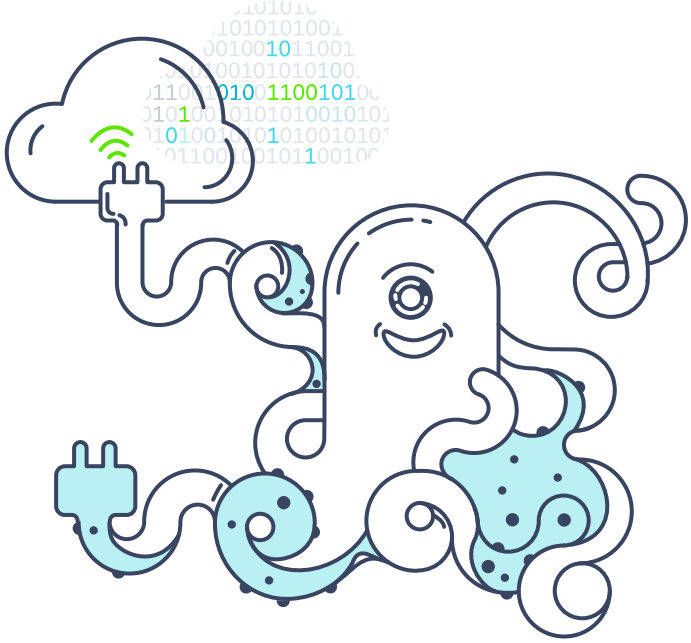
Progress®

ASP.NET CORE MVC CHANGES EVERY DEVELOPER SHOULD KNOW

CONTENTS

OVERVIEW



ASP.NET Core MVC (formerly MVC 6) is a ground-up rewrite of the popular .NET web framework. While the fundamental concepts of Model View Controller remain the same, many of the underlying layers of the framework have changed. This new version of MVC pushes the framework forward with improved modularity, cross-platform adoption and web standardization.

Modularity

ASP.NET Core MVC enables developers to use as little or as many features as they see fit. In this version, developers have the option to write directly "to the metal." In other words, developers can create **middleware** that interacts directly with the **request pipeline**. In ASP.NET Core MVC itself consists of a collection of **middleware** or modules that can be added to the pipeline to compose an application.

ASP.NET Core runs just as a normal console application. In the chapter Taking Control of Configuration in ASP.NET Core MVC, we'll look at how an ASP.NET Core MVC applications entry point and how the application is initialized.

Cross-Platform

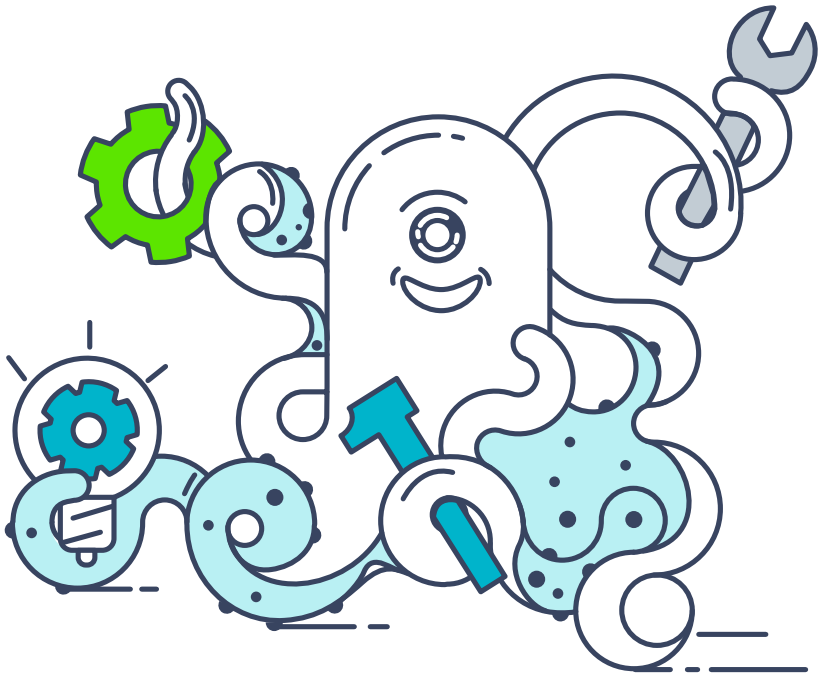
Rewriting the framework from scratch provides an added benefit: there is no longer a dependency on IIS. ASP.NET applications can run in new environments. Now, it's possible to self-host MVC applications, as well as host them on Linux, OS X, and Docker platforms.

Web Standardization

Web development technologies move at a rapid pace. New tools such as Bower, Grunt, Gulp and Node have become mainstream in a relatively short period of time. Microsoft chose to not resist the trend but embrace it by supporting these new tools and workflows in Visual Studio and in Core MVC. The tools also create common ground for cross-platform developers.

Embracing Change

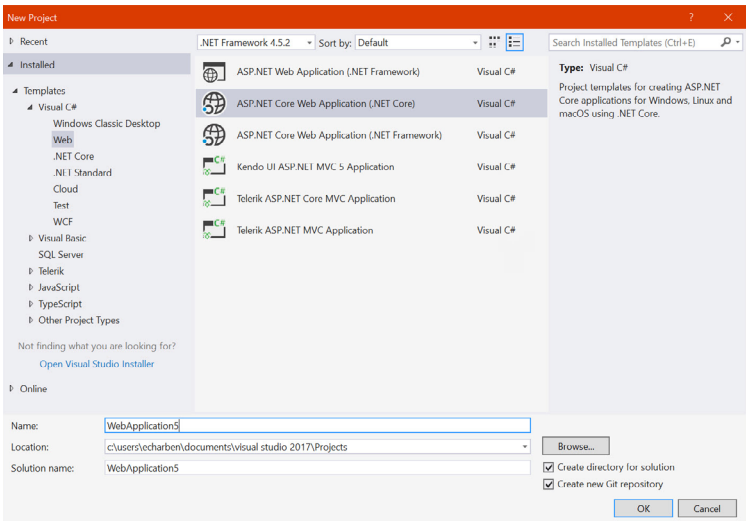
Understanding why MVC is changing is only part of learning about what's new in Core MVC. Throughout this piece, we'll look at big changes vital to developing an Core MVC project such as **the new project template, routing, configuration, dependency injection and tag helpers**. We'll explore each of these topics through practical examples, and get an understanding of how to implement them in a project.

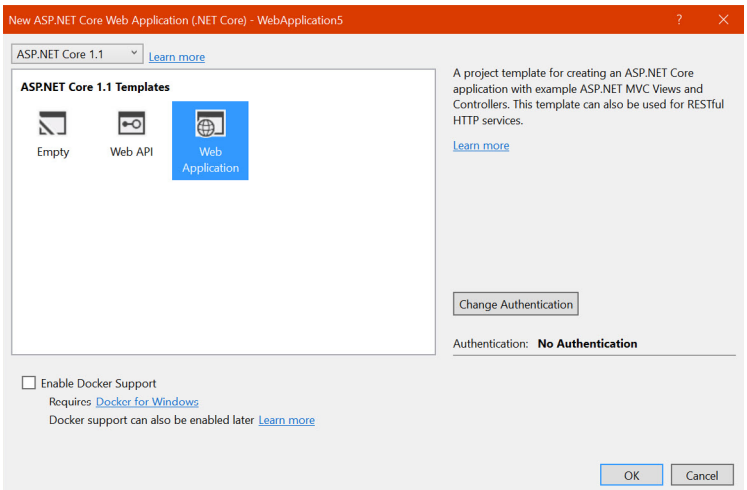


ASP.NET CORE MVC NEW PROJECT TOUR

Sweeping changes were made throughout MVC; even some of the most basic elements have been reorganized. These changes are apparent immediately, when starting a new Core MVC project, especially if you're familiar with previous versions of the framework.

Let's click File > New Project and take a tour of the new Core MVC project template. We'll look at what's missing from MVC 5, what we can expect to stay the same and what's new.





What's Missing

Before beginning work on a new project, it's important to understand where some familiar items have gone. Considering Core MVC is a complete rewrite, you should expect some changes; however, you may be surprised to find some key items missing.

We'll explain all items with replacement counterparts in detail under "What's new."

- **App_Start:** The App_Start folder previously contained various startup processes and settings such as configuration, identity and routing. Those items have been replaced by the **Startup.cs** class, which is now responsible for all app startup tasks.
- **App_data:** The App_data folder once held application data such as local database files and log files. The folder isn't included in this release, but you can add it back. If you choose to use the app_data folder, proceed with caution as to not make files publicly available by accident. See: App_Data directory in ASP.NET5 Core MVC on Stack Overflow for details.
- **Global.ASAX:** The Global.ASAX is no longer needed, because it was yet another place for startup routines. Instead, all startup functionality has been placed in startup.cs.
- **Web.Config:** The Web.Config was once the XML equivalent to a settings junk drawer; now all application settings are found in **appsettings.json**.
  Note: A Web.Config can still be found in MVC for configuring static application resources.
- **Scripts:** The scripts directory used to house the application's JavaScript files has a new home. All JavaScript files now reside under wwwroot/js as a static resource.
- **Content:** Much like the aforementioned Scripts folder, static site resources are in wwwroot.

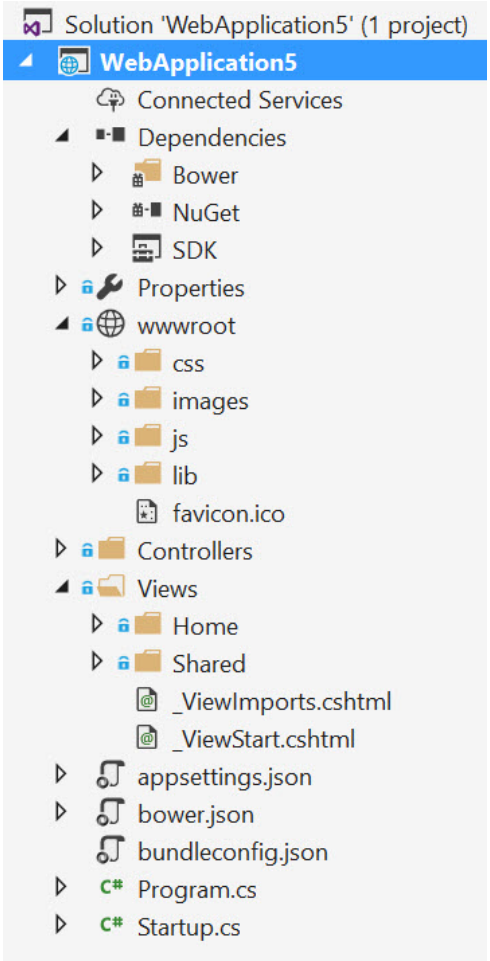
What's (Mostly) the Same

Very few things remain unchanged in the Core MVC project template. In fact, the only three items that stayed the same are the fundamental components of the MVC pattern itself: Models, Views and Controllers.

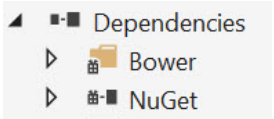
- **Models:** The models folder remains with a minor change: it now contains data Models only.
- **Views:** Views in Core MVC are as they were in previous versions; they are dynamic HTML (or .cshtml) rendered on the server before being sent to the client. Views contain the application UI and are, by default, built with Bootstrap. One new addition to the views folder is the **_ViewImports.cshtml**. The _ViewImports file provides namespaces that can be used by all other views. In previous MVC projects, this functionality was the responsibility of the web.config file in the Views folder. However, the web.config no longer exists, and global namespaces are now provided by _ViewImports.
- **Controllers:** In Core MVC, the controllers folder retains its responsibility to hold application controllers. Controllers were commonly used to return views, but can serve as Web API endpoints, now that Web API and MVC have merged. In Core MVC, both MVC controllers and Web API controllers use the same routes and Controller base class.

What's New

At first glance, you'll see a lot of new parts to an MVC project. From the root folder down, there are many new files and folders that come with all new conventions. Let's explore the new items and understand their purpose in the project.



- **Wwwroot:** The wwwroot folder is used by the host to serve static resources. Sub-folders include js (JavaScript), CSS, Images and lib. The lib folder contains third-party JavaScript libraries added via Bower package manager.
- **Dependencies:** More package management options are available in ASP.NET Core MVC. This version includes Bower and NPM support, which you can configure by GUI. Additionally, you can manage configuration by their respective .json files, found in the root src folder.



- **Data/Migrations:** ASP.NET Core MVC ships with Entity Framework Core, which no longer supports EDMX database modeling. Because EF Core focuses on code first, database creation, initialization and migration code are located in the migrations folder.
- **Services:** Services are at the forefront of ASP.NET Core MVC. Because ASP.NET Core MVC was built with dependency injection at its core, you can easily instantiate and use services throughout the application (see the chapter Dependency Injection in ASP.NET Core MVC for more details).
- **bower.json** & **package.json:** To support "all things web," ASP.NET Core MVC now offers first-class support for Bower and NPM. These popular package management systems were born from the web and open-source development communities. Bower hosts popular packages such as Bootstrap, while NPM brings in dependencies such as Gulp. The bower.json and package.json files are used to register and install Bower and NPM packages with full intellisense support. These files are hidden in the Solution Explorer window. *package.json is supported but not included by default.*

```
{
   "name": "ASP.NET",
   "private": true,
   "dependencies": {
      "bootstrap": "3.0.0" ,
      "bootstrap-touch-carousel": "0.8.0",
      "hammer.
      "jquery"-
      "jquery-
      "jquery-
   }
}
```

bootstrap-touch-carousel

github.com/ixisio/bootstrap-touch-carousel

bootstrap-touch-carousel

- **gulpfile.js:** Gulp is another tool built “for the web, by the web.” It is given first-class support in ASP.NET Core MVC. Gulp is a Node.js-based task runner with many plug-ins available from NPM. There are packages for compiling, minifying and bundling CSS. There are also packages for .NET developers for invoking MSBuild, NuGet, NUnit and more. gulpfile.js is where you can define Gulp tasks for the application. *Supported by not included by default*

- **startup.cs:** In previous versions, MVC application startup was handled in App_Start and Global.asax, with ASP.NET Core startup handled in Startup.cs. The Startup method is the first method in the application to run, and is only run once. During startup, the application's configuration is read, dependencies are resolved and injected, and routes created.
- **appsettings.json:** The appsettings.json file is the primary replacement for the **Web.Config**. In this file, you'll specify application settings such as connection string, email endpoints and more. The new configuration model supports the **JSON**.

Wrapping Up

The ASP.NET Core MVC project template embraces the web in many ways. From the root folder and below, most of the project structure has been revised to align with the ever-changing web. Including NPM and Bower support in addition to NuGet provides developers a wide range of options for bringing modular components to an application. While many things have changed in the project template, the core MVC components have remained.

"File > New Project" may be a bit intimidating at first, but knowing where to find each piece and its purpose will give you a head start.

ASP.NET CORE MVC ROUTING REDEFINED

Routing has evolved over the MVC framework's history. In ASP.NET Core MVC, routing continues to improve upon advances made in MVC 5. In this section, we'll look at those changes by examining where and how routes are defined, reviewing improvements made to attribute-based routes, and learning how special attributes can help fine-tune application routing.

Defining Routes

The ASP.NET Routing module is responsible for mapping incoming browser requests to particular MVC controller actions. Although defining routes in MVC is relatively simple, this task has a long history of being moved throughout the framework. In past versions, routes were defined in Global.ASAX or App_Start/RouteConfig.cs; in ASP.NET Core MVC, the location has changed yet again.

Routes in ASP.NET Core MVC are defined in the Configure method of Startup.cs, when MVC is registered as middleware, upon app.UseMvc execution.

Convention-Based Routing

Routes defined during configuration are convention-based routes. In ASP.NET Core MVC, you can define convention-based routes using an Action<IRouteBuilder>, which is a parameter of the UseMvc method.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
        {

        // Add MVC to the request pipeline.
            app.UseMvc(routes =>
            {
                // Route to "~/Home/Index/123"
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
                // Route to "~/public/blog/posts/show/123"
                routes.MapRoute(
                    name: "blogPost",
                    template: "public/blog/{controller}/{action}/{postId}");
            });

        }
```

While convention-based routes look and work much like they did in previous versions, a new default way of specifying default values has been added. Now you can define default values for controller, action or value in-line; previously, you had to set them in an additional attribute. Simply using an equal sign inside the template specifies which value to use if no value is present {controller=defalutValue}. Likewise, you can use nullable types as placeholders for optional parameters {id?}.

```
// defaults in ASP.NET Core MVC
routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");

// defaults in previous MVC
routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new
{
            controller = "Home",
            action = "Index",
            id = UrlParameter.Optional }
    );
```

While convention-based routes have been updated in this version, they are completely optional. In ASP.NET Core MVC, attribute-based routing is enabled from the start.

Attribute-Based Routing

Attribute-based routing is an improvement that debuted in MVC 5 and is enabled in ASP.NET Core MVC. Attribute-based routing uses attributes to define routes and gives finer control over the URIs in your web application.

The following code shows how you can use attribute-based routing to specify a route for /home/index. Notice we combine the controller-level attribute with the action-level attribute to complete the full route.

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("Index")]
    public IActionResult Index()
    {
        return View();
    }
}
```

You can combine multiple route attributes, as well. For example, you can add a secondary route of [Route("/")] to enable the index action to be visited from the path http://myapp.com/home/index as well as from the root of the URL of the application http://myapp.com.

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("Index")]
    [Route("/")]
    public IActionResult Index()
    {
        return View();
    }
}
```

In ASP.NET Core MVC, routes have new [controller] and [action] tokens that can reference controller and action names in the route template. These tokens enable you to name routes after their respective controller or action, while maintaining flexibility. If you rename the controller or action, the route changes without requiring you to update the route.

```
// Route to "~/public/blog/posts/show/123"
[Route("public/blog/[controller]")]
public class PostsController : Controller
{
    [Route("[action]/{id}")]
    public IActionResult Show(int id)
    {
        return View();
    }
}
```

Conclusion

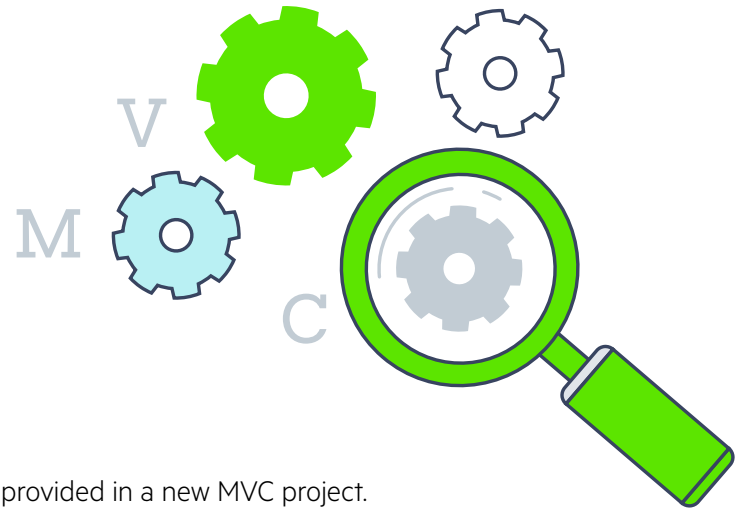
Routing in ASP.NET Core MVC provides multiple options for mapping URIs. The inclusion of both convention-based and attribute-based routes enables you to mix and match to fit your application's needs.

TAKING CONTROL OF CONFIGURATION IN ASP.NET CORE MVC

If you're coming to ASP.NET Core MVC from using prior versions, you'll quickly begin to notice changes. One of the major changes in this version is the lack of features in the Web.Config file. In this section, we'll learn where configuration has moved, what the new features are and how to get values from configuration to the application.

Where's My Web.Config?

XML configuration is a thing of the past. In ASP.NET Core MVC, the Web.Config was completely removed and the configuration experience overhauled. The new configuration consists of a variety of options, including JSON-based files and environment variables.



Let's take a look at what's provided in a new MVC project.

Application Initialization

In previous versions of ASP.NET applications were loaded by Internet Information Services (IIS) executable (InetMgr.exe) creates and calls a managed web application's entry point. The HttpApplication.Application_Start() event is fired during initialization. A developer's first chance to execute code was to handle the Application_Start event in Global.asax.

In an ASP.NET Core application, ASP.NET applications are a .NET Core Console application that calls into ASP.NET specific libraries. This is a fundamental change in how ASP.NET Core applications are developed. Instead of the application being hosted by IIS, all of the ASP.NET hosting libraries executed from within Program.cs. This means that a single .NET tool chain can be used for both .NET Core Console applications and ASP.NET Core applications. The benefits of the new architecture allow for more control by the developer over initialization and the hosting environment.

In this excerpt from the Main method of an ASP.NET Core app, Main is responsible for configuring and running the app. Additional settings and hosting functionality can be configured and added to the WebHostBuilder.

```
public class Program
{
  public static void Main(string[] args)
  {
    var host = new WebHostBuilder()
          .UseKestrel()
          .UseIISInegration()
          .UseStartup<Startup>()
          .Build();
      host.Run();
  }
}
```

Startup

ASP.NET Core provides complete control of how individual requests are handled by your application. The Startup class is called by the Main method of the application, this is where configuration takes place and services for the application are specified.

In the request pipeline, middleware components are initialized. The Configure method is used to specify how the ASP.NET application will respond to individual HTTP requests in the form of middleware. Middleware in ASP.NET Core are built from extension methods on IApplicationBuilder and give developers direct access to HTTP requests. Typical Use *Action* middleware will handle requests and responses and passes the request on to the next component in the pipeline for further action. In addition, Run and Map middleware delegates are used for less common scenarios, these delegates will terminate or branch the request pipeline.

New Configuration Options

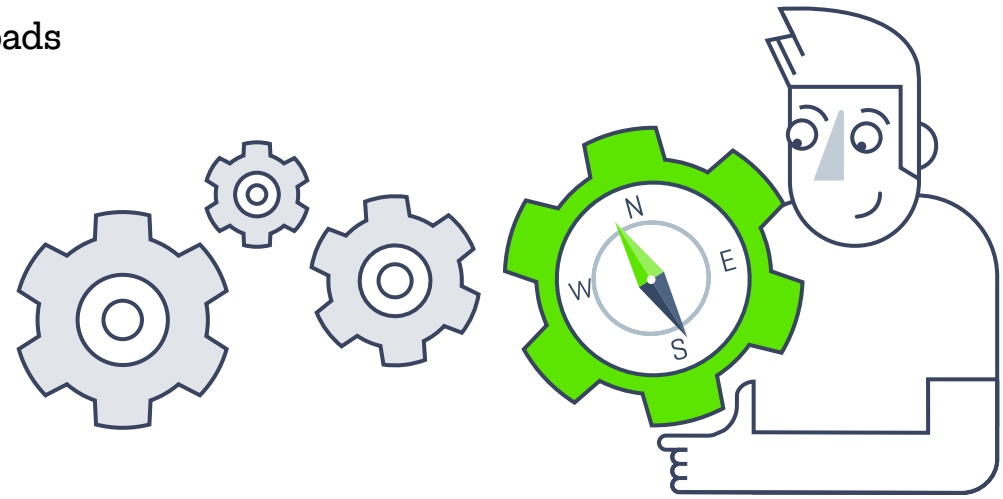
The new configuration options are available through the Startup routine in startup.cs. In the Startup, method is a ConfigurationBuilder; it is new to ASP.NET Core MVC and provides a chain-able API with which you can define multiple configurations. Out-of-the-box, you have two configuration options in Startup: appsettings.json and appsettings.{env.EnvironmentName}.json.

```
// Setup configuration sources.
    var builder = new ConfigurationBuilder()
            .SetBasePath(appEnv.ApplicationBasePath)
            .AddJsonFile("appsettings.json")
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);
```

The default configurations use AddJsonFile, which specifies that the configuration will be in JSON format. If JSON isn't your preferred format, multiple configuration builders are available, including INI, Command Line and Environment Variables. You can even create your own by implementing the IConfigurationBuilder interface.

In the standard MVC template, we have appsettings.json and a second optional appsettings defined at runtime, based on the EnvironmentName, if available. This second appsettings comes in handy when you need to overload an option.

Overloads



In previous versions of MVC, managing application settings for different environments such as development and production meant using transformations. In ASP.NET Core MVC, this system is replaced by configuration overloading.

When you register a configuration in the ConfigurationBuilder, the last value applied wins over any previous value of the same key.

```
//foo.json
{ "mykey" : "Foo" }

//bar.json
{ "mykey" : "Bar" }

//startup.cs

var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    .AddJsonFile("foo.json")
    .AddJsonFile("bar.json");

Configuration.GetValue<string>("mykey"); // => Bar
```

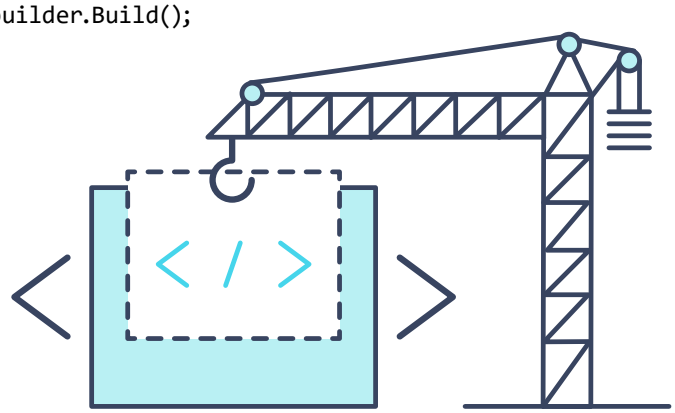
Using configuration overloads, we can have much greater control over how the application behaves in a given environment.

Getting Values from Configuration in ASP.NET Core MVC

Getting values from configuration has also changed from previous versions of MVC. Before, you could retrieve values from the ConfigurationManager at the controller level; however, in ASP.NET Core MVC, the values are available to the application through dependency injection.

At the end of the Startup method, we use ConfigurationBuilder to create a single instance of IConfiguration when the Build method is called.

```
Configuration = builder.Build();
```



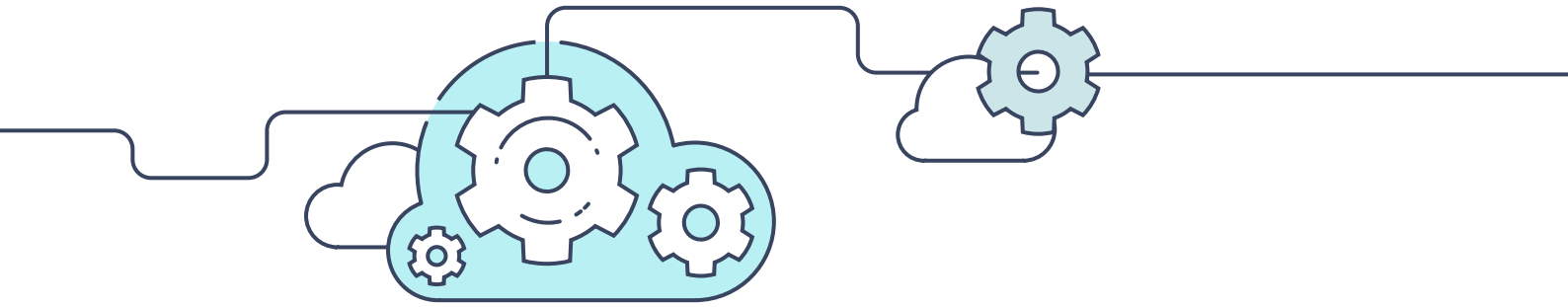
Once the configuration is built, the values are injected into the project in the ConfigureServices method. Inside the ConfigureServices method, you can add settings using services.Configure<T>. You can fetch the value from configuration by convention, when the key matches the property name, or by specifying the key explicitly.

In this example, we fetch MyOptions, a POCO with the property ConfigMessage, from configuration and add it to the services collection.

```
//MyOptions.cs (POCO)
public class MyOptions
{
    public string ConfigMessage { get; set; }
}

//appsettings.json
{
    "MyOptions": {
        "ConfigMessage": "Hello from appsettings.json"
    }
}

//Startup.cs

//Add framework services.
services.Configure<MyOptions>(Configuration.GetSection("MyOptions"));
```

We make the values available to the application's controller by adding the options to any controller's constructor. Since we're using the dependency injection that has already been made available in ASP.NET Core MVC, we'll just need to reference the value. Using IOptions<T> as an argument in the constructor will resolve the dependency. This works because the service provider is resolved for us when we create a controller instance.

```
//HomeController.cs
private readonly string configMessage;
public HomeController(IOptions<MyOptions> myConfigOptions)
{
    configMessage = myConfigOptions.Value.ConfigMessage;
}

public IActionResult Index()
{
    ViewData["Message"] = configMessage; //=> Hello from appsettings.json
    return View();
}
```



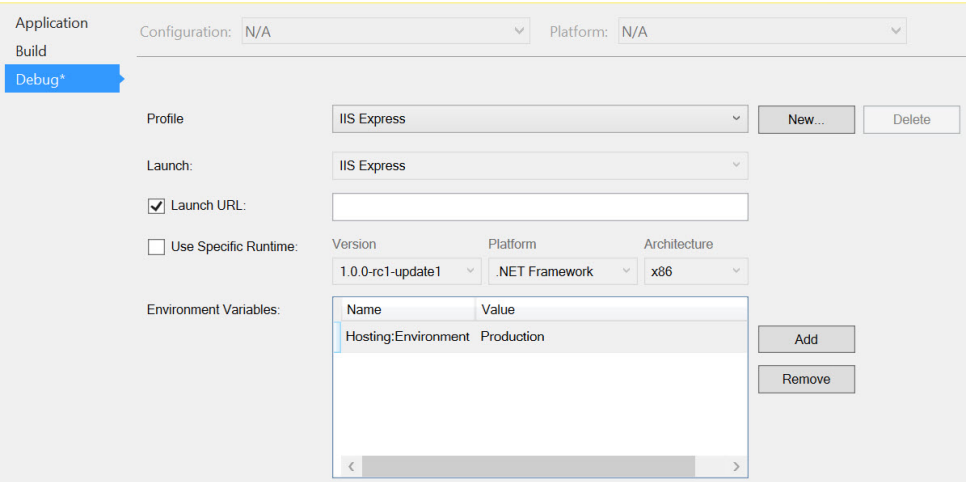
Having seen how appsettings.json works, what overloads do and how to retrieve values, we can put it to use. We'll continue with the MyOptions example and see how different overrides take place in scenarios such development, staging and cloud deployment on Azure.

Toggling Config Files

First, let's revisit ConfigurationBuilder in Startup.cs. The configuration builder has two configs, appsettings.json and config.{env.EnvironmentName}.json. We can use the second config to override settings when specific environment variables are available. We will start by adding the following configs to our project:

```
//appsettings.development.json
{
  "MyOptions": {
    "ConfigMessage": "Hello from config.development.json"
  }
}

//appsettings.staging.json
{
  "MyOptions": {
    "ConfigMessage": "Hello from config.staging.json"
  }
}

//appsettings.production.json
{
  "MyOptions": {
    "ConfigMessage": "Hello from config.production.json"
  }
}
```

Running the application with the default environment variables will use the ConfigMessage "Hello from config.development.json." This is because the default EnvironmentName is "development." Changing the Hosting:Environment value to "staging" or "production" will get the configuration from the corresponding config file. To see this in action using Visual Studio, open the project's properties and change Hosting:Environment value under the debug menu.



Overriding Using Environment Variables

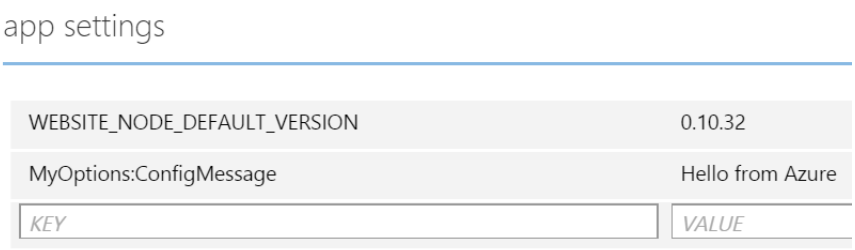
In addition to toggling appsettings files, you can override set values through environment variables. Environment variables are ideal for Azure deployment because they separate the server configuration from the project. Using environment variables for values such as database connection strings and API keys will ensure the deployed application is always using production values.

Add environment variables to ConfigurationBuilder near the end of the Startup method using the AddEnvironmentVariables(); method:

```
public Startup(IHostingEnvironment env, IApplicationEnvironment appEnv)
    {
        // Setup configuration sources.

        var builder = new ConfigurationBuilder(appEnv.ApplicationBasePath)
            .AddJsonFile("appsettings.json")
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

        if (env.IsDevelopment())
        {
            // 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=532709
            builder.AddUserSecrets();
        }
        builder.AddEnvironmentVariables();
        Configuration = builder.Build();
    }
```

You can set a value in Azure through the management portal from the application's Configure menu. Under App Settings, enter a key/value pair that corresponds to the value needed by the application. If the setting has a hierarchy, be sure to use the full name space for the key using : as a separator.

Continuing with the example, we can set the ConfigMessage by adding an environment variable with the key MyOptions:ConfigMessage and the value Hello from Azure.

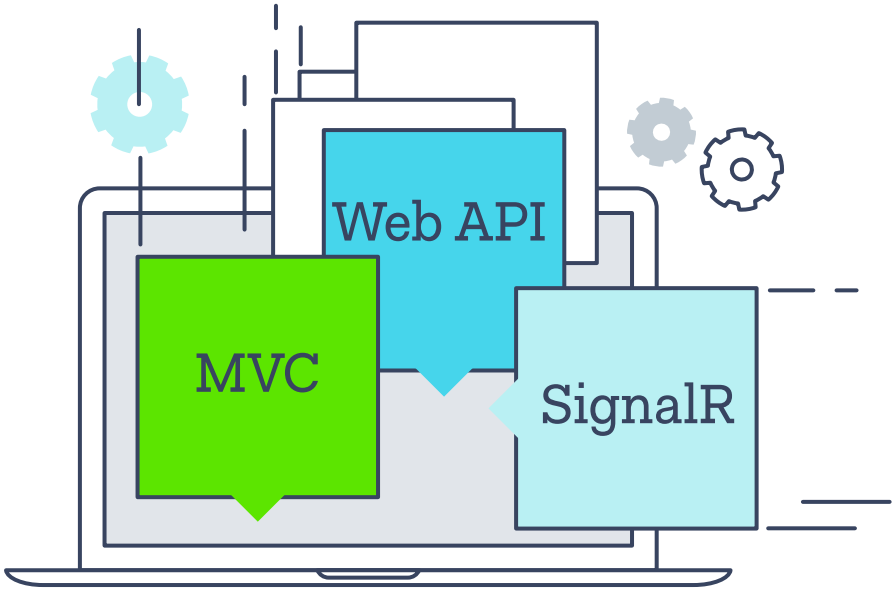


The application will only receive the value Hello from Azure when it is deployed; otherwise, it will get its value from the config file.

Conclusion

Configuration is completely different in ASP.NET Core MVC. The new implementation follows a more modular development approach common throughout ASP.NET Core MVC. In addition, the JSON format feels ubiquitous with modern web practices. The new style of configuration may come with a learning curve, but it allows for greater flexibility.

DEPENDENCY INJECTION IN ASP.NET CORE MVC



Dependency injection (DI) has been possible in previous versions of MVC. With each new version, DI has become easier to implement, and ASP.NET Core MVC supplies DI right out of the box. In this section, we'll look at how the new DI implementation works, its weaknesses and how can we replace it with our favorite DI framework.

What's New

The unification of APIs across ASP.NET is a common theme throughout ASP.NET Core 1.0, and DI is no different. The new ASP.NET stack, which includes MVC, SignalR, Web API and more, relies on a built-in minimalistic DI container. The core features of the DI container have been abstracted to the **IServiceProvider** interface and are available throughout the stack. Because the IServiceProvider is the same across all components of the ASP.NET framework, you can resolve a single dependency from any part of the application.

The DI container supports just four modes of operation:

- **Instance:** A specific instance is given all the time. You are responsible for its initial creation.
- **Transient:** A new instance is created every time.
- **Singleton:** A single instance is created and acts like a singleton.
- **Scoped:** A single instance is created inside the current scope. It is equivalent to Singleton in the current scope.

Basic Setup

Let's walk through setting up DI in an MVC application. To demonstrate the basics, we'll resolve the dependency for the service used to get project data. We don't need to know anything about the service ,other than that it implements the IProjectService interface, an interface custom to our demo project. IProjectService has one method, GetOrganization(). This method retrieves an organization and its corresponding list of projects.

```
public interface IProjectService
{
    string Name { get; }
    Organization GetOrganization();
}

public class Organization
{
    public string Name { get; set; }
    [JsonProperty("Avatar_Url")]
    public string AvatarUrl { get; set; }
    public IQueryable<Project> Projects { get; set; }
}
```

We'll use the IProjectService to get the organization data and display it in a view. Let's start by setting up the controller where the service will be used. We'll use constructor injection by creating a new constructor method for our controller that accepts an IProjectService. Next, the Index action will call GetOrganization, sending the data to the view to be rendered.

```
private readonly IProjectService projectService;
public HomeController(IProjectService projectService)
{
    this.projectService = projectService;
}
public IActionResult Index()
{
    Organization org = projectService.GetOrganization();
    return View(org);
}
```

If we try to run the application at this point, we'll receive an exception, because we haven't yet added a concrete implementation of our IProjectService to the DI container.

InvalidOperationException: Unable to resolve service for type 'DependencyInjection ASP.NET Core MVCDemo. Services. IProjectService' while attempting to activate 'DependencyInjection ASP.NET Core MVCDemo. Controllers. HomeController'.

Microsoft. Framework. DependencyInjection.

ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, Boolean isDefaultParameterRequired)

The exception message shows that the code fails during a call to ActivatorUtilities.GetService. This is valuable information, because it shows that in ASP.NET Core MVC, the DI container is already involved in the controller's construction. Now we just need to tell the container how to resolve the dependency.

To resolve the dependency, we need a concrete implementation of IProjectService. We'll add a DemoService class, and for simplicity, it will use static dummy data.

```
public class DemoService : IProjectService
{
    public string Name { get; } = "Demo";

    public Organization GetOrganization() => new Organization
    {
        Name = this.Name,
        AvatarUrl = $"http://placehold.it/100&text={this.Name}",
        Projects = GetProjects()
    };

private IQueryable<Project> GetProjects() => new List<Project> {
         new Project {
             Id = 0,
             Description = "Test project 0",
             Name = "Test 0",
             Stars = 120
         },
         //...
         new Project {
             Id = 4,
             Description = "Test project 4",
             Name = "Test 4",
             Stars = 89
         }
    }.AsQueryable();
}
```

Finally, we'll instruct the DI container to instantiate a new DemoService whenever IProjectService is required. To configure the container, we'll modify the ConfigureServices method in Startup.cs. We'll add our configuration to the end of this method.

```
// This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        //... other services
        // Add MVC services to the services container.
        services.AddMvc();

        //our services
    }
```

We add the service by using the AddTransient extension method on the services collection, and setting the IProjectService as the type of Service and the DemoService as the implementation.

```
  public void ConfigureServices(IServiceCollection services)
    {
        //... other services
        // Add MVC services to the services container.
        services.AddMvc();

        //our services
        services.AddTransient<IProjectService, DemoService>();
    }
```

With the service added, DemoService will now be instantiated when we create the controller, and the exception will no longer be thrown.

GitHub Repos

Demo

Demo

Test 0 (120 Stars)
Test project 0

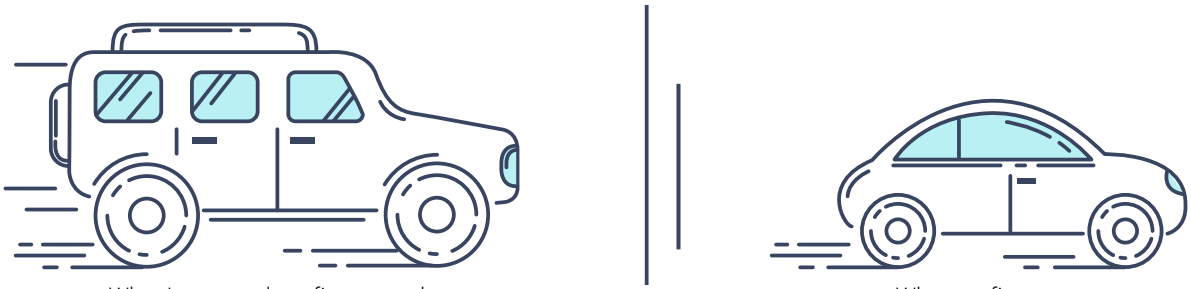
Test 1 (0 Stars)
Test project 1

Test 2 (5 Stars)
Test project 2

Test 3 (15 Stars)
Test project 3

Test 4 (89 Stars)
Test project 4

Weaknesses



What I expected my first car to be...

What my first car was...

Having a baked in DI layer throughout the ASP.NET stack is helpful, but you should reserve your expectations. While it is useful for simple scenarios, it is very limited. The default container only supports constructor injection, and it can only resolve types with one public constructor. There's no need to worry though; you have complete control over the DI container, and it's easy to replace with your favorite DI solution.

Replacing Default ASP.NET Core MVC DI Container

Setting up a replacement DI container does not require a lot of code, but the process could be more discoverable. We'll continue with our previous example to show exactly where the extensibility point is.

The method we are using, `ConfigureServices`, is actually one of two delegates the application will use to register the DI container and its services. The first and default delegate is an `Action<IServiceCollection>`, which was used in the previous example to resolve our `DemoService`. The second is a `Func<IServiceCollection, IServiceProvider>`, which is used to replace the default container by returning a custom implementation `IServiceProvider`.

To change from the default and enable use of an alternative `IServiceProvider`, we'll need to change the method signature. We can test this easily by modifying our example; we'll change the method from returning void to `IServiceProvider`, then at the very end of the method return services. `BuildServiceProvider()`. At this point the method does exactly the same thing as before, and we're still using the default DI container.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //... other services
        // Add MVC services to the services container.
        services.AddMvc();


        //our services
        services.AddTransient<IProjectService, DemoService>();
        return services.BuildServiceProvider();
    }
```

The application will still build and run just as it did before, except now we've exposed the extensibility point we need.

Next, we need to add a third-party DI container. There are a variety of DI contaiers to choose from, all with their own strengths and weaknesses. For this example, we'll be using a DNX compatible alpha version of Autofac.

To install Autofac, we'll open our package.json file and add the Autofac binaries.

```
"dependencies": {
    ...
   "Autofac": "4.5.0",
   "Autofac.Extensions.DependencyInjection": "4.1.0"
}
```

In the Startup.cs, we'll modify the ConfigureServices method again. This time, we'll add our dependencies to Autofac and resolve the container back to an IServiceProvider, replacing the default container.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //... other services
        // Add MVC services to the services container.
        services.AddMvc();

        //Autofac config
        var builder = new ContainerBuilder();

        builder.RegisterType<DemoService>()
            .As<IProjectService>().InstancePerLifetimeScope();

        //Populate the container with services that were previously registered
        builder.Populate(services);

        var container = builder.Build();

        return container.Resolve<IServiceProvider>();
}
```

Now our application is using the Autofac container in place of the default container. We can fully take advantage of Autofac's features, and because ASP.NET was developed against the IServiceProvider, we wont need to change any other code in our project.

Let's make one final change to the example by swapping our DemoService to another implementation using GitHub's REST API to populate our project list. The GitHubService implements the same IProjectService as DemoService; however, it takes a name parameter in its constructor. Using Autofac, we can set the Name value of the constructor in the configuration.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //...
        builder.Register(svc => new GitHubService("Telerik"))
            .As<IProjectService>().InstancePerLifetimeScope();

        //builder.RegisterType<DemoService>()
        //   .As<IProjectService>().InstancePerLifetimeScope();
    }
```

Restarting the application reveals our new GitHubService was successfully resolved without touching our controller code.

GitHub Repos

Telerik

razor-converter (143 Stars)
Tool for converting WebForms Views to Razor (C# Only)

less.js (4 Stars)
Leaner CSS

kendo-examples-asp-net (59 Stars)

kendo-examples-asp-net-mvc (81 Stars)
Kendo UI Examples for ASP.NET MVC

jquery-ui-vs-kendo-ui (8 Stars)
A technical comparison of jQuery UI and Kendo UI.

Wrap up & Resources

The new ASP.NET provides out-of-the-box DI throughout the stack. The new DI container is basic enough to get the job done, but lacks robust configurations. The container is easy enough to replace if you know where to look, and enables you to use feature-rich third-party tools. You can find the source code for **both the framework and examples for this project** at GitHub.

Microsoft.Framework.DependencyInjection
https://github.com/aspnet/DependencyInjection

Article Demo
https://github.com/EdCharbeneau/DependencyInjectionASP.NET_Core_MVCDemo

STAY SHARP WITH RAZOR TAGHELPERS

TagHelpers syntax looks like HTML: it uses elements and attributes but is processed by Razor on the server. TagHelpers are an alternative syntax to HTML Helper methods, and they improve the developer experience by providing a rich and seamless API. This can't be achieved with Razor alone. By seeing HTMLHelpers used alongside the new TagHelpers, we can easily identify the benefits they provide.

Let's use a simple button as an example. Here we have an HTML link that uses the bootstrap button style btn btn-primary and an href of /index. The button also includes the text Go Home.

```
<a href="/index" class="btn btn-primary">Go Home</a>
```

We could create this same markup using the ActionLink HTMLHelper to render the button. However, the developer experience starts to suffer once we begin adding the button style to the element.

```
<!-- just a link, no style -->
@Html.ActionLink("Go Home", "Index")

<!-- desired style -->
@Html.ActionLink("Go Home", "Index", "Home", null, new { @class = "btn btn-primary" })
```

When we add a CSS class to an ActionLink helper, the method signature must be satisfied completely. In addition to the long method signature, the syntax for adding a CSS class requires additional declaration, including escape characters. All of this leads to overly complex code intermixed with HTML markup.

The TagHelper Way

ASP.NET Core MVC implemented TagHelpers to help remediate the mess that HTMLHelpers made. TagHelpers continue to use Razor to generate HTML, but they do so in a less obtrusive way. Because TagHelpers rely on **elements** and **attributes** instead of escaping into server-side code, they can behave more like HTML.

Let's revisit the previous example and see how a TagHelper compares to the HTML and HTMLHelper syntax. This time, we'll start with an anchor tag <a and use the asp-action attribute to initialize the tag helper. When Razor sees the asp-action attribute, it recognizes the attribute's value is C# code.

When the parser recognizes TagHelper code, the code changes to bold with a purple highlight. (Note: colors may vary due to system preferences.)

```
<!-- HTMLHelper -->
@Html.ActionLink("Go Home", "Index", "Home", null, new { @class = "btn btn-primary" })

<!-- TagHelper -->
<a asp-action="Index" class="btn btn-primary">Go Home</a>

<!-- Rendered HTML -->
<a href="/index" class="btn btn-primary">Go Home</a>
```

Simplified Forms

TagHelpers also excel is in HTML form creation. Previously, MVC relied on complicated HTMLHelpers to accomplish this task. HTMLHelpers are not ideal for scenarios in which the helper must contain child elements. To render content within an HTMLHelper, the helper must encapsulate the content within a using statement. The using statement leads to a poor developer experience by interrupting the normal flow of HTML with pure C# syntax.

In the following example, the same password reset form is shown using both TagHelpers and HTMLHelpers. In the code below, notice how TagHelpers provide a much clearer picture of what the rendered HTML will be like.

```
<!-- Form using TagHelpers -->
<form asp-controller="Manage" asp-action="ChangePassword" method="post" class="form-
horizontal" role="form">
    <h4>Change Password Form</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="OldPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="OldPassword" class="form-control" />
            <span asp-validation-for="OldPassword" class="text-danger"></span>
        </div>
    </div>
    <>...</>
    <>...</>
    <>...</>
</form>

<!-- Form using HTMLHelpers -->
@using (Html.BeginForm("ChangePassword", "Manage", FormMethod.Post, new { @class = "form-
horizontal", role = "form" }))
{
    <h4>Change Password Form</h4>
    <hr />
        @Html.ValidationSummary(false,"", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.OldPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.OldPassword, new { @class = "form-control" })
            @Html.ValueFor(m => m.OldPassword)
        </div>
    </div>
  <>...</>
  <>...</>
  <>...</>
}
```

Your Choice

TagHelpers are a new addition but not the only choice. TagHelpers improve the developer experience by making client- and server-side HTML creation virtually seamless. Although TagHelpers are great for the examples shown here, HTMLHelpers excel at scenarios in which you use Fluent API chaining to construct complex UIs such as Charts, Grids and Spreadsheets. Consider using a "right tool for the job" approach when deciding which type of helper to use to build the view.

A NEW DIRECTION

ASP.NET Core MVC takes familiar concepts in new directions. While ASP.NET Core MVC still follows its theoretical roots, it is a completely different framework built for the future of web development.

Starting with the new project template, we examined how many of the basic elements have moved, changed or improved. Routing received updates that enable better developer control by mixing convention- and attribute-based routing. Vast improvements in handling configuration and DI can have direct impact on an application. Finally, new TagHelpers enable a better developer experience by making client- and server-side HTML creation virtually seamless. Understanding these essential changes to MVC will help you excel on your next project.

About the Author

Ed Charbeneau is a web enthusiast, speaker, writer, design admirer and Developer Advocate for Telerik. He has designed and developed web-based applications for business, manufacturing, systems integration as well as customer facing websites. Ed enjoys geeking out to cool new tech, brainstorming about future technology and admiring great design. You can find out more at http://edcharbeneau.com.