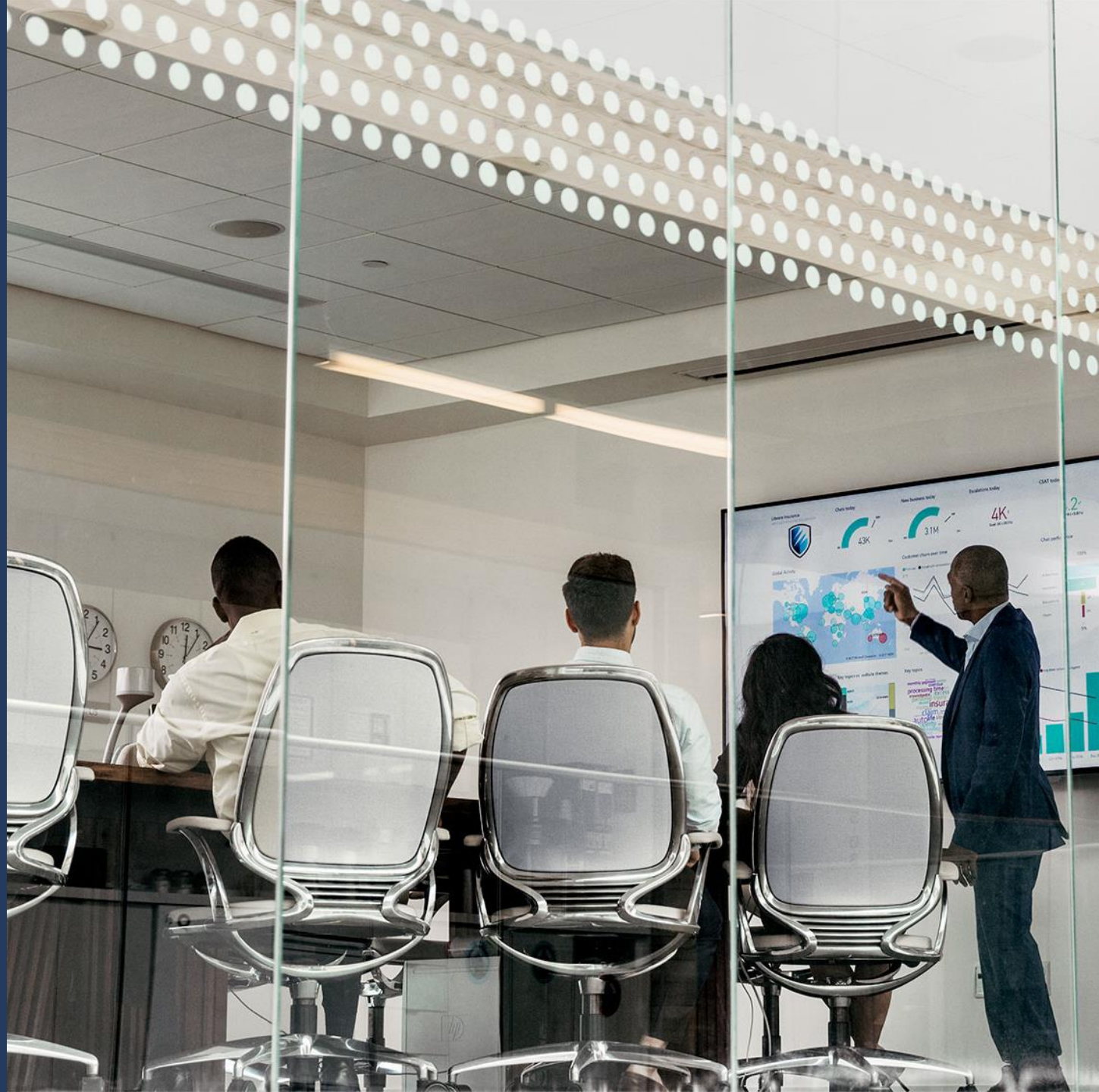
Microsoft

Configuration in NET Core

Orestis Meikopoulos
Microsoft Azure & AI Consultant

https://www.linkedin.com/in/ormikopo



Agenda

- Configuration in .NET Core
- Security guidelines – Secret Manager & Azure Key Vault

Configuration - Overview

- When writing applications we may have some settings / values that we want to
  - Avoid hard-coding inside our codebase to be able to easily change them during runtime, without having to recompile and redeploy our code
  - Vary the exact values of them depending on the environment in which the application is running
- These values might include sensitive data such as
  - **Passwords**, **connection strings** and **API keys**
- ASPNET Core deals with these by providing us with a **dictionary of settings** through DI using the **IConfiguration** interface

Audience Question 1

What exactly is a Dictionary in C#?

Configuration in NET Core - Default Configuration

- Configuration in ASP.NET Core is performed using one or more **configuration providers**
- Configuration providers read configuration data from key-value pairs using a variety of **configuration sources**, such as
  - Settings files, such as appsettings.json
  - Environment variables
  - Azure Key Vault
  - Azure App Configuration
  - Command-line arguments
  - Other sources, like custom providers, directory files or in-memory .NET objects

Configuration in NET Core - Default Configuration

- **CreateDefaultBuilder** provides default configuration for the app in the following order
  - appsettings.json using the **JSON configuration provider**
  - appsettings.{Environment}.json using the **JSON configuration provider**
  - App secrets when the app runs in the Development environment using the **Secret Manager**
  - Environment variables using the **Environment Variables configuration provider**
  - Command-line arguments using the **Command-line configuration provider**
- Config providers that are added later **override** previous key settings

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

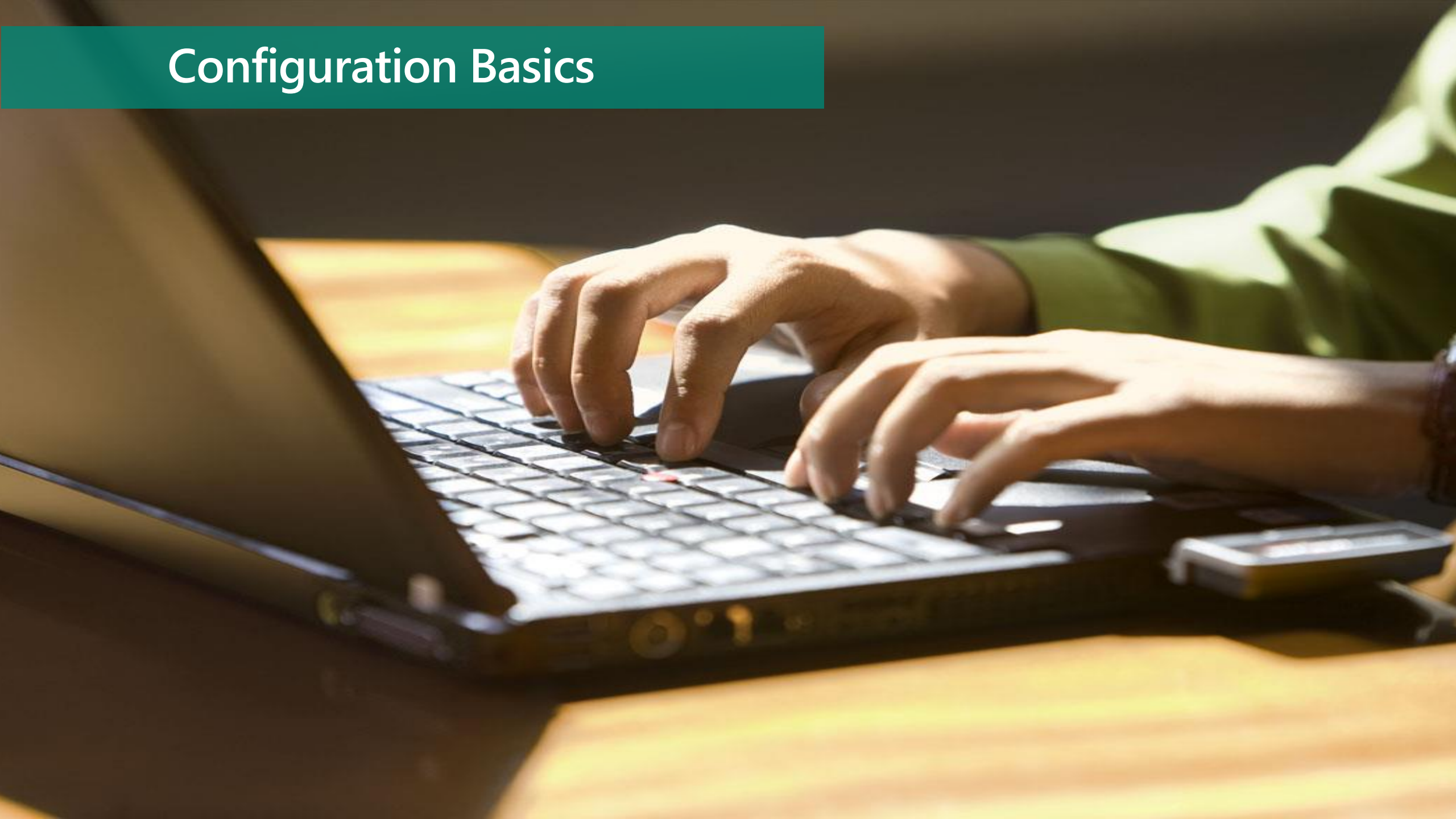
Configuration in NET Core - Read Values & Connection Strings

- Reading **simple values** from **appsettings** file with **IConfiguration**
  - The IConfiguration is auto-registered with the DI and can be injected in any class
  - This IConfiguration has a string-based indexer that allow reading values with JSON-style keys
- Reading **connection strings**
  - Connections strings are kept in any configuration file and can be read the same way as any configuration value
  - But, following the conventions, if **ConnectionStrings** is kept at top level of appsettings.json, there is a handy method to read them

```
public class TestController : Controller
{
    IConfiguration _configuration;

    public TestController(IConfiguration configuration,)
    {
        _configuration = configuration;
    }

    public IActionResult Get()
    {
        //get config value with IConfiguration
        var logLevel = _configuration["Logging:Debug:LogLevel:Default"];
        //from array with index
        var firstServerName = _configuration["Servers:0:Name"];
        //casting with (optional) default value
        var country = _configuration.GetValue<string>("Address:Country", "India");

        return new OkObjectResult($"Log level: {logLevel}");
    }
}
```

```
var primaryConnStr = Configuration.GetConnectionString("PrimaryDB");
//which is simply a short-hand for
var secondaryConnStr = Configuration.GetSection("ConnectionStrings")["SecondaryDB"];
```

Configuration Basics



Audience Question 2

What would be the output of logLevelValue and serverName?

appsettings.json

Schema: https://json.schemastore.org/appsettings

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "Position": {
    "Title": "Editor",
    "Name": "John Doe"
  },
  "OAuthSettings": {
    "ClientId": "123",
    "ClientSecret": "somesecret",
    "Scope": "scope",
    "RedirectUrl": "redirect_url"
  },
  "Servers": [
    {
      "Name": "Server1"
    },
    {
      "Name": "Server2"
    }
  ],
```

6 references | Orestis Meikopoulos, 4 days ago | 1 author, 1 change

```dotnet
public class ReadSimpleValueModel : PageModel
{
    private readonly IConfiguration _configuration;

    0 references | Orestis Meikopoulos, 4 days ago | 1 author, 1 change
    public ReadSimpleValueModel(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    0 references | Orestis Meikopoulos, 4 days ago | 1 author, 1 change
    public void OnGet()
    {
        // Get config values with IConfiguration
        var logLevelValue = _configuration["Logging:LogLevel:Microsoft"];
        var serverName = _configuration["Servers:1:Name"];
    }
}
```

Configuration in NET Core - Options Pattern

- A complete configuration file or any part of it can be bound to a simple POCO object and then can be injected where it makes sense as strongly-typed configuration
- Bind hierarchical configuration data and read related configuration values using the **options pattern**
- For this, create an **options class** that is compatible with the section of configuration file that needs to be casted

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; }
    public string Name { get; set; }
}
```

appsettings.json

Schema: https://json.schemastore.org/appsettings

```
1  {
2    "Position": {
3      "Title": "Editor",
4      "Name": "John Doe"
5    },
```

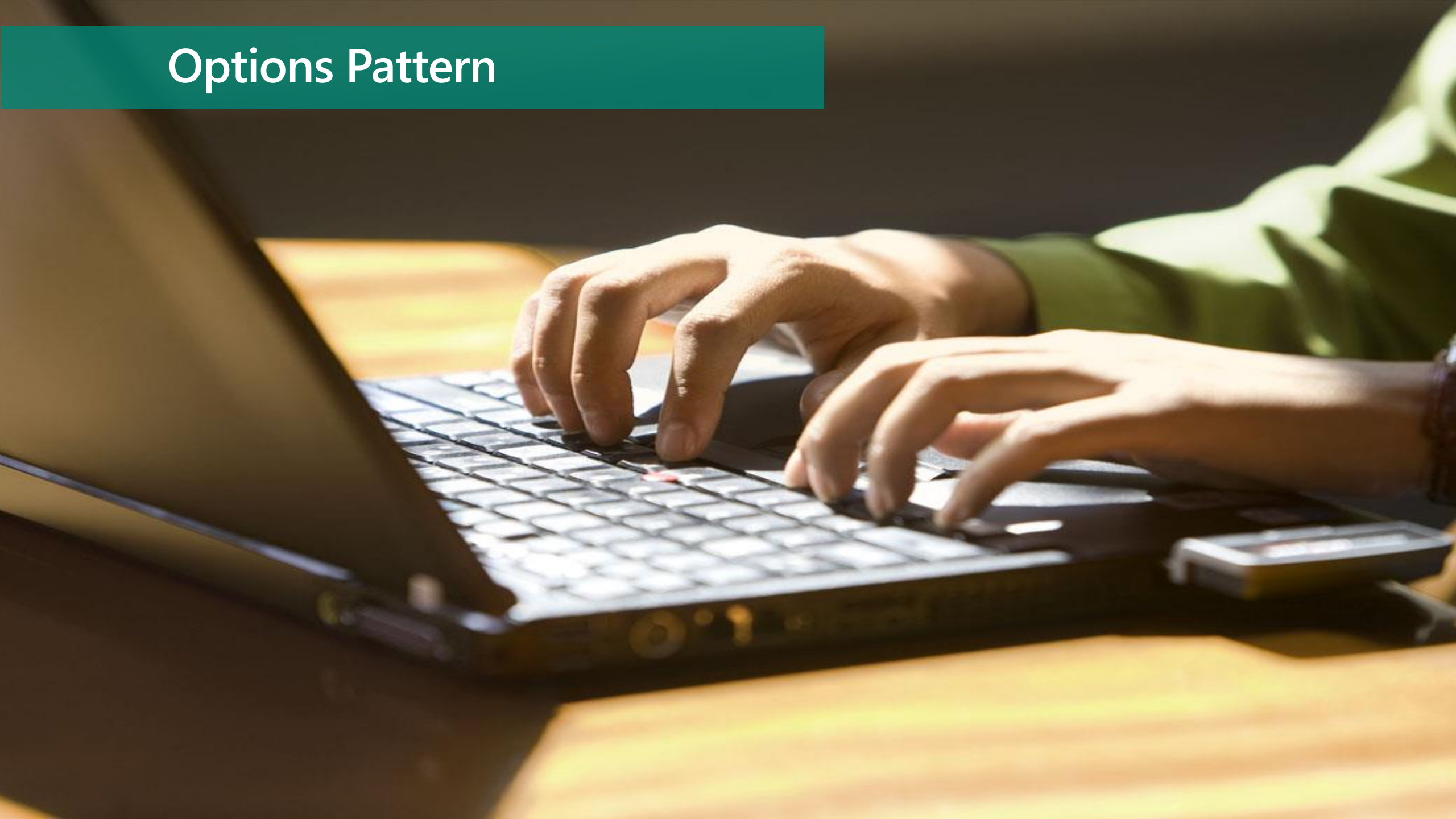
Configuration in NET Core - Options Pattern

- Code inside the ConfigureServices() method shows how a configuration is casted to an object and registered with the DI container
- The configuration registered in code can be injected with an **IOptions** wrapper of the created instance type e.g. IOptions<PositionOptions> for our code sample

```
2 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
public class Startup
{
    0 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    2 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public void ConfigureServices(IServiceCollection services)
    {
        // For use in OptionsPattern page
        services.Configure<PositionOptions>(
            Configuration.GetSection(PositionOptions.Position)
        );

        services.AddRazorPages();
    }
```

```
0 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
public OptionsPatternModel(IConfiguration configuration, IOptions<PositionOptions> options)
{
    Configuration = configuration;
    _options = options.Value;
}
```

Options Pattern



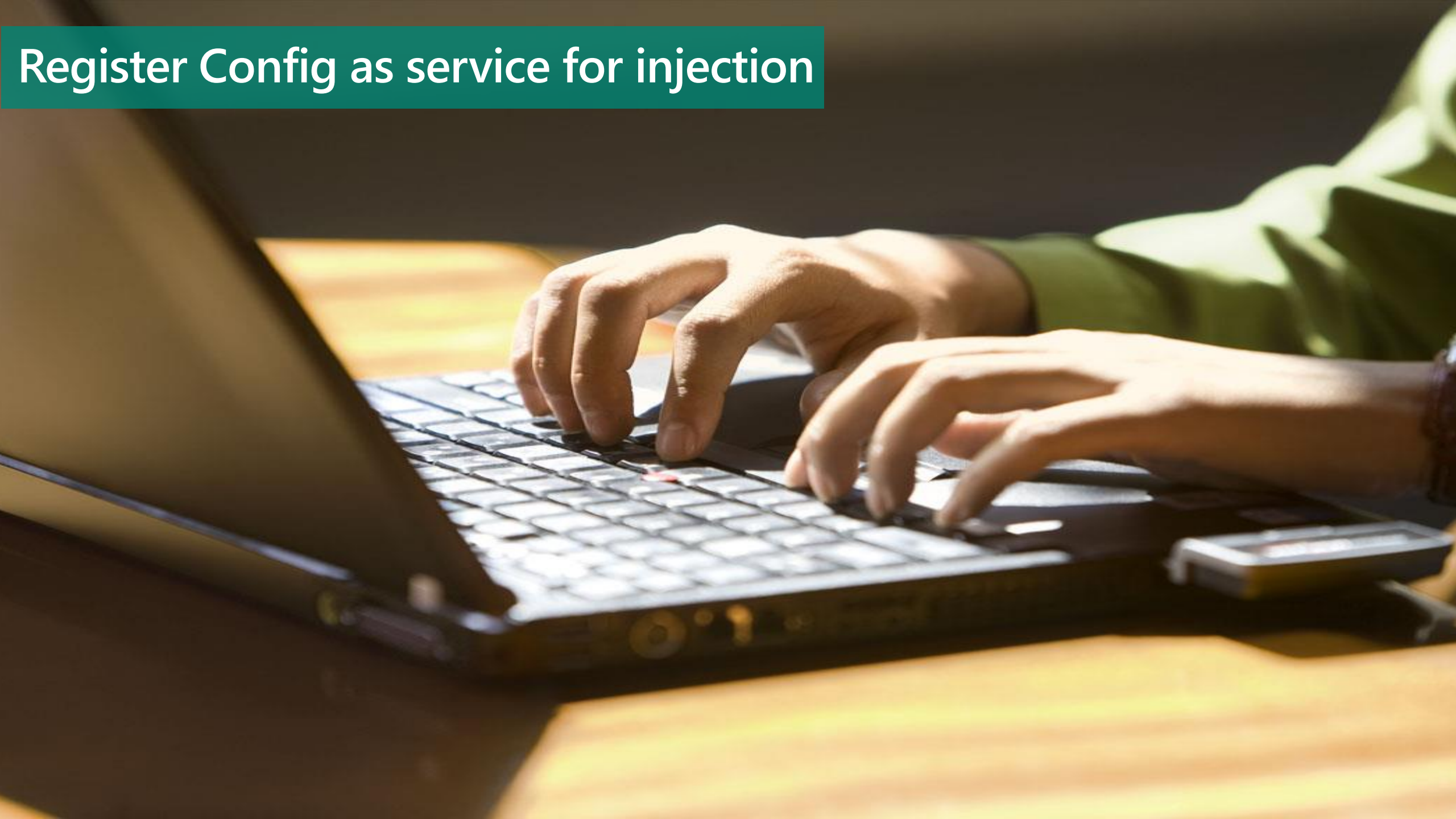
Configuration in NET Core - Register Config as service for DI

- To use that strongly-typed configuration object **without the IOptions<> wrapper**, we need to create an instance of that object in Startup with the required config section populated, and register that as injectable object
- Once that object has been registered as singleton, it can be injected with the DI framework into any class by the config object type

```
public class Startup
{
    0 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    2 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public void ConfigureServices(IServiceCollection services)
    {
        // Register the settings for MyOAuthConfig section
        var settings = new OAuthSettings();
        Configuration.Bind("OAuthSettings", settings);
        services.AddSingleton(settings);

        services.AddRazorPages();
    }
```

```
6 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
public class RegisterConfigAsServiceForInjectionModel : PageModel
{
    private readonly OAuthSettings _settings;

    0 references | Orestis Meikopoulos, 54 days ago | 1 author, 1 change
    public RegisterConfigAsServiceForInjectionModel(OAuthSettings settings)
    {
        _settings = settings;
    }
```

Register Config as service for injection



Configuration in NET Core – Security Data Guidelines

- Never store passwords or other sensitive data (e.g. production connection strings) in configuration code
- The **Secret manager** can be used to store secrets in **development**
- Don't use production secrets in development or test environments
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository
- Use **environment variables** or **Azure Key Vault** to read secret data and passwords in **production**

Audience Question 3

Which property of the CrmAppDbContext class could we extract to a configuration value (e.g. inside appSettings.json file)? Is it something we should consider keeping secret and be careful not to be committed inside our codebase?

```
public class CrmAppDbContext:DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderProduct> OrderProducts { get; set; }

    public readonly static string connectionString =
        "Server =localhost; " +
        "Database =crm; " +
        "User Id =sa; " +
        "Password =passw0rd;";

    protected override void OnConfiguring
        (DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }
    public CrmAppDbContext(DbContextOptions<CrmAppDbContext> options)
          : base(options)
    { }
    public CrmAppDbContext()
    { }
}
```

Configuration in NET Core - Security & Secret Manager

- The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored
  - Values are stored in a simple JSON file inside a user folder on the local machine
  - e.g. on Windows: %APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
    - **<user_secrets_id>** is the **UserSecretsId** value specified in the .csproj file when enabling the Secret Storage
- To enable Secret Storage
  - Either run **dotnet user-secrets init** or
  - In VS, right-click the project and select **Manage User Secrets** from the context menu

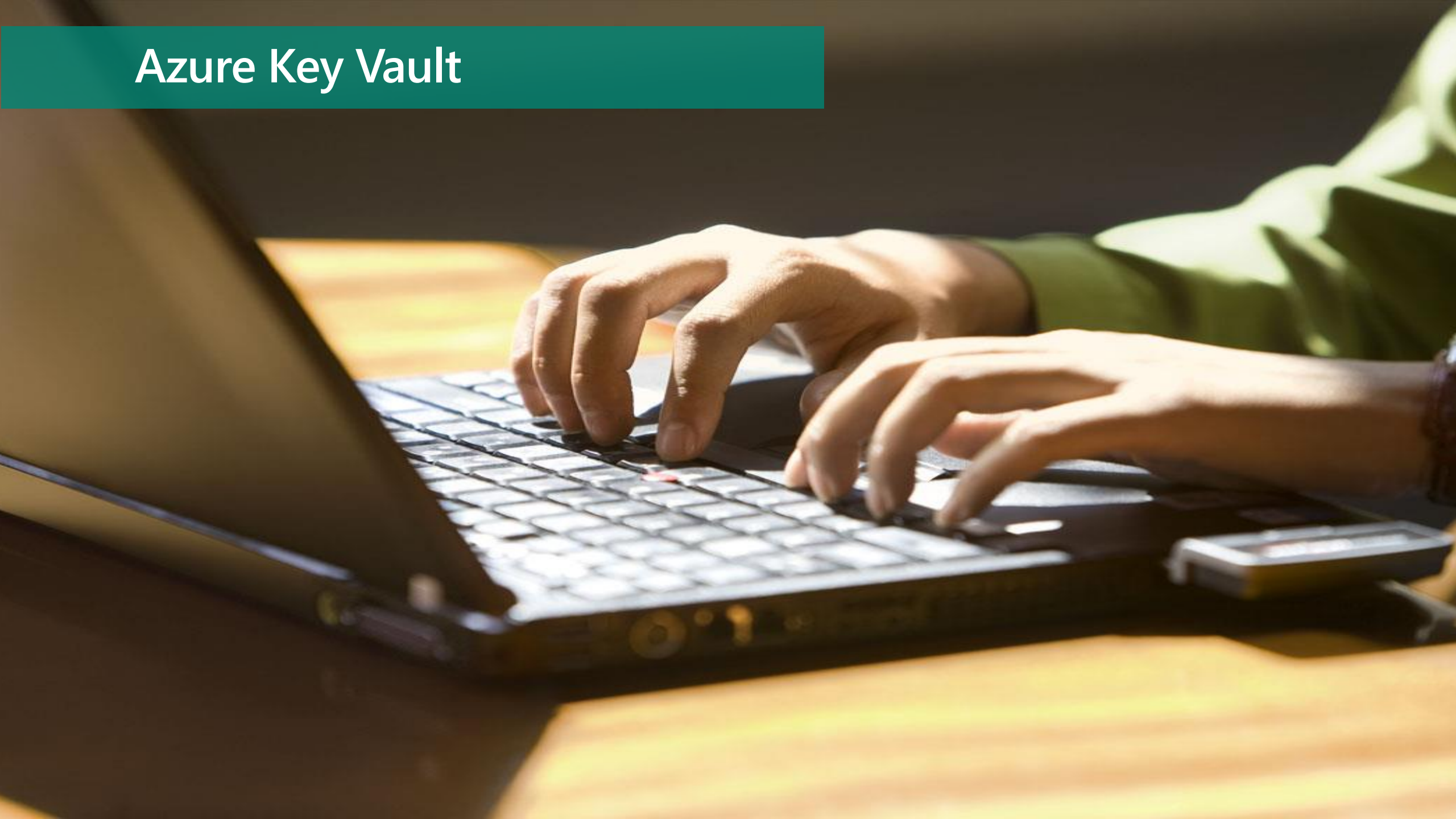
```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

Secret Manager

Configuration in NET Core – Azure Key Vault

- Azure Key Vault is a cloud-based service that assists in
  - Safeguarding cryptographic keys and secrets used by apps and services
- Common scenario for using Azure Key Vault with ASP.NET Core apps include:
  - Controlling access to sensitive configuration data
- You can use the Microsoft Azure Key Vault Configuration Provider to
  - Load app configuration values from Azure Key Vault secrets when your app is running in **production**

Azure Key Vault



Microsoft

Thank You

- For the opportunity
- For participating
- For listening

© Copyright Microsoft Corporation. All rights reserved.