**Adding options to my project:**

Almost every project will have some settings that need to be configured and changed depending on the environment, or secrets that you don't want to hard code into your repository. The classic example is connection strings and passwords etc.

Application settings can be stored in multiple places - environment variables, appsettings.json, user secrets.

**Strongly typed configuration:**

In ASP.NET Core, there is now no default AppSettings["MySettingKey"] way to get settings. Instead, the recommended approach is to create a strongly typed configuration class with a structure that matches a section in your configuration file (or wherever your configuration is being loaded from):

public class UnitOptions

{

public string Temp { get; set; } = String.Empty;

public string Distance { get; set; } = String.Empty;

}

Would map to the lower section in the appsettings.json below.

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Debug",

"System": "Information",

"Microsoft": "Information"

}

},

"Units": {

"Temp": "Celsius",

"Distance": "Miles"

}}

Bind the UnitOptions class to the corresponding section in appsettings.json by registering configuration instance in Program.cs

builder.Services.Configure<UnitOptions>(builder.Configuration.GetSection("Units"));

**IOptions:**

**IOptions** is singleton and hence can be used to read configuration data within any service lifetime. Being singleton, it cannot read changes to the configuration data after the app has started.

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

[ApiController]

public class OptionsDemoController : ControllerBase

{

private readonly ITransientService \_transientService;

public OptionsDemoController(TransientService transientService)

{

\_transientService = transientService;

}

[HttpGet]

[Route("/units/transient")]

public IActionResult GetUnitsTransient() => Ok(\_transientService.GetUnits());

}

public interface ITransientService

{

UnitOptions GetUnits();

}

public class TransientService : ITransientService

{

private readonly UnitOptions \_unitOptions;

public TransientService(IOptions<UnitOptions> unitOptions)

{

\_unitOptions = unitOptions.Value;

}

public UnitOptions GetUnits()

{

return \_unitOptions;

}

}

**IOptionsSnapshot**

IOptionsSnapshot is scoped and hence it can be used only with transient and scoped service lifetimes. Being scoped, it can recompute config data for each request.

public interface IScopedService

{

UnitOptions GetUnits();

}

public class ScopedService : IScopedService

{

private readonly UnitOptions \_unitOptions;

public ScopedService(IOptionsSnapshot<UnitOptions> unitOptions)

{

\_unitOptions = unitOptions.Value;

}

public UnitOptions GetUnits()

{

return \_unitOptions;

}

}

While the app is still running, change the value of distance unit from ‘Miles’ to ‘Kilometres’ in the appsettings.json file and hit the same API controller action again. The response reflects the changes to the config data.

**IOptionsMonitor**

IOptionsMonitor is singleton and hence can be used to read configuration data in any service lifetime. However, as opposed to IOptions, it can retrieve current config data at any time.

Create a singleton service with an injected IOptionsMonitor instance as follows:-

public interface ISingletonService

{

UnitOptions GetUnits();

}

public class SingletonService : ISingletonService

{

private readonly IOptionsMonitor<UnitOptions> \_unitOptions;

public SingletonService(IOptionsMonitor<UnitOptions> unitOptions)

{

\_unitOptions = unitOptions;

}

public UnitOptions GetUnits()

{

return \_unitOptions.CurrentValue;

}

}

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

[ApiController]

public class OptionsDemoController : ControllerBase

{

private readonly ITransientService \_transientService;

private readonly IScopedService \_scopedService;

private readonly ISingletonService \_singletonService;

public OptionsDemoController(ITransientService transientService, IScopedService scopedService, ISingletonService singletonService)

{

\_transientService = transientService;

\_scopedService = scopedService;

\_singletonService = singletonService;

}

[HttpGet]

[Route("/units/transient")]

public IActionResult GetUnitsTransient() => Ok(\_transientService.GetUnits());

[HttpGet]

[Route("/units/scoped")]

public IActionResult GetUnitsScoped() => Ok(\_scopedService.GetUnits());

[HttpGet]

[Route("/units/singleton")]

public IActionResult GetUnitsSingleton() => Ok(\_singletonService.GetUnits());

}

**Environment variables:**

Environment variables are used to avoid storage of app secrets in code or in local configuration files. Environment variables override configuration values for all previously specified configuration sources.

**Environment.GetEnvironmentVariable("envVar");**

**Secret Manager:**

The Secret Manager tool stores sensitive data during the development of an ASP.NET Core project. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

Use of nuget: Microsoft.Extensions.Configuration.UserSecrets

Generate Your User Secrets File

To generate your user secrets file, right-click on the common/config project (whichever utilizes connection strings) and select Manage User Secrets. A file named secrets.json should be opened. Any configuration values you want to store for local use should be stored here.

In file add key value pairs: "MySecret" : "12345"

builder.Configuration.AddUserSecrets(Assembly.GetExecutingAssembly(), true);

Access: builder.Configuration["MySecret"];