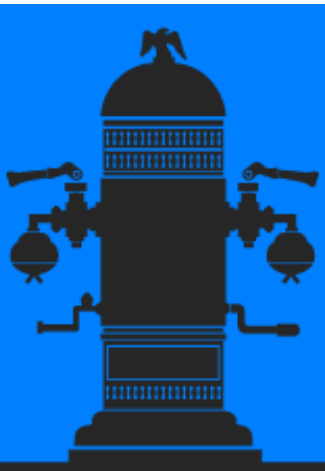

OWIN

Succinctly

by Ugo Lattanzi and Simone Chiaretta

Syncfusion | Technology Resource Portal

OWIN Succinctly

By
Ugo Lattanzi and Simone Chiaretta

Foreword by Daniel Jebaraj

Technical Reviewer: Tugberk Ugurlu
Copy Editor: Ben Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, 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!

About the Authors

Ugo Lattanzi is a programmer who specializes in enterprise applications, with focuses in web applications, service-oriented applications, and generally all environments where scalability is a top priority.

Thanks to the experience earned in recent years, Ugo is now focused on technologies like ASP.NET MVC, Node.js, Microsoft Azure, NServiceBus, AngularJS and HTML 5.

Thanks to this passion in web development using ASP.NET MVC, Microsoft recognized him as a Microsoft MVP in the technology.

Ugo is also a speaker for many important Italian technology communities, the author of several articles, and a co-organizer of the widely appreciated Web.NET European Conference in 2012. Away from the keyboard, he is a bad snowboarder but a good father.

Simone Chiaretta is a web architect and developer who enjoys sharing his development experiences and more than 15 years' worth of knowledge on web development with ASP.NET and other web technologies.

He has been a Microsoft MVP on ASP.NET for seven years, authored several books about ASP.NET MVC (*Beginning ASP.NET MVC 1.0* and *What's New in ASP.NET MVC 2*, both published by Wrox*,* and *OWIN Succinctly*, published by Syncfusion), spoken at many international developer conferences, and contributed to online developer portals like Simple-Talk. Simone also co-founded the Italian ALT.NET user group ugialt.NET and is the co-organizer of many conferences in Milan, including the widely appreciated Web.NET European Conference in 2012.

When not writing code, blog posts, or taking part in the worldwide .NET community, he likes to play with Arduino, drones, and underwater robots, and tries to train for triathlons.

He is one of the many expats living and working in the capital of Europe, Brussels.

About the Book

The world of web frameworks is evolving fast, mainly driven by the fast pace of open source communities. To keep up with this pace, even traditionally slow software companies are starting to build their solutions on top of those community-driven tools and standards.

In this book, we are going to look at one of those community-driven standards, OWIN, and how Microsoft has fully embraced this standard. Microsoft has created Katana, an OWIN-based web server, and the first step toward a fully compatible OWIN web stack.

Chapter 1 OWIN

Introduction

In this first chapter, you are going to learn how the world of web development has dramatically changed in the last few years, and how a group of .NET developers have defined a small set of specifications to try to make web development in the .NET world easier. You will also learn how Microsoft embraced these specifications, and why it is basing the future of ASP.NET on it.

The web is evolving fast, and web development needs to keep up with it. For this reason, all framework vendors need to offer good solutions for developers to deal with their latest needs.

One of these needs is to have an abstraction between the web application and the web server. This has several advantages, which we will discuss later in the book.

OWIN bridges this gap between web application and web server in the .NET world by offering different solutions like Katana or the open source community-driven Nowin.

What is OWIN?

OWIN is the acronym of **Open Web Server Interface for .NET**. It is defined on the project website as follows.

> *OWIN defines a standard interface between .NET web servers and web applications. The goal of the OWIN interface is to decouple server and application, encourage the development of simple modules for .NET web development, and, by being an open standard, stimulate the open source ecosystem of .NET web development tools.*

As you can imagine after reading the definition, there is no code for OWIN, just specifications for the real implementation of a custom web server.

After two years (2010–2012), OWIN's specification was finally completed and the current version is 1.0. If you are curious and would like to see the history of the specification, the draft history is also available on the OWIN project website.

Why Do We Need OWIN?

You have probably already figured out some of the reasons that make OWIN an important factor in web applications, but there are others that are not as obvious.

Firstly, OWIN aims at defining a standard interface for web servers and .NET client applications to interact, whereas Katana and Nowin are the implementations.

Moreover, like the OWIN website says, the design of OWIN is inspired by Node.js, WSGI on Python, and Rack, which is a similar lightweight server for Ruby.

Despite being an inspiration and having many similarities, there are also important differences between Node.js and OWIN: the OWIN specification mentions a web server like something that runs on the server, answers to HTTP requests, and forwards them to its middleware. On the other hand, Node.js is the web server that runs under your code, so you have total control of it.

So then, why is OWIN so cool, and why do we need it?

Well, Internet Information Services (IIS) is a super-tested web server and it will probably have a long life in .NET applications, but it has several limitations.

First of all, IIS is related to the operating system. It means that you have to wait for the new release of Windows to have new features (e.g., WebSockets are available only on the latest version of Windows). You have to completely update the web server. Often it is not easy to update a web server, and system admins don't like a request like "I need the latest version of Windows because I need to use WebSockets."

With OWIN, your code is not related to the OS (specifically to System.Web, the “huge” monolithic library that lies behind the execution of ASP.NET). This means that you can use whatever you want instead of IIS (i.e. Katana or Nowin) and update it when necessary, instead of updating the OS. Moreover, if you need it, you can build your custom host and insert whatever you want in the HTTP request processing pipeline (i.e. your custom authentication logic).

All these things are possible thanks to OWIN, which provides the specifications for writing those "extension points" that make it easy to insert into the chain. In a perfect world, where all the web servers observe OWIN, you could also change the web server, keeping your extension points without writing one line of code.

Another important point could be performance, but we will explain it later when we talk about the pipeline.

When is it Right to Use OWIN?

The short answer is: *"Whenever the framework you are using allows it."*

This is because OWIN, and in particular Katana, is becoming part of the web development stack from Microsoft (and with ASP.NET vNext due in 2015, this will be even more true). Some frameworks have integrated the paradigms of OWIN, while others are not yet compatible. For example, Web API and SignalR are compatible with OWIN, and can be used in the OWIN middleware pipeline, while ASP.NET MVC (prior to v5) is still incompatible. Nancy, FubuMVC, and other third-party web frameworks are also compatible with OWIN. In Chapter 3, we will look at how to use these frameworks with Katana.

The question of compatibility will not be relevant for long, because the new wave of ASP.NET web frameworks will be based on OWIN, including the new version of ASP.NET MVC (v6). Soon you will be using software built on top of OWIN specifications even without knowing it. Some of the current version of ASP.NET MVC is also based on OWIN, as you will see in Chapter 5.

OWIN Specifications

Now that we have an idea of what OWIN is and the scenarios in which a server that respects OWIN specs is useful, it is time to analyze the specifications in detail.

As anticipated at the beginning of the chapter, the OWIN specifications are pretty simple. They define a set of layers, the order in which they are stacked, and the interface, referred to as the application delegate or **`AppFunc`**, which is used by these layers to communicate with each other.

OWIN Layers

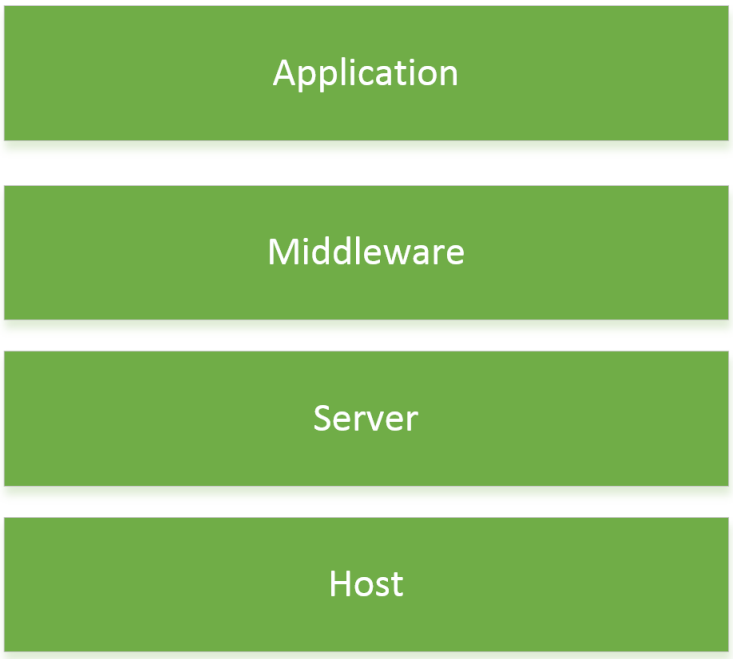

Figure 1: OWIN layers

The OWIN specs define four layers:

- **Host**: The process inside which all the other layers are executed. It is mainly responsible for the application configuration and startup of the process, including launching the server.
- **Server**: This is the *actual* HTTP server, the one binding to the network socket to listen to requests and sending them to the pipeline of OWIN middleware components.

- **Middleware**: These pass-through components stand between the server and the final application code, and handle the requests sent through the OWIN pipeline. These components can be as simple as a logger or as complex as full web frameworks like Web API or SignalR. As far as the server is concerned, a middleware component is anything that implements the application delegate.
- **Application**: This is the specific application code, possibly built on top of a web framework. Since the code of your application will only interact with the web framework, the only interaction you will have with the OWIN part of the pipeline will be configuration of the pipeline, but nothing more than this.

The Application Delegate

The other important part of the specs is the definition of the interface used by middleware components to interact with the server in the OWIN pipeline. It is not an interface in the strict sense of the term, but it is a delegate that each OWIN middleware component should provide.

```
using AppFunc = Func<
  IDictionary<string, object>, // Environment
  Task>; // Done
```

Code Listing 1

In plain English, the previous code means that each OWIN middleware must have a method that receives as input a variable of type **`IDictionary<string, object>`** (known as the environment dictionary) with the state of the running request, and must return a **`Task`** object to allow the asynchronous execution of the component.

The Environment Dictionary

The environment dictionary contains all the information about the request, response object, and any server state. It is the responsibility of the server to create the dictionary, fill it with the streams used to write the response, and read the possible request body with all the headers.

Through the pipeline, each component and layer may add additional data, but the specs define a set of mandatory keys that always have to be present.

Table 1: Mandatory Keys in Request Data

Key Name	Description
owin.RequestBody	A **`Stream`** with the request body, if any. **`Stream.Null`** MAY be used as a placeholder if there is no request body.

Key Name	Description
owin.RequestHeaders	A **IDictionary<string, string[]>** of request headers.
owin.RequestMethod	A **string** containing the HTTP request method of the request (e.g., GET, POST).
owin.RequestPath	A **string** containing the request path. The path must be relative to the root of the application delegate.
owin.RequestPathBase	A **string** containing the portion of the request path corresponding to the root of the application delegate.
owin.RequestProtocol	A **string** containing the protocol name and version (e.g., HTTP/1.0 or HTTP/1.1).
owin.RequestQueryString	A **string** containing the query string component of the HTTP request URI, without the leading “?” (e.g., "foo=bar&baz=quux"). The value may be an empty string.
owin.RequestScheme	A **string** containing the URI scheme used for the request (e.g., http, https).

Table 2: Mandatory Keys in Response Data

Key Name	Description
owin.ResponseBody	A **Stream** used to write out the response body, if any.
owin.ResponseHeaders	An **IDictionary<string, string[]>** of response headers.

Key Name	Description
owin.ResponseStatusCode	An optional **`int`** containing the HTTP response status code as defined in RFC 2616 section 6.1.1. The default is 200.
owin.ResponseReasonPhrase	An optional **`string`** containing the reason phrase associated with the given status code. If none is provided then the server should provide a default as described in RFC 2616 section 6.1.1.
owin.ResponseProtocol	An optional **`string`** containing the protocol name and version (e.g. HTTP/1.0 or HTTP/1.1). If none is provided, then the owin.RequestProtocol key's value is the default.

Table 3: Other Mandatory Keys

Key Name	Description
owin.CallCancelled	A **`CancellationToken`** indicating if the request has been cancelled or aborted.
owin.Version	The **`string`** 1.0 indicating OWIN version.

The Application Startup

The specs, specifically Section 4, also define how the startup sequence must be, and how new middleware should be added to the application. Until a few months ago, the OWIN group provided a non-normative **`IAppBuilder`** to formalize how the OWIN pipeline starts and how middleware is added, but it has recently been removed as this topic is still being discussed in the mailing list. Expect changes on this part of the spec in the future.

General Consideration of the Specs

These specs might seem small, but it is in their simplicity that lies their importance: by requiring the dependency on basically only one type (the `AppFunc`), they lower the entry barrier for developers who want to write OWIN components. Also, using a dictionary allows interoperability among components and frameworks without requiring all the different developers to agree on a specific structured object model, something that is very difficult to achieve. The three different web frameworks built by Microsoft—Web Forms, ASP.NET MVC, and Web API—have three different object models for maintaining the state of the current request.

Introducing Katana

As we have just seen, OWIN is nothing but a few specs. For them to be usable in a real project, there must be something that implements them. At the time of writing, only two publicly released servers implement these specs: Katana, developed by Microsoft, and Nowin. In the rest of the book, we will use the Katana implementation every time we need to refer to a concrete implementation.

A Brief History of Katana

Before looking at how to use Katana, we need to understand how it came to be by looking back more than 10 years through the evolution of ASP.NET.

ASP.NET Web Forms

First came ASP.NET Web Forms in 2001, which was built with two classes of developers in mind:

- Developers coming from ASP Classic, who were used to building dynamic websites using a mix of HTML markup and server-side scripting. In addition, the ASP runtime abstracted the underlying HTTP connection and web server, and provided to the developers a set of objects that allowed them to easily interact with the current HTTP request. It also provided additional services such as session management, state, caching, and more.
- Developers coming from the more traditional Windows Forms application development. Those developers did not know how to write HTML, and were used to building applications by dragging elements on a design canvas.

To cater to both classes of developers, the first ASP.NET web framework, called Web Forms, was built with the goal of hiding the stateless nature of the web with a server-side event model, mainly implemented via the infamous ViewState.

The result was successful, with a feature-rich web framework and approachable programming model. But all this came with a few limitations:

- All the features were shipped together in a monolithic framework, heavily coupled with the core web abstracting library, System.Web.

- Being heavily based on a design time programming model, ASP.NET was tied to the larger .NET framework and also to Visual Studio; this meant that, apart from quick fixes, years were passing between releases of the web framework.
- Last but not least, the ASP.NET framework (via its core library System.Web) was also coupled and usable only with Microsoft's Internet Information Services (IIS) web server.

ASP.NET MVC

Over time, those limitations, which at the beginning probably did not seem very dangerous, began to show their costs. It was becoming very difficult for Microsoft to adapt to the fast pace of other frameworks and languages, which were built as small and very focused components, rather than huge frameworks.

In addition, the style of web development changed: while at the beginning having the framework abstract away HTTP and even HTML markup was helping non-web developers build web applications, now developers wanted more control, especially on the markup that was being rendered on pages.

To address these two problems, the ASP.NET team developed the ASP.NET MVC framework, adopting the Model-View-Controller design pattern to keep a clean separation between business logic and presentation logic, while allowing developers to have complete control of the markup. In addition, they decided to release the framework out of band and not coupled with the .NET framework, making faster and more frequent releases possible.

Some problems still remained, most notably the dependence on the huge System.Web DLL, which made ASP.NET MVC not totally independent from the larger .NET framework.

ASP.NET Web API

A few years later, another big change happened in the way web applications were built. Instead of server-generated data-driven pages, applications were being built as mostly static pages, some portions of which were generated by interacting with web APIs via AJAX calls.

The ASP.NET team started building a new framework, called ASP.NET Web API, to better approach this paradigm of web development. They also took the opportunity to move toward an even more modular component model by ditching the dependency on System.Web and building a framework that could evolve independently from the rest of ASP.NET, and much quicker thanks to the recent introduction of NuGet, the package manager from which all .NET libraries are easily installed. On top of this, since it didn't depend on System.Web, it also did not depend on IIS, allowing the possibility of running on custom hosts and other web servers.

Here Comes Katana

With Web API being able to run in its own lightweight host, and with the growth of more modular and very focused frameworks, there was the concrete risk that developers had to start separate processes to handle the various components of a modern web application, like static files, dynamic files, web APIs, and web sockets.

To avoid the proliferation of processes, all of which would need to be started, stopped, and managed independently, a common hosting process was needed to which additional features and modules could be easily added.

This is why OWIN was born, to standardize the way frameworks and components could be easily plugged into those hosting processes. Katana is the hosting process and server built by Microsoft on top of these specifications.

How to Get Katana

Katana is released out of band and is available as a NuGet package that you can directly include in your projects.

In the next chapter, we will see how to create your first simple web application running on Katana.

License

Like every other open source project, OWIN and Katana have a license to which you have to comply when you want to use them.

Luckily, the license used is the Apache License 2.0, which is very permissive and basically gives you the right to use the software however you want, as long as:

- The original copyright is retained.
- The license is included in modified software.
- If significant changes are made to the software, they must be stated somewhere.

Conclusions

In this chapter, you have learned what OWIN is, how OWIN came to be, and what the specifications of OWIN are.

You had a brief recap on the history of the web development stacks from Microsoft, why now is the right time for Microsoft to embrace the OWIN specs. Microsoft has built its own implementation called Katana, and plans to build its future web development platforms on top of OWIN.

In the next chapter, you will learn what exactly Katana is, how its components map to the OWIN specs, and how to start building you first "Hello World" web apps on Katana.

Chapter 2 Katana

Introduction

In the previous chapter, we learned what OWIN is, when it is a good idea to use it (if you have forgotten, the answer is *always*), its specifications, and how Microsoft came to embrace this standard by building its own implementation called Katana.

In this chapter, you will learn how exactly Katana implements the OWIN specs, how it enhances them, and how to write your first "Hello World" web application with Katana and its different hosting options.

Later in the chapter, you will also learn how to customize the startup process by choosing which **`Startup`** class to use.

What is Katana

As seen in Chapter 1, Katana is the Microsoft implementation of the OWIN specs, and provides all the layers, sometimes in more than one flavor, specified by OWIN. In addition to implementing hosts and servers, Katana provides a series of APIs to facilitate the development of OWIN applications, including some functional components like authentication, diagnostics, static files serving, and bindings for ASP.NET Web API and SignalR. To avoid confusion, remember that Katana is not a full-fledged web server, but just the “glue” between the OWIN world and IIS.

The Principles of Katana

In line with the guiding principles of OWIN, Katana was built with three main objectives in mind:

- **Portable**: Every component in the pipeline should be replaceable, from the server and host to the web frameworks. Frameworks built by other companies or open source projects should be able to run on top of Microsoft OWIN servers and Microsoft middleware components; frameworks should potentially run on top of server and hosts coming from other OWIN-based projects.
- **Modular**: All components shipped with Katana should be very focused and provide just one functionality so that application developers can mix and match the components that they need. This keeps the size of the application small, instead of giving developers a myriad of features they don't need.
- **Lightweight and performant**: A consequence of the modular approach is that applications will consume only the resources that are really needed. In addition, since components are developed to be portable, they can be replaced with more performant implementations whenever they become available.

Katana Architecture

Before diving into coding your first Katana application, it is important to understand its architecture, and how the layers defined by the OWIN specs, **Host**, **Server**, **Middleware,** and **Application**, map on top of Katana components.

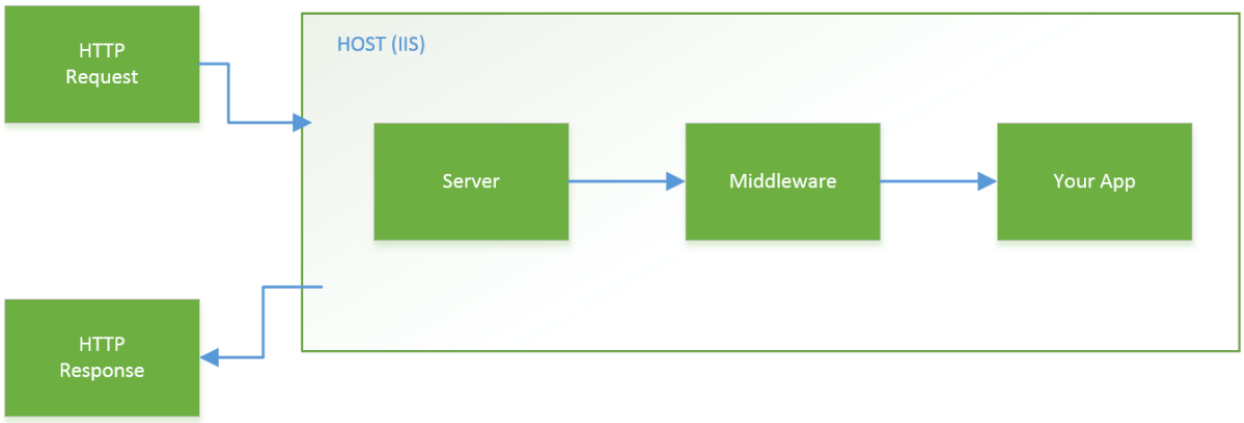

Figure 2: Architecture of Katana

Host

The first and lower layer defined in the OWIN specs is the host. It is responsible for the management of the underlying process (startup, shut down, and so on) and the initialization of the OWIN pipeline that will later be used to process incoming requests.

There are three options for Katana-based applications:

- **IIS/ASP.NET**—The easiest and more backward compatible option is using IIS: in this scenario the OWIN pipeline is started via a standard **`HttpModule`** and **`HttpHandler`**. Using this hosting option you are also obliged to use the OWIN server that relies on System.Web as IIS is both host and server.
- **Custom Host**—If you want the option to use a different server instead of System.Web, or want to have more control on the hosting process and application configuration, another option is to build a custom hosting process. This allows developers to host their server in a Windows Service, a console application, or even, for example, easily embed their own server inside a desktop application.
- OwinHost—The last option is for developers who don't want to use IIS/ASP.NET to host their application but also don't want to build their own custom host and would rather rely on something already built. For this purpose, the Katana suite includes OwinHost.exe: a command line application that, when launched from a project's root folder, starts the server using the **`HttpListener`** server and finds the startup class by convention. It has a few command line options that can be set to customize the way it behaves. You can, for example, point it manually to the application startup class or change the server used (if you don't want to use the default **`HttpListener`**).

Server

The next layer is the server, which is responsible for opening the network socket connection on which the application will respond. It listens for requests and, once one arrives, it populates the environment dictionary with the values coming from the HTTP request and sends it down through the pipeline of OWIN middleware defined by the developer in the application startup code.

System.Web

As previously mentioned, System.Web and the IIS/ASP.NET hosting option cannot be chosen independently one from the other. When using this option, the Katana System.Web-based server registers itself as **`HttpModule`** and **`HttpHandler`** and will process the requests sent to it by IIS. It will then map the well-known **`HttpRequest`** and **`HttpResponse`** objects to OWIN's environment dictionary and send the request to the OWIN pipeline.

HttpListener

The other option currently available is based on the lower-level networking class **`HttpListener`**. This is the default server when using OwinHost or self-host.

WebListener

This is the super lightweight OWIN server that will come with ASP.NET vNext. It can't currently be used with Katana, but it's worth knowing what's in the pipeline.

Middleware

After the first two layers, which basically just wire up the process, the request finally enters the pipeline of OWIN middleware. Middleware is a small component that implements the OWIN application delegate **`AppFunc`**.

Middleware should process the request and pass the execution to the next component defined in the pipeline. What middleware does while processing the request can be either very simple or very complicated. It could just be a simple logging component, but it can also be a full web framework like Nancy, SignalR, or ASP.NET Web API. It can even be a small, self-contained web application that you just want to plug into your web project.

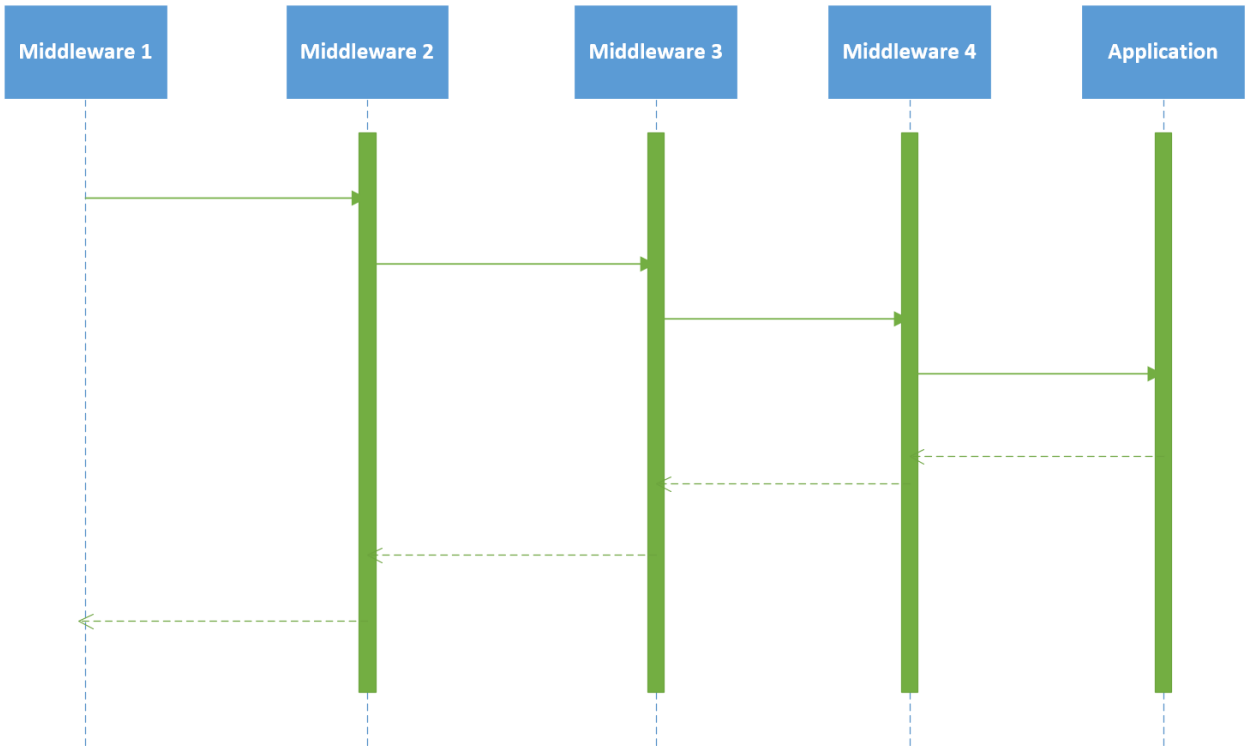

Figure 3: Sequence diagram of the invocation of middleware

In Chapter 4, you will learn how to build your own middleware component.

Application

Finally, the top layer of the OWIN specs is the code of your application.

Katana on IIS

Now that you've learned about the architecture and the options available with the Katana suite, it is time to get your hands dirty writing a Katana-based application. The application will just print the message **`Hello World!`** in the output, which is a webpage in this case.

Create the Project

We will first start with building a Katana application that runs using the IIS/ASP.NET host and the System.Web server. To do so, first you need to open Visual Studio and start by creating a new **ASP.NET Web Application** on the .NET Framework 4.5.

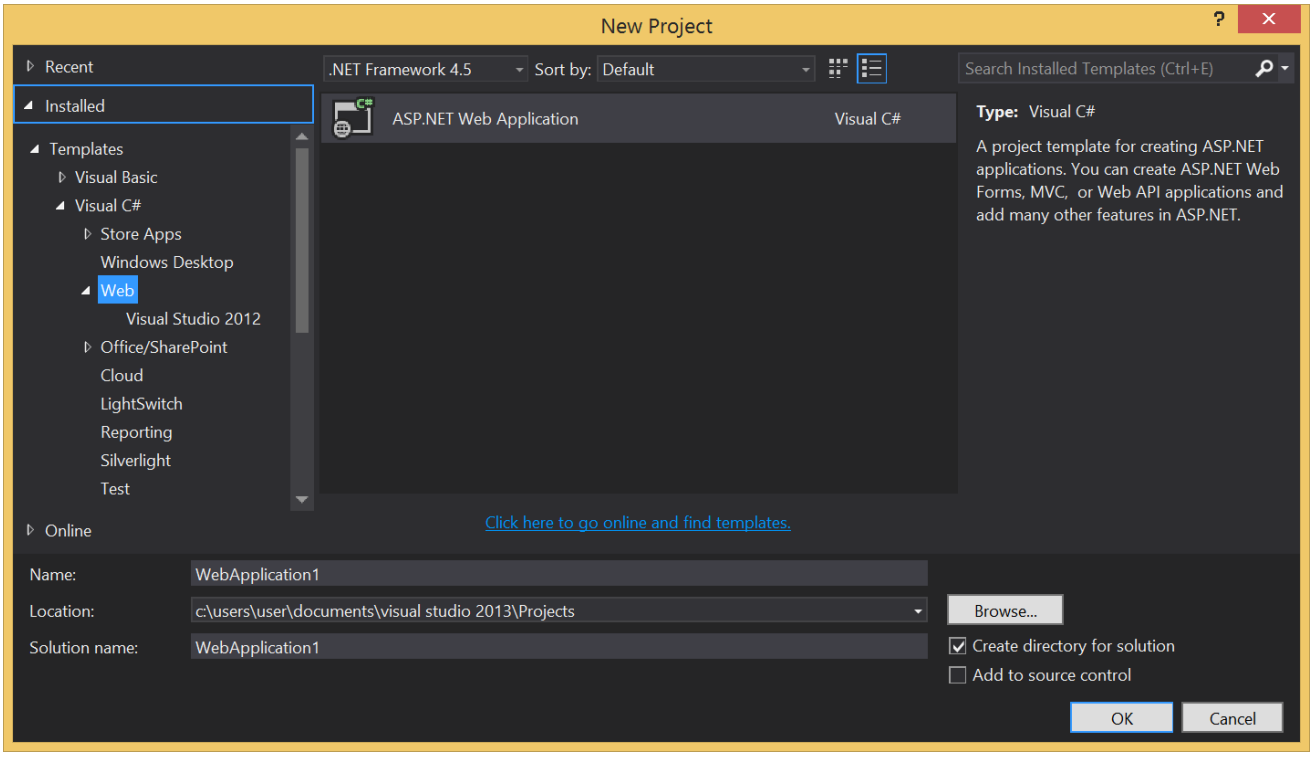

Figure 4: New Project window

Once you've chosen the kind of project, the **New ASP.NET Project** creation wizard will appear. Make sure you use the following settings:

- Select the **Empty** template.
- Select **No Authentication**.
- Do not select any additional folder or core reference.
- Do not host in Azure.

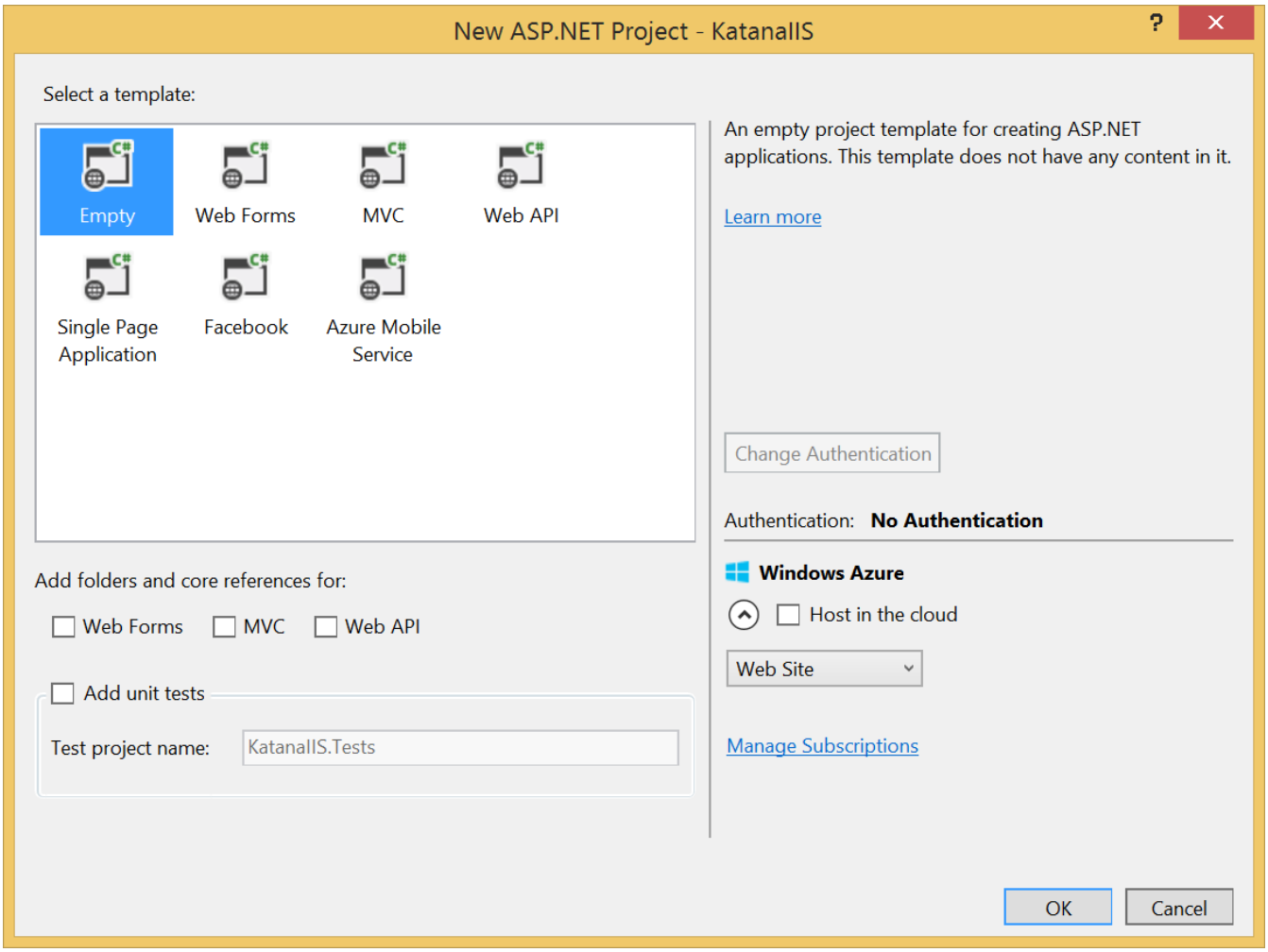

Figure 5: New ASP.NET Project window

The previous figure shows how the **New ASP.NET Project** window should look.

Include the Relevant NuGet Packages

The project has been created; now it is time to include Katana via NuGet.

There are many NuGet packages available to build Katana-based applications, often organized with a very granular approach. While this makes it better for modularity, it makes it harder to know which package to choose for the feature needed by an application. And to make things a bit more complicated, none of the packages have *Katana* in the name, but all are referred to as *Owin.* The Appendix at the end of the book describes the features of each OWIN NuGet package released by Microsoft as part of the Katana suite.

From within Visual Studio, open the **NuGet Package Manager**, and search using the keyword *owin*. The first package in the list will be **`Microsoft.Owin.Host.SystemWeb`**, which is the package you have to install.

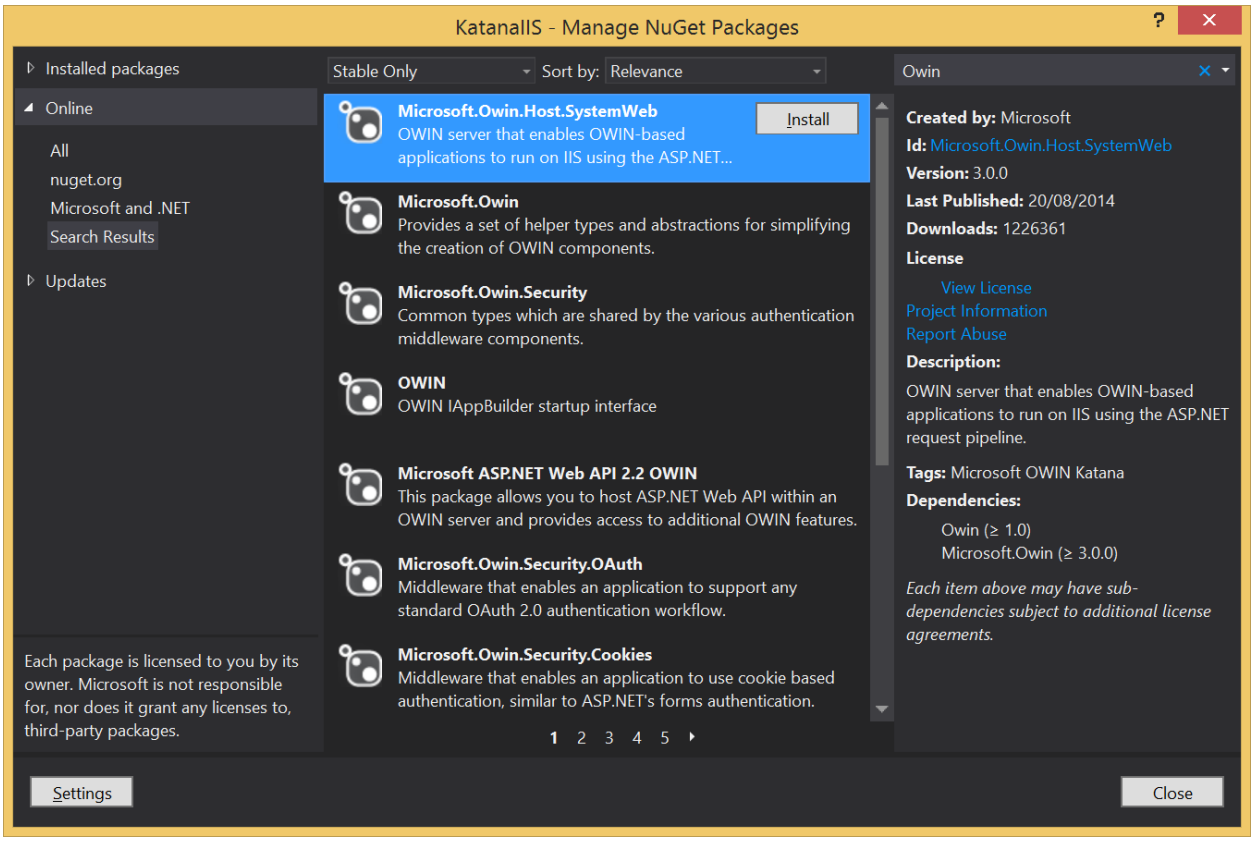

Figure 6: Manage NuGet Packages window with package to install

If you prefer using the package manager console, you can install the needed package by typing:

```
PM> Install-Package Microsoft.Owin.Host.SystemWeb
```

The System.Web-based hosting package depends on two other NuGet packages, **`Owin`** and **`Microsoft.Owin`**. After having installed the package, the **Manage NuGet Packages** window will look like the following figure.

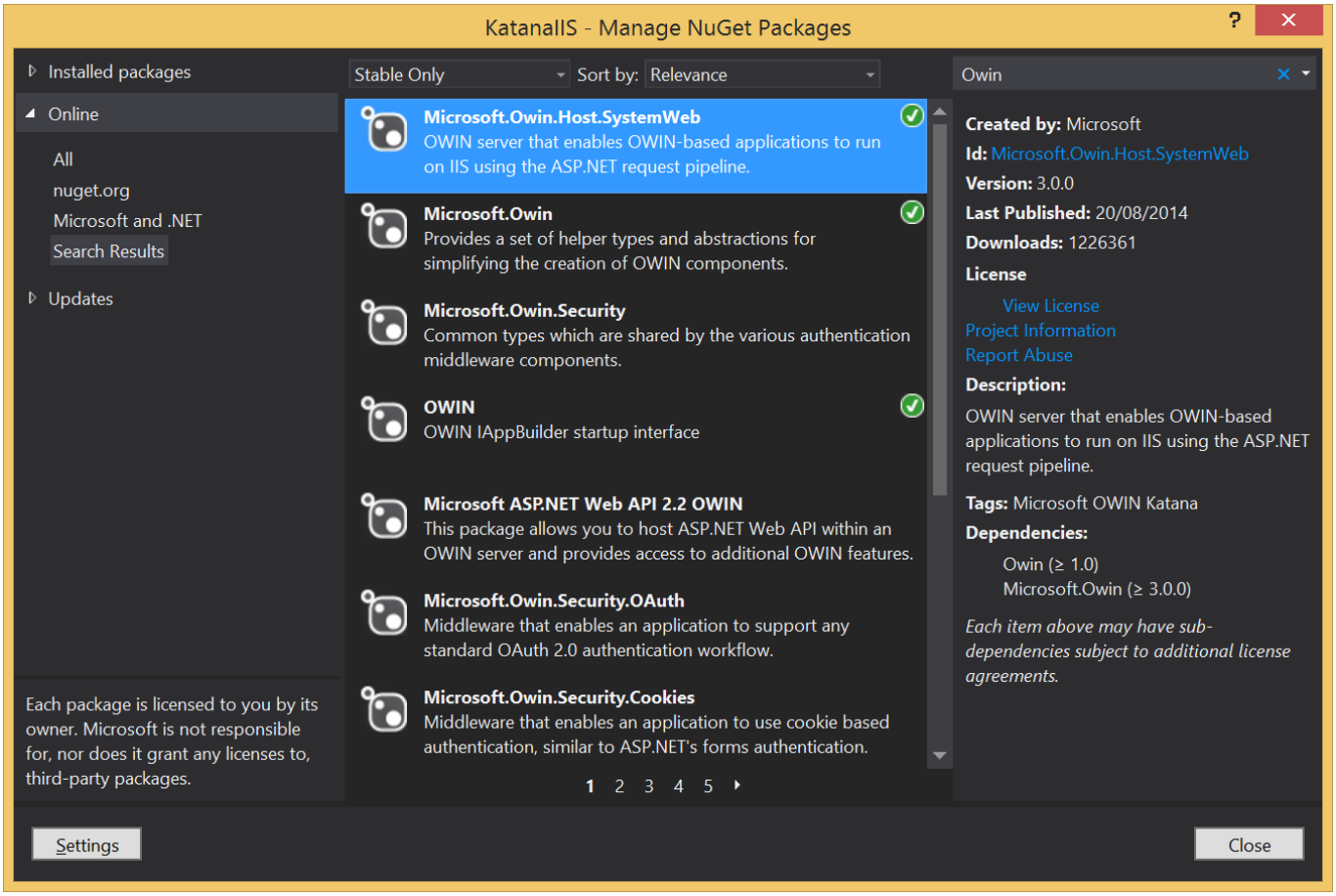

Figure 7: Manage NuGet Packages window with installed packages

Adding the Startup Class

It is finally time to write some code and instruct the application how to respond to web requests. This is done in the **Startup** class, which by convention is nothing but a class named **`Startup`** in the root of the assembly. The code to configure the application is specified by convention in the **`Configuration`** method.

Now open the **Add New Item** window, drill down inside the Web > General tree (or type **`owin`** in the search box in the top right corner of the window), and select the **OWIN Startup class** template, as shown in the following figure.

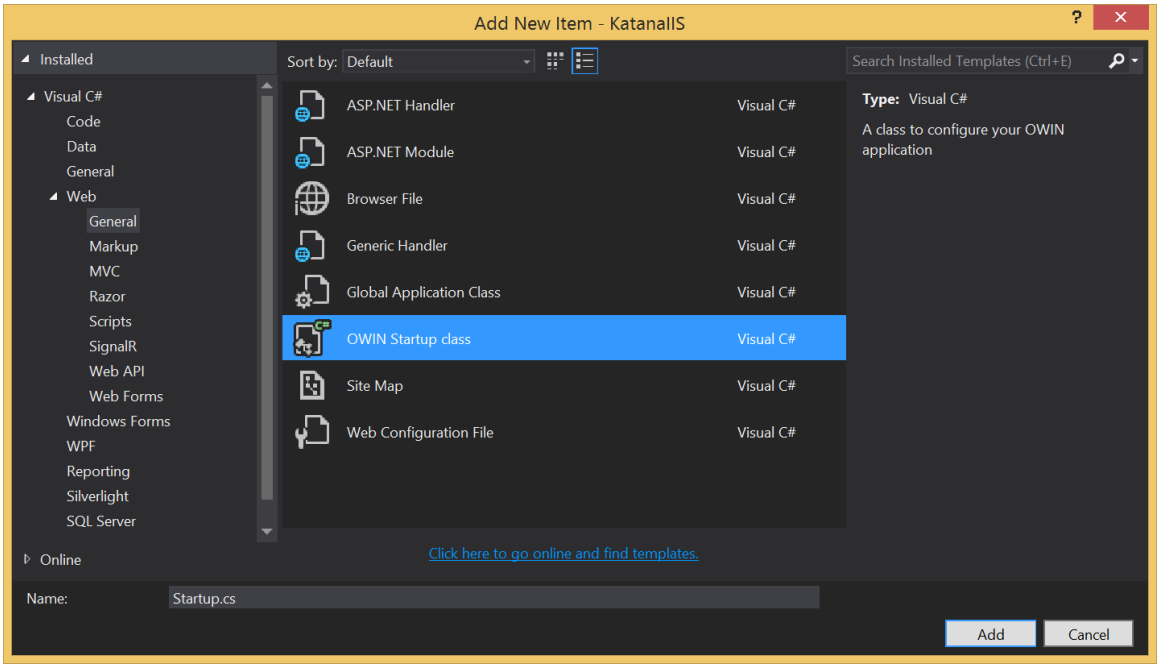

Figure 8: Add New Item window with OWIN Startup class

This will create a **`Startup.cs`** file in the root of the application folder with the following code.

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(Syncfusion.OwinSuccinctly.KatanaIIS.Startup))]

namespace Syncfusion.OwinSuccinctly.KatanaIIS
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // For more information on how to configure your application, visit
http://go.microsoft.com/fwlink/?LinkID=316888
        }
    }
}
```

Code Listing 2

As you can see, besides creating a class following the conventions, the template also added an assembly-level attribute to specify the startup class.

```
[assembly: OwinStartup(typeof(Syncfusion.OwinSuccinctly.KatanaIIS.Startup))]
```

Code Listing 3

The attribute is needed in case the name of the class is different from **Startup**. It will come in handy later in the chapter where you will learn other ways to specify the startup class which also use the **web.config** file.

To add your own logic, you have to implement the **Configuration** method by adding operations in the OWIN pipeline as demonstrated in the following code.

```
public void Configuration(IAppBuilder app)
{
    app.Run(context =>
    {
        context.Response.ContentType = "text/plain";
        return context.Response.WriteAsync("Hello World!");
    });
}
```

Code Listing 4

The previous method inserts as the last element of the OWIN pipeline a lambda function that receives an **IOwinContext** object as input, sets the response type to **text/plain**, and asynchronously writes to the response stream the text "Hello World!".

In this short sample, you already see two of the helpers that Katana adds on top of OWIN specs, as well as the de-facto standard **IAppBuilder** interface:

- The **IOwinContext** class, which wraps the OWIN environment dictionary with a strongly typed and easier to use class.
- The **Run** extension method, used to add a component as a final step of the OWIN pipeline. Its signature is **Run(System.Func<IOwinContext,Task> handler)**.

Using the bare OWIN specs and only **Owin.dll**, the same class would have been a little more complicated to write.

```
using AppFunc = Func<IDictionary<string, object>, Task>;
...
app.Use(new Func<AppFunc, AppFunc>(next => (env =>
{
    string text = "Hello World!";
    var response = env["owin.ResponseBody"] as Stream;
    var headers = env["owin.ResponseHeaders"] as IDictionary<string, string[]>;
    headers["Content-Type"] = new[] { "text/plain" };
    return response.WriteAsync(
        Encoding.UTF8.GetBytes(text), 0, text.Length);
})));
```

Code Listing 5

Notice that without the Katana helpers, you have to call the **`Use`** method to instruct the pipeline to run the OWIN middleware object with that long signature, and ignore the **`next`** parameter, which contains the following element in the pipeline. Also, you have to get the response keys out of the environment dictionary and cast them to the right type before using.

Note: You are never going to use this syntax, but I put it in the chapter to show how things would be using only the OWIN specs. It is more code, but not that much more.

Going a Bit Further

Inside the handler function you can also write more logic, for example, like reading a query parameter to write a personalized **`Hello Readers!`** message.

```
public void Configuration(IAppBuilder app)
{
    app.Run(context =>
    {
        var name = context.Request.Query["name"];
        context.Response.ContentType = "text/plain";
        return context.Response.WriteAsync("Hello " + name + "!");
    });
}
```

Code Listing 6

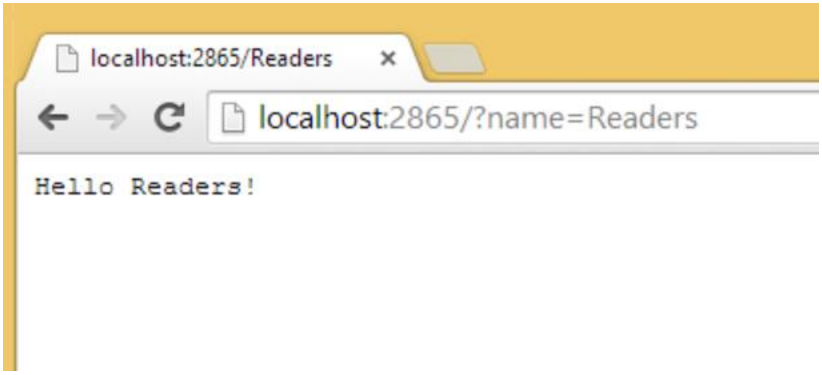

Figure 9: Hello Readers! via query string

If you are not a big fan of query string parameters and prefer a more RESTful approach with clean URLs, you could parse the requested path and extract the segments.

```
public void Configuration(IAppBuilder app)
{
    app.Run(context =>
```

```
    {
        var name = "World";
        var uriSegments = context.Request.Uri.Segments;
        if (uriSegments.Length > 1)
            name = uriSegments[1];
        context.Response.ContentType = "text/plain";
        return context.Response.WriteAsync("Hello " + name + "!");
    });
}
```

Code Listing 7

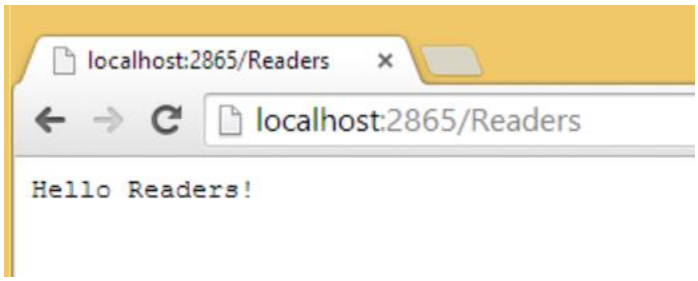

Figure 10: Hello Readers via path segment

Now the code is becoming too complicated to fit just into a lambda expression. It is time to move it to an external method. This way you will also better understand what the handler is: a function that takes in an **`IOwinContext`** and returns a **`Task`** (in this case using the **`async`**/**`await`** paradigm).

```
public void Configuration(IAppBuilder app)
{
    app.Run(WriteResponse);
}

private Task WriteResponse(IOwinContext context)
{
    var name = "World";
    var uriSegments = context.Request.Uri.Segments;
    if (uriSegments.Length > 1)
        name = uriSegments[1];
    context.Response.ContentType = "text/plain";
    return context.Response.WriteAsync("Hello " + name + "!");
}
```

Code Listing 8

With this working simple web app, we will now take it out of IIS and host it inside a custom host.

Katana via a Custom Host

Hosting the Katana application inside a custom host doesn't differ much from hosting it inside IIS. The startup class and the code of the application remain exactly the same as before. The only things that change are the project type used to create the application—a console application—and the NuGet packages to install.

Note: Hosting your application inside a custom host like a console application, a Windows service, or even a WPF desktop application, is an easy way to add a web interface for your applications. Be careful when choosing this option instead of choosing IIS. You'll lose all the features that come with IIS (like SSL, event logging, easy management console, etc.) and will have to implement them yourself.

Create the project

As before, start by creating a new project, this time by selecting **Console Application** as shown in Figure 11.

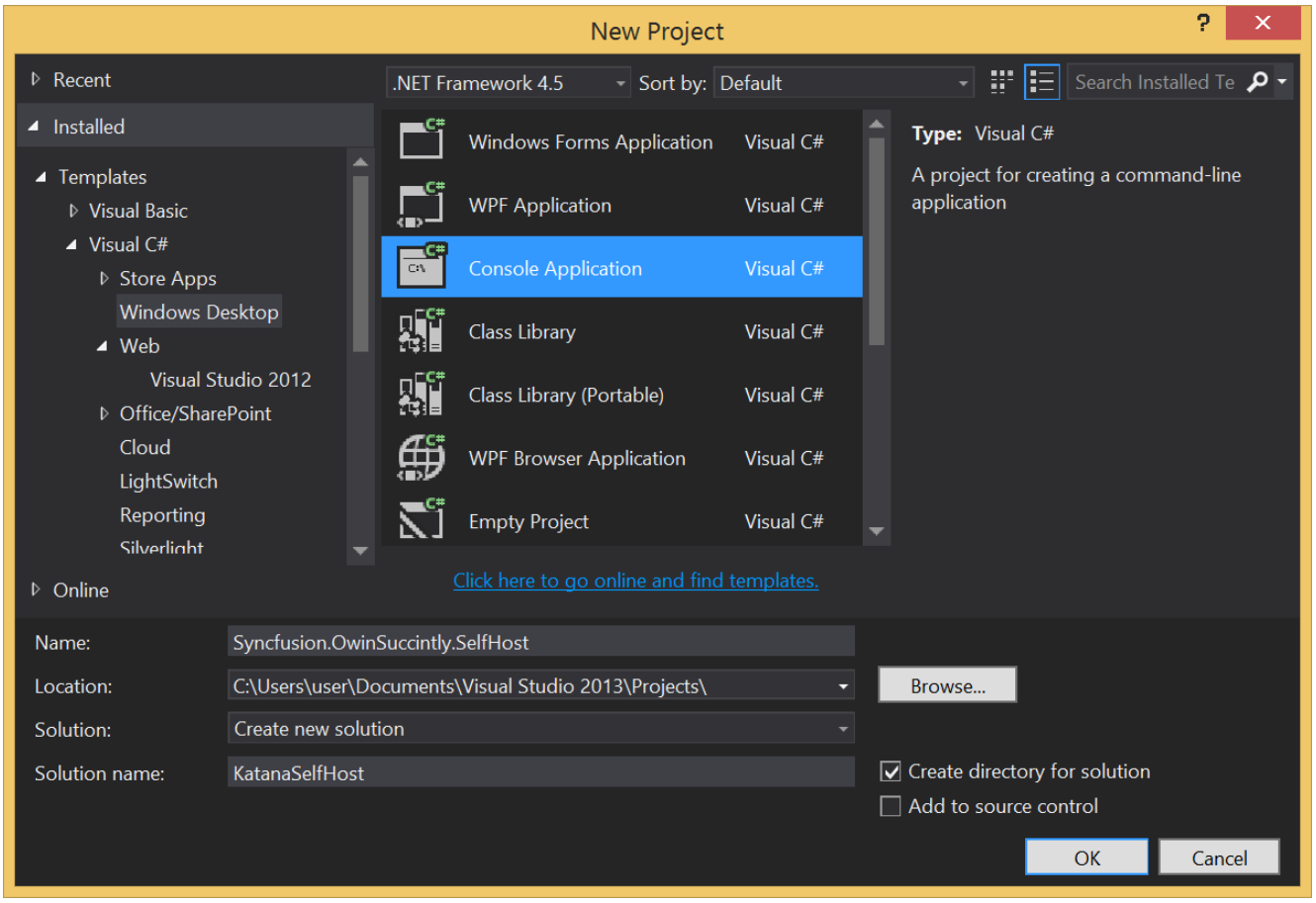

Figure 11: New Project window

Include the Relevant NuGet Packages

With the empty console application created, open the NuGet Package Manager, and search using the keyword owin **selfhost** and choose **`Microsoft.Owin.SelfHost`**.

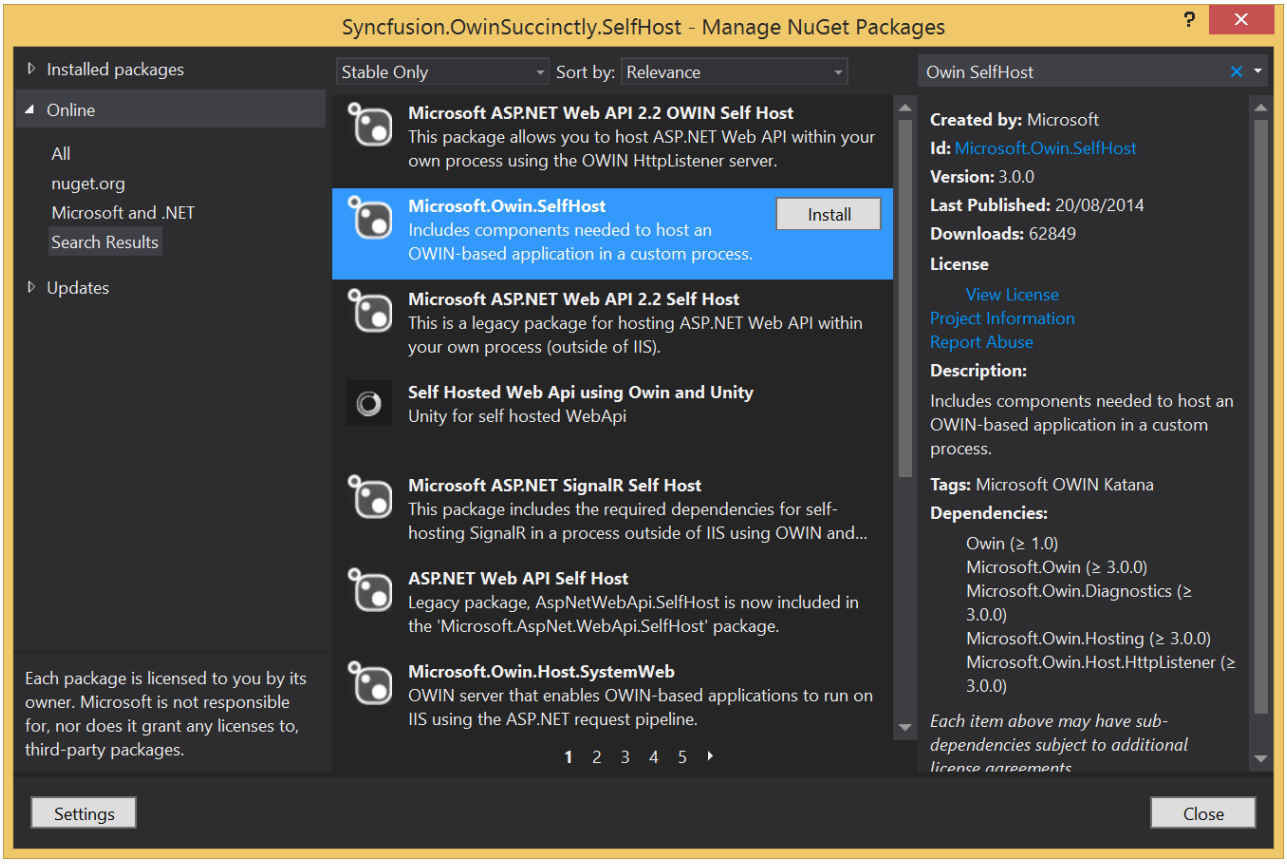

Figure 12: Manage NuGet Packages window showing package to install

Or you can also use the following command in the Package Manager console.

```
PM> Install-Package Microsoft.Owin.SelfHost
```

When the download and installation are finished, you will see seven NuGet packages installed.

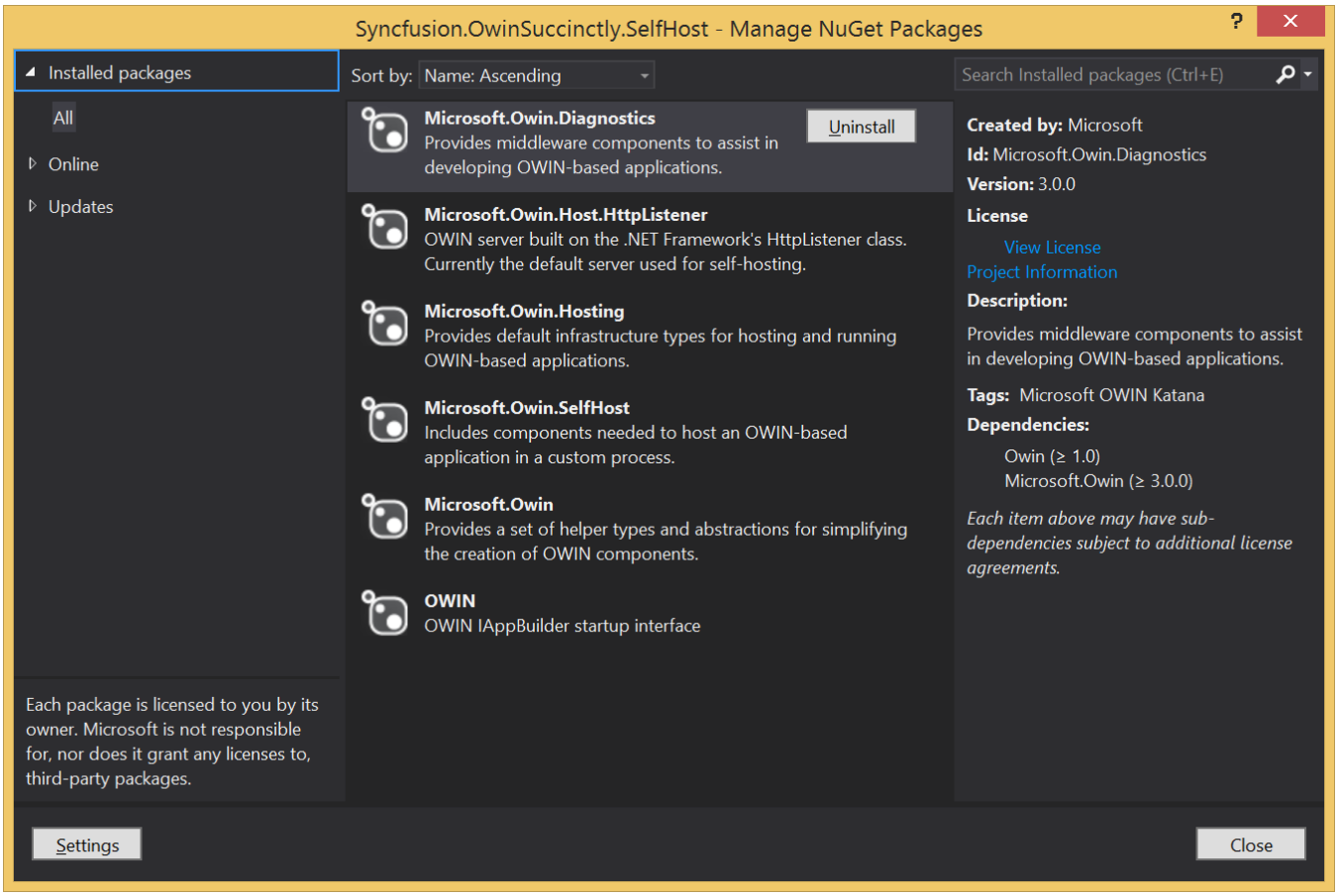

Figure 13: Manage NuGet Packages window with installed packages

In addition to **`Owin`** and **`Microsoft.Owin`**, it will also download the **`Microsoft.Owin.Hosting`** package, which contains the infrastructure for hosting OWIN-based applications, **`Microsoft.Owin.Host.HttpListener`**. It will include the implementation of the OWIN host layer based on the .NET networking class **`HttpListener`** and **`Microsoft.Owin.Diagnostic`**, which contains some useful components. You need these components now that you are not relying on IIS anymore, including the default welcome page and the equivalent of the Yellow Screen of Death (YSOD).

Write the Hosting Code

Everything is in place now, so the next step is writing the code to launch the server and host the OWIN-based application.

```
static void Main(string[] args)
{
    using (WebApp.Start<Startup>("http://localhost:9000"))
    {
        Console.WriteLine("Launched site on http://localhost:9000");
        Console.WriteLine("Press [enter] to quit...");
```

```
        Console.ReadLine();
    }
}
```

Code Listing 9

Basically, the only relevant line of code is the **`WebApp.Start<Startup>("http://localhost:9000")`** you specify with the generics notation, which is the class that contains the startup configuration, and you specify which domain and port to listen to as its parameter.

Starting and Running the Server

Now we can add the startup class by writing the same code as before, and the application will run perfectly fine. Just press F5 and the console application will launch and show that the server has been started.

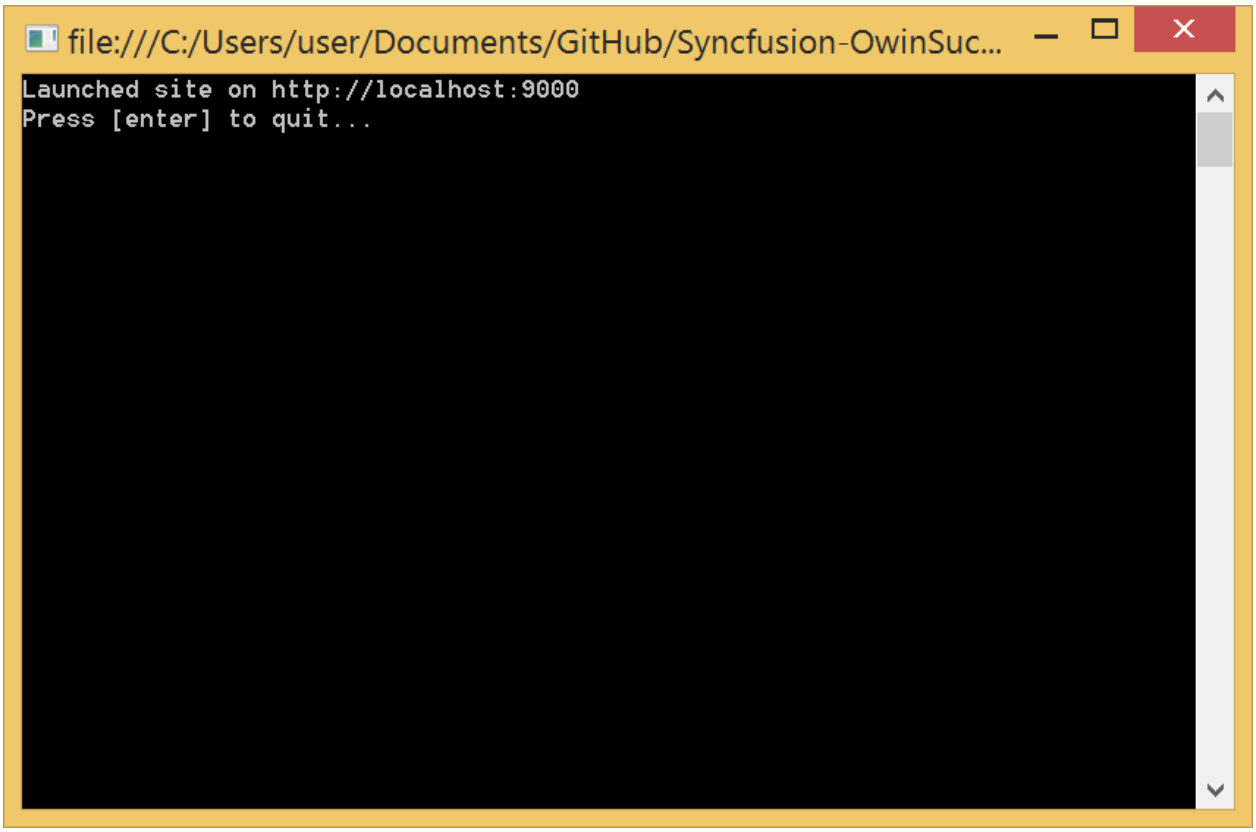

Figure 14: Console application with custom host

Using the Diagnostic Helpers

By running our application in a custom host, we are losing all the management and diagnostic features of IIS. By using the features included in the **`Microsoft.Owin.Diagnostic`** package, we can at least bring back the welcome screen and a debugging view to show in case of unhandled exceptions.

```
public void Configuration(IAppBuilder app)
{
    app.UseWelcomePage("/");
    app.UseErrorPage();
    app.Run(context =>
    {
        Trace.WriteLine(context.Request.Uri);
        //Line to show the ErrorPage
        if (context.Request.Path.ToString().Equals("/dotnotcallme"))
        {
            throw new Exception("You requested the wrong URL :)");
        }

        context.Response.ContentType = "text/plain";
        return context.Response.WriteAsync("Hello, world.");
    });
}
```

Code Listing 10

The previous code listing shows the basic **`Configuration`** class with a few additional lines.

The first is **`app.UseWelcomePage("/")`**, which adds into the pipeline an OWIN middleware component that shows the default welcome page. It is added when the URL specified as parameter is requested, which is the root of the site in this case.

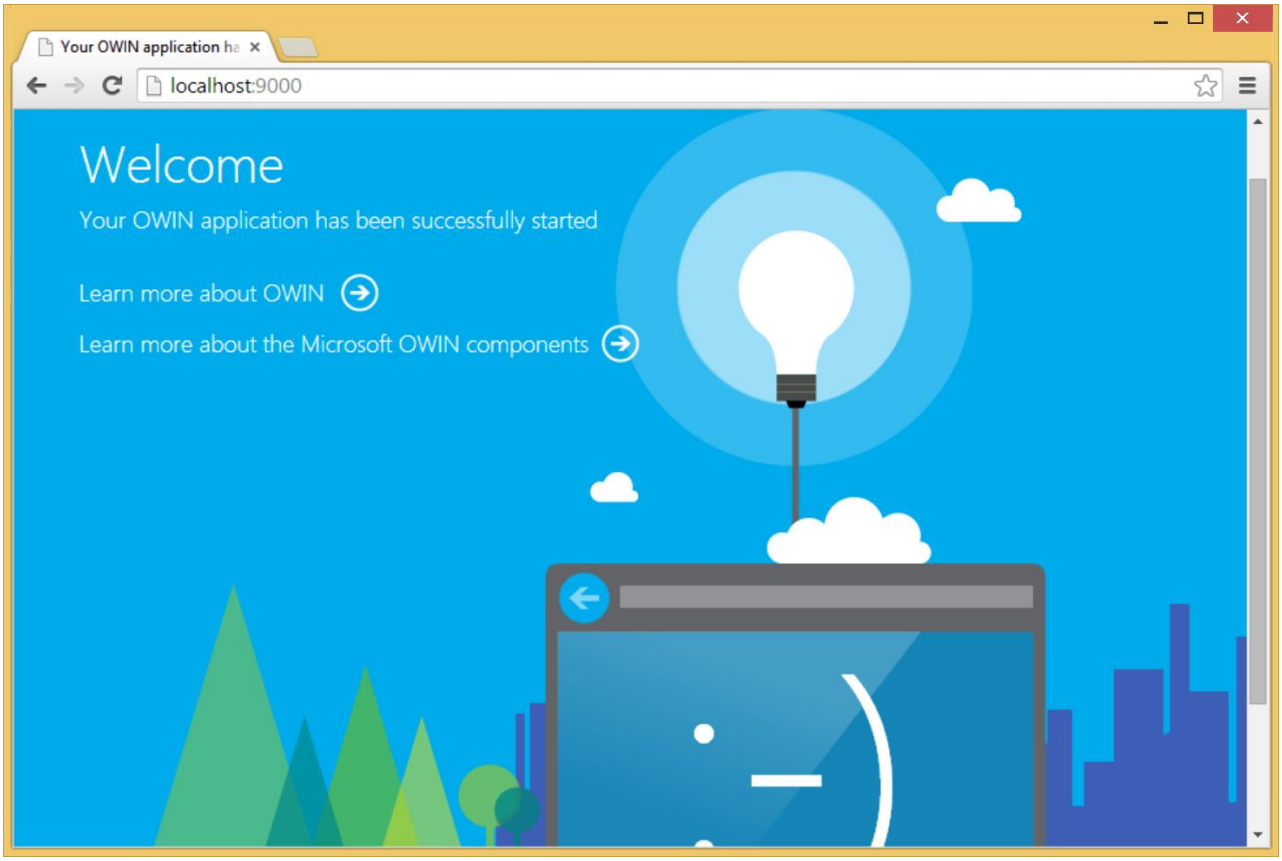

Figure 15: Welcome page

The second line **`app.UseErrorPage()`** configures the pipeline to show an error page if an unhandled exception happens during the execution of the request.

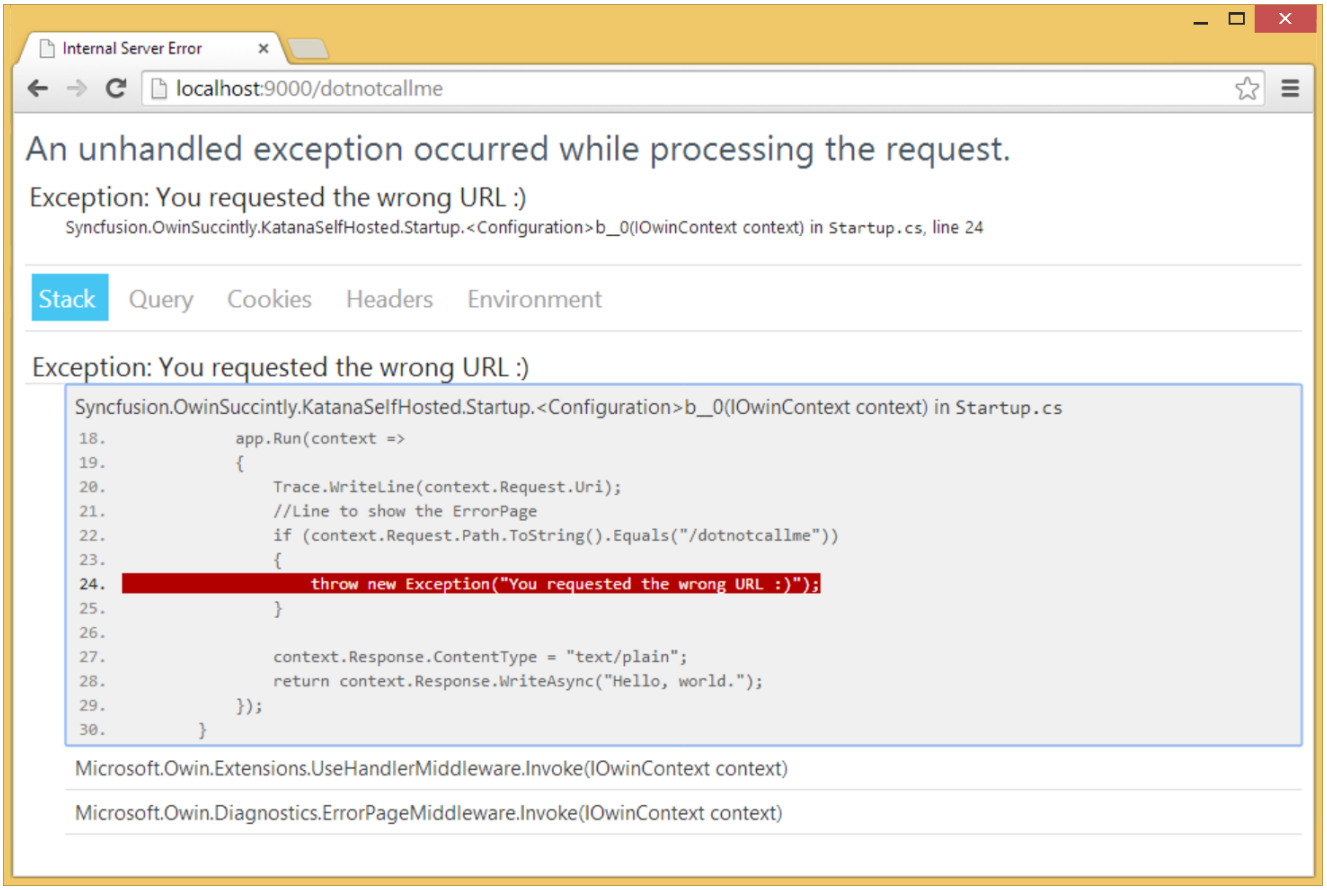

Figure 16: Error page, stack trace tab

Figure 16 shows the default view of the error page, with the stack trace tab that is equivalent to the YSOD. It shows the stack trace and source code with the line of code that generated the exception highlighted.

As you can see, there are many others panes: Query, Cookies, Headers, and, most interesting for an OWIN application, Environment, with a list of all the entries in the OWIN environment dictionary.

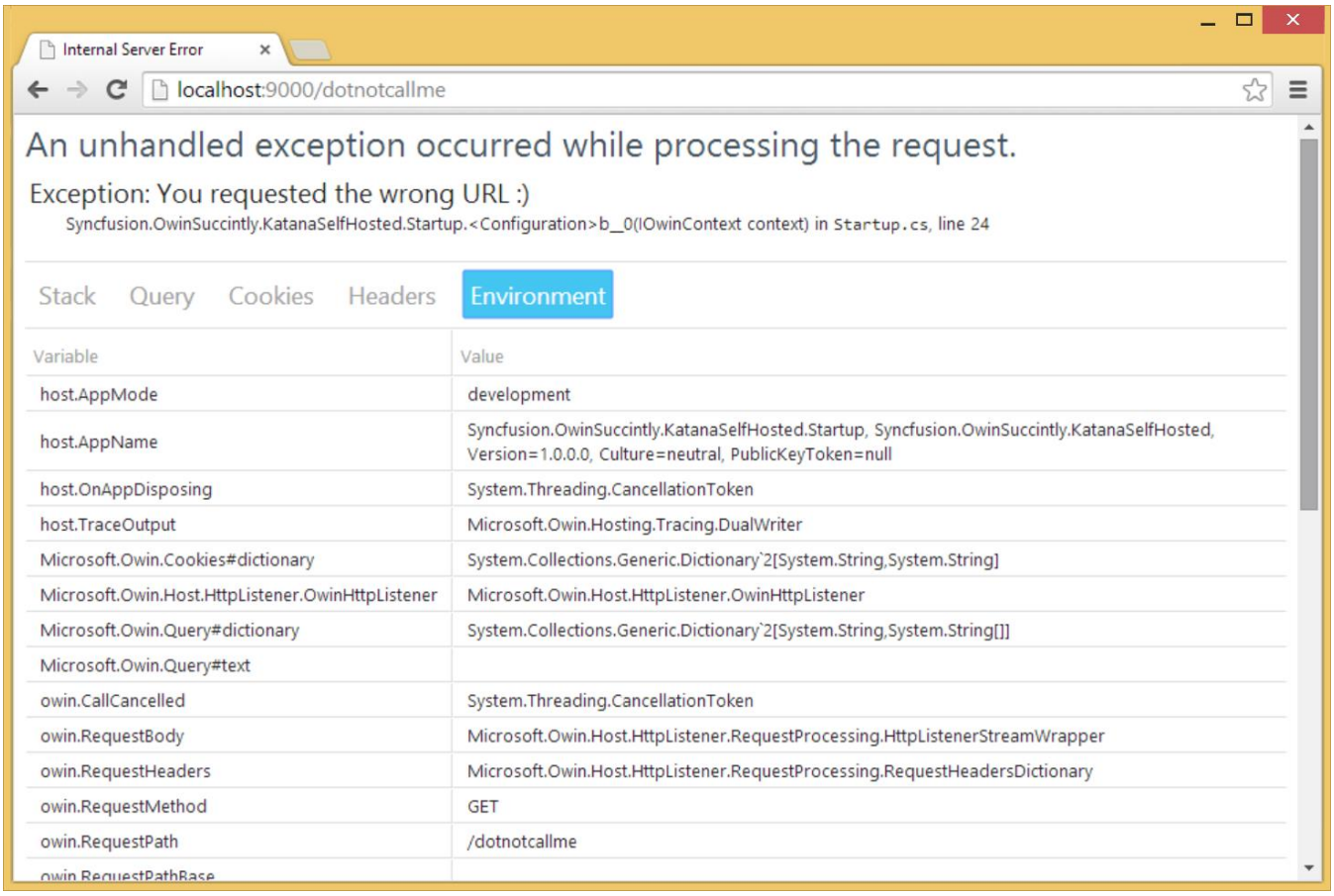

Figure 17: Error page, environment dictionary tab

Note: If you want to see the error page in action, the code listing above is made to throw a fake exception when the URL `/donotcallme` is requested.

Using Tracing

Another diagnostic possibility is to use the **`Trace`** class from **`System.Diagnostic`** and write directly to the trace output stream, which in the case of the console application is the console window. For example, you can add the following line of code inside the request handler to print on the console window all the URLs requested.

```
Trace.WriteLine(context.Request.Uri);
```

Code Listing 11

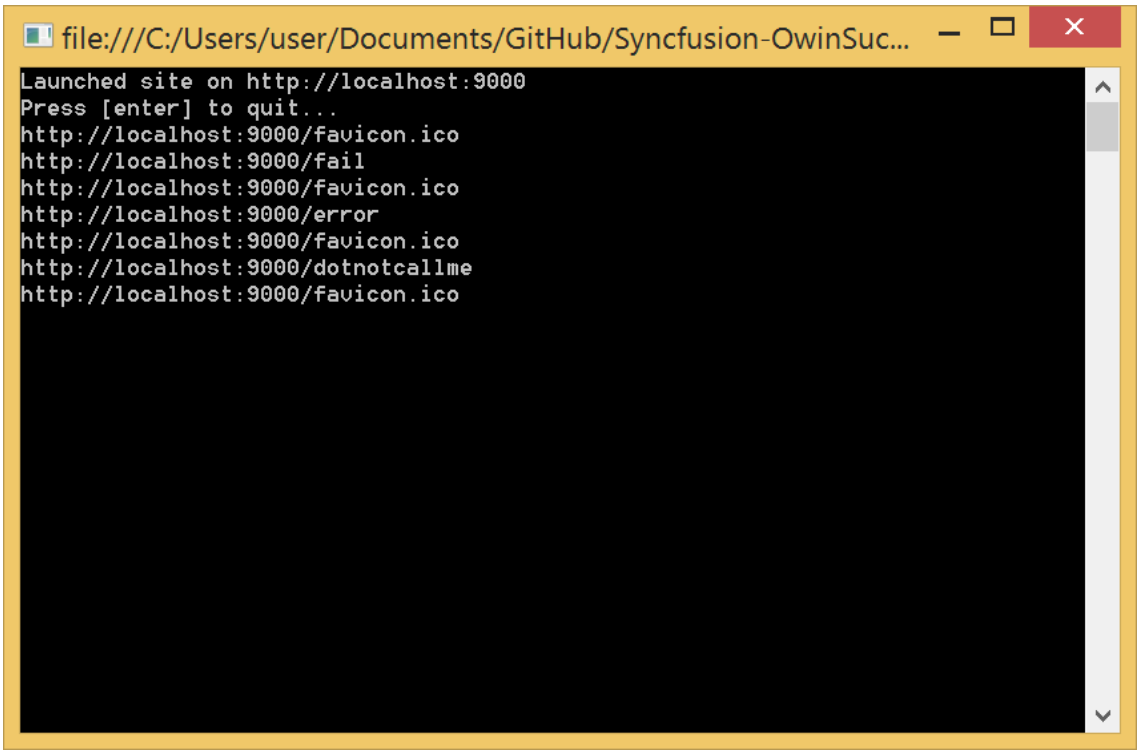

Figure 18: Tracing in the console window

Note that the **`http://localhost:9000/`** is never printed in the console window. This is because the root URL is intercepted by the **`WelcomePage`** OWIN component that renders the welcome page and then closes the response without forwarding the execution to the rest of the OWIN pipeline. If you wanted to also trace the welcome page, you should have registered a custom middleware before registering the **`WelcomePage`** component.

Katana via OwinHost.exe

The last hosting option is using the OwinHost.exe application, which, unsurprisingly, is also distributed via a NuGet package.

The configuration code for the Katana-based application doesn't change from what you have seen so far. Also, you can create the application either as a normal class library or as a web application.

The difference between the two approaches is mainly the development and debugging experience:

- With a class library you get a very easy setup of the project with the bare minimum references needed, but you need to manually launch the host and attach it manually to the process for debugging. Also, you do not have all the web tooling available with other options.

- With a web application, after you create the project, you have to do a bit of cleaning up of unneeded references, but you will have an integrated debugging experience (using Visual Studio 2013) and the web tools.

Use a Class Library

Let's start by creating a new project. This time select the **Class Library** project.

Write the Application

Once you have the project ready, delete the default Class1.cs file and add a new **OWIN Startup class** like you did when creating the startup class in the previous approaches.

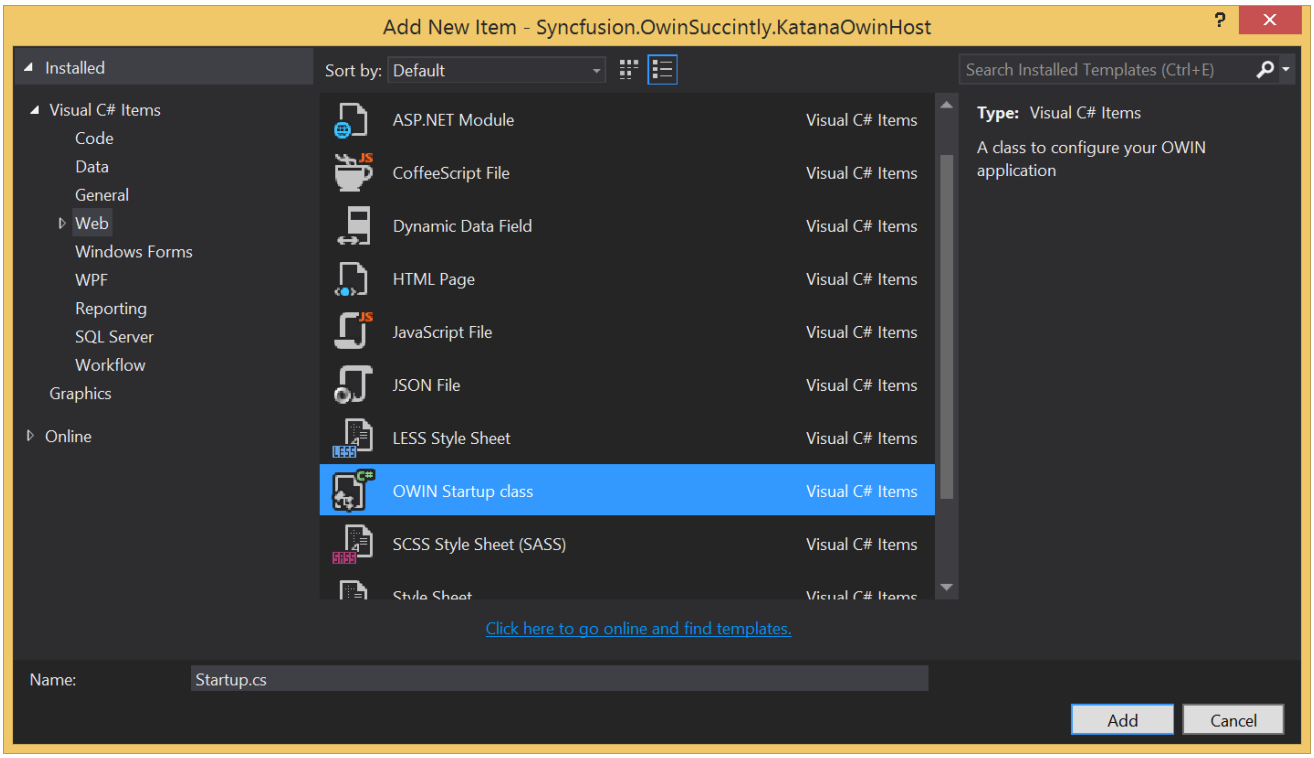

Figure 19: Add OWIN Startup class

This will automatically add the reference to both Owin.dll and Microsoft.Owin.dll (by referencing their NuGet packages).

```
[assembly: OwinStartup(typeof(Syncfusion.OwinSuccinctly.KatanaOwinHost.Startup))]

namespace Syncfusion.OwinSuccinctly.KatanaOwinHost
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
```

```
            app.Run(context => context.Response.WriteAsync("Hello World!"));
        }
    }
}
```

Code Listing 12

After writing the usual **`Configuration`** method in the **`Startup`** class like the one in Code Listing 11, your class library project should look like the one in Figure 20.

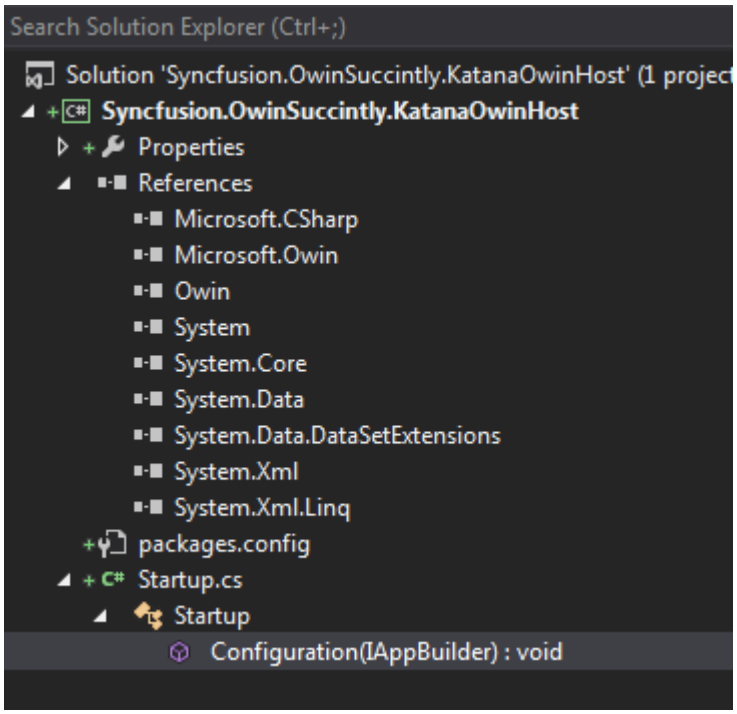

Figure 20: Visual Studio Solution Explorer

Pay attention to the list of references: the two OWIN DLLs were automatically referenced when you added the **Owin Startup Class**, and the others are only the bare minimum needed for a class library to run.

Launch and Debug the Application

Now that you have the class library with the application, you need to launch OwinHost.exe and then possibly debug it.

First open the NuGet Package Manager and download the **`OwinHost`** package.

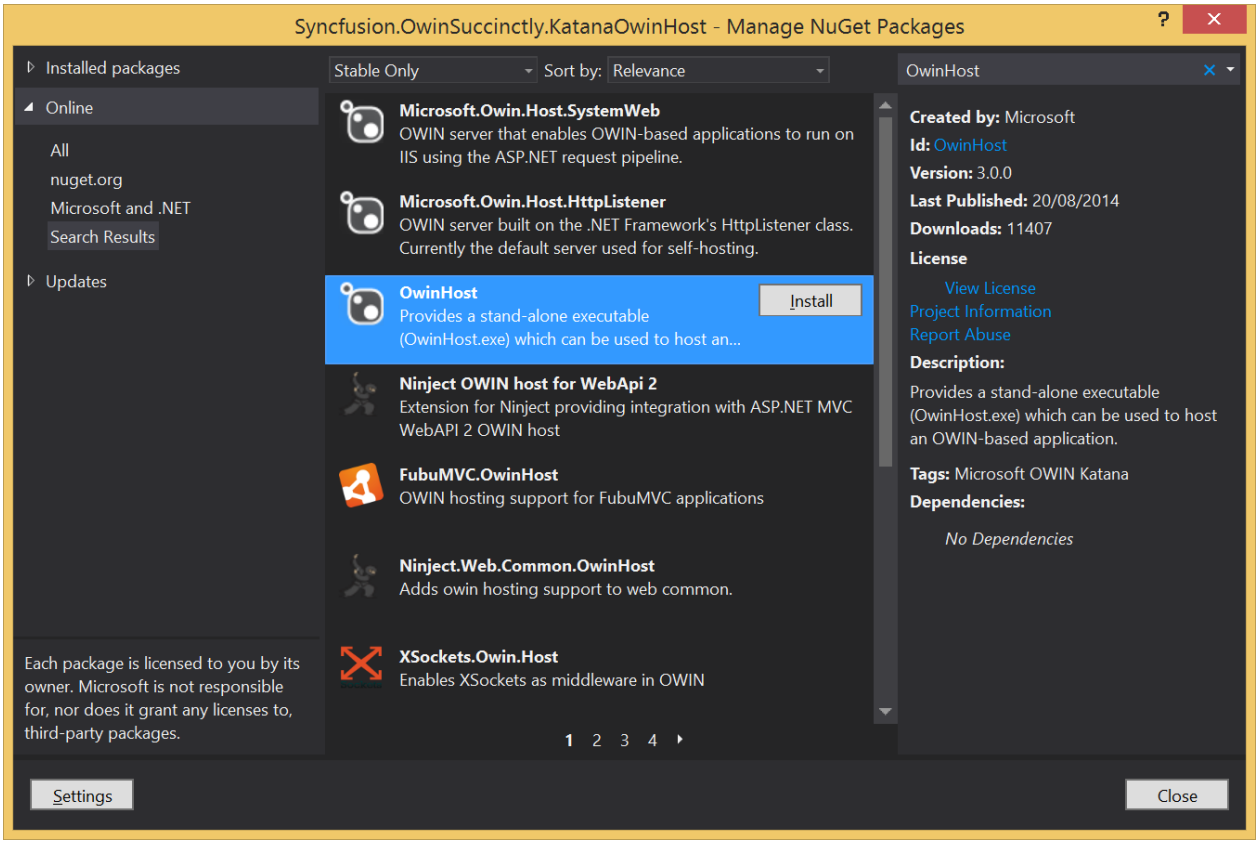

Figure 21: OwinHost NuGet Package

The OwinHost.exe file, like all NuGet packages, will be installed in the <solution root>/packages/OwinHost.(version)/tools folder (at the time of writing, the location is <solution root>/packages/OwinHost.3.0.0/tools).

By convention, when called without any parameters, OwinHost.exe will load all the assemblies in the .\bin folder relative to where the application is called from, and will call the startup class as specified with the **`OwinStartup`** assembly attribute.

Unfortunately, a Class Library project, after it is built, puts the DLLs in a .\bin\Debug folder, so we have to tell Visual Studio to save the files in another folder. To do this, open the **Project Properties** window, select the **Build** tab, and change the **Output path** property to bin\.

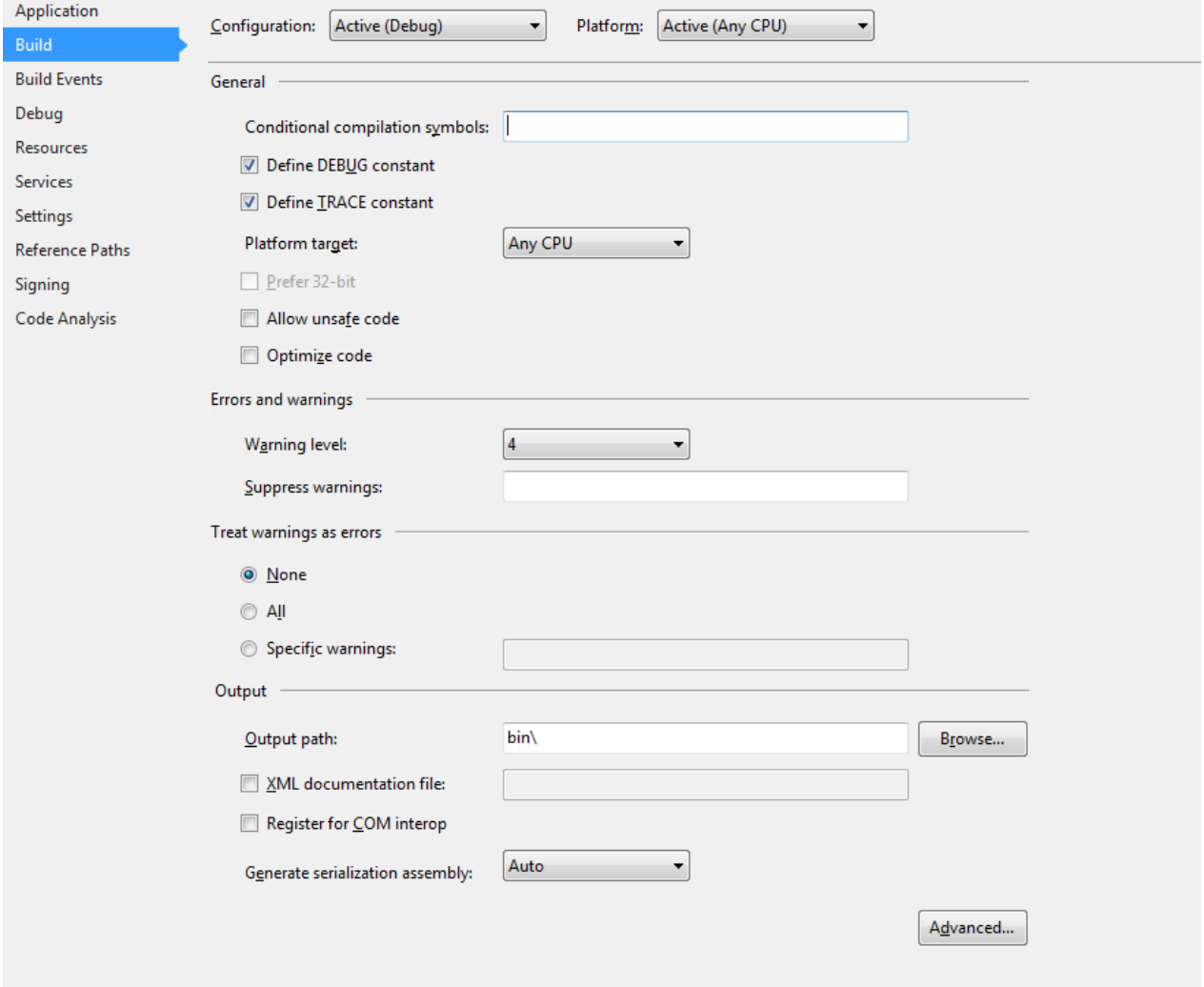

Figure 22: Project Properties window

Finally, open a console window, move to the root of the project, and launch OwinHost.exe without any parameters using the relative path.

```
C:\..\Syncfusion.OwinSuccinctly.KatanaOwinHost>..\packages\OwinHost.3.0.0\tools\OwinHost.exe
```

Code Listing 13

The host will start, listening by default on port 5000 of **`localhost`**.

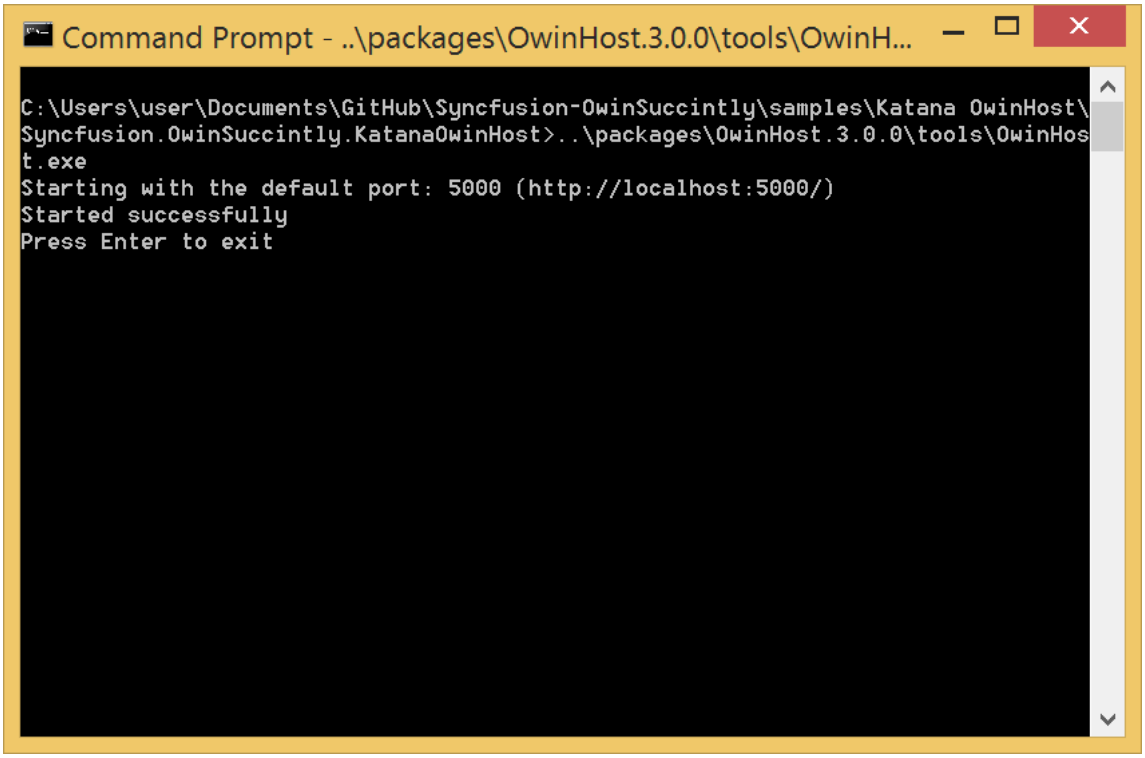

Figure 23: OwinHost.exe console output

You can change the default behavior using command line options and by manually specifying the startup class as an argument. To see the help screen with all the options, just launch the application with the **-h** parameter.

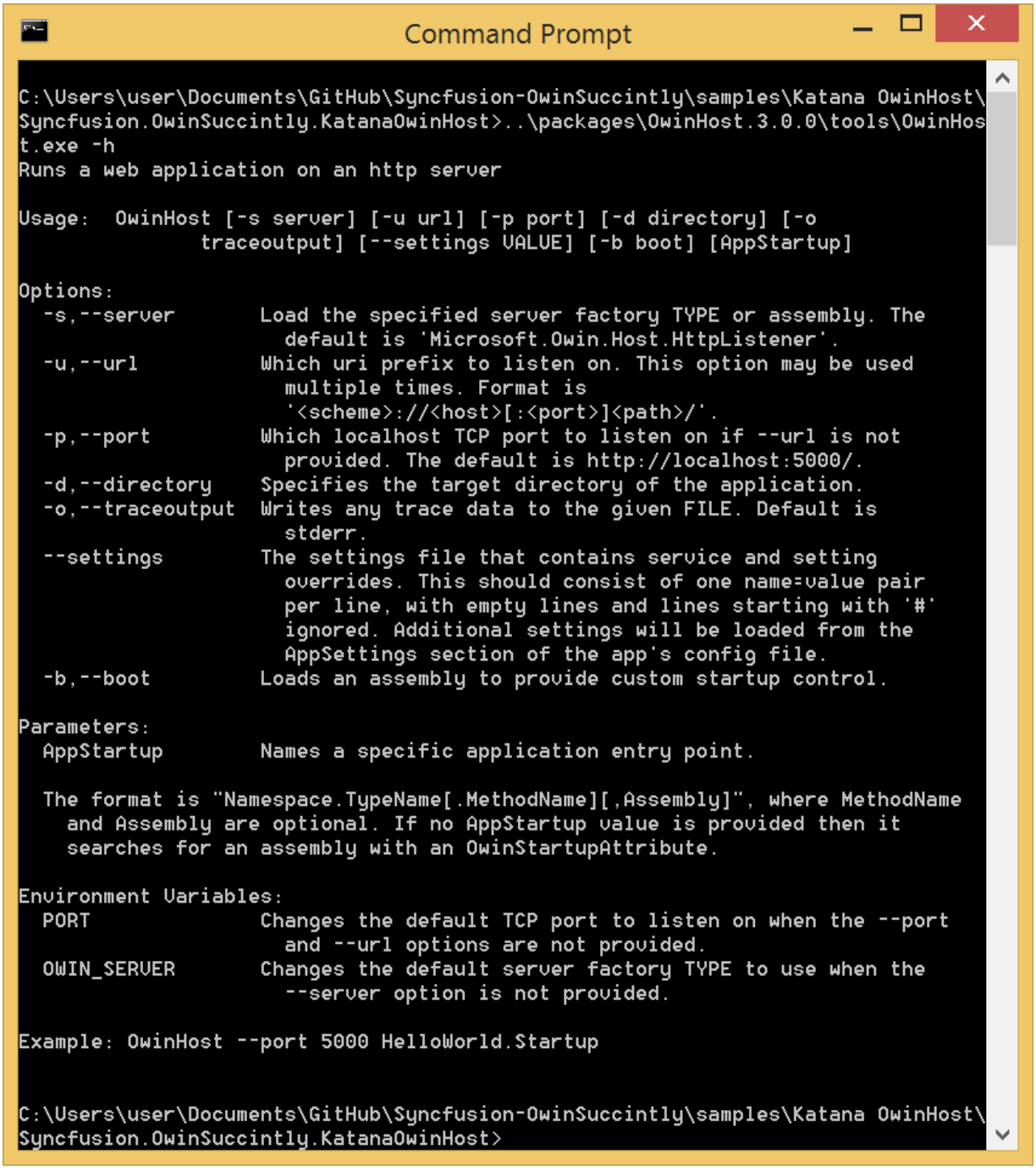

Figure 24: OwinHost.exe help

If you want to debug, you have also to attach the debugger to the OwinHost.exe process.

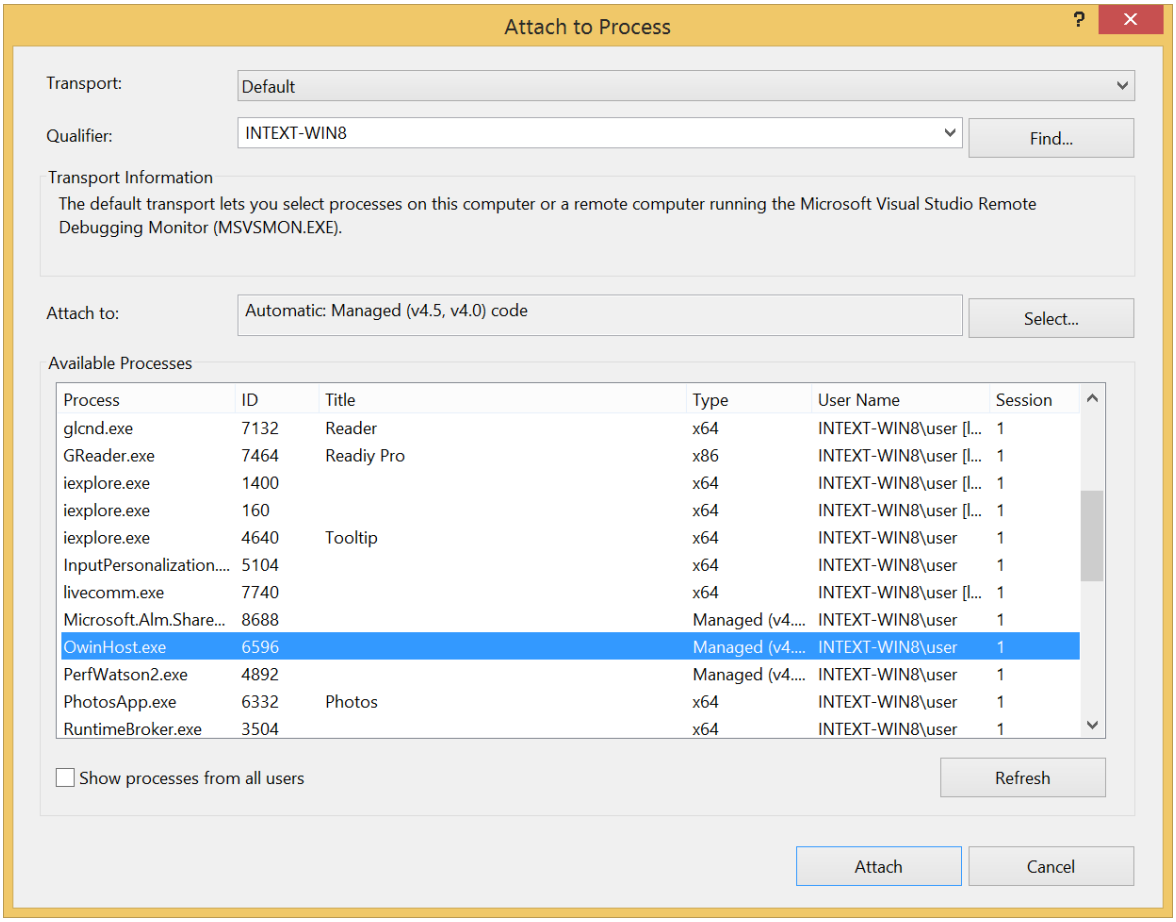

Figure 25: Attach to Process

It's not very convenient, but if you want a better debugging experience like using F5 to debug in Visual Studio, you have to create your project as a web application.

Get the F5 Debugging Back

When you reference the **`OwinHost`** NuGet package inside a Web Application project, its installation procedure leverages a new feature introduced with Visual Studio 2013: external host extensibility. This powerful feature allows tools to easily register an additional custom host together with the default IIS Express and local IIS.

The **`OwinHost`** NuGet package does so, and adds itself to the list of servers Visual Studio can launch to debug the application. The following figure shows how the **Project Properties** window looks after having installed the **`OwinHost`** NuGet package.

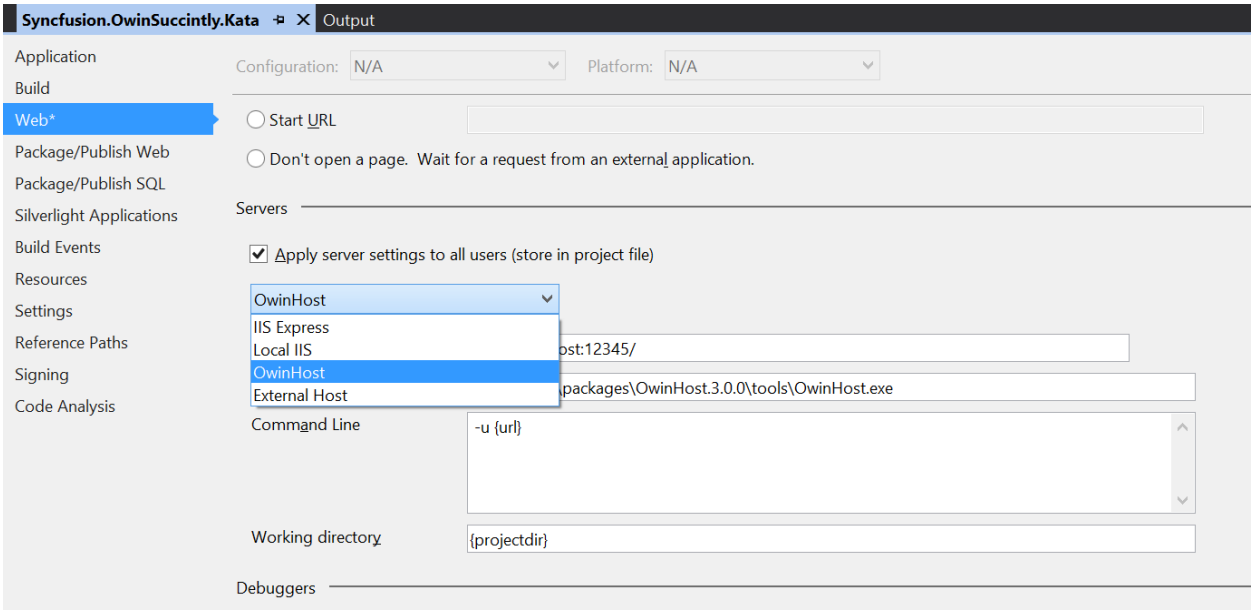

Figure 26: Project Properties window, Web tab

But registering an additional custom host works only within a Web Application project, so instead of building the Katana application as a console application, create a **Web Application** using the **Empty** template. Then, like you did for the console application, add the OWIN **`Startup`** class. Afterwards, delete all the unneeded references (for example, System.Web) until your Solution Explorer tree looks like the one shown for the console application in Figure 20.

Finally, install the **`OwinHost`** NuGet package and configure the project to use **OwinHost** as the server when launching the application from Visual Studio.

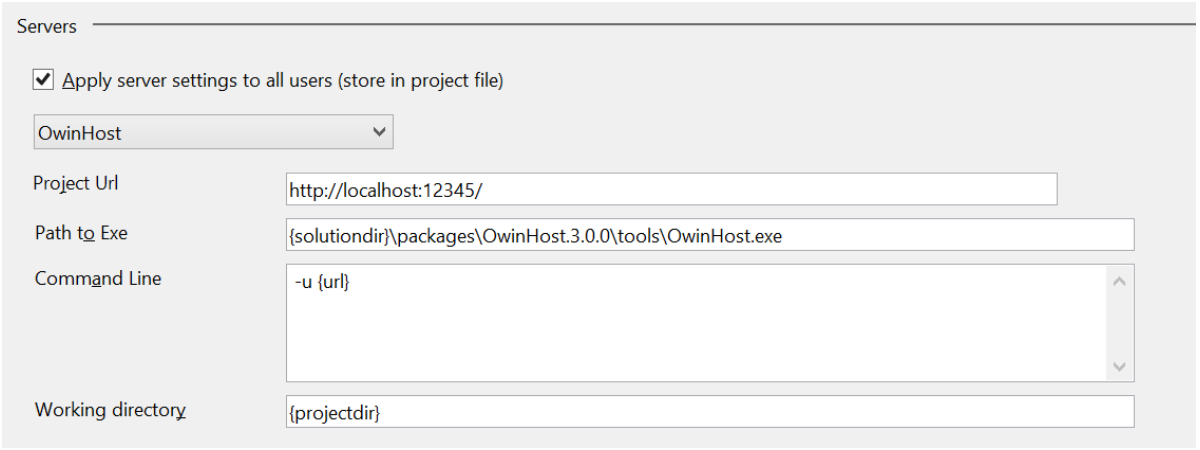

Figure 27: Configuring servers in the Web tab of the Project Properties window

From the form that appears when you choose **OwinHost** as the server, you can customize the URL to listen to and pass custom parameters in the command line.

Now you can press **F5** and directly launch your OWIN application from within Visual Studio.

Note: This works only in Visual Studio 2013. Do not install the additional server option (you will be prompted by the NuGet Manager for this) if you want to open the same project in Visual Studio 2012, as you won't be able to open it anymore in Visual Studio 2013.

Katana on Azure

OWIN and Katana are great for on-premise applications, but are even better on the cloud. They fit very well with all Windows Server installations, virtual machines, or something more powerful like a full-scaling application.

Microsoft Azure is one of the most complete cloud services with different options. Basically, you can deploy your application wherever you want, but the most used services are:

- Azure Website
- Cloud Service (Worker Role and Web Role)

Because everything on Azure is hosted on a Windows installation (except for some Linux virtual machines), there aren't differences between using OWIN and Katana on-premise or on Microsoft Azure. In fact, your websites will run on IIS.

The only thing to do if you are going to use Microsoft Azure is to follow the instructions explained at the beginning of the chapter about running Katana with IIS, and then deploy it to Azure.

Other Ways to Specify the Startup Class

By now, you understand that an OWIN application needs a startup class that contains the bootstrapping code, and that this class is normally found by convention based on its name, **`Startup`**, or by specifying the class name with the **`OwinStartupAttribute`**. But these are not the only ways to specify the startup class. Let's review the options available:

- **Naming convention**: By default, the host looks for a class called **`Startup`** in the root of the application namespace.
- **`OwinStartup`** Attribute: An assembly-level attribute of type **`OwinStartupAttribute`** specifies which class contains the startup class.

```
[assembly: OwinStartup("Default",
    typeof(Syncfusion.OwinSuccinctly.KatanaStartup.StartupDefault))]
```

Code Listing 14

With this attribute, you can also specify a *friendly* mnemonic name for the class, which can be used later to refer to this particular startup class, and even the name of the method to call if you don't want to use the **`Configuration`** method.

- **appSettings in a configuration file**: By using an entry in the **appSettings** section of the web.config or app.config files, you can override both naming conventions and **OwinStartup** attributes.

```
<appSettings>
  <add key="owin:appStartup"
value="Syncfusion.OwinSuccinctly.KatanaStartup.StartupDefault" />
</appSettings>
```

Code Listing 15

You can also use the fully qualified name of the type if the startup class is in another assembly, or you can specify the friendly name. We will come back to the friendly name later.

- **Startup option**: If you are hosting your application in a custom host, you can specify which class to use when you spin up the host by specifying the class as a generic parameter for the **Start** method.

```
WebApp.Start<StartupDefault>("http://localhost:9000");
```

Code Listing 16

- **Command line argument**: If you are using **OwinHost**, you can also specify the startup class as an argument of the command line tool.

```
..\packages\OwinHost.3.0.0\tools\OwinHost.exe
Syncfusion.OwinSuccinctly.KatanaStartup.StartupDefault
```

Code Listing 17

Using the Friendly Name

Let's look a bit more in detail at using the friendly name to refer to configurations. For this to work, you have to combine two of the five options explained previously:

- The **OwinStartup** attribute.
- Either **appSettings** or **OwinHost.exe** argument.

You need to have at least two startup classes, both with the **OwinStartupAttribute** specified (okay, here we will have three).

```
[assembly:
OwinStartup("Development",typeof(Syncfusion.OwinSuccinctly.KatanaStartup.StartupDe
velopment))]

namespace Syncfusion.OwinSuccinctly.KatanaStartup
```

```
{
    public class StartupDevelopment
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                return context.Response.WriteAsync("Hello from Development!");
            });
        }
    }
}

[assembly:
OwinStartup("Test",typeof(Syncfusion.OwinSuccinctly.KatanaStartup.StartupTest))]

namespace Syncfusion.OwinSuccinctly.KatanaStartup
{
    public class StartupTest
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                return context.Response.WriteAsync("Hello from Test!");
            });
        }
    }
}

[assembly:
OwinStartup("Production",typeof(Syncfusion.OwinSuccinctly.KatanaStartup.StartupPro
duction))]

namespace Syncfusion.OwinSuccinctly.KatanaStartup
{
    public class StartupProduction
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                return context.Response.WriteAsync("Hello from Production!");
            });
        }
    }
}
```

Code Listing 18

Since each of the three classes has a different friendly name associated with it, we can now decide via a setting in the configuration file which of them to use as a startup class. The following configuration will launch the application using the **`Test`** configuration.

```
<appSettings>
  <add key="owin:appStartup" value="Test" />
</appSettings>
```

Code Listing 19

Or using OwinHost.exe, the same result will be achieved by typing:

```
..\packages\OwinHost.3.0.0\tools\OwinHost.exe Test
```

Code Listing 20

In these examples, the difference between the configurations is nonexistent, but in a real life scenario you might want to add different middleware components based on the configuration. For example, you could enable the diagnostic page only in development and testing, but not in production, or enable caching, JavaScript bundling, and minification only in production.

Conclusions

In this chapter, you have learned what Katana is and how its architecture maps on top of the OWIN specs.

Later in the chapter, you also saw how to create Katana applications and host them inside IIS, inside a custom host, and finally inside the provided OwinHost.exe. You also learned about a little-known feature of Visual Studio 2013, and how to deploy a Katana-based application on Azure.

Finally, you've also seen the many ways in which you can change the startup class of an OWIN application.

In the next chapter, you will learn how to use Katana to run applications built with the web frameworks you already know, like ASP.NET Web API, MVC frameworks, and SignalR.

Chapter 3 Using Katana with Other Web Frameworks

Introduction

In the previous chapter, you learned how to write applications with Katana using all the hosting options available: IIS/ASP.NET, self-host, and OwinHost.exe. In particular, you learned that writing the startup configuration is a very easy process that only involves creating a class with a `Configuration` method that contains all the setup instructions.

Leveraging the knowledge you just acquired, in this chapter you will learn how to use the following web frameworks you are probably already familiar with, but on top of Katana:

- ASP.NET Web API
- NancyFx
- SignalR

Unfortunately, existing web frameworks have to be adapted to work with OWIN middleware components; for those that depend on System.Web, the porting is not easy. One of those is ASP.NET MVC, which is not compatible with OWIN, but ASP.NET MVC version 6 will be.

Since you know already how to include the OWIN and Katana NuGet packages and how to create a startup class, I will skip over those steps and focus mainly on the usage of the web frameworks.

Using Any Web Framework with Katana

Before going into the specifics of individual frameworks, this is the general recipe for using almost any web framework with OWIN and Katana:

1. First you create the Visual Studio project based on which host you want to use:
    - **Web Application** project if you want to host it with IIS/ASP.NET.
    - **Console Application** (or your custom host) if you want to use the self-hosting option.
    - **Class Library** or **Web Application** if you want to use the OwinHost.exe host.
2. Get the NuGet package for the hosting option you decide to use:
    - `Microsoft.Owin.Host.SystemWeb` if you are using the IIS/ASP.NET host.
    - `Microsoft.Owin.Host.HttpListener` if you are using self-host or OwinHost.exe.

3. Get the NuGet packages for the framework you want to work with, usually by downloading the compatibility package for your framework, which should have the references already.
4. Add the **`OwinStartup`** class via Visual Studio, which adds the startup class and the code OWIN packages in case they have not been added automatically during the previous steps.
5. Configure OWIN to use your web framework, and write the application based on your web framework as you would normally do.

ASP.NET Web API

Probably the most common scenario for using an OWIN-based application is the creation of a RESTful service using ASP.NET Web API.

Creating an ASP.NET Web API application with OWIN is not much different from doing it with the normal ASP.NET stack. Microsoft has slightly adapted the Web API framework to interface with the OWIN pipeline, but nothing else has changed: the only difference is that now you have to manually register Web API as OWIN middleware inside the pipeline.

Create the project

The hosting project can be any of the ones seen in the previous chapter, but for keeping it simple we will make an **ASP.NET Web Application**, using the **Empty** template, and selecting the **Web API** option under **Add folders and core references for**.

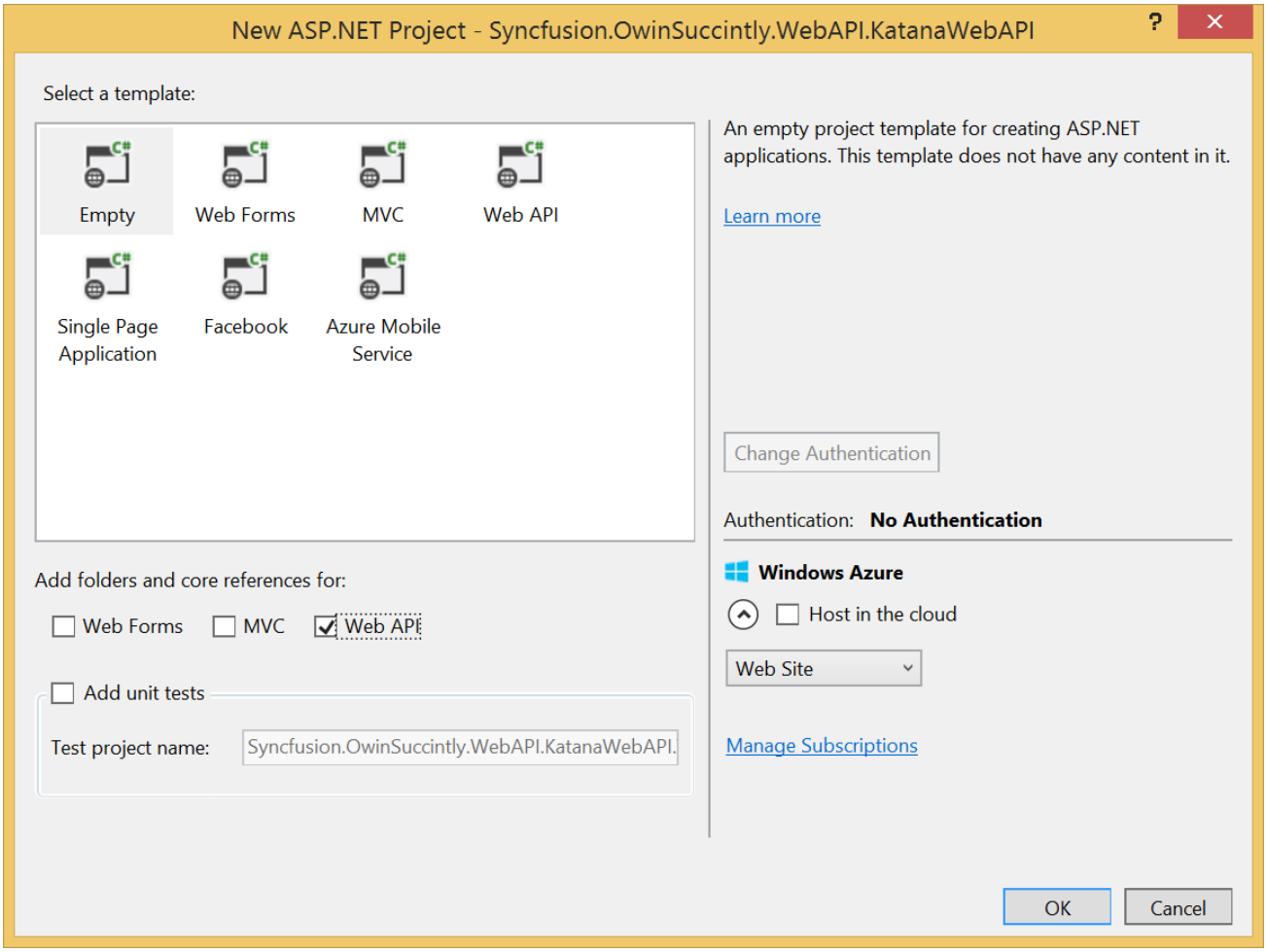

Figure 28: New ASP.NET Project

Note: You can also create the project using the Web API template, but that will also include references to ASP.NET MVC (which relies on System.Web) that are used to provide the API help page. In general you don't need it, especially under OWIN. It will not work as ASP.NET MVC is not yet compatible with OWIN.

Once you have created the project, you have to include the usual OWIN NuGet packages and the compatibility package that makes ASP.NET Web API understand the OWIN **`AppFunc`** and environment dictionary. Luckily, the compatibility package, **`Microsoft.AspNet.WebApi.Owin`**, comes with all the dependencies needed, except for the one about the OWIN server you want to use.

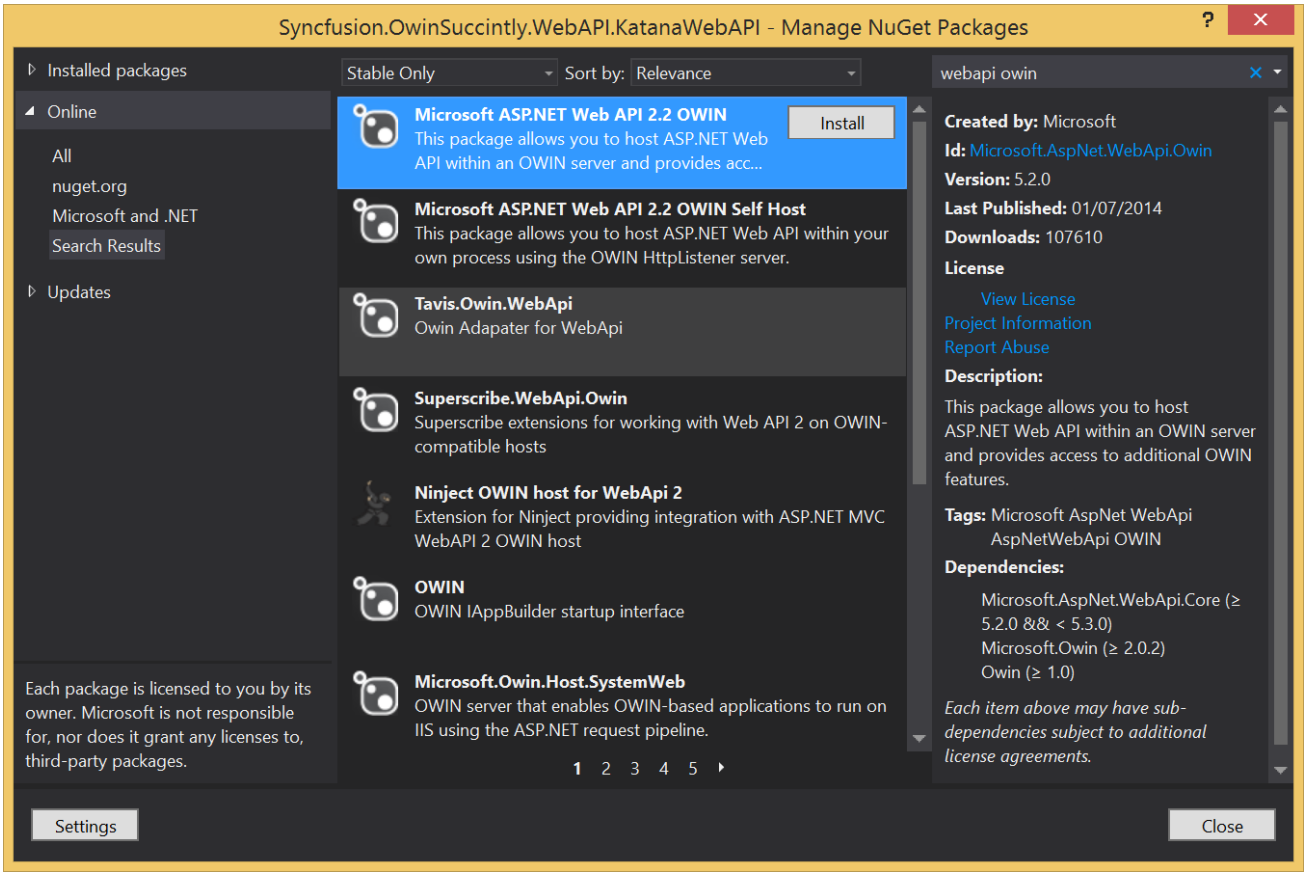

Figure 29: NuGet Package Manager with Microsoft.AspNet.WebApi.Owin selected

The previous figure shows which NuGet package to download. You could also download it via the package manager console:

```
PM> Install-Package Microsoft.AspNet.WebApi.Owin
```

Now you have to add the NuGet package for the server you want to use. In this case, since we are running on IIS, you would also need to get the **`Microsoft.Owin.Host.SystemWeb`** package.

Add Startup class and configure Web API

Now add the **`Startup`** class as usual, and add to it the configuration for Web API.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        HttpConfiguration config = new HttpConfiguration();
```

```
        config.Routes.MapHttpRoute(
            "default",
            "api/{controller}",
            new { controller = "Chapter" }
            );

        config.Formatters.XmlFormatter.UseXmlSerializer = true;
        config.Formatters.Remove(config.Formatters.JsonFormatter);

        app.UseWebApi(config);
    }
}
```

Code Listing 21

You should notice the code is not different from the configuration code that you would normally write in the **`WebApiConfig`** class of a standard Web API project.

Tip: If you want to test your Web API with a browser that doesn't have a JSON visualizer, you can remove the `JsonFormatter` ***and set the*** `XmlSerializer` ***by default to always return the XML representation of the results, like in the previous code listing.***

In addition there is the last line of code **`app.UseWebApi(config);`** that is used to add the Web API framework in the OWIN pipeline, supplying its configuration.

Since we are bootstrapping the application with OWIN, we don't need the default configuration classes from the Web API folders and references, so you can safely delete the **`Global.asax`** and the **`App_Start\WebApiConfig.cs`** files that have been created by the Visual Studio template.

The actual Web API code

Now that the application startup has been configured, you can write the Web API application like you are used to. The configuration in the previous code listing refers to a controller named **`Chapter`** as the default controller to redirect to, so let's write it.

```
public class ChapterController : ApiController
{
    // GET api/<controller>
    public IEnumerable<string> Get()
    {
        return new string[] {
            "Owin",
            "Katana",
            "Web Frameworks",
            "Azure",
```

```
            "Custom Middleware",
            "Authentication",
            "Appendixes"};
    }
}
```

Code Listing 22

The previous code listing is a very simple controller that returns a list of strings with the names of the chapters of this e-book. From this you can see that if you know how to write normal Web API applications, you also know how to write Web API applications under OWIN.

To test the application, just launch it by pressing **F5** and then browse to **`http://localhost:<port>/api`**.

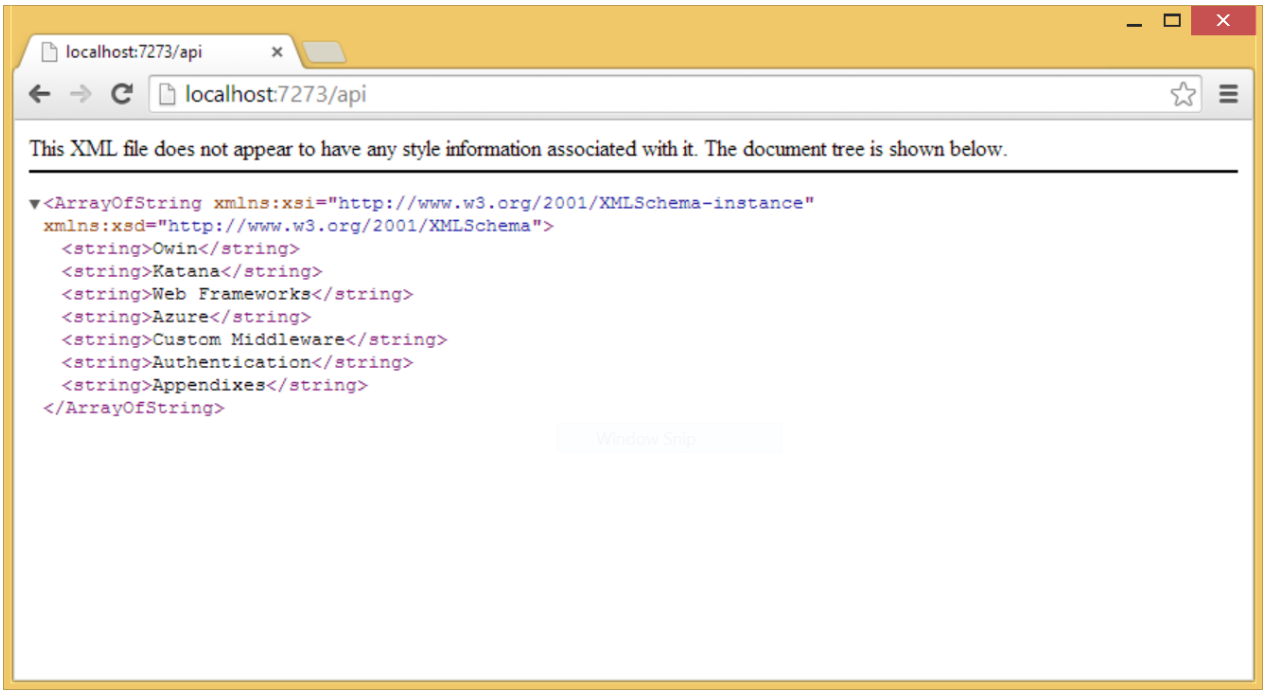

Figure 30: Web API response

ASP.NET MVC

The second scenario that could be hosted on top of OWIN is the creation of MVC-based web applications.

The framework of choice for this kind of application is ASP.NET MVC. Unfortunately, as already mentioned, it heavily relies on System.Web and thus cannot run on top of OWIN as middleware. This will happen with ASP.NET vNext, but for the moment, if you want to build an MVC-based application, you have only two choices:

- Still use ASP.NET MVC, which can be used without any problem in combination with OWIN (as you will see in Chapter 5), but you will lose the ability to run the OWIN application outside of IIS via either custom host or OwinHost.exe.
- Use another web framework, like NancyFx (which is very easy to use as you will see in the next section), FubuMVC, or Simple.Web.

NancyFx

NancyFx is, quoting from the official website nancyfx.org, a "lightweight, low-ceremony framework for building HTTP-based services on .NET and Mono. The goal of the framework is to stay out of the way as much as possible and provide a super-duper-happy-path to all interactions."

Brief Introduction to Nancy

Nancy is inspired by Sinatra (www.sinatrarb.com), a very popular web framework in Ruby. It is not an MVC framework, but rather a domain-specific language (DSL) for creating web applications, with no default dependencies on any hosting environment or data persistence framework.

Applications are built in **modules**, which define how the application behaves by specifying the routes and the actions that have to be performed when a route is matched.

```
public class ChaptersModule : NancyModule
{
  public ChaptersModule()
  {
      Get["/"] = _ =>
      {
          //do something for homepage
      };

      Get["/Chapters"] = _ =>
      {
          //return chapters
      };

      Get["/Chapters/{id}"] = params =>
      {
          //return chapter with given id, retrieved via param.id
      };
  }
}
```

Code Listing 23

The previous code listing demonstrates how to quickly set up an application with Nancy. Each action defined in the constructor of the module is a function that receives a dynamic dictionary, with the URL segments as input, and returns either the response directly or a ViewModel.

Using Nancy with Katana

The process of using Nancy is not different from the usual, so I'll just quickly go through the steps that are specific to Nancy.

After having created the project (the type you prefer) and adding the NuGet package for the hosting option you have chosen, get the compatibility package which, in the case of NancyFx, is called **`Nancy.Owin`**.

```
PM> Install-Package Nancy.Owin
```

This package also comes with the **`Nancy`** package, so there's no need to manually add it.

Once this has been done, you have to configure OWIN to use Nancy to handle the requests. This is done by adding one simple line of code to the **`Startup`** class.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.UseNancy();
    }
}
```

Code Listing 24

Finally, create a Nancy module to handle the requests for your web application.

```
public class HomeModule : NancyModule
{
    public HomeModule()
    {
        Get["/"] = _ => "Hello World!";

        Get["/{name}"] = parameters => "Hello " + parameters.name+"!";
    }
}
```

Code Listing 25

The code listing implements the same functionality of the last code listing of Chapter 2. It says "Hello World!" when requested without a URL segment, and says "Hello *[name]*!" when called with a URL segment.

Additional OWIN-Nancy Interaction Details

Before moving to the next framework, there are still a few more details to cover about using Nancy with OWIN:

- How to read the environment dictionary.
- How to configure Nancy as a pass-through component.

Read the OWIN Environment Dictionary

To access the OWIN environment dictionary from within an action, you can call the method **`this.Context.GetOwinEnvironment()`**, and then you can read the dictionary by specifying the key, for example, **`(string)env["owin.RequestPath"]`**.

Configure Nancy as a Pass-Through Component

Typically, applications are developed using more than one framework, and for that to be possible, all middleware components must pass the execution back to the pipeline when they are done with it, so you don't want Nancy to process all the requests. But the way we configured the OWIN pipeline using **`app.UseNancy()`** doesn't allow it because it registers Nancy as the final step in the pipeline. To allow Nancy to pass all requests it cannot handle to the next component in the pipeline, you need to configure Nancy slightly differently, using the **`PerformPassThrough`** option.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app
            .UseNancy(options =>
                options.PerformPassThrough = context =>
                    context.Response.StatusCode == HttpStatusCode.NotFound)
            .PageNotFoundMiddleware();
    }
```

Code Listing 26

The previous code listing instructs Nancy to pass the execution to the next middleware component in case its execution results in a **404 Not found**.

SignalR

As the last framework demonstrated in this chapter, SignalR is the solution provided by Microsoft for real-time communication on the web, which uses web sockets and a few other fallback protocols in case the browser or server doesn't support them.

Since the second version of the framework, the only way to build an application using SignalR is by using OWIN, which means that if you are using SignalR, you are already using it in combination with OWIN. Actually, the only thing you have to do besides create the required SignalR Hubs or Connections is configure the OWIN pipeline with the following code.

```
public void Configuration(IAppBuilder app)
{
    app.MapSignalR();
}
```

Code Listing 27

Using SignalR with Self-Host

A bit more challenging would be configuring SignalR to run inside a custom host. Since static files are not served with custom hosts, you have two options: the first is to create another static website with the HTML and JavaScript needed to invoke the SignalR Hub (hosted on IIS), but you would need to configure the SignalR endpoint with CORS (cross-origin resource sharing) in order to accept connections from a different domain. The second option is to configure the self-host to serve static files without having to build a different website.

Using a Separate Website

Build the Host

We start by creating a console application as we did in the previous chapter. Get the **`Microsoft.AspNet.SignalR.SelfHost`** NuGet package. It contains references to all the packages needed for self-hosting OWIN applications, and also contains the server-side packages of SignalR.

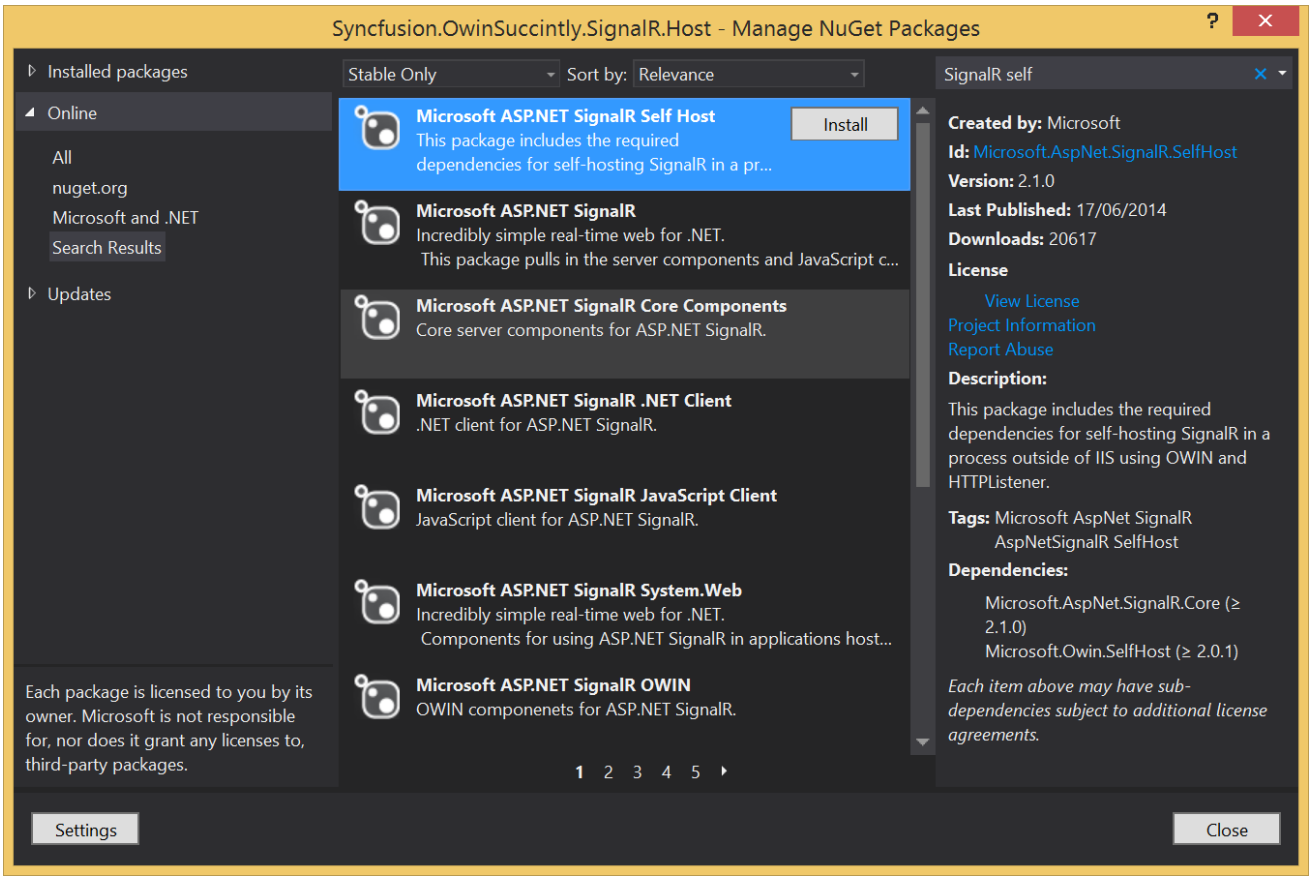

Figure 31: SignalR Self-Host package

Or you can also it install via the Package Manager.

```
PM> Install-Package Microsoft.AspNet.SignalR.SelfHost
```

You also need to reference the **Cors** library for OWIN to enable cross-origin access.

```
PM> Install-Package Microsoft.Owin.Cors.SignalR.SelfHost
```

With all the packages in the right place, we still need to add three classes:

- The code to launch the host, in **Program.cs**.
- The code to configure the OWIN application, in **Startup.cs**.
- The code of the actual SignalR application, in the form of a SignalR Hub, in a file called **ChatHub.cs**.

The code to launch the host is not different from what we have written already in the previous chapter.

```
static void Main(string[] args)
{
    string uri = "http://localhost:9999/";
    using (WebApp.Start<Startup>(uri))
    {
        Console.WriteLine("Started Listening on http://localhost:9999/");
        Console.ReadKey();
        Console.WriteLine("Finished");
    }
}
```

Code Listing 28

Also, the **`Startup`** class will not be very different from what we've seen previously.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.UseCors(CorsOptions.AllowAll);
        app.MapSignalR();
    }
}
```

Code Listing 29

The only specificity here is the line used to configure CORS inserted in the pipeline as additional OWIN middleware.

Finally, we write the SignalR-specific code. The "Hello World" example for web sockets applications is a chat app, so the following example code creates a simple **`ChatHub`**.

```
public class ChatHub : Hub
{
    public void Send(string message)
    {
        Clients.All.addMessage(message);
    }
}
```

Code Listing 30

If you are not familiar with SignalR Hubs, the code in the previous listing instructs the application to broadcast to all the connected clients the **`message`** that it receives when one client calls the **`Send`** method.

To test that everything works—well, at least the server part—launch the application and open your browser to the URL **http://localhost:9000/signalr/negotiate** to verify that SignalR is running and responding.

To verify that the proxy for the **ChatHub** has been created, browse to **http://localhost:9999/signalr/hubs**. This file is automatically generated by SignalR, and contains all the proxies needed so that client-side development is easier.

```
...
proxies['chatHub'] = this.createHubProxy('chatHub');
proxies['chatHub'].client = { };
proxies['chatHub'].server = {
    send: function (message) {
        return proxies['chatHub'].invoke.apply(proxies['chatHub'],
          $.merge(["Send"], $.makeArray(arguments)));
    }
};
...
```

Code Listing 31

The previous lines are an extract from the auto-generated JavaScript file **signalr/hubs**.

Build the Site

Now, to consume the SignalR application, we need an HTML page with a bit of JavaScript code. The easiest way to do it is by creating a **Web Application** project and downloading the NuGet package with the client-side libraries for SignalR.

The following command will download jQuery and the SignalR JavaScript library.

```
PM> Install-Package Microsoft.AspNet.SignalR.JS
```

Now create a **default.html** page and write the following code.

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR Simple Chat</title>
</head>
<body>
    <div class="container">
        <input type="text" id="message" />
        <input type="button" id="sendmessage" value="Send" />
        <ul id="discussion"></ul>
    </div>

    <script src="Scripts/jquery-1.6.4.min.js"></script>
```

```
    <script src="Scripts/jquery.signalR-2.1.0.min.js"></script>
    <script src="http://localhost:9999/signalr/hubs"></script>

    <script type="text/javascript">
        $(function () {
            $.connection.hub.URL = "http://localhost:9999/signalr";
            var chat = $.connection.chatHub;
            chat.client.addMessage = function (message) {
                var encodedMsg = $('<div />').text(message).html();
                $('#discussion').append('<li>'+ encodedMsg + '</li>');
            };
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    chat.server.send($('#message').val());
                    $('#message').val('').focus();
                });
            });
        });
    </script>
</body>
</html>
```

Code Listing 32

I won't go into much detail since this is not a book about SignalR, but I'd like to comment on a few lines just to give you the context in case you are not familiar with it.

The page references three JavaScript libraries: jQuery, the SignalR library, and the auto-generated proxies.

```
<script src="Scripts/jquery-1.6.4.min.js"></script>
<script src="Scripts/jquery.signalR-2.1.0.min.js"></script>
<script src="http://localhost:9999/signalr/hubs"></script>
```

Code Listing 33

As the first thing in the JavaScript code, you configure the URL of the Hub and get a reference of the proxy class.

```
$.connection.hub.URL = "http://localhost:9999/signalr";
var chat = $.connection.chatHub;
```

Code Listing 34

After the connection is established, you can configure the button to call the **send** message of the proxy.

```
$.connection.hub.start().done(function () {
    $('#sendmessage').click(function () {
        chat.server.send($('#message').val());
    });
```

Code Listing 35

You also need a function that the SignalR Hub can call to update the page when messages are broadcasted.

```
chat.client.addMessage = function (message) {
    var encodedMsg = $('<div />').text(message).html();
    $('#discussion').append('<li>'+ encodedMsg + '</li>');
};
```

Code Listing 36

Notice that **Send** and **addMessage** are the same method names used in the **ChatHub** previously defined. This is how the communication between client and server happens.

Now start both projects and open **default.html** in two separate windows of your browser and see that messages typed in one window are also displayed in the other.

Serving Static Files from Self-Host

This option is a bit trickier to set up, but is also probably the most common scenario for self-hosted solutions: you want all the resources for your self-hosted site served from your custom host.

Start by creating a self-host for SignalR and configure it as you did before, just don't add the **Cors** NuGet package (the requests will come from the same domain).

Now we have to configure the application to serve static files; there is a NuGet package for this called **Microsoft.Owin.StaticFiles**.

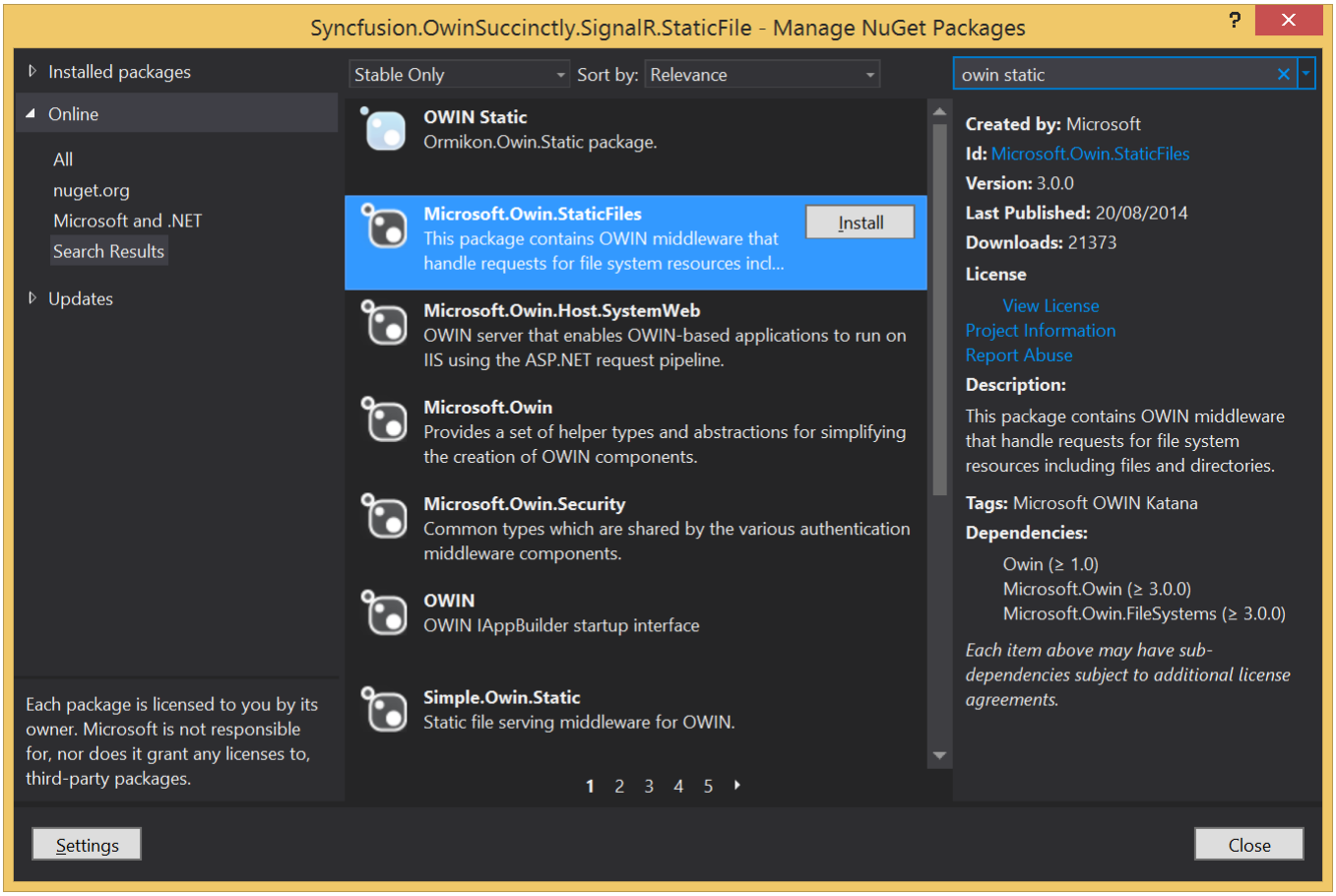

Figure 32: OWIN StaticFiles package

With this package, you can configure your application to serve static files too. All you need is to add a line in the **`Configuration`** method to enable it.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.UseStaticFiles("/files");
        app.MapSignalR();
    }
}
```

Code Listing 37

With this **`UseStaticFiles`** method, you add a static file OWIN middleware that serves all the files located under the **`/files`** folder. To configure static file serving, you can also use the **`UseFileServer`** method, which allows you to specify default files and folder browsing.

The last step is to create a folder called files and copy all the files from the static website created previously (the **`default.html`** and all the JavaScript files in the **`Scripts`** folder) into it. The final project structure should look like the following figure.

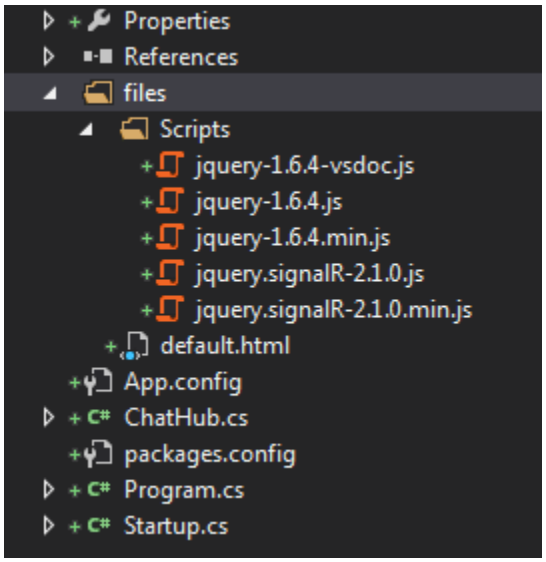

Figure 33: Project folder structure

This is a console application, so static files are not copied by default to the output folder. Remember to specify the **Build Action** as **Content** and **Copy to Output Directory** as **Copy always**.

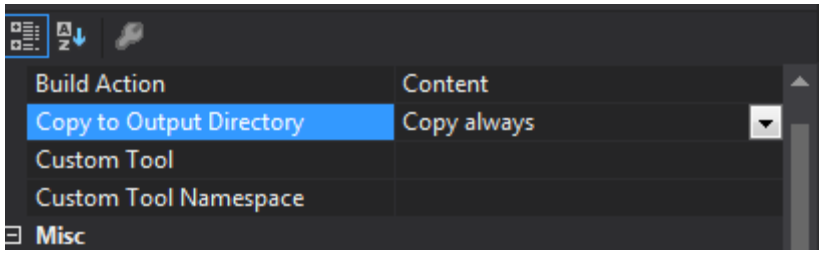

Figure 34: Build action

Now launch the project and open the **`http://localhost:[port]/files/default`** URL and you will see the same chat application as before, but all served from the same host.

Storing Files in Embedded Resources

If you want your hosting application to be even tidier, you can also configure the static file middleware to serve files from resource files. This is done by specifying which **`IFileSystem`** implementation to use. Two are available in the **`StaticFiles`** middleware: the one used by default, **`PhysicalFileSystem`**, and **`EmbeddedResourceFileSystem`**.

To avoid always repeating the initial part of the resource identifier when creating the instance of **`EmbeddedResourceFileSystem`**, you can specify the root of your files' structure.

```
app.UseStaticFiles(new StaticFileOptions()
{
    FileSystem = new
EmbeddedResourceFileSystem("Syncfusion.OwinSuccinctly.SignalR.StaticFiles.files")
});
```

Code Listing 38

Change the **Build Action** to **Embedded Resource** and change the path to physical scripts to use **.** as the path separator instead of **/**.

```
<script src="Scripts.jquery-1.6.4.min.js"></script>
<script src="Scripts.jquery.signalR-2.1.0.min.js"></script>
<script src="/signalr/hubs"></script>
```

Code Listing 39

Now open the URL **http://localhost:9000/default.html** and use the same chat application.

If you are stuck and always get **404** errors when trying to get to files, you can enable directory browsing on the whole site and call the root of the site to get the full name of the files after they have been converted into embedded resources.

```
app.UseDirectoryBrowser(new DirectoryBrowserOptions()
{
    FileSystem = new EmbeddedResourceFileSystem(@""),
});
```

Code Listing 40

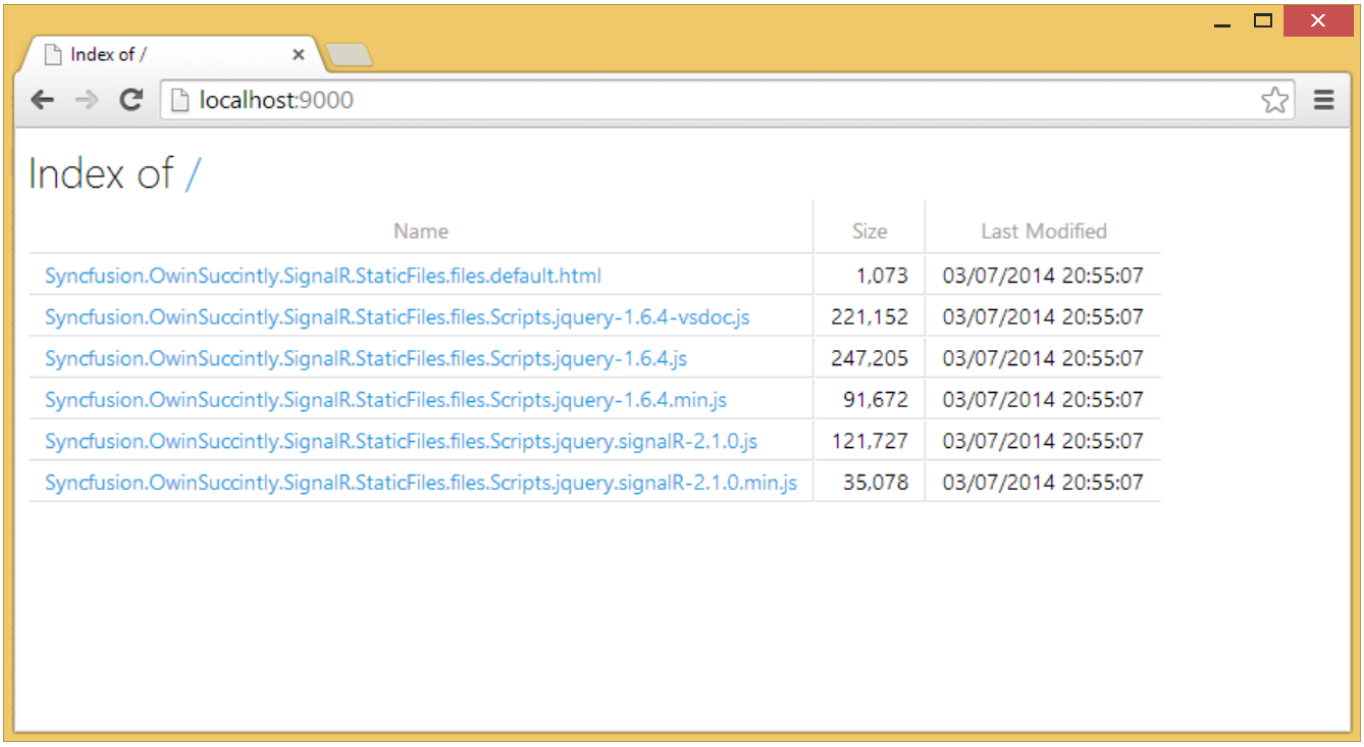

Figure 35: Directory browsing

Combining More Frameworks

In this chapter, you've seen how to use all the frameworks on their own, but one of the nicer features of OWIN is that you can combine many frameworks in a single application and use standard OWIN middleware components along the pipeline for cross-cutting concerns, like logging, caching, or authentication. You could use many frameworks without OWIN, but you would have to implement those horizontal functionalities for each of the frameworks you were integrating.

```
public void Configuration(IAppBuilder app)
{
    HttpConfiguration config = new HttpConfiguration();
    config.Routes.MapHttpRoute(
        "default",
        "api/{controller}"
        );

    app.Use(async (context, next) =>
        {
            Trace.WriteLine("-> " +context.Request.Path);
            await next();
        });
```

```
    app.UseNancy(options =>
        options.PerformPassThrough = context =>
            context.Response.StatusCode == HttpStatusCode.NotFound);

    app.UseWebApi(config);
}
```

Code Listing 41

The previous code listing shows Nancy and Web API used together, and both use common logging code written as inline OWIN middleware. In this case it was important to configure Nancy as a pass-through component, otherwise it would have thrown a 404 HTTP error for all the URLs it couldn't handle (like in the case of Web API routes).

Conclusions

In this chapter, you have learned how to use your favorite web frameworks with Katana and theoretically every other host or server that implements the OWIN specs.

You have seen how to use Web API, Nancy, and SignalR. You have also discovered that the current ASP.NET MVC cannot be used with OWIN servers, but it will be possible with the next version of ASP.NET MVC (version 6). You have also learned how to serve static files from OWIN self-hosted applications.

In the next chapter, you will learn how to add custom components to the OWIN pipeline, including building your own middleware.

Chapter 4 Building Custom Middleware

In the previous chapter, you learned how to build OWIN-based applications using web frameworks and middleware components written by others, but you will probably need to write custom components specific to your application.

In this chapter, you will learn how to do it, both as simple inline functions and as more isolated components in their own class and library.

Defining Middleware Using the Core OWIN Specs

As you have seen in Chapter 1, an OWIN middleware component is basically just a function with the **AppFunc** signature.

```
using AppFunc = Func<
IDictionary<string, object>, // Environment
Task>; // Done
```

Code Listing 42

To use a middleware component, you also have to register it in the OWIN pipeline via the **IAppBuilder** interface. Even if not strictly defined in the specs (this topic is actually being discussed in the OWIN list at the time of writing), you register a middleware component by calling the **Use** method of that interface and supplying a function with the following signature.

```
Func<AppFunc, AppFunc>
```

Code Listing 43

Basically, you pass a function that takes the next component in the OWIN pipeline and returns a new middleware component to the **Use** method. Stated like this, it sounds complicated, so we better look at an example.

```
app.Use(new Func<Func<IDictionary<string, object>, Task> /*next*/
    , Func<IDictionary<string, object> /*env*/, Task>>
    (next => (async env =>
{
    var response = env["owin.ResponseBody"] as Stream;
    string pre = "\tMW 2 - Before (Inline AppFunc)\r\n";
    string post = "\tMW 2 - After (Inline AppFunc)\r\n";
    await response.WriteAsync(Encoding.UTF8.GetBytes(pre), 0, pre.Length);
    await next.Invoke(env);
```

```
    await response.WriteAsync(Encoding.UTF8.GetBytes(post), 0, post.Length);
}));
```

Code Listing 44

The one shown in the previous code listing is a very simple middleware that, using only what comes with the OWIN specs, writes two strings ("**Before**" and "**After**") to the response stream while calling the next component in the pipeline. The **next** variable is the parameter that gets passed in the "outer" function, while **env** is the input parameter of the "inner" function. Also notice that the function accesses the environment dictionary and casts the value to the right type before using them.

This inline method is okay if you just want to add some tracing, but in case your middleware class is more complicated, you might want to create a separate class.

```
public class RawOwinMiddleware
{
    private Func<IDictionary<string, object>, Task> next;

    public RawOwinMiddleware(Func<IDictionary<string, object>, Task> next)
    {
        this.next = next;
    }

    public async Task Invoke(IDictionary<string, object> env)
    {
        var response = env["owin.ResponseBody"] as Stream;
        string pre = "\t\t\tMW 4 - Before (RawOwinMiddleware)\r\n";
        string post = "\t\t\tMW 4 - After (RawOwinMiddleware)\r\n";
        await response.WriteAsync(Encoding.UTF8.GetBytes(pre), 0, pre.Length);
        await next.Invoke(env);
        await response.WriteAsync(Encoding.UTF8.GetBytes(post), 0, post.Length);
    }
}
```

Code Listing 45

It does essentially the same things, but the **IAppBuilder** instance behaves differently when it receives a **Type** instead of an inline lambda expression. In this case, it creates an instance of the middleware type, passing to the constructor the next element in the pipeline, and then it calls the **Invoke** method when the middleware has to be executed.

To register this middleware component, you need to pass its type to the **Use** method: **app.Use(typeof(RawOwinMiddleware))**.

Defining Middleware with the Help of Katana Helpers

Using the OWIN specs directly is good if you want to better understand how middleware works and gets registered in the pipeline, or if you want to release it so that it works in all OWIN servers without taking a dependency on the **`Microsoft.Owin`** DLL. But if none of these constraints are a problem, you can build middleware using the **`Use`** extension method provided by the Katana project, which needs a **`Func<IOwinContext, Func<Task>, Task>`**—basically a function that receives the **`IOwinContext`** and the next element in the pipeline and returns the **`Task`** to execute.

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("MW 1 - Before (Katana inline
middleware)\r\n");
    await next();
    await context.Response.WriteAsync("MW 1 - After (Katana inline
middleware)\r\n");
});
```

Code Listing 46

This code example does the same thing as the one in the previous section, but is much more succinct.

Of course, with the Katana helpers you can write middleware as a separate class.

```
public class LoggerMiddleware : OwinMiddleware
{
    public LoggerMiddleware(OwinMiddleware next)
        : base(next)
    {

    }

    public override async Task Invoke(IOwinContext context)
    {
        await context.Response.WriteAsync("\t\t\t\tMW 5 - Before (Katana helped
middleware class)\r\n");
        await this.Next.Invoke(context);
        await context.Response.WriteAsync("\t\t\t\tMW 5 - After (Katana helped
middleware class)\r\n");
    }
}
```

Code Listing 47

In this case, you have to write a class that inherits from **OwinMiddleware**, but the rest is almost the same as what you wrote using just the OWIN specs. Also you need to register the component by passing its type to the **Use** method: **app.Use(typeof(LoggerMiddleware))**.

As seen in Chapter 2, the big advantage of using the **Microsoft.Owin** helpers is the **IOwinContext** that wraps all the elements in the OWIN environment dictionary in a strongly typed fashion.

Before closing the chapter, I want to remind you of a pretty important point about middleware: all their "before" actions are executed in the order in which the components are registered in the pipeline, while the "after" actions are executed in the opposite order. This is also well-visualized in Figure 36.

```
   public void Configuration(IAppBuilder app)
    {
        app.Use(async (context, next) =>
        {
            await context.Response.WriteAsync("MW 1 - Before (Katana helped inline
 middleware)"+Environment.NewLine);
            await next();
            await context.Response.WriteAsync("MW 1 - After (Katana helped inline
middleware)" + Environment.NewLine);
        });

        app.Use(new Func<Func<IDictionary<string, object>, Task> /*next*/
            , Func<IDictionary<string, object> /*env*/, Task>>
            (next => (async env =>
        {
            var response = env["owin.ResponseBody"] as Stream;
            string pre = "\tMW 2 - Before (Inline AppFunc)\r\n";
            string post = "\tMW 2 - After (Inline AppFunc)\r\n";
            await response.WriteAsync(Encoding.UTF8.GetBytes(pre), 0, pre.Length);
            await next.Invoke(env);
            await response.WriteAsync(Encoding.UTF8.GetBytes(post), 0, post.Length
);
        })));

        app.Use(new Func<Func<IDictionary<string, object>, Task> /*next*/
            , Func<IDictionary<string, object> /*env*/, Task>>
            ( next => env => Invoke(next, env)));

        app.Use(typeof(RawOwinMiddleware));
        app.Use(typeof(LoggerMiddleware));

        app.Run(ctx =>
        {
            return ctx.Response.WriteAsync("\t\t\t\t\tApp - Hello World!" + Enviro
nment.NewLine);
        });
    }

    private async Task Invoke(Func<IDictionary<string, object>, Task> next
```

```
    , IDictionary<string, object> env)
    {
        var response = env["owin.ResponseBody"] as Stream;
        string pre = "\t\tMW 3 - Before (Delegate)\r\n";
        string post = "\t\tMW 3 - After (Delegate)\r\n";
        await response.WriteAsync(Encoding.UTF8.GetBytes(pre), 0, pre.Length);
        await next.Invoke(env);
        await response.WriteAsync(Encoding.UTF8.GetBytes(post), 0, post.Length);

    }
}

}
```

Code Listing 48

The following figure shows how all the components defined in this chapter are executed if all of them were registered in a pipeline as in the previous code listing.

```
MW 1 - Before (Katana helped inline middleware)
        MW 2 - Before (Inline AppFunc)
                MW 3 - Before (Delegate)
                        MW 4 - Before (RawOwinMiddleware)
                                MW 5 - Before (Katana helped middleware class)
                                        App - Hello World!
                                MW 5 - After (Katana helped middleware class)
                        MW 4 - After (RawOwinMiddleware)
                MW 3 - After (Delegate)
        MW 2 - After (Inline AppFunc)
MW 1 - After (Katana helped inline middleware)
```

Figure 36: Trace of calls to middleware

Conclusions

In this chapter, you have learned how to build custom middleware components.

In the next chapter, you will learn how to secure your application, configure other middleware components to enable the standard form authentication, and also how to leverage social authentication with Twitter, Facebook, Google, and others.

Chapter 5 Authentication with Katana

Introduction

So far we have learned what OWIN is, what Katana is, the idea behind them, their approach, and also how to build custom middleware components.

In this chapter, we are going to apply what we learned until now, starting from one of the most useful middleware components available for Katana, the authentication process.

Given that the web is evolving quickly and many websites require different kinds of authentication, we will see how to use the classic **form authentication**, and also how to integrate **social authentication** with the most used social networks (Twitter, Facebook, and Google).

Authentication

Authentication is one of the topics that best fits with the idea of middleware, since it is something that has to be verified at each request and is the same in all applications.

As explained in the previous chapters, middleware is something that can handle a request, do something with it, and then forward it to the application, or stop the execution if something is wrong.

The authentication process is also one of the best real-life examples to understand how middleware works within OWIN.

What happens when someone requests a URL that requires authentication?

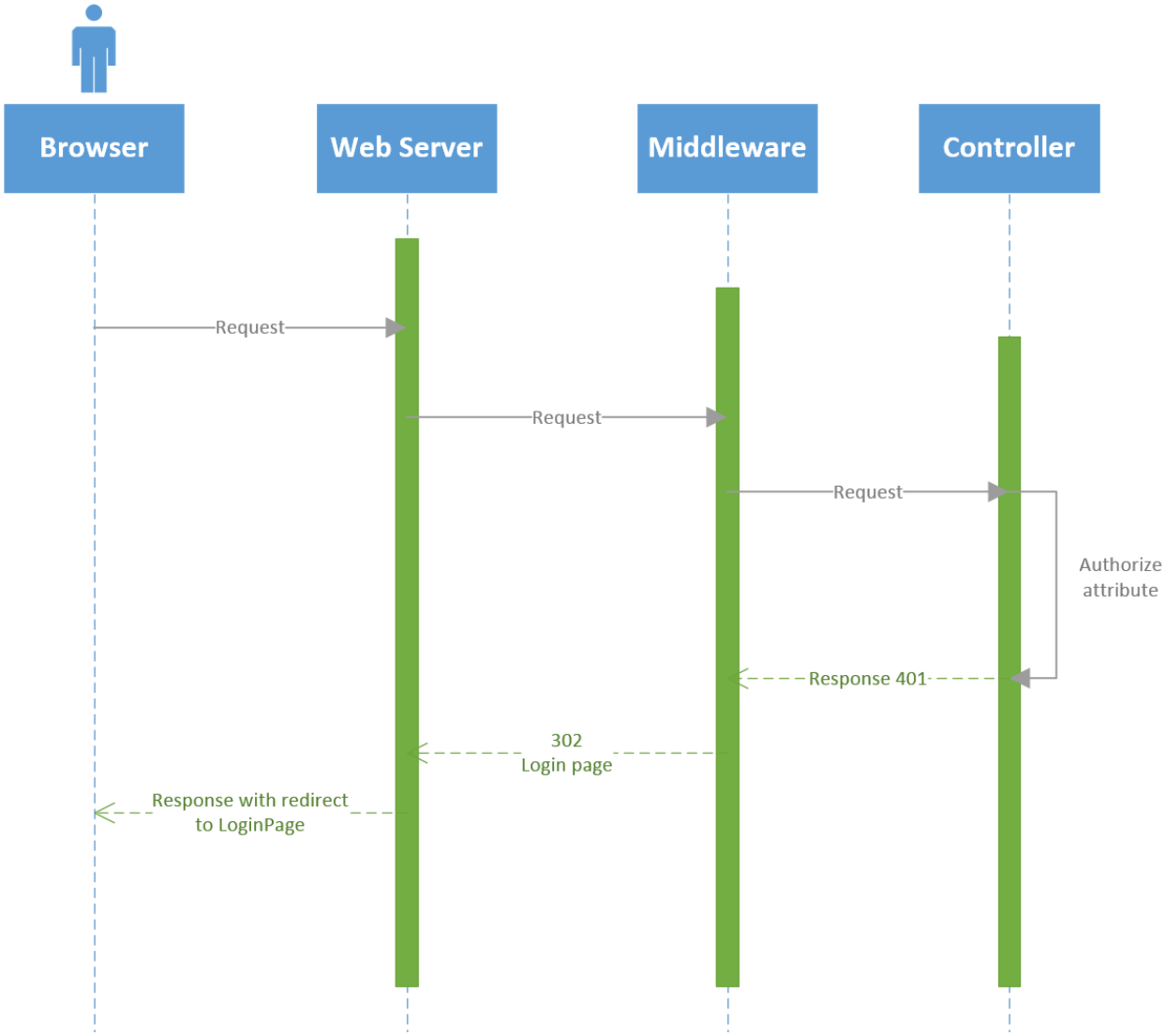

Figure 37: Sequence diagram of login process

As you can see in the sequence diagram in Figure 37, the middleware changes the 401 unauthorized response coming from the controller to a 302, redirecting the user to the login page.

Using Katana, you can choose from several packages for authentication. By using NuGet, you can choose from the following types of authentication:

- Forms authentication
- Twitter authentication
- Facebook authentication
- Google authentication
- Windows Azure
- Active Directory

- OpenID
- Microsoft account authentication
- Many more

Form Authentication

Form authentication is the most widespread authentication and, in the .NET world, is most likely to be used with ASP.NET MVC because you will need to render HTML.

Before we go too deep into the code, it is important to understand the difference between the form authentication managed by OWIN and the form authentication managed by ASP.NET MVC and WebForms.

Features

The following table compares the form authentication features between ASP.NET MVC and OWIN.

Table 4: Form Authenteication Features in ASP.NET MVC and OWIN

Features	MVC Authentication	OWIN Authentication
Cookie Authentication	Yes	Yes
Cookieless Authentication	Yes	No
Expiration	Yes	Yes
Sliding Expiration	Yes	Yes
Token Protection	Yes	Yes
Claims Support	No	Yes
Web Farm Support	Yes	Yes

Features	MVC Authentication	OWIN Authentication
Unauthorized Redirection	Yes	Yes

As you can see, Katana's implementation covers almost all the features released by ASP.NET MVC plus claim authentication. This means that it is a good choice for your .NET web application. Beware though, if you need to keep cookieless authentication in your website, you have to create your own custom middleware (or do not use OWIN).

Tip: Form authentication with Katana requires ASP.NET Identity, so if you don't know it I recommend taking a look at the ASP.NET website: http://www.asp.net/identity.

Installation

There are two ways to create a project using OWIN for authentication. The first is to use the template included with ASP.NET MVC 5 and create your application on top of it.

The second is to install and configure the right packages manually, which is probably the best way if you want to add OWIN to an existing application.

MVC 5 Template

The absolute easiest way is to use an ASP.NET MVC 5 template. To do so, open Visual Studio and create a new web application project with the MVC template.

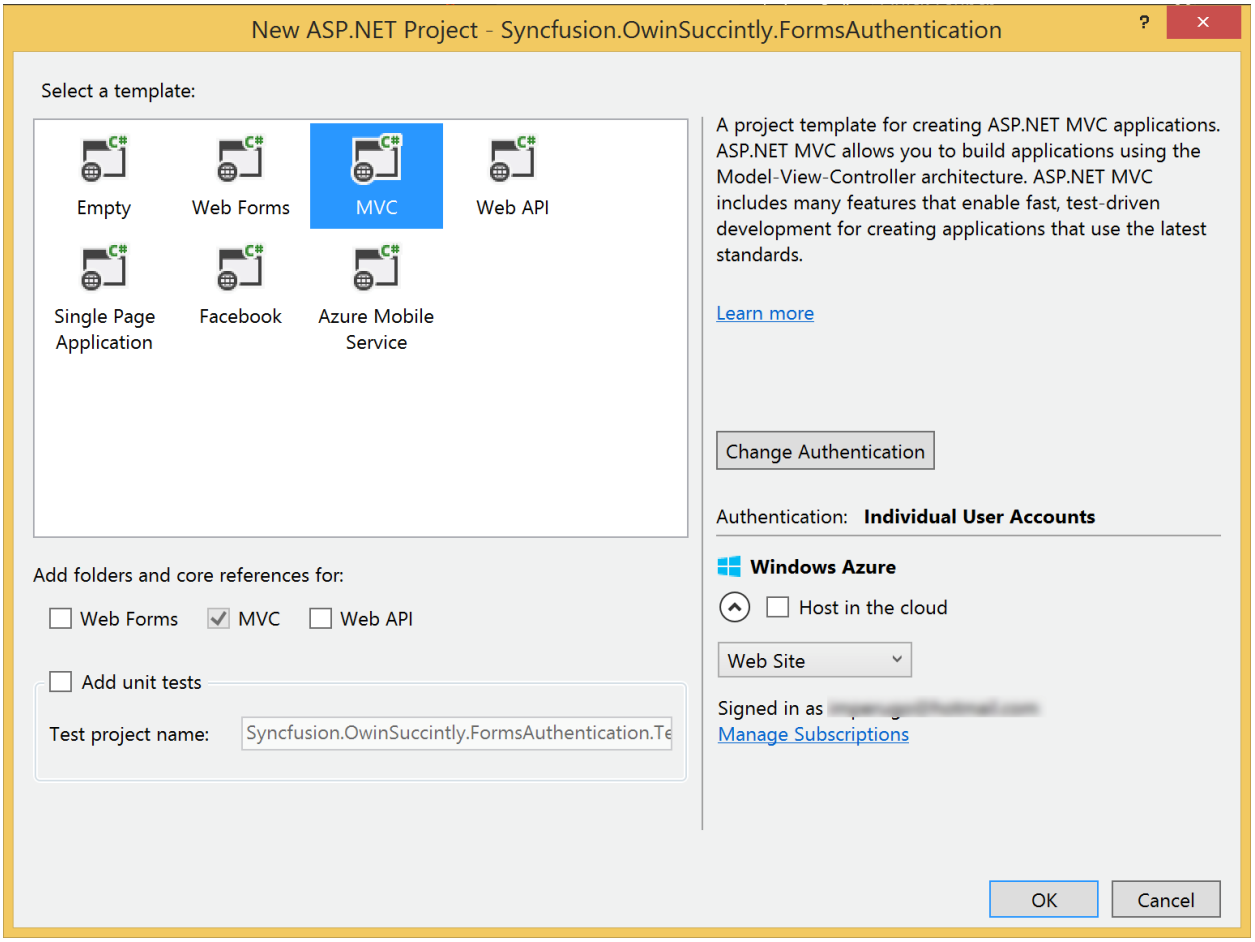

Figure 38: New ASP.NET MVC project

Before clicking **OK**, it is important to specify the authentication mode by clicking the **Change Authentication** button on the right side of the window.

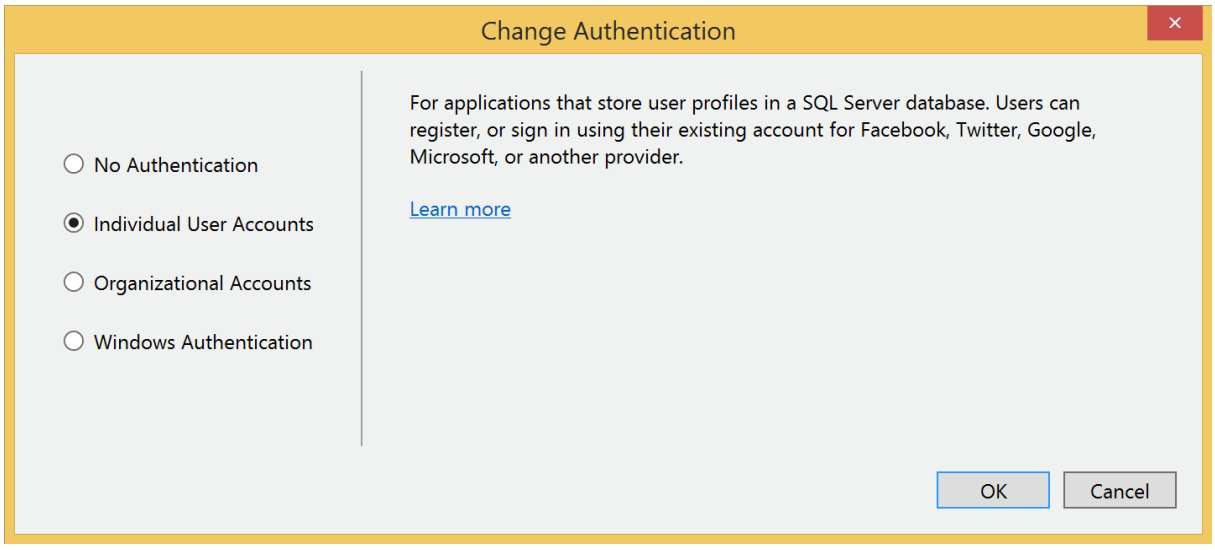

Figure 39: Changing the authentication mode

For form authentication, we are going to use the **Individual User Accounts** option.

The result is a web application that supports form authentication and social authentication (Facebook, Google, Twitter, and so on), and uses Entity Framework to create and save credentials into a SQL database.

From Scratch or an Existing Application

Another way to implement form authentication is to manually add OWIN. This is probably the most useful approach if you're adding the authentication to an existing project.

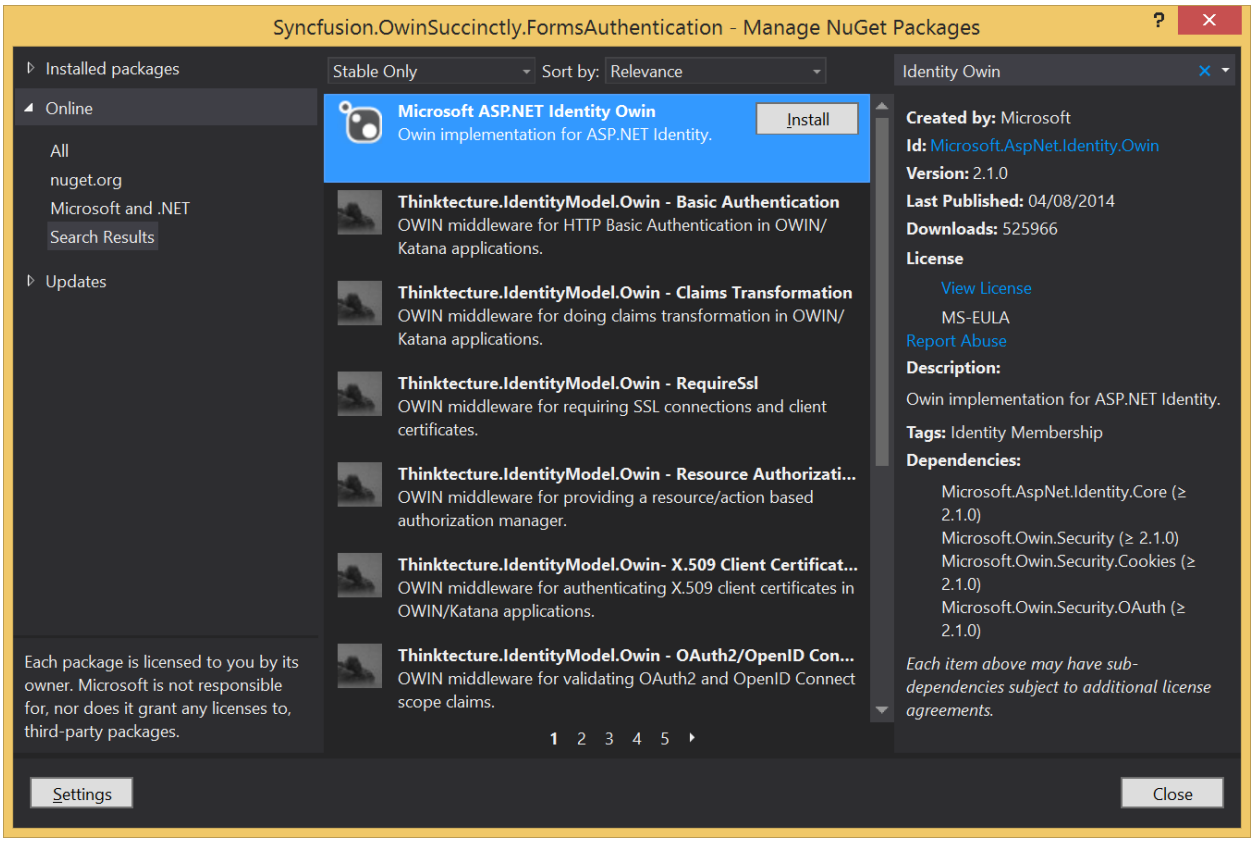

Figure 40: Adding NuGet package for OWIN implementation of ASP.NET Identity

The first step is to install the right packages: **`Microsoft.AspNet.Identity.Owin`** (as shown in Figure 40) and **`Microsoft.Owin.Host.SystemWeb`**.

The following figure shows what you should see installed in your project after installing the two packages and their dependencies.

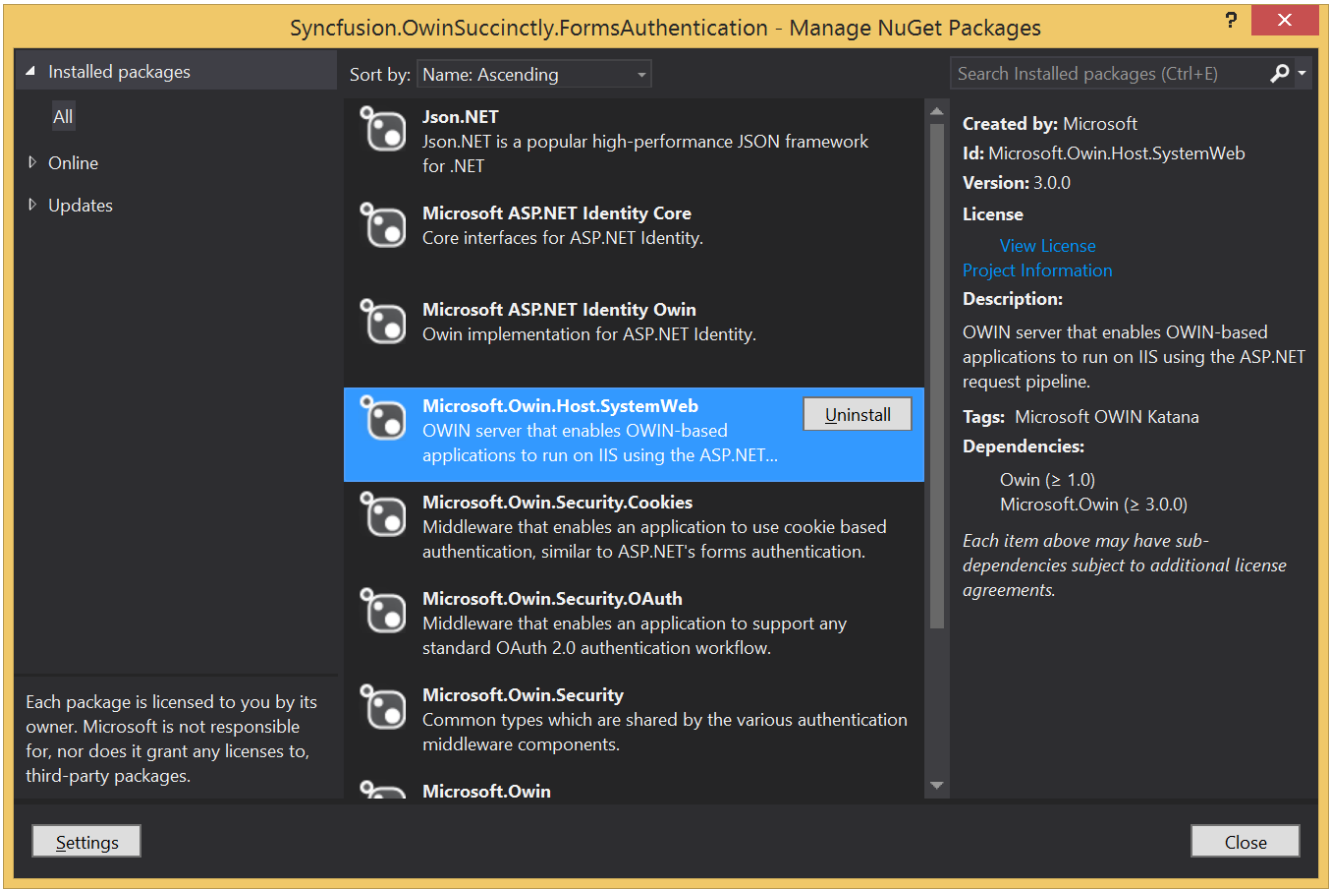

Figure 41: Installed NuGet packages

The next step is to create the startup class needed by OWIN.

```
using Microsoft.Owin;
using Owin;

[assembly:
OwinStartupAttribute(typeof(Syncfusion.OwinSuccinctly.FormsAuthentication.Startup)
)]
namespace Syncfusion.OwinSuccinctly.FormsAuthentication
{
    using System;

    using Microsoft.AspNet.Identity;
    using Microsoft.AspNet.Identity.Owin;
    using Microsoft.Owin.Security;
    using Microsoft.Owin.Security.Cookies;

    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseCookieAuthentication(
```

```
                new CookieAuthenticationOptions
                    {
                        AuthenticationType =
DefaultAuthenticationTypes.ApplicationCookie,
                        AuthenticationMode = AuthenticationMode.Active,
                        LoginPath = new PathString("/Account/Login")
                    });
        }
    }
}
```

Code Listing 49

The code is pretty simple. The extension method **`UseCookieAuthentication`** enables the authentication and **`CookieAuthenticationOptions`** is its configuration.

The first important thing to notice is the two properties **`AuthenticationType`** and **`AuthenticationMode`**.

The first is part of ASP.NET Identity. The second one is part of OWIN, for which there are two possible values as shown in the following table. The descriptions come from the official MSDN site.

Table 5: AuthenticationMode values

Value	Description
Active	In Active mode the authentication middleware will alter the user identity as the request arrives, and will also alter a plain 401 as the response leaves.
Passive	In Passive mode the authentication middleware will only provide user identity when asked, and will only alter 401 responses where the authentication type named in the extra challenge data.

Of course, there are other important settings we have to manage like authentication cookies, login URL, and so on. The following table shows all the available options you can manage with the **`CookieAuthenticationOptions`** class.

Table 6: Cookie authentication options

Options	Description
`CookieDomain`	Defines the cookie domain.

Options	Description
CookieHttpOnly	True (default) or False defines whether the cookie is **HttpOnly**.
CookieName	The name of the cookie (e.g. **AuthorizationCookie**).
CookiePath	The path of the cookie. The default is **/**.
CookieSecure	An enum with three options: **Always** (HTTPS scenario), **Never** (HTTP), and **SameAsRequest** (hybrid HTTP/HTTPS).
ExpireTimeSpan	Defines the expiration of the cookie. The default is 14.
ReturnUrlParameter	The name of the return parameter used for the redirect after a successful login.
SlidingExpiration	Manages the expiration extension for active users.

As explained in the authentication section at the beginning of the chapter, the **MVC controller** is really important because it contains the authorize attribute necessary to prevent anonymous access.

Now that the OWIN part is complete, it is time to write some code on MVC.

Note: In this book, we are not going to explain ASP.NET MVC, so it is important to know what a controller, action, and routing is, or how to manage a POST request.

Because the login page specified in the OWIN configuration is **/Account/Login**, the first thing to do is to create the **AccountController** and the **Login** action.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Web;
using System.Web.Mvc;

namespace Syncfusion.OwinSuccinctly.FormsAuthentication.Controllers
{
    using System.Security.Claims;
    using System.Threading.Tasks;

    using Microsoft.AspNet.Identity;
    using Microsoft.Owin.Security;

    using Syncfusion.OwinSuccinctly.FormsAuthentication.Models;

    public class AccountController : Controller
    {
        //
        // GET: /Account/Login
        [AllowAnonymous]
        public ActionResult Login(string returnURL)
        {
            ViewBag.ReturnURL = returnURL;
            return View();
        }


        //
        // POST: /Account/Login
        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Login(LoginViewModel model, string
returnURL)
        {
            if (ModelState.IsValid)
            {
                if (model.Email == "user@example.com" && model.Password ==
"Password1")
                {

AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie);

                    var user = new IdentityUser("1") { UserName = "user" };

                    var manager = new UserManager<IdentityUser>(new
IdentityStore());

                    ClaimsIdentity identity = await
manager.CreateIdentityAsync(user, DefaultAuthenticationTypes.ApplicationCookie);
                    AuthenticationManager.SignIn(new AuthenticationProperties() {
IsPersistent = model.RememberMe }, identity);
                    return this.Redirect(returnURL);
                }
                else
                {
```

```
                    ModelState.AddModelError("", "Invalid username or password.");
                }
            }

            // If we got this far, something failed, redisplay form
            return View(model);
        }

        private IAuthenticationManager AuthenticationManager
        {
            get
            {
                return HttpContext.GetOwinContext().Authentication;
            }
        }
    }
}
```

Code Listing 50

If you have never used ASP.NET Identity, this code probably seems complex, but it is really powerful. In fact, we will see later how to combine different kinds of authentication using the same approach.

The most important method of the previous code block is **`AuthenticationManager.SignIn`**. It creates the authentication cookie used to check credentials across the requests. The other methods (**`CreateIdentityAsync`** and **`UserManager<IdentityUser>`**) are needed by ASP.NET Identity.

Testing

If you did everything right, running the app and typing *user@example.com* as the username and *Password1* as the password in the login form should log you in.

Social Authentication

As we have just seen, using form authentication with OWIN and ASP.NET Identity is really simple. Now we will see how to extend the authentication, allowing users to log into the application using social networks like Twitter, Facebook, Google, and so on.

A Brief of Social Authentication

Before looking at how to implement social authentication in our application, it is important to know that all the most famous social networks use OAuth 1.x/2.x as their authentication workflow.

OAuth was not designed for authentication but for authorization decisions across a network of web-enabled applications and APIs. In fact, you can use OAuth to authenticate a user or an application that needs to use external APIs.

Another important authentication protocol (not explained in this book) is OpenID which is a simple identity layer, designed on top of OAuth 2.0 protocol, that offers the basic profile information about the end-user in an interoperable and REST-like manner.

Introducing OAuth

In this book, we are not going to explain how OAuth works or how to implement an OAuth server, but we need to know what it is.

Here is a good description from the OAuth website:

> *An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications.*

OAuth is really helpful because users don't have to complete another registration process, but can simply log in via one of the social networks they already use.

In addition, no user credentials are shared with the third-party application because all the communication between the client app (your website in this case) and the authorization server (Facebook, Twitter, Google and so on) is token-based.

The next figure shows the sequence diagram of an OAuth authorization flow.

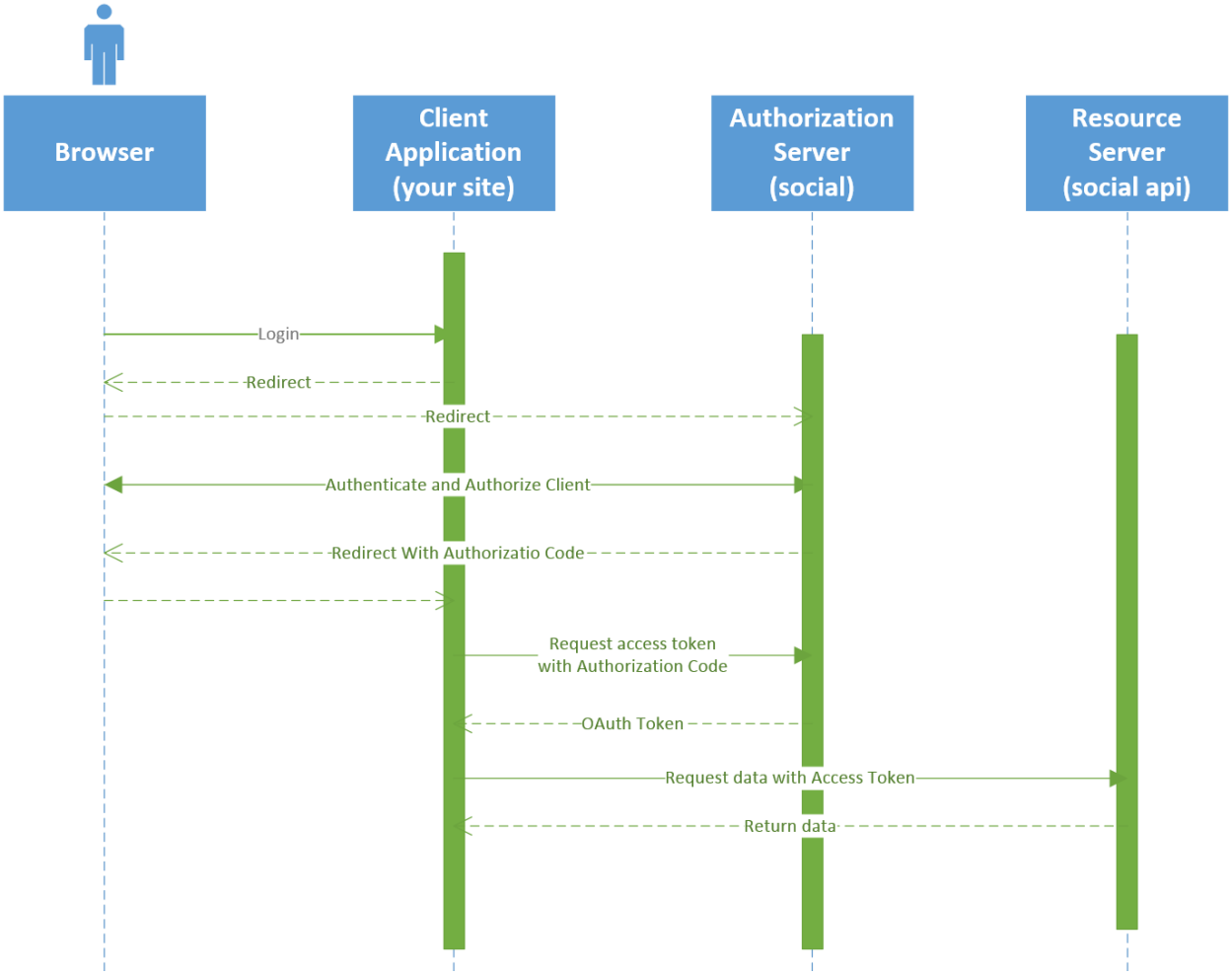

Figure 42: OAuth sequence diagram

Even though this workflow may seem complex, in fact it is not. We are already used to this kind of workflow when we log into an application using a social network like Facebook. In fact, we have seen this screen thousands of times.

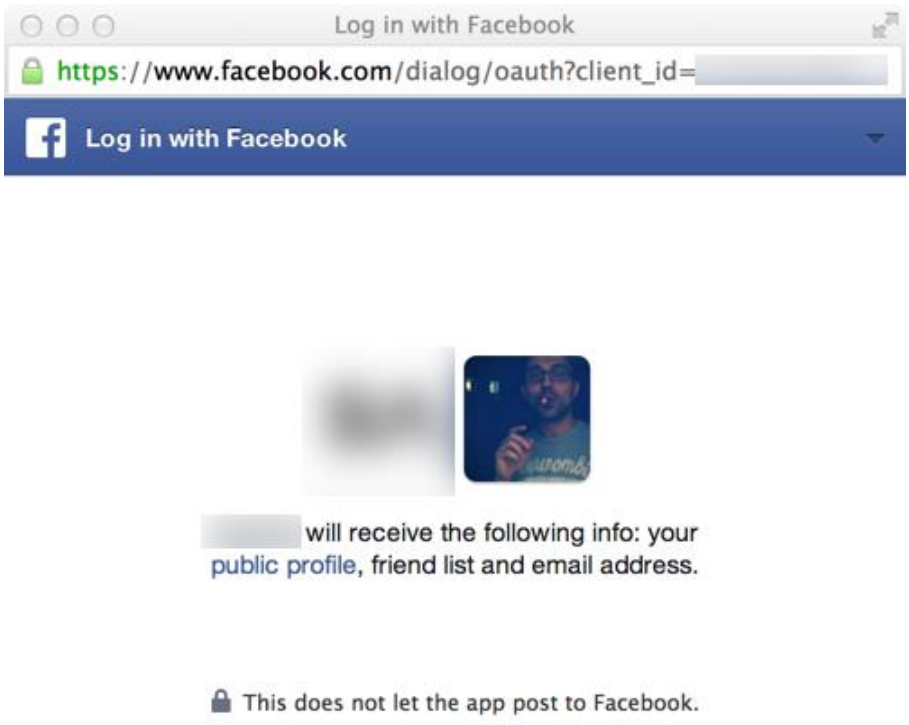

Figure 43: Facebook authorization request

This screen is the **Authenticate and Authorize Client** step in the diagram in Figure 42.

Setting Up OWIN

Now that it is clear what OAuth is and how it works, it is time to set up your OWIN application.

Note: The code for the rest of the explanation is built on top of the sample created in the Form Authentication section of this chapter.

Because the authentication comes from an external application (the social network), it is important to configure our middleware to accept external cookies for authentication.

```
app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);
```

Code Listing 51

Now the application is almost ready, and we need to configure the desired social media and the right endpoint (MVC actions) to manage communication between our application and the authentication server.

The last part is the same for all social authentications because all of them are based on the same protocol (OAuth precisely). We will explain this final part further in the OAuth Token Validation section.

Note: Twitter, Facebook, and Google sections are very similar. If you don't need to implement all these providers, you can choose your favorite and jump to the OAuth Token Validation section.

Twitter Authentication

Twitter, along with Facebook, is one of the most popular social networks, especially if you are a developer. As such, it offers the opportunity to use its authentication in your application.

Register Your Application

As we saw in the previous section, OAuth requires registering with the authorization server. To use Twitter's authentication, we need to go to the Twitter application management site at apps.twitter.com and follow the steps to register an application.

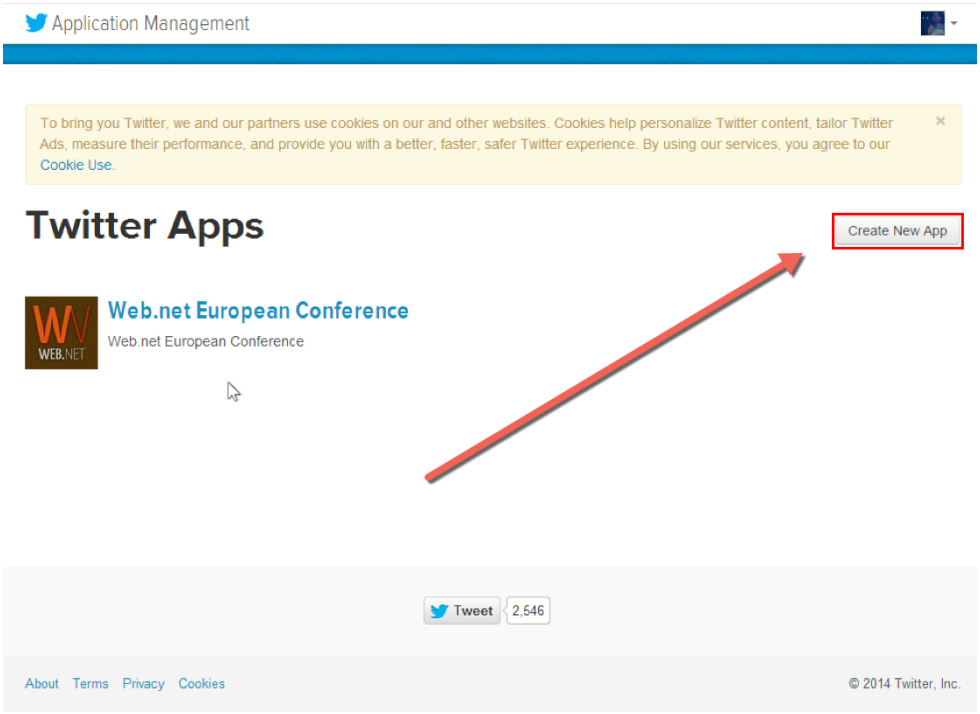

Figure 44: Create new application

With OWIN, you don't have to specify the Callback URL because it uses the default value **`signin-twitter`**.

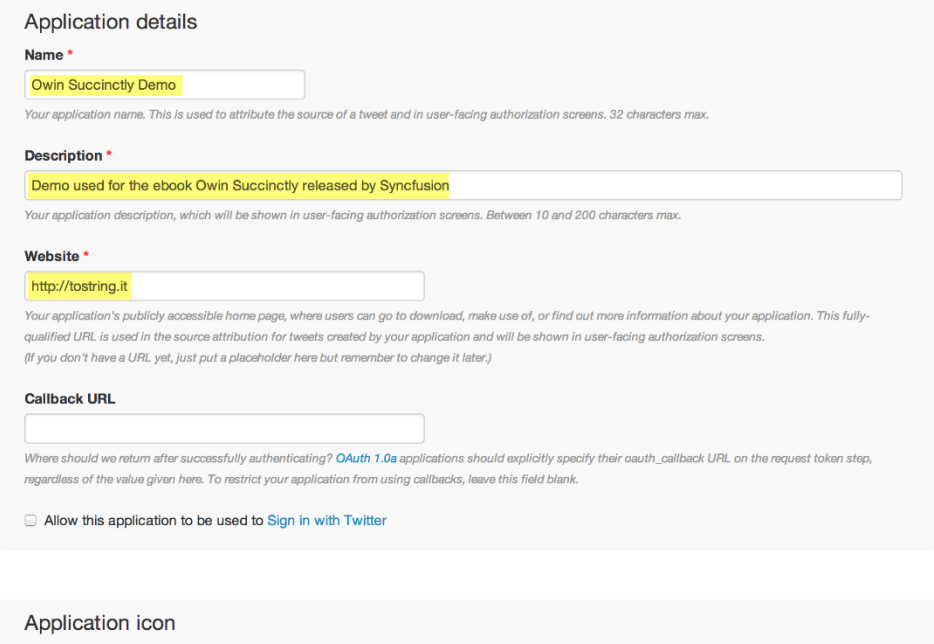

Figure 45: Register application form

To complete the integration with Twitter, we need the API key and the API secret for the application we just registered. Open the **Keys and Access Tokens** tab.

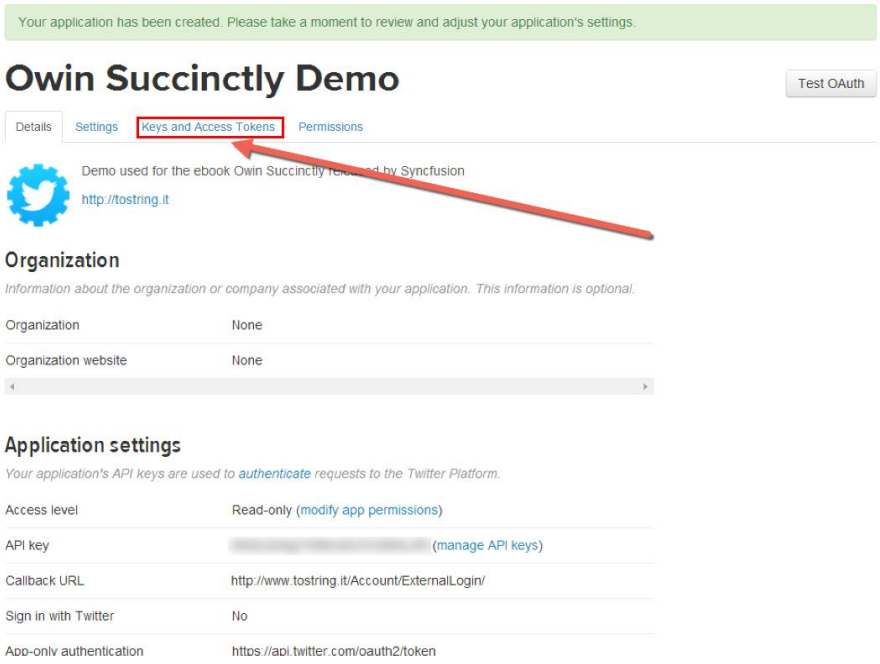

Figure 46: Registered application summary

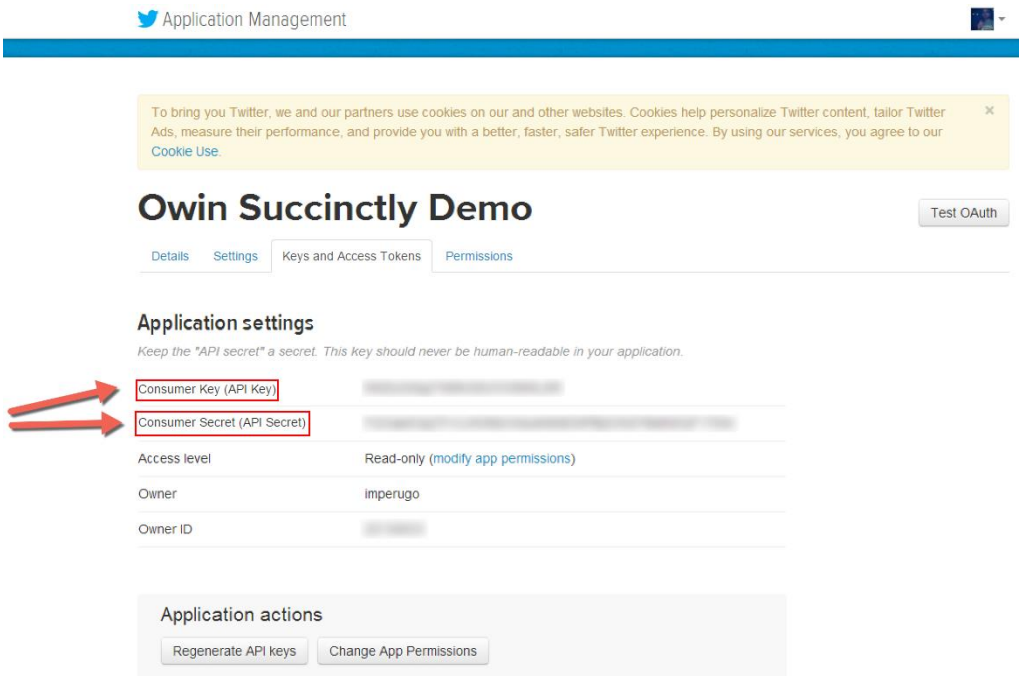

Figure 47: Register application API

Note: By default, a Twitter app is read-only. This means your application can't tweet for the user but can only read data. If you want to do more, you can change this by clicking the** Change App Permissions **button.

Now that we have all the ingredients ready, we can go back to writing code on our application and add the OWIN Twitter package, **`Microsoft.Owin.Security.Twitter`**.

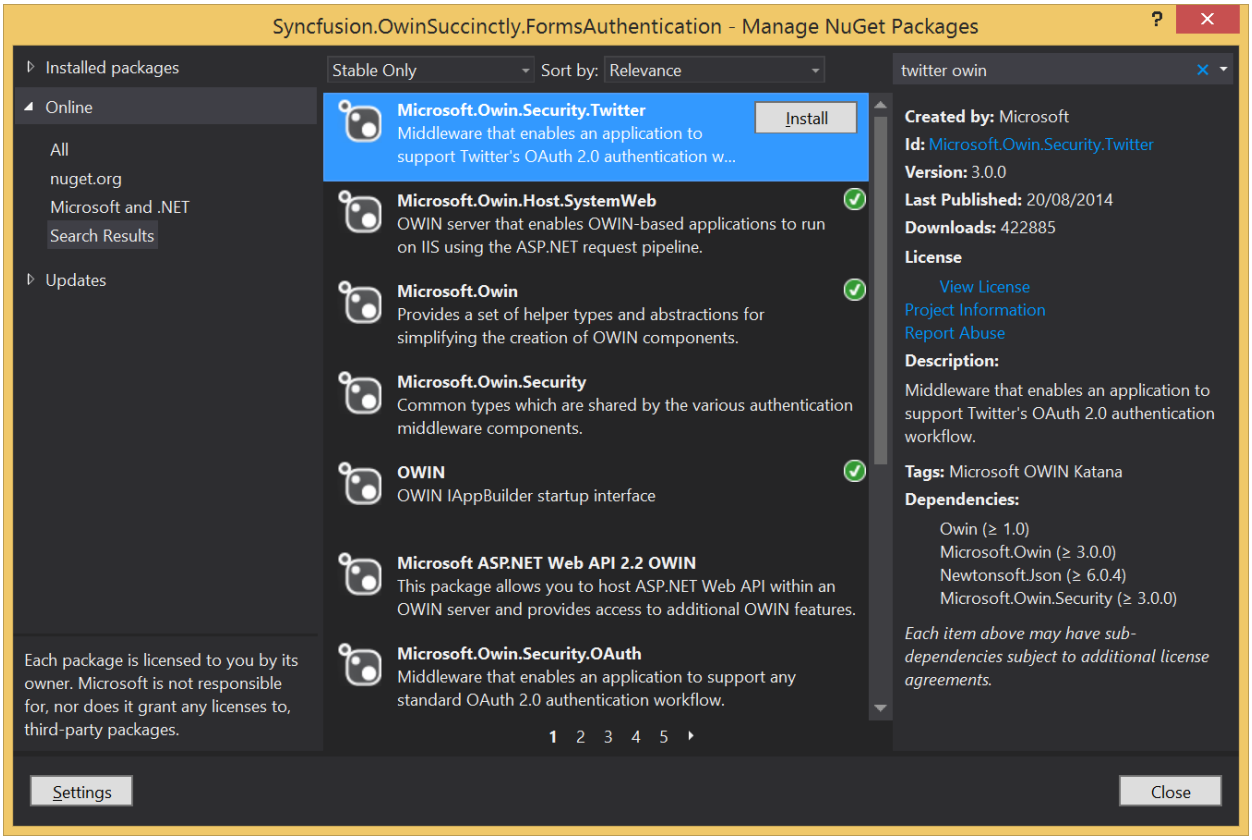

Figure 48: NuGet Package Manager with Twitter-related OWIN package

Finally, paste your API keys in the OWIN configuration file.

```
app.UseTwitterAuthentication(
    consumerKey: "my-api-key",
    consumerSecret: "my-api-secret");
```

Code Listing 52

Now the OWIN part is complete. We don't have to write more code here (that's why OWIN is so cool). The next step is to write the necessary code to manage tokens, cookies, and all the other stuff required by OAuth. We will cover this part in the OAuth Token Validation section.

Facebook Authentication

Facebook authentication is not so different from Twitter because it is based on the same protocol, so we'll follow the same steps for Twitter but on the Facebook website.

Register your Application

Because OAuth requires registering with the authorization server, we need to go to the Facebook Developer Website at developers.facebook.com/apps and register our application to obtain the necessary keys to integrate the login between the two applications.

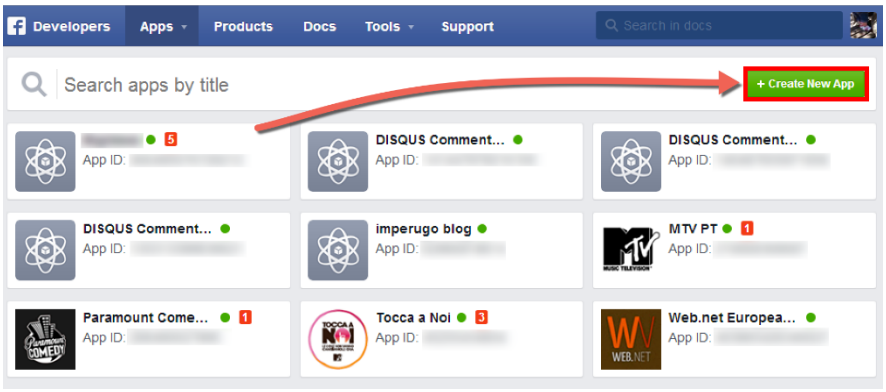

Figure 49: Register a new Facebook app

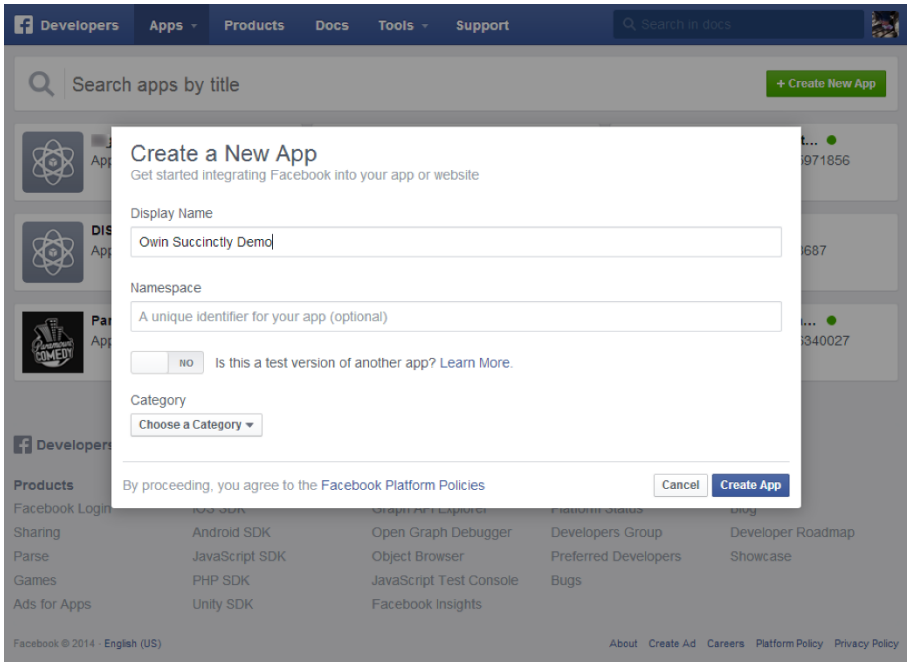

Figure 50: Creating a new application

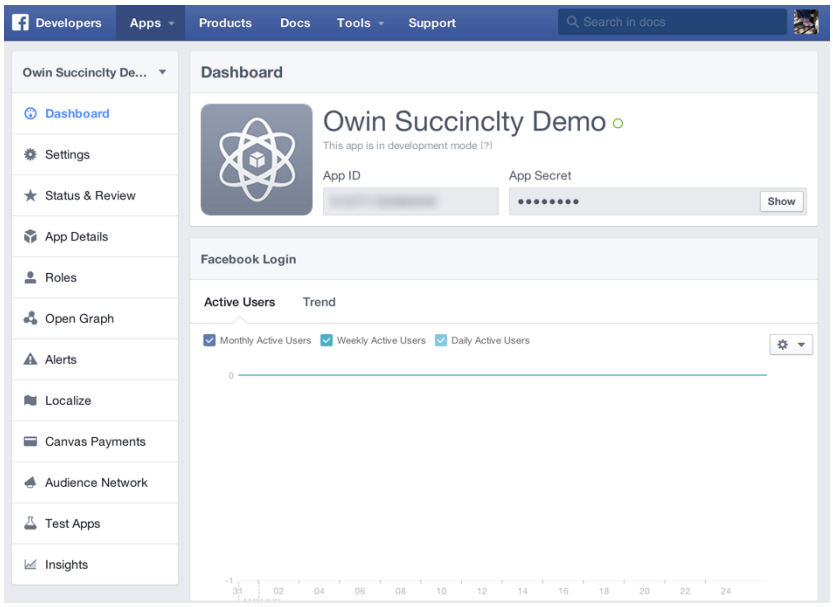

Figure 51: Registered application summary

Now that we have created our application, we have to specify the allowed domains. In our case it will be *localhost*, but you also need to specify your production domain. The steps are demonstrated in the following figures.

After creating the app on the Facebook developer site, click **Settings** on the left side of the page and then click the **Add Platform** button.

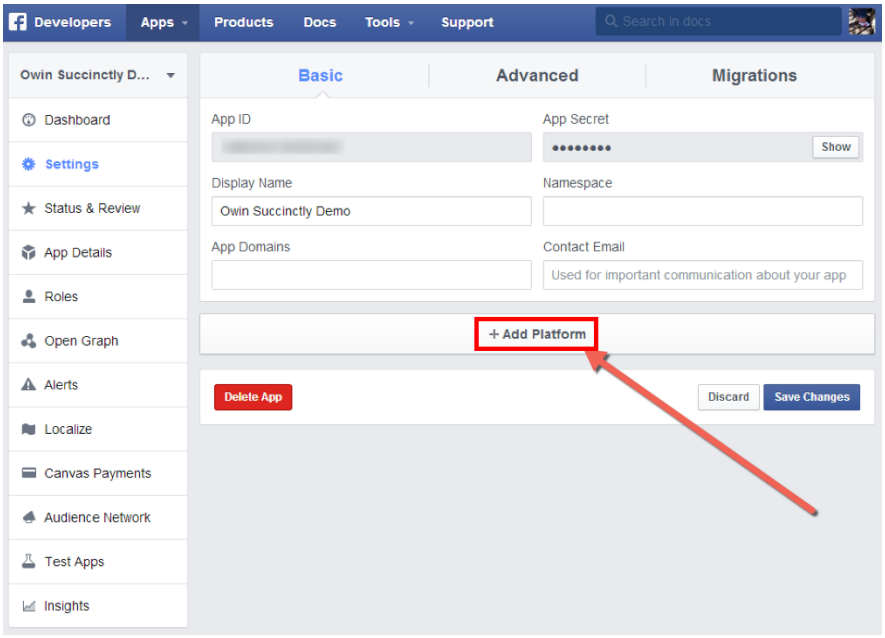

Figure 52: Application settings

Select the **Website** option.

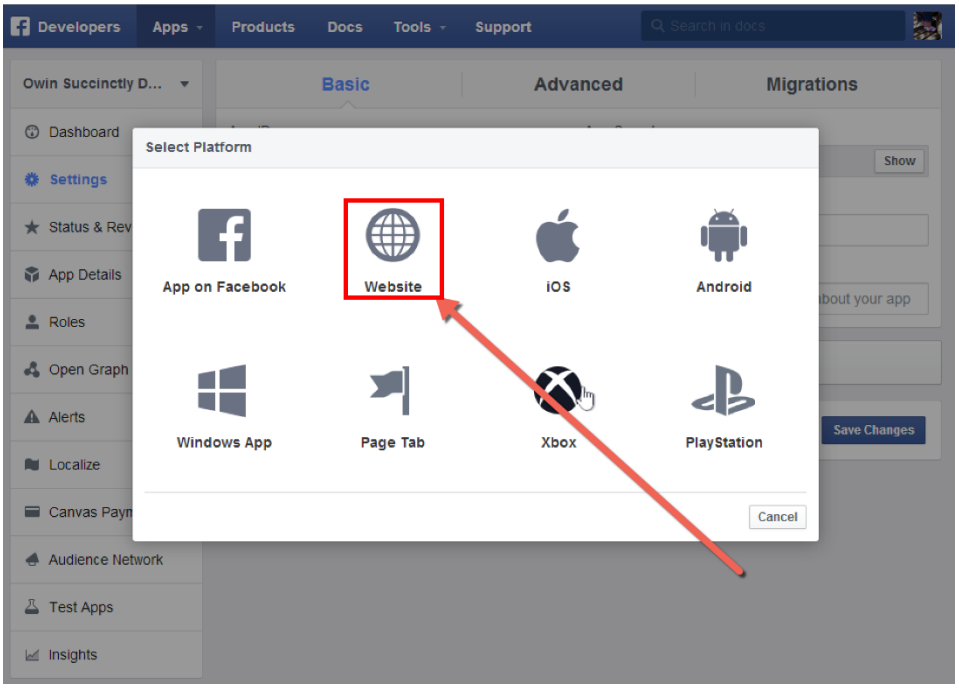

Figure 53: Adding a new platform

Specify **`localhost`** as the **App Domain** and the **Site URL**.

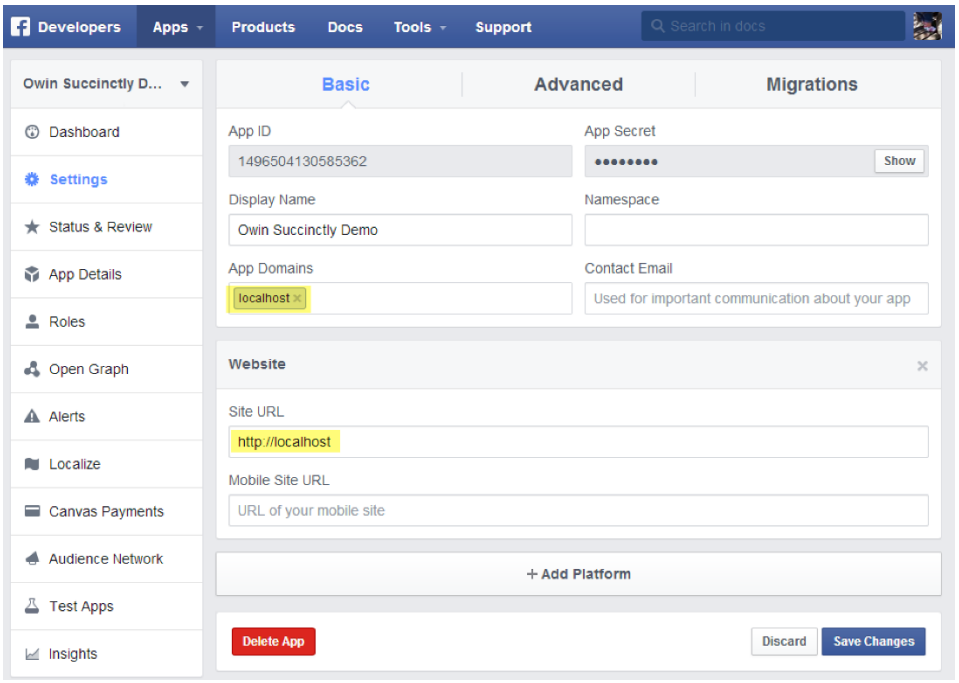

Figure 54: Application settings

Finally, everything is ready and we can register the **App ID** and **App Secret** into OWIN, but first, it is necessary to install the right OWIN Facebook package from NuGet, **`Microsoft.Owin.Security.Facebook`**.

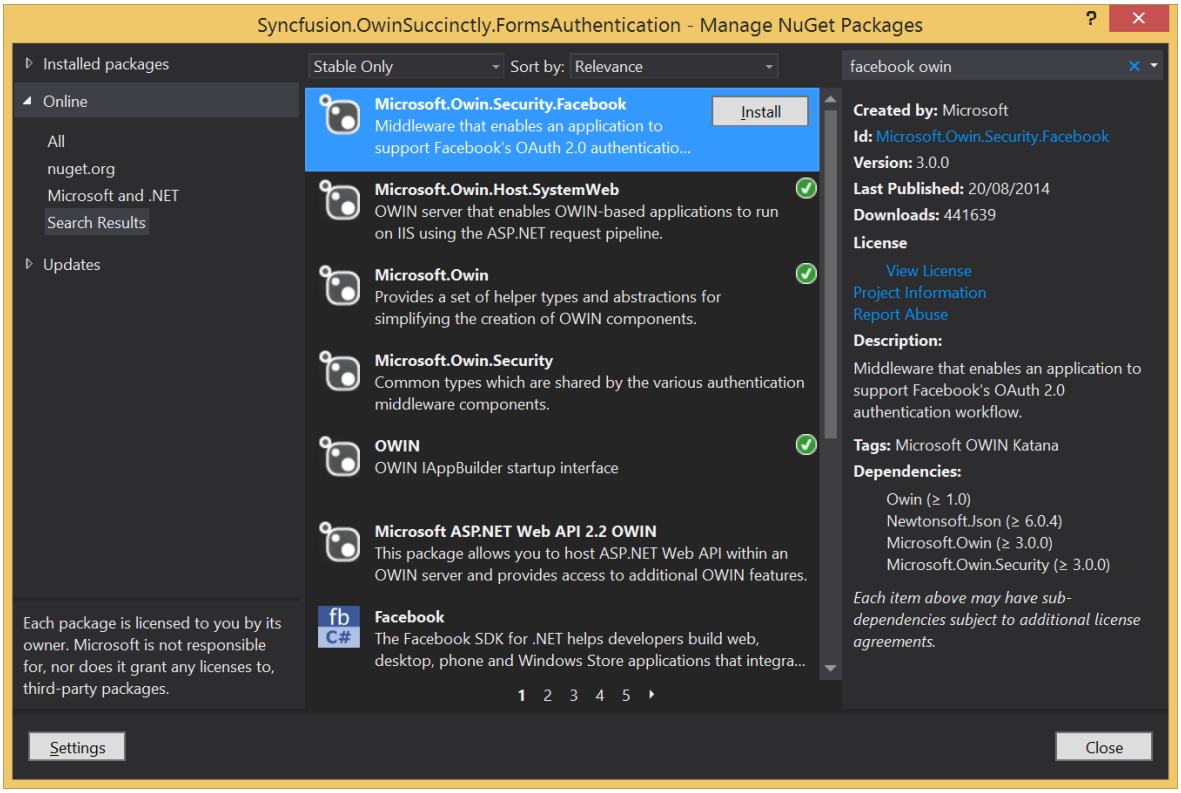

Figure 55: Manage NuGet packages with Facebook-related OWIN package

As you did for Twitter authentication, specify the **`appId`** and **`appSecret`** in the OWIN **`Startup`** class.

```
app.UseFacebookAuthentication(
    appId: "my-app-id",
    appSecret: "my-app-secret");
```

Code Listing 53

We are done. Facebook integration with OWIN is now complete, but it is necessary to manage the cookies between the requests. We will cover that part in the OAuth Token Validation section.

Google Authentication

Until a few months ago, Google was offering the opportunity to use its authentication base on different protocols:

- Open ID 2.0
- OAuth 1.0
- OAuth 2.0

For security reasons, it is no longer possible to use the first two types of authentication, so we have to do the same thing we did with Facebook and Twitter but on the Google Developers Console.

Create Your Project

Google uses a different naming for the registration process. In fact, here you are not registering your application, but instead creating a project.

Go to the Google Developers Console at console.developers.google.com/project and create a new project.

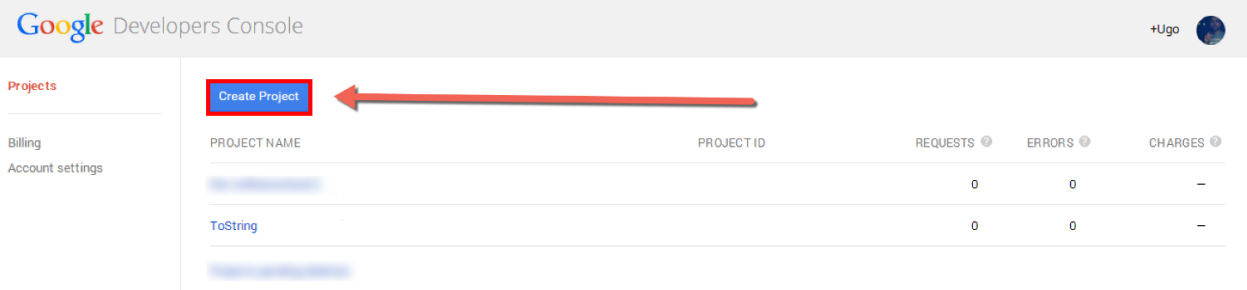

Figure 56: Create Project

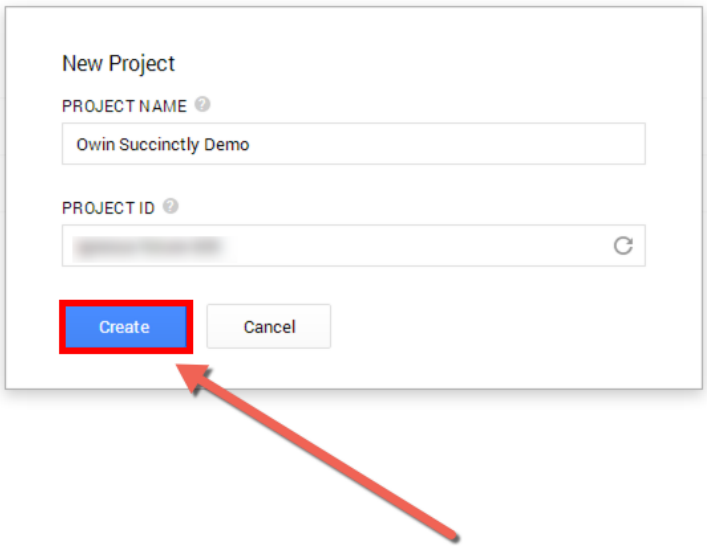

Figure 57: New Project form

After a few seconds, the project should be created and we can manage our client credentials. Click the **Credentials** option on the left side of the screen.

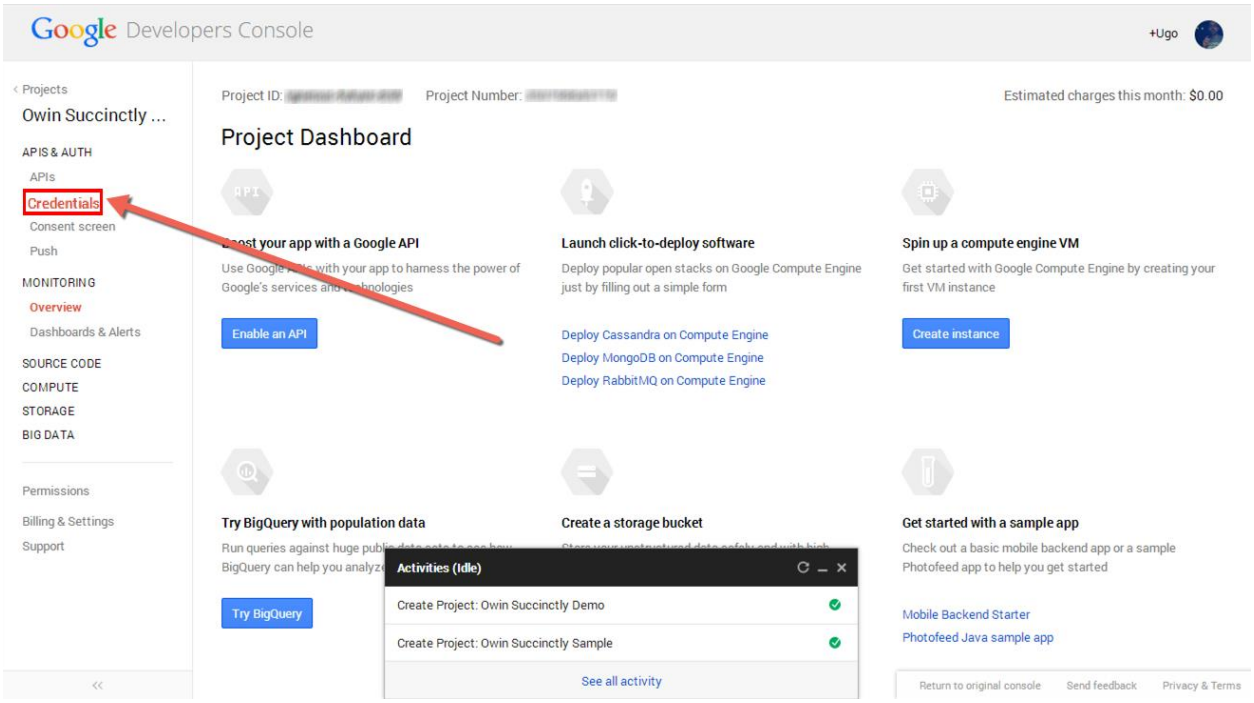

Figure 58: Credentials settings

Under the OAuth header, click the **Create new Client ID** button.

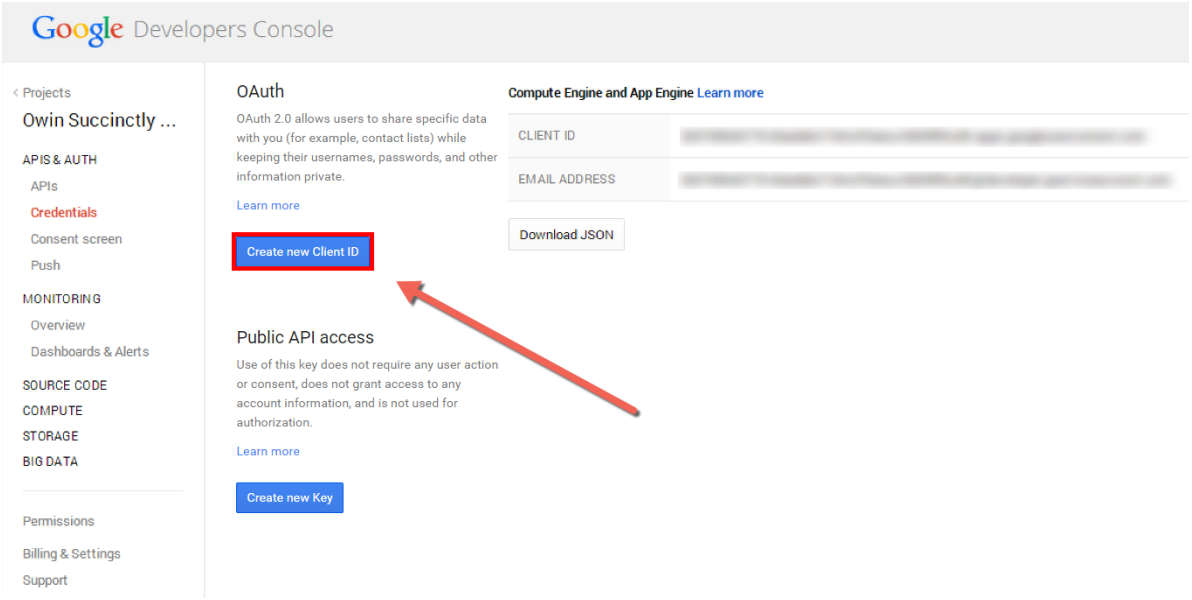

Figure 59: Create new client ID

Specify **`/signin-google`** as the redirect URI. OWIN manages callback URLs automatically, with different callbacks per social authentication.

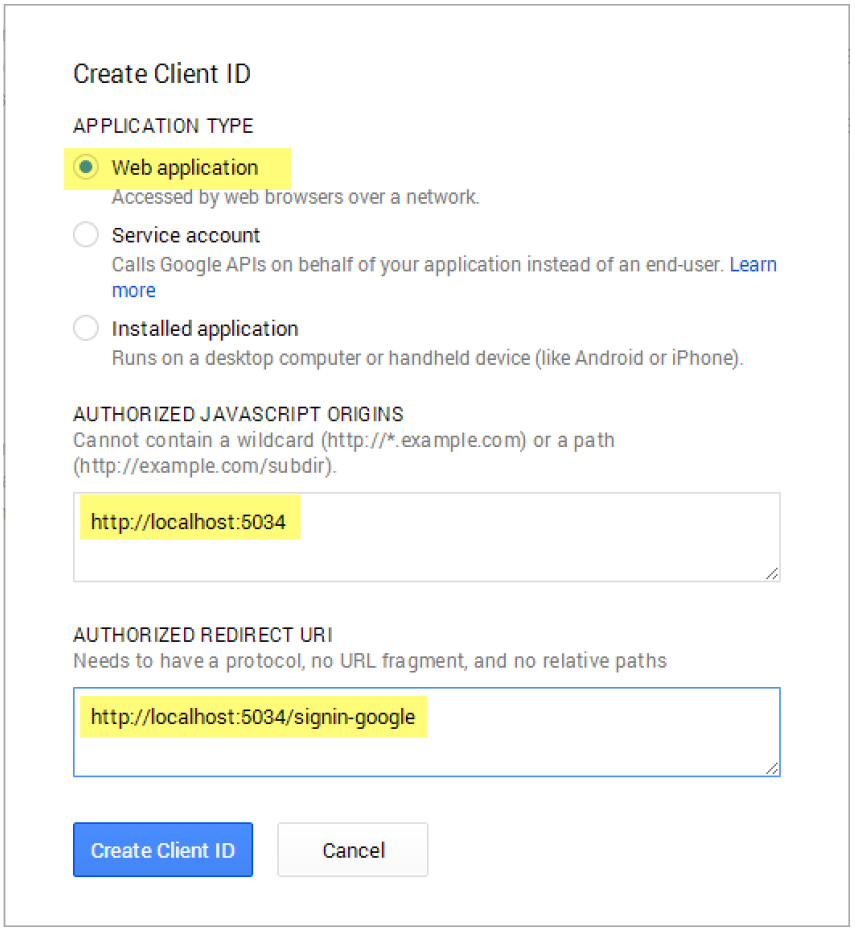

Figure 60: Create Client ID form

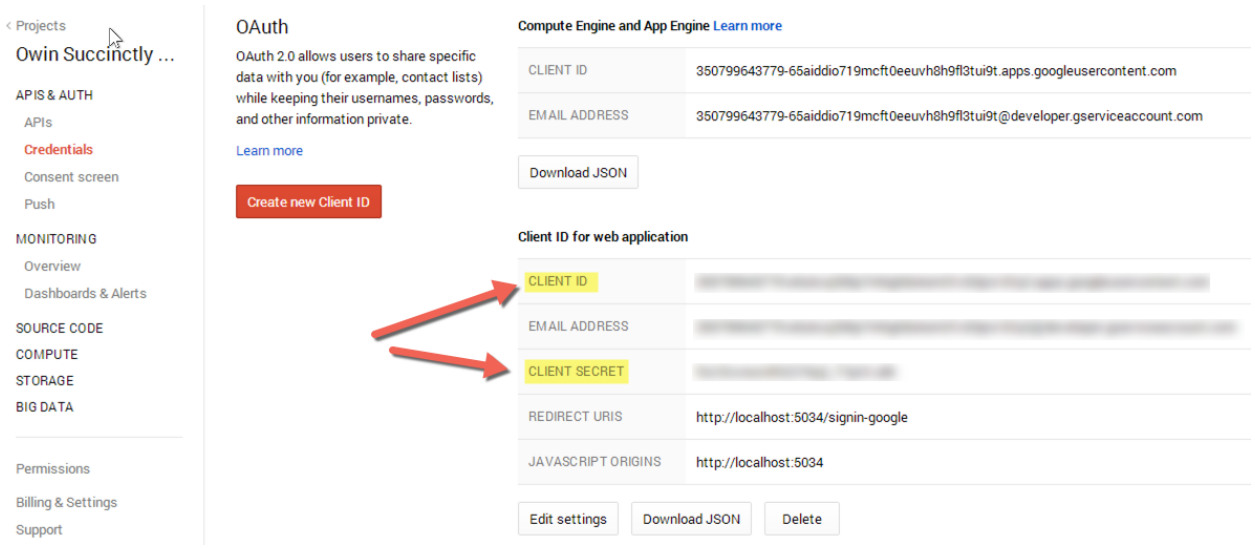

Figure 61: Credentials

Now that your project is registered for Google authentication, install the Google authentication provider package via NuGet.

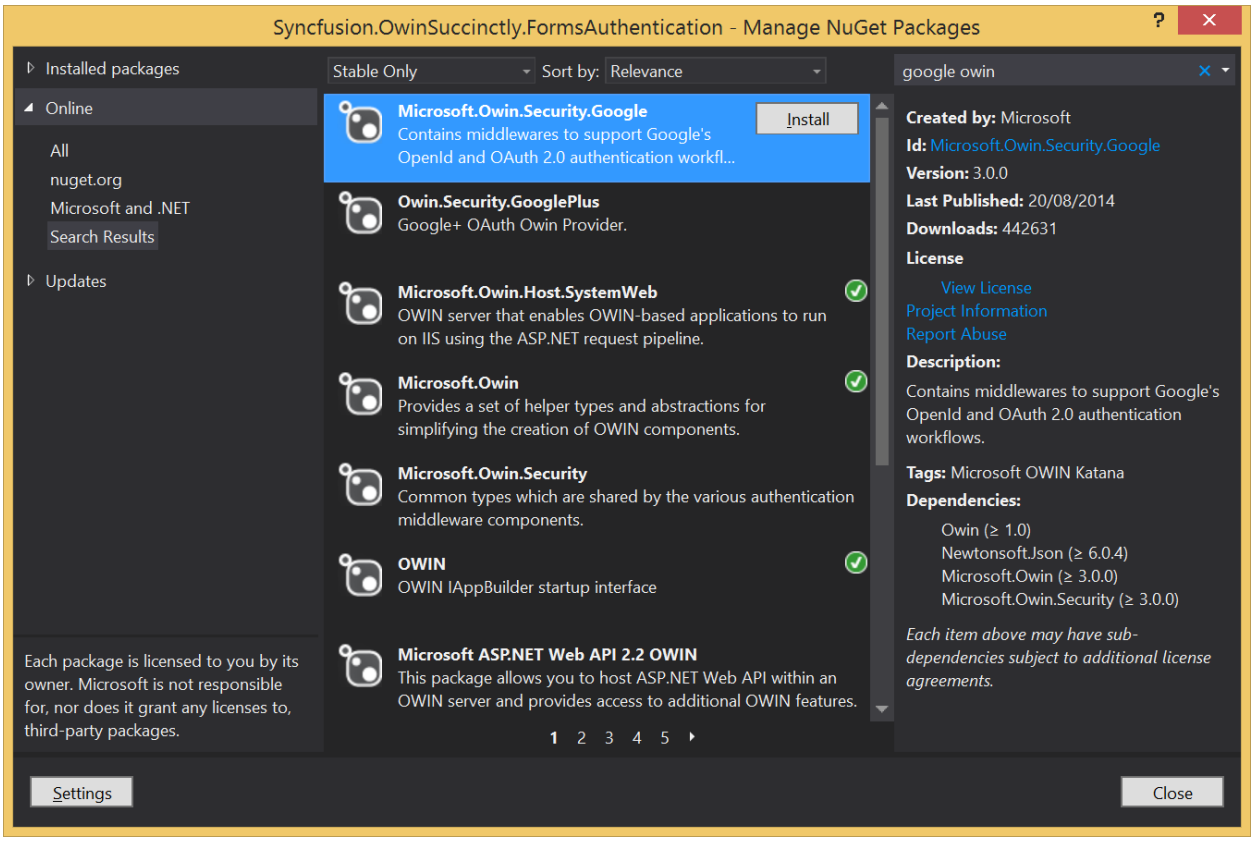

Figure 62: Google Security NuGet package

Finally, copy the **Client ID** and **Client Secret** values into the OWIN **`Startup`** class.

```
app.UseGoogleAuthentication(
    clientId: "my-client-id",
    clientSecret: "my-client-secret");
```

Code Listing 54

The OWIN Google integration is done, but let's see how to let your users log in via their social network accounts.

OAuth Token Validation

Now that all social media sites are ready to communicate with our application, it is important to correctly configure the application to accept their callbacks.

Note: The code we are going to see in this section is part ASP.NET MVC code and ASP.NET Identity. Nothing here is OWIN, but it is necessary to complete the integration. Moreover, there are several ways to create this part. ASP.NET Identity is not the only one, but it is suggested.

Show Login Buttons on Login Page

Of course, we have to show the right login buttons on the login page. This means not only the classic login form, but also our social authentication integration. The goal here is to create a "Log in with…" button for each social network we registered in our application.

To do that, we have to retrieve all registered social authentications using a specific extension method for the **`HttpContextBase`** class that returns an **`OwinContext`** with all we need.

In our **`Login.cshtml`** file, we have to retrieve all the registered social authentications using the following code.

```
@{
    var loginProviders = Context.GetOwinContext()
        .Authentication.GetExternalAuthenticationTypes();
}
```

Code Listing 55

Now we have to iterate over them and create the login button, but before doing it, it is important to understand why we can't simply put a link in our page.

When a user clicks on an authentication link on Twitter, for example, we have to send a set of information to specify to Twitter which application it is calling, the redirect URL, and other info. Part of this info must be specified in HTTP request headers, so we have to create a POST form, submit the provider info to our MVC action, and then redirect the user to the social network for the authentication.

Still working with our **`Login.cshtml`** file, the following code does just that.

```
@using (Html.BeginForm("ExternalLogin", "Account"))
{
@Html.AntiForgeryToken()
<div id="socialLoginList">
    <p>
        @foreach (AuthenticationDescription p in loginProviders)
        {
            <button type="submit" class="btn btn-default"
                    id="@p.AuthenticationType" name="provider"
                    value="@p.AuthenticationType"
                    title="Log in using your @p.Caption account">
```

```
                    @p.AuthenticationType
                </button>
            }
        </p>
    </div>
}
```

Code Listing 56

Until now, it hasn't been anything complex; we retrieved the registered providers, iterated the collection, and created a button. Each button submits to an action called **`ExternalLogin`** in the **`AccountController`** sending the authentication provider.

Now we have to get this information and redirect it to the social authentication. Open the **`AccountController`** we created in the Form Authentication section and add the following code.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult ExternalLogin(string provider, string returnURL)
{
    // Request a redirect to the external login provider
    return new ChallengeResult(
        provider,
        URL.Action(
            "ExternalLoginCallback",
            "Account",
            new { ReturnURL = returnURL }
            )
        );
}
```

Code Listing 57

The code is really simple. The action gets the provider and redirects the user to the login page, sending the callback URL (**`ExternalLoginCallback`**).

The unusual code for MVC developers is the class **`ChallengeResult`** that's not a part of MVC, Identity, or OWIN, so we have to create it.

```
internal class ChallengeResult : HttpUnauthorizedResult
{
    public ChallengeResult(string provider, string redirectUri)
        : this(provider, redirectUri, null)
    {
    }

    public ChallengeResult(string provider, string redirectUri, string userId)
    {
```

```
        LoginProvider = provider;
        RedirectUri = redirectUri;
        UserId = userId;
    }

    public string LoginProvider { get; set; }
    public string RedirectUri { get; set; }
    public string UserId { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        var properties = new AuthenticationProperties() { RedirectUri =
RedirectUri };
        if (UserId != null)
        {
            properties.Dictionary["XsrfId"] = UserId;
        }
        context.HttpContext.GetOwinContext().Authentication.Challenge(properties,
LoginProvider);
    }
}
```

Code Listing 58

All this is necessary to create the right redirect request. Now that we have the action ready, let's try the application.

If we did everything right, the login layout should result in the following in our application.

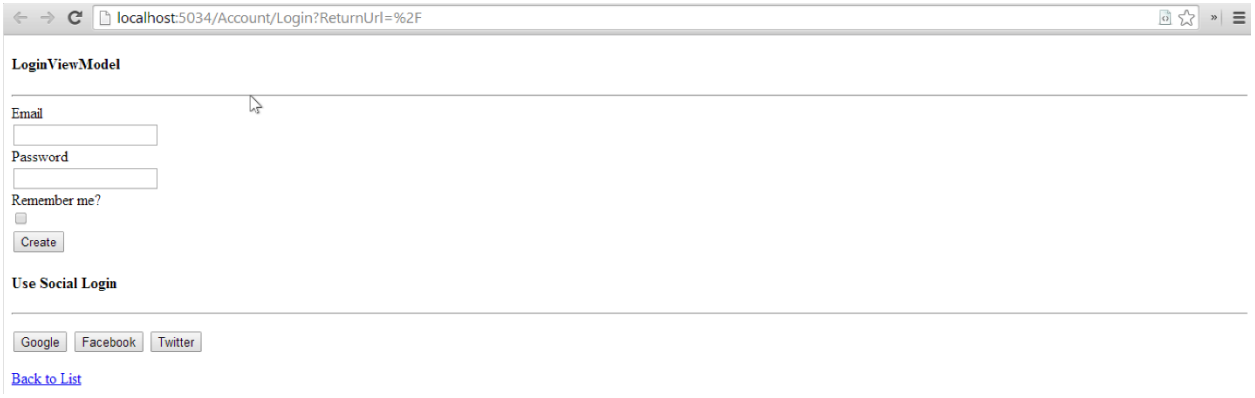

Figure 63: Login page

Clicking the **Twitter** button should redirect to Twitter page that asks to authorize the app.

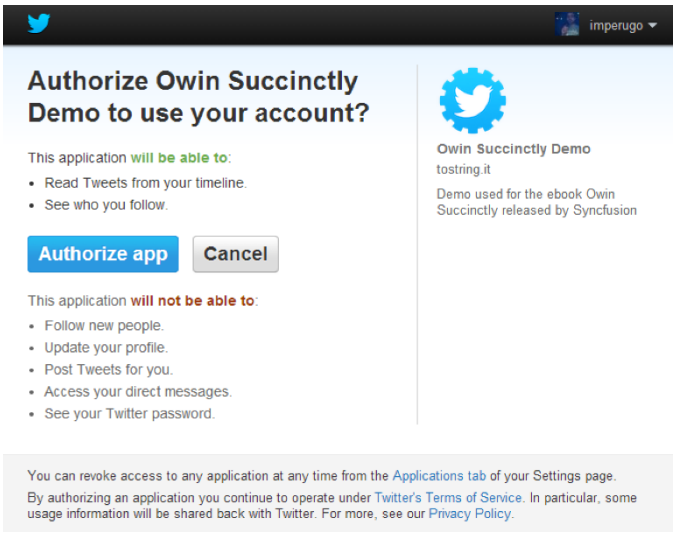

Figure 64: Twitter authorize app page

Next, after providing the right credentials, we should be sent back to our application where we will get an error.

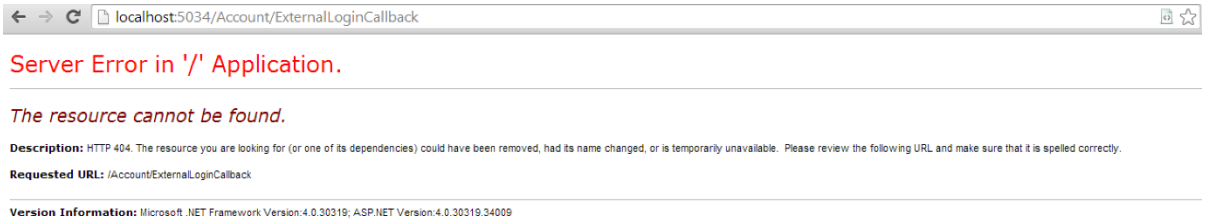

Figure 65: 404 error on calling callback page

No worries. That's okay. The social login is redirecting the user to **`Account/ExternalLoginCallback`** because we specified it before. Now we have to create the action to process the callback.

```
[AllowAnonymous]
public async Task<ActionResult> ExternalLoginCallback(string returnURL)
{
    var loginInfo = await AuthenticationManager.GetExternalLoginInfoAsync();
    if (loginInfo == null)
    {
        return RedirectToAction("Login");
    }


    //Verify if a user with loginInfo.Login exists in your database
    var user = new IdentityUser("1") { UserName = "imperugo" };

    if (user != null)
    {
```

```
        AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie);

        var manager = new UserManager<IdentityUser>(new IdentityStore());

        ClaimsIdentity identity = await manager.CreateIdentityAsync(user,
DefaultAuthenticationTypes.ApplicationCookie);
        AuthenticationManager.SignIn(new AuthenticationProperties(), identity);
        return this.RedirectToAction("Index", "Home");
    }
    else
    {
        // If the user does not have an account, then prompt the user to create an
account
        return View("ExternalLoginConfirmation", new
ExternalLoginConfirmationViewModel { Email = loginInfo.Email });
    }
}
```

Code Listing 59

By running the demo application again and logging in using one of the registered social networks, you should be able to see the protected page.

Of course, this code is only a demo. In a final application you have to implement your login persistence, but the demo works and allows you to test the login integration easily.

Conclusions

In this chapter, you have learned how to manage authentication using OWIN and Katana with different providers, starting with the classic form authentication and trying the most common social networks like Twitter, Facebook, and Google.

NuGet offers other providers for Microsoft, Yahoo, and many others, so browse NuGet to see what else is out there.

Appendix

NuGet Packages

Katana is released by Microsoft via NuGet, but given the very granular nature of Katana, there are a lot of those packages. If you search NuGet for "Microsoft.Owin," you get more than 200 results. This appendix is a list of the most important official Katana packages, with a short description of what they provide.

Table 7: Katana NuGet Packages

Package Name	Description
`Owin`	Contains the base OWIN class.
`Microsoft.Owin`	Helpers and additional abstractions for simplifying development.
`Microsoft.Owin.Host.SystemWeb`	OWIN server for running OWIN-based applications on IIS through the ASP.NET pipeline.
`Microsoft.Owin.Hosting`	Provides infrastructural components for hosting OWIN-based applications.
`Microsoft.Owin.SelfHost`	Components to run OWIN-based applications inside a custom process.
`Microsoft.Owin.Host.HttpListener`	OWIN server built with .NET core networking class, **`HttpListener`**.

Package Name	Description
OwinHost	Standalone executable to host OWIN-based applications.
Microsoft.Owin.Diagnostics	Middleware components to help development, providing an enhanced error page and a Welcome page.
Microsoft.Owin.StaticFiles	Middleware components to enable serving of static files, both physical and in embedded resources.
Microsoft.AspNet.WebApi.Owin	Middleware component to host ASP.NET Web API application within an OWIN application.
Microsoft.AspNet.WebApi.OwinSelfHost	Component to host ASP.NET Web API inside a custom process.
Microsoft.AspNet.SignalR	Strictly not an OWIN-related package, but built on OWIN.
Microsoft.AspNet.SignalR.SystemWeb	Component to run SignalR on top of the System.Web-based OWIN server.
Microsoft.AspNet.SignalR.SelfHost	Component to host SignalR inside a custom process.
Microsoft.Owin.Security	Common types shared among all other middleware components.

Package Name	Description
Microsoft.Owin.Security.OAuth	Middleware that enables support to any standard OAuth 2.0 authentication workflow.
Microsoft.Owin.Security.Twitter	Middleware that enables support to Twitter's OAuth 2.0 authentication workflow.
Microsoft.Owin.Security.Facebook	Middleware that enables support to Facebook's OAuth 2.0 authentication workflow.
Microsoft.Owin.Security.Google	Middleware that enables support to Google's OpenId and OAuth 2.0 authentication workflows.
Microsoft.Owin.Security.MicrosoftAccount	Middleware that enables support to the Microsoft Account authentication workflow.
Microsoft.Owin.Security.Cookes	Middleware that enables support to cookie-based authentication, like the ASP.NET forms authentication.
Microsoft.Owin.Security.ActiveDirectory	Middleware that enables support to Microsoft's Active Directory authentication.
Microsoft.Owin.Security.WindowsAzure	Middleware that enables support to Microsoft Azure Active Directory authentication.