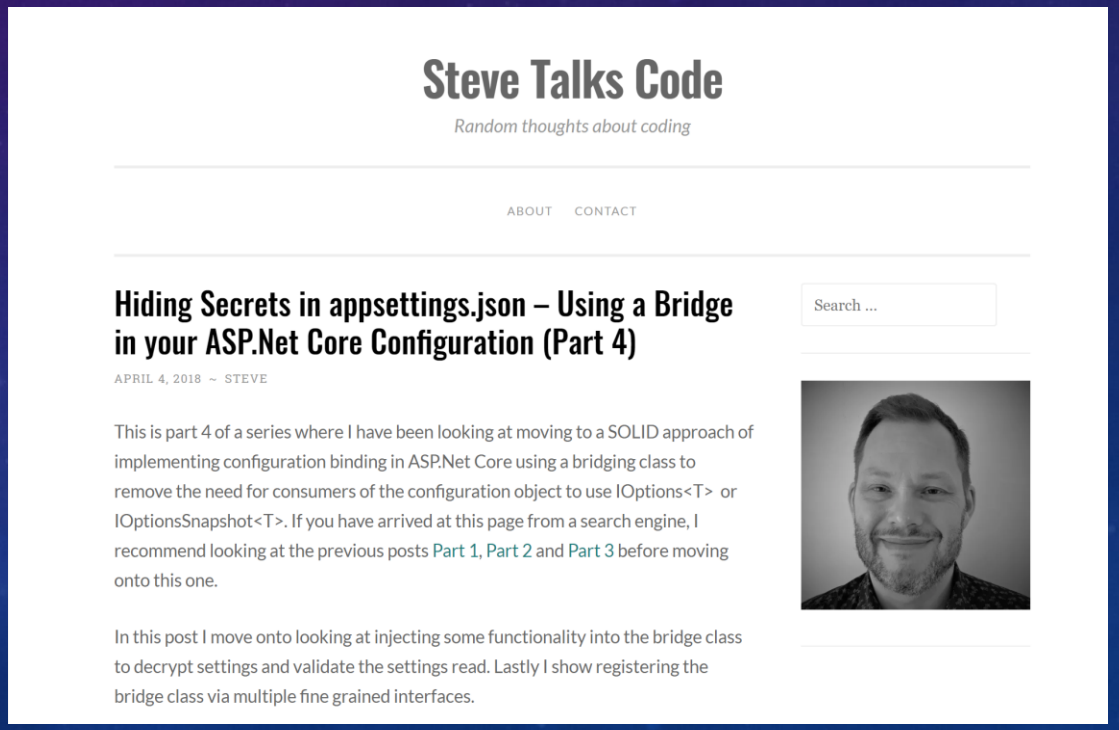
.NET Configuration is Easy … Right?

A SOLID Approach to Configuration in ASP.NET Core

Steve Collins

WHO AM I?

- Contract Microsoft stack developer/architect for last 25+ years
- Blog at https://stevetalkscode.co.uk
- @stevetalkscode



EXPECTATIONS WARNING

- Not an expert on ASP.NET Core and configuration, but …
- Want to introduce the complexities of configuration to new developers and give some food for thought to existing developers
- Show some best practices when using configuration
- Show using the practices can make your code more readable, maintainable and testable
- Assume some familiarity with the SOLID principles

AGENDA

- Why do we use configuration? — Beginner
- A very brief history of .NET configuration from .NET Framework 1.0 to .NET Core — Beginner
- .NET Core configuration sources and security considerations — Beginner / Intermediate
- How ConfigurationBuilder composes configuration from multiple sources — Beginner / Intermediate
- How CreateDefaultWebhost presets multiple configuration sources for you — Beginner / Intermediate
- Making your configuration testable — Beginner / Intermediate
- Direct IConfiguration binding vs .Configure<T> & the Options pattern — Intermediate / Expert
- Unwrapping IOptions*<T> variations using lambda expressions — Intermediate / Expert
- Using the Bridge pattern to unwrap IOptions*<T> — Intermediate / Expert
- Validating, Transforming and Encrypting values — Intermediate / Expert

WHY DO WE USE CONFIGURATION?

- For values that may change in the future
- For values that differ between deployment environments
- For sensitive values that should not be in source code
- For values that may disable/enable functionality at startup or while running

HISTORICAL PROBLEMS WITH CONFIGURING ASP.NET

- Configuration drift between servers when not controlled through deployment automation
- Usually only read when web application starts, so changes require an application restart
- Can become large and unwieldy to manage in a single file
- System may fail to start if errors in configuration such as invalid characters
- Accidental leaking of secrets into source control

The History of .NET Configuration

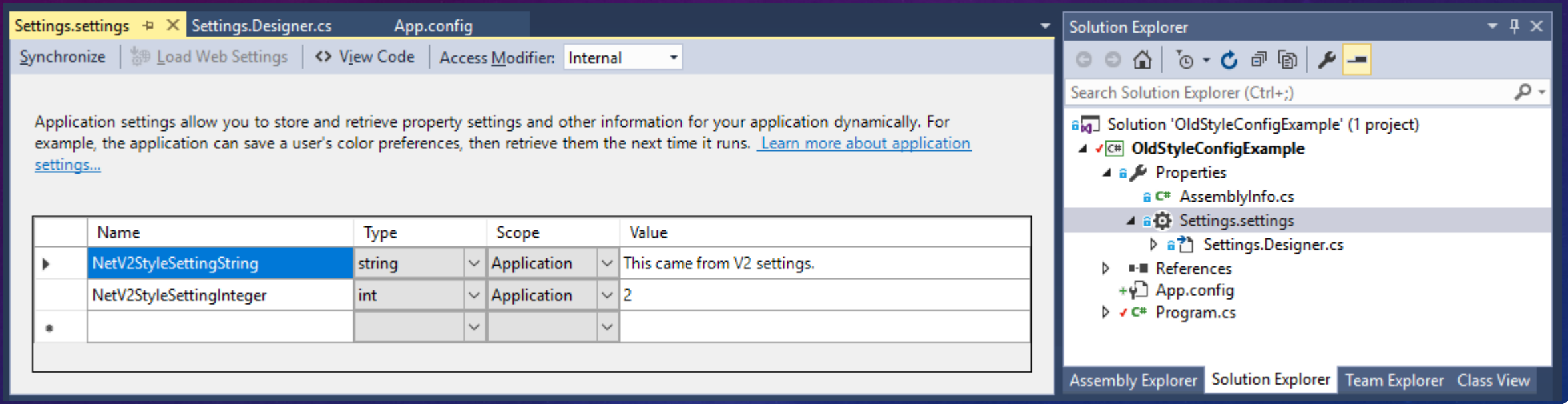
WEB.CONFIG FILES

- XML Web.Config files introduced with .NET 1.0 to replace INI files
- Use an XML schema to define expected elements and attributes
- Limited to string key/value pairs in appSettings section (unless writing own section handler) so other types needed manually casting in code
- Accessed through the static ConfigurationManager class' AppSettings property
- Transformation files used to apply XML transformations for different build configurations

```
var netV1StyleSettingString = ConfigurationManager.AppSettings["NetV1StyleSettingString"];
var netV1StyleSettingInt = int.Parse(ConfigurationManager.AppSettings["NetV1StyleSettingInteger"]);
```

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="NetV1StyleSettingString" value="This came from old style V1 appSettings." />
    <add key="NetV1StyleSettingInteger" value="1" />
  </appSettings>
</configuration>
```

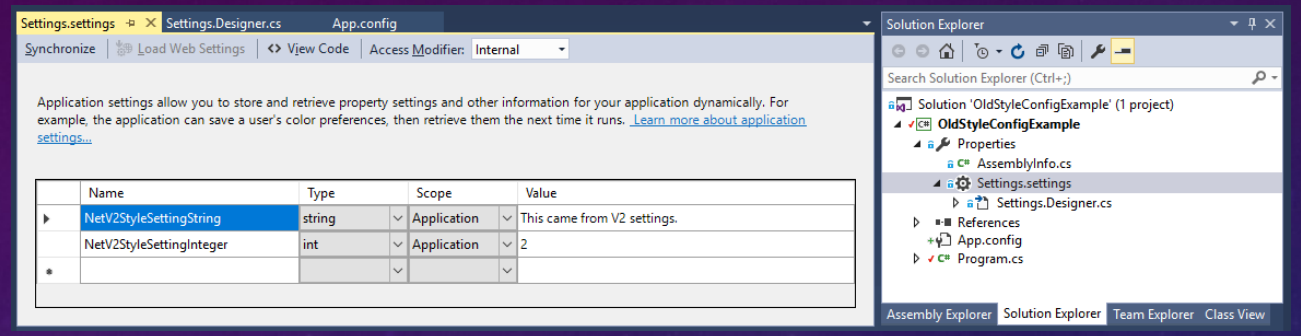
WEB.CONFIG FILES



- Visual Studio 2005 / .NET 2.0 introduced the Settings.Settings file and a designer

Microsoft .NET

WEB.CONFIG FILES



- Auto-generated code looks after the casting from string to required data type
  - Requires additional code for custom types to be serialised and attributing of properties
- Visual Studio tries to keep consistency between the designer and the configuration file
  - Gets messy when the configuration file manually changed or is transformed on build
  - Being auto-generated code, any changes made to the class are lost if regenerated
  - The auto-generated class is declared as sealed so cannot inherit
  - Whilst the properties are not static, a singleton instance (Default) is provided as a static property
- Adds a SectionGroup handler to the configuration file
  - Requires additional code for custom types to be serialised and attributing of properties

Microsoft® .NET

SO WHAT IS BAD ABOUT WEB.CONFIG FILES?

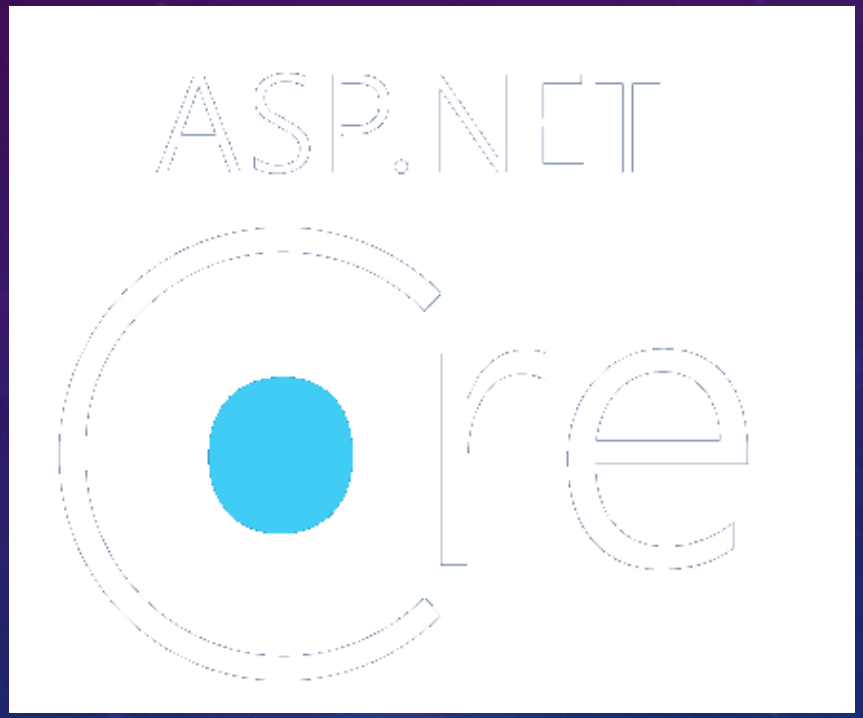
- Uses XML with rigid schema – very easy to get things wrong by putting elements in wrong place
- Not very extensible without a lot of pain …
  - Requires detailed knowledge of Configuration Sections, Groups, Handlers,
  - Custom classes must be deserialisable via the XMLSerializer
  - Hard to debug
- Requires transformation files for different environments which are either defined by the build configuration or requires deployment tool to be able to run transformation

SO WHAT IS BAD ABOUT WEB.CONFIG FILES?

- Access to configuration through the ConfigurationManager static class has given developers a gun to shoot themselves in the foot with it being littered across whole code bases
- Very hard to unit test when code accesses the static class directly
- Requires lots of code to go beyond the simple out-of-the-box handling
- Can become very fragile and hard to spot and catch errors

@stevetalkscode

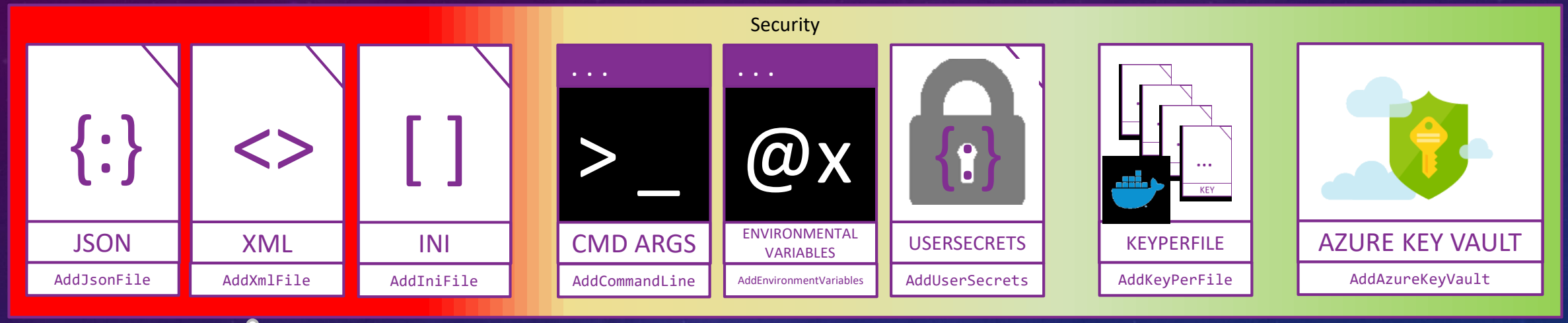
The Future of .NET Configuration


ASP.NET
core

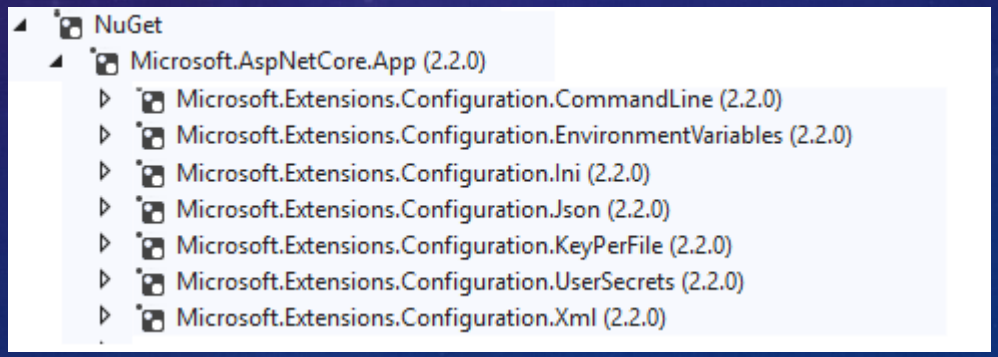
IN .NET CORE, WEB.CONFIG IS DEAD …

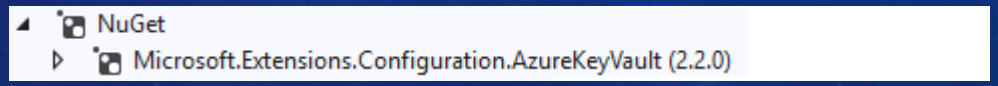
- The Web.Config file and ConfigurationManager class are not in .NET Core
- In .NET Core, all configuration is accessed through the IConfiguration interface
- An instance of a class implementation is created using the ConfigurationBuilder class using configuration sources
- Not limited to a single configuration source – composed from multiple sources in one or more formats
- Not limited to files
- Depending on the source, does not require an application restart to use changed configuration values
- Out of the box ability to create class instances with properties bound to configuration values

COMMON CONFIGURATION SOURCES



NOT WEB.CONFIG!



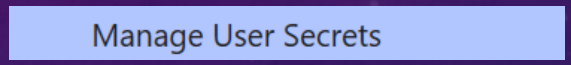


INSECURE CONFIGURATION

- Configuration files are inherently insecure
- Saved ‘in the clear’ as text
- Usually saved in source control with the application source
- Usually deployed to the same directory as the application

KEEPING SECRETS OUTSIDE SOURCE CONTROL

- Make use of user secrets while developing




```
<PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
   <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

- Associated with a GUID in the project file
- Save location depends on platform
  - Windows - `%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json`
  - Linux / Mac - `~/.microsoft/usersecrets/<user_secrets_id>/secrets.json`
- Saved ‘in the clear’ as text, so still visible with anyone with access to the file location
- Edit JSON file manually in text editor, in Visual Studio via the Manage Secrets menu option on the project node
- Can set and list values via the `dotnet user-secrets` command
- More info at https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets

SECURE KEY VAULT PROVIDERS

Azure Key Vault

OFFICAL
NUGET PACKAGE

OFFICAL
NUGET PACKAGE

UNOFFICIAL
NUGET PACKAGES

SECURE KEY VAULT PROVIDERS

- Azure Key Vault – requires an Azure subscription and has costs per 10,000 transactions
  - Options for software or hardware protected keys
  - Great if you are already hosting in Azure
  - Latency if not hosting in Azure
  - May not be able to use Azure for corporate policy reasons
  - Requires the application to be registered in Azure AD for access to the key vault and assigned as API
  - Requires **Microsoft.Extensions.Configuration.AzureKeyVault** NuGet package
  - AddAzureKeyVault extension method which requires
    - a URL to the vault location (over HTTPS)
    - either a certificate or a clientId/client secret … but where to you store these securely?

SECURE KEY VAULT PROVIDERS

- AWS Key Management Service (KMS) – requires an AWS subscription and has costs per 10,000 transactions
  - Options for software or hardware protected keys
  - Great if you are already hosting in AWS
  - Latency if not hosting in AWS
  - May not be able to use AWS for corporate policy reasons
  - Requires an application prefix
  - Requires **Amazon.Extensions.Configuration.SystemsManager** NuGet package
  - AddSystemsManager extension method which requires
    - Application prefix
    - How often to reload the settings based on a TimeSpan
    - Whether the configuration is optional

CONFIGURATION SOURCES AND PROVIDERS

- Each configuration source has two elements

- An implementation of IConfigurationSource – this reads the configuration data from the source

  - Has single method Build(IConfigurationBuilder buider)
  - Returns an instance of IConfigurationProvider

- An implementation of IConfigurationProvider – this converts the source data into Key/Value pairs
  - The keys are composed of a logical hierarchy path to a single value
  - E.g. Section1:SubSection1:KeyName
  - The value is a string
  - Also supports a ReloadToken for the configuration to be notified of changes in the source data

IConfigurationSource

WorkingDaySettings:DefaultValues:StartTime	09:00
WorkingDaySettings:DefaultValues:LunchStart	12:30
WorkingDaySettings:DefaultValues:LunchEnd	13:30
WorkingDaySettings:DefaultValues:EndTime	17:00

IConfigurationProvider

COMPOSING CONFIGURATION

- Layered approach to configuration
- Each additional layer can override and/or append to the previous layers
- Layers can be of the same configuration source type or different source type
  - E.g. By default, appSettings.json has an override appSettings.{environmentname}.json
  - If in Development envrionment, a local usersettings.json file is then layered over this for sensitive data

```
{
    "WorkingDaySettings":
    {
        "DefaultValues" :
        {
            "StartTime": "09:30"
        }
    }
}
```

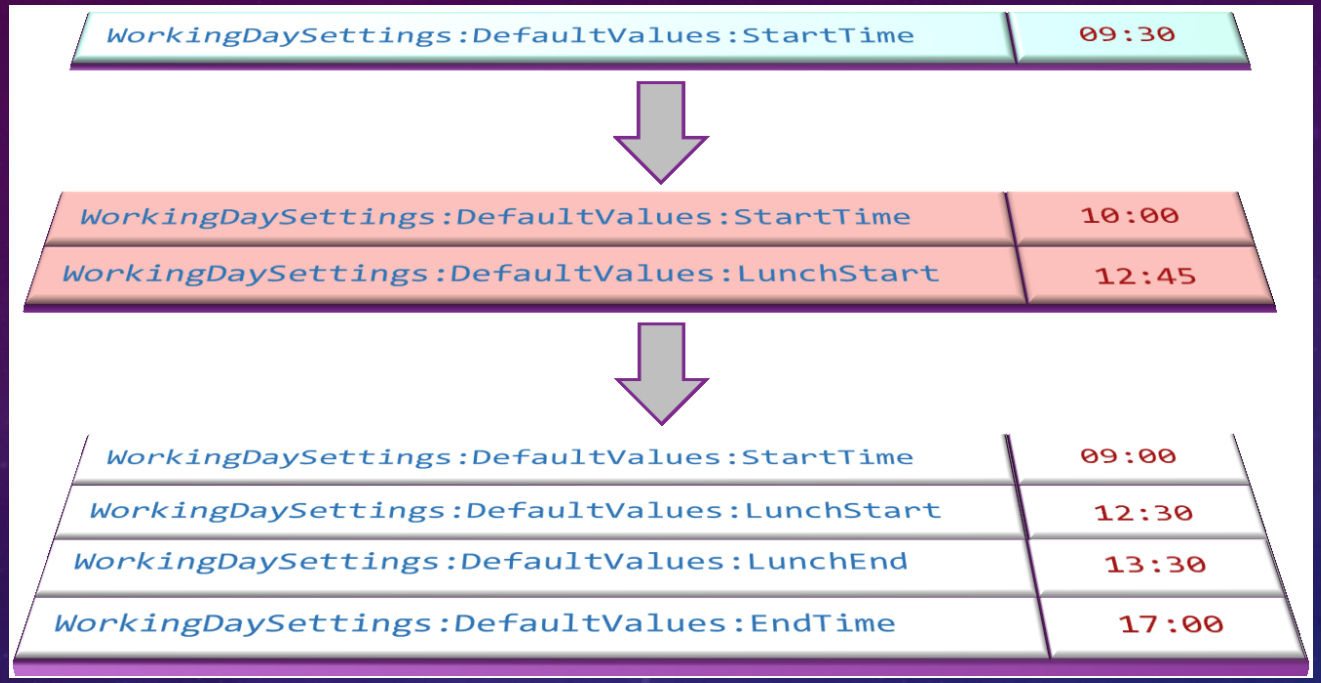
usersecrets.json

```
{
    "WorkingDaySettings":
    {
        "DefaultValues" :
        {
            "StartTime": “10:00",
            "LunchStart": "12:45"
        }
    }
}
```

appsettings.development.json

```
{
    "WorkingDaySettings":
    {
        "DefaultValues" :
        {
            "StartTime": "09:00",
            "LunchStart": "12:30",
            "LunchEnd": "13:30“,
            "EndTime": "17:00",
        }
    }
}
```

appsettings.json


WorkingDaySettings:DefaultValues:StartTime
09:30
WorkingDaySettings:DefaultValues:StartTime
10:00
WorkingDaySettings:DefaultValues:LunchStart
12:45
WorkingDaySettings:DefaultValues:StartTime
09:00
WorkingDaySettings:DefaultValues:LunchStart
12:30
WorkingDaySettings:DefaultValues:LunchEnd
13:30
WorkingDaySettings:DefaultValues:EndTime
17:00


WorkingDaySettings:DefaultValues:StartTime
09:30
WorkingDaySettings:DefaultValues:LunchStart
12:45
WorkingDaySettings:DefaultValues:LunchEnd
13:30
WorkingDaySettings:DefaultValues:EndTime
17:00

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
            .AddEnvironmentVariables();
        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; }

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

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
    {
        loggerFactory.AddConsole(Configuration.GetSection("Logging"));
        loggerFactory.AddDebug();

        app.UseMvc();
    }
}
```

```
public class Startup
{
    public Startup(IConfiguration config)
    {
        Configuration = config;
    }

    public IConfigurationRoot Configuration { get; }

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

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
    {
        loggerFactory.AddConsole(Configuration.GetSection("Logging"));
        loggerFactory.AddDebug();

        app.UseMvc();
    }
}
```

DON’T REPEAT WHAT’S ALREADY THERE

- If using WebHost.CreateDefaultBuilder, the common configuration sources are already registered for you in the following order
  - appsettings.json
  - appsettings.<environment>.json
  - User secrets (if Development)
  - Environment variables
  - Command line arguments

- If you subsequently add more configuration sources, remember to re-add User secrets again at the end if using them to override values in the added sources

CUSTOMISING CONFIGURATION FOR TESTING

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(CreateWebHostBuilder(args)).Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .ConfigureAppConfiguration(AddCustomConfiguration);


    public static IWebHost BuildWebHost(IWebHostBuilder webHostBuilder) => webHostBuilder.Build();

    private static void AddCustomConfiguration(WebHostBuilderContext context, IConfigurationBuilder builder)
    {
        builder.AddJsonFile(path: "someothersettings.json", optional: true, reloadOnChange: true);
    }
}
```

```
AddCommandLine                IConfigurationBuilder
AddConfiguration              IConfigurationBuilder
AddEnvironmentVariables       IConfigurationBuilder
AddInMemoryCollection         IConfigurationBuilder
AddIniFile                    IConfigurationBuilder
AddJsonFile                   IConfigurationBuilder
AddKeyPerFile                 IConfigurationBuilder
AddUserSecrets                IConfigurationBuilder
AddUserSecrets<>              IConfigurationBuilder
AddXmlFile                    IConfigurationBuilder
```

CUSTOMISING CONFIGURATION FOR TESTING

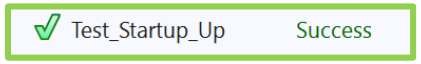
```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(CreateWebHostBuilder(args)).Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .ConfigureAppConfiguration(AddCustomConfiguration);


    public static IWebHost BuildWebHost(IWebHostBuilder webHostBuilder) => webHostBuilder.Build();

    private static void AddCustomConfiguration(WebHostBuilderContext context, IConfigurationBuilder builder)
    {
        builder.AddJsonFile(path: "someothersettings.json", optional: true, reloadOnChange: true);
    }
}
```

```
[Fact]
public void Test_Startup_Up()
{
    var message = "Goodbye from args";
    var args = new string[] { $"MySettings:GoodbyeMessage={message}" };
    var testConfig = new [] {new KeyValuePair<string, string>("MySettings:HelloMessage", "Hello World from Unit Test")};
    var defaultBuilder = Program.CreateWebHostBuilder(args);
    defaultBuilder.ConfigureAppConfiguration((context, builder) => { builder.AddInMemoryCollection(testConfig); });
    var sut = (IConfiguration)Program.BuildWebHost(defaultBuilder).Services.GetService(typeof(IConfiguration));
    Assert.Equal(message, sut["MySettings:GoodbyeMessage"]);
    Assert.Equal(testConfig[0].Value, sut[testConfig[0].Key]);
}
```



USING YOUR CONFIGURATION

- Tempting to use IConfiguration in the same way that ConfigurationManager was used

IConfiguration.Item[String] Property

Namespace: Microsoft.Extensions.Configuration

Assembly: Microsoft.Extensions.Configuration.Abstractions.dll

Gets or sets a configuration value.

C# Copy

```
public string this[string key] { get; set; }
```

Parameters

key String

The configuration key.

Property Value

String

The configuration value.

USING YOUR CONFIGURATION

- Tempting to use IConfiguration in the same way that ConfigurationManager was used
- Please don't do this …
  - Have to know the path to the value E.g. Section1:SubSection1:SubSubSection1:Key
  - Still makes use of 'Magic Strings' littered throughout the code
  - Have to go hunting for usage if key changes (or the hierarchy to the key as part of the string)
  - Have to go hunting for usage if another related key is added
  - Liable to typos – if one of the references has a typo, how do you find it?
- Unless you are limiting its use to setting up Dependency Injections services
  - In other words, limit its use to just the Startup class …
  - Or encapsulate inside a class that is registered with the DI container … but there is an easier way …

USING YOUR CONFIGURATION WITH BINDING

- Wouldn't it be nice if we could encapsulate our settings in a class?

```
{
    "WorkingDaySettings":
    {
        "DefaultValues" :
        {
            "StartTime": "09:00",
            "LunchStart": "12:30",
            "LunchEnd": "13:30",
            "EndTime": "17:00",
        }
    }
}
```

```
public class WorkingDay
{
    public string StartTime { get; set; }
    public string LunchStart { get; set; }
    public string LunchEnd { get; set; }
    public string EndTime { get; set; }
}
```

```
Configuration.Bind();
```

(this IConfiguration configuration, **object instance**):void

(this IConfiguration configuration, **object instance**, Action<BinderOptions> configureOptions):void

(this IConfiguration configuration, **string key**, object instance):void

Attempts to bind the given object instance to the configuration section specified by the key by matching property names against configuration keys recursively.

key: The key of the configuration section to bind.

USING YOUR CONFIGURATION WITH BINDING

- Out of the box extension method `Configure<T>` to map a configuration section to a class instance
- Extension method maps to class using the Options pattern

  https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-3.0

```
3 references
public class Startup
{
    1 reference
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    3 references
    public IConfiguration Configuration { get; }

    1 reference
    public void ConfigureServices(IServiceCollection services)
    {

        services.AddMvc();
        services.AddOptions();
        services.Configure<MyAppSettings>(Configuration.GetSection("Settings"));
    }
```

Options pattern in ASP.NET Core

08/19/2019 • 36 minutes to read • +6

By Luke Latham

The options pattern uses classes to represent groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- The Interface Segregation Principle (ISP) or Encapsulation – Scenarios (classes) that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns – Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data. For more information, see the Options validation section.

… BUT THERE IS A BETTER WAY !

OPTIONS PATTERN IN .NET CORE

- The Options pattern registers three generic interfaces
  - IOptions<T> - creates a Singleton instance with properties bound to the configuration values at time of application start up through the .Value property
  - IOptionsMonitor<T> - creates a Singleton instance with properties bound to the current configuration values at time of calling via the .CurrentValue property
  - IOptionsSnapshot<T> - creates an instance with the properties bound to the current configuration values at time of being created in a Scoped lifetime calling via the . Value property
- Getting updated configuration values depends on whether the configuration source supports notifications via a ReloadToken
  - File based configuration sources usually based on file monitor with notifies a change when the file is changed
  - Others may be based on a polling mechanism that compares the current cached values to the values currently in the source

OPTIONS PATTERN IN .NET CORE – NOT IDEAL

```
public WeekdaysController(WorkingDay workingDayOptions)
{
    _workingDay = workingDayOptions;
}
```

OPTIONS PATTERN IN .NET CORE – NOT IDEAL

```
using Microsoft.Extensions.Options;
```

```
public WeekdaysController(IOptionsMonitor<WorkingDay> workingDayOptions)
{
    _workingDay = workingDayOptions.CurrentValue;
}
```

```
public WeekdaysController(IOptionsSnapshot<WorkingDay> workingDayOptions)
{
    _workingDay = workingDayOptions.Value;
}
```

```
public WeekdaysController(IOptions<WorkingDay> workingDayOptions)
{
    _workingDay = workingDayOptions.Value;
}
```

OPTIONS PATTERN IN .NET CORE – NOT IDEAL

- Not obvious when you use .Configure<T> that the pattern is being used
  - If you haven't read the documentation, you expect to be able to inject T into your classes
  - Instead, you have to inject one of IOptions<T>, IOptionsSnapshot<T> or IOptionsMonitor<T>
  - You then need to unwrap the instance of T using the Value or CurrentValue property (as appropriate)
- If you want to use T in a project outside of the main ASP.NET Core project you either need to
  - import the **Microsoft.Extensions.Options** NuGet package or
  - unwrap T using .Value or .CurrentValue property before passing to the library class that needs T
  - A bit dirty as introducing 'plumbing' code to domain or business logic projects which doesn't really belong there
- Have to mock to IOptions*<T> interfaces in unit tests for classes that accept these in the constructor

UNWRAPPING OPTIONS VIA DI REGISTRATION

- To make things a cleaner, can perform the value unwrapping in the Dependency Injection registration
- As its simplest, can register a lambda function with the DI container to unwrap an instance of T

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.Configure<WorkingDay>(Configuration.GetSection("WorkingDaySettings:DefaultValues"));
    services.AddScoped(sp => sp.GetRequiredService<IOptionsSnapshot<WorkingDay>>().Value);
}
```

- Gets more complicated if wanting to do something more exotic such as injecting into the constructor of the WorkingDay class … for example what if we want to encrypt some of the values?
- What if we want to change the type of one or more properties. E.g. change the strings to TimeSpan values
- What if we want to read configuration into a single class, but apply Interface Segregation Principle to expose as smaller finely grained interfaces
- What if we want to make the class that gets passed around immutable … we can't write back to the configuration sources so why make the class properties read/write?

TAKE ME TO THE BRIDGE

- We can improve things by using the Gang of Four Bridge pattern by replacing the lambda with a class instance that meets the requirements
  - Rename the WorkingDay class to WorkingDayReader
  - Create a new WorkingDay class with the same property names, but this time typed as TimeSpan and readonly by only having a { get; } accessor
  - Add a constructor to the new WorkingDay class that has a parameter of type IOptionsMonitor<WorkingDayReader>
  - In the constructor, unwrap the IOptionsMonitor<WorkingDayReader> using the .CurrentValue property
  - For each TimeSpan property in the new class, map from the string version in the unwrapped WorkingDayReader instance

TAKE ME TO THE BRIDGE

```
public class WorkingDay
{
    public string StartTime { get; set; }
    public string LunchStart { get; set; }
    public string LunchEnd { get; set; }
    public string EndTime { get; set; }
}
```

TAKE ME TO THE BRIDGE

```
public class WorkingDayReader
{
    public string StartTime { get; set; }
    public string LunchStart { get; set; }
    public string LunchEnd { get; set; }
    public string EndTime { get; set; }
}
```

```
public class WorkingDay
{
    public TimeSpan StartTime { get;}
    public TimeSpan LunchStart { get; }
    public TimeSpan LunchEnd { get; }
    public TimeSpan EndTime { get; }
}
```

TAKE ME TO THE BRIDGE

```
public class WorkingDayReader
{
    public string StartTime { get; set; }
    public string LunchStart { get; set; }
    public string LunchEnd { get; set; }
    public string EndTime { get; set; }
}
```

```
public interface IWorkingDay
{
    TimeSpan StartTime { get;}
    TimeSpan LunchStart { get; }
    TimeSpan LunchEnd { get; }
    TimeSpan EndTime { get; }
}
```

```
public class WorkingDay : IWorkingDay
{
 public WorkingDay(IOptionsMonitor<WorkingDayReader> workingDayReader)
    {
        var reader = workingDayReader.CurrentValue;
        StartTime = TimeSpanFromHoursAndMinutes(reader.StartTime);
        LunchStart = TimeSpanFromHoursAndMinutes(reader.LunchStart);
        LunchEnd = TimeSpanFromHoursAndMinutes(reader.LunchEnd);
        EndTime = TimeSpanFromHoursAndMinutes(reader.EndTime);
    }
    public TimeSpan StartTime { get;}
    public TimeSpan LunchStart { get; }
    public TimeSpan LunchEnd { get; }
    public TimeSpan EndTime { get;

    private static TimeSpan TimeSpanFromHoursAndMinutes(string hoursAndMinutes) =>
        TimeSpan.ParseExact(hoursAndMinutes, "hh\\:mm", CultureInfo.InvariantCulture);
}
```

TAKE ME TO THE BRIDGE

```
public class WorkingDayReader
{
    public string StartTime { get; set; }
    public string LunchStart { get; set; }
    public string LunchEnd { get; set; }
    public string EndTime { get; set; }
}
```

```
public interface IWorkingDay
{
    TimeSpan StartTime { get;}
    TimeSpan LunchStart { get; }
    TimeSpan LunchEnd { get; }
    TimeSpan EndTime { get; }
}
```

```
public class WorkingDay : IWorkingDay
{
    public TimeSpan StartTime { get;}
    public TimeSpan LunchStart { get; }
    public TimeSpan LunchEnd { get; }
    public TimeSpan EndTime { get; }
}
```

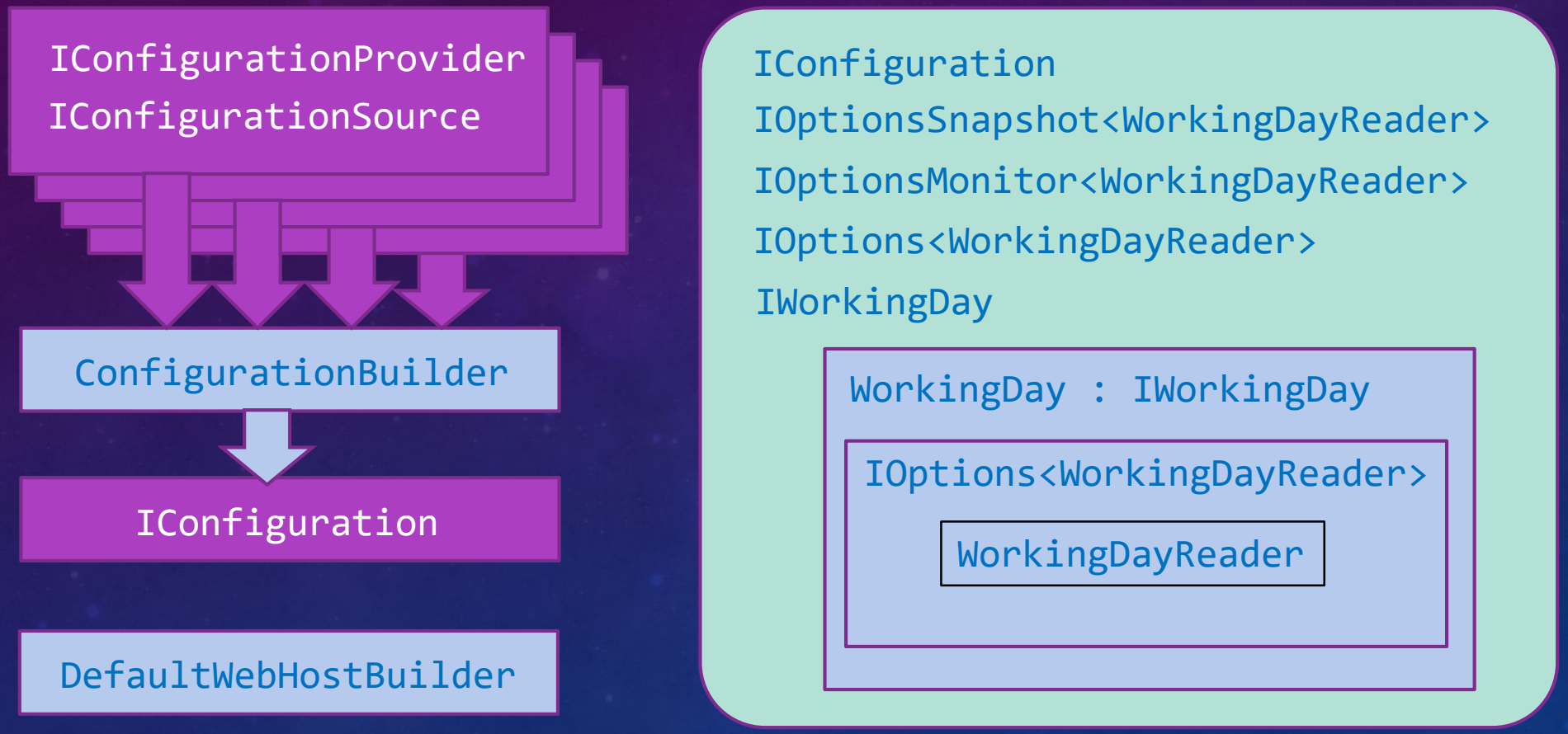
```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        services.Configure<WorkingDayReader>(Configuration.GetSection("WorkingDaySettings:DefaultValues"));
        services.AddTransient<IWorkingDay, WorkingDay>();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        // Contents excluded from presentation to save space ….
    }
}
```

HOW THE DI RESOLVES THE CONFIGURATION

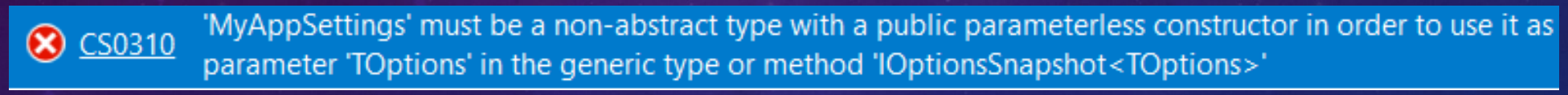


SO HOW DOES ALL THIS HELP?

- Means that the only part of code that needs to know about IConfiguration is the Startup class which converts configuration into class instances that are accessible through interfaces
- Means that the configuration is easy to mock for unit tests as all that is being mocked are the interfaces
- The configuration can be broken down into many finely grained interfaces that can all be registered with the DI container.
- If the interfaces are defined in a common library, the configuration can be shared between multiple projects without the need to reference the Microsoft.Extensions.Options NuGet package

SO HOW DOES ALL THIS HELP?

- Future developers do not need to worry about the Options pattern to consume configuration, especially having to decide which type of IOptions*<T> is required (inconsistent lifetimes)
- Gets around the problem of classes used by Configure<T> not being able to have custom constructors



- By decoupling the class that is used by Configure<T> from the class (interface) that is consumed, allows for customisation … changing data types, adding custom validation, and adding encryption

SENSITIVE CONFIGURATION VALUES

- Anything that is used for authentication – passwords, connection strings, API keys etc.
- Remember text based configuration files (appSettings.json etc.) are insecure
- Sensitive values should not be part of configuration files placed in source control
- Preferably use one of the key vault providers such as Azure or AWS
- If those are not available, consider using a configuration source that is outside of source control
  - Command Line arguments
  - Docker secrets
  - Environmental variables
  - Appsettings.json – deployed outside of the application folder

Still insecure if plain text

ENCRYPTING SENSITIVE CONFIGRATION VALUES

- Base64 Encoding IS NOT ENCRYPTION!
- Ideally use certificate based encryption as works cross-platform and not stored in source code, but this is not always possible
- Do not ‘roll your own’ encryption algorithm
- You may be considering making use of ASP.NET Core's own Data Protection APIs, but …

> The ASP.NET Core 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's nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data.

ENCRYPTING SENSITIVE CONFIGURATION VALUES

- Base64 Encoding IS NOT ENCRYPTION!
- Ideally use certificate based encryption as works cross-platform and not stored in source code, but this is not always possible
- Do not ‘roll your own’ encryption algorithm
- You may be considering making use of ASP.NET Core's own Data Protection APIs, but … not suitable
- Use a strong algorithm – consider AES256 with SHA2 hashed key & initialisation vector
- Do not store key/IV pair in source code – consider putting in environmental variable, command line args, docker secrets or as last resort, a json file outside the application AND NOT IN SOURCE CONTROL!
- Use different salt/hashes for different purposes if you have multiple encrypted values
- Once you have the encrypted byte array, THEN you Base64 encode to put into your configuration source THAT IS NOT IN SOURCE CONTROL!

DECRYPTING CONFIGURATION VALUES USING BRIDGE

- Create an interface that decrypts values
- Create a class implementation using the algorithm you have choses to use to decrypt values
- Register the interface and implementation with the DI container
- Make the interface a parameter to the bridge class along with the IOptions<T>
- When the IOptions<T> value is unwrapped, apply the decryption to the encrypted properties

VALIDATING CONFIGURATION VALUES

- Options validation added in .NET Core 2.2, making use of System.ComponentModel.DataAnnotations
  - Requires some additional code to work – does not automatically ‘wire-up’ with Configure<T>
  - Can use the AddOptions<T> method instead to do the wiring up, but loses a lot of the functionality that Configure<T> provides
  - Fairly limited
- Alternative is to create your own validator interface and class
  - Register the validator with the DI container as implementation of the interface
  - Can inject the validator into the bridge class in similar way to the decryption

PUTTING IT ALL TOGETHER

```
public interface IAppSettingsResolved : IAppSettings, ISqlConnectionSettings, IOracleConnectionSettings
{
}

public interface IAppSettings
{
    string ApplicationName { get; }
}

public interface ISqlConnectionSettings
{
    string SqlConnectionSting { get; }
}

public interface IOracleConnectionSettings
{
    string OracleConnectionSting { get; }
}

public interface ISettingsDecrypt
{
    string Decrypt(string key, IDictionary<string, string> keyValues);
}

public interface ISettingsValidator
{
    bool TryValidate(IAppSettingsStructure settings, out AggregateException validationExceptions);
}
```

PUTTING IT ALL TOGETHER

```
public class MyAppSettingsBridge : IAppSettingsResolved
{
    public MyAppSettingsBridge(
        IOptionsMonitor<MyAppSettings> appSettings,
        ISettingsDecrypt decryptor,
        ISettingsValidator validator)
    {
        var appSettingsUnwrapped = appSettings?.CurrentValue ?? throw new ArgumentNullException(nameof(appSettings));

        if (decryptor is null) throw new ArgumentException(nameof(decryptor));

        if (validator is null) throw new ArgumentNullException(nameof(validator));

        if (!validator.TryValidate(appSettingsUnwrapped, out var validationException)) throw validationException;

        ApplicationName = appSettingsUnwrapped.ApplicationName;
        SqlConnectionSting = decryptor.Decrypt("Sql", appSettingsUnwrapped.Secrets);
        OracleConnectionSting = decryptor.Decrypt("Oracle", appSettingsUnwrapped.Secrets);
    }

    public string ApplicationName { get; }

    public string SqlConnectionSting { get; }

    public string OracleConnectionSting { get; }
}
```

PUTTING IT ALL TOGETHER

```
public class Program
{
    [ExcludeFromCodeCoverage]
    public static void Main(string[] args)
    {
        BuildWebHost(CreateWebHostBuilder(args)).Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .ConfigureAppConfiguration(AddCustomConfiguration);

    private static IWebHost BuildWebHost(IWebHostBuilder builder) => builder.Build();

    private static void AddCustomConfiguration(WebHostBuilderContext ctx, IConfigurationBuilder builder)
    {
        builder.AddJsonFile(new PhysicalFileProvider(@"C:\"),@"secrets.json",true,true);
    }
}
```

PUTTING IT ALL TOGETHER

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
        services.Configure<MyAppSettings>(Configuration.GetSection("MyAppSettings"));
        services.AddSingleton(Configuration);

        var salt = Configuration["CryptoSalt"];
        var pwd = Configuration["CryptoPwd"];

        services.AddSingleton( x => new CryptoFactory().Create<AesManaged>(pwd, salt));
        services.AddSingleton<ISettingsDecrypt, SettingsDecryptor>();
        services.AddSingleton<ISettingsValidator, SettingsValidator>();
        services.AddScoped<IAppSettingsResolved, MyAppSettingsBridge>();

        // add the other interfaces implemented by MyAppSettingsBridge to allow for resolution by those interfaces (interface segregation)
        services.AddScoped<IAppSettings>(provider => provider.GetService<IAppSettingsResolved>());
        services.AddScoped<ISqlConnectionSettings>(provider => provider.GetService<IAppSettingsResolved>());
        services.AddScoped<IOracleConnectionSettings>(provider => provider.GetService<IAppSettingsResolved>());
    }

}
```

PUTTING IT ALL TOGETHER

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
        services.Configure<MyAppSettings>(Configuration.GetSection("MyAppSettings"));
        services.AddSingleton(Configuration);

        var salt = Configuration["CryptoSalt"];
        var pwd = Configuration["CryptoPwd"];

        services.AddSingleton( x => new CryptoFactory().Create<AesManaged>(pwd, salt));
        services.AddSingleton<ISettingsDecrypt, SettingsDecryptor>();
        services.AddSingleton<ISettingsValidator, SettingsValidator>();
        services.AddScoped<IAppSettingsResolved, MyAppSettingsBridge>();

        // add the other interfaces implemented by MyAppSettingsBridge to allow for resolution by those interfaces (interface segregation)
        services.AddScoped<IAppSettings>(provider => provider.GetService<IAppSettingsResolved>());
        services.AddScoped<ISqlConnectionSettings>(provider => provider.GetService<IAppSettingsResolved>());
        services.AddScoped<IOracleConnectionSettings>(provider => provider.GetService<IAppSettingsResolved>());
    }

}
```

PUTTING IT ALL TOGETHER

```
public class HomeController : Controller
{
    private readonly IAppSettingsResolved _settings;

    public HomeController(IAppSettingsResolved settings)
    {
        _settings = settings ?? throw new ArgumentNullException(nameof(settings));
    }

    public IActionResult Index()
    {
        return View(_settings);
    }

    public IActionResult About()
    {
        ViewData["Message"] = "Your application description page.";
        return View(_settings);
    }

    public IActionResult Contact()
    {
        ViewData["Message"] = "Your contact page.";
        return View(_settings);
    }

    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

SO THAT'S IT …

- We have negated the need for code (other than DI registration) needing to reference the IConfiguration implementation by using interfaces registered with the DI container
- We have hidden the Options pattern so it does not pollute outside of DI registration by introducing bridge classes
- Each moving part has a single responsibility
- Library code does not need to know about the Microsoft.Extensions.Options NuGet package

RESOURCES

- Microsoft Docs
  - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration
- User Secrets Documentation
  - https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets
- Andrew Lock's blog
  - https://andrewlock.net/tag/configuration/
  - ASP.NET Core In Action book
- Steve Gordon's Pulralsight Course
  - Coming soon … follow @stevejgordon for announcements

THANK YOU

Find out more …

- Four part series on my blog – https://stevetalkscode.co.uk
- Git hub repos
  - bit.ly/NetConfigIsEasy
-  @stevetalkscode