Create a Web API in MVC 6

By [Mike Wasson](http://www.asp.net/vnext/overview/aspnet-vnext/create-a-web-api-with-mvc-6#author-36870)Mike Wasson|last updated January 16, 2015

One of the goals of ASP.NET 5.0 is to unify the MVC and Web API frameworks.

In this topic you will learn:

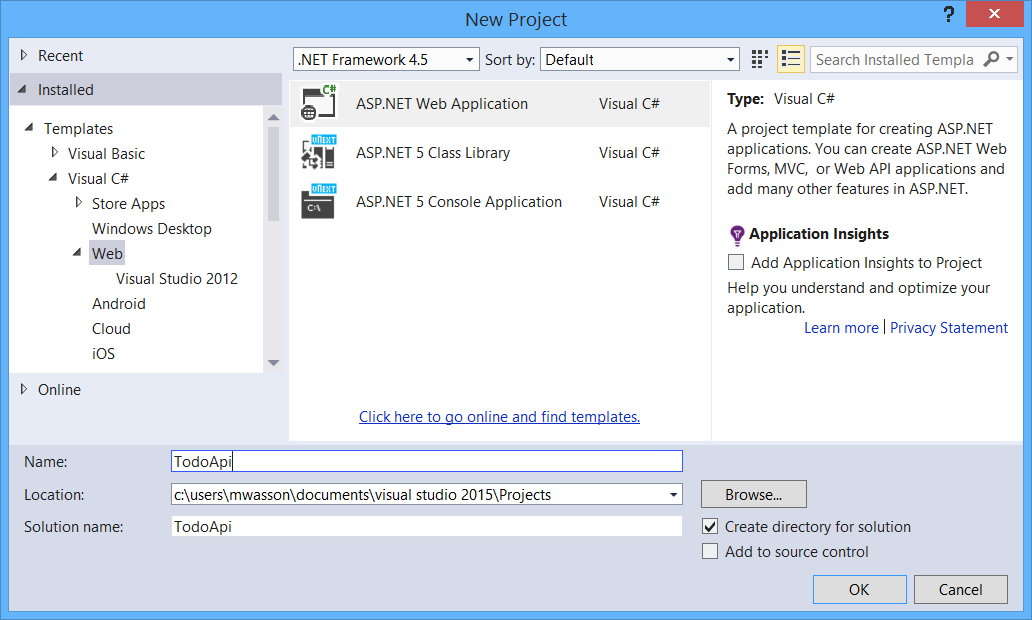
* How to create a simple web API in ASP.NET MVC 6.
* How to start from the Empty project template and add components to your app, as you need them.
* How to configure the ASP.NET 5.0 pipeline.
* How to self-host the app outside of IIS.

My approach in this tutorial is to start with nothing and build up the app. You can also start with the “Starter Web” template, which configures MVC 6, authentication, logging, and other features for you, and includes example controllers and views. Either approach is valid.

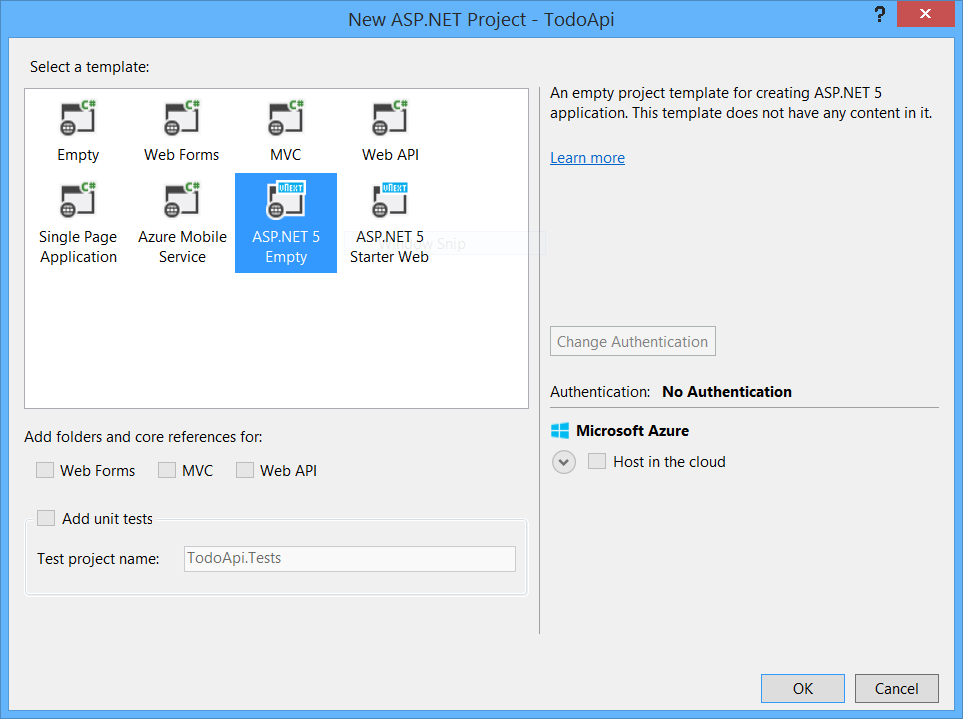
Create an Empty ASP.NET 5 Project

Start Visual Studio 2015. From the **File** menu, select **New** > **Project**.

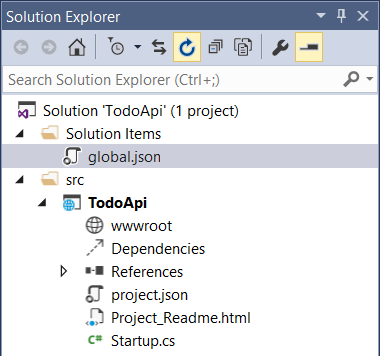
In the **New Project** dialog, click **Templates** > **Visual C#** > **Web**, and select the ASP.NET Web Application project template. Name the project "TodoApi" and click OK.



In the **New ASP.NET Project** dialog, select the "ASP.NET 5.0 Empty" template.



The following screen shot shows the project structure.



The project includes these files:

* global.json contains solution-level settings, and enables project-to-project references.
* project.json contains project settings.
* Project\_Readme.html is a readme file.
* Startup.cs contains startup and configuration code.

The Startup class, defined in Startup.cs, configures the ASP.NET request pipeline. When you use the empty project template, the Startup class starts out with literally nothing added to the pipeline:

public class Startup

{

public void Configure(IApplicationBuilder app)

{

// Nothing here!

}

}

You can run the app now, but it has no functionality. We’ll be adding functionality as we go. By comparison, the "Starter Web" template configures various parts of the framework, such as MVC 6, Entity Framework, authentication, logging, and so forth.

Add a Welcome Page

Open the project.json file. This file contains project settings for the app. The dependencies section lists required NuGet packages and class libraries. Add the Microsoft.AspNet.Diagnostics package to this list:

"dependencies": {

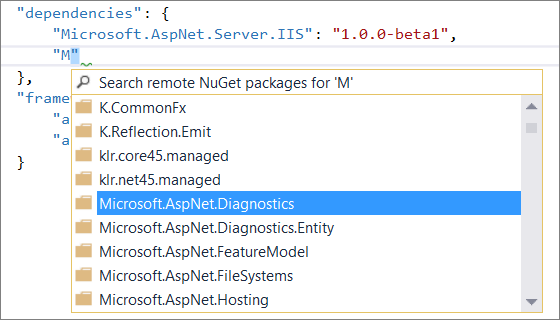
"Microsoft.AspNet.Server.IIS": "1.0.0-beta1",

// Add this:

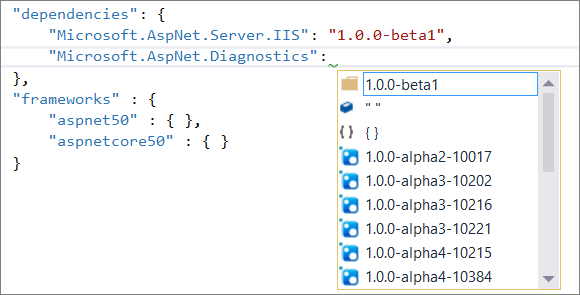
"Microsoft.AspNet.Diagnostics": "1.0.0-beta1"

},

As you type, Visual Studio gives you IntelliSense for a list of available packages.



You also get IntelliSense for the package version:



Next, open Startup.cs and add the code shown below.

using System;

using Microsoft.AspNet.Builder;

namespace TodoApi

{

public class Startup

{

public void Configure(IApplicationBuilder app)

{

// New code

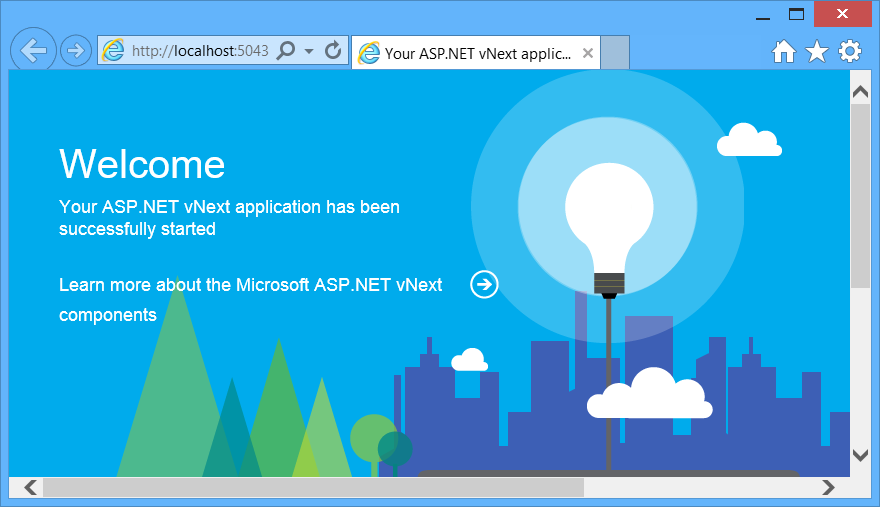
app.UseWelcomePage();

}

}

}

Press F5 to start debugging. Visual Studio launches a browser and navigates to http://localhost:*port*/, where *port* is a random port number, assigned by Visual Studio. You should see a welcome page that looks like this:



The welcome page is a quick way to see something running, without writing application code.

Create a Web API

In this section, you’ll create a web API to manage a list of ToDo items. First, we need to add ASP.NET MVC 6 to the app.

Add the MVC 6 package to the list of dependencies in project.json:

"dependencies": {

"Microsoft.AspNet.Server.IIS": "1.0.0-beta1",

"Microsoft.AspNet.Diagnostics": "1.0.0-beta1",

// New:

"Microsoft.AspNet.Mvc": "6.0.0-beta1"

},

Next, add MVC to the request pipeline. In Startup.cs,

1. Add a using statement for Microsoft.Framework.DependencyInjection.
2. Add the following method to the Startup class.
3. public void ConfigureServices(IServiceCollection services)
4. {
5. services.AddMvc();

}

This code adds all of the dependencies that MVC 6 requires. The framework automatically calls ConfigureServices on startup.

1. In the Configure method, add the following code. The UseMvc method adds MVC 6 to the pipeline.
2. public void Configure(IApplicationBuilder app)
3. {
4. // New:
5. app.UseMvc();

}

Here is the complete Startup class after these changes:

using System;

using Microsoft.AspNet.Builder;

using Microsoft.AspNet.Http;

// New using:

using Microsoft.Framework.DependencyInjection;

namespace TodoApi

{

public class Startup

{

// Add this method:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

}

public void Configure(IApplicationBuilder app)

{

// New:

app.UseMvc();

app.UseWelcomePage();

}

}

}

Add a Model

A *model* is a class that represents domain data in an application. In this case, the model is a ToDo item. Add the following class to the project:

using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models

{

public class TodoItem

{

public int Id { get; set; }

[Required]

public string Title { get; set; }

public bool IsDone { get; set; }

}

}

To keep the project organized, I created a Models folder and put the model class there. You don’t need to follow this convention.

Add a Controller

A *controller* is a class that handles HTTP requests. Add the following class to the project:

using Microsoft.AspNet.Mvc;

using System.Collections.Generic;

using System.Linq;

using TodoApi.Models;

namespace TodoApi.Controllers

{

[Route("api/[controller]")]

public class TodoController : Controller

{

static readonly List<TodoItem> \_items = new List<TodoItem>()

{

new TodoItem { Id = 1, Title = "First Item" }

};

[HttpGet]

public IEnumerable<TodoItem> GetAll()

{

return \_items;

}

[HttpGet("{id:int}", Name = "GetByIdRoute")]

public IActionResult GetById (int id)

{

var item = \_items.FirstOrDefault(x => x.Id == id);

if (item == null)

{

return HttpNotFound();

}

return new ObjectResult(item);

}

[HttpPost]

public void CreateTodoItem([FromBody] TodoItem item)

{

if (!ModelState.IsValid)

{

Context.Response.StatusCode = 400;

}

else

{

item.Id = 1+ \_items.Max(x => (int?)x.Id) ?? 0;

\_items.Add(item);

string url = Url.RouteUrl("GetByIdRoute", new { id = item.Id },

Request.Scheme, Request.Host.ToUriComponent());

Context.Response.StatusCode = 201;

Context.Response.Headers["Location"] = url;

}

}

[HttpDelete("{id}")]

public IActionResult DeleteItem(int id)

{

var item = \_items.FirstOrDefault(x => x.Id == id);

if (item == null)

{

return HttpNotFound();

}

\_items.Remove(item);

return new HttpStatusCodeResult(204); // 201 No Content

}

}

}

I created a Controllers folder for the controller. Again, you don’t need to follow this convention.

We'll examine the code more closely in the next section. This controller implements some basic CRUD operations:

|  |  |
| --- | --- |
| **Request (HTTP method + URL)** | **Description** |
| GET /api/todo | Returns all ToDo items |
| GET /api/todo/id | Returns the ToDo item with the ID given in the URL. |
| POST /api/todo | Creates a new ToDo item. The client sends the ToDo item in the request body. |
| DELETE /api/todo/id | Deletes a ToDo item. |

For example, here is the HTTP request to get the list of ToDo items:

GET http://localhost:5000/api/todo HTTP/1.1

User-Agent: Fiddler

Host: localhost:5000

Here is the response:

HTTP/1.1 200 OK

Content-Type: application/json;charset=utf-8

Server: Microsoft-HTTPAPI/2.0

Date: Thu, 30 Oct 2014 22:40:31 GMT

Content-Length: 46

[{"Id":1,"Title":"First Item","IsDone":false}]

Exploring the Code

In this section, I’ll briefly explain some of the code in the TodoController class.

Routing

The **[Route]** attribute defines a URL template for the controller:

[Route("api/[controller]")]

Any HTTP requests that match the template are routed to the controller. In this example, “[controller]” means to substitute the controller class name, minus the “Controller” suffix. For the TodoController class, therefore, the route template is “api/todo”.

HTTP methods

The **[HttpGet]**, **[HttpPost]** and **[HttpDelete]** attributes define the HTTP methods for the controller actions. (There are also **[HttpPut]** and **[HttpPatch]** attributes, not used in this tutorial.)

[HttpGet]

public IEnumerable<TodoItem> GetAll() {}

[HttpGet("{id:int}", Name = "GetByIdRoute")]

public IActionResult GetById (int id) {}

[HttpPost]

public void CreateTodoItem([FromBody] TodoItem item) {}

[HttpDelete("{id:int}")]

public IActionResult DeleteItem(int id) {}

In the case of GetById and DeleteItem, the string argument adds more segments to the route. So for these methods, the complete route template is “api/[controller]/{id:int}”.

In the “{id:int}” segment, *id* is a variable, and “:int” constrains the variable to match an integer. So these URLs would match:

http://localhost/api/todo/1

http://localhost/api/todo/42

but not:

http://localhost/api/todo/abc

Notice that GetById and DeleteItem also have method parameters named *id*. The framework populates the value of the *id* parameter from the corresponding URL segment. For example, if the request URL is http://localhost/api/todo/42, the value of *id* is set to 42. This process is called *parameter binding*.

The CreateTodoItem shows another form of parameter binding:

[HttpPost]

public void CreateTodoItem([FromBody] TodoItem item) {}

Here the **[FromBody]** attribute tells the framework to deserialize the TodoItem parameter from the request body.

The following table lists some example requests and the controller actions that match:

|  |  |
| --- | --- |
| **Request** | **Controller Action** |
| GET /api/todo | GetAll |
| POST /api/todo | CreateTodoItem |
| GET /api/todo/1 | GetById |
| DELETE /api/todo/1 | DeleteItem |
| GET /api/todo/abc | none – returns 404 |
| PUT /api/todo | none – returns 404 |

The last two examples return 404 errors, but for different reasons. In the case of 'GET /api/todo/abc', the 'abc' segment does not match the integer constraint for the GetById method. In the case of 'PUT /api/todo', there is no controller action with the **[HttpPut]** attribute.

Action Return Values

The TodoController class shows several ways to return a value from a controller action.

The GetAll method returns a CLR object.

[HttpGet]

public IEnumerable<TodoItem> GetAll()

{

return \_items;

}

The returned object is serialized in the body of the response message. The default format is JSON, but the client can request another format. For example, here is an HTTP request that asks for an XML response.

GET http://localhost:5000/api/todo HTTP/1.1

User-Agent: Fiddler

Host: localhost:5000

Accept: application/xml

Response:

HTTP/1.1 200 OK

Content-Type: application/xml;charset=utf-8

Server: Microsoft-HTTPAPI/2.0

Date: Thu, 30 Oct 2014 22:40:10 GMT

Content-Length: 228

<ArrayOfTodoItem xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/TodoApi.Models"><TodoItem><Id>1</Id><IsDone>false</IsDone><Title>First Item</Title></TodoItem></ArrayOfTodoItem>

The GetById method returns an IActionResult:

[HttpGet("{id:int}", Name = "GetByIdRoute")]

public IActionResult GetById (int id)

{

var item = \_items.FirstOrDefault(x => x.Id == id);

if (item == null)

{

return HttpNotFound();

}

return new ObjectResult(item);

}

The method returns ObjectResult if it finds a ToDo item with a matching ID. Returning ObjectResult is equivalent to returning the CLR model, but it makes the return type IActionResult. That way, the method can return a different action result for other code paths.

When the ID is not found, the method returns HttpNotFound, which translates into a 404 response.

Finally, the CreateTodoItem method shows how to create the response by setting values directly on the Response property in the controller. In this case, the return type is void.

[HttpPost]

public void CreateTodoItem([FromBody] TodoItem item)

{

// (some code not shown here)

Context.Response.StatusCode = 201;

Context.Response.Headers["Location"] = url;

}

A drawback to this approach is that it’s harder to unit test. (For a relevant discussion of testing action results, see [Unit Testing Controllers in ASP.NET Web API](http://www.asp.net/web-api/overview/testing-and-debugging/unit-testing-controllers-in-web-api). That topic is specifically about ASP.NET Web API, but the general ideas apply.)

Dependency Injection

MVC 6 has dependency injection built in to the framework. To see this, let’s create a repository class to hold the list of ToDo items.

First, define an interface for the repository:

using System.Collections.Generic;

namespace TodoApi.Models

{

public interface ITodoRepository

{

IEnumerable<TodoItem> AllItems { get; }

void Add(TodoItem item);

TodoItem GetById(int id);

bool TryDelete(int id);

}

}

Then define a concrete implementation.

using System;

using System.Collections.Generic;

using System.Linq;

namespace TodoApi.Models

{

public class TodoRepository : ITodoRepository

{

readonly List<TodoItem> \_items = new List<TodoItem>();

public IEnumerable<TodoItem> AllItems

{

get

{

return \_items;

}

}

public TodoItem GetById(int id)

{

return \_items.FirstOrDefault(x => x.Id == id);

}

public void Add(TodoItem item)

{

item.Id = 1 + \_items.Max(x => (int?)x.Id) ?? 0;

\_items.Add(item);

}

public bool TryDelete(int id)

{

var item = GetById(id);

if (item == null)

{

return false;

}

\_items.Remove(item);

return true;

}

}

}

Use constructor injection to inject the repository into the controller:

[Route("api/[controller]")]

public class TodoController : Controller

{

// Remove this code:

//static readonly List<TodoItem> \_items = new List<TodoItem>()

//{

// new TodoItem { Id = 1, Title = "First Item" }

//};

// Add this code:

private readonly ITodoRepository \_repository;

public TodoController(ITodoRepository repository)

{

\_repository = repository;

}

Then update the controller methods to use the repository:

[HttpGet]

public IEnumerable<TodoItem> GetAll()

{

return \_repository.AllItems;

}

[HttpGet("{id:int}", Name = "GetByIdRoute")]

public IActionResult GetById(int id)

{

var item = \_repository.GetById(id);

if (item == null)

{

return HttpNotFound();

}

return new ObjectResult(item);

}

[HttpPost]

public void CreateTodoItem([FromBody] TodoItem item)

{

if (!ModelState.IsValid)

{

Context.Response.StatusCode = 400;

}

else

{

\_repository.Add(item);

string url = Url.RouteUrl("GetByIdRoute", new { id = item.Id }, Request.Scheme, Request.Host.ToUriComponent());

Context.Response.StatusCode = 201;

Context.Response.Headers["Location"] = url;

}

}

[HttpDelete("{id}")]

public IActionResult DeleteItem(int id)

{

if (\_repository.TryDelete(id))

{

return new HttpStatusCodeResult(204); // 201 No Content

}

else

{

return HttpNotFound();

}

}

For dependency injection to work, we need to register the repository with the dependency injection system. In the Startup class, add the following code:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

// New code

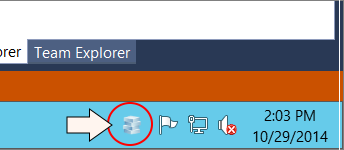
services.AddSingleton<ITodoRepository, TodoRepository>();

}

When the application runs, the framework automatically injects the TodoRepository into the controller, whenever it creates a controller instance. Because we used AddSingleton to register ITodoRepository, the same instance is used throughout the lifetime of the app.

Run the app outside of IIS.

By default, when you press F5, the app runs in IIS Express. You can see this by the IIS Express icon that appears in the taskbar.



ASP.NET 5.0 is designed with a [pluggable server layer](https://github.com/aspnet/Home/wiki/Servers). That means you can swap in a different web server. In this section, we’ll switch to WebListener, which runs directly on the Http.Sys kernel driver, instead of running inside IIS.

To be clear, there are many advantages to running inside IIS (security, process management, management GUI, and so on). The point is that ASP.NET 5.0 is not directly tied to IIS. With that said, let's run the app hosted inside a console app.

In the project.json file, add the Microsoft.AspNet.Server.WebListener package:

"dependencies": {

"Microsoft.AspNet.Server.IIS": "1.0.0-beta1",

"Microsoft.AspNet.Diagnostics": "1.0.0-beta1",

"Microsoft.AspNet.Mvc": "6.0.0-beta1",

// New:

"Microsoft.AspNet.Server.WebListener": "6.0.0-beta1"

},

Then add the following top-level option to project.json.

{

// Other sections not shown

"commands": {

"web ": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5000"

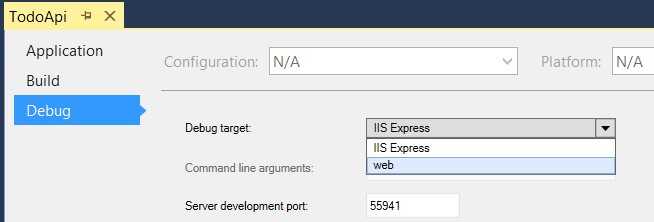
}

}

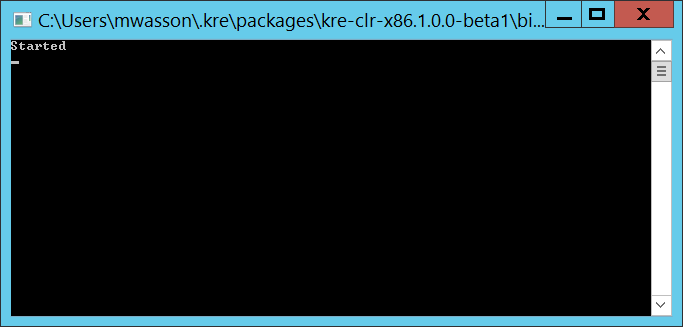
The “commands” option contains a list of pre-defined commands that you can pass to the K runtime. In this example, “web” is the name of the command, which can be an arbitrary string. The value is the actual command.

* Microsoft.AspNet.Hosting is an assembly that is used to host ASP.NET 5.0 application. The assembly contains an entry point that is used for self-hosting.
* The --server flag specifies the server, in this case WebListener.
* The --server.urls flag gives the URL to listen on.

Save the project.json file. Then, in Solution Explorer, right click the project and select **Properties**. In the **Properties** tab, click **Debug**. Under **Debug target**, change the dropdown from “IIS Express” to “web”



Now hit F5 to run the app. Instead of starting IIS Express and opening a browser window, Visual Studio runs a console app that starts the WebListener.



Open a browser and navigate to http://localhost:5000. (That’s the value that you specified in the project.json file.) You should see the welcome page.

To run in IIS Express again, just switch the Debug Target back to “IIS Express”.

*This article was originally created on November 12, 2014*