ASP.NET WEBHOOKS

SUCCINCTLY

BY GAURAV ARORA

SUCCINCTLY E-BOOK SERIES

Syncfusion

ASP.NET WebHooks Succinctly

By

Gaurav Arora

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: Nick Harrison
Copy Editor: Courtney Wright
Acquisitions Coordinator: Morgan Weston, social media manager, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

In the memories of

My angel

Kanchan
(1986 - 1997)

About the Author

Gaurav Arora has completed an M.Phil course in computer science. He is a Microsoft MVP, certified as a scrum trainer/coach, XEN for ITIL-F, and APMG for PRINCE-F and PRINCE-P. Gaurav serves as a mentor at IndiaMentor, webmaster of dotnetspider, contributor to TechNet Wiki, and co-founder of Innatus Curo Software LLC. In the 19+ years of his career, he has mentored thousands of students and industry professionals.

He is a lifetime member of the Computer Society of India (CSI) and is involved in technical communities such as CodeProject, dotnetspider, and IndiaMentor, where he is a mentor.

Gaurav is a writer, blogger, speaker, and open-source contributor. He is an active contributor of ASP.Net WebHooks; his contribution is listed here.

You can find him on Twitter @g_arora, LinkedIn, and on his blog.

Preface

Why did I write this book?

I wrote this book for all those self-learners who really want to make something different (including myself).

During my talk in Delhi, India on ASP.NET WebHooks, I faced many questions from people wanted to know how they can write their own API that supports ASP.NET WebHooks. There is plenty of material (to start) available on WebHooks, but in this book I have crafted this material so that anyone can follow the step-by-step guidelines to write their own ASP.NET WebHooks Receiver/Sender.

Who is this book for?

This book is intended for practical, day-to-day programmers— developers, architects, consultants, solution-providers, etc. The current version of this book requires a few basics to understand the code:

- Understand basic concepts of C# and ASP.NET
- Understand basic idea of OOPs
- Can write or represent code, program, or code-snippet
- Understand basic idea of REST

Yes, that is all that's really required toget started with this book.

How and where to get source code?

Each chapter has source code and all is in working/running condition. The complete source code is available here.

Chapter 1 Introduction

Today we live in a world where we try to have automation make it simpler for us to live and do the hard part for us in less time with greater efficiency. We cook using machines, which reminds me of the robot chefs becoming the old news already

Now, imagine a scenario where we are working on an Enterprise application with millions of activities and events being registered as a result of dynamic actions. And many of these activities resulting in a cascading set of database actions with events already scheduled upon the new data being generated. As an architect or the lead of the team working on this – I am already being bombarded with thoughts whether to take care of the event first or the data which is yet to be generated. Let's simplify.

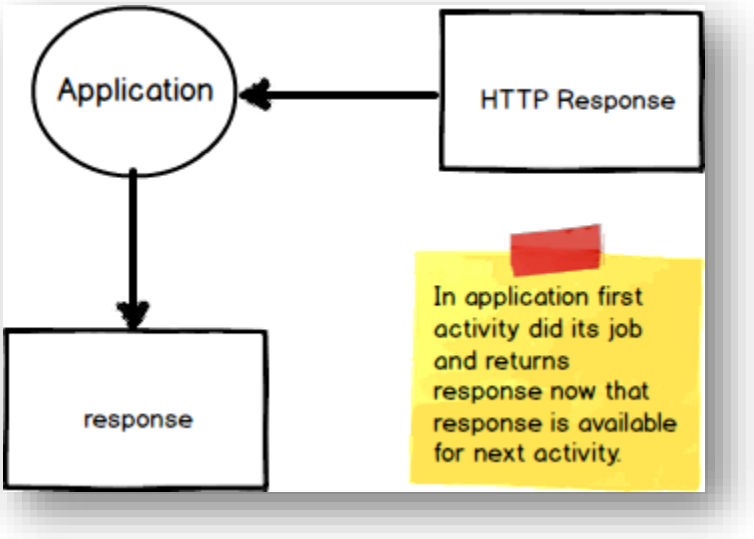

Figure 1: Pictorial view

Figure 1 depicts the story for us to simplify. In our application, the first activity completed its job and returns *response*, which is available for next activity. This response becomes an input for the next activity. Let us call this **`Callback`** for the first step towards simplification of this scenario.

Callback function

In the preceding section, we discussed **`Callback`** in a layman language. Let's now understand what a callback function is, in our bid to further simplify this as our next step.

Simply defined, a callback function is:

- A function with a prospect of generating a “callback” that can be passed to another function, which can further wait on this function to finish processing before beginning its own execution
- Called by being passed as an argument to another function (that is, a function would be an argument of another function)
- Invoked with an event (for example, an HTTP post)

The following is a graphical representation of the last bullet:

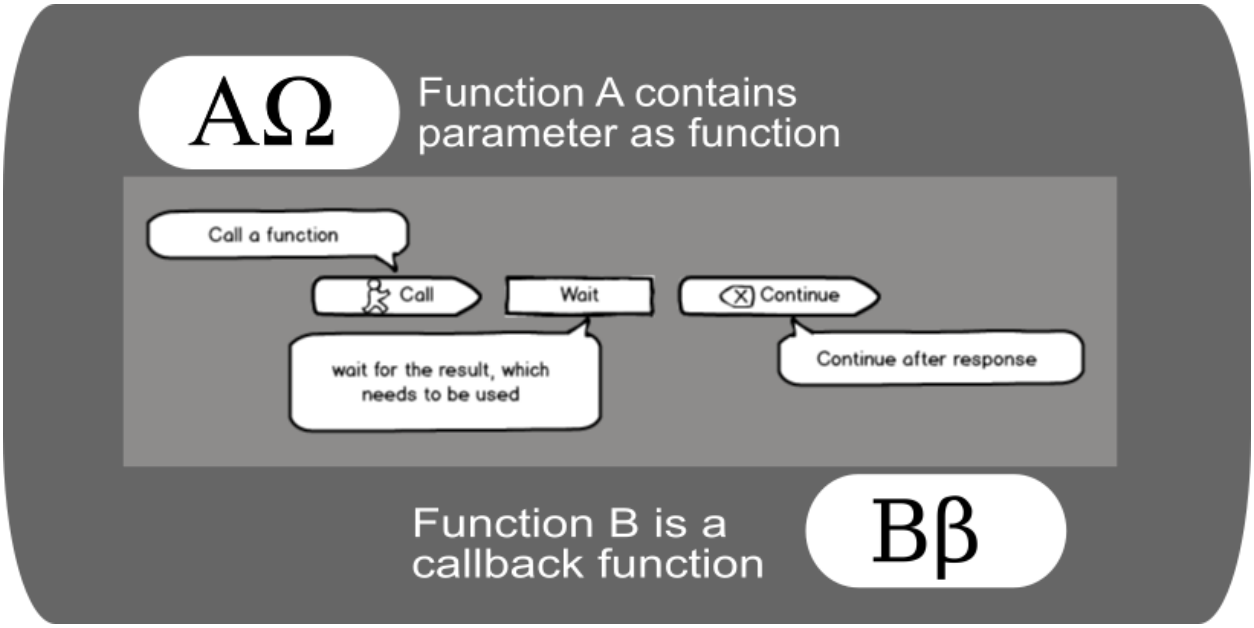

Figure 2: Pictorial view of callback flow

Here we are depicting function A being called with a parameter, which happens to be function B. Before starting its own execution, function A decides to check out the result expected from function B by making a call to it. Voila—B is our callback function.

Code Listing 1 Simple callback function

```
<!DOCTYPE html>

<html>

<head>

<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></s
cript>

<script>

$(document).ready(function(){

    $("button").click(function(){
```

```
        $("div").hide("drop", function(){
            alert("Hi! I called, callback.");
        });
    });
});
</script>
</head>
<body>

<button>Callback</button>

<div>Contents in this area will be hide using callback function.</div>

</body>
</html>
```

In Code Listing 1, we used jQuery's **hide** method, which simply hides the **div**. So, here first, the **click** function of the button waits until **div** gets hidden, and then shows **alert**.

Note: A complete discussion of the callback function is beyond the scope of this book.

As per Wikipedia, a **callback** is "a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time."

Publisher/subscriber model (aka pub/sub model)

In the last section we introduced you to the **callback** function. Let's look at another aspect of our original challenge associated with events and their subscription.

"How an application can send the notifications/messages to applications who are interested in receiving without knowing the identities of receivers"

You guessed it right—this is the pub/sub model. Many of you would already be aware of this. Without going into the complexities and vast possibilities of this model, let us dive in and see how this is done in WebHooks. The publisher/subscriber (or pub/sub) model is one of the solutions applied commonly to this challenge.

As an example, consider a scenario where a few applications need to receive a message from an application that is not in their scope (that is, it is not subscribed to for any events). In this scenario, we need at least one mediator to facilitate communication between both of the applications.

It's easy to think of more complex scenarios where an application requires messages from more than one applications (or vice-versa).

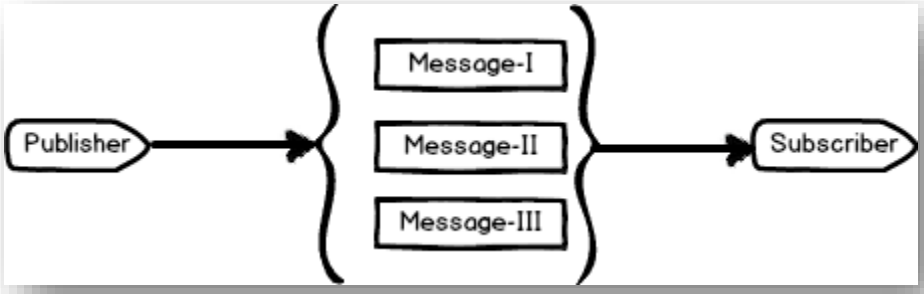

Figure 3: Pictorial view of pub/sub

In Figure 3, there is one publisher (application) and one subscriber (or receiver), which is receiving three types of messages. The Microsoft Biztalk Server, which uses the publisher/subscriber pattern internally, is one example of this model.

WebHooks

Simply put, WebHooks are user-defined **`HTTP Callbacks`**, which are configured with an event and will invoke an event trigger (callback). The magic about it is that the source of the event could lie in the same application or be external. Callbacks can be configured and managed by users of the applications and developers.

Tip: WebHook presence means intention to perform an action. In other words, WebHooks are primarily the wiring of the right events containing the actual code that is intended to be executed.

How it works

Consider working with REST API, where the API is called when the triggering event occurs (say, HTTP **POST**). In simple words, end users are subscribing to a specific trigger, which will be notified via an HTTP **POST** by the developers. WebHooks gives you a great deal of power with minimal setup because notification over HTTP opens up a vast and thriving ecosystem of Web applications.

As per Wikipedia: "A webhook in web development is a method of augmenting or altering the behavior of a web page, or web application, with custom callbacks. These callbacks may be maintained, modified, and managed by third-party users and developers who may not necessarily be affiliated with the originating website or application."

Note: The term "WebHook" was introduced by Jeff Lindsay in 2007.

Real-life scenarios from a developer's perspective could be like pushing code in repository, continuous build, and posting a data over HTTP **POST**. A WebHook could indicate that a file has changed in Dropbox, a code change has been committed in GitHub, a payment has been initiated in PayPal, or a card has been created in Trello, for example.

ASP.NET WebHooks

ASP.NET WebHooks are the implementation of WebHooks in ASP.NET. Currently, ASP.NET WebHooks are in preview (not available for ASP.NET Core). Implementation in ASP.NET supports both sender and receiver. Under the hood, it uses ASP.NET Web API to send and receive notifications.

How it works

The complete cycle of publishing and subscribing in ASP.NET WebHook goes like this:

- Sender exposes events for client to subscribe
- Receiver subscribes by registering a WebHook

Yes, it's that simple. If this doesn't make sense yet, just try to stay along for a few more chapters, and it will be worth it.

Conclusion

In this chapter we have explored WebHooks and the publisher/subscriber model.

Chapter 2 Working with WebHooks

One of my favorite aspects of ASP.NET WebHook is that it is open source, and you can easily get its source code to customize and update as per your own requirements.

As we will see shortly, getting started with WebHooks is easy. But before we dive into that, we need to first understand a few concepts like WebHook receivers, WebHook receiver URIs, and WebHook Handler.

WebHook receivers

A package/module created to receive WebHooks from others is called a WebHook receiver. Some senders might require simple configurations, while others ask for complex configurations. There might be some extra steps involved during configuration, like registration and verification. This also depends upon the security model that the sender is using. This is a choice that is to be made by the sender.

Some senders work on the push-to-pull model, where the HTTP **POST** request does not contain any data, but contains reference to the event triggered. In this scenario, we might require additional data such a, a separate request on the basis of triggered event to pull the data separately for our use.

Microsoft ASP.NET WebHooks are meant to both accept and verify the WebHooks from a particular sender. A single WebHook receiver can support more than one WebHook based on their own configuration settings. In other words, we can say that if we configured 10 Bitbucket repositories for WebHook, then our one Bitbucket WebHook receiver can accept WebHook from all 10 of these repositories.

WebHook receiver URI

All WebHook providers need different information at the time of configuration for the WebHooks; you might require a different URI for different WebHook receivers. Table 1 describes the different URI formats for various WebHook receivers.

Table 1: URI formats – WebHook receivers

URI formats for various WebHook receivers	
Receiver	**URI format**
Bitbucket	`https://<host>/api/webhooks/incoming/bitbucket/?code=secret_key`
Dropbox	`https://<host>/api/webhooks/incoming/bitbucket/?{id}`
GitHub	`https://<host>/api/webhooks/incoming/bitbucket/{id}?code=secret_key`

URI formats for various WebHook receivers	
Receiver	**URI format**
Instagram	`https://<host>/api/webhooks/incoming/instagram/{id}`
MailChimp	`https://<host>/api/mailchimp/?code=secret_key`
MyGet	`https://<host>/api/webhooks/incoming/myget?code=secret_key`
Slack	`https://<host>/api/webhooks/incoming/slack/{id}`
Stripe	`https://<host>/api/webhooks/incoming/stripe/{id}?code=secret_key`
VSTS	`https://<host>/api/webhooks/incoming/vsts?code=secret_key`
Zendesk	`https://<host>/api/webhooks/incoming/zendesk/{id}?code=secret_key`

A typical URI would be in the form of:

`https://<host>/api/webhooks/incoming/<receiver>/{id}`

Where:

- **`<host>`** is the public URL of your application
- **`<receiver>`** is the name of the receiver (in our example, it is **`bitbucket`**)
- **`{id}`** is an identifier (and optional in most of cases), which can be used to identify a particular WebHook receiver configuration

WebHook handler

A **`WebHookHandler`** is an abstract class and implements **`IWebHookHandler`** interface. While creating the application or our handler, like the **`Bitbucket`** handler in our case, do not use **`IWebhookhandler`** interface, but directly use **`WebHookHandler`** abstract class.

WebHook requests can be processed by one or more handlers, which are called in order based with the use of their own order property from lowest to highest (between 1 to 100).

Tip: Order is an integer, and DefaultOrder value is 50.

Importance of setting a response for WebHook handler

There might be some scenario where we have to send back a response to the originating API. In order to do so, we can set the **`Response`** property for the handler. A good example to set **`Response`** property would be HTTP status code **`410 – Gone`**. This indicates that WebHook is not active, and no further requests can be entertained.

Tip: The Response property is optional for the handler, so if a response is not required by the business case, leave it out.

Consuming WebHooks

Now let's put it all together. There are two ways for you to get started with this technology and start consuming the WebHooks.

- NuGet – Add a reference, and you are all set to go
- The examples provided can be an entry point for you. However, you would need to set up up the WebHook source repository to get these examples to compile correctly. The reason for this requirement is that the samples use the source code, and not the NuGet packages.

Creating a sample application using NuGet packages

All WebHooks are available in NuGet packages; you just need to get the package from the NuGet resource.

Tip: All NuGet packages are available at: Microsoft.AspNet.WebHooks.

Note: You can also get the advantage of Visual Studio Extension—refer here for details.

Parts of NuGet packages

The NuGet packages have three modules:

- A module named Common, which is shared between senders and receivers. It can be found here.
- A module specifically developed for a scenario where you want to send your own WebHooks to others and named as Sender. It can be found here.
- A module named Receivers, which provides support to receive WebHooks from others. It can be found here.

Creating a WebHook receiver

In this section, we will discuss in detail how to use Bitbucket WebHook receivers in your own application. We will refer to a sample application and use NuGet packages instead of referring directly to our projects.

First, let's create a receiver project; in this application we are going to use Visual Studio 2015.

1. Create new project from within Visual Studio.

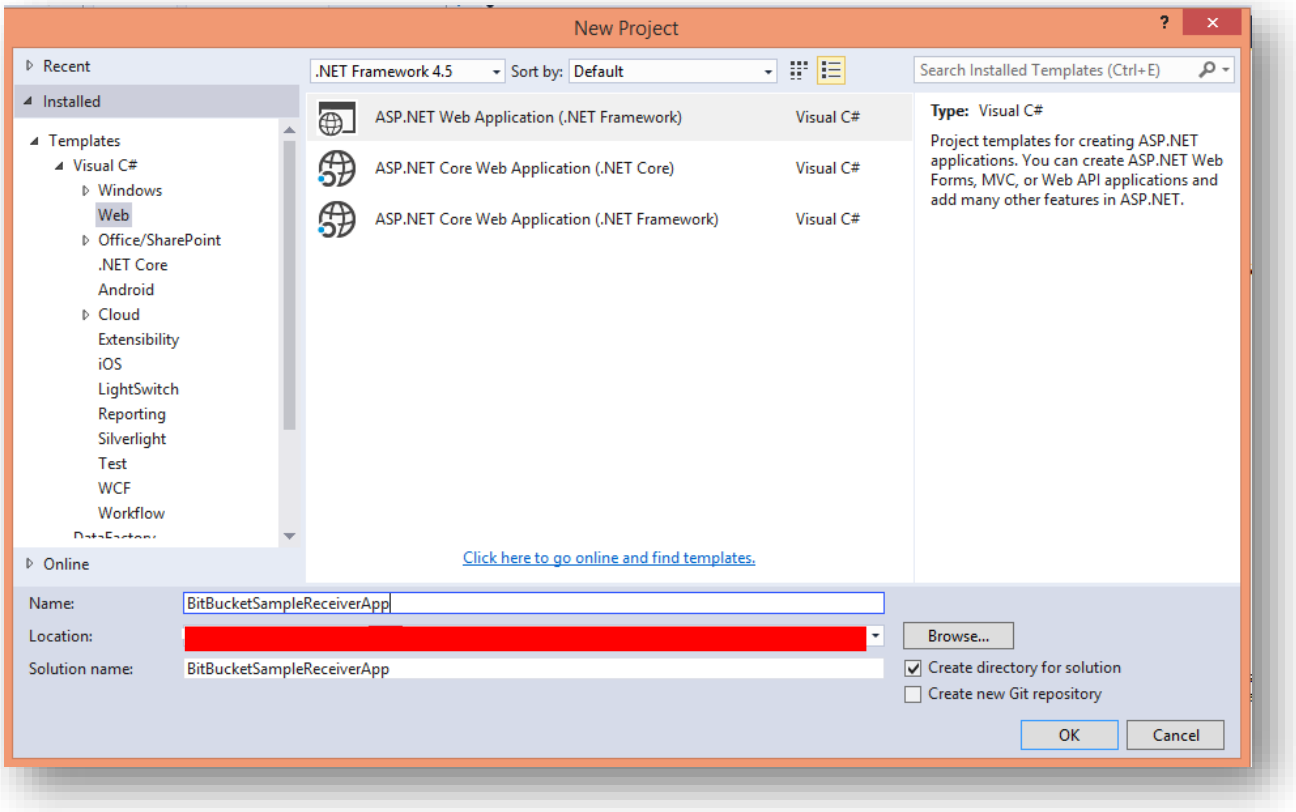

Figure 4: Create new project

2. Choose an empty project with the Web API template, as shown in Figure 5.

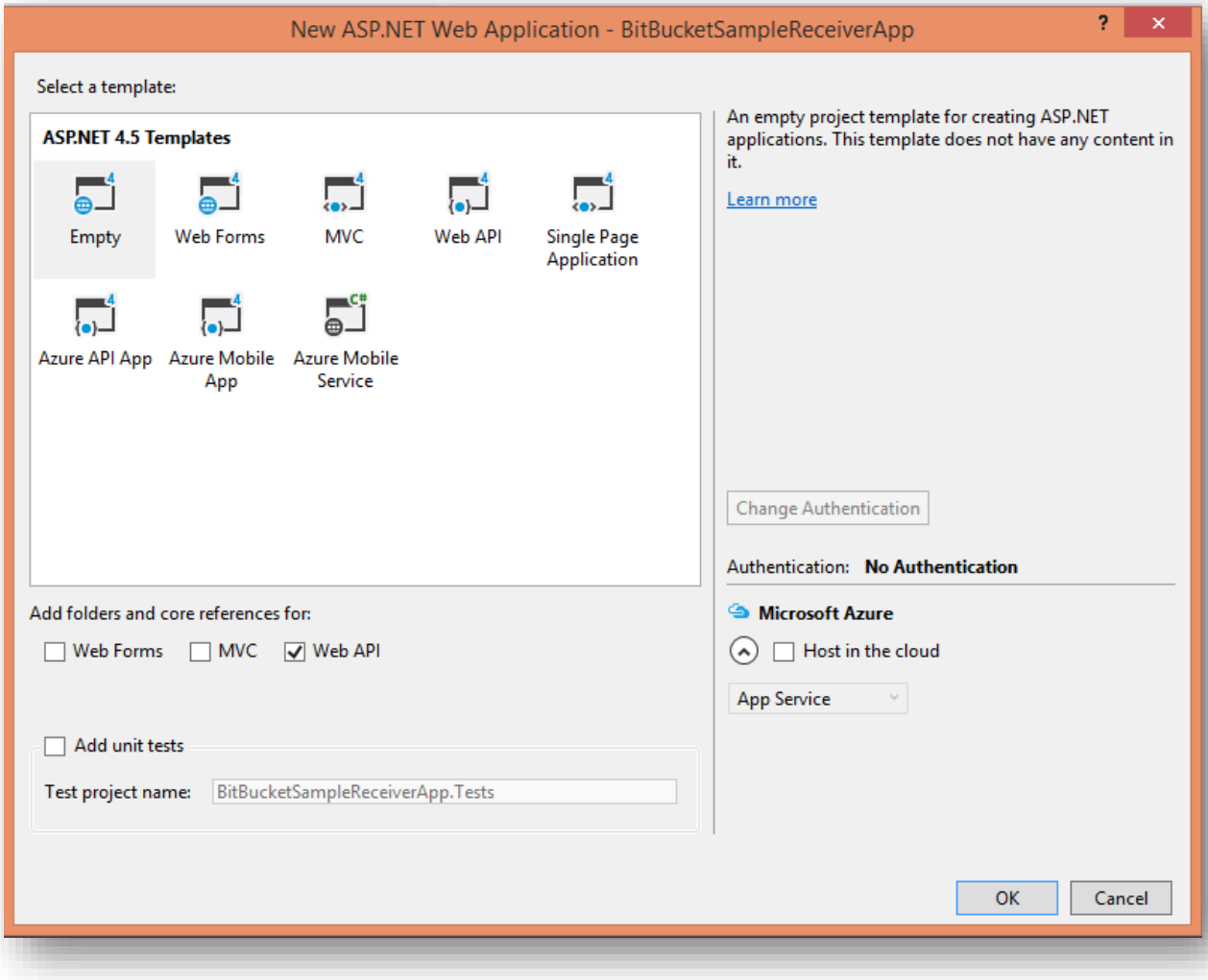

Figure 5: Select project template

3. Install the NuGet package using either the **Manage NuGet Package Manager** dialog box or the **Package Manager** console.

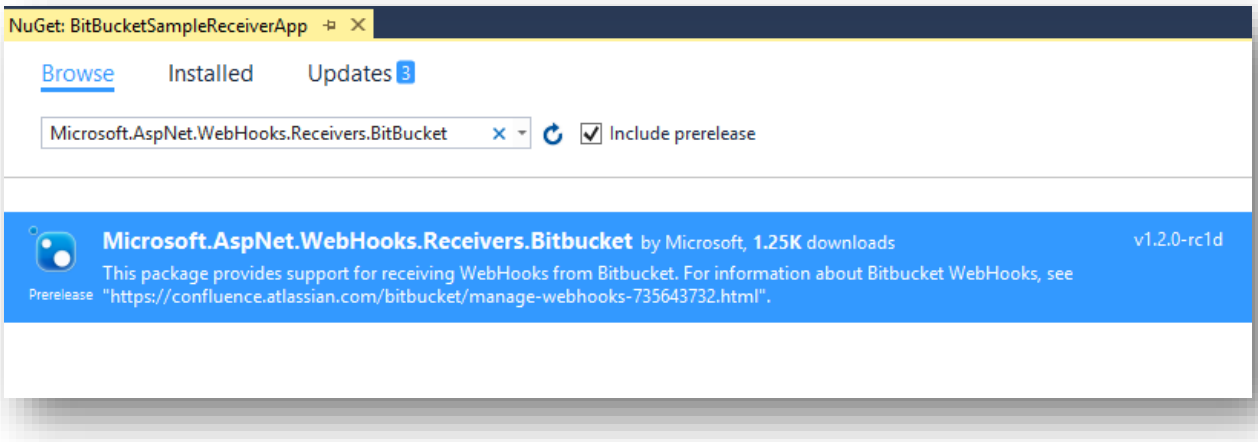

Figure 6: NuGet package manager dialog

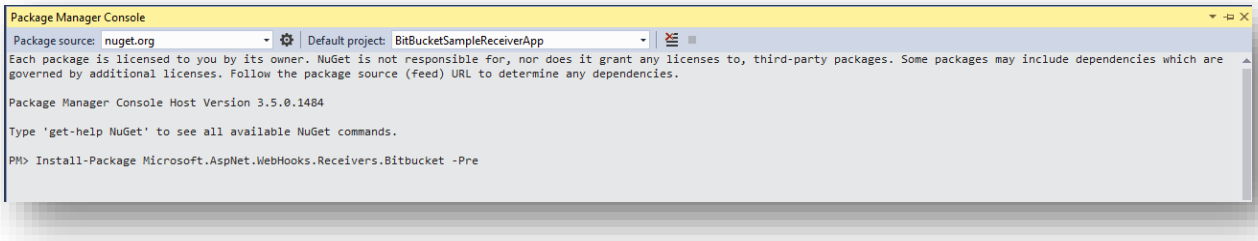

Figure 7: NuGet package manager console

- Search **`Microsoft.AspNet.WebHooks.Receivers.BitBucket`** and make sure you have checked the **Include prerelease** option.
- Type **`Install-Package Microsoft.AspNet.WebHooks.Receivers.BitBucket -Pre`** if you are using the Package Manager console.

Note: You will notice new assemblies added in the project:
`Microsoft.AspNet.WebHooks.Common`, `Microsoft.AspNet.WebHooks.Receivers`, `Microsoft.AspNet.WebHooks.Receivers.Bitbucket`

Unique config key

A unique config key is nothing but a unique key for every WebHook receiver. It contains a **`secret_key`**, which should be between 32 and 128 characters long. This key is nothing but a HMAC.

There are plenty of tools available to generate the secret key. In all examples in this book, we have used the tool available here to generate HMAC (or in other words, our WebHook receiver's **`secret_key`**).

Table 2 lists the key of every receiver.

Table 2: Unique config key

Unique Config key	
Receiver	**Key**
Bitbucket	`MS_WebHookReceiverSecret_Bitbucket`
Dropbox	`MS_WebHookReceiverSecret_Dropbox`
GitHub	`MS_WebHookReceiverSecret_GitHub`
Instagram	It required two keys: `MS_WebHookReceiverSecret_InstagramId` – Client ID `MS_WebHookReceiverSecret_Instagram` – Client Secrete
MailChimp	`MS_WebHookReceiverSecret_MailChimp`
MyGet	`MS_WebHookReceiverSecret_MyGet`
Slack	`MS_WebHookReceiverSecret_Slack`
Stripe	`MS_WebHookReceiverSecret_Stripe`
VSTS	`MS_WebHookReceiverSecret_VSTS`
Zendesk	`MS_WebHookReceiverSecret_Zendesk`

Set up unique config key

To set up a unique config key with a **`secret_key`** value, we need to add a config key with a unique name. In our case, we are going to add the unique key **`MS-WebHookReceiverSecret_Bitbucket`**

To set up our application for successfully receiving the intended notification, we must add a unique config key containing the **`secret_key`** value. For this, just open up the **`Web.Config`** field of our newly created application. Add a key here; as already discussed, every WebHook receiver has a unique key. Code Listing 2 shows how to set this up for Bitbucket. If you are planning to create the new application based upon the WebHook receiver other than the Bitbucket receiver, please refer to Table 3, listing all the unique keys for the respective WebHook receivers.

Code Listing 2

```
<appSettings>
```

```
    <add key="MS_WebHookReceiverSecret_Bitbucket"
value="a5bfbf7b2d2c4ec07757183a61e9570611112ddc2724bde63ba85c0664c75403"/>
  </appSettings>
```

Initializing or configuring a WebHook receiver

So far, we have created a small application to consume a Bitbucket WebHook receiver, and configured it with a unique key to get event notifications from a particular WebHook.

The next step is to initialize a WebHook receiver. To do that, open the WebAPIConfig file from your project and modify it by adding: **`config.InitializeReceiveBitbucketWebHooks`**(), as shown in Code Listing 3.

WebHook receivers are configurable using the **`IWebHookReceiveConfig`** interface, and can be implemented by using any dependency injection.

Code Listing 3

```
 namespace BitBucketSampleReceiverApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

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

```
            // Initialize Bitbucket WebHook receiver
            config.InitializeReceiveBitbucketWebHooks();
        }
    }
}
```

Note: InitializeReceiveBitbucketWebHooks() is an extension method that is defined in Microsoft.AspNet.WebHooks.Receivers.Bitbucket.dll. For more information, refer here.

Creating a WebHook handler

A WebHook handler is just a simple class with a special operation.

The next step is to create a handler for receiving the event notifications from Bitbucket. In order to achieve this, we need to do the following:

- Add new folder and name it WebHooks. You can put any kind of WebHook handlers under this folder.
- Add new class under folder **`WebHooks`** and name it: BitBucketWebHookhandler.cs. It should be inherited from **`WebHookHandler`**.
- In this sample, we are going to track push notifications. Let's write our code to receive the push notifications (although we can write code to receive any other kind of event notification as well).

Tip: We can write our application to receive any kind of event notification from Bitbucket, provided we have added the appropriate settings for those event notifications.

Note: Documentation of Bitbucket push notifications can be found here.

Code Listing 4

```
    public class BitBucketWebHookhandler : WebHookHandler
    {
        public BitBucketWebHookhandler()
        {
            Receiver = BitbucketWebHookReceiver.ReceiverName;
        }
```

```
        public override Task ExecuteAsync(string receiver,
WebHookHandlerContext context)
        {
            var dataJObject = context.GetDataOrDefault<JObject>();
            var action = context.Actions.First();
            switch (action)
            {
                case BitBucketAction.Push:
                    var repository =
dataJObject["repository"].ToObject<BitbucketRepository>();
                    var actor =
dataJObject["actor"].ToObject<BitbucketUser>();
                    AssessChanges(dataJObject);
                    break;

                default:
                    var data = dataJObject.ToString();
                    break;
            }
            return Task.FromResult(true);
        }

        private static void AssessChanges(JObject dataJObject)
        {
            foreach (var change in dataJObject["push"]["changes"])
            {
                var previousValue =
change["old"]["target"].ToObject<BitbucketTarget>();
```

```
                var newValue =
change["new"]["target"].ToObject<BitbucketTarget>();
            }
        }
    }
```

In Code Listing 4, we are have our handler, which specifically handles Bitbucket events. If you see a receiver in the constructor, this receiver will return **`BitBucket`** only.

Our **`BitBucketWebHookHandler`** class is inheriting **`WebHookHandler`** (an abstract class). Here, we are overriding the **`ExecuteAsync`** method, which is actually having the receiver and **`WebHookhandlerContext`** (context of incoming) method.

In the preceding code listing, we are getting data as per the documentation of Bitbucket mentioned here.

When we have a single generic handler for multiple WebHook receivers, we check for a particular receiver like: **`if("BitBucket".Equals(receiver, StringComparison.CurrentCultureIgnoreCase)).`** Our modified code would be look like the following:

Code Listing 5

```
namespace BitBucketSampleReceiverApp.WebHooks
{
    public class BitBucketWebHookhandler : WebHookHandler
    {
        public BitBucketWebHookhandler()
        {
            Receiver = BitbucketWebHookReceiver.ReceiverName;
        }

        public override Task ExecuteAsync(string receiver,
WebHookHandlerContext context)
        {
            if ("BitBucket".Equals(receiver,
System.StringComparison.CurrentCultureIgnoreCase))
```

```
            {
                ReceivefromBitbucket(context);
            }

            return Task.FromResult(true);
        }
```

In the preceding code listing, we have extracted our main method to a small method **`ReceivefromBitbucket()`**; similarly, we can add more conditions for different WebHook receivers while working with a generic receiver.

Publishing the application

We have created a receiver application with a Bitbucket handler. Our small application is now capable of communicating with Bitbucket, or in other words, it can now listen or receive event notifications from the Bitbucket repository for which we have configured the WebHook.

Importance of SSL support with public URI

Due to security reasons, almost all WebHook providers require that the URI is public and with SSL support so it can use the '**`https`**' protocol. In other words, if our URI is not public, then how can a Bitbucket repository connect to your local URI?

Getting the public URI

As we discussed in the preceding section, our application should have a public URI with **`https`** protocol—in other words, it should be SSL-enabled. There are lot of ways to make your application public with SSL:

- Purchase a domain
- Purchase a hosting
- Purchase a SSL certificate
- Deploy your application
- Bind your domain with SSL

These steps require money and plenty of time. The easiest way to achieve our deployment is by using Azure. Azure URLs have SSL enabled internally, so we can use those for our application. We are not going to discuss Azure deployment in detail here.

Note: Refer to [Windows Azure Websites Succinctly] for more details.

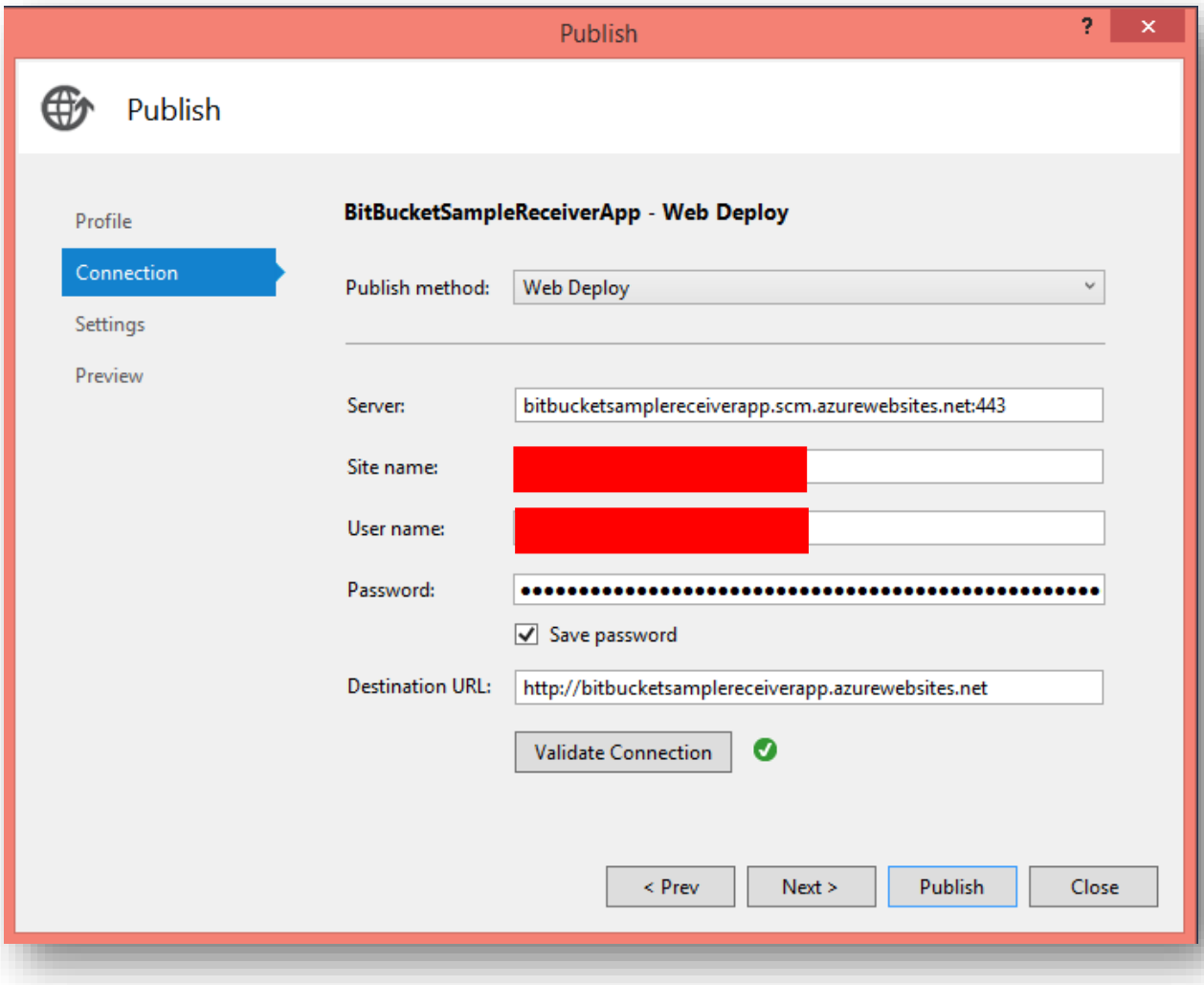

Figure 8: Publish dialog

Configure the Bitbucket repository

We've now gone through the entire cycle of consuming WebHooks with the help of NuGet packages. Let's now look at how to configure the Bitbucket repository, in case you wanted to use the second option of going through the samples provided. As these samples use the source code and not the NuGet packages, you must perform the following steps. If you are not interested in performing the exercise outlined here, you can skip to the next chapter.

Since we have already created a **BitbucketWebHook** receiver application that can talk to the Bitbucket repository and listen for the event notifications, let's move to our next step.

The next step would be to configure our Bitbucket repository, which will send event notifications to our **`BitBucketWebHook`** receiver.

1. Go to your Bitbucket (https://bitbucket.org) account, login with valid credentials or create one, in case you do not have account with Bitbucket.
2. Go to your repository (we are using this one).
3. Click on Setting Under Navigation**.**
4. Click **WebHooks** in the **Integrations** section.
5. On the **WebHook** page, click Add webhook (for more information, go here).
6. Enter a meaningful title (for example, BitbucketWebHook)
7. Enter the correct URI (in the format of https://<host>/api/webhooks/incoming/bitbucket/?code=secret_key) of the public application we have just deployed with the URL: http://bitbucketsamplereceiverapp.azurewebsites.net/. So, our actual WebHook URI would be: https://bitbucketsamplereceiverapp.azurewebsites.net/api/webhooks/incoming/bitbucket/?code=secret_key
   Note: Please refer to the next section for a better understanding of this part.
8. Our **`secret_key`** would be a generated HMAC—refer to preceding section on how to generate the **`secret_key`**.
9. Customize your choices. You can bypass SSL/TLS certificate verification.
10. Triggers:
    a. Choose **Repository push** or
    b. Select the triggers of your choice from the complete trigger list.
11. Once done, click **Save**.

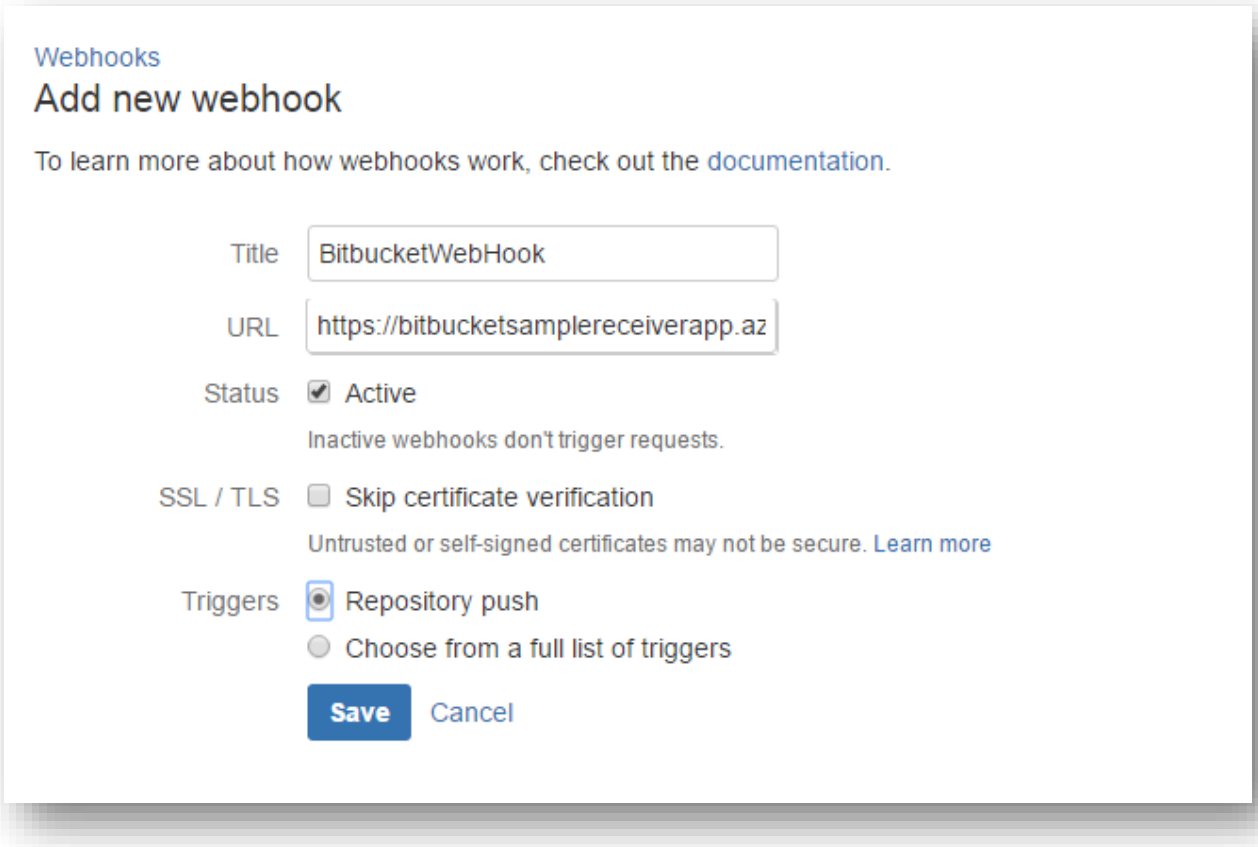

Figure 9: Add new webhook

Tip: The WebHook URI format is:
https://<host>/api/webhooks/incoming/bitbucket/?code=secret_key

Note: *`secret_key`* ***should contain the same value as the*** *`MS_WebHookReceiverSecret_bitbucket`* ***key in configurations,***

Assigning a secret_key value

We are all set to go with just one exception, we still need to set **`secret_key`** in our **`webconfig.cs.`** We have already added the key **`MS_WebHookReceiverSecret_Bitbucket`**, but without any value, so our next step is to assign a value to our config key.

There is no need to add value in our application (you can also add value to your **`webconfig`** at the time of creation of the application), as we have already deployed the application. If we are going to edit the **`webconfig`** file, then we need to re-deploy our application, which really makes no sense. Here, we can use the power of Azure to add our config key. To do so, go over to your Azure portal and access the Application settings, and add the key and value, as shown in following image.

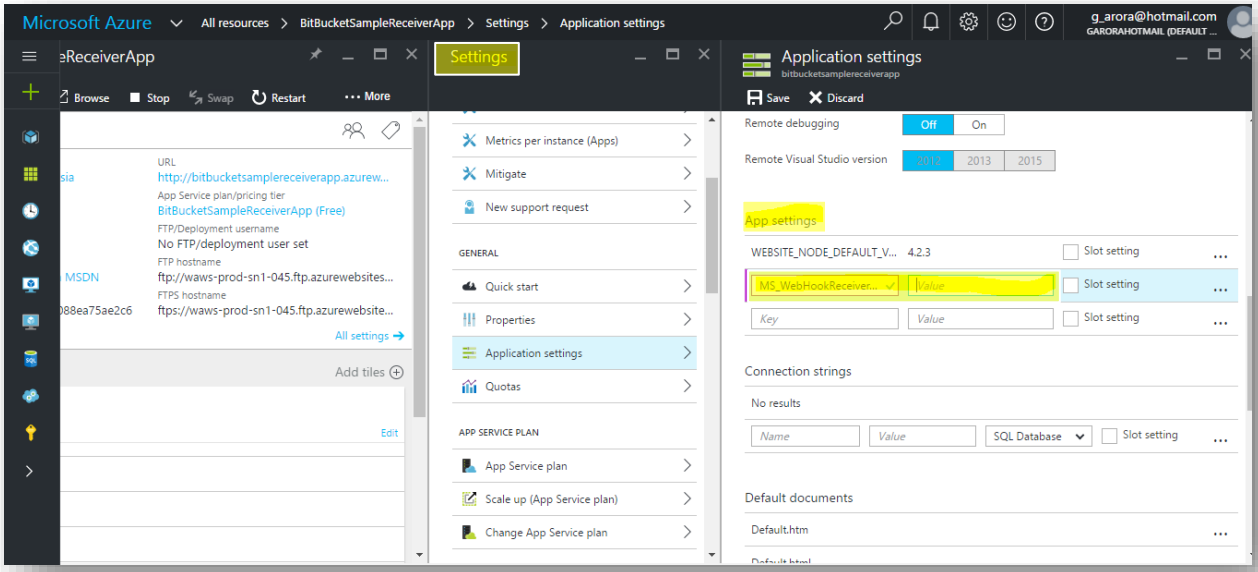

Figure 10: Configuring app settings

A quick note about the WebHooks process

As soon as the WebHook receiver validate the WebHooks request, it tells the application or user code to process. In other words, first WebHook receiver validates the WebHook requests and WebHook handlers start processing.

Verifying WebHook receiver application

Finally, we are ready to test our WebHook receiver. Go and make few changes to any of your files locally from your Bitbucket repository, for which you have configured WebHook, and then push it over to remote repository. We can verify the event notification in two ways:

- Verification from the Bitbucket View request page
- Verify the incoming data on application (can debug remotely)

Verification from Bitbucket

To make it easier let us edit event notifications, go back to the WebHook settings of your Bitbucket repository, and set them to receive events/notifications for Push and Issues.

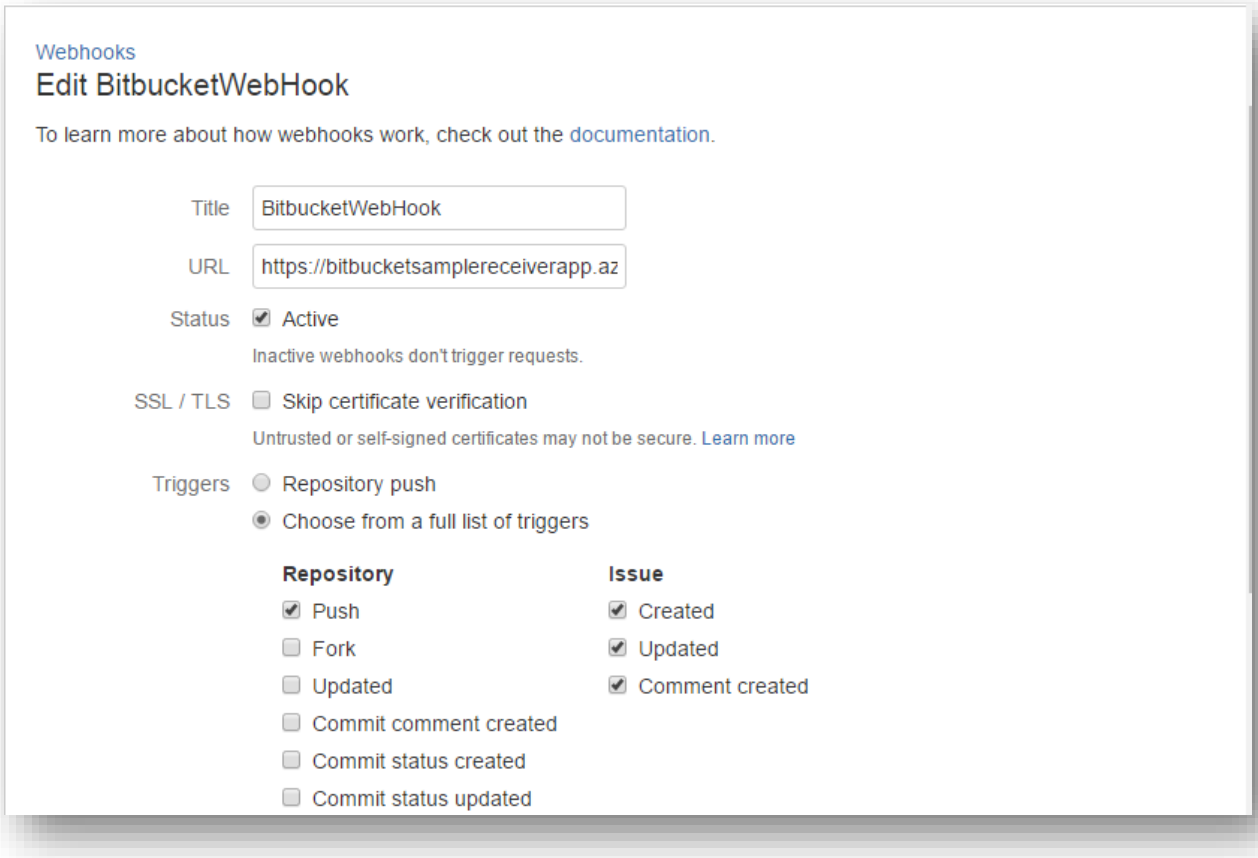

Figure 11: Edit existing WebHook

Now, go back to the repository and create a new issue or edit few changes in an existing issue. This will trigger a particular event notification.

You can verify this by visiting Settings > WebHook > View from within your Bitbucket repository.

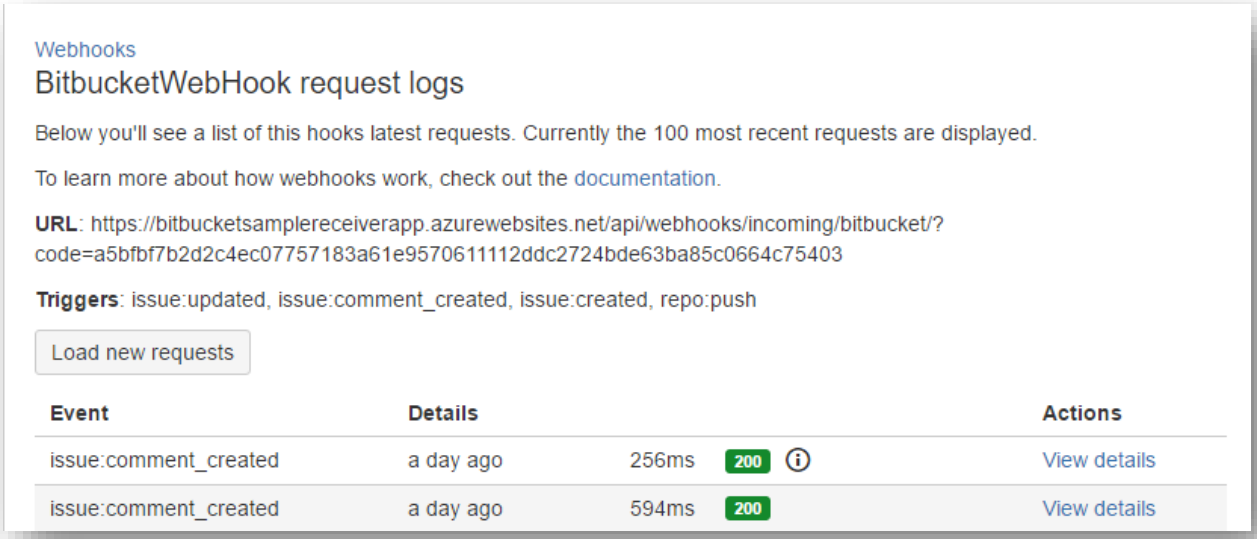

Figure 12: WebHook request logs

I have created one issue and then commented. In Figure 12, you can see what event triggered and what the details are. For more details, you can click on the View details link.

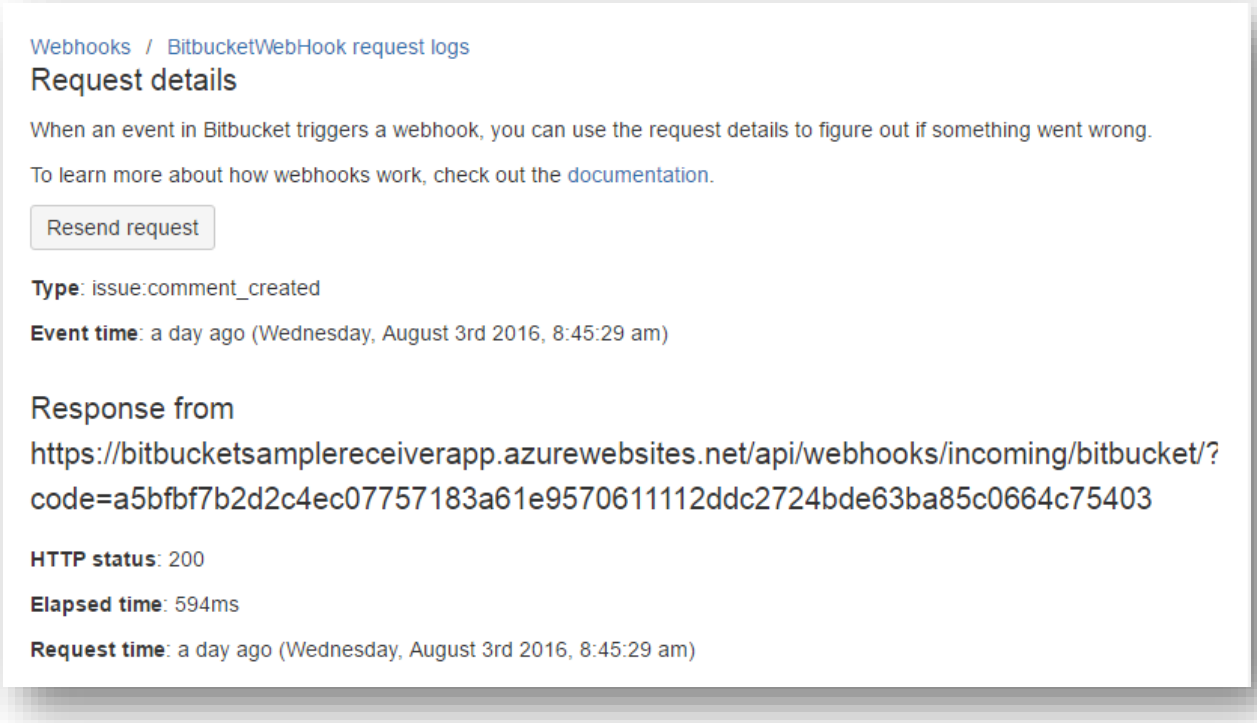

Figure 13: WebHook request details

On the Details page, you can get all details of the request (HTTP status, elapsed time, request time, etc.) You can also get the request and body details.

Debugging your application

We have verified that our WebHook is working fine by using the Bitbucket settings page, where we saw which event triggered and what we received. When such an application is being developed, the primary focus is on the end result rather than the configured settings. So if there is an analysis required to troubleshoot a probable bug, one would like to immediately debug the application.

So, let's check whether or not our application is receiving the data as expected. Attach the debugger to your application and make a few changes in your repository.

> ***Note: We are not going to discuss how to debug an Azure application—this is beyond the scope of this book. Refer to this article to learn more about debugging an Azure application.***

To verify our app, I have just modified the ReadMe file, and then pushed the changes back to the remote repository. Here, the Push notification gets triggered and notifies our WebHook receiver, so our application should hit the Debug point shown in Figure 14. We should get the triggered data.

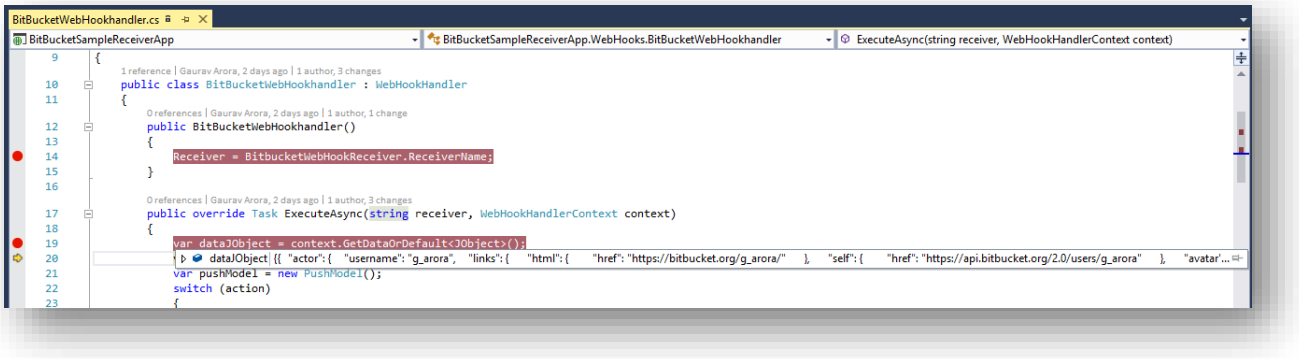

Figure 14: Debugging WebHook receiver application

Here, we received everything as required—our WebHook is working fine. In this application, we verified and understood how our WebHook receiver talks with the Bitbucket repository and how we get the requested data back to our subscribing application.

In this application we did not use the data, but in the actual environment where we would be using hooks, the data would be put to a more prudent use. In the next section, we will create a simple application to consume and process the data received.

Setting up a WebHook source repository

If you have decided to use the samples provided, then you already know that you need to set up your WebHook source repository as it is used these samples. This section will guide you through just that.

To create or customize a WebHook, one should have the complete source code. In this section, we will discuss how to set up your WebHook source repository and get ready for creating or customizing WebHooks.

Setting up a source code repository

ASP.NET WebHook is open source, and its source code is available from the GitHub repository here. If you are planning to add or customize the WebHook, you should follow these steps to set up the repository:

1. Access https://github.com and log in to your account.
2. Go to the ASP.NET WebHook repository URL.
3. Fork the repository and follow the instructions.
4. Use your favorite GUI or CLI to start, locally. I prefer to use GitBash or GitExtensions for Windows.
5. Clone the repository to your local system on your preferred location.
6. Check out the **`dev`** branch for any additions or modifications.
7. After any modifications, make a **`pull`** request (**`PR`**) to the main repository from which you forked your repository.

To create a new **`pull`** request, you just need to have a valid GitHub account, and most importantly, you have to sign the **cla for open source contribution**. For more information about cla, please refer here.

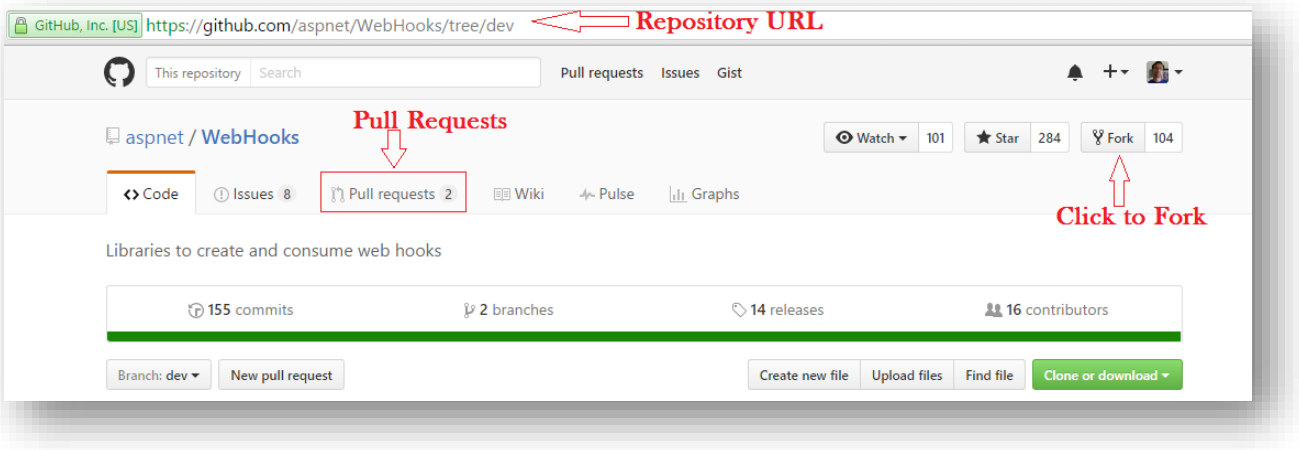

Figure 15: ASP.NET WebHook repository

Note: Please refer to GitHub Succinctly to learn how to work with GitHub repositories.

There are a lot of WebHook senders and receivers available; you can get started with one of these WebHooks using repository samples or creating your own custom application using NuGet packages.

WebHook receiver samples

WebHook receiver samples are available in the GitHub repository. To use samples, you just need to go this site and select your sample. Currently, available samples (for receivers) are:

- Azure
- Bitbucket
- DropBox
- GitHub
- Instagram
- MyGet
- Slack
- Stripe
- VSTS
- Zendesk

Note: All samples are referring to actual WebHook receiver projects, and are part of the ASP.NET WebHook solution. These are important when you are going to debug actual source code.

Conclusion

In this chapter we have discussed all ASP.NET WebHooks. We have looked at WebHook receivers, WebHook senders, and URI formats.

Chapter 3 Creating a Real-time Application

In this chapter, we will create an application that will talk with Bitbucket and receive event notification data. We will then work with the received data.

Introduction

We will create an application that will listen to our Bitbucket repository to get notifications. Our application will then:

- Receive notification using the **`handler`** class
- Store received data
- Display data using Web API

Prerequisite

Here are the prerequisites for continuing with the sample application:

- Basic knowledge of ASP.NET WebHooks (including WebHook receivers, handlers, etc.)
- Visual Studio 2015
- Basic knowledge of MVC5, WebAPI, and JQuery
- SQL Server 2008 R2 or later
- A valid Azure subscription
- Basic knowledge of AngularJS
- Basic knowledge of Bootstrap
- Basic knowledge of Entity Framework

Application: architecture and design

In this section we will discuss the architecture and design of the application we want to build. This includes how our application will interact with the Bitbucket repository, and what technologies and frameworks we are going to use in this application.

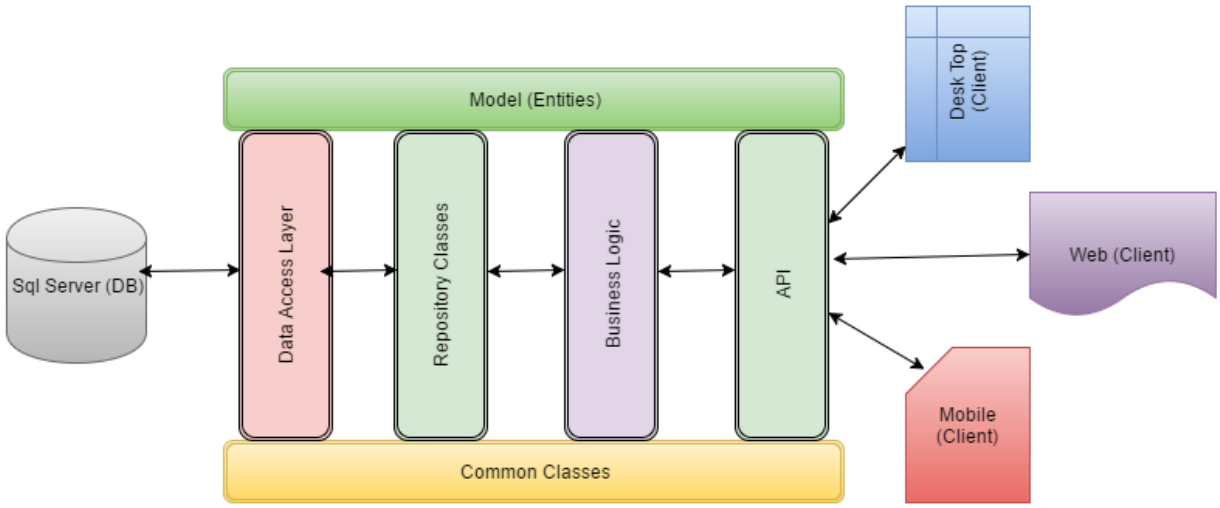

Figure 16: Overview of application

Figure 16 shows an overview of our application. Our main application will contain Restful APIs (Web API) and will be hosted on Azure as App Services. This API application will talk with the SQL Server Database hosted on Azure itself. These APIs can then be consumed by any client (desktop, web or mobile). In this sample, we are only going to develop a web client using AngularJS.

Note: RESTful is short form of "Representational state transfer," and RESTful APIs or RESTful Webservices provide a way to access services in a manner of stateless operations.

We can divide the preceding figure into three parts:

Database

This part of application is important to persist all the application activities. Here we will be using SQL Server database hosted on Azure. Furthermore, our database will be:

- Named: **`Activity`** (or name of your choice)
- Having table : **`ActivityModel`** (or name of your choice)

The database will be used to persist and retrieve all data received from Bitbucket repository.

API

This is the main part of our sample application. To make it simple and more flexible, we are using WebAPI2.0, and it is further divided into:

- Data access layer
- Business layer
- Repository classes
- Common classes
- Model entities

User Interface (UI)

This part represents a client application that will be consuming these APIs. Our client can be one of the following:

- Desktop application
- Web application
- Mobile application

To make our application simple, we will create a web application using MVC5 and AngularJS.

Creating the sample application

In Chapter 2, we created a sample application of BitbucketWebHook Receiver, where we discussed how our WebHook receiver receives the data from the Bitbucket repository. The scope for that application was just to showcase that the data was finally received.

In this section, we will discuss all the steps required to create a production application, and take it one step further by persisting the data received locally and making it available through a WebAPI.

New web application

To get started with this application, just create a simple WebAPI project using Visual Studio. You can also refer to Chapter 2 for help creating a WebAPI project.

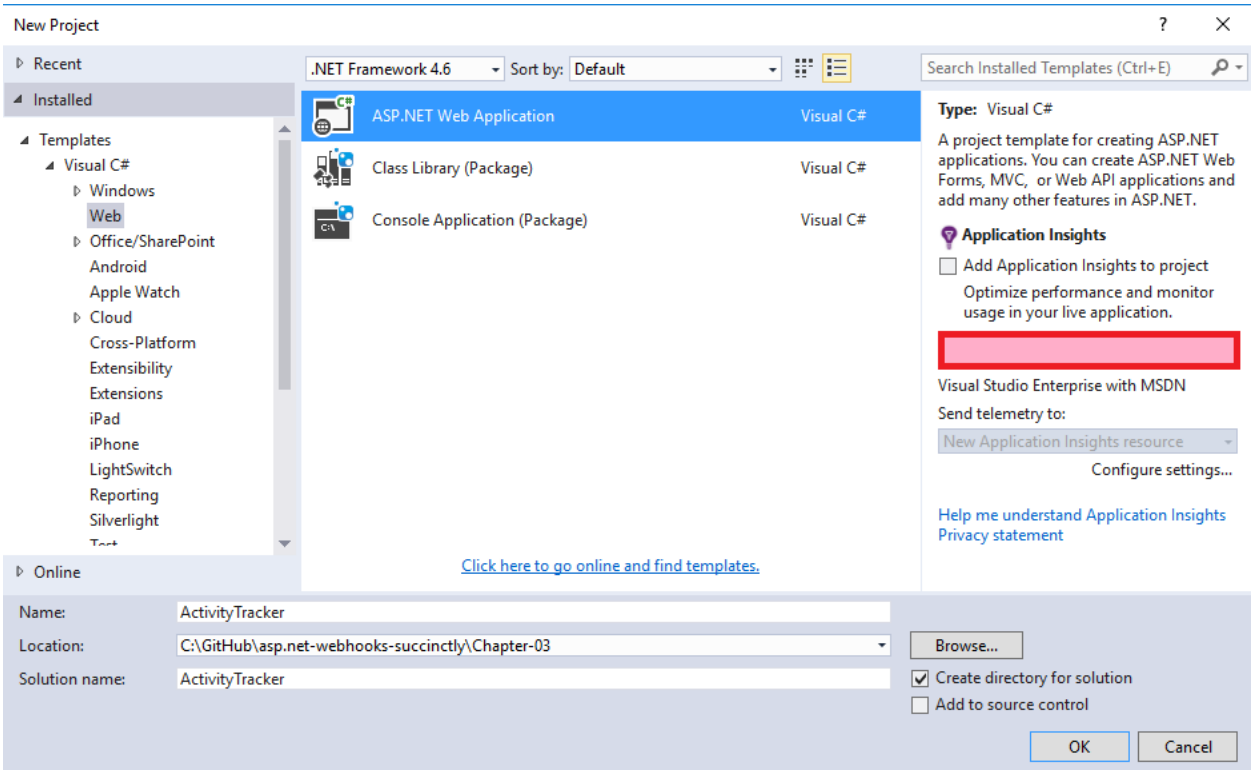

Figure 17: Creating web project

We will name our new project ActivityTracker, select a location, and provide a suitable Solution name. Click **OK** once you're done.

Select template

From the next screen, select an empty template and check the Web API checkbox, as shown in Figure 18.

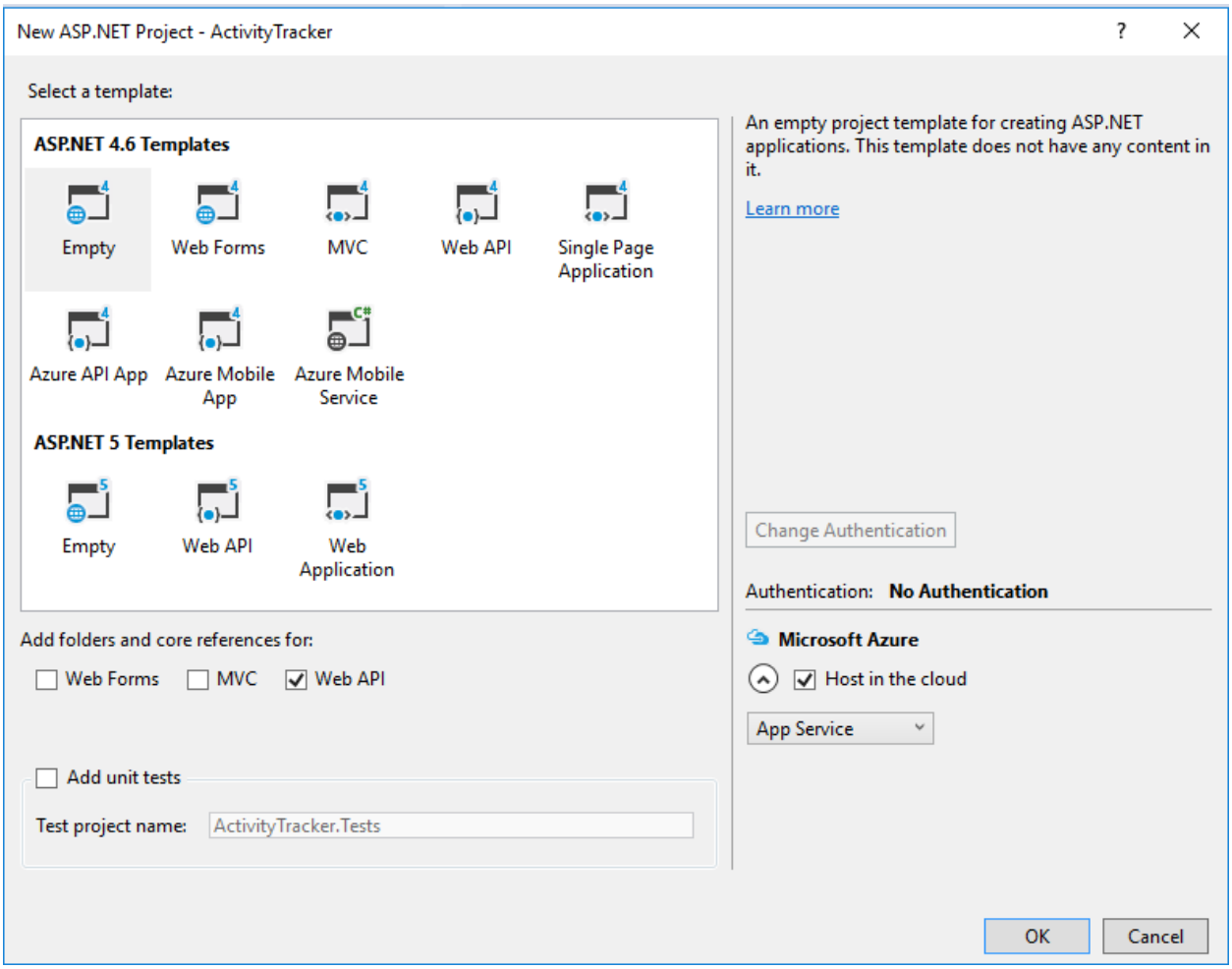

Figure 18: Selecting a template

As we will be deploying this app to Azure, it's good if you check the option **Host in the cloud** in the **Microsoft Azure** section. If you select this option, you will be be prompted to select an Azure app name and location. Please follow all the steps. You can just click on Cancel if you don't want to do this at the moment (we can publish it later).

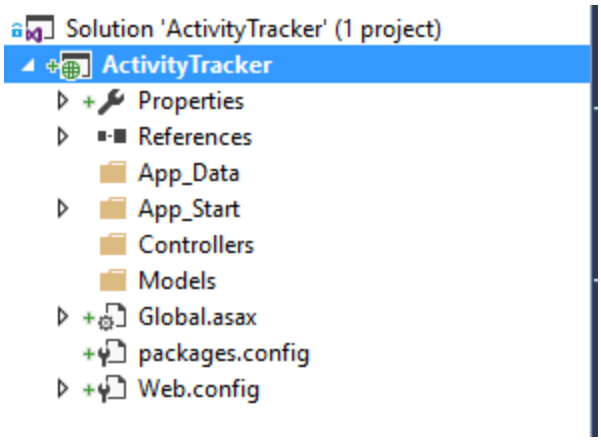

Figure 19: Project structure

Visual Studio will create an empty project, as shown in Figure 19.

Adding support of Bitbucket WebHook receiver

We have created an empty project, and now we need to add support for our Bibucket WebHook receiver to start collecting the data sent by Bitbucket.

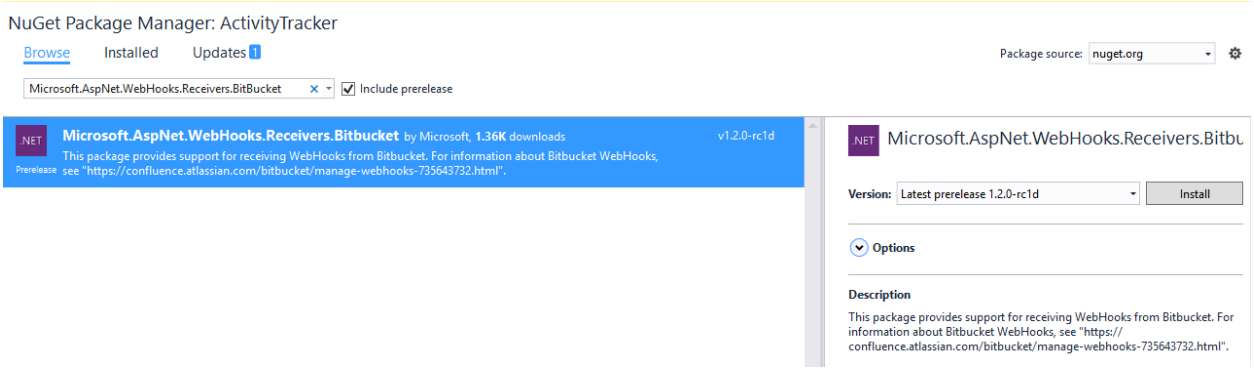

Figure 20: Adding NuGet packages

Open NuGet Package Manager, and under the **Browse** tab, enter Microsoft.AspNet.WebHooks.Receivers.BitBucket. Check the **Include prerelease** option, and then install the NuGet package.

Adding Entity Framework support

We are using Entity Framework to support our persistence models. Now, in the NuGet Package Manager dialog box, enter EntityFramework. Do not forget to uncheck the **Include prerelease** option.

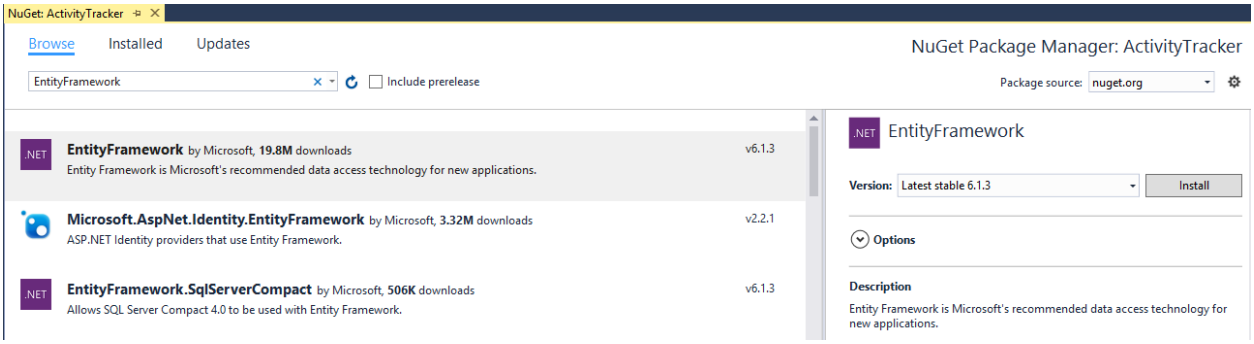

Figure 21 Adding EntityFramework

Click on **Install**, and it will install EF6.1.3 to the project.

Now that we have met all basic framework requirements for our project, we are ready to write our code.

Note: We created an application and named it** ActivityTracker**. This application will show us our all activities (what we have configured for WebHook).

Start writing Handler class

Let us start to write code to our sample application.

From Visual Studio, open the **Solution Explorer** and the new folder WebHooks. Add a **`BitBucketWebhookHandler`** class to the application under the newly created folder. This class should inherit an abstract class, **`WebHookHandler`**.

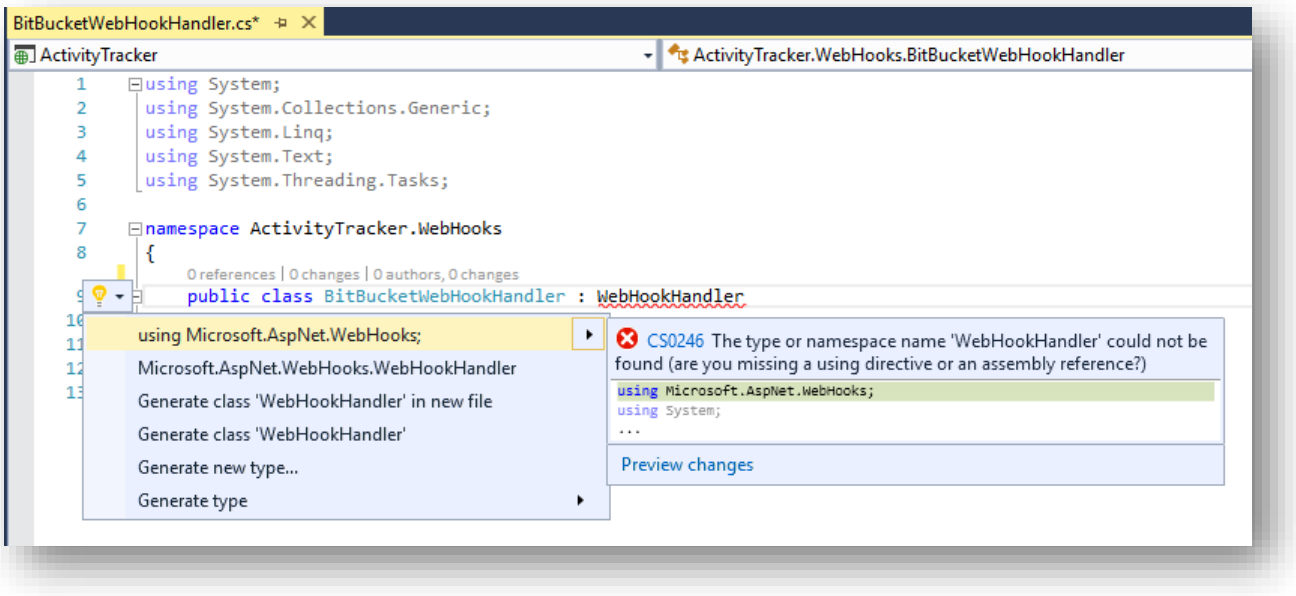

Figure 22: Missing namespace

Add required the namespace (**`Microsoft.AspNet.Webhooks`**) as shown in Figure 22, and implement its method (**`ExecuteAsync`**).

Note: We have already discussed the `ExecuteAsync` method in Chapter 2.

Implement the **`ExecuteAsync`** method to receive data from Bitbucket repository; the parameter **`WebHookhandlerContext`** contains all actions occurring during an operation. **`Action`** is a list of **`string`**. Always get **`First`** action, and then grab the data.

Code Listing 6

```
public class BitBucketWebHookHandler : WebHookHandler

    {

        public override Task ExecuteAsync(string receiver,
WebHookHandlerContext context)

        {

            if (Common.IsBitBucketReceiver(receiver))

            {

                var dataJObject = context.GetDataOrDefault<JObject>();

                var action = context.Actions.First();

                switch (action)

                {

                    case "repo:push":

                        //do something

                        break;

                    case "repo:fork":

                        //do something

                        break;

                    case "repo:updated":

                        //do something

                        break;


                    default:
```

```
                        var data = dataJObject.ToString();
                        break;
                }

            }

            return Task.FromResult(true);
        }
    }
```

In Code Listing 6, we are evaluating **`action`** and implementing code based upon its value. Bitbucket has these pre-defined actions, such as **`repo:fork`** and **`repo:updated`**. Based upon the value of **`action`** received by us, we will perform some specific operations. If it does not match any of these conditions, then we will simply get the data and store it in a **`var`** data type.

Sometimes it can be erroneous when comparing hard-coded strings. As humans, we can somehow add spaces or truncate some words (for example, **`repo:fork`** could be **`repo:fork`** or **`rep : fork`**).

To avoid scenarios where we might create errors, we can go with:

- constants
- enums

If we go with constants, then we can have a **`constant`** class, which can look something like this:

Code Listing 7

```
public class BitBucketRepoAction
    {
        //Refer to: https://confluence.atlassian.com/bitbucket/event-
payloads-740262817.html#EventPayloads-RepositoryEvents
        public const string Push = "repo:push";
        public const string Fork = "repo:fork";
        public const string Updated = "repo:updated";
        public const string CommitCommentCreated =
"repo:commit_comment_created";
```

```
        public const string CommitStatusCreated =
"repo:commit_status_created";

        public const string CommitStatusUpdated =
"repo:commit_status_updated";

    }
```

With the use of constants, the **`ExecuteAsync`** method of our **`BitBucketWebHookHandler`** class would look like the following:

Code Listing 8

```
public override Task ExecuteAsync(string receiver, WebHookHandlerContext
context)

{

            if (Common.IsBitBucketReceiver(receiver))

            {

                var dataJObject = context.GetDataOrDefault<JObject>();

                var action = context.Actions.First();

                switch (action)

                {

                    case BitBucketRepoAction.Push:

                        //do something

                        break;

                    case BitBucketRepoAction.Fork:

                        //do something

                        break;

                    case BitBucketRepoAction.Updated:

                        //do something

                        break;

                    case BitBucketRepoAction.CommitCommentCreated:

                        //do something

                        break;
```

```
                    case BitBucketRepoAction.CommitStatusCreated:
                        //do something
                        break;
                    case BitBucketRepoAction.CommitStatusUpdated:
                        //do something
                        break;

                    default:
                        var data = dataJObject.ToString();
                        break;
                }
            }
        return Task.FromResult(true);
}
```

If we go with enum, then we can have an **enum** declaration, which goes something like this:

Code Listing 9

```
namespace ActivityTracker.Enums
{
    public enum EnumRepository
    {
        [EnumDisplayName("Changes pushed to remote")]
        [EnumDisplayCode("repo:push")]
        push,
        [EnumDisplayName("Fork")]
        [EnumDisplayCode("repo:fork")]
        fork,
        [EnumDisplayName("Updated")]
```

```
        [EnumDisplayCode("repo:updated")]
        updated,
        [EnumDisplayName("Commit comment created")]
        [EnumDisplayCode("repo:commit_comment_created")]
        commitcommentcreated,
        [EnumDisplayName("Commit status created")]
        [EnumDisplayCode("repo:commit_status_created")]
        commitstatuscreated,
        [EnumDisplayName("Commit status updated")]
        [EnumDisplayCode("repo:commit_status_updated")]
        commitstatusupdated

    }
}
```

In Code Listing 9, we have created **EnumRepository** and declared all possible enums that fall in **BitBucket Repository Events.**

Note: Details of Bitbucket repository events can be found here.

We have created two custom attributes, **EnumDisplayName** and **EnumDisplayCode**, which represent a short description of **enum** and an action of the Bitbucket repository event. We will use these **EnumDisplayCode** to analyze the **WebHookHandlerContext** action.

In Code Listings 10 and 11, we will see how these attributes look, and how we can get the values of these attributes.

Code Listing 10

```
namespace ActivityTracker.Attributes
{
    /// <summary>
    ///     A custom attribute used to associate a display name with an
enum
```

```
    ///     value
    /// </summary>
    [AttributeUsage(AttributeTargets.All)]
    public class EnumDisplayNameAttribute : Attribute
    {
        /// <summary>
        ///     initializes an instance of the <see
cref="EnumDisplayNameAttribute" />
        ///     custom attribute with the specified displayName
        /// </summary>
        /// <param name="displayName">
        ///     the displayName to associate with the
        ///     custom attribute
        /// </param>
        public EnumDisplayNameAttribute(string displayName)
        {
            DisplayName = displayName;
        }

        /// <summary>
        ///     gets the displayName property for the <see
cref="EnumDisplayNameAttribute" />
        ///     custom attribute
        /// </summary>
        public string DisplayName { get; }
    }
}
```

Code Listing 11

```
namespace ActivityTracker.Attributes
{
    /// <summary>
    ///  A custom attribute used to associate a display code with an enum
    ///     value
    /// </summary>
    [AttributeUsage(AttributeTargets.All)]
    public class EnumDisplayCodeAttribute : Attribute
    {
        /// <summary>
        /// initializes an instance of <see
cref="EnumDisplayCodeAttribute"/>
        /// custome attribute with the specified displaycode
        /// </summary>
        /// <param name="displayCode">
        /// the displayCode to associate with the
        /// custom attribute
        /// </param>
        public EnumDisplayCodeAttribute(string displayCode)
        {
            DisplayCode = displayCode;
        }


        /// <summary>
        /// Gets the displaycode property for the <see
cref="EnumDisplayCodeAttribute"/>
        /// custom attribute
        /// </summary>
```

```
        public string DisplayCode { get; }
    }
}
```

These custom attribute classes simply contain the properties **DisplayName** and **DisplayCode**. Both classes are inherited from **System.Attribute**, and their usage is set for all targets.

Note: For more details regarding attributes, refer here.

We have created custom attributes, but to access these attributes we need to write some custom code. Our best bet to access attribute values is by using **reflection**. To achieve this, I have written a helper class. In order to access the attribute values, we would have two methods: one to access **DisplayName**, and another to access **DisplayCode**.

We won't go into great detail for this class as a whole, but we will cover these two methods in detail.

Note: The complete source code is available here.

Code Listing 12

```
/// <summary>
///     Retrieves the display name (specified in the <see
cref="EnumDisplayNameAttribute" />
///     custom attribute) for the passed in <see cref="System.Enum" />
instance.
/// </summary>
/// <param name="enumeratedValue"></param>
/// <param name="enumProp"></param>
/// <returns>the display name for the specified enum</returns>
/// <remarks>
///     The enum specified must implement the <see
cref="EnumDisplayNameAttribute" />
///     custom attribute.  If it does not, an empty string is returned
/// </remarks>
```

```
public static string GetEnumDisplayName(System.Enum enumeratedValue,
EnumProp enumProp)
{
  var fieldInfo =
enumeratedValue.GetType().GetField(enumeratedValue.ToString());
  var attribArray = GetCustomAttributes(fieldInfo, enumProp);
  return attribArray.Length == 0
                ? string.Empty
                : GetEnumDisplay(enumProp, attribArray);
}
```

This method is accepting **enumeratedValue** and **enumProp** and finding the custom attribute. It then calls another method **GetEnumDisplay(enumProp, attribArray)** to display the value.

Code Listing 13

```
public enum EnumProp
    {
        /// <summary>
        ///     display name custom property
        /// </summary>
        DisplayName,

        /// <summary>
        ///     display code custom property
        /// </summary>
        DisplayCode
    }
```

EnumProp is nothing but another enum that contains two enums: **DisplayName** and **DisplayCode**.

Code Listing 14

```
private static object[] GetCustomAttributes(FieldInfo fieldInfo, EnumProp
enumProp)
{
   return enumProp == EnumProp.DisplayName
         ?
fieldInfo.GetCustomAttributes(typeof(EnumDisplayNameAttribute), false)
         :
fieldInfo.GetCustomAttributes(typeof(EnumDisplayCodeAttribute), false);
}
```

`GetCustomAttributes()` first locates the value of **`DisplayName`**, and if it is not found, it then gets the custom attribute using the reflection method **`GetCustomAttributes()`**, and returns an object array.

Code Listing 15

```
private static string GetEnumDisplay(EnumProp enumProp, object[]
attribArray)
{
  switch (enumProp)
  {
    case EnumProp.DisplayName:
         {
           var attrib = attribArray[0] as EnumDisplayNameAttribute;
           return attrib != null ? attrib.DisplayName : string.Empty;
         }
    case EnumProp.DisplayCode:
         {
          var attrib = attribArray[0] as EnumDisplayCodeAttribute;
          return attrib != null ? attrib.DisplayCode : string.Empty;
         }
    default:
```

```
            return string.Empty;
    }
}
```

The **GetEnumDisplay()** method is getting a **DisplayName** or a **DisplayCode** as per **enumProp**, and it returns an empty string in case of no match. There might be cases where the provided custom attribute name is actually not available; in such conditions we will get an empty string, which is by design.

We have seen that it will make our code easier to read and understand if we go with either a constant or an enum.

In our application, I will go with enum. The reason I prefer to go with enum is that our customized attribute will give us more facility while we will work with either **action** or **dataobject** in our **BitBucketWebHookHandler**. With the help of the helper class and extension methods, we can easily access the custom attribute as per our need.

Our **BitBucketWebHookhandler** class would look like Code Listing 16, when we are using this for **Repository events**.

Note: There are different approaches to achieve this; we will discuss a few in coming sections.

Code Listing 16

```
public class BitBucketWebHookHandler : WebHookHandler
{
    public override Task ExecuteAsync(string receiver,
WebHookHandlerContext context)
    {
       if (Common.IsBitBucketReceiver(receiver))
       {
         var dataJObject = context.GetDataOrDefault<JObject>();
         var action = context.Actions.First();
         var enumAction = EnumHelper.GetEnumValue<EnumRepository>(action,
                                              EnumProp.DisplayCode,
false);
         switch (enumAction)
```

```
        {
           case EnumRepository.push:
                //do something
                break;
           case EnumRepository.fork:
                //do something
                break;
           case EnumRepository.updated:
               //do something
                break;
           case EnumRepository.commitcommentcreated:
               //do something
                break;
           case EnumRepository.commitstatuscreated:
               //do something
                break;
           case EnumRepository.commitstatusupdated:
               //do something
               break;

           default:
               var data = dataJObject.ToString();
               break;
         }

        }
        return Task.FromResult(true);
    }
```

```
}
```

Now, our **BitBucketWebHookHandler** class looks neat and clean, and we can easily go to an enum and read its description. In every case, we can also use our **EnumHelper** class for various operations.

Before going any further, let us draft out what we want to do with the data. Our data is nothing but some information that our WebHook received via **WebHookHandlerContext**.

Entity framework: Using code-first approach

We have already added Entity Framework support to our project. Now the only thing remaining is to add few models, mapping, and persistence classes.

Under the folder **Models**, add a new class with the name **ActivityModel**. The model class looks like Code Listing 17:

Code Listing 17

```
public class ActivityModel
    {
        [Key]
        public int ActivityId { get; set; }
        [MaxLength(10)]
        public string Activity { get; set; }
        [MaxLength(25)]
        public string Action { get; set; }
        [MaxLength(65)]
        public string Description { get; set; }
        public string Data { get; set; }
    }
```

Note: The validation attributes may cause validation errors in API method calls if the data passed in is longer than specified by the attributes, but the attributes are used to create the data model.

Setting up the repository

Before writing code for our repository, which will enforce the code-first approach, let's add a connection string in the webconfig file. From the Solution Explorer, open the **web.config** file and add the connection string, as shown in Code Listing 18.

Code Listing 18

```
  <connectionStrings>
    <add name="ActivityDBConnectionString" connectionString="data
source=.\sqlexpress;initial catalog=Activity;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient"/>
  </connectionStrings>
```

Note: We used SQL Express as a database, so we used a connection string of SQLServer. If you are using MySQL or any other database, you would be required to change connection string accordingly.

Now we are ready to write our repository classes. Open Solution Explorer and add a new folder, **`Persistance`**. Add a new class called **`ActivityContext`**, which should inherit from **`DbContext`**. Our context class would look like the following:

Code Listing 19

```
public class ActivityContext : DbContext
    {
        public ActivityContext() :
base("name=ActivityDBConnectionString")
        {


        }
        public DbSet<ActivityModel> ActivityTrackers { get; set; }
    }
```

In our context, we passed **`connectionstring`** to let **`Dbcontext`** know about our database. We have also added one **`DbSet`** of our **`ActivityModel`** class.

To complete our repository classes, we also added **`IActivityRepository`** interface and a class **`ActivityRepository`** which implements the interface.

Rewriting the BitBucketWebHookHandler class

Our handler class is not persisting data. Let us rewrite our **BitBucketWebHookhandler** class so it will start persisting our data received from the Bitbucket repository. The new code would look like the following:

Code Listing 20

```
public class BitBucketWebHookHandler : WebHookHandler
    {
        public BitBucketWebHookHandler()
        {

        }

        public override Task ExecuteAsync(string receiver,
WebHookHandlerContext context)
        {
            if (Common.IsBitBucketReceiver(receiver))
            {
                var dataJObject = context.GetDataOrDefault<JObject>();
                var action = context.Actions.First();

                var processActivities = new
ProcessActivities(dataJObject, action);
                processActivities.Process();
            }

            return Task.FromResult(true);
        }
    }
```

In our new code, we have introduced a new class, **ProcessActivities**, and upon calling the **Process** method, our data will be persisted in our SQL Database.

Code Listing 21

```
public void Process()
  {
    var firstPart = _action.Split(':');
    if (firstPart[0].ToLower() == "repository")
          ProcessRepository();
    else if (firstPart[0].ToLower() == "pullrequest")
          ProcessPullRequest();
    else if (firstPart[0].ToLower() == "issue")
          ProcessIssue();
    else
          throw new System.Exception("UnIdentified action!");
  }
```

Write APIController

The next step is to add write to our WebAPI controller (**TrackerController**) code to complete the Web API.

Code Listing 22

```
public class TrackerController : ApiController
    {
        private readonly IActivityRepository _activityRepository;

        public TrackerController()
        {
            _activityRepository = new ActivityRepository();
        }

        public TrackerController(IActivityRepository activityRepository)
```

```
        {
            _activityRepository = activityRepository;
        }
        // GET api/<controller>
        public IEnumerable<ActivityModel> Get()
        {
            return _activityRepository.GetAll();
        }

        // GET api/<controller>/5
        public ActivityModel Get(int activityId)
        {
            return _activityRepository.GetBy(activityId);
        }

    }
```

We are ready with two **GET** resources. One retrieves complete records from database, while another fetches records of a particular **ActivityId**.

We are done with our Web API, the next step is to create a client. As we discussed previously, we are going to create a web client using MVC5 and AngularJS. To achieve this, we need to follow these steps:

Wire up API using AngularJS

Add the MVC controller

Open Solution Explorer and right-click on the **controllers** folder, and add a new MVC controller called **DashboardController**. This task will add MVC5 support to our project. Now we just need to add the following code snippet to our config file.

Code Listing 23

```
public class WebApiApplication : HttpApplication
```

```
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
            //Add these lines, after adding MVC5 Controller
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
```

Add AngularJS references

To start with angularjs, open the NuGet Package Manager and add the AngularJS NuGet packages:

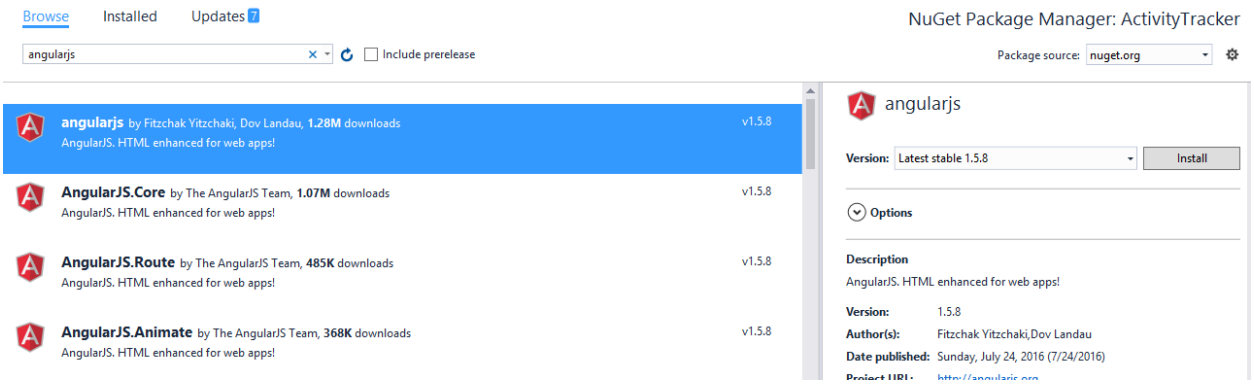

Figure 23: AngularJS

In Solution Explorer, open the new folder **app** in the **scripts** folder. Right-click the folder and add a new item from the **New Item** dialog box, and select the **angularjs** controller.

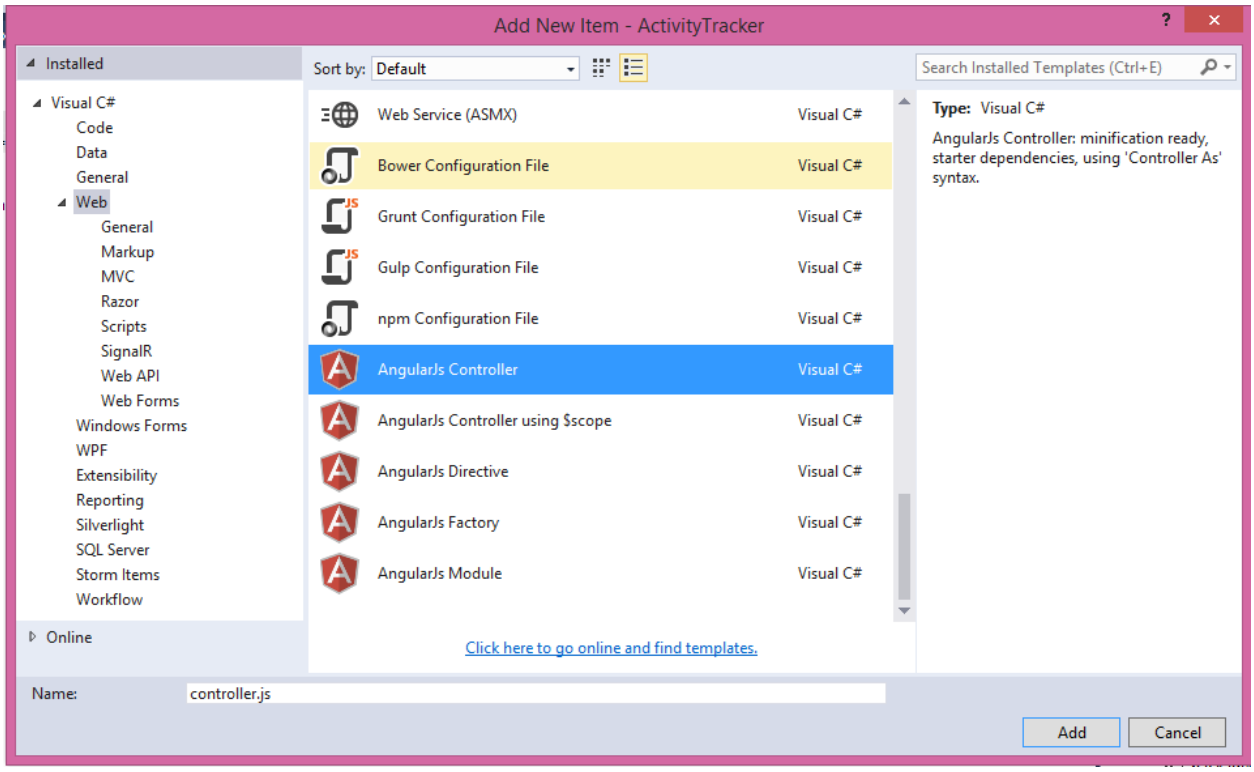

Figure 24: Adding AngularJS controller

Wire up the API call

Now add AngularJS services, and name it **service.js**. Then add this code to the js file:

Code Listing 24

```
app.service("activityService", function($http) {
    'use strict';
    //Get Activities
    this.getAll = function() {
        return $http.get("api/Tracker");
    };

    //Get single activity

    this.getActivity = function(activityId) {
        return $http.get("api/Tracker/" + activityId);
```

```
    };
});
```

In our service.js file, we are just wrapping up our API calls in such a way that our **getAll** function will use the Web API resource **api/Tracker**.

Now, get back to our AngularJS controller and add code so that we can use it in our View.

Code Listing 25

```
app.controller('activityController', function($scope, $log,
activityService) {


    refreshGrid();


    function refreshGrid() {
        activityService.getAll().then(function(promise) {
$scope.Activities = promise.data },
            function(err) {
                $log.error('error while connecting API', err);


            });
    };


    $scope.get = function(activityId) {
        activityService.get(activityId).then(function(promise) {
                $scope.Id = promise.ActivityId;
                $scope.Activity = promise.Activity;
                $scope.Action = promise.Action;
                $scope.Description = promise.Description;
                $scope.Data = promise.Data;
            },
```

```
            function(err) {
                $log.error('error while connecting API', err);
            });
    };


});
```

In **controller.js**, we are calling the **refreshGrid()** function, which will fetch complete data from our database as soon as our application is initiated.

a. Add View

Add Razor view by right-clicking on the Index action method of **DashboardController**, and select **Blank** view. It adds the **Index.cshtml** view under the folder **Dashboard** of the **Views** folder. Add the following code to our view to initiate the data:

Code Listing 26

```
<div ng-app="activityModule">
    <div class="container" ng-controller="activityController">
        <h2>Activity Dashboard</h2>
        <p>Glimpse of all activities of Bitbucket repository
<c>repository name</c></p>
        <table class="table">
            <thead>
                <tr>
                    <th>Activity</th>
                    <th>Action</th>
                    <th>Description</th>
                    <th>Data</th>
                </tr>
            </thead>
            <tbody ng-repeat="activity in Activities">
```

```
                <tr ng-class-odd="'warning'" ng-class-even="'success'">
                    <td>{{activity.Activity}}</td>
                    <td>{{activity.Action}}</td>
                    <td>{{activity.Description}}</td>
                    <td ng-
click="getActivity({{activity.ActivityId}})">View Details</td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

Our view represents a simple table, and we are using **ng-repeat** to bind our table with whole data.

b. Initialize WebHook controller

Open **WebAPIconfig.cs** and add following code:

Code Listing 27

```
// Initialize Bitbucket WebHook receiver
 config.InitializeReceiveBitbucketWebHooks();
```

This will initialize our Bitbucket WebHook receiver.

Note: Please make sure you have configured the Bitbucket repository to receive notifications and have added a secret key to your web.config file. Refer to Chapter 2 for more details.

1. Deploying to Azure

   In Chapter 2, we explained how to publish an application over Azure—if you skipped this chapter, please refer back for more details.

2. Perform operation and validate results

   Go to your Bitbucket repository and perform a few actions to push changes, create an issue, etc. If you have configured everything correctly, you will get the result on your index page.

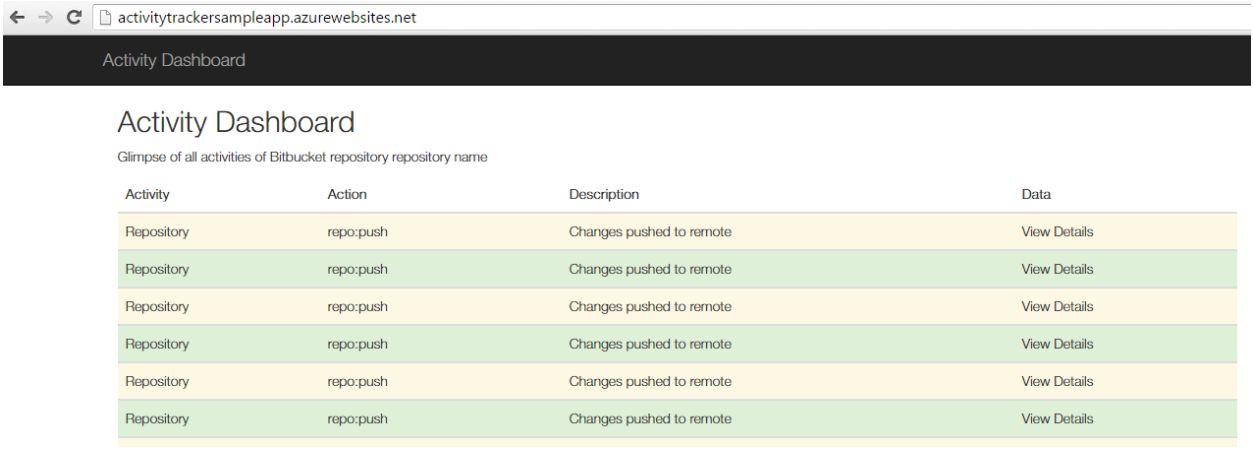

Figure 25: Activity Dashboard

Conclusion

In this chapter we have developed a real-time application using the Bitbucket Webhook Receiver. There is no limit to the ways you can extend the application. To keep it simple, we will just save the data to our database and display it in a table using MVC5 and AngularJS.

Chapter 4 Creating a WebHook Receiver

In previous chapters we have discussed how to consume existing WebHook receivers in our applications. To consume an existing Webhook receiver, we need to take a few steps (already discussed) and customize for our own application. At times it might be the case that the available WebHook receiver does not fit your design needs. In that case, you can create your own WebHook easily. We will dedicate this chapter to creating a new WebHook receiver for Zendesk. We will also walk through all steps for the creation of our new WebHook receiver.

Getting started

Any WebHook receiver is created for receiving notifications from a sender. In the same way, our purpose in creating a Zendesk WebHook receiver is to receive notifications from the Zendesk app.

Before we can start receiving notifications, we need to know what URI to register our WebHook with Zendesk. Additional information is also required, such as available Zendesk events, actions, and notifications available for subscription.

Note: To learn more about Zendesk push notifications, refer here.

Everything is documented at the Zendesk developers' website, which describes the expectations for push notifications for Android devices. The documentation summarizes all that is required to register our application and our device. A valid account is required with Zendesk portal for the registration process.

Tip: If you do not have an account with Zendesk, request a trial account to test your receiver.

Prerequisite

The following are prerequisites to get started with Zendesk WebHook Receiver:

- A valid account with Zendesk
- Understanding of the push notifications for Android devices from the Zendesk developer documentations portal
- Knowledge of ASP.NET Web
- Visual Studio 2015 or later
- Knowledge of ASP.Net WebHooks

Categories of WebHook receivers

I divided WebHooks receivers into two categories based on their availability for end users (who want to receive notifications) and developers (who implement these to their own applications). It will not always be the case that you want to make your WebHook Receivers available for everyone. Sometimes, there might be requirements to use a WebHook receiver internally within your own application or a in private network.

You can give any appropriate name to these categories. These names are based on my own views and understanding of WebHook receivers.

Private WebHook receivers

These WebHook receivers are not available publicly, so there are no publicly available NuGet packages for these receivers. However, you can create NuGet packages and use them on a private NuGet server (within your organization).

The main advantages of these WebHook receivers are:

- There are no development rules defined to write code. You can write code as per your own convenience, and in your own styles and coding standards.
- ASP.NET WebHooks source code is not required, so there's no need to set up a Github repository.
- No source code is required, so there's no need to make any **`pull`** request.

The only thing you should require for the creation of WebHook receivers is NuGet packages to setup base for WebHook Receiver, and these are:

- Microsoft.AspNet.WebHooks.Common (available here) – provides a common functionality and can be used for both Receiver and Sender
- Microsoft.AspNet.WebHooks.Receivers (available here) – provides a standard way to receive WebHooks.

Public WebHook receivers

These WebHook receivers are available publically and can be easily obtained as NuGet packages. Anyone can download NuGet packages and start their own implementation. These WebHook receivers are also part of ASP.Net WebHooks repository and can be released as official WebHook Receivers.

To start with public WebHook receivers, you need:

- To set up GitHub repository
- Follow documentation to write code
- While you write code, you should follow TDD (Test Driven Development)
- WebHook Receiver project name should be in the pattern of `Microsoft.AspNet.WebHooks.Receivers.<product-name-for-which-this-receiver-is>`. In our case, it will be `Microsoft.AspNet.WebHooks.Receivers.Zendesk`.
- All **`pull`** requests should be on **`Dev`** branch of the repository.

Your WebHook receiver will available once it's approved and the **`pull`** request is merged into the **`Dev`** branch. NuGet packages will be available once released.

Note: We will create Public WebHook receivers, so our Zendesk WebHook receiver will be available as a NuGet package.

Started writing code

In preceding sections, we covered the basic steps to create our Zendesk WebHook receiver. As we are going to create public WebHook receivers, we need to set up a GitHub repository so we can get the source code and make a **`pull`** request.

We won't discuss setting up GitHub repository, and getting the source code, or making a **`pull`** request, since these are all already discussed in Chapter 2. If you skipped it, please refer to Chapter 2 before proceeding.

Finally, we are ready to start adding a project and code for our Zendesk WebHook receiver.

Adding a project

Start Visual Studio and add a new Library Project under the folder **`src`**, and name it **`Microsoft.AspNet.WebHooks.Receivers.Zendesk`**.

Configuring the project

Open Solution Explorer and right-click on the project to get the Properties page of the project. Here you need to make few changes:

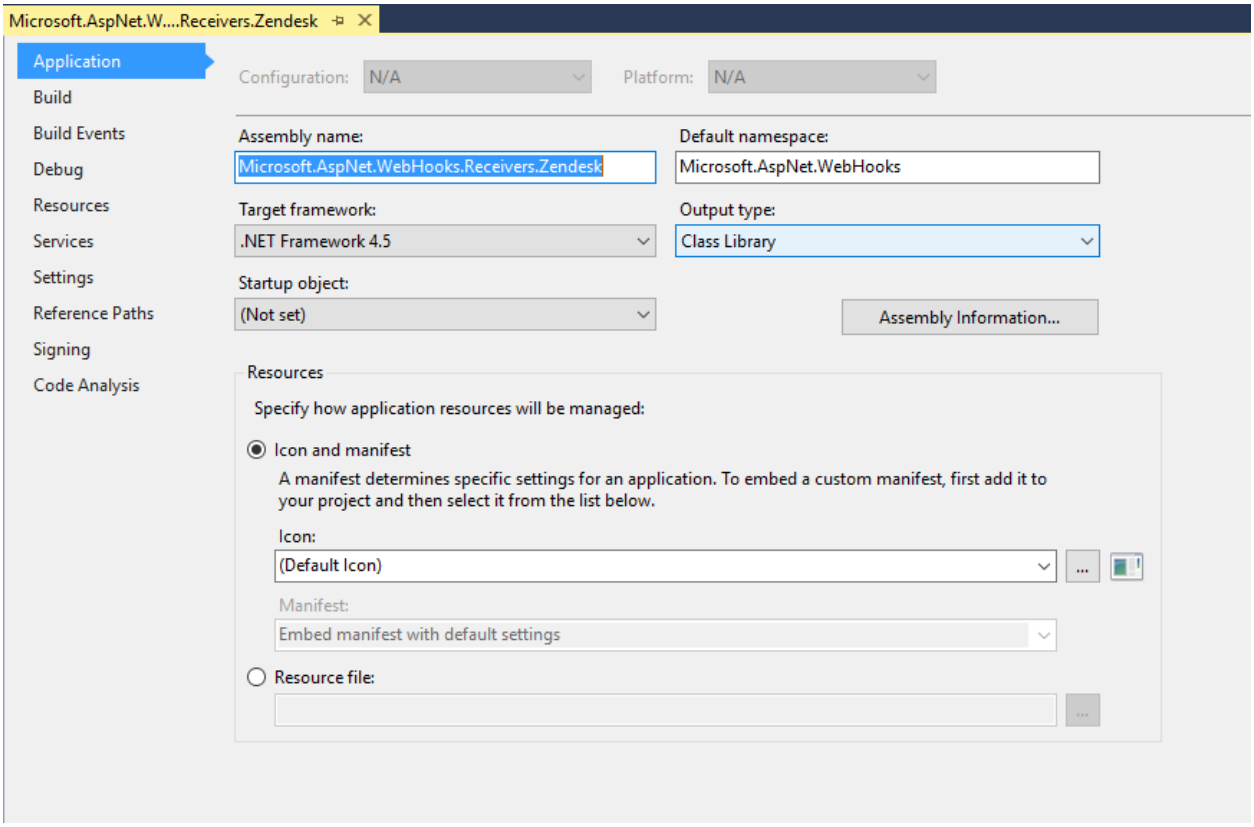

Figure 26: Project properties

Application

In this section of Project properties page, enter the **Assembly name** **`Microsoft.Aspnet.WebHooks.Receivers.Zendesk`**, and set the **Default namespace** to **`Microsoft.AspNet.WebHooks`**.

Build

Set **Conditional compilation symbols** as **`CODE_ANALYSIS;ASPNETWEBHOOKS`**.

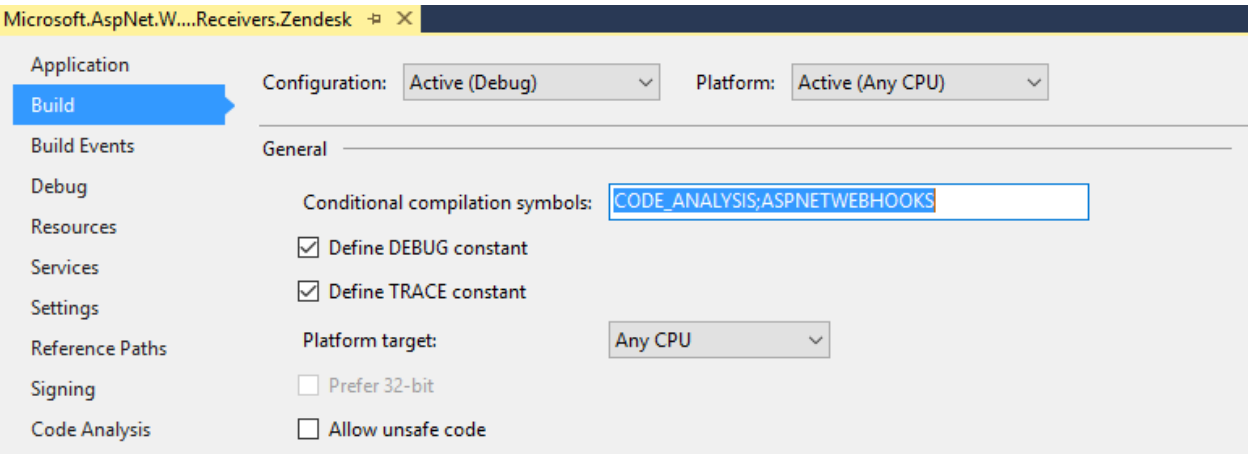

Figure 27: Conditional compilation symbols

Set **Output** to the common bin folder, including XML documentations.

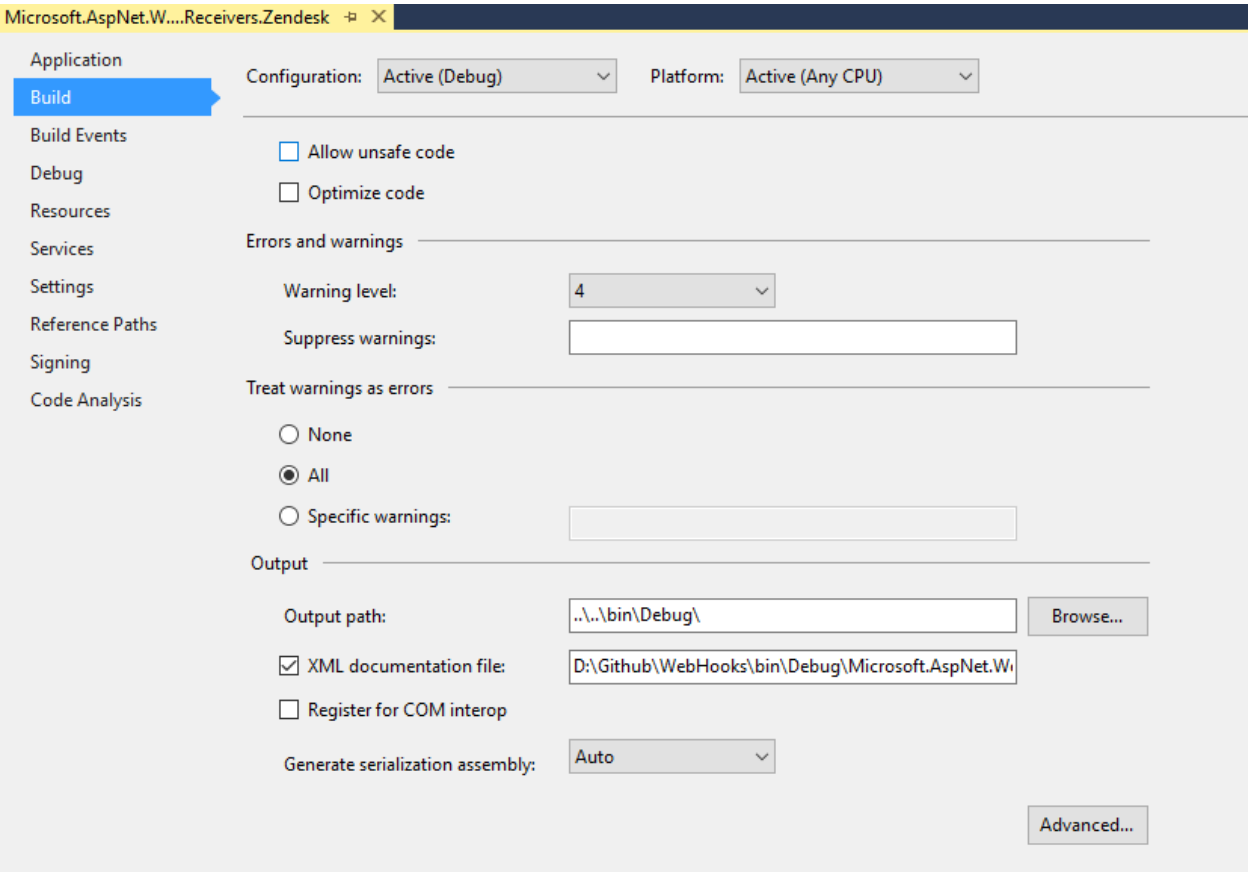

Figure 28: Setting output path

Signing

In the **Signing** section, choose a strong name key file, **35MSSharedLib1024.snk,** it is available here.

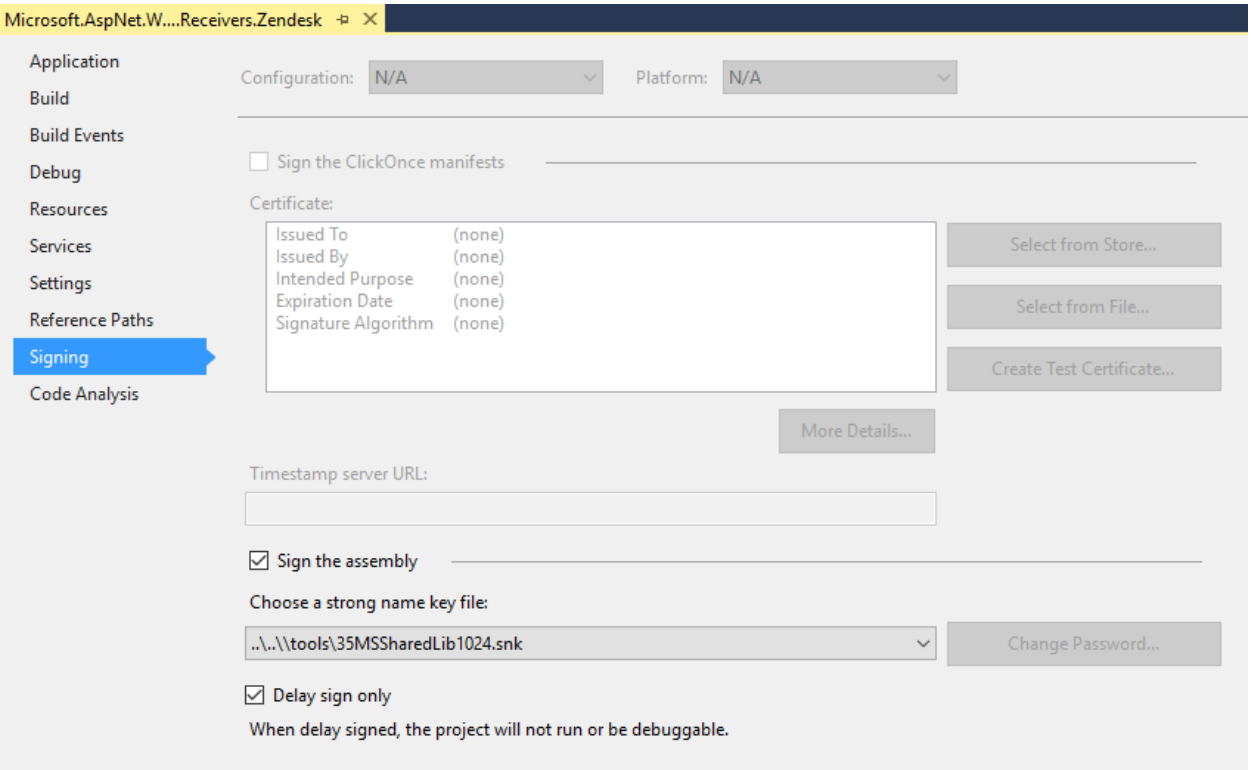

Figure 29 Sign the assembly

Code analysis

In the **Code Analysis** section, set FxCop rules and select the file **`FxCop.rulset`** available here.

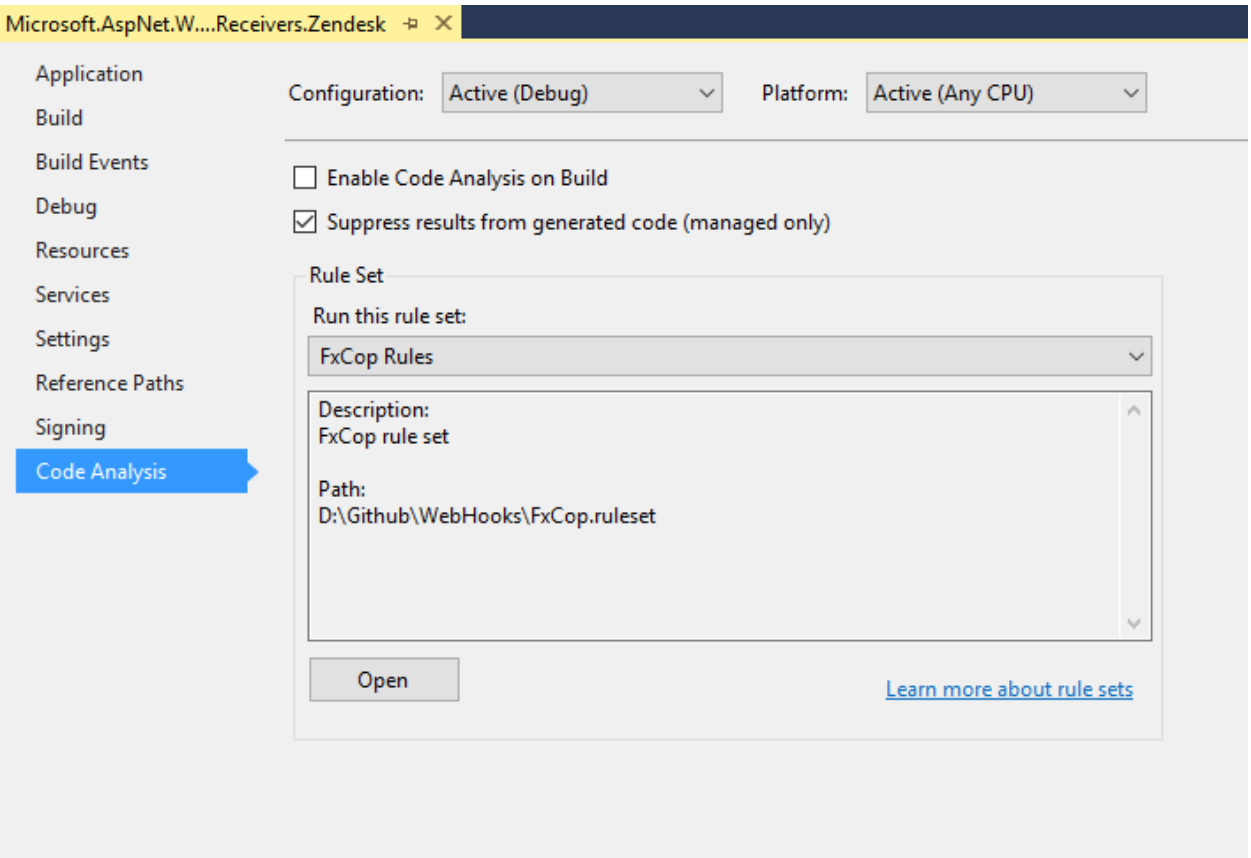

Figure 30: Setting Code Analysis rules

After all the changes, click **Save** to save the changes, and close the Properties pages.

Adding required files to project

To start writing actual code, we need to add certain files.

1. Open Solution Explorer, right-click on the project name, and select **Existing Item**. Select **CustomDictionary.xml** and add as a link. Now, right-click on the name of the same file from Solution Explorer and set **Build Action** as **CodeAnalysisDictionary**.
2. Open Solution Explorer and add the folders **`Extensions`** and **`Webhooks`**
3. Add the file **`HttpConfigurationExtensions.cs`** under the **`Extensions`** folder.
4. Under the **`WebHooks`** folder, add these files:
    a. **`ZendeskDevice.cs`**
    b. **`ZendeskNotification.cs`**
    c. **`ZendeskPost.cs`**
    d. **`ZendeskWebHookReceiver.cs`** (which is our main file, and responsible for receiving notifications).
5. You also need to set a common assembly and a resource file.

Initialization of WebHook

Our extension file **`HttpConfiguration Extension.cs`** is responsible for initializing the Zendesk WebHook receiver when in use from the application. It contains a simple extension method:

Code Listing 28

```
public static void InitializeReceiveZendeskWebHooks(this
HttpConfiguration config)
{
    WebHooksConfig.Initialize(config);
}
```

Writing the WebHook Receiver class

We have the WebHook receiver **`ZendeskWebhookReceiver`**, which implements the abstract class **`WebHookReceiver`**. Add the following code to this class:

Code Listing 29

```
public override async Task<HttpResponseMessage> ReceiveAsync(string id,
HttpRequestContext context, HttpRequestMessage request)
        {
            if (id == null)
            {
                throw new ArgumentNullException(nameof(id));
            }
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }
            if (request == null)
            {
                throw new ArgumentNullException(nameof(request));
            }
```

```
            if (request.Method == HttpMethod.Post)
            {
                // Ensure that we use https and have a valid code
parameter
                await EnsureValidCode(request, id);

                // Read the request entity body.
                JObject data = await ReadAsJsonAsync(request);

                // Call registered handlers
                return await ExecuteWebHookAsync(id, context, request,
new string[] { DefaultAction }, data);
            }
            else
            {
                return CreateBadMethodResponse(request);
            }
        }
```

We are done with the creation of Zendesk WebHook receivers. Please note that we did not write tests as per test-driven development (TDD); we should write tests simultaneously.

Writing tests

We should follow test-driven development while writing Zendesk WebHook receivers. To start with writing the test, we should follow these steps:

- Add new test project under the Test folder.
- Name the project Microsoft.AspNet.WebHooks.Receivers.Zendesk.Test
- Open the project Properties page and perform customization (refer to the “Configuring Project” section of Chapter 4).
- Add the folders **Webhooks** and **Messages**
- Under folder WebHooks, add the following files:
    - **ZendeskDeviceTest.cs**
    - **ZendeskNotificationTest.cs**

    - **ZendeskPostTest.cs**
    - **ZendeskWebHookReceiverTest.cs** (the main test file)
- Add NuGet package of **xunit**

Here is a code snippet of our test file:

Code Listing 30

```
[Fact]
        public void ReceiverName_IsConsistent()
        {
            // Arrange
            IWebHookReceiver rec = new ZendeskWebHookReceiver();
            string expected = "zendesk";

            // Act
            string actual1 = rec.Name;
            string actual2 = ZendeskWebHookReceiver.ReceiverName;

            // Assert
            Assert.Equal(expected, actual1);
            Assert.Equal(actual1, actual2);
        }

        [Fact]
        public async Task ReceiveAsync_Throws_IfPostIsNotUsingHttps()
        {
            // Arrange
            Initialize(TestSecret);
            _postRequest.RequestUri = new Uri("http://some.no.ssl.host");

            // Act
```

```
            HttpResponseException ex = await
Assert.ThrowsAsync<HttpResponseException>(() =>
ReceiverMock.Object.ReceiveAsync(TestId, RequestContext, _postRequest));


            // Assert

            HttpError error = await
ex.Response.Content.ReadAsAsync<HttpError>();

            Assert.Equal("The WebHook receiver
'ZendeskWebHookReceiverProxy' requires HTTPS in order to be secure.
Please register a WebHook URI of type 'https'.", error.Message);

            ReceiverMock.Protected()

                .Verify<Task<HttpResponseMessage>>("ExecuteWebHookAsync",
Times.Never(), TestId, RequestContext, _postRequest,
ItExpr.IsAny<IEnumerable<string>>(), ItExpr.IsAny<object>());

        }
```

Committing and creating a pull request

We are done with the creation of our Zendesk WebHook receiver, and now we have to commit our changes to the repository and make a pull request. We are not going to discuss this in details. Refer to GitHub Succinctly book for more details.

Tip: To learn more about creating a pull request, refer to this article.

We created a **`pull`** request for Zendesk, which has been approved and is now available publicly as a NuGet package here.

Conclusion

In this chapter we have discussed the creation of a WebHook receiver with a sample application.

Chapter 5 Writing Senders

In Chapter 4, we discussed and walked through all the steps required to create a new WebHook receiver. Now think of a scenario, where you want to create your own sender so, others can create or consume the WebHook receiver for your sender. In this chapter, we will walk you through the process of creating your own sender step-by-step.

WebHook senders

Writing your own sender is slightly different from consuming a receiver; the process is outlined in the following sections.

Subscription

A WebHook sender application should provide an interface, an API, or some way for users to subscribe to the events and receive notifications. These events may vary from application to application. Let us say that we would like to create WebHooks senders for Docunate™—for this our events would be:

- Document added, updated, deleted
- Document keywords added, updated, deleted

Management

A WebHook sender application should be capable of managing all users/subscribers, including registered WebHooks by them. There should be a way to persist this information. Don't let this statement sway your thoughts towards a database persistence—there is a way out, and we will be discussing that shortly.

Mapping

Subscribers and their subscriptions should be mapped in such a way that a particular event notification is sent to a subscriber who is registered for that event. In other words, if user **`A`** is subscribed for event the **`document:added`**, and user **`B`** is subscribed for the event **`keywords:added`**, the WebHook sender should send notifications to user **`A`** for **`document:added`**, and to user **`B`** for **`keywords:added`**.

Using ASP.NET WebHooks to create custom senders

It would be complex and require more effort than usual to implement everything from scratch. The managing of senders can also be sometimes painful. If we want to write our own senders from scratch, then we have to keep all these points (discussed in the preceding section) in mind, as ignoring these could make our life difficult. Thankfully, ASP.NET WebHooks provides a way to create your own senders and make your work very simple.

ASP.NET WebHooks provides us inherent support for registering subscribed events apart from managing and persisting subscription in Azure storage. More importantly, it provides us a means of handling errors and retrying notifications as many times as required. It also provides a means of maintaining queue and load balancing between sender and the receiver app. In short, ASP.NET WebHooks gives us pretty much everything we need to create our own sender.

Creating a sender depends on the nature of its application, so all senders will have some differences. For an example: Bitbucket sends notifications differently from GitHub notifications. The common denominator for any sender is event subscription, as every sender pushes notification for an event. In other words, both Bitbucket and GitHub send notifications, whenever a push event triggers (if someone has subscribed to these events, then they will get a notification for the same).

ASP.NET sample web app

In this section we will discuss a sample app meant to be our sender, which can be downloaded here. This sample application is meant to send notifications to subscribed users for preferred events. This app would also provide an interface to subscribe to all or specified events, so subscribers can receive notifications for those events. Our sample web application will have certain web APIs to perform specific tasks, and will expose all events so that subscribers can subscribe to all or specific events.

Tip: For more information on Web API, read ASP.NET Web API Succinctly.

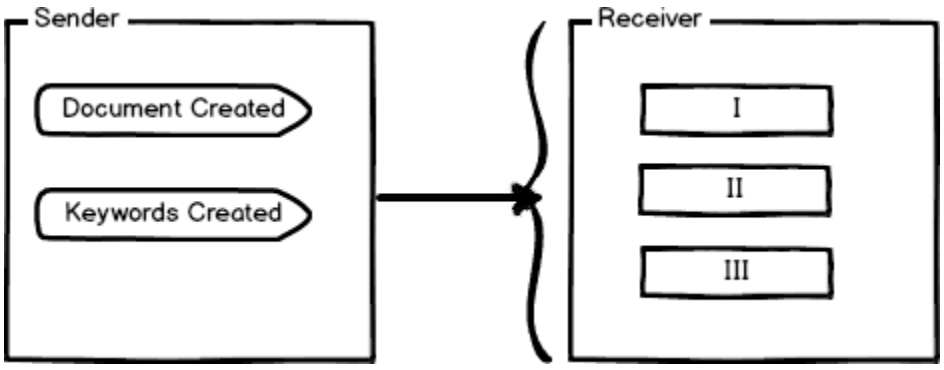

Figure 31: Pictorial view Senders to Receiver

Sender applications can send event notifications to all the receivers that may have subscribed, at once. However, receivers may or may not subscribe for more than one event. Receivers will receive notifications for all the events they have subscribed. It works in a way similar to having one post resource of a Web API; the post resource returns back the data to the user as soon as a post request completes.

Scenarios for sending notifications

Here are few scenarios where we can treat notification differently:

Consider a scenario where all users or a group of users need event notifications for all or one of the subscribed events. In this likely scenario, it becomes the responsibility of the sender to manage it. Our sample app (sender) is capable doing this as well. In the coming sections, we will discuss what base framework or projects we require to work with this application.

Another important scenario is where an anonymous user needs to receive notifications for subscribed events by a group to which this user belongs. Internally, the application sends notifications to a group and not to the individual user. So it's the responsibility of the subscribers to subscribe in such a way that whenever the receiver receives the notification, it will deliver it to all the group members. ASP.NET provides a way to send notifications to all users at once.

There could be times when we need to send notifications to all users except one or two in the group, or when the users are subscribed for a particular event only. In this scenario, ASP.NET helps us send notifications to all subscribers/users, and we can easily add simple logic to skip a particular user for notifications. We'd use something like: **`sendToAllUsers.where(s=> s.user != 'userA')`**.

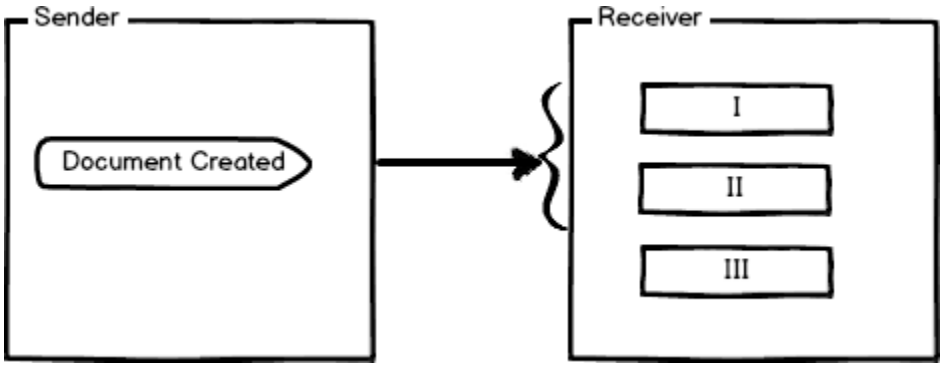

Figure 32: Pictorial view of sending notifications to limited number of users

Figure 32 is pictorial view for sending notifications to subscribed subscribers. Here, three subscribers are subscribed for the event **`document:created`**, but the sender is sending notifications to only Receiver I and Receiver II, while skipping Receiver III.

Requirement for creation of sample application

ASP.NET WebHooks provides NuGet packages with which we can easily create our sample application. The following NuGet packages are required to get started with our sample application:

- To start with the sample application, we first need to add support for WebHooks. This would enable the user to register for WebHooks. The **`Custom`** NuGet package fulfills this requirement, and is available here.
- The sample application needs to let the users register and provide a facility to subscribe to the events required. The **`Custom MVC`** NuGet package fulfills this requirement, and is available here.
- To fulfill Restful API support, we have a **`Custom API`** NuGet package, which is available here.

Apart from these NuGet packages, there is a need to persist and store subscription data. NuGet offers a package to fulfill this requirement with **AzureStorage**. It can be found here.

SqlStorage is another option for this persistence, and it can be found here.

Starting with the sample application

By now, we have a fundamental grasp of how to create senders and know the basic requirements to create our sample application. Here are a few points that will help us get started with the sample application:

- The sample application will use Azure storage to persist the subscriptions
- Rest API will be available for all subscribers

To get started with sample application, open Solution Explorer in Visual Studio, and then open the **WebAPIConfig** file. Add the following lines at the end of the **Register** method.

Code Listing 31

```
public static void Register(HttpConfiguration config)
        {
            // Other stuff related to configuration goes here

            // Load basic support for sending WebHooks
            config.InitializeCustomWebHooks();

            // Load Azure Storage for persisting subscriptions
            config.InitializeCustomWebHooksAzureStorage();

            // Load Web API controllers for managing subscriptions
            config.InitializeCustomWebHooksApis();
        }
```

In the preceding code listing, we are initializing support of **WebHook** in our sample app, and then support for **AzureStorage** and **Apis**.

Exposing events for subscription

As we have discussed in the preceding sections, the sender should expose events so that subscribers can subscribe to desired event for notifications. ASP.NET WebHooks provides a facility to declare events with the help of the **`IWebHookFilterProvider`** interface. These events are just a class derived from **`WebHookFilter`**. Since you can have more than one event, always make the event declaration a collection.

Code Listing 32

```
public class CustomFilterProvider : IWebHookFilterProvider

    {

        private readonly Collection<WebHookFilter> filters = new
Collection<WebHookFilter>

    {

        new WebHookFilter { Name = "created", Description = "Document
created." },

        new WebHookFilter { Name = "updated", Description = "Document
updated." },

        new WebHookFilter { Name = "deleted", Description = "Document
deleted." },

    };


        public Task<Collection<WebHookFilter>> GetFiltersAsync()

        {

            return Task.FromResult(this.filters);

        }

    }
```

In the preceding code listing, we defined three events, representing document creation, updates, and deletion. You can add more events here if you want to expose for subscribers.

Serving notifications

Now we just have to provide a way for our subscribers to get notifications to the subscribed events for the registered WebHook receivers.

Code Listing 33

```
public class DocumentApiController : ApiController

    {

        [HttpPost]
        public async Task<IHttpActionResult> Create(DocumentModel
document)

        {

            //Stuff to create document goes here

            DocumentModel documentModel=
DocumentProvider.Create(document);


            //Notify for doc:create action

            await this.NotifyAsync("doc:create",documentModel);

            return Ok();

        }

    }
```

In Code Listing 30, our API responsible for creation of a new document code is self-explanatory; here we are creating a new document by accepting a document model in the parameter of the **`Post`** resource, and immediately after creation of the document, we are notifying with data of a newly created document. This code is not complete, and is only an example to explain sending notifications.

Wrapping up WebHook sender

Earlier, we created a WebHook sender to create notifications for the WebHook receiver; the sample application we discussed is available here.

The next step would be to create a receiver to receive these notifications, which is beyond the scope for this chapter, but would follow the same steps outlined in Chapter 4.

Conclusion

In this chapter we have discussed the creation of a custom sender with a sample application. We looked at the basic framework required to create a sender, and discussed the complexity and scope of sending notifications.

Chapter 6 Diagnostics

One of the most important things in any production environment is to track any bugs and maintain the logs. In case of some weird behavior being exhibited by the application, this comes in handy, as we can't debug or track the issues directly. There might be a scenario like a broken external link or some external server being unavailable. ASP.NET WebHooks provides us with a way to diagnose our applications using WebHook handlers. In this chapter we will discuss logging and debugging an application.

Logging

Logging provides us an inside view of all the weird exceptions, errors, or hidden problems within our application, and is helpful in any production environment.

An application or software should be written such that it can report issues and problems effectively, and logging is the best way to report issues and problems. ASP.NET WebHooks provides inherent support for logging.

Tip: System.Diagnostics.Trace is a default method for writing a log internally by ASP.NET WebHooks.

Internally there is a **`TraceLogger.cs`** class, which implements the **`ILogger`** interface and writes the log. This has been implemented in such a manner that anyone can implement the logging system while utilizing frameworks like Log4Net or NLog.

In this book we are using an Azure app for our example applications. Let's discuss how to implement logging in the Azure apps.

Working with logs: Azure apps

On Azure, all logs are automatically available after logging is enabled. Take any of the examples discussed in the preceding chapters where we deployed our application as an Azure web app.

Enabling diagnostic logs

In this section we will discuss the steps to enable logs by using Azure portal, and also directly from within Visual Studio.

Using Azure portal

Azure provides web server diagnostics and application diagnostics. With application diagnostics, we can see or retrieve logs that are helpful in tracking down various issues.

Let us take a look at how to enable application diagnostics:

Log in to the Azure portal using valid credentials. Go to **Web Apps** and click on your app (**`ActivityTrackerSampleAPP`** in our case).

Tip: If you don't have an Azure account, you can create a free one here.

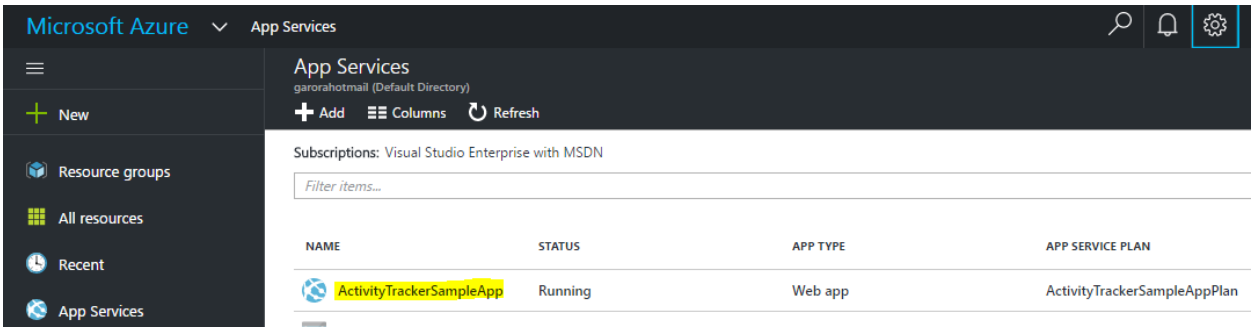

Figure 33: Web App - Azure portal

Select **Diagnostics logs** and enable the application diagnostics as shown in Figure 34.

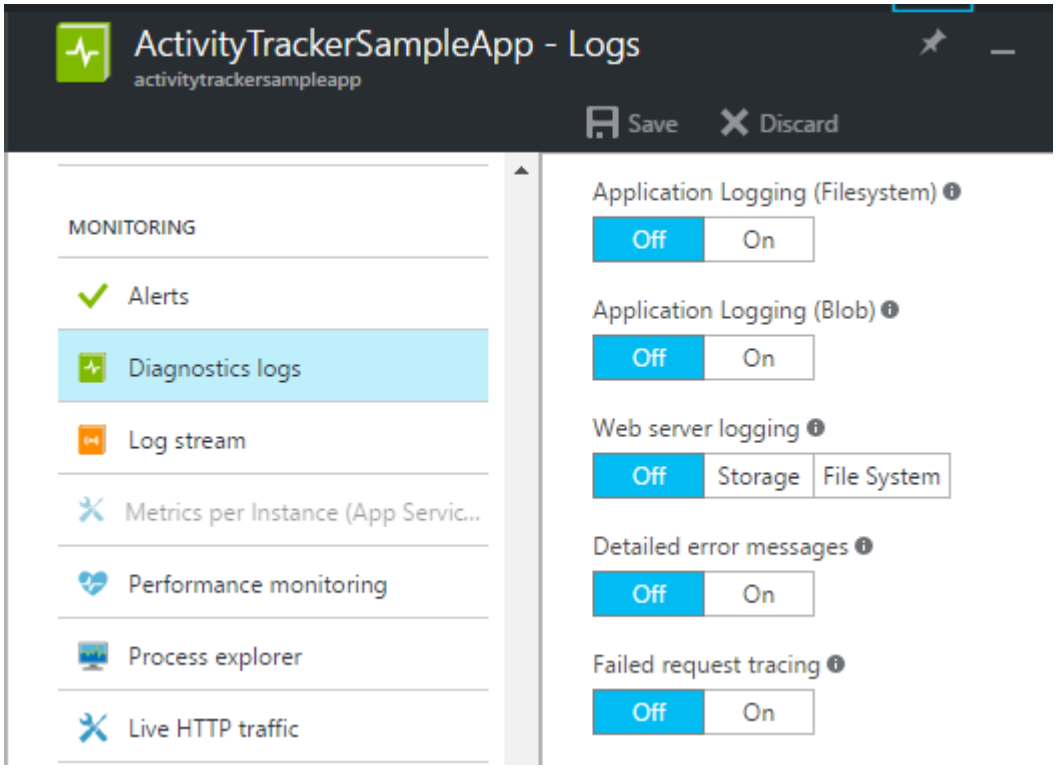

Figure 34: Enabling application logging

You can select **`verbose`** to capture everything (you can also select informational, warnings, or errors level as per your choice) with the help of the Azure portal; this is an easy way to enable diagnostics logs.

Note: To know more about enabling diagnostics, refer here.

Using Visual Studio

In this section we will discuss enabling the application logging using Visual Studio. All the examples are using Visual Studio 2015.

Start Visual Studio and open your ActivityTracker application, which we created in Chapter 3. This application has already been published to Azure—if you skipped Chapter 3, you can use the sample app from the Bitbucket repository by going to Chapter 3, opening up the application, and publishing over Azure.

Tip: Using Visual Studio with elevated administrator rights is recommended.

From Visual Studio, open Server Explorer and expand the App Service node, right-click on ActivityTrackerSampleApp (we have published this app), and then click View Settings.

Note: If you are not already logged in, Visual Studio will prompt you to log in using your Azure subscription credentials.

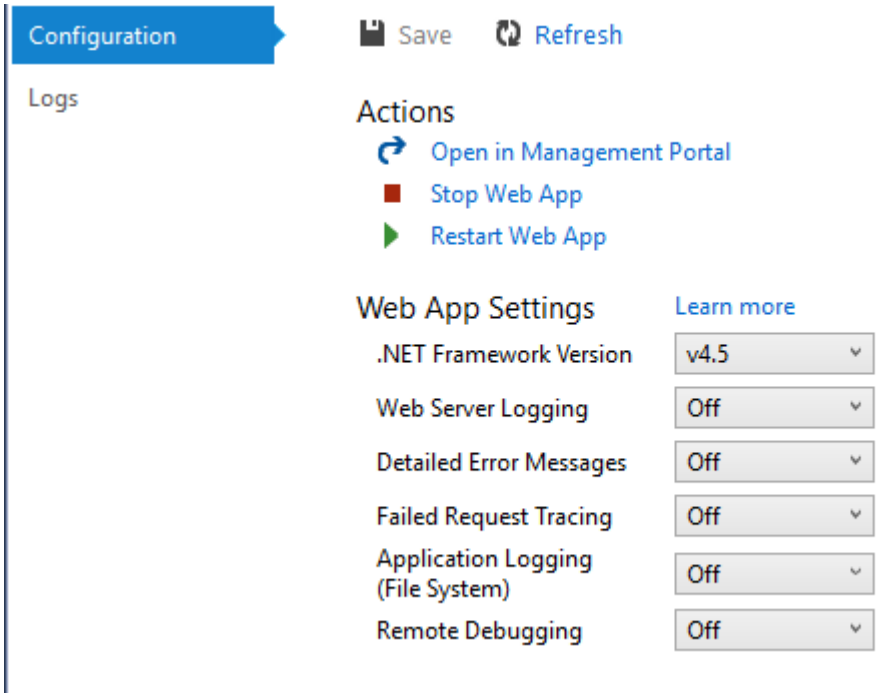

Figure 35: Enabling application logging - Visual Studio

On the View Setting page, under the Configuration tab, you can enable logging. Select **verbose** from the Application Logging (File System) dropdown menu, and click **S**ave to enable logging for all levels. The application is now ready to write logs.

Implement or redirect own logging implementation

As discussed previously, ASP.NET WebHooks internally writes logs using **`System.Diagnostics.Trace`**. But if you would like to use some other logging framework such as Log4Net or NLog, we just need to provide implementation of the **`ILogger`** interface, and simply register using any dependency injection engine.

Debugging

As we are deploying applications to Azure, we would like to debug our production or deployed code to track various issues. This requires remote debugging (where we attach our production environment and debug using Visual Studio on our local machine). This plays a very important role when you need to drill-down in your code to fix a problem.

In this section we will take sample application developed in Chapter 03. If it's not already deployed, you can deploy the application to Azure now.

To start, open the ActivityTracker sample application (created in Chapter 3) using Visual Studio. Open the Server Explorer, expand the **App Service** node, right-click on ActiviTyTrackerSampleApp, and choose Attach Debugger.

Note: Set break point before attaching process to start debugging.

As soon as you select **Attach Debugger**, Visual Studio starts applying debugger settings and invokes a web browser using the application URL. Now, go and push some changes to the repository for which you added a WebHook receiver (refer to Chapter 3).

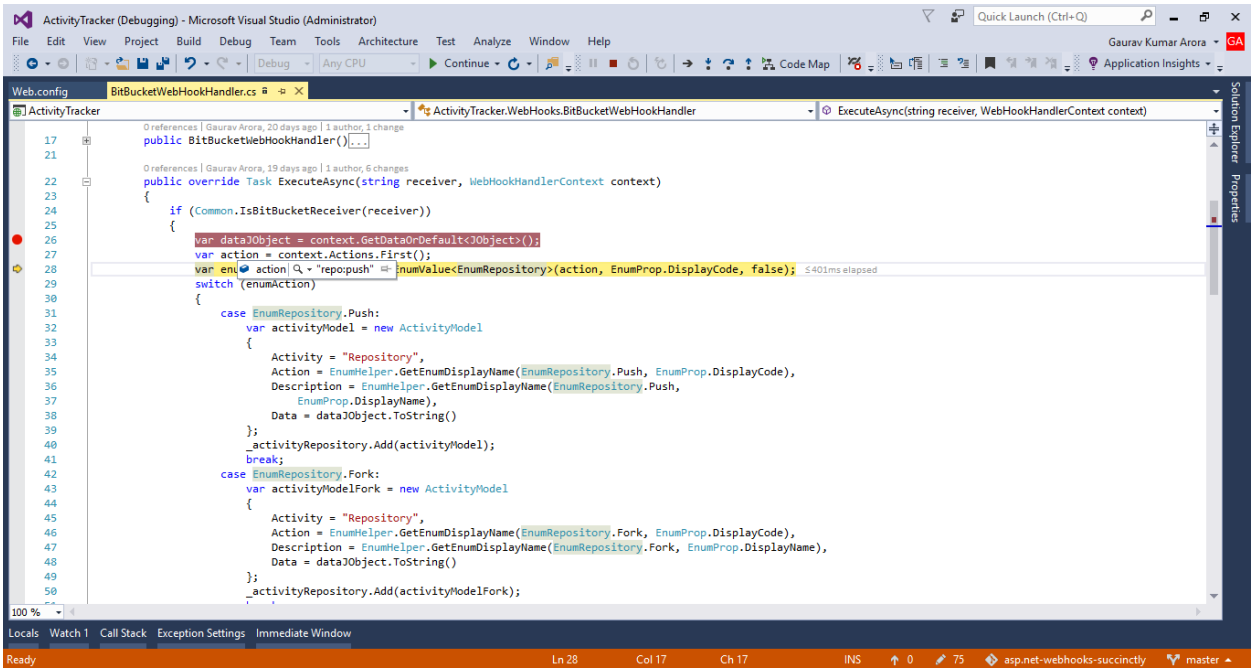

Figure 36: Debugging application

As soon as Bitbucket triggers a **`push`** event, the break point is hit, and you can see the value of data and the flow of logic while debugging.

Debugging directly with ASP.NET WebHooks

Directly debugging with the source code plays an important role while you are creating or customizing existing code. This allows you to directly debug the entire source code of ASP.NET WebHooks. In order to achieve this, you will have to follow few steps as mentioned in the official documentation of ASP.NET WebHooks.

Conclusion

In this chapter we have looked at diagnostics, and also discussed the importance of logging and debugging. We have debugged a live application and looked in detail at how to get application logs from Azure.

Chapter 7 Tips & Tricks

In this chapter we are going to discuss few tips and tricks for ASP.NET WebHooks. These tips and tricks will be helpful when you try to implement or customize WebHooks.

Working locally

In Chapter 2, we discussed the importance of SSL with public URL, and we deployed our sample applications to **`Azure`** as a **`Web APP`**. It is not possible or practical to create and deploy your application to Azure every time just for testing. To resolve this issue, there is a helpful tool that allows you to work locally to test your applications: ngrock. A full discussion is beyond the scope of this book, but you can get full details form the project's site.

Visual Studio extension

In Chapter 2, we discussed consuming WebHooks and went through the process of writing an application and deployment to Azure. While working well, this process takes time. Imagine if we had a tool that could helps us in a way similar to Plug and Play. There is a great Visual Studio extension available for this, which provides us an interface to set few configuration parameters and allow us to get notifications. You can get this extension here.

Most importantly, this is an open source extension, which mean you can add your own provider (if not listed in the extension). To do so, you just need to fork the repository and a create **`pull`** request for your provider.

Dependency injection implementation

ASP.NET WebHooks is built with the implementation of dependency injection, and you can use any dependency injection of your choice. For implementation, please refer to Dependency Scope Extension to know the list of all dependencies for **`Receivers`**. There is a default implementation, which is defined here: Receiver Services.

We are not going to explain our sample application, which is available on Chapter 7 of the Bitbucket repository.

Customizing logging framework

In Chapter 6, we discussed diagnostics where we went through the logging way of ASP.NET WebHooks. We also discuss the redirecting log by implementing any logging framework. We are not going to explain our sample application, which demonstrates the implementation of the Log4Net logging framework. You can find this sample application on Chapter 7 of the Bitbucket repository.

Consuming multiple WebHook receivers

In Chapter 2, we worked through consuming WebHook receivers, and covered the format of a valid URI and configuring a secret key. We also created a sample application to consume a WebHook receiver for single receiver. Our application can have multiple receivers at the same time. You can find the sample application on Chapter 7 of the Bitbucket repository.

Similarly, we can use multiple configurations for a single WebHook receiver. For example, our Bitbucket WebHook receiver can be set for three different secret keys—this is demonstrated in our sample application, and is available here.

Customizing notifications

In section WebHook processing we discussed about processing of Webhook. Typically, data is JSON or HTML, but we can cast to a specific type. Here's a typical example of Custom WebHook, where we demonstrate notifications to cast in CustomNotifications.

References

The following reference links will provide more information on various topics we've discussed in this book.

- https://en.wikipedia.org/wiki/Callback_(computer_programming)
- http://stackoverflow.com/questions/9596276/how-to-explain-callbacks-in-plain-english-how-are-they-different-from-calling-o
- https://msdn.microsoft.com/en-us/library/ff649664.aspx
- https://en.wikipedia.org/wiki/Webhook
- https://docs.asp.net/projects/webhooks/en/latest/
- https://github.com/aspnet/WebHooks/tree/master/samples/CustomSender
- https://blogs.msdn.microsoft.com/jjameson/2009/04/03/shared-assembly-info-in-visual-studio-projects/
- https://msdn.microsoft.com/en-in/library/7k989cfy(v=vs.90).aspx
- https://github.com/aspnet/WebHooks/tree/master/src/Microsoft.AspNet.WebHooks.Receivers.Zendesk
- https://ngrok.com/docs
- http://blogs.msdn.com/b/webdev/archive/2015/09/29/announcing-the-asp-net-webhooks-visual-studio-extension-preview.aspx
- https://visualstudiogallery.msdn.microsoft.com/bbfce065-ad62-4020-bed9-4b7d079f51e5/
- https://github.com/bradygaster/AspNet.WebHooks.ConnectedService
- https://github.com/aspnet/WebHooks/blob/master/src/Microsoft.AspNet.WebHooks.Receivers/Extensions/DependencyScopeExtensions.cs
- https://github.com/aspnet/WebHooks/blob/master/src/Microsoft.AspNet.WebHooks.Receivers/Services/ReceiverServices.cs
- https://en.wikipedia.org/wiki/Dependency_injection