsoft as a solution architect in the Americas Office of the CTO and is the lead architect for the US Azure PaaS Center of Expertise (CoE). He is a public speaker and has appeared at a variety of industry conferences nationally and internationally at events, such as TechEd North America, TechEd EMEA, Microsoft SOA and BPM conference, as well as many internal Microsoft conferences over the years discussing application development, integration, and cloud computing, as well as a variety of other related topics.


In addition, he is a board certified architect (CITA-P-IASA Global) and continues to work mentoring and sitting on architecture certification review boards.

Lastly, Stephen was a contributing author for the Azure Architecture Certification Exam 70-534, Architecting Azure Solutions.
Nikhil Sachdeva is a senior consultant at Microsoft. He has over 11 years of experience in architecting and implementing scalable web applications and services using Microsoft technologies. He has been involved with Microsoft Azure since its early days and currently works as a subject matter expert in building custom Platform as a Service (PaaS) solutions on the Azure platform. He has a passion for writing and is a contributing writer for *Introducing Windows Azure for IT Professionals*, *Microsoft Press*, and has contributed to several other Microsoft articles and blogs on Microsoft Azure and related technologies. His recent passion is building highly scalable and available solutions for the Internet of Things (IoT) and frequently rants his experiences at http://connectedstuff.net.
Building Web Services with Microsoft Azure

With multiple cloud platforms out there, it is easy to get confused when making a technology decision for your projects. This gets further complicated with the plethora of development tools and frameworks available today. Microsoft Azure simplifies this problem by providing a scalable and manageable platform for customers to easily deploy, monitor, and troubleshoot their cloud-based applications. Its seamless integration with new and existing Microsoft tools and inherent support for open source software makes it an obvious choice for building cloud-based applications and services.

Whether you are new to Microsoft Azure cloud development or you have been creating cloud applications, there will be something new for you in this book. We will cover the full application development architecture and cover all tiers of an application. We will also cover a number of patterns that you will encounter, from solutions that are completely hosted in the cloud to hybrid solutions where applications are split between the cloud and on-premises networks.

What This Book Covers

Chapter 1, Getting Started with the ASP.NET Web API, introduces the ASP.NET Web API framework and provides an overview of its application and internals. It will guide you through the stages of creating a Web API and deploying it in Microsoft Azure.

Chapter 2, Extending the ASP.NET Web API, discusses various extensibility and customization options available in the ASP.NET Web API framework. It guides the reader through various extension points, such as custom routing, message formatters, content negotiation, and securing a Web API. It also discusses various hosting options for deploying Web APIs.

Chapter 3, API Management, provides a set of tools that assist API developers in managing and monitoring Web APIs. We will discuss various options of publishing, marketing, monitoring, and managing a Web API using API management.

Chapter 4, Developing a Web API for Mobile Apps, provides an overview of Mobile Services and walks through a scenario of creating a Web API using Mobile Services. Mobile Services provides an easy-to-use environment to rapidly build cross-platform apps for Windows, iOS, Android, and other platforms. Its rich built-in capabilities for managing backend login, data, authentication, and notifications makes it a compelling option for developing mobile applications.
Chapter 5, Connecting Applications with Microsoft Azure Service Bus, discusses Windows Azure Service Bus, which allows for related and brokered messaging using a range of different features (such as Topics and Queues).

Chapter 6, Creating Hybrid Services, demonstrates how to create Hybrid Services to connect on-premises Large Object (LOB) / database to cloud-based applications. Essentially, this chapter builds on the previous chapter, demonstrating how you can use the elements of the Service Bus to create Hybrid applications. It will also demonstrate how to effectively maintain these applications.

Chapter 7, Data Services in the Cloud – an Overview of ADO.NET and Entity Framework, explores how to create data services in the cloud using Entity Framework and ADO.NET.

Chapter 8, Data Services in the Cloud – Microsoft Azure Storage, explores how you can use cloud-based Azure Storage technologies.

Chapter 9, Data Services in the Cloud – NoSQL in Microsoft Azure, explores how to use DocumentDB, a fully managed, highly scalable NoSQL data management service based on Azure, as well as ways to start using other open source Azure options such as MongoDB.
ASP.NET Web API is a framework for building HTTP Services. It is part of the ASP.NET platform, which is a free web framework for building websites and services.

Microsoft Azure makes building web services extremely easy. It provides the backbone for hosting scalable Web APIs and then efficiently managing and monitoring them.

In this chapter, we get to know the ASP.NET Web API framework. We delve into the fundamentals of the components that encompass the ASP.NET Web API, walk through the request-response pipeline of ASP.NET Web API, and also talk about the main features provided by the framework that make development of ASP.NET Web API a breeze. We then begin with the essential step of creating our first ASP.NET Web API and then deploy it in Microsoft Azure.

The ASP.NET Web API framework

ASP.NET Web API provides an easy and extensible mechanism to expose data and functionality to a broad range of clients, including browsers and modern web and mobile apps. The most significant aspect of ASP.NET Web API is that it is targeted to enable support for HTTP and RESTful services. In this section, we will discuss the nuts and bolts of the Web API framework.

In the context of this book, the words ASP.NET Web API, Web API, and HTTP Services represent the same thing unless explicitly mentioned. This chapter assumes that the reader has knowledge about HTTP, REST, and Web API, in general. For more information on these technologies, please refer to the introduction chapter of this book.
Background

Before we delve into the building blocks and design principles behind ASP.NET Web API, it is important to understand that ASP.NET Web API is an evolution of the existing continuous Microsoft efforts to enable support for HTTP Services. A timeline of events that describe this evolution are outlined next.

Windows Communication Foundation (WCF) ([https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx](https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx)) was launched with .NET 3.0 for SOAP-based services; the primary aim was to abstract the transport layer and enable support for the WS-* protocol. No HTTP features were enabled except HTTP POST for requests.

In .NET 3.5, [WebHttpBinding](https://msdn.microsoft.com/en-us/library/system.servicemodel.webhttpbinding%28v=vs.110%29.aspx) was introduced in WCF with the intent to support services that were not based on SOAP. It allowed systems to configure endpoints for WCF services that are exposed through HTTP requests instead of SOAP messages. The implementation was very basic and most HTTP protocol features were still missing or needed to be coded separately.

WCF Starter Kit ([https://aspnet.codeplex.com/releases/view/24644](https://aspnet.codeplex.com/releases/view/24644)) preview was launched to provide a suite of helper classes, extension methods, and Visual Studio project templates for building and consuming HTTP REST-based services. A new `WebServiceHost2` type was added to host RESTful services. Also, client HTTP support was added to provide a more natural experience for HTTP programming. The project never got released and eventually migrated into WCF Web API. As of August 2009 the project is in preview status.

During the same time, ASP.NET released some basic support to create REST APIs with its .NET 4.0 release. Its capabilities were limited, and HTTP features such as content negotiation were not available. ASP.NET was still targeted towards building web applications.

WCF Web API ([http://wcf.codeplex.com/wikipage?title=WCF%20HTTP](http://wcf.codeplex.com/wikipage?title=WCF%20HTTP)) was technically the first attempt to create a framework to support HTTP services from the ground up. However, the development efforts still leveraged pieces from WCF REST Starter Kit and .NET 3.5. A new rich, high-level HTTP programming model was adopted that included full support for content negotiation. A variety of traditional formats were supported (XML, JSON, and OData), and server-side query composition, ETags, hypermedia, and much more was enabled. The development was simplified by introducing the use of `HttpRequestMessage` and `HttpResponseMessage` to access requests and responses. The WCF Web API's second release was compatible with ASP.NET and allowed registering routes for Web APIs similar to ASP.NET MVC.
The WCF Web API project merged with the ASP.NET team to create an integrated Web API framework, and ASP.NET Web API was born. It shipped with the MVC4 Beta release in February 2012 and went to RTM in August 2012.

ASP.NET Web API 2 was released in October 2013 with new features such as attribute routing and authorization with OAuth 2.0. Another version, 2.1, was released in January 2014 with improvements to the existing infrastructure and additional features such as global exception handling.

At the time of writing, ASP.NET Web API 2.2 is the current stable version of the ASP.NET Web API framework. It includes more improvements to existing features such as enabling client support for Windows 8.1 devices along with bug fixes.

**Building blocks**

As described in the previous section, ASP.NET Web API is built on top of existing frameworks and technologies. Web API leverages features from ASP.NET MVC, the core ASP.NET framework, and the .NET framework to reduce the overall learning curve for developers and at the same time provides abstraction to make programming easier. The following figure highlights the key features that make up the ASP.NET Web API framework.
ASP.NET Web API leverages some standard features from the ASP.NET MVC framework. Though expertise in ASP.NET MVC is not a requirement for developing HTTP Services using ASP.NET Web API, a fundamental understanding of the ASP.NET MVC framework and the MVC design pattern is useful. More information about the ASP.NET MVC framework is available at http://www.asp.net/mvc.

### Design principles behind the ASP.NET Web API

Now that we understand the intent behind ASP.NET Web API, let's discuss some design principles on which ASP.NET Web API is based:

- **Distributed by design**: ASP.NET Web API considers HTTP as a first-class citizen and has inherent support for REST. It enables a framework for creating distributed services that leverage HTTP features such as stateless, caching, and compression.

- **Russian Doll Model**: This is also called the Matryoshka doll model, which refers to a set of wooden dolls of decreasing sizes placed one inside the other. Architecturally, it denotes a recognizable relationship of nested objects (object within an object). The ASP.NET Web API messaging handlers leverage this principle to define the request-response pipeline. Each handler in the pipeline is responsible for processing the incoming request and then delegating the request to another handler. When a request reaches a handler, it may opt to process it, validate it, and then delegate the request to another handler or break the chain by sending a response. Of course, this is not true for the last handler in the chain since it will not delegate but rather work to get the response back up the chain. The response will follow the same logic but in the reverse direction and each handler will get the response before sending it back to the caller. We will discuss message handlers in detail in the next section. The follow figure describes the Russian Doll model being employed by ASP.NET Web API framework.
• **Asynchronous to the core**: The ASP.NET Web API leverages the .NET Task-based Asynchronous Pattern (TAP) model to the core. TAP is based on the `Task` and `Task<TResult>` types in the `System.Threading.Tasks` namespace, which are used to represent arbitrary asynchronous operations. TAP provides a new pattern to work with both CPU-intensive and non-CPU-intensive asynchronous invocations. It simplifies the overall programming model for asynchronous operations in .NET 4.5 through the `async` and `await` model. Let's look at an example:

```csharp
async Task<int> GetContentLengthAsync(string uri)
{
    int contentLength;
    using (var client = new HttpClient())
    {
        var content = await client.GetStringAsync(uri);
        contentLength = content.Length;
    }
    return contentLength;
}
```
In the preceding example:

- We create an asynchronous method `GetContentLengthAsync` that accepts a web URI and returns the length of the response.
- The return type for this method is `Task<TResult>`, the return type can also be `Task` or `void`.
- An `async` method typically includes at least one `await` call, which will mark a path where the code cannot continue unless an asynchronous operation is completed. It does not block the thread though. Instead, the method is suspended and the control returns to the caller.
- The method execution continues after the response is returned from the asynchronous call.

A TAP-based pattern simplifies writing asynchronous programming code and improves overall responsiveness of the system.

- ASP.NET incorporates dependency injection at various levels in the core types of the system. The ASP.NET Web API exposes a set of interfaces such as `IDependencyResolver` (http://www.asp.net/web-api/overview/advanced/dependency-injection), which enables developers to modify almost everything in the message pipeline by injecting their custom instances. Another good side effect of Dependency Injection in ASP.NET Web API is that we can easily create unit tests for services built on the framework thus improving testability.
- ASP.NET Web API is intended to be an open framework, and it leverages the ASP.NET infrastructure for enabling multiple hosting options. We can host a Web API from a simple console application to a powerful server such as IIS. We will cover the different options in Chapter 2, Extending the ASP.NET Web API.

**Application scenarios**

As mentioned before, ASP.NET Web API enables customers to easily and rapidly develop HTTP-based services that can result in new business opportunities or improved customer satisfaction. Some of these scenarios are described here:

- **Mashup**: This refers to accessing content from more than one service and then creating a new service typically via a user interface. For example, multiple services exposed by the Bing Maps API can be used by a consumer to create a map-plotting user interface that can show the location of nearby location on a map.
• **Smartphone apps:** This is a no-brainer; the mobile app market is one of the most booming markets today. An independent survey suggests that around 56 billion apps were downloaded in 2013 generating revenue of around $20-25 billion, and this number is only going to increase in the coming years. ASP.NET Web API provides multiple client libraries to offer developers with quick and easy ways to create HTTP Services. Moreover, Microsoft Azure Mobile Services provide a robust backend to host mobile apps and Web APIs for multiple platforms, such as iOS, Android, and Windows. It provides baked-in capabilities such as push notifications, social integration, and database management, which seem like a perfect fit for developing mobile applications. We will explore more about Azure Mobile Services and Web API in *Chapter 4, Developing a Web API for Mobile Apps.*

• **Single Page Application (SPA):** In this, all the necessary code artifacts such as HTML, JavaScript, and CSS are retrieved in a single page load. The appropriate resources are then dynamically rendered by making asynchronous calls to services exposing data and functionality. ASP.NET provides inherent support to populate JS frameworks, such as Knockout.js and Angular JS, and provides full support for HTML5 and Web Sockets (via SignalR). These technologies work together with the backend HTTP Services layer built using ASP.NET Web API and Microsoft Azure websites to provide a comprehensive solution to develop SPA.

• **Intranet and partner integration:** When thinking about Web API, we usually think about publically published APIs; however, there is much value in developing Web API in a corporate or intranet scenario as well. If we go back in history, one of the first Web APIs from Amazon was a byproduct of the work they had been doing in their internal applications. Web API brings reusability, and this brings down the development and testing cost when dealing with multiple teams and large organizations.

• **Internet of Things:** In the past year, **Internet of Things (IoT)** has become the hottest buzzword. It is a phenomena bigger than anything that has happened in the software industry since its inception. IoT coupled with cloud technologies enables scenarios that never existed. For example, a thermostat talking to a cloud service and providing telemetry data to the manufacturer can open avenues of new businesses via predictive maintenance using data analytics or partner integration to provide better customer service. As devices are becoming more powerful, they are no longer restricted to short-range networks or lightweight protocols. HTTP infrastructure and Web APIs can play a pivotal role in such scenarios in the coming years.
Behind the scenes with the ASP.NET Web API

Let's talk about some of the internal workings of ASP.NET Web API and how a request is received and a corresponding response generated.

Anatomy of the API of ASP.NET Web API

Before we understand the request and response lifecycle within ASP.NET Web API, it is important to comprehend some of the principal types and their usage within the pipeline.

The core assemblies for ASP.NET Web API can be installed using the Microsoft.AspNet.WebApi NuGet package, which is distributed under the MS license, this will install all the required dependencies to develop an ASP.NET Web API service.

NuGet is the package manager for the Microsoft development platform, including .NET. The NuGet client tools provide the ability to produce and consume packages. NuGet is installed as part of the Visual Studio 2013 release. To know more about NuGet, visit https://www.nuget.org/.

To install the runtime packages via the NuGet Package Manager Console in Visual Studio use the following command:

PM> Install-Package Microsoft.AspNet.WebApi

Alternatively, Visual Studio provides a NuGet user interface dialog that can be used to install the package. For more information on using the NuGet dialog in Visual Studio, visit https://docs.nuget.org/consume/package-manager-dialog.

Note that a project created using the ASP.NET Visual Studio 2013 template does not require adding these packages explicitly.
The following figure describes the core assemblies and their dependencies that get downloaded when the Microsoft.AspNet.WebApi package is installed. There are other NuGet packages that are published by the ASP.NET Web API team (such as CORS support, Help Pages, OWIN host). This figure provides the bare minimal to get started with ASP.NET Web API development:

Now, let’s look at some of the important runtime types that enable the communication and execution in the ASP.NET Web API pipeline.

**DelegatingHandler**

As mentioned earlier, the execution of a request implements a Russian Doll Model. A DelegatingHandler is a runtime type that enables the creation of a handler that can participate in the chain of request and response pipeline for ASP.NET Web API. Although used heavily by the ASP.NET Web API infrastructure, DelegatingHandler is a type defined in System.Net.Http.
DelegatingHandler is derived from HttpMessageHandler, which is the abstract base class for all HTTP message handlers. The most critical method exposed by HttpMessageHandler is the abstract SendAsync method. It is responsible for maintaining the chain of the request and response pipeline. DelegatingHandler can enrich the request message before calling SendAsync; once called, it sends the incoming request to the next handler for processing.

protected internal abstract Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken cancellationToken);

DelegatingHandler acts as a base type for built-in message handlers such as HttpServer. In its base implementation, it enables assigning an inner handler during construction and delegating the SendAsync request to the inner handler, thus enabling a chain between the handlers. Any type that inherits from DelegatingHandler can then participate in the pipeline by ensuring that SendAsync on DelegatingHandler is called.

public class CustomMessageHandler: DelegatingHandler
{
    protected async override Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
{
    // Call the inner handler.
    var response = await base.SendAsync(request,
cancellationToken);
    return response;
}
}

Alternatively, an inherited type may return a response instead of invoking base.SendAsync. For example, a handler generating a response. Another useful scenario could be where a handler detects an exception and wants to stop processing the request.

Note that the call to SendAsync is asynchronous, thus satisfying the "asynchronous to the core" design principle of REST.

All message handlers in ASP.NET Web API are contained in an ordered collection as part of the HttpConfiguration.MessageHandlers collection. What this means is that the handlers are always executed in sequence of their addition. The ASP.NET Web API framework allows for defining message handlers at the configuration level on a per route basis using HttpRoutingDispatcher. We discuss more details about this feature in the Routing and dispatching section.
HttpRequestMessage

HttpRequestMessage is a runtime type defined in the System.Net.Http namespace; it acts as an unified abstraction for all HTTP requests irrespective of client or server. Any incoming message is first converted into HttpRequestMessage before it traverses through the pipeline; this provides an ability to have a strong type HTTP message throughout the pipeline.

An HttpRequestMessage may contain values for the following attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>This specifies the method of the request (GET, POST, PUT, DELETE, OPTIONS, TRACE, HEAD) and returns an HttpMethod type.</td>
</tr>
<tr>
<td>Content</td>
<td>This specifies the body of the request. It is contained in an HttpContent object.</td>
</tr>
<tr>
<td>Headers</td>
<td>This contains all headers in the HTTP request. These can be used to retrieve authorization headers and user agents. The Content property also exposes a Headers collection, which is useful to define header values for the content itself, such as ContentType and MediaFormatters.</td>
</tr>
<tr>
<td>Properties</td>
<td>This represents a dictionary that can contain custom values to be passed as part of the request, such as error details.</td>
</tr>
</tbody>
</table>

HttpResponseMessage

HttpResponseMessage represents the other side of the story and provides an abstraction for all responses generated for corresponding HTTP requests. It is also defined in the System.Net.Http namespace. Any response within the pipeline will be represented as HttpResponseMessage before it gets converted and sent to the caller.

 HttpResponseMessage may contain values for the following attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>This specifies the body of the request. It is contained in an HttpContent object.</td>
</tr>
<tr>
<td>Headers</td>
<td>This contains all headers in the HTTP response. It is a key-value pair collection of the HttpResponseMessage type.</td>
</tr>
<tr>
<td>RequestMessage</td>
<td>This is a copy of the request message (HttpRequestMessage) that leads to the response.</td>
</tr>
<tr>
<td>IsSuccessStatusCode</td>
<td>This is a bool value that indicates whether the HTTP response is successful.</td>
</tr>
</tbody>
</table>
### Getting Started with the ASP.NET Web API

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatusCode</td>
<td>The HTTP status code for the response (for example, a 202 status code would mean success) returns the HttpStatusCode enum.</td>
</tr>
<tr>
<td>ReasonPhrase</td>
<td>This is a string description that corresponds to the status code that has been returned (for example, a reason phrase for a 202 status code will be &quot;Accepted&quot;).</td>
</tr>
<tr>
<td>Version</td>
<td>This is the HTTP protocol version; the default is 1.1, which returns a version type.</td>
</tr>
</tbody>
</table>

HttpRequest also exposes the **EnsureSuccessStatusCode** method, which is especially useful in testing scenarios where a successful response is expected. The method throws an exception if HttpResponseMessage does not return with a success HttpStatusCode in the range of 200 to 209. Note that in case of a failure response, the **EnsureSuccessStatusCode** internally calls Dispose on the stream if the Content property for the response is not null. Essentially, we cannot access the Content stream post with this method when a failure status code is returned.

### ApiController

ASP.NET MVC has the concept of controllers since its inception; conceptually, Web API controllers follow a similar approach to leverage the goodies from the ASP.NET framework. However, Web API controllers are targeted towards providing an abstraction over the HTTP request-response pipeline. Moreover, Web API controllers return only response data in contrast to HTML views in ASP.NET MVC.

All Web API controllers implement the IHttpController interface to classify them as Web API controllers. As we will see in the next section, the Web API request dispatch, and route matching pipeline attempts to look for an IHttpController instead of a particular controller implementation. The following code snippet shows the definition for the IHttpController interface.

```csharp
{
    public interface IHttpController
    {
        Task<HttpResponseMessage>
        ExecuteAsync(HttpControllerContext controllerContext, CancellationToken cancellationToken);
    }
}
```
While we can create our controllers by implementing the IHttpController interface, ASP.NET Web API provides a useful base class that abstracts most of the low-level plumbing required for the request-response pipeline.

A typicalApiController.ExecuteAsync implementation looks like this:

```csharp
public virtual Task<HttpResponseMessage> ExecuteAsync(HttpControllerContext controllerContext, CancellationToken cancellationToken)
{
    if (this._initialized)
    {
        throw Error.InvalidOperation
            (SRResources.CannotSupportSingletonInstance,
            new object[2]
            {
                (object) typeof (ApiController).Name,
                (object) typeof (IHttpControllerActivator).Name
            });
    }
    else
    {
        this.Initialize(controllerContext);
        if (this.Request != null)
            HttpRequestMessageExtensions.RegisterForDispose
                (this.Request, (IDisposable) this);
        ServicesContainer services =
            controllerContext.ControllerDescriptor.
            Configuration.Services;
        HttpActionDescriptor actionDescriptor =
            ServicesExtensions.GetActionSelector(services).
            SelectAction(controllerContext);
        this.ActionContext.ActionDescriptor = actionDescriptor;
        if (this.Request != null)
            HttpRequestMessageExtensions.SetActionDescriptor
                (this.Request, actionDescriptor);
        FilterGrouping filterGrouping =
            actionDescriptor.GetFilterGrouping();
        IActionFilter[] actionFilters =
            filterGrouping.ActionFilters;
        IAuthenticationFilter[] authenticationFilters =
            filterGrouping.AuthenticationFilters;
        IAuthorizationFilter[] authorizationFilters =
            filterGrouping.AuthorizationFilters;
        IExceptionFilter[] exceptionFilters =
            filterGrouping.ExceptionFilters;
```
IHttpActionResult innerResult =
   (IHttpActionResult) new ActionFilterResult
   (actionDescriptor.ActionBinding, this.ActionContext,
   services, actionFilters);
if (authorizationFilters.Length > 0)
   innerResult = (IHttpActionResult) new
   AuthorizationFilterResult(this.ActionContext,
   authorizationFilters, innerResult);
if (authenticationFilters.Length > 0)
   innerResult = (IHttpActionResult) new
   AuthenticationFilterResult(this.ActionContext,
   this, authenticationFilters, innerResult);
if (exceptionFilters.Length > 0)
{
   IExceptionLogger logger =
      ExceptionServices.GetLogger(services);
   IExceptionHandler handler =
      ExceptionServices.GetHandler(services);
   innerResult = (IHttpActionResult) new
   ExceptionFilterResult(this.ActionContext,
   exceptionFilters, logger, handler, innerResult);
}
return innerResult.ExecuteAsync(cancellationToken);

The series of actions performed by the API controller in the preceding code is as follows:

1. The API controller invokes the action selection logic for the HTTP request.
2. It then initializes and invokes the registered authentication filters.
3. Subsequently, it initializes and invokes the authorization filters.
4. Next, the action filters are initialized and invoked followed by the exception filters in case of exceptions.
5. In the last step, it invokes the Parameter Binding and content negotiation for the requests and responses.

Finally, the operation is executed.
### Other important types

There are other important types in the `System.Net.Http` and `System.Web.Http` namespaces that enrich the ASP.NET Web API framework, these are described as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HttpRoutingDispatcher</code></td>
<td>This is the default endpoint message handler, which examines the IHttpRoute of the matched route, and chooses which message handler to call. If Handler is null, then it delegates the control to HttpControllerDispatcher.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>HttpControllerDispatcher</code></td>
<td>This dispatches an incoming HttpRequestMessage to an IHttpController implementation for processing.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>HttpControllerContext</code></td>
<td>This contains the metadata about a single HTTP operation and provides access to the request, response, and configuration objects for the HTTP operation.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>HttpParameterBinding</code></td>
<td>This provides a definition of how a parameter will be bound.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>IContentNegotiator</code></td>
<td>This is the primary interface for enabling content negotiation by selecting a response writer (formatter) in compliance with header values.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>IHttpControllerSelector</code></td>
<td>This assists in selecting a controller during the request pipeline.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>HttpActionSelector</code></td>
<td>This assists in selecting an action in a controller during request execution.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td><code>HttpControllerHandler</code></td>
<td>This passes ASP.NET requests into the HttpServer pipeline and writes the result back.</td>
<td>System.Web.Http</td>
</tr>
</tbody>
</table>
## Getting Started with the ASP.NET Web API

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>MediaTypeFormatter</td>
<td>This is the base class to handle serializing and deserializing strongly-typed objects. This plays a pivotal role in converting requests and responses to their desired formats.</td>
<td>System.Web.Http</td>
</tr>
<tr>
<td>HttpServer</td>
<td>This defines an implementation of HttpResponseMessage, which dispatches an incoming HttpRequestMessage and creates an HttpResponseMessage as a result.</td>
<td>System.Web.Http</td>
</tr>
</tbody>
</table>

We will get into the details of most of these types in later sections and in Chapter 2, *Extending the ASP.NET Web API*.

### Message lifecycle

Now that we have some context of the main APIs involved in the Web API request-response pipeline, let's take a look at how a request flows through the Web API pipeline and how a response is directed back to the client.

At a broad level, the message pipeline can be categorized into three main stages:

![Message pipeline stages](image)

Let's walk through the message pipeline using a sample Web API. The Web API is a simple Hello World API that allows the client to issue a GET request and returns the string *Hello world!* in JSON. For now, don't worry about the implementation details of this Web API. In the next sections, we will get into deeper detail on how to create, test, and deploy a Web API.

<table>
<thead>
<tr>
<th>Request</th>
<th><a href="http://localhost:12356/api/hello">http://localhost:12356/api/hello</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>Hello world!</td>
</tr>
</tbody>
</table>
Host listener

A host listener refers to a component listening for incoming requests. For years, Microsoft Web Components had an inherent dependency of using Internet Information Services (IIS) as the host. The ASP.NET Web API breaks out of this legacy model and allows itself to be hosted outside IIS. As a matter of fact, we can host a Web API in an executable, which then acts as the server receiving requests for the Web API. There are many options available to host a Web API; we will explore some of them in detail in Chapter 2, Extending the ASP.NET Web API. For now, we will focus on using either of the hosting options and enable listening for requests and responding with a response.

Considering the earlier Hello World example, the incoming request will be processed as shown in the following diagram:

In the preceding diagram:

1. The client code issues a GET request using HTTP 1.1 as the protocol. The request is issued to an endpoint, which represents a resource on the server.

   ```
   GET http://localhost:12356/api/hello HTTP/1.1
   Host: localhost:12356
   ```

2. The configured host listener will then receive the request and convert it into `HTTPRequestMessage`. The host inherits from the `HTTPServer` type, which by itself is a message handler and implements `DelegatingHandler`. So, essentially it is just another message handler that delegates the request to the next handler (remember the Russian Doll Model we talked about earlier).

3. The request is then sent through to the routing and dispatching components and then to the controller components for further processing and returning the response.
Routing and dispatching

The routing and dispatching stage primarily involves the execution of a series of message handlers, which process the incoming request and then delegate them to the next message handler for processing. In earlier versions of ASP.NET Web API, we could only configure message handlers globally per application. All the message handlers were added to the `HttpConfiguration.MessageHandlers` collection and were executed for each request. The global configuration was enabled using the `Register` method in the `WebApiConfig.cs` file, which gets added when creating a new ASP.NET Web API project. We talk more about the `WebApiConfig` and `Register` method in the Creating our first ASP.NET Web API section. The following snippet show a sample `Register` method definition:

```csharp
public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
    // add a new message handler to all routes
    config.MessageHandlers.Add(new CustomMessageHandler());
}
```

In the preceding code, we configured `CustomMessageHandler()` to be called for all routes in the Web API. While this worked well for most basic scenarios, it became a challenge for applications where specialization for a route was required. For example, specific authorization handlers for a route. Web API 2 introduced a per route handler flow that allows granularity when routing requests to message handlers. The following code is added to `WebApiConfig.cs` to configure a per route custom handler in the ASP.NET Web API 2:

```csharp
public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/common/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "CustomApi",
        routeTemplate: "api/custom/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.MessageHandlers.Add(new CustomMessageHandler());
}
```
name: "CustomDefaultApi",
routeTemplate: "api/{controller}/{id}",
defaults: new { id = RouteParameter.Optional },
constraints: null,
handler: new CustomMessageHandler() 
); 
}

In the preceding code, we configured CustomMessageHandler, which inherits from DelegatingHandler to a particular route. We also have a global route with API/common that when called will not invoke our custom message handler, it will rather follow a global configuration. The internals of CustomMessageHandler have been omitted since they are not necessary at this stage; all we are concerned here is how the internal routing logic works. We will discuss these concepts in greater detail in Chapter 2, Extending the ASP.NET Web API.

Let's look at how the per-route message handler routing is performed by ASP.NET Web API 2:

1. When a request is sent, the host listener (HttpServer) invokes HttpRoutingDispatcher, which acts as the default endpoint message handler. HttpRoutingDispatcher is responsible for discovering the route and invoking the corresponding handlers for the route.

2. If no route is matched, HttpRoutingDispatcher returns an HTTP standard response code: Not Found (404).

3. If a route is found, the routing handler checks if there is a handler associated with the current route. It does this by checking the Route.Handler property on the route.

4. If no handler is specified for the route, it is assumed that the global configuration will be applied, and the request is delegated to HttpControllerDispatcher for further processing.
5. If a handler is attached to the route, `HttpControllerDispatcher` attempts to invoke the message handler registered for the particular route. The invoked message handler may then return to the main path and delegate to `HttpControllerDispatcher`, as shown:

6. `HttpControllerDispatcher` is a message handler, which is then responsible for selecting and creating the controller before delegating the request to the controller, as shown:

7. If no controller is resolved, a 404 status code is returned by `HttpControllerDispatcher`. 
Once the controller is created successfully, its `ExecuteAsync` is called to delegate processing of the request:

```csharp
httpResponseMessage = await controller.ExecuteAsync(controllerContext, cancellationToken);
```

**Controller processing**

The final and the most exciting stage of the pipeline is processing the request by the controller; this is where all the magic happens to generate the appropriate response for the incoming request. As explained earlier, if the controller inherits from `APIController`, much of the plumbing work is already abstracted from the developer. The following pipeline walks through a controller pipeline that inherits from `APIController`:

1. The controller first determines the action to be invoked:
   ```csharp
   actionDescriptor = ServicesExtensions.GetActionSelector(services).SelectAction(controllerContext);
   ```

2. Next, a series of filters is invoked, which provides authentication and authorization. At any point, if there is a failure response (for example, client not authenticated), the filter can invoke an error response and break the response chain, as shown in the following figure:
3. The next step is model binding. It is a technique provided by the ASP.NET framework to bind request values to existing models. It is an abstraction layer that automatically populates controller action parameters, taking care of the mundane property mapping and type conversion code typically involved in working with ASP.NET request data. We will explore more about model and parameter binding in Chapter 2, Extending the ASP.NET Web API.

4. Next, the pipeline executes the actual action and any associated action filters. This will process the code that we write in our action method for processing incoming requests. The framework actually provides an ability to execute the action filters as a pre and post step to the actual method execution.

5. Once the controller action and the post filters are executed, the final step is to convert the response generated into an appropriate HttpResponseMessage and traverse the chain back to return the response to the client. This is shown in the following figure:
6. Once HttpResponseMessage has been created, the response traverses back through the pipeline, and is returned to the client as an HTTP response.

**Creating our first ASP.NET Web API**

In this section, we will start building our Web API, covering the necessary environment required for creating a Web API. We will also discuss our problem scenario, and then create a Web API from scratch. We will then delve into the details of Microsoft Azure websites and deploy our first Web API in Microsoft Azure.

**Prerequisites**

The following must be configured to run the example scenarios we discuss in this and following chapters:

- **Developer machine**: We need a development environment that can support Microsoft Visual Studio and ASP.NET Web tools. Although we can create ASP.NET Web API solutions from a local PC, for the purpose of this book, we will host a new virtual machine in Microsoft Azure and use it as our development environment. Setting up and managing an environment on Microsoft Azure is so straightforward and elegant, that it is now used as a preferred way to develop applications.

- **A Microsoft Azure subscription**: We need a Microsoft Azure subscription to host our Web API. A free trial is available at http://azure.microsoft.com/pricing/free-trial/?WT.mc_id=A261C142F. For MSDN subscribers, Azure subscription can be activated using their Microsoft Azure credits.
• **Visual Studio 2013**: Once we have the virtual machine up and running, we will need Visual Studio 2013. Any version of Visual Studio Professional 2013 or later should be sufficient for our samples. There is also a free full feature version of Visual Studio referred to as Community Edition, which enables ASP.NET web development. MSDN subscribers can also get access to a set of preconfigured Visual Studio virtual machine images available as part of the Microsoft Azure subscription. The source and samples for this book are built using Visual Studio 2013 Community Edition Update 4.

• **Visual Studio Online**: It is always good practice to use a code management and collaboration system when starting a project. **Visual Studio Online** (VSO) is a Microsoft Azure-based service that provides the capabilities of Team Foundation Server in the cloud. We will use VSO for source control throughout all samples, we will also be using Git as our source control system; Visual Studio Online provides integral support for this. VSO also provides integration with Microsoft Azure for **Continuous Integration** (CI) and **Continuous Delivery** (CD) capabilities.


An alternative is to use GitHub as a source control strategy as well. For the purpose of this book, we will use VSO and Git.

Now that we have the development tools installed and ready, let's quickly take an example to put these to test. We will build an ASP.NET Web API for a fictitious transport service provider that provides consumers with package tracking abilities. The following figure shows how clients interact with the solution:
Creating the ASP.NET Web API project

We will start by creating a new ASP.NET Web Application project in Visual Studio.

Uncheck the application insights to project box for now.

In the new ASP.NET project page, we select the empty template and check Web API in the Add folders and core references for section. Let Authentication be set to No Authentication for now. The empty template skips many other sample controllers and folders such as the MVC helper and documentation folders.
Also, we unselect **Host in the Cloud** under Microsoft Azure for now. We will add this in the next section when we talk about the Microsoft Azure website.

At this stage, the solution should look like this:
A few things have been added to jump start our Web API project:

A static WebApiConfig type has been added to the App_Start folder. The WebApiConfig.cs file is responsible for registering configuration settings for the Web API during startup. Some of the common uses of WebApiConfig.cs are to define default routes, customize the behavior of global and per route matching routes, and define media type formatters.

An important thing to note is the Register method that contains the following code:

```csharp
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
)
```

This method defines the default route template for any request; what this means is that any request that has a route api/{controller}/{id} will be considered a valid route. In this route we have the following parameters:

- `{controller}`: This refers to the name of the controller class without the suffix, for example, a class with name PackageController will be represented as Package in the incoming request
- `{id}`: This is used for parameter binding, these together define the controller and action route for the request

With Web API 2, we now have the option to specify attribute-based routes on each API call to provide granular route definitions for our Web API. We will cover this in more detail in Chapter 2, Extending the ASP.NET Web API.

The Global.asax file has the following defined in Application_Start, this registers a delegate for the WebApiConfig.Register method in GlobalConfiguration. Note that GlobalConfiguration belongs to System.Web.Http.WebHost, which indicates that we are currently hosting our Web API in IIS, as shown:

```csharp
GlobalConfiguration.Configure(WebApiConfig.Register);
```

## Defining an ASP.NET data model

A model in ASP.NET Web API is similar to a data model defined in the ASP.NET MVC framework. It is a type that represents the entity in the application and manages its state. ASP.NET Web API has inherent capabilities to serialize a model automatically to JSON, XML, or any other format, and then write the serialized data into the body of the HTTP response message. The client can then deserialize the object to obtain the response.

For the scope of this chapter, we will keep the data access very simple. We will have an in-memory store and will define an ASP.NET model to represent it. In later chapters, we will fetch data using tools like Entity Framework and also discuss how expose OData endpoints for building query capabilities from our Web APIs.

We follow these steps to create an ASP.NET model for our solution:

1. Right-click on the Models folder. Click on Add and then on New Item.
2. From the Add New Item dialog, add a new C# class. Name it Package.cs.
3. Add properties to the data model. The package data model should look like this:

   namespace Contoso.Transport.Services.Models
   {
   public class Package
   {
   public int Id { get; set; }  
   public Guid AccountNumber { get; set; }  
   public string Destination { get; set; }  
   public string Origin { get; set; }  
   public double Weight { get; set; }  
   public double Units { get; set; }  
   public int StatusCode { get; set; }  
   }
   }

Defining an ASP.NET Web API controller

We will now implement a Web API controller that will allow clients to retrieve packages based on a unique package ID. We will use the empty controller template in Visual Studio and then add the following code to our controller type:

1. Right-click on the Controllers folder in the Contoso.Transport.Services project. Click on Add and select Controller.
2. Choose an empty Web API 2 controller. Name the controller `PackageController`.

3. Add a reference to the `Contoso.Transport.Service.Models` namespace:
   ```csharp
   ```

4. Replace the code in `PackageController` with the following code snippet:
   ```csharp
   namespace Contoso.Transport.Services.Controllers
   {
   public class PackageController : ApiController
   {
   private static IEnumerable<Package> packages;
   
   public Package Get(int id)
   {
   ```
return packages.SingleOrDefault(p => p.Id == id);
}

protected override void
Initialize(HttpControllerContext
controllerContext)
{
    base.Initialize(controllerContext);

    // Create package for testing purposes. In the real
    world use a repository pattern or entity framework to fetch this
data.
    GenerateStubs();
}

private static void GenerateStubs()
{
    packages = new List<Package>
    {
        new Package
        {
            Id = 1,
            AccountNumber = Guid.NewGuid(),
            Origin = "CA",
            Destination = "TX",
            StatusCode = 1,
            Units = 1,
            Weight = 2.5,
            Created = DateTime.UtcNow,
        },
        new Package
        {
            Id = 2,
            AccountNumber = Guid.NewGuid(),
            Origin = "AZ",
            Destination = "AL",
            StatusCode = 1,
            Units = 2,
            Weight = 1,
            Created = DateTime.UtcNow.AddDays(-2),
        },
        new Package
        {

        
```
Id = 3,
AccountNumber = Guid.NewGuid(),
Origin = "FL",
Destination = "GA",
StatusCode = 3,
Units = 1,
Weight = 2.5,
Created = DateTime.UtcNow,
};

The preceding code is relatively straightforward; it creates an in-memory store for packages and a GET operation to search for a package based on its ID. A few important things to consider are discussed next.

We override the Initialize method for the controller. The method is called when the controller is initialized by the Web API pipeline. It can be used for setting up member variables and populating metadata for a controller.

Note that because this is a sample scenario, we are using an in-memory collection for the data entities. In real-world scenarios, we should use tools such as Entity Framework along with the Repository pattern to make the code more maintainable. The later part of this book discusses the Entity Framework.

The return type here is our Package entity, when a response is generated, ASP.NET Web API will serialize the entity using the MediaFormatter configured for the project. To facilitate control over the HTTP Response, ASP.NET Web API also provides additional types such as IHttpActionResult and HttpResponseMessage. We discuss these in Chapter 2, Extending the ASP.NET Web API.

The signature of the GET operation matches HttpRoute that we defined in the WebAPIConfig.cs file (api/{controller}/{id}). The signature allows us to determine which HTTP verb needs to be executed for the incoming request. We may define multiple operations for a controller, but each needs to match to a particular route. This approach is also referred to as routing by convention. If no route is found, an exception is thrown. ASP.NET Web API 2 also provides specific routes per controller or even per action through attribute routing. We will discuss these approaches in detail in Chapter 2, Extending the ASP.NET Web API.
Next, build the project and that is it! We just created our first Web API. We now have a controller that can search for packages based on IDs and return the package model as a response. In the next section, we put our Web API to test in order to verify our operations.

Testing the Web API

Now that we have a Web API created, we will look at options to validate our Web API functionality. Testing is a crucial stage in the creation of Web API development and publishing. There are a couple of different ways of doing this. We can just test it by requesting the URL in a browser or write code and leverage the `System.Net.HttpClient` type. This section discusses both of these approaches.

Testing in a browser

Testing a Web API is as simple as developing it, since each action in a Web API controller represents a resource, we can just type the URL of the resource and fetch the results.

1. Press F5 in Visual Studio to launch the Web API in a browser.
2. Visit the following URL:
   
   ```
   http://localhost:<PORT>/api/package/1
   ```

   Note that the port will be allocated by IIS Express and will be different for each installation.

3. This yields a result similar to the following in the browser:

   ```
   {
   "Id": 1,
   "AccountNumber": "43a2a3eb-e0b8-4840-9e5e-192214a79d58",
   "Destination": "TX",
   "Origin": "CA",
   "Weight": 2.5,
   "Units": 1,
   "StatusCode": 1,
   "Created": "2015-03-16T23:57:33.1372091Z",
   "Properties": null
   }
   ```
What happened here?

The client (in this case, the browser) initiated a GET request by hitting the URL:

```
GET /api/package/1 HTTP/1.1
Host: localhost:49435
Cache-Control: no-cache
```

When the server received the request, it scanned all controllers to check for a matching route. When the route was found, `PackageController` was selected to respond to the request, and its `Initialize` method was invoked. The GET action in `PackageController` was then identified as a route match for the HTTP GET request. The method gets called, and 1 is passed as the parameter.

**Testing with HttpClient**

In the previous section, we discussed how to use a browser to test our Web API. In scenarios where we call our Web API from within a business logic or client application, this may not work. Fortunately, the `System.Net.HttpClient` type can be used to invoke an HTTP-based Web API from a .NET client.

We will use our Visual Studio test project that we created earlier to demonstrate this example:


   The Visual Studio ranger’s team has built an extension to generate unit tests for class files. It is a useful tool to generate Test methods for multiple methods in the class files. The extensions can be found here: https://visualstudiogallery.msdn.microsoft.com/45208924-e7b0-45df-8cff-165b505a38d7.

2. Add the following assembly references to the project:
   
   - `System.Net.Http` assembly
   - `Contoso.Transport.Services`

3. Add the following NuGet packages to the project:
   
   - `Newtonsoft.JSON`

4. Replace the code in `PackageControllerTest.cs` with the following code:

   ```csharp
   using System;
   using System.Net.Http;
   ```
using Newtonsoft.Json;
using System.Threading.Tasks;
using System.Net.Http;

namespace Contoso.Services.Tests
{
    [TestClass]
    public class PackageControllerTests
    {

        [TestMethod]
        public async Task FindPackageByIdNotNullTest()
        {
            var packageid = 1;
            var packageUrl =
                string.Format("http://localhost:<PORT>/api/package/{0}", packageid);
            using (HttpClient client = new HttpClient())
            {
                var response = await client.GetStringAsync(packageUrl);
                Assert.IsNotNull(response);
                var package = await Task.Factory.StartNew(() =>
                    JsonConvert.DeserializeObject<Package>(response));
                Assert.IsNotNull(package);
                Assert.AreEqual(packageid, package.Id);
            }
        }
    }
}

5. Change packageUrl to the URL of the created Web API.
6. At this point, ensure that the Web API is running. Please refer to the Testing in a browser section for more details.
7. Build the test project.
8. Right-click on **Test Method** and click on **Run Test** (*Ctrl* + *R* + *T*).

9. The test explorer should display the success results, as follows:

We verified our Web API using a .NET client as well. `HttpClient` exposes a set of utility methods that allow us to perform various HTTP operations. In this case, we used the `GetStringAsync` method that creates an HTTP GET request and returns the response as a string.
Committing changes to Git

In the previous section, we built and tested our Web API. Now, we discuss how to use Visual Studio Online and Git for source control management. Note that this section assumes that we have a Git repository created and available. It does not give a walkthrough of setting up a Git repository within VSO.

The Microsoft Virtual Academy has an excellent tutorial for using Visual Studio Online with Git it is highly recommended to understand these concepts before proceeding with this section: http://www.microsoftvirtualacademy.com/training-courses/using-git-with-visual-studio-2013-jump-start.

We will now commit the change to Visual Studio Online. Visual Studio 2013 provides a rich integration with Git so we can commit the changes without opening the Git command prompt:

2. A dialog box is displayed with options for available source control systems. Select Git and click on OK.
3. Right-click on the solution again and click on Commit.
4. Enter comments for the changes, and click on **Commit** to create a local commit.

5. We can now use the **Sync/Push** Git commands from Visual Studio to commit our changes remotely:

![Sync/Push Git commands from Visual Studio](image)

---

### Deploying the ASP.NET Web API using Azure Websites

Microsoft Azure Websites is a **Platform as a Service (PaaS)** offering from Microsoft, which allows publishing web apps in Azure. The key focus when building Microsoft websites is to enable a true enterprise-ready PaaS service by allowing rapid deployment, better manageability, global scale at high throughput, and support for multiple platforms. The development of Azure Websites started about 3 years ago under the internal project name **Antares** with the primary intent to create a cheap and scalable web hosting framework for Microsoft Azure. The main target was shared and cross-platform hosting scenarios. The first preview release for Azure Websites came out in June 2012 and went **General Availability (GA)** in mid-2013.

Azure Websites was recently renamed and is now part of Azure App Services. REST based service development is now referred to as **API apps**. For the scope of this book, we will focus on the existing service implementation. To learn more about App Services, please refer [http://azure.microsoft.com/en-us/services/app-service/](http://azure.microsoft.com/en-us/services/app-service/).

There are other PaaS offerings available in Microsoft Azure, such as cloud services that provide Web and worker roles. There is also Microsoft virtual machine through the **Infrastructure as a Service (IaaS)**, so why choose websites?
Let's look at some of the features provided by Azure Websites, which make them an appealing candidate for Web API and web app deployment:

- **Cloud-First by design**: This is a no-brainer, Azure Websites was started to eliminate the dependency of IIS for cloud developers. Azure Websites leverage SQL Server as a configuration backend, which allows to achieve a greater website density in multitenant environments. It is also intended as a stateless and scalable server architecture.

- **Dynamic provisioning**: The allocation of resources for Azure Websites is determined based on dynamic rules to ensure optimum allocation and performance of the system overall.

- **Network share support**: All instances can leverage shared content storage, this ensures optimum reliability in case the frontend machine goes down.

- **Stateless servers and smart load balancing**: Following the cloud design patterns, all VMs are stateless by design.

- **Build on existing PaaS platform**: Internally, Azure Websites are hosted in PaaS VMs, so it automatically leverages most of the goodies provided by the cloud services platform today. The following figure shows the main components that make up Azure Websites:
As we can see in the preceding figure, Azure Websites are built on existing technologies, such as Azure storage and web and worker hosting model. It adds additional abstraction to enable rapid deployment and easy configuration management through a set of hosting site and site content databases. Furthermore, Azure load balancers are used in conjunction with IIS Application Request Routing (ARR) load balancers to achieve high availability and Hyper Scale. Azure Websites provides the following benefits:

- **Development support:** Websites can be used for developing solutions in an array of languages such as ASP.NET, Java, PHP, Python, and Node.js. Moreover, the development platform does not need to be Windows, OSX, or Linux. CMS solutions such as WordPress, Drupal, and Joomla are fully supported. The integration with code management tools, such as Git, and source control systems, such as VSO and GitHub, is seamless. To add the cherry to the cake, the tools provided by Visual Studio 2013 and above are so easy to use that development, testing, and deployment just take minutes.

- **Built for DevOps:** Azure Websites are targeted to reduce the overall time taken to market and provide an effective TCO. Some of these features include high available environment with automatic patching support, automatic scale, automatic disaster recovery and backups, comprehensive monitoring and alter, and quick and easy deployment.

With such great features, is there any scenario where Azure Websites might not be a good fit?

Well, there are some scenarios where we may be better off using Azure Cloud Services and perhaps even Azure Virtual Machines:

- **Low-level machine access:** Although Azure Websites provide a standard tier that allows its deployment VM, customization that can be done on that VM is still restricted. For example, elevated privileges are not supported as of writing this.

- **Background processes:** There are scenarios where we want to have backend tasks that run complex processing jobs such as math calculations. Worker roles in cloud services seem appropriate for these scenarios. Having said that, from an application perspective, we can still leverage websites as a frontend. For example, a Web API hosted in Azure Websites may push calculation requests to an Azure Service Bus queue, and a backend worker role can then listen to these requests and process them. Another aspect that is supported by Azure Websites is WebJobs, which is associated with an Azure Website, and as of today, can be used for processing lightweight background tasks, such as startup tasks. WebJobs may become more powerful in the future and could then be considered as an alternative for worker roles.
• **Virtual Network support**: Since Azure Websites VMs are controlled by Microsoft PaaS infrastructure, there are minimal options to support Virtual Networks or customer domain integration. As of September 2014, a preview release of supporting websites on Virtual Network has been added as a feature. For the latest information on Virtual Network support, please visit http://azure.microsoft.com/en-us/documentation/articles/web-sites-integrate-with-vnet/.

• **Remote desktop**: As of writing of this book there is no support for remote desktop on Azure Websites. It may not be a big limitation though, as websites now support remote debugging from within Visual Studio for all pricing tiers.

• **In-role cache**: Due to the stateless nature of Azure Websites, in-role caching is not supported. However, we can use a distributed cache such as Redis and make it work with websites.

Note that some of the scenarios mentioned here might not be available as of today, but the team may provide support for these in future releases. As a rule of thumb, consider these as recommendations and not protocols. Considerable planning should be conducted based on customer requirements and timelines before making a decision to choose any of the platform options.

### Deploying to Azure Websites

Now that we have a fair understanding of Azure Websites' capabilities, let's deploy the Web API we built in the previous section to an Azure Website.

This sample assumes that the user is signed into an Azure account in Visual Studio. It is required to view the Azure subscription when creating the ASP.NET Web API project. For more information on Azure tools for Visual Studio, please visit http://msdn.microsoft.com/en-us/library/azure/ee405484.aspx.

Since we did not associate our Web API project to an Azure Website at the time of its creation, we will now provide details on the Azure subscription and website where we want the deployment to be hosted:

1. Right-click on the Web API project. Click on **Publish** to open the Publish wizard, and select **Microsoft Azure Websites** as the publish target.
2. In the **Microsoft Azure Websites** dialog, click on **New** to create a new Azure Website. Here, we provide details about our Azure subscription and give a name to the website. When deciding on a name, ensure that it is unique; the **Create Site** dialog box validates the fully qualified URL to ensure that there are no existing sites with this name. In case the site name already exists, an error is thrown by the **Create Site** dialog box.

3. By default, the domain name is set to **azurewebsites.net**. It is the default domain for all tiers; we can also provide a custom domain for our deployments (for example, **www.mywebsite.com**).
4. Once the Website is created, we are ready to publish our Web API. Azure Websites provides the following publish methods for web applications:

- **Web deploy**: This is also known as msdeploy. It streamlines the deployment of web application to Azure Websites. It enables the packaging of Web application content, configuration, databases, and any other artifacts, which can be used for storage or redeployment. If the package needs to be redeployed to a different environment, configuration values within the package can be parameterized during deployment without requiring modifications to the packages. Use this option to delegate deployment to Azure Websites.

- **WebDeploy package**: We can use this option when we have an existing WebDeploy package.

- **FTP**: This allows to deploy source from an FTP server.

- **File system**: This deploys source from a local filesystem location.

5. Once the deployment is complete (should not take more than a few seconds), we will have our Web API running in the cloud.

6. To test the deployment, replace the local domain name we used in the **Testing in a browser** section with the cloud domain name of the Azure Website as shown:

```json
Request
http://dotnetbook.azurewebsites.net/api/package/1

Response
{
    "Id": 1,
    "AccountNumber": "ac09ae02-00cf-426d-b969-909fae655ab9",
    "Destination": "TX",
    "Origin": "CA",
    "Weight": 2,
    "Units": 1,
    "StatusCode": 1,
    "History": null,
    "Properties": null
}
```

If we now browse to the Azure management portal and go to the Azure Website we just created, we can right away see incoming requests and how the traffic is being monitored by Azure Websites similar to how it is shown in the following diagram; now that is true PaaS!
The new Azure portal (currently in preview) provides more detailed statistics about the resources leveraged by Azure Websites. These include storage, network, and compute. The preview portal is accessible at https://portal.azure.com/.

Continuous Deployment using Azure Websites

In the previous section, we used Visual Studio tools to deploy our Web API in Microsoft Azure Websites. The Visual Studio deployment option works well for one-off deployments or small development projects. However, in real-world scenarios, customers prefer integrating builds with automated deployments. Azure Websites provides continuous deployment from source code control and repository tools such as BitBucket, CodePlex, Dropbox, Git, GitHub, Mercurial, and Team Foundation Server. We can, in fact, do a Git deployment from a local machine!

For the purpose of this sample, we use VSO and Git as our repository. Let’s take a look at how we can configure Visual Studio Online for our Continuous Deployments (CD):

1. Go to the Azure portal and browse to the Azure Website we just created. From the dashboard, select setup deployment from source control. When using the new preview portal, this option will be in the Deployment section as Set up continuous deployment.
2. Select VSO to designate the source code and then select the repository to deploy:

Azure will now create a link to Visual Studio Online. Once the link is established, when a change is committed, it will get deployed to Azure Website automatically. The link creates a build definition for the solution and configures it to run in CD mode. If we go to the build definition in Visual Studio, we can see that a new build definition is added. The linkage also sets the deployment attributes for the build. It ensures that a commit on the solution triggers a deployment. Note that if any unit test projects, which are part of the solution will also automatically get triggered.
3. Let's make a change to our existing sample to verify this. We will add a new attribute to our Package ASP.NET data model that we created earlier. This attribute will get populated with the current UTC timestamp when the package is created:

```csharp
namespace Contoso.Transport.Services.Models
{
    public class Package
    {
        public int Id { get; set; }
        public Guid AccountNumber { get; set; }
        public string Destination { get; set; }
        public string Origin { get; set; }
        public double Weight { get; set; }
        public double Units { get; set; }
        public int StatusCode { get; set; }
        public DateTime Created { get; set; }
    }
}
```

We will also update the in-memory collection in PackageController to ensure that a created timestamp is associated with each record. The code for updating the controller is left as an exercise.
4. When we make a commit and push to the Visual Studio Online repository, the build definition will invoke a new build, which will also update the deployment in our existing Azure Website. This can be verified by looking at the build report:

![Build Succeeded](dotnetbook_CD_20140914.4 - Build succeeded)

View Summary | View Log | Open Drop Folder | Diagnostics | Actions

Nikhil Sachdeva triggered dotnetbook_CD (dotnetbook) for branch refs/heads/master (9eb33bd)
Ran for 2.2 minutes (Hosted Build Controller), completed 6 seconds ago

Latest Activity
Build last modified by Elastic Build (niksaccdev) 6 seconds ago.

Request Summary
Request 5, requested by Nikhil Sachdeva 3 minutes ago, Completed

Deployment Summary
Your Website was deployed to: http://dotnetbook.azurewebsites.net/

Summary
Debug | Any CPU
0 error(s), 0 warning(s)
▷ C:\src\chapter\Contoso.Chapter1.sln compiled
▷ 1 test run completed - 100% pass rate
No Code Coverage Results
Our Web API is now successfully deployed in Microsoft Azure Website and configured for CD.

**Summary**

This chapter provided an overview of the what, why, and how of an ASP.NET Web API. ASP.NET Web API is a next generation service that enables support for a first-class modern HTTP programming model. It provides abundant support for formats and content negotiation, and request validation.

We talked about the foundational elements of a Web API. We then discussed its usage scenarios and components, and also discussed the classes that make up the API. We went through the request-response pipeline and walked through the message lifecycle of a request. We then created our first simple Web API for a simplified customer scenario and walked through its development, testing, and deployment using Azure Websites. In the next chapter, we will look into advanced capabilities of ASP.NET Web API, and see how these capabilities can be used to customize different aspects of the ASP.NET Web API request-response pipeline.
Where to buy this book

You can buy Building Web Services with Microsoft Azure from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.