ASP.NET 5 CORE MVC

REVIEW AND DEPENDENCY INJECTION WITH IOC

OVERVIEW

- ASP.NET Core MVC is based on the Open Web Interface for .NET or OWIN. OWIN provides the underlying pipeline infrastructure (application example in later class).
- MVC itself consists of a collection of middleware or modules that can be added to the pipeline to compose an application.
- In addition to supporting open-source tooling, ASP.NET Core MVC adopts other web practices such as using JSON-based configuration files. JSON parses faster than XML, and is easier to read and edit than XML.

WHAT'S MISSING

- App_Start
- App_data
- Global.ASAX
- Web.Config not in root but has a new home
- Scripts has a new home
- Content has a new home

WHAT'S (MOSTLY) THE SAME

- Models
- Views
- ViewModels
- Controllers

WHAT'S NEW

- Src: new root folder for used to identify the source code of the project
- Wwwroot: used by the host to serve static resources and sub-folders include js (JavaScript), CSS, Images and lib
- Bower and NPM support can be configured by GUI
- Managing configuration by their respective .json files in the root src folder.
- Core MVC ships with Entity Framework 7 (EF7) that no longer supports EDMX database first modeling

WHAT'S NEW (CONTINUED)

- Services: Core MVC built with dependency injection (more later) at its core so services can be easily instantiated and used
- bower.json & package.json: Bower hosts Bootstrap, while NPM brings in dependencies such as Gulp with full intellisense support
- gulpfile.js: Gulp is a Node.js-based task runner with many plug-ins (MSBuild, NuGet, Nunit) and NPM with packages for compiling, minifying and bundling CSS
- hosting.ini: a pluggable server layer, which eliminates the hard dependency on IIS mainly used for configuring WebListener for hosting without IIS if desired

WHAT'S NEW (CONTINUED)

- project.json: describes the application and its .NET dependencies, and unlike prior versions of MVC it allows adding and removing .NET dependencies.
- appsettings.json: the primary replacement for the Web.Config where you'll specify application settings such as connection string, email endpoints and more.
- The new configuration model supports the JSON format which is used throughout ASP.NET Core MVC.
- startup.cs: previously handled in App_Start and Global.asax where 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.

THE STARTUP ROUTINE IN STARTUP.CS

- ConfigurationBuilder 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
- default configurations use AddJsonFile, which specifies that the configuration will be in JSON format
- Core MVC has configuration overloading. When you register a configuration in the ConfigurationBuilder, the last value applied wins over any previous value with same key

STARTUP.CS

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

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

//startup.cs

var builder = new ConfigurationBuilder()
                .SetBasePath(appEnv.ApplicationBasePath)
                        .AddJsonFile("foo.json")
                        .AddJsonFile("bar.json");

//result
Configuration.Get("mykey"); // => Bar
```

GETTING VALUES FROM CONFIGURATION

- Values are available to application through dependency injection (more later).
- At end of Startup method, use ConfigurationBuilder to create single instance of IConfiguration when Build method called: Configuration = builder.Build();
- Once configuration built, values are injected into project in ConfigureServices method.
- Inside ConfigureServices method, add settings using services.Configure<T>.
- Optionally, fetch value from configuration by convention, when key matches property name, or by specifying key explicitly.

FETCH MYOPTIONS, A POCO WITH THE PROPERTY CONFIGMESSAGE

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

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

//Startup.cs

//convention
services.Configure<MyOptions>(Configuration.GetConfigurationSection("MyOptions"));

//explicit
services.Configure<MyOptions>(Configuration("MyOptions:ConfigMessage"));
```

CONTROLLERS

- Controllers return views, but can serve as Web API endpoints now that Web API and MVC have merged with same routes and same Controller base class.
- Make values available to controller by adding options to any controller's constructor
- Since dependency injection already made available in Core MVC, we'll just need to reference the value
- Using IOptions<T> as argument in the constructor resolves the dependency.
- Works because the service provider resolved when we create a controller instance.

CONTROLLERS (CONTINUED)

```
//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();
}
```

TOGGLING CONFIG FILES

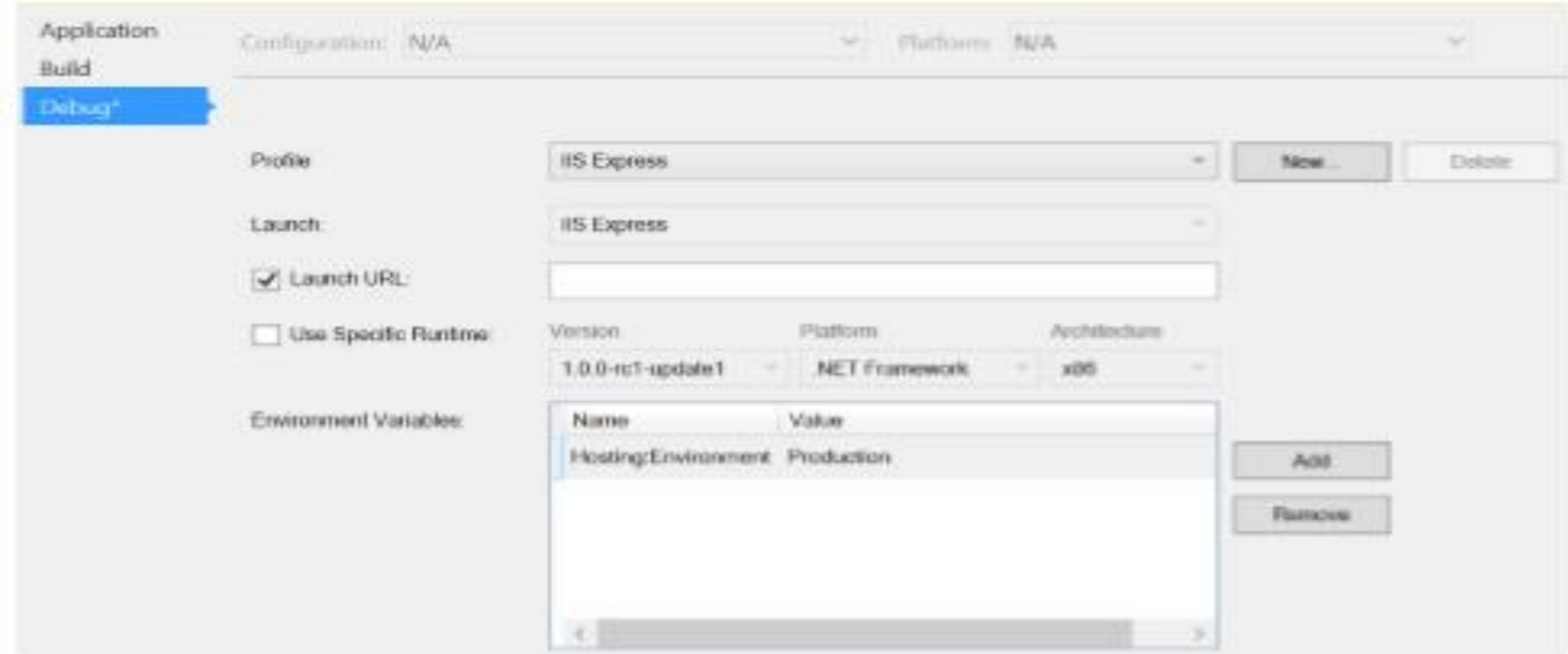
- Configuration builder has two configs, appsettings. json and config.{env.EnvironmentName}.json. Use the second to override settings when specific environment variables are available

```
//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"
   }
}
```

TOGGLING CONFIG FILES (CONTINUED)

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

TOGGLING CONFIG FILES (CONTINUED)



DEFINING ROUTES

- The ASP.NET Routing module responsible for mapping incoming browser requests to particular MVC controller actions.
- Routes in 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
- Attribute-Based Routing: uses attributes to define routes and gives finer control over URIs in web application.

CONVENTION-BASED ROUTES USE ACTION<IROUTEBUILDER> PARAMETER

```
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}");
            });

        }
```

CONVENTION BASED ROUTING (CONTINUED)

- Convention-based routes look and work much like 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?}.

CONVENTION BASED ROUTING (CONTINUED)

```
// 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 }
     );
```

ATTRIBUTE-BASED ROUTING

- Convention-based routes updated in this version, but are completely optional.
- In Core MVC, attribute-based routing enabled from the start
- The code in next slide shows using attribute-based routing to specify route for /home/index.
- Notice combining controller-level attribute with action-level attribute to complete full route.

ATTRIBUTE-BASED ROUTING (CONTINUED)

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

ATTRIBUTE-BASED ROUTING (CONTINUED)

- You can combine multiple route attributes. For example, you can add a secondary route of [Route("/")] to enable 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();
    }
}
```

ATTRIBUTE-BASED ROUTING (CONTINUED)

- Routes have new [controller] and [action] tokens that reference controller and action names in route template. These tokens enable naming routes after their respective controller or action, while maintaining flexibility. If renaming controller or action, route changes without requiring an update to 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();
    }
}
```

DEPENDENCY INJECTION IN ASP.NET CORE MVC

- Dependency injection (DI) is technique for achieving loose coupling between objects and their collaborators, or dependencies.
- Follows the Dependency Inversion Principle, which states that *"high level modules should not depend on low level modules; both should depend on abstractions" (like interfaces).*
- Rather than directly instantiating collaborators, or using static references, objects a class needs in order to perform its actions are provided to class in some fashion.
- Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle.
- This approach is known as "constructor injection"

DI AND THE STRATEGY PATTERN

- Instead of referencing specific implementations, classes request abstractions (typically interface or abstract classes) which are provided to them when they are constructed.
- Extracting dependencies into interfaces and providing implementations of these interfaces as parameters is example of Strategy design pattern.
- The Strategy Design Pattern allows object to have some or all of its behavior defined in terms of another object which follows a particular interface.
- A particular instance of this interface is provided to client when it is instantiated or invoked, providing concrete behavior to be used.

DI AND IOC

- DI, with many classes requesting their dependencies via their constructor (or properties), it's helpful to have a class dedicated to creating these classes with their associated dependencies.
- These classes are referred to as *containers*, or more specifically, Inversion of Control (IoC) containers or Dependency Injection (DI) containers.
- A container is essentially a factory that is responsible for providing instances of types that are requested from it. If a given type has declared that it has dependencies, and the container has been configured to provide the dependency types, it will create the dependencies as part of creating the requested instance.
- In this way, complex dependency graphs can be provided to classes without need for any hard-coded object construction. In addition to creating objects with their dependencies, containers typically manage object lifetimes within application.

INVERSION OF CONTROL (IOC)

- ASP.NET 5 includes a simple built-in container (represented by the IServiceProvider interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI. ASP.NET's container refers to the types it manages as *services*.
- You configure the built-in container's services in the ConfigureServices method in your application's Startup class.

DI CONFIGURATION

- The ConfigureServices method in the Startup class is responsible for defining the services the application will use, including platform features like Entity Framework and ASP.NET MVC.
- Initially, the IServiceCollection provided to ConfigureServices has just a handful of services defined.
- The default web template shows an example of how to add additional services to the container using a number of extensions methods like AddEntityFramework, AddIdentity, and AddMvc.

DI CONFIGURATION (CONTINUED)

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddEntityFramework()
            .AddSqlServer()
            .AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]
));

    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

DI CONFIGURATION (CONTINUED)

- The features and middleware provided by ASP.NET, such as MVC, follow a convention of using a single AddService() extension method to register all of the services required by that feature.
- Of course, in addition to configuring the application to take advantage of various framework features, you can also use ConfigureServices to configure your own application services.

DI CONTAINER

- 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 IServiceProvider interface and are available throughout stack. Because the IServiceProvider is same across all components of 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.

SUMMARY

- Discussed new features of an ASP.NET Core MVC project in VS 2015
- Configuration and Routing with Core MVC
- Dependency Injection
- IoC