
ASP.NET 5 Documentation

Release 0.0.1

Microsoft

September 02, 2015

1	Frameworks	3
2	Topics	5
2.1	Getting Started	5
2.2	Tutorials	13
2.3	Conceptual Overview	42
2.4	Fundamentals	62
2.5	.NET Execution Environment (DNX)	119
2.6	Working with Data	128
2.7	Publishing and Deployment	128
2.8	Client-Side Development	139
2.9	Mobile	228
2.10	Security	228
2.11	Performance	313
2.12	Migration	313
2.13	Contribute	314
3	Related Resources	321
4	Contribute	323

Note: This documentation is a work in progress. Topics marked with a are placeholders that have not been written yet. You can track the status of these topics through our public documentation [issue tracker](#). Learn how you can [contribute](#) on GitHub.

Frameworks

- MVC

2.1 Getting Started

2.1.1 Installing ASP.NET 5 On Windows

By [Steve Smith](#)

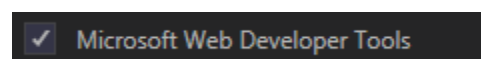
This article describes how to install ASP.NET 5 on Windows, showing both standalone installation as well as installation with Visual Studio 2015.

In this article:

- *[Install ASP.NET with Visual Studio](#)*
- *[Install ASP.NET Standalone](#)*

Install ASP.NET with Visual Studio

The easiest way to get started building application with ASP.NET 5 is to install the latest version of Visual Studio 2015 (including the freely available Community edition). Visual Studio is an Integrated Development Environment (IDE), which means it's not just an editor, but also many of the tools you need to build applications, in this case ASP.NET 5 web applications. When installing Visual Studio 2015, you'll want to be sure to specify that you want to install the Microsoft Web Developer Tools.



Once Visual Studio is installed, ASP.NET 5 is installed as well. You're ready to [build your first ASP.NET application](#).

Install ASP.NET Standalone

Visual Studio isn't the only way to install ASP.NET, and installing an IDE may not be appropriate in some scenarios. You can also install ASP.NET on its own from a command prompt. There are a few steps involved, since we'll need to install and configure the environment in which ASP.NET runs, known as the .NET Execution Environment (DNX). Before installing DNX, we need one more tool, the .NET Version Manager (DNVM).

Install the .NET Version Manager (DNVM)

The .NET Version Manager is used to install one or more versions of the .NET Execution Environment, and to manage which version is currently active. To install DNVM on Windows, you need to open a command prompt as an

Administrator, and run the following Powershell script:

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "&{$Branch='dev';iex ((new-object net.w
```

After the script has run, open a new command prompt and confirm DNVM is working by typing: `dnvm`

Assuming DNVM is configured correctly, you should see a result like this:

```
C:\>dnvm
You must specify a command!

.NET Version Manager v1.0.0-beta4-10338
By Microsoft Open Technologies, Inc.

usage: dnvm <command> [<arguments...>]

commands:
alias      Lists and manages aliases
help       Displays a list of commands, and help for specific commands
install    Installs a version of the runtime
list       Lists available runtimes
name       Gets the full name of a runtime
setup      Installs the version manager into your User profile directory
upgrade    Installs the latest version of the runtime and reassigns the spec
ified alias to point at it
use        Adds a runtime to the PATH environment variable for your current
shell

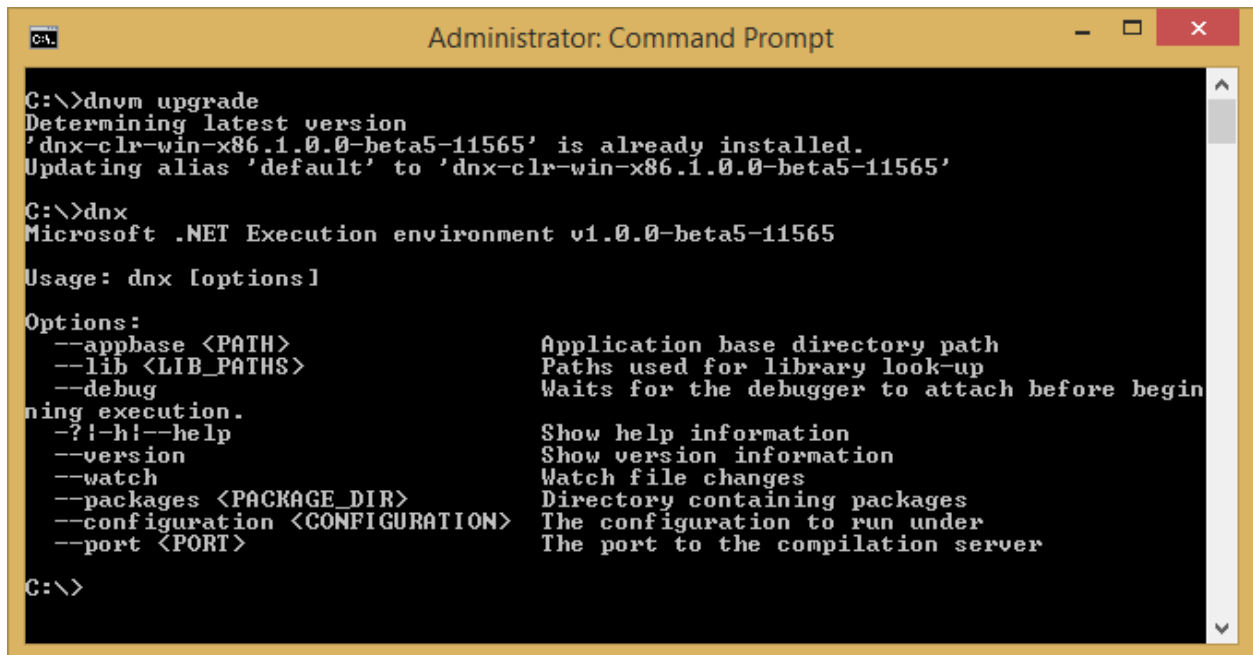
C:\>
```

Install the .NET Execution Environment (DNX)

To install the latest version of DNX using DNVM, run: `dnvm upgrade`

This command downloads the latest version of DNX and puts it on your user profile so that it is ready to use.

After this command completes, run: `dnx` to confirm DNX is configured correctly.



```
C:\>dnvm upgrade
Determining latest version
'dnx-clr-win-x86.1.0.0-beta5-11565' is already installed.
Updating alias 'default' to 'dnx-clr-win-x86.1.0.0-beta5-11565'

C:\>dnx
Microsoft .NET Execution environment v1.0.0-beta5-11565

Usage: dnx [options]

Options:
  --appbase <PATH>           Application base directory path
  --lib <LIB_PATHS>          Paths used for library look-up
  --debug                     Waits for the debugger to attach before begin
                             ning execution.
  -?|-h|--help               Show help information
  --version                  Show version information
  --watch                    Watch file changes
  --packages <PACKAGE_DIR>  Directory containing packages
  --configuration <CONFIGURATION> The configuration to run under
  --port <PORT>              The port to the compilation server

C:\>
```

Now that DNX is installed, you're ready to begin using ASP.NET 5!

Summary

You can install ASP.NET 5 on Windows either as a standalone installation, or as part of Visual Studio 2015. In either case, installation is straightforward, and once complete, you're ready to get [started building your first ASP.NET application](#).

Related Resources

- [Installing ASP.NET 5 on OS X](#)
- [Your First ASP.NET 5 Application Using Visual Studio](#)

2.1.2 Installing ASP.NET 5 On Mac OS X

By Steve Smith

ASP.NET 5 runs on the .NET Execution Environment (DNX), which is available on multiple platforms, including OS X. This article describes how to install DNX, and therefore ASP.NET 5, on OS X, using [Homebrew](#).

In this article:

- [*Install ASP.NET 5 on OS X*](#)

Install ASP.NET 5 on OS X

Install Mono

Currently, ASP.NET 5 on OS X requires the [Mono](#) runtime. Mono is an ongoing effort to port the .NET Framework to other platforms. It is one of the ways .NET applications can run on platforms other than Windows.

Using the Mono Installer You can install the latest version of Mono from the [Mono project downloads page](#), just select Mac OS X and click download. The latest version should be compatible with ASP.NET 5 (specifically, you need at least Mono 4.0.1).

Using Homebrew If you use the [Homebrew](#) package manager, you can install mono using the following command:

```
brew install mono
```

Installing DNVM

Next, you need to install the .NET Version Manager (DNVM). You can do so by running an automatic installation script we provide. If you'd rather do it manually, instructions for manual installation are also provided.

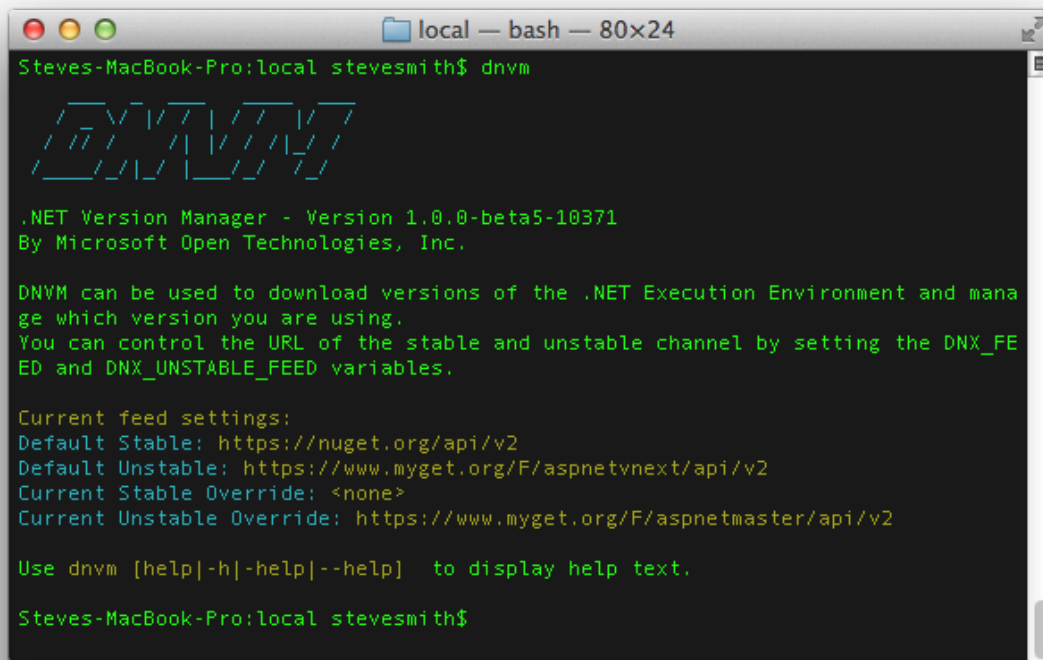
Automated Install Script To install DNVM using the automatic installations script, run the following command from your Terminal:

```
curl -sSL https://raw.githubusercontent.com/aspnet/Home/dev/dnvminstall.sh | sh && source ~/.dnx/dnvm
```

Manual Install If you'd rather install DNVM manually, you can download the `dnvm.sh` file from the [aspnet/Home](#) repository and place it in the `/Users/[your user name here]/.dnx/dnvm` directory on your file system. Then, add the following lines to your shell startup script (normally `/Users/[your user name here]/.bash_profile`, or `/Users/[your user name here]/.zshrc` if you are using ZSH):

```
[ -s "$HOME/.dnx/dnvm/dnvm.sh" ] && source "$HOME/.dnx/dnvm/dnvm.sh"
```

Verifying your installation You can verify that DNVM is installed properly by running `dnvm` in a Terminal window. If your shell does not recognize it `dnvm` as a command, run `source dnvm.sh` to load it, then try running `dnvm` again. You should see something like this:



```
Steves-MacBook-Pro:local stevesmith$ dnvm

  /-  \-  /-  \-  /-  \-  /-  \-
 /-  \-  /-  \-  /-  \-  /-  \-
/-  \-  /-  \-  /-  \-  /-  \-

.NET Version Manager - Version 1.0.0-beta5-10371
By Microsoft Open Technologies, Inc.

DNVM can be used to download versions of the .NET Execution Environment and manage which version you are using.
You can control the URL of the stable and unstable channel by setting the DNX_FEED and DNX_UNSTABLE_FEED variables.

Current feed settings:
Default Stable: https://nuget.org/api/v2
Default Unstable: https://www.myget.org/F/aspnetnext/api/v2
Current Stable Override: <none>
Current Unstable Override: https://www.myget.org/F/aspnetmaster/api/v2

Use dnvm [help|-h|--help] to display help text.

Steves-MacBook-Pro:local stevesmith$
```

Installing DNX

Once you have DNVM installed, you need to install the .NET Execution Environment (DNX). To install the latest version of DNX using DNVM, run:

```
dnvm upgrade
```

Now that DNX is installed, you're ready to begin using ASP.NET 5! Learn how you can [create a cross-platform console application](#) or a simple ASP.NET MVC application that runs within DNX.

Summary

ASP.NET 5 is built on the cross-platform .NET Execution Environment, which can be installed on OS X as well as Linux and Windows. Installing DNX and ASP.NET 5 on OS X takes just a few minutes, using a few Terminal commands.

Related Resources

- [Installing ASP.NET 5 on Windows](#)
- [Your First ASP.NET 5 Application Using Visual Studio](#)

2.1.3 Installing ASP.NET 5 On Linux

By Daniel Roth

ASP.NET 5 runs on the .NET Execution Environment (DNX), which is available on multiple platforms, including Linux. This article describes how to install DNX, and therefore ASP.NET 5, on Linux using Mono.

Note: You can also run DNX on Mac and Linux using .NET Core. .NET Core for Mac and Linux is still in the early stages of development, but if you want to experiment with it please follow along on [GitHub](#).

In this article:

- *Using Docker*
- *Installing on Debian, Ubuntu and derivatives*
- *Installing on CentOS, Fedora and derivatives*

Using Docker

Instructions on how to use the ASP.NET 5 Docker image can be found here:
<http://blogs.msdn.com/b/webdev/archive/2015/01/14/running-asp-net-5-applications-in-linux-containers-with-docker.aspx>

The rest of this section deals with setting up a machine to run applications without the Docker image.

Installing on Debian, Ubuntu and derivatives

The following instructions were tested using Ubuntu 14.04 and Mint 17.01

Install Mono

Mono is an ongoing effort to port the .NET Framework to other platforms. Mono is one of the ways .NET applications can run on platforms other than Windows. ASP.NET 5 requires a version of Mono greater than 4.0.1.

To install Mono:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 3FA7E0328081BFF6A14DA29AA6A199318D5D8464308571683A226633287F601E1
echo "deb http://download.mono-project.com/repo/debian wheezy main" | sudo tee /etc/apt/sources.list
sudo apt-get update
sudo apt-get install mono-complete
```

Install libuv

Libuv is a multi-platform asynchronous IO library that is used by the **KestrelHttpServer** that we will use to host our ASP.NET 5 web applications.

To build libuv you should do the following:

```
sudo apt-get install automake libtool curl
curl -sSL https://github.com/libuv/libuv/archive/v1.4.2.tar.gz | sudo tar xzfv - -C /usr/local/src
cd /usr/local/src/libuv-1.4.2
sudo sh autogen.sh
sudo ./configure
sudo make
```

```
sudo make install
sudo rm -rf /usr/local/src/libuv-1.4.2 && cd ~/
sudo ldconfig
```

Note: `make install` puts `libuv.so.1` in `/usr/local/lib`, in the above commands `ldconfig` is used to update `ld.so.cache` so that `dlopen` (see `man dlopen`) can load it. If you are getting `libuv` some other way or not running `make install` then you need to ensure that `dlopen` is capable of loading `libuv.so.1`.

Install the .NET Version Manager (DNVM)

Now let's get DNVM. To do this run:

```
curl -sSL https://raw.githubusercontent.com/aspnet/Home/dev/dnvminstall.sh | DNX_BRANCH=dev sh && so
```

Once this step is complete you should be able to run `dnvm` and see some help text.

Note: `dnvm` needs `unzip` to function properly. If you don't have it installed, run `sudo apt-get install unzip` to install it before installing a runtime.

Add NuGet sources

Now that we have DNVM and the other tools needed to run an ASP.NET 5 application you can configure additional NuGet package sources to get access to the dev builds of all the ASP.NET 5 packages.

The nightly package source is: <https://www.myget.org/F/aspnetvnext/api/v2/>

You specify your package sources through your `NuGet.Config` file.

Edit: `~/ .config/NuGet/NuGet.Config`

The `NuGet.Config` file should look something like the following

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="AspNetVNext" value="https://www.myget.org/F/aspnetvnext/api/v2/" />
    <add key="nuget.org" value="https://www.nuget.org/api/v2/" />
  </packageSources>
  <disabledPackageSources />
</configuration>
```

You should now be able to restore packages from both the official public feed on <https://nuget.org> and also from the ASP.NET 5 dev builds.

Installing on CentOS, Fedora and derivatives

Note: Installation steps for CentOS, Fedora and derivatives are not currently available but should be available soon. The commands are mostly the same, with some differences to account for the different package managers used on these systems. Learn how you can [contribute](#) on GitHub.

2.1.4 Choosing the Right .NET For You on the Server

By [Daniel Roth](#)

ASP.NET 5 is based on the [.NET Execution Environment \(DNX\)](#), which supports running cross-platform on Windows, Mac and Linux. When selecting a DNX to use you also have a choice of .NET flavors to pick from: .NET Framework (CLR), [.NET Core](#) (CoreCLR) or [Mono](#). Which .NET flavor should you choose? Let's look at the pros and cons of each one.

.NET Framework

The .NET Framework is the most well known and mature of the three options. The .NET Framework is a mature and fully featured framework that ships with Windows. The .NET Framework ecosystem is well established and has been around for well over a decade. The .NET Framework is production ready today and provides the highest level of compatibility for your existing applications and libraries.

The .NET Framework runs on Windows only. It is also a monolithic component with a large API surface area and a slower release cycle. While the code for the .NET Framework is [available for reference](#) it is not an active open source project.

.NET Core

.NET Core 5 is a modular runtime and library implementation that includes a subset of the .NET Framework. Currently it is feature complete on Windows, and in-progress builds exist for both Linux and OS X. .NET Core consists of a set of libraries, called "CoreFX", and a small, optimized runtime, called "CoreCLR". .NET Core is open-source, so you can follow progress on the project and contribute to it on [GitHub](#).

The CoreCLR runtime (Microsoft.CoreCLR) and CoreFX libraries are distributed via [NuGet](#). Because .NET Core has been built as a componentized set of libraries you can limit the API surface area your application uses to just the pieces you need. You can also run .NET Core based applications on much more constrained environments (ex. [Windows Server Nano](#)).

The API factoring in .NET Core was updated to enable better componentization. This means that existing libraries built for the .NET Framework generally need to be recompiled to run on .NET Core. The .NET Core ecosystem is relatively new, but it is rapidly growing with the support of popular .NET packages like JSON.NET, Autofac, xUnit.net and many others.

Developing on .NET Core allows you to target a single consistent platform that can run on multiple platforms. However, the .NET Core support for Mac and Linux is still very new and not ready for production workloads.

Please see [Introducing .NET Core](#) for more details on what .NET Core has to offer.

Mono

[Mono](#) is a port of the .NET Framework built primarily for non-Windows platforms. Mono is open source and cross-platform. It also shares a similar API factoring to the .NET Framework, so many existing managed libraries work on Mono today. Mono is not a supported platform by Microsoft. It is however a good proving ground for cross-platform development while cross-platform support in .NET Core matures.

Summary

The .NET Execution Environment (DNX) and .NET Core make .NET development available to more scenarios than ever before. DNX also gives you the option to target your application at existing available .NET platforms. Which

.NET flavor you pick will depend on your specific scenarios, timelines, feature requirements and compatibility requirements.

2.2 Tutorials

2.2.1 Your First ASP.NET 5 Application Using Visual Studio

By [Steve Smith](#)

ASP.NET 5 provides a host of improvements over its predecessors, including improved performance, better support for modern web development standards and tools, and improved integration between WebAPI, MVC, and WebForms. In addition, you can easily develop ASP.NET 5 applications using a variety of tools and editors, but Visual Studio continues to provide a very productive way to build web applications. In this article, we'll walk through creating your first ASP.NET 5 web application using Visual Studio 2015.

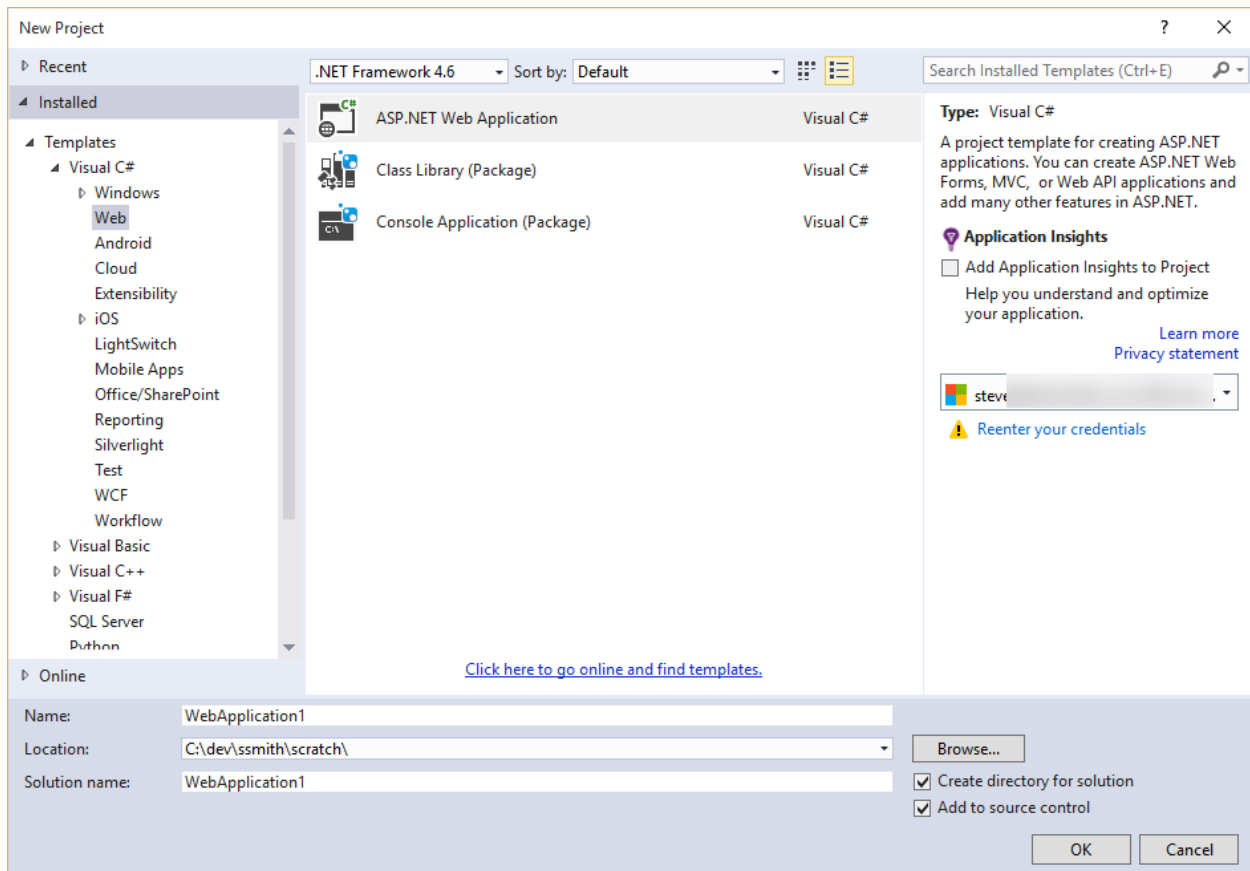
In this article:

- *Create a New ASP.NET 5 Project*
- *Running the Application*
- *Server-Side vs. Client-Side Behavior*
- *Adding Server-Side Behavior*
- *Adding Client-Side Behavior*

[View or download sample from GitHub.](#)

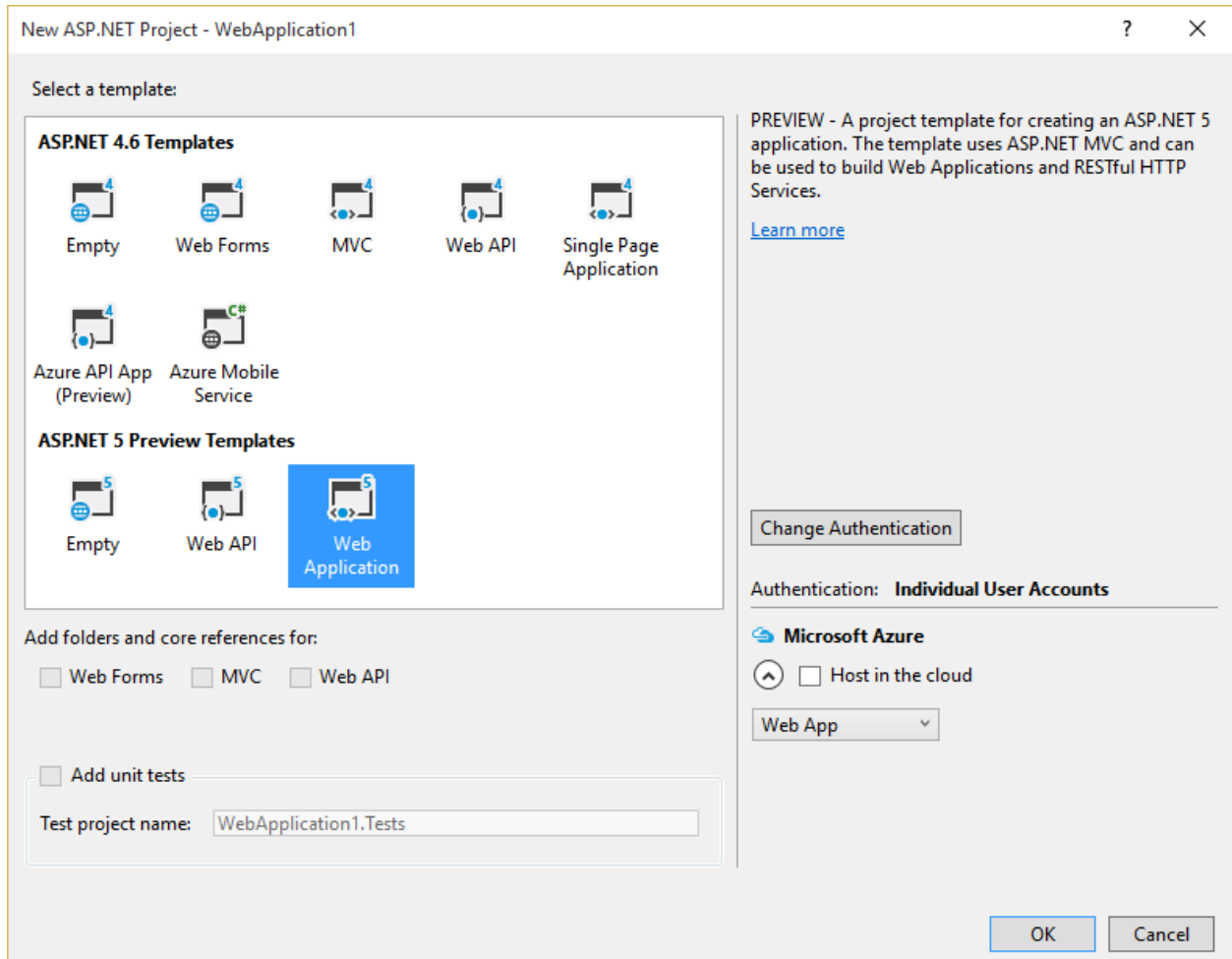
Create a New ASP.NET 5 Project

To get started, open Visual Studio 2015. Next, create a New Project (from the Start Page, or via File - New - Project). On the left part of the New Project window, make sure the Visual C# templates are open and “Web” is selected, as shown:



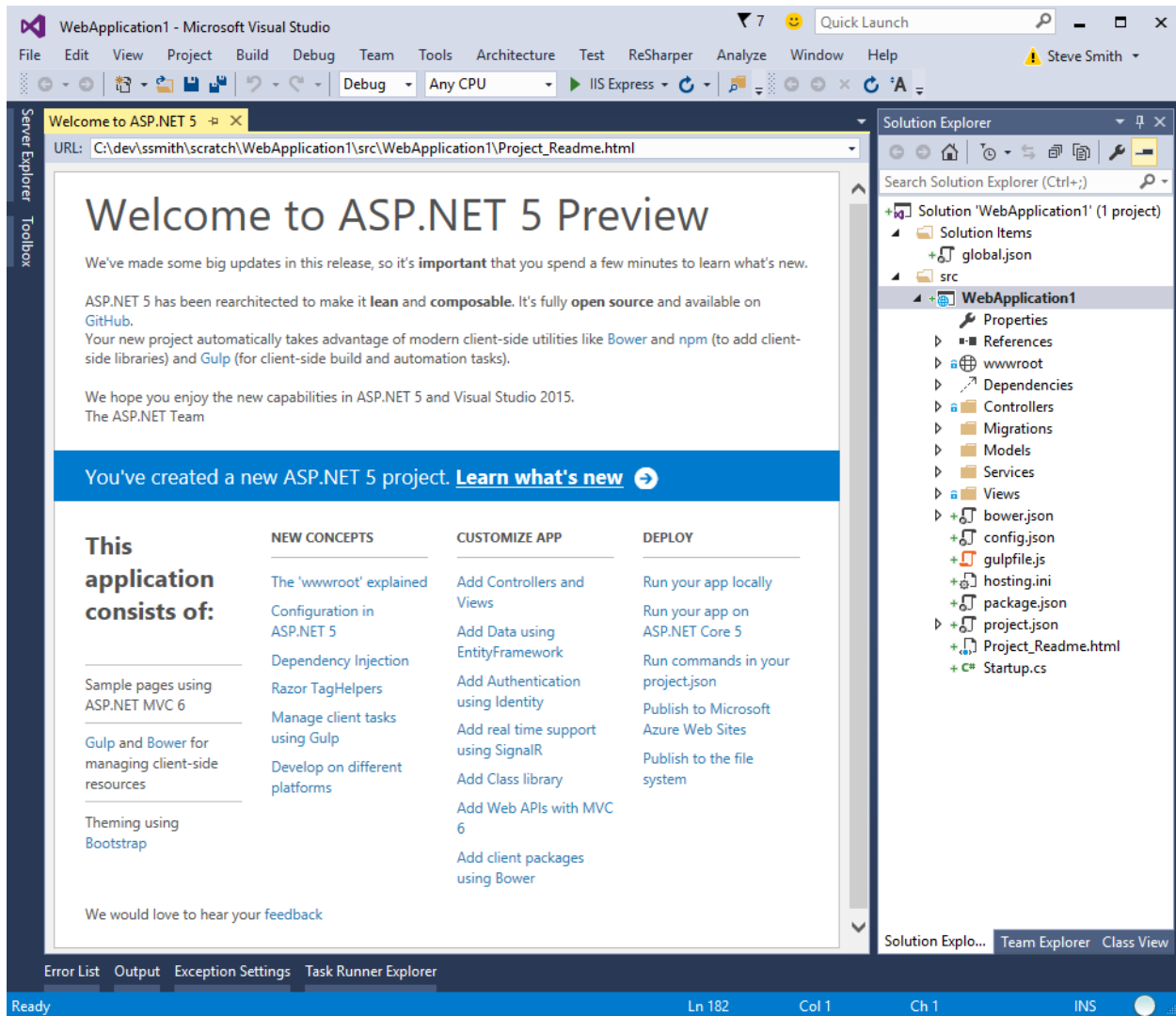
On the right, choose ASP.NET Web Application. Make sure the framework specified at the top of the window is .NET Framework 4.6. Enter a name and confirm where you would like the project to be created, and click OK.

Next you should see another dialog, the New ASP.NET Project window:



Select the Web Application from the set of ASP.NET 5 Preview templates. These are distinct from the ASP.NET 4.6 templates, which can be used to create ASP.NET projects using the previous version of ASP.NET. Note that you can choose to configure hosting in Microsoft Azure directly from this dialog by checking the box on the right. After selecting Web Application, click OK.

At this point, the project is created. If you are prompted to select a source control option, choose whichever you prefer (for this example, I've chosen Git). It may take a few moments for the project to load, and you may notice Visual Studio's status bar indicates that Visual Studio is downloading some resources as part of this process. Visual Studio ensures some required files are pulled into the project when a solution is opened (or a new project is created), and other files may be pulled in at compile time. Your project, once fully loaded, should look like this:



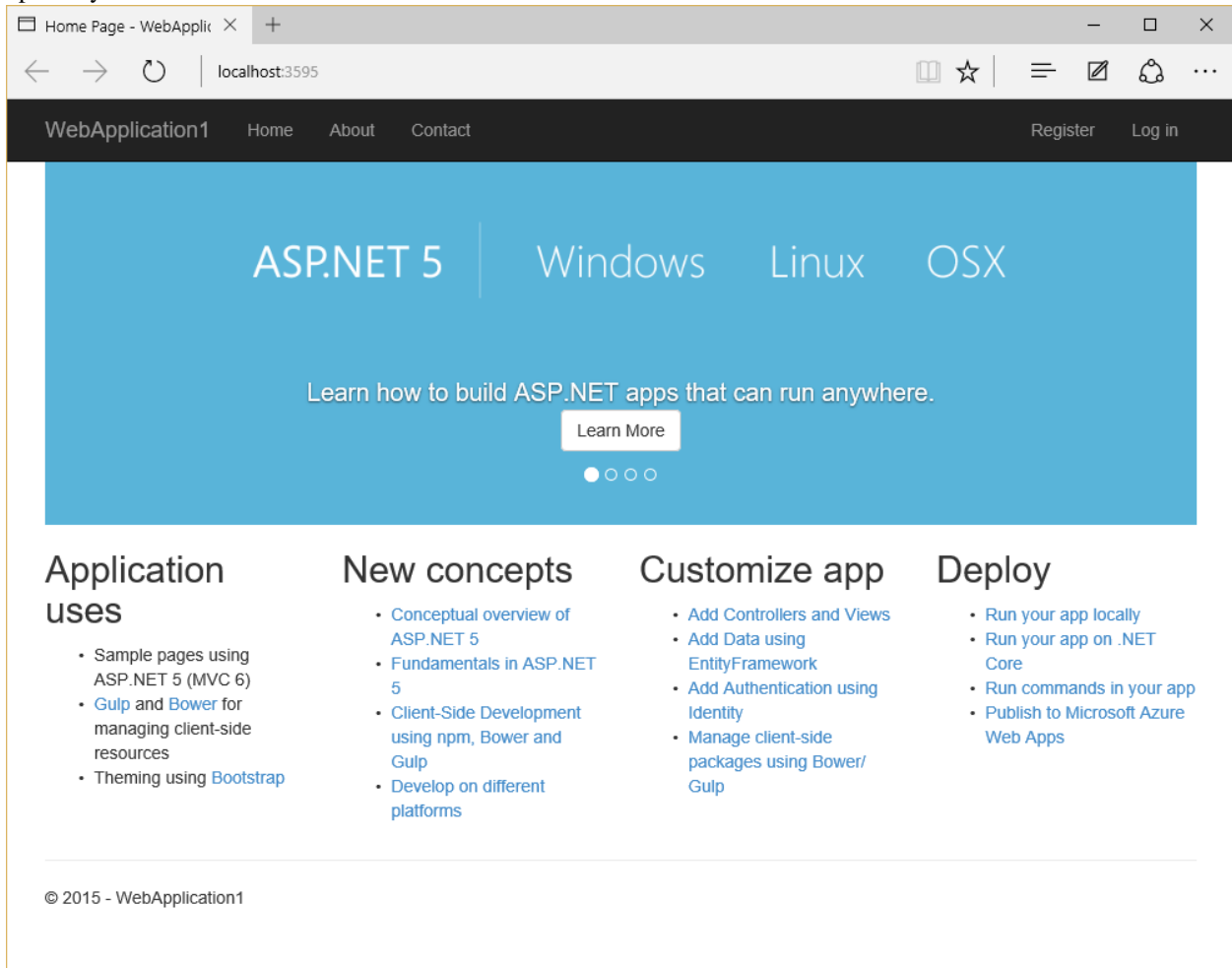
Looking at the Solution Explorer and comparing the elements with what we're familiar with in previous versions of ASP.NET, a few things stick out as being new and different. There's now a `wwwroot` folder, with its own icon. Similarly, there's a *Dependencies* folder **and** still a *References* folder - we'll discuss the differences between these two in a moment. Rounding out the list of folders, we have *Controllers*, *Models*, and *Views*, which make sense for an ASP.NET MVC project. This template also includes a *Services* folder, initially holding *MessageServices* used by ASP.NET Identity, and a *Migrations* folder, which holds classes used by Entity Framework to track updates to our model's database schema.

Looking at the files in the root of the project, we may notice the absence of a few files. `Global.asax` is no longer present, nor is `web.config`, both mainstays from the start of ASP.NET. Instead, we find a `Startup.cs` file and a `config.json` file. Adding to this mix are `bower.json`, `gulpfile.js`, `package.json`, and `project.json` (the `Project_Readme.html` file you can see in the browser tab). Clearly the success of Javascript in web development has had an effect on how ASP.NET 5 projects are configured, compiled, and deployed, with JavaScript Object Notation (JSON) files replacing XML for configuration purposes.

While we're at it, you may not notice it from the Solution Explorer, but if you open Windows Explorer you'll see that there is no longer a `.csproj` file, either. Instead you'll find an `.xproj` file, an MSBuild file that serves the same purpose from a build process perspective, but which is much simpler than its `csproj/vbproj` predecessor.

Running the Application

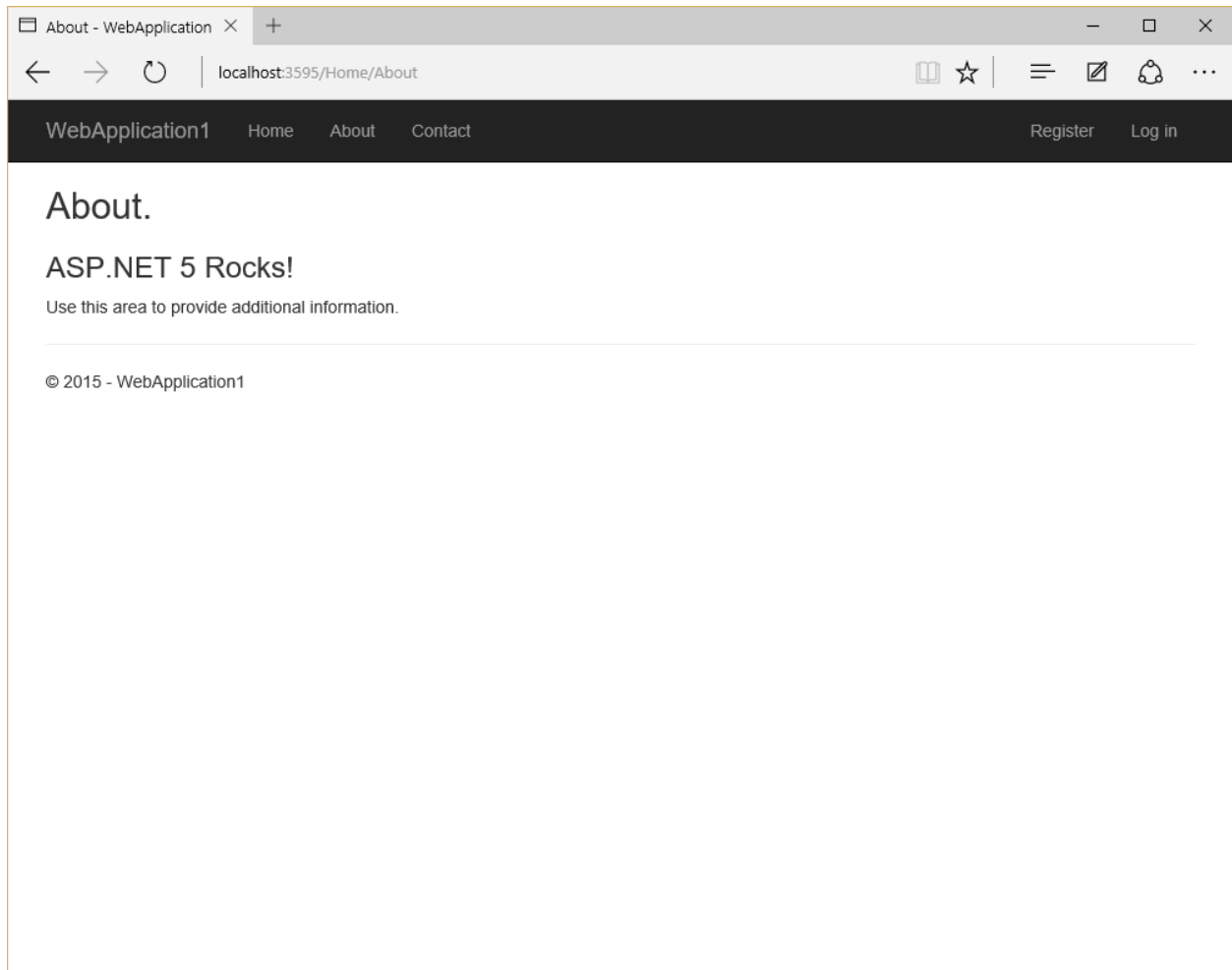
Run the application (Debug -> Start Without Debugging) and after a quick build step, you should see it open in your web browser.



Click on the About link, and note the text on the page. Now, open the `HomeController.cs` file in the Controllers folder, and change the `ViewBag.Message` as follows:

```
ViewBag.Message = "ASP.NET 5 Rocks!";
```

Save the file and, **without rebuilding the project**, refresh your web browser. You should see the updated text. ASP.NET 5 no longer requires that you manually build your server-side logic before viewing it, making small updates much faster to inspect during development.



Server-Side vs. Client-Side Behavior

Modern web applications frequently make use of a combination of server-side and client-side behavior. Over time, ASP.NET has evolved to support more and more client-side behavior, and with ASP.NET 5 it now includes great support for Single Page Applications (SPAs) that shift virtually all of the application logic to the web client, and use the server only to fetch and store data. Your application's approach to where its behavior resides will depend on a variety of factors. The more comfortable your team is with client-side development, the more likely it is that much of your application's behavior will run on the client. If your web site will include a great deal of public content that should be discoverable by search engines, you may wish to ensure the server returns this content directly, rather than having it built up by client-side scripts, since the latter requires [special effort](#) to be indexed by search engines.

On the server, ASP.NET MVC 6 (part of ASP.NET 5) works similarly to its predecessor, including support for Razor-formatted Views as well as integrated support for Web API. On the client, there are many options available for managing client application state, binding to UI elements, and communication with APIs. Learn more about [Client-Side Development](#).

Now we can add a bit of behavior to both the server and the client of the default application, to demonstrate how easy it is to get started building your own ASP.NET 5 application.

Adding Server-Side Behavior

We've already tweaked the behavior of the HomeController's `About` method to change the Message passed to the View. We can add additional server-side behavior by further modifying the HomeController's `About` action and its associated View. Then, we'll enhance this basic information by adding some client-side behavior that makes API calls back to the server.

To start, add a new ViewModel class called `ServerInfoViewModel`. I'm adding this in a new `ViewModels` folder as a convention, but you can place the file in another folder if you prefer.

```
namespace WebApplication1.ViewModels
{
    public class ServerInfoViewModel
    {
        public string Name { get; set; }
        public string LocalAddress { get; set; }
        public string Software { get; set; }
    }
}
```

Next, update the HomeController's `About` method to instantiate this class, set its properties, and pass it to the View.

```
1 using System;
2 using Microsoft.AspNet.Mvc;
3 using WebApplication1.ViewModels;
4
5 namespace WebApplication1.Controllers
6 {
7     public class HomeController : Controller
8     {
9         public IActionResult Index()
10        {
11            return View();
12        }
13
14        public IActionResult About()
15        {
16            string appName = "Your First ASP.NET 5 App";
17            ViewBag.Message = "Your Application Name: " + appName;
18
19            var serverInfo = new ServerInfoViewModel()
20            {
21                Name = Environment.MachineName,
22                Software = Environment.OSVersion.ToString()
23            };
24            return View(serverInfo);
25        }
26
27        public IActionResult Contact()
28        {
29            ViewData["Message"] = "Your contact page.";
30
31            return View();
32        }
33
34        public IActionResult Error()
35        {
36            return View("~/Views/Shared/Error.cshtml");
37        }
38    }
39 }
```

```
38     }  
39 }
```

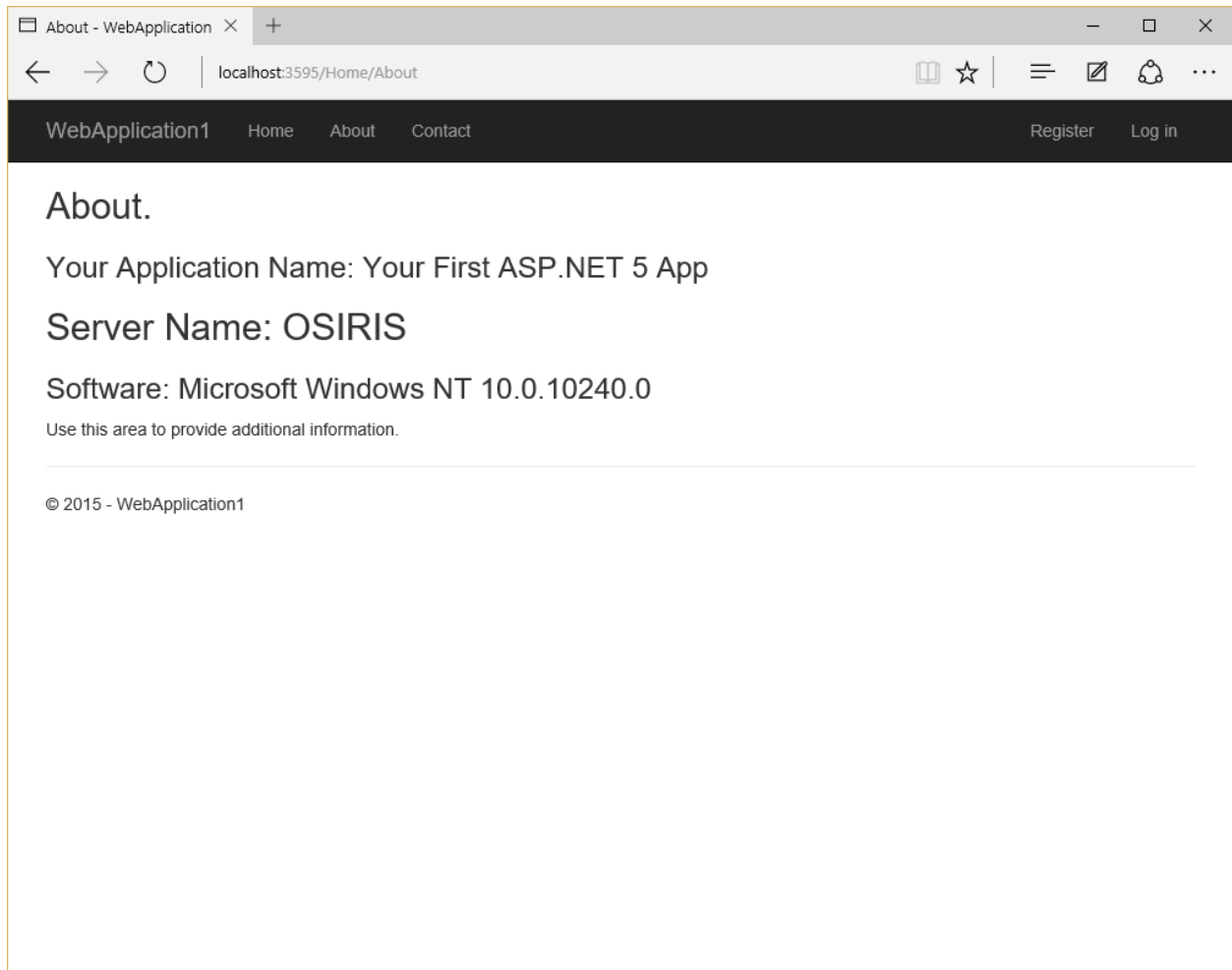
Now we need to update the View to give it a strongly-typed model and display the information using Razor syntax. Modify `Views/Home/About.cshtml` as follows:

```
1 @model WebApplication1.ViewModels.ServerInfoViewModel  
2 @{  
3     ViewData["Title"] = "About";  
4 }  
5 <h2>@ViewData["Title"]</h2>  
6 <h3>@ViewData["Message"]</h3>  
7 <h2>Server Name: @Model.Name</h2>  
8 <h3>Software: @Model.Software</h3>
```

Now we can build the solution. Since the default web template targets both the full .NET and .NET Core, we expect the build to fail when it tries to access the `Environment.MachineName` and `Environment.OSVersion` variables in `HomeController`. This behavior won't work in .NET Core (currently), so we will remove .NET Core from our list of targeted frameworks. Open `project.json` and modify the "frameworks" key as shown:

```
"frameworks": {  
    "dnx451": { }  
},
```

Now we should be able to build and run the solution. Navigate to the About page and you should see your server name and OS version displayed.



Adding server-side behavior in ASP.NET 5 should be very familiar if you have been working with previous versions of ASP.NET MVC.

Adding Client-Side Behavior

Modern web applications frequently make use of rich client-side behavior, whether as part of individual pages generated by the server, or as a Single Page Application. Popular JavaScript frameworks like AngularJS provide rich functionality for SPAs (see [Using Angular for Single Page Applications \(SPAs\)](#)), but in this example we will take advantage of another framework that is included in the basic web project template: jQuery. Our goal is to allow the user to click a button on the About page and have it load a list of the current processes running on the server, without refreshing the page. To do this, we will need to add some HTML and JavaScript to our About.cshtml view, as well as create a Web API controller that our client-side code can call to get the list of processes running on the web server.

Let's begin with the client-side code. We are going to need a button and a list that will be populated with the result of the call to the server. Since we are going to need to refer to the button and list programmatically in our JavaScript code, we will give each one an id. Since the default web project template includes [Bootstrap](#), we can use some of its CSS classes to style the elements. Add the following code to the bottom of the About.cshtml page.

```
<button id="listButton" class="btn-success">List Processes</button>
<ul id="processList" class="list-group"></ul>
```

Next, we need to add some script that will run when the `listButton` button is clicked, and will populate the contents of the `processList` list. Since we want this script to run after jQuery is loaded (in the `_Layout.cshtml` razor file),

we need to place it into a Razor Section called scripts. In this section, we will include a script block that will define a function for binding the list to some data, and a click handler that will make a GET request to our API and call the binding function with the resulting data. Update About.cshtml to add a `@section scripts` as shown:

```

1  @model WebApplication1.ViewModels.ServerInfoViewModel
2  @{
3      ViewData["Title"] = "About";
4  }
5  <h2>@ViewData["Title"].</h2>
6  <h3>@ViewData["Message"]</h3>
7  <h2>Server Name: @Model.Name</h2>
8  <h3>Software: @Model.Software</h3>
9
10 <p>Use this area to provide additional information.</p>
11
12 <button id="listButton" class="btn-success">List Processes</button>
13 <ul id="processList" class="list-group"></ul>
14
15 @section scripts {
16     <script type="text/javascript">
17         function bindData(element, data) {
18             $('<li/>')
19                 .addClass('list-group-item active')
20                 .text('Processes')
21                 .appendTo(element);
22             $.each(data, function (id, option) {
23                 $('<li/>')
24                     .addClass('list-group-item')
25                     .text(option.Name)
26                     .appendTo(element);
27             });
28         }
29         $(document).ready(function () {
30             $("#listButton").bind("click", function (e) {
31                 $.ajax({
32                     url: "/api/processes",
33                     data: "",
34                     type: "GET",
35                     success: function (data) {
36                         bindData($("#processList"), data);
37                     }
38                 });
39             });
40         });
41     </script>
42 }

```

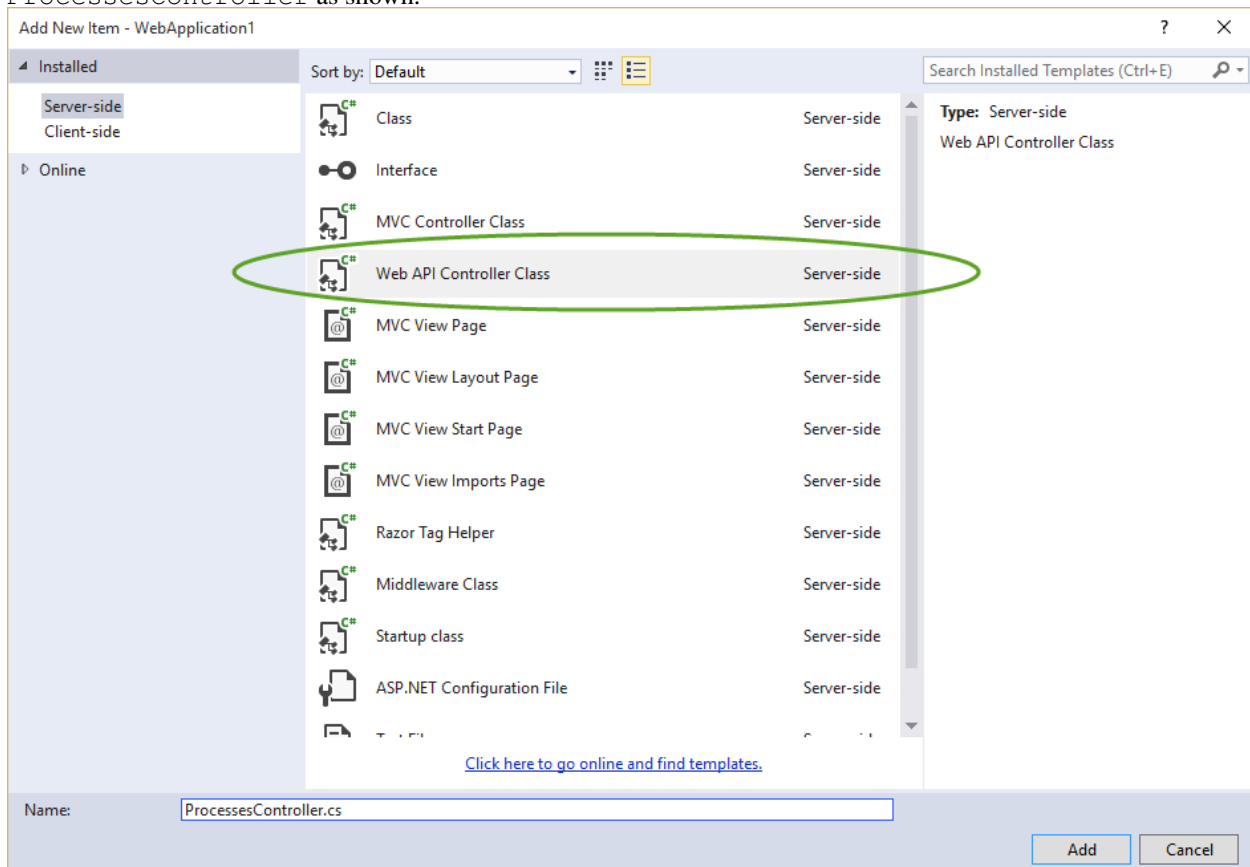
At this point, we're done with the client code and we need to add the Web API code that will respond to a GET request to the `"/api/processes"` URL. Add a new class, `ProcessInfoViewModel` (not to be confused with the .NET framework library's `ProcessInfo` class), to the `ViewModels` folder, and give it just one string property, `Name`:

```

namespace WebApplication1.ViewModels
{
    public class ProcessInfoViewModel
    {
        public string Name { get; set; }
    }
}

```

Now add a new item to the Controllers folder, and choose a new Web API Controller Class. Call it `ProcessesController` as shown.



Delete all of the methods except for the `Get()` method, and update the `Get()` method to return an enumeration of `ProcessInfoViewModel` items as shown.

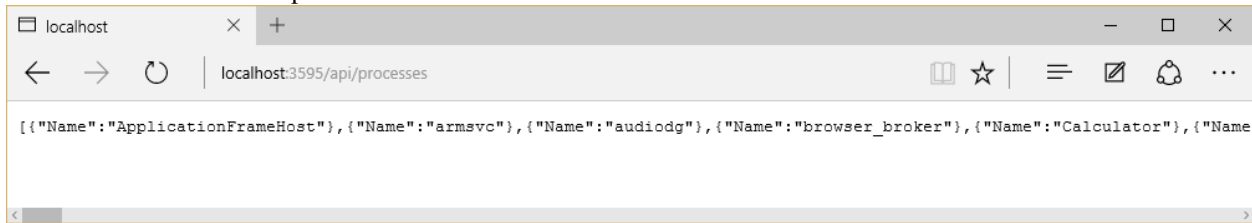
```

1 using System.Collections.Generic;
2 using System.Diagnostics;
3 using System.Linq;
4 using Microsoft.AspNet.Mvc;
5 using WebApplication1.ViewModels;
6
7 namespace WebApplication1.Controllers
8 {
9     [Route("api/[controller]")]
10    public class ProcessesController : Controller
11    {
12        // GET: api/values
13        [HttpGet]
14        public IEnumerable<ProcessInfoViewModel> Get()
15        {
16            var processList = Process.GetProcesses().OrderBy(p => p.ProcessName).ToList();
17
18            return processList.Select(p => new ProcessInfoViewModel() { Name = p.ProcessName });
19        }
20    }
21 }

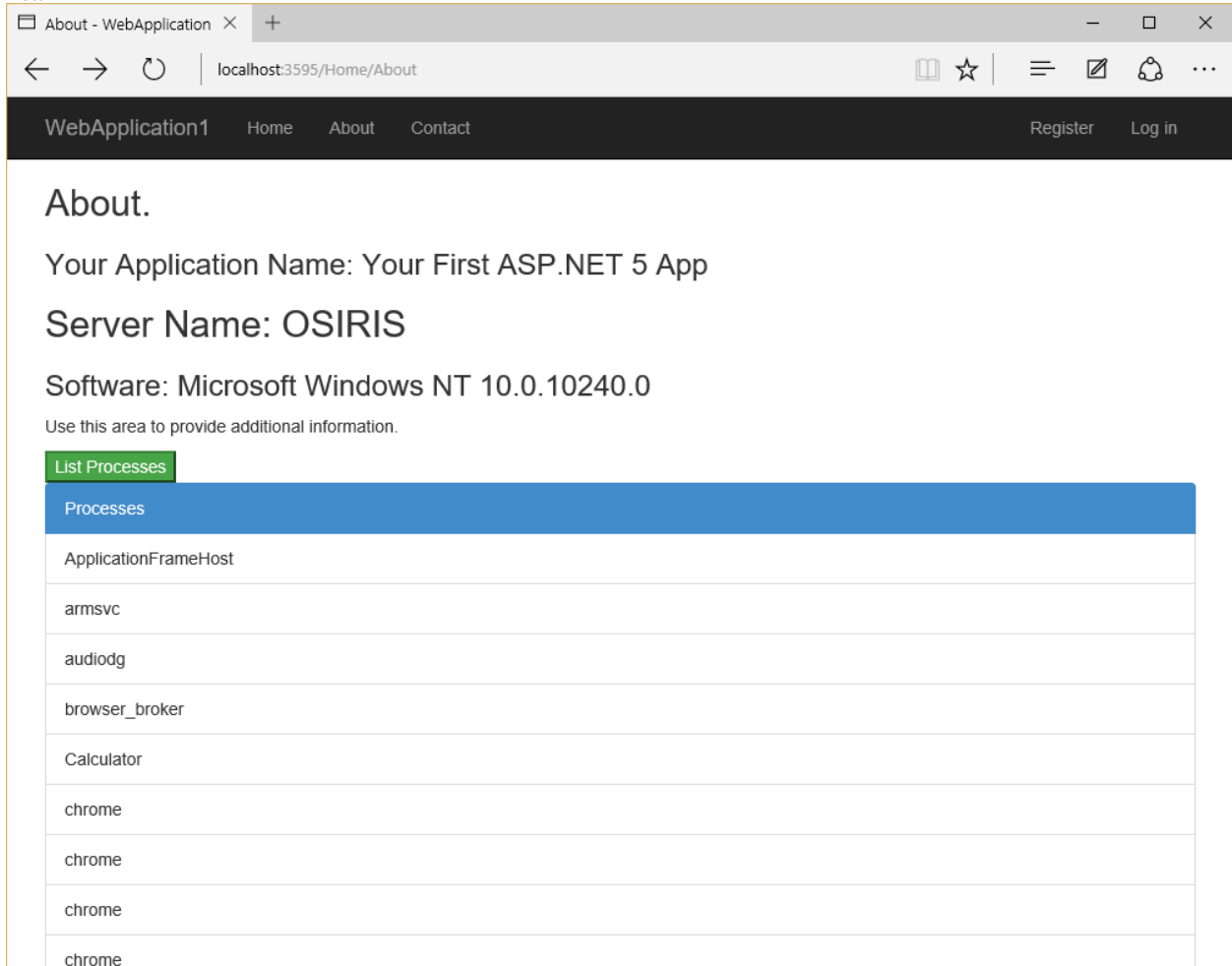
```

At this point, you should be able to test the API by navigating to the `/api/processes` path in your browser. You should

see some JSON-formatted process names.



Navigating back to the About page, clicking on the button should similarly load the list of processes into the HTML list.



Now the application includes both server-side and client-side behavior, running together on the About page.

Summary

ASP.NET 5 introduces a few new concepts, but should be very familiar to developers who have used previous versions of ASP.NET. Creating a new web application that includes both server-side and client-side behavior takes only a few minutes using the Visual Studio ASP.NET 5 Starter Web template.

2.2.2 Your First ASP.NET 5 Application on a Mac

By Steve Smith

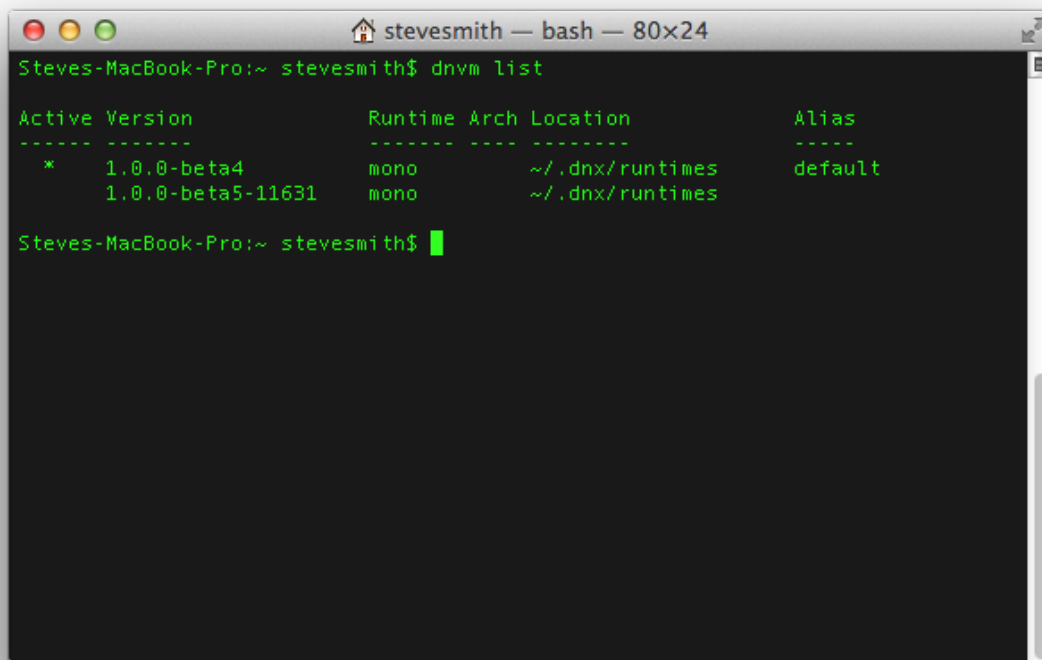
ASP.NET 5 is cross-platform and can be developed and run on Mac OS X as well as Linux and Windows. See how you can quickly install, scaffold, run, debug, and deploy ASP.NET applications on a Mac.

In this article:

- *Setting Up Your Development Environment*
- *Scaffolding Applications Using Yeoman*
- *Developing ASP.NET Applications on a Mac With Visual Studio Code*
- *Running Locally Using Kestrel*
- *Publishing to Azure*

Setting Up Your Development Environment

First, make sure you have installed ASP.NET on your Mac OS X machine. This step will include installing Homebrew and configuring DNVM. This article has been tested with DNX beta4. You can confirm that you are running the correct version of dnx by running the command `dnvm list`. You should see `default` listed under the `Alias` column for `1.0.0-beta4` as shown:



```
stevesmith$ dnvm list

Active Version      Runtime Arch Location      Alias
-----
*   1.0.0-beta4      mono    ~/.dnx/runtimes default
    1.0.0-beta5-11631 mono    ~/.dnx/runtimes
```

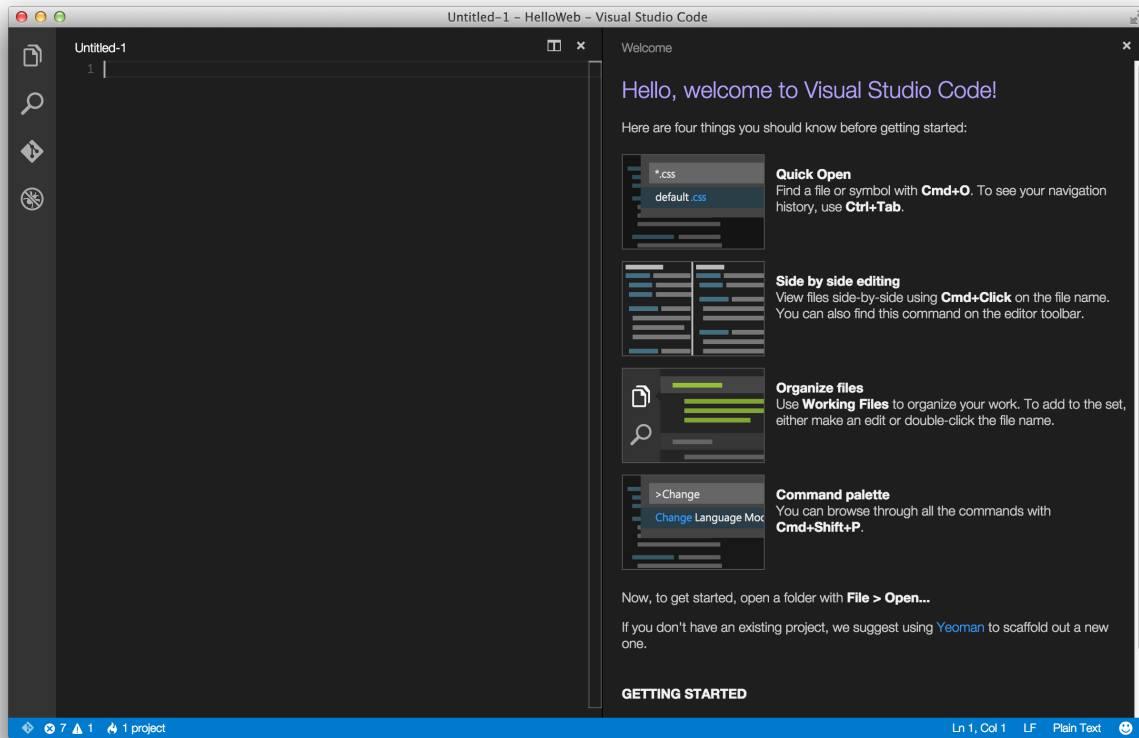
Scaffolding Applications Using Yeoman

Coming soon: instructions for getting started with Yeoman.

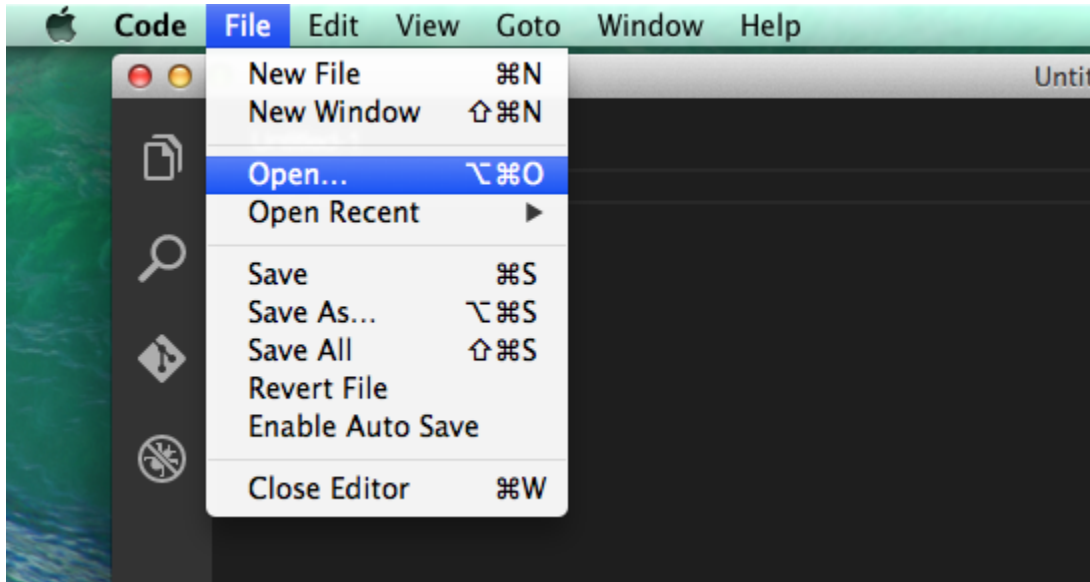
Download a sample [Empty Web Site](#) to use with this article.

Developing ASP.NET Applications on a Mac With Visual Studio Code

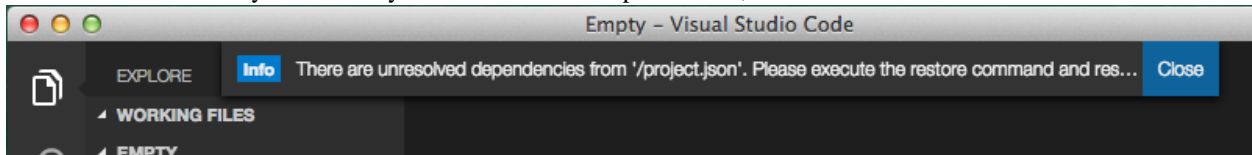
Now, install Visual Studio Code from code.visualstudio.com. Unzip the application and open it - the first time you should see a welcome screen:



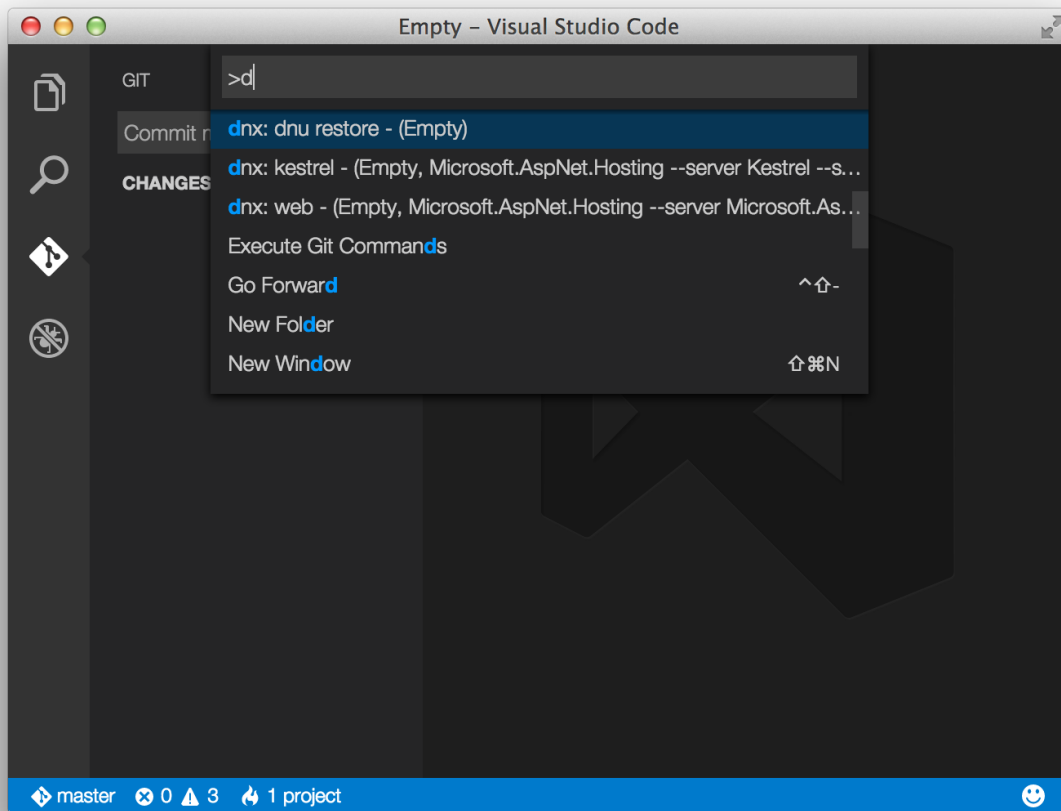
To get started with your first ASP.NET application on a Mac, select File -> Open and choose the folder where you unzipped the empty web site (or scaffolded a site with Yeoman).



Visual Studio Code may detect that you need to restore dependencies, as shown in this screenshot:



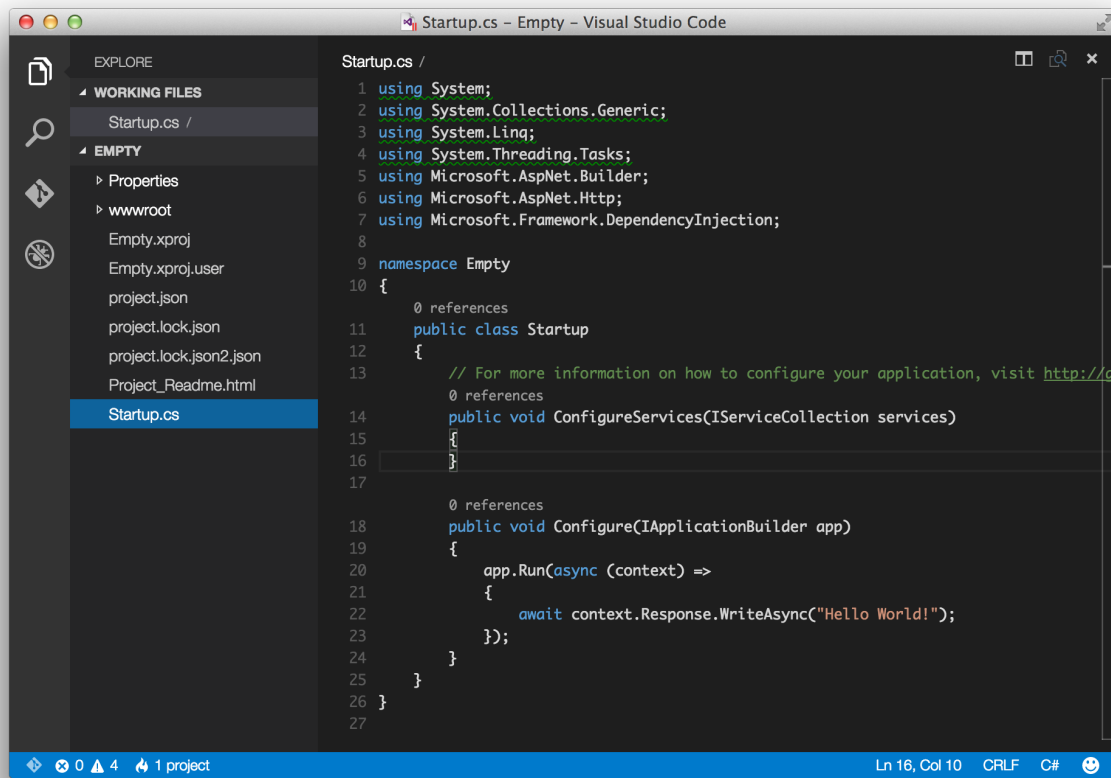
From a Terminal / bash prompt, run `dnx restore` to restore the project's dependencies. Alternately, you can press command `shift p` and then type `>d` as shown:



This will allow you to run commands directly from within Visual Studio Code, including `dnu restore` and any commands defined in `project.json`.

At this point, you should be able to host and browse to this simple ASP.NET web application, which we'll see in a moment.

This empty project template simply displays “Hello World!”. Open `Startup.cs` in Visual Studio Code to see how this is configured:



If this is your first time using Visual Studio Code (or just *Code* for short), note that it provides a very streamlined, fast, clean interface for quickly working with files, while still providing tooling to make writing code extremely productive.

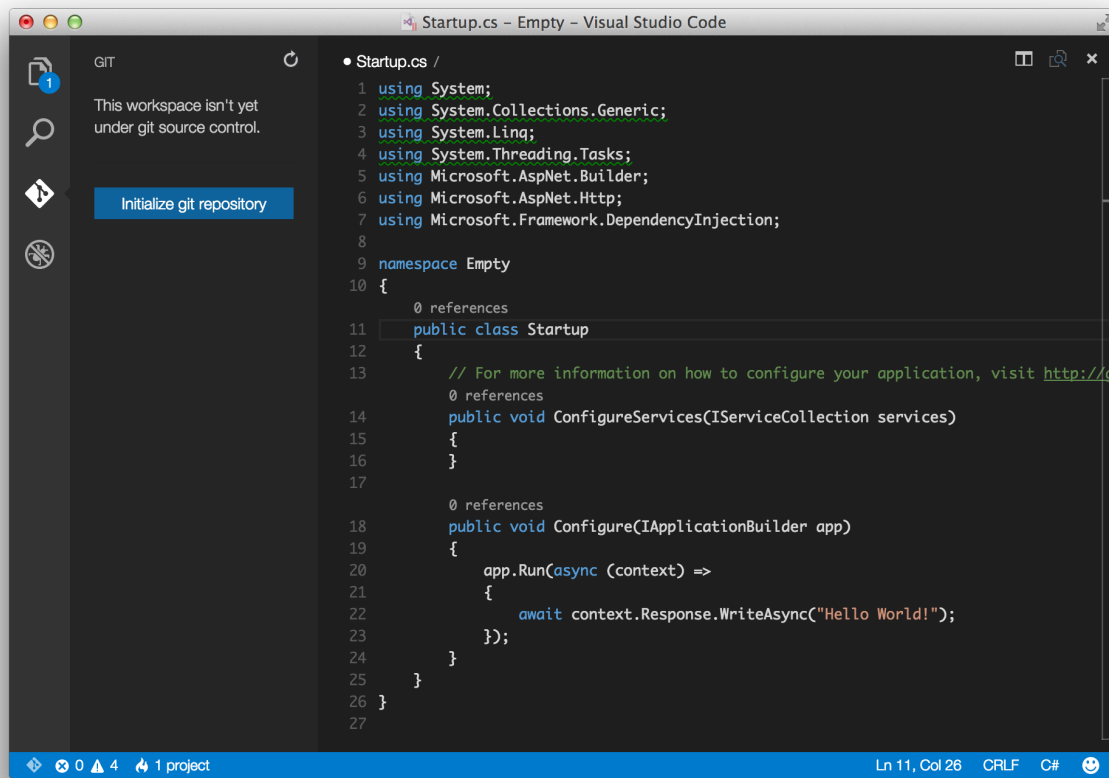
In the left navigation bar, there are four icons, representing four viewlets:

- Explore
- Search
- Git
- Debug

The Explore viewlet allows you to quickly navigate within the folder system, as well as easily see the files you are currently working with. It displays a badge to indicate whether any files have unsaved changes, and new folders and files can easily be created (without having to open a separate dialog window). You can easily Save All from a menu option that appears on mouse over, as well.

The Search viewlet allows you to quickly search within the folder structure, searching filenames as well as contents.

Code will integrate with git if it is installed on your system. You can easily initialize a new repository, make commits, and push changes from the Git viewlet.



The Debug viewlet supports interactive debugging of applications. Currently only node.js and mono applications are supported by the interactive debugger.

Finally, Code's editor has a ton of great features. You should note right away that several using statements are underlined, because Code has determined they are not necessary. Note that classes and methods also display how many references there are in the project to them. If you're coming from Visual Studio, Code includes many of the keyboard shortcuts you're used to, such as command k c to comment a block of code, and command k u to uncomment.

Running Locally Using Kestrel

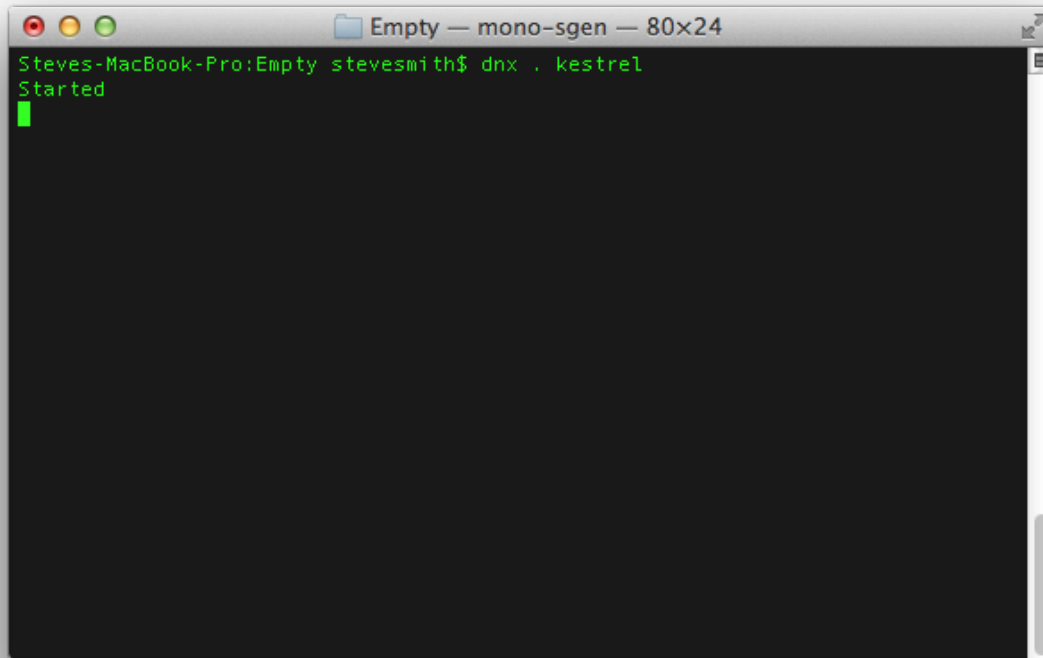
The sample we're using is configured to use Kestrel as its web server. You can see it configured in the project.json file, where it is specified as a dependency and as a command.

```

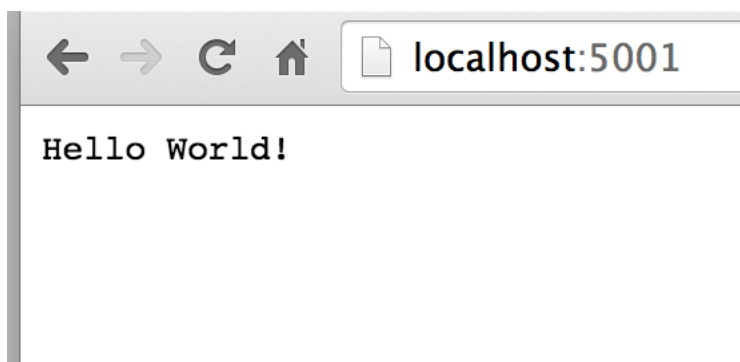
1 {
2   "webroot": "wwwroot",
3   "version": "1.0.0-*",
4
5   "dependencies": {
6     "Microsoft.AspNet.Server.IIS": "1.0.0-beta4",
7     "Microsoft.AspNet.Server.WebListener": "1.0.0-beta4",
8     "Kestrel": "1.0.0-beta4"
9   },
10
11   "commands": {
12     "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5001",
13     "kestrel": "Microsoft.AspNet.Hosting --server Kestrel --server.urls http://localhost:5001"
14   }
15 }
```

```
14 },  
15 // more deleted
```

In order to run the `kestrel` command, which will launch the web application on localhost port 5001, run `dnx . kestrel` from a command prompt:



Navigate to `localhost:5001` and you should see:



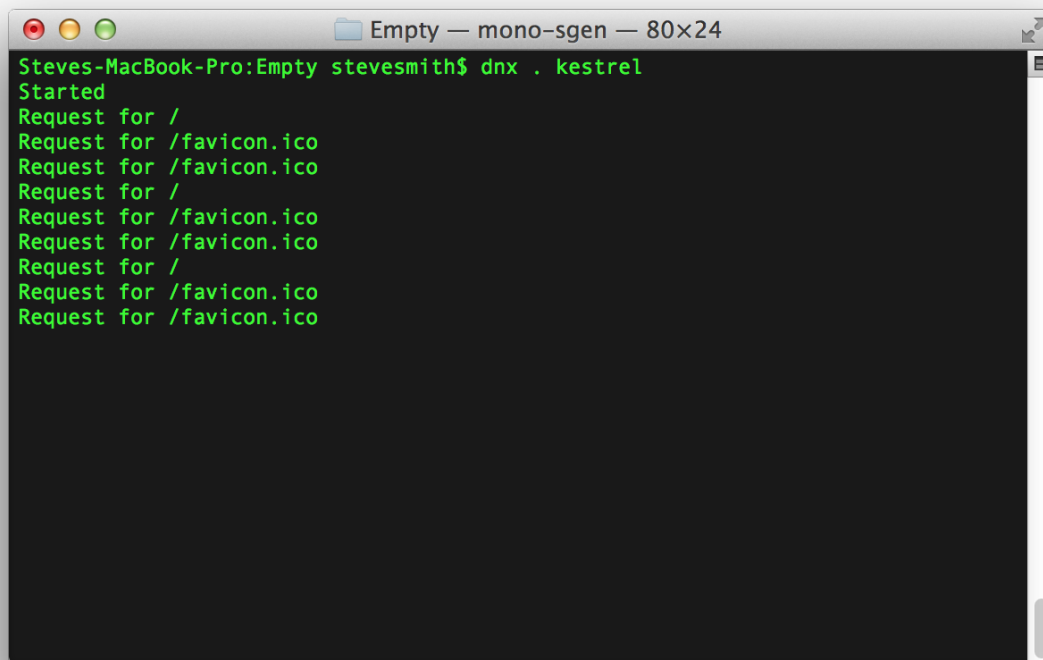
It's not necessarily obvious, but if you want to stop the web server once you've started it, simply press `enter`.

We can update the application to output information to the console whenever a request is received. Update the `Configure()` method as follows:

```
1 public void Configure(IApplicationBuilder app)  
2 {  
3     app.Run(async (context) =>  
4     {
```

```
5     Console.WriteLine("Request for " + context.Request.Path);
6     await context.Response.WriteAsync("Hello World!");
7   });
8 }
```

Save the file and restart the web server. Make a few requests to the URL. You should see the request information output in the Terminal window (recall that most browsers will automatically attempt to request a `favicon.ico` file when making a request to a new domain):

A screenshot of a terminal window titled "Empty — mono-sgen — 80x24". The terminal shows the command `dnx . kestrel` being executed, followed by the output "Started". Below this, there are several lines of "Request for" messages, including requests for "/" and "/favicon.ico".

```
Steve's-MacBook-Pro:Empty stevesmith$ dnx . kestrel
Started
Request for /
Request for /favicon.ico
Request for /favicon.ico
Request for /
Request for /favicon.ico
Request for /favicon.ico
Request for /
Request for /favicon.ico
Request for /favicon.ico
```

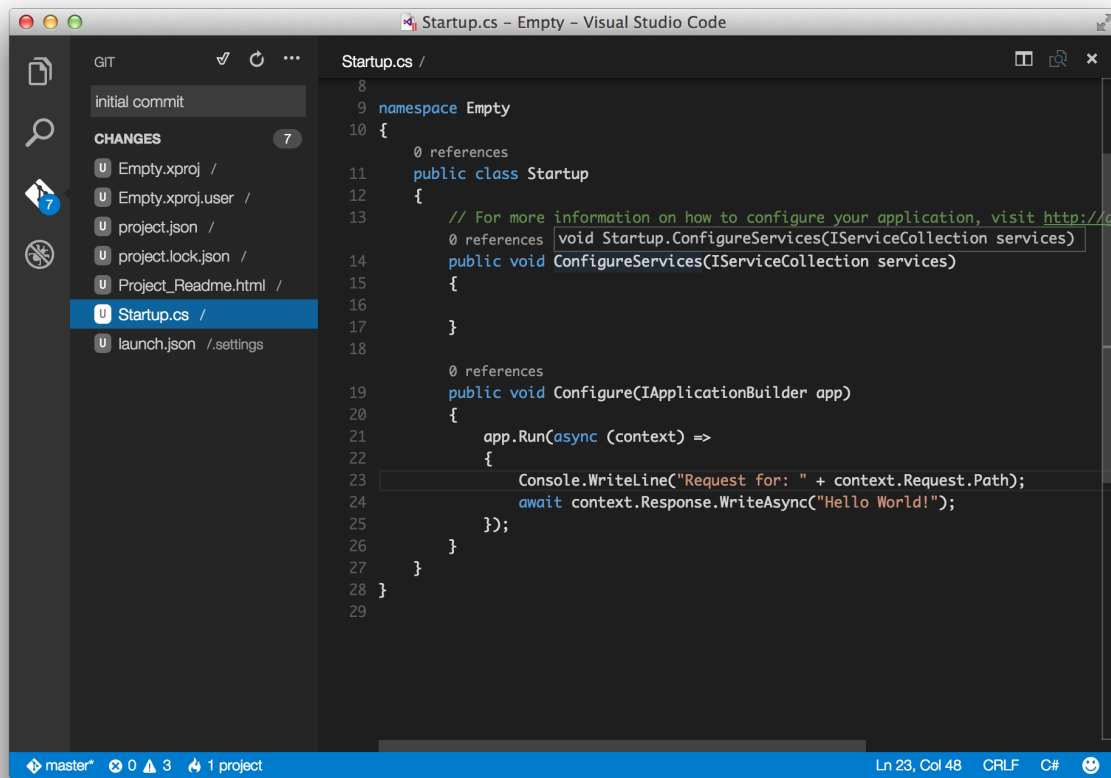
As you can see, it's quite straightforward, especially if you're already familiar with command line tooling, to get started building ASP.NET applications using Visual Studio Code on Mac OS X.

Publishing to Azure

Once you've developed your application, you can easily use the git integration built into Visual Studio Code to push updates to production, hosted on [Windows Azure](#).

Initialize Git

First, if you haven't already done so, initialize git in the folder you're working in. Simply click on the git viewlet and click the `Initialize git repository` button.





Add a commit message as shown in the image above, and press enter or click the checkmark icon to commit the staged files. Now git is tracking changes, so if you make an update to a file, the git viewlet will display how many files have changed since your last commit.

Initialize Azure Website

If you're unfamiliar with Windows Azure, you may not know that you can deploy to Azure Web Apps directly using git. They also support other workflows, but being able to simply perform a `git push` to a remote can be a very convenient way to make updates.

Note: Learn more about [configuring Azure to support deployment from source control](#).

Create a new Web App in Azure, and configure it to support git deployment. If you don't have an Azure account, you can [create a free trial](#). Once it's configured, you should see a page like this:

YOUR REPOSITORYGIT URL  

Push my local files to Windows Azure

1

Download Git[Get it here](#) if you don't already have it.

2

Commit your local files

At the command prompt, change to the root directory for your app, and then type this command:

To commit local files

```
git init
git add .
git commit -m "initial commit"
```

Or to clone content that's already on the app

```
git clone https://ardalis@vscode-mac.scm.azurewebsites.net:443/vscode-mac.git
```

3

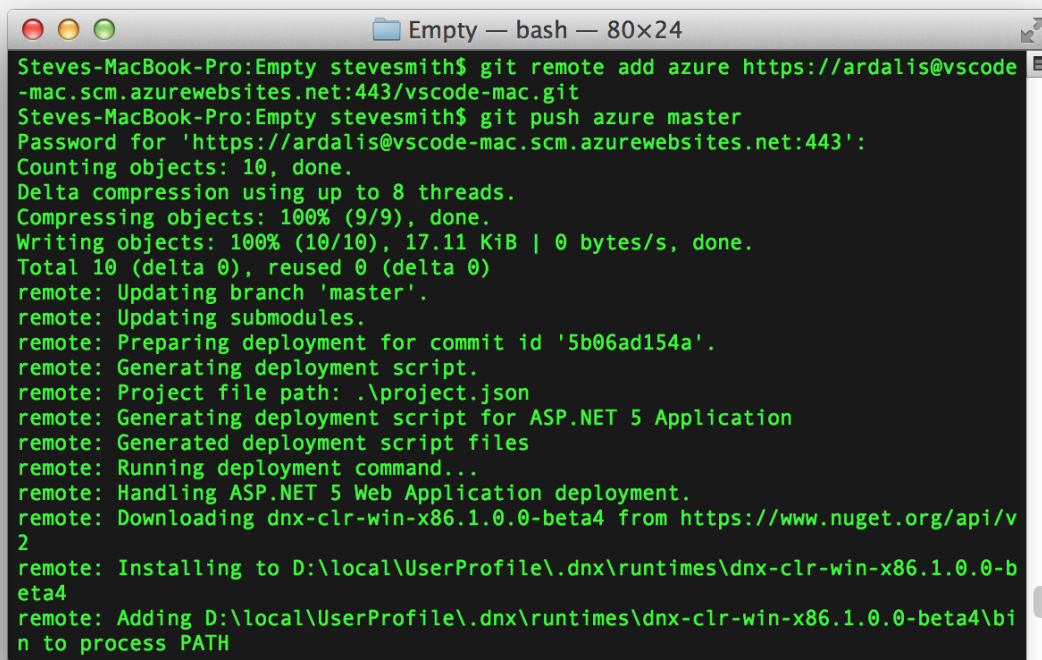
Add remote Windows Azure repository and push your stuff

At the command prompt, change to the root directory for your app, and then type this command:

```
git remote add azure https://ardalis@vscode-mac.scm.azurewebsites.net:443/vscode-mac.git
git push azure master
```

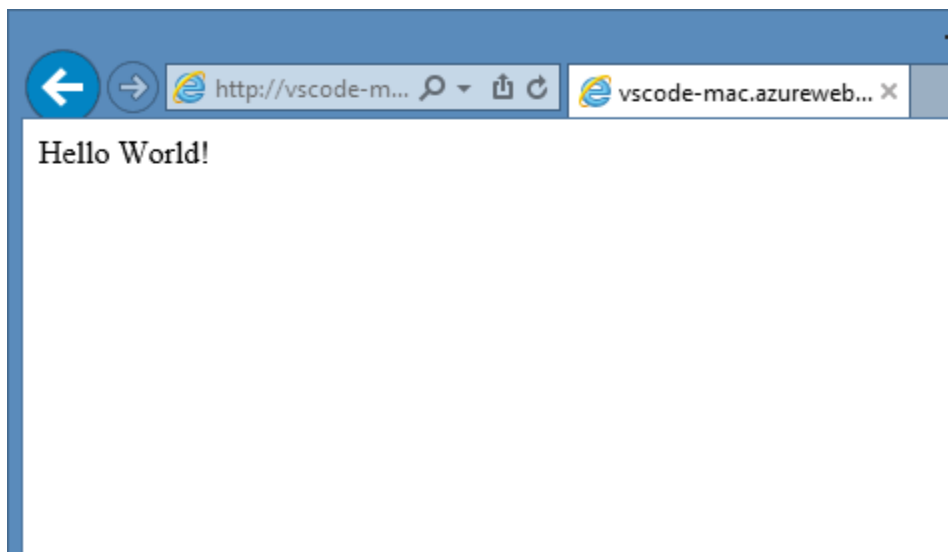
Did you forget your password? [Reset your deployment credentials](#)

Note the GIT URL, which is also shown in step 3. Assuming you've been following along, you can skip steps 1 and 2 and go directly to step 3. In a Terminal window, add a remote named `azure` with the GIT URL shown, and then perform `git push azure master` to deploy. You should see output similar to the following:

A terminal window titled 'Empty — bash — 80x24' showing the execution of git commands and the resulting deployment of an ASP.NET 5 application to Azure. The output includes details about object counting, compression, and the generation of deployment scripts.

```
Steves-MacBook-Pro:Empty stevesmith$ git remote add azure https://ardalis@vscode-mac.scm.azurewebsites.net:443/vscode-mac.git
Steves-MacBook-Pro:Empty stevesmith$ git push azure master
Password for 'https://ardalis@vscode-mac.scm.azurewebsites.net:443':
Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (10/10), 17.11 KiB | 0 bytes/s, done.
Total 10 (delta 0), reused 0 (delta 0)
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id '5b06ad154a'.
remote: Generating deployment script.
remote: Project file path: .\project.json
remote: Generating deployment script for ASP.NET 5 Application
remote: Generated deployment script files
remote: Running deployment command...
remote: Handling ASP.NET 5 Web Application deployment.
remote: Downloading dnx-clr-win-x86.1.0.0-beta4 from https://www.nuget.org/api/v2
remote: Installing to D:\local\UserProfile\dnx\runtimes\dnx-clr-win-x86.1.0.0-beta4
remote: Adding D:\local\UserProfile\dnx\runtimes\dnx-clr-win-x86.1.0.0-beta4\bin to process PATH
```

Now you can browse to your Web App and you should see your newly deployed application.

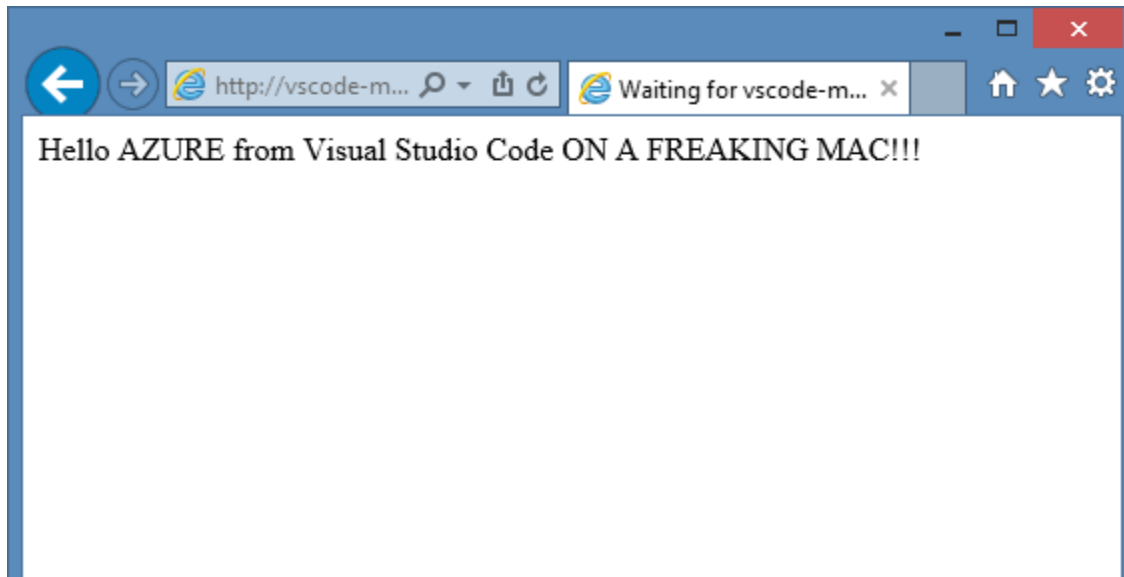


At this point, you can make additional changes to the application, commit them, and whenever you're ready to deploy, simply perform another `git push azure master` from a Terminal prompt. To demonstrate, let's update the message being printed:

```
1 public void Configure(IApplicationBuilder app)
2 {
3     app.Run(async (context) =>
```

```
4 {  
5     Console.WriteLine("Request for " + context.Request.Path);  
6     await context.Response.WriteAsync(  
7         "Hello AZURE from Visual Studio Code ON A FREAKING MAC!!!");  
8     });  
9 }
```

Save the changes. Commit them using the git viewlet. Run `git push azure master` from a Terminal prompt, once more. Then refresh your browser:



Summary

ASP.NET 5 and DNX support installation on Mac OS X. Developers can quickly install the necessary tools to get started, including Yeoman for app scaffolding and [Visual Studio Code](#) for rapid lightweight editing with built-in support for debugging, git integration, and Intellisense.

Additional Reading

Learn more about Visual Studio Code:

- [Announcing Visual Studio Code Preview](#)
- code.visualstudio.com
- [Visual Studio Code Documentation](#)

2.2.3 Create a New NuGet Package with DNX

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.2.4 Publish to an Azure Web App using Visual Studio

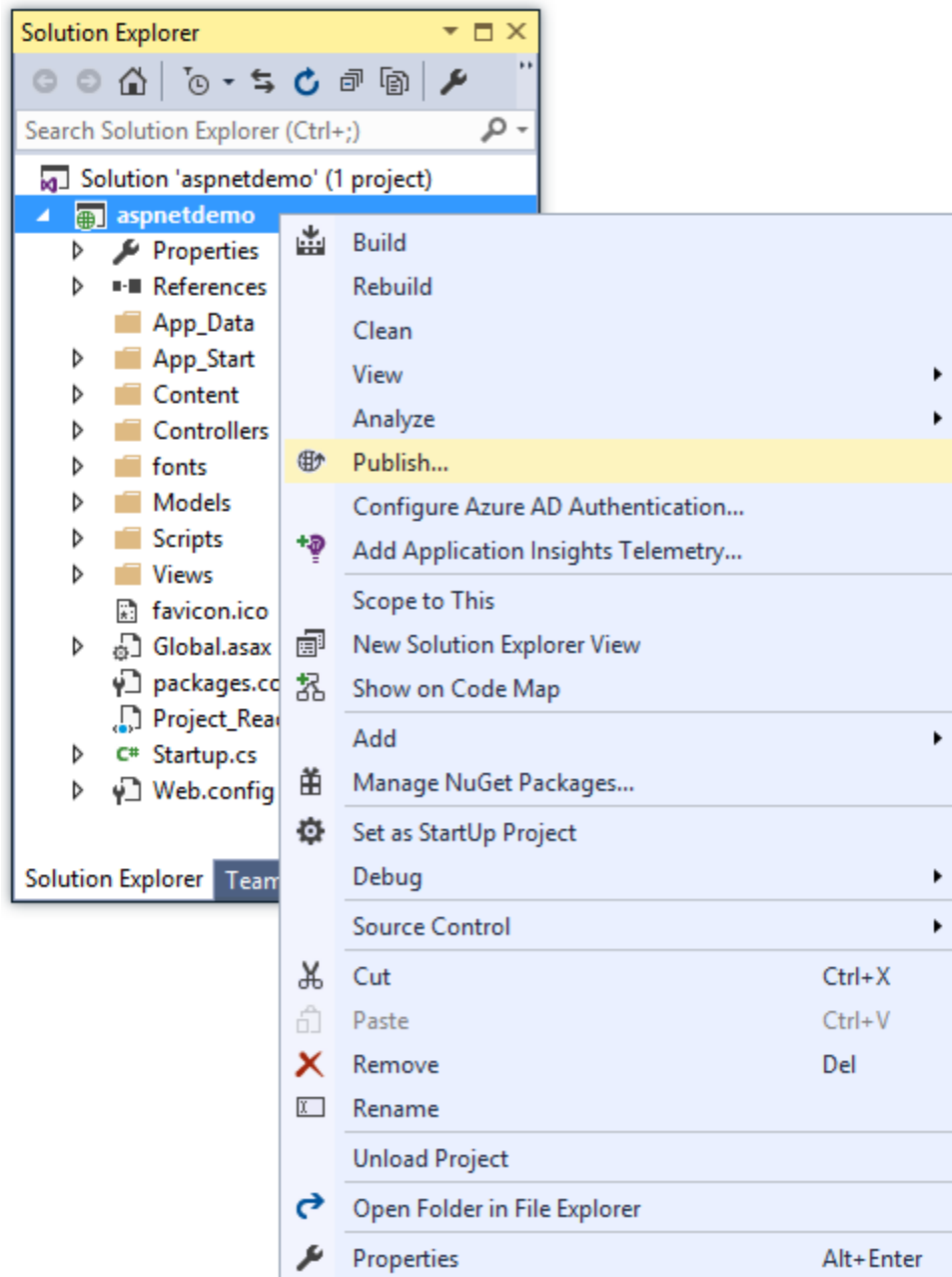
By Erik Reitan

This article describes how to publish an ASP.NET web app to Azure using Visual Studio.

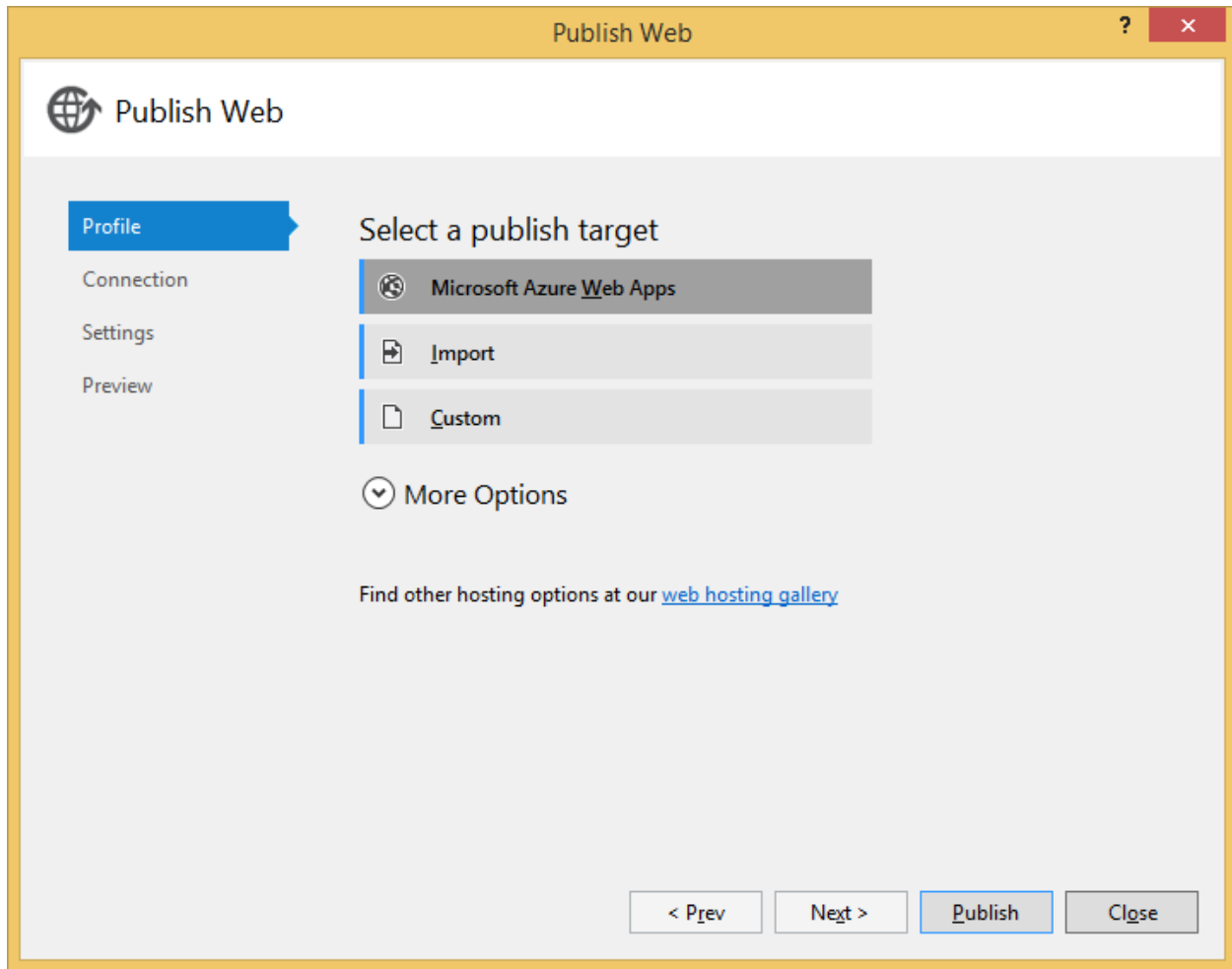
Note: To complete this tutorial, you need a Microsoft Azure account. If you don't have an account, you can [activate your MSDN subscriber benefits](#) or [sign up for a free trial](#).

Start by either creating a new ASP.NET web app or opening an existing ASP.NET web app.

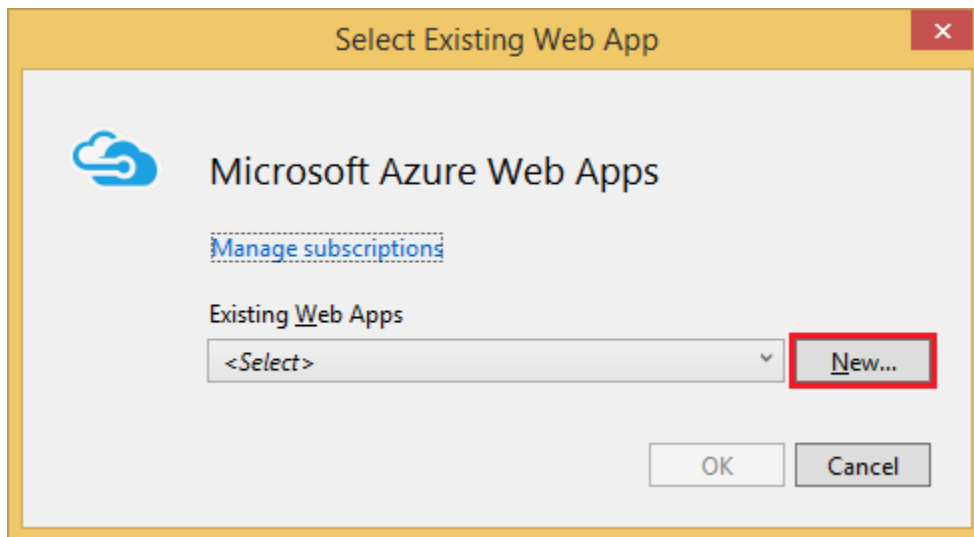
1. In **Solution Explorer** of Visual Studio, right-click on the project and select **Publish**.



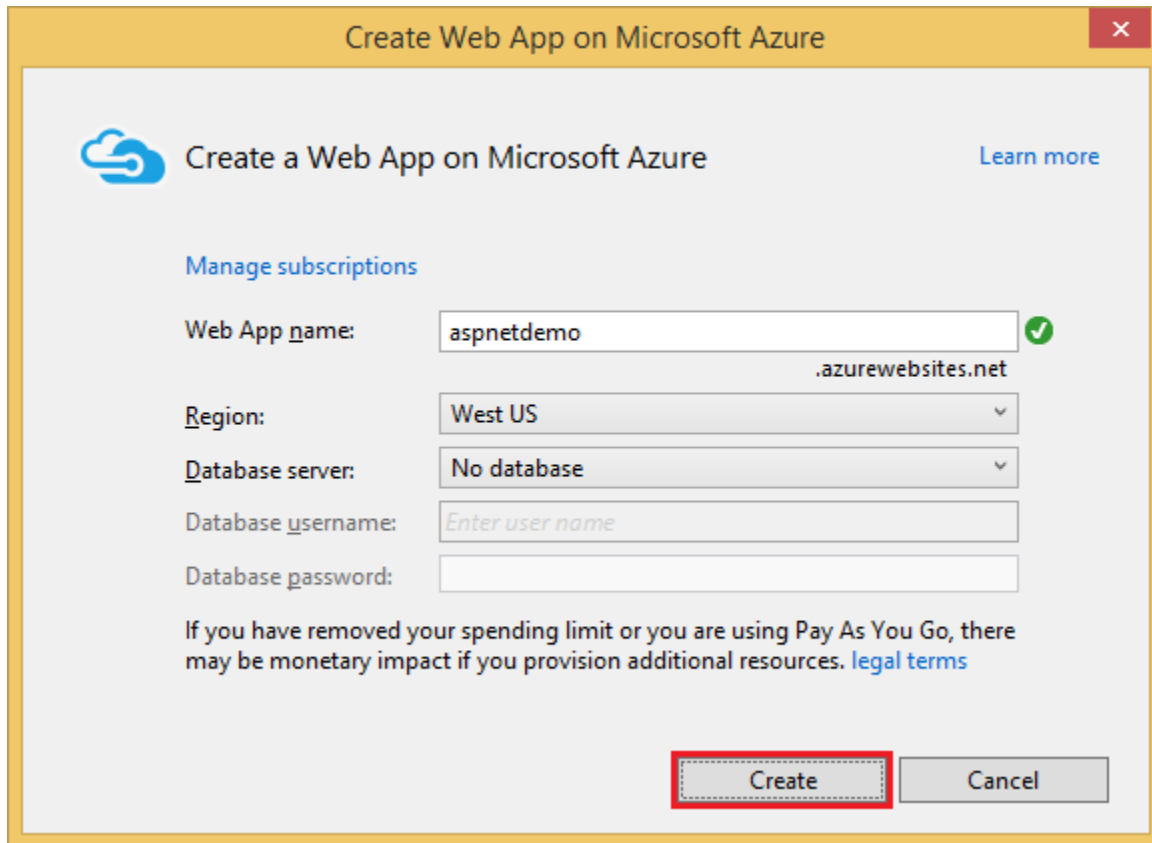
2. In the **Publish Web** dialog box, click on **Microsoft Azure Web Apps** and log into your Azure subscription.



3. Click **New** in the **Select Existing Web App** dialog box to create a new Web app in Azure.




4. Enter a site name and region. You can optionally create a new database server, however if you've created a database server in the past, use that. When you're ready to continue, click **Create**.



Create Web App on Microsoft Azure

Create a Web App on Microsoft Azure [Learn more](#)

[Manage subscriptions](#)

Web App name:  [.azurewebsites.net](#)

Region:

Database server:

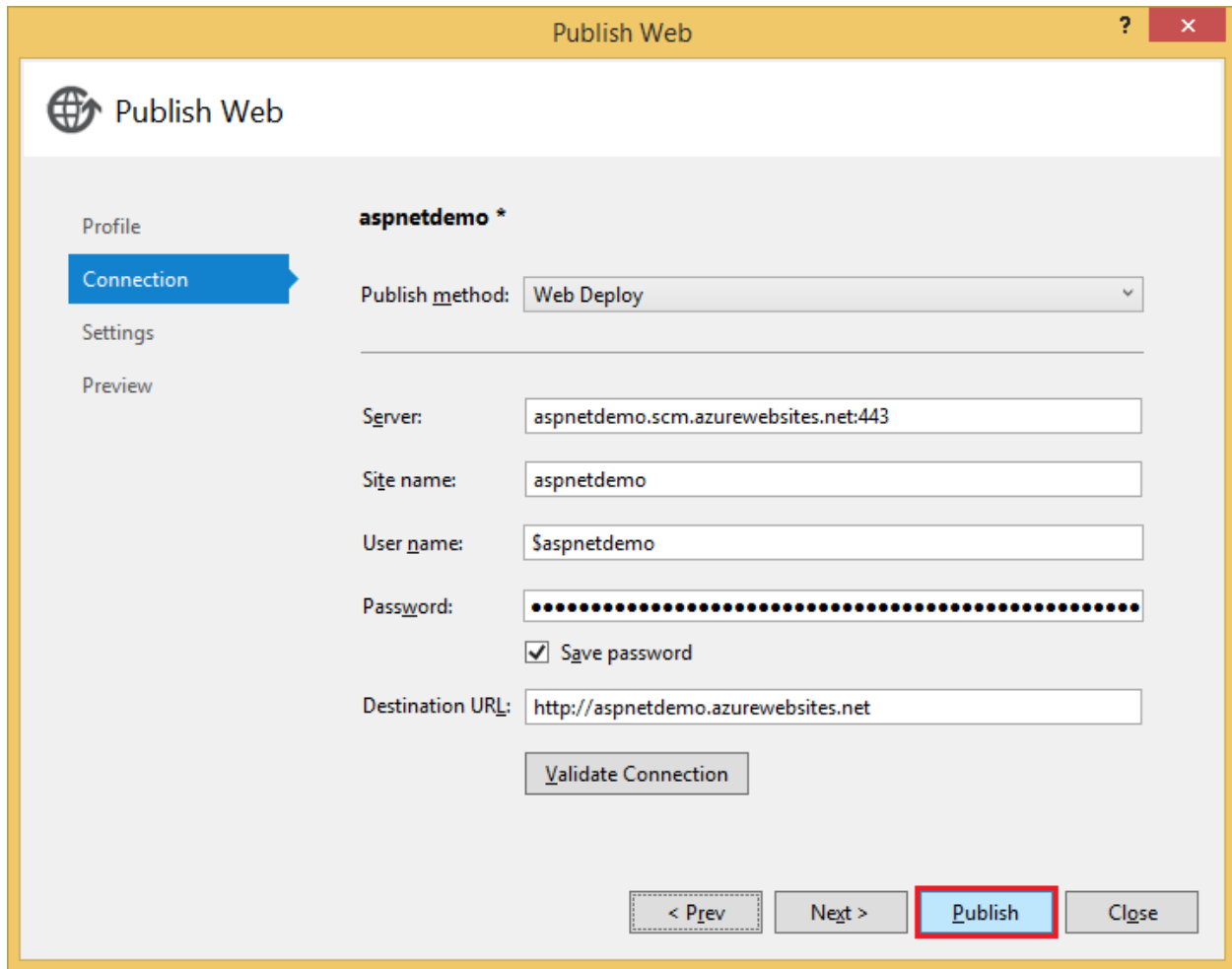
Database username:

Database password:

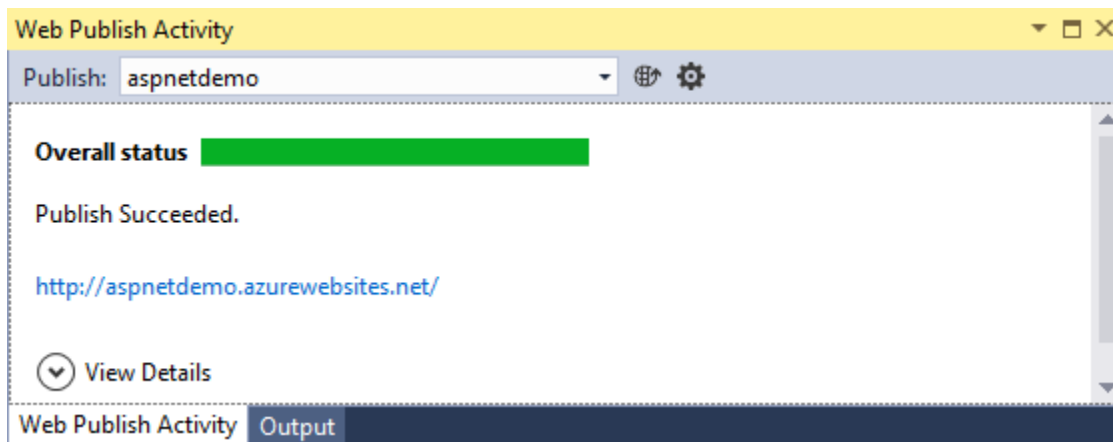
If you have removed your spending limit or you are using Pay As You Go, there may be monetary impact if you provision additional resources. [legal terms](#)

Database servers are a precious resource. For test and development it's best to use an existing server. There is **no** validation on the database password, so if you enter an incorrect value, you won't get an error until your web app attempts to access the database.

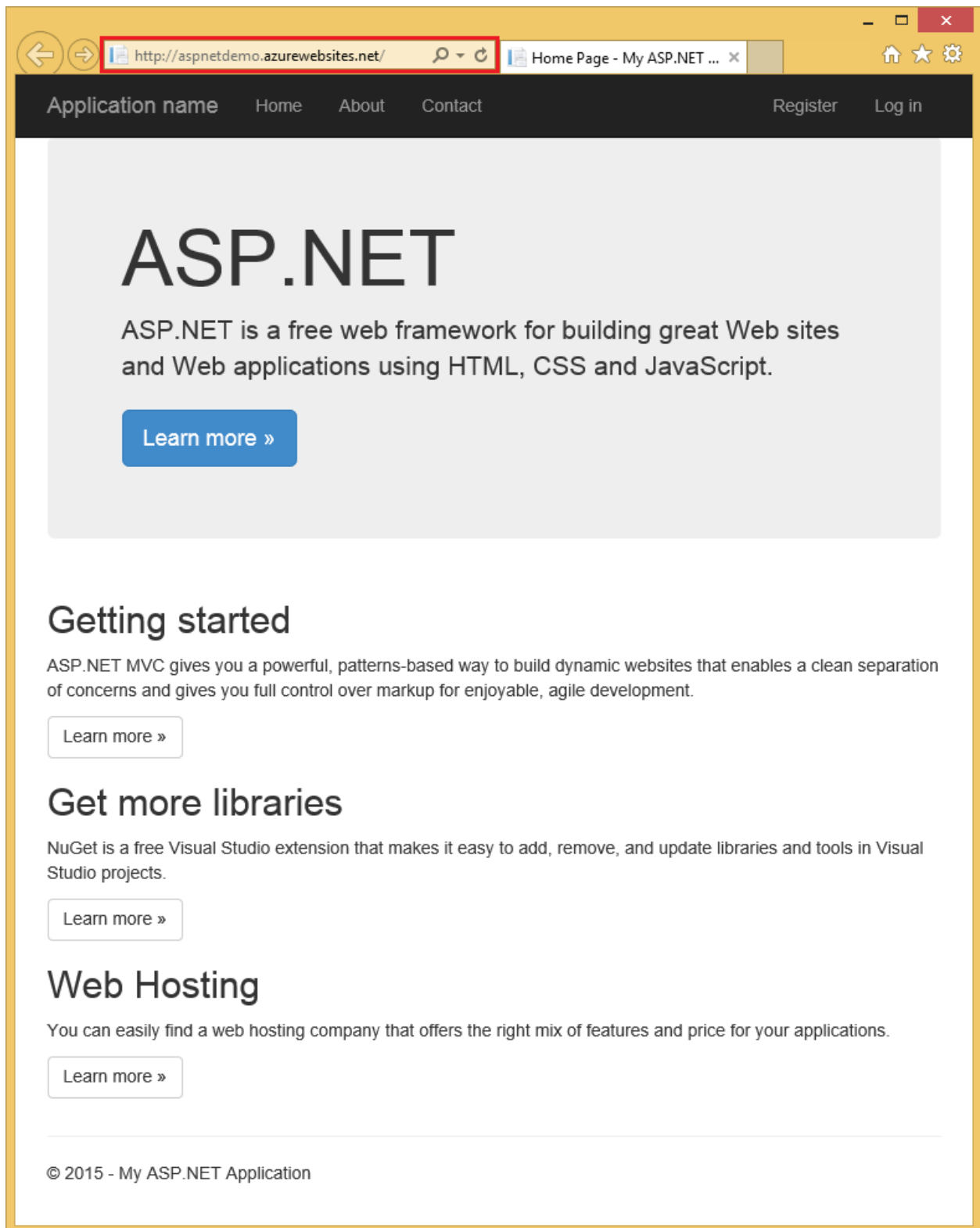
5. On the **Connection** tab of the **Publish Web** dialog box, click **Publish**.



You can view the publishing progress in the **Web Publish Activity** window within Visual Studio.



When publishing to Azure is complete, your web app will be displayed in a browser running on Azure.



2.2.5 ASP.NET 5 on Nano Server

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.2.6 ASP.NET 5 on Azure Service Fabric

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

Learn more about [Azure Service Fabric](#).

2.3 Conceptual Overview

2.3.1 Introduction to ASP.NET 5

By [Daniel Roth](#)

ASP.NET 5 is a significant redesign of ASP.NET. This topic introduces the new concepts in ASP.NET 5 and explains how they help you develop modern web apps.

What is ASP.NET 5?

ASP.NET 5 is a new open-source and cross-platform framework for building modern cloud-based Web applications using .NET. We built it from the ground up to provide an optimized development framework for apps that are either deployed to the cloud or run on-premises. It consists of modular components with minimal overhead, so you retain flexibility while constructing your solutions. You can develop and run your ASP.NET 5 applications cross-platform on Windows, Mac and Linux. ASP.NET 5 is fully open source on [GitHub](#).

Why build ASP.NET 5?

The first preview release of ASP.NET 1.0 came out almost 15 years ago. Since then millions of developers have used it to build and run great web applications, and over the years we have added and evolved many, many capabilities to it.

With ASP.NET 5 we are making a number of architectural changes that make the core web framework much leaner and more modular. ASP.NET 5 is no longer based on System.Web.dll, but is instead based on a set of granular and well factored NuGet packages allowing you to optimize your app to have just what you need. You can reduce the surface area of your application to improve security, reduce your servicing burden and also to improve performance in a true pay-for-what-you-use model.

ASP.NET 5 is built with the needs of modern Web applications in mind, including a unified story for building Web UI and Web APIs that integrate with today's modern client-side frameworks and development workflows. ASP.NET 5 is also built to be cloud-ready by introducing environment-based configuration and by providing built-in dependency injection support.

To appeal to a broader audience of developers, ASP.NET 5 supports cross-platform development on Windows, Mac and Linux. The entire ASP.NET 5 stack is open source and encourages community contributions and engagement. ASP.NET 5 comes with a new, agile project system in Visual Studio while also providing a complete command-line interface so that you can develop using the tools of your choice.

In summary, with ASP.NET 5 you gain the following foundational improvements:

- New light-weight and modular HTTP request pipeline

- Ability to host on IIS or self-host in your own process
- Built on .NET Core, which supports true side-by-side app versioning
- Ships entirely as NuGet packages
- Integrated support for creating and using NuGet packages
- Single aligned web stack for Web UI and Web APIs
- Cloud-ready environment-based configuration
- Built-in support for dependency injection
- New tooling that simplifies modern web development
- Build and run cross-platform ASP.NET apps on Windows, Mac and Linux
- Open source and community focused

Application anatomy

ASP.NET 5 applications are built and run using the new [.NET Execution Environment \(DNX\)](#). Every ASP.NET 5 project is a [DNX project](#). ASP.NET 5 integrates with DNX through the [ASP.NET Application Hosting](#) package.

ASP.NET 5 applications are defined using a public `Startup` class:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app)
    {
    }
}
```

The `ConfigureServices` method defines the services used by your application and the `Configure` method is used to define what middleware makes up your request pipeline. See [Understanding ASP.NET 5 Web Apps](#) for more details.

Services

A service is a component that is intended for common consumption in an application. Services are made available through dependency injection. ASP.NET 5 includes a simple built-in inversion of control (IoC) container that supports constructor injection by default, but can be easily replaced with your IoC container of choice. See [Dependency Injection](#) for more details.

Services in ASP.NET 5 come in three varieties: singleton, scoped and transient. Transient services are created each time they're requested from the container. Scoped services are created only if they don't already exist in the current scope. For Web applications, a container scope is created for each request, so you can think of scoped services as per request. Singleton services are only ever created once.

Middleware

In ASP.NET 5 you compose your request pipeline using [Middleware](#). ASP.NET 5 middleware perform asynchronous logic on an `HttpContext` and then optionally invoke the next middleware in the sequence or termi-

nate the request directly. You generally “Use” middleware by invoking a corresponding extension method on the `IApplicationBuilder` in your `Configure` method.

ASP.NET 5 comes with a rich set of prebuilt middleware:

- [Working with Static Files](#)
- [Routing](#)
- [Diagnostics](#)
- [Authentication](#)

You can also author your own [custom middleware](#).

You can use any [OWIN](#)-based middleware with ASP.NET 5. See [OWIN](#) for details.

Servers

The ASP.NET Application Hosting model does not directly listen for requests, but instead relies on an HTTP server implementation to surface the request to the application as a set of feature interfaces that can be composed into an `HttpContext`.

ASP.NET 5 includes server support for running on IIS or self-hosting in your own process. On Windows you can host your application outside of IIS using the [WebListener](#) server, which is based on HTTP.sys. You can also host your application on a non-Windows environment using the cross-platform [Kestrel](#) web server.

Web root

The Web root of your application is the root location in your project from which HTTP requests are handled (ex. handling of static file requests). The Web root of an ASP.NET 5 application is configured using the “webroot” property in your `project.json` file.

Configuration

ASP.NET 5 uses a new configuration model for handling of simple name-value pairs that is not based on `System.Configuration` or `web.config`. This new configuration model pulls from an ordered set of configuration providers. The built-in configuration providers support a variety of file formats (XML, JSON, INI) and also environment variables to enable environment-based configuration. You can also write your own custom configuration providers. Environments, like Development and Production, are a first-class notion in ASP.NET 5 and can also be set up using environment variables:

```
// Setup configuration sources.
var configuration = new Configuration()
    .AddJsonFile("config.json")
    .AddJsonFile($"config.{env.EnvironmentName}.json", optional: true);

if (env.IsEnvironment("Development"))
{
    // This reads the configuration keys from the secret store.
    // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=53271
    configuration.AddUserSecrets();
}
configuration.AddEnvironmentVariables();
Configuration = configuration;
```

See [Configuration](#) for more details on the new configuration system and [Working with Multiple Environments](#) for details on how to work with environments in ASP.NET 5.

Client-side development

ASP.NET 5 is designed to integrate seamlessly with a variety of client-side frameworks, including [AngularJS](#), [KnockoutJS](#) and [Bootstrap](#). See [Client-Side Development](#) for more details.

2.3.2 Introducing .NET Core

By [Steve Smith](#)

.NET Core is a small, optimized runtime that can be targeted by ASP.NET 5 applications. In fact, the new ASP.NET 5 project templates target .NET Core by default, in addition to the .NET Framework. Learn what targeting .NET Core means for your ASP.NET 5 application.

In this article:

- *[What is .NET Core?](#)*
- *[Motivation Behind .NET Core](#)*
- *[Building Applications with .NET Core](#)*
- *[.NET Core and NuGet](#)*
- *[Additional Reading](#)*

What is .NET Core

.NET Core 5 is a modular runtime and library implementation that includes a subset of the .NET Framework. Currently it is feature complete on Windows, and in-progress builds exist for both Linux and OS X. .NET Core consists of a set of libraries, called “CoreFX”, and a small, optimized runtime, called “CoreCLR”. .NET Core is open-source, so you can follow progress on the project and contribute to it on [GitHub](#):

- [.NET Core Libraries \(CoreFX\)](#)
- [.NET Core Common Language Runtime \(CoreCLR\)](#)

The CoreCLR runtime (Microsoft.CoreCLR) and CoreFX libraries are distributed via NuGet. The CoreFX libraries are factored as individual NuGet packages according to functionality, named “System.[module]” on [nuget.org](#).

One of the key benefits of .NET Core is its portability. You can package and deploy the CoreCLR with your application, eliminating your application’s dependency on an installed version of .NET (e.g. .NET Framework on Windows). You can host multiple applications side-by-side using different versions of the CoreCLR, and upgrade them individually, rather than being forced to upgrade all of them simultaneously.

CoreFX has been built as a componentized set of libraries, each requiring the minimum set of library dependencies (e.g. System.Collections only depends on System.Runtime, not System.Xml). This approach enables minimal distributions of CoreFX libraries (just the ones you need) within an application, alongside CoreCLR. CoreFX includes collections, console access, diagnostics, IO, LINQ, JSON, XML, and regular expression support, just to name a few libraries. Another benefit of CoreFX is that it allows developers to target a single common set of libraries that are supported by multiple platforms.

Motivation Behind .NET Core

When .NET first shipped in 2002, it was a single framework, but it didn’t take long before the .NET Compact Framework shipped, providing a smaller version of .NET designed for mobile devices. Over the years, this exercise was repeated multiple times, so that today there are different flavors of .NET specific to different platforms. Add to this the further platform reach provided by Mono and Xamarin, which target Linux, Mac, and native iOS and Android devices. For each platform, a separate vertical stack consisting of runtime, framework, and app model is required to

develop .NET applications. One of the primary goals of .NET Core is to provide a single, modular, cross-platform version of .NET that works the same across all of these platforms. Since .NET Core is a fully open source project, the Mono community can benefit from CoreFX libraries. .NET Core will not replace Mono, but it will allow the Mono community to reference and share, rather than duplicate, certain common libraries, and to contribute directly to CoreFX, if desired.

In addition to being able to target a variety of different device platforms, there was also pressure from the server side to reduce the overall footprint, and more importantly, surface area, of the .NET Framework. By factoring the CoreFX libraries and allowing individual applications to pull in only those parts of CoreFX they require (a so-called “pay-for-play” model), server-based applications built with ASP.NET 5 can minimize their dependencies. This, in turn, reduces the frequency with which patches and updates to the framework will impact these applications, since only changes made to the individual pieces of CoreFX leveraged by the application will impact the application. A smaller deployment size for the application is a side benefit, and one that makes more of a difference if many applications are deployed side-by-side on a given server.

Note: The overall size of .NET Core doesn’t intend to be smaller than the .NET Framework over time, but since it is pay-for-play, most applications that utilize only parts of CoreFX will have a smaller deployment footprint.

Building Applications with .NET Core

.NET Core can be used to build a variety of applications using different application models including Web applications, console applications and native mobile applications. The .NET Execution Environment (DNX) provides a cross-platform runtime host that you can use to build .NET Core based applications that can run on Windows, Mac and Linux and is the foundation for running ASP.NET applications on .NET Core. Applications running on DNX can target the .NET Framework or .NET Core. In fact, DNX projects can be cross-compiled, targeting both of these frameworks in a single project, and this is how the project templates ship with Visual Studio 2015. For example, the `frameworks` section of `project.json` in a new ASP.NET 5 web project will target `dnx451` and `dnxcore50` by default:

```
"frameworks": {  
  "dnx451": { },  
  "dnxcore50": { }  
},
```

`dnx451` represents the .NET Framework, while `dnxcore50` represents .NET Core 5 (5.0). You can use compiler directives (`#if`) to check for symbols that correspond to the two frameworks: `DNX451` and `DNXCORE50`. If for instance you have code that uses resources that are not available as part of .NET Core, you can surround them in a conditional compilation directive:

```
#if DNX451  
    // utilize resource only available with .NET Framework  
#endif
```

The recommendation from the ASP.NET team is to target both frameworks with new applications. If you want to only target .NET Core, remove `dnx451`, or only target .NET Framework, remove `dnxcore50`, from the `frameworks` listed in `project.json`. Note that ASP.NET 4.6 and earlier target and require the .NET Framework, as they always have.

.NET Core and NuGet

Using NuGet allows for much more agile usage of the individual libraries that comprise .NET Core. It also means that an application can list a collection of NuGet packages (and associated version information) and this will comprise both system/framework as well as third-party dependencies required. Further, third-party dependencies can now also express their specific dependencies on framework features, making it much easier to ensure the proper packages and versions are pulled together during the development and build process.

If, for example, you need to use immutable collections, you can install the `System.Collections.Immutable` package via NuGet. The NuGet version will also align with the assembly version, and will use [semantic versioning](#).

Note: Although CoreFX will be made available as a fairly large number of individual NuGet packages, it will continue to ship periodically as a full unit that Microsoft has tested as a whole. These distributions will most likely ship at a lower cadence than individual packages, allowing time to perform necessary testing, fixes, and the distribution process.

Summary

.NET Core is a modular, streamlined subset of the .NET Framework and CLR. It is fully open-source and provides a common set of libraries that can be targeted across numerous platforms. Its factored approach allows applications to take dependencies only on those portions of the CoreFX that they use, and the smaller runtime is ideal for deployment to both small devices (though it doesn't yet support any) as well as cloud-optimized environments that need to be able to run many small applications side-by-side. Support for targeting .NET Core is built into the ASP.NET 5 project templates that ship with Visual Studio 2015.

Additional Reading

Learn more about .NET Core:

- [Immo Landwerth Explains .NET Core](#)
- [What is .NET Core 5 and ASP.NET 5](#)
- [.NET Core 5 on dotnetfoundation.org](#)
- [.NET Core is Open Source](#)
- [.NET Core on GitHub](#)

2.3.3 DNX Overview

By [Daniel Roth](#)

What is the .NET Execution Environment?

The .NET Execution Environment (DNX) is a software development kit (SDK) and runtime environment that has everything you need to build and run .NET applications for Windows, Mac and Linux. It provides a host process, CLR hosting logic and managed entry point discovery. DNX was built for running cross-platform ASP.NET Web applications, but it can run other types of .NET applications, too, such as cross-platform console apps.

Why build DNX?

Cross-platform .NET development DNX provides a consistent development and execution environment across multiple platforms (Windows, Mac and Linux) and across different .NET flavors (.NET Framework, .NET Core and Mono). With DNX you can develop your application on one platform and run it on a different platform as long as you have a compatible DNX installed on that platform. You can also contribute to DNX projects using your development platform and tools of choice.

Build for .NET Core DNX dramatically simplifies the work needed to develop cross-platform applications using .NET Core. It takes care of hosting the CLR, handling dependencies and bootstrapping your application. You can easily define projects and solutions using a lightweight JSON format (*project.json*), build your projects and publish them for distribution.

Package ecosystem Package managers have completely changed the face of modern software development and DNX makes it easy to create and consume packages. DNX provides tools for installing, creating and managing NuGet packages. DNX projects simplify building NuGet packages by cross-compiling for multiple target frameworks and can output NuGet packages directly. You can reference NuGet packages directly from your projects and transitive dependencies are handled for you. You can also build and install development tools as packages for your project and globally on a machine.

Open source friendly DNX makes it easy to work with open source projects. With DNX projects you can easily replace an existing dependency with its source code and let DNX compile it in-memory at runtime. You can then debug the source and modify it without having to modify the rest of your application.

Projects

A DNX project is a folder with a *project.json* file. The name of the project is the folder name. You use DNX projects to build NuGet packages. The *project.json* file defines your package metadata, your project dependencies and which frameworks you want to build for:

```
1 {
2   "version": "1.0.0-*",
3   "description": "ClassLibrary1 Class Library",
4   "authors": [ "author" ],
5   "tags": [ "" ],
6   "projectUrl": "",
7   "licenseUrl": "",
8
9   "dependencies": {
10    "System.Collections": "4.0.10-beta-23109",
11    "System.Linq": "4.0.0-beta-23109",
12    "System.Threading": "4.0.10-beta-23109",
13    "System.Runtime": "4.0.10-beta-23109",
14    "Microsoft.CSharp": "4.0.0-beta-23109"
15  },
16
17  "frameworks": {
18    "dotnet": { }
19  }
20 }
```

All the files in the folder are by default part of the project unless explicitly excluded in *project.json*.

You can also define commands as part of your project that can be executed (see [Commands](#)).

You specify which frameworks you want to build for under the “frameworks” property. DNX will cross-compile for each specified framework and create the corresponding lib folder in the built NuGet package.

You can use the .NET Development Utility (DNU) to build, package and publish DNX projects. Building a project produces the binary outputs for the project. Packaging produces a NuGet package that can be uploaded to a package feed (for example, <http://nuget.org>) and then consumed. Publishing collects all required runtime artifacts (the required DNX and packages) into a single folder so that it can be deployed as an application.

For more details on working with DNX projects see [Working with DNX Projects](#).

Dependencies

Dependencies in DNX consist of a name and a version number. Version numbers should follow [Semantic Versioning](#). Typically dependencies refer to an installed NuGet package or to another DNX project. Project references are resolved using peer folders to the current project or project paths specified using a *global.json* file at the solution level:


```

1 {
2   "projects": [ "src", "test" ],
3   "sdk": {
4     "version": "1.0.0-beta6"
5   }
6 }

```

The *global.json* file also defines the minimum DNX version (“sdk” version) needed to build the project.

Dependencies are transitive in that you only need to specify your top level dependencies. DNX will handle resolving the entire dependency graph for you using the installed NuGet packages. Project references are resolved at runtime by building the referenced project in memory. This means you have the full flexibility to deploy your DNX application as package binaries or as source code.

Packages and feeds

For package dependencies to resolve they must first be installed. You can use DNU to install a new package into an existing project or to restore all package dependencies for an existing project. The following command downloads and installs all packages that are listed in the *project.json* file:

```

dnu restore

```

Packages are restored using the configured set of package feeds. You configure the available package feeds using [NuGet configuration files \(NuGet.config\)](#).

Commands

A command is a named execution of a .NET entry point with specific arguments. You can define commands in your *project.json* file:

```

1 "commands": {
2   "web": "Microsoft.AspNet.Hosting --config hosting.ini",
3   "ef": "EntityFramework.Commands"
4 },

```

You can then use DNX to execute the commands defined by your project, like this:

```

dnx . web

```

Commands can be built and distributed as NuGet packages. You can then use DNU to install commands globally on a machine:

```

dnu commands install MyCommand

```

For more information on using and creating commands see [Using Commands](#).

Application Host

The DNX application host is typically the first managed entry point invoked by DNX and is responsible for handling dependency resolution, parsing *project.json*, providing additional services and invoking the application entry point.

Alternatively, you can have DNX invoke your application’s entry point directly. Doing so requires that your application be fully built and all dependencies located in a single directory. Using DNX without using the DNX Application Host is not common.

The DNX application host provides a set of services to applications through dependency injection (for example, *IServiceProvider*, *IApplicationEnvironment* and *ILoggerFactory*). Application host services can be injected in the constructor of the class for your *Main* entry point or as additional method parameters to your *Main* entry point.

Compile Modules

Compile modules are an extensibility point that let you participate in the DNX compilation process. You implement a compile module by implementing the `ICompileModule` interface and putting your compile module in a compiler/preprocess or compiler/postprocess in your project.

DNX Version Manager

You can install multiple DNX versions and flavors on your machine. To install and manage different DNX versions and flavors you use the .NET Version Manager (DNVM). DNVM lets you list the different DNX versions and flavors on your machine, install new ones and switch the active DNX.

See [Getting Started](#) for instructions on how to acquire and install DNVM for your platform.

2.3.4 Introduction to NuGet

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

NuGet is the package management system used by the .NET Execution Environment and ASP.NET 5. You can learn all about NuGet and working with NuGet packages at <https://docs.nuget.org>.

2.3.5 Basics of Web Development

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.3.6 Understanding ASP.NET 5 Web Apps

By [Steve Smith](#)

ASP.NET 5 introduces several new fundamental concepts of web programming that are important to understand in order to productively create web apps. These concepts are not necessarily new to web programming in general, but are new to ASP.NET and thus are likely new to many developers whose experience with web programming has mainly been using ASP.NET and Visual Studio.

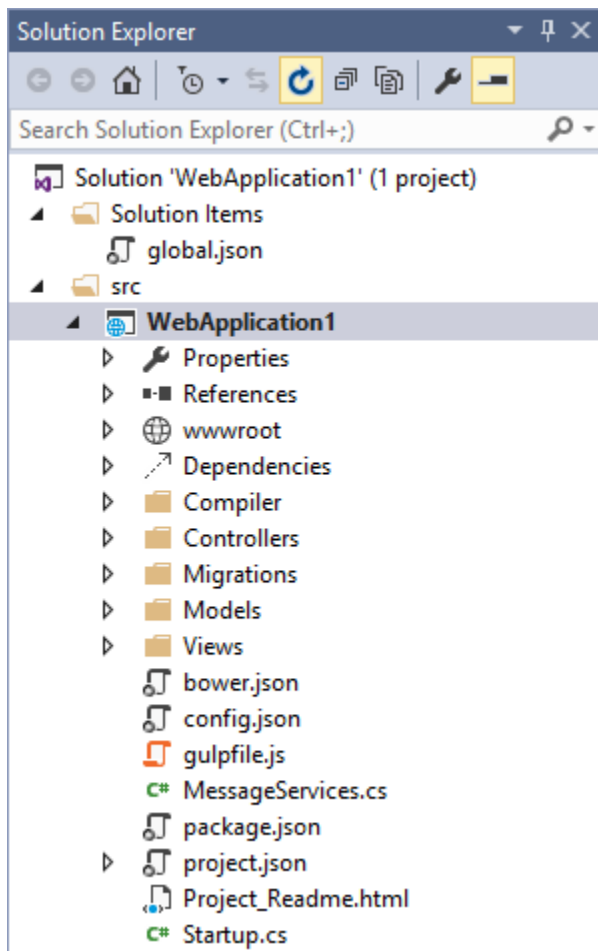
In this article:

- *ASP.NET Project Structure*
- *Framework Target*
- *The project.json File*
- *The global.json File*
- *The wwwroot folder*
- *Client Side Dependency Management*

- *Server Side Dependency Management*
- *Configuring the Application*
- *Application Startup*

ASP.NET Project Structure

ASP.NET 5's project structure adds new concepts and replaces some legacy elements found in previous versions of ASP.NET projects. The new default web project template creates a solution and project structure like the one shown here:

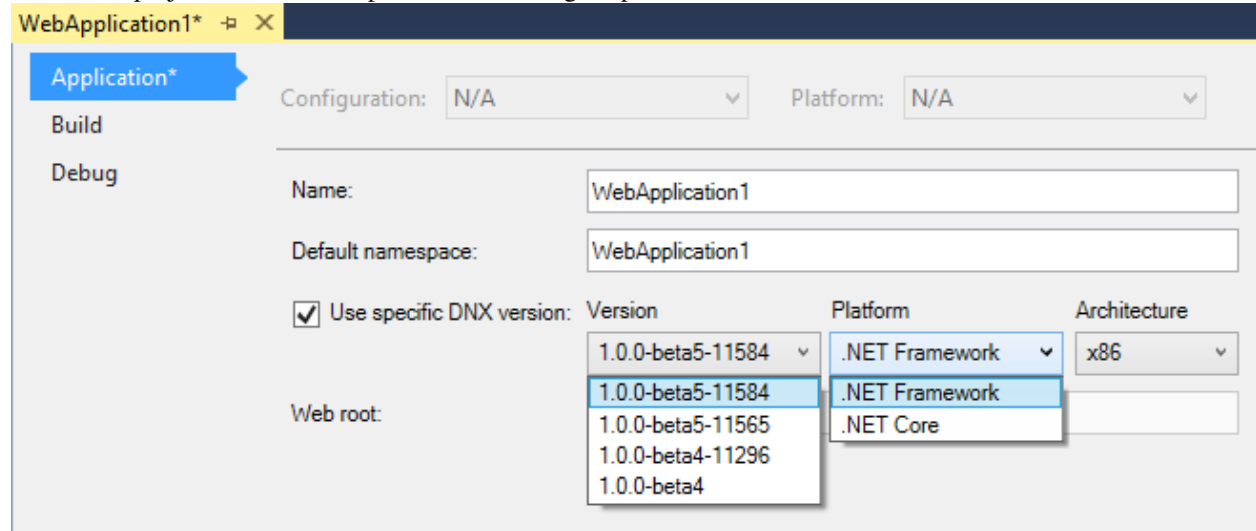


The first thing you may notice about this new structure is that it includes a Solution Items folder with a *global.json* file, and the web project itself is located within a *src* folder within the solution. The new structure also includes a special *wwwroot* folder and a Dependencies section in addition to the References section that was present in past versions of ASP.NET (but which has been updated in this version). In the root the project there are also several new files such as *bower.json*, *config.json*, *gulpfile.js*, *package.json*, *project.json*, and *Startup.cs*. You may notice that the files *global.asax*, *packages.config*, and *web.config* are gone. In previous versions of ASP.NET, a great deal of application configuration was stored in these files and in the project file. In ASP.NET 5, this information and logic has been refactored into files that are generally smaller and more focused.

Framework Target

ASP.NET 5 can target multiple frameworks, allowing the application to be deployed into different hosting environments. By default applications will target the full version of .NET, but they can also target the .NET Core. Most legacy apps will target the full ASP.NET 5, at least initially, since they're likely to have dependencies that include framework base class libraries that are not available in .NET Core today. .NET Core is a small version of the .NET framework that is optimized for web apps and supports Linux and Mac environments. It can be deployed with an application, allowing multiple apps on the same server to target different versions of .NET Core. It is also modular, allowing additional functionality to be added only when it is required, as separate NuGet packages ([learn more about .NET Core](#)).

You can see which framework is currently being targeted in the web application project's properties, by right-clicking on the web project in Solution Explorer and selecting Properties:



By default, the checkbox for *Use specific DNX version* is unchecked. To target a specific version, check the box and choose the appropriate *Version*, *Platform*, and *Architecture*.

The project.json File

The project.json file is new to ASP.NET 5. It is used to define the project's *server side dependencies* (discussed below), as well as other project-specific information. The sections included in *project.json* by default with the default web project template are shown below.

```
{
  "webroot": "wwwroot",
  "version": "1.0.0-*",
  "dependencies": ...,
  "commands": ...,
  "frameworks": ...,
  "exclude": ...,
  "bundleExclude": ...,
  "scripts": ...
}
```

The **webroot** section specifies the folder that should act as the root of the web site, which by convention defaults to *the wwwroot folder*. The version property specifies the current version of the project. You can also specify other metadata about the project such as **authors** and **description**.

ASP.NET 5 has a great deal of support for command line tooling, and the **commands** section allows you to configure what certain command line commands should do (for instance, launch a web site or run tests).

```
"commands": {
  "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5000",
  "gen": "Microsoft.Framework.CodeGeneration",
  "ef": "EntityFramework.Commands"
},
```

The **frameworks** section designates which targeted frameworks will be built, and what dependencies need to be included. For instance, if you were using LINQ and collections, you could ensure these were included with your .NET Core build by adding them to the `dnxcore50` list of dependencies as shown.

See Also: [Diagnosing dependency issues with ASP.NET 5](#)

The **exclude** section is used to identify files and folders that should be excluded from builds. Likewise, **bundleExclude** is used to identify content portions of the project that should be excluded when bundling the site (for example, in production).

```
"exclude": [
  "wwwroot",
  "node_modules",
  "bower_components"
],

"bundleExclude": [
  "node_modules",
  "bower_components",
  "**.kproj",
  "**.user",
  "**.vsspscc"
],
```

The **scripts** section is used to specify when certain build automation scripts should run. Visual Studio now has built-in support for running such scripts before and after certain events. The default ASP.NET project template has scripts in place to run during *postrestore* and *prepare* that install *client side dependencies* using `npm` and `bower`.

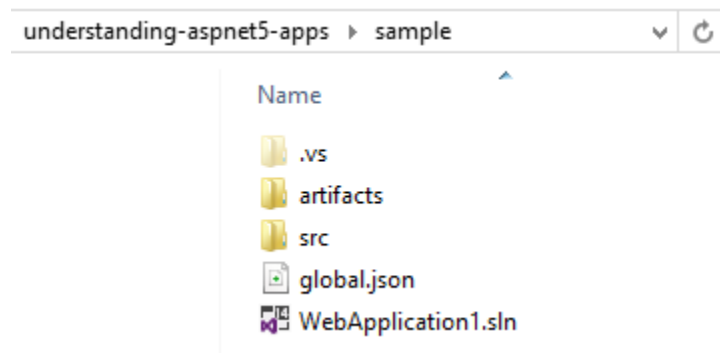
```
"scripts": {
  "postrestore": [ "npm install" ],
  "prepare": [ "grunt bower:install" ]
}
```

The global.json File

The `global.json` file is used to configure the solution as a whole. It includes just two sections, `projects` and `sdk` by default.

```
{
  "projects": [ "src", "test" ],
  "sdk": {
    "version": "1.0.0-beta6"
  }
}
```

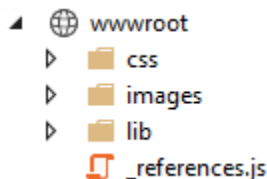
The `projects` property designates which folders contain source code for the solution. By default the project structure places source files in a `src` folder, allowing build artifacts to be placed in a sibling folder, making it easier to exclude such things from source control.



The `sdk` property specifies the version of the DNX (.Net Execution Environment) that Visual Studio will use when opening the solution. It's set here, rather than in `project.json`, to avoid scenarios where different projects within a solution are targeting different versions of the SDK.

The wwwroot Folder

In previous versions of ASP.NET, the root of the project was typically the root of the website. If you placed a `Default.aspx` file in the project root of an early version of ASP.NET, it would load if a request was made to the web application's root. In later versions of ASP.NET, support for routing was added, making it possible to decouple the locations of files from their corresponding URLs (thus, `HomeController` in the `Controllers` folder is able to serve requests made to the root of the site, using a default route implementation). However, this routing was used only for ASP.NET-specific application logic, not static files needed by the client to properly render the resulting page. Resources like images, script files, and stylesheets were generally still loaded based on their location within the file structure of the application, based off of the root of the project.



The file based approach presented several problems. First, protecting sensitive project files required framework-level protection of certain filenames or extensions, to prevent having things like `web.config` or `global.asax` served to a client in response to a request. Having to specifically block access (also known as blacklisting) to certain files is much less secure than granting access only to those files which should be accessible (also known as whitelisting). Typically different versions were required for dev/test and production (for example `web.config`). Scripts would typically be referenced individually and in a readable format during development, but would be minified and bundled together

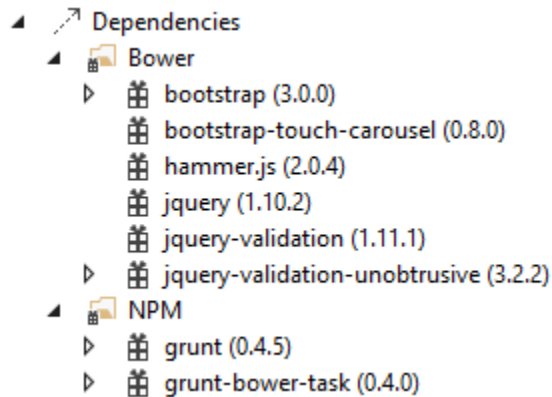
when deployed for production. It's beneficial to deploy only production files to production, but handling these kinds of scenarios was difficult with the previous file structure.

Enter the *wwwroot* folder in ASP.NET 5. The *wwwroot* folder represents the actual root of the web app when running on a web server. Static files, like *config.json*, which are not located in *wwwroot* will never be accessible, and there is no need to create special rules to block access to sensitive files. Instead of blacklisting access to sensitive files, a more secure whitelist approach is taken whereby only those files in the *wwwroot* folder are accessible via web requests.

In addition to the security benefits, the *wwwroot* folder also simplifies common tasks like bundling and minification, which can now be more easily incorporated into a standard build process and automated using tools like Grunt.

Client Side Dependency Management

The Dependencies folder contains two subfolders: Bower and NPM. These folders correspond to two package managers by the same names, and they're used to pull in client-side dependencies and tools (e.g. jQuery, bootstrap, or grunt). Expanding the folders reveals which dependencies are currently managed by each tool, and the current version being used by the project.



The bower dependencies are controlled by the *bower.json* file. You'll notice that each of the items listed in the figure above correspond to dependencies listed in *bower.json*:



```

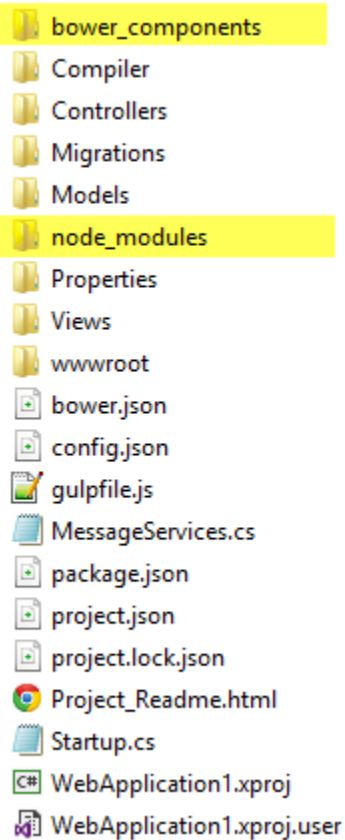
{
  "name": "WebApplication",
  "private": true,
  "dependencies": {
    "bootstrap": "3.0.0",
    "jquery": "1.10.2",
    "jquery-validation": "1.11.1",
    "jquery-validation-unobtrusive": "3.2.2",
    "hammer.js": "2.0.4",
    "bootstrap-touch-carousel": "0.8.0"
  },
  "exportsOverride": {
    "bootstrap": {
      "js": "dist/js/*.js",
      "css": "dist/css/*.css",
      "fonts": "dist/fonts/*.font"
    },
    "bootstrap-touch-carousel": {
      "js": "dist/js/*.js",
      "css": "dist/css/*.css"
    },
    "jquery": {
      "": "jquery.{js,min.js,min.map}"
    },
    "jquery-validation": {
      "": "jquery.validate.js"
    },
    "jquery-validation-unobtrusive": {
      "": "jquery.validate.unobtrusive.{js,min.js}"
    },
    "hammer": {
      "": "hammer.{js,min.js}"
    }
  }
}

```

Each dependency is then further configured in its own section within the *bower.json* file, indicating how it should be deployed to the *wwwroot* folder when the bower task is executed.

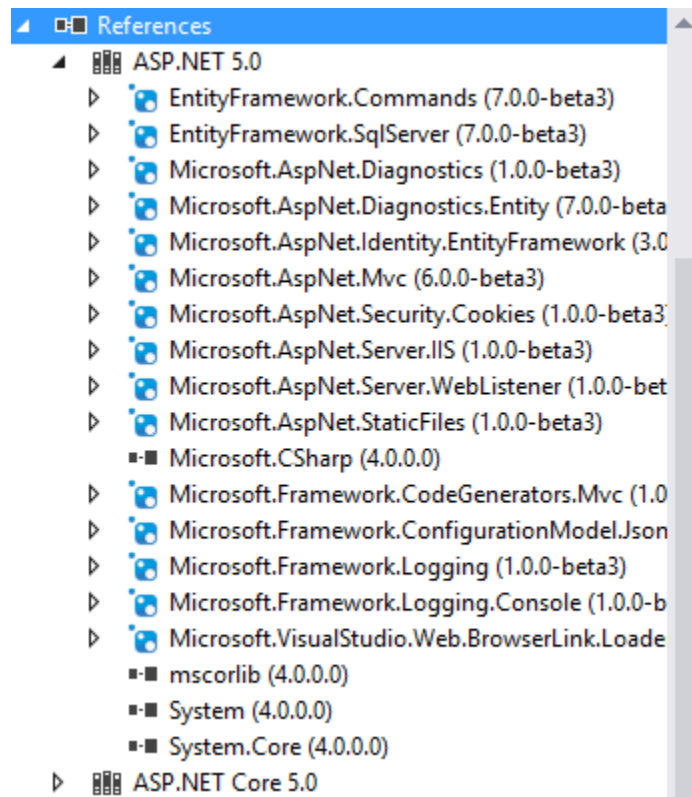
By default, the bower task is executed using gulp, which is configured in *gulpfile.js*. The current web template's gulpfile includes tasks for copying and cleaning script and CSS files from the bower folder to a */lib* folder in *wwwroot*.

Name



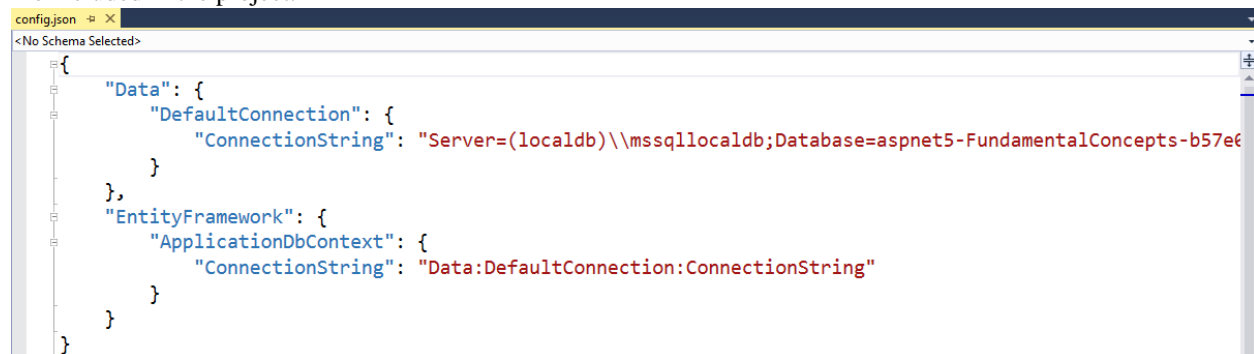
Server Side Dependency Management

The *References* folder details the server-side references for the project. It should be familiar to ASP.NET developers, but it has been modified to differentiate between references for different framework targets, such as the full ASP.NET 5.0 vs. ASP.NET Core 5.0. Within each framework target, you will find individual references, with icons indicating whether the reference is to an assembly, a NuGet package, or a project. Note that these dependencies are checked at compile time, with missing dependencies downloaded from the configured NuGet package source (specified under Options – NuGet Package Manager – Package Sources).



Configuring the Application

ASP.NET 5 no longer stores configuration settings in XML files (*web.config* and *machine.config*). Configuration is now stored in *config.json*, which was designed specifically for storing app configuration settings. The default ASP.NET project template includes Entity Framework, and so specifies the database connection string details in the *config.json* file included in the project.



Individual entries within *config.json* are not limited to name-value pairs, but can specify rich objects. Entries can also reference other entries, as you can see the EF configuration does above.

There's nothing special about the *config.json* filename - it's specified by name in *Startup.cs*. You can add as many different configuration files as makes sense for your app, rather than having to add to an ever-growing *web.config* file. You're also not limited to just JSON-formatted files - you can still use XML or even .INI files if you prefer.

Accessing configuration data from your app is best done by injecting the *IConfiguration* interface into your controller, and then simply calling its *Get* method with the name of the configuration element you need. For example, to store the application name in config and display it on the About page, you would need to make three changes to the default

project. First, add the entry to *config.json*.



Next, make sure ASP.NET knows what to return when a constructor requires an instance of *IConfiguration* . In this case, we can specify that the configuration value is a singleton, since we don't expect it to change throughout the life of the application. We'll address *Startup.cs* in a moment, but for this step just add one line to the end of the *ConfigureServices()* method in *Startup.cs*:

```
services.AddSingleton(_ => Configuration);
```

The third and final step is to specify that your controller expects an *IConfiguration* instance via its constructor. Following the [Explicit Dependencies Principle](#) with your classes is a good habit to get into, and will allow ASP.NET 5's built-in support for Dependency Injection to work correctly. Assign the instance to a local field, and then access the configuration value by calling the *Get* method on this instance.

You will need to ensure you have this using statement:

```
using Microsoft.Framework.ConfigurationModel;
```

Then, update the controller as shown:

```

public class HomeController : Controller
{
    private readonly IConfiguration _config;
    public HomeController(IConfiguration config)
    {
        _config = config;
    }

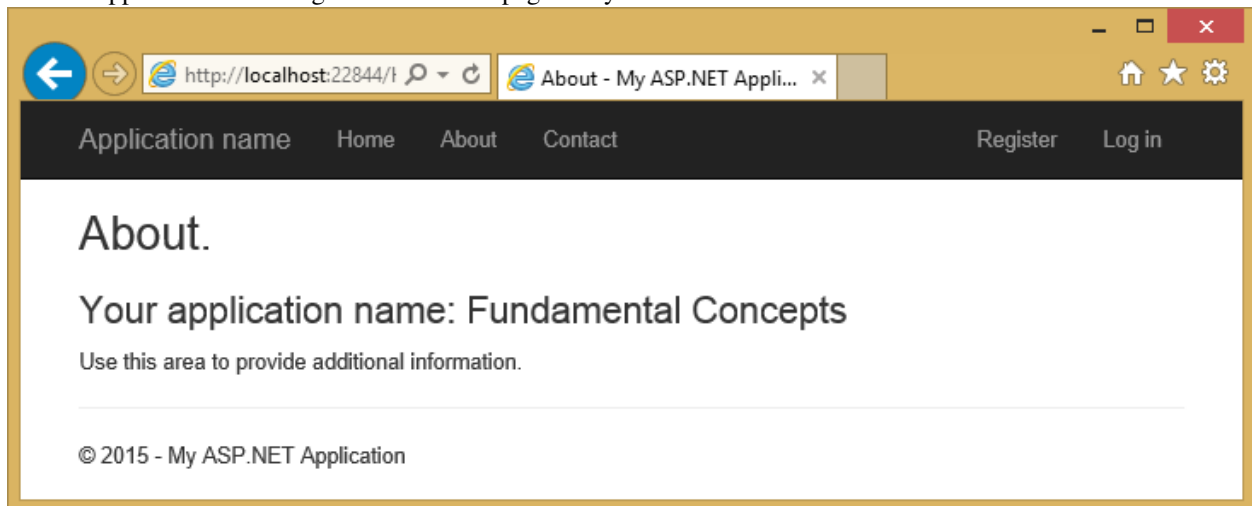
    public IActionResult Index()...

    public IActionResult About()
    {
        string appName = _config.Get("ApplicationName");
        ViewBag.Message = "Your application name: " + appName;

        return View();
    }
}

```

Run the application and navigate to the About page and you should see the result.



Application Startup

ASP.NET 5 has decomposed its feature set into a variety of modules that can be individually added to a web app. This allows for lean web apps that do not import or bring in features they don't use. When your ASP.NET app starts, the ASP.NET runtime calls `Configure` in the `Startup` class. If you create a new ASP.NET web project using the Empty template, you will find that the `Startup.cs` file has only a couple lines of code. The default Web project's `Startup` class wires up configuration, MVC, EF, Identity services, logging, routes, and more. It provides a good example for how to configure the services used by your ASP.NET app. There are three parts to the sample startup class: a constructor, `ConfigureServices`, and `Configure`. The `Configure` method is called after `ConfigureServices` and is used to configure middleware.

The constructor specifies how configuration will be handled by the app. Configuration is a property of the `Startup`

class and can be read from a variety of file formats as well as from environment variables. The default project template wires up Configuration to use a *config.json* and environment variables.

```
public Startup(IHostingEnvironment env)
{
    // Setup configuration sources.
    Configuration = new Configuration()
        .AddJsonFile("config.json")
        .AddEnvironmentVariables();
}
```

The `ConfigureServices` method is used to specify which services are available to the app. The default template uses helper methods to add a variety of services used for EF, Identity, and MVC. This is also where you can add your own services, as we did above to expose the configuration as a service. The complete `ConfigureServices` method, including the call to add `Configuration` as a singleton, is shown here:

```
// This method gets called by the runtime.
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddEntityFramework(Configuration)
        .AddSqlServer()
        .AddDbContext<ApplicationDbContext>();

    // Add Identity services to the services container.
    services.AddIdentity<ApplicationUser, IdentityRole>(Configuration)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    // Add MVC services to the services container.
    services.AddMvc();

    services.AddSingleton(_ => Configuration);
}
```

Finally, the `Configure` method will be called by the runtime after `ConfigureServices`. In the sample project, `Configure` is used to wire up a console logger, add several useful features for the development environment, add support for static files, Identity, and MVC routing. Note that adding Identity and MVC in `ConfigureServices` isn't sufficient - they also need to be configured in the request pipeline via these calls in `Configure`.

```
// Configure is called after ConfigureServices is called.
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerfactory)
{
    // Configure the HTTP request pipeline.
    // Add the console logger.
    loggerfactory.AddConsole();

    // Add the following to the request pipeline only in development environment.
    if (string.Equals(env.EnvironmentName, "Development", StringComparison.OrdinalIgnoreCase))
    {
        app.UseBrowserLink();
        app.UseErrorPage(ErrorPageOptions.ShowAll);
        app.UseDatabaseErrorPage(DatabaseErrorPageOptions.ShowAll);
    }
    else
    {
        // Add Error handling middleware which catches all application specific errors and
        // send the request to the following path or controller action.
        app.UseExceptionHandler("/Home/Error");
    }
}
```

```
// Add static files to the request pipeline.
app.UseStaticFiles();

// Add cookie-based authentication to the request pipeline.
app.UseIdentity();

// Add MVC to the request pipeline.
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller}/{action}/{id?}",
        defaults: new { controller = "Home", action = "Index" });
});
}
```

As you can see, configuring which services are available and how the request pipeline is configured is now done completely in code in the `Startup` class, as opposed to using HTTP Modules and Handlers managed via *web.config*.

Summary

ASP.NET 5 introduces a few concepts that didn't exist in previous versions of ASP.NET. Rather than working with *web.config*, *packages.config*, and a variety of project properties stored in the *.csproj/.vbproj* file, developers can now work with specific files and folders devoted to specific purposes. Although at first there is some learning curve, the end result is more secure, more maintainable, works better with source control, and has better separation of concerns than the approach used in previous versions of ASP.NET.

2.4 Fundamentals

2.4.1 Application Startup

By [Steve Smith](#)

ASP.NET 5 provides complete control of how individual requests are handled by your application. The `Startup` class is the entry point to the application, setting up configuration and wiring up services the application will use. Developers configure a request pipeline in the `Startup` class that is used to handle all requests made to the application.

In this article:

- [The `Startup` class](#)
- [The `Configure` method](#)
- [The `ConfigureServices` method](#)

The `Startup` class

In ASP.NET 5, the `Startup` class provides the entry point for an application. It's possible to have environment-specific startup classes and methods (see [Working with Multiple Environments](#)), but regardless, one `Startup` class will serve as the entry point for the application. ASP.NET searches the primary assembly for a class named `Startup` (in any namespace). You can specify a different assembly to search using the *Hosting:Application* configuration key. It doesn't matter whether the class is defined as `public`; ASP.NET will still load it if it conforms to the naming convention. If there are multiple `Startup` classes, this will not trigger an exception. ASP.NET will select one based

on its namespace (matching the project's root namespace first, otherwise using the class in the alphabetically first namespace).

The Startup class can optionally accept dependencies in its constructor that are provided through [dependency injection](#). Typically, the way an application will be configured is defined within its Startup class's constructor (see [Configuration](#)). The Startup class must define a `Configure` method, and may optionally also define a `ConfigureServices` method, which will be called when the application is started.

The Configure method

The `Configure` method is used to specify how the ASP.NET application will respond to individual HTTP requests. At its simplest, you can configure every request to receive the same response. However, most real-world applications require more functionality than this. More complex sets of pipeline configuration can be encapsulated in [middleware](#) and added using extension methods on `IApplicationBuilder`.

Your `Configure` method must accept an `IApplicationBuilder` parameter. Additional services, like `IHostingEnvironment` and `ILoggerFactory` may also be specified, in which case these services will be [injected](#) by the server if they are available. In the following example from the default web site template, you can see several extension methods are used to configure the pipeline with support for [BrowserLink](#), error pages, static files, ASP.NET MVC, and Identity.

```

1 // Configure is called after ConfigureServices is called.
2 public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
3 {
4     loggerFactory.MinimumLevel = LogLevel.Information;
5     loggerFactory.AddConsole();
6
7     // Configure the HTTP request pipeline.
8
9     // Add the following to the request pipeline only in development environment.
10    if (env.IsDevelopment())
11    {
12        app.UseBrowserLink();
13        app.UseErrorPage();
14        app.UseDatabaseErrorPage(DatabaseErrorPageOptions.ShowAll);
15    }
16    else
17    {
18        // Add Error handling middleware which catches all application specific errors and
19        // sends the request to the following path or controller action.
20        app.UseExceptionHandler("/Home/Error");
21    }
22
23    // Add static files to the request pipeline.
24    app.UseStaticFiles();
25
26    // Add cookie-based authentication to the request pipeline.
27    app.UseIdentity();
28
29    // Add authentication middleware to the request pipeline. You can configure options such as Id and
30    // For more information see http://go.microsoft.com/fwlink/?LinkID=532715
31    // app.UseFacebookAuthentication();
32    // app.UseGoogleAuthentication();
33    // app.UseMicrosoftAccountAuthentication();
34    // app.UseTwitterAuthentication();
35
36    // Add MVC to the request pipeline.

```

```
37 app.UseMvc(routes =>
38 {
39     routes.MapRoute(
40         name: "default",
41         template: "{controller=Home}/{action=Index}/{id?}");
42
43     // Uncomment the following line to add a route for porting Web API 2 controllers.
44     // routes.MapWebApiRoute("DefaultApi", "api/{controller}/{id?}");
45 });
46 }
```

You can see what each of these extensions does by examining the source. For instance, the `UseMvc` extension method is defined in `BuilderExtensions` available on [GitHub](#). Its primary responsibility is to ensure MVC was added as a service (in `ConfigureServices`) and to correctly set up routing for an ASP.NET MVC application.

You can learn all about middleware and using `IApplicationBuilder` to define your request pipeline in the [Middleware](#) topic.

The `ConfigureServices` method

Your `Startup` class can optionally include a `ConfigureServices` method for configuring services that are used by your application. The `ConfigureServices` method is a public method on your `Startup` class that take a `IServiceCollection` as a parameter and optionally returns an `IServiceProvider`. The `ConfigureServices` method is called before `Configure`. This is important, because some features like ASP.NET MVC require certain services to be added in `ConfigureServices` before they can be wired up to the request pipeline.

Just as with `Configure`, it is recommended that features that require substantial setup within `ConfigureServices` be wrapped up in extension methods on `IServiceCollection`. You can see in this example from the default web site template that several `Add[Something]` extension methods are used to configure the app to use services from Entity Framework, Identity, and MVC:

```
1 // This method gets called by the runtime. Use this method to add services to the container.
2 public void ConfigureServices(IServiceCollection services)
3 {
4     // Add Entity Framework services to the services container.
5     services.AddEntityFramework()
6         .AddSqlServer()
7         .AddDbContext<ApplicationDbContext>(options =>
8             options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));
9
10    // Add Identity services to the services container.
11    services.AddIdentity<ApplicationUser, IdentityRole>()
12        .AddEntityFrameworkStores<ApplicationDbContext>()
13        .AddDefaultTokenProviders();
14
15    // Configure the options for the authentication middleware.
16    // You can add options for Google, Twitter and other middleware as shown below.
17    // For more information see http://go.microsoft.com/fwlink/?LinkID=532715
18    services.Configure<FacebookAuthenticationOptions>(options =>
19    {
20        options.AppId = Configuration["Authentication:Facebook:AppId"];
21        options.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
22    });
23
24    services.Configure<MicrosoftAccountAuthenticationOptions>(options =>
25    {
```



```

26     options.ClientId = Configuration["Authentication:MicrosoftAccount:ClientId"];
27     options.ClientSecret = Configuration["Authentication:MicrosoftAccount:ClientSecret"];
28 });
29
30 // Add MVC services to the services container.
31 services.AddMvc();
32
33 // Uncomment the following line to add Web API services which makes it easier to port Web API 2
34 // You will also need to add the Microsoft.AspNet.Mvc.WebApiCompatShim package to the 'dependencies'
35 // services.AddWebApiConventions();
36
37 // Register application services.
38 services.AddTransient<IEmailSender, AuthMessageSender>();
39 services.AddTransient<ISmsSender, AuthMessageSender>();
40 }

```

Adding services to the services container makes them available within your application via [dependency injection](#). Just as the `Startup` class is able to specify dependencies its methods require as parameters, rather than hard-coding to a specific implementation, so too can your middleware, MVC controllers and other classes in your application.

The `ConfigureServices` method is also where you should add configuration option classes, like `AppSettings` in the example above, that you would like to have available in your application. See the [Configuration](#) topic to learn more about configuring options.

Summary

In ASP.NET 5, the `Startup` class is responsible for setting up the application, including its configuration, the services it will use, and how it will process requests.

Additional Resources

- [Working with Multiple Environments](#)
- [Middleware](#)
- [OWIN](#)

2.4.2 HTTP Abstractions

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.3 Middleware

By [Steve Smith](#)

Small application components that can be incorporated into an HTTP request pipeline are known collectively as middleware. ASP.NET 5 has integrated support for middleware, which are wired up in an application's `Configure` method during [Application Startup](#).

In this article:

- [What is middleware](#)

- *Creating a middleware pipeline with IApplicationBuilder*
- *Built-in middleware*
- *Writing middleware*

Download sample from [GitHub](#).

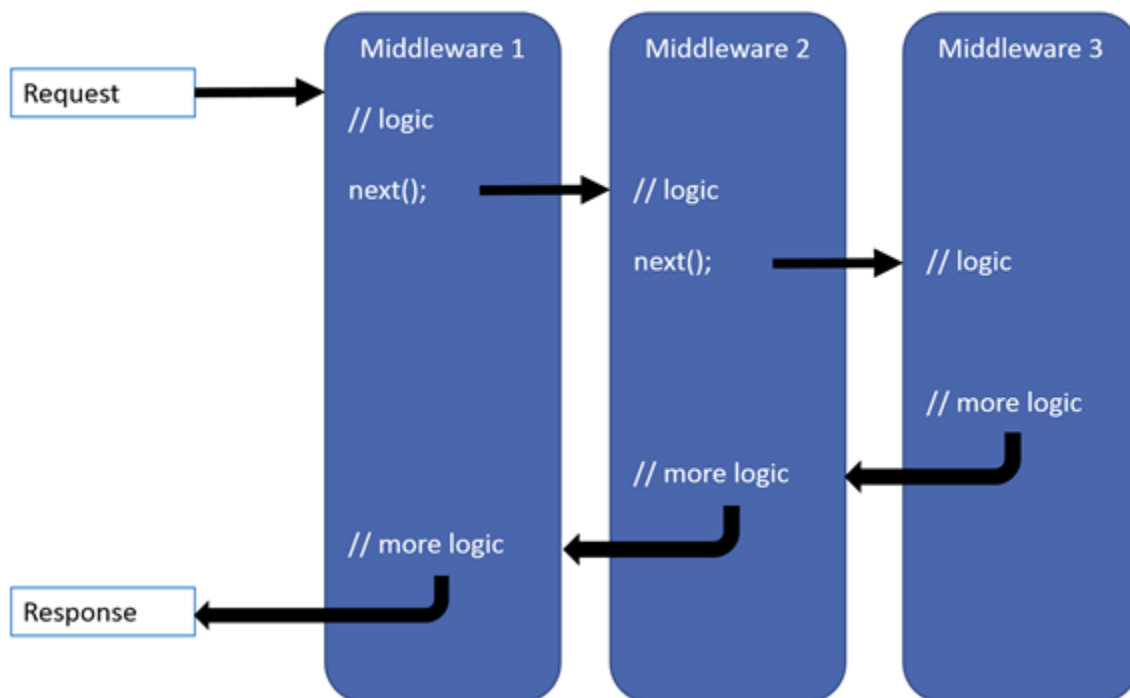
What is middleware

Middleware are components that are assembled into an application pipeline to handle requests and responses. Each component can choose whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component in the pipeline. Request delegates are used to build this request pipeline, which are then used to handle each incoming HTTP request to your application.

Request delegates are configured using `Run`, `Map`, and `Use` extension methods on the `IApplicationBuilder` type that is passed into the `Configure` method in the `Startup` class. An individual request delegate can be specified in-line as an anonymous method, or it can be defined in a reusable class. These reusable classes are *middleware*, or *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the chain, or choosing to short-circuit the chain if appropriate.

Creating a middleware pipeline with IApplicationBuilder

The ASP.NET request pipeline consists of a sequence of request delegates, called one after the next, as this diagram shows (the thread of execution follows the black arrows):



Each delegate has the opportunity to perform operations before and after the next delegate. Any delegate can choose to stop passing the request on to the next delegate, and instead handle the request itself. This is referred to as short-circuiting the request pipeline, and is desirable because it allows unnecessary work to be avoided. For example, an authorization middleware function might only call the next delegate in the pipeline if the request is authenticated,

otherwise it could short-circuit the pipeline and simply return some form of “Not Authorized” response. Exception handling delegates need to be called early on in the pipeline, so they are able to catch exceptions that occur in later calls within the call chain.

You can see an example of setting up a request pipeline, using a variety of request delegates, in the default web site template that ships with Visual Studio 2015. Its `Configure` method, shown below, first wires up error pages (in development) or the site’s production error handler, then builds out the pipeline with support for static files, ASP.NET Identity authentication, and finally, ASP.NET MVC.

```

1  }
2
3  // Configure is called after ConfigureServices is called.
4  public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
5  {
6      loggerFactory.MinimumLevel = LogLevel.Information;
7      loggerFactory.AddConsole();
8
9      // Configure the HTTP request pipeline.
10
11     // Add the following to the request pipeline only in development environment.
12     if (env.IsDevelopment())
13     {
14         app.UseBrowserLink();
15         app.UseErrorPage();
16         app.UseDatabaseErrorPage(DatabaseErrorPageOptions.ShowAll);
17     }
18     else
19     {
20         // Add Error handling middleware which catches all application specific errors and
21         // sends the request to the following path or controller action.
22         app.UseExceptionHandler("/Home/Error");
23     }
24
25     // Add static files to the request pipeline.
26     app.UseStaticFiles();
27
28     // Add cookie-based authentication to the request pipeline.
29     app.UseIdentity();
30
31     // Add authentication middleware to the request pipeline. You can configure options such as Id and
32     // For more information see http://go.microsoft.com/fwlink/?LinkID=532715
33     // app.UseFacebookAuthentication();
34     // app.UseGoogleAuthentication();
35     // app.UseMicrosoftAccountAuthentication();
36     // app.UseTwitterAuthentication();
37
38     // Add MVC to the request pipeline.
39     app.UseMvc(routes =>
40     {
41         routes.MapRoute(
42             name: "default",
43             template: "{controller=Home}/{action=Index}/{id?}");
44
45         // Uncomment the following line to add a route for porting Web API 2 controllers.
46         // routes.MapWebApiRoute("DefaultApi", "api/{controller}/{id?}");

```

Because of the order in which this pipeline was constructed, the middleware configured by the `UseExceptionHandler` method will catch any exceptions that occur in later calls (in non-development environments). Also, in this example a design decision has been made that static files will not be protected by any authentication. This is a tradeoff that

improves performance when handling static files since no other middleware (such as authentication middleware) needs to be called when handling these requests (ASP.NET 5 uses a specific `wwwroot` folder for all files that should be accessible by default, so there is typically no need to perform authentication before sending these files). If the request is not for a static file, it will flow to the next piece of middleware defined in the pipeline (in this case, Identity). Learn more about [Working with Static Files](#).

Note: Remember: the order in which you arrange your `Use[Middleware]` statements in your application's `Configure` method is very important. Be sure you have a good understanding of how your application's request pipeline will behave in various scenarios.

The simplest possible ASP.NET application sets up a single request delegate that handles all requests. In this case, there isn't really a request "pipeline", so much as a single anonymous function that is called in response to every HTTP request.

```
1 app.Run(async context =>
2 {
3     await context.Response.WriteAsync("Hello, World!");
4 });
```

It's important to realize that request delegate, as written here, will terminate the pipeline, regardless of other calls to `App.Run` that you may include. In the following example, only the first delegate ("Hello, World!") will be executed and displayed.

```
1 public void Configure(IApplicationBuilder app)
2 {
3     app.Run(async context =>
4     {
5         await context.Response.WriteAsync("Hello, World!");
6     });
7
8     app.Run(async context =>
9     {
10        await context.Response.WriteAsync("Hello, World, Again!");
11    });
```

You chain multiple request delegates together making a different call, with a `next` parameter representing the next delegate in the pipeline. Note that just because you're calling it "next" doesn't mean you can't perform actions both before and after the next delegate, as this example demonstrates:

```
1 public void ConfigureLogInline(IApplicationBuilder app, ILoggerFactory loggerfactory)
2 {
3     loggerfactory.AddConsole(minLevel: LogLevel.Information);
4     var logger = loggerfactory.CreateLogger(_environment);
5     app.Use(async (context, next) =>
6     {
7         logger.LogInformation("Handling request.");
8         await next.Invoke();
9         logger.LogInformation("Finished handling request.");
10    });
11
12    app.Run(async context =>
13    {
14        await context.Response.WriteAsync("Hello from " + _environment);
15    });
16 }
```

Warning: Be wary of modifying `HttpResponse` after invoking `next`, since one of the components further down the pipeline may have written to the response, causing it to be sent to the client.

Note: This `ConfigureLogInline` method is called when the application is run with an environment set to `LogInline`. Learn more about [Working with Multiple Environments](#). We will be using variations of `Configure[Environment]` to show different options in the rest of this article. The easiest way to run the samples in Visual Studio is with the web command, which is configured in `hosting.ini` to listen on `http://localhost:5000`. See also [Application Startup](#).

In the above example, the call to `await next.Invoke()` will call into the delegate on line 14. The client will receive the expected response (“Hello from LogInline”), and the server’s console output includes both the before and after messages, as you can see here:

```

C:\WINDOWS\system32\cmd.exe
info : [Microsoft.Net.Http.Server.WebListener] Start
info : [Microsoft.Net.Http.Server.WebListener] Listening on prefix: http://localhost:5000/
Started
info : [LogInline] Handling request.
info : [LogInline] Finished handling request.

```

Run, Map, and Use

You configure the HTTP pipeline using the `extensions` `Run`, `Map`, and `Use`. By convention, the `Run` method is simply a shorthand way of adding middleware to the pipeline that doesn’t call any other middleware (that is, it will not call a `next` request delegate). Thus, `Run` should only be called at the end of your pipeline. `Run` is a convention, and some middleware components may expose their own `Run[Middleware]` methods that should only run at the end of the pipeline. The following two examples (one using `Run` and the other `Use`) are equivalent to one another, since the second one doesn’t use its `next` parameter:

```

1 public void ConfigureEnvironmentOne(IApplicationBuilder app)
2 {
3     app.Run(async context =>
4     {
5         await context.Response.WriteAsync("Hello from " + _environment);
6     });
7 }
8
9 public void ConfigureEnvironmentTwo(IApplicationBuilder app)
10 {
11     app.Use(next => async context =>
12     {
13         await context.Response.WriteAsync("Hello from " + _environment);

```

```
14     });  
15 }
```

Note: The `IApplicationBuilder` interface itself exposes a single `Use` method, so technically they're not all *extension* methods.

We've already seen several examples of how to build a request pipeline with `Use`. `Map*` extensions are used as a convention for branching the pipeline. The current implementation supports branching based on the request's path, or using a predicate. The `Map` extension method is used to match request delegates based on a request's path. `Map` simply accepts a path and a function that configures a separate middleware pipeline. In the following example, any request with the base path of `/maptest` will be handled by the pipeline configured in the `HandleMapTest` method.

```
1 private static void HandleMapTest(IApplicationBuilder app)  
2 {  
3     app.Run(async context =>  
4     {  
5         await context.Response.WriteAsync("Map Test Successful");  
6     });  
7 }  
8  
9 public void ConfigureMapping(IApplicationBuilder app)  
10 {  
11     app.Map("/maptest", HandleMapTest);  
12 }  
13 }
```

Note: When `Map` is used, the matched path segment(s) are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

In addition to path-based mapping, the `MapWhen` method supports predicate-based middleware branching, allowing separate pipelines to be constructed in a very flexible fashion. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a simple predicate is used to detect the presence of a querystring variable `branch`:

```
1 private static void HandleBranch(IApplicationBuilder app)  
2 {  
3     app.Run(async context =>  
4     {  
5         await context.Response.WriteAsync("Branch used.");  
6     });  
7 }  
8  
9 public void ConfigureMapWhen(IApplicationBuilder app)  
10 {  
11     app.MapWhen(context => {  
12         return context.Request.Query.ContainsKey("branch");  
13     }, HandleBranch);  
14  
15     app.Run(async context =>  
16     {  
17         await context.Response.WriteAsync("Hello from " + _environment);  
18     });  
19 }
```

Using the configuration shown above, any request that includes a querystring value for `branch` will use the pipeline

defined in the `HandleBranch` method (in this case, a response of “Branch used.”). All other requests (that do not define a `querystring` value for `branch`) will be handled by the delegate defined on line 17.

Built-in middleware

ASP.NET ships with the following middleware components:

Table 2.1: Middleware

Middleware	Description
Authentication	Provides authentication support.
CORS	Configures Cross-Origin Resource Sharing.
Diagnostics	Includes support for error pages and runtime information.
Routing	Define and constrain request routes.
Session	Provides support for managing user sessions.
Static Files	Provides support for serving static files, and directory browsing.

Writing middleware

For more complex request handling functionality, the ASP.NET team recommends implementing the middleware in its own class, and exposing an `IApplicationBuilder` extension method that can be called from the `Configure` method. The simple logging middleware shown in the previous example can be converted into a middleware class that takes in the next `RequestDelegate` in its constructor and supports an `Invoke` method as shown:

Listing 2.1: RequestLoggerMiddleware.cs

```

1 using Microsoft.AspNet.Builder;
2 using Microsoft.AspNet.Http;
3 using Microsoft.Framework.Logging;
4 using System.Threading.Tasks;
5
6 namespace MiddlewareSample
7 {
8     public class RequestLoggerMiddleware
9     {
10         private readonly RequestDelegate _next;
11         private readonly ILogger _logger;
12
13         public RequestLoggerMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)
14         {
15             _next = next;
16             _logger = loggerFactory.CreateLogger<RequestLoggerMiddleware>();
17         }
18
19         public async Task Invoke(HttpContext context)
20         {
21             _logger.LogInformation("Handling request: " + context.Request.Path);
22             await _next.Invoke(context);
23             _logger.LogInformation("Finished handling request.");
24         }
25     }
26 }
```

The middleware follows the [Explicit Dependencies Principle](#) and exposes all of its dependencies in its constructor. Middleware can take advantage of the `UseMiddleware<T>` extension to inject services directly into their constructors,

as shown in the example below. Dependency injected services are automatically filled, and the extension takes a params array of arguments to be used for non-injected parameters.

Listing 2.2: RequestLoggerExtensions.cs

```

1 public static class RequestLoggerExtensions
2 {
3     public static IApplicationBuilder UseRequestLogger(this IApplicationBuilder builder)
4     {
5         return builder.UseMiddleware<RequestLoggerMiddleware>();
6     }
7 }

```

Using the extension method and associated middleware class, the Configure method becomes very simple and readable.

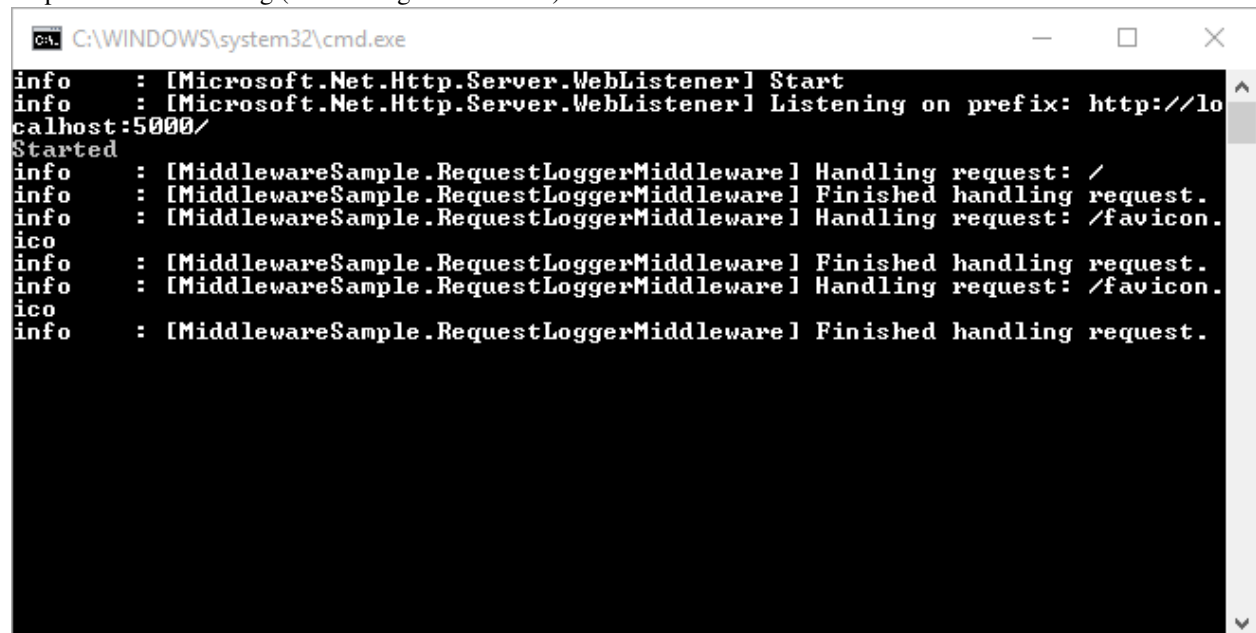
```

1 public void ConfigureLogMiddleware(IApplicationBuilder app,
2     ILoggerFactory loggerfactory)
3 {
4     loggerfactory.AddConsole(minLevel: LogLevel.Information);
5
6     app.UseRequestLogger();
7
8     app.Run(async context =>
9     {
10         await context.Response.WriteAsync("Hello from " + _environment);
11     });
12 }

```

Although RequestLoggerMiddleware requires an ILoggerFactory parameter in its constructor, neither the Startup class nor the UseRequestLogger extension method need to explicitly supply it. Instead, it is automatically provided through dependency injection performed within UseMiddleware<T>.

Testing the middleware (by setting the ASPNET_ENV environment variable to LogMiddleware) should result in output like the following (when using WebListener):



```

C:\WINDOWS\system32\cmd.exe
info : [Microsoft.Net.Http.Server.WebListener] Start
info : [Microsoft.Net.Http.Server.WebListener] Listening on prefix: http://localhost:5000/
Started
info : [MiddlewareSample.RequestLoggerMiddleware] Handling request: /
info : [MiddlewareSample.RequestLoggerMiddleware] Finished handling request.
info : [MiddlewareSample.RequestLoggerMiddleware] Handling request: /favicon.ico
info : [MiddlewareSample.RequestLoggerMiddleware] Finished handling request.
info : [MiddlewareSample.RequestLoggerMiddleware] Handling request: /favicon.ico
info : [MiddlewareSample.RequestLoggerMiddleware] Finished handling request.

```

Note: You can see another example of UseMiddleware<T> in action in the [UseStaticFiles](#) extension method, which is used to create the [StaticFileMiddleware](#) with its required constructor parameters. In this

case, the `StaticFileOptions` parameter is passed in, but other constructor parameters are supplied by `UseMiddleware<T>` and dependency injection.

Summary

Middleware provide simple components for adding features to individual web requests. Applications configure their request pipelines in accordance with the features they need to support, and thus have fine-grained control over the functionality each request uses. Developers can easily create their own middleware to provide additional functionality to ASP.NET applications.

Additional Resources

- [Application Startup](#)
- [Request Features](#)

2.4.4 Working with Static Files

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.5 File System

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.6 Routing

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.7 Configuration

By [Steve Smith](#) and [Daniel Roth](#)

ASP.NET 5 supports a variety of different configuration options. Application configuration data can come from files using built-in support for JSON, XML, and INI formats, as well as from environment variables. You can also write your own *custom configuration provider*.

In this article:

- *Getting and setting configuration settings*
- *Using the built-in providers*
- *Using Options and configuration objects*
- *Writing custom providers*

[Download sample from GitHub.](#)

Getting and setting configuration settings

ASP.NET 5's configuration system has been re-architected from previous versions of ASP.NET, which relied on `System.Configuration` and XML configuration files like `web.config`. The new [configuration model](#) provides streamlined access to key/value based settings that can be retrieved from a variety of sources. Applications and frameworks can then access configured settings using the new [Options pattern](#).

To work with settings in your ASP.NET application, it is recommended that you only instantiate an instance of `Configuration` in your application's `Startup` class. Then, use the [Options pattern](#) to access individual settings.

At its simplest, the `Configuration` class is just a collection of `Sources`, which provide the ability to read and write name/value pairs. You must configure at least one source in order for `Configuration` to function correctly. The following sample shows how to test working with `Configuration` as a key/value store:

```
1 // assumes using Microsoft.Framework.ConfigurationModel is specified
2 var builder = new ConfigurationBuilder();
3 builder.Add(new MemoryConfigurationSource());
4 var config = builder.Build();
5 config.Set("somekey", "somevalue");
6
7 // do some other work
8
9 string setting = config.Get("somekey"); // returns "somevalue"
10 // or
11 string setting2 = config["somekey"]; // also returns "somevalue"
```

Note: You must set at least one configuration source.

It's not unusual to store configuration values in a hierarchical structure, especially when using external files (e.g. JSON, XML, INI). In this case, configuration values can be retrieved using a `:` separated key, starting from the root of the hierarchy. For example, consider the following `config.json` file:

```
1 {
2   "Data": {
3     "DefaultConnection": {
4       "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=aspnet5-WebApplication1-3bd99dbb-a
5     }
6   }
7 }
```

The application uses configuration to configure the right connection string. Access to the `ConnectionString` setting is achieved through this key: `Data:DefaultConnection:ConnectionString`.

The settings required by your application and the mechanism used to specify those settings (configuration being one example) can be decoupled using the [options pattern](#). To use the options pattern you create your own settings class (probably several different classes, corresponding to different cohesive groups of settings) that you can inject into your application using an options service. You can then specify your settings using configuration or whatever mechanism you choose.

Note: You could store your `Configuration` instance as a service, but this would unnecessarily couple your application to a single configuration system and specific configuration keys. Instead, you can use the [Options pattern](#) to avoid these issues.

Using the built-in providers

The configuration framework has built-in support for JSON, XML, and INI configuration files, as well as support for in-memory configuration (directly setting values in code) and the ability to pull configuration from environment variables and command line parameters. Developers are not limited to using a single configuration source. In fact several may be set up together such that a default configuration is overridden by settings from another source if they are present.

Adding support for additional configuration file sources is accomplished through extension methods. These methods can be called on a `ConfigurationBuilder` instance in a standalone fashion, or chained together as a fluent API, as shown.

```
1 var builder = new ConfigurationBuilder(".");
2 builder.AddJsonFile("config.json");
3 builder.AddEnvironmentVariables();
4 var config = builder.Build();
```

The order in which configuration sources are specified is important, as this establishes the precedence with which settings will be applied if they exist in multiple locations. In the example above, if the same setting exists in both `config.json` and in an environment variable, the setting from the environment variable will be the one that is used. Essentially, the last configuration source specified “wins” if a setting exists in more than one location.

It can be useful to have environment-specific configuration files. This can be achieved using the following:

```
1 public Startup(IHostingEnvironment env, IApplicationEnvironment appEnv)
2 {
3     // Setup configuration sources.
4
5     var builder = new ConfigurationBuilder(appEnv.ApplicationBasePath)
6         .AddJsonFile("config.json")
7         .AddJsonFile($"config.{env.EnvironmentName}.json", optional: true);
8
9     if (env.IsDevelopment())
10    {
11        // This reads the configuration keys from the secret store.
12        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=527346
13        builder.AddUserSecrets();
14    }
15    builder.AddEnvironmentVariables();
16    Configuration = builder.Build();
17 }
```

The `IHostingEnvironment` service is used to get the current environment. In the Development environment, the highlighted line of code above would look for a file named `config.Development.json` and use its values, overriding any other values, if it’s present. Learn more about [Working with Multiple Environments](#).

Warning: You should never store passwords or other sensitive data in source code or in plain text configuration files. You also shouldn’t use production secrets in your development or test environments. Instead, such secrets should be specified outside the project tree, so they cannot be accidentally committed into the source repository. Learn more about [Working with Multiple Environments](#) and managing [Safe Storage of Application Secrets](#).

One way to leverage the order precedence of Configuration is to specify default values, which can be overridden. In this simple console application, a default value for the username setting is specified in a `MemoryConfigurationSource`, but this is overridden if a command line argument for username is passed to the application. You can see in the output how many configuration sources are configured at each stage of the program.

```

1 using System;
2 using System.Linq;
3 using Microsoft.Framework.Configuration;
4
5 namespace ConfigConsole
6 {
7     public class Program
8     {
9         public void Main(string[] args)
10        {
11            var builder = new ConfigurationBuilder();
12            Console.WriteLine("Initial Config Sources: " + builder.Sources.Count());
13
14            var defaultSettings = new MemoryConfigurationSource();
15            defaultSettings.Set("username", "Guest");
16            builder.Add(defaultSettings);
17            Console.WriteLine("Added Memory Source. Sources: " + builder.Sources.Count());
18
19            builder.AddCommandLine(args);
20            Console.WriteLine("Added Command Line Source. Sources: " + builder.Sources.Count());
21
22            var config = builder.Build();
23            string username = config.Get("username");
24
25            Console.WriteLine($"Hello, {username}!");
26        }
27    }
28 }

```

When run, the program will display the default value unless a command line parameter overrides it.

```

C:\Windows\System32\cmd.exe
>dnx . ConfigConsole
Initial Config Sources: 0
Added Memory Source. Sources: 1
Added Command Line Source. Sources: 2
Hello, Guest!

>dnx . ConfigConsole --username Steve
Initial Config Sources: 0
Added Memory Source. Sources: 1
Added Command Line Source. Sources: 2
Hello, Steve!

>

```

Using Options and configuration objects

Using [Options](#) you can easily convert any class (or POJO - Plain Old CLR Object) into a settings class. It's recommended that you create well-factored settings objects that correspond to certain features within your application, thus following the Interface Segregation Principle (ISP) (classes depend only on the configuration settings they use) as well as Separation of Concerns (settings for disparate parts of your app are managed separately, and thus are less likely to negatively impact one another).

A simple `MyOptions` class is shown here:

```
1 public class MyOptions
2 {
3     public string Option1 { get; set; }
4     public int Option2 { get; set; }
5 }
```

Options can be injected into your application using the `IOptions<TOptions>` service. For example, the following MVC controller uses `IOptions<MyOptions>` to access the settings it needs to render the Index view:

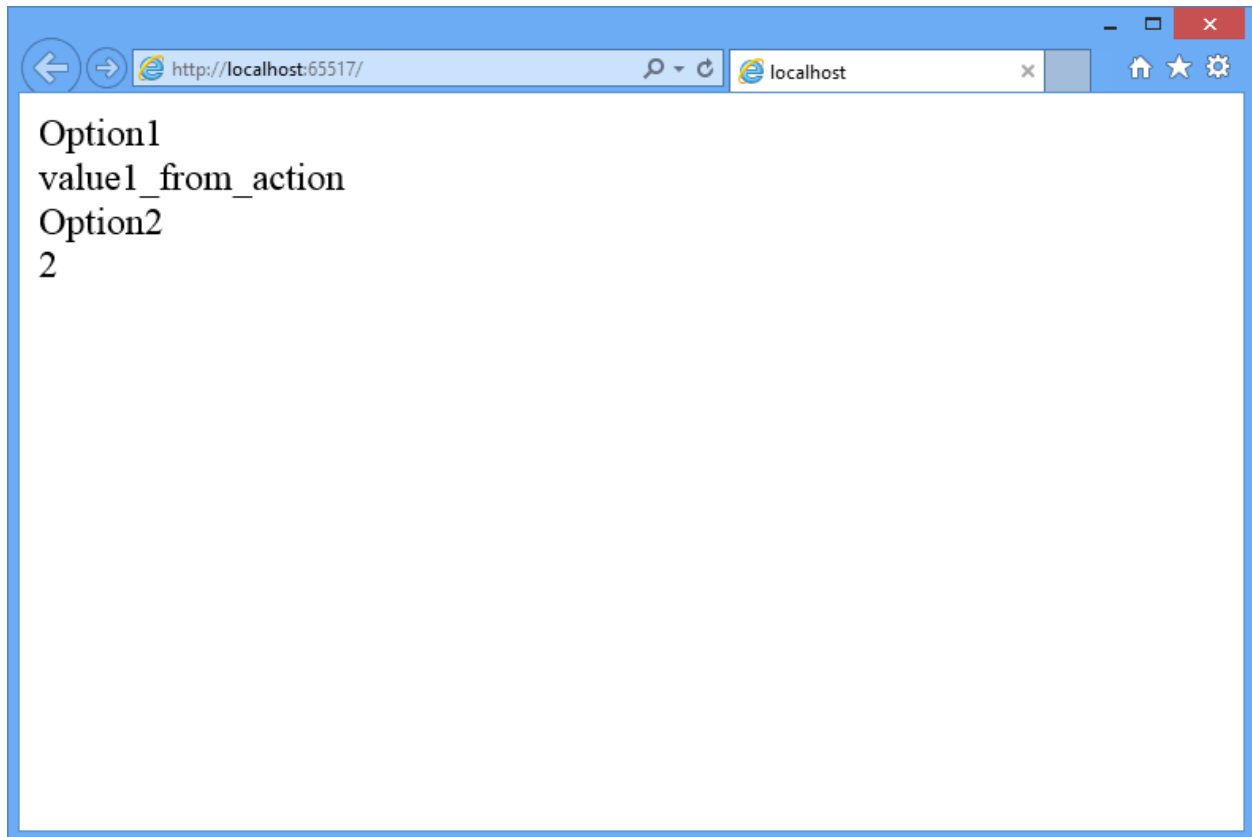
```
1 public class HomeController : Controller
2 {
3     public HomeController(IOptions<MyOptions> optionsAccessor)
4     {
5         Options = optionsAccessor.Options;
6     }
7
8     MyOptions Options { get; }
9
10    public IActionResult Index()
11    {
12        return View(Options);
13    }
14 }
```

Learn more about [Dependency Injection](#).

To setup the `IOptions<TOption>` service you call the `AddOptions()` extension method during startup in your `ConfigureServices` method:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     // Setup options with DI
4     services.AddOptions();
5 }
```

The Index view displays the configured options:



You configure options using the `Configure<TOption>` extension method. You can configure options using a delegate or by binding your options to configuration:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     // Setup options with DI
4     services.AddOptions();
5
6     // Configure MyOptions using config
7     services.Configure<MyOptions>(Configuration);
8
9     // Configure MyOptions using code
10    services.Configure<MyOptions>(myOptions =>
11    {
12        myOptions.Option1 = "value1_from_action";
13    });
14
15    services.AddMvc();
16 }
```

When you bind options to configuration each property in your options type is bound to a configuration key of the form `property:subproperty:...`. For example, the `MyOptions.Option1` property is bound to the key `Option1`, which is read from the `option1` property in `config.json`. Note that configuration keys are case insensitive.

Each call to `Configure<TOption>` adds an `IConfigureOptions<TOption>` service to the service container that is used by the `IOptions<TOption>` service to provide the configured options to the application or framework. If you want to configure your options some other way (ex. reading settings from a data base) you can use the `ConfigureOptions<TOptions>` extension method to you specify a custom

`IConfigureOptions<TOption>` service directly.

You can have multiple `IConfigureOptions<TOption>` services for the same option type and they are all applied in order. In the *example* above value of `Option1` and `Option2` are both specified in `config.json`, but the value of `Option1` is overridden by the configured delegate.

Writing custom providers

In addition to using the [built-in configuration source providers](#), you can also write your own. To do so, you simply inherit from `ConfigurationSource`, and populate the `Data` property with the settings from your configuration source.

Example: Entity Framework Settings

You may wish to store some of your application's settings in a database, and access them using Entity Framework (EF). There are many ways in which you could choose to store such values, ranging from a simple table with a column for the setting name and another column for the setting value, to having separate columns for each setting value. In this example, I'm going to create a simple configuration source that reads name-value pairs from a database using EF.

To start off we'll define a simple `ConfigurationValue` entity for storing configuration values in the database:

```
1 public class ConfigurationValue
2 {
3     public string Id { get; set; }
4     public string Value { get; set; }
5 }
```

We also need a `ConfigurationContext` to store and access the configured values using EF:

```
1 public class ConfigurationContext : DbContext
2 {
3     public ConfigurationContext(DbContextOptions options) : base(options)
4     {
5     }
6
7     public DbSet<ConfigurationValue> Values { get; set; }
8 }
```

Next, create the custom configuration source by inheriting from `ConfigurationSource`. The configuration data is loaded by overriding the `Load` method, which reads in all of the configuration data from the configured database. For demonstration purposes, the configuration source also takes care of initializing the database if it hasn't already been created and populated:

```
1 public class EntityFrameworkConfigurationSource : ConfigurationSource
2 {
3     public EntityFrameworkConfigurationSource(Action<DbContextOptionsBuilder> optionsAction)
4     {
5         OptionsAction = optionsAction;
6     }
7
8     Action<DbContextOptionsBuilder> OptionsAction { get; }
9
10    public override void Load()
11    {
12        var builder = new DbContextOptionsBuilder<ConfigurationContext>();
13        OptionsAction(builder);
14    }
```

```

15     using (var dbContext = new ConfigurationContext(builder.Options))
16     {
17         dbContext.Database.EnsureCreated();
18         Data = !dbContext.Values.Any()
19             ? CreateAndSaveDefaultValues(dbContext)
20             : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
21     }
22 }
23
24 private IDictionary<string, string> CreateAndSaveDefaultValues(ConfigurationContext dbContext)
25 {
26     var configValues = new Dictionary<string, string>
27     {
28         { "key1", "value_from_ef_1" },
29         { "key2", "value_from_ef_2" }
30     };
31     dbContext.Values.AddRange(configValues
32         .Select(kvp => new ConfigurationValue() { Id = kvp.Key, Value = kvp.Value })
33         .ToArray());
34     dbContext.SaveChanges();
35     return configValues;
36 }
37 }

```

By convention we also add an `AddEntityFramework` extension method for adding the configuration source:

```

1 public static class EntityFrameworkExtensions
2 {
3     public static IConfigurationBuilder AddEntityFramework(this IConfigurationBuilder builder, Action<IConfigurationBuilder> setup)
4     {
5         return builder.Add(new EntityFrameworkConfigurationSource(setup));
6     }
7 }

```

You can see an example of how to use this custom `ConfigurationSource` in your application in the following example. Create a new `ConfigurationBuilder` to setup your configuration sources. To add the `EntityFrameworkConfigurationSource` you first need to specify the data provider and connection string. How should you configure the connection string? Using configuration of course! Add a `config.json` file as a configuration source to bootstrap setting up the `EntityFrameworkConfigurationSource`. By reusing the same `ConfigurationBuilder` any settings specified in the database will override settings specified in `config.json`:

```

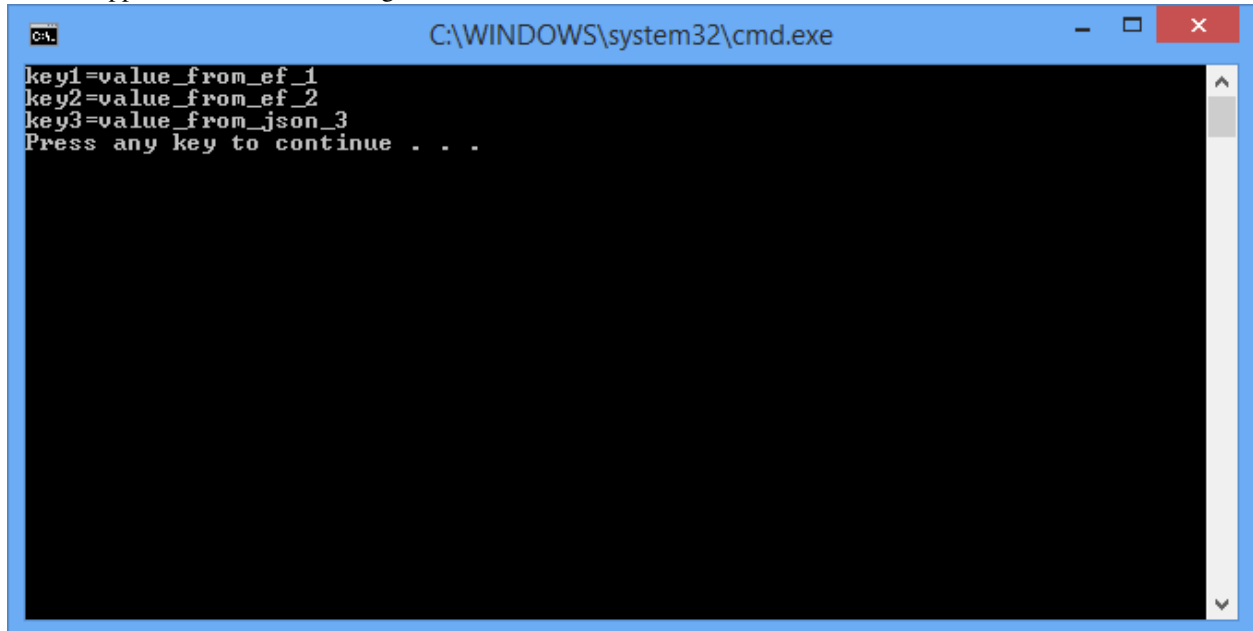
1 public class Program
2 {
3     public void Main(string[] args)
4     {
5         var builder = new ConfigurationBuilder(".");
6         builder.AddJsonFile("config.json");
7         builder.AddEnvironmentVariables();
8         var config = builder.Build();
9
10        builder.AddEntityFramework(options => options.UseSqlServer(config["Data:DefaultConnection:ConnectionString"]));
11        config = builder.Build();
12
13        Console.WriteLine("key1={0}", config.Get("key1"));
14        Console.WriteLine("key2={0}", config.Get("key2"));
15        Console.WriteLine("key3={0}", config.Get("key3"));
16    }
17 }

```


18

}

Run the application to see the configured values:

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a blue title bar and a black background. The text displayed in the command prompt is:

```
key1=value_from_ef_1
key2=value_from_ef_2
key3=value_from_json_3
Press any key to continue . . .
```

Summary

ASP.NET 5 provides a very flexible configuration model that supports a number of different file-based options, as well as command-line, in-memory, and environment variables. It works seamlessly with the options model so that you can inject strongly typed settings into your application or framework. You can create your own custom configuration source providers as well, which can work with or replace the built-in providers, allowing for extreme flexibility.

2.4.8 Dependency Injection

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.9 Options Model

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.10 Diagnostics

By [Steve Smith](#)

ASP.NET 5 includes a number of new features that can assist with diagnosing problems. Configuring different handlers for application errors or to display additional information about the application can easily be achieved in the application's startup class.

In this article:

- *Configuring an error handling page*
- *Using the error page during development*
- *HTTP 500 errors on Azure*
- *The runtime info page*
- *The welcome page*

Browse or download samples on [GitHub](#).

Configuring an error handling page

In ASP.NET 5, you configure the pipeline for each request in the `Startup` class's `Configure()` method (learn more about [Configuration](#)). You can add a simple error page, meant only for use during development, very easily. All that's required is to add a dependency on `Microsoft.AspNet.Diagnostics` to the project (and a `using` statement to `Startup.cs`), and then add one line to `Configure()` in `Startup.cs`:

```
1 using Microsoft.AspNet.Builder;
2 using Microsoft.AspNet.Diagnostics;
3 using Microsoft.AspNet.Hosting;
4 using Microsoft.AspNet.Http;
5 using Microsoft.Framework.DependencyInjection;
6 using System;
7
8 namespace DiagDemo
9 {
10     public class Startup
11     {
12         // For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?LinkId=301871
13         public void ConfigureServices(IServiceCollection services)
14         {
15         }
16
17         public void Configure(IApplicationBuilder app, IHostingEnvironment env)
18         {
19             if (string.Equals(env.EnvironmentName, "Development", StringComparison.OrdinalIgnoreCase))
20             {
21                 app.UseErrorPage();
22
23                 app.UseRuntimeInfoPage(); // default path is /runtimeinfo
24             }
25             else
26             {
27                 // specify production behavior for error handling, for example:
28                 // app.UseExceptionHandler("/Home/Error");
29                 // if nothing is set here, web server error page will be displayed
30             }
31
32             app.UseWelcomePage("/welcome");
33
34             app.Run(async (context) =>
35             {
36                 if (context.Request.Query.ContainsKey("throw")) throw new Exception("Exception triggered");
37                 context.Response.ContentType = "text/html";
38                 await context.Response.WriteAsync("<html><body>Hello World!");
39                 await context.Response.WriteAsync("<ul>");
```

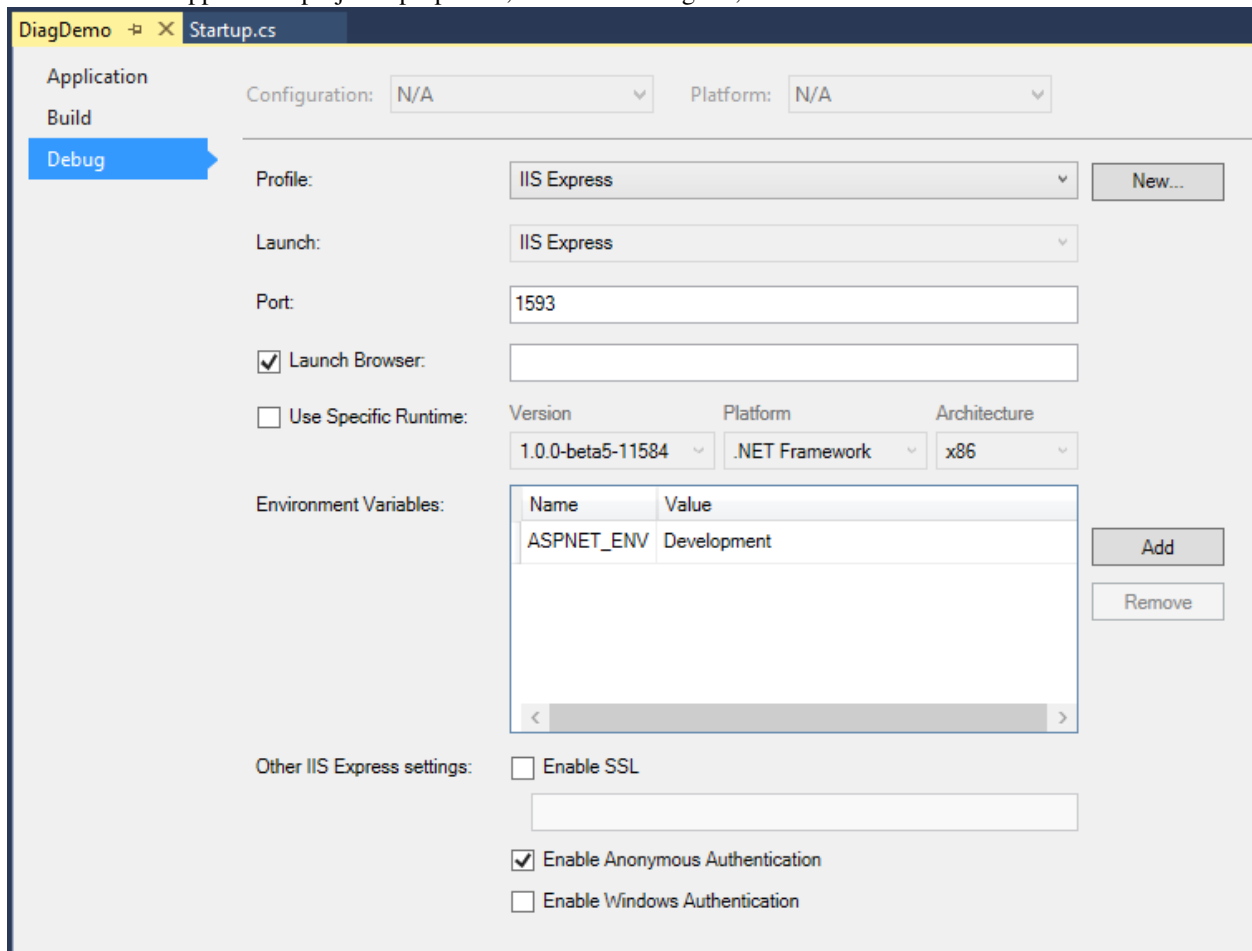
```

40         await context.Response.WriteAsync("<li><a href=\"/welcome\">Welcome Page</a></li>");
41         await context.Response.WriteAsync("<li><a href=\"/?throw=true\">Throw Exception</a></li>");
42         await context.Response.WriteAsync("</ul>");
43         await context.Response.WriteAsync("</body></html>");
44     });
45 }
46 }
47 }

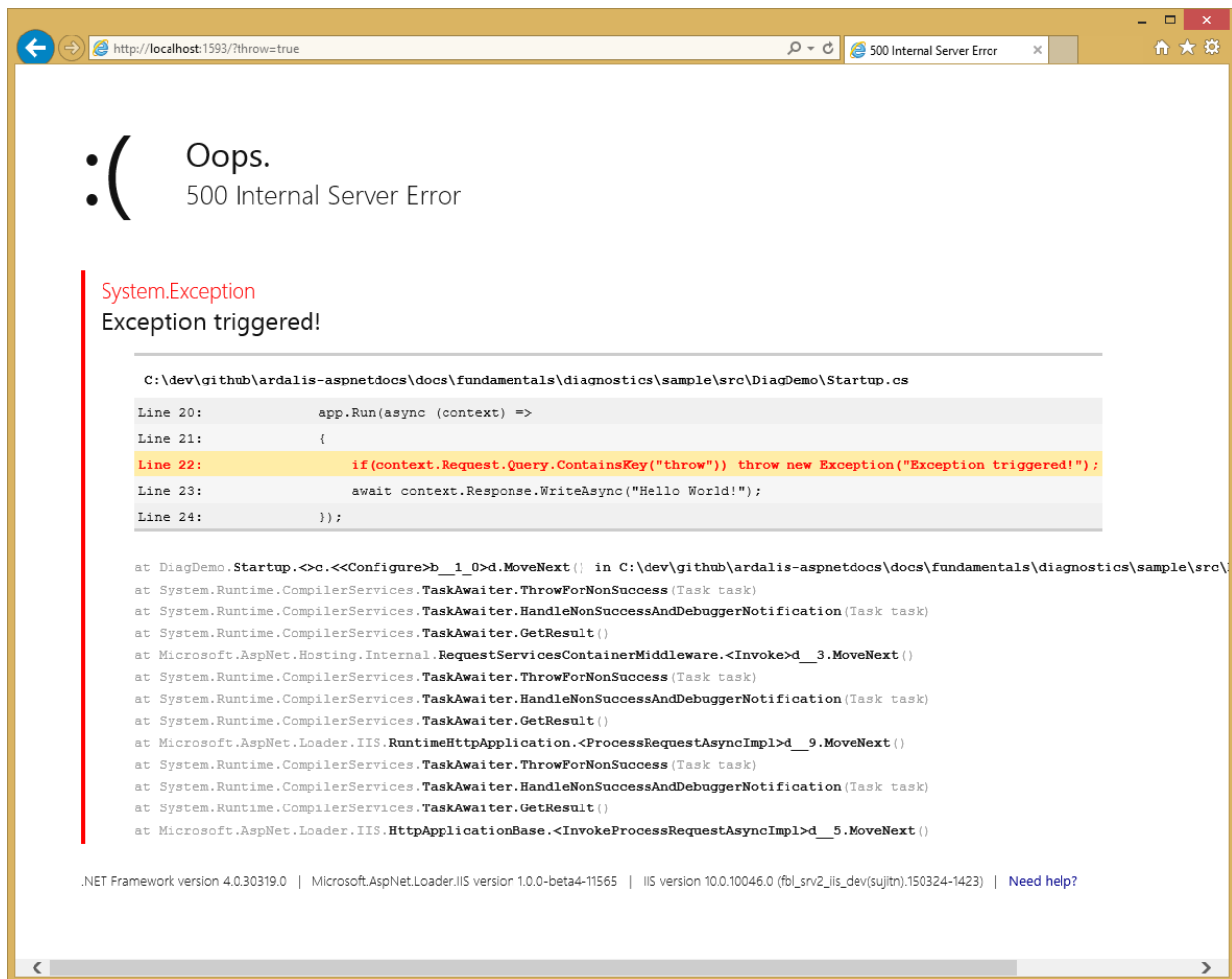
```

The above code, which is built from the ASP.NET 5 Empty template, includes a simple mechanism for creating an exception on line 36. If a request includes a non-empty querystring parameter for the variable `throw` (e.g. a path of `/?throw=true`), an exception will be thrown. Line 21 makes the call to `UseErrorPage()` to enable the error page middleware.

Notice that the call to `UseErrorPage()` is wrapped inside an `if` condition that checks the current `EnvironmentName`. This is a good practice, since you typically do not want to share detailed diagnostic information about your application publicly once it is in production. This check uses the `ASPNET_ENV` environment variable. If you are using Visual Studio 2015, you can customize the environment variables used when the application runs in the web application project's properties, under the `Debug` tab, as shown here:



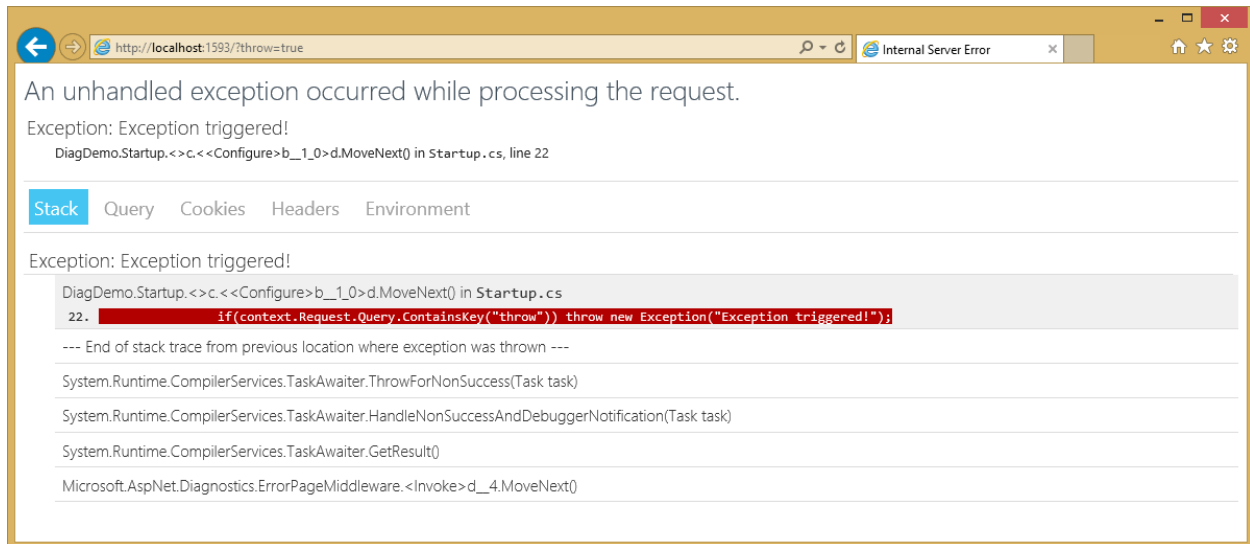
Setting the `ASPNET_ENV` variable to anything other than `Development` (e.g. `Production`) will cause the application not to call `UseErrorPage()`, and thus any exceptions will be handled by the underlying web server package (in this case, `Microsoft.AspNet.Server.IIS`) as shown here:



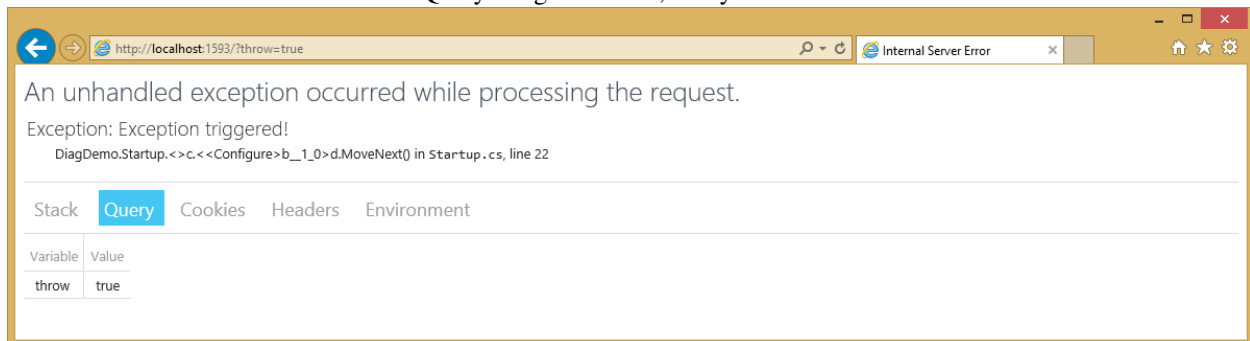
We will cover the features provided by the error page in the next section (ensure `ASPNET_ENV` is reset to Development if you are following along).

Using the error page during development

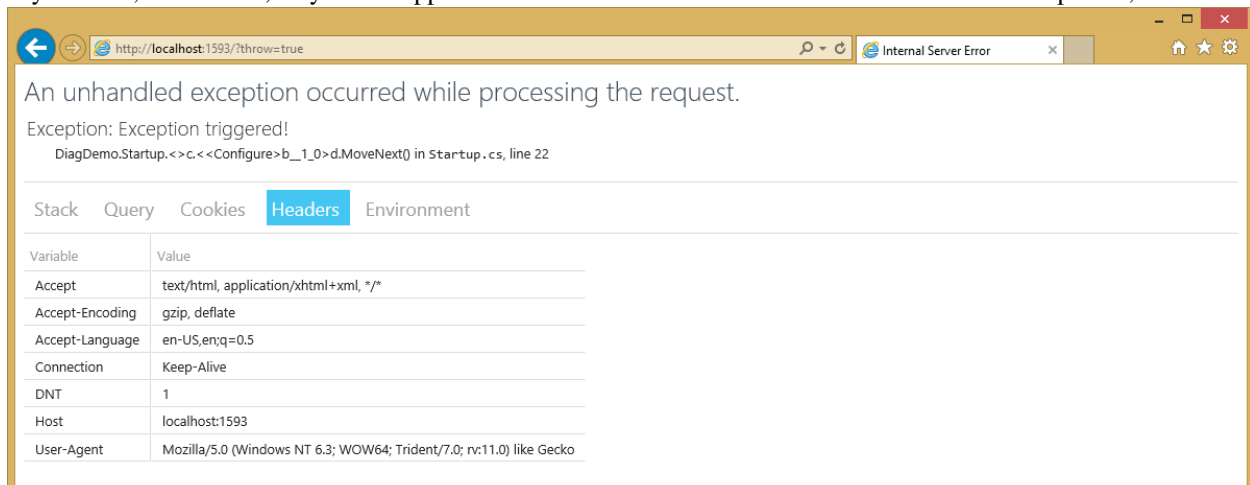
The default error page will display some useful diagnostics information when an unhandled exception occurs within the web processing pipeline. The error page includes several tabs with information about the exception that was triggered and the request that was made. The first tab shows the stack trace:



The next tab shows the contents of the Querystring collection, if any:



In this case, you can see the value of the `throw` parameter that was passed to this request. This request didn't have any cookies, but if it did, they would appear on the Cookies tab. You can see the headers that were passed, here:



Note: In the current pre-release build, the Cookies section of ErrorPage is not yet enabled. [View ErrorPage Source](#).

HTTP 500 errors on Azure

If your app throws an exception before the `Configure` method in *Startup.cs* completes, the error page won't be configured. For local development using IIS Express, you'll still get a call stack showing where the exception occurred. The same app deployed to Azure (or another production server) will return an HTTP 500 error with no message details. ASP.NET 5 uses a new configuration model that is not based on *web.config*, and when you create a new web app with Visual Studio 2015, the project no longer contains a *web.config* file. (See [Understanding ASP.NET 5 Web Apps](#).)

The publish wizard in Visual Studio 2015 creates a *web.config* file if you don't have one. If you have a *web.config* file in the *wwwroot* folder, deploy inserts the markup into the *web.config* file it generates.

To get detailed error messages on Azure, add the following *web.config* file to the *wwwroot* folder.

Note: Security warning: Enabling detailed error message can leak critical information from your app. You should never enable detailed error messages on a production app.

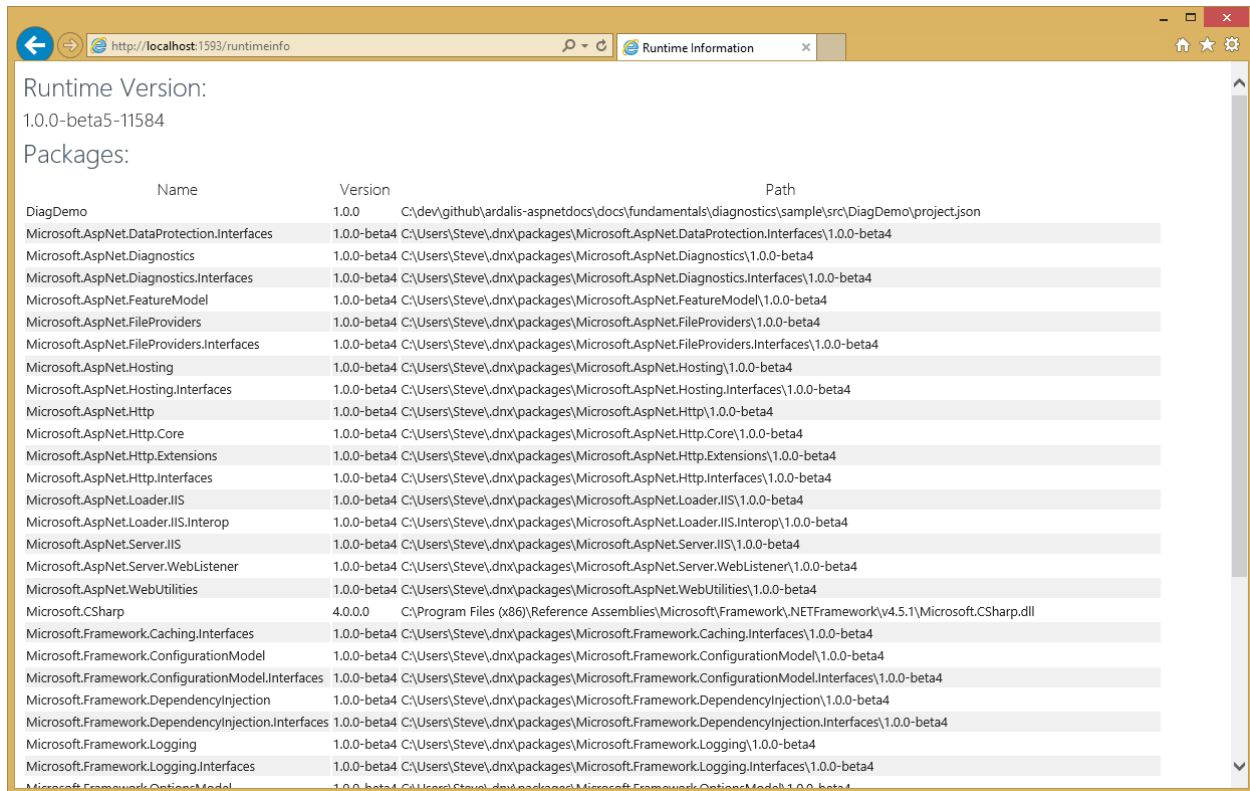
```
<configuration>
  <system.web>
    <customErrors mode="Off"/>
  </system.web>
</configuration>
```

The runtime info page

In addition to *configuring and displaying an error page*, you can also add a runtime info page by simply calling an extension method in *Startup.cs*. The following line, is used to enable this feature:

```
app.UseRuntimeInfoPage(); // default path is /runtimeinfo
```

Once this is added to your ASP.NET application, you can browse to the specified path (*/runtimeinfo*) to see information about the runtime that is being used and the packages that are included in the application, as shown below:



Name	Version	Path
DiagDemo	1.0.0	C:\dev\github\jardalis-aspnetdocs\docs\fundamentals\diagnostics\sample\src\DiagDemo\project.json
Microsoft.AspNet.DataProtection.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.DataProtection.Interfaces\1.0.0-beta4
Microsoft.AspNet.Diagnostics	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Diagnostics\1.0.0-beta4
Microsoft.AspNet.Diagnostics.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Diagnostics.Interfaces\1.0.0-beta4
Microsoft.AspNet.FeatureModel	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.FeatureModel\1.0.0-beta4
Microsoft.AspNet.FileProviders	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.FileProviders\1.0.0-beta4
Microsoft.AspNet.FileProviders.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.FileProviders.Interfaces\1.0.0-beta4
Microsoft.AspNet.Hosting	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Hosting\1.0.0-beta4
Microsoft.AspNet.Hosting.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Hosting.Interfaces\1.0.0-beta4
Microsoft.AspNet.Http	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Http\1.0.0-beta4
Microsoft.AspNet.Http.Core	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Http.Core\1.0.0-beta4
Microsoft.AspNet.Http.Extensions	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Http.Extensions\1.0.0-beta4
Microsoft.AspNet.Http.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Http.Interfaces\1.0.0-beta4
Microsoft.AspNet.Loader.IIS	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Loader.IIS\1.0.0-beta4
Microsoft.AspNet.Loader.IIS.Interop	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Loader.IIS.Interop\1.0.0-beta4
Microsoft.AspNet.Server.IIS	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Server.IIS\1.0.0-beta4
Microsoft.AspNet.Server.WebListener	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.Server.WebListener\1.0.0-beta4
Microsoft.AspNet.WebUtilities	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.AspNet.WebUtilities\1.0.0-beta4
Microsoft.CSharp	4.0.0.0	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\NETFramework\v4.5.1\Microsoft.CSharp.dll
Microsoft.Framework.Caching.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.Caching.Interfaces\1.0.0-beta4
Microsoft.Framework.ConfigurationModel	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.ConfigurationModel\1.0.0-beta4
Microsoft.Framework.ConfigurationModel.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.ConfigurationModel.Interfaces\1.0.0-beta4
Microsoft.Framework.DependencyInjection	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.DependencyInjection\1.0.0-beta4
Microsoft.Framework.DependencyInjection.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.DependencyInjection.Interfaces\1.0.0-beta4
Microsoft.Framework.Logging	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.Logging\1.0.0-beta4
Microsoft.Framework.Logging.Interfaces	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.Logging.Interfaces\1.0.0-beta4
Microsoft.Framework.OptionsModel	1.0.0-beta4	C:\Users\Steve\dnx\packages\Microsoft.Framework.OptionsModel\1.0.0-beta4

The path for this page can be optionally specified in the call to `UseRuntimeInfoPage()`. It accepts a [RuntimeInfoPageOptions](#) instance as a parameter, which has a `Path` property. For example, to specify a path of `/info` you would call `UseRuntimeInfoPage()` as shown here:

```
app.UseRuntimeInfoPage("/info");
```

As with `UseErrorPage()`, it is a good idea to limit public access to diagnostic information about your application. As such, in our sample we are only implementing `UseRuntimeInfoPage()` when the `EnvironmentName` is set to `Development`.

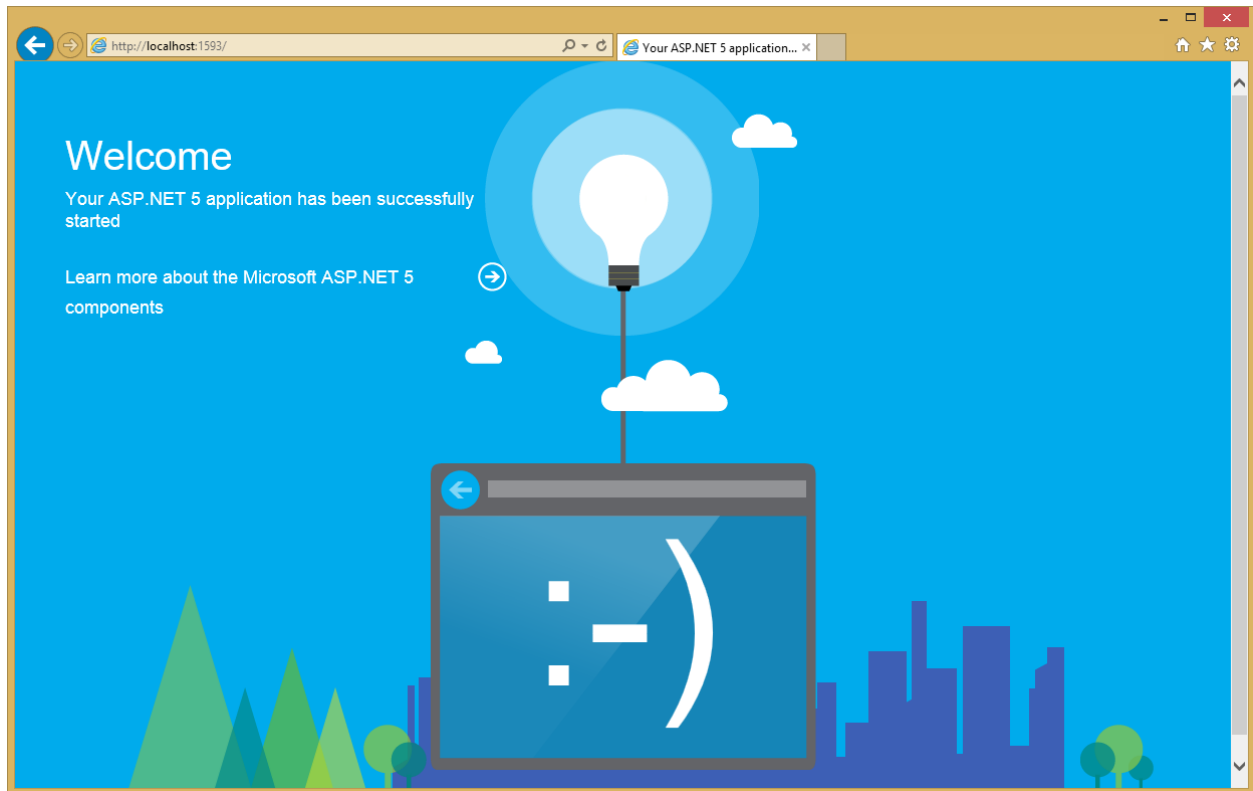
Note: Remember that the `Configure()` method in `Startup.cs` is defining the pipeline that will be used by all requests to your application, which means the order is important. If for example you move the call to `UseRuntimeInfoPage()` after the call to `app.Run()` in the examples shown here, it will never be called because `app.Run()` will handle the request before it reaches the call to `UseRuntimeInfoPage()`.

The welcome page

Another extension method you may find useful, especially when you're first spinning up a new ASP.NET 5 application, is the `UseWelcomePage()` method. Add it to `Configure()` like so:

```
app.UseWelcomePage();
```

Once included, this will handle all requests (by default) with a cool hello world page that uses embedded images and fonts to display a rich view, as shown here:



You can optionally configure the welcome page to only respond to certain paths. The code shown below will configure the page to only be displayed for the `/welcome` path (other paths will be ignored, and will fall through to other handlers):

```
app.UseWelcomePage("/welcome");
```

Configured in this manner, the *startup.cs* shown above will respond to requests as follows:

Table 2.2: Requests

Path	Result
<code>/runtimeinfo</code>	<code>UseRuntimeInfoPage</code> will handle and display runtime info page
<code>/welcome</code>	<code>UseWelcomePage</code> will handle and display welcome page
paths without <code>?throw=</code>	<code>app.Run()</code> will respond with "Hello World!"
paths with <code>?throw=</code>	<code>app.Run()</code> throws an exception; <code>UseErrorPage</code> handles, displays an error page

Summary

In ASP.NET 5, you can easily add error pages, view diagnostic information, or respond to requests with a simple welcome page by adding just one line to your app's *Startup.cs* class.

Additional Resources

- [Using Application Insights](#) - Collect detailed usage and diagnostic data for your application.

2.4.11 Application Insights

By *Steve Smith*

[Application Insights](#) provides development teams with a 360° view across their live application's performance, availability, and usage. It can also [detect and diagnose issues and exceptions](#) in these applications. Telemetry data may be collected from web servers and web clients, as well as desktop and mobile applications.

In this article:

- [Getting started](#)
- [Viewing activity](#)

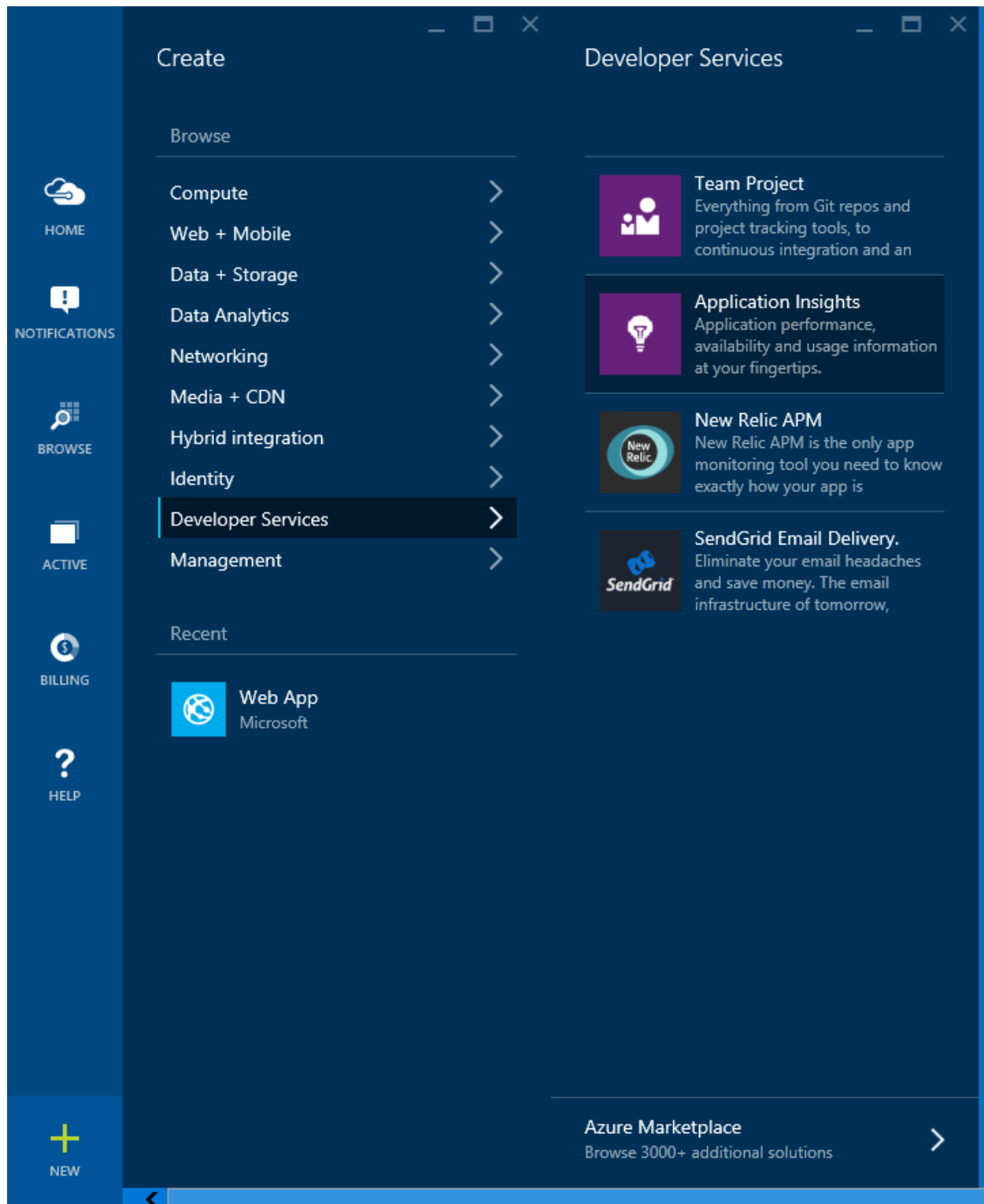
[View or download sample on GitHub.](#)

Getting started

Application Insights, like ASP.NET 5, is in preview.

To get started with Application Insights, you will need a subscription to Microsoft Azure. If your team or organization already has a subscription, you can ask the owner to add you to it using your Microsoft account.

Sign in to the [Azure portal](#) with your account and create a new Application Insights resource.



Choose ASP.NET as the application type. Note the *Instrumentation Key* (under Settings, Properties) associated with the Application Insights resource you've created ([see detailed instructions with more screenshots here](#)). You will need the instrumentation key in a few moments when you configure your ASP.NET 5 application to use Application Insights.

Next, update `project.json` to add a new reference to `Microsoft.ApplicationInsights.AspNet` in

your dependencies section, as shown:

```

1 dependencies: {
2   "EntityFramework.SqlServer": "7.0.0-beta6",
3   "EntityFramework.Commands": "7.0.0-beta6",
4   "Microsoft.AspNet.Mvc": "6.0.0-beta6",
5   "Microsoft.AspNet.Mvc.TagHelpers": "6.0.0-beta6",
6   "Microsoft.AspNet.Authentication.Cookies": "1.0.0-beta6",
7   "Microsoft.AspNet.Authentication.Facebook": "1.0.0-beta6",
8   "Microsoft.AspNet.Authentication.Google": "1.0.0-beta6",
9   "Microsoft.AspNet.Authentication.MicrosoftAccount": "1.0.0-beta6",
10  "Microsoft.AspNet.Authentication.Twitter": "1.0.0-beta6",
11  "Microsoft.AspNet.Diagnostics": "1.0.0-beta6",
12  "Microsoft.AspNet.Diagnostics.Entity": "7.0.0-beta6",
13  "Microsoft.AspNet.Identity.EntityFramework": "3.0.0-beta6",
14  "Microsoft.AspNet.Server.IIS": "1.0.0-beta6",
15  "Microsoft.AspNet.Server.WebListener": "1.0.0-beta6",
16  "Microsoft.AspNet.StaticFiles": "1.0.0-beta6",
17  "Microsoft.AspNet.Tooling.Razor": "1.0.0-beta6",
18  "Microsoft.Framework.Configuration.Abstractions": "1.0.0-beta6",
19  "Microsoft.Framework.Configuration.Json": "1.0.0-beta6",
20  "Microsoft.Framework.Configuration.UserSecrets": "1.0.0-beta6",
21  "Microsoft.Framework.Logging": "1.0.0-beta6",
22  "Microsoft.Framework.Logging.Console": "1.0.0-beta6",
23  "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0-beta6",
24  "Microsoft.ApplicationInsights.AspNet": "1.0.0-beta6"

```

Saving the `project.json` file will download and install the required packages.

Next, edit (or create) the `config.json` file, adding the instrumentation key you noted above from your Application Insights resource in Windows Azure. Specify an “ApplicationInsights” section with a key named “Instrumentation-Key”. Set its value to the instrumentation key.

```

1 {
2   "ApplicationInsights": {
3     "InstrumentationKey": "YOUR KEY GOES HERE"
4   },
5   "Data": {
6     "DefaultConnection": {
7       "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=aspnet5-AppInsightsDemo-8cf4d67e-5
8     }
9   }
10 }

```

Next, in `Startup.cs` you need to configure Application Insights in a few places. In the constructor, where you configure Configuration, add a block to configure Application Insights for development:

```

1 if (env.IsDevelopment())
2 {
3     // This reads the configuration keys from the secret store.
4     // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=53
5     builder.AddUserSecrets();
6
7     // This will push telemetry data through Application Insights pipeline faster, allowing you to v
8     builder.AddApplicationInsightsSettings(developerMode: true);
9 }

```

Note: Setting `AppInsights` in `developerMode` (`configuration.AddApplicationInsightsSettings(developerMode: true)`) will expedite your telemetry through the pipeline so that you can see results immediately ([learn more](#)).

Add the Application Insights telemetry services in your `ConfigureServices()` method:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     // Add Application Insights data collection services to the services container.
4     services.AddApplicationInsightsTelemetry(Configuration);
```

Then, in the `Configure()` method add middleware to allow Application Insights to track exceptions and log information about individual requests. Note that the request tracking middleware should be as the first middleware in the pipeline, while the exception middleware should follow the configuration of error pages or other error handling middleware.

An edited `Startup.cs` is shown below, highlighting the necessary Application Insights code ([view full Startup.cs](#)):

```
1 public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
2 {
3     loggerFactory.MinimumLevel = LogLevel.Information;
4     loggerFactory.AddConsole();
5
6     // Configure the HTTP request pipeline.
7
8     // Add Application Insights to the request pipeline to track HTTP request telemetry data.
9     app.UseApplicationInsightsRequestTelemetry();
10
11     // Add the following to the request pipeline only in development environment.
12     if (env.IsDevelopment())
13     {
14         app.UseBrowserLink();
15         app.UseErrorPage();
16         app.UseDatabaseErrorPage(DatabaseErrorPageOptions.ShowAll);
17     }
18     else
19     {
20         // Add Error handling middleware which catches all application specific errors and
21         // sends the request to the following path or controller action.
22         app.UseExceptionHandler("/Home/Error");
23     }
24
25     // Track data about exceptions from the application. Should be configured after all error handling
26     app.UseApplicationInsightsExceptionTelemetry();
```

Now add the Application Insights scripts to your views. Add the following to the very top of the `_ViewImports.cshtml` file:

```
1 @using AppInsightsDemo
2 @using AppInsightsDemo.Models
3 @using Microsoft.AspNet.Identity
4 @addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
5 @inject Microsoft.ApplicationInsights.Extensibility.TelemetryConfiguration TelemetryConfiguration
```

Then, add the following line of code in your `_Layouts.cshtml` file at the end of the `<head>` section (before any other JavaScript blocks specified there):

```
1 <head>
2     <meta charset="utf-8" />
3     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
4     <title>@ViewData["Title"] - AppInsightsDemo</title>
5
```

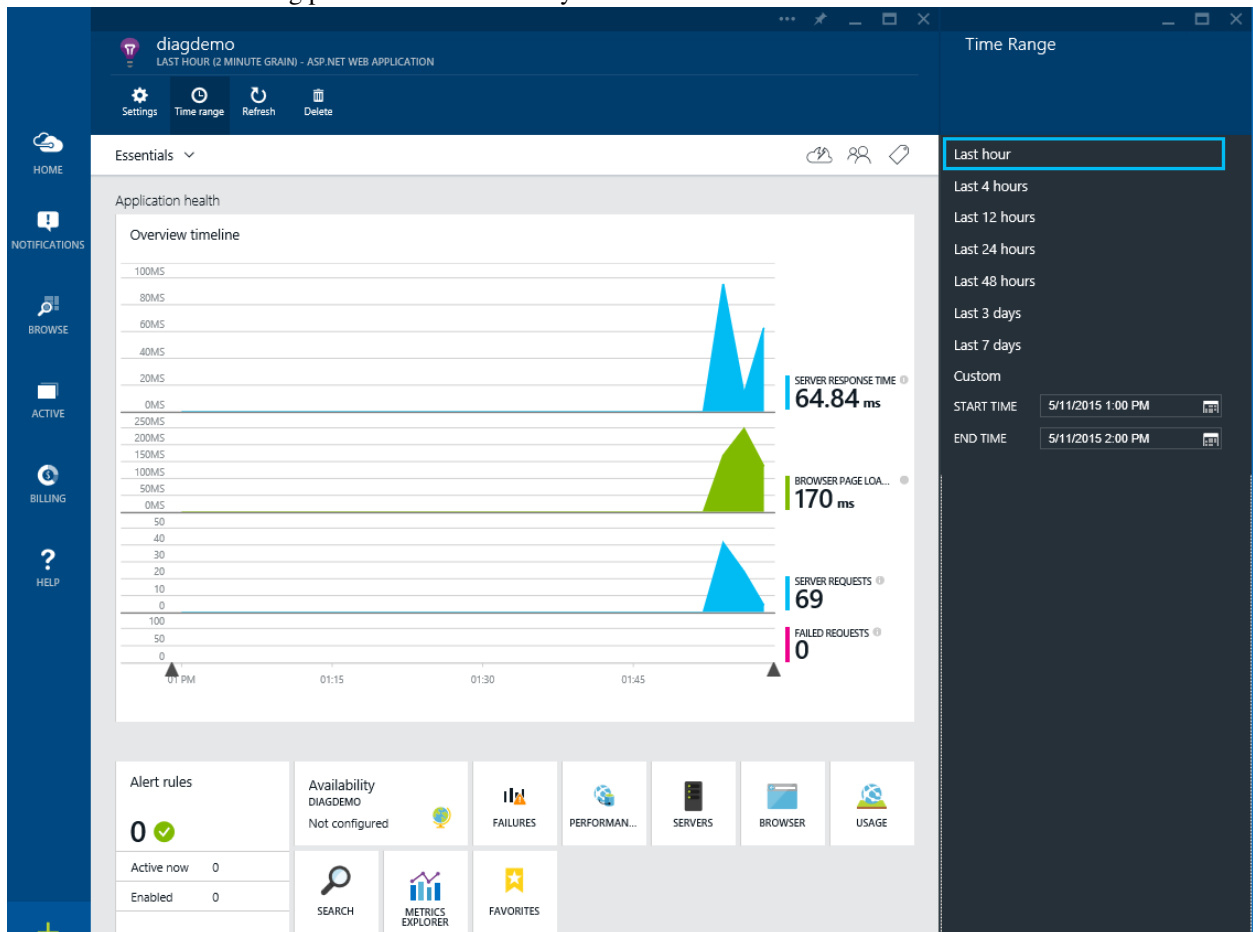
```

6 <environment names="Development">
7   <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
8   <link rel="stylesheet" href="~/lib/bootstrap-touch-carousel/dist/css/bootstrap-touch-carousel.css" />
9   <link rel="stylesheet" href="~/css/site.css" />
10 </environment>
11 <environment names="Staging,Production">
12   <link rel="stylesheet" href="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/bootstrap.min.css"
13       asp-fallback-href="~/lib/bootstrap/css/bootstrap.min.css"
14       asp-fallback-test-class="hidden" asp-fallback-test-property="visibility" asp-fallback-test-value="hidden" />
15   <link rel="stylesheet" href="//ajax.aspnetcdn.com/ajax/bootstrap-touch-carousel/0.8.0/css/bootstrap-touch-carousel.min.css"
16       asp-fallback-href="~/lib/bootstrap-touch-carousel/css/bootstrap-touch-carousel.css"
17       asp-fallback-test-class="carousel-caption" asp-fallback-test-property="display" asp-fallback-test-value="display" />
18   <link rel="stylesheet" href="~/css/site.css" asp-file-version="true" />
19 </environment>
20 @Html.ApplicationInsightsJavaScript (TelemetryConfiguration)
21 </head>

```

Viewing activity

You can view the activity from your site once it's been configured and you've made some requests to it by navigating to the Azure portal. There, you will find the Application Insights resource you configured previously, and you will be able to view charts showing performance and activity data:



In addition to tracking activity and performance data on every page, you can also track specific events. For instance, if you want to know any time a user completes a certain transaction, you can create and track such events individually.

To do so, you should inject the `TelemetryClient` into the controller in question, and call its `TrackEvent` method. In the included sample, we've added event tracking for user registration and successful and failed login attempts. You can see the required code in the excerpt from `AccountController.cs` shown below:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Security.Claims;
5  using System.Security.Principal;
6  using System.Threading.Tasks;
7  using Microsoft.AspNet.Authentication;
8  using Microsoft.AspNet.Authorization;
9  using Microsoft.AspNet.Identity;
10 using Microsoft.AspNet.Mvc;
11 using Microsoft.AspNet.Mvc.Rendering;
12 using Microsoft.Data.Entity;
13 using Microsoft.Data.Entity.Infrastructure;
14 using AppInsightsDemo;
15 using AppInsightsDemo.Models;
16 using AppInsightsDemo.Services;
17 using Microsoft.ApplicationInsights;
18
19 namespace AppInsightsDemo.Controllers
20 {
21     [Authorize]
22     public class AccountController : Controller
23     {
24         private readonly UserManager<ApplicationUser> _userManager;
25         private readonly SignInManager<ApplicationUser> _signInManager;
26         private readonly IEmailSender _emailSender;
27         private readonly ISmsSender _smsSender;
28         private readonly ApplicationDbContext _applicationDbContext;
29         private static bool _databaseChecked;
30         private readonly TelemetryClient _telemetryClient;
31
32         public AccountController(
33             UserManager<ApplicationUser> userManager,
34             SignInManager<ApplicationUser> signInManager,
35             IEmailSender emailSender,
36             ISmsSender smsSender,
37             ApplicationDbContext applicationDbContext,
38             TelemetryClient telemetryClient)
39         {
40             _userManager = userManager;
41             _signInManager = signInManager;
42             _emailSender = emailSender;
43             _smsSender = smsSender;
44             _applicationDbContext = applicationDbContext;
45             _telemetryClient = telemetryClient;
46         }
47
48         //
49         // POST: /Account/Login
50         [HttpPost]
51         [AllowAnonymous]
52         [ValidateAntiForgeryToken]
53         public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
54         {
55             EnsureDatabaseCreated(_applicationDbContext);

```

```

56     ViewData["ReturnUrl"] = returnUrl;
57     if (ModelState.IsValid)
58     {
59         // This doesn't count login failures towards account lockout
60         // To enable password failures to trigger account lockout, set lockoutOnFailure: true
61         var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, mo
62         if (result.Succeeded)
63         {
64             _telemetryClient.TrackEvent("LoginSuccess");
65             return RedirectToLocal(returnUrl);
66         }
67         if (result.RequiresTwoFactor)
68         {
69             return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe
70         }
71         if (result.IsLockedOut)
72         {
73             return View("Lockout");
74         }
75         else
76         {
77             _telemetryClient.TrackEvent("LoginFailure");
78             ModelState.AddModelError(string.Empty, "Invalid login attempt.");
79             return View(model);
80         }
81     }
82
83     // If we got this far, something failed, redisplay form
84     return View(model);
85 }
86
87 //
88 // POST: /Account/Register
89 [HttpPost]
90 [AllowAnonymous]
91 [ValidateAntiForgeryToken]
92 public async Task<IActionResult> Register(RegisterViewModel model)
93 {
94     EnsureDatabaseCreated(_applicationDbContext);
95     if (ModelState.IsValid)
96     {
97         var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
98         var result = await _userManager.CreateAsync(user, model.Password);
99         if (result.Succeeded)
100         {
101             // For more information on how to enable account confirmation and password reset
102             // Send an email with this link
103             //var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
104             //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id,
105             //await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
106             //    "Please confirm your account by clicking this link: <a href=\"" + callback
107             await _signInManager.SignInAsync(user, isPersistent: false);
108             _telemetryClient.TrackEvent("NewRegistration");
109             return RedirectToAction(nameof(HomeController.Index), "Home");
110         }
111         AddErrors(result);
112     }
113 }

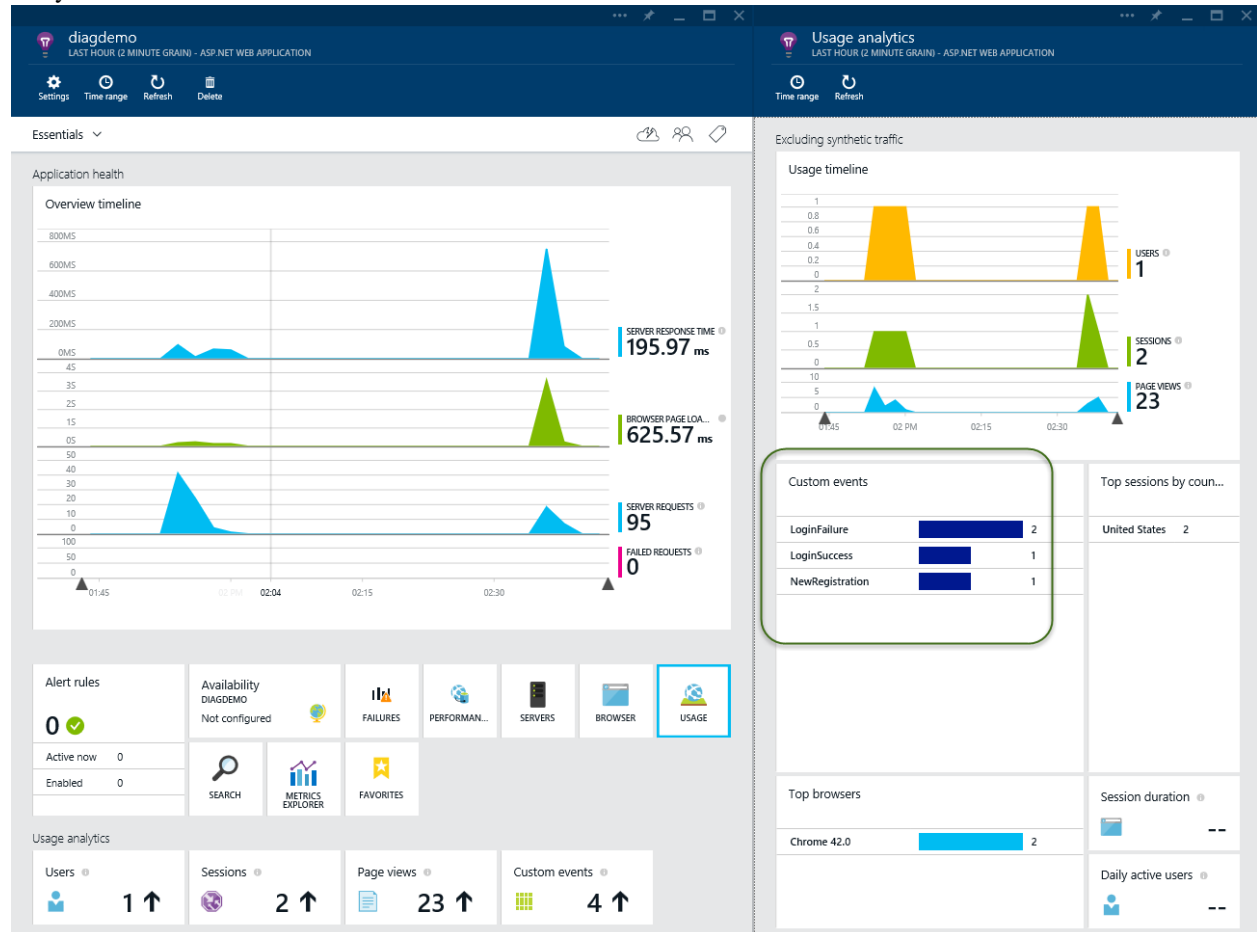
```

```

114     // If we got this far, something failed, redisplay form
115     return View(model);
116 }
117 }
118 }

```

With this in place, testing the application's registration and login feature results in the following activity available for analysis:



Note: Application Insights is still in development. To view the latest release notes and configuration instructions, please [refer to the project wiki](#).

Summary

Application Insights allows you to easily add application activity and performance tracking to any ASP.NET 5 app. With Application Insights in place, you can view live reports showing information about the users of your application and how it is performing, including both client and server performance information. In addition, you can track custom events, allowing to you capture user activities unique to your application.

Additional Resources

- [Application Insights API for custom events and metrics](#)

- [Application Insights for ASP.NET 5](#)

2.4.12 Logging

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.13 Managing Application State

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.14 Working with Multiple Environments

By [Steve Smith](#)

ASP.NET 5 introduces improved support for controlling application behavior across multiple environments, such as development, staging, and production. Environment variables are used to indicate which environment the application is running in, allowing the app to be configured appropriately.

In this article:

- *Development, Staging, Production*
- *Determining the environment at runtime*
- *Startup conventions*

[Browse or download samples on GitHub.](#)

Development, Staging, Production

ASP.NET 5 references a particular [environment variable](#), `ASPNET_ENV`, to describe the environment the application is currently running in. This variable can be set to any value you like, but three values are used by convention: `Development`, `Staging`, and `Production`. You will find these values used in the samples and templates provided with ASP.NET 5.

The current environment setting can be detected programmatically from within ASP.NET 5. In addition, ASP.NET MVC 6 introduces an [Environment Tag Helper](#) that allows MVC Views to include certain sections based on the current application environment.

Note: The specified environment name is case insensitive. Whether you set the variable to `Development` or `development` or `DEVELOPMENT` the results will be the same.

Development

This should be the environment used when developing an application. When using Visual Studio 2015, this setting can be specified in your project's debug profiles, such as for IIS Express, shown here:

The screenshot shows the 'Environments' window in Visual Studio with the 'Debug' tab selected. The 'Profile' is 'IIS Express', 'Launch' is 'IIS Express', and 'Port' is '12345'. The 'Launch Browser' checkbox is checked. Under 'Use Specific Runtime', the 'Version' is '1.0.0-beta4-11566', 'Platform' is '.NET Framework', and 'Architecture' is 'x86'. The 'Environment Variables' table has one entry: 'ASPNET_ENV' with the value 'Development'. The 'Other IIS Express settings' section shows 'Enable SSL' unchecked, 'Enable Anonymous Authentication' checked, and 'Enable Windows Authentication' unchecked.

Name	Value
ASPNET_ENV	Development

When you modify the default settings created with the project, your changes are persisted in `launchSettings.json` in the Properties folder. After modifying the `ASPNET_ENV` variable in the web profile to be set to Staging, the `launchSettings.js` file in our sample project is shown below:

Listing 2.3: `launchSettings.json`

```
{
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNET_ENV": "Development"
      }
    },
    "web": {
      "commandName": "web",
      "environmentVariables": {
        "ASPNET_ENV": "Staging"
      }
    }
  }
}
```

Note: Changes made to project profiles or to `launchSettings.json` directly will not take effect until the web

server being used is restarted.

Staging

By convention, a `Staging` environment is a pre-production environment used for final testing before deployment to production. Ideally, its physical characteristics should mirror that of production, so that any issues that may arise in production occur first in the staging environment, where they can be addressed without impact to users.

Production

The `Production` environment is the environment in which the application runs when it is live and being used by end users. This environment should be configured to maximize security, performance, and application robustness. Some common settings that a production environment might have that would differ from development include:

- Turn on caching
- Ensure all client-side resources are bundled, minified, and potentially served from a CDN
- Turn off diagnostic `ErrorPages`
- Turn on friendly error pages
- Enable production logging and monitoring (e.g. `AppInsights`)

This is by no means meant to be a complete list. It's best to avoid scattering environment checks in many parts of your application. Instead, the recommended approach is to perform such checks within the application's `Startup` class(es) wherever possible

Determining the environment at runtime

The `IHostingEnvironment` service provides the core abstraction for working with environments. This service is provided by the ASP.NET hosting layer, and can be injected into your startup logic via [Dependency Injection](#). The ASP.NET 5 web site template in Visual Studio uses this approach to load environment-specific configuration files (if present) and to customize the app's error handling settings. In both cases, this behavior is achieved by referring to the currently specified environment by calling `EnvironmentName` or `IsEnvironment` on the instance of `IHostingEnvironment` passed into the appropriate method.

If you need to check whether the application is running in a particular environment, use `env.IsEnvironment("environmentname")` since it will correctly ignore case (instead of checking if `env.EnvironmentName == "Development"` for example).

The following code is taken from the default ASP.NET 5 web site template:

The highlighted sections in the example above show several examples of adjusting application configuration based on environment.

In the `Startup()` method (constructor), the configuration of the application is set up to optionally allow for environment-specific configuration files (e.g. `config.Development.json`) that will override any other config settings (because the file is added last in the configuration setup chain - [learn more about ASP.NET configuration](#)). A call to `IsEnvironment()` is also used to ensure that User Secrets are only configured for use in `Development` ([learn more about User Secrets and the Secret Manager](#)).

In `Configure()`, the environment is checked once more, and if the app is running in a `Development` environment, then it enables `BrowserLink` and error pages (which typically should not be run in production). Otherwise, if the app is not running in a development environment, a standard error handling page is configured to be displayed in response to any unhandled exceptions.

Listing 2.4: Startup.cs (some parts removed for brevity)

```
1 public class Startup
2 {
3     public Startup(IHostingEnvironment env, IApplicationEnvironment appEnv)
4     {
5         // Setup configuration sources.
6
7         var builder = new ConfigurationBuilder(appEnv.ApplicationBasePath)
8             .AddJsonFile("config.json")
9             .AddJsonFile($"config.{env.EnvironmentName}.json", optional: true);
10
11         if (env.IsDevelopment())
12         {
13             // This reads the configuration keys from the secret store.
14             // For more details on using the user secret store see http://go.microsoft.com/fwlink/?L
15             builder.AddUserSecrets();
16         }
17         builder.AddEnvironmentVariables();
18         Configuration = builder.Build();
19     }
20
21     public IConfiguration Configuration { get; set; }
22     public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFact
23     {
24         loggerFactory.MinimumLevel = LogLevel.Information;
25         loggerFactory.AddConsole();
26
27         // Configure the HTTP request pipeline.
28
29         // Add the following to the request pipeline only in development environment.
30         if (env.IsDevelopment())
31         {
32             app.UseBrowserLink();
33             app.UseErrorPage();
34             app.UseDatabaseErrorPage(DatabaseErrorPageOptions.ShowAll);
35         }
36         else
37         {
38             // Add Error handling middleware which catches all application specific errors and
39             // sends the request to the following path or controller action.
40             app.UseExceptionHandler("/Home/Error");
41         }
42     }
```

Startup conventions

ASP.NET 5 supports a convention-based approach to configuring an application's startup based on the current environment. You can also programmatically control how your application behaves according to which environment it is in, allowing you to create and manage your own conventions.

When an ASP.NET 5 application starts, the `Startup` class is used to bootstrap the application, load its configuration settings, etc. ([learn more about ASP.NET startup](#)). However, if a class exists named `Startup{EnvironmentName}`, e.g. `StartupDevelopment`, and the `ASPNET_ENV` environment variable matches that name, then that `Startup` class is used instead. Thus, you could configure `Startup` for development, but have a separate `StartupProduction` that would be used when the app is run in production. Or vice versa.

The following `StartupDevelopment` file from this article's sample project is run when the application is set to run in a Development environment:

Listing 2.5: `StartupDevelopment.cs`

```

1 using Microsoft.AspNet.Builder;
2
3 namespace Environments
4 {
5     public class StartupDevelopment
6     {
7         public void Configure(IApplicationBuilder app)
8         {
9             app.UseWelcomePage();
10        }
11    }
12 }
```

Run the application in development, and a welcome screen is displayed. The sample also includes a `StartupStaging` class:

Listing 2.6: `StartupStaging.cs`

```

1 using Microsoft.AspNet.Builder;
2 using Microsoft.AspNet.Http;
3
4 namespace Environments
5 {
6     public class StartupStaging
7     {
8         public async void Configure(IApplicationBuilder app)
9         {
10             app.Run(async context =>
11             {
12                 context.Response.ContentType = "text/plain";
13                 await context.Response.WriteAsync("Staging environment.");
14             });
15         }
16     }
17 }
```

When the application is run with `ASPNET_ENV` set to `Staging`, this `Startup` class is used, and the application will simply display a string stating it's running in a staging environment. The application's default `Startup` class will only run when the environment is not set to either `Development` or `Staging` (presumably, this would be when it is set to `Production`, but you're not limited to only these three options. Also note that if no environment is set, the default `Startup` will run).

In addition to using an entirely separate `Startup` class based on the current environment, you can also make adjustments to how the application is configured within a `Startup` class. The `Configure()` and `ConfigureServices()` methods support environment-specific versions similar to the `Startup` class itself, of the form `Configure[Environment]()` and `Configure[Environment]Services()`. If you define a method `ConfigureDevelopment()` it will be called instead of `Configure()` when the environment is set to development. Likewise, `ConfigureDevelopmentServices()` would be called instead of `ConfigureServices()` in the same environment.

Summary

ASP.NET 5 provides a number of features and conventions that allow developers to easily control how their applications behave in different environments. When publishing an application from development to staging to production, environment variables set appropriately for the environment allow for optimization of the application for debugging, testing, or production use, as appropriate.

Additional Resources

- [Tag Helpers in ASP.NET MVC 6](#) including the Environment Tag Helper
- [Configuration](#)

2.4.15 WebSockets

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.16 Caching

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.17 Error Handling

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.18 Request Features

By [Steve Smith](#)

Individual web server features related to how HTTP requests and responses are handled have been factored into separate interfaces, defined in the [HttpAbstractions repository](#) (the [Microsoft.AspNet.Http.Features package](#)). These abstractions are used by individual server implementations and middleware to create and modify the application's hosting pipeline.

In this article:

- *Feature interfaces*

- *Feature collections*
- *Middleware and request features*

Feature interfaces

ASP.NET 5 defines a number of [Http Feature Interfaces](#), which are used by servers to identify which features they support. The most basic features of a web server are the ability to handle requests and return responses, as defined by the following feature interfaces:

IHttpRequestFeature Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body.

IHttpResponseFeature Defines the structure of an HTTP response, including the status code, headers, and body of the response.

IHttpAuthenticationFeature Defines support for identifying users based on a `ClaimsPrincipal` and specifying an authentication handler.

IHttpUpgradeFeature Defines support for [HTTP Upgrades](#), which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

IHttpBufferingFeature Defines methods for disabling buffering of requests and/or responses.

IHttpConnectionFeature Defines properties for local and remote addresses and ports.

IHttpRequestLifetimeFeature Defines support for aborting connections, or detecting if a request has been terminated prematurely, such as by a client disconnect.

IHttpSendFileFeature Defines a method for sending files asynchronously.

IHttpWebSocketFeature Defines an API for supporting web sockets.

IRequestIdentifierFeature Adds a property that can be implemented to uniquely identify requests.

ISessionFeature Defines `ISessionFactory` and `ISession` abstractions for supporting user sessions.

ITlsConnectionFeature Defines an API for retrieving client certificates.

ITlsTokenBindingFeature Defines methods for working with TLS token binding parameters.

Note: `ISessionFeature` is not a server feature, but is implemented by [SessionMiddleware](#).

Feature collections

The `HttpAbstractions` repository includes a `FeatureModel` package. Its main ingredient is the [FeatureCollection](#) type, which is used frequently by [Servers](#) and their requests, as well as [Middleware](#), to identify which features they support. The `HttpContext` type defined in `Microsoft.AspNet.Http.Abstractions` (not to be confused with the `HttpContext` defined in `System.Web`) provides an interface for getting and setting these features. Since feature collections are mutable, even within the context of a request, middleware can be used to modify the collection and add support for additional features.

Middleware and request features

While servers are responsible for creating the feature collection, middleware can both add to this collection and consume features from the collection. For example, the [StaticFileMiddleware](#) accesses a feature (`IHttpSendFileFeature`) through the [StaticFileContext](#):

Listing 2.7: StaticFileContext.cs

```
public async Task SendAsync()
{
    ApplyResponseHeaders(Constants.Status200Ok);

    string physicalPath = _fileInfo.PhysicalPath;
    var sendFile = _context.GetFeature<IHttpSendFileFeature>();
    if (sendFile != null && !string.IsNullOrEmpty(physicalPath))
    {
        await sendFile.SendFileAsync(physicalPath, 0, _length, _context.RequestAborted);
        return;
    }

    Stream readStream = _fileInfo.CreateReadStream();
    try
    {
        await StreamCopyOperation.CopyToAsync(readStream, _response.Body, _length, _context.RequestAborted);
    }
    finally
    {
        readStream.Dispose();
    }
}
```

In the code above, the `StaticFileContext` class's `SendAsync` method accesses the server's implementation of the `IHttpSendFileFeature` feature (by calling `GetFeature` on `HttpContext`). If the feature exists, it is used to send the requested static file from its physical path. Otherwise, a much slower workaround method is used to send the file (when available, the `IHttpSendFileFeature` allows the operating system to open the file and perform a direct kernel mode copy to the network card).

Note: Use the pattern shown above for feature detection from middleware or within your application. Calls made to `GetFeature` will return an instance if the feature is supported, or `null` otherwise.

Additionally, middleware can add to the feature collection established by the server, by calling `SetFeature<>`. Existing features can even be replaced by middleware, allowing the middleware to augment the functionality of the server. Features added to the collection are available immediately to other middleware or the underlying application itself later in the request pipeline.

The `WebSocketMiddleware` follows this approach, first detecting if the server supports upgrading (`IHttpUpgradeFeature`), and then adding a new `IHttpWebSocketFeature` to the feature collection if it doesn't already exist. Alternately, if configured to replace the existing implementation (via `_options.ReplaceFeature`), it will overwrite any existing implementation with its own.

```
public Task Invoke(HttpContext context)
{
    // Detect if an opaque upgrade is available. If so, add a websocket upgrade.
    var upgradeFeature = context.GetFeature<IHttpUpgradeFeature>();
    if (upgradeFeature != null)
    {
        if (_options.ReplaceFeature || context.GetFeature<IHttpWebSocketFeature>() == null)
        {
            context.SetFeature<IHttpWebSocketFeature>(new UpgradeHandshake(context,
                upgradeFeature, _options));
        }
    }

    return _next(context);
}
```



```
}
```

By combining custom server implementations and specific middleware enhancements, the precise set of features an application requires can be constructed. This allows missing features to be added without requiring a change in server, and ensures only the minimal amount of features are exposed, thus limiting attack surface area and improving performance.

Summary

Feature interfaces define specific HTTP features that a given request may support. Servers define collections of features, and the initial set of features supported by that server, but middleware can be used to enhance these features.

Additional Resources

- [Servers](#)
- [Middleware](#)
- [OWIN](#)

2.4.19 Servers

By [Steve Smith](#)

ASP.NET 5 is completely decoupled from the web server environment that hosts the application. As released, ASP.NET 5 supports IIS and IIS Express, WebListener, and Kestrel web servers, which run on a variety of platforms. Developers and third party software vendors can create their own custom servers as well within which to host their ASP.NET 5 applications.

In this article:

- *[Servers and commands](#)*
- *[Supported features by server](#)*
- *[IIS and IIS Express](#)*
- *[WebListener](#)*
- *[Kestrel](#)*
- *[Choosing a server](#)*
- *[Custom Servers](#)*

[Browse or download samples on GitHub.](#)

Servers and commands

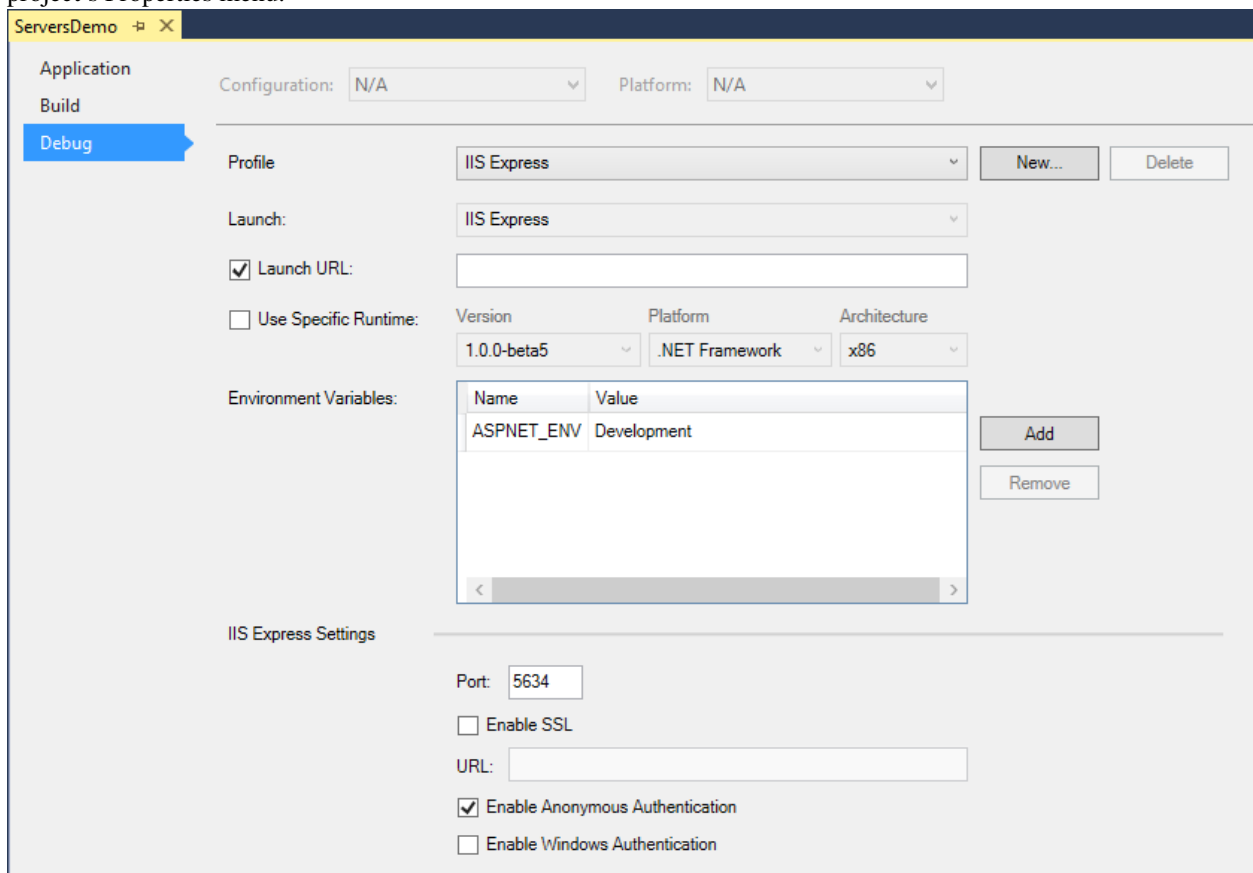
ASP.NET 5 is designed to decouple web applications from the underlying web server that hosts them. Traditionally, ASP.NET applications have been windows-only (unless [hosted via mono](#)) and hosted on the built-in web server for windows, Internet Information Server (IIS) (or a development server, like [IIS Express](#) or earlier development web servers). While IIS is still the recommended way to host production ASP.NET applications on Windows, the cross-platform nature of ASP.NET allows it to be hosted in any number of different web servers, on multiple operating systems.

ASP.NET 5 ships with support for 3 different servers:

- Microsoft.AspNet.Server.IIS
- Microsoft.AspNet.Server.WebListener (WebListener)
- Microsoft.AspNet.Server.Kestrel (Kestrel)

ASP.NET 5 does not directly listen for requests, but instead relies on the HTTP server implementation to surface the request to the application as a set of [feature interfaces](#) composed into an `HttpContext`. Both IIS and WebListener are Windows-only; Kestrel is designed to run cross-platform. You can configure your application to be hosted by any or all of these servers by specifying commands in your `project.json` file. You can even specify an application entry point for your application, and run it as an executable (using `dnx . run`) rather than hosting it in a separate process.

The default web host for ASP.NET applications developed using Visual Studio 2015 is IIS / IIS Express. The “Microsoft.AspNet.Server.IIS” dependency is included in `project.json` by default, even with the Empty web site template. Visual Studio provides support for multiple profiles, associated with IIS Express and any other commands defined in `project.json`. You can manage these profiles and their settings in the Debug tab of your web application project’s Properties menu.



Note: IIS doesn’t support commands; Visual Studio launches IIS Express and loads your application into it when you choose its profile.

I’ve configured the sample project for this article to support each of the different server options in its `project.json` file:

The `run` command will execute the application via its `void main()` method defined in `program.cs`. In this case, this has been set up to configure and start an instance of `WebListener`. This is not a typical means of launching a server, but is shown to demonstrate the possibility (the [Music Store sample application](#) also demonstrates this option).

Listing 2.8: project.json (truncated)

```

1 {
2   "webroot": "wwwroot",
3   "version": "1.0.0-*",
4
5   "dependencies": {
6     "Microsoft.AspNet.Server.IIS": "1.0.0-beta6",
7     "Microsoft.AspNet.Server.WebListener": "1.0.0-beta6",
8     "Kestrel": "1.0.0-beta6"
9   },
10
11  "commands": {
12    "kestrel": "Microsoft.AspNet.Hosting --server Kestrel --server.urls http://localhost:5004",
13    "run": "run server.urls=http://localhost:5003",
14    "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http
15  },
16
17  "frameworks": {

```

Supported Features by Server

ASP.NET defines a number of [Request Features](#) which may be supported on different server implementations. The following table lists the different features and the servers supporting them.

Feature	IIS	WebListener	Kestrel
IHttpRequestFeature	Yes	Yes	Yes
IHttpResponseFeature	Yes	Yes	Yes
IHttpAuthenticationFeature	Yes	Yes	No
IHttpUpgradeFeature	No	Yes (with limits)	Yes
IHttpBufferingFeature	Yes	Yes	No
IHttpConnectionFeature	Yes	Yes	No
IHttpRequestLifetimeFeature	Yes	Yes	No
IHttpSendFileFeature	Yes	Yes	No
IHttpWebSocketFeature	Yes	Yes	No*
IRequestIdentifierFeature	Yes	Yes	No
ITlsConnectionFeature	Yes	Yes	No
ITlsTokenBindingFeature	Yes	Yes	No

To add support for web sockets in Kestrel, use the [WebSocketMiddleware](#)

Configuration options

When launching a server, you can provide it with some configuration options. This can be done directly using command line parameters, or a configuration file containing the settings can be specified. The `Microsoft.AspNet.Hosting` command supports parameters for the server to use (such as Kestrel or WebListener) as well as a `server.urls` configuration key, which should contain a semicolon-separated list of URL prefixes the server should handle.

The `project.json` file shown above demonstrates how to pass the `server.urls` parameter directly:

```
"kestrel": "Microsoft.AspNet.Hosting --server Kestrel --server.urls http://localhost:5004"
```

Alternately, a configuration file can be referenced, instead:

Listing 2.9: program.cs

```
1 using System;
2 using System.Threading.Tasks;
3 using Microsoft.AspNet.Hosting;
4 using Microsoft.Framework.Configuration;
5
6 namespace ServersDemo
7 {
8     /// <summary>
9     /// This demonstrates how the application can be launched in a console application.
10    /// "dnx . run" command in the application folder will invoke this.
11    /// </summary>
12    public class Program
13    {
14        private readonly IServiceProvider _serviceProvider;
15
16        public Program(IServiceProvider serviceProvider)
17        {
18            _serviceProvider = serviceProvider;
19        }
20        public Task<int> Main(string[] args)
21        {
22            //Add command line configuration source to read command line parameters.
23            var builder = new ConfigurationBuilder();
24            builder.AddCommandLine(args);
25            var config = builder.Build();
26
27            using (new WebHostBuilder(_serviceProvider, config)
28                .UseServer("Microsoft.AspNet.Server.WebListener")
29                .Build()
30                .Start())
31            {
32                Console.WriteLine("Started the server..");
33                Console.WriteLine("Press any key to stop the server");
34                Console.ReadLine();
35            }
36            return Task.FromResult(0);
37        }
38    }
39 }
```

```
"kestrel": "Microsoft.AspNet.Hosting --config hosting.ini"
```

Then, `hosting.ini` can include the settings the server will use (including the server parameter, as well):

```
server=Kestrel
server.urls=http://localhost:5004
```

Programmatic configuration

In addition to specifying configuration options, the server hosting the application can be referenced programmatically via the [IApplicationBuilder interface](#), available in the `Configure` method in `Startup`. `IApplicationBuilder` exposes a `Server` property of type `ServerInformation`. `IServerInformation` only exposes a `Name` property, but different server implementations may expose additional functionality. For instance, `WebListener` exposes a `Listener` property that can be used to configure the server's authentication:

```
1 public void Configure(IApplicationBuilder app)
2 {
3     var webListenerInfo = app.Server as Microsoft.AspNet.Server.WebListener.ServerInformation;
4     if (webListenerInfo != null)
5     {
6         webListenerInfo.Listener.AuthenticationManager.AuthenticationSchemes =
7             AuthenticationSchemes.AllowAnonymous;
8     }
9
10    app.Run(async (context) =>
11    {
12        var message = String.Format("Hello World from {0}",
13                                   ((IServerInformation)app.Server).Name);
14        await context.Response.WriteAsync(message);
15    });
16 }
```

IIS and IIS Express

Working with IIS as your server for your ASP.NET application is the default option, and should be familiar to ASP.NET developers who have worked with previous versions of the framework. IIS currently provides support for the largest number of features, and includes IIS management functionality and access to other IIS modules. Hosting ASP.NET 5 on IIS bypasses the legacy `System.Web` infrastructure used by prior versions of ASP.NET, providing a substantial performance gain. Most production ASP.NET web applications being deployed to Windows servers will run on IIS.

WebListener

`WebListener` is a Windows-only server that allows ASP.NET applications to be hosted outside of IIS. It runs directly on the [Http.Sys kernel driver](#), and has very little overhead. It supports the same feature interfaces as IIS; in fact, you can think of `WebListener` as a library version of IIS.

You can add support for `WebListener` to your ASP.NET application by adding the “Microsoft.AspNet.Server.WebListener” dependency in `project.json` and the following command:

```
"web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5004"
```

Kestrel

Kestrel is a cross-platform web server based on [libuv](#), a cross-platform asynchronous I/O library. Kestrel is open-source, and you can [view the Kestrel source on GitHub](#). Kestrel is a great option to at least include support for in your ASP.NET 5 projects so that your project can be easily run by developers on any of the supported platforms. You add support for Kestrel by including “Kestrel” in your project’s dependencies listed in `project.json`.

Learn more about working with Kestrel to create [Your First ASP.NET 5 Application on a Mac](#).

Choosing a server

If you intend to deploy your application on a Windows server, you should prefer the Windows optimized servers (IIS and WebListener). IIS remains the best choice for typical web application deployments on Windows servers, and provides the richest set of features. Choose WebListener instead of IIS only if you need to host your application within your own process or service. Use Kestrel if you intend to develop on or deploy to non-Windows environments.

Custom Servers

In addition to the options listed above, you can create your own server in which to host your ASP.NET application, or use other open source servers. Forking and modifying the `KestrelHttpServer` is one way to quickly create your own custom server, and at the time of this writing the `KestrelHttpServer` repository on GitHub has been forked 55 times. When implementing your own server, you’re free to implement just the feature interfaces your application needs, though at a minimum you must support `IHttpRequestFeature` and `IHttpResponseFeature`.

Since Kestrel is open source, it makes an excellent starting point if you need to implement your own custom server. In fact, like all of ASP.NET 5, you’re welcome to [contribute](#) any improvements you make back to the project.

Kestrel currently supports a limited number of feature interfaces, including `IHttpRequestFeature`, `IHttpResponseFeature`, and `IHttpUpgradeFeature`, but additional features will be added in the future. You can see how these interfaces are implemented and supported by Kestrel in its [ServerRequest class](#), a portion of which is shown below.

Listing 2.10: Kestrel ServerRequest.cs class snippet

```
using Microsoft.AspNet.FeatureModel;
using Microsoft.AspNet.Http.Features;

namespace Kestrel
{
    public class ServerRequest : IHttpRequestFeature,
                                IHttpResponseFeature,
                                IHttpUpgradeFeature
    {
        private FeatureCollection _features;

        private void PopulateFeatures()
        {
            _features.Add(typeof(IHttpRequestFeature), this);
            _features.Add(typeof(IHttpResponseFeature), this);
            _features.Add(typeof(IHttpUpgradeFeature), this);
        }
    }
}
```

The `IHttpUpgradeFeature` interface consists of only one property and one method. Kestrel’s implementation of this interface is shown here for reference:

Listing 2.11: Kestrel ServerRequest.cs IHttpUpgradeFeature implementation

```
bool IHttpUpgradeFeature.IsUpgradableRequest
{
    get
    {
        string[] values;
        if (_frame.RequestHeaders.TryGetValue("Connection", out values))
        {
            return values.Any(value => value.IndexOf("upgrade",
                StringComparison.OrdinalIgnoreCase) != -1);
        }
        return false;
    }
}

async Task<Stream> IHttpUpgradeFeature.UpgradeAsync()
{
    _frame.StatusCode = 101;
    _frame.ReasonPhrase = "Switching Protocols";
    _frame.ResponseHeaders["Connection"] = new string[] { "Upgrade" };
    if (!_frame.ResponseHeaders.ContainsKey("Upgrade"))
    {
        string[] values;
        if (_frame.RequestHeaders.TryGetValue("Upgrade", out values))
        {
            _frame.ResponseHeaders["Upgrade"] = values;
        }
    }
    _frame.ProduceStart();
    return _frame.DuplexStream;
}
```

Summary

Because ASP.NET 5 has completely decoupled ASP.NET applications from IIS or any other web server, it's now possible to host ASP.NET applications on any number of different servers. ASP.NET supports three: IIS, WebListener, and Kestrel, which provide two great options for Windows environments and a third, open-source option that can be used on several different operating systems.

Additional Reading

- [Request Features](#)
- [Hosting](#)

2.4.20 OWIN

By Steve Smith

ASP.NET 5 supports OWIN, the Open Web Interface for .NET, which allows web applications to be decoupled from web servers. In addition, OWIN defines a standard way for middleware to be used in a pipeline to handle individual requests and associated responses. ASP.NET 5 applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

In this article:

- *[Running OWIN middleware in the ASP.NET pipeline](#)*
- *[Using ASP.NET Hosting on an OWIN-based server](#)*
- *[Run ASP.NET 5 on an OWIN-based server and use its WebSockets support](#)*
- *[OWIN keys](#)*

[Browse or download samples on GitHub.](#)

Running OWIN middleware in the ASP.NET pipeline

ASP.NET 5's OWIN support is deployed as part of the [Microsoft.AspNet.Owin](#) package, and the source is in the [HttpAbstractions repository](#). You can import OWIN support into your project by adding this package as a dependency in your project .json file, as shown here:

```
"dependencies": {
  "Microsoft.AspNet.Hosting": "1.0.0-beta5",
  "Microsoft.AspNet.Owin": "1.0.0-beta5",
  "Microsoft.AspNet.Server.IIS": "1.0.0-beta5",
  "Microsoft.AspNet.Server.WebListener": "1.0.0-beta5"
},
```

OWIN middleware conform to the [OWIN specification](#), which defines a `Properties IDictionary<string, object>` interface that must be used, and also requires certain keys be set (such as `owin.ResponseBody`). We can construct a very simple example of middleware that follows the OWIN specification to display “Hello World”, as shown here:

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);
```



```
// OWIN Environment Keys: http://owin.org/spec/owin-1.0.0.html
var responseStream = (Stream)environment["owin.ResponseBody"];
var responseHeaders = (IDictionary<string, string>)environment["owin.ResponseHeaders"];

responseHeaders["Content-Length"] = new string[] { responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
responseHeaders["Content-Type"] = new string[] { "text/plain" };

return responseStream.WriteAsync(responseBytes, 0, responseBytes.Length);
}
```

In the above example, notice that the method returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN. Within the method, this parameter is used to retrieve the `owin.ResponseBody` and `owin.ResponseHeaders` objects from the environment dictionary. Once the headers are set appropriately for the content being returned, a task representing the asynchronous write to the response stream is returned.

Adding OWIN middleware to the ASP.NET pipeline is most easily done using the `UseOwin` extension method. Given the `OwinHello` method shown above, adding it to the pipeline is a simple matter:

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can of course configure other actions to take place within the OWIN pipeline. Remember that response headers should only be modified prior to the first write to the response stream, so configure your pipeline accordingly.

Note: Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        // do something before
        return OwinHello;
        // do something after
    });
});
```

Note: The OWIN support in ASP.NET 5 is an evolution of the work that was done for the [Katana project](#). Katana's `IAppBuilder` component has been replaced by `IApplicationBuilder`, but if you have existing Katana-based middleware, you can use it within your ASP.NET 5 application through the use of a bridge, as shown in the [Owin.IAppBuilderBridge example on GitHub](#).

Using ASP.NET Hosting on an OWIN-based server

OWIN-based servers can host ASP.NET applications, since ASP.NET conforms to the OWIN specification. One such server is [Nowin](#), a .NET OWIN web server. In the sample for this article, I've included a very simple project that references [Nowin](#) and uses it to create a simple server capable of self-hosting ASP.NET 5.

```
1 using System;
2 using System.Collections.Generic;
```

```
3 using System.Net;
4 using System.Threading.Tasks;
5 using Microsoft.AspNet.FeatureModel;
6 using Microsoft.AspNet.Hosting.Server;
7 using Microsoft.AspNet.Owin;
8 using Microsoft.Framework.Configuration;
9 using Nowin;
10
11 namespace NowinSample
12 {
13     public class NowinServerFactory : IServerFactory
14     {
15         private Func<IFeatureCollection, Task> _callback;
16
17         private Task HandleRequest(IDictionary<string, object> env)
18         {
19             return _callback(new OwinFeatureCollection(env));
20         }
21
22         public IServerInformation Initialize(IConfiguration configuration)
23         {
24             var builder = ServerBuilder.New()
25                 .SetAddress(IPAddress.Any)
26                 .SetPort(5000)
27                 .SetOwinApp(OwinWebSocketAcceptAdapter.AdaptWebSockets(HandleRequest));
28
29             return new NowinServerInformation(builder);
30         }
31
32         public IDisposable Start(IServerInformation serverInformation,
33             Func<IFeatureCollection, Task> application)
34         {
35             var information = (NowinServerInformation)serverInformation;
36             _callback = application;
37             INowinServer server = information.Builder.Build();
38             server.Start();
39             return server;
40         }
41
42         private class NowinServerInformation : IServerInformation
43         {
44             public NowinServerInformation(ServerBuilder builder)
45             {
46                 Builder = builder;
47             }
48
49             public ServerBuilder Builder { get; private set; }
50
51             public string Name
52             {
53                 get
54                 {
55                     return "Nowin";
56                 }
57             }
58         }
59     }
60 }
```

`IServerFactory` is an ASP.NET interface that requires an `Initialize` and a `Start` method. `Initialize` must return an instance of `IServerInformation`, which simply includes the server's name (but the specific implementation may provide additional functionality). In this example, the `NowinServerInformation` class is defined as a private class within the factory, and is returned by `Initialize` as required.

`Initialize` is responsible for configuring the server, which in this case is done through a series of fluent API calls that hard code the server to listen for requests (to any IP address) on port 5000. Note that the final line of the fluent configuration of the `builder` variable specifies that requests will be handled by the private method `HandleRequest`.

`Start` is called after `Initialize` and accepts the `IServerInformation` created by `Initialize`, and a callback of type `Func<IFeatureCollection, Task>`. This callback is assigned to a local field and is ultimately called on each request from within the private `HandleRequest` method (which was wired up in `Initialize`).

With this in place, all that's required to run an ASP.NET application using this custom server is the following command in `project.json`:

```

1 {
2   "webroot": "wwwroot",
3   "version": "1.0.0-*",
4
5   "dependencies": {
6     "Microsoft.AspNet.Hosting": "1.0.0-beta5",
7     "Microsoft.AspNet.Owin": "1.0.0-beta5",
8     "Microsoft.AspNet.Http": "1.0.0-beta5",
9     "Nowin": "0.17.0"
10  },
11
12  "commands": {
13    "web": "Microsoft.AspNet.Hosting --server NowinSample"
14  },
15 }
```

When run, this command (equivalent to running `dnx . web` from a command line) will search for a package called “NowinSample” that contains an implementation of `IServerFactory`. If it finds one, it will initialize and start the server as detailed above. Learn more about the built-in ASP.NET [Servers](#).

Run ASP.NET 5 on an OWIN-based server and use its WebSockets support

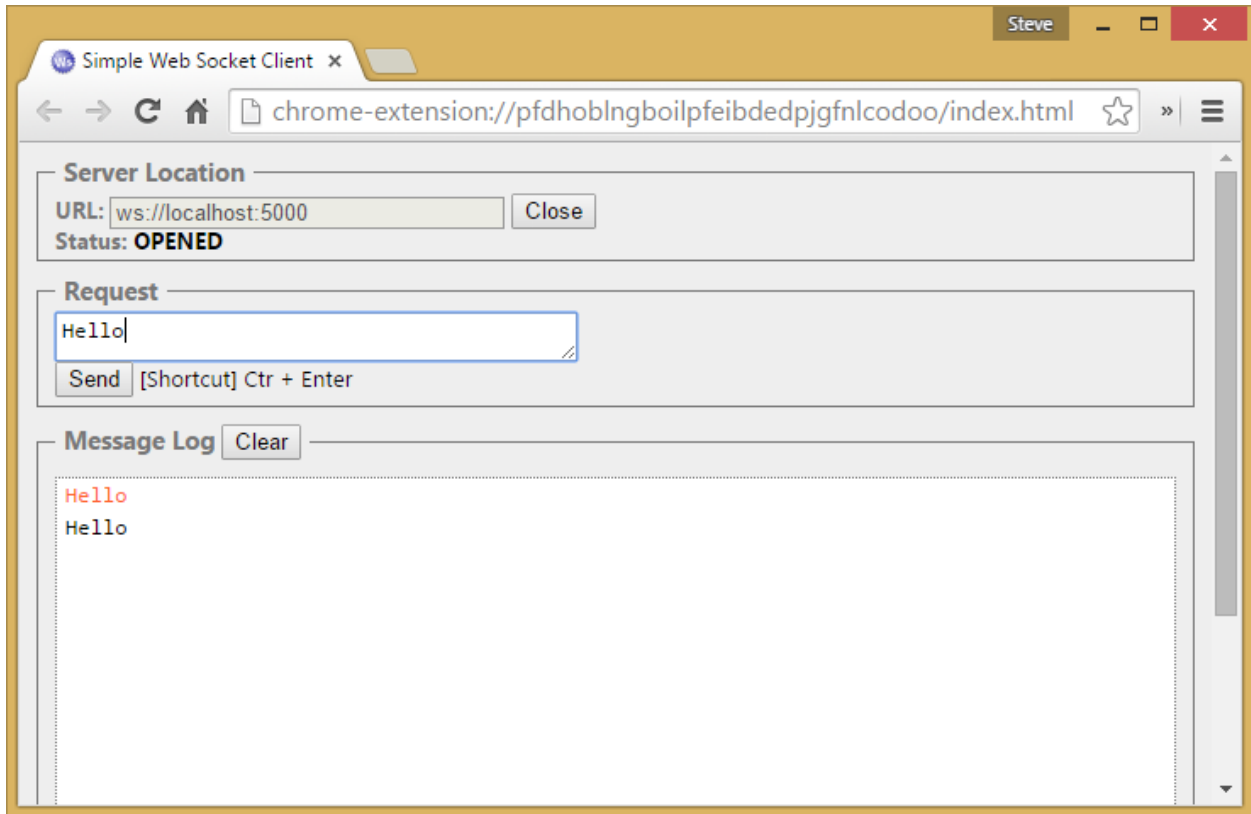
Another example of how OWIN-based servers' features can be leveraged by ASP.NET 5 is access to features like WebSockets. The .NET OWIN web server used in the previous example has support for Web Sockets built in, which can be leveraged by an ASP.NET 5 application. The example below shows a simple web application that supports Web Sockets and simply echos back anything sent to the server via WebSockets.

```

1 public class Startup
2 {
3     public void Configure(IApplicationBuilder app)
4     {
5         app.Use(async (context, next) =>
6         {
7             if (context.WebSockets.IsWebSocketRequest)
8             {
9                 WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
10                await EchoWebSocket(webSocket);
11            }
12            else
13            {
14                await next();
15            }
16        });
17    }
18 }
```

```
15         }
16     });
17
18     app.Run(context =>
19     {
20         return context.Response.WriteAsync("Hello World");
21     });
22 }
23
24 private async Task EchoWebSocket(WebSocket webSocket)
25 {
26     byte[] buffer = new byte[1024];
27     WebSocketReceiveResult received = await webSocket.ReceiveAsync(
28         new ArraySegment<byte>(buffer), CancellationToken.None);
29
30     while (!webSocket.CloseStatus.HasValue)
31     {
32         // Echo anything we receive
33         await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, received.Count),
34             received.MessageType, received.EndOfMessage, CancellationToken.None);
35
36         received = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
37             CancellationToken.None);
38     }
39
40     await webSocket.CloseAsync(webSocket.CloseStatus.Value,
41         webSocket.CloseStatusDescription, CancellationToken.None);
42 }
43 }
44 }
```

This sample ([available on GitHub](#)) is configured using the same `NowinServerFactory` as the previous one - the only difference is in how the application is configured in its `Configure` method. A simple test using a [simple websocket client](#) demonstrates that the application works as expected:



OWIN keys

OWIN depends heavily on an `IDictionary<string, object>` used to communicate information throughout an HTTP Request/Response exchange. ASP.NET 5 implements all of the required and optional keys outlined in the OWIN specification, as well as some of its own. Note that any keys not required in the OWIN specification are optional and may only be used in some scenarios. When working with OWIN keys, it's a good idea to review the list of [OWIN Key Guidelines and Common Keys](#)

Request Data (OWIN v1.0.0)

Key	Value (type)	Description
owin.RequestScheme	String	
owin.RequestMethod	String	
owin.RequestPathBase	String	
owin.RequestPath	String	
owin.RequestQueryString	String	
owin.RequestProtocol	String	
owin.RequestHeaders	<code>IDictionary<string, string[]></code>	
owin.RequestBody	Stream	

Request Data (OWIN v1.1.0)

Key	Value (type)	Description
owin.RequestId	String	Optional

Response Data (OWIN v1.0.0)

Key	Value (type)	Description
owin.ResponseStatusCode	int	Optional
owin.ResponseReasonPhrase	String	Optional
owin.ResponseHeaders	IDictionary<string, string[]>	
owin.ResponseBody	Stream	

Other Data (OWIN v1.0.0)

Key	Value (type)	Description
owin.CallCancelled	CancellationToken	
owin.Version	String	

Common Keys

Key	Value (type)	Description
ssl.ClientCertificate	X509Certificate	
ssl.LoadClientCertAsync	Func<Task>	
server.RemoteIpAddress	String	
server.RemotePort	String	
server.LocalIpAddress	String	
server.LocalPort	String	
server.IsLocal	bool	
server.OnSendingHeaders	Action<Action<object>, object>	

SendFiles v0.3.0

Key	Value (type)	Description
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

Key	Value (type)	Description
opaque.Version	String	
opaque.Upgrade	OpaqueUpgrade	See delegate signature
opaque.Stream	Stream	
opaque.CallCancelled	CancellationToken	

WebSocket v0.3.0

Key	Value (type)	Description
websocket.Version	String	
websocket.Accept	WebSocketAccept	See delegate signature .
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	String	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	WebSocketSendAsync	See delegate signature .
websocket.ReceiveAsync	WebSocketReceiveAsync	See delegate signature .
websocket.CloseAsync	WebSocketCloseAsync	See delegate signature .
websocket.CallCancelled	CancellationToken	
websocket.ClientCloseStatus	int	Optional
websocket.ClientCloseDescription	String	Optional

Summary

ASP.NET 5 has built-in support for the OWIN specification, providing compatibility to run ASP.NET 5 applications within OWIN-based servers as well as supporting OWIN-based middleware within ASP.NET 5 servers.

Additional Resources

- [Middleware](#)
- [Servers](#)

2.4.21 Hosting

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.4.22 Testing

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5 .NET Execution Environment (DNX)

2.5.1 DNX Overview

By [Daniel Roth](#)

What is the .NET Execution Environment?

The .NET Execution Environment (DNX) is a software development kit (SDK) and runtime environment that has everything you need to build and run .NET applications for Windows, Mac and Linux. It provides a host process,

CLR hosting logic and managed entry point discovery. DNX was built for running cross-platform ASP.NET Web applications, but it can run other types of .NET applications, too, such as cross-platform console apps.

Why build DNX?

Cross-platform .NET development DNX provides a consistent development and execution environment across multiple platforms (Windows, Mac and Linux) and across different .NET flavors (.NET Framework, .NET Core and Mono). With DNX you can develop your application on one platform and run it on a different platform as long as you have a compatible DNX installed on that platform. You can also contribute to DNX projects using your development platform and tools of choice.

Build for .NET Core DNX dramatically simplifies the work needed to develop cross-platform applications using .NET Core. It takes care of hosting the CLR, handling dependencies and bootstrapping your application. You can easily define projects and solutions using a lightweight JSON format (*project.json*), build your projects and publish them for distribution.

Package ecosystem Package managers have completely changed the face of modern software development and DNX makes it easy to create and consume packages. DNX provides tools for installing, creating and managing NuGet packages. DNX projects simplify building NuGet packages by cross-compiling for multiple target frameworks and can output NuGet packages directly. You can reference NuGet packages directly from your projects and transitive dependencies are handled for you. You can also build and install development tools as packages for your project and globally on a machine.

Open source friendly DNX makes it easy to work with open source projects. With DNX projects you can easily replace an existing dependency with its source code and let DNX compile it in-memory at runtime. You can then debug the source and modify it without having to modify the rest of your application.

Projects

A DNX project is a folder with a *project.json* file. The name of the project is the folder name. You use DNX projects to build NuGet packages. The *project.json* file defines your package metadata, your project dependencies and which frameworks you want to build for:

```
1 {
2   "version": "1.0.0-*",
3   "description": "ClassLibrary1 Class Library",
4   "authors": [ "author" ],
5   "tags": [ "" ],
6   "projectUrl": "",
7   "licenseUrl": "",
8
9   "dependencies": {
10    "System.Collections": "4.0.10-beta-23109",
11    "System.Linq": "4.0.0-beta-23109",
12    "System.Threading": "4.0.10-beta-23109",
13    "System.Runtime": "4.0.10-beta-23109",
14    "Microsoft.CSharp": "4.0.0-beta-23109"
15  },
16
17  "frameworks": {
18    "dotnet": { }
19  }
20 }
```

All the files in the folder are by default part of the project unless explicitly excluded in *project.json*.

You can also define commands as part of your project that can be executed (see [Commands](#)).

You specify which frameworks you want to build for under the “frameworks” property. DNX will cross-compile for each specified framework and create the corresponding lib folder in the built NuGet package.

You can use the .NET Development Utility (DNU) to build, package and publish DNX projects. Building a project produces the binary outputs for the project. Packaging produces a NuGet package that can be uploaded to a package feed (for example, <http://nuget.org>) and then consumed. Publishing collects all required runtime artifacts (the required DNX and packages) into a single folder so that it can be deployed as an application.

For more details on working with DNX projects see [Working with DNX Projects](#).

Dependencies

Dependencies in DNX consist of a name and a version number. Version numbers should follow [Semantic Versioning](#). Typically dependencies refer to an installed NuGet package or to another DNX project. Project references are resolved using peer folders to the current project or project paths specified using a *global.json* file at the solution level:

```

1 {
2   "projects": [ "src", "test" ],
3   "sdk": {
4     "version": "1.0.0-beta6"
5   }
6 }
```

The *global.json* file also defines the minimum DNX version (“sdk” version) needed to build the project.

Dependencies are transitive in that you only need to specify your top level dependencies. DNX will handle resolving the entire dependency graph for you using the installed NuGet packages. Project references are resolved at runtime by building the referenced project in memory. This means you have the full flexibility to deploy your DNX application as package binaries or as source code.

Packages and feeds

For package dependencies to resolve they must first be installed. You can use DNU to install a new package into an existing project or to restore all package dependencies for an existing project. The following command downloads and installs all packages that are listed in the *project.json* file:

```
dnu restore
```

Packages are restored using the configured set of package feeds. You configure the available package feeds using [NuGet configuration files \(NuGet.config\)](#).

Commands

A command is a named execution of a .NET entry point with specific arguments. You can define commands in your *project.json* file:

```

1 "commands": {
2   "web": "Microsoft.AspNet.Hosting --config hosting.ini",
3   "ef": "EntityFramework.Commands"
4 },
```

You can then use DNX to execute the commands defined by your project, like this:

```
dnx . web
```

Commands can be built and distributed as NuGet packages. You can then use DNU to install commands globally on a machine:

```
dnu commands install MyCommand
```

For more information on using and creating commands see [Using Commands](#).

Application Host

The DNX application host is typically the first managed entry point invoked by DNX and is responsible for handling dependency resolution, parsing *project.json*, providing additional services and invoking the application entry point.

Alternatively, you can have DNX invoke your application's entry point directly. Doing so requires that your application be fully built and all dependencies located in a single directory. Using DNX without using the DNX Application Host is not common.

The DNX application host provides a set of services to applications through dependency injection (for example, *IServiceProvider*, *IApplicationEnvironment* and *ILoggerFactory*). Application host services can be injected in the constructor of the class for your *Main* entry point or as additional method parameters to your *Main* entry point.

Compile Modules

Compile modules are an extensibility point that let you participate in the DNX compilation process. You implement a compile module by implementing the [ICompileModule](#) interface and putting your compile module in a compiler/preprocess or compiler/postprocess in your project.

DNX Version Manager

You can install multiple DNX versions and flavors on your machine. To install and manage different DNX versions and flavors you use the .NET Version Manager (DNVM). DNVM lets you list the different DNX versions and flavors on your machine, install new ones and switch the active DNX.

See [Getting Started](#) for instructions on how to acquire and install DNVM for your platform.

2.5.2 Creating a Cross-Platform Console App with DNX

By [Steve Smith](#)

Using the .NET Execution environment (DNX), it's very easy to run a simple console application.

In this article:

- [Creating a Console App](#)
- [Specifying Project Settings](#)
- [Running the App](#)

You can [view and download the source](#) from the project created in this article.

Creating a Console App

Before you begin, make sure you have successfully installed DNX on your system:

- [Installing on Windows](#)
- [Installing on Mac OS X](#)

Open a console or terminal window in an empty working folder, where `dnx` is configured.

Creating a console application is extremely straightforward. For this article, we're going to use the following C# class, which has just one line of executable code:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5
6 namespace ConsoleApp1
7 {
8     public class Program
9     {
10         public void Main(string[] args)
11         {
12             Console.WriteLine("Hello from DNX!");
13         }
14     }
15 }
```

It really doesn't get any simpler than this. Create a file with these contents and save it as `Program.cs` in your current folder.

Specifying Project Settings

Next, we need to provide the project settings DNX will use. Create a new `project.json` file in the same folder, and edit it to match the listing shown here:

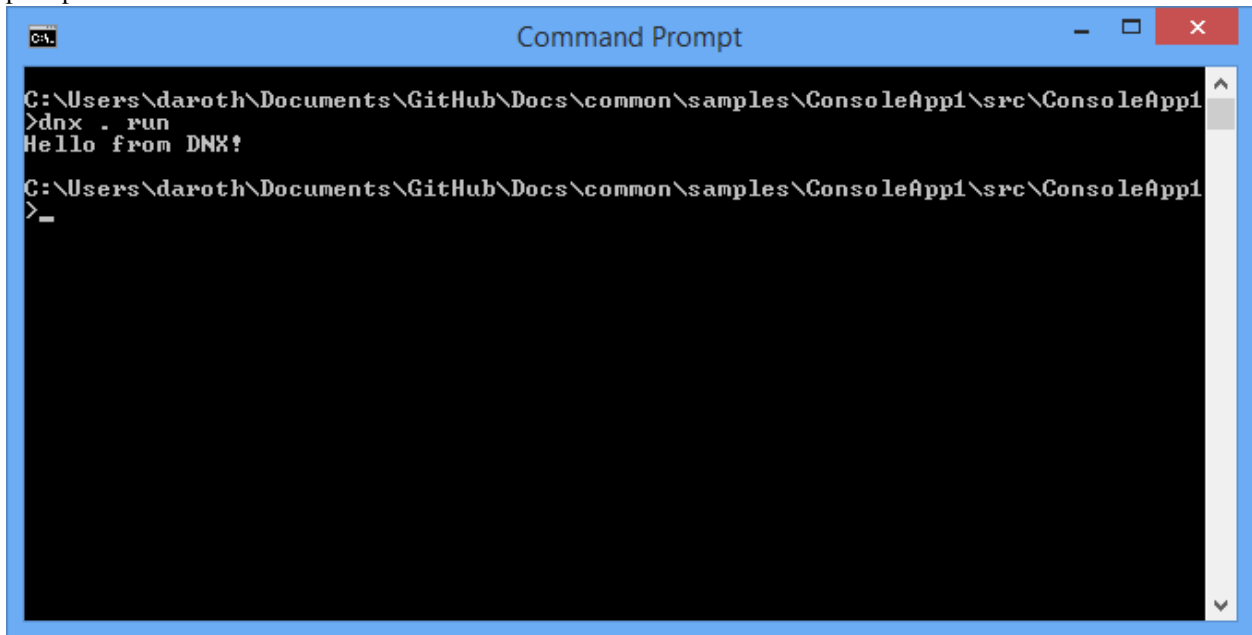
```
1 {
2     "version": "1.0.0-*",
3     "description": "ConsoleApp1 Console Application",
4     "authors": [ "author" ],
5     "tags": [ "" ],
6     "projectUrl": "",
7     "licenseUrl": "",
8
9     "dependencies": {
10     },
11
12     "commands": {
13         "ConsoleApp1": "ConsoleApp1"
14     },
15
16     "frameworks": {
17         "dnx451": { },
18         "dnxcore50": {
19             "dependencies": {
20                 "System.Collections": "4.0.10-beta-23109",
21                 "System.Console": "4.0.0-beta-23109",
22                 "System.Linq": "4.0.0-beta-23109",
23                 "System.Threading": "4.0.10-beta-23109",
24                 "Microsoft.CSharp": "4.0.0-beta-23109"
25             }
26         }
27     }
28 }
```

The `project.json` file defines the app dependencies and target frameworks in addition to various metadata properties about the app. See [Working with DNX Projects](#) for more details.

Save your changes.

Running the App

At this point, we're ready to run the app. You can do this by simply entering `dnx . run` from the command prompt. You should see a result like this one:

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a blue title bar with standard Windows window controls (minimize, maximize, close). The command prompt shows the following text:

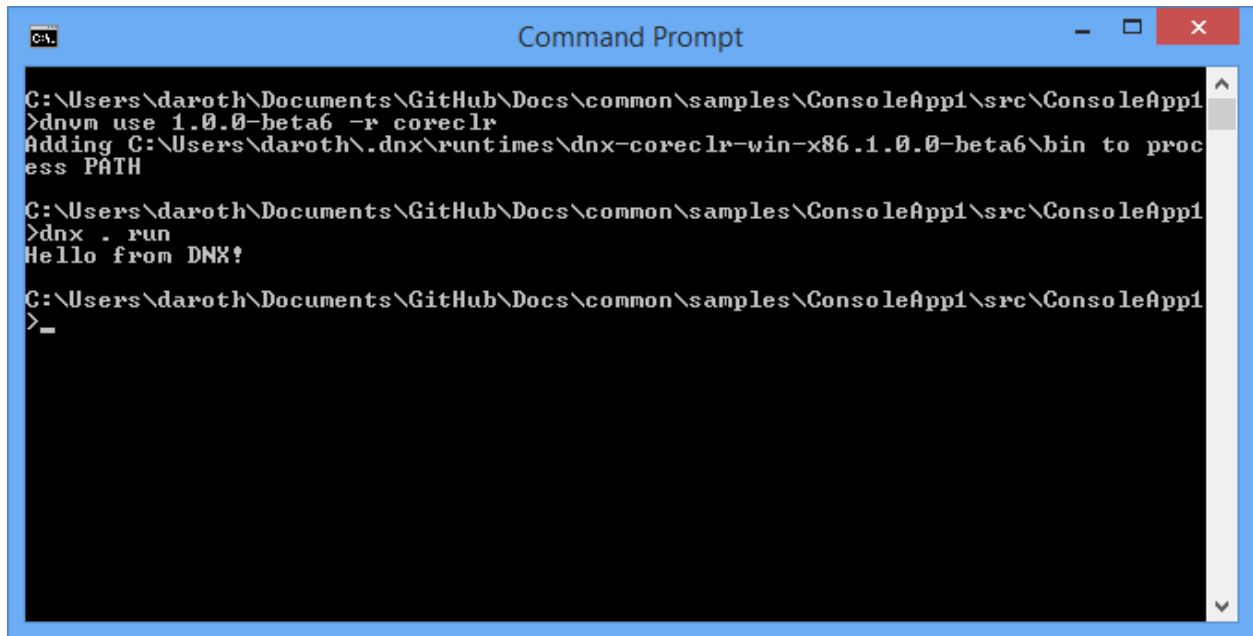
```
C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>dnx . run
Hello from DNX!

C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>_
```

Note: The `dnx` command is used to execute a managed entry point (a `Program.Main` function) in an assembly. The `dnx . run` command is a shorthand for executing the entry point in the current project. It is equivalent to `dnx . [project_name]`

You can select which CLR to run on using the .NET Version Manager (DNVM). To run on CoreCLR first run `dnvm use [version] -r CoreCLR`. To return to using the .NET Framework CLR run `dnvm use [version] -r CLR`.

You can see the app continues to run after switching to use CoreCLR:

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a blue title bar and standard Windows window controls (minimize, maximize, close). The command history is as follows:
C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>dnvm use 1.0.0-beta6 -r coreclr
Adding C:\Users\daroth\dnx\runtimes\dnx-coreclr-win-x86.1.0.0-beta6\bin to process PATH

C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>dnx . run
Hello from DNX!

C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>
The prompt is now a single underscore character.

The `dnx` command references several [environment variables](#), such as `DNX_TRACE`, that affect its behavior.

Set the `DNX_TRACE` environment variable to 1, and run the application again. You should see a great deal more output:

```

C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>set DNX_TRACE=1

C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>dnx . run
The servicing index file at: C:\Program Files (x86)\Microsoft DNX\Servicing\inde
x.txt does not exist or could not be opened.
Loaded module: dnx.clr.dll
Found export: CallApplicationMain
Information: [DomainManager] Using Desktop CLR v4.5.1
Information: [Bootstrapper] Runtime Framework: DNX,Version=v4.5.1
Information: [DefaultHost]: Project path: C:\Users\daroth\Documents\GitHub\Docs\
common\samples\ConsoleApp1\src\ConsoleApp1
Information: [DefaultHost]: Project root: C:\Users\daroth\Documents\GitHub\Docs\
common\samples\ConsoleApp1
Information: [DefaultHost]: Packages path: C:\Users\daroth\dnx\packages
Information: [DependencyWalker]: Walking dependency graph for 'ConsoleApp1 DNX,U
ersion=v4.5.1'
Information: [WalkContext]: Graph walk stage 1 took in 31ms
Information: [DependencyWalker]: Graph walk took 34ms.
Information: [WalkContext]: Populate took 10ms
Information: [DependencyWalker]: Resolved dependencies for ConsoleApp1 in 47ms
Information: [LoaderContainer]: Load name=ConsoleApp1
Information: [ProjectLibraryExportProvider]: GetLibraryExport(ConsoleApp1, DNX,U
ersion=v4.5.1, Debug, >)
Information: [Microsoft.Framework.Runtime.Roslyn.RoslynProjectCompiler]: GetProj
ectReference(ConsoleApp1, DNX,Version=v4.5.1, Debug, >)
Information: [DependencyWalker]: Walking dependency graph for 'ConsoleApp1 DNX,U
ersion=v4.5.1'
Information: [WalkContext]: Graph walk stage 1 took in 0ms
Information: [DependencyWalker]: Graph walk took 0ms.
Information: [WalkContext]: Populate took 0ms
Information: [DependencyWalker]: Resolved dependencies for ConsoleApp1 in 1ms
Information: [ProjectExportProviderHelper]: Resolving references for 'ConsoleApp
1'
Information: [ProjectExportProviderHelper]: Resolved 4 references for 'ConsoleAp
p1' in 8ms
Information: [RoslynCompiler]: Compiling 'ConsoleApp1'
Information: [LoaderContainer]: Load name=System.Security.Cryptography.Hashing,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Information: [LoaderContainer]: Load name=System.Security.Cryptography.Hashing,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Information: [LoaderContainer]: Load name=System.Security.Cryptography.Hashing.A
lgorithms, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Information: [LoaderContainer]: Load name=System.Security.Cryptography.Hashing.A
lgorithms, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Information: [LoaderContainer]: Load name=System.IO.FileSystem, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Information: [LoaderContainer]: Load name=System.IO.FileSystem, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Information: [RoslynCompiler]: Compiled 'ConsoleApp1' in 96ms
Information: [RoslynProjectReference]: Emitting assembly for ConsoleApp1
Information: [CompilationContext]: Generating resources for ConsoleApp1
Information: [CompilationContext]: Generated resources for ConsoleApp1 in 4ms
Information: [RoslynProjectReference]: Emitted ConsoleApp1 in 163ms
Information: [ProjectAssemblyLoader]: Loaded name=ConsoleApp1 in 315ms
Hello from DNX!

C:\Users\daroth\Documents\GitHub\Docs\common\samples\ConsoleApp1\src\ConsoleApp1
>_

```

Summary

Creating and running your first console application on DNX is very simple, and only requires two files.

2.5.3 Working with DNX Projects

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.4 Compilation

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.5 Loaders

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.6 Services

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.7 Using Commands

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.8 Servicing and Updates

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.9 Design Time Host

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.10 Diagnosing Project Dependency Issues

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

For information on diagnosing project dependency issues please see <http://davidfowl.com/diagnosing-dependency-issues-with-asp-net-5/>.

2.5.11 Create a New NuGet Package with DNX

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.12 Migrating an Existing NuGet Package Project

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.13 Global.json Reference

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.5.14 Project.json Reference

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.6 Working with Data

2.6.1 Building Web Applications with Entity Framework

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.7 Publishing and Deployment

2.7.1 Publishing to IIS

By [Rick Anderson](#)

In this article:

- *Publish from Visual Studio*
- *Xcopy to IIS Server*
- *Additional Resources*

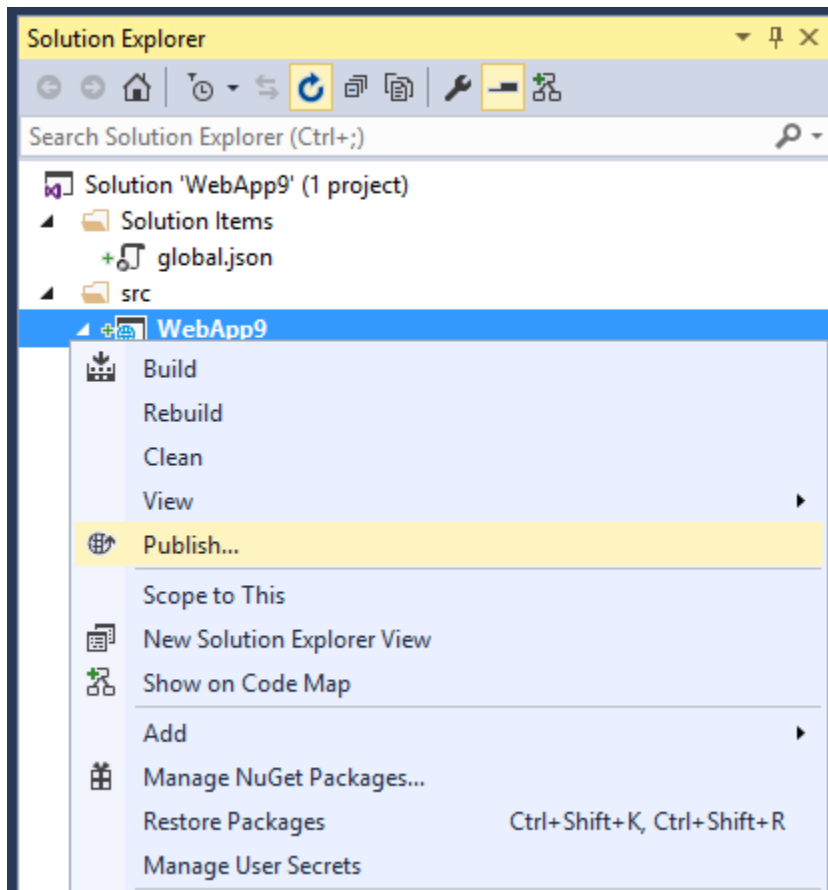
Publish from Visual Studio

1. Create an ASP.NET 5 app. In this sample, I'll create a MVC 6 app using the **Web Site** template under **ASP.NET 5 Preview Templates**. If you're not using the web and gen commands in your development and production work flow, you can remove them from the *project.json* file.

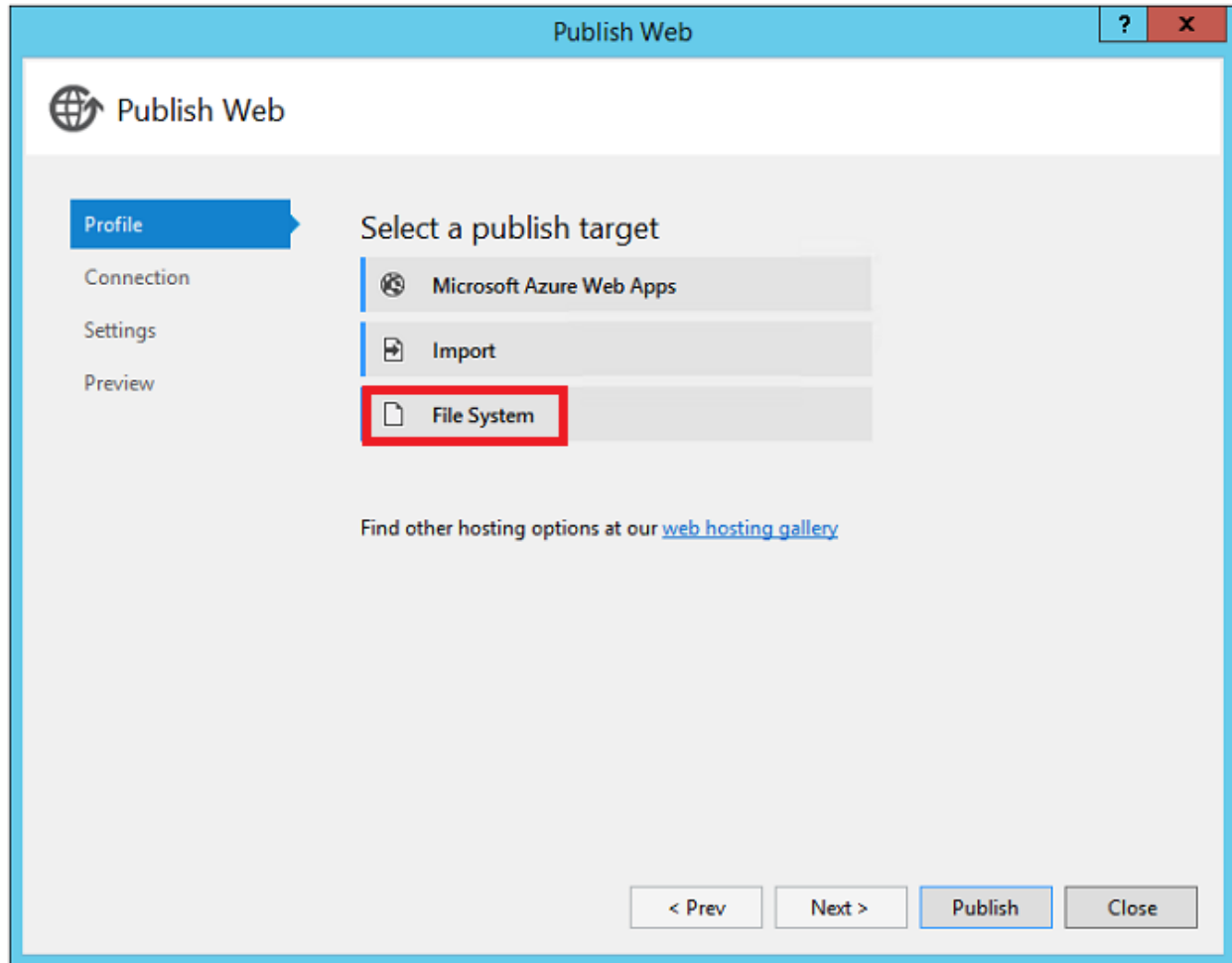
```
"commands": {
  "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5000",
  "gen": "Microsoft.Framework.CodeGeneration"
},
```

With these commands in the *project.json* file, the publish folder will contain web and gen scripts. See [DNX Commands](#) for more information.

2. In **Solution Explorer**, right-click on the project and select **Publish**.



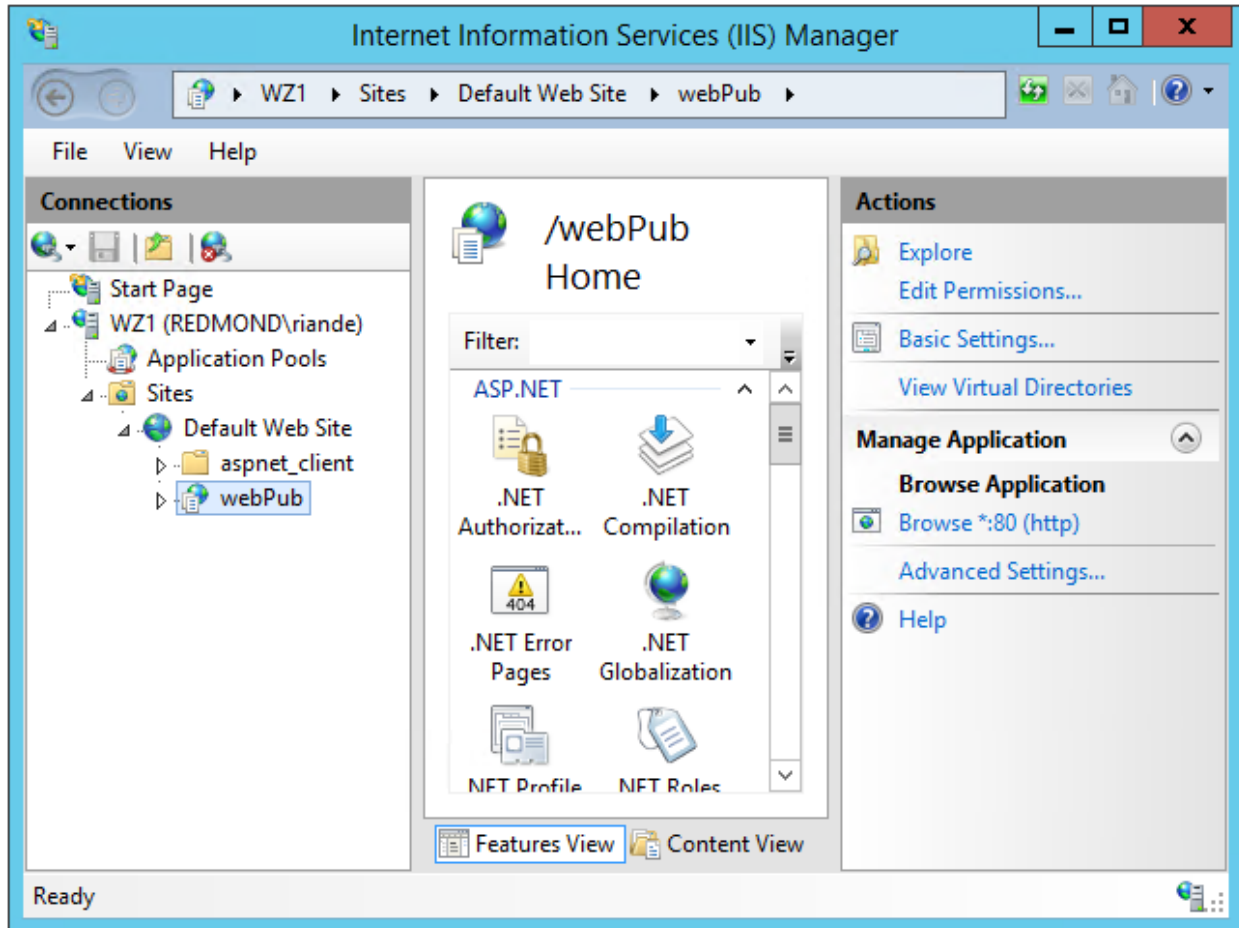
3. In the **Publish Web** dialog, on the **Profile** tab, select **File System**.



4. Enter a profile name. Click **Next**.
5. On the **Connection** tab you can change the publishing target path from the default `..\artifacts\bin\WebApp9\Release\Publish folder`. Click **Next**.
6. On the **Settings** tab you can select the configuration, target DNX version and publish options. Currently your deployment server must have .NET 4.5.1 or higher to use DNX core. We hope to have a native module in the future that will allow you to use DNX core in the app pool regardless of the full CLR. If you select **Precompile during publishing**, the `Publish\approot\src` directory and source files will not be created. If you don't check **Precompile during publishing**, your source files will be found in the `Publish\approot\src` directory. Click **Next**.
7. The **Preview** tab shows you the publish path (by default, the same directory as the ".sln" solution file).

Xcopy to IIS Server

1. Navigate to the the publish folder (`..\artifacts\bin\WebApp9\Release\Publish folder` in this sample).
2. Copy the **approot** and **wwwroot** directories to the target IIS server.
3. In IIS manager, configure the app with application path to the **wwwroot** path. You can click on **Browse *.80(http)** to see your deployed app in the browser.



Additional Resources

- [Understanding ASP.NET 5 Web Apps](#)
- [Introducing .NET Core](#)

2.7.2 Publishing to a Windows Virtual Machine on Azure

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.7.3 Publishing to an Azure Web App with Continuous Deployment

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.7.4 Publish to a Docker Image

Docker is a lightweight container engine, similar in some ways to a virtual machine, which you can use to host applications and services.

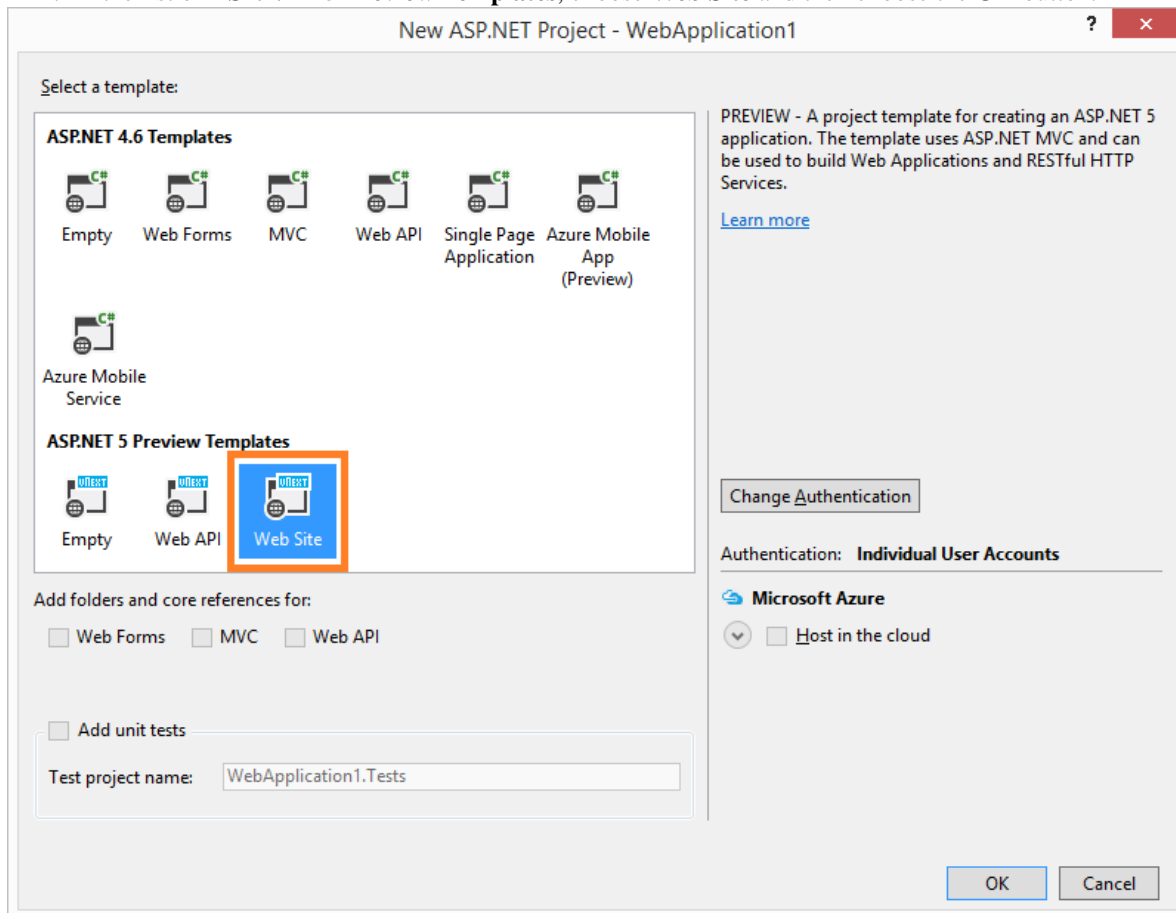
This example shows you how to use the Visual Studio 2015 RC Tools for Docker extension to publish an ASP.NET 5 app to an Ubuntu Linux virtual machine (referred to here as a Docker host) on Azure with the Docker extension installed along with an ASP.NET 5 web application. You can publish the app to a new Docker host hosted on Azure, or to an on-premise server, Hyper-V, or Boot2Docker host by using the **Custom Host** setting. After publishing your app to a Docker host, you can use Docker command-line tools to interact with the container your app has been published to.

Create and publish a new Docker container

In these procedures, you create a new ASP.NET 5 web application project, publish a Docker container to Azure, and then publish the web app project to the Docker container.

Add an ASP.NET 5 web application project

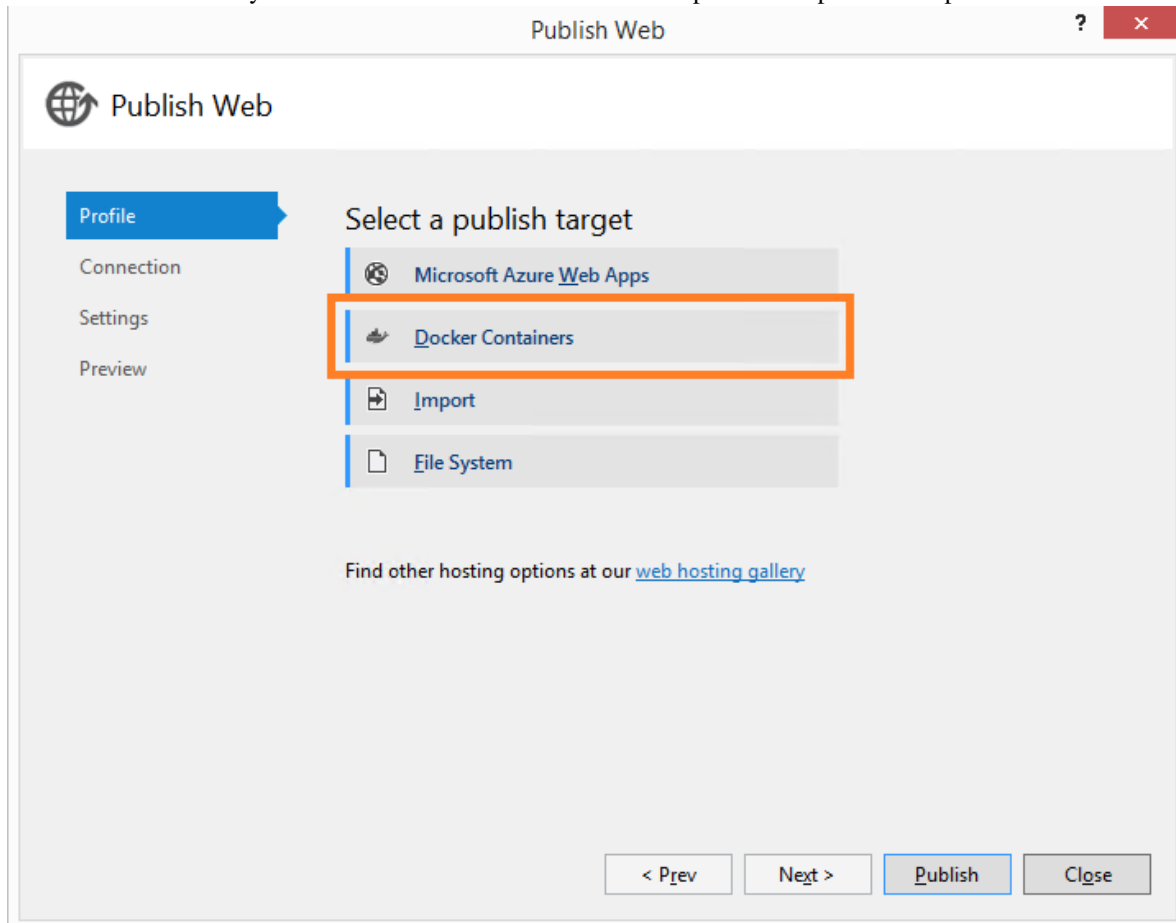
1. Create a new ASP.NET web application project. On the main menu, choose **File, New Project**. Under **C#, Web**, choose **ASP.NET Web Application**.
2. In the list of **ASP.NET 5 Preview Templates**, choose **Web Site** and then choose the **OK** button.



Publish the project

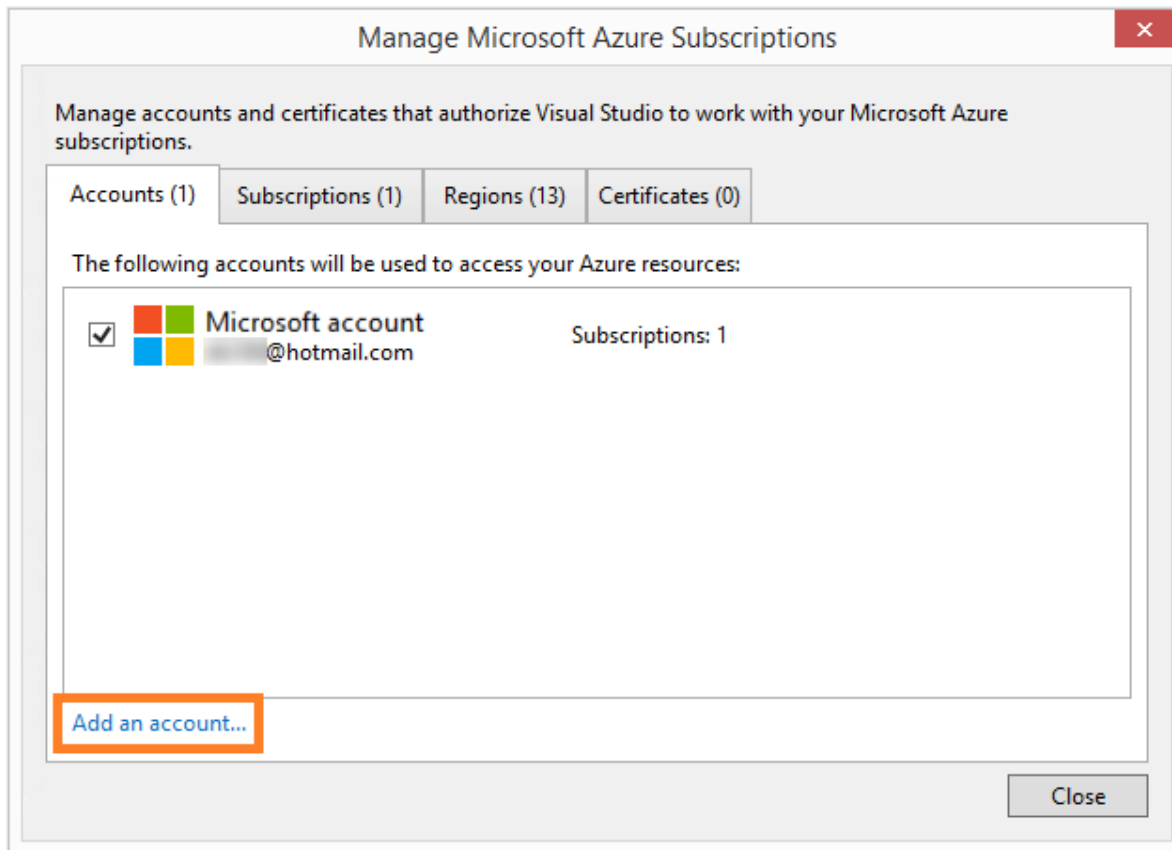
1. On the ASP.NET project's context menu, choose **Publish**.
2. In the **Select a publish target** section of the **Publish Web** dialog box, choose the **Docker Containers** button.

If you don't see a Docker Containers option, make sure you have installed the [Visual Studio 2015 RC Tools for Docker](#) and that you selected an ASP.NET 5 Web Site template in the previous step.

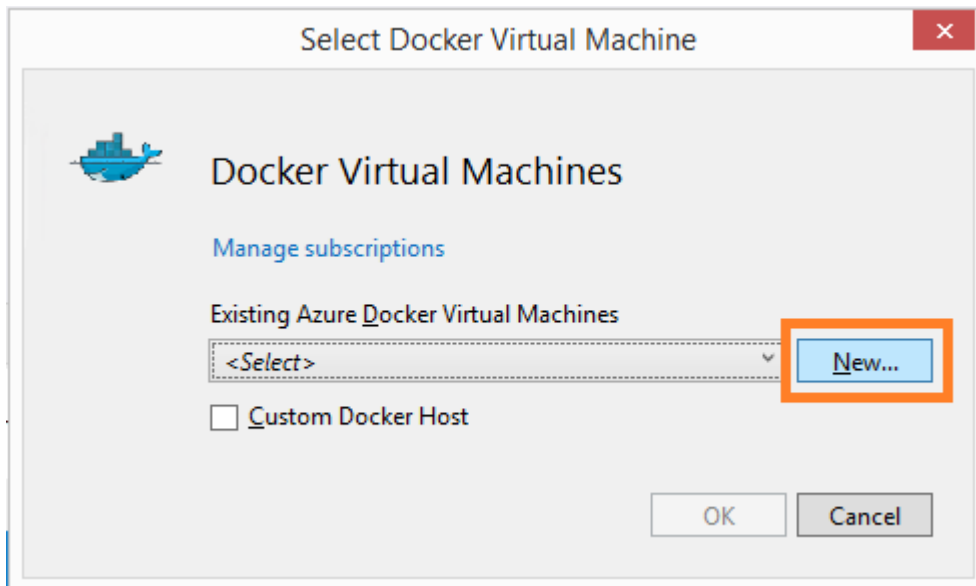


The **Select Docker Virtual Machine** dialog box appears. This lets you specify the Docker host in which you want to publish the project. You can choose to create a new Docker host or choose an existing VM hosted on Azure or elsewhere. For this example, we'll use an Azure Docker host.

3. If you're already logged into Azure, skip to step 5. If you're not logged in, choose the **Manage subscriptions** link.
4. In the **Manage Microsoft Azure Subscriptions** dialog box and choose an existing Azure account. If you aren't logged into Azure, choose the **Add an account** link, sign in to Azure, and then click the **Close** button.



5. Choose an existing Docker host or create a new one. If you're using an existing Docker host, choose it in the **Existing Azure Docker Virtual Machines** list, choose the **OK** button, and then go to step 7. Otherwise, choose the **New** button and continue to the next step.



As an alternative, you can choose to publish to a custom Docker host. See *Provide a custom Docker host* later in this topic for more information.

6. Enter the following information in the **Create a virtual machine on Microsoft Azure** dialog box. When you're

done, choose the **OK** button. This creates a Linux virtual machine with a configured Docker extension.

Create virtual machine on Microsoft Azure

Create a virtual machine on Microsoft Azure [Learn more](#)

[Manage Accounts...](#)

Subscription: Visual Studio Ultimate with MSDN ()@hotmail. ▾

DNS name: DockerTest987 ✓

Image: Ubuntu Server 14.04 LTS ▾

Size: Basic_A0 (1 cores, 768 MB) ▾

Username:

New password:

Confirm password:

Location: West US ▾

Subnet: ▾

Docker server port: 2376

Docker CA certificate: ...

Docker server certificate: ...

Docker server key: ...

☒ Auto-generate Docker certificates

[Online privacy statement](#) **OK** **Cancel**

Property Name	Setting
DNS Name	Enter a unique name for the virtual machine. If the name is accepted by Azure, a green circle with a white checkmark appears to the right. If the name isn't accepted, a red circle with a white x appears. In that case, enter a new unique name.
Image	Enter an OS image to use in the Docker host, if any. For this example, leave this setting at Ubuntu Server 14.04 LTS .
Username	Enter a unique user name for the virtual machine.
Password	Enter a password for the local user and then confirm it.
Location	Change this setting to the region closest to your location.
Auto-generate Docker certificates	Check this box if you want certificates and keys to be automatically generated for you. Clear this box if you want to provide existing certificates and keys.

7. After you choose **OK**, the virtual machine will begin to be created.

You'll get a message that the virtual machine is being created in Azure. You can check on the progress of this operation in the **Output** window.

8. After the Docker host is fully provisioned in Azure, you can check your account on the Azure portal. The virtual machine will appear under the **Virtual Machine** category on the Azure portal.
9. Now that the Docker host is ready, go back and publish the web app project. On the context menu for the web application project node in **Solution Explorer** choose **Publish**.
10. On the **Connection** tab in the **Publish Web** dialog box, choose the **Validate Connection** box to make sure the Docker host is ready. If the connection is good, choose the **Publish** button to publish the web app.

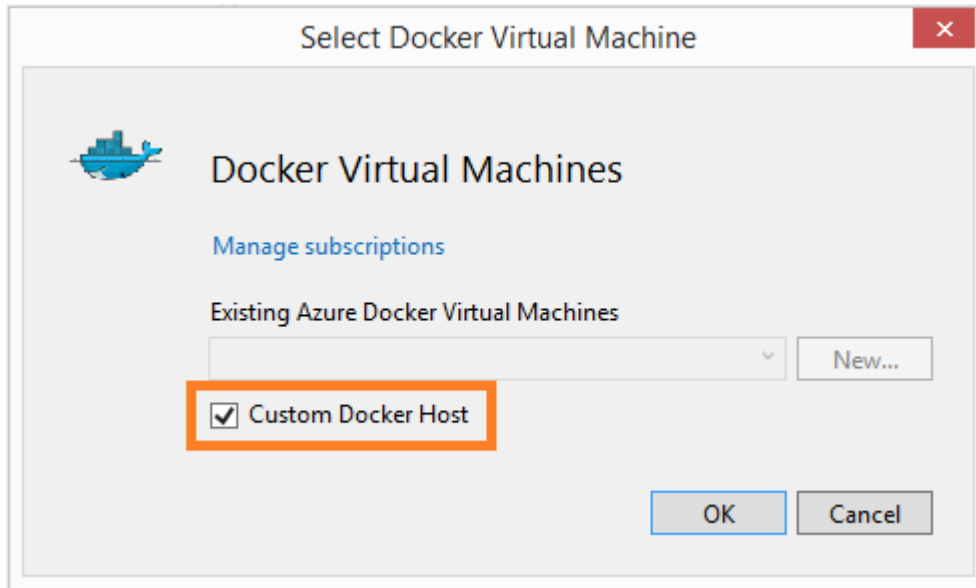
The first time you publish an app to a Docker host, it will take time to download any of the base images that are referenced in your Dockerfile (such as **FROM** *imagename* in the Dockerfile).

Provide a custom Docker host

The previous procedure had you create a Docker virtual machine hosted on Azure. However, if you already have an existing Docker host elsewhere, you can choose to publish to it instead of Azure.

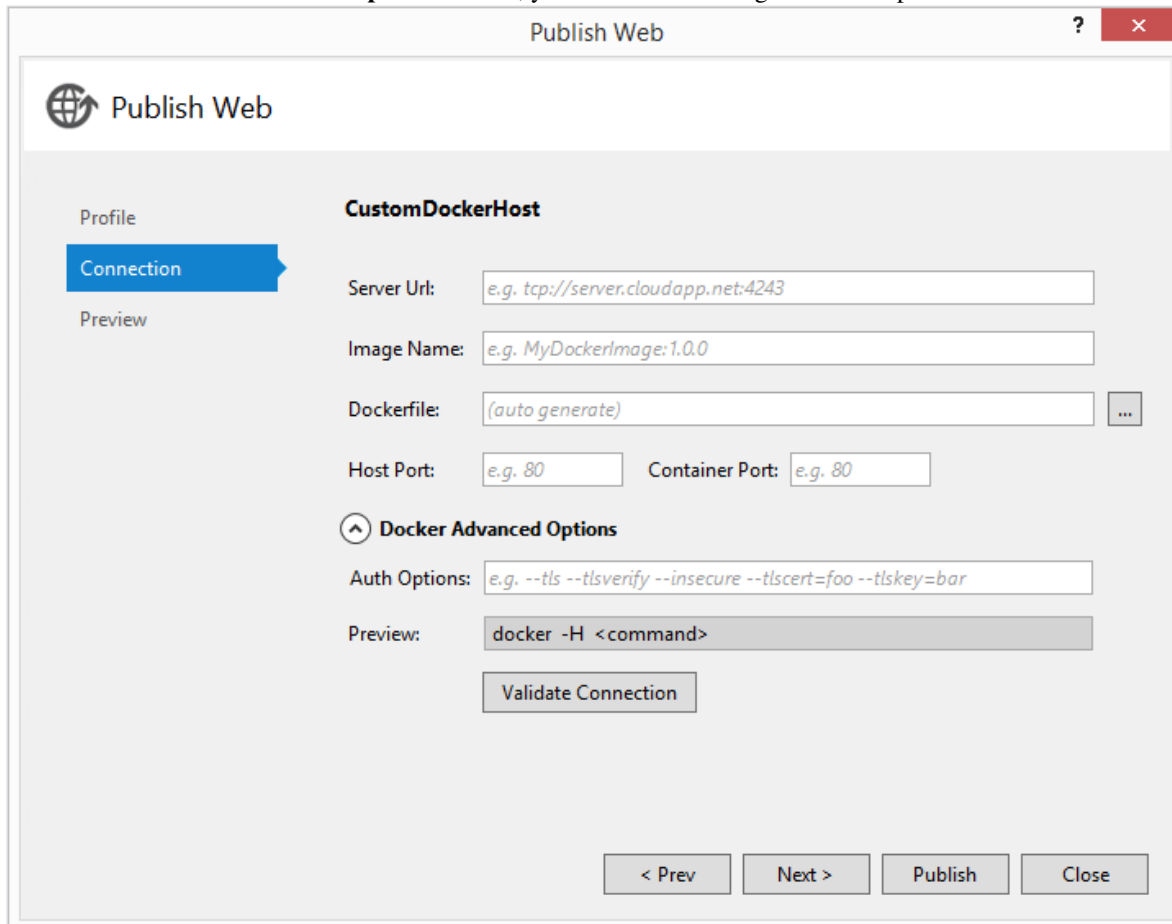
How to provide a custom Docker host

1. In the **Select Docker Virtual Machine** dialog box, select the **Custom Docker Host** check box.



2. Choose the **OK** button.
3. In the **Publish Web** dialog box, add values to the settings in the **CustomDockerHost** section, such as: the server URL, image name, Dockerfile location, and host and container port numbers.

In the **Docker Advanced Options** section, you can view or change the Auth options and Docker command line.



4. After you've entered in all the required values, choose the **Validate Connection** button to ensure the connection to the Docker host works properly.
5. If the connection works properly, choose the **Next** button to see a list of the components that will be published, or choose the **Publish** button to immediately publish the project.

Test the Docker host

Now that the project has been published to a Docker host on Azure, let's test it by checking its settings. Because the Docker command line tools install with the Visual Studio extension, you can issue commands to Docker from a Windows command prompt.

The procedure below is for communicating with a Docker host that's been deployed to Azure.

How to test the Docker host

1. Open a Windows command prompt.
2. Assign the Docker host to an environment variable. To do this, enter the following command (Substitute the name of your Docker host for <NameofAzureVM>):

```
Set docker_host=tcp://<NameofAzureVM>.cloudapp.net:2376
```

Invoking this command prevents you from having to add `-H (Host)` `tcp://<NameofAzureVM>.cloudapp.net:2376` to every command you issue.

3. If you want, you can issue commands like these to test that the Docker host is present and functioning.

Command line	Description
<code>docker --tls info</code>	Get Docker version info.
<code>docker --tls ps</code>	Get a list of running containers.
<code>docker --tls ps -a</code>	Get a list of containers, including ones that are stopped.
<code>docker --tls logs <Docker container name></code>	Get a log for the specified container.
<code>docker --tls images</code>	Get a list of images.

For a full list of Docker commands, simply enter the command `docker` in the command prompt. For more information, see [Docker Command Line](#).

Next steps

Now that you have a Docker host, you can issue Docker commands to it. To learn more about Docker, see the [Docker documentation](#) and the [Docker online tutorial](#).

See also

[Troubleshooting Docker Errors](#)

2.7.5 How to Customize Publishing

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.8 Client-Side Development

2.8.1 Using Gulp

By [Noel Rice](#) and [Mike Wasson](#)

Modern web development has lots of moving parts. To build a typical web app, you might:

- Compile LESS or SASS files to CSS.
- Compile CoffeeScript or TypeScript files to JS.
- Bundle and minify your JS files.
- Run tools like JSHint.

A *task runner* is an app that automates these routine development tasks. Visual Studio 2015 provides built-in support for two popular JavaScript-based task runners, [Gulp](#) and [Grunt](#).

Using Gulp

This topic will cover the following typical steps in using Gulp in your ASP.NET project.

- Use NPM to add the Gulp.js package.
- Use NPM to add one or more [Gulp plugins](#). By itself, Gulp is just an engine for running tasks. Plugins are the modules that actually do the work.
- Define Gulp tasks.
- Run tasks in Task Runner Explorer.
- Bind tasks to build events.

ASP.NET Project Templates

New ASP.NET Web Applications created in Visual Studio 2015 start with the *Empty*, *Web API*, and **Web Site* templates. The *Web Site* template includes NPM and Gulp by default. NPM is preconfigured to add the Gulp.js package. Gulp is configured to copy Bootstrap, Hammer and jQuery client libraries to your web app directory.

Add the Gulp package

NPM (Node Package Manager) is a package manager that was originally created for Node.js. We'll use NPM to install the Gulp.js package. Add an NPM configuration file.

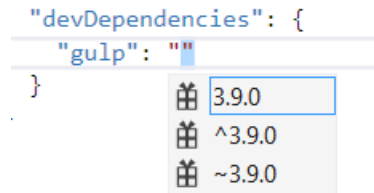
1. In Solution Explorer, right-click the project.
2. Select **Add > New Item**.
3. Select **NPM configuration file**.
4. Leave the default name, "package.json".
5. Click **Add**.

Open package.json. In the dependencies section, add an entry for "gulp":

```
{
  "version": "1.0.0",
  "name": "MyWebApp",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.0"
  }
}
```

The packages are identified using the numbering scheme **<major>.<minor>.<patch>** according to the [SemVer](#) (semantic versioning) standard.

After typing the name of the package and a colon, Intellisense shows a simplified list of versions with the most common choices:



```
"devDependencies": {
  "gulp": ""
}
```

- The top item in the Intellisense list is considered the latest stable version of the package.
- The carat ^ symbol matches the most recent major version.
- The tilde ~ matches the most recent minor version.

Add the Gulp config file

1. In Solution Explorer, right-click the project.
2. Select Add > New Item.
3. Select **Gulp Configuration file**.
4. Leave the default name, “gulpfile.js”.
5. Click **Add**.
6. Right-click **Dependencies > NPM** and select **Restore Packages** from the context menu.

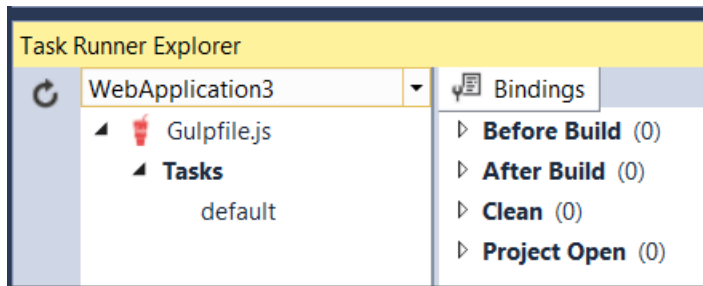
Define a “Hello World” task

Now you’re ready to define some Gulp tasks. Open gulpfile.js and add the following line.

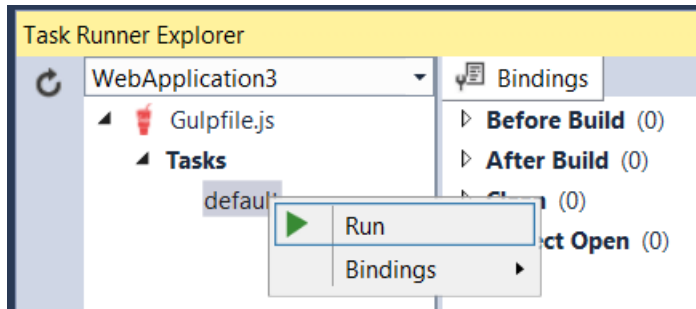
```
var gulp = require('gulp');

gulp.task('default', function () {
  console.log('hello, world');
});
```

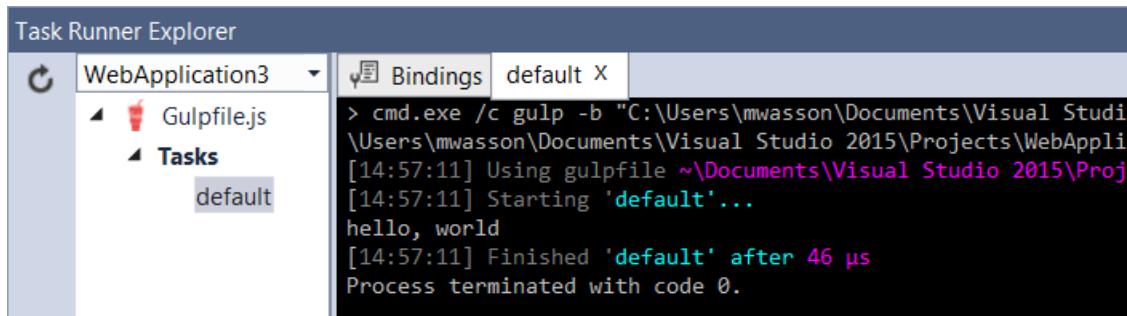
In Solution Explorer, right-click gulpfile.js and select Task Runner Explorer. This opens the Task Runner Explorer window. Task Runner Explorer shows the list of Gulp tasks. So far, we’ve only defined one (‘default’).



Right-click the default task and select **Run**.



You will see the output from the task. In this case, the task just writes “hello, world” to the console.



Use Gulp to copy files

Here is an example of a more useful task.

```
var gulp = require('gulp');

var paths = {
  src: "./Assets/**/*.js",
  dest: "./wwwroot/js/"
}

gulp.task('default', function () {
  return gulp.src(paths.src) // Returns a stream
    .pipe(gulp.dest(paths.dest)) // Pipes the stream somewhere
});
```

This task copies JS files from an *Assets* folder into *wwwroot/js*.

- The `gulp.src` method returns a stream of files. You can use [file globbing](#) to match multiple files. In this case, we are matching every `.js` file under `Assets`.
- The `gulp.dest` method writes the streamed files to a destination folder.

- The pipe method pipes the files from src to dest.

The real power of Gulp is that you can pipe a file through multiple plugins. The output from each stage becomes the input to the next. For example, your pipeline might have these stages:

compile TypeScript > run JSHint > minify

Using Gulp to run JSHint

To do real work with Gulp, you'll use [plugins](#). This section shows an example of using the *JSHint* plugin to detect JavaScript problems. The example also demonstrates creating a “cleanup” task to remove files and directories.

Open package.json and add entries for “gulp-jshint” and “del”:

```
{
  "version": "1.0.0",
  "name": " MyWebApp",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.0",
    "gulp-jshint": "1.11.0",
    "del": "1.2.0"
  }
}
```

In the Solution Explorer, right-click Dependencies > NPM and choose **Restore Packages** from the context menu.

Edit gulpfile.js:

```
var gulp = require('gulp');
var jshint = require('gulp-jshint');
var del = require('del');

var paths = {
  src: "./Assets/**/*.js",
  dest: "./wwwroot/js/"
}

gulp.task("clean", function () {
  del(paths.dest + '**/*');    // Delete everything in 'wwwroot/js'
});

gulp.task('default', ['clean'], function () {
  return gulp.src(paths.src)    // Returns a stream
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(gulp.dest(paths.dest)) // Pipes the stream somewhere
});
```

Now the ‘default’ task includes JSHint in the pipeline:

```
.pipe(jshint())
.pipe(jshint.reporter('default'))
```

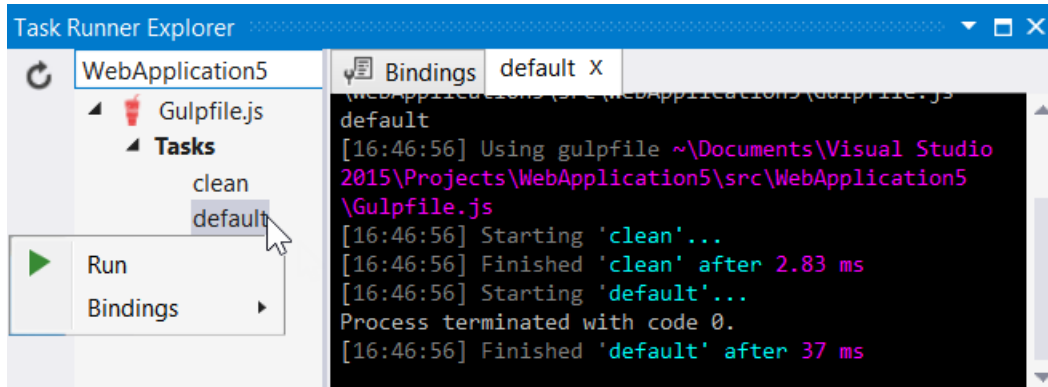
In addition, ‘default’ now includes the ‘clean’ task as a dependency:

```
gulp.task('default', ['clean'], function () {
```

The ‘clean’ task deletes everything under wwwroot/js, using the *del* module. (For more information, see <https://github.com/gulpjs/gulp/blob/master/docs/recipes/delete-files-folder.md>) Now when you run the ‘default’ task,

Gulp will run ‘clean’ first.

In Task Runner Explorer, right-click the **default** task and select **Run**.

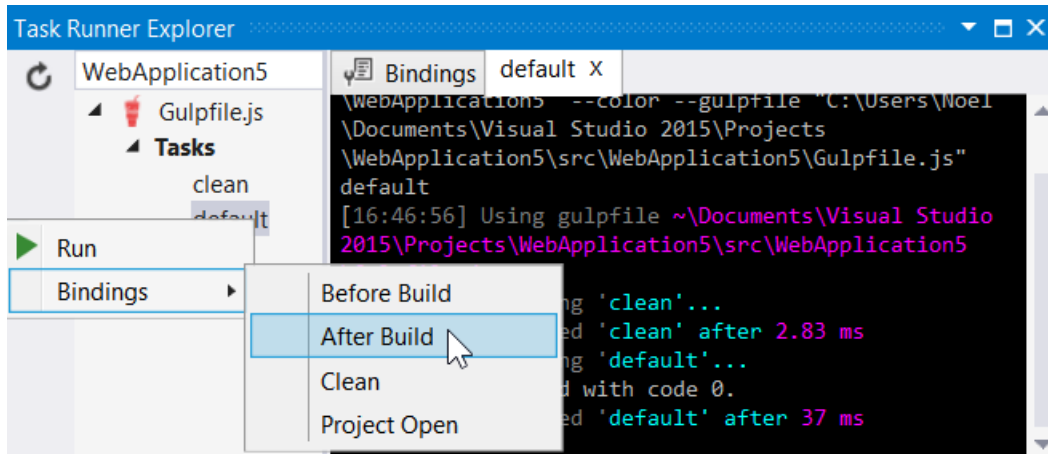


Now the Task Runner Explorer first runs the ‘clean’ task, then the ‘default’ task.

Bind tasks to build events

Unless you want to manually start every task in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Let’s bind ‘default’ so that it runs every time Visual Studio builds the project. In Task Runner Explorer, right-click the ‘default’ task and select **Bindings > After Build** from the context menu.



In the Solution Explorer, right-click and **Build** the project. After the project build, the ‘default’ task runs automatically.

See Also

- [Using Grunt](#)

2.8.2 Using Grunt

By Noel Rice

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality “lint” tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio 2015, though the ASP.NET project templates use Gulp by default (see [Using Gulp](#)).

In this article:

- *Preparing the application*
- *Configuring NPM*
- *Configuring Grunt*
- *Watching for changes*
- *Binding to Visual Studio events*

This example uses the Empty ASP.NET 5 project template as its starting point, to show how to automate the client build process from scratch.

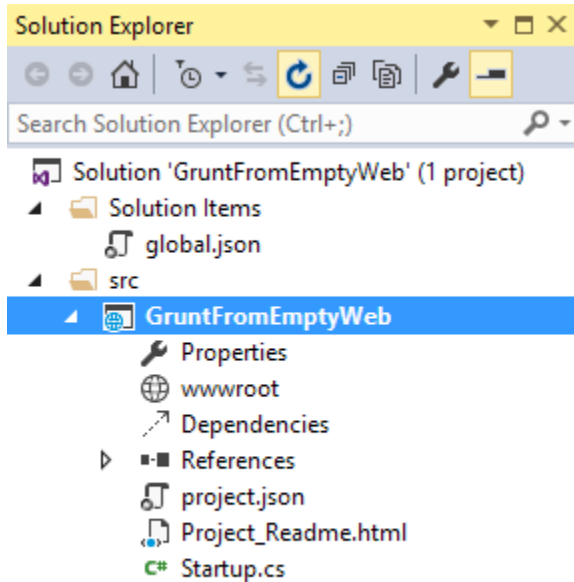
The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.
- **grunt-contrib-clean**: A task that removes files or directories.
- **grunt-contrib-jshint**: A task that reviews JavaScript code quality.
- **grunt-contrib-concat**: A task that joins files into a single file.
- **grunt-contrib-uglify**: A task that minifies JavaScript to reduce size.
- **grunt-contrib-watch**: A task that watches file activity.

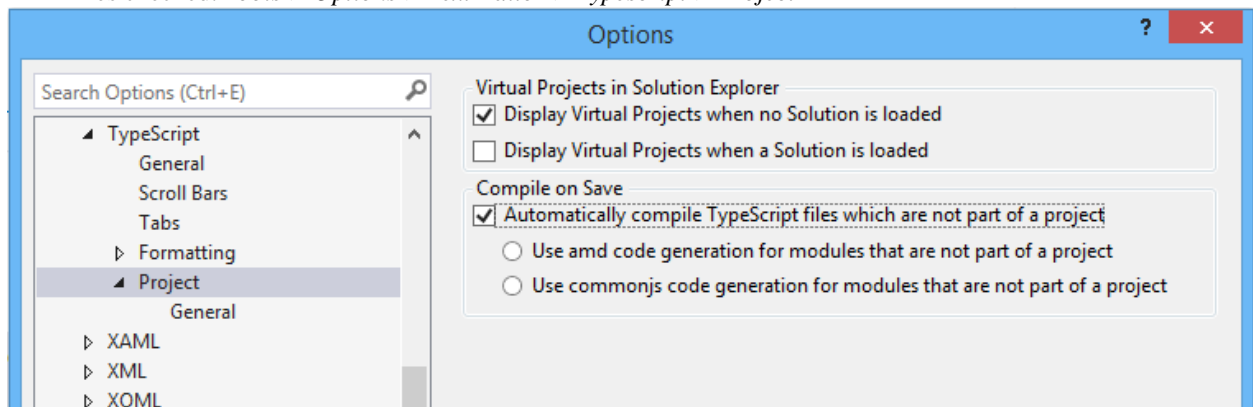
Preparing the application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio 2015 settings and will be our raw material to process using Grunt.

1. In Visual Studio 2015, create a new ASP.NET Web Application.
2. In the **New ASP.NET Project** dialog, select the **ASP.NET 5 Empty** template and click the OK button.
3. In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.



4. Add a new folder named TypeScript to your project directory.
5. Before adding any files, let's make sure that Visual Studio 2015 has the option 'compile on save' for TypeScript files checked. *Tools > Options > Text Editor > TypeScript > Project*



6. Right-click the TypeScript directory and select **Add > New Item** from the context menu. Select the **JavaScript file** item and name the file **Tastes.ts** (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new Tastes.js file will appear with the JavaScript source). `enum Tastes { Sweet, Sour, Salty, Bitter }`

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it **Food.ts**. Copy the code below into the file.

```
class Food {
    constructor(name: string, calories: number) {
        this._name = name;
        this._calories = calories;
    }

    private _name: string;
    get Name() {
        return this._name;
    }
}
```

```

    private _calories: number;
    get Calories() {
        return this._calories;
    }

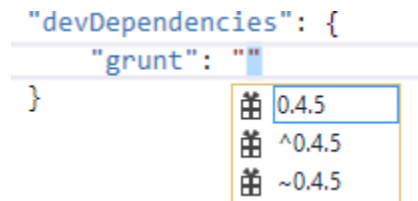
    private _taste: Tastes;
    get Taste(): Tastes { return this._taste }
    set Taste(value: Tastes) {
        this._taste = value;
    }
}

```

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

1. In the Solution Explorer, right-click the project and select **Add > New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, `package.json`, and click the **Add** button.
2. In the `package.json` file, inside the `devDependencies` object braces, enter “grunt”. Select grunt from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press Ctrl-Space if Intellisense does not appear).



```

"devDependencies": {
    "grunt": "0.4.5"
}

```

Note: NPM uses [semantic versioning](#) to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme `<major>.<minor>.<patch>`. Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The carat `^` symbol matches the most recent major version and the tilde `~` matches the most recent minor version. See the [NPM semver version parser reference](#) as a guide to the full expressivity that SemVer provides.

3. Add more dependencies to load grunt-contrib* packages for *clean*, *jshint*, *concat*, *uglify* and *watch* as shown in the example below. The versions do not need to match the example.

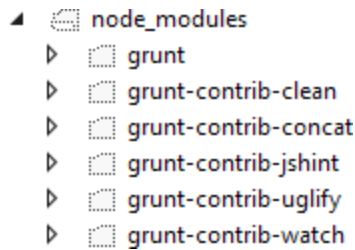
```

"devDependencies": {
    "grunt": "0.4.5",
    "grunt-contrib-clean": "0.6.0",
    "grunt-contrib-jshint": "0.11.0",
    "grunt-contrib-concat": "0.5.1",
    "grunt-contrib-uglify": "0.8.0",
    "grunt-contrib-watch": "0.6.1"
}

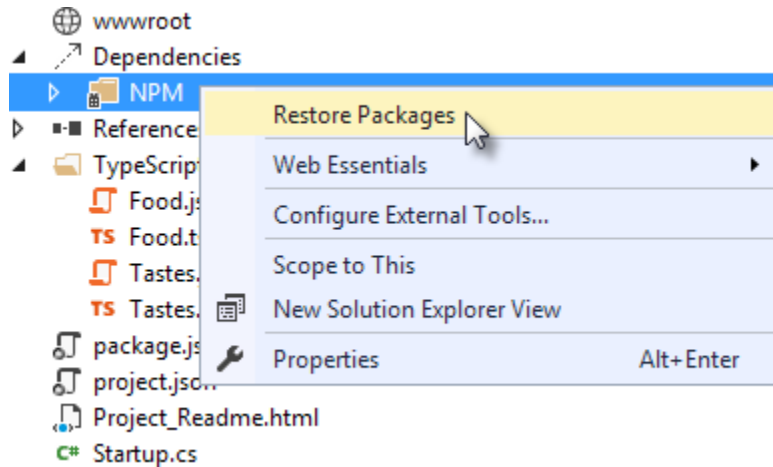
```

4. Save the `package.json` file.

The packages for each `devDependencies` item will download, along with any files that each package requires. You can find the package files in the `node_modules` directory by enabling the **Show All Files** button in the Solution Explorer.



Note: If you need to, you can manually restore dependencies in Solution Explorer by right-clicking on Dependencies\NPM and selecting the **Restore Packages** menu option.



Configuring Grunt

Grunt is configured using a manifest named `gruntfile.js` that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add > New Item**. Select the **Grunt Configuration file** option, leave the default name, `Gruntfile.js`, and click the **Add** button.

The initial code includes a module definition and the `grunt.initConfig()` method. The `initConfig()` is used to set options for each package, and the remainder of the module will load and register tasks.

```
module.exports = function (grunt) {
    grunt.initConfig({
    });
};
```

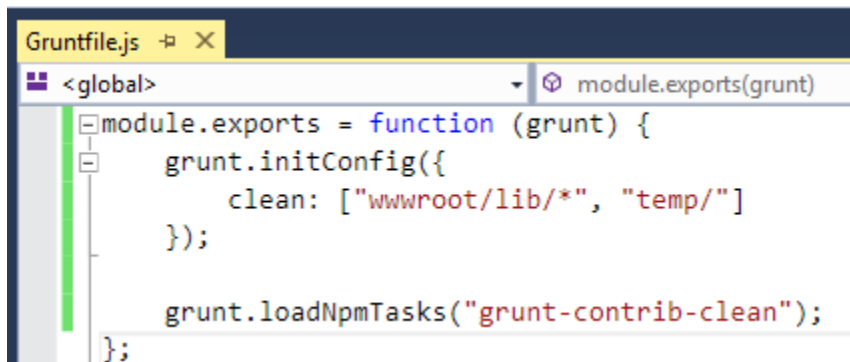
2. Inside the `initConfig()` method, add options for the `clean` task as shown in the example `Gruntfile.js` below. The `clean` task accepts an array of directory strings. This task removes files from `wwwroot/lib` and removes the entire `/temp` directory.

```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"],
    });
};
```

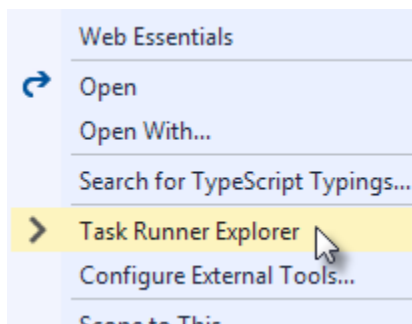
3. Below the `initConfig()` method, add a call to `grunt.loadNpmTasks()`. This will make the task runnable from Visual Studio.

```
grunt.loadNpmTasks("grunt-contrib-clean");
```

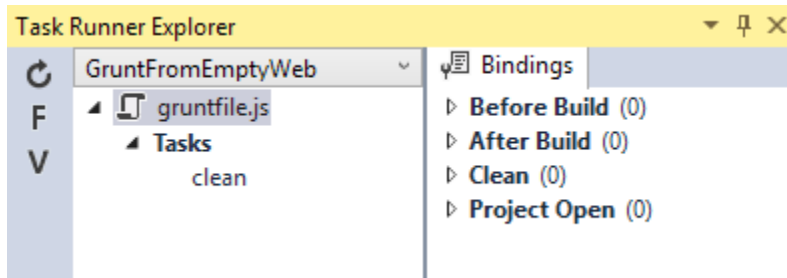
4. Save Gruntfile.js. The file should look something like the screenshot below.



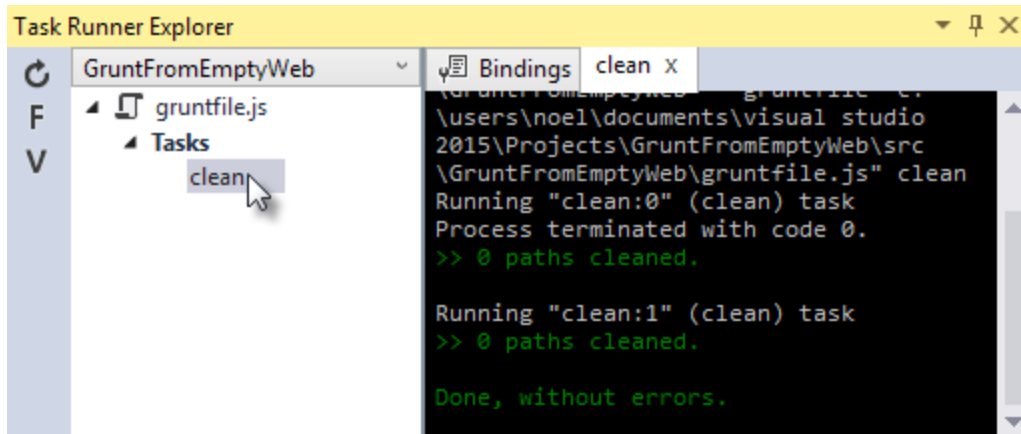
5. Right-click Gruntfile.js and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.



6. Verify that `clean` shows under **Tasks** in the Task Runner Explorer.



7. Right-click the `clean` task and select **Run** from the context menu. A command window displays progress of the task.



Note: There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

8. In the `initConfig()` method, add an entry for `concat` using the code below.

The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that is produced.

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

Note: The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using Intellisense or assign your own.

9. Add the `jshint` task using the code below.

The `jshint` code-quality utility is run against every JavaScript file found in the `temp` directory.

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

Note: The option “-W069” is an error produced by `jshint` when JavaScript uses bracket syntax to assign a property instead of dot notation, i.e. `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

10. Add the `uglify` task using the code below.

The task minifies the `combined.js` file found in the `temp` directory and creates the result file in `wwwroot/lib` following the standard naming convention `<file name>.min.js`.

```
uglify: {
  all: {
    src: ['temp/combined.js'],
    dest: 'wwwroot/lib/combined.min.js'
  }
},
```

```
    }  
  },
```

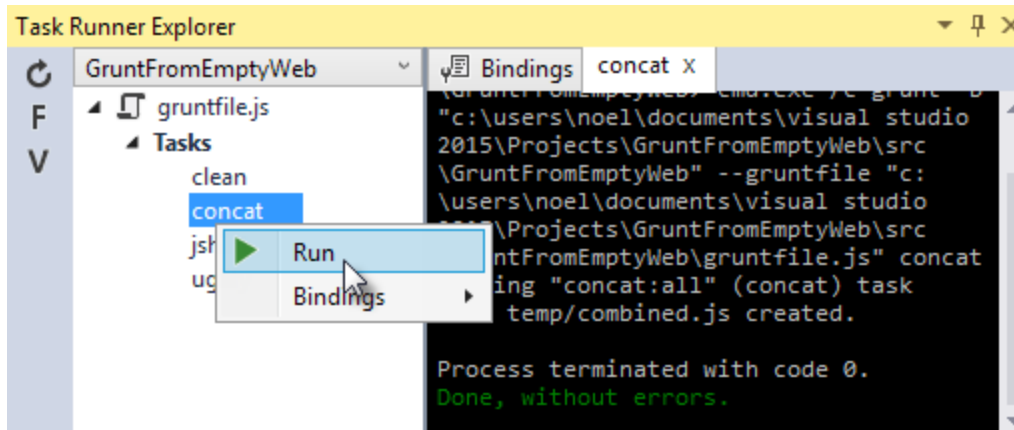
11. Under the call `grunt.loadNpmTasks()` that loads `grunt-contrib-clean`, include the same call for `jshint`, `concat` and `uglify` using the code below.

```
grunt.loadNpmTasks('grunt-contrib-jshint');  
grunt.loadNpmTasks('grunt-contrib-concat');  
grunt.loadNpmTasks('grunt-contrib-uglify');
```

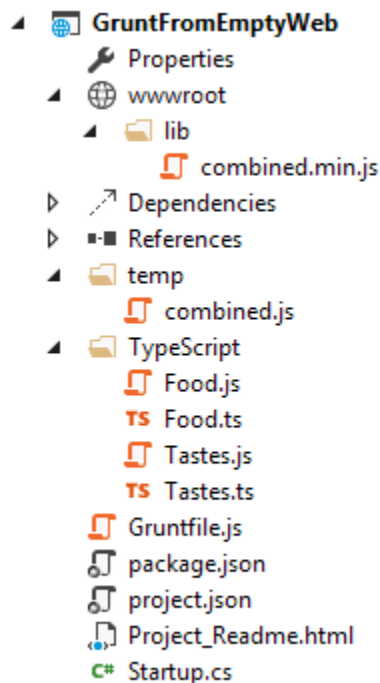
12. Save `Gruntfile.js`. The file should look something like the example below.



13. Notice that the Task Runner Explorer Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in Solution Explorer. Each task should run without errors.



The concat task creates a new combined.js file and places it into the temp directory. The jshint task simply runs and doesn't produce output. The uglify task creates a new combined.min.js file and places it into wwwrootlib. On completion, the solution should look something like the screenshot below:



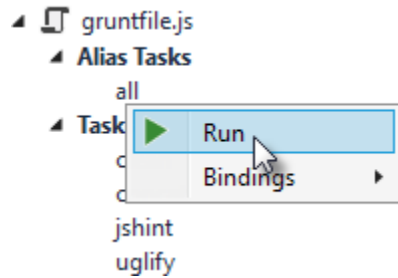
Note: For more information on the options for each package, visit <https://www.npmjs.com/> and lookup the package name in the search box on the main page. For example, you can look up the grunt-contrib-clean package to get a documentation link that explains all of its parameters.

All Together Now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the `loadNpmTasks()` calls, outside `initConfig`.

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.



Watching for changes

A `watch` task keeps an eye on files and directories. The `watch` triggers tasks automatically if it detects changes. Add the code below to `initConfig` to watch for changes to `*.js` files in the `TypeScript` directory. If a JavaScript file is changed, `watch` will run the `all` task.

```
watch: {  
  files: ["TypeScript/*.js"],  
  tasks: ["all"]  
}
```

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

Right-click the `watch` task in Task Runner Explorer and select `Run` from the context menu. The command window that shows the `watch` task running will display a `waiting...` message. Open one of the `TypeScript` files, add a space, and then save the file. This will trigger the `watch` task and trigger the other tasks to run in order. The screenshot below shows a sample run.


```

ψ Bindings watch (running) x
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" watch
Running "watch" task
Waiting...
>> File "TypeScript\Tastes.js" changed.
Running "clean:0" (clean) task
>> 1 path cleaned.

Running "clean:1" (clean) task
>> 1 path cleaned.

Running "concat:all" (concat) task
File temp/combined.js created.

Running "jshint:files" (jshint) task
>> 1 file lint free.

Running "uglify:all" (uglify) task
>> 1 file created.

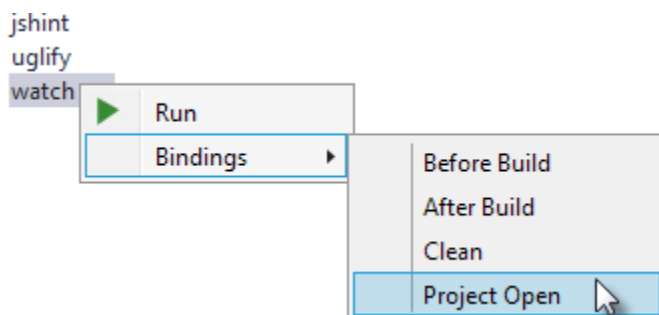
Done, without errors.
Completed in 1.236s at Fri Mar 13 2015
17:12:36 GMT-0700 (Pacific Daylight
Time) - Waiting...

```

Binding to Visual Studio Events

Unless you want to manually start your tasks every time you work in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Let's bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the `watch` task and select **Bindings > Project Open** from the context menu.



Unload and reload the project. When the project loads again, the `watch` task will start running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio 2015. Visual Studio's Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

See Also

- Using Gulp

2.8.3 Manage Client-Side Packages with Bower

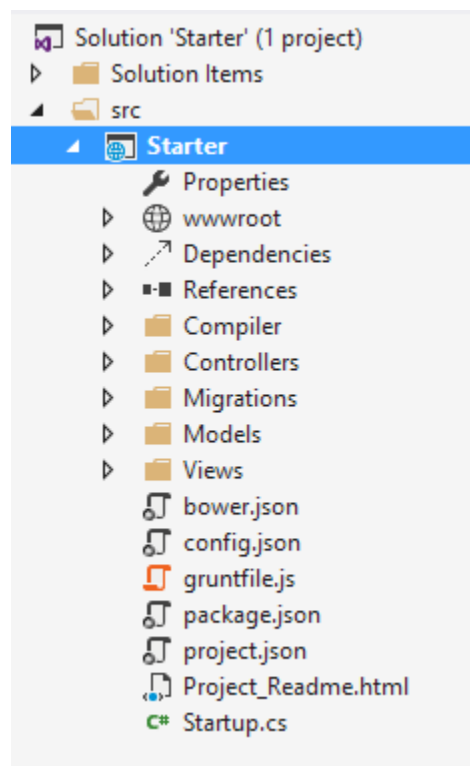
By Noel Rice

Bower is a “package manager for the web.” Bower lets you install and restore client-side packages, include JavaScript and CSS libraries. For example, with Bower you can install CSS files, fonts, client frameworks, and JavaScript libraries from external sources. Bower resolves dependencies and will automatically download and install all the packages you need. For example, if you configure Bower to load the Bootstrap package, the right jQuery package will automatically come along for the ride. (For server-side libraries like the MVC 6 framework, you will still use NuGet Package Manager.)

Note: Visual Studio developers are already familiar with NuGet, so why not use NuGet instead of Bower? Mainly because Bower already has a rich eco-system with about 18 thousand packages in play, and integrates well with the Gulp and Grunt task runners.

Getting Started with Bower

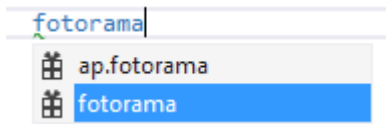
The ASP.NET 5 Starter Web MVC project pre-constructs the client build process for you. The ubiquitous jQuery and Bootstrap packages are installed, and plumbing for NPM, Grunt, and Bower is already in place. The following screenshot shows the initial project in Solution Explorer.



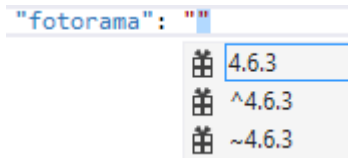
Client-side packages are listed in the bower.json file. The ASP.NET 5 Starter Web project pre-configures bower.json with jQuery, jQuery validation, Bootstrap, and [Hammer.js](#).

Let’s add support for photo albums by installing the *Fotorama* <<http://fotorama.io/>>_jQuery plugin.

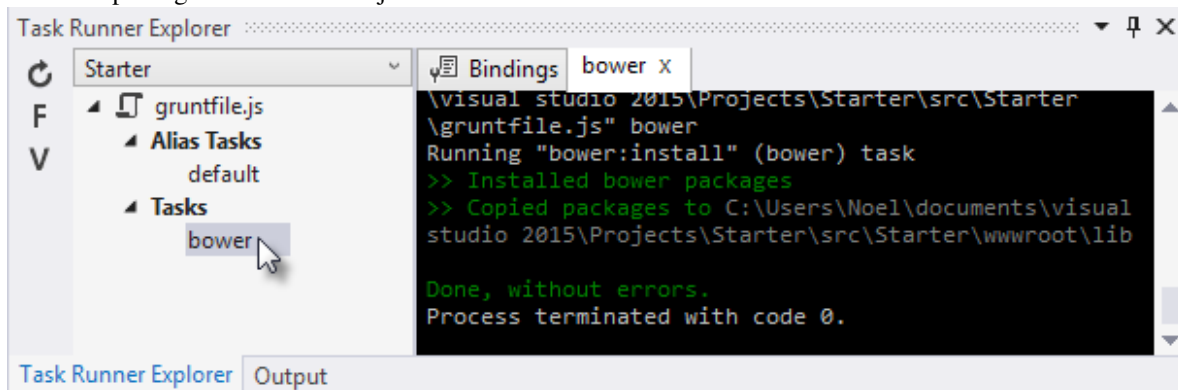
1. At the end of the `dependencies` section in `bower.json`, add a comma and type “fotorama”. Notice as you type that you get IntelliSense with a list of available packages. Select “fotorama” from the list.



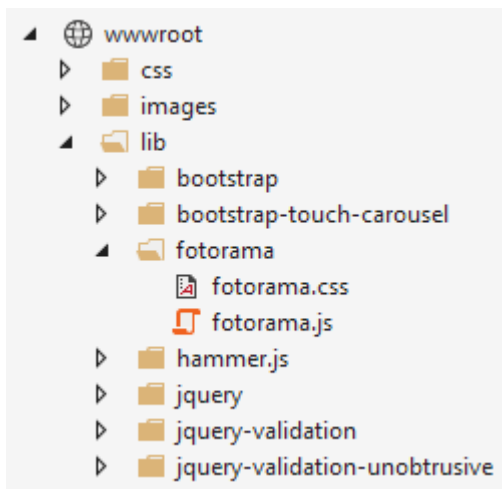
2. Add a colon and then select the latest stable version of the package from the drop down list.



3. Save the `bower.json` file.
4. Right-click `gruntfile.js` and select **Task Runner Explorer**.
5. Double-click **Tasks > bower** to run the Bower deployment task. This task runs Bower to download and install the packages listed in `bower.json`.



6. In Solution Explorer, expand the `wwwroot` node. The `lib` directory should now contain all of the packages, including the fotorama package.

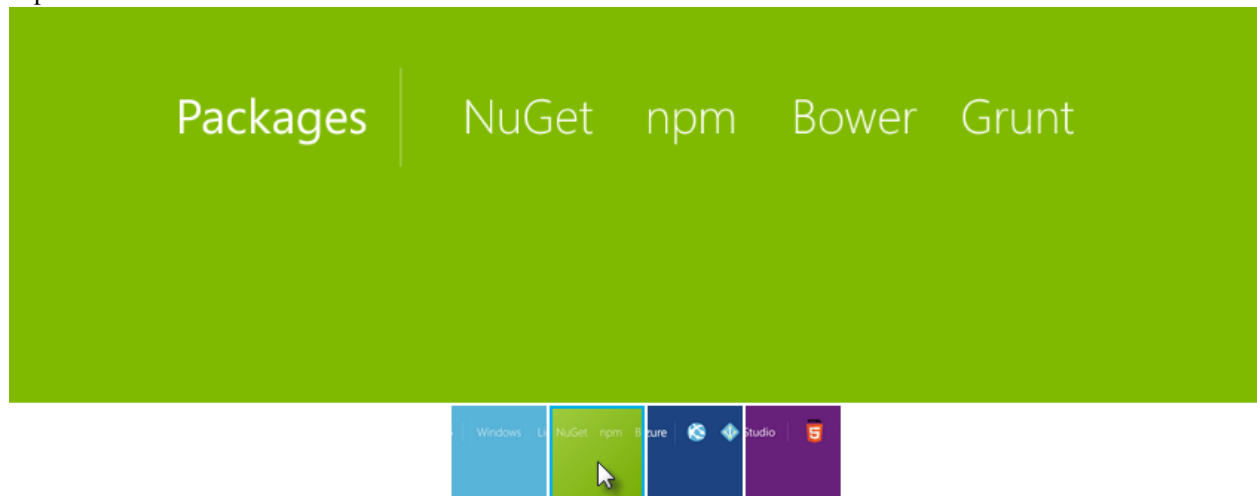


Next, let's add an HTML page to the project. In Solution Explorer, right-click `wwwroot` node and select **Add > New Item > HTML Page**. Name the page `Index.html`. Replace the contents of the file with the following:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Bower and Fotorama</title>
  <link href="lib/fotorama/fotorama.css" rel="stylesheet" />
</head>
<body>
  <div class="fotorama" data-nav="thumbs">
    
    
    
    
  </div>
  <script src="lib/jquery/jquery.js"></script>
  <script src="lib/fotorama/fotorama.js"></script>
</body>
</html>
```

This example uses images currently available inside *wwwroot\images*, but you can add any images on hand.

Press Ctrl-Shift-W to display the page in the browser. The control displays the images and allows navigation by clicking the thumbnail list below the main image. This quick test shows that Bower installed the correct packages and dependencies.



Exploring the Client Build Process

The **ASP.NET 5 Starter Web** project has everything you need for Bower already set up. This next walk-through starts with the **Empty** project template, and adds each piece manually, so you can get feel for how Bower is used in a project. See what happens to the project structure and the run-time output as each configuration change is made to the project.

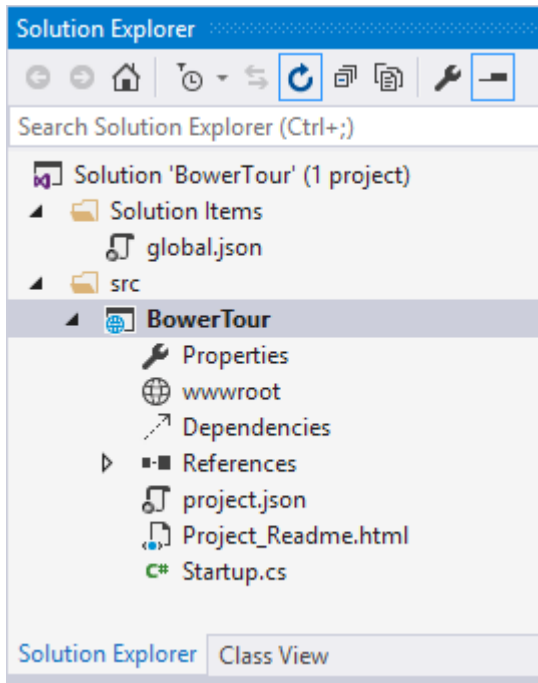
The general steps to use the client-side build process with Bower are:

- Define and download packages used in your project.
- Install the packages to the root of your web application.
- Reference packages from your web pages.

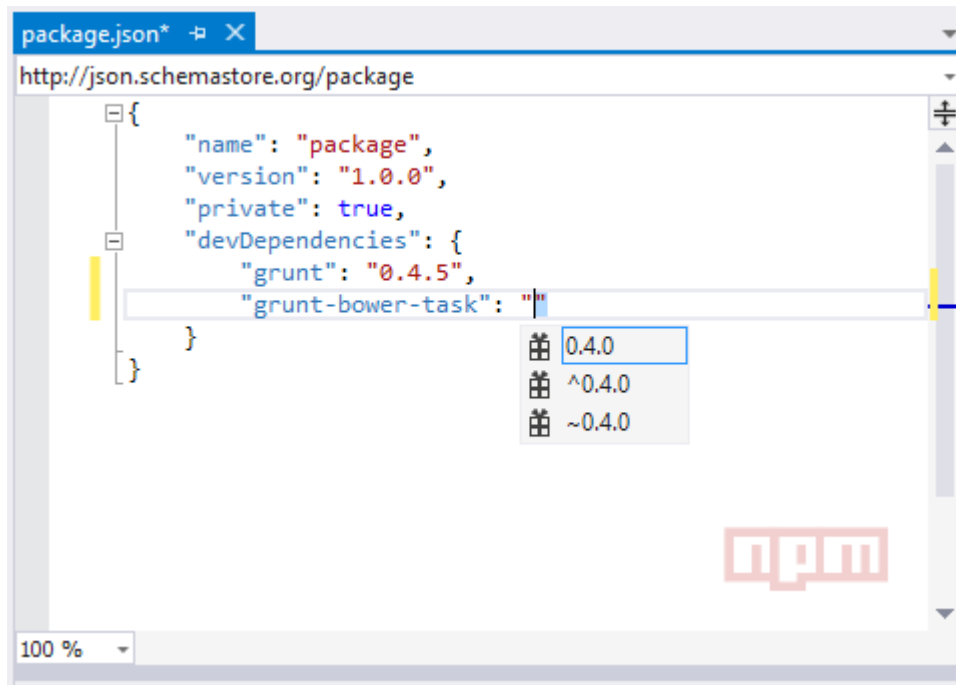
Define Packages

The first step is to define the packages your application needs and download them. This example uses Bower to load jQuery and Bootstrap. Start by configuring NPM to install design-time packages such as the Grunt task runner. Then use Grunt to run Bower so that Bower installs run-time packages jQuery and Bootstrap.

1. In Visual Studio 2015, create a new ASP.NET Web Application.
2. In the **New ASP.NET Project** dialog, select the **ASP.NET 5 Empty** template and click **OK**.
3. In Solution Explorer, the *src* directory includes a *project.json* file, and *wwwroot* and *Dependencies* nodes. The project directory will look like the screenshot below, where the *Properties* and *wwwroot* directories are empty.



4. In the Solution Explorer toolbar, enable **Show All Files**.
5. In Solution Explorer, right-click the project and add the following items:
 - NPM configuration file – *package.json*
 - Grunt configuration file – *gruntfile.js*
 - Bower configuration file – *bower.json*
6. The *package.json* file is the NPM package definition that loads all the files, include the grunt and grunt-bower-task dependencies.



7. In `gruntfile.js`, define a task that runs Bower. This is used later to manage run-time packages, like jQuery or Bootstrap, on the client. The `grunt.initConfig` task options dictate that files be copied to the `wwwroot/lib` directory. Grunt loads the `grunt-bower-task` that triggers Bower to install packages to your web application.

```
module.exports = function (grunt) {
  grunt.initConfig({
    bower: {
      install: {
        options: {
          targetDir: "wwwroot/lib",
          layout: "byComponent",
          cleanTargetDir: false
        }
      }
    },
  });

  grunt.registerTask("default", ["bower:install"]);

  grunt.loadNpmTasks("grunt-bower-task");
};
```

8. In Solution Explorer, right-click the **DependenciesNPM** node and click **Restore Packages**.
9. In Solution Explorer, view the restored packages:
 - Open the **DependenciesNPM** grunt node to see all packages that Grunt depends on.
 - Open the `node_modules` directory to view the files copied to your local machine during the package restoration.

Note: If you don't see the `node_modules` directory, make sure that **Show All Files** is enabled in the Solution Explorer toolbar.

10. Open `bower.json` and remove the `exportsOverride` section for the time being. We will replace it later after

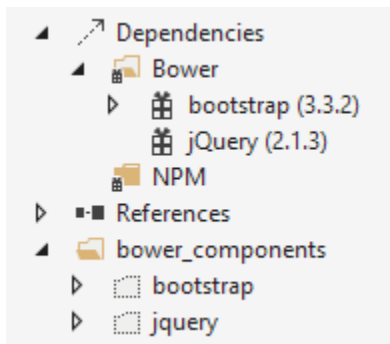
you see how Bower deploys files without this section.

11. Add jquery and bootstrap to the dependencies section. The resulting bower.json file should look like the example here. The versions will change over time, so use the latest stable build version from the drop down list.

```
{
  "name": "bower",
  "license": "Apache-2.0",
  "private": true,
  "dependencies": {
    "jquery": "2.1.3",
    "bootstrap": "3.3.2"
  }
}
```

12. Save the bower.json file.

The project should now include *bootstrap* and *jQuery* directories in two locations: *DependenciesBower* and *bower_components*.

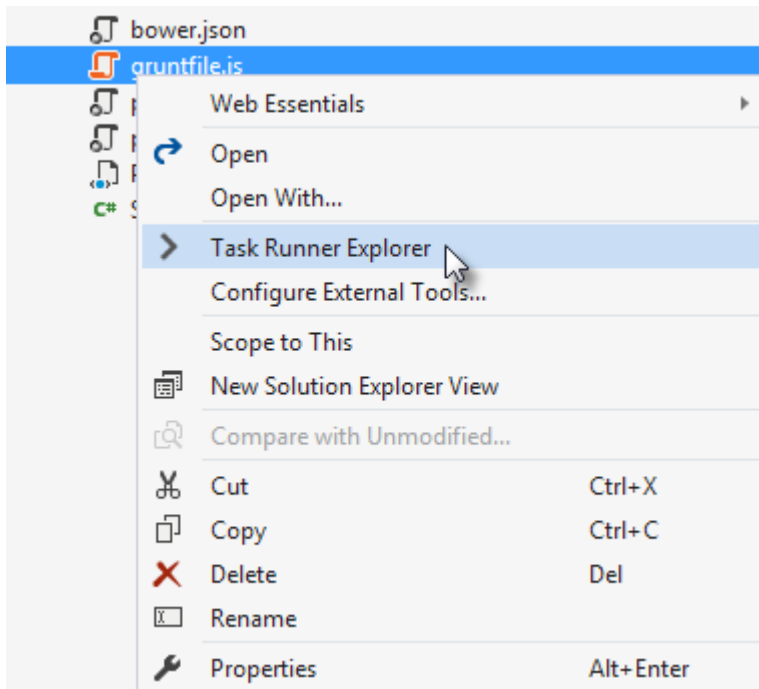


Note: If you don't see the *bower_components* directory, make sure that the Show All Files button is enabled in Solution Explorer's toolbar.

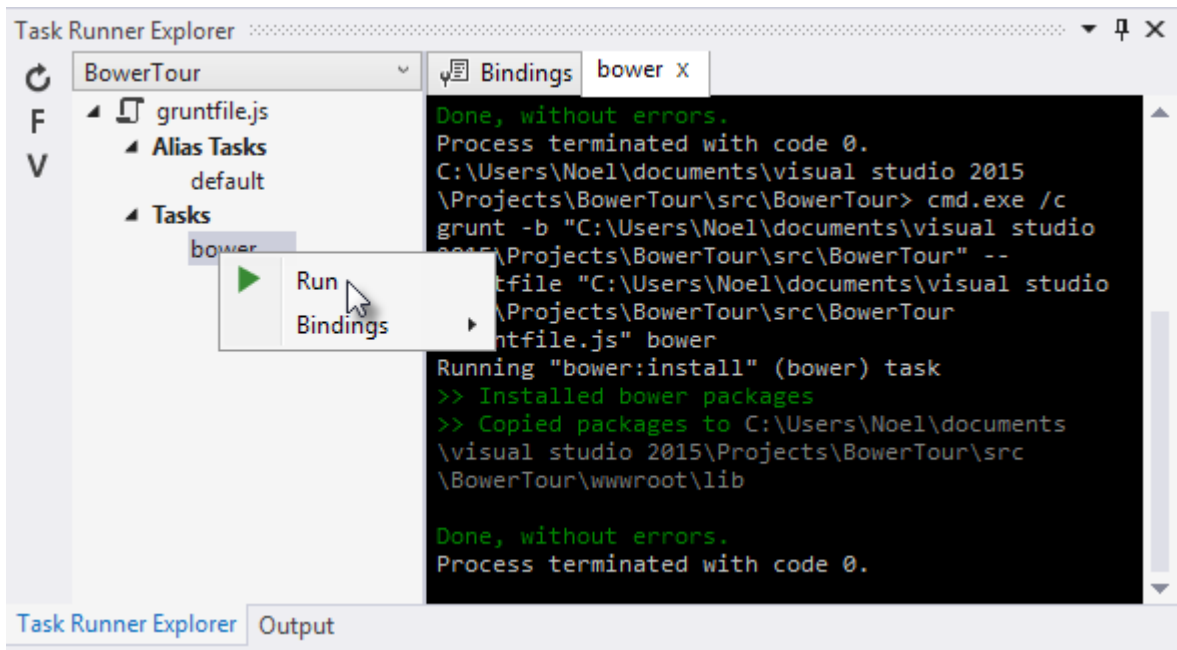
Install Packages to the Web Application

You've installed all the required files to your machine but haven't deployed them yet. In this step, you will use Bower to copy from *bower_components* to the *lib* directory under the web application root.

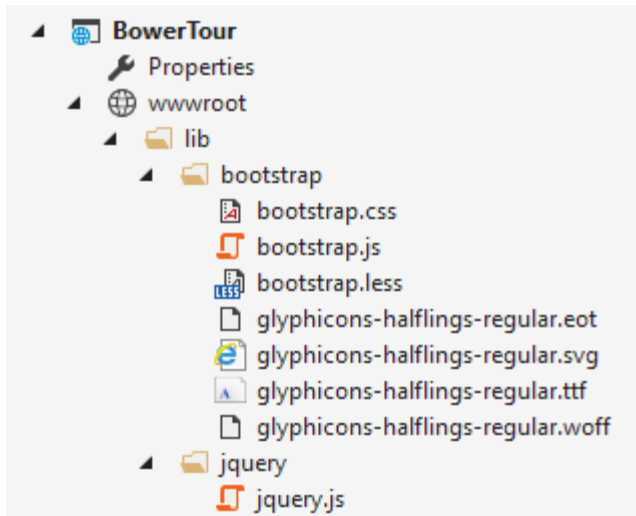
1. Right-click *gruntfile.js* and select **Task Runner Explorer**. You can also reach Task Runner Explorer through the **View > Other Windows** menu.



2. In Task Runner Explorer, right-click **Tasks > Bower** and select **Run**. This step copies the Bower packages to the root of the project (the default is *wwwroot*) under the *lib* directory.



3. In Solution Explorer, expand the *wwwrootlibbootstrap* and *wwwrootlibjquery* directories. You should see the deployed files there.

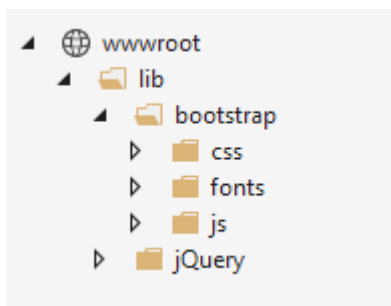


4. Now delete the `wwwroot` node. You will be able to replace it easily in the next step.
5. Open `bower.json` and add the `exportsOverride` element as shown in the listing below.

```
"exportsOverride": {
  "bootstrap": {
    "js": "dist/js/*.js",
    "css": "dist/css/*.css",
    "fonts": "dist/fonts/*.woff"
  },
  "jquery": {
    "js": "dist/jquery.{js,min.js,min.map}"
  }
}
```

The `exportsOverride` element defines source directories and target directories. For example, Bootstrap JavaScript files are copied from `bower_componentsbootstrapdistjs` to `wwwrootlibbootstrapjs`.

6. From Task Runner Explorer, run the Bower task a second time. The files are now organized under the target `css`, `fonts`, and `js` directories.



Reference Packages

Now that Bower has copied the client support packages needed by the application, you can test that an HTML page can use the deployed jQuery and Bootstrap functionality.

1. Right-click **wwwroot** and select **Add > New Item > HTML Page**.
2. Add the CSS and JavaScript references.

- In Solution Explorer, expand **wwwroot** and locate `bootstrap.css`. Drag this file into the `head` element of the HTML page.
- Drag `jquery.js` and `bootstrap.js` to the end of the `body` element.

Make sure `bootstrap.js` follows `jquery.js`, so that jQuery is loaded first.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Bower Example</title>
  <link href="lib/bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>

  <script src="lib/jquery/jquery.js"></script>
  <script src="lib/bootstrap/js/bootstrap.js"></script>
</body>
</html>
```

Use the Installed Packages

Add jQuery and Bootstrap components to the page to verify that the web application is configured correctly.

1. Inside the body tag, above the script references, add a **div** element with the Bootstrap **jumbotron** class and an anchor tag.

```
<div class="jumbotron">
  <h1>Using the jumbotron style</h1>
  <p><a class="btn btn-primary btn-lg" role="button">
    Stateful button</a></p>
</div>
```

2. Add the following code after the jQuery and Bootstrap references.

```
<script>
  $(".btn").click(function() {
    $(this).text('loading')
    .delay(1000)
    .queue(function () {
      $(this).text('reset');
      $(this).dequeue()
    });
  });
</script>
```

3. Press Ctrl-Shift-W to view the HTML page in the browser. Verify that the jumbotron styling is applied, the jQuery code responds when the button is clicked, and that the Bootstrap button changes state.

Using the jumbotron style



2.8.4 Building Beautiful, Responsive Sites with Bootstrap

By [Steve Smith](#)

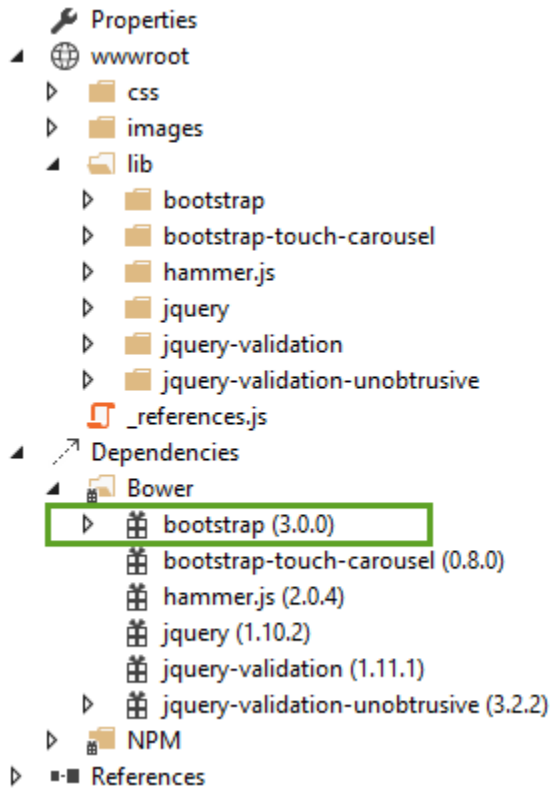
Bootstrap is currently the most popular web framework for developing responsive web applications. It offers a number of features and benefits that can improve your users' experience with your web site, whether you're a novice at front-end design and development or an expert. Bootstrap is deployed as a set of CSS and JavaScript files, and is designed to help your website or application scale efficiently from phones to tablets to desktops.

In this article:

- [Getting Started](#)
- [Basic Templates and Features](#)
- [More Themes](#)
- [Components](#)
- [JavaScript Support](#)

Getting Started

There are several ways to get started with Bootstrap. If you're starting a new web application in Visual Studio, you can choose the default starter template for ASP.NET 5, in which case Bootstrap will come pre-installed:



Adding Bootstrap to an ASP.NET 5 project is simply a matter of adding it to `bower.json` as a dependency:

```
1 {  
2   "name": "ASP.NET",  
3   "private": true,  
4   "dependencies": {  
5     "bootstrap": "3.0.0",  
6     "bootstrap-touch-carousel": "0.8.0",  
7     "hammer.js": "2.0.4",  
8     "jquery": "2.1.4",  
9     "jquery-validation": "1.11.1",  
10    "jquery-validation-unobtrusive": "3.2.2"  
11  }  
12 }
```

This is the recommended way to add Bootstrap to an ASP.NET 5 project.

You can also install bootstrap using one of several package managers, such as bower, npm, or NuGet. In each case, the process is essentially the same:

Bower

```
bower install bootstrap
```

npm

```
npm install bootstrap
```

NuGet

```
Install-Package bootstrap
```

Note: The recommended way to install client-side dependencies like Bootstrap in ASP.NET 5 is via Bower (using `bower.json`, as shown above). The use of npm/NuGet are shown to demonstrate how easily Bootstrap can be added to other kinds of web applications, including earlier versions of ASP.NET.

If you're referencing your own local versions of Bootstrap, you'll need to reference them in any pages that will use it. In production you should reference bootstrap using a CDN. In the default ASP.NET site template, the `_Layout.cshtml` file does so like this:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>@ViewData["Title"] - WebApplication1</title>
7
8     <environment names="Development">
9       <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
10      <link rel="stylesheet" href="~/lib/bootstrap-touch-carousel/dist/css/bootstrap-touch-carou
11      <link rel="stylesheet" href="~/css/site.css" />
12    </environment>
13    <environment names="Staging,Production">
14      <link rel="stylesheet" href="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/bootstrap.min
15      asp-fallback-href="~/lib/bootstrap/css/bootstrap.min.css"
16      asp-fallback-test-class="hidden" asp-fallback-test-property="visibility" asp-fallbo
17      <link rel="stylesheet" href="//ajax.aspnetcdn.com/ajax/bootstrap-touch-carousel/0.8.0/css
18      asp-fallback-href="~/lib/bootstrap-touch-carousel/css/bootstrap-touch-carousel.css"
19      asp-fallback-test-class="carousel-caption" asp-fallback-test-property="display" asp
20      <link rel="stylesheet" href="~/css/site.css" asp-file-version="true" />
21    </environment>
22  </head>
23  <body>
24    <div class="navbar navbar-inverse navbar-fixed-top">
25      <div class="container">
26        <div class="navbar-header">
27          <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="
28            <span class="icon-bar"></span>
29            <span class="icon-bar"></span>
30            <span class="icon-bar"></span>
31          </button>
32          <a asp-controller="Home" asp-action="Index" class="navbar-brand">WebApplication1
33        </div>
34        <div class="navbar-collapse collapse">
35          <ul class="nav navbar-nav">
36            <li><a asp-controller="Home" asp-action="Index">Home</a></li>
37            <li><a asp-controller="Home" asp-action="About">About</a></li>
38            <li><a asp-controller="Home" asp-action="Contact">Contact</a></li>
39          </ul>
40          @await Html.PartialAsync("_LoginPartial")
41        </div>
42      </div>
43    </div>
44    <div class="container body-content">
45      @RenderBody()
```

```

46     <hr />
47     <footer>
48         <p>&copy; 2015 - WebApplication1</p>
49     </footer>
50 </div>
51
52 <environment names="Development">
53     <script src="~/lib/jquery/dist/jquery.js"></script>
54     <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
55     <script src="~/lib/hammer.js/hammer.js"></script>
56     <script src="~/lib/bootstrap-touch-carousel/dist/js/bootstrap-touch-carousel.js"></script>
57 </environment>
58 <environment names="Staging,Production">
59     <script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
60         asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
61         asp-fallback-test="window.jQuery">
62     </script>
63     <script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/bootstrap.min.js"
64         asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
65         asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
66     </script>
67     <script src="//ajax.aspnetcdn.com/ajax/hammer.js/2.0.4/hammer.min.js"
68         asp-fallback-src="~/lib/hammer.js/hammer.js"
69         asp-fallback-test="window.Hammer">
70     </script>
71     <script src="//ajax.aspnetcdn.com/ajax/bootstrap-touch-carousel/0.8.0/js/bootstrap-touch-
72         asp-fallback-src="~/lib/bootstrap-touch-carousel/dist/js/bootstrap-touch-carouse
73         asp-fallback-test="window.Hammer && window.Hammer.Instance">
74     </script>
75     <script src="~/js/site.js" asp-file-version="true"></script>
76 </environment>
77
78 @RenderSection("scripts", required: false)
79 </body>
80 </html>

```

Note: If you’re going to be using any of Bootstrap’s jQuery plugins, you will also need to reference jQuery.

Basic Templates and Features

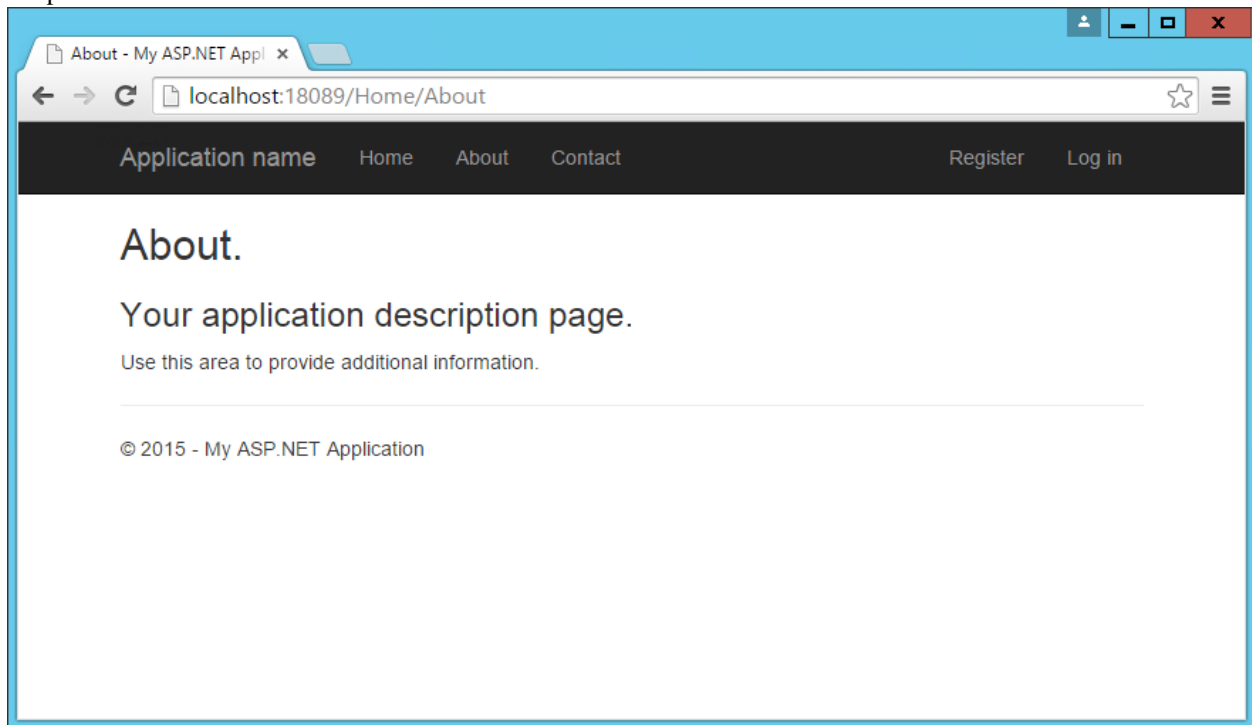
The most basic Bootstrap template looks very much like the `_Layout.cshtml` file shown above, and simply includes a basic menu for navigation and a place to render the rest of the page.

Basic Navigation

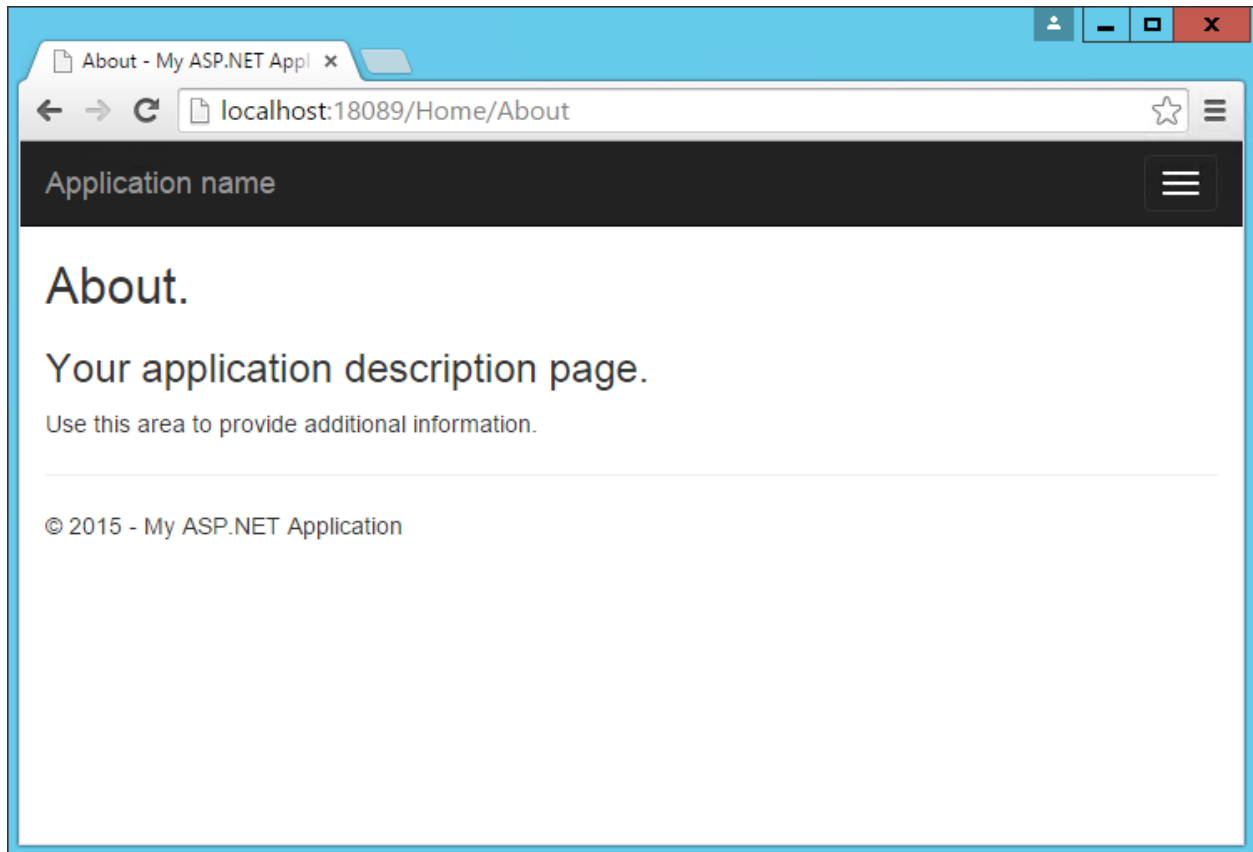
The default template uses a set of `<div>` elements to render a top navbar and the main body of the page. If you’re using HTML5, you can replace the first `<div>` tag with a `<nav>` tag to get the same effect, but with more precise semantics. Within this first `<div>` you can see there are several others. First, a `<div>` with a class of “container”, and then within that, two more `<div>` elements: “navbar-header” and “navbar-collapse”. The navbar-header div includes a button that will appear when the screen is below a certain minimum width, showing 3 horizontal lines (a so-called “hamburger icon”). The icon is rendered using pure HTML and CSS; no image is required. This is the code that displays the icon, with each of the `` tags rendering one of the white bars:

```
<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
  <span class="icon-bar"></span>
  <span class="icon-bar"></span>
  <span class="icon-bar"></span>
</button>
```

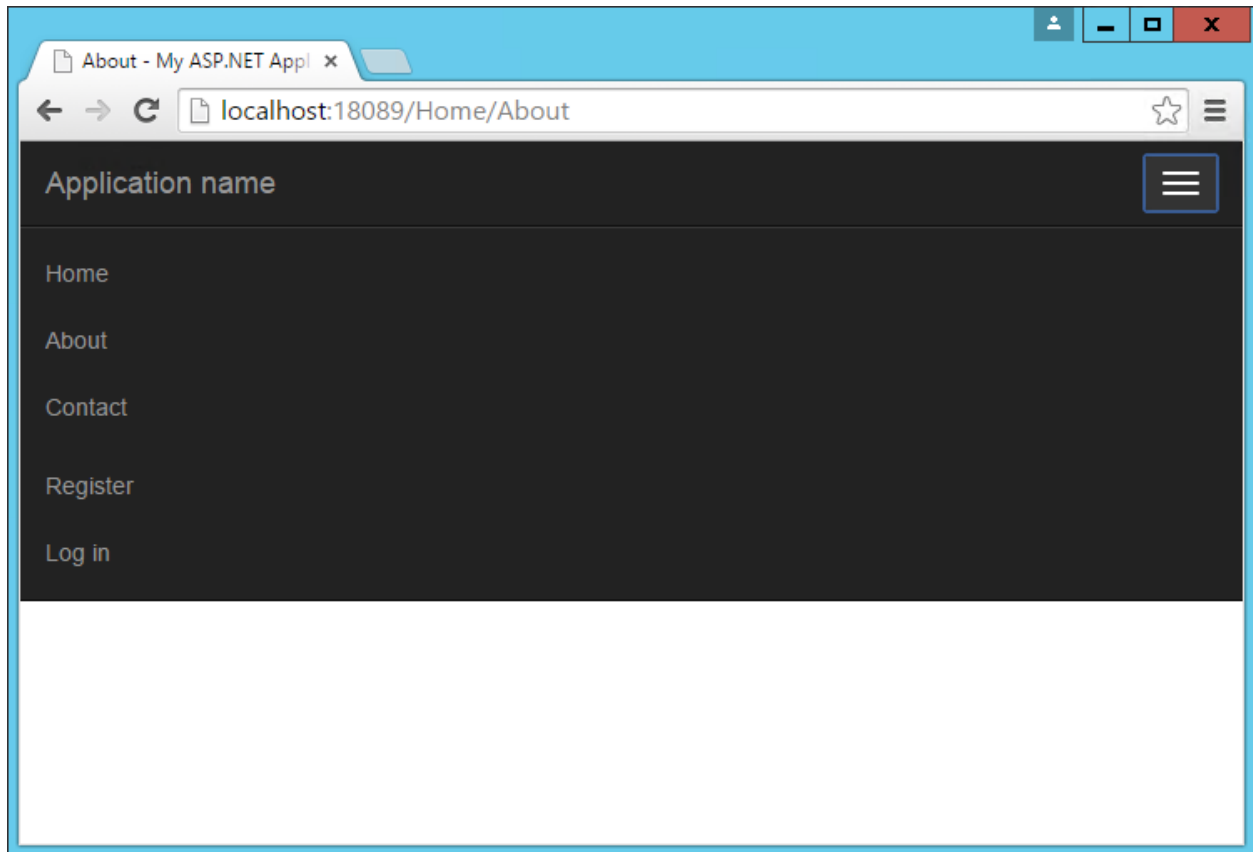
It also includes the application name, which appears in the top left. The main navigation menu is rendered by the `` element within the second div, and includes links to Home, About, and Contact. Additional links for Register and Login are added by the `_LoginPartial` line on line 29. Below the navigation, the main body of each page is rendered in another `<div>`, marked with the “container” and “body-content” classes. In the simple default `_Layout` file shown here, the contents of the page are rendered by the specific View associated with the page, and then a simple `<footer>` is added to the end of the `<div>` element. You can see how the built-in About page appears using this template:



The collapsed navbar, with “hamburger” button in the top right, appears when the window drops below a certain width:



Clicking the icon reveals the menu items in a vertical drawer that slides down from the top of the page:



Typography and Links

Bootstrap sets up the site's basic typography, colors, and link formatting in its CSS file. This CSS file includes default styles for tables, buttons, form elements, images, and more ([learn more](#)). One particularly useful feature is the grid layout system, covered next.

Grids

One of the most popular features of Bootstrap is its grid layout system. Modern web applications should avoid using the `<table>` tag for layout, instead restricting the use of this element to actual tabular data. Instead, columns and rows can be laid out using a series of `<div>` elements and the appropriate CSS classes. There are several advantages to this approach, including the ability to adjust the layout of grids to display vertically on narrow screens, such as on phones.

Bootstrap's [grid layout system](#) is based on twelve columns. This number was chosen because it can be divided evenly into 1, 2, 3, or 4 columns, and column widths can vary to within 1/12th of the vertical width of the screen. To start using the grid layout system, you should begin with a container `<div>` and then add a row `<div>`, as shown here:

```
<div class="container">
  <div class="row">

    </div>
</div>
```

Next, add additional `<div>` elements for each column, and specify the number of columns that `<div>` should occupy (out of 12) as part of a CSS class starting with "col-md-". For instance, if you want to simply have two columns of

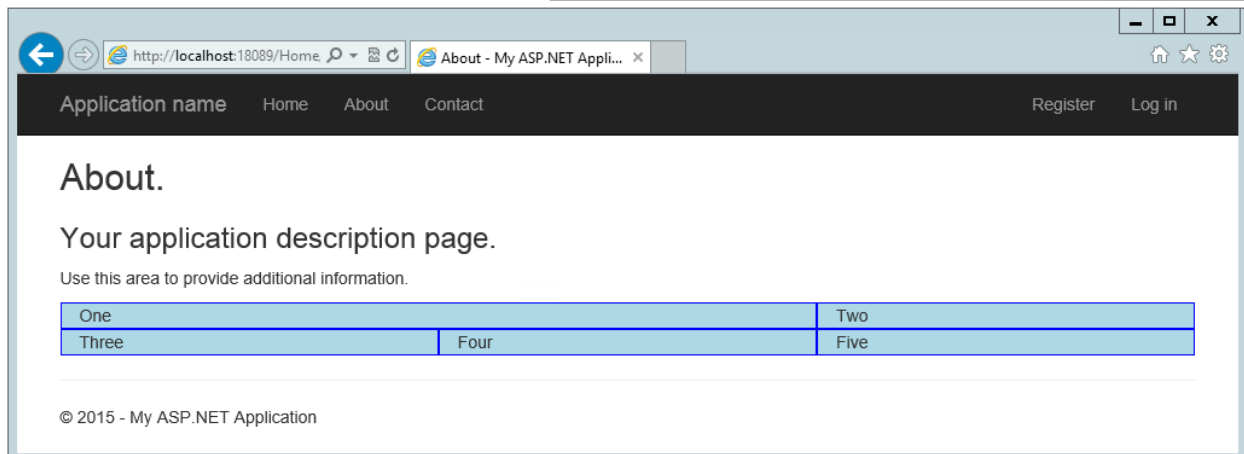
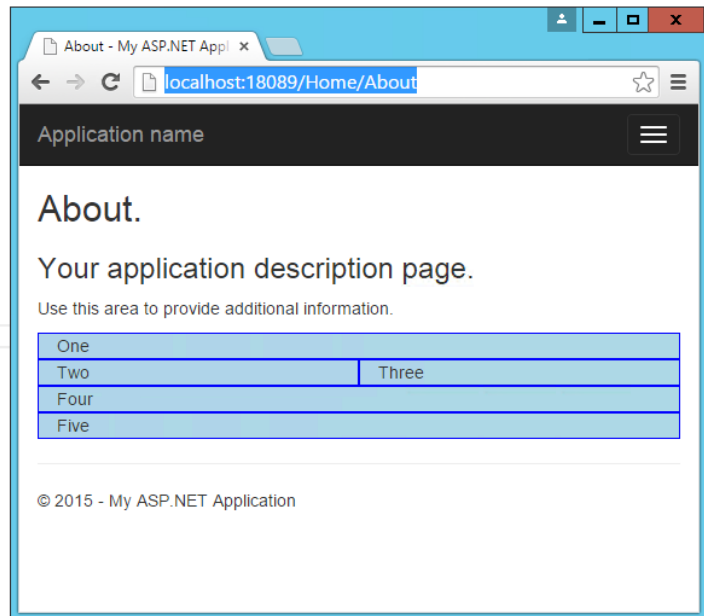
equal size, you would use a class of “col-md-6” for each one. In this case “md” is short for “medium” and refers to standard-sized desktop computer display sizes. There are four different options you can choose from, and each will be used for higher widths unless overridden (so if you want the layout to be fixed regardless of screen width, you can just specify xs classes).

CSS Class Prefix	Device Tier	Width
col-xs-	Phones	< 768px
col-sm-	Tablets	>= 768px
col-md-	Desktops	>= 992px
col-lq-	Larger Desktop Displays	>= 1200px

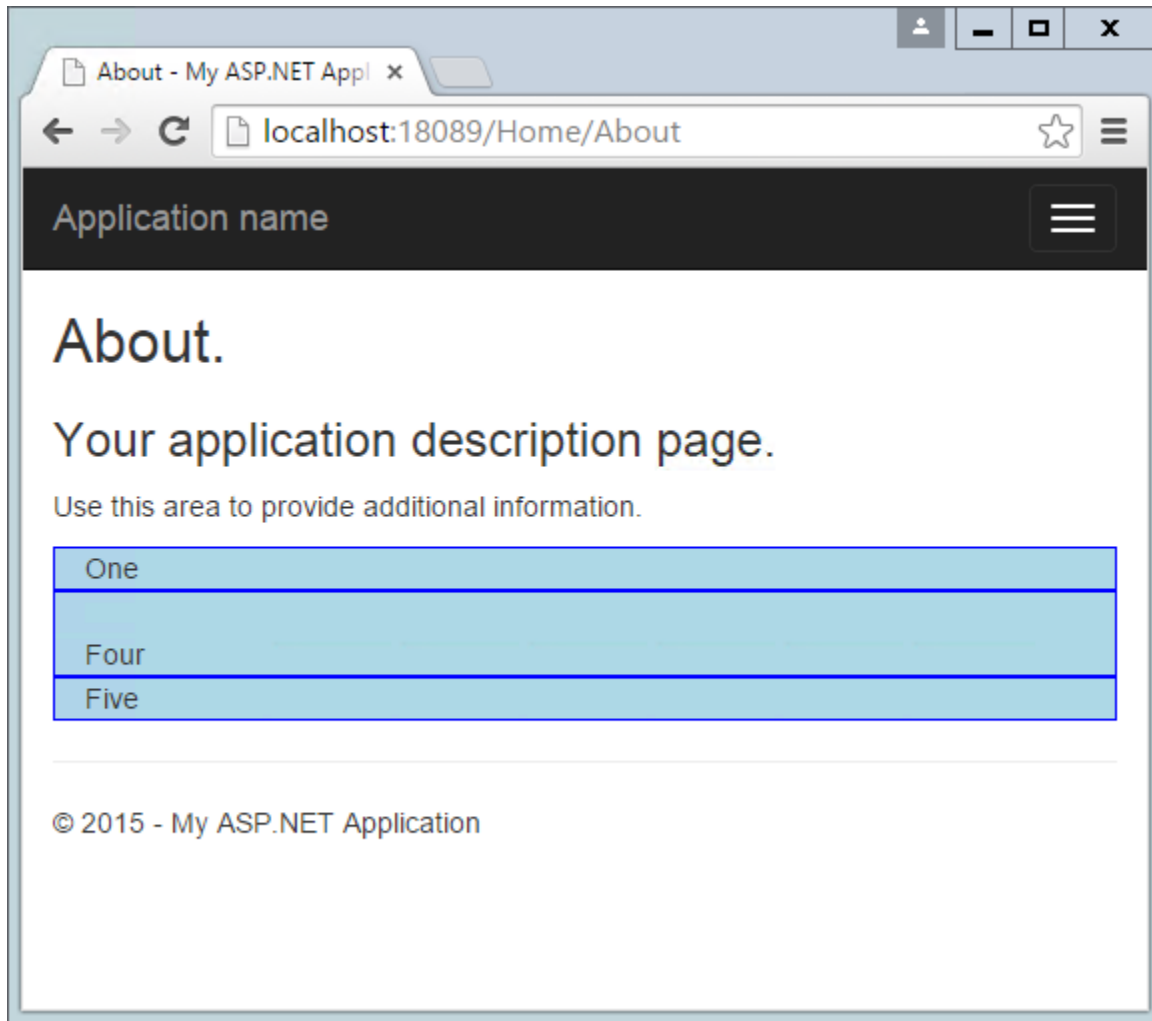
When specifying two columns both with “col-md-6” the resulting layout will be two columns at desktop resolutions, but these two columns will stack vertically when rendered on smaller devices (or a narrower browser window on a desktop), allowing users to easily view content without the need to scroll horizontally.

Bootstrap will always default to a single-column layout, so you only need to specify columns when you want more than one column. The only time you would want to explicitly specify that a `<div>` take up all 12 columns would be to override the behavior of a larger device tier. When specifying multiple device tier classes, you may need to reset the column rendering at certain points. Adding a clearfix div that is only visible within a certain viewport can achieve this, as shown here:

```
<p>Use this area to provide additional information.</p>
<style>
  [class*="col-"] {
    background-color: lightblue;
    border: 1px solid blue;
  }
</style>
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-md-8">
      One
    </div>
    <div class="col-xs-6 col-md-4">
      Two
    </div>
    <div class="col-xs-6 col-md-4">
      Three
    </div>
    <div class="clearfix visible-xs"></div>
    <div class="col-xs-12 col-md-4">
      Four
    </div>
    <div class="col-xs-12 col-md-4">
      Five
    </div>
  </div>
</div>
```



In the above example, One and Two share a row in the “md” layout, while Two and Three share a row in the “xs” layout. Without the clearfix `<div>`, Two and Three are not shown correctly in the “xs” view (note that only One, Four, and Five are shown):



In this example, only a single row `<div>` was used, and Bootstrap still mostly did the right thing with regard to the layout and stacking of the columns. Typically, you should specify a row `<div>` for each horizontal row your layout requires, and of course you can nest Bootstrap grids within one another. When you do, each nested grid will occupy 100% of the width of the element in which it is placed, which can then be subdivided using column classes.

Jumbotron

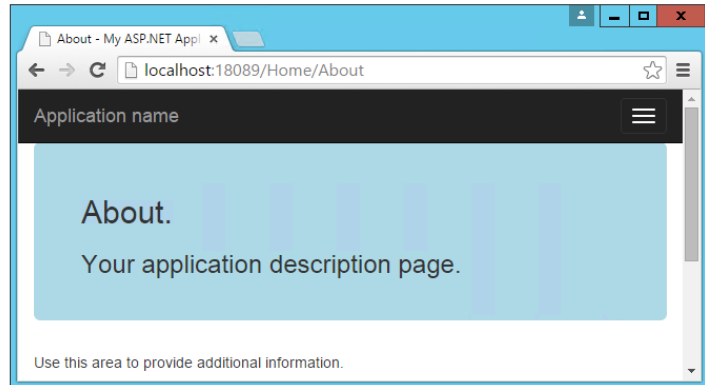
If you’ve used the default ASP.NET MVC templates in Visual Studio 2012 or 2013, you’ve probably seen the Jumbotron in action. It refers to a large full-width section of a page that can be used to display a large background image, a call to action, a rotator, or similar elements. To add a jumbotron to a page, simply add a `<div>` and give it a class of “jumbotron”, then place a container `<div>` inside and add your content. We can easily adjust the standard About page to use a jumbotron for the main headings it displays:

```

<style>
.jumbotron {
    background-color: lightblue;
}
</style>

<div class="jumbotron">
    <div class="container">
        <h2>@ViewBag.Title.</h2>
        <h3>@ViewBag.Message</h3>
    </div>
</div>
<p>Use this area to provide additional information.</p>

```



Buttons

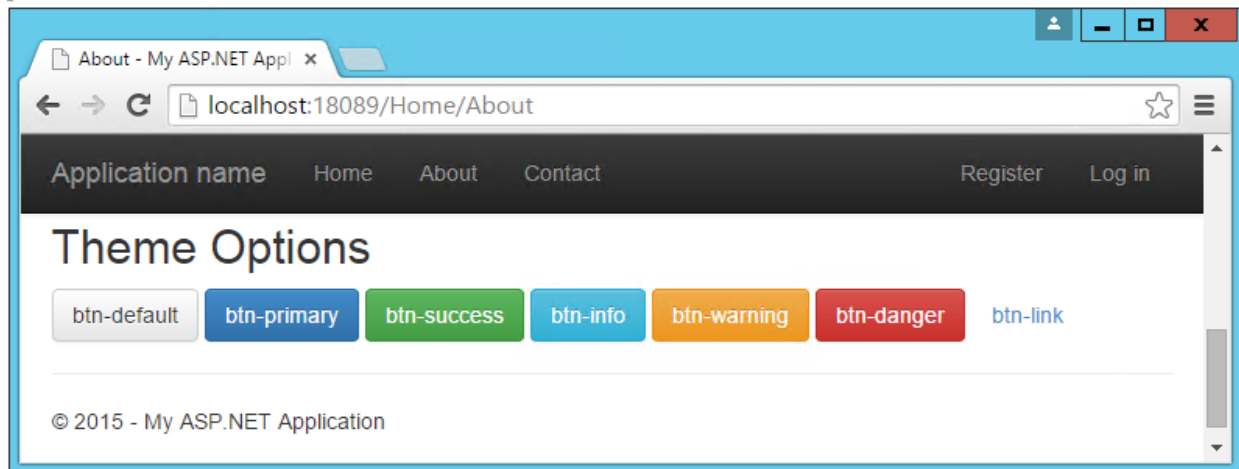
The default button classes and their colors are shown in the figure below.

```

<h2>Theme Options</h2>

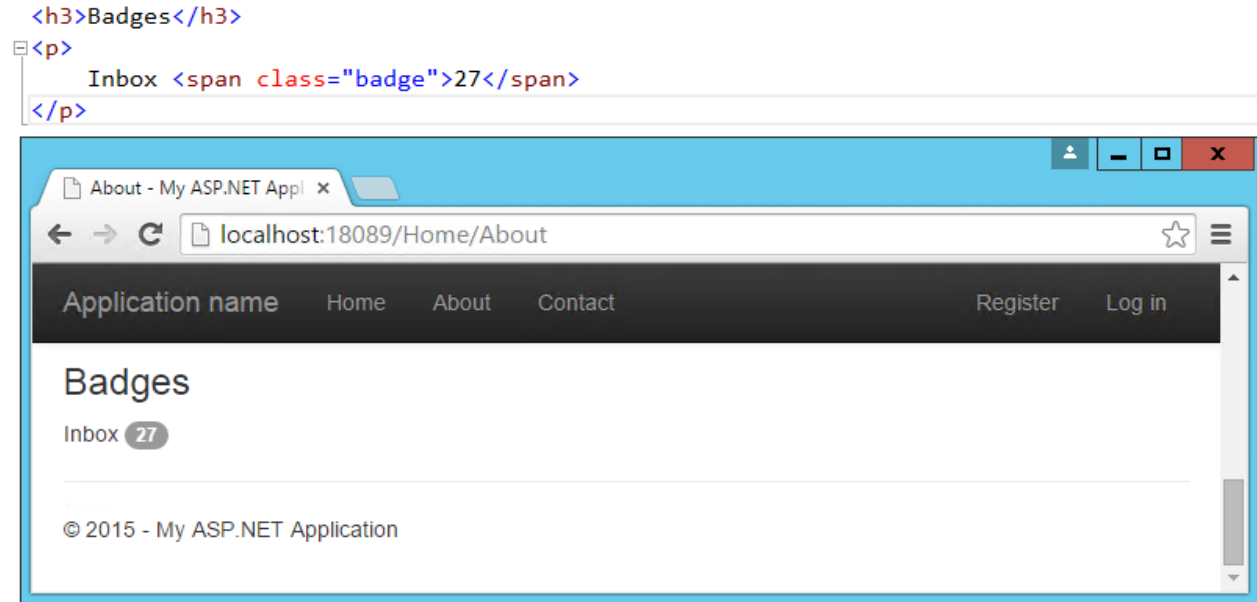
<p>
    <button type="button" class="btn btn-default">btn-default</button>
    <button type="button" class="btn btn-primary">btn-primary</button>
    <button type="button" class="btn btn-success">btn-success</button>
    <button type="button" class="btn btn-info">btn-info</button>
    <button type="button" class="btn btn-warning">btn-warning</button>
    <button type="button" class="btn btn-danger">btn-danger</button>
    <button type="button" class="btn btn-link">btn-link</button>
</p>

```



Badges

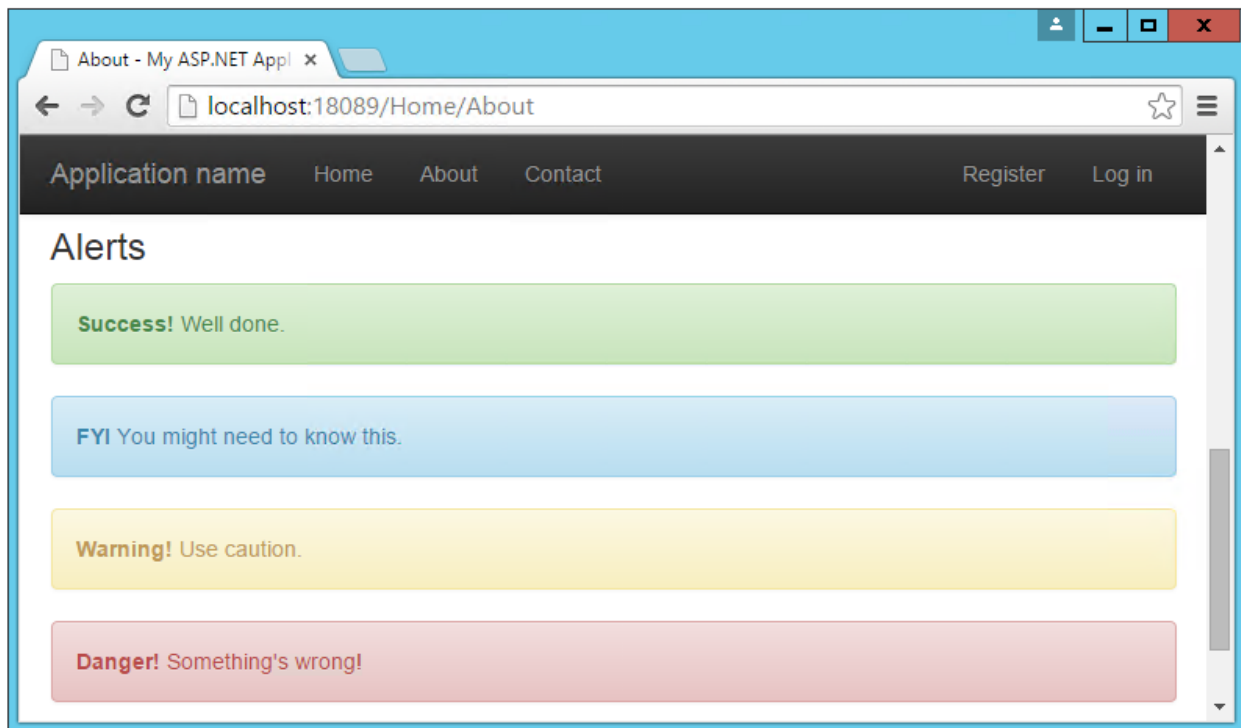
Badges refer to small, usually numeric callouts next to a navigation item. They can indicate a number of messages or notifications waiting, or the presence of updates. Specifying such badges is as simple as adding a `` containing the text, with a class of "badge":



Alerts

You may need to display some kind of notification, alert, or error message to your application's users. That's where the standard alert classes come in. There are four different severity levels, with associated color schemes:

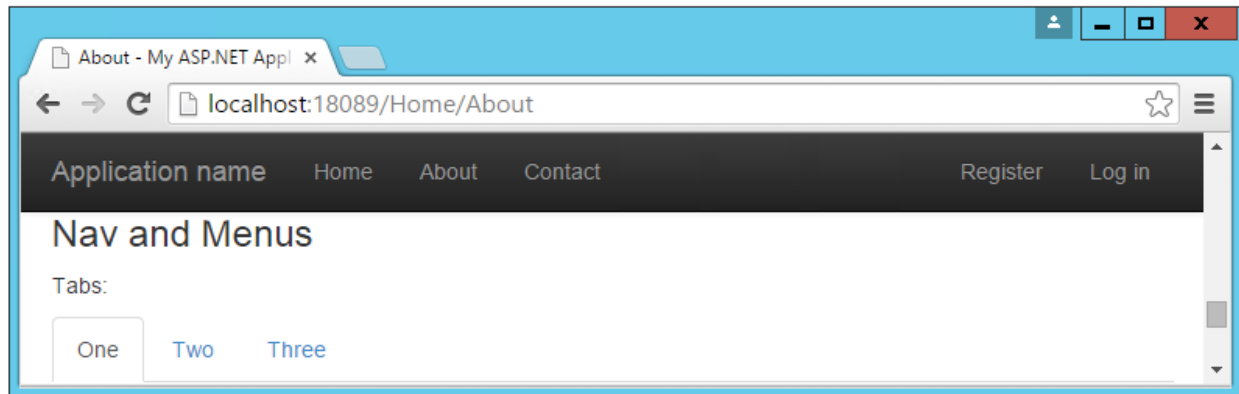
```
<h3>Alerts</h3>
<div class="alert alert-success">
  <strong>Success!</strong> Well done.
</div>
<div class="alert alert-info">
  <strong>FYI</strong> You might need to know this.
</div>
<div class="alert alert-warning">
  <strong>Warning!</strong> Use caution.
</div>
<div class="alert alert-danger">
  <strong>Danger!</strong> Something's wrong!
</div>
```



Navbars and Menus

Our layout already includes a standard navbar, but the Bootstrap theme supports additional styling options. We can also easily opt to display the navbar vertically rather than horizontally if that's preferred, as well as adding sub-navigation items in flyout menus. Simple navigation menus, like tab strips, are built on top of `` elements. These can be created very simply by just providing them with the CSS classes "nav" and "nav-tabs":

```
<h3>Nav and Menus</h3>
<p>Tabs:</p>
<ul class="nav nav-tabs">
  <li class="active"><a href="#">One</a></li>
  <li><a href="#">Two</a></li>
  <li><a href="#">Three</a></li>
</ul>
```

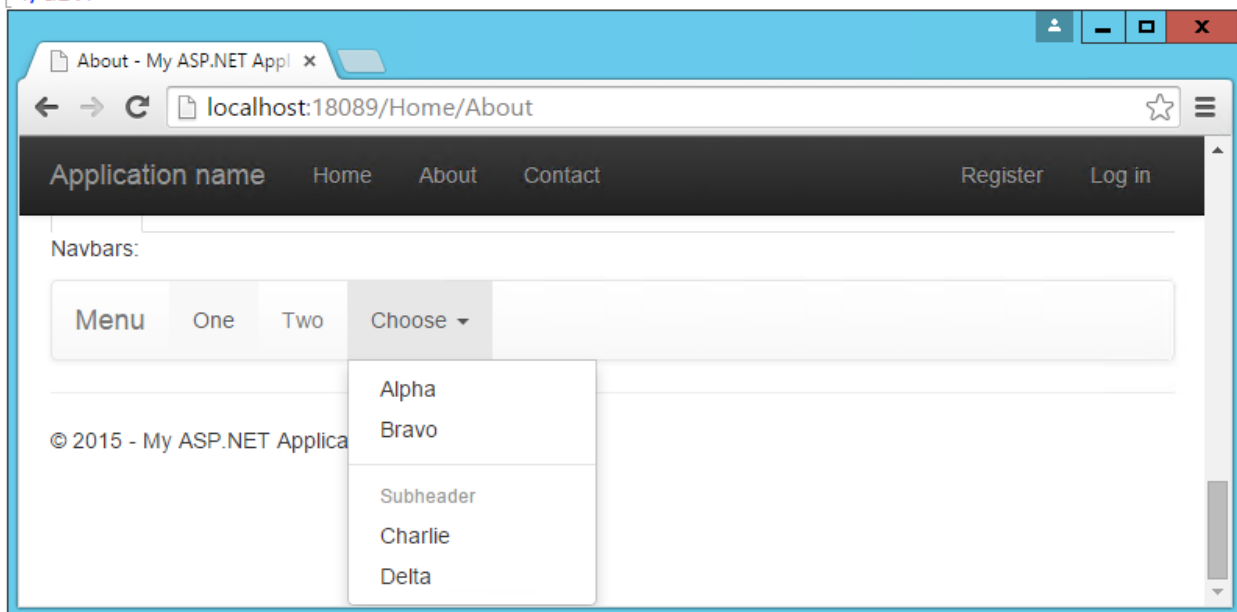


Navbars are built similarly, but are a bit more complex. They start with a `<nav>` or `<div>` with a class of “navbar”, within which a container div holds the rest of the elements. Our page includes a navbar in its header already – the one shown below simply expands on this, adding support for a dropdown menu:

```

<p>Navbars:</p>
<div class="navbar navbar-default">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse"
        data-target=".navbar-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Menu</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li class="active"><a href="#">One</a></li>
        <li><a href="#two">Two</a></li>
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown"
            role="button" aria-expanded="false">Choose <span class="caret"></span></a>
          <ul class="dropdown-menu" role="menu">
            <li><a href="#">Alpha</a></li>
            <li><a href="#">Bravo</a></li>
            <li class="divider"></li>
            <li class="dropdown-header">Subheader</li>
            <li><a href="#">Charlie</a></li>
            <li><a href="#">Delta</a></li>
          </ul>
        </li>
      </ul>
    </div>
  </div>
</div>

```



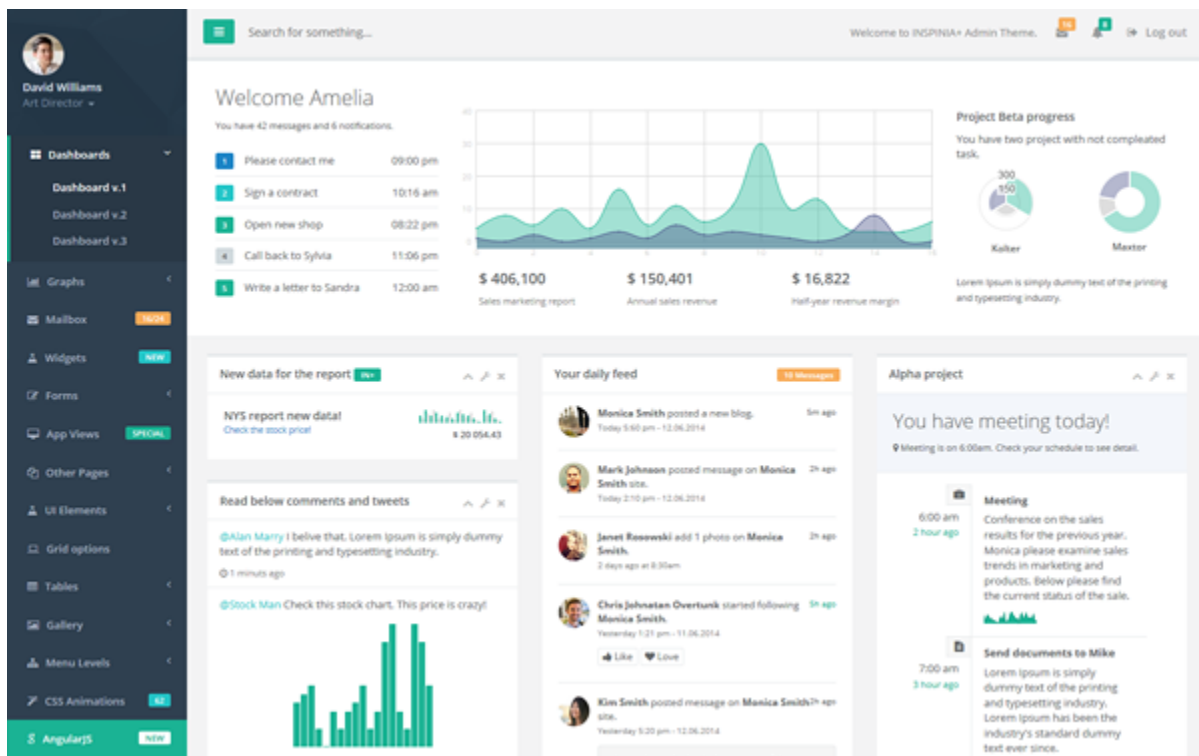
Additional Elements

The default theme can also be used to present HTML tables in a nicely formatted style, including support for striped views. There are labels with styles that are similar to those of the buttons. You can create custom Dropdown menus that support additional styling options beyond the standard HTML `<select>` element, along with Navbars like the one our default starter site is already using. If you need a progress bar, there are several styles to choose from, as well as List Groups and panels that include a title and content. Explore additional options within the standard Bootstrap Theme here:

<http://getbootstrap.com/examples/theme/>

More Themes

You can extend the standard Bootstrap Theme by overriding some or all of its CSS, adjusting the colors and styles to suit your own application's needs. If you'd like to start from a ready-made theme, there are several theme galleries available online that specialize in Bootstrap Themes, such as WrapBootstrap.com (which has a variety of commercial themes) and Bootswatch.com (which offers free themes). Some of the paid templates available provide a great deal of functionality on top of the basic Bootstrap theme, such as rich support for administrative menus, and dashboards with rich charts and gauges. An example of a popular paid template is Inspinia, currently for sale for \$18, which includes an ASP.NET MVC5 template in addition to AngularJS and static HTML versions. A sample screenshot is shown below.











































If you're interested in building your own dashboard, you may wish to start from the free example available here: <http://getbootstrap.com/examples/dashboard/>.

Components

In addition to those elements already discussed, Bootstrap includes support for a variety of built-in UI components.

Glyphicons

Bootstrap includes icon sets from Glyphicons (<http://glyphicons.com>), with over 200 icons freely available for use within your Bootstrap-enabled web application. Here's just a small sample:

glyphicon indent-right	glyphicon facetime-video	glyphicon picture	glyphicon map-marker	glyphicon adjust	glyphicon share	glyphicon share	glyphicon share
							
glyphicon glyphicon-check	glyphicon glyphicon-move	glyphicon glyphicon-step-backward	glyphicon glyphicon-fast-backward	glyphicon glyphicon-backward	glyphicon glyphicon-play	glyphicon glyphicon-pause	glyphicon glyphicon-stop
							
glyphicon glyphicon-forward	glyphicon glyphicon-fast-forward	glyphicon glyphicon-step-forward	glyphicon glyphicon-eject	glyphicon glyphicon-chevron-left	glyphicon glyphicon-chevron-right	glyphicon glyphicon-plus-sign	glyphicon glyphicon-minus-sign
							
glyphicon glyphicon-remove-sign	glyphicon glyphicon-ok-sign	glyphicon glyphicon-question-sign	glyphicon glyphicon-info-sign	glyphicon glyphicon-screenshot	glyphicon glyphicon-remove-circle	glyphicon glyphicon-ok-circle	glyphicon glyphicon-ban-circle
							
glyphicon glyphicon-arrow-left	glyphicon glyphicon-arrow-right	glyphicon glyphicon-arrow-up	glyphicon glyphicon-arrow-down	glyphicon glyphicon-share-alt	glyphicon glyphicon-resize-full	glyphicon glyphicon-resize-small	glyphicon glyphicon-exclamation-sign
							
glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon

Input Groups

Input groups allow bundling of additional text or buttons with an input element, providing the user with a more intuitive experience:

Breadcrumbs

Breadcrumbs are a common UI component used to show a user their recent history or depth within a site's navigation hierarchy. Add them easily by applying the “breadcrumb” class to any `` list element. Include built-in support for pagination by using the “pagination” class on a `` element within a `<nav>`. Add responsive embedded slideshows and video by using `<iframe>`, `<embed>`, `<video>`, or `<object>` elements, which Bootstrap will style automatically. Specify a particular aspect ratio by using specific classes like “embed-responsive-16by9”.

JavaScript Support

Bootstrap's JavaScript library includes API support for the included components, allowing you to control their behavior programmatically within your application. In addition, bootstrap.js includes over a dozen custom jQuery plugins, providing additional features like transitions, modal dialogs, scroll detection (updating styles based on where the user

has scrolled in the document), collapse behavior, carousels, and affixing menus to the window so they do not scroll off the screen. There's not sufficient room to cover all of the JavaScript add-ons built into Bootstrap – to learn more please visit <http://getbootstrap.com/javascript/>.

Summary

Bootstrap provides a web framework that can be used to quickly and productively lay out and style a wide variety of websites and applications. Its basic typography and styles provide a pleasant look and feel that can easily be manipulated through custom theme support, which can be hand-crafted or purchased commercially. It supports a host of web components that in the past would have required expensive third-party controls to accomplish, while supporting modern and open web standards.

2.8.5 Knockout.js MVVM Framework

By [Steve Smith](#)

Knockout is a popular JavaScript library that simplifies the creation of complex data-based user interfaces. It can be used alone or with other libraries, such as jQuery. Its primary purpose is to bind UI elements to an underlying data model defined as a JavaScript object, such that when changes are made to the UI, the model is updated, and vice versa. Knockout facilitates the use of a Model-View-ViewModel (MVVM) pattern in a web application's client-side behavior. The two main concepts one must learn when working with Knockout's MVVM implementation are Observables and Bindings.

In this article:

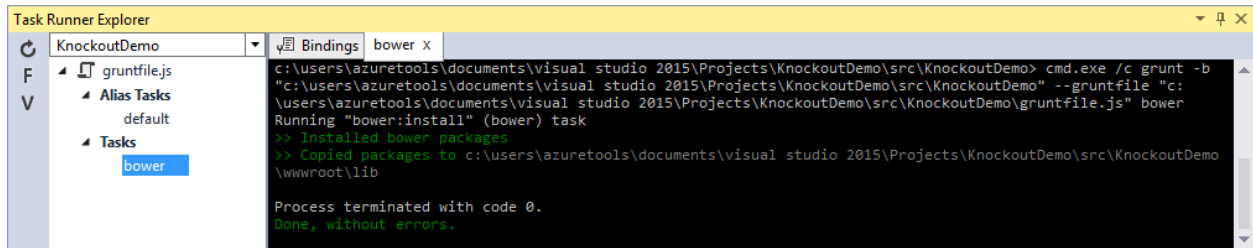
- *Getting Started with Knockout in ASP.NET 5*
- *Observables, ViewModels, and Simple Binding*
- *Control Flow*
- *Templates*
- *Components*
- *Communicating with APIs*

Getting Started with Knockout in ASP.NET 5

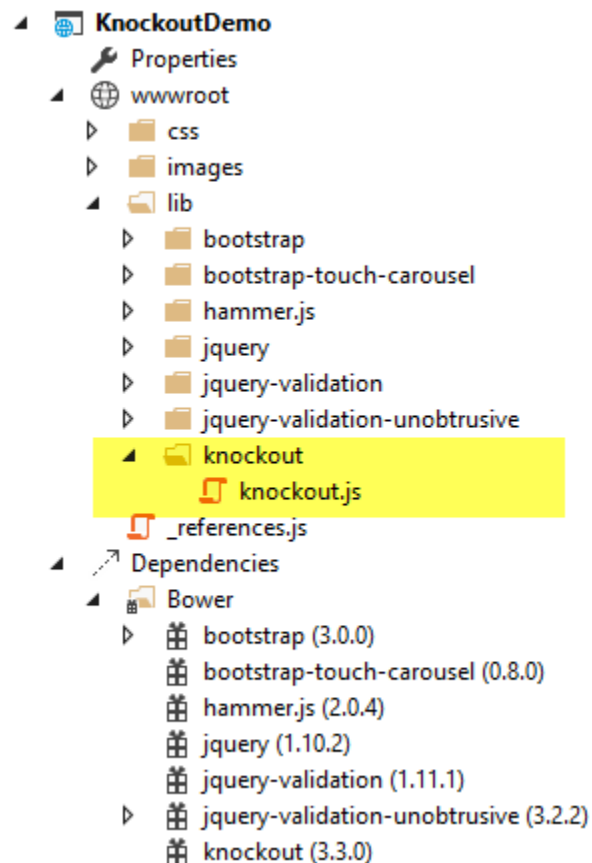
Knockout is deployed as a single JavaScript file, so installing and using it is very straightforward. In Visual Studio 2015, you can simply add knockout as a dependency and Visual Studio will use bower to retrieve it. Assuming you already have bower and gulp configured (the ASP.NET 5 Starter Template comes with them already set up), open bower.json in your ASP.NET 5 project, and add the knockout dependency as shown here:

```
{
  "name": "KnockoutDemo",
  "private": true,
  "dependencies": {
    "knockout" : "^3.3.0"
  },
  "exportsOverride": {
  }
}
```

With this in place, you can then manually run bower by opening the Task Runner Explorer (under *View* → *Other Windows* → *Task Runner Explorer*) and then under Tasks, right-click on bower and select Run. The result should appear similar to this:



Now if you look in your project's `wwwroot` folder, you should see knockout installed under the lib folder.



It's recommended that in your production environment you reference knockout via a Content Delivery Network, or CDN, as this increases the likelihood that your users will already have a cached copy of the file and thus will not need to download it at all. Knockout is available on several CDNs, including the Microsoft Ajax CDN, here:

<http://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js>

To include Knockout on a page that will use it, simply add a `<script>` element referencing the file from wherever you will be hosting it (with your application, or via a CDN):

```
<script type="text/javascript" src="knockout-3.3.0.js"></script>
```

Observables, ViewModels, and Simple Binding

You may already be familiar with using JavaScript to manipulate elements on a web page, either via direct access to the DOM or using a library like jQuery. Typically this kind of behavior is achieved by writing code to directly set element values in response to certain user actions. With Knockout, a declarative approach is taken instead, through which

elements on the page are bound to properties on an object. Instead of writing code to manipulate DOM elements, user actions simply interact with the ViewModel object, and Knockout takes care of ensuring the page elements are synchronized.

As a simple example, consider the page list below. It includes a `` element with a `data-bind` attribute indicating that the text content should be bound to `authorName`. Next, in a JavaScript block a variable `viewModel` is defined with a single property, `authorName`, set to some value. Finally, a call to `ko.applyBindings` is made, passing in this `viewModel` variable.

```

1      <html>
2          <head>
3              <script type="text/javascript" src="lib/knockout/knockout.js"></script>
4          </head>
5          <body>
6              <h1>Some Article</h1>
7              <p>
8                  By <span data-bind="text: authorName"></span>
9              </p>
10             <script type="text/javascript">
11                 var viewModel = {
12                     authorName: 'Steve Smith'
13                 };
14                 ko.applyBindings(viewModel);
15             </script>
16         </body>
17     </html>

```

When viewed in the browser, the content of the `` element is replaced with the value in the `viewModel` variable:



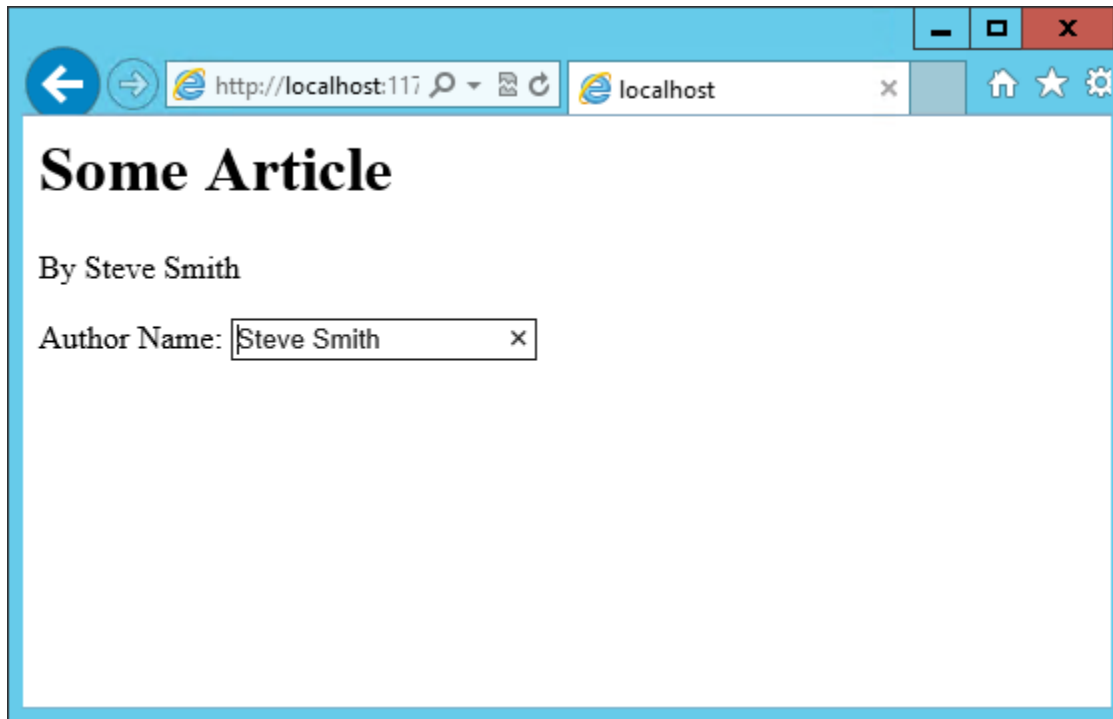
We now have simple one-way binding working. Notice that nowhere in the code did we write JavaScript to assign a value to the `span`'s contents. If we want to manipulate the `ViewModel`, we can take this a step further and add an HTML input textbox, and bind to its value, like so:

```

<p>
    Author Name: <input type="text" data-bind="value: authorName" />
</p>

```

Reloading the page, we see that this value is indeed bound to the input box:



However, if we change the value in the textbox, the corresponding value in the `` element doesn't change. Why not?

The issue is that nothing notified the `` that it needed to be updated. Simply updating the `ViewModel` isn't by itself sufficient, unless the `ViewModel`'s properties are wrapped in a special type. We need to use **observables** in the `ViewModel` for any properties that need to have changes automatically updated as they occur. By changing the `ViewModel` to use `ko.observable("value")` instead of just `"value"`, the `ViewModel` will update any HTML elements that are bound to its value whenever a change occurs. Note that input boxes don't update their value until they lose focus, so you won't see changes to bound elements as you type.

Note: Adding support for live updating after each keypress is simply a matter of adding `valueUpdate: "afterkeydown"` to the `data-bind` attribute's contents.

Our `viewModel`, after updating it to use `ko.observable`:

```
var viewModel = {
    authorName: ko.observable('Steve Smith')
};
ko.applyBindings(viewModel);
```

Knockout supports a number of different kinds of bindings. So far we've seen how to bind to `text` and to `value`. You can also bind to any given attribute. For instance, to create a hyperlink with an anchor tag, the `src` attribute can be bound to the `viewModel`. Knockout also supports binding to functions. To demonstrate this, let's update the `viewModel` to include the author's twitter handle, and display the twitter handle as a link to the author's twitter page. We'll do this in three stages.

First, add the HTML to display the hyperlink, which we'll show in parentheses after the author's name:

```
<h1>Some Article</h1>
<p>
    By <span data-bind="text: authorName"></span>
```

```
</p> (<a data-bind="attr: { href: twitterUrl}, text: twitterAlias" ></a>)
```

Next, update the viewModel to include the twitterUrl and twitterAlias properties:

```
var viewModel = {
    authorName: ko.observable('Steve Smith'),
    twitterAlias: ko.observable('@ardalis'),
    twitterUrl: ko.computed(function() {
        return "https://twitter.com/";
    }, this)
};
ko.applyBindings(viewModel);
```

Notice that at this point we haven't yet updated the twitterUrl to go to the correct URL for this twitter alias – it's just pointing at twitter.com. Also notice that we're using a new Knockout function, `computed`, for twitterUrl. This is an observable function that will notify any UI elements if it changes. However, for it to have access to other properties in the viewModel, we need to change how we are creating the viewModel, so that each property is its own statement.

The revised viewModel declaration is shown below. It is now declared as a function. Notice that each property is its own statement now, ending with a semicolon. Also notice that to access the twitterAlias property value, we need to execute it, so its reference includes `()`.

```
function viewModel() {
    this.authorName = ko.observable('Steve Smith');
    this.twitterAlias = ko.observable('@ardalis');

    this.twitterUrl = ko.computed(function() {
        return "https://twitter.com/" + this.twitterAlias().replace('@', '');
    }, this)
};
ko.applyBindings(viewModel);
```

The result works as expected in the browser:



Knockout also supports binding to certain UI element events, such as the click event. This allows you to easily and declaratively bind UI elements to functions within the application's viewModel. As a simple example, we can add a button that, when clicked, modifies the author's twitterAlias to be all caps.

First, we add the button, binding to the button's click event, and referencing the function name we're going to add to the viewModel:

```
<p>  
    <button data-bind="click: capitalizeTwitterAlias">Capitalize</button>  
</p>
```

Then, add the function to the viewModel, and wire it up to modify the viewModel's state. Notice that to set a new value to the twitterAlias property, we call it as a method and pass in the new value.

```
function viewModel() {  
    this.authorName = ko.observable('Steve Smith');  
    this.twitterAlias = ko.observable('@ardalis');  
  
    this.twitterUrl = ko.computed(function() {  
        return "https://twitter.com/" + this.twitterAlias().replace('@', '');  
    }, this);  
  
    this.capitalizeTwitterAlias = function() {  
        var currentValue = this.twitterAlias();  
        this.twitterAlias(currentValue.toUpperCase());  
    };  
};  
ko.applyBindings(viewModel);
```

Running the code and clicking the button modifies the displayed link as expected:



Control Flow

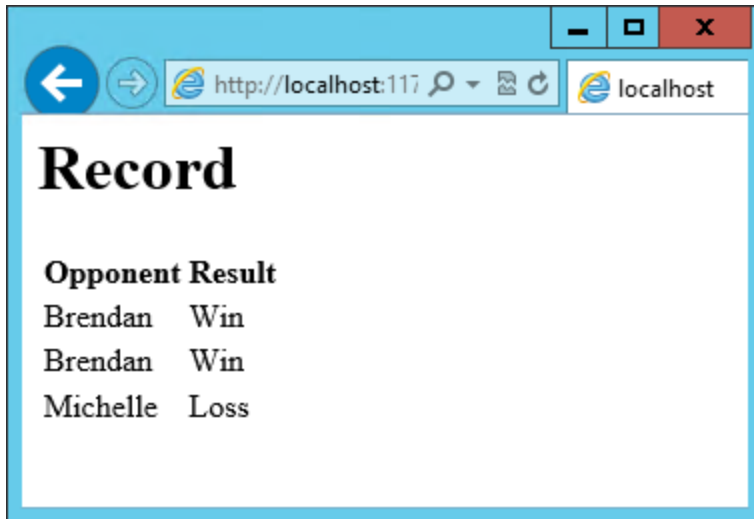
Knockout includes bindings that can perform conditional and looping operations. Looping operations are especially useful for binding lists of data to UI lists, menus, and grids or tables. The `foreach` binding will iterate over an array. When used with an observable array, it will automatically update the UI elements when items are added or removed from the array, without re-creating every element in the UI tree. The following example uses a new `viewModel` which includes an observable array of game results. It is bound to a simple table with two columns using a `foreach` binding on the `<tbody>` element. Each `<tr>` element within `<tbody>` will be bound to an element of the `gameResults` collection.

```

1 <h1>Record</h1>
2 <table>
3     <thead>
4         <tr>
5             <th>Opponent</th>
6             <th>Result</th>
7         </tr>
8     </thead>
9     <tbody data-bind="foreach: gameResults">
10         <tr>
11             <td data-bind="text:opponent"></td>
12             <td data-bind="text:result"></td>
13         </tr>
14     </tbody>
15 </table>
16 <script type="text/javascript">
17     function GameResult(opponent, result) {
18         var self = this;
19         self.opponent = opponent;
20         self.result = ko.observable(result);
21     }
22
23     function ViewModel() {
24         var self = this;
25
26         self.resultChoices = ["Win", "Loss", "Tie"];
27
28         self.gameResults = ko.observableArray([
29             new GameResult("Brendan", self.resultChoices[0]),
30             new GameResult("Brendan", self.resultChoices[0]),
31             new GameResult("Michelle", self.resultChoices[1])
32         ]);
33     };
34     ko.applyBindings(new ViewModel());
35 </script>

```

Notice that this time we're using `ViewModel` with a capital "V" because we expect to construct it using "new" (in the `applyBindings` call). When executed, the page results in the following output:



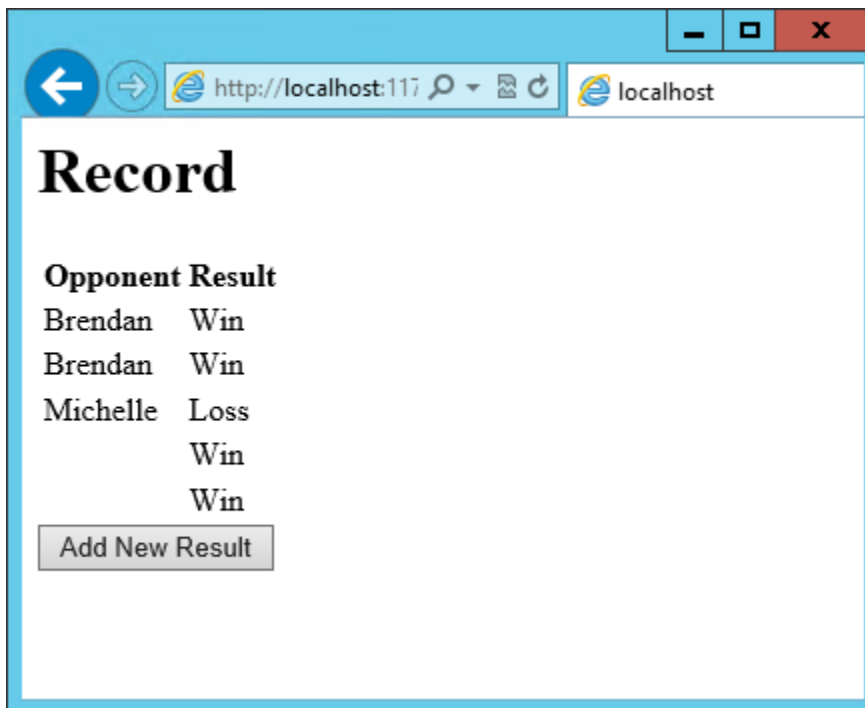
To demonstrate that the observable collection is working, let's add a bit more functionality. We can include the ability to record the results of another game to the ViewModel, and then add a button and some UI to work with this new function. First, let's create the `addResult` method:

```
// add this to ViewModel()
self.addResult = function() {
    self.gameResults.push(new GameResult("", self.resultChoices[0]));
}
```

Bind this method to a button using the `click` binding:

```
<button data-bind="click: addResult">Add New Result</button>
```

Open the page in the browser and click the button a couple of times, resulting in a new table row with each click:



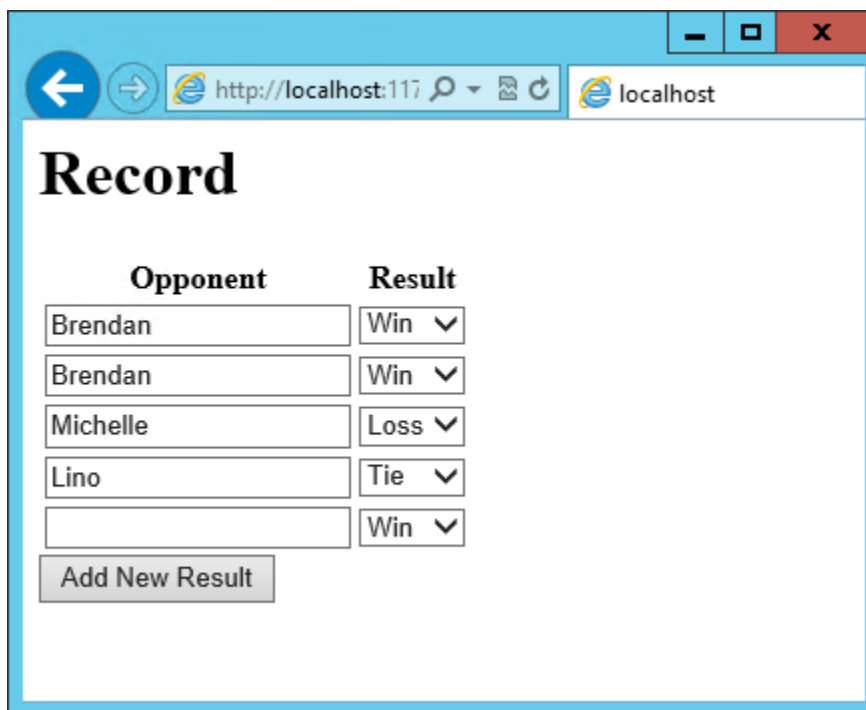
There are a few ways to support adding new records in the UI, typically either inline or in a separate form. We can

easily modify the table to use textboxes and dropdownlists so that the whole thing is editable. Just change the `<tr>` element as shown:

```
<tbody data-bind="foreach: gameResults">
  <tr>
    <td><input data-bind="value:opponent" /></td>
    <td><select data-bind="options: $root.resultChoices,
      value:result, optionsText: $data"></select></td>
  </tr>
</tbody>
```

Note that `$root` refers to the root ViewModel, which is where the possible choices are exposed. `$data` refers to whatever the current model is within a given context - in this case it refers to an individual element of the `resultChoices` array, each of which is a simple string.

With this change, the entire grid becomes editable:



If we weren't using Knockout, we could achieve all of this using jQuery, but most likely it would not be nearly as efficient. Knockout tracks which bound data items in the ViewModel correspond to which UI elements, and only updates those elements that need to be added, removed, or updated. It would take significant effort to achieve this ourselves using jQuery or direct DOM manipulation, and even then if we then wanted to display aggregate results (such as a win-loss record) based on the table's data, we would need to once more loop through it and parse the HTML elements. With Knockout, displaying the win-loss record is trivial. We can perform the calculations within the ViewModel itself, and then display it with a simple text binding and a ``.

To build the win-loss record string, we can use a computed observable. Note that references to observable properties within the ViewModel must be function calls, otherwise they will not retrieve the value of the observable (i.e. `gameResults()` not `gameResults` in the code shown):

```
self.displayRecord = ko.computed(function () {
  var wins = self.gameResults().filter(function (value) { return value.result() == "Win"; }).length;
  var losses = self.gameResults().filter(function (value) { return value.result() == "Loss"; }).length;
  var ties = self.gameResults().filter(function (value) { return value.result() == "Tie"; }).length;
  return wins + " - " + losses + " - " + ties;
});
```

```
}, this);
```

Bind this function to a span within the `<h1>` element at the top of the page:

```
<h1>Record <span data-bind="text: displayRecord"></span></h1>
```

The result:



Adding rows or modifying the selected element in any row's Result column will update the record shown at the top of the window.

In addition to binding to values, you can also use almost any legal JavaScript expression within a binding. For example, if a UI element should only appear under certain conditions, such as when a value exceeds a certain threshold, you can specify this logically within the binding expression:

```
<div data-bind="visible: customerValue > 100"></div>
```

This `<div>` will only be visible when the `customerValue` is over 100.

Templates

Knockout has support for templates, so that you can easily separate your UI from your behavior, or incrementally load UI elements into a large application on demand. We can update our previous example to make each row its own template by simply pulling the HTML out into a template and specifying the template by name in the `data-bind` call on `<tbody>`.

```
<tbody data-bind="template: { name: 'rowTemplate', foreach: gameResults">
</tbody>
<script type="text/html" id="rowTemplate">
  <tr>
    <td><input data-bind="value:opponent" /></td>
    <td><select data-bind="options: $root.resultChoices,
      value:result, optionsText: $data"></select></td>
```

```
</tr>  
</script>
```

Knockout also supports other templating engines, such as the jQuery.tmpl library and Underscore.js's templating engine.

Components

Components allow you to organize and reuse UI code, usually along with the ViewModel data on which the UI code depends. To create a component, you simply need to specify its template and its viewModel, and give it a name. This is done by calling `ko.components.register()`. In addition to defining the templates and viewModel inline, they can be loaded from external files using a library like `require.js`, resulting in very clean and efficient code.

Communicating with APIs

Knockout can work with any data in JSON format. A common way to retrieve and save data using Knockout is with jQuery, which supports the `$.getJSON()` function to retrieve data, and the `$.post()` method to send data from the browser to an API endpoint. Of course, if you prefer a different way to send and receive JSON data, Knockout will work with it as well.

Summary

Knockout provides a simple, elegant way to bind UI elements to the current state of the client application, defined in a ViewModel. Knockout's binding syntax uses the data-bind attribute, applied to HTML elements that are to be processed. Knockout is able to efficiently render and update large data sets by tracking UI elements and only processing changes to affected elements. Large applications can break up UI logic using templates and components, which can be loaded on demand from external files. Currently version 3, Knockout is a stable JavaScript library that can improve web applications that require rich client interactivity.

2.8.6 Using Angular for Single Page Applications (SPAs)

By [Venkata Koppaka](#)

In this article, you will learn how to build a SPA-style ASP.NET application using AngularJS.

In this article:

- *[What is AngularJS?](#)*
- *[Getting Started](#)*
- *[Key Components](#)*
- *[Angular 2.0](#)*

[View this article's samples on GitHub.](#)

What is AngularJS?

AngularJS is a modern JavaScript framework from Google commonly used to work with Single Page Applications (SPAs). AngularJS is open sourced under MIT license and the development progress of AngularJS can be followed on its [GitHub repository](#). The library is called Angular because HTML uses angular brackets.

AngularJS is not a DOM manipulation library like jQuery but it uses a subset of jQuery called jQLite. AngularJS is primarily based on declarative HTML attributes that you can add to your HTML tags. You can try AngularJS in your browser using the [Code School website](#).

Version 1.3.x is the current stable version and the Angular team is working towards a big rewrite of AngularJS for V2.0 which is currently still in development. This article focuses on Angular 1.X with some notes on where Angular is heading with 2.0.

Getting Started

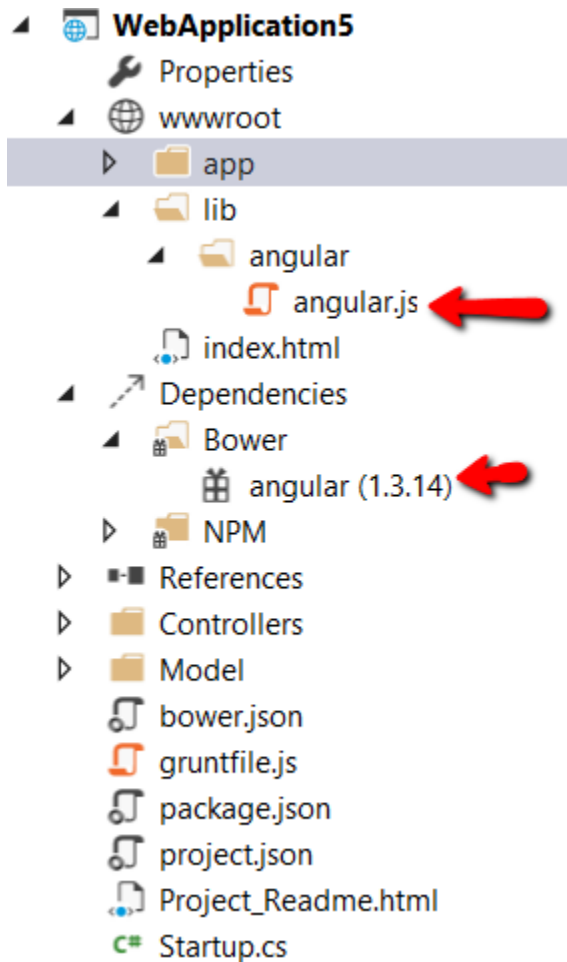
To start using AngularJS in your ASP.NET application, you must either install it as part of your project, or reference it from a content delivery network (CDN).

Installation

There are several ways to add AngularJS to your application. If you're starting a new web application in Visual Studio 2015 and ASP.NET 5, you can add AngularJS using the built-in NPM and Bower support. Simply open `bower.json` and add an entry to the `dependencies` property:

```
1 {
2   "name": "ASP.NET",
3   "private": true,
4   "dependencies": {
5     "bootstrap": "3.0.0",
6     "bootstrap-touch-carousel": "0.8.0",
7     "hammer.js": "2.0.4",
8     "jquery": "2.1.4",
9     "jquery-validation": "1.11.1",
10    "jquery-validation-unobtrusive": "3.2.2",
11    "angular": "1.3.15",
12    "angular-route": "1.3.15"
13  }
14 }
```

Once you save the file, Angular will be installed for your project, located in the Bower folder. You can then use [Grunt](#) or [Gulp](#) to copy the appropriate files into your `wwwroot/lib` folder, as shown:



Next, add a `<script>` reference to the bottom of the `<body>` section of your html page or `_Layout.cshtml` file, as shown here:

```

1  <environment names="Development">
2    <script src="~/lib/jquery/dist/jquery.js"></script>
3    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
4    <script src="~/lib/hammer.js/hammer.js"></script>
5    <script src="~/lib/bootstrap-touch-carousel/dist/js/bootstrap-touch-carousel.js"></script>
6    <script src="~/lib/angular/angular.js"></script>
7  </environment>

```

It's recommended that production applications utilize CDNs for common libraries like Angular. You can reference Angular from one of several CDNs, such as this one:

```

1  <environment names="Staging,Production">
2    <script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
3      asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
4      asp-fallback-test="window.jQuery">
5    </script>
6    <script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/bootstrap.min.js"
7      asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
8      asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
9    </script>
10   <script src="//ajax.aspnetcdn.com/ajax/hammer.js/2.0.4/hammer.min.js"
11     asp-fallback-src="~/lib/hammer.js/hammer.js"
12     asp-fallback-test="window.Hammer">

```

```
13     </script>
14     <script src="//ajax.aspnetcdn.com/ajax/bootstrap-touch-carousel/0.8.0/js/bootstrap-touch-
15         asp-fallback-src=~\lib/bootstrap-touch-carousel/dist/js/bootstrap-touch-carouse
16         asp-fallback-test="window.Hammer && window.Hammer.Instance">
17     </script>
18     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"
19         asp-fallback-src=~\lib/angular/angular.min.js"
20         asp-fallback-test="window.angular">
21     </script>
22     <script src=~\js/site.js" asp-file-version="true"></script>
23 </environment>
```

Once you have a reference to the angular.js script file, you're ready to begin using Angular in your web pages.

Key Components

AngularJS includes a number of major components, such as *directives*, *templates*, *repeaters*, *modules*, *controllers*, and more. Let's examine how these components work together to add behavior to your web pages.

Directives

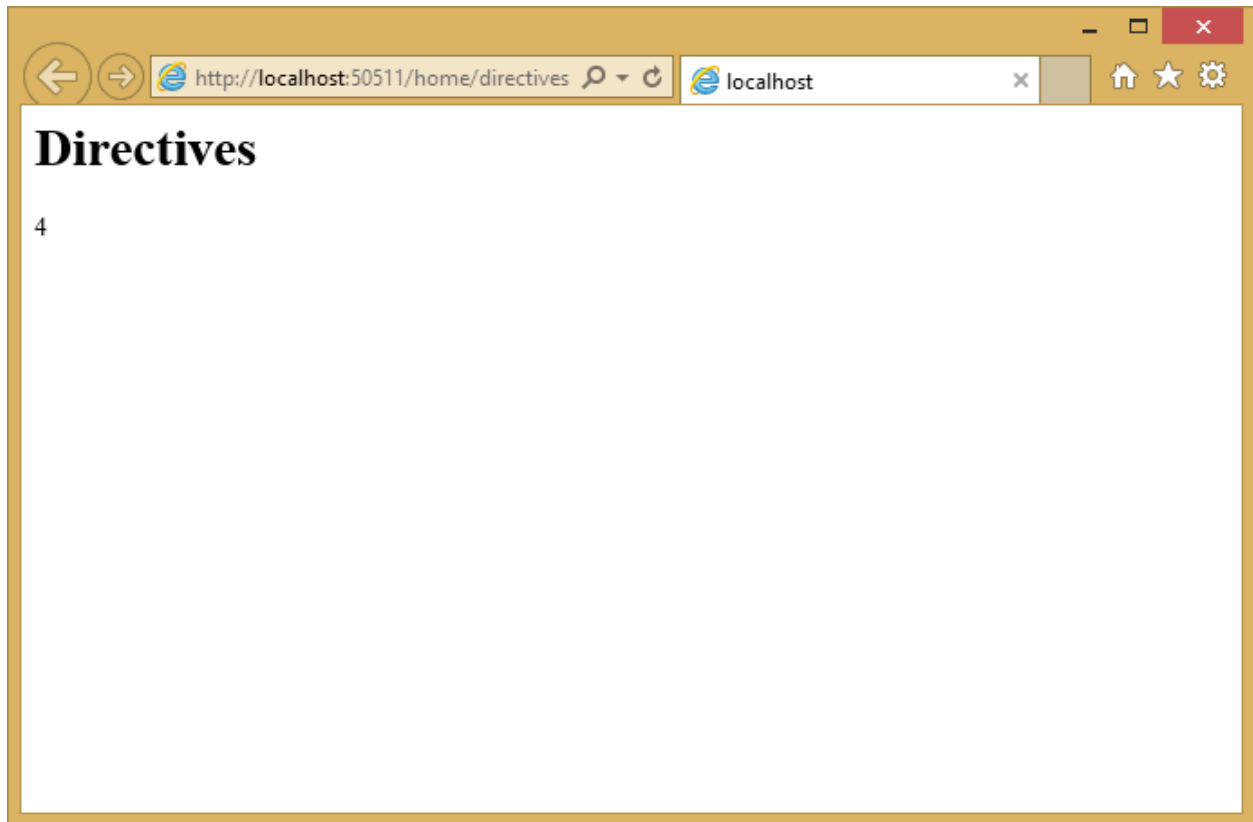
AngularJS uses *directives* to extend HTML with custom attributes and elements. AngularJS directives are defined via `data-ng-*` or `ng-*` prefixes (`ng` is short for angular). There are two types of AngularJS directives:

- Primitive Directives: These are pre-defined by the Angular team and are part of the AngularJS framework.
- Custom Directives: These are custom directives that you can define.

One of the primitive directives used in all AngularJS applications is the `ng-app` directive used to bootstrap the AngularJS application. This directive can be added to the whole document body or to specific pieces of the body. Let's see an example in action. Assuming you're in an ASP.NET project, you can either add an HTML file to the `wwwroot` folder, or add a new controller action and associated view. In this case, I've added a new `Directives()` action method to `HomeController.cs` and the associated view is shown here:

```
1 @{ Layout = "";
2 }
3 <html>
4 <body ng-app>
5     <h1>Directives</h1>
6     {{2+2}}
7     <script src=~\lib/angular/angular.js"></script>
8 </body>
9 </html>
```

To keep these samples independent of one another, I'm not using the shared layout file. You can see that we decorated the body tag with the `ng-app` directive to indicate this page is an AngularJS application. The `{{2+2}}` is an Angular data binding expression that you will learn more about in a moment. Here is the result if you run this application:



Other primitive directives in AngularJS include:

ng-controller Determines which JavaScript controller is bound to which view.

ng-model Determines what model the values of an HTML element's properties are bound to.

ng-init Used to initialize the application data in the form of an expression for the current scope.

ng-if If clause used within your AngularJS application; usually used with an expression.

ng-repeat Repeats a given block of HTML over a set of data.

ng-show Shows or hides the given HTML element based on the expression provided.

For a full list of all primitive directives supported in AngularJS please refer to the [directive documentation section on the AngularJS documentation website](#).

Data Binding

AngularJS provides [data binding](#) support out-of-the-box using either the `ng-bind` directive or a data binding expression syntax such as `{{expression}}`. AngularJS supports two-way data binding where data from a model is kept in synchronization with a view template at all times. Any changes to the view are automatically reflected in the model and any changes in the model are likewise reflected in the view.

Create a new HTML file or controller action and view called `Databinding` and include the following:

```

1 @ { Layout = "";
2 }
3 <html>
4 <body ng-app>
5   <h1>Databinding</h1>

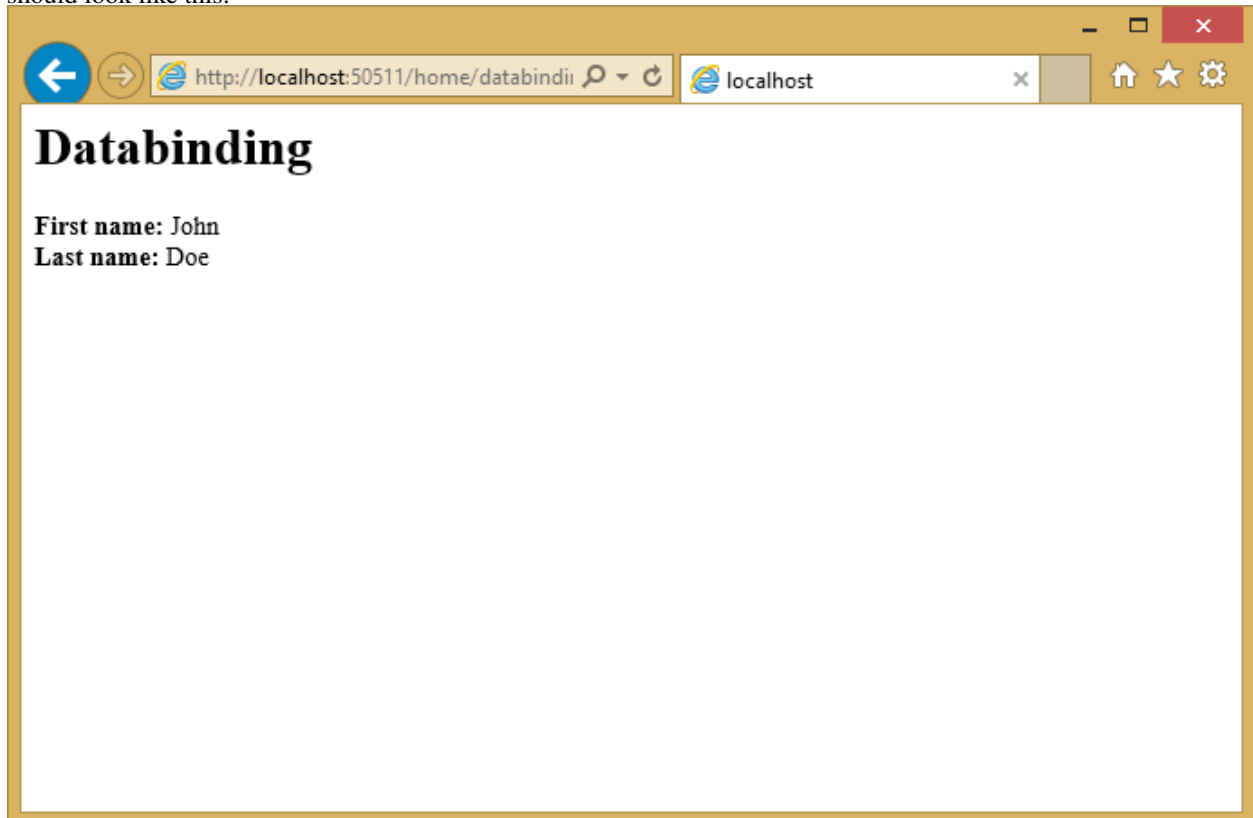
```

```

6      <div ng-init="firstName='John'; lastName='Doe';">
7          <strong>First name:</strong> {{firstName}} <br />
8          <strong>Last name:</strong> <span ng-bind="lastName" />
9      </div>
10
11      <script src="~/lib/angular/angular.js"></script>
12 </body>
13 </html>
14

```

Notice that you can display model values using either directives or data binding (`ng-bind`). The resulting page should look like this:



Templates

Templates in AngularJS are just plain HTML pages decorated with AngularJS directives and artifacts. A template in AngularJS is a mixture of directives, expressions, filters, and controls that combine with HTML to form the view.

Add another view to demonstrate templates, and add the following to it:

```

1  @{{ Layout = "";
2  }}
3  <html>
4  <body ng-app>
5      <h1>Templates</h1>
6
7      <div ng-init="personName='John Doe'">
8          <input ng-model="personName" /> {{personName}}
9      </div>

```

```

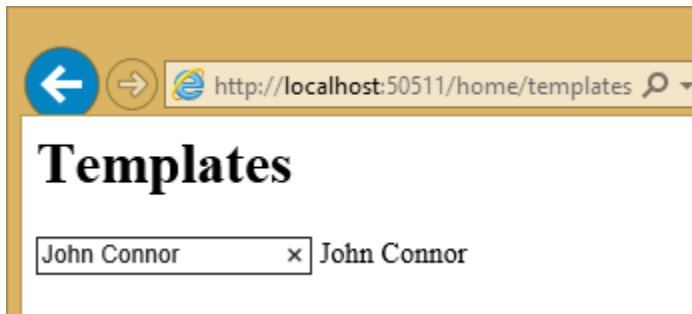
10  <script src="/lib/angular/angular.js"></script>
11  </body>
12  </html>
13

```

The template has AngularJS directives like `ng-app`, `ng-init`, `ng-model` and data binding expression syntax to bind the `personName` property. Running in the browser, the view looks like the screenshot below:



If you change the name by typing in the input field, you will see the text next to the input field dynamically update, showing Angular two-way data binding in action.



Expressions

Expressions in AngularJS are JavaScript-like code snippets that are written inside the `{{ expression }}` syntax. The data from these expressions are bound to HTML the same way as `ng-bind` directives. The main difference between AngularJS expressions and regular JavaScript expressions is that AngularJS expressions are evaluated against the `$scope` object in AngularJS.

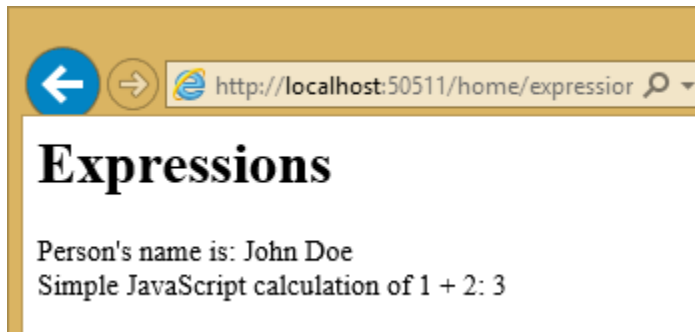
The AngularJS expressions in the sample below bind `personName` and a simple JavaScript calculated expression:

```

1  @{{ Layout = "";
2  }}
3  <html>
4  <body ng-app>
5      <h1>Expressions</h1>
6
7      <div ng-init="personName='John Doe'">
8          Person's name is: {{personName}} <br />
9          Simple JavaScript calculation of 1 + 2: {{1+2}}
10     </div>
11
12     <script src="/lib/angular/angular.js"></script>
13 </body>
14 </html>

```

The example running in the browser displays the `personName` data and the results of the calculation:

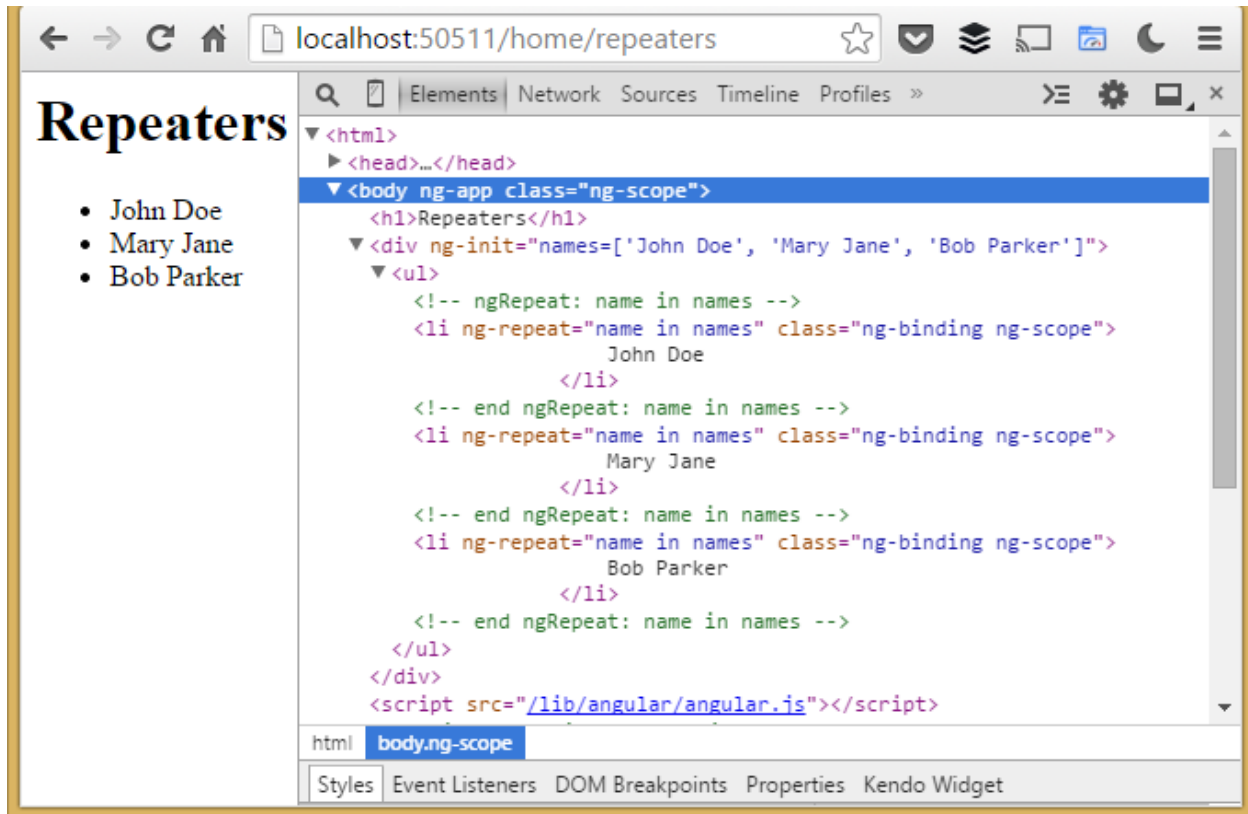


Repeaters

Repeating in AngularJS is done via a primitive directive called `ng-repeat`. The `ng-repeat` directive repeats a given HTML element in a view over the length of a repeating data array. Repeaters in AngularJS can repeat over an array of strings or objects. Here is a sample usage of repeating over an array of strings:

```
1 @{{ Layout = "";
2 }
3 <html>
4 <body ng-app>
5   <h1>Repeaters</h1>
6
7   <div ng-init="names=['John Doe', 'Mary Jane', 'Bob Parker']">
8     <ul>
9       <li ng-repeat="name in names">
10         {{ name }}
11       </li>
12     </ul>
13   </div>
14
15   <script src="~/lib/angular/angular.js"></script>
16 </body>
17 </html>
```

The `repeat directive` outputs a series of list items in an unordered list, as you can see in the developer tools shown in this screenshot:



Here is an example that repeats over an array of objects. The `ng-init` directive establishes a names array, where each element is an object containing first and last names. The `ng-repeat` assignment, `name in names`, outputs a list item for every array element.

```

1  @ { Layout = "";
2  }
3  <html>
4  <body ng-app>
5      <h1>Repeaters2</h1>
6
7      <div ng-init="names=[
8          {firstName:'John', lastName:'Doe'},
9          {firstName:'Mary', lastName:'Jane'},
10         {firstName:'Bob', lastName:'Parker'}]">
11
12         <ul>
13             <li ng-repeat="name in names">
14                 {{name.firstName + ' ' + name.lastName}}
15             </li>
16         </ul>
17     </div>
18
19     <script src="~/lib/angular/angular.js"></script>
20 </body>
</html>

```

The output in this case is the same as in the previous example.

Angular provides some additional directives that can help provide behavior based on where the loop is in its execution.

\$index Use `$index` in the `ng-repeat` loop to determine which index position your loop currently is on.

\$even and \$odd Use `$even` in the `ng-repeat` loop to determine whether the current index in your loop is an

even indexed row. Similarly, use `$odd` to determine if the current index is an odd indexed row.

\$first and \$last Use `$first` in the `ng-repeat` loop to determine whether the current index in your loop is the first row. Similarly, use `$last` to determine if the current index is the last row.

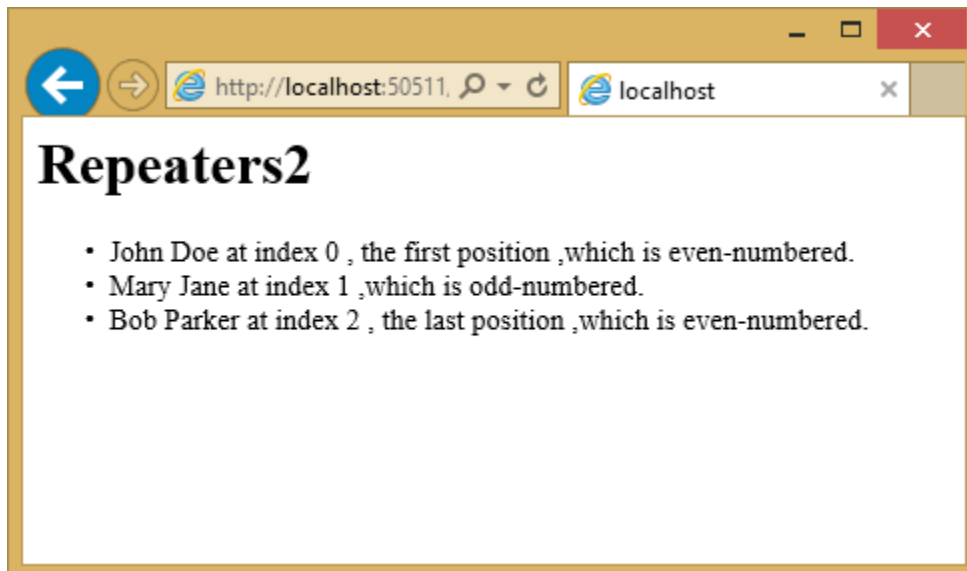
Below is a sample that shows `$index`, `$even`, `$odd`, `$first`, `$odd` in action:

```

1  @{{ Layout = "";
2  }}
3  <html>
4  <body ng-app>
5    <h1>Repeaters2</h1>
6
7    <div ng-init="names=[
8      {firstName:'John', lastName:'Doe'},
9      {firstName:'Mary', lastName:'Jane'},
10     {firstName:'Bob', lastName:'Parker'}]">
11
12     <ul>
13       <li ng-repeat="name in names">
14         {{name.firstName + ' ' + name.lastName}} at index {{ $index }}
15         <span ng-show="{{ $first }}">, the first position</span>
16         <span ng-show="{{ $last }}">, the last position</span>
17         <span ng-show="{{ $odd }}">, which is odd-numbered.</span>
18         <span ng-show="{{ $even }}">, which is even-numbered.</span>
19       </li>
20     </ul>
21   </div>
22
23   <script src="~/lib/angular/angular.js"></script>
24 </body>
</html>

```

Here is the resulting output:



\$scope

`$scope` is a JavaScript object that acts as glue between the view (template) and the controller (explained below). A view template in AngularJS only knows about the values that are set on the `$scope` object from the controller.

Note: In the MVVM world, the `$scope` object in AngularJS is often defined as the ViewModel. The AngularJS team refers to the `$scope` object as the Data-Model. [Learn more about Scopes in AngularJS](#)

Below is a simple example showing how to set properties on `$scope` within a separate JavaScript file, `scope.js`:

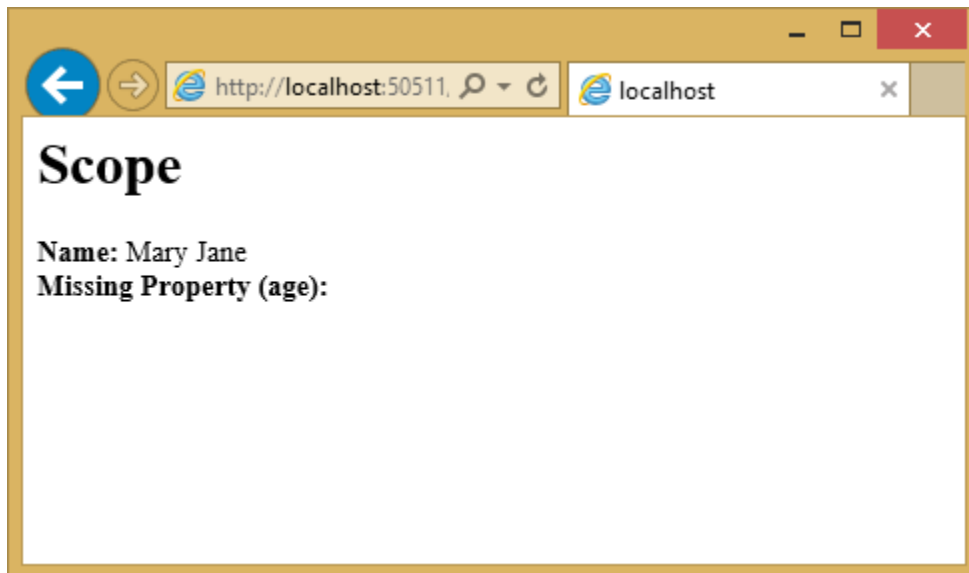
```
1 var personApp = angular.module('personApp', []);
2 personApp.controller('personController', ['$scope', function ($scope) {
3     $scope.name = 'Mary Jane';
4 }]);
```

Observe the `$scope` parameter passed to the controller on line #2. This object is what the view knows about. In line #3, we are setting a property called “name” to “Mary Jane”.

What happens when a particular property is not found by the view? The view defined below refers to name and age properties:

```
1 @{{ Layout = "";
2 }}
3 <html>
4 <body ng-app="personApp">
5     <h1>Scope</h1>
6
7     <div ng-controller="personController">
8         <strong>Name:</strong> {{name}} <br />
9         <strong>Missing Property (age):</strong> {{age}}
10    </div>
11
12    <script src="~/lib/angular/angular.js"></script>
13    <script src="~/app/scope.js"></script>
14 </body>
15 </html>
```

Notice in line #8 that we are asking Angular to show the “name” property using expression syntax. Line #9 then refers to “age”, a property that does not exist. The running example shows the name set to “Mary Jane” and nothing for age - missing properties are ignored.



Modules

A **module** in AngularJS is a collection of controllers, services, directives etc. The `angular.module()` function call is used to create, register and retrieve modules in AngularJS. All modules, including those shipped by the AngularJS team and third party libraries should be registered using the `angular.module()` function.

Below is a snippet of code that shows how to create a new module in AngularJS. The first parameter is the name of the module. The second parameter defines dependencies on other modules. Later in this article we will be showing how to pass these dependencies to an `angular.module()` method call.

```
var personApp = angular.module('personApp', []);
```

Use the `ng-app` directive to represent an AngularJS module on the page. To use a module, assign the name of the module, `personApp` in this example, to the `ng-app` directive in our template.

```
<body ng-app="personApp">
```

Controllers

Controllers in AngularJS are the first point of entry for your code. The `<module name>.controller()` function call is used to create and register controllers in AngularJS. The `ng-controller` directive is used to represent an AngularJS controller on the HTML page. The role of the controller in Angular is to set state and behavior of the data model (`$scope`). Controllers should not be used to manipulate the DOM directly.

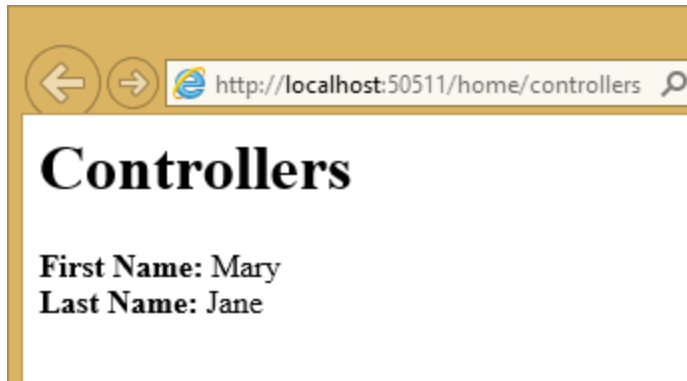
Below is a snippet of code that registers a new controller. The `personApp` variable in the snippet is an Angular module, defined on line 5.

```
1 // module
2 var personApp = angular.module('personApp', []);
3
4 // controller
5 personApp.controller('personController', function ($scope) {
6     $scope.firstName = "Mary";
7     $scope.lastName = "Jane"
8 });
```

The view using the `ng-controller` directive assigns the controller name:

```
1 @{} Layout = "";
2 }
3 <html>
4 <body ng-app="personApp">
5     <h1>Controllers</h1>
6
7     <div ng-controller="personController">
8         <strong>First Name:</strong> {{firstName}} <br />
9         <strong>Last Name:</strong> {{lastName}}
10    </div>
11
12    <script src="~/lib/angular/angular.js"></script>
13    <script src="~/app/controllers.js"></script>
14 </body>
15 </html>
```

The page shows “Mary” and “Jane” that correspond to the `firstName` and `lastName` properties assigned to the `$scope` object.



Services

Services in AngularJS are commonly used shared code that are abstracted away into a file that can be used throughout the lifetime of an angular application. Services are lazily instantiated, meaning that there will not be an instance of a service unless a component that depends on the service gets used. Factories are an example of a service used in AngularJS applications. Factories are created using the `myApp.factory()` function call, where `myApp` is the module.

Below is an example that shows how to use factories in AngularJS:

```

1 personApp.factory('personFactory', function () {
2     function getName() {
3         return "Mary Jane";
4     }
5
6     var service = {
7         getName: getName
8     };
9
10    return service;
11 });

```

To call this factory from the controller, pass `personFactory` as a parameter to the `controller()` function:

```

personApp.controller('personController', function($scope, personFactory) {
    $scope.name = personFactory.getName();
});

```

Using services to talk to a REST endpoint

Below is an end-to-end example using services in AngularJS to interact with an ASP.NET 5 Web API endpoint. The example gets data from the Web API built using ASP.NET 5 and displays the data in a view template. Let's start with the view first:

```

1 @Layout = "";
2 }
3 <html>
4 <body ng-app="PersonsApp">
5     <h1>People</h1>
6
7     <div ng-controller="personController">
8         <ul>

```

```
9         <li ng-repeat="person in people">
10             <h2>{{person.FirstName}} {{person.LastName}} </h2>
11         </li>
12     </ul>
13 </div>
14
15 <script src="~/lib/angular/angular.js"></script>
16 <script src="~/app/personApp.js"></script>
17 <script src="~/app/personFactory.js"></script>
18 <script src="~/app/personController.js"></script>
19 </body>
20 </html>
```

In this view, we have an Angular module called `PersonsApp` and a controller called `personController`. We are using `ng-repeat` to iterate over the list of persons. We are referencing three separate script files on lines 16-18.

The `personApp.js` file is used to register the `PersonsApp` module. The syntax is similar to previous examples. We are using the `angular.module()` function to create a new instance of the module that we will be working with.

```
1 (function () {
2     'use strict';
3     var app = angular.module('PersonsApp', []);
4 }) ();
```

Let's take a look at `personFactory.js`, below. We are calling the module's `factory()` method to create a factory. Line #12 shows the built-in Angular `$http` service retrieving people information from a web service.

```
1 (function () {
2     'use strict';
3
4     var serviceId = 'personFactory';
5
6     angular.module('PersonsApp').factory(serviceId,
7         ['$http', personFactory]);
8
9     function personFactory($http) {
10
11         function getPeople() {
12             return $http.get('/api/people');
13         }
14
15         var service = {
16             getPeople: getPeople
17         };
18
19         return service;
20     }
21 }) ();
```

In `personController.js`, we are calling the module's `controller()` method to create the controller. The `$scope` object's `people` property is assigned the data returned from the `personFactory` (line #13).

```
1 (function () {
2     'use strict';
3
4     var controllerId = 'personController';
5
6     angular.module('PersonsApp').controller(controllerId,
```

```

7      ['$scope', 'personFactory', personController]);
8
9      function personController($scope, personFactory) {
10         $scope.people = [];
11
12         personFactory.getPeople().success(function (data) {
13             $scope.people = data;
14         }).error(function (error) {
15             // log errors
16         });
17     }
18 }) ();

```

Let's take a quick look at the ASP.NET 5 Web API and the model behind it. The Person model is a plain POCO (Plain Old CLR Object) with Id, FirstName and LastName properties:

```

1 namespace AngularSample.Models
2 {
3     public class Person
4     {
5         public int Id { get; set; }
6         public string FirstName { get; set; }
7         public string LastName { get; set; }
8     }
9 }

```

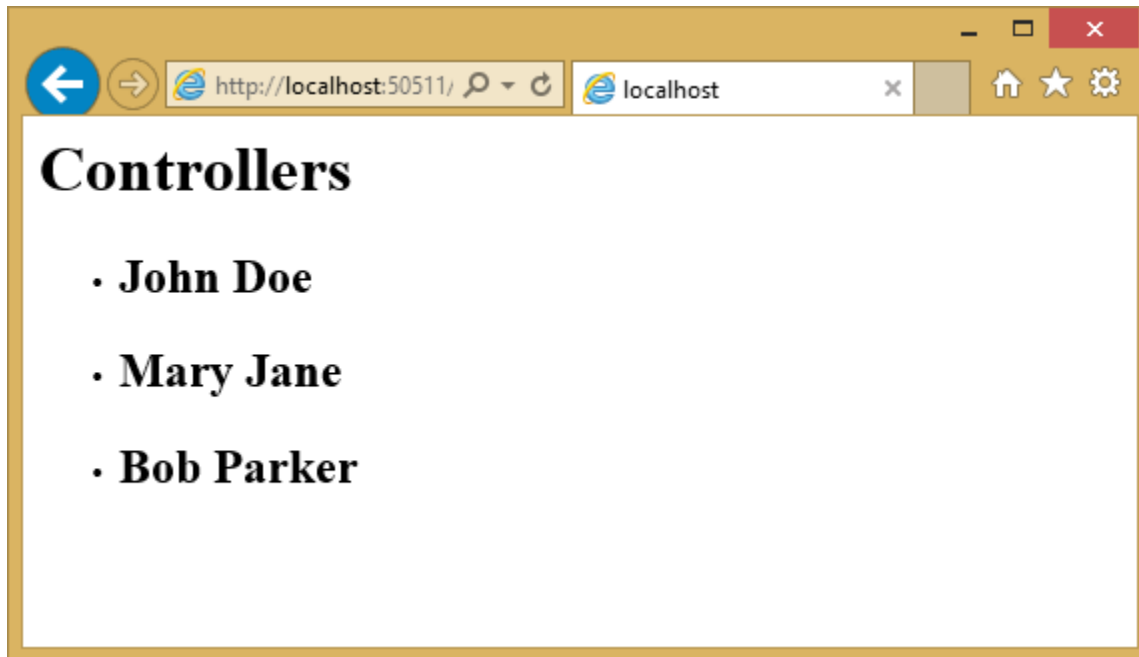
The Person controller returns a JSON-formatted list of Persons.

```

1 using AngularSample.Models;
2 using Microsoft.AspNet.Mvc;
3 using System.Collections.Generic;
4
5 namespace AngularSample.Controllers.Api
6 {
7     public class PersonController : Controller
8     {
9         [Route("/api/people")]
10        public JsonResult GetPeople()
11        {
12            var people = new List<Person>()
13            {
14                new Person { Id = 1, FirstName = "John", LastName = "Doe" },
15                new Person { Id = 1, FirstName = "Mary", LastName = "Jane" },
16                new Person { Id = 1, FirstName = "Bob", LastName = "Parker" }
17            };
18
19            return Json(people);
20        }
21    }
22 }

```

Let's see the application in action:



You can [view the application's structure on GitHub](#).

Note: For more on structuring AngularJS applications, see [John Papa's Angular Style Guide](#)

Note: To create AngularJS module, controller, factory, directive and view files easily, be sure to check out Sayed Hashimi's [SideWaffle template pack for Visual Studio](#). Sayed Hashimi is a Senior Program Manager on the Visual Studio Web Team at Microsoft and SideWaffle templates are considered the gold standard. At the time of this writing, SideWaffle is only available for Visual Studio 2012 and 2013.

Routing and Multiple Views

AngularJS has a built-in route provider to handle SPA (Single Page Application) based navigation. To work with routing in AngularJS you have to add the `angular-route` library using NPM or Bower. You can see in the [project.json](#) file referenced at the start of this article that we are already referencing it in our project.

After you install the package, add the script reference (`angular-route.js`) to your view.

Now let's take the Person App we have been building and add navigation to it. First, we will make a copy of the app by creating a new `PeopleController` action called `Spa` and a corresponding `Spa.cshtml` View by copying the `Index.cshtml` view in the `People` folder. Add a script reference to `angular-route` (see line #11). Also add a `div` marked with the `ng-view` directive (see line #6) as a placeholder to place views in. We are going to be using several additional `.js` files which are referenced on lines 12-15.

```
1  @{ Layout = "";  
2  }  
3  <html>  
4  <body ng-app="personApp">  
5  
6      <div ng-view>  
7  
8      </div>  
9  </body>  
10 </html>
```

```

10 <script src="~/lib/angular/angular.js"></script>
11 <script src="~/lib/angular-route/angular-route.js"></script>
12
13 <script src="~/app/personModule.js"></script>
14 <script src="~/app/personRoutes.js"></script>
15 <script src="~/app/personListController.js"></script>
16 <script src="~/app/personDetailController.js"></script>
17 </body>
18 </html>

```

Let's take a look at `personModule.js` file to see how we are instantiating the module with routing. We are passing `ngRoute` as a library into the module. This module handles routing in our application.

```

1 var personApp = angular.module('personApp', ['ngRoute']);

```

The `personRoutes.js` file, below, defines routes based on the route provider. Lines 4-7 define navigation by effectively saying, when a URL with `/persons` is requested, use a template called `partials/personList` by working through `personListController`. Lines 8-11 indicate a detail page with a route parameter of `personId`. If the URL doesn't match one of the patterns, Angular defaults to the `/persons` view.

```

1 personApp.config(['$routeProvider',
2     function ($routeProvider) {
3         $routeProvider.
4             when('/persons', {
5                 templateUrl: '/app/partials/personlist.html',
6                 controller: 'personListController'
7             }).
8             when('/persons/:personId', {
9                 templateUrl: '/app/partials/persondetail.html',
10                controller: 'personDetailController'
11             }).
12             otherwise({
13                 redirectTo: '/persons'
14             })
15         }
16 ]);

```

The `personlist.html` file is a partial view, which only has HTML that is needed to show person list.

```

1 <div>
2     <h1>PERSONS PAGE</h1>
3     <span ng-bind="message"/>
4 </div>

```

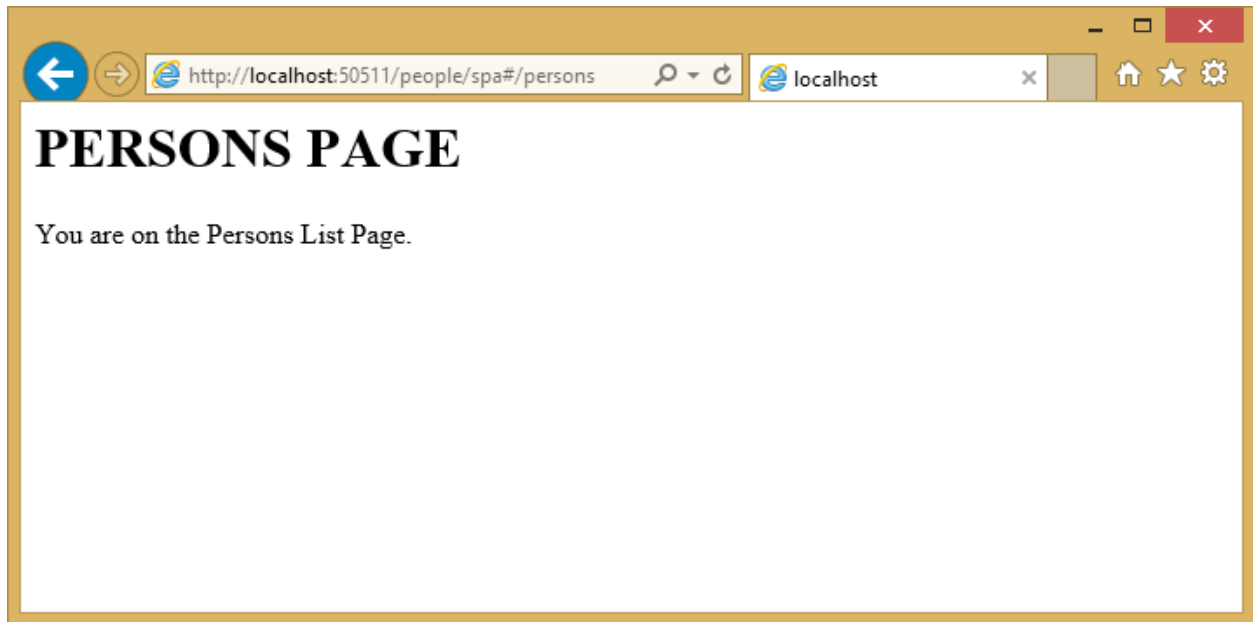
The controller is defined by using the module's `controller()` function in `personListController.js`.

```

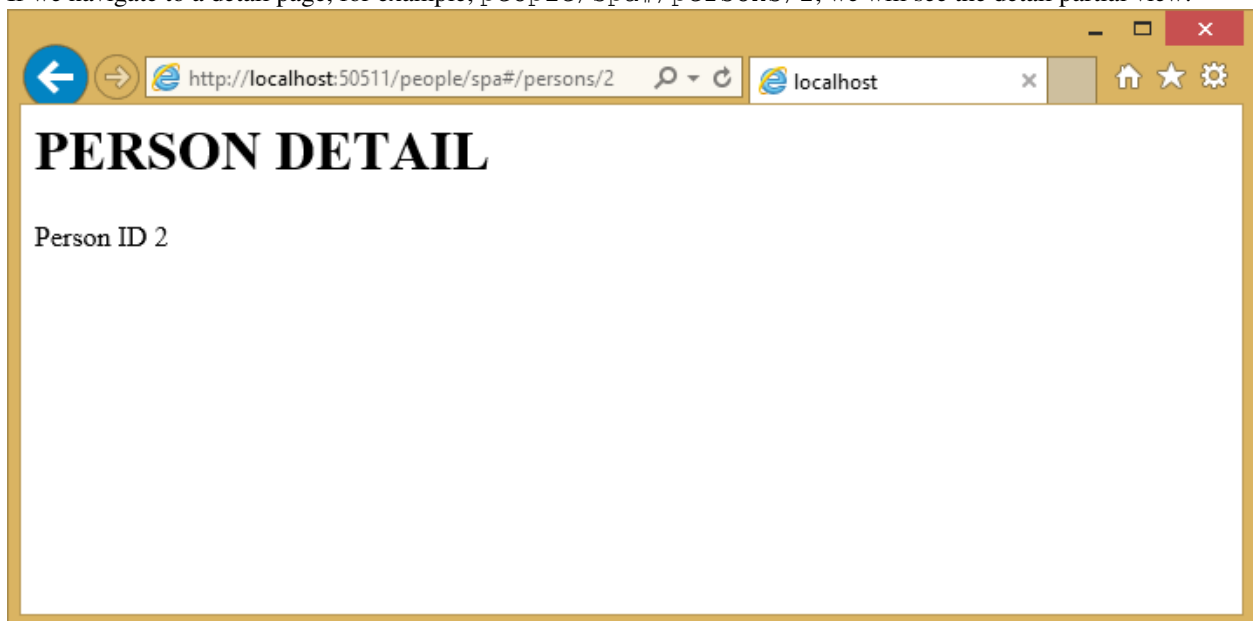
1 personApp.controller('personListController', function ($scope) {
2     $scope.message = "You are on the Persons List Page.";
3 })

```

If we run this application and go to the `people/spa#/persons` URL we will see:



If we navigate to a detail page, for example, `people/spa#/persons/2`, we will see the detail partial view:



You can view the full source and any files not shown in this article on [GitHub](#).

Event Handlers

There are a number of directives in AngularJS that add event-handling capabilities to the input elements in your HTML DOM. Below is a list of the events that are built into AngularJS.

- `ng-click`
- `ng-dbl-click`
- `ng-mousedown`
- `ng-mouseup`

- ng-mouseenter
- ng-mouseleave
- ng-mousemove
- ng-keydown
- ng-keyup
- ng-keypress
- ng-change

Note: You can add your own event handlers using the [custom directives](#) feature in AngularJS.

Let's look at how the ng-click event is wired up. Create a new JavaScript file, `eventHandlerController.js`, and add the following to it.

```

1 personApp.controller('eventHandlerController', function ($scope) {
2     $scope.firstName = 'Mary';
3     $scope.lastName = 'Jane';
4
5     $scope.sayName = function () {
6         alert('Welcome, ' + $scope.firstName + ' ' + $scope.lastName);
7     }
8 });

```

Notice in the `eventHandlerController` we now have a new `sayName()` function (line #5). All the method is doing for now is showing a JavaScript alert to the user with a welcome message.

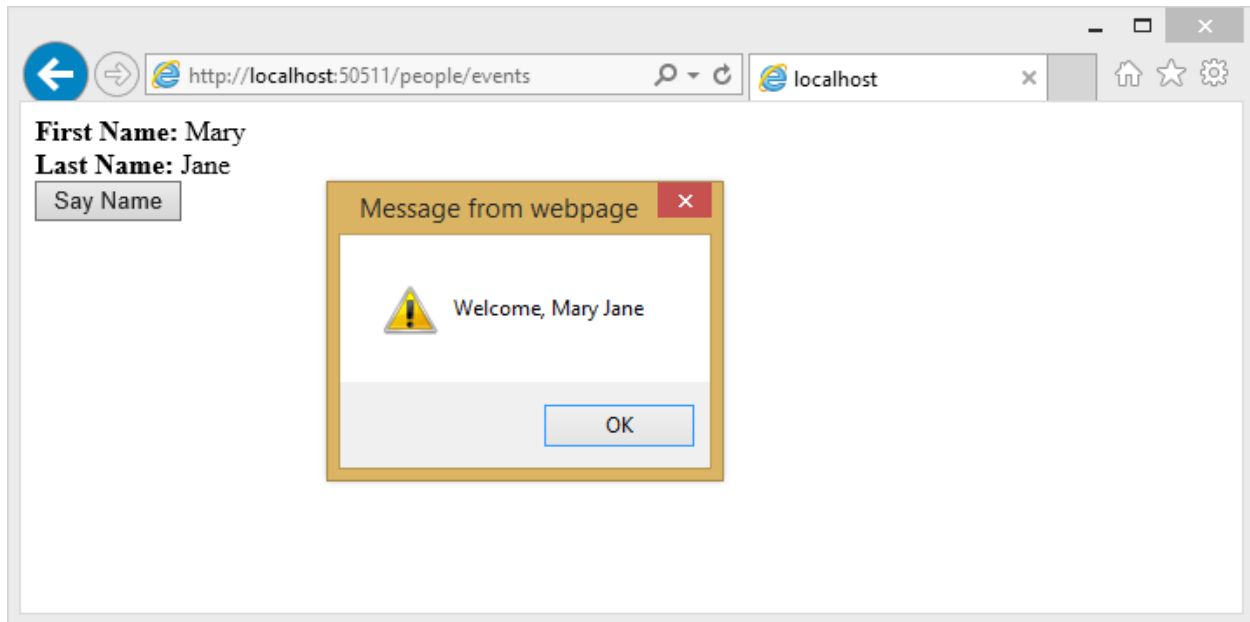
The view below binds a controller function to an AngularJS event. Line #8 has a new `<input>` element of type button marked with the `ng-click` angular directive that calls our `sayName()` function which is part of the `$scope` object passed to this view.

```

1 @{{ Layout = "";
2 }
3 <html>
4 <body ng-app="personApp">
5     <div ng-controller="eventHandlerController">
6         <strong>First Name:</strong> {{firstName}} <br />
7         <strong>Last Name:</strong> {{lastName}} <br />
8         <input ng-click="sayName()" type="button" value="Say Name" />
9     </div>
10    <script src="~/lib/angular/angular.js"></script>
11    <script src="~/lib/angular-route/angular-route.js"></script>
12
13    <script src="~/app/personModule.js"></script>
14    <script src="~/app/eventHandlerController.js"></script>
15 </body>
16 </html>

```

The running example demonstrates that the controller's `sayName()` function is called automatically when the button is clicked.



For more detail on AngularJS built-in event handler directives, be sure to head to the [documentation website](#) of AngularJS.

Angular 2.0

Angular 2.0 is the next version of AngularJS, which is totally reimagined with ES6 and mobile in mind. It's built using Microsoft's TypeScript language. Angular 2.0 is supposed to be released towards the end of calendar year 2015. There are many changes in Angular 2.0 when compared to 1.X but the Angular team is working hard to provide guidance to developers and things will become more clear closer to the release. If you wish to play with Angular 2.0 now, the Angular team has released a website, [Angular.io](#) to show their progress, provide early documentation, and to gather feedback.

Summary

This article provides an overview of AngularJS for ASP.NET developers, and should help developers new to the framework get up to speed with AngularJS quickly.

Related Resources

- [Angular Docs](#)
- [Angular 2 Info](#)

2.8.7 Styling Applications with Less, Sass, and Font Awesome

By [Steve Smith](#)

Users of web applications have increasingly high expectations when it comes to style and overall experience. Modern web applications frequently leverage rich tools and frameworks for defining and managing their look and feel in a consistent manner. Frameworks like [Bootstrap](#) can go a long way toward defining a common set of styles and layout options for the web sites. However, most non-trivial sites also benefit from being able to effectively define and maintain styles and cascading style sheet (CSS) files, as well as having easy access to non-image icons that help make the site's

interface more intuitive. That's where languages and tools that support [Less](#) and [Sass](#), and libraries like [Font Awesome](#), come in.

In this article:

- [CSS Preprocessor Languages](#)
- [Less](#)
- [Sass](#)
- [Less or Sass?](#)
- [Font Awesome](#)

CSS Preprocessor Languages

Languages that are compiled into other languages, in order to improve the experience of working with the underlying language, are referred to as pre-processors. There are two popular pre-processors for CSS: Less and Sass. These pre-processors add features to CSS, such as support for variables and nested rules, which improve the maintainability of large, complex stylesheets. CSS as a language is very basic, lacking support even for something as simple as variables, and this tends to make CSS files repetitive and bloated. Adding real programming language features via preprocessors can help reduce duplication and provide better organization of styling rules. Visual Studio provides built-in support for both Less and Sass, as well as extensions that can further improve the development experience when working with these languages.

As a quick example of how preprocessors can improve readability and maintainability of style information, consider this CSS:

```
.header {
    color: black;
    font-weight: bold;
    font-size: 18px;
    font-family: Helvetica, Arial, sans-serif;
}

.small-header {
    color: black;
    font-weight: bold;
    font-size: 14px;
    font-family: Helvetica, Arial, sans-serif;
}
```

Using Less, this can be rewritten to eliminate all of the duplication, using a mixin (so named because it allows you to “mix in” properties from one class or rule-set into another):

```
.header {
    color: black;
    font-weight: bold;
    font-size: 18px;
    font-family: Helvetica, Arial, sans-serif;
}

.small-header {
    .header;
    font-size: 14px;
}
```

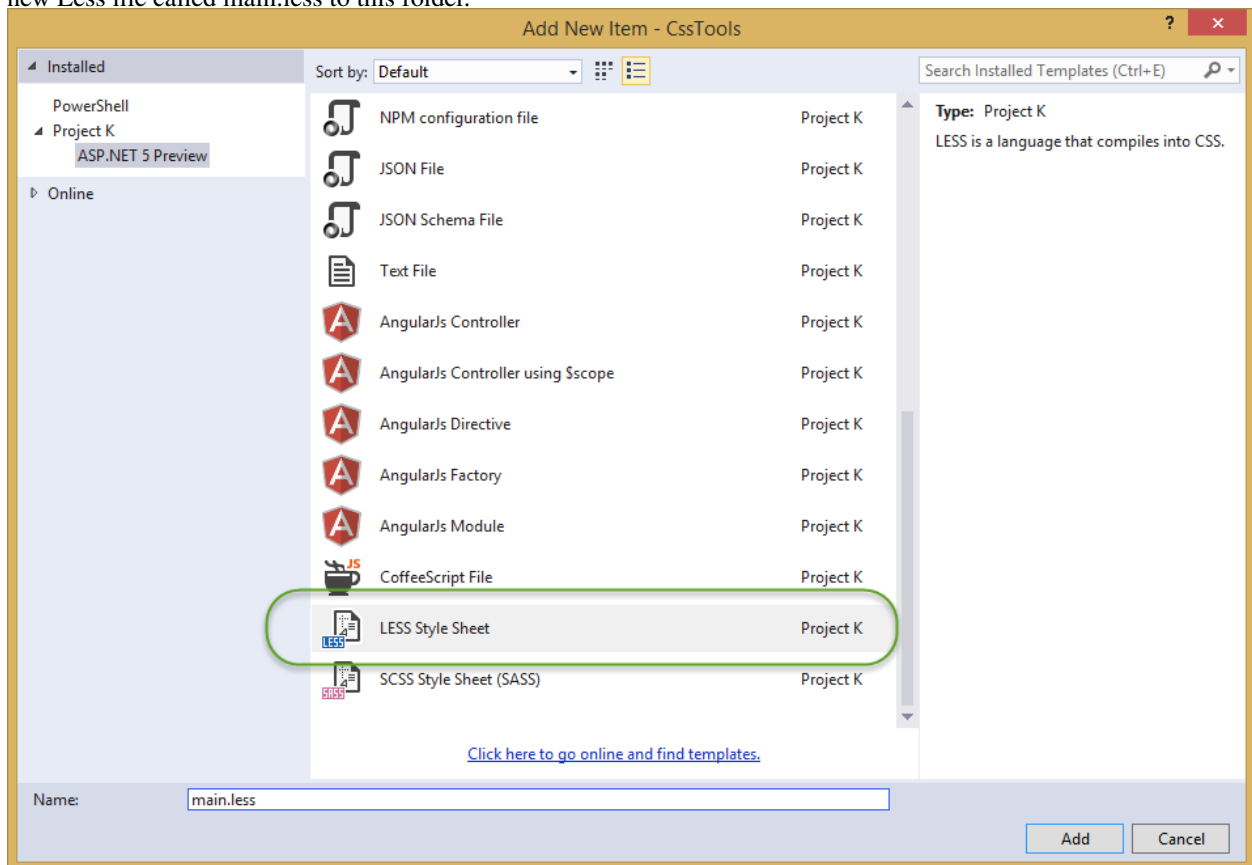
Visual Studio 2015 adds a great deal of built-in support for Less and Sass. You can also add support for earlier versions of Visual Studio by installing the [Web Essentials](#) extension.

Less

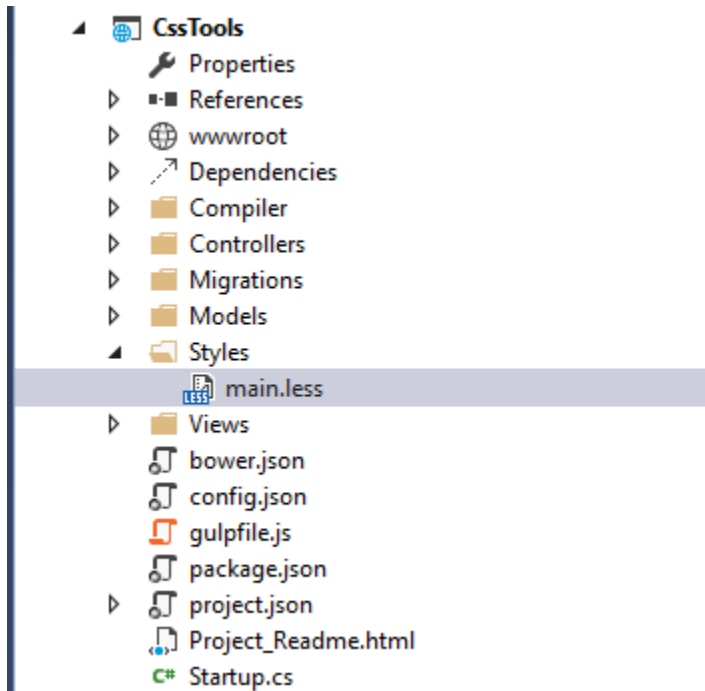
The Less CSS pre-processor runs using Node.js. You can quickly install it using the Node Package Manager (NPM), with:

```
npm install -g less
```

If you're using Visual Studio 2015, you can get started with Less by adding one or more Less files to your project, and then configuring Gulp (or Grunt) to process them at compile-time. Add a Styles folder to your project, and then add a new Less file called `main.less` to this folder.



Once added, your folder structure should look something like this:



Now we can add some basic styling to the file, which will be compiled into CSS and deployed to the `wwwroot` folder by Gulp.

Modify `main.less` to include the following content, which creates a simple color palette from a single base color.

```
@base: #663333;
@background: spin(@base, 180);
@lighter: lighten(spin(@base, 5), 10%);
@lighter2: lighten(spin(@base, 10), 20%);
@darker: darken(spin(@base, -5), 10%);
@darker2: darken(spin(@base, -10), 20%);

body {
    background-color:@background;
}

.baseColor {color:@base}
.bgLight {color:@lighter}
.bgLight2 {color:@lighter2}
.bgDark {color:@darker}
.bgDark2 {color:@darker2}
```

`@base` and the other `@`-prefixed items are variables. Each of them represents a color. Except for `@base`, they are set using color functions: `lighten`, `darken`, and `spin`. `Lighten` and `darken` do pretty much what you would expect; `spin` adjusts the hue of a color by a number of degrees (around the color wheel). The less processor is smart enough to ignore variables that aren't used, so to demonstrate how these variables work, we need to use them somewhere. The classes `.baseColor`, etc. will demonstrate the calculated values of each of the variables in the CSS file that is produced.

Getting Started

If you don't already have one in your project, add a new Gulp configuration file. Make sure `package.json` includes `gulp` in its `devDependencies`, and add "gulp-less":

```
"devDependencies": {
  "gulp": "3.8.11",
  "gulp-less": "3.0.2",
  "rimraf": "2.3.2"
}
```

Save your changes to the package.json file, and you should see that all of the files referenced can be found in the Dependencies folder under NPM. If not, right-click on the NPM folder and select “Restore Packages.”

Now open gulpfile.js. Add a variable at the top to represent less:

```
var gulp = require("gulp"),
    rimraf = require("rimraf"),
    fs = require("fs"),
    less = require("gulp-less");
```

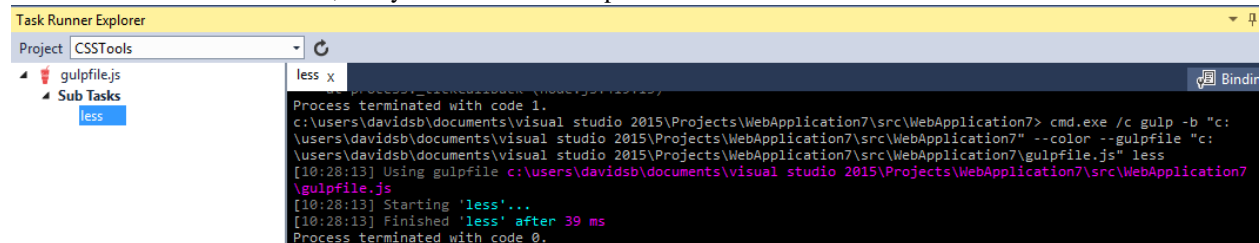
add another variable to allow you to access project properties:

```
var project = require('./project.json');
```

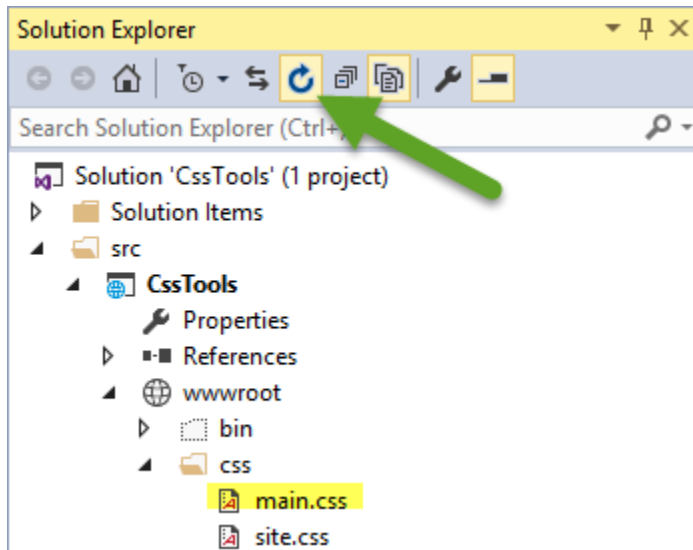
Next, add a task to run less, using the syntax shown here:

```
gulp.task("less", function () {
return gulp.src('Styles/main.less')
    .pipe(less())
    .pipe(gulp.dest(project.webroot + '/css'))
});
```

Open the Task Runner Explorer (view>Other Windows > Task Runner Explorer). Among the tasks, you should see a new task named less. Run it, and you should have output similar to what is shown here:



Now refresh your Solution Explorer and inspect the contents of the wwwroot/css folder. You should find a new file, main.css, there:



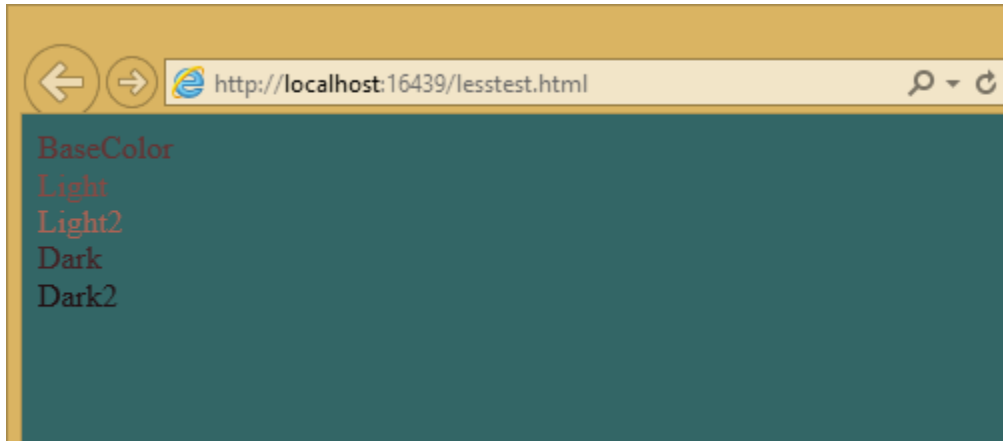
Open main.css and you should see something like the following:

```
body {
    background-color: #336666;
}
.baseColor {
    color: #663333;
}
.bgLight {
    color: #884a44;
}
.bgLight2 {
    color: #aa6355;
}
.bgDark {
    color: #442225;
}
.bgDark2 {
    color: #221114;
}
```

Add a simple HTML page to the wwwroot folder and reference main.css to see the color palette in action.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link href="css/main.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        <div class="baseColor">BaseColor</div>
        <div class="bgLight">Light</div>
        <div class="bgLight2">Light2</div>
        <div class="bgDark">Dark</div>
        <div class="bgDark2">Dark2</div>
    </div>
</body>
</html>
```

You can see that the 180 degree spin on @base used to produce @background resulted in the color wheel opposing color of @base:



Less also provides support for nested rules, as well as nested media queries. For example, defining nested hierarchies like menus can result in verbose CSS rules like these:

```
nav {  
    height: 40px;  
    width: 100%;  
}  
nav li {  
    height: 38px;  
    width: 100px;  
}  
nav li a:link {  
    color: #000;  
    text-decoration: none;  
}  
nav li a:visited {  
    text-decoration: none;  
    color: #CC3333;  
}  
nav li a:hover {  
    text-decoration: underline;  
    font-weight: bold;  
}  
nav li a:active {  
    text-decoration: underline;  
}
```

Ideally all of the related style rules will be placed together within the CSS file, but in practice there is nothing enforcing this rule except convention and perhaps block comments.

Defining these same rules using Less looks like this:

```
nav {  
    height: 40px;  
    width: 100%;  
    li {  
        height: 38px;  
        width: 100px;  
        a {  
            color: #000;  
            &:link { text-decoration:none}
```

```

&:visited { color: #CC3333; text-decoration:none}
&:hover { text-decoration:underline; font-weight:bold}
&:active {text-decoration:underline}
    }
}
}

```

Note that in this case, all of the subordinate elements of `nav` are contained within its scope. There is no longer any repetition of parent elements (`nav`, `li`, `a`), and the total line count has dropped as well (though some of that is a result of putting values on the same lines in the second example). It can be very helpful, organizationally, to see all of the rules for a given UI element within an explicitly bounded scope, in this case set off from the rest of the file by curly braces.

The `&` syntax is a Less selector feature, with `&` representing the current selector parent. So, within the `a {...}` block, `&` represents an `a` tag, and thus `&:link` is equivalent to `a:link`.

Media queries, extremely useful in creating responsive designs, can also contribute heavily to repetition and complexity in CSS. Less allows media queries to be nested within classes, so that the entire class definition doesn't need to be repeated within different top-level `@media` elements. For example, this CSS for a responsive menu:

```

.navigation {
    margin-top: 30%;
    width: 100%;
}
@media screen and (min-width: 40em) {
    .navigation {
        margin: 0;
    }
}
@media screen and (min-width: 62em) {
    .navigation {
        width: 960px;
        margin: 0;
    }
}

```

This can be better defined in Less as:

```

.navigation {
    margin-top: 30%;
    width: 100%;
    @media screen and (min-width: 40em) {
        margin: 0;
    }
    @media screen and (min-width: 62em) {
        width: 960px;
        margin: 0;
    }
}

```

Another feature of Less that we have already seen is its support for mathematical operations, allowing style attributes to be constructed from pre-defined variables. This makes updating related styles much easier, since the base variable can be modified and all dependent values change automatically.

CSS files, especially for large sites (and especially if media queries are being used), tend to get quite large over time, making working with them unwieldy. Less files can be defined separately, then pulled together using `@import` directives. Less can also be used to import individual CSS files, as well, if desired.

Mixins can accept parameters, and Less supports conditional logic in the form of mixin guards, which provide a declarative way to define when certain mixins take effect. A common use for mixin guards is to adjust colors based

on how light or dark the source color is. Given a mixin that accepts a parameter for color, a mixin guard can be used to modify the mixin based on that color:

```
.box (@color) when (lightness(@color) >= 50%) {  
    background-color: #000;  
}  
.box (@color) when (lightness(@color) < 50%) {  
    background-color: #FFF;  
}  
.box (@color) {  
    color: @color;  
}  
  
.feature {  
    .box (@base);  
}
```

Given our current @base value of #663333, this Less script will produce the following CSS:

```
.feature {  
    background-color: #FFF;  
    color: #663333;  
}
```

Less provides a number of additional features, but this should give you some idea of the power of this preprocessing language.

Sass

Sass is similar to Less, providing support for many of the same features, but with slightly different syntax. It is built using Ruby, rather than JavaScript, and so has different setup requirements. The original Sass language did not use curly braces or semicolons, but instead defined scope using white space and indentation. In version 3 of Sass, a new syntax was introduced, **SCSS** (“Sassy CSS”). SCSS is similar to CSS in that it ignores indentation levels and whitespace, and instead uses semicolons and curly braces.

To install Sass, typically you would first install Ruby (pre-installed on Mac), and then run:

```
gem install sass-lang
```

However, assuming you’re running Visual Studio, you can get started with Sass in much the same way as you would with Less. Open package.json and add the “gulp-sass” package to devDependencies:

```
"devDependencies": {  
    "gulp": "3.8.11",  
    "gulp-less": "3.0.2",  
    "gulp-sass": "1.3.3",  
    "rimraf": "2.3.2"  
}
```

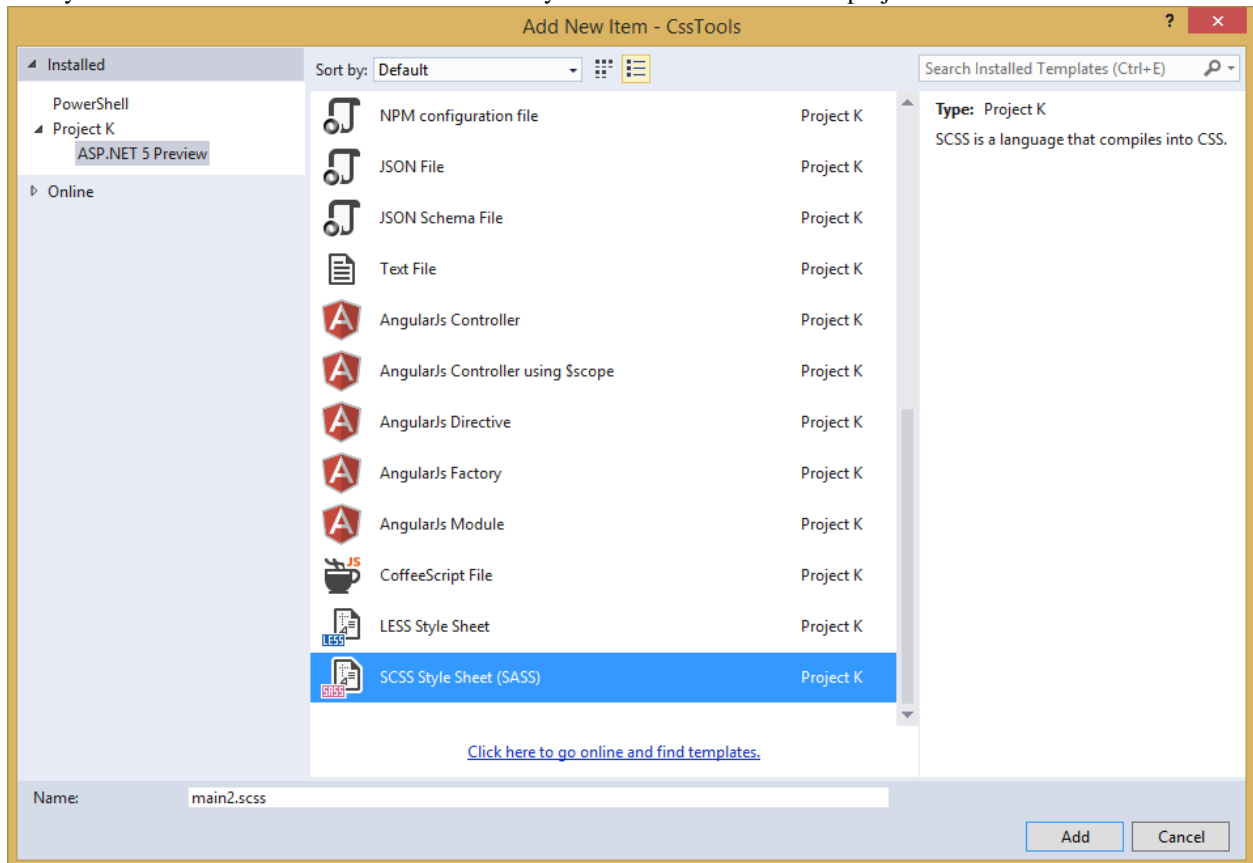
Next, modify gulpfile.js to add a sass variable and a task to compile your Sass files and place the results in the wwwroot folder:

```
var gulp = require("gulp"),  
    rimraf = require("rimraf"),  
    fs = require("fs"),  
    less = require("gulp-less"),  
    sass = require("gulp-sass");  
  
// other content removed
```



```
gulp.task("sass", function () {
    return gulp.src('Styles/main2.scss')
        .pipe(sass())
        .pipe(gulp.dest(project.webroot + '/css'))
});
```

Now you can add the Sass file main2.scss to the Styles folder in the root of the project:



Open main2.scss and add the following:

```
$base: #CC0000;
body {
    background-color: $base;
}
```

Save all of your files. Now in Task Runner Explorer, you should see a sass task. Run it, refresh solution explorer, and look in the /wwwroot/css folder. There should be a main2.css file, with these contents:

```
body {
    background-color: #CC0000; }
```

Sass supports nesting in much the same way that Less does, providing similar benefits. Files can be split up by function and included using the `@import` directive:

```
@import 'anotherfile';
```

Sass supports mixins as well, using the `@mixin` keyword to define them and `@include` to include them, as in this example from sass-lang.com:

```
@mixin border-radius($radius) {  
  -webkit-border-radius: $radius;  
  -moz-border-radius: $radius;  
  -ms-border-radius: $radius;  
  border-radius: $radius;  
}  
  
.box { @include border-radius(10px); }
```

In addition to mixins, Sass also supports the concept of inheritance, allowing one class to extend another. It's conceptually similar to a mixin, but results in less CSS code. It's accomplished using the `@extend` keyword. First, let's see how we might use mixins, and the resulting CSS code. Add the following to your `main2.scss` file:

```
@mixin alert {  
  border: 1px solid black;  
  padding: 5px;  
  color: #333333;  
}  
  
.success {  
  @include alert;  
  border-color: green;  
}  
  
.error {  
  @include alert;  
  color: red;  
  border-color: red;  
  font-weight: bold;  
}
```

Examine the output in `main2.css` after running the sass task in Task Runner Explorer:

```
.success {  
  border: 1px solid black;  
  padding: 5px;  
  color: #333333;  
  border-color: green;  
}  
  
.error {  
  border: 1px solid black;  
  padding: 5px;  
  color: #333333;  
  color: red;  
  border-color: red;  
  font-weight: bold;  
}
```

Notice that all of the common properties of the `alert` mixin are repeated in each class. The mixin did a good job of helping use eliminate duplication at development time, but it's still creating CSS with a lot of duplication in it, resulting in larger than necessary CSS files - a potential performance issue. It would be great if we could follow the [Don't Repeat Yourself \(DRY\) Principle](#) at both development time and runtime.

Now replace the `alert` mixin with a `.alert` class, and change `@include` to `@extend` (remembering to extend `.alert`, not `alert`):

```
.alert {  
  border: 1px solid black;
```

```

        padding: 5px;
        color: #333333;
    }

    .success {
        @extend .alert;
        border-color: green;
    }

    .error {
        @extend .alert;
        color: red;
        border-color: red;
        font-weight: bold;
    }

```

Run Sass once more, and examine the resulting CSS:

```

.alert, .success, .error {
    border: 1px solid black;
    padding: 5px;
    color: #333333; }

.success {
    border-color: green; }

.error {
    color: red;
    border-color: red;
    font-weight: bold; }

```

Now the properties are defined only as many times as needed, and better CSS is generated.

Sass also includes functions and conditional logic operations, similar to Less. In fact, the two languages' capabilities are very similar.

Less or Sass?

There is still no consensus as to whether it's generally better to use Less or Sass (or even whether to prefer the original Sass or the newer SCSS syntax within Sass). A recent poll conducted on twitter of mostly ASP.NET developers found that the majority preferred to use Less, by about a 2-to-1 margin. Probably the most important decision is to **use one of these tools**, as opposed to just hand-coding your CSS files. Once you've made that decision, both Less and Sass are good choices.

Font Awesome

In addition to CSS pre-compilers, another great resource for styling modern web applications is Font Awesome. Font Awesome is a toolkit that provides over 500 scalable vector icons that can be freely used in your web applications. It was originally designed to work with Bootstrap, but has no dependency on that framework, or on any JavaScript libraries.

The easiest way to get started with Font Awesome is to add a reference to it, using its public content delivery network (CDN) location:

```

<link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">

```

Of course, you can also quickly add it to your Visual Studio 2015 project by adding it to the “dependencies” in bower.json:

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.0.0",
    "jquery": "1.10.2",
    "jquery-validation": "1.11.1",
    "jquery-validation-unobtrusive": "3.2.2",
    "hammer.js": "2.0.4",
    "bootstrap-touch-carousel": "0.8.0",
    "Font-Awesome": "4.3.0"
  }
}
```

Then, to get the stylesheet added to the wwwroot folder, modify gulpfile.js as follows:

```
gulp.task("copy", ["clean"], function () {
  var bower = {
    "angular": "angular/angular*.js,map",
    "bootstrap": "bootstrap/dist/**/*.{js,map,css,ttf,svg,woff,eot}",
    "bootstrap-touch-carousel": "bootstrap-touch-carousel/dist/**/*.{js,css}",
    "hammer.js": "hammer.js/hammer*.js,map",
    "jquery": "jquery/jquery*.js,map",
    "jquery-validation": "jquery-validation/jquery.validate.js",
    "jquery-validation-unobtrusive": "jquery-validation-unobtrusive/jquery.validate.unobtrusive.js",
    "font-awesome": "Font-Awesome/**/*.{css,otf,eot,svg,ttf,woff,wof2}"
  };

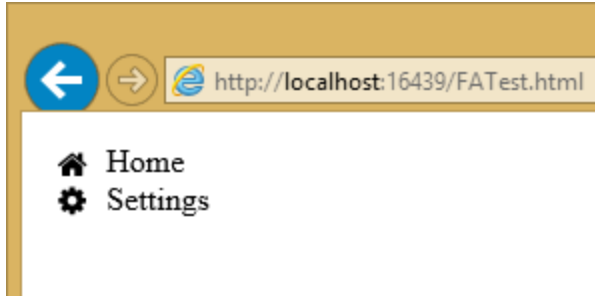
  for (var destinationDir in bower) {
    gulp.src(paths.bower + bower[destinationDir])
      .pipe(gulp.dest(paths.lib + destinationDir));
  }
});
```

Once this is in place (and saved), running the ‘copy’ task in Task Runner Explorer should copy the font awesome fonts and css files to /lib/font-awesome.

Once you have a reference to it on a page, you can add icons to your application by simply applying Font Awesome classes, typically prefixed with “fa-”, to your inline HTML elements (such as or <i>). As a very simple example, you can add icons to simple lists and menus using code like this:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <link href="lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</head>
<body>
  <ul class="fa-ul">
    <li><i class="fa fa-li fa-home"></i> Home</li>
    <li><i class="fa fa-li fa-cog"></i> Settings</li>
  </ul>
</body>
</html>
```

This produces the following in the browser - note the icon beside each item:



You can view a complete list of the available icons here:

<http://fortawesome.github.io/Font-Awesome/icons/>

Summary

Modern web applications increasingly demand responsive, fluid designs that are clean, intuitive, and easy to use from a variety of devices. Managing the complexity of the CSS stylesheets required to achieve these goals is best done using a pre-processor like Less or Sass. In addition, toolkits like Font Awesome quickly provide well-known icons to textual navigation menus and buttons, improving the overall user experience of your application.

2.8.8 Bundling and Minification

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.8.9 Working with a Content Delivery Network (CDN)

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.8.10 Responsive Design for the Mobile Web

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.8.11 Introducing TypeScript

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.8.12 Building Projects with Yeoman

By [Noel Rice](#)

Yeoman generates complete, running projects for a given set of client-tools. You may be overwhelmed by “feature-shock” for the latest glut of client-tools (NPM, Gulp, Angular, etc.) or have simply run out of time to work through each feature set, much less the details, of every client tool. Yeoman generates everything you need to get over your first hurdle, the “hello world” that demonstrates that a group of technologies function together.

Yeoman is an open-source tool that works like a Visual Studio project template, but targets a wide developer audience that ranges from Ruby, browser extensions, PhoneGap, FaceBook React, jQuery-Mobile, and Microsoft technologies like ASP.NET.

Yeoman is *opinionated*, that is, it prescribes tools and best practices for your target technologies so you don’t have to decide “Use version X or Y?” or “What directory structure should I use?” Yeoman gets you started with a known-good project that runs.

The Yeoman command line tool **yo** works alongside a Yeoman generator. Generators define the technologies that go into a project. Here are a few sample **generators**:

- **AngularJS Generator** creates a starting point for a new single-page Angular application.
- **jQuery Generator** creates the shell code for a jQuery plug-in.
- **Chrome App Generator** generates everything you need to create an extension for the Chrome browser.
- The **Flux-React generator** creates an application based on Facebook’s Flux/React architecture.
- **ASP.NET generator** creates ASP.NET 5, DNX projects.
- You can even **generate your own Yeoman generator**.

In this article:

- *Getting Started*
- *Building and Running from Visual Studio*
- *Client-Side Build Support*
- *Restoring, Building and Hosting from the Command Line*
- *Adding to Your Project with Sub Generators*

Getting Started

The **ASP.NET generator** creates ASP.NET 5, DNX projects that can be loaded into Visual Studio 2015 or run from the command line. The generator creates the following project types:

- **Empty Application:** An empty Web application with minimal dependencies.
- **Console Application:** A DNX-based console application.
- **Web Application:** A complete MVC web application with a simple home page and examples for managing accounts and login.
- **Web API Application:** A Web API built with MVC.
- **Nancy ASP.NET Application:** A light-weight HTTP service with one module built using Nancy.
- **Class Library:** A DNX-based class library.

This walk-through demonstrates how to use Yeoman to generate an ASP.NET 5 web application.

1. Follow the instructions on <http://yeoman.io/learning/> to install **yo** and other required tools.
2. On the command line, install the ASP.NET generator:

```
npm install -g generator-aspnet
```

Note: The `-g` flag installs the generator globally so that you can use it from any path on your system.

3. Make a new directory where your project will be generated:

```
mkdir c:\MyYo
```

4. On the command line, make the new directory the current directory.

```
cd c:\MyYo
```

5. Run the `yo` command and pass the name of the generator.

```
yo aspnet
```

6. The `aspnet` generator displays a menu. Select the **Web Application** and press Enter.

```
C:\myyo>yo aspnet

  --(o)--
  |  U  |
  |  A  |
  |  ~  |
  |  T  |
  |  Y  |

Welcome to the
marvellous ASP.NET 5
generator!

? What type of application do you want to create?
Empty Application
Console Application
> Web Application
Web API Application
Nancy ASP.NET Application
Class Library
```

7. Provide an application name “MyWebApp” and press Enter.

```
? What type of application do you want to create? Web Application
? What's the name of your ASP.NET application? (WebApplication) MyWebApp_
```

Yeoman will create the project and supporting files.

```
create MyWebApp\wwwroot\lib\bootstrap\js\bootstrap.min.js
create MyWebApp\wwwroot\lib\hammer.js\hammer.js
create MyWebApp\wwwroot\lib\hammer.js\hammer.min.js
create MyWebApp\wwwroot\lib\hammer.js\hammer.min.map
create MyWebApp\wwwroot\lib\jquery-validation-unobtrusive\jquery.validate
create MyWebApp\wwwroot\lib\jquery-validation\jquery.validate.js
create MyWebApp\wwwroot\lib\jquery\jquery-migrate.js
create MyWebApp\wwwroot\lib\jquery\jquery-migrate.min.js
create MyWebApp\wwwroot\lib\jquery\jquery.js
create MyWebApp\wwwroot\lib\jquery\jquery.min.js
create MyWebApp\wwwroot\lib\jquery\jquery.min.map

Your project is now created, you can use the following commands to get going
  dnu restore
  dnu build
  dnx . run for console projects
  dnx . kestrel or dnx . web for web projects

PS C:\MyYo>
```

Client-Side Build Support

The ASP.NET generator creates support files to configure client-side build tools. Grunt or Gulp task runners files are added to your project to automate build tasks for Web Application projects. By default, aspnet-generator creates **gruntfile.js** to run tasks. Running the generator with the **--gulp** argument generates **gulpfile.js** instead.

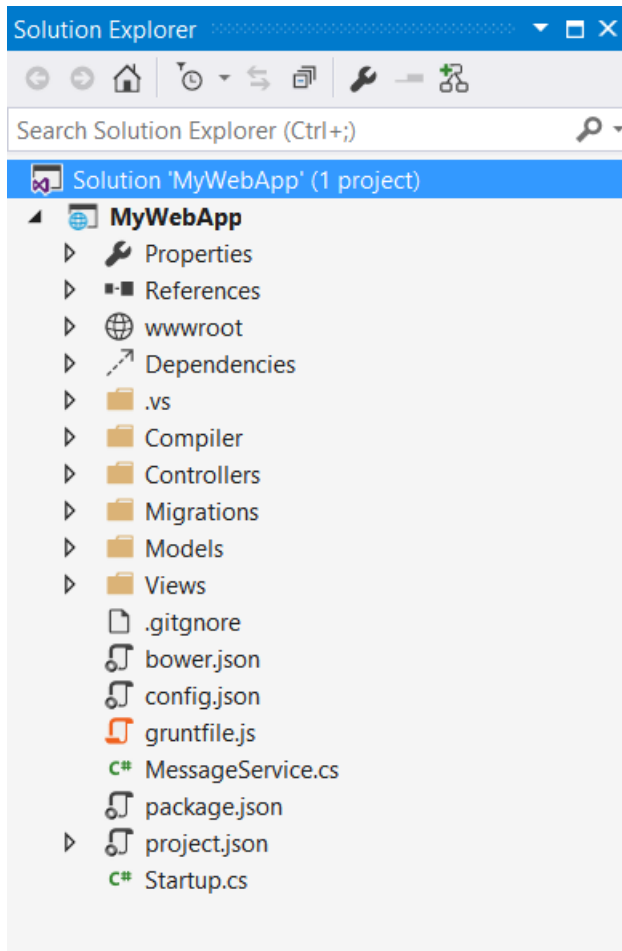
```
yo aspnet --gulp
```

The generator also configures **package.json** to load Grunt or Gulp and adds bower.json to restore client-side packages using the Bower client-side package manager.

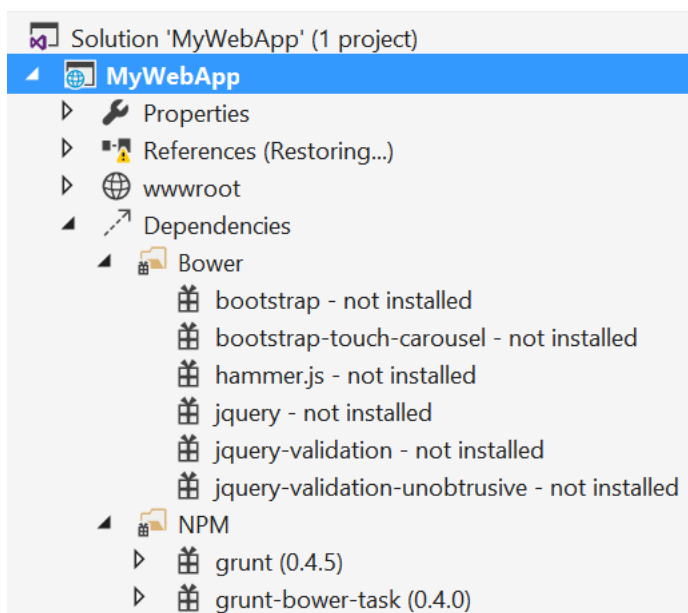
Building and Running from Visual Studio

You can load your generated ASP.NET 5 web project directly into Visual Studio 2015, then build and run your project from there.

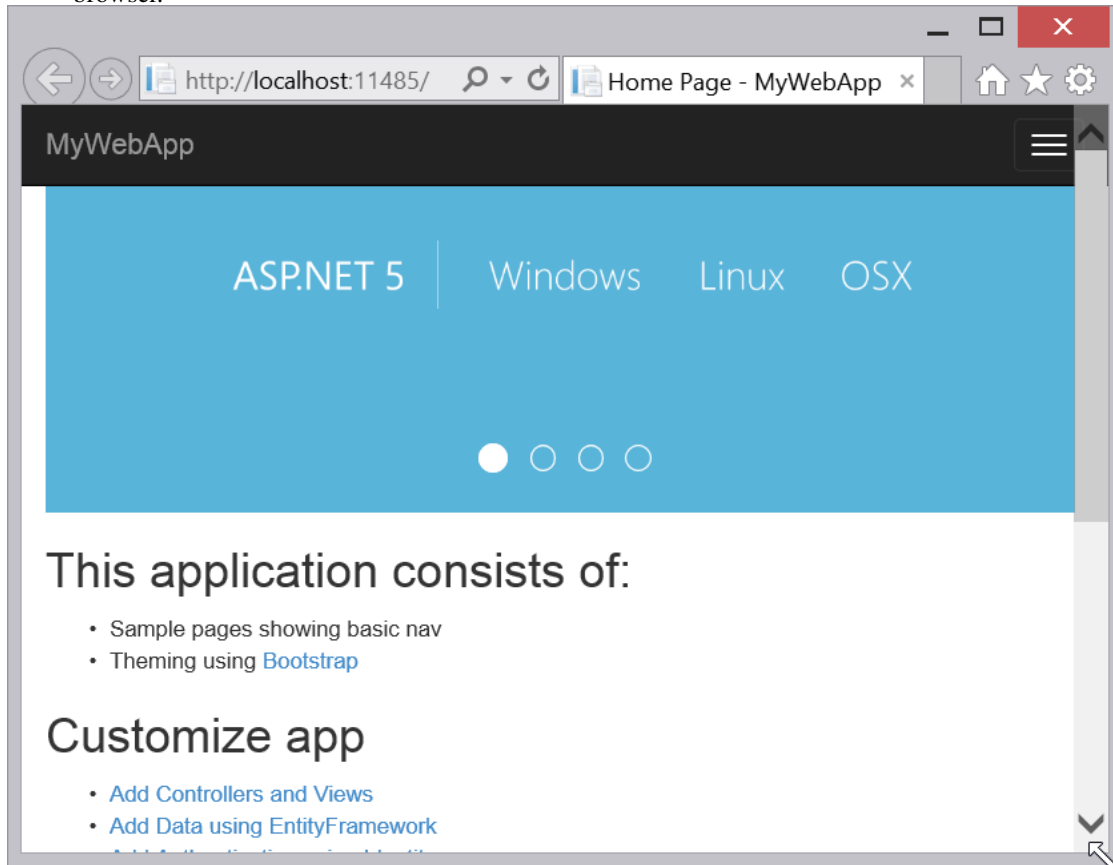
1. Open Visual Studio 2015. From the File menu select *Open* → *Project/Solution*.
2. In the Open Project dialog, navigate to the `project.json` file, select it and click the **Open** button. In the Solution Explorer, the project should look something like the screenshot below.



Note: Yeoman creates a MVC web application complete with server and client side build support. Server-side dependencies are listed under the **References** node, and client-side dependencies in the **Dependencies** node/ of Solution Explorer. Dependencies are restored automatically when you load this project.



- When all the dependencies are restored, press **F5** to run the project. The default home page displays in the browser.



Restoring, Building and Hosting from the Command Line

You can prepare and host your web application using commands **dnu** (Microsoft .NET Development Utility) and **dnx** (Microsoft .NET Execution Environment).

Note: For more information on DNX see [DNX Overview](#)

- From the command line, change the current directory to the folder containing the project (that is, the folder that contains the *project.json* file).

```
cd c:\MyYo\MyWebApp
```

- From the command line, restore the project's package dependencies.

```
dnu restore
```

- Also from the command line, build the project assemblies.

```
dnu build
```

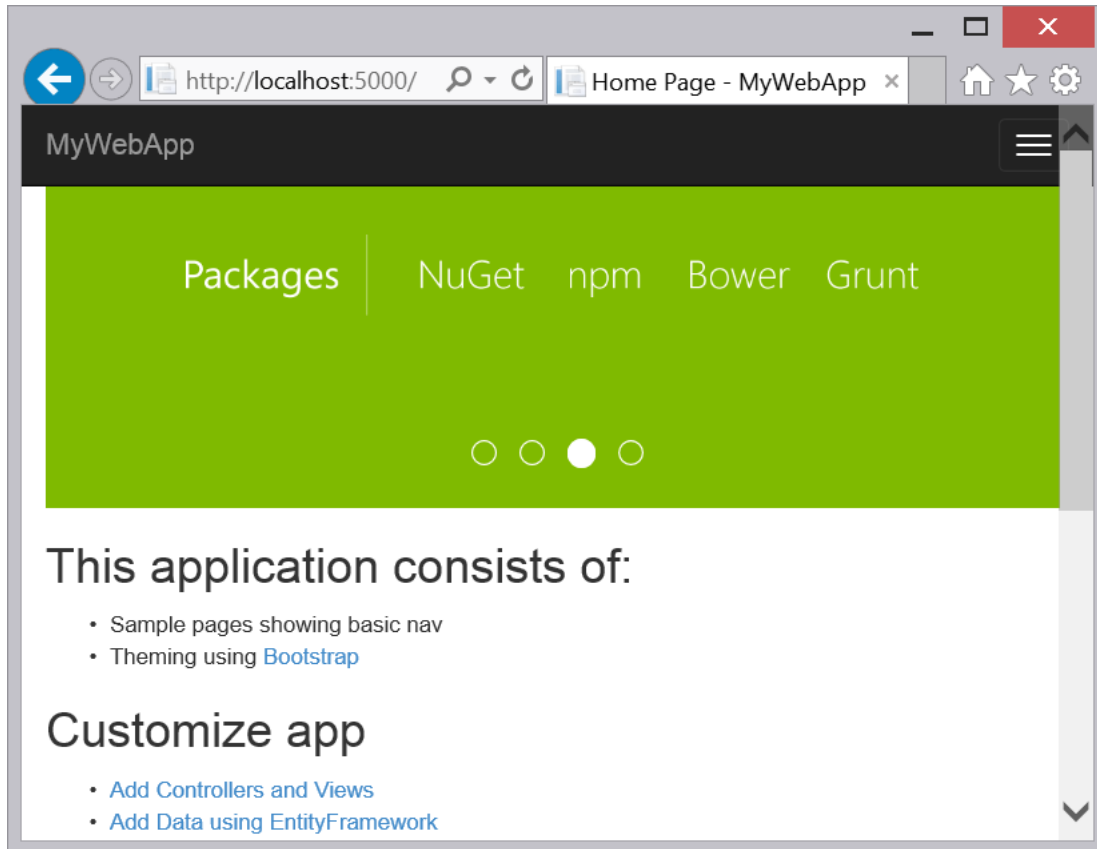
- To run the development web server run **dnx** command.

```
dnx . web
```

The web server will listen on port 5000. The URL and port are defined in `project.json` in the **commands** section.

```
"commands": {
  "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5000",
  "kestrel": "Microsoft.AspNet.Hosting --server Kestrel --server.urls http://localhost:5001",
  "gen": "Microsoft.Framework.CodeGeneration",
  "ef": "EntityFramework.Commands"
},
```

5. Open a web browser and navigate to <http://localhost:5000>.



Note: You can also run the cross-platform Kestrel development server using the `dnx . kestrel` command. By default, Kestrel listens on port 5001 as defined in the **project.json, commands** section.

Adding to Your Project with Sub Generators

You can add new generated files using Yeoman even after the project is created. Use [sub generators](#) to add any of the file types that make up your project. For example, to add a new class to your project, enter the `yo aspnet:Class` command followed by the name of the class. Run the command from the directory where the file should be created.

```
yo aspnet:Class Person
```

The command creates `Person.cs`.

```
using System;

namespace MyNamespace
{
    public class Person
    {
    }
}
```

Summary

Yeoman generates complete running projects for a wide range of technology combinations. The generated files can be loaded into Visual Studio. Task Runner Explorer and other Visual Studio tooling help configure and automate unfamiliar technologies in a familiar environment.

2.9 Mobile

2.9.1 Responsive Design for the Mobile Web

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

Also check out the following topics on building mobile views and native mobile backends with ASP.NET MVC:

- [Building Mobile Specific Views](#)
- [Creating Backend Services for Native Mobile Applications](#)

2.10 Security

2.10.1 Enabling authentication using external providers

By [Pranav Rastogi](#)

This tutorial shows you how to build an ASP.NET 5 Web application that enables users to log in using OAuth 2.0 with credentials from an external authentication provider, such as Facebook, Twitter, LinkedIn, Microsoft, or Google. For simplicity, this tutorial focuses on working with credentials from Facebook and Google.

Enabling these credentials in your web sites provides a significant advantage because millions of users already have accounts with these external providers. These users may be more inclined to sign up for your site if they do not have to create and remember a new set of credentials.

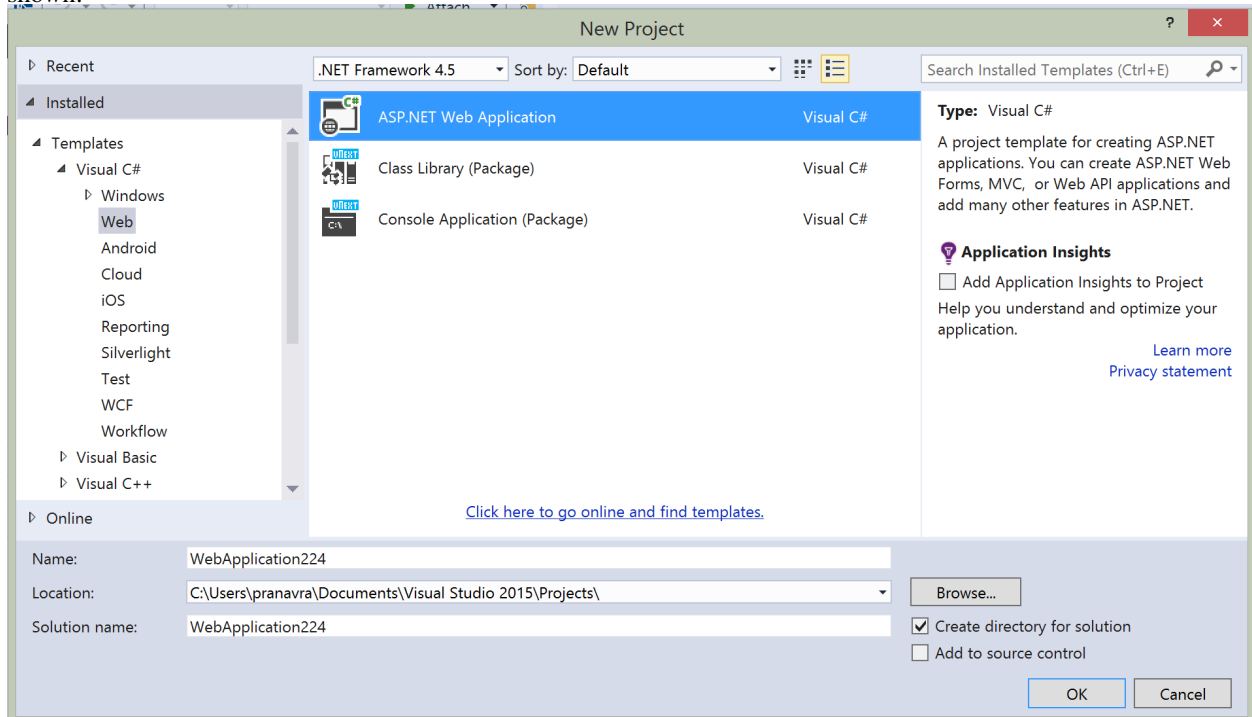
In this article:

- *Create a New ASP.NET 5 Project*
- *Running the Application*
- *Creating the app in Facebook*
- *Use SecretManager to store Facebook AppId and AppSecret*
- *Enable Facebook middleware*

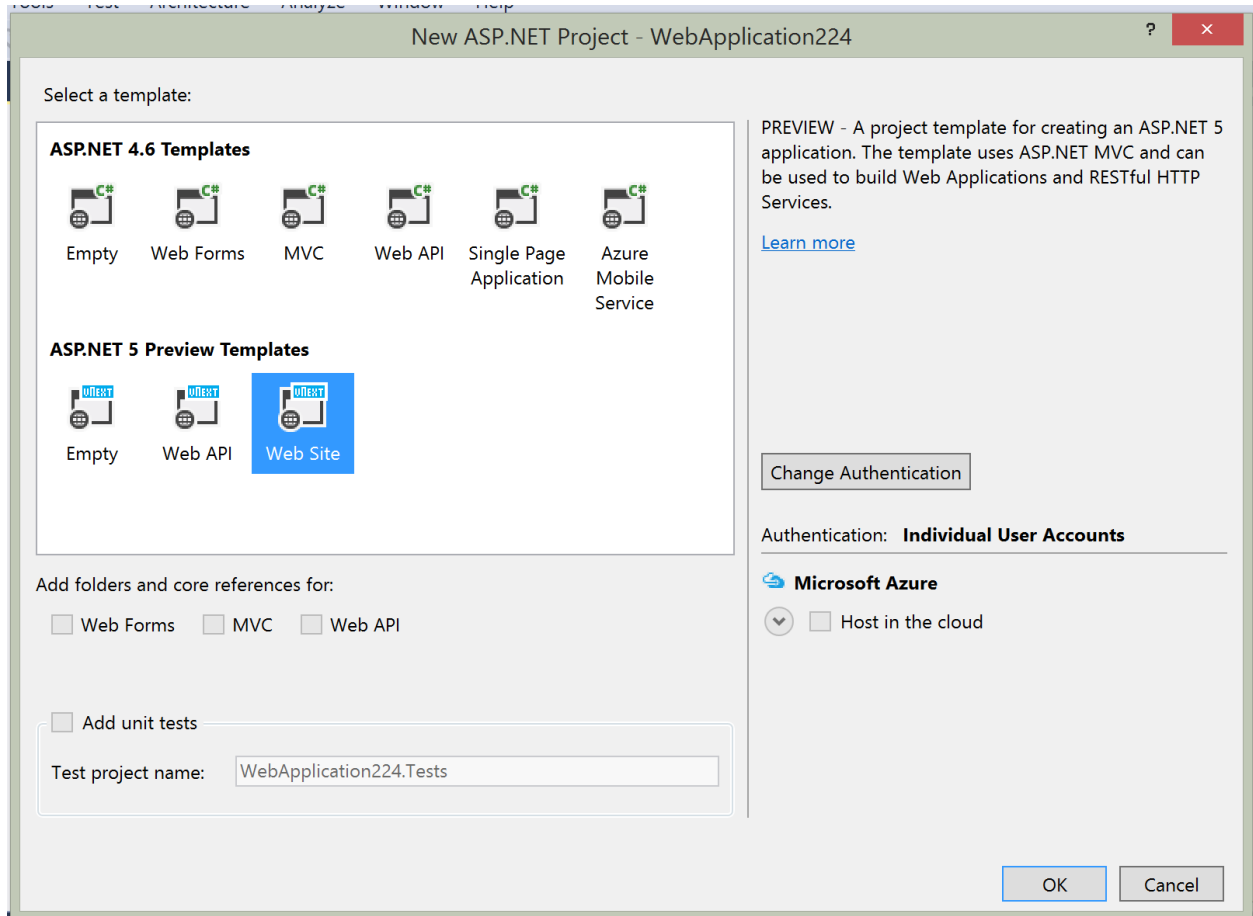
- *Login with Facebook*
- *Optionally set password*
- *Next steps*
- *Summary*

Create a New ASP.NET 5 Project

To get started, open Visual Studio 2015. Next, create a New Project (from the Start Page, or via File - New - Project). On the left part of the New Project window, make sure the Visual C# templates are open and “Web” is selected, as shown:



Next you should see another dialog, the New ASP.NET Project window:

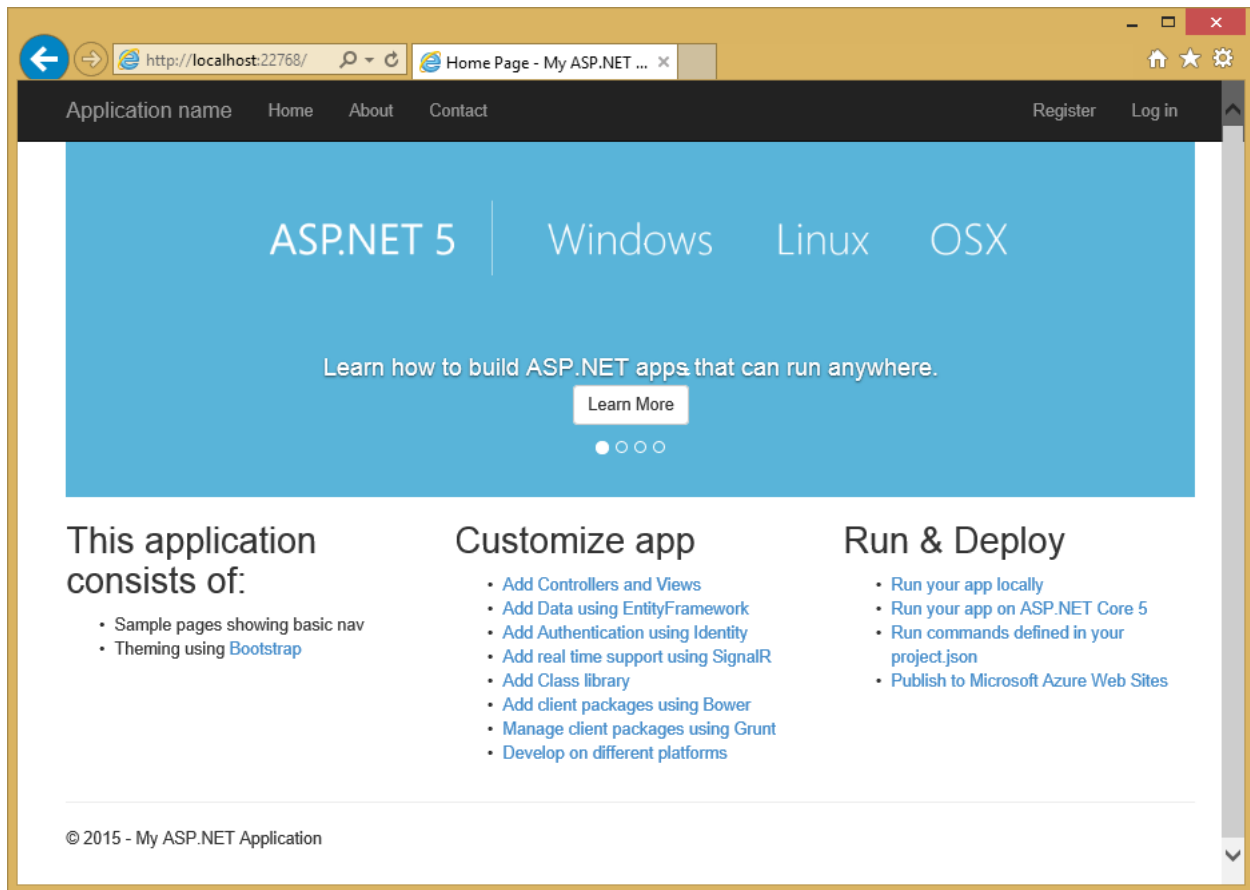


Select the ASP.NET 5 Web site template from the set of ASP.NET 5 templates. Make sure you have Individual Authentication selected for this template. After selecting, click OK.

At this point, the project is created. It may take a few moments to load, and you may notice Visual Studio's status bar indicates that Visual Studio is downloading some resources as part of this process. Visual Studio ensures some required files are pulled into the project when a solution is opened (or a new project is created), and other files may be pulled in at compile time.

Running the Application

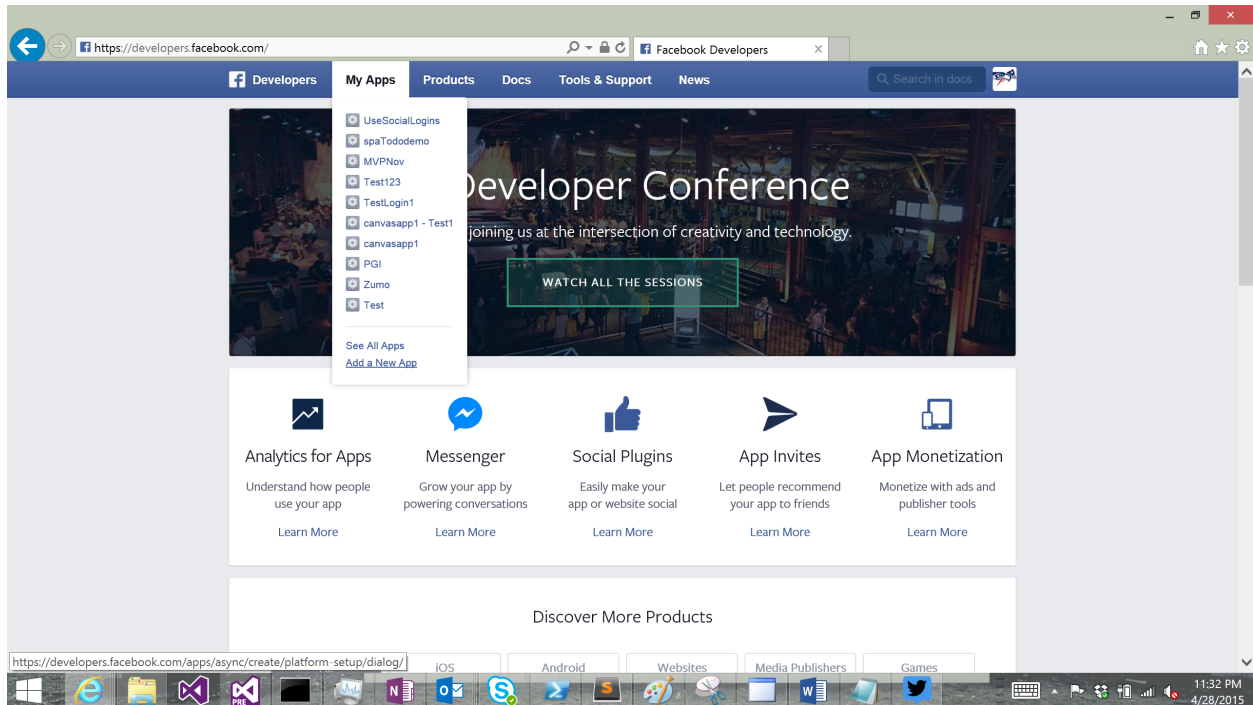
Run the application and after a quick build step, you should see it open in your web browser.



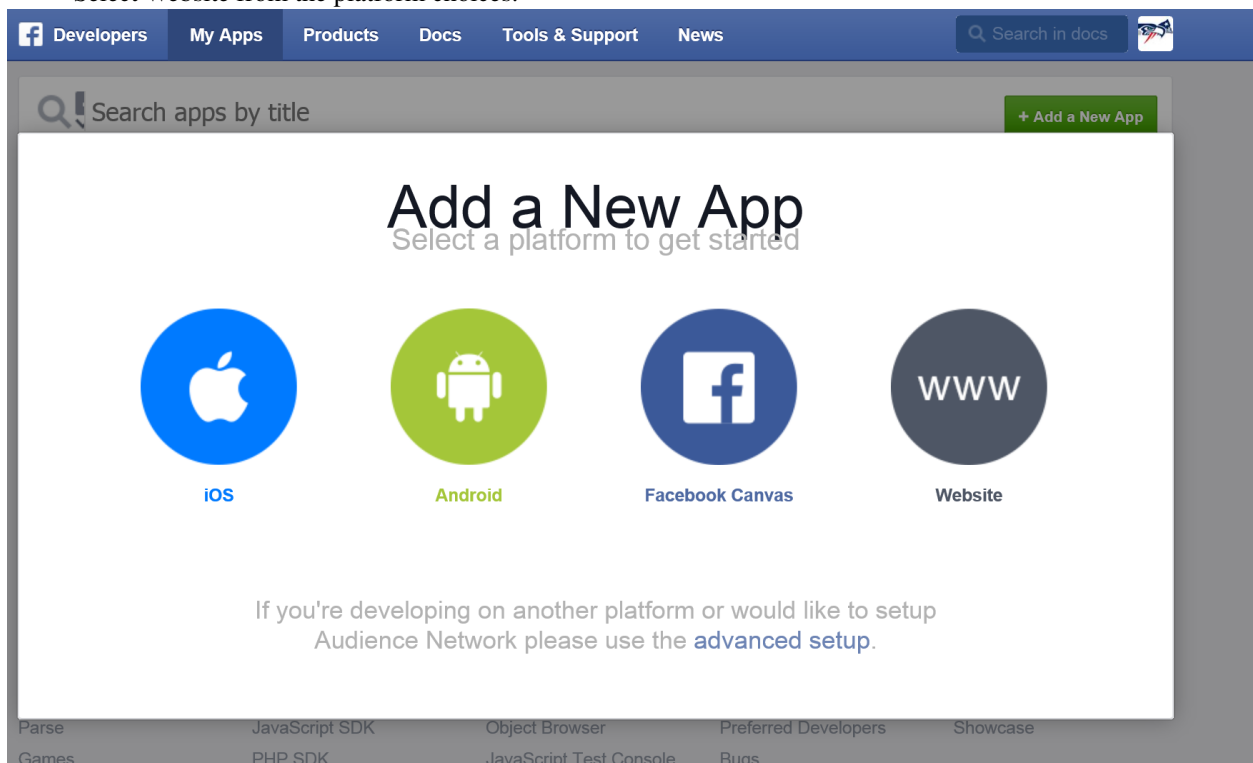
Creating the app in Facebook

For Facebook OAuth2 authentication, you need to copy to your project some settings from an application that you create in Facebook.

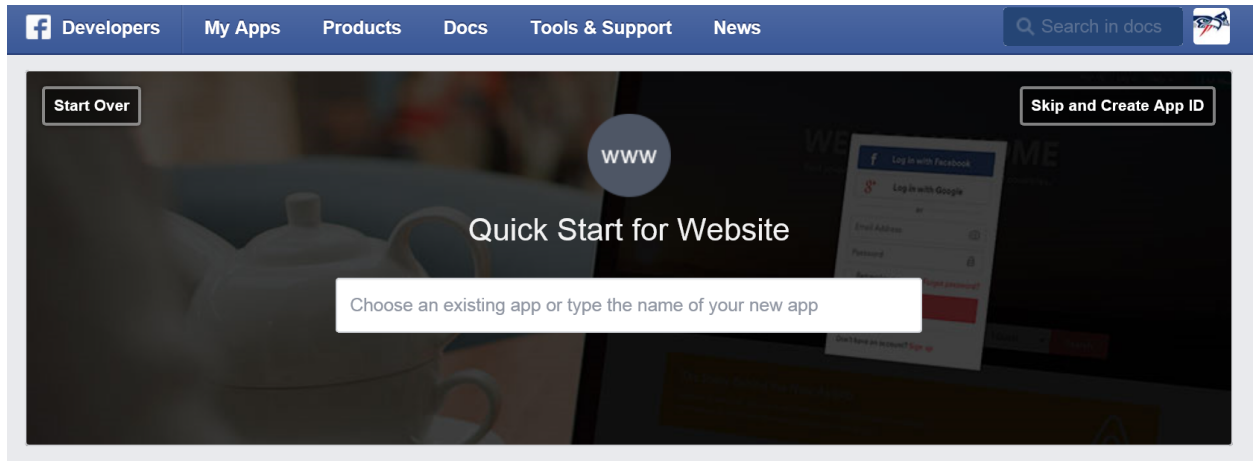
- In your browser, navigate to <https://developers.facebook.com/apps> and log in by entering your Facebook credentials.
- If you aren't already registered as a Facebook developer, click Register as a Developer and follow the directions to register.
- On the Apps tab, click Create New App.



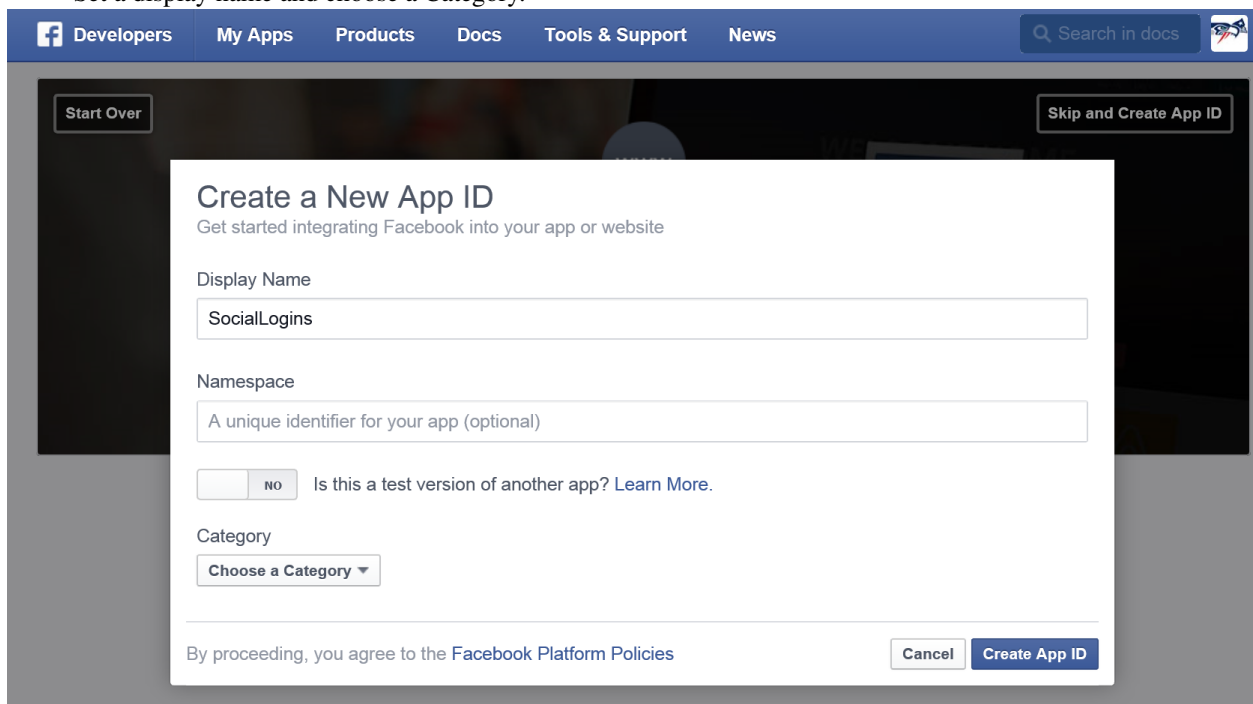
- Select Website from the platform choices.



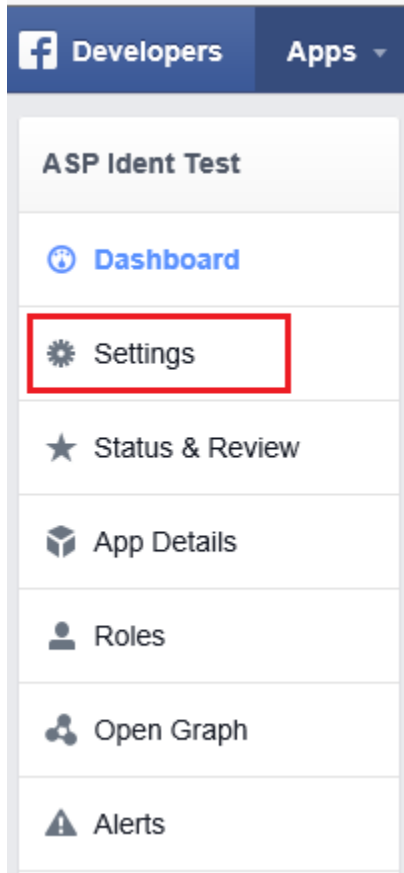
- Click **Skip and Create App ID**



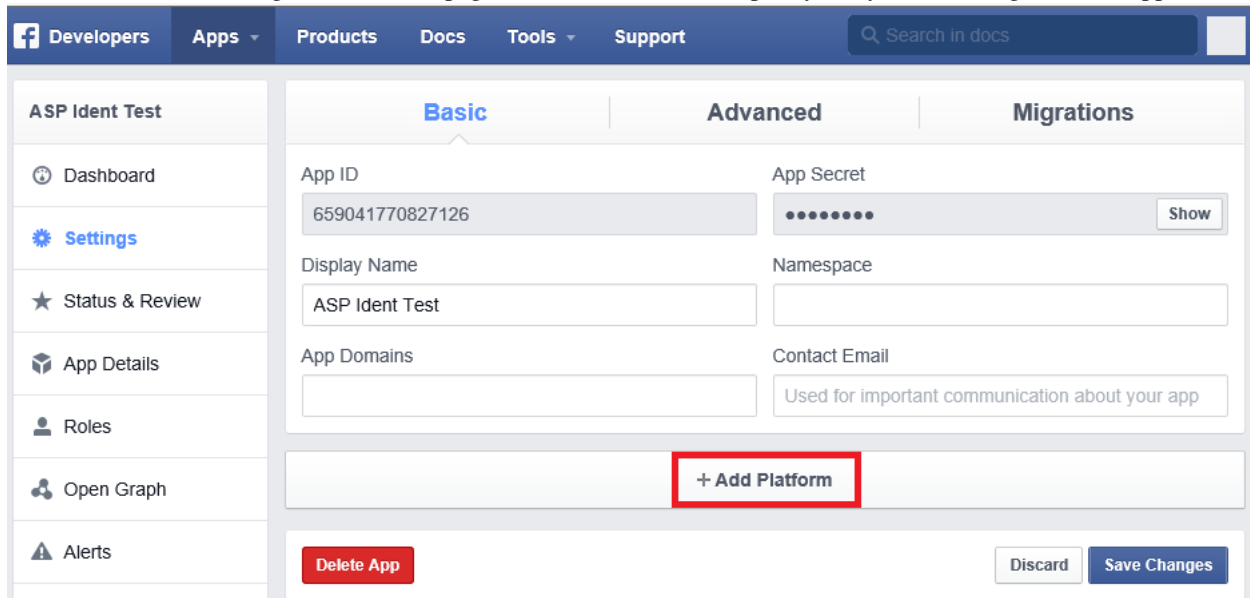
- Set a display name and choose a Category.



- Select **Settings** from the left menu bar.



- On the **Basic** settings section of the page select Add Platform to specify that you are adding a website application.



- Select Website from the platform choices.

Select Platform



App on Facebook



Website



iOS



Android



Windows App



Page Tab



Xbox



Play Station

Cancel

- Add your Site URL (<http://localhost:port/>)
- Make a note of your App ID and your App Secret so that you can add both into your ASP.NET 5 Web site later in this tutorial. Also, Add your Site URL (<https://localhost:44300/>) to test your application.

SocialLogins	Basic	Advanced	Migrations
<ul style="list-style-type: none"> Dashboard Settings Status & Review App Details Roles Open Graph Alerts Localize 	<p>App ID 862373430475128</p> <p>Display Name SocialLogins</p> <p>App Domains <input type="text"/></p>	<p>App Secret Show</p> <p>Namespace <input type="text"/></p> <p>Contact Email Used for important communication about your app</p>	<p>Website Quick Start ×</p> <p>Site URL http://localhost:55832/</p>

Use SecretManager to store Facebook AppId and AppSecret

The project created has the following code in Startup which reads the configuration values from a secret store. As a best practice, it is not recommended to store the secrets in a configuration file in the application since they can be checked into source control which may be publicly accessible.

Follow these steps to add the Facebook AppId and AppSecret to the Secret Manager:

- Open a Command Prompt and navigate to the folder of project.json for your project.
- Use DNVM (.NET Version Manager) to set a runtime version by running **dnvm use 1.0.0-beta5**

```
C:\Users\pranavra>dnvm list
```

Active	Version	Runtime	Architecture	Location	Alias
	1.0.0-beta4	clr	x64	C:\Users\pranavra\.dnx\runtimes	
	1.0.0-beta4	clr	x86	C:\Users\pranavra\.dnx\runtimes	default
	1.0.0-beta4	coreclr	x64	C:\Users\pranavra\.dnx\runtimes	
	1.0.0-beta4	coreclr	x86	C:\Users\pranavra\.dnx\runtimes	

```
C:\Users\pranavra>dnvm use 1.0.0-beta4
Adding C:\Users\pranavra\.dnx\runtimes\dnx-clr-win-x86.1.0.0-beta4\bin to process PATH

C:\Users\pranavra>dnvm list
```

Active	Version	Runtime	Architecture	Location	Alias
	1.0.0-beta4	clr	x64	C:\Users\pranavra\.dnx\runtimes	
*	1.0.0-beta4	clr	x86	C:\Users\pranavra\.dnx\runtimes	default
	1.0.0-beta4	coreclr	x64	C:\Users\pranavra\.dnx\runtimes	
	1.0.0-beta4	coreclr	x86	C:\Users\pranavra\.dnx\runtimes	

```
C:\Users\pranavra>
```

- Install the SecretManager tool using DNU (Microsoft .NET Development Utility) by running **dnu commands install SecretManager**
- Set the Facebook AppId by running **user-secret set Authentication:Facebook:AppId 862373430475128**
- Set the Facebook AppSecret by running **user-secret set Authentication:Facebook:AppSecret 862373430475128**
- The following code in the template reads the configuration values from the SecretManager. To learn more about SecretManager see [Secret Manager](#)

```
var configuration = new Configuration()
.AddJsonFile("config.json")
.AddJsonFile($"config.{env.EnvironmentName}.json", optional: true);

if (env.IsEnvironment("Development"))
{
    // This reads the configuration keys from the secret store.
    // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=53271
    configuration.AddUserSecrets();
}
```

Enable Facebook middleware

- You can add the options for Facebook middleware such as Facebook AppId and AppSecret in the ConfigureServices method in Startup.

```
services.Configure<FacebookAuthenticationOptions>(options =>
{
    options.AppId = Configuration["Authentication:Facebook:AppId"];
```

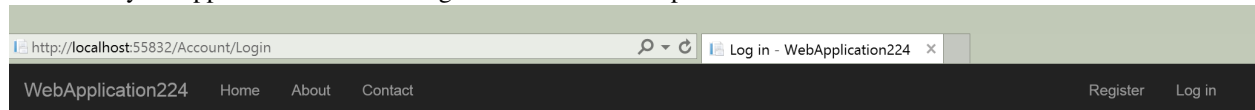
```
options.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
});
```

- Add the Facebook middleware by adding it to the HTTP request pipeline by uncommenting the following line in the Configure method in Startup.

```
app.UseFacebookAuthentication();
```

Login with Facebook

- Run your application and click Login. You will see an option for Facebook.



Log in.

Use a local account to log in.

Email

Password

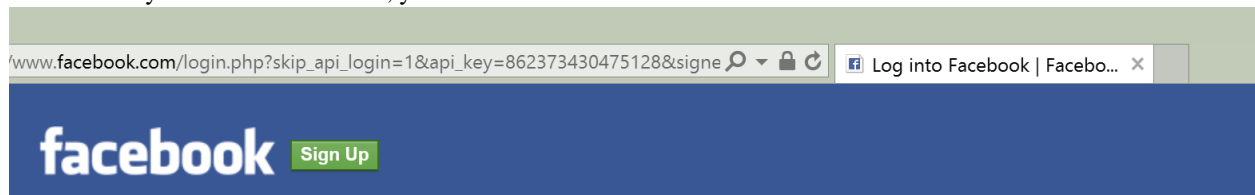
☐ Remember me?

[Register as a new user?](#)

[Forgot your password?](#)

Use another service to log in.

- When you click on Facebook, you will be redirected to Facebook for authentication.



Facebook Login

Email or Phone:

Password:

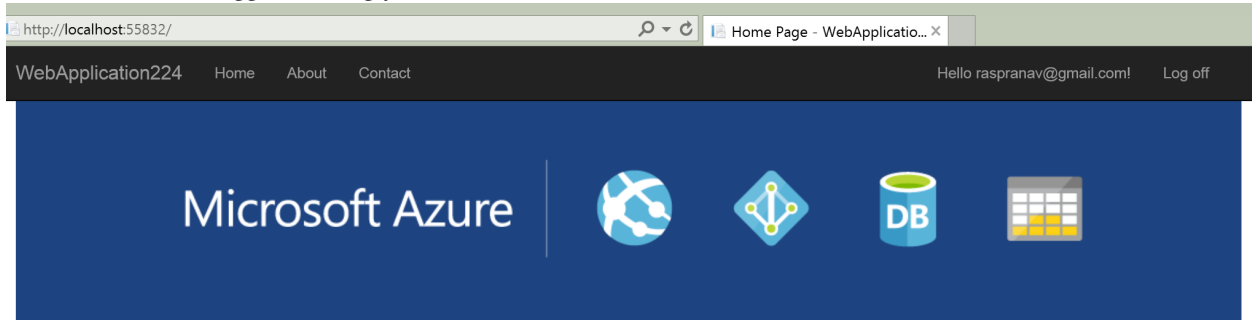
☐ Keep me logged in

or [Sign up for Facebook](#)

[Forgot your password?](#)

[English \(US\)](#) [Español](#) [Français \(France\)](#) [中文\(简体\)](#) [العربية](#) [Português \(Brasil\)](#) [Italiano](#) [한국어](#) [Deutsch](#) [हिन्दी](#)

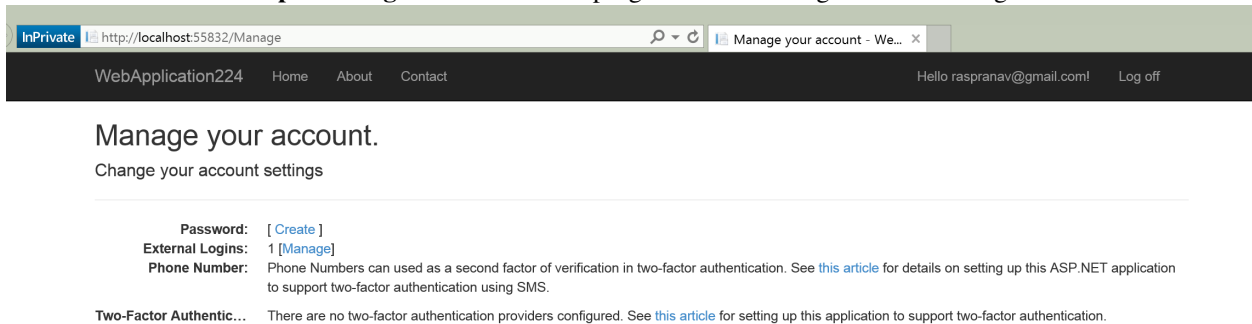
- Once you enter your Facebook credentials, then you will be redirected back to the Web site where you can set your email.
- You are now logged in using your Facebook credentials.



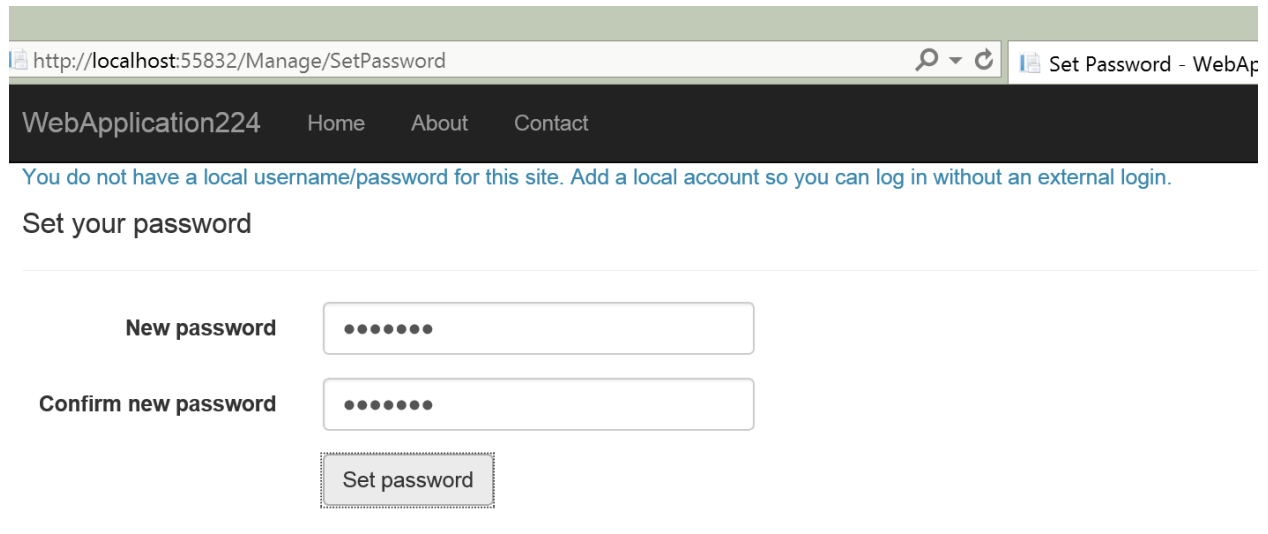
Optionally set password

When you authenticate with External Login providers, then you do not have to set a password locally on the Web site. This is useful since you do not have to create an extra password that you have to remember and maintain. However sometimes you might want to create a password and login using your email that you set during the login process with external providers. To set the password once you have logged in with an external provider:

- Click the **Hello raspranav@gmail.com** at the top right corner to navigate to the Manage view.



- Click **Create** next to the Password text.



http://localhost:55832/Manage/SetPassword

Set Password - WebAp

WebApplication224 Home About Contact

You do not have a local username/password for this site. Add a local account so you can log in without an external login.

Set your password

New password

Confirm new password

© 2015 - WebApplication224

- Set a valid password and you can use this to login with your email.

Next steps

- This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with Microsoft Account, Twitter, Google and other providers.
- Once you publish your Web site to Azure Web App, you should reset the AppSecret in the Facebook developer portal.
- Set the Facebook AppId and AppSecret as application setting in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Summary

ASP.NET Identity and Security middleware can be used to authenticate with external providers.

2.10.2 Account Confirmation and Password Recovery with ASP.NET Identity

By [Pranav Rastogi](#)

This tutorial shows you how to build an ASP.NET 5 Web site with email confirmation and password reset using ASP.NET Identity.

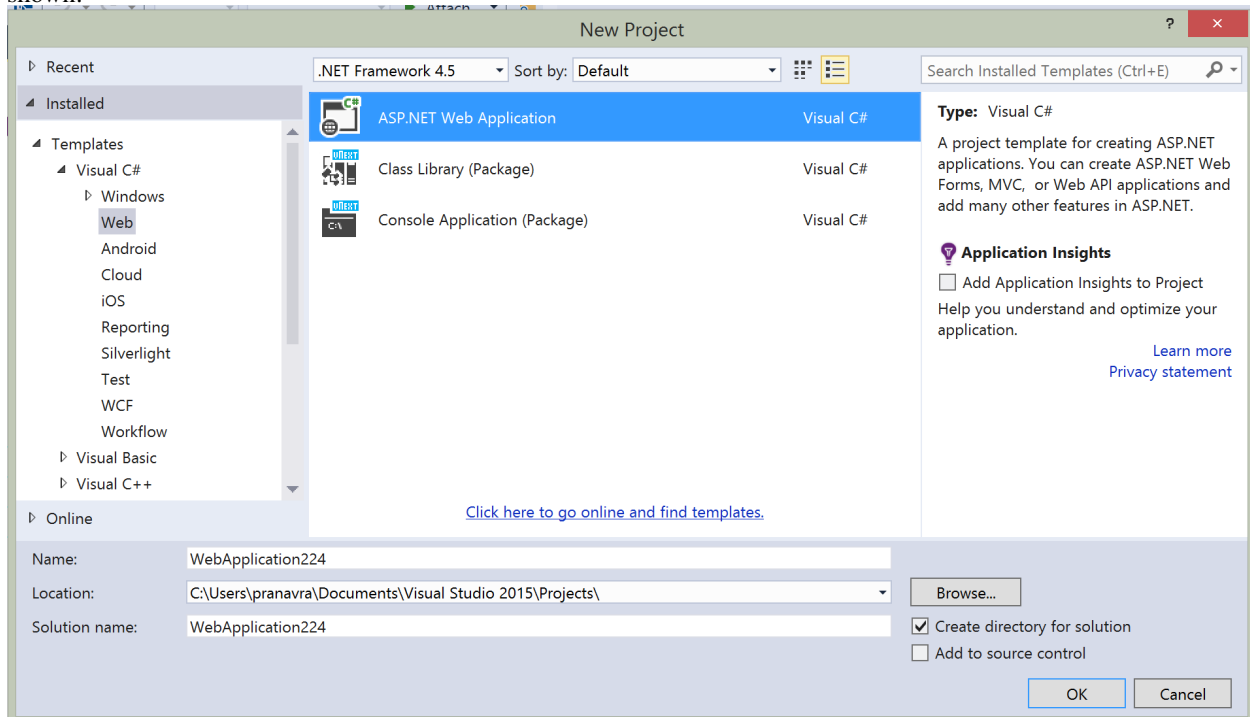
In this article:

- *Create a New ASP.NET 5 Project*
- *Running the Application*
- *Setup up Email provider*
- *Enable Account confirmation and Password recovery*
- *Register, confirm email, and reset password*
- *Require email confirmation before login*

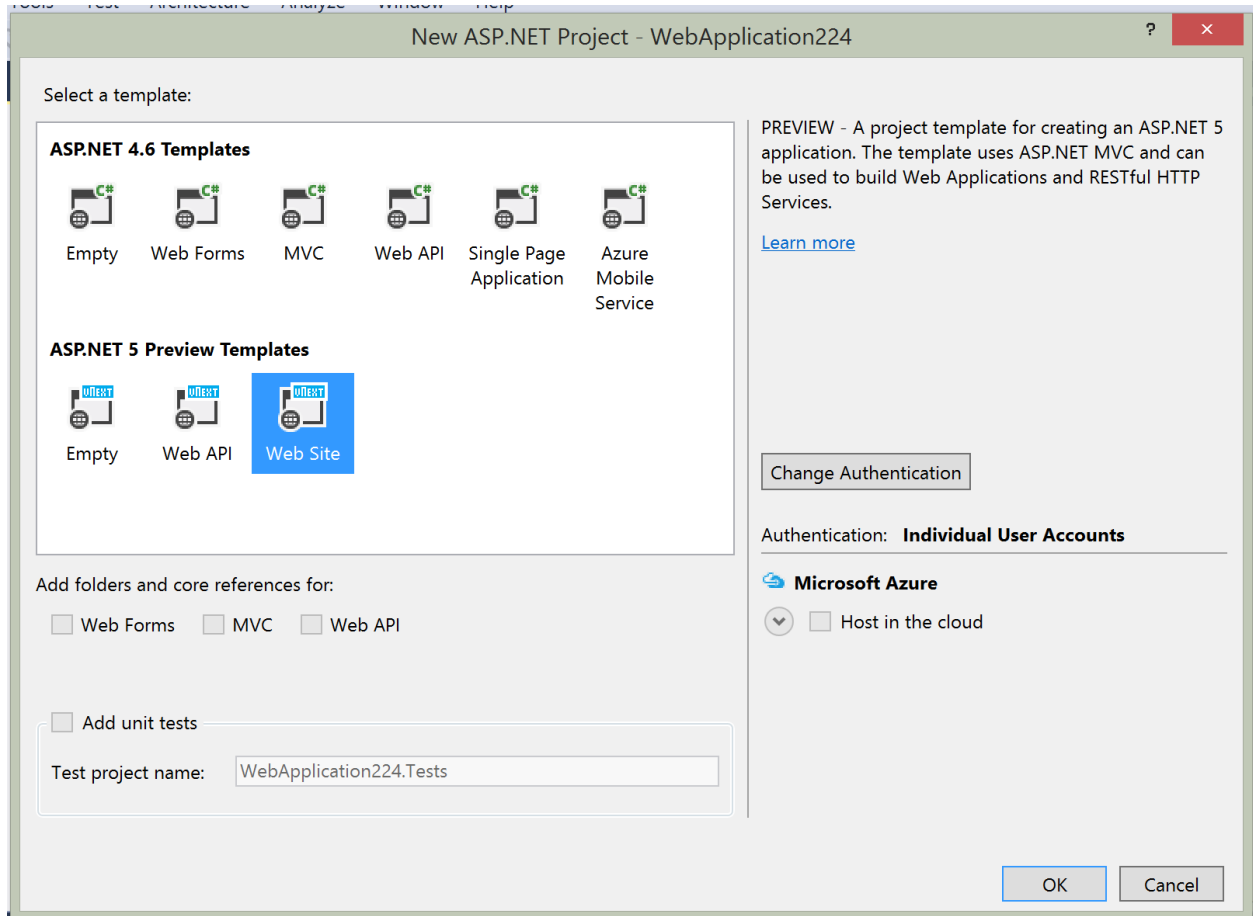
- *Next steps*
- *Summary*

Create a New ASP.NET 5 Project

To get started, open Visual Studio 2015. Next, create a New Project (from the Start Page, or via File - New - Project). On the left part of the New Project window, make sure the Visual C# templates are open and “Web” is selected, as shown:



Next you should see another dialog, the New ASP.NET Project window:

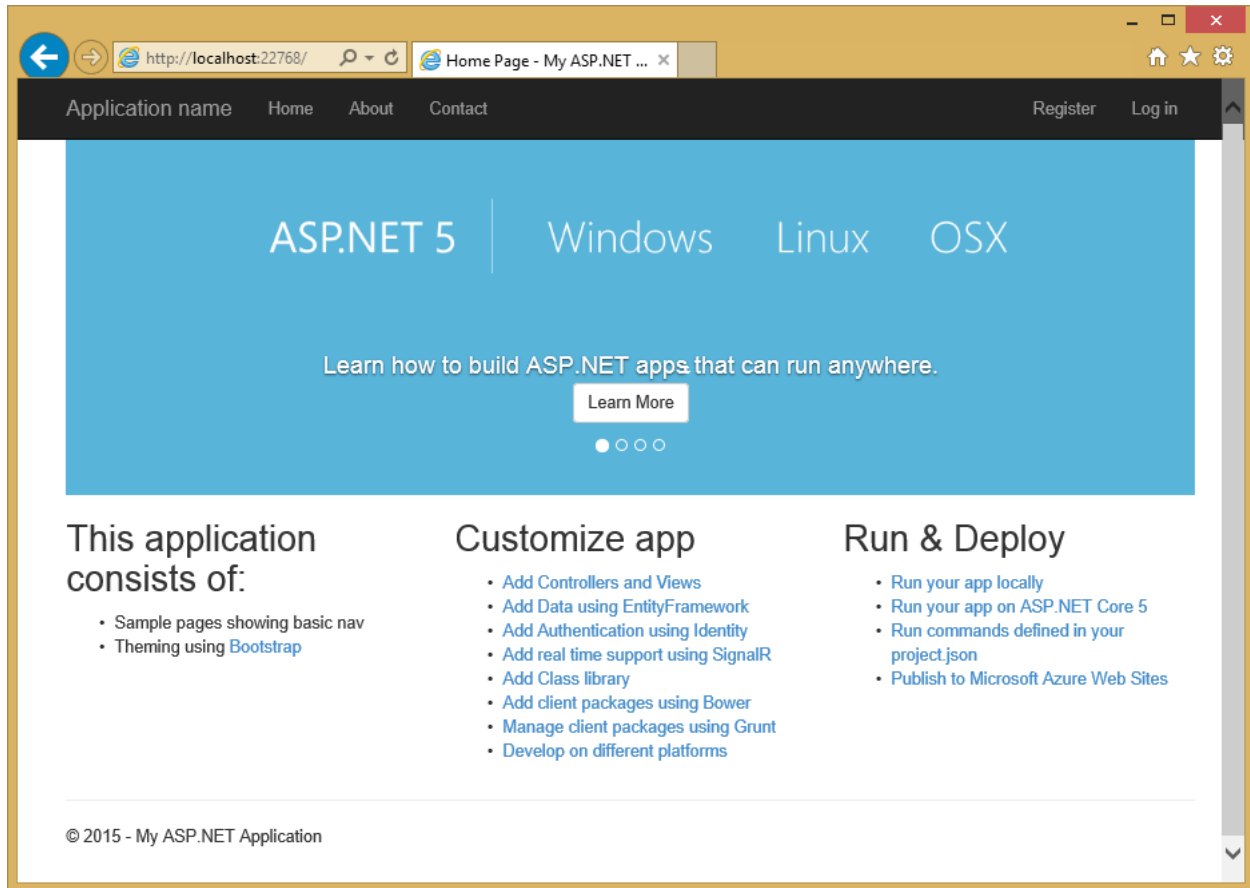


Select the ASP.NET 5 Web site template from the set of ASP.NET 5 templates. Make sure that authentication is set to **Individual User Accounts**. After selecting, click OK.

At this point, the project is created. It may take a few moments to load, and you may notice that Visual Studio's status bar indicates that it is downloading some resources as part of this process. Visual Studio ensures that some required files are pulled into the project when a solution is opened (or a new project is created), and other files may be pulled in at compile time.

Running the Application

Run the application and after a quick build step, you should see it open in your web browser.



Setup up Email provider

Although this tutorial only shows how to add email notification through [SendGrid](#), you can send email using SMTP and other m

- Install SendGrid NuGet package
- Go to the [Azure SendGrid sign up page](#) and register for a free SendGrid account.
- Add code in **MessageServices.cs** similar to the following to configure SendGrid

```
public static Task SendEmailAsync(string email, string subject, string message)
{
    // Plug in your email service here to send an email.
    var myMessage = new SendGridMessage();
    myMessage.AddTo(email);
    myMessage.From = new System.Net.Mail.MailAddress("Joe@contoso.com", "Joe S.");
    myMessage.Subject = subject;
    myMessage.Text = message;
    myMessage.Html = message;
    var credentials = new NetworkCredential("SendGridUser", "SendGridKey");
    // Create a Web transport for sending email.
    var transportWeb = new Web(credentials);
    // Send the email.
    if (transportWeb != null)
    {
        return transportWeb.DeliverAsync(myMessage);
    }
}
```

```

    }
    else
    {
        return Task.FromResult(0);
    }
}

```

Note: SendGrid cannot target dnxcore50: If you build your project then you will get compilation errors. This is because SendGrid does not have a package for dnxcore50 and some APIs such as System.Mail are not available on .NET Core. You can remove dnxcore50 from project.json or call the REST API from SendGrid to send email.

Note: Security Note: Never store sensitive data in your source code. The account and credentials are added to the code above to keep the sample simple. Follow the steps on how to store secrets using the [Secret Manager](#). The template code is setup to read configuration values from the SecretManager.

Enable Account confirmation and Password recovery

The template already has the code for account confirmation and password recovery. Follow these steps to enable it:

- In your project open **AccountController** and look at the **Register** action.
- Uncomment the code to enable account confirmation.

```

public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset please v
            // Send an email with this link
            var code = await UserManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code = co
            await MessageServices.SendEmailAsync(model.Email, "Confirm your account",
                "Please confirm your account by clicking this link: <a href=\"\" + callbackUrl + \"\">
            //await SignInManager.SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

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

```

- Enable password recovery by uncommenting the code in the **ForgotPassword** action and its associated view:

```

public async Task<IActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindByNameAsync(model.Email);
        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user)))
        {

```

```
        // Don't reveal that the user does not exist or is not confirmed
        return View("ForgotPasswordConfirmation");
    }

    // For more information on how to enable account confirmation and password reset please visit
    // Send an email with this link
    var code = await UserManager.GeneratePasswordResetTokenAsync(user);
    var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = code });
    await MessageServices.SendEmailAsync(model.Email, "Reset Password",
        "Please reset your password by clicking here: <a href=\"" + callbackUrl + "\">link</a>");
    return View("ForgotPasswordConfirmation");
}

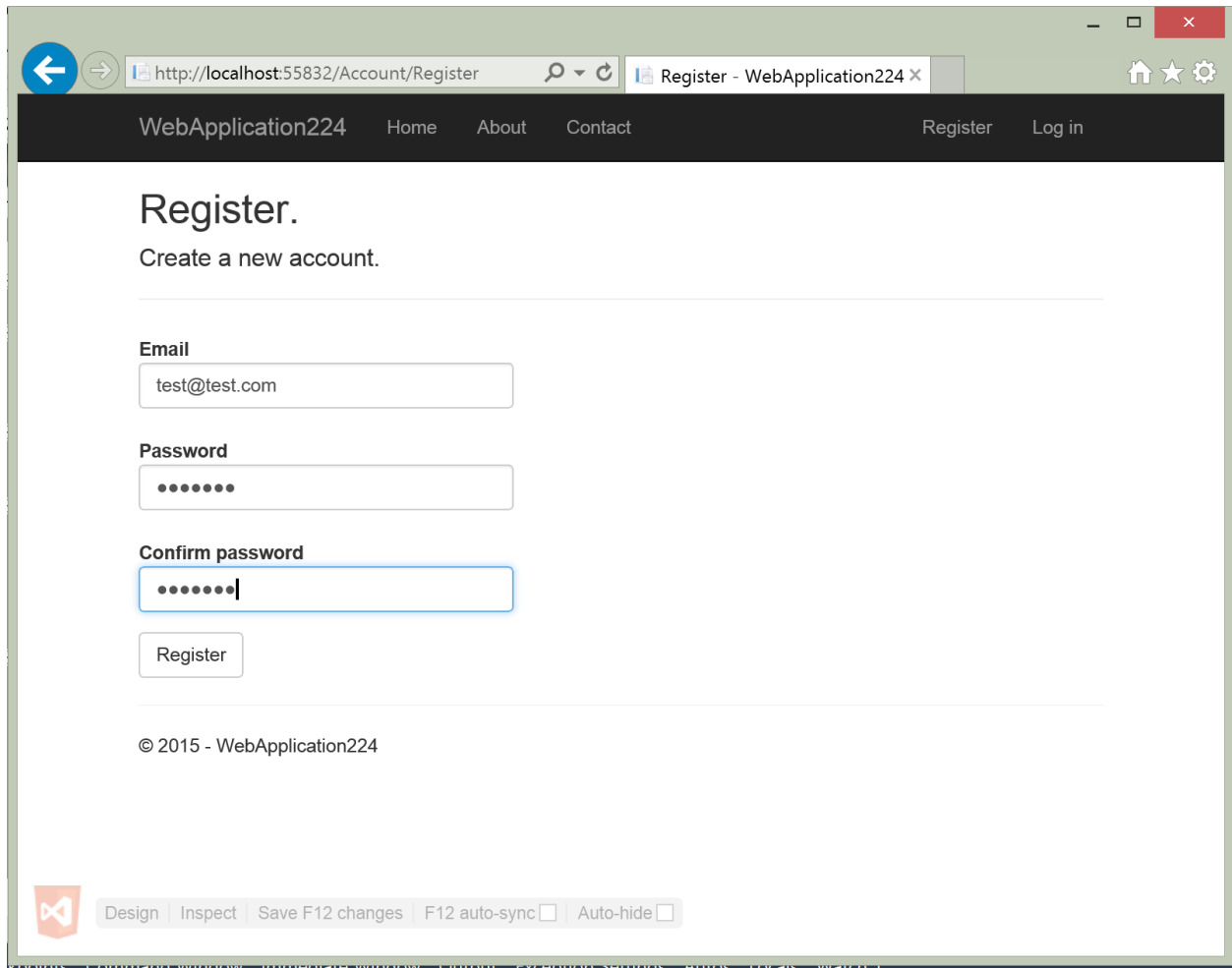
// If we got this far, something failed, redisplay form
return View(model);
}
```

```
<form asp-controller="Account" asp-action="ForgotPassword" method="post" class="form-horizontal" role="form">
    <h4>Enter your email.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Submit" />
        </div>
    </div>
</form>
```

Register, confirm email, and reset password

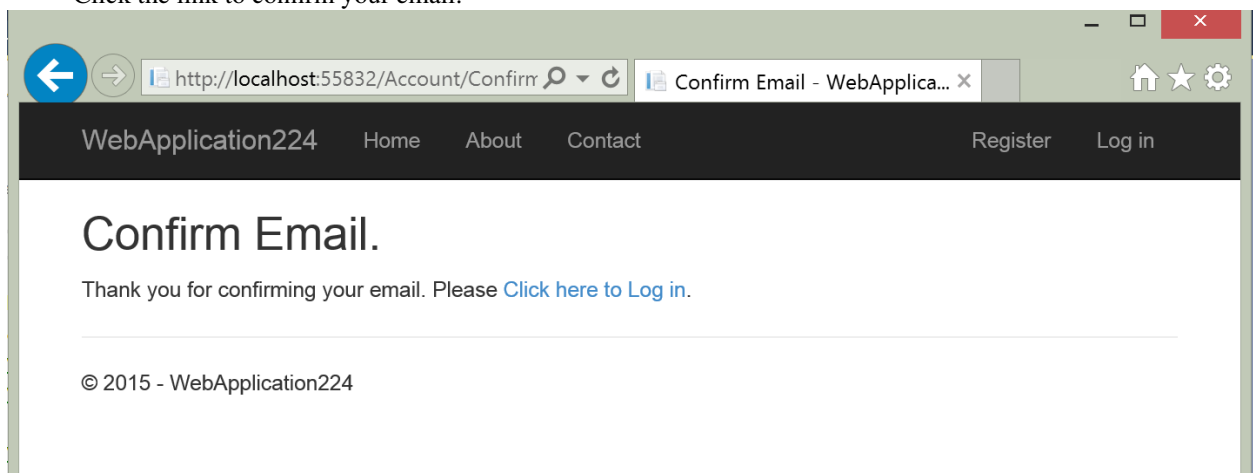
Let us run the Web site and show the account confirmation and password recovery flow.

- Run the application and register a new user



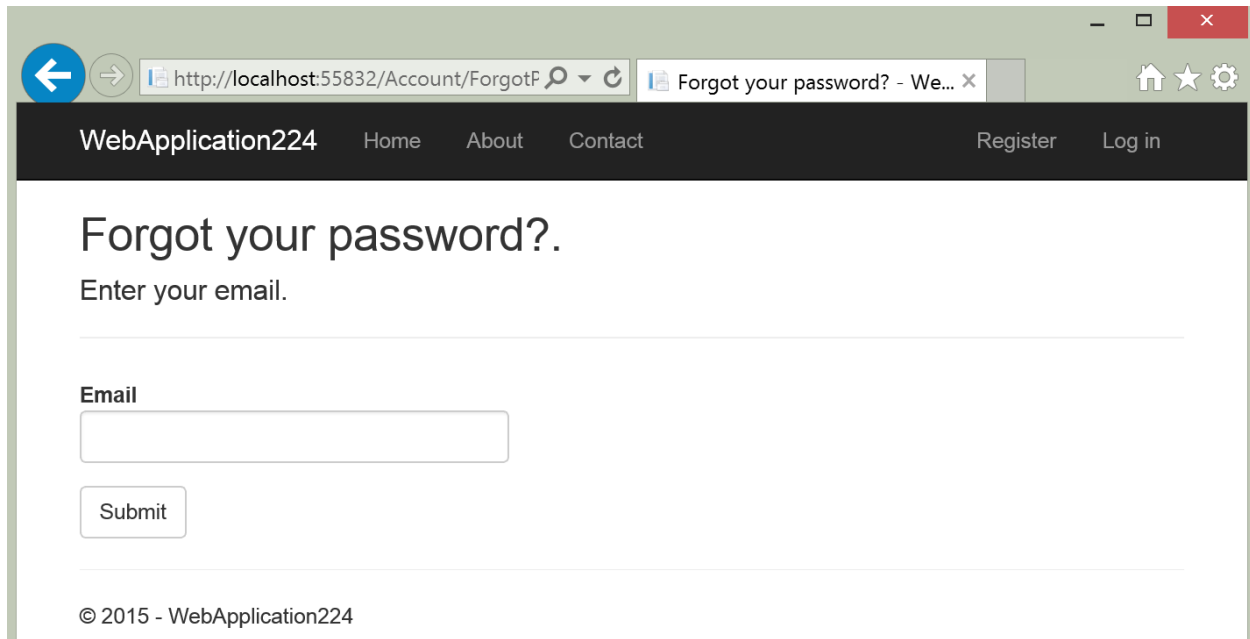
The screenshot shows a web browser window with the URL `http://localhost:55832/Account/Register`. The page title is "Register - WebApplication224". The page has a dark navigation bar with links: "WebApplication224", "Home", "About", "Contact", "Register", and "Log in". The main content area is titled "Register." and "Create a new account." It contains three input fields: "Email" with the value "test@test.com", "Password" with masked characters ".....", and "Confirm password" with masked characters ".....". Below the fields is a "Register" button. At the bottom, there is a copyright notice "© 2015 - WebApplication224".

- Check your email for the account confirmation link.
- Click the link to confirm your email.



The screenshot shows a web browser window with the URL `http://localhost:55832/Account/Confirm`. The page title is "Confirm Email - WebApplica...". The page has a dark navigation bar with links: "WebApplication224", "Home", "About", "Contact", "Register", and "Log in". The main content area is titled "Confirm Email." and "Thank you for confirming your email. Please [Click here to Log in](#)." At the bottom, there is a copyright notice "© 2015 - WebApplication224".

- Login with your email and password.
- Log Off.
- Click Login and select **Forgot your password?**



WebApplication224 Home About Contact Register Log in

Forgot your password?.

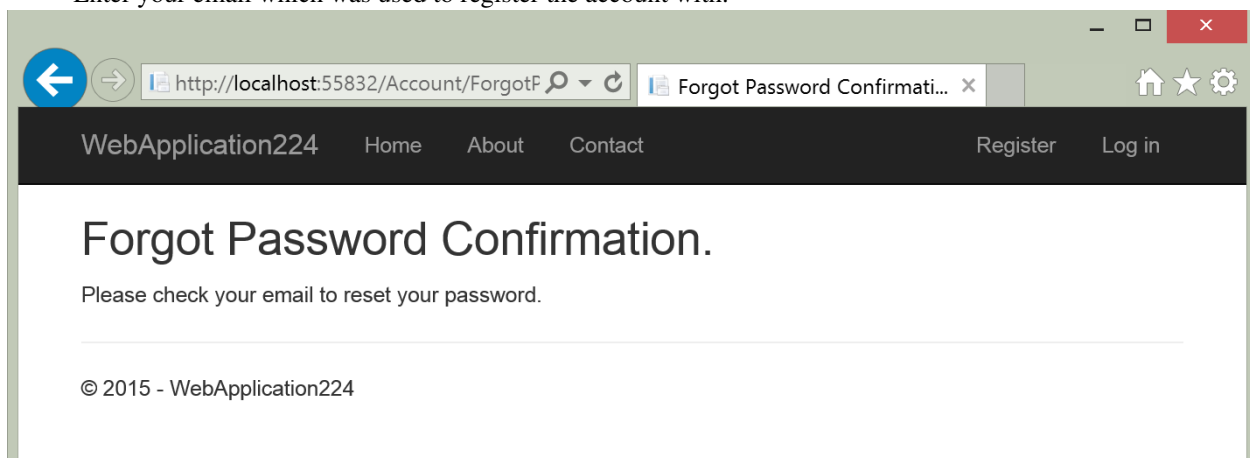
Enter your email.

Email

Submit

© 2015 - WebApplication224

- Enter your email which was used to register the account with.



WebApplication224 Home About Contact Register Log in

Forgot Password Confirmation.

Please check your email to reset your password.

© 2015 - WebApplication224

- An email with the link to reset your password will be sent. Check your email and click it to reset your password.

WebApplication224 Home About Contact Register Log in

Reset password.

Reset your password.

Email

Password

Confirm password

Reset

- After your password has been successfully reset, you can login with your email and new password.

WebApplication224 Home About Contact Register Log in

Reset password confirmation.

Your password has been reset. Please [Click here to log in.](#)

© 2015 - WebApplication224

Require email confirmation before login

Currently once a user completes the registration form, they are logged in. You generally want to confirm their email before logging them in. In the section below, we will modify the code to require new users to have a confirmed email before they are logged in (authenticated). Update the HttpPost Login action with the following highlighted changes.

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewBag.ReturnUrl = returnUrl;
    if (ModelState.IsValid)
```

```
{
    // Require the user to have a confirmed email before they can log on.
    var user = await UserManager.FindByNameAsync(model.Email);
    if (user != null)
    {
        if (!await UserManager.IsEmailConfirmedAsync(user))
        {
            ModelState.AddModelError(string.Empty, "You must have a confirmed email to log in.");
            return View(model);
        }
    }
    // Code removed for brevity. You should have the code that was in the project.
}

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

Next steps

- Once you publish your Web site to Azure Web App, you should reset the secrets for SendGrid. Set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Summary

ASP.NET Identity can be used to add account confirmation and password recovery.

2.10.3 Two-factor Authentication with SMS Using ASP.NET Identity

By [Pranav Rastogi](#)

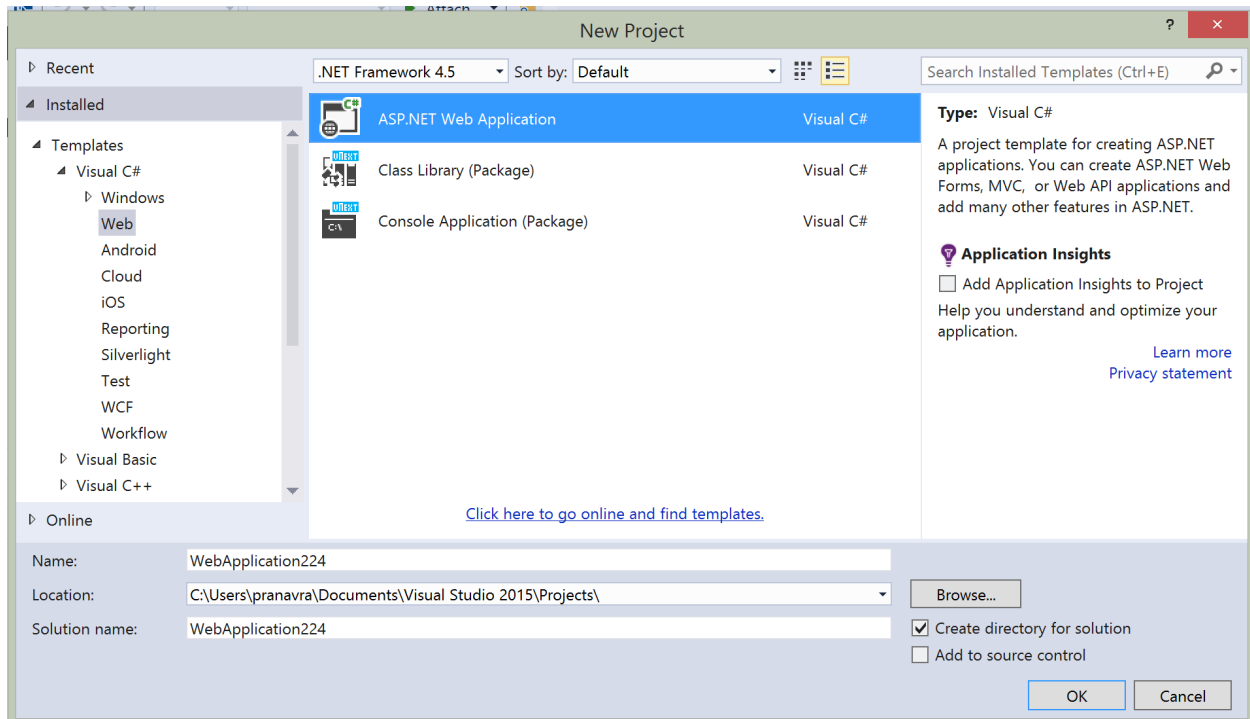
This tutorial will show you how to set up Two-factor authentication (2FA) using SMS.

In this article:

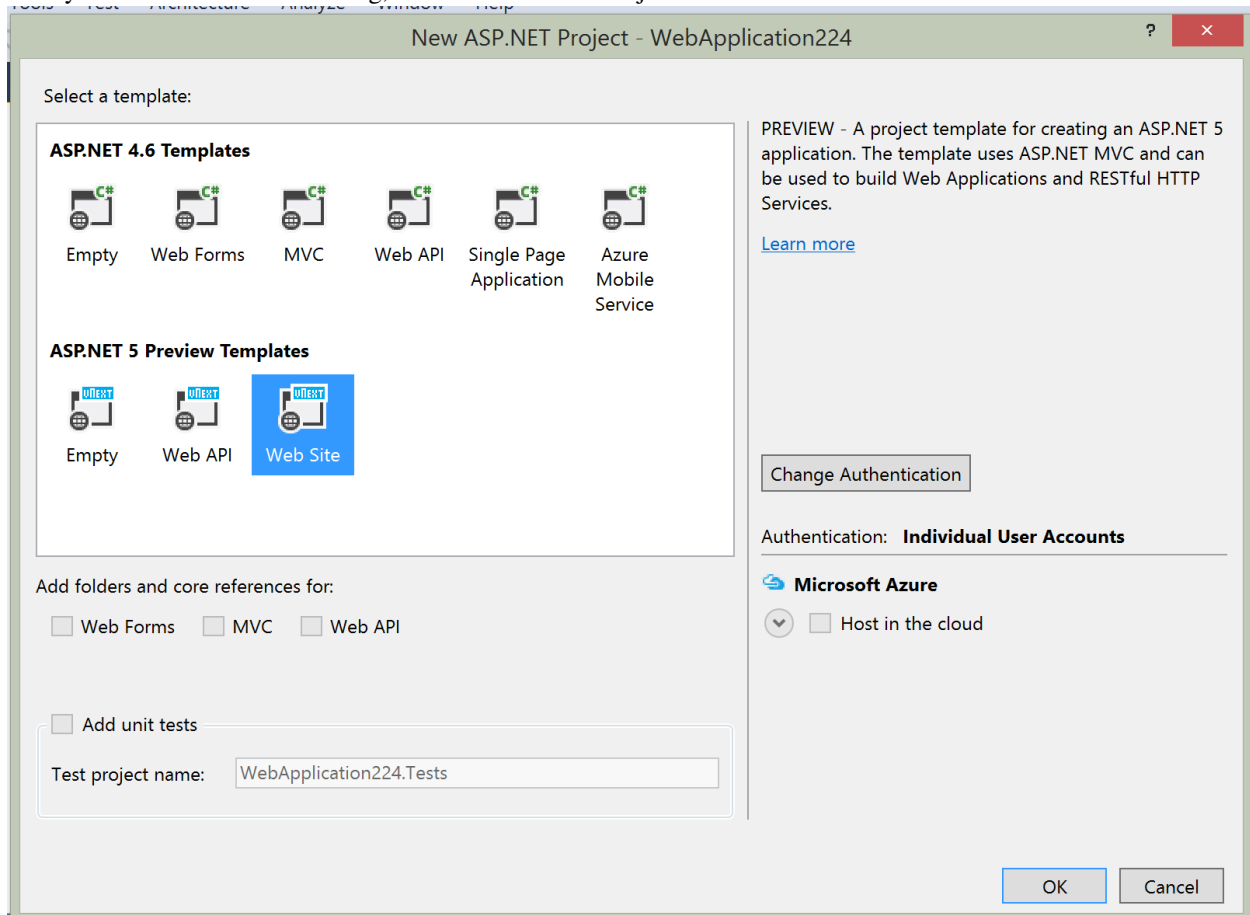
- *Create a New ASP.NET 5 Project*
- *Running the Application*
- *Setup up SMS for Two-factor authentication*
- *Enable Two-factor authentication*
- *Login with Two-factor authentication*
- *Account lockout for protecting against brute force attacks*
- *Next steps*
- *Summary*

Create a New ASP.NET 5 Project

To get started, open Visual Studio 2015. Next, create a New Project (from the Start Page, or via File - New - Project). On the left part of the New Project window, make sure the Visual C# templates are open and “Web” is selected, as shown:



Next you should see another dialog, the New ASP.NET Project window:

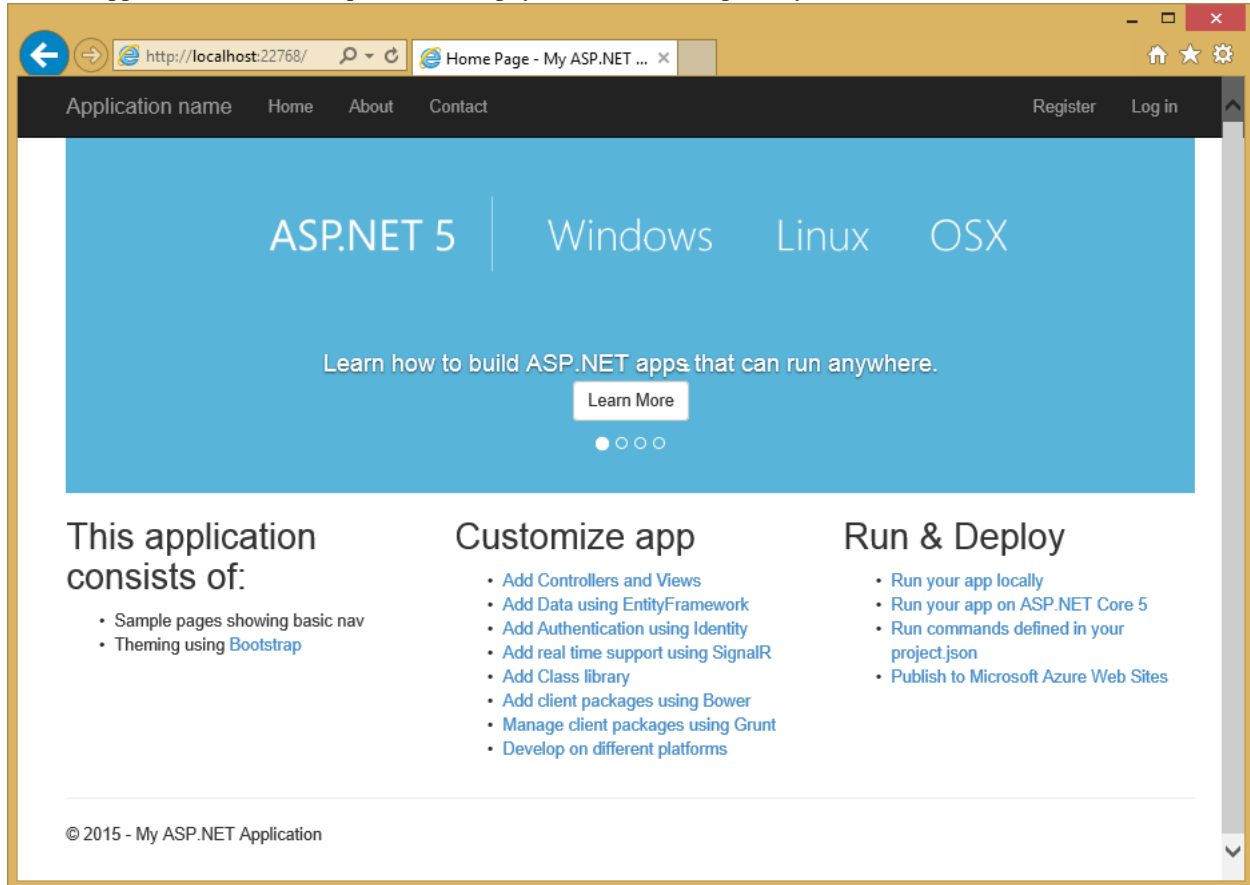


Select the ASP.NET 5 Web site template from the set of ASP.NET 5 templates. Make sure you have Individual Authentication selected for this template. After selecting, click OK.

At this point, the project is created. It may take a few moments to load, and you may notice Visual Studio's status bar indicates that Visual Studio is downloading some resources as part of this process. Visual Studio ensures some required files are pulled into the project when a solution is opened (or a new project is created), and other files may be pulled in at compile time.

Running the Application

Run the application and after a quick build step, you should see it open in your web browser.



Setup up SMS for Two-factor authentication

This tutorial uses Twilio, but you can use any SMS provider.

- Create a [Twilio](#) account.
- From the Dashboard tab of your Twilio account, copy the Account SID and Auth token.
- From the Numbers tab, copy your Twilio phone number.
- Make the SID, account token and phone number available to the app:
- Add the Twilio NuGet package to the app.
- Add code in MessageServices to send SMS

```
public static Task SendSmsAsync(string number, string message)
{
    // Plug in your SMS service here to send a text message.
    var twilio = new TwilioRestClient("YourTwilioSid", "YourTwilioToken");
    var result = twilio.SendMessage("YourTwilioPhoneNumber", number, message);
    return Task.FromResult(0);
}
```

Note: Twilio cannot target dnxcore50: If you build your project then you will get compilation errors. This is because Twilio does not have a package for dnxcore50. You can remove dnxcore50 from project.json or call the REST API from Twilio to send SMS.

Note: Security Note: Never store sensitive data in your source code. The account and credentials are added to the code above to keep the sample simple. Follow the steps on how to store secrets using the [Secret Manager](#). The template code is setup to read configuration values from the SecretManager.

Enable Two-factor authentication

The app already has the code to enable two-factor authentication. Follow these steps to enable two-factor authentication:

- In your project open Index view in Manage folder.
- Uncomment the code to add a phone number to an user account.

```
<dt>Phone Number:</dt>
<dd>
    @(Model.PhoneNumber ?? "None") [
        @if (Model.PhoneNumber != null)
        {
            <a asp-controller="Manage" asp-action="AddPhoneNumber">Change</a>
            @: &nbsp;|&nbsp;
            <a asp-controller="Manage" asp-action="RemovePhoneNumber">Remove</a>
        }
        else
        {
            <a asp-controller="Manage" asp-action="AddPhoneNumber">Add</a>
        }
    ]
</dd>
```

- Uncomment the code to enable/ disable two-factor authentication for an user account.

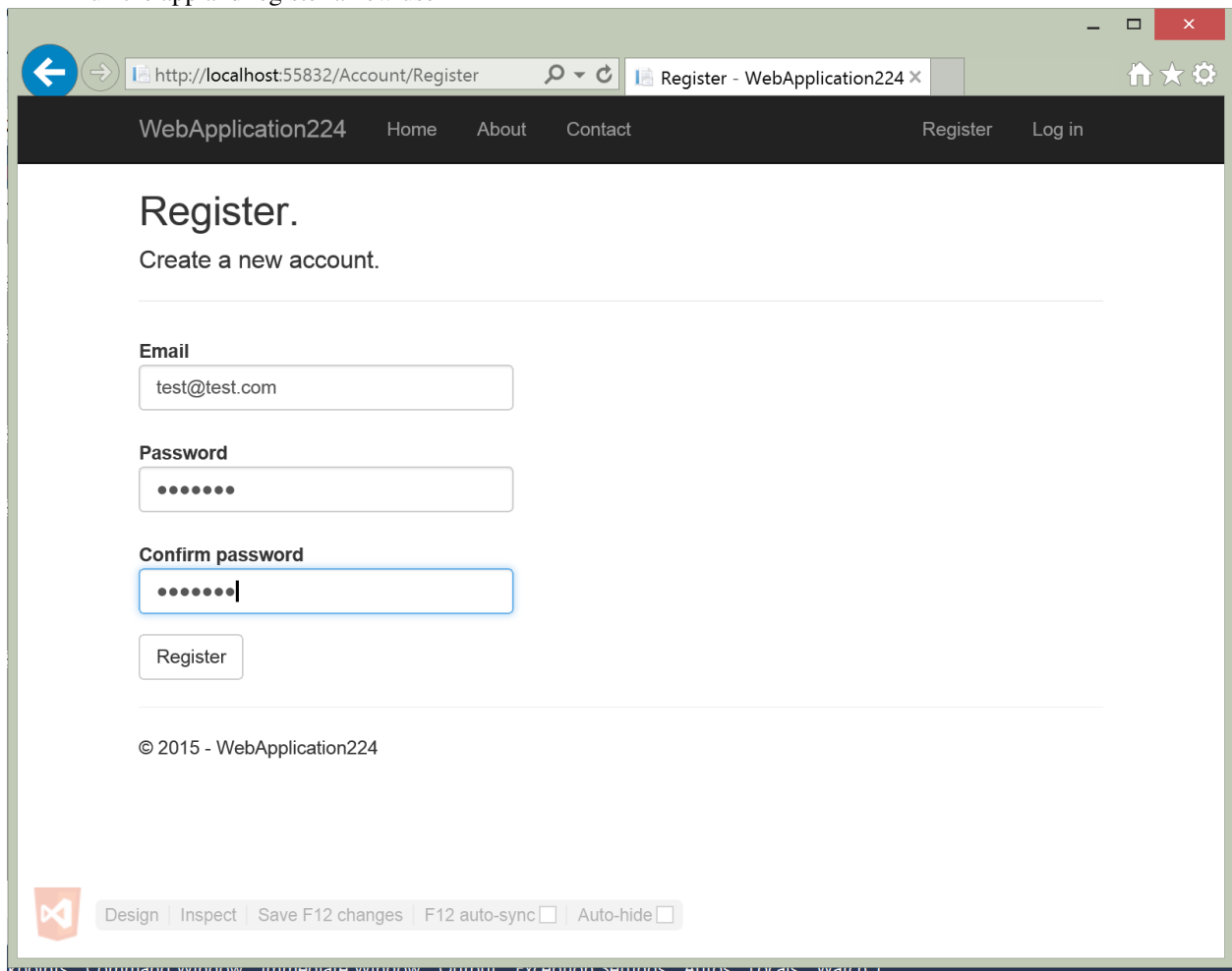
```
<dt>Two-Factor Authentication:</dt>
<dd>
    @if (Model.TwoFactor)
    {
        <form asp-controller="Manage" asp-action="DisableTwoFactorAuthentication" method="post">
            <text>
                Enabled
                <input type="submit" value="Disable" class="btn btn-link" />
            </text>
        </form>
    }
    else
    {
        <form asp-controller="Manage" asp-action="EnableTwoFactorAuthentication" method="post">
```

```
        <text>
            Disabled
        <input type="submit" value="Enable" class="btn btn-link" />
        </text>
    </form>
}
</dd>
```

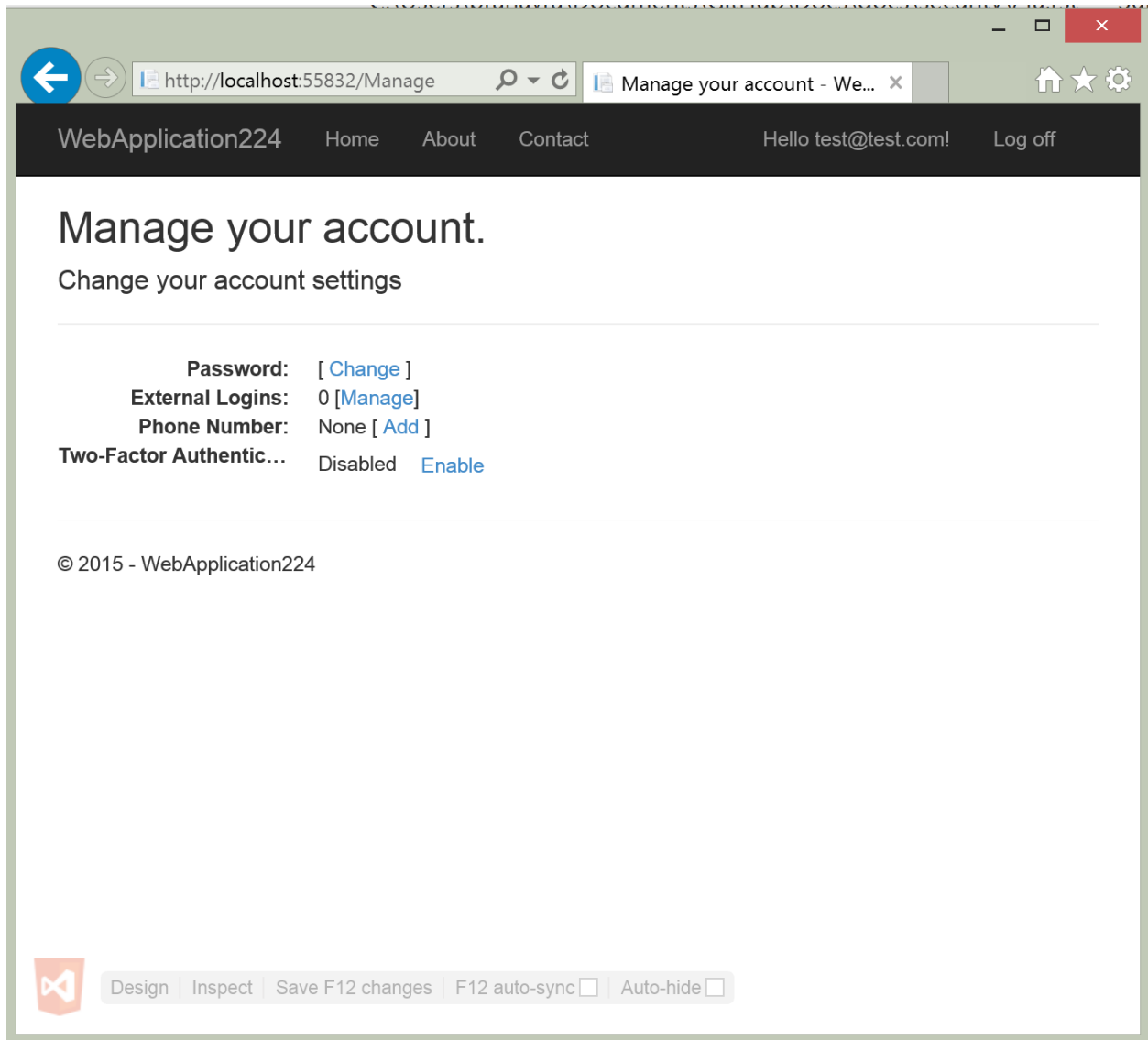
Login with Two-factor authentication

Let us run the Web site and show the two-factor authentication flow.

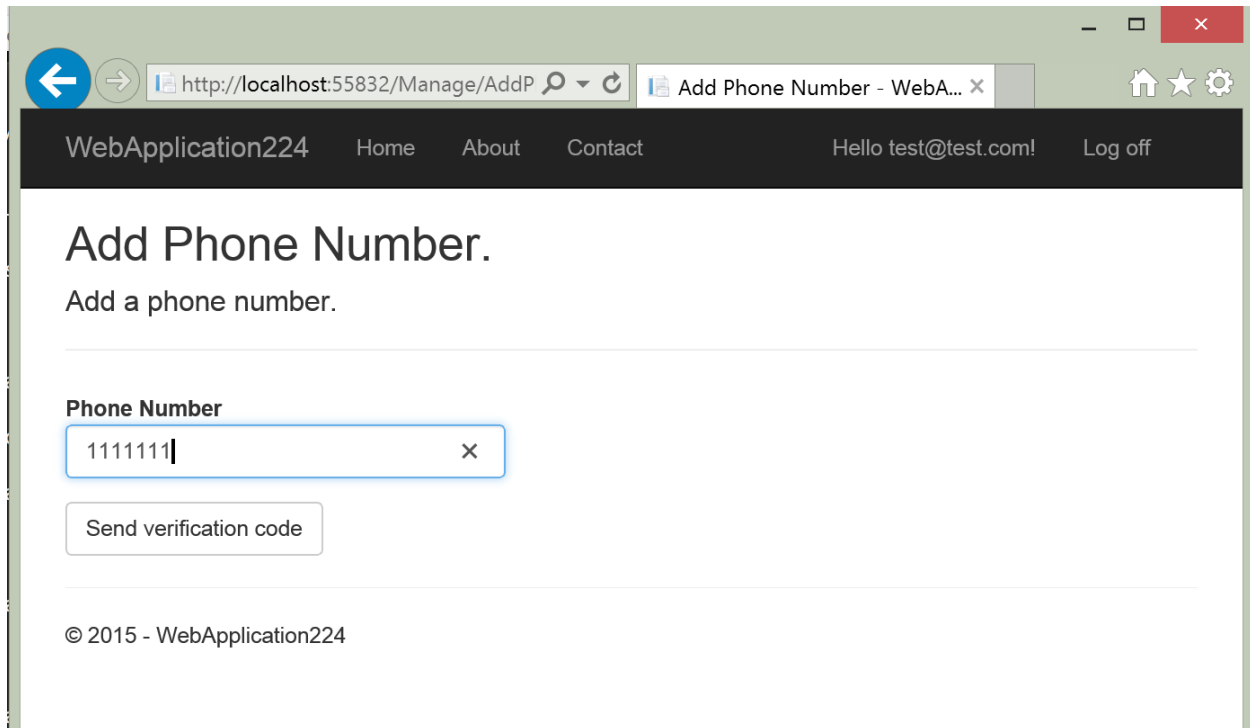
- Run the app and register a new user



- Click on your user name, which activates the Index action method in Manage controller.

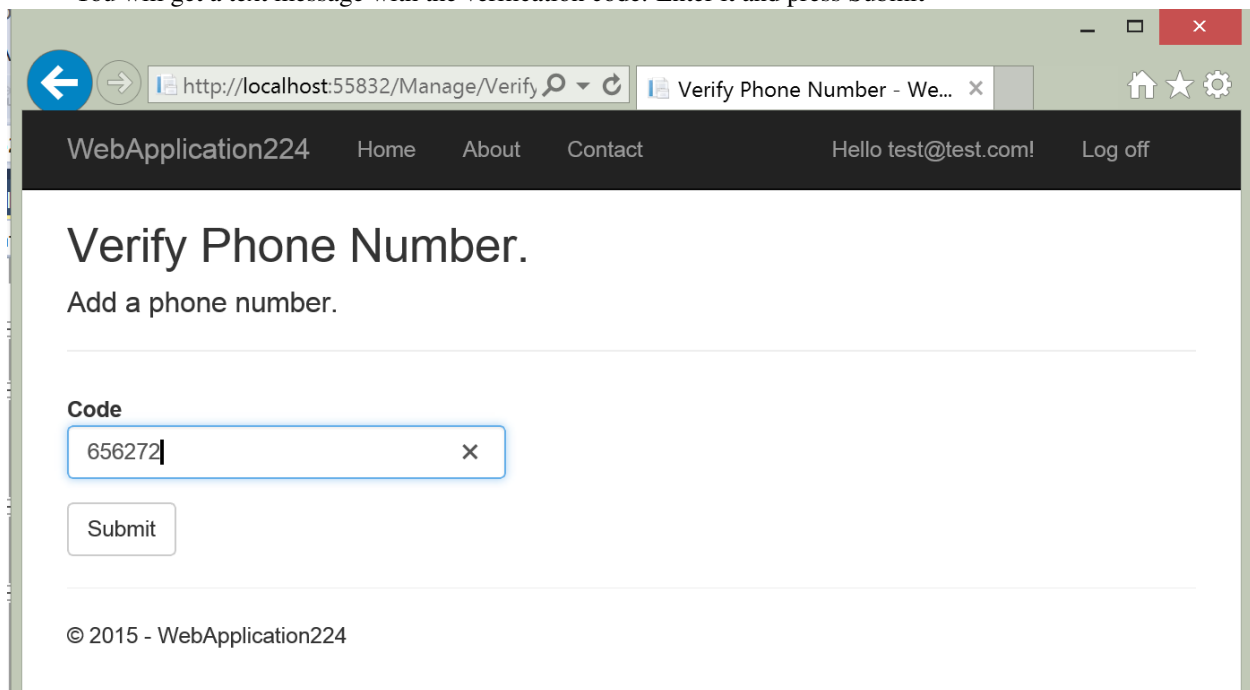


- Add Phone Number



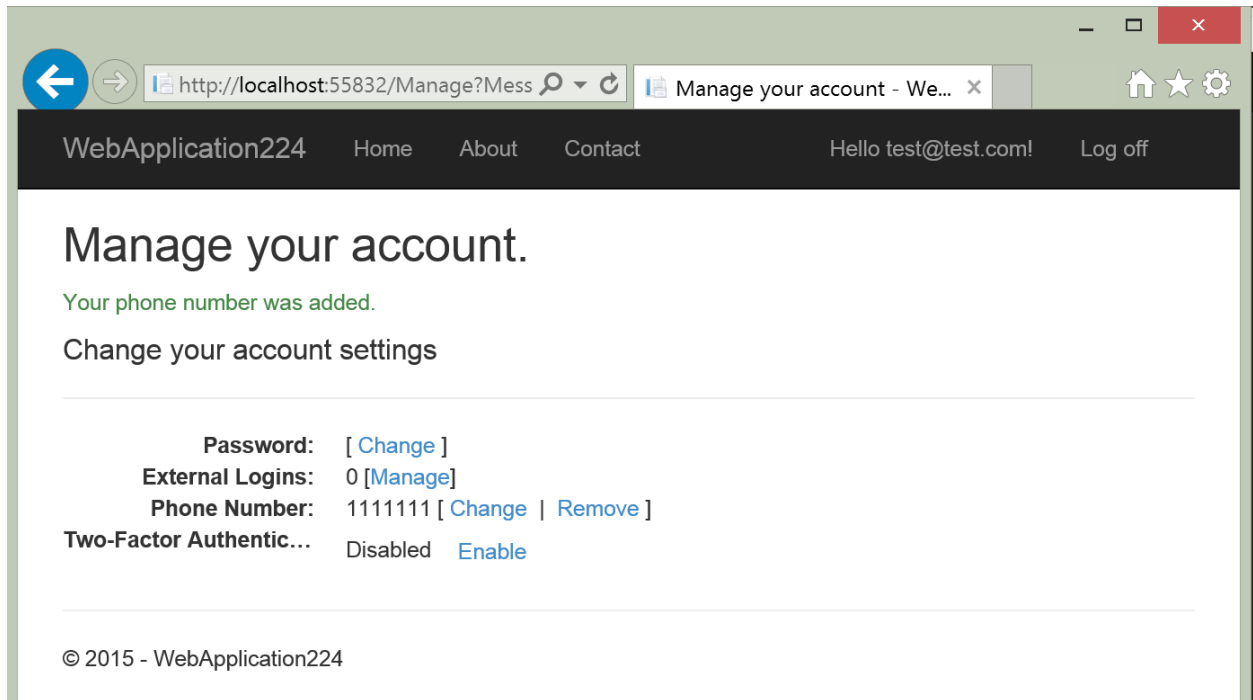
A screenshot of a web browser window showing the 'Add Phone Number' page of 'WebApplication224'. The browser's address bar shows 'http://localhost:55832/Manage/AddP'. The page has a dark navigation bar with links for 'Home', 'About', and 'Contact', and a user greeting 'Hello test@test.com!' with a 'Log off' link. The main content area has the heading 'Add Phone Number.' followed by the instruction 'Add a phone number.' Below this is a form with a label 'Phone Number' and a text input field containing '1111111'. To the right of the input field is a small 'x' icon. Below the input field is a button labeled 'Send verification code'. At the bottom of the page is a copyright notice '© 2015 - WebApplication224'.

- You will get a text message with the verification code. Enter it and press Submit

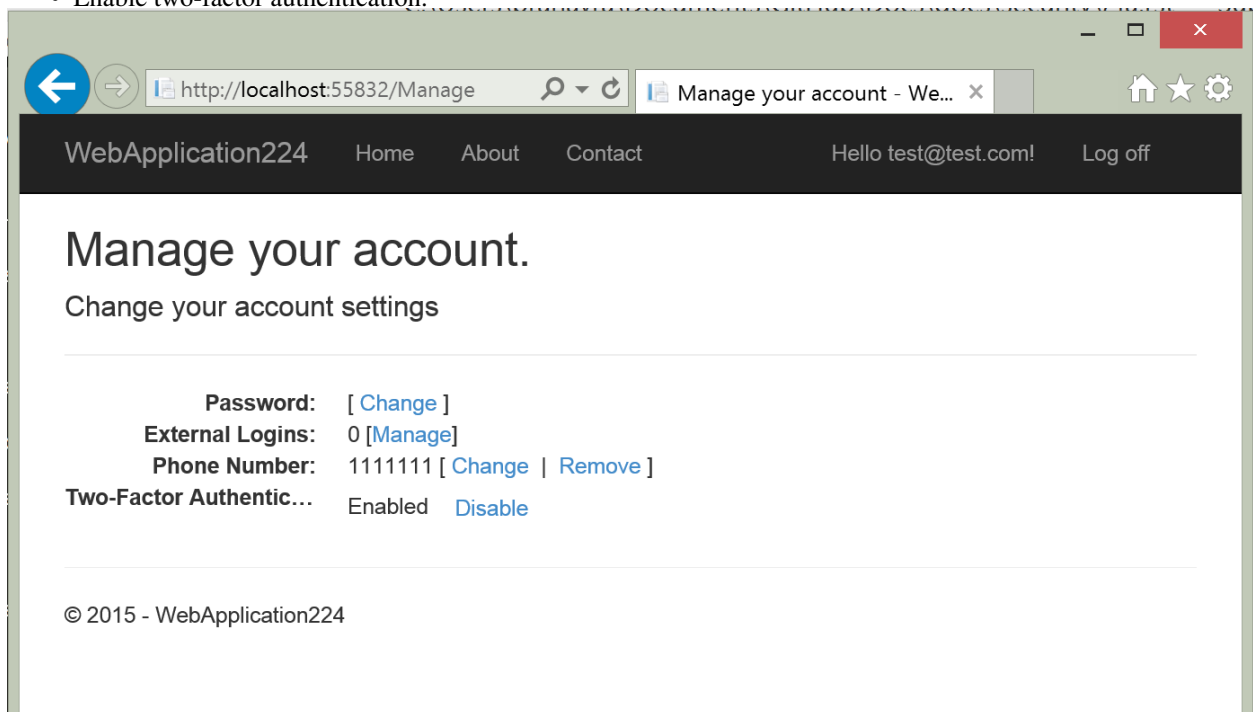


A screenshot of a web browser window showing the 'Verify Phone Number' page of 'WebApplication224'. The browser's address bar shows 'http://localhost:55832/Manage/Verify'. The page has the same dark navigation bar as the previous screenshot. The main content area has the heading 'Verify Phone Number.' followed by the instruction 'Add a phone number.' Below this is a form with a label 'Code' and a text input field containing '656272'. To the right of the input field is a small 'x' icon. Below the input field is a button labeled 'Submit'. At the bottom of the page is a copyright notice '© 2015 - WebApplication224'.

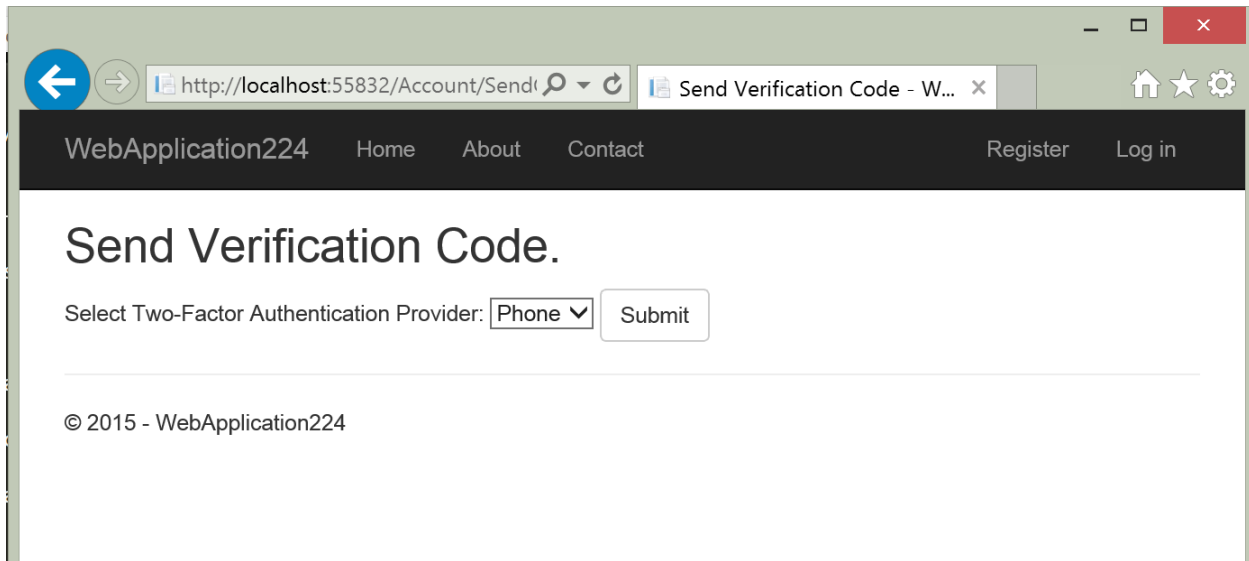
- The Manage view shows your phone number was added successfully.



- Enable two-factor authentication.

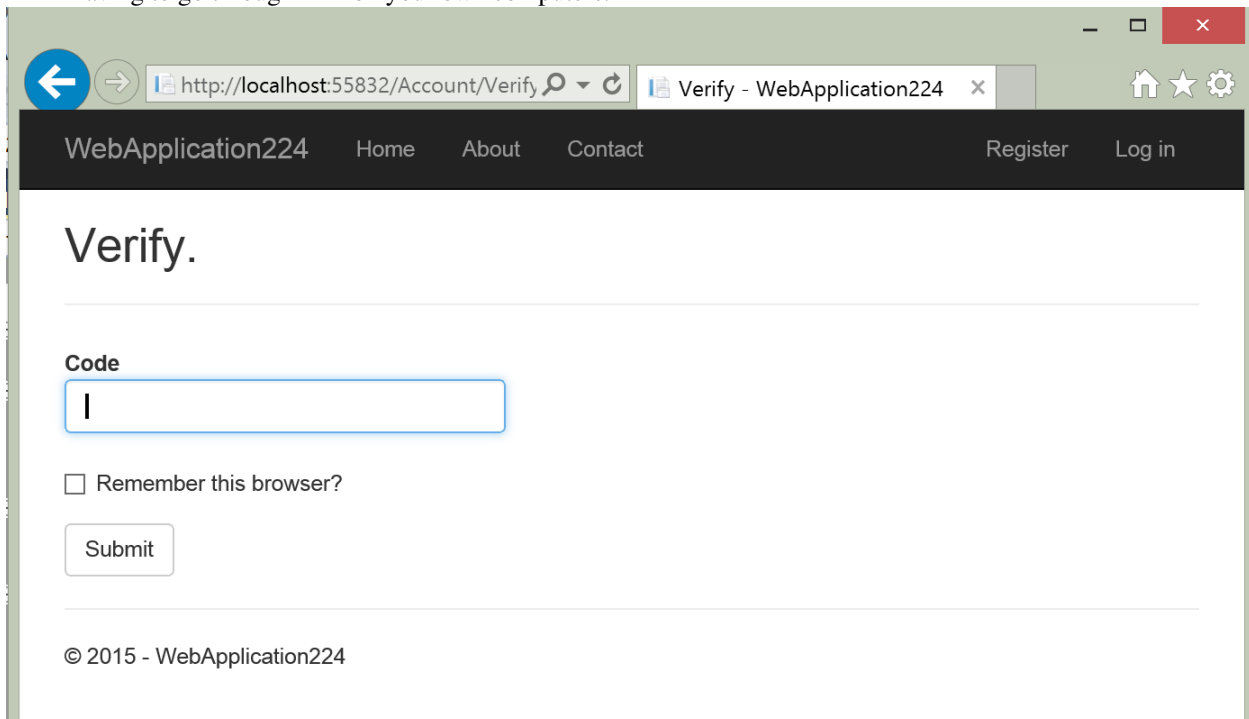


- Log Off.
- Login with user name and password.
- Since you have enable two-factor authentication, you have to verify the second step. In this case you have a verified phone number so you can use it to verify the PIN. If you had other factors such as email, QR code generators, then you could use those as a verification step.



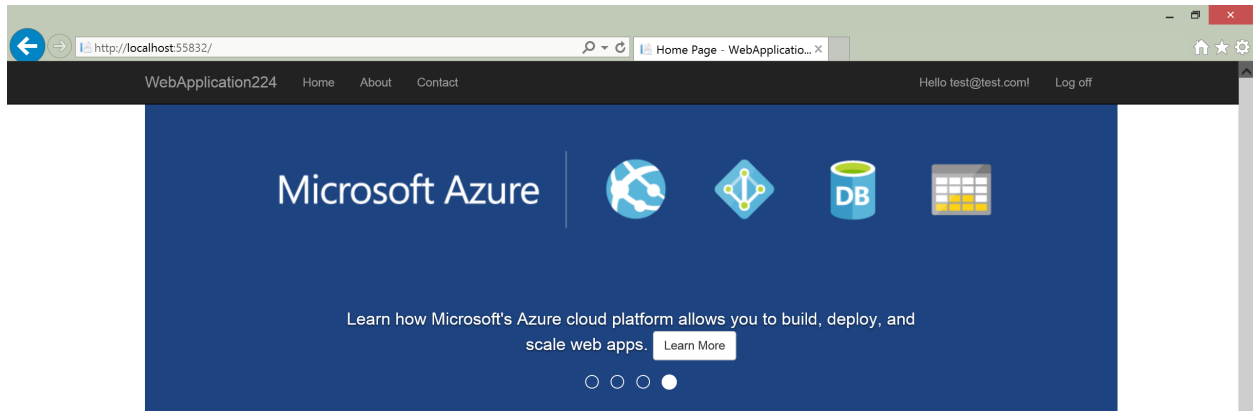
The screenshot shows a web browser window with the address bar at `http://localhost:55832/Account/SendVerificationCode`. The page title is "Send Verification Code - WebApplication224". The navigation bar includes "WebApplication224", "Home", "About", "Contact", "Register", and "Log in". The main content area has the heading "Send Verification Code." and a form with the label "Select Two-Factor Authentication Provider:" followed by a dropdown menu showing "Phone" and a "Submit" button. At the bottom, there is a copyright notice: "© 2015 - WebApplication224".

- You will get a text message with the verification code. Enter it.
- Clicking on the Remember this browser check box will exempt you from needing to use 2FA to log on with that computer and browser. Enabling 2FA and clicking on the Remember this browser will provide you with strong 2FA protection from malicious users trying to access your account, as long as they don't have access to your computer. You can do this on any private machine you regularly use. By setting Remember this browser, you get the added security of 2FA from computers you don't regularly use, and you get the convenience on not having to go through 2FA on your own computers.



The screenshot shows a web browser window with the address bar at `http://localhost:55832/Account/Verify`. The page title is "Verify - WebApplication224". The navigation bar is the same as the previous page. The main content area has the heading "Verify." and a form with the label "Code" followed by a text input field containing a single character "I". Below the input field is a checkbox labeled "Remember this browser?". At the bottom of the form is a "Submit" button. At the bottom of the page, there is a copyright notice: "© 2015 - WebApplication224".

- Log In.



Account lockout for protecting against brute force attacks

We recommend you use account lockout with 2FA. Once a user logs in (through local account or social account), each failed attempt at 2FA is stored, and if the maximum attempts (default is 5) is reached, the user is locked out for five minutes (you can set the lock out time with `DefaultAccountLockoutTimeSpan`). The following configures Account to be locked out for 10 min after 10 failed attempts.

```
services.Configure<IdentityOptions>(options =>
{
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
    options.Lockout.MaxFailedAccessAttempts = 10;
});
```

Next steps

- Once you publish your Web site to Azure Web App, you should reset the AppSecret in the Facebook developer portal.
- Set the Facebook AppId and AppSecret as application setting in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

Summary

ASP.NET Identity can be used to add two-factor authentication.

2.10.4 Authorization

Roles-Based Authorization

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

Claims-Based Authorization

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

Supporting Third Party Clients using OAuth 2.0

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.10.5 Data Protection

The ASP.NET data protection stack provides a simple, easy to use cryptographic API a developer can use to protect data, including key management and rotation.

Introduction

Web applications often need to store security-sensitive data. Windows provides DPAPI for desktop applications but this is unsuitable for web applications. The ASP.NET data protection stack provide a simple, easy to use cryptographic API a developer can use to protect data, including key management and rotation.

The ASP.NET 5 data protection stack is designed to serve as the long-term replacement for the <machineKey> element in ASP.NET 1.x - 4.x. It was designed to address many of the shortcomings of the old cryptographic stack while providing an out-of-the-box solution for the majority of use cases modern applications are likely to encounter.

Problem statement

The overall problem statement can be succinctly stated in a single sentence: I need to persist trusted information for later retrieval, but I do not trust the persistence mechanism. In web terms, this might be written as “I need to round-trip trusted state via an untrusted client.”

The canonical example of this is an authentication cookie or bearer token. The server generates an “I am Groot and have xyz permissions” token and hands it to the client. At some future date the client will present that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token. Thus the first requirement: authenticity (a.k.a. integrity, tamper-proofing).

Since the persisted state is trusted by the server, we anticipate that this state might contain information that is specific to the operating environment. This could be in the form of a file path, a permission, a handle or other indirect reference, or some other piece of server-specific data. Such information should generally not be disclosed to an untrusted client. Thus the second requirement: confidentiality.

Finally, since modern applications are componentized, what we've seen is that individual components will want to take advantage of this system without regard to other components in the system. For instance, if a bearer token component is using this stack, it should operate without interference from an anti-CSRF mechanism that might also be using the same stack. Thus the final requirement: isolation.

We can provide further constraints in order to narrow the scope of our requirements. We assume that all services operating within the cryptosystem are equally trusted and that the data does not need to be generated or consumed outside of the services under our direct control. Furthermore, we require that operations are as fast as possible since each request to the web service might go through the cryptosystem one or more times. This makes symmetric cryptography ideal for our scenario, and we can discount asymmetric cryptography until such a time that it is needed.

Design philosophy

We started by identifying problems with the existing stack. Once we had that, we surveyed the landscape of existing solutions and concluded that no existing solution quite had the capabilities we sought. We then engineered a solution based on several guiding principles.

- The system should offer simplicity of configuration. Ideally the system would be zero-configuration and developers could hit the ground running. In situations where developers need to configure a specific aspect (such as the key repository), consideration should be given to making those specific configurations simple.
- Offer a simple consumer-facing API. The APIs should be easy to use correctly and difficult to use incorrectly.
- Developers should not learn key management principles. The system should handle algorithm selection and key lifetime on the developer's behalf. Ideally the developer should never even have access to the raw key material.
- Keys should be protected at rest when possible. The system should figure out an appropriate default protection mechanism and apply it automatically.

With these principles in mind we developed a simple, [easy to use](#) data protection stack.

The ASP.NET 5 data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like [Windows CNG DPAPI](#) and [Azure Rights Management](#) are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET 5 data protection APIs for long-term protection of confidential data.

Audience

The data protection system is divided into five main packages. Various aspects of these APIs target three main audiences;

1. The [Consumer APIs Overview](#) target application and framework developers.

“I don’t want to learn about how the stack operates or about how it is configured. I simply want to perform some operation in as simple a manner as possible with high probability of using the APIs successfully.”
2. The [configuration APIs](#) target application developers and system administrators.

“I need to tell the data protection system that my environment requires non-default paths or settings.”
3. The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs would be limited to rare situations and experienced, security aware developers.

“I need to replace an entire component within the system because I have truly unique behavioral requirements. I am willing to learn uncommonly-used parts of the API surface in order to build a plugin that fulfills my requirements.”

Package Layout

The data protection stack consists of five packages.

- `Microsoft.AspNetCore.DataProtection.Interfaces` contains the basic `IDataProtectionProvider` and `IDataProtector` interfaces. It also contains useful extension methods that can assist working with these types (e.g., overloads of `IDataProtector.Protect`). See the consumer interfaces section for more information. If somebody else is responsible for instantiating the data protection system and you are simply consuming the APIs, you’ll want to reference `Microsoft.AspNetCore.DataProtection.Interfaces`.
- `Microsoft.AspNetCore.DataProtection` contains the core implementation of the data protection system, including the core cryptographic operations, key management, configuration, and extensibility. If you’re responsible for

instantiating the data protection system (e.g., adding it to an `IServiceCollection`) or modifying or extending its behavior, you'll want to reference `Microsoft.AspNetCore.DataProtection`.

- `Microsoft.AspNetCore.DataProtection.Extensions` contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains a simple "instantiate the system pointing at a specific key storage directory with no dependency injection setup" API (more info). It also contains extension methods for limiting the lifetime of protected payloads (more info).
- `Microsoft.AspNetCore.DataProtection.SystemWeb` can be installed into an existing ASP.NET 4.x application to redirect its `<machineKey>` operations to instead use the new data protection stack. See [compatibility](#) for more information.
- `Microsoft.AspNetCore.Cryptography.KeyDerivation` provides an implementation of the PBKDF2 password hashing routine and can be used by systems which need to handle user passwords securely. See [Password Hashing](#) for more information.

Getting Started with the Data Protection APIs

At its simplest protecting data is consists of the following steps:

1. Create a data protector from a data protection provider.
2. Call the `Protect` method with the data you want to protect.
3. Call the `Unprotect` method with the data you want to turn back into plain text.

Most frameworks such as ASP.NET or SignalR already configure the data protection system and add it to a service container you access via dependency injection. The following sample demonstrates configuring a service container for dependency injection and registering the data protection stack, receiving the data protection provider via DI, creating a protector and protecting then unprotecting data

```
1 using System;
2 using Microsoft.AspNetCore.DataProtection;
3 using Microsoft.Framework.DependencyInjection;
4
5 public class Program
6 {
7     public static void Main(string[] args)
8     {
9         // add data protection services
10        var serviceCollection = new ServiceCollection();
11        serviceCollection.AddDataProtection();
12        var services = serviceCollection.BuildServiceProvider();
13
14        // create an instance of MyClass using the service provider
15        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
16        instance.RunSample();
17    }
18
19    public class MyClass
20    {
21        IDataProtector _protector;
22
23        // the 'provider' parameter is provided by DI
24        public MyClass(IDataProtectionProvider provider)
25        {
26            _protector = provider.CreateProtector("Contoso.MyClass.v1");
27        }
28    }
```

```

29     public void RunSample()
30     {
31         Console.Write("Enter input: ");
32         string input = Console.ReadLine();
33
34         // protect the payload
35         string protectedPayload = _protector.Protect(input);
36         Console.WriteLine($"Protect returned: {protectedPayload}");
37
38         // unprotect the payload
39         string unprotectedPayload = _protector.Unprotect(protectedPayload);
40         Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
41     }
42 }
43
44
45 /*
46  * SAMPLE OUTPUT
47  *
48  * Enter input: Hello world!
49  * Protect returned: CfDJ8ICcgQwZZhlAltZT...OdfH66ilPnGmpCR5e441xQ
50  * Unprotect returned: Hello world!
51  */

```

When you create a protector you must provide one or more [Purpose Strings](#). A purpose string provides isolation between consumers, for example a protector created with a purpose string of “green” would not be able to unprotect data provided by a protector with a purpose of “purple”.

Tip: Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It is intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`.

A call to `Unprotect` will throw `CryptographicException` if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch `CryptographicException` instead of swallowing all exceptions.

Consumer APIs

Consumer APIs Overview

The `IDataProtectionProvider` and `IDataProtector` interfaces are the basic interfaces through which consumers use the data protection system. They are located in the `Microsoft.AspNetCore.DataProtection.Interfaces` package.

IDataProtectionProvider The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an `IDataProtector` by calling `IDataProtectionProvider.CreateProtector(purpose)`, where `purpose` is a string that describes the intended consumer use case. See [Purpose Strings](#) for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector The protector interface is returned by a call to `CreateProtector`, and it is this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the `Protect` method. The basic interface defines a method which converts `byte[]` -> `byte[]`, but there is also an overload (provided as an extension method) which converts `string` -> `string`. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for his use case. Irrespective of the overload chosen, the value returned by the `Protect` method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the `Unprotect` method. (There are `byte[]`-based and `string`-based overloads for developer convenience.) If the protected payload was generated by an earlier call to `Protect` on this same `IDataProtector`, the `Unprotect` method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different `IDataProtector`, the `Unprotect` method will throw `CryptographicException`.

The concept of same vs. different `IDataProtector` ties back to the concept of purpose. If two `IDataProtector` instances were generated from the same root `IDataProtectionProvider` but via different purpose strings in the call to `IDataProtectionProvider.CreateProtector`, then they are considered **different protectors**, and one will not be able to unprotect payloads generated by the other.

Consuming these interfaces For a DI-aware component, the intended usage is that the component take an `IDataProtectionProvider` parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

Note: Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the [Non DI Aware Scenarios](#) document for more information on getting an instance of an `IDataProtection` provider without going through DI.

The following sample demonstrates three concepts:

1. [Adding the data protection system](#) to the service container,
2. Using DI to receive an instance of an `IDataProtectionProvider`, and
3. Creating an `IDataProtector` from an `IDataProtectionProvider` and using it to protect and unprotect data.

```
1 using System;
2 using Microsoft.AspNet.DataProtection;
3 using Microsoft.Framework.DependencyInjection;
4
5 public class Program
6 {
7     public static void Main(string[] args)
8     {
9         // add data protection services
10        var serviceCollection = new ServiceCollection();
11        serviceCollection.AddDataProtection();
12        var services = serviceCollection.BuildServiceProvider();
13
14        // create an instance of MyClass using the service provider
15        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
16        instance.RunSample();
17    }
18
19    public class MyClass
20    {
21        IDataProtector _protector;
22
23        // the 'provider' parameter is provided by DI
24        public MyClass(IDataProtectionProvider provider)
25        {
```

```

26     _protector = provider.CreateProtector("Contoso.MyClass.v1");
27 }
28
29 public void RunSample()
30 {
31     Console.Write("Enter input: ");
32     string input = Console.ReadLine();
33
34     // protect the payload
35     string protectedPayload = _protector.Protect(input);
36     Console.WriteLine($"Protect returned: {protectedPayload}");
37
38     // unprotect the payload
39     string unprotectedPayload = _protector.Unprotect(protectedPayload);
40     Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
41 }
42 }
43 }
44
45 /*
46  * SAMPLE OUTPUT
47  *
48  * Enter input: Hello world!
49  * Protect returned: CfDJ8ICcgQwZZhlAlTzT...OdfH66ilPnGmpCR5e441xQ
50  * Unprotect returned: Hello world!
51  */

```

The package `Microsoft.AspNet.DataProtection.Interfaces` contains an extension method `IServiceProvider.GetDataProtector` as a developer convenience. It encapsulates as a single operation both retrieving an `IDataProtectionProvider` from the service provider and calling `IDataProtectionProvider.CreateProtector`. The following sample demonstrates its usage.

```

1  using System;
2  using Microsoft.AspNet.DataProtection;
3  using Microsoft.Framework.DependencyInjection;
4
5  public class Program
6  {
7      public static void Main(string[] args)
8      {
9          // add data protection services
10         var serviceCollection = new ServiceCollection();
11         serviceCollection.AddDataProtection();
12         var services = serviceCollection.BuildServiceProvider();
13
14         // get an IDataProtector from the IServiceProvider
15         var protector = services.GetDataProtector("Contoso.Example.v2");
16         Console.Write("Enter input: ");
17         string input = Console.ReadLine();
18
19         // protect the payload
20         string protectedPayload = protector.Protect(input);
21         Console.WriteLine($"Protect returned: {protectedPayload}");
22
23         // unprotect the payload
24         string unprotectedPayload = protector.Unprotect(protectedPayload);
25         Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
26     }

```

}

Tip: Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It is intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`.

A call to `Unprotect` will throw `CryptographicException` if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch `CryptographicException` instead of swallowing all exceptions.

Purpose Strings

Components which consume `IDataProtectionProvider` must pass a unique *purposes* parameter to the `CreateProtector` method. The *purposes* parameter is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.



In the diagram above `IDataProtector` instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

Tip: Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.

A Contoso-authored component which is responsible for minting bearer tokens might use `Contoso.Security.BearerToken` as its purpose string. Or - even better - it might use `Contoso.Security.BearerToken.v1` as its purpose string. Appending the version number allows a future version to use `Contoso.Security.BearerToken.v2` as its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the *purposes* parameter to `CreateProtector` is a string array, the above could have been instead specified as [`"Contoso.Security.BearerToken", "v1"`]. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

Warning: Components should not allow untrusted user input to be the sole source of input for the purposes chain. For example, consider a component `Contoso.Messaging.SecureMessage` which is responsible for storing secure messages. If the secure messaging component were to call `CreatePurpose([username])`, then a malicious user might create an account with username `"Contoso.Security.BearerToken"` in an attempt to get the component to call `CreatePurpose(["Contoso.Security.BearerToken"])`, thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.

A better purposes chain for the messaging component would be `CreatePurpose(["Contoso.Messaging.SecureMessage", "User: username"])`, which provides proper isolation.

The isolation provided by and behaviors of `IDataProtectionProvider`, `IDataProtector`, and purposes are as follows:

- For a given `IDataProtectionProvider` object, the `CreatePurpose` method will create an `IDataProtector` object uniquely tied to both the `IDataProtectionProvider` object which created it and the purposes parameter which was passed into the method.
- The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.
- Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.
- Two `IDataProtector` objects are equivalent if and only if they are created from equivalent `IDataProtectionProvider` objects with equivalent purposes parameters.
- For a given `IDataProtector` object, a call to `Unprotect(protectedData)` will return the original unprotectedData if and only if `protectedData := Protect(unprotectedData)` for an equivalent `IDataProtector` object.

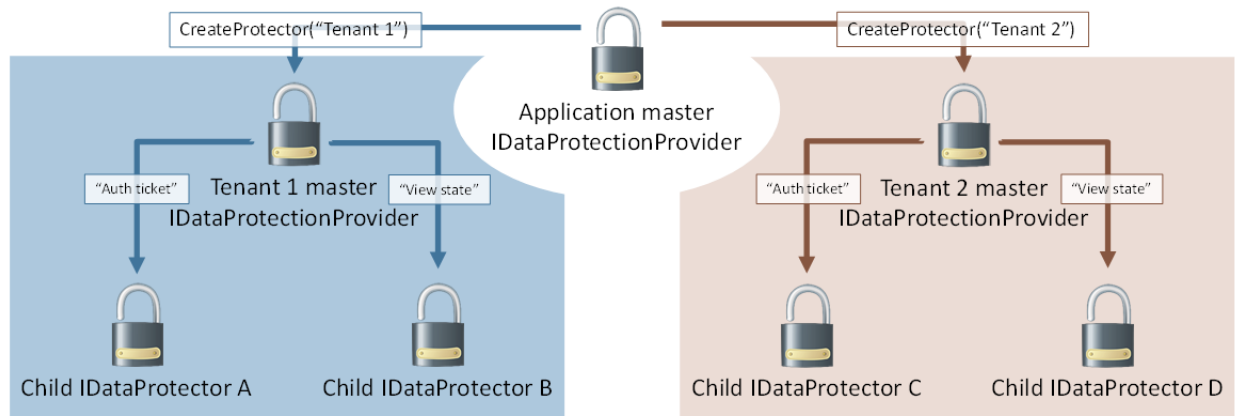
Note: We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system is not intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

Purpose hierarchy and multi-tenancy

Since an `IDataProtector` is also implicitly an `IDataProtectionProvider`, purposes can be chained together. In this sense `provider.CreateProtector(["purpose1", "purpose2"])` is equivalent to `provider.CreateProtector("purpose1").CreateProtector("purpose2")`.

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of `Contoso.Messaging.SecureMessage`, the `SecureMessage` component can call `provider.CreateProtector("Contoso.Messaging.SecureMessage")` once upfront and cache the result into a private `_myProvider` field. Future protectors can then be created via calls to `_myProvider.CreateProtector("User: username")`, and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls `provider.CreateProtector("Tenant 1")` and `provider.CreateProtector("Tenant 2")` to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically this is represented as below.



Warning: This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a tenant can execute arbitrary code, he could perform private reflection to break the isolation guarantees, or he could just read the master keying material directly and derive whatever subkeys he desires.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it is actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to *isolate individual applications* from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Password Hashing

The data protection code base includes a package `Microsoft.AspNet.Cryptography.KeyDerivation` which contains cryptographic key derivation functions. This package is technically its own standalone component, has no dependencies on the rest of the data protection system, and can be used completely independently. (Though this package is technically not part of the data protection system, its source exists alongside the data protection code base as a convenience.)

The package currently offers a method `KeyDerivation.Pbkdf2` which allows hashing a password using the [PBKDF2 algorithm](#). This API is very similar to the .NET Framework's existing `Rfc2898DeriveBytes` type, but there are three important distinctions:

1. The `KeyDerivation.Pbkdf2` method supports consuming multiple PRFs (currently HMACSHA1, HMACSHA256, and HMACSHA512), whereas the `Rfc2898DeriveBytes` type only supports HMACSHA1.
2. The `KeyDerivation.Pbkdf2` method detects the current operating system and attempts to choose the most optimized implementation of the routine, providing much better performance in certain cases. (On Windows 8, it offers around 10x the throughput of `Rfc2898DeriveBytes`.)
3. The `KeyDerivation.Pbkdf2` method requires the caller to specify all parameters (salt, PRF, and iteration count). The `Rfc2898DeriveBytes` type provides default values for these.

```

1 using System;
2 using System.Security.Cryptography;
3 using Microsoft.AspNet.Cryptography.KeyDerivation;
4
5 public class Program
6 {

```

```

7 public static void Main(string[] args)
8 {
9     Console.WriteLine("Enter a password: ");
10    string password = Console.ReadLine();
11
12    // generate a 128-bit salt using a secure PRNG
13    byte[] salt = new byte[128 / 8];
14    using (var rng = RandomNumberGenerator.Create())
15    {
16        rng.GetBytes(salt);
17    }
18    Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");
19
20    // derive a 256-bit subkey (use HMACSHA1 with 10,000 iterations)
21    string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
22        password: password,
23        salt: salt,
24        prf: KeyDerivationPrf.HMACSHA1,
25        iterationCount: 10000,
26        numBytesRequested: 256 / 8));
27    Console.WriteLine($"Hashed: {hashed}");
28 }
29 }
30
31 /*
32  * SAMPLE OUTPUT
33  *
34  * Enter a password: Xtw9NMgx
35  * Salt: NZsP6NnmfBuYeJrrAKNuVQ==
36  * Hashed: /OOoOer10+tGwTRDTrQSoeCxVTfr6dtYly7d0cPxIak=
37  */

```

See also the source code for ASP.NET Identity's [PasswordHasher](#) type for a real-world use case.

Limiting the lifetime of protected payloads

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It is certainly possible for the developer to create his own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package `Microsoft.AspNetCore.DataProtection.Extensions` contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs hang off of the `ITimeLimitedDataProtector` type.

API usage The `ITimeLimitedDataProtector` interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an `ITimeLimitedDataProtector`, you'll first need an instance of a regular [IDataProtector](#) constructed with a specific purpose. Once the `IDataProtector` instance is available, call the `IDataProtector.ToTimeLimitedDataProtector` extension method to get back a protector with built-in expiration capabilities.

`ITimeLimitedDataProtector` exposes the following API surface and extension methods:

- `CreateProtector(string purpose) : ITimeLimitedDataProtector` This API is similar to the existing `IDataProtectionProvider.CreateProtector` in that it can be used to create [purpose chains](#) from a root time-limited protector.

- `Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]`
- `Protect(byte[] plaintext, TimeSpan lifetime) : byte[]`
- `Protect(byte[] plaintext) : byte[]`
- `Protect(string plaintext, DateTimeOffset expiration) : string`
- `Protect(string plaintext, TimeSpan lifetime) : string`
- `Protect(string plaintext) : string`

In addition to the core `Protect` methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a `DateTimeOffset`) or as a relative time (from the current system time, via a `TimeSpan`). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

- `Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]`
- `Unprotect(byte[] protectedData) : byte[]`
- `Unprotect(string protectedData, out DateTimeOffset expiration) : string`
- `Unprotect(string protectedData) : string`

The `Unprotect` methods return the original unprotected data. If the payload hasn't yet expired, the absolute expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the `Unprotect` method will throw `CryptographicException`.

Warning: It is not advised to use these APIs to protect payloads which require long-term or indefinite persistence. “Can I afford for the protected payloads to be permanently unrecoverable after a month?” can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the [non-DI code paths](#) for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the `Microsoft.AspNet.DataProtection.Extensions` package.

```
1 using System;
2 using System.IO;
3 using System.Threading;
4 using Microsoft.AspNet.DataProtection;
5
6 public class Program
7 {
8     public static void Main(string[] args)
9     {
10         // create a protector for my application
11
12         var provider = new DataProtectionProvider(new DirectoryInfo(@"c:\myapp-keys\"));
13         var baseProtector = provider.CreateProtector("Contoso.TimeLimitedSample");
14
15         // convert the normal protector into a time-limited protector
16         var timeLimitedProtector = baseProtector.ToTimeLimitedDataProtector();
17
18         // get some input and protect it for five seconds
19         Console.Write("Enter input: ");
20         string input = Console.ReadLine();
21         string protectedData = timeLimitedProtector.Protect(input, lifetime: TimeSpan.FromSeconds(5));
22         Console.WriteLine($"Protected data: {protectedData}");
23
24         // unprotect it to demonstrate that round-tripping works properly
25         string roundtripped = timeLimitedProtector.Unprotect(protectedData);
```

```

26         Console.WriteLine($"Round-tripped data: {roundtripped}");
27
28         // wait 6 seconds and perform another unprotect, demonstrating that the payload self-expires
29         Console.WriteLine("Waiting 6 seconds...");
30         Thread.Sleep(6000);
31         timeLimitedProtector.Unprotect(protectedData);
32     }
33 }
34
35 /*
36  * SAMPLE OUTPUT
37  *
38  * Enter input: Hello!
39  * Protected data: CfDJ8Hu5z0zwxn...nLk7Ok
40  * Round-tripped data: Hello!
41  * Waiting 6 seconds...
42  * <<throws CryptographicException with message 'The payload expired at ...'>>
43
44  */

```

Unprotecting payloads whose keys have been revoked

The ASP.NET 5 data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like [Windows CNG DPAPI](#) and [Azure Rights Management](#) are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET 5 data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so `IDataProtector.Unprotect` can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as `IDataProtector.Unprotect` will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having `Unprotect` throw could lead to unacceptable data loss.

IPersistedDataProtector To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an `IPersistedDataProtector` type. To get an instance of `IPersistedDataProtector`, simply get an instance of `IDataProtector` in the normal fashion and try casting the `IDataProtector` to `IPersistedDataProtector`.

Note: Not all `IDataProtector` instances can be cast to `IPersistedDataProtector`. Developers should use the C# `as` operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case appropriately.

`IPersistedDataProtector` exposes the following API surface:

```

DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
    out bool requiresMigration, out bool wasRevoked) : byte[]

```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There is no string-based overload. The two out parameters are as follows.

- `requiresMigration`: will be set to true if the key used to protect this payload is no longer the active default key, e.g., the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on his business needs.

- wasRevoked: will be set to true if the key used to protect this payload was revoked.

Warning: Exercise extreme caution when passing `ignoreRevocationErrors: true` to the `DangerousUnprotect` method. If after calling this method the `wasRevoked` value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case only continue operating on the unprotected payload if you have some separate assurance that it is authentic, e.g. that it's coming from a secure database rather than being sent by an untrusted web client.

```
1 using System;
2 using System.IO;
3 using System.Text;
4 using Microsoft.AspNet.DataProtection;
5 using Microsoft.AspNet.DataProtection.KeyManagement;
6 using Microsoft.Framework.DependencyInjection;
7
8 public class Program
9 {
10     public static void Main(string[] args)
11     {
12         var serviceCollection = new ServiceCollection();
13         serviceCollection.AddDataProtection();
14         serviceCollection.ConfigureDataProtection(configure =>
15         {
16             // point at a specific folder and use DPAPI to encrypt keys
17             configure.PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"));
18             configure.ProtectKeysWithDpapi();
19         });
20         var services = serviceCollection.BuildServiceProvider();
21
22         // get a protector and perform a protect operation
23         var protector = services.GetDataProtector("Sample.DangerousUnprotect");
24         Console.WriteLine("Input: ");
25         byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
26         var protectedData = protector.Protect(input);
27         Console.WriteLine($"Protected payload: {Convert.ToBase64String(protectedData)}");
28
29         // demonstrate that the payload round-trips properly
30         var roundTripped = protector.Unprotect(protectedData);
31         Console.WriteLine($"Round-tripped payload: {Encoding.UTF8.GetString(roundTripped)}");
32
33         // get a reference to the key manager and revoke all keys in the key ring
34         var keyManager = services.GetService<IKeyManager>();
35         Console.WriteLine("Revoking all keys in the key ring...");
36         keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");
37
38         // try calling Protect - this should throw
39         Console.WriteLine("Calling Unprotect...");
40         try
41         {
42             var unprotectedPayload = protector.Unprotect(protectedData);
43             Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
44         }
45         catch (Exception ex)
46         {
47             Console.WriteLine($"{{ex.GetType().Name}}: {{ex.Message}}");
48         }
49     }
50 }
```

```

50         // try calling DangerousUnprotect
51         Console.WriteLine("Calling DangerousUnprotect...");
52         try
53         {
54             IPersistedDataProtector persistedProtector = protector as IPersistedDataProtector;
55             if (persistedProtector == null)
56             {
57                 throw new Exception("Can't call DangerousUnprotect.");
58             }
59
60             bool requiresMigration, wasRevoked;
61             var unprotectedPayload = persistedProtector.DangerousUnprotect(
62                 protectedData: protectedData,
63                 ignoreRevocationErrors: true,
64                 requiresMigration: out requiresMigration,
65                 wasRevoked: out wasRevoked);
66             Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
67             Console.WriteLine($"Requires migration = {requiresMigration}, was revoked = {wasRevoked}");
68         }
69         catch (Exception ex)
70         {
71             Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
72         }
73     }
74 }
75
76 /*
77  * SAMPLE OUTPUT
78  *
79  * Input: Hello!
80  * Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
81  * Round-tripped payload: Hello!
82  * Revoking all keys in the key ring...
83  * Calling Unprotect...
84  * CryptographicException: The key {...} has been revoked.
85  * Calling DangerousUnprotect...
86  * Unprotected payload: Hello!
87  * Requires migration = True, was revoked = True
88  */

```

Configuration

Configuring Data Protection

When the data protection system is initialized it applies some *default settings* based on the operational environment. These settings are generally good for applications running on a single machine. There are some cases where a developer may want to change these (perhaps because his application is spread across multiple machines or for compliance reasons), and for these scenarios the data protection system offers a rich configuration API. There is an extension method `ConfigureDataProtection` hanging off of `IServiceCollection`. This method takes a callback, and the parameter passed to the callback object allows configuration of the system. For instance, to store keys at a UNC share instead of `%LOCALAPPDATA%` (the default), configure the system as follows:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection();
    services.ConfigureDataProtection(configure =>

```

```
{
    configure.PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));
});
}
```

Warning: If you change the key persistence location, the system will no longer automatically encrypt keys at rest since it doesn't know whether DPAPI is an appropriate encryption mechanism.

You can configure the system to protect keys at rest by calling any of the `ProtectKeysWith*` configuration APIs. Consider the example below, which stores keys at a UNC share and encrypts those keys at rest with a specific X.509 certificate.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection();
    services.ConfigureDataProtection(configure =>
    {
        configure.PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));
        configure.ProtectKeysWithCertificate("thumbprint");
    });
}
```

See [key encryption at rest](#) for more examples and for discussion on the built-in key encryption mechanisms.

To configure the system to use a default key lifetime of 14 days instead of 90 days, consider the following example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection();
    services.ConfigureDataProtection(configure =>
    {
        configure.SetDefaultKeyLifetime(TimeSpan.FromDays(14));
    });
}
```

By default the data protection system isolates applications from one another, even if they're sharing the same physical key repository. This prevents the applications from understanding each other's protected payloads. To share protected payloads between two different applications, configure the system passing in the same application name for both applications as in the below example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection();
    services.ConfigureDataProtection(configure =>
    {
        configure.SetApplicationName("my application");
    });
}
```

Finally, you may have a scenario where you do not want an application to automatically roll keys as they approach expiration. One example of this might be applications set up in a primary / secondary relationship, where only the primary application is responsible for key management concerns, and all secondary applications simply have a read-only view of the key ring. The secondary applications can be configured to treat the key ring as read-only by configuring the system as below:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection();
    services.ConfigureDataProtection(configure =>
```



```
{
    configure.DisableAutomaticKeyGeneration();
});
}
```

Per-application isolation When the data protection system is provided by an ASP.NET host, it will automatically isolate applications from one another, even if those applications are running under the same worker process account and are using the same master keying material. This is somewhat similar to the `IsolateApps` modifier from `System.Web`'s `<machineKey>` element.

The isolation mechanism works by considering each application on the local machine as a unique tenant, thus the `IDataProtector` rooted for any given application automatically includes the application ID as a discriminator. The application's unique ID comes from one of two places.

1. If the application is hosted in IIS, the unique identifier is the application's configuration path. If an application is deployed in a farm environment, this value should be stable assuming that the IIS environments are configured similarly across all machines in the farm.
2. If the application is not hosted in IIS, the unique identifier is the physical path of the application.

The unique identifier is designed to survive resets - both of the individual application and of the machine itself.

This isolation mechanism assumes that the applications are not malicious. A malicious application can always impact any other application running under the same worker process account. In a shared hosting environment where applications are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between applications, including separating the applications' underlying key repositories.

If the data protection system is not provided by an ASP.NET host (e.g., if the developer instantiates it himself via the `DataProtectionProvider` concrete type), application isolation is disabled by default, and all applications backed by the same keying material can share payloads as long as they provide the appropriate purposes. To provide application isolation in this environment, call the `SetApplicationName` method on the configuration object, see the [code sample](#) above.

Changing algorithms The data protection stack allows changing the default algorithm used by newly-generated keys. The simplest way to do this is to call `UseCryptographicAlgorithms` from the configuration callback, as in the below example.

```
services.ConfigureDataProtection(configure =>
{
    configure.UseCryptographicAlgorithms(new AuthenticatedEncryptionOptions()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
});
```

The default `EncryptionAlgorithm` and `ValidationAlgorithm` are `AES-256-CBC` and `HMACSHA256`, respectively. The default policy can be set by a system administrator via [Machine Wide Policy](#), but an explicit call to `UseCryptographicAlgorithms` will override the default policy.

Calling `UseCryptographicAlgorithms` will allow the developer to specify the desired algorithm (from a predefined built-in list), and the developer does not need to worry about the implementation of the algorithm. For instance, in the scenario above the data protection system will attempt to use the CNG implementation of AES if running on Windows, otherwise it will fall back to the managed `System.Security.Cryptography.Aes` class.

The developer can manually specify an implementation if desired via a call to `UseCustomCryptographicAlgorithms`, as show in the below examples.

Tip: Changing algorithms does not affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms To specify custom managed algorithms, create a `ManagedAuthenticatedEncryptionOptions` instance that points to the implementation types.

```
services.ConfigureDataProtection(configure =>
{
    configure.UseCustomCryptographicAlgorithms(new ManagedAuthenticatedEncryptionOptions()
    {
        // a type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // specified in bits
        EncryptionAlgorithmKeySize = 256,

        // a type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
});
```

Generally the `*Type` properties must point to concrete, instantiable (via a public parameterless ctor) implementations of `SymmetricAlgorithm` and `KeyedHashAlgorithm`, though the system special-cases some values like `typeof(Aes)` for convenience.

Note: The `SymmetricAlgorithm` must have a key length of 128 bits and a block size of 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The `KeyedHashAlgorithm` must have a digest size of ≥ 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The `KeyedHashAlgorithm` is not strictly required to be HMAC.

Specifying custom Windows CNG algorithms To specify a custom Windows CNG algorithm using CBC-mode encryption + HMAC validation, create a `CngCbcAuthenticatedEncryptionOptions` instance that contains the algorithmic information.

```
services.ConfigureDataProtection(configure =>
{
    configure.UseCustomCryptographicAlgorithms(new CngCbcAuthenticatedEncryptionOptions()
    {
        // passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // specified in bits
        EncryptionAlgorithmKeySize = 256,

        // passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
});
```

Note: The symmetric block cipher algorithm must have a key length of 128 bits and a block size of 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of ≥ 128 bits and must support being opened with the `BCRYPT_ALG_HANDLE_HMAC_FLAG` flag. The `*Provider` properties

can be set to null to use the default provider for the specified algorithm. See the [BCryptOpenAlgorithmProvider](#) documentation for more information.

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption + validation, create a `CngGcmAuthenticatedEncryptionOptions` instance that contains the algorithmic information.

```
services.ConfigureDataProtection(configure =>
{
    configure.UseCustomCryptographicAlgorithms(new CngGcmAuthenticatedEncryptionOptions()
    {
        // passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // specified in bits
        EncryptionAlgorithmKeySize = 256
    });
});
```

Note: The symmetric block cipher algorithm must have a key length of 128 bits and a block size of exactly 128 bits, and it must support GCM encryption. The `EncryptionAlgorithmProvider` property can be set to null to use the default provider for the specified algorithm. See the [BCryptOpenAlgorithmProvider](#) documentation for more information.

Specifying other custom algorithms Though not exposed as a first-class API, the data protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it is possible to keep all keys contained within an HSM and to provide a custom implementation of the core encryption and decryption routines. See `IAuthenticatedEncryptorConfiguration` in the core cryptography extensibility section for more information.

See also [Non DI Aware Scenarios](#)

[Machine Wide Policy](#)

Default Settings

Key Management The system tries to detect its operational environment and provide good zero-configuration behavioral defaults. The heuristic used is as follows.

1. If the system is being hosted in Azure Web Sites, keys are persisted to the “%HOME%\ASP.NET\DataProtection-Keys” folder. This folder is backed by network storage and is synchronized across all machines hosting the application. Keys are not protected at rest.
2. If the user profile is available, keys are persisted to the “%LOCALAPPDATA%\ASP.NET\DataProtection-Keys” folder. Additionally, if the operating system is Windows, they’ll be encrypted at rest using DPAPI.
3. If the application is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.
4. If none of these conditions matches, keys are not persisted outside of the current process. When the process shuts down, all generated keys will be lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should good defaults for most applications similar to how the ASP.NET <machineKey> auto-generation routines worked in the past. The final, fall back option is the only scenario that truly requires the developer to specify [configuration](#) upfront if he wants key persistence, but this fall-back would only occur in rare situations.

Warning: If the developer overrides this heuristic and points the data protection system at a specific key repository, automatic encryption of keys at rest will be disabled. At rest protection can be re-enabled via [configuration](#).

Key Lifetime Keys by default have a 90-day lifetime. When a key expires, the system will automatically generate a new key and set the new key as the active key. As long as retired keys remain on the system you will still be able to decrypt any data protected with them. See [key lifetime](#) for more information.

Default Algorithms The default payload protection algorithm used is AES-256-CBC for confidentiality and HMAC-SHA256 for authenticity. A 512-bit master key, rolled every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See [subkey derivation](#) for more information.

Machine Wide Policy

When running on Windows, the data protection system has limited support for setting default machine-wide policy for all applications which consume data protection. The general idea is that an administrator might wish to change some default setting (such as algorithms used or key lifetime) without needing to manually update every application on the machine.

Warning: The system administrator can set default policy, but he cannot enforce it. The application developer can always override any value with one of his own choosing. The default policy only affects applications where the developer has not specified an explicit value for some particular setting.

Setting default policy To set default policy, an administrator can set known values in the system registry under the following key.

Reg key: HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNet.DataProtection

If you're on a 64-bit operating system and want to affect the behavior of 32-bit applications, remember also to configure the Wow6432Node equivalent of the above key.

The supported values are:

- `EncryptionType` [string] - specifies which algorithms should be used for data protection. This value must be "CNG-CBC", "CNG-GCM", or "Managed" and is described in more detail [below](#).
- `DefaultKeyLifetime` [DWORD] - specifies the lifetime for newly-generated keys. This value is specified in days and must be 7.
- `KeyEscrowSinks` [string] - specifies the types which will be used for key escrow. This value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly qualified name of a type which implements `IKeyEscrowSink`.

Encryption types If `EncryptionType` is "CNG-CBC", the system will be configured to use a CBC-mode symmetric block cipher for confidentiality and HMAC for authenticity with services provided by Windows CNG (see [Specifying custom Windows CNG algorithms](#) for more details). The following additional values are supported, each of which corresponds to a property on the `CngCbcAuthenticatedEncryptionOptions` type:

- `EncryptionAlgorithm` [string] - the name of a symmetric block cipher algorithm understood by CNG. This algorithm will be opened in CBC mode.
- `EncryptionAlgorithmProvider` [string] - the name of the CNG provider implementation which can produce the algorithm `EncryptionAlgorithm`.

- `EncryptionAlgorithmKeySize` [DWORD] - the length (in bits) of the key to derive for the symmetric block cipher algorithm.
- `HashAlgorithm` [string] - the name of a hash algorithm understood by CNG. This algorithm will be opened in HMAC mode.
- `HashAlgorithmProvider` [string] - the name of the CNG provider implementation which can produce the algorithm `HashAlgorithm`.

If `EncryptionType` is “CNG-GCM”, the system will be configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see [Specifying custom Windows CNG algorithms](#) for more details). The following additional values are supported, each of which corresponds to a property on the `CngGcmAuthenticatedEncryptionOptions` type:

- `EncryptionAlgorithm` [string] - the name of a symmetric block cipher algorithm understood by CNG. This algorithm will be opened in Galois/Counter Mode.
- `EncryptionAlgorithmProvider` [string] - the name of the CNG provider implementation which can produce the algorithm `EncryptionAlgorithm`.
- `EncryptionAlgorithmKeySize` [DWORD] - the length (in bits) of the key to derive for the symmetric block cipher algorithm.

If `EncryptionType` is “Managed”, the system will be configured to use a managed `SymmetricAlgorithm` for confidentiality and `KeyedHashAlgorithm` for authenticity (see [Specifying custom managed algorithms](#) for more details). The following additional values are supported, each of which corresponds to a property on the `ManagedAuthenticatedEncryptionOptions` type:

- `EncryptionAlgorithmType` [string] - the assembly-qualified name of a type which implements `SymmetricAlgorithm`.
- `EncryptionAlgorithmKeySize` [DWORD] - the length (in bits) of the key to derive for the symmetric encryption algorithm.
- `ValidationAlgorithmType` [string] - the assembly-qualified name of a type which implements `KeyedHashAlgorithm`.

If `EncryptionType` has any other value (other than null / empty), the data protection system will throw an exception at startup.

Warning: When configuring a default policy setting that involves type names (`EncryptionAlgorithmType`, `ValidationAlgorithmType`, `KeyEscrowSinks`), the types must be available to the application. In practice, this means that for applications running on Desktop CLR, the assemblies which contain these types should be GACed. For ASP.NET 5 applications running on Core CLR, the packages which contain these types should be referenced in `project.json`.

Non DI Aware Scenarios

The data protection system is normally designed to be added to a service container and to be provided to dependent components via a DI mechanism. However, there may be some cases where this is not feasible, especially when importing the system into an existing application.

To support these scenarios the package `Microsoft.AspNet.DataProtection.Extensions` provides a concrete type `DataProtectionProvider` which offers a simple way to use the data protection system without going through DI-specific code paths. The type itself implements `IDataProtectionProvider`, and constructing it is as easy as providing a `DirectoryInfo` where this provider’s cryptographic keys should be stored.

For example:

```

1 using System;
2 using System.IO;
3 using Microsoft.AspNet.DataProtection;
4
5 public class Program
6 {
7     public static void Main(string[] args)
8     {
9         // get the path to %LOCALAPPDATA%\myapp-keys
10        string destFolder = Path.Combine(
11            Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
12            "myapp-keys");
13
14        // instantiate the data protection system at this folder
15        var dataProtectionProvider = new DataProtectionProvider(
16            new DirectoryInfo(destFolder));
17
18        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
19        Console.Write("Enter input: ");
20        string input = Console.ReadLine();
21
22        // protect the payload
23        string protectedPayload = protector.Protect(input);
24        Console.WriteLine($"Protect returned: {protectedPayload}");
25
26        // unprotect the payload
27        string unprotectedPayload = protector.Unprotect(protectedPayload);
28        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
29    }
30 }
31
32 /*
33  * SAMPLE OUTPUT
34  *
35  * Enter input: Hello world!
36  * Protect returned: CfDJ8FWbAn6...ch3hAPm1NJA
37  * Unprotect returned: Hello world!
38  */

```

Warning: By default the `DataProtectionProvider` concrete type does not encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share, in which case the data protection system cannot automatically deduce an appropriate at-rest key encryption mechanism. Additionally, the `DataProtectionProvider` concrete type does not *isolate applications* by default, so all applications pointed at the same key directory can share payloads as long as their purpose parameters match.

The application developer can address both of these if desired. The `DataProtectionProvider` constructor accepts an *optional configuration callback* which can be used to tweak the behaviors of the system. The sample below demonstrates restoring isolation via an explicit call to `SetApplicationName`, and it also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption instead via a call to *`ProtectKeysWithCertificate`*.

```

1 using System;
2 using System.IO;
3 using Microsoft.AspNet.DataProtection;
4
5 public class Program

```

```

6 {
7     public static void Main(string[] args)
8     {
9         // get the path to %LOCALAPPDATA%\myapp-keys
10        string destFolder = Path.Combine(
11            Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
12            "myapp-keys");
13
14        // instantiate the data protection system at this folder
15        var dataProtectionProvider = new DataProtectionProvider(
16            new DirectoryInfo(destFolder),
17            configuration =>
18            {
19                configuration.SetApplicationName("my app name");
20                configuration.ProtectKeysWithDpapi();
21            });
22
23        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
24        Console.Write("Enter input: ");
25        string input = Console.ReadLine();
26
27        // protect the payload
28        string protectedPayload = protector.Protect(input);
29        Console.WriteLine($"Protect returned: {protectedPayload}");
30
31        // unprotect the payload
32        string unprotectedPayload = protector.Unprotect(protectedPayload);
33        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
34    }
35 }

```

Tip: Instances of the `DataProtectionProvider` concrete type are expensive to create. If an application maintains multiple instances of this type and if they're all pointing at the same key storage directory, application performance may be degraded. The intended usage is that the application developer instantiate this type once then keep reusing this single reference as much as possible. The `DataProtectionProvider` type and all `IDataProtector` instances created from it are thread-safe for multiple callers.

Extensibility APIs

Core cryptography extensibility

Warning: Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There is generally one `IAuthenticatedEncryptor` per key, and the `IAuthenticatedEncryptor` instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

- `Decrypt(ArraySegment<byte> ciphertext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`
- `Encrypt(ArraySegment<byte> plaintext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except `ArgumentNullException` and similar) should be homogenized to `CryptographicException`.

Note: The `IAuthenticatedEncryptor` instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

IAuthenticatedEncryptorDescriptor The `IAuthenticatedEncryptorDescriptor` interface represents a type that knows how to create an *IAuthenticatedEncryptor* instance. Its API is as follows.

- `CreateEncryptorInstance() : IAuthenticatedEncryptor`
- `ExportToXml() : XmlSerializedDescriptorInfo`

Like `IAuthenticatedEncryptor`, an instance of `IAuthenticatedEncryptorDescriptor` is assumed to wrap one specific key. This means that for any given `IAuthenticatedEncryptorDescriptor` instance, any authenticated encryptors created by its `CreateEncryptorInstance` method should be considered equivalent, as in the below code sample.

```
// we have an IAuthenticatedEncryptorDescriptor instance
IAuthenticatedEncryptorDescriptor descriptor = ...;

// get an encryptor instance and perform an authenticated encryption operation
ArraySegment<byte> plaintext = new ArraySegment<byte>(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = descriptor.CreateEncryptorInstance();
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated decryption operation
var encryptor2 = descriptor.CreateEncryptorInstance();
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);

// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

XML Serialization The primary difference between `IAuthenticatedEncryptor` and `IAuthenticatedEncryptorDescriptor` is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an `IAuthenticatedEncryptor` whose implementation relies on `SymmetricAlgorithm` and `KeyedHashAlgorithm`. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (e.g., it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset. The descriptor can be serialized via its `ExportToXml` routine. This routine returns an `XmlSerializedDescriptorInfo` which contains two properties: the `XElement` representation of the descriptor and the `Type` which represents an *IAuthenticatedEncryptorDescriptorDeserializer* which can be used to resurrect this descriptor given the corresponding `XElement`.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name "requiresEncryption" (xmlns "<http://schemas.asp.net/2015/03/dataProtection>"), value "true".

Tip: There's a helper API for setting this attribute. Call the `extension method XElement.MarkAsRequiresEncryption()` located in namespace `Microsoft.AspNet.DataProtection.AuthenticatedEncryption.ConfigurationModel`.

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM will not expose the material in plaintext form. Instead, the descriptor might write out the key-wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an **IAuthenticatedEncryptorDescriptor** instance from an **XElement**. It exposes a single method:

- **ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor**

The **ImportFromXml** method takes the **XElement** that was returned by *IAuthenticatedEncryptorDescriptor.ExportToXml* and creates an equivalent of the original **IAuthenticatedEncryptorDescriptor**.

Types which implement **IAuthenticatedEncryptorDescriptorDeserializer** should have one of the following two public constructors:

- **.ctor(IServiceProvider)**
- **.ctor()**

Note: The **IServiceProvider** passed to the constructor may be null.

IAuthenticatedEncryptorConfiguration The **IAuthenticatedEncryptorConfiguration** interface represents a type which knows how to create *IAuthenticatedEncryptorDescriptor* instances. It exposes a single API.

- **CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor**

Think of **IAuthenticatedEncryptorConfiguration** as the top-level factory. The configuration serves as a template. It wraps algorithmic information (e.g., this configuration produces descriptors with an AES-128-GCM master key), but it is not yet associated with a specific key.

When **CreateNewDescriptor** is called, fresh key material is created solely for this call, and a new **IAuthenticatedEncryptorDescriptor** is produced which wraps this key material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to **CreateNewDescriptor** should never create equivalent **IAuthenticatedEncryptorDescriptor** instances.

The **IAuthenticatedEncryptorConfiguration** type serves as the entry point for key creation routines such as *automatic key rolling*. To change the implementation for all future keys, register a singleton **IAuthenticatedEncryptorConfiguration** in the service container.

Key management extensibility

Tip: Read the *key management* section before reading this section, as it explains some of the fundamental concepts behind these APIs.

Warning: Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key The **IKey** interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of “cryptographic key material”. A key has the following properties:

- Activation, creation, and expiration dates
- Revocation status
- Key identifier (a GUID)

Additionally, `IKey` exposes a `CreateEncryptorInstance` method which can be used to create an *`IAuthenticatedEncryptor`* instance tied to this key.

Note: There is no API to retrieve the raw cryptographic material from an `IKey` instance.

IKeyManager The `IKeyManager` interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

- Create a new key and persist it to storage.
- Get all keys from storage.
- Revoke one or more keys and persist the revocation information to storage.

Warning: Writing an `IKeyManager` is a very advanced task, and the majority of developers should not attempt it. Instead, most developers should take advantage of the facilities offered by the *`XmlKeyManager`* class.

XmlKeyManager The `XmlKeyManager` type is the in-box concrete implementation of `IKeyManager`. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, *`XElement`*).

`XmlKeyManager` depends on several other components in the course of fulfilling its tasks:

- *`IAuthenticatedEncryptorConfiguration`*, which dictates the algorithms used by new keys.
- *`IXmlRepository`*, which controls where keys are persisted in storage.
- *`IXmlEncryptor`* [optional], which allows encrypting keys at rest.
- *`IKeyEscrowSink`* [optional], which provides key escrow services.

Below are high-level diagrams which indicate how these components are wired together within `XmlKeyManager`.

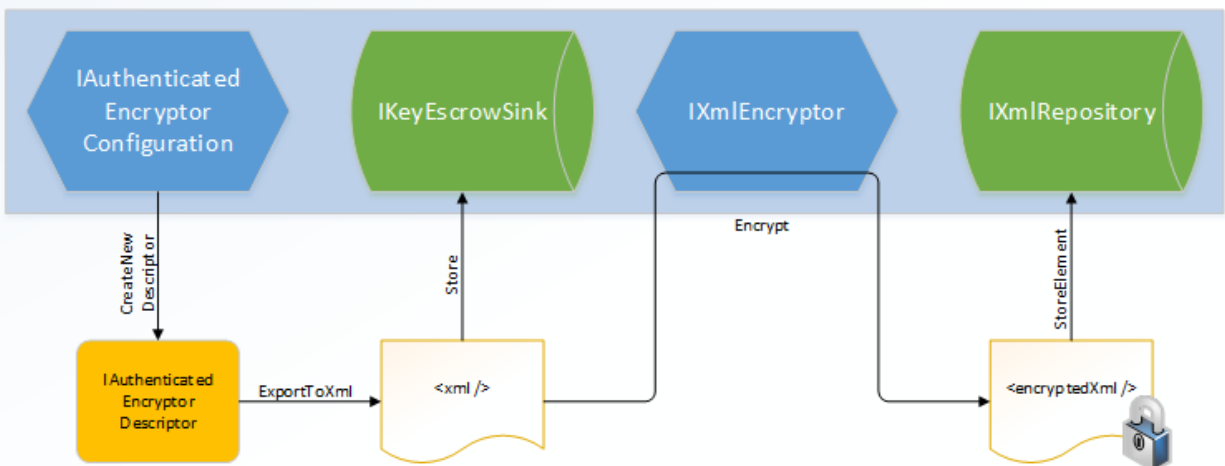


Fig. 2.1: Key Creation / CreateNewKey

In the implementation of `CreateNewKey`, the `IAuthenticatedEncryptorConfiguration` component is used to create a unique `IAuthenticatedEncryptorDescriptor`, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an `IXmlEncryptor` (if required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the `IXmlRepository`. (If no `IXmlEncryptor` is configured, the unencrypted document is persisted in the `IXmlRepository`.)

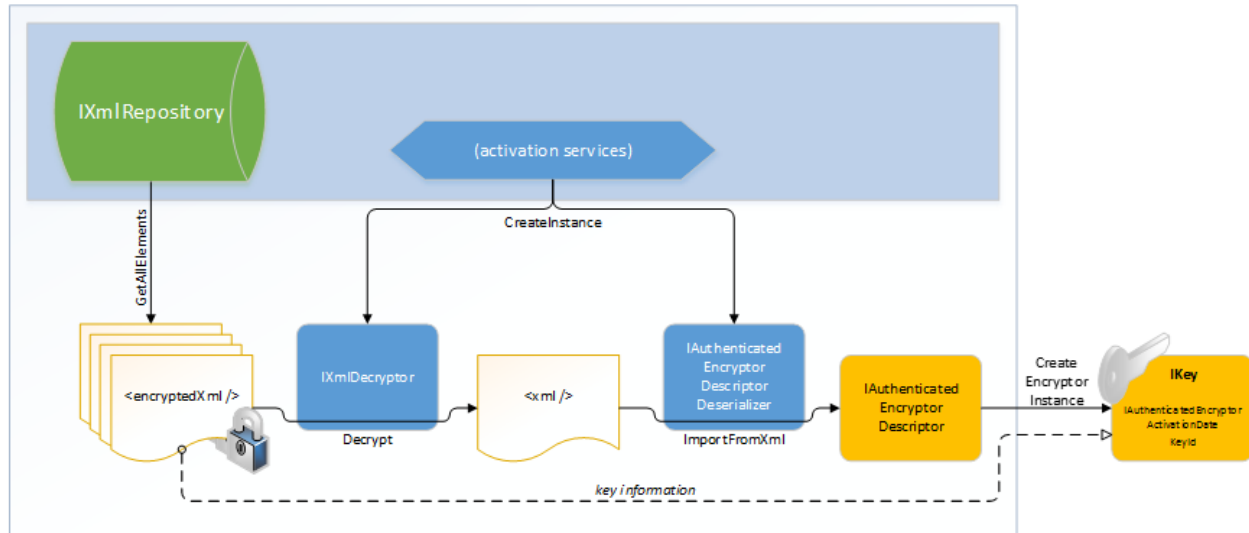


Fig. 2.2: Key Retrieval / GetAllKeys

In the implementation of `GetAllKeys`, the XML documents representing keys and revocations are read from the underlying `IXmlRepository`. If these documents are encrypted, the system will automatically decrypt them. `XmlKeyManager` creates the appropriate `IAuthenticatedEncryptorDescriptorDeserializer` instances to deserialize the documents back into `IAuthenticatedEncryptorDescriptor` instances, which are then wrapped in individual `IKey` instances. This collection of `IKey` instances is returned to the caller.

Further information on the particular XML elements can be found in the [key storage format document](#).

IXmlRepository The `IXmlRepository` interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

- `GetAllElements()` : `ICollection<XElement>`
- `StoreElement(XElement element, string friendlyName)`

Implementations of `IXmlRepository` don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are two built-in concrete types which implement `IXmlRepository`: `FileSystemXmlRepository` and `RegistryXmlRepository`. See the [key storage providers document](#) for more information. Registering a custom `IXmlRepository` would be the appropriate manner to use a different backing store, e.g., Azure Blob Storage. To change the default repository application-wide, register a custom singleton `IXmlRepository` in the service provider.

IXmlEncryptor The `IXmlEncryptor` interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

- `Encrypt(XElement plaintextElement)` : `EncryptedXmlInfo`

If a serialized `IAuthenticatedEncryptorDescriptor` contains any elements marked as “requires encryption”, then `XmIKeyManager` will run those elements through the configured `IXmlEncryptor`’s `Encrypt` method, and it will persist the enciphered element rather than the plaintext element to the `IXmlRepository`. The output of the `Encrypt` method is an `EncryptedXmlInfo` object. This object is a wrapper which contains both the resultant enciphered `XElement` and the `Type` which represents an `IXmlDecryptor` which can be used to decipher the corresponding element.

There are four built-in concrete types which implement `IXmlEncryptor`: `CertificateXmlEncryptor`, `DpapiNGXmlEncryptor`, `DpapiXmlEncryptor`, and `NullXmlEncryptor`. See the [key encryption at rest document](#) for more information. To change the default key-encryption-at-rest mechanism application-wide, register a custom singleton `IXmlEncryptor` in the service provider.

IXmlDecryptor The `IXmlDecryptor` interface represents a type that knows how to decrypt an `XElement` that was enciphered via an `IXmlEncryptor`. It exposes a single API:

- `Decrypt(XElement encryptedElement) : XElement`

The `Decrypt` method undoes the encryption performed by `IXmlEncryptor.Encrypt`. Generally each concrete `IXmlEncryptor` implementation will have a corresponding concrete `IXmlDecryptor` implementation.

Types which implement `IXmlDecryptor` should have one of the following two public constructors:

- `.ctor(IServiceProvider)`
- `.ctor()`

Note: The `IServiceProvider` passed to the constructor may be null.

IKeyEscrowSink The `IKeyEscrowSink` interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the [IXmlEncryptor](#) type in the first place. However, accidents happen, and keyrings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it is transformed by any configured [IXmlEncryptor](#). The interface exposes a single API:

- `Store(Guid keyId, XElement element)`

It is up to the `IKeyEscrowSink` implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a known corporate X.509 certificate where the certificate’s private key has been escrowed; the `CertificateXmlEncryptor` type can assist with this. The `IKeyEscrowSink` implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can [configure this globally](#). It can also be configured programmatically via the `DataProtectionConfiguration.AddKeyEscrowSink` method as shown in the sample below. The `AddKeyEscrowSink` method overloads mirror the `IServiceCollection.AddSingleton` and `IServiceCollection.AddInstance` overloads, as `IKeyEscrowSink` instances are intended to be singletons. If multiple `IKeyEscrowSink` instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There is no API to read material from an `IKeyEscrowSink` instance. This is consistent with the design theory of the escrow mechanism: it’s intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn’t have access to its own escrowed material.

The following sample code demonstrates creating and registering an `IKeyEscrowSink` where keys are escrowed such that only members of “CONTOSODomain Admins” can recover them.

Note: To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the

domain controller must be Windows Server 2012 or later.

```

1 using System;
2 using System.IO;
3 using System.Xml.Linq;
4 using Microsoft.AspNet.DataProtection.KeyManagement;
5 using Microsoft.AspNet.DataProtection.XmlEncryption;
6 using Microsoft.Framework.DependencyInjection;
7
8 public class Program
9 {
10     public static void Main(string[] args)
11     {
12         var serviceCollection = new ServiceCollection();
13         serviceCollection.AddDataProtection();
14         serviceCollection.ConfigureDataProtection(configure =>
15         {
16             // point at a specific folder and use DPAPI to encrypt keys
17             configure.PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"));
18             configure.ProtectKeysWithDpapi();
19             configure.AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
20         });
21         var services = serviceCollection.BuildServiceProvider();
22
23         // get a reference to the key manager and force a new key to be generated
24         Console.WriteLine("Generating new key...");
25         var keyManager = services.GetService<IKeyManager>();
26         keyManager.CreateNewKey(
27             activationDate: DateTimeOffset.Now,
28             expirationDate: DateTimeOffset.Now.AddDays(7));
29     }
30
31     // A key escrow sink where keys are escrowed such that they
32     // can be read by members of the CONTOSO\Domain Admins group.
33     private class MyKeyEscrowSink : IKeyEscrowSink
34     {
35         private readonly IXmlEncryptor _escrowEncryptor;
36
37         public MyKeyEscrowSink(IServiceProvider services)
38         {
39             // Assuming I'm on a machine that's a member of the CONTOSO
40             // domain, I can use the Domain Admins SID to generate an
41             // encrypted payload that only they can read. Sample SID from
42             // https://technet.microsoft.com/en-us/library/cc778824(v=ws.10).aspx.
43             _escrowEncryptor = new DpapiNGXmlEncryptor(
44                 "SID=S-1-5-21-1004336348-1177238915-682003330-512",
45                 DpapiNGProtectionDescriptorFlags.None,
46                 services);
47         }
48
49         public void Store(Guid keyId, XElement element)
50         {
51             // Encrypt the key element to the escrow encryptor.
52             var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);
53
54             // A real implementation would save the escrowed key to a
55             // write-only file share or some other stable storage, but
56             // in this sample we'll just write it out to the console.

```

```

57         Console.WriteLine($"Escrowing key {keyId}");
58         Console.WriteLine(encryptedXmlInfo.EncryptedElement);
59
60         // Note: We cannot read the escrowed key material ourselves.
61         // We need to get a member of CONTOSO\Domain Admins to read
62         // it for us in the event we need to recover it.
63     }
64 }
65 }
66
67 /*
68  * SAMPLE OUTPUT
69  *
70  * Generating new key...
71  * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
72  * <encryptedKey>
73  *   <!-- This key is encrypted with Windows DPAPI-NG. -->
74  *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
75  *   <value>MIIfAYJKoZIhvcNAQcDoIIbTCCCGkCAQ...T5rA4g==</value>
76  * </encryptedKey>
77 */

```

Miscellaneous APIs

Warning: Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret The ISecret interface represents a secret value, such as cryptographic key material. It contains the following API surface.

- Length : int
- Dispose() : void
- WriteSecretIntoBuffer(ArraySegment<byte> buffer) : void

The WriteSecretIntoBuffer method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a byte[] directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The Secret type is a concrete implementation of ISecret where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via [CryptProtectMemory](#).

Implementation

Authenticated encryption details.

Calls to IDataProtector.Protect are authenticated encryption operations. The Protect method offers both confidentiality and authenticity, and it is tied to the purpose chain that was used to derive this particular IDataProtector instance from its root IDataProtectionProvider.

IDataProtector.Protect takes a byte[] plaintext parameter and produces a byte[] protected payload, whose format is described below. (There is also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be [base64url-encoded](#).)

Protected payload format The protected payload format consists of three primary components:

- A 32-bit magic header that identifies the version of the data protection system.
- A 128-bit key id that identifies the key used to protect this particular payload.
- The remainder of the protected payload is *specific to the encryptor encapsulated by this key*. In the example below the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows: * A 128-bit key modifier. * A 128-bit initialization vector. * 48 bytes of AES-256-CBC output. * An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```

1 09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
2 AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
3 84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
4 79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
5 61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
6 5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
7 8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
8 52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
9 52 C9 74 A0

```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

Warning: All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {0c819c80-6619-4019-9536-53f8aaffee57}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it is reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey Derivation and Authenticated Encryption

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating “CBC-mode encryption + HMAC validation” or “GCM encryption + validation”. In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

Note: Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of `IAuthenticatedEncryptor` rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation The `IAuthenticatedEncryptor` interface serves as the core interface for all authenticated encryption operations. Its `Encrypt` method takes two buffers: plaintext and additionalAuthenticatedData (AAD). The plaintext contents flow unchanged the call to `IDataProtector.Protect`, but the AAD is generated by the system and consists of three components:

1. The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.
2. The 128-bit key id.

3. A variable-length string formed from the purpose chain that created the IDataProtector that is performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from K_M instead of using K_M itself in all of our cryptographic operations. For every call to `IAuthenticatedEncryptor.Encrypt`, the following key derivation process takes place:

$(K_E, K_H) = \text{SP800_108_CTR_HMACSHA512}(K_M, \text{AAD}, \text{contextHeader} \parallel \text{keyModifier})$

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see [NIST SP800-108](#), Sec. 5.1) with the following parameters:

- Key derivation key (KDK) = K_M
- PRF = HMACSHA512
- label = additionalAuthenticatedData
- context = contextHeader \parallel keyModifier

The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving K_E and K_H . The key modifier is a 128-bit string randomly generated for each call to `Encrypt` and serves to ensure with overwhelming probability that K_E and K_H are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, $|K_E|$ is the length of the symmetric block cipher key, and $|K_H|$ is the digest size of the HMAC routine. For GCM encryption + validation operations, $|K_H| = 0$.

CBC-mode encryption + HMAC validation Once K_E is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key K_H to produce the MAC. This process and the return value is represented graphically below.

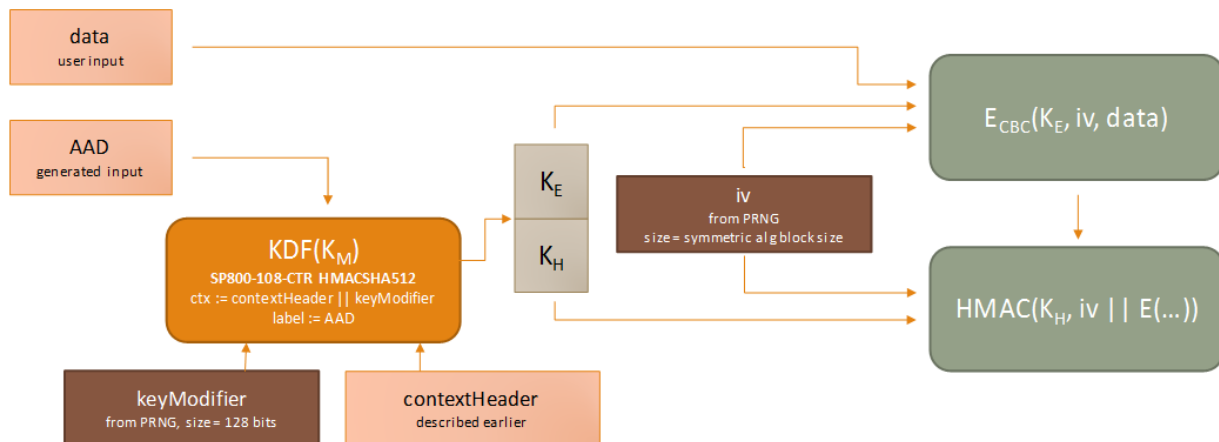


Fig. 2.3: $\text{output} := \text{keyModifier} \parallel \text{iv} \parallel E_{\text{cbc}}(K_E, \text{iv}, \text{data}) \parallel \text{HMAC}(K_H, \text{iv} \parallel E_{\text{cbc}}(K_E, \text{iv}, \text{data}))$

Note: The `IDataProtector.Protect` implementation will *prepend the magic header and key id* to output before returning it to the caller. Because the magic header and key id are implicitly part of *AAD*, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation Once K_E is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.

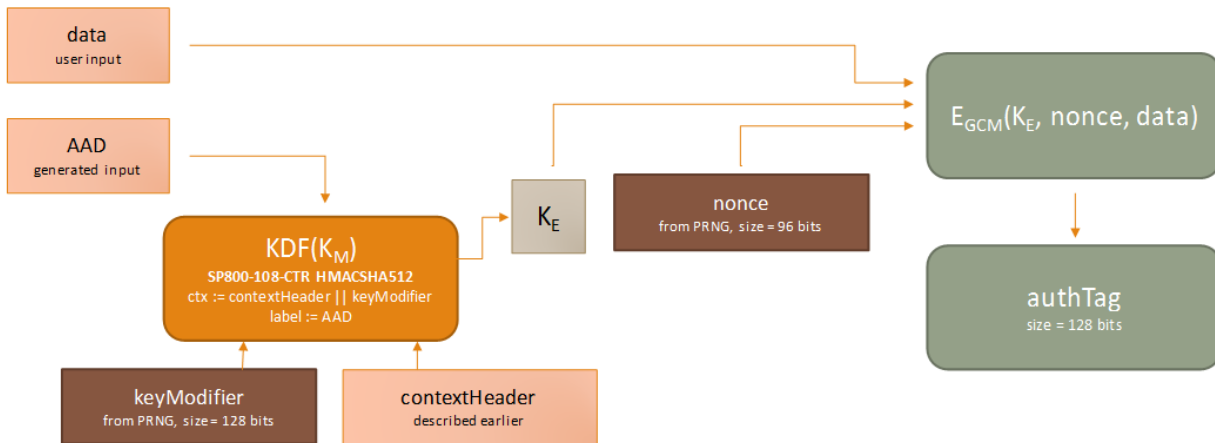


Fig. 2.4: $\text{output} := \text{keyModifier} \parallel \text{nonce} \parallel E_{\text{gcm}}(K_E, \text{nonce}, \text{data}) \parallel \text{authTag}$

Note: Even though GCM natively supports the concept of AAD, we’re still feeding AAD only to the original KDF, opting to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, *to support agility* we never want to use K_M directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^{-32} . If we fix K_E we cannot perform more than 2^{32} encryption operations before we run afoul of the 2^{-32} limit. This might seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^{-32} probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given K_M . For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there is no need to forward it to the GCM routine.

Context headers

Background and theory In the data protection system, a “key” means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It is intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of *cryptographic agility*, which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm’s OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: “AES” (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we’d need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), he’d have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the *subkey derivation process*. There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication The context header consists of the following components:

- [16 bits] The value 00 00, which is a marker meaning “CBC encryption + HMAC authentication”.
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)
- [32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.
- $\text{EncCBC}(K_E, \text{IV}, \text{“”})$, which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of K_E is described below.
- $\text{MAC}(K_H, \text{“”})$, which is the output of the HMAC algorithm given an empty string input. The construction of K_H is described below.

Ideally we could pass all-zero vectors for K_E and K_H . However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see [NIST SP800-108](#), Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive $|K_E| + |K_H|$ bytes of output, then decompose the result into K_E and K_H themselves. Mathematically, this is represented as follows.

$(K_E \parallel K_H) = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = \text{“”}, \text{label} = \text{“”}, \text{context} = \text{“”})$

Example: AES-192-CBC + HMACSHA256 As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let $(K_E \parallel K_H) = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = \text{“”}, \text{label} = \text{“”}, \text{context} = \text{“”})$, where $|K_E| = 192$ bits and $|K_H| = 256$ bits per the specified algorithms. This leads to $K_E = 5\text{BB}6\text{..}21\text{DD}$ and $K_H = \text{A}04\text{A}\text{..}00\text{A}9$ in the example below:

5B	B6	C9	83	13	78	22	1D	8E	10	73	CA	CF	65	8E	B0
61	62	42	71	CB	83	21	DD	A0	4A	05	00	5B	AB	C0	A2
49	6F	A5	61	E3	E2	49	87	AA	63	55	CD	74	0A	DA	C4
B7	92	3D	BF	59	90	00	A9								

Next, compute $\text{Enc}_{\text{CBC}}(K_E, \text{IV}, \text{“”})$ for AES-192-CBC given $\text{IV} = 0^*$ and K_E as above.

result := F474B1872B3B53E4721DE19C0841DB6F

Next, compute $\text{MAC}(K_H, \text{""})$ for HMACSHA256 given K_H as above.

result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C

This produces the full context header below:

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described [above](#) are:

- the marker (00 00)
- the block cipher key length (00 00 00 18)
- the block cipher block size (00 00 00 10)
- the HMAC key length (00 00 00 20)
- the HMAC digest size (00 00 00 20)
- the block cipher PRP output (F4 74 - DB 6F) and
- the HMAC PRF output (D4 79 - end).

Note: The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and Keyed-HashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the Keyed-HashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1 First, let $(K_E \parallel K_H) = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = \text{""}, \text{label} = \text{""}, \text{context} = \text{""})$, where $|K_E| = 192$ bits and $|K_H| = 160$ bits per the specified algorithms. This leads to $K_E = \text{A219..E2BB}$ and $K_H = \text{DC4A..B464}$ in the example below:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

Next, compute $\text{Enc}_{\text{CBC}}(K_E, \text{IV}, \text{""})$ for 3DES-192-CBC given $\text{IV} = 0^*$ and K_E as above.

result := ABB100F81E53E10E

Next, compute $\text{MAC}(K_H, \text{""})$ for HMACSHA1 given K_H as above.

result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

The components break down as follows:

- the marker (00 00)
- the block cipher key length (00 00 00 18)

- the block cipher block size (00 00 00 08)
- the HMAC key length (00 00 00 14)
- the HMAC digest size (00 00 00 14)
- the block cipher PRP output (AB B1 - E1 0E) and
- the HMAC PRF output (76 EB - end).

Galois/Counter Mode encryption + authentication The context header consists of the following components:

- [16 bits] The value 00 01, which is a marker meaning “GCM encryption + authentication”.
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our system, this is fixed at nonce size = 96 bits.)
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)
- [32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)
- [128 bits] The tag of $\text{Enc}_{\text{GCM}}(K_E, \text{nonce}, \text{“”})$, which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

K_E is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there is no K_H in play here, we essentially have $|K_H| = 0$, and the algorithm collapses to the below form.

$K_E = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = \text{“”}, \text{label} = \text{“”}, \text{context} = \text{“”})$

Example: AES-256-GCM First, let $K_E = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = \text{“”}, \text{label} = \text{“”}, \text{context} = \text{“”})$, where $|K_E| = 256$ bits.

$K_E := 22\text{BC}6\text{F}1\text{B}171\text{C}08\text{C}4\text{AE}2\text{F}27444\text{AF}8\text{FC}8\text{B}3087\text{A}90006\text{CAEA}91\text{FDCFB}47\text{C}1\text{B}8733\text{B}8$

Next, compute the authentication tag of $\text{Enc}_{\text{GCM}}(K_E, \text{nonce}, \text{“”})$ for AES-256-GCM given nonce = 096 and K_E as above.

result := E7DCCE66DF855A323A6BB7BD7A59BE45

This produces the full context header below:

00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45

The components break down as follows:

- the marker (00 01)
- the block cipher key length (00 00 00 20)
- the nonce size (00 00 00 0C)
- the block cipher block size (00 00 00 10)
- the authentication tag size (00 00 00 10) and
- the authentication tag from running the block cipher (E7 DC - end).

Key Management

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages.

- **Created** - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.
- **Active** - the key exists in the key ring and should be used for all new Protect operations.
- **Expired** - the key has run its natural lifetime and should no longer be used for new Protect operations.
- **Revoked** - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can *override this behavior* if necessary.

Warning: The developer might be tempted to delete a key from the key ring (e.g., by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there is no emergency override like there is with revoked keys. Deleting a key is truly destructive behavior, and consequently the data protection system exposes no first-class API for performing this operation.

Default key selection When the data protection system reads the key ring from the backing repository, it will attempt to locate a “default” key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has not disabled automatic key generation, then a new key will be generated with immediate activation per the *key expiration and rolling* policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There is an exception. If the application developer has *disabled automatic key generation*, then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling When a key is created, it is automatically given an activation date of { now + 2 days } and an expiration date of { now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring does not already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

```
services.ConfigureDataProtection(configure =>
{
    // use 14-day lifetime instead of 90-day lifetime
    configure.SetDefaultKeyLifetime(TimeSpan.FromDays(14));
});
```

An administrator can also change the default system-wide, though an explicit call to `SetDefaultKeyLifetime` will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic keyring refresh When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows `Protect` and `Unprotect` operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

Warning: Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface `IKeyManager` that can be used to inspect and make changes to the key ring. The DI system that provided the instance of `IDataProtectionProvider` can also provide an instance of `IKeyManager` for your consumption. Alternatively, you can pull the `IKeyManager` straight from the `IServiceProvider` as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to `Protect` or `Unprotect` will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the `IKeyManager` interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

```
1 using System;
2 using System.IO;
3 using System.Threading;
4 using Microsoft.AspNet.DataProtection;
5 using Microsoft.AspNet.DataProtection.KeyManagement;
6 using Microsoft.Framework.DependencyInjection;
7
8 public class Program
9 {
10     public static void Main(string[] args)
11     {
12         var serviceCollection = new ServiceCollection();
13         serviceCollection.AddDataProtection();
14         serviceCollection.ConfigureDataProtection(configure =>
15         {
16             // point at a specific folder and use DPAPI to encrypt keys
17             configure.PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"));
18             configure.ProtectKeysWithDpapi();
19         });
20         var services = serviceCollection.BuildServiceProvider();
21
22         // perform a protect operation to force the system to put at least
23         // one key in the key ring
24         services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
25         Console.WriteLine("Performed a protect operation.");
26         Thread.Sleep(2000);
```

```

27
28     // get a reference to the key manager
29     var keyManager = services.GetService<IKeyManager>();
30
31     // list all keys in the key ring
32     var allKeys = keyManager.GetAllKeys();
33     Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
34     foreach (var key in allKeys)
35     {
36         Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked = {key.IsRevoked}");
37     }
38
39     // revoke all keys in the key ring
40     keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation reason here.");
41     Console.WriteLine("Revoked all existing keys.");
42
43     // add a new key to the key ring with immediate activation and a 1-month expiration
44     keyManager.CreateNewKey(
45         activationDate: DateTimeOffset.Now,
46         expirationDate: DateTimeOffset.Now.AddMonths(1));
47     Console.WriteLine("Added a new key.");
48
49     // list all keys in the key ring
50     allKeys = keyManager.GetAllKeys();
51     Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
52     foreach (var key in allKeys)
53     {
54         Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked = {key.IsRevoked}");
55     }
56 }
57
58
59 /*
60  * SAMPLE OUTPUT
61  *
62  * Performed a protect operation.
63  * The key ring contains 1 key(s).
64  * Key {1b948618-belf-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = False
65  * Revoked all existing keys.
66  * Added a new key.
67  * The key ring contains 2 key(s).
68  * Key {1b948618-belf-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = True
69  * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18 22:20:51Z, IsRevoked = False
70  */

```

Key storage The data protection system has a heuristic whereby it tries to deduce an appropriate key storage location and encryption at rest mechanism automatically. This is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

- [In-box key storage providers](#)
- [In-box key encryption at rest providers](#)

Key Storage Providers

By default the data protection system *employs a heuristic* to determine where cryptographic key material should be persisted. The developer can override the heuristic and manually specify the location.

Note: If you specify an explicit key persistence location, the data protection system will deregister the default key encryption at rest mechanism that the heuristic provided, so keys will no longer be encrypted at rest. It is recommended that you additionally *specify an explicit key encryption mechanism* for production applications.

The data protection system ships with two in-box key storage providers.

File system We anticipate that the majority of applications will use a file system-based key repository. To configure this, call the `PersistKeysToFileSystem` configuration routine as demonstrated below, providing a `DirectoryInfo` pointing to the repository where keys should be stored.

```
sc.ConfigureDataProtection(configure =>
{
    // persist keys to a specific directory
    configure.PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
});
```

The `DirectoryInfo` can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only applications on the local machine will need to use this repository), consider using *Windows DPAPI* to encrypt the keys at rest. Otherwise consider using an *X.509 certificate* to encrypt keys at rest.

Registry Sometimes the application might not have write access to the file system. Consider a scenario where an application is running as a virtual service account (such as `w3wp.exe`'s app pool identity). In these cases, the administrator may have provisioned a registry key that is appropriate ACLed for the service account identity. Call the `PersistKeysToRegistry` configuration routine as demonstrated below to take advantage of this, providing a `RegistryKey` pointing to the location where cryptographic key material should be stored.

```
sc.ConfigureDataProtection(configure =>
{
    // persist keys to a specific location in the system registry
    configure.PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
});
```

If you use the system registry as a persistence mechanism, consider using *Windows DPAPI* to encrypt the keys at rest.

Custom key repository If the in-box mechanisms are not appropriate, the developer can specify his own key persistence mechanism by providing a custom `IXmlRepository`.

Key Encryption At Rest

By default the data protection system *employs a heuristic* to determine how cryptographic key material should be encrypted at rest. The developer can override the heuristic and manually specify how keys should be encrypted at rest.

Note: If you specify an explicit key encryption at rest mechanism, the data protection system will deregister the default key storage mechanism that the heuristic provided. You must *specify an explicit key storage mechanism*, otherwise the data protection system will fail to start. The data protection system ships with three in-box key encryption mechanisms.

Windows DPAPI *This mechanism is available only on Windows.*

When Windows DPAPI is used, key material will be encrypted via `CryptProtectData` before being persisted to storage. DPAPI is an appropriate encryption mechanism for data that will never be read outside of the current machine (though

it is possible to back these keys up to Active Directory; see [DPAPI and Roaming Profiles](#)). For example to configure DPAPI key-at-rest encryption.

```
sc.ConfigureDataProtection(configure =>
{
    // only the local user account can decrypt the keys
    configure.ProtectKeysWithDpapi();
});
```

If `ProtectKeysWithDpapi` is called with no parameters, only the current Windows user account can decipher the persisted key material. You can optionally specify that any user account on the machine (not just the current user account) should be able to decipher the key material, as shown in the below example.

```
sc.ConfigureDataProtection(configure =>
{
    // all user accounts on the machine can decrypt the keys
    configure.ProtectKeysWithDpapi(protectToLocalMachine: true);
});
```

X.509 certificate *This mechanism is not yet available on Core CLR.*

If your application is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and to configure applications to use this certificate for encryption of keys at rest. See below for an example.

```
sc.ConfigureDataProtection(configure =>
{
    // searches the cert store for the cert with this thumbprint
    configure.ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
});
```

Because this mechanism uses `X509Certificate2` and `EncryptedXml` under the covers, this feature is currently only available on Desktop CLR. Additionally, due to .NET Framework limitations only certificates with CAPI private keys are supported. See [Certificate-based encryption with Windows DPAPI-NG](#) below for possible workarounds to these limitations.

Windows DPAPI-NG *This mechanism is available only on Windows 8 / Windows Server 2012 and later.*

Beginning with Windows 8, the operating system supports DPAPI-NG (also called CNG DPAPI). Microsoft lays out its usage scenario as follows.

Cloud computing, however, often requires that content encrypted on one computer be decrypted on another. Therefore, beginning with Windows 8, Microsoft extended the idea of using a relatively straightforward API to encompass cloud scenarios. This new API, called DPAPI-NG, enables you to securely share secrets (keys, passwords, key material) and messages by protecting them to a set of principals that can be used to unprotect them on different computers after proper authentication and authorization.

From [https://msdn.microsoft.com/en-us/library/windows/desktop/hh706794\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/hh706794(v=vs.85).aspx)

The principal is encoded as a protection descriptor rule. Consider the below example, which encrypts key material such that only the domain-joined user with the specified SID can decrypt the key material.

```
sc.ConfigureDataProtection(configure =>
{
    // uses the descriptor rule "SID=S-1-5-21-..."
    configure.ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
        flags: DpapiNGProtectionDescriptorFlags.None);
});
```

There is also a parameterless overload of `ProtectKeysWithDpapiNG`. This is a convenience method for specifying the rule “SID=mine”, where mine is the SID of the current Windows user account.

```
sc.ConfigureDataProtection(configure =>
{
    // uses the descriptor rule "SID={current account SID}"
    configure.ProtectKeysWithDpapiNG();
});
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user will be able to decipher the encrypted payload from any domain-joined machine (provided that the process is running under his identity).

Certificate-based encryption with Windows DPAPI-NG If you’re running on Windows 8.1 / Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption, even if the application is running on Core CLR. To take advantage of this, use the rule descriptor string “CERTIFICATE=HashId:thumbprint”, where thumbprint is the hex-encoded SHA1 thumbprint of the certificate to use. See below for an example.

```
sc.ConfigureDataProtection(configure =>
{
    // searches the cert store for the cert with this thumbprint
    configure.ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0",
        flags: DpapiNGProtectionDescriptorFlags.None);
});
```

Any application which is pointed at this repository must be running on Windows 8.1 / Windows Server 2012 R2 or later to be able to decipher this key.

Custom key encryption If the in-box mechanisms are not appropriate, the developer can specify his own key encryption mechanism by providing a custom `IXmlEncryptor`.

Key Immutability and Changing Settings

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it is immutable. Its creation, activation, and expiration dates can never be changed, though it can be revoked by using `IKeyManager`. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes will not go into effect until the next time a key is generated, either via an explicit call to `IKeyManager.CreateNewKey` or via the data protection system’s own *automatic key generation* behavior. The settings that affect key persistence are as follows:

- *The default key lifetime*
- *The key encryption at rest mechanism*
- *The algorithmic information contained within the key*

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to `IKeyManager.CreateNewKey` to force the creation of a new key. Remember to provide an explicit activation date (`{ now + 2 days }` is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

Tip: All applications touching the repository should specify the same settings in the call to `ConfigureDataProtection`, otherwise the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key Storage Format

Objects are stored at rest in XML representation. The default directory for key storage is %LOCALAPPDATA%\ASP.NETDataProtection-Keys\.

The <key> element Keys exist as top-level objects in the key repository. By convention keys have the filename **key-{guid}.xml**, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<key id="80732141-ec8f-4b80-af9c-c4d2d1ff8901" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor serializerType="{serializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

The <key> element contains the following attributes and child elements:

- The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.
- The version of the <key> element, currently fixed at 1.
- The key's creation, activation, and expiration dates.
- A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {80732141-ec8f-4b80-af9c-c4d2d1ff8901}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element The outer <descriptor> element contains an attribute `serializerType`, which is the assembly-qualified name of a type which implements `IAuthenticatedEncryptorDescriptorDeserializer`. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though, this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element An <encryptedSecret> element which contains the encrypted form of the secret key material may be present if *encryption of secrets at rest is enabled*. The attribute `decryptorType` will be the

assembly-qualified name of a type which implements `IXmlDecryptor`. This type is responsible for reading the inner `<encryptedKey>` element and decrypting it to recover the original plaintext.

As with `<descriptor>`, the particular format of the `<encryptedSecret>` element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The `<revocation>` element Revocations exist as top-level objects in the key repository. By convention revocations have the filename `revocation-{timestamp}.xml` (for revoking all keys before a specific date) or `revocation-{guid}.xml` (for revoking a specific key). Each file contains a single `<revocation>` element.

For revocations of individual keys, the file contents will be as below.

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="eb4fc299-8808-409d-8a34-23fc83d026c9" />
  <reason>human-readable reason</reason>
</revocation>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

The `<reason>` element is never read by the system. It is simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers

There are scenarios where an application needs a throwaway `IDataProtectionProvider`. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the package `Microsoft.AspNetCore.DataProtection` includes a type `EphemeralDataProtectionProvider`. This type provides a basic implementation of `IDataProtectionProvider` whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of `EphemeralDataProtectionProvider` uses its own unique master key. Therefore, if an `IDataProtector` rooted at an `EphemeralDataProtectionProvider` generates a protected payload, that payload can only be unprotected by an equivalent `IDataProtector` (given the same *purpose* chain) rooted at the same `EphemeralDataProtectionProvider` instance.

The following sample demonstrates instantiating an `EphemeralDataProtectionProvider` and using it to protect and unprotect data.

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";
```

```

        // create an ephemeral provider and demonstrate that it can round-trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); // THROWS
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAAAAA...uGoxWLjGKtmlSkNACQ
 * Unprotect returned: Hello!
 * << throws CryptographicException >>
 */

```

Compatibility

Sharing cookies between applications.

Web sites commonly consist of many individual web applications, all working together harmoniously. If an application developer wants to provide a good single-sign-on experience, he'll often need all of the different web applications within the site to share authentication tickets between each other.

To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET 5 cookie authentication tickets.

Sharing authentication cookies between ASP.NET 5 applications. To share authentication cookies between two different ASP.NET 5 applications, configure each application that should share cookies as follows.

1. Install the package `Microsoft.AspNet.Authentication.Cookies.Shareable` into each of your ASP.NET 5 applications.
2. In `Startup.cs`, locate the call to `UseIdentity`, which will generally look like the following.

```

// Add cookie-based authentication to the request pipeline.
app.UseIdentity();

```

3. Remove the call to `UseIdentity`, replacing it with four separate calls to `UseCookieAuthentication`. (`UseIdentity` calls these four methods under the covers.) In the call to `UseCookieAuthentication` that sets up the application cookie, provide an instance of a `DataProtectionProvider` that has been initialized to a key storage location.

```
// Add cookie-based authentication to the request pipeline.
// NOTE: Need to decompose this into its constituent components
// app.UseIdentity();

app.UseCookieAuthentication(null, IdentityOptions.ExternalCookieAuthenticationScheme);
app.UseCookieAuthentication(null, IdentityOptions.TwoFactorRememberMeCookieAuthenticationScheme);
app.UseCookieAuthentication(null, IdentityOptions.TwoFactorUserIdCookieAuthenticationScheme);
app.UseCookieAuthentication(null, IdentityOptions.ApplicationCookieAuthenticationScheme,
    dataProtectionProvider: new DataProtectionProvider(
        new DirectoryInfo(@"c:\shared-auth-ticket-keys\")));
```

Caution: When used in this manner, the `DirectoryInfo` should point to a key storage location specifically set aside for authentication cookies. The application name is ignored (intentionally so, since you're trying to get multiple applications to share payloads). You should consider configuring the `DataProtectionProvider` such that keys are encrypted at rest, as in the below example.

```
app.UseCookieAuthentication(null, IdentityOptions.ApplicationCookieAuthenticationScheme,
    dataProtectionProvider: new DataProtectionProvider(
        new DirectoryInfo(@"c:\shared-auth-ticket-keys\"),
        configure =>
        {
            configure.ProtectKeysWithCertificate("thumbprint");
        }
    ));
```

The cookie authentication middleware will use the explicitly provided implementation of the `DataProtectionProvider`, which due to taking an explicit directory in its constructor is isolated from the data protection system used by other parts of the application.

Sharing authentication cookies between ASP.NET 4.x and ASP.NET 5 applications. ASP.NET 4.x applications which use Katana cookie authentication middleware can be configured to generate authentication cookies which are compatible with the ASP.NET 5 cookie authentication middleware. This allows upgrading a large site's individual applications piecemeal while still providing a smooth single sign on experience across the site.

Tip: You can tell if your existing application uses Katana cookie authentication middleware by the existence of a call to `UseCookieAuthentication` in your project's `Startup.Auth.cs`. ASP.NET 4.x web application projects created with Visual Studio 2013 and later use the Katana cookie authentication middleware by default.

Note: Your ASP.NET 4.x application must target .NET Framework 4.5.1 or higher, otherwise the necessary NuGet packages will fail to install.

To share authentication cookies between your ASP.NET 4.x applications and your ASP.NET 5 applications, configure the ASP.NET 5 application as stated above, then configure your ASP.NET 4.x applications by following the steps below.

1. Install the package `Microsoft.Owin.Security.Cookies.Shareable` into each of your ASP.NET 4.x applications.
2. In `Startup.Auth.cs`, locate the call to `UseCookieAuthentication`, which will generally look like the following.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    // ...
});
```

3. Modify the call to `UseCookieAuthentication` as follows, changing the `AuthenticationType` and `CookieName` to match those of the ASP.NET 5 cookie authentication middleware, and providing an instance of a `DataProtectionProvider` that has been initialized to a key storage location.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = DefaultCompatibilityConstants.ApplicationCookieAuthenticationType,
    CookieName = DefaultCompatibilityConstants.CookieName,
    // CookiePath = "...", (if necessary)
    // ...
},
dataProtectionProvider: new DataProtectionProvider(
    new DirectoryInfo(@"c:\shared-auth-ticket-keys\")));
```

The DirectoryInfo has to point to the same storage location that you pointed your ASP.NET 5 application to.

4. In IdentityModels.cs, change the call to ApplicationUserManager.CreateIdentity to use the same authentication type as in the cookie middleware.

```
public ClaimsIdentity GenerateUserIdentity(ApplicationUserManager manager)
{
    // Note the authenticationType must match the one defined in CookieAuthenticationOptions.AuthenticationType
    var userIdentity = manager.CreateIdentity(this, DefaultCompatibilityConstants.ApplicationCookieAuthenticationType);
    // ...
}
```

The ASP.NET 4.x and ASP.NET 5 applications are now configured to share authentication cookies.

Note: You'll need to make sure that the ASP.NET Identity system for each application is pointed at the same user database. Otherwise the identity system will produce failures at runtime when it tries to match the information in the authentication cookie against the information in its database.

Replacing <machineKey> in ASP.NET 4.5.1

As of ASP.NET 4.5, the implementation of the <machineKey> element is [replaceable](#). This allows most calls to ASP.NET 4.5+'s cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

Note: The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or higher. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package Microsoft.AspNet.DataProtection.SystemWeb. This will instantiate the data protection system using the [default configuration](#) settings.

When you install the package, it inserts a line into Web.config that tells ASP.NET to use it for [most cryptographic operations](#), including forms authentication, view state, and calls to MachineKey.Protect. The line that's inserted reads as follows.

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

Tip: You can tell if the new data protection system is active by inspecting fields like __VIEWSTATE, which should begin with "CfDJ8" as in the below example. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZOpk..." />
```

Package configuration The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses `DataProtectionStartup` and overrides its `ConfigureServices` method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

```
using System;
using System.IO;
using Microsoft.AspNet.DataProtection.SystemWeb;
using Microsoft.Framework.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.ConfigureDataProtection(configure =>
            {
                configure.SetApplicationName("my-app");
                configure.PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\myapp-keys\"));
                configure.ProtectKeysWithCertificate("thumbprint");
            });
        }
    }
}
```

Tip: You can also use `<machineKey applicationName="my-app" ... />` in place of an explicit call to `SetApplicationName`. This is a convenience mechanism to avoid forcing the developer to create a `DataProtectionStartup`-derived type if all he wanted to configure was setting the application name.

To enable this custom configuration, go back to `Web.config` and look for the `<appSettings>` element that the package install added to the config file. It will look like the below.

```
<appSettings>
  <!--
    If you want to customize the behavior of the ASP.NET 5 Data Protection stack, set the
    "aspnet:dataProtectionStartupType" switch below to be the fully-qualified name of a
    type which subclasses Microsoft.AspNet.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

Fill in the blank value with the assembly-qualified name of the `DataProtectionStartup`-derived type you just created. If the name of the application is `DataProtectionDemo`, this would look like the below.

```
<add key="aspnet:dataProtectionStartupType"
      value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />
```

The newly-configured data protection system is now ready for use inside the application.

2.10.6 Safe Storage of Application Secrets

By Rick Anderson

This tutorial shows how your application can securely store and access secrets in the local development environment. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode. The secret manager tool was written to help prevent sensitive data from being checked into source control. The [Configuration](#) system that is used by default in DNX based apps can read secrets stored with the secret manager tool described in this article.

In this article:

- [Environment variables](#)
- [Installing the secret manager tool](#)
- [How the secret manager tool works](#)
- [Additional Resources](#)

Environment variables

Your `Startup` class should call `AddEnvironmentVariables` as the last configuration method so when [DNX](#) reads environment variables, and a key is found in a configuration file and the environment, the environment value takes precedence over the configuration file. (See [Configuration](#).) For example, if you create a new ASP.NET web site app with individual user accounts, it will add a default connection string to the `config.json` file. The `Data:DefaultConnection:ConnectionString` key value in the `config.json` file uses LocalDB, which runs in user mode and doesn't require a password. When you deploy your application to a test or production server, you can override the `Data:DefaultConnection:ConnectionString` key value with an environment variable setting that connects to a test or production SQL Server. That key value would also contain a password to connect to the SQL Server.

Apps frequently require secrets, such as a client ID for OAuth. These passwords and other sensitive data should never be added to `config` files inside your project's source tree, as configuration files can be accidentally checked into source control. The secret manager tool provides a mechanism to store sensitive data for development work outside your project tree. The secret manager tool is a DNX console application that is used to store secrets used by DNX and ASP.NET applications in the development environment.

Installing the secret manager tool

- Create a new ASP.NET web app. We will use this to test secrets stored with the secret manager tool.
- Open a command prompt and navigate to the project folder (the folder with `config.json`).
- Set the runtime version using the .Net Version Manager (DNVM). DNVM is a tool that lets you list, install and switch [DNX](#) versions on your machine. Run the following command:

```
dnvm use 1.0.0-beta5
```

The `dnvm` tool is the .NET Version Manager used to update and configure the .NET Execution Environment (DNX). The command `dnvm use 1.0.0-beta5` instructs the .NET Version Manager to add the DNX to the `PATH` environment variable for the current shell. After running this command the following is displayed:

```
Adding C:\Users\<user>\.dnx\runtimes\dnx-clr-win-x86.1.0.0-beta5\bin to process PATH
```

- Install the secret manager tool using DNU (Microsoft .NET Development Utility). DNU is used to build, package and publish DNX projects.

```
dnv commands install SecretManager
```

- Test the secret manager tool by running the following command:

```
user-secret -h
```

The secret manager tool will display usage, options and command help.

- Use the secret manager tool to set a secret. For example, in the command window enter the following:

```
user-secret set MySecret ValueOfMySecret
```

- Add the following code to the end of the Startup method:

```
string testConfig = configuration.Get("MySecret");  
Trace.WriteLine(testConfig);
```

The output window of Visual Studio will display “ValueOfMySecret”.

How the secret manager tool works

The tool operates on project specific configuration settings that are stored in your user account. In the example above, the command window was opened in the project folder (containing the file *project.json*). You can run the secret manager tool from other directories, but you must use the `-project` switch and pass in the path to the *project.json* file.

The secret manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. In the current version, the values are stored in a **JSON** configuration file in the user profile directory:

- Windows: %APPDATA%\microsoft\UserSecrets\- Linux: ~/.microsoft/usersecrets/<applicationId>/secrets.json
- Mac: ~/.microsoft/usersecrets/<applicationId>/secrets.json

The `applicationId` comes from the *project.json* file and is arbitrary, but should be unique unless you have a reason for it not to be. The following markup shows a portion of the *project.json* file with the `applicationId` highlighted:

```
{  
  "webroot": "wwwroot",  
  "userSecretsId": "aspnet5-WebApplication1-f7fd3f56-2899-4eea-a88e-673d24bd7090",  
  "version": "1.0.0-*"  
}
```

The `userSecretsId` key for the `applicationId` highlighted above was generated by Visual Studio.

You should not write code that depends on the location or format of the data saved with the secret manager tool, as these implementation details might change. For example, the secret values are currently not encrypted today, but could be someday.

Additional Resources

- [Configuration](#).
- [DNX Overview](#).

2.10.7 Anti-Request Forgery

Note: This topic hasn’t been written yet! You can track the status of this [issue](#) through our public GitHub issue

tracker. Learn how you can [contribute](#) on GitHub.

2.10.8 Preventing Open Redirect Attacks

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.10.9 Preventing Cross-Site Scripting

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.10.10 Enabling Cross-Origin Requests (CORS)

By [Mike Wasson](#)

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy*, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites make cross-origin requests to your web app.

[Cross Origin Resource Sharing](#) (CORS) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others. CORS is safer and more flexible than earlier techniques such as [JSONP](#). This topic shows how to enable CORS in your ASP.NET 5 application.

In this article:

- *What is “same origin”?*
- *Add the CORS package*
- *Configure CORS in your app*
- *CORS policy options*
- *How CORS works*

Note: This topic covers general ASP.NET 5 apps. For information about CORS support in ASP.NET MVC 6, see [Specifying a CORS Policy](#).

What is “same origin”?

Two URLs have the same origin if they have identical schemes, hosts, and ports. ([RFC 6454](#))

These two URLs have the same origin:

- <http://example.com/foo.html>
- <http://example.com/bar.html>

These URLs have different origins than the previous two:

- <http://example.net> - Different domain
- <http://example.com:9000/foo.html> - Different port

- <https://example.com/foo.html> - Different scheme
- <http://www.example.com/foo.html> - Different subdomain

Note: Internet Explorer does not consider the port when comparing origins.

Add the CORS package

In your project.json file, add the following:

```
"dependencies": {  
  "Microsoft.AspNet.Cors": "1.0.0-beta6"  
},
```

Configure CORS in your app

This section shows how to configure CORS. First, add the CORS service. In Startup.cs:

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddCors();  
}
```

Next, configure a cross-origin policy, using the `CorsPolicyBuilder` class. There are two ways to do this. The first is to call `UseCors` with a lambda:

```
public void Configure(IApplicationBuilder app)  
{  
    app.UseCors(builder =>  
        builder.WithOrigins("http://example.com"));  
}
```

The lambda takes a `CorsPolicyBuilder` object. I'll describe all of the configuration options later in this topic. In this example, the policy allows cross-origin requests from "<http://example.com>" and no other origins.

Note that `CorsPolicyBuilder` has a fluent API, so you can chain method calls:

```
app.UseCors(builder =>  
    builder.WithOrigins("http://example.com")  
        .AllowAnyHeader()  
    );
```

The second approach is to define one or more named CORS policies, and then select the policy by name at run time.

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddCors();  
    services.ConfigureCors(options =>  
    {  
        options.AddPolicy("AllowSpecificOrigin",  
            builder => builder.WithOrigins("http://example.com"));  
    });  
}  
  
public void Configure(IApplicationBuilder app)  
{  
    app.UseCors("AllowSpecificOrigin");  
}
```

```
app.Run(async (context) =>
{
```

This example adds a CORS policy named “AllowSpecificOrigin”. To select the policy, pass the name to UseCors.

CORS policy options

This section describes the various options that you can set in a CORS policy.

- *Set the allowed origins*
- *Set the allowed HTTP methods*
- *Set the allowed request headers*
- *Set the exposed response headers*
- *Credentials in cross-origin requests*
- *Set the preflight expiration time*

For some options it may be helpful to read *How CORS works* first.

Set the allowed origins

To allow one or more specific origins:

```
options.AddPolicy("AllowSpecificOrigins",
builder =>
{
    builder.WithOrigins("http://example.com", "http://www.contoso.com");
});
```

To allow all origins:

```
options.AddPolicy("AllowAllOrigins",
builder =>
{
    builder.AllowAnyOrigin();
});
```

Consider carefully before allowing requests from any origin. It means that literally any website can make AJAX calls to your app.

Set the allowed HTTP methods

To specify which HTTP methods are allowed to access the resource.

```
options.AddPolicy("AllowSpecificMethods",
builder =>
{
    builder.WithOrigins("http://example.com")
        .WithMethods("GET", "POST", "HEAD");
});
```

To allow all HTTP methods:

```
options.AddPolicy("AllowAllMethods",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .AllowAnyMethod();
    });
```

This affects pre-flight requests and Access-Control-Allow-Methods header.

Set the allowed request headers

A CORS preflight request might include an Access-Control-Request-Headers header, listing the HTTP headers set by the application (the so-called “author request headers”).

To whitelist specific headers:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .WithHeaders("accept", "content-type", "origin", "x-custom-header");
    });
```

To allow all author request headers:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .AllowAnyHeader();
    });
```

Browsers are not entirely consistent in how they set Access-Control-Request-Headers. If you set headers to anything other than “*”, you should include at least “accept”, “content-type”, and “origin”, plus any custom headers that you want to support.

Set the exposed response headers

By default, the browser does not expose all of the response headers to the application. (See <http://www.w3.org/TR/cors/#simple-response-header>.) The response headers that are available by default are:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

The CORS spec calls these *simple response headers*. To make other headers available to the application:

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
```

```
        .WithExposedHeaders("x-custom-header");
    });
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser does not send any credentials with a cross-origin request. Credentials include cookies as well as HTTP authentication schemes. To send credentials with a cross-origin request, the client must set `XMLHttpRequest.withCredentials` to `true`.

Using `XMLHttpRequest` directly:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'http://www.example.com/api/test');
xhr.withCredentials = true;
```

In jQuery:

```
$.ajax({
    type: 'get',
    url: 'http://www.example.com/home',
    xhrFields: {
        withCredentials: true
    }
});
```

In addition, the server must allow the credentials. To allow cross-origin credentials:

```
options.AddPolicy("AllowCredentials",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .AllowCredentials();
    });
```

Now the HTTP response will include an `Access-Control-Allow-Credentials` header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials, but the response does not include a valid `Access-Control-Allow-Credentials` header, the browser will not expose the response to the application, and the AJAX request fails.

Be very careful about allowing cross-origin credentials, because it means a website at another domain can send a logged-in user's credentials to your app on the user's behalf, without the user being aware. The CORS spec also states that setting origins to `"*"` (all origins) is invalid if the `Access-Control-Allow-Credentials` header is present.

Set the preflight expiration time

The `Access-Control-Max-Age` header specifies how long the response to the preflight request can be cached. To set this header:

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
    {
        builder.WithOrigins("http://example.com")
            .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

How CORS works

This section describes what happens in a CORS request, at the level of the HTTP messages. It's important to understand how CORS works, so that you can configure your CORS policy correctly, and troubleshoot if things don't work as you expect.

The CORS specification introduces several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests; you don't need to do anything special in your JavaScript code.

Here is an example of a cross-origin request. The "Origin" header gives the domain of the site that is making the request:

```
GET http://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: http://myclient.azurewebsites.net/
Accept: */*
Accept-Language: en-US
Origin: http://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

If the server allows the request, it sets the Access-Control-Allow-Origin header. The value of this header either matches the Origin header, or is the wildcard value "*", meaning that any origin is allowed.:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Date: Wed, 20 May 2015 06:27:30 GMT
Content-Length: 12

Test message
```

If the response does not include the Access-Control-Allow-Origin header, the AJAX request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser does not make the response available to the client application.

Preflight Requests

For some CORS requests, the browser sends an additional request, called a "preflight request", before it sends the actual request for the resource. The browser can skip the preflight request if the following conditions are true:

- The request method is GET, HEAD, or POST, and
- The application does not set any request headers other than Accept, Accept-Language, Content-Language, Content-Type, or Last-Event-ID, and
- The Content-Type header (if set) is one of the following:
 - application/x-www-form-urlencoded
 - multipart/form-data
 - text/plain

The rule about request headers applies to headers that the application sets by calling `setRequestHeader` on the `XMLHttpRequest` object. (The CORS specification calls these "author request headers".) The rule does not apply to headers the browser can set, such as User-Agent, Host, or Content-Length.

Here is an example of a preflight request:

```
OPTIONS http://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: */*
Origin: http://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

The pre-flight request uses the HTTP OPTIONS method. It includes two special headers:

- **Access-Control-Request-Method:** The HTTP method that will be used for the actual request.
- **Access-Control-Request-Headers:** A list of request headers that the application set on the actual request. (Again, this does not include headers that the browser sets.)

Here is an example response, assuming that the server allows the request:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 20 May 2015 06:33:22 GMT
```

The response includes an **Access-Control-Allow-Methods** header that lists the allowed methods, and optionally an **Access-Control-Allow-Headers** header, which lists the allowed headers. If the preflight request succeeds, the browser sends the actual request, as described earlier.

2.11 Performance

2.11.1 Measuring Application Performance

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.12 Migration

2.12.1 Migrating from ASP.NET Identity 2.x to 3.x

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.12.2 Migrating HTTP Modules to Middleware

Note: This topic hasn't been written yet! You can track the status of this [issue](#) through our public GitHub issue tracker. Learn how you can [contribute](#) on GitHub.

2.13 Contribute

2.13.1 ASP.NET Docs Style Guide

By [Steve Smith](#)

This document provides an overview of how articles published on [docs.asp.net](#) should be formatted. You can actually use this file, itself, as a template when contributing articles.

In this article:

- [Article Structure](#)
- [ReStructuredText Syntax](#)

Article Structure

Articles should be submitted as individual text files with a **.rst** extension. Authors should be sure they are familiar with the [Sphinx Style Guide](#), but where there are disagreements, this document takes precedence. The article should begin with its title on line 1, followed by a line of === characters. Next, the author should be displayed with a link to an author specific page (ex. the author's GitHub user page, Twitter page, etc.).

Articles should typically begin with a brief abstract describing what will be covered, followed by a bulleted list of topics, if appropriate. If the article has associated sample files, a link to the samples should be included following this bulleted list.

Articles should typically include a Summary section at the end, and optionally additional sections like Next Steps or Additional Resources. These should not be included in the bulleted list of topics, however.

Headings

Typically articles will use at most 3 levels of headings. The title of the document is the highest level heading and must appear on lines 1-2 of the document. The title is designated by a row of === characters.

Section headings should correspond to the bulleted list of topics set out after the article abstract. *Article Structure*, above, is an example of a section heading. A section heading should appear on its own line, followed by a line consisting of — characters.

Subsection headings can be used to organize content within a section. *Headings*, above, is an example of a subsection heading. A subsection heading should appear on its own line, followed by a line of ^^^ characters.

```
Title (H1)
=====

Section heading (H2)
-----

Subsection heading (H3)
^^^^^^^^^^^^^^^^^^^^
```

For section headings, only the first word should be capitalized:

- Use this heading style

- Do Not Use This Style

More on sections and headings in ReStructuredText: <http://sphinx-doc.org/rest.html#sections>

ReStructuredText Syntax

The following ReStructuredText elements are commonly used in ASP.NET documentation articles. Note that **indentation and blank lines are significant!**

Inline Markup

Surround text with:

- One asterisk for **emphasis** (*italics*)
- Two asterisks for ****strong emphasis**** (**bold**)
- Two backticks for `“code samples”` (an `<html>` element)

..note:: Inline markup cannot be nested, nor can surrounded content start or end with whitespace (`*foo*` is wrong).

Escaping is done using the `\` backslash.

Format specific items using these rules:

- **Italics (surround with *)**
 - Files, folders, paths (for long items, split onto their own line)
 - New terms
 - URLs (unless rendered as links, which is the default)
- **Strong (surround with **)**
 - UI elements
- **Code Elements (surround with “)**
 - Classes and members
 - Command-line commands
 - Database table and column names
 - Language keywords

Links

Inline hyperlinks are formatted like this:

```
Learn more about `ASP.NET <http://www.asp.net>`_.
```

Learn more about [ASP.NET](#).

Surround the link text with backticks. Within the backticks, place the target in angle brackets, and ensure there is a space between the end of the link text and the opening angle bracket. Follow the closing backtick with an underscore.

In addition to URLs, documents and document sections can also be linked by name:

```
For example, here is a link to the `Inline Markup`_ section, above.
```

For example, here is a link to the *Inline Markup* section, above.

Any element that is rendered as a link should not have any additional formatting or styling.

Lists

Lists can be started with a `-` or `*` character:

```
- This is one item
- This is a second item
```

Numbered lists can start with a number, or they can be autonumbered by starting each item with the `#` character. Please use the `#` syntax.

```
1. Numbered list item one.(don't use numbers)
2. Numbered list item two.(don't use numbers)

#. Auto-numbered one.
#. Auto-numbered two.
```

Source Code

Source code is very commonly included in these articles. Images should never be used to display source code - instead use `code-block` or `literalinclude`. You can refer to it using the `code-block` element, which must be declared precisely as shown, including spaces, blank lines, and indentation:

```
.. code-block:: c#

public void Foo()
{
    // Foo all the things!
}
```

This results in:

```
public void Foo()
{
    // Foo all the things!
}
```

The code block ends when you begin a new paragraph without indentation. Sphinx supports quite a few different languages. Some common language strings that are available include:

- `c#`
- `javascript`
- `html`

Code blocks also support line numbers and emphasizing or highlighting certain lines:

```
.. code-block:: c#
   :linenos:
   :emphasize-lines: 3

public void Foo()
{
    // Foo all the things!
}
```

This results in:

```
1 public void Foo()  
2 {  
3     // Foo all the things!  
4 }
```

Note: `caption` and `name` will result in a code-block not being displayed due to our builds using a Sphinx version prior to version 1.3. If you don't see a code block displayed above this note, it's most likely because the version of Sphinx is < 1.3.

Images

Images such as screen shots and explanatory figures or diagrams should be placed in a `_static` folder within a folder named the same as the article file. References to images should therefore always be made using relative references, e.g. `article-name/style-guide/_static/asp-net.png`. Note that images should always be saved as all lower-case file names, using hyphens to separate words, if necessary.

Note: Do not use images for code. Use `code-block` or `literalinclude` instead.

To include an image in an article, use the `.. image` directive:

```
.. image:: style-guide/_static/asp-net.png
```

Note: No quotes are needed around the file name.

Here's an example using the above syntax:



Images are responsively sized according to the browser viewport when using this directive. Currently the maximum width supported by the <http://docs.asp.net> theme is 697px.

Notes

To add a note callout, like the ones shown in this document, use the `.. note::` directive.

```
.. note:: This is a note.
```

This results in:

Note: This is a note.

Including External Source Files

One nice feature of ReStructuredText is its ability to reference external files. This allows actual sample source files to be referenced from documentation articles, reducing the chances of the documentation content getting out of sync with the actual, working code sample (assuming the code sample works, of course). However, if documentation articles are referencing samples by filename and line number, it is important that the documentation articles be reviewed whenever changes are made to the source code, otherwise these references may be broken or point to the wrong line number. For this reason, it is recommended that samples be specific to individual articles, so that updates to the sample will only affect a single article (at most, an article series could reference a common sample). Samples should therefore be placed in a subfolder named the same as the article file, in a sample folder (e.g. /article-name/sample/).

External file references can specify a language, emphasize certain lines, display line numbers (recommended), similar to *Source Code*. Remember that these line number references may need to be updated if the source file is changed.

```
.. literalinclude:: style-guide/_static/startup.cs
   :language: c#
   :emphasize-lines: 19,25-27
   :linenos:
```

```
1  using System;
2  using Microsoft.AspNet.Builder;
3  using Microsoft.AspNet.Hosting;
4  using Microsoft.AspNet.Http;
5  using Microsoft.Framework.DependencyInjection;
6
7  namespace ProductsDnx
8  {
9      public class Startup
10     {
11         public Startup(IHostingEnvironment env)
12         {
13         }
14
15         // This method gets called by a runtime.
16         // Use this method to add services to the container
17         public void ConfigureServices(IServiceCollection services)
18         {
19             services.AddMvc();
20         }
21
22         // Configure is called after ConfigureServices is called.
23         public void Configure(IApplicationBuilder app, IHostingEnvironment env)
24         {
25             app.UseStaticFiles();
26             // Add MVC to the request pipeline.
27             app.UseMvc();
28         }
29     }
30 }
```

You can also include just a section of a larger file, if desired:

```
.. literalinclude:: style-guide/_static/startup.cs
   :language: c#
   :lines: 1,4,20-
   :linenos:
```

This would include the first and fourth line, and then line 20 through the end of the file.

Literal includes also support *Captions* and names, as with `code-block` elements. If the `caption` is left blank, the file name will be used as the caption. Note that captions and names are available with Sphinx 1.3, which the ReadTheDocs theme used by this system is not yet updated to support.

Tables

Tables can be constructed using grid-like “ASCII Art” style text. In general they should only be used where it makes sense to present some tabular data. Rather than include all of the syntax options here, you will find a detailed reference at <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#grid-tables>.

UI navigation

When documenting how a user should navigate a series of menus, use the `:menuselection:` directive:

```
:menuselection:`Windows --> Views --> Other...`
```

This will result in *Windows* → *Views* → *Other...*

Additional Reading

Learn more about Sphinx and ReStructuredText:

- [Sphinx documentation](#)
- [RST Quick Reference](#)

Summary

This style guide is intended to help contributors quickly create new articles for docs.asp.net. It includes the most common RST syntax elements that are used, as well as overall document organization guidance. If you discover mistakes or gaps in this guide, please [submit an issue](#).

Related Resources

- [.NET Core Documentation](#)

Contribute

The documentation on this site is the handiwork of our many [contributors](#).

We accept pull requests! But you're more likely to have yours accepted if you follow these guidelines:

1. Read <https://github.com/aspnet/Docs/blob/master/CONTRIBUTING.md>
2. Follow the *ASP.NET Docs Style Guide*