ASP.NET Core - True Ultimate Guide

Section 14: Configuration

ASP.NET Core Configuration

Configuration is the cornerstone of any application, providing essential settings and values that drive its behavior. ASP.NET Core's configuration system is flexible and extensible, allowing you to retrieve configuration data from various sources and prioritize them according to your needs.

Core Concepts

- **Configuration Providers:** These components read configuration data from different sources and populate a central configuration store.
- **Configuration Sources:** The actual locations or mechanisms where your configuration data resides (e.g., files, environment variables, command-line arguments).
- **Key-Value Pairs:** Configuration data is stored as key-value pairs, where the key is a string identifier, and the value is the configuration data (string, number, boolean, etc.).

Common Configuration Sources

1. **Files (JSON, XML, INI):**
    - **Purpose:** Storing configuration data in structured files. JSON is the default and most common format in ASP.NET Core.
    - **Pros:** Easy to read and edit, supports hierarchical structure.
    - **Cons:** Might not be suitable for storing secrets or highly sensitive data.
2. **Environment Variables:**
    - **Purpose:** Reading configuration values from environment variables.
    - **Pros:** Ideal for environment-specific settings (e.g., database connection strings) and secrets.
    - **Cons:** Can be difficult to manage for complex configurations or large numbers of settings.
3. **Command-Line Arguments:**
    - **Purpose:** Overriding configuration values when running the application from the command line.
    - **Pros:** Provides flexibility for dynamic configuration on the fly.
    - **Cons:** Might not be suitable for storing complex or sensitive data.

4. **In-Memory .NET Objects:**
    - **Purpose:** Storing configuration data in a dictionary or custom objects directly in your code.
    - **Pros:** Flexibility for dynamic or programmatic configuration scenarios.
    - **Cons:** Not persistent, less suitable for managing a large number of settings.
5. **Azure Key Vault:**
    - **Purpose:** Securely storing secrets and sensitive configuration data in the cloud.
    - **Pros:** Highly secure, centralized management of secrets.
    - **Cons:** Requires Azure subscription and setup.
6. **Azure App Configuration:**
    - **Purpose:** A powerful cloud-based service for managing feature flags and configuration settings.
    - **Pros:** Feature flag management, centralized configuration, dynamic updates.
    - **Cons:** Requires Azure subscription and setup.
7. **User Secrets (Development):**
    - **Purpose:** Storing sensitive data (e.g., API keys) during development without committing them to source control.
    - **Pros:** Secure and convenient for local development.
    - **Cons:** Not intended for production environments.

Adding and Managing Configuration Sources in Program.cs

```
var builder = WebApplication.CreateBuilder(args);

var configuration = builder.Configuration;


// Add configuration sources in the desired order of precedence (last added wins)

configuration.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true);

configuration.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
reloadOnChange: true);

configuration.AddEnvironmentVariables();

configuration.AddUserSecrets<Program>(); // For development secrets

// ... other sources ...
```

1. AddJsonFile: Loads configuration from JSON files.
2. AddEnvironmentVariables: Loads configuration from environment variables.
3. AddUserSecrets<Program>(): Loads configuration from the user secrets store (for development).

When to Use Which Configuration Source

- **appsettings.json:** For default settings, base configurations, non-sensitive data.
- **appsettings.{Environment}.json:** For environment-specific overrides.
- **Environment Variables:** For environment-specific settings, sensitive data (API keys, connection strings).
- **Command-Line Arguments:** For overriding settings during development or deployment.
- **User Secrets:** For sensitive data during local development.
- **Azure Key Vault:** For storing secrets and other sensitive data securely in production.
- **Azure App Configuration:** For dynamic configuration updates, feature flags, and centralized management.

Best Practices

- **Layered Configuration:** Use multiple sources with a well-defined order of precedence to keep your configuration organized and flexible.
- **Environment-Specific Settings:** Separate sensitive and environment-specific settings into appropriate files.
- **Secrets Management:** Use Azure Key Vault or other secure mechanisms to store sensitive data.
- **Strong Typing:** Create strongly typed configuration classes using the Options pattern (IOptions<T>) for improved type safety and easier access to your settings in code.
- **Validation:** Validate your configuration values during startup to catch errors early.
- **Logging:** Log configuration-related events to help with troubleshooting and debugging.

IConfiguration

In ASP.NET Core, the IConfiguration interface is the heart of the configuration system. It represents a set of key-value pairs that can be loaded from various sources (JSON files, environment variables, etc.). This interface provides a unified way to access your application's settings, regardless of where they are stored.

Key Methods, Properties, and Indexers

1. **GetSection(string key):**
    - **Purpose:** Retrieves a specific section of the configuration as an IConfigurationSection. Sections allow you to group related settings.
    - **Example:**

      var connectionStrings = configuration.GetSection("ConnectionStrings");

2. **GetValue<T>(string key):**
    - **Purpose:** Retrieves a configuration value as a specified type T.
    - **Example:**

      var port = configuration.GetValue<int>("Server:Port");

3. **GetConnectionString(string name):**
    - **Purpose:** Retrieves a connection string from the "ConnectionStrings" section of the configuration.
    - **Example:**

      var connectionString = configuration.GetConnectionString("DefaultConnection");

4. **GetChildren():**
    - **Purpose:** Returns an enumerable collection of IConfigurationSection objects representing the immediate children of the current section.
    - **Example:**

      var sections = configuration.GetSection("Logging").GetChildren();

5. **Indexer (this[string key]):**
    - **Purpose:** Retrieves a configuration value as a string.
    - **Example:**

      var value = configuration["Logging:LogLevel:Default"];

Injecting IConfiguration

- **In Controllers:**

```
public class HomeController : Controller
{
   private readonly IConfiguration _configuration; // Field to store IConfiguration

   public HomeController(IConfiguration configuration)
   {
      _configuration = configuration;
   }

   public IActionResult Index()
   {
      var myKeyValue = _configuration["MyKey"]; // Access configuration value
      return View();
   }
}
```

- **In Services:**

```
public class EmailService : IEmailService
{
   private readonly IConfiguration _configuration;

   public EmailService(IConfiguration configuration)
   {
      _configuration = configuration;
   }

   public void SendEmail(string to, string subject, string body)
   {
```

```
        var smtpServer = _configuration["Email:SmtpServer"]; // Use configuration for email settings
        // ... (email sending logic)
    }
}
```

In both cases, the IConfiguration is injected through the constructor using ASP.NET Core's dependency injection.

Best Practices

- **Strongly Typed Configuration:** Use the Options pattern (IOptions<T>) to map your configuration values to strongly typed objects for easier access and type safety.
- **Environment-Specific Settings:** Use appsettings.{Environment}.json files to store configuration values that vary depending on the environment (Development, Production, etc.).
- **Secret Management:** Store sensitive information (e.g., passwords, API keys) in Azure Key Vault or other secure storage mechanisms.
- **Layered Configuration:** Combine multiple configuration sources (files, environment variables, etc.) with a well-defined order of precedence.
- **Reload On Change:** Consider using reloadOnChange: true in your configuration providers to automatically reload configuration changes without restarting the application.

Example: Options Pattern

```
// MyOptions.cs
public class MyOptions
{
    public string Option1 { get; set; }
    public int Option2 { get; set; }
}

// Program.cs (or Startup.cs)
builder.Services.Configure<MyOptions>(builder.Configuration.GetSection("MyOptions"));
```

```
// MyService.cs
public class MyService : IMyService
{
    private readonly IOptions<MyOptions> _options;

    public MyService(IOptions<MyOptions> options)
    {
        _options = options;
    }

    public void DoSomething()
    {
        var option1Value = _options.Value.Option1;
        // ...
    }
}
```

In this example, the MyOptions class represents a section of your configuration. The IOptions<MyOptions> interface provides a strongly typed way to access those settings within your services.

By following these best practices and leveraging the power of IConfiguration, you can build robust and adaptable ASP.NET Core applications with well-organized and easily manageable configuration settings.

Hierarchical Configuration

In ASP.NET Core, you can organize your configuration settings into a hierarchical structure using JSON, XML, or INI files. This hierarchical structure allows you to group related settings under sections and subsections, making your configuration more readable, maintainable, and scalable.

JSON-Based Hierarchical Configuration (appsettings.json):

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=MyDatabase;Trusted_Connection=True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Inventory": {
    "StockAlertThreshold": 20,
    "WarehouseLocations": [
      "New York",
      "London",
      "Tokyo"
    ]
  }
}
```

In this example:

- **Sections:** The top-level keys (ConnectionStrings, Logging, Inventory) define sections within the configuration.
- **Nested Sections:** The Logging section further contains a nested LogLevel section.
- **Arrays:** The WarehouseLocations setting is an array of strings within the Inventory section.

Accessing Hierarchical Configuration with IConfiguration

The IConfiguration interface provides methods to easily navigate and retrieve values from this hierarchical structure.

- **GetSection(string key):**
    - Returns an IConfigurationSection object representing the specified section.
    - Use this to drill down into nested sections.
- **GetValue<T>(string key):**
    - Retrieves a configuration value as the specified type T.
    - The key can include the entire path to the value, using colons (:) to separate sections.
- **Indexer (this[string key]):**
    - Retrieves a configuration value as a string.
    - Works like the GetValue<string>() method.

Code Examples

```
var connectionString = _configuration.GetConnectionString("DefaultConnection");


var logLevel = _configuration.GetValue<string>("Logging:LogLevel:Default");


// Using IConfigurationSection:

var inventorySection = _configuration.GetSection("Inventory");

var stockAlertThreshold = inventorySection.GetValue<int>("StockAlertThreshold");


// Get an array

var warehouseLocations = inventorySection.GetSection("WarehouseLocations").Get<string[]>();
```

Best Practices

- **Clear Structure:** Organize your settings into logical sections and subsections for better readability and maintainability.
- **Consistent Naming:** Use meaningful and consistent naming conventions for your configuration keys.

- **Strong Typing with Options Pattern:** Use the Options pattern (IOptions<T>) to map your configuration sections to strongly typed classes, which provides type safety and makes your code easier to work with.
- **Environment Variables:** Consider using environment variables for settings that may vary across environments (e.g., ASPNETCORE_ENVIRONMENT).
- **Secret Management:** Never store sensitive information (passwords, API keys) directly in configuration files. Use Azure Key Vault, Secret Manager, or other secure mechanisms to manage secrets.

Example: Options Pattern

```
// InventoryOptions.cs
public class InventoryOptions
{
    public int StockAlertThreshold { get; set; }
    public string[] WarehouseLocations { get; set; }
}

// Program.cs (or Startup.cs)
builder.Services.Configure<InventoryOptions>(builder.Configuration.GetSection("Inventory"));

// In your service or controller
public class InventoryService : IInventoryService
{
    private readonly InventoryOptions _options;

    public InventoryService(IOptions<InventoryOptions> options)
    {
        _options = options.Value;
    }

    // ... use _options.StockAlertThreshold and _options.WarehouseLocations
}
```

Options Pattern

The Options pattern is a design pattern in ASP.NET Core that enables you to access configuration values in a strongly typed manner. Instead of retrieving configuration values as strings and manually converting them to the appropriate types, you define POCO (Plain Old CLR Object) classes that represent the structure of your configuration sections. These classes, known as "options" classes, make your configuration code more readable, maintainable, and less error-prone.

Benefits of the Options Pattern

- **Strongly Typed Access:** Access your configuration values directly as properties of your options classes, eliminating the need for manual type conversions and reducing the risk of runtime errors.
- **IntelliSense Support:** Get code completion and type checking in your IDE when working with your configuration settings.
- **Validation:** You can easily add validation logic to your options classes to ensure that configuration values are valid.
- **Clean Separation:** Keep your configuration settings separate from your business logic, improving the overall organization of your code.

When to Use the Options Pattern

- **Related Settings:** When you have groups of related configuration settings that logically belong together (e.g., database connection settings, email settings, feature flags).
- **Strongly Typed Access:** When you want to work with your configuration values in a type-safe manner.
- **Validation:** When you want to add validation logic to ensure your configuration values are valid.

How to Implement the Options Pattern

1. **Create an Options Class:** Define a class that mirrors the structure of your configuration section. Make sure the property names match the keys in your configuration file.

```
public class EmailOptions
{
    public string SmtpServer { get; set; } = string.Empty;
    public int SmtpPort { get; set; } = 25;
```

```
    public string SenderEmail { get; set; } = string.Empty;

    public string SenderPassword { get; set; } = string.Empty;

}
```

2. **Register the Options:** In your Program.cs (or Startup.cs in older versions), register your options class using the Configure<T> extension method on IServiceCollection:

```
builder.Services.Configure<EmailOptions>(builder.Configuration.GetSection("Email"));
```

This tells the DI container to bind the settings in the Email section of your configuration to an instance of EmailOptions.

3. **Inject IOptions<T>:** Inject the IOptions<T> interface into your controllers or services to access the bound options:

```
public class EmailService : IEmailService

{

    private readonly EmailOptions _options;


    public EmailService(IOptions<EmailOptions> options)

    {

        _options = options.Value;

    }


    // ... use _options.SmtpServer, _options.SmtpPort, etc. ...

}
```

Related Methods for Configuration Access

- **ConfigurationBinder.Get<T>(IConfiguration configuration):** Binds and returns the entire configuration section to a strongly typed object of type T.
- **ConfigurationBinder.Get(IConfiguration configuration, Type type):** Binds and returns the entire configuration section to an object of the specified type.
- **ConfigurationBinder.Bind(IConfiguration configuration, object instance):** Binds the configuration to an existing object instance.

Example: Options Pattern with GetSection and Bind

```
// Program.cs (or Startup.cs)

var emailOptions = new EmailOptions();

builder.Configuration.GetSection("Email").Bind(emailOptions);

builder.Services.AddSingleton(emailOptions); // Add the bound object as a singleton
```

Environment-Specific Configuration Files

ASP.NET Core allows you to create configuration files that are specific to different environments. By convention, these files are named appsettings.{Environment}.json, where {Environment} is replaced with the name of the environment (e.g., appsettings.Development.json, appsettings.Production.json).

Purpose:

- **Environment-Specific Settings:** These files store configuration values that are unique to each environment. This could include database connection strings, API keys, logging levels, or feature flags.
- **Customization:** You can tailor your application's behavior for development, testing, staging, and production environments without having to manually modify configuration settings every time you deploy.

Order of Precedence:

ASP.NET Core loads configuration from multiple sources, and the order in which they are loaded determines which values take precedence in case of conflicts. The general order of precedence (from highest to lowest) is:

1. **Command-Line Arguments:** Any configuration values specified as command-line arguments when you run your application (e.g., dotnet run --Logging:LogLevel:Default=Debug) override all other sources.
2. **Environment Variables:** Configuration values set as environment variables on your system take precedence over values in configuration files. ASP.NET Core automatically maps environment variables to configuration keys using a convention. For example, the environment variable ConnectionStrings__DefaultConnection would map to the configuration key ConnectionStrings:DefaultConnection.

3. **User Secrets (Development Only):** If you're in the Development environment, values from the user secrets store (secrets.json) override those from appsettings.json and appsettings.Development.json. This is useful for storing sensitive information during development.
4. **appsettings.{Environment}.json:** If present, settings from this file override values from the base appsettings.json file. This allows you to customize settings for specific environments.
5. **appsettings.json:** This is the base configuration file that is always loaded. It contains the default settings for your application.

Example: Overriding Connection Strings

```
// appsettings.json
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=MyDatabaseDev;Trusted_Connection=True;"
  }
}

// appsettings.Production.json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=myprodserver;Database=MyDatabaseProd;User
Id=myuser;Password=mypassword;"
  }
}
```

If the ASPNETCORE_ENVIRONMENT variable is set to "Production", the connection string from appsettings.Production.json will be used.

Code Example: GetSection() and GetValue()

```
var connectionString = _configuration.GetConnectionString("DefaultConnection");

var logLevel = _configuration.GetValue<string>("Logging:LogLevel:Default");
```

- GetConnectionString("DefaultConnection") is a convenience method to fetch a connection string specifically from the ConnectionStrings section.

- GetValue<string>() retrieves values from specific configuration sections or keys.

Best Practices

- **Logical Structure:** Organize your settings into sections and subsections to make your configuration files easy to read and understand.
- **Consistent Naming:** Use consistent naming conventions for your configuration keys (e.g., kebab-case, snake_case).
- **Environment Variables for Sensitive Data:** Store sensitive information like API keys and connection strings in environment variables or Azure Key Vault, not in configuration files that might be committed to source control.
- **User Secrets for Development:** Use user secrets to store sensitive data during development without exposing it in your code repository.
- **Order Matters:** Be mindful of the order of precedence when adding configuration sources. Place the most important or specific overrides later in the process.
- **Validation:** Consider validating your configuration during application startup to ensure that all required settings are present and have valid values.

Secrets Management in ASP.NET Core

In the world of web development, you'll often need to work with sensitive information like API keys, database connection strings, or passwords. Hardcoding these values directly into your source code is a security risk. That's where Secrets Manager comes into play.

Secrets Manager: Your Digital Vault

Secrets Manager is a tool that provides secure storage and management for your application's secrets. It keeps your sensitive data out of your source code and makes it easier to manage and rotate secrets without redeploying your application.

User Secrets: Keeping Development Secrets Safe

User Secrets is a developer-friendly feature of Secrets Manager specifically designed for local development environments. It allows you to store secrets for a particular project on your local machine without having to commit them to source control, keeping them out of your code repository.

How to Set User Secrets Using the dotnet Command

1. **Initialize:** If you haven't already, initialize user secrets for your project:

dotnet user-secrets init

This command adds a UserSecretsId property to your project's .csproj file, which links the project to a user secrets store.

2. **Set a Secret:** Use the set command to store a secret:

dotnet user-secrets set "MySecretName" "MySecretValue"

Replace "MySecretName" with the desired key and "MySecretValue" with the actual secret value.

3. **List Secrets (Optional):**

dotnet user-secrets list

This command lists all the secrets you've stored for the project.

4. **Remove a Secret (Optional):**

dotnet user-secrets remove "MySecretName"

Accessing User Secrets in Your Code

```
var builder = WebApplication.CreateBuilder(args);

var configuration = builder.Configuration;


// In Program.cs (or Startup.cs):

if (builder.Environment.IsDevelopment())

{

   configuration.AddUserSecrets<Program>();

}
```

This will add a configuration source that can read user secrets, but only when the environment is set to "Development".

Then, to access a user secret, you can use the same techniques you would for any other configuration value:

```
var mySecret = configuration["MySecretName"];
```

Best Practices for Secrets Management

- **Never Hardcode Secrets:** Always store sensitive information in a secure store like Secrets Manager.

- **Least Privilege:** Grant your application the minimum necessary permissions to access secrets.
- **Rotate Secrets Regularly:** Regularly change your secrets to minimize the risk of exposure.
- **Separate Environments:** Use different secrets for different environments (development, staging, production).
- **Automation:** Consider automating the process of secret rotation to enhance security.

Example: Storing an API Key as a User Secret

1. **Initialize:** dotnet user-secrets init
2. **Set Secret:** dotnet user-secrets set "StripeApiKey" "sk_test_1234567890"

Accessing in Your Code (Example):

```
var stripeApiKey = configuration["StripeApiKey"];
```

Caveats

- **Development Only:** User secrets are intended for development environments and should not be used in production.
- **Local Storage:** User secrets are stored in a JSON file on your local machine. Ensure this file is protected.

Set Configuration Values from Environment Variables

- **Flexibility:** You can dynamically change your application's settings without modifying code or configuration files.
- **Security:** Environment variables are a secure way to store sensitive information like API keys, connection strings, or passwords without embedding them in your code.
- **Deployment Environments:** Different environments (development, staging, production) often require distinct configuration values. Environment variables can be easily set and managed per environment.
- **Automation:** This approach lends itself well to automation scripts for deployment and configuration.

How It Works

1. **Environment Variable Prefix:** ASP.NET Core's configuration system recognizes environment variables that start with a specific prefix, by default, ASPNETCORE_. This allows you to namespace your environment variables to avoid conflicts with other variables on your system.
2. **Key Mapping:** The part of the environment variable name after the prefix is used as the configuration key. For example, the environment variable ASPNETCORE_Logging__LogLevel__Default will map to the configuration key Logging:LogLevel:Default. Double underscores (__) are used to represent colons (:) in the hierarchy.
3. **Configuration Provider:** ASP.NET Core has a built-in configuration provider called EnvironmentVariablesConfigurationProvider that automatically reads these environment variables and adds them to the configuration system.

Setting Environment Variables from the Command Line

PowerShell (Windows, macOS, Linux)

```
$env:ASPNETCORE_MyKey = "myvalue"        # Simple key-value

$env:ASPNETCORE_Logging__LogLevel__Default = "Debug"  # Hierarchical key
```

In PowerShell, use the $env: prefix to set environment variables within the current session.

Command Prompt (Windows)

```
set ASPNETCORE_MyKey=myvalue        # Simple key-value

set ASPNETCORE_Logging__LogLevel__Default=Debug  # Hierarchical key
```

Bash (macOS, Linux)

```
export ASPNETCORE_MyKey="myvalue"        # Simple key-value

export ASPNETCORE_Logging__LogLevel__Default="Debug"  # Hierarchical key
```

Example: Setting a Database Connection String

Let's say you want to set your database connection string using an environment variable. Here's how you would do it:

1. **Set the Environment Variable:**

   ```
   # In PowerShell

   $env:ASPNETCORE_ConnectionStrings__DefaultConnection =
   "Server=myServer;Database=myDb;Trusted_Connection=True;"
   ```

```
# In Command Prompt (Windows)

set
ASPNETCORE_ConnectionStrings__DefaultConnection="Server=myServer;Database=myDb;T
rusted_Connection=True;"


# In Bash (macOS/Linux)

    export
    ASPNETCORE_ConnectionStrings__DefaultConnection="Server=myServer;Database=
    myDb;Trusted_Connection=True;"
```

Note the double underscores (__) used to represent the colon (:) in the configuration path.

2. **Access in Your Code:** You can then retrieve this connection string in your ASP.NET Core application using:

```
var connectionString = _configuration.GetConnectionString("DefaultConnection");
```

Notes

- **Prefix:** Remember to use the ASPNETCORE_ prefix for your environment variables.
- **Key Mapping:** Double underscores (__) in the environment variable name are translated to colons (:) in the configuration key.
- **Override:** Environment variable values will override those set in appsettings.json or appsettings.{Environment}.json.
- **Sensitive Data:** This is an excellent way to manage sensitive data without exposing it in your code or configuration files.
- **Deployment:** Make sure to set the appropriate environment variables on your production server before deploying your application.

The Mechanics of Environment Variable Configuration

1. **Environment Variable Prefix:** ASP.NET Core's configuration system recognizes environment variables that start with a specific prefix. By default, this prefix is ASPNETCORE_. You can customize this prefix if needed. This prefix helps to namespace your environment variables and avoid conflicts with other variables on your system.
2. **Key Mapping:** The part of the environment variable name after the prefix is used as the configuration key. A double underscore (__) is used to represent a colon (:) in the hierarchical structure of your configuration. For example:
    - Environment Variable: ASPNETCORE_Logging__LogLevel__Default
    - Configuration Key: Logging:LogLevel:Default
3. **Configuration Provider:** ASP.NET Core includes a built-in configuration provider called EnvironmentVariablesConfigurationProvider. This provider automatically reads environment variables that match the prefix and adds them to the application's configuration. The values from environment variables override any matching values found in appsettings.json or environment-specific configuration files.

Setting Environment Variables from the Command Line

PowerShell (Windows, macOS, Linux)

```
$env:ASPNETCORE_MyKey = "myvalue"        # Simple key-value
$env:ASPNETCORE_Logging__LogLevel__Default = "Debug"  # Hierarchical key
```

Command Prompt (Windows)

```
set ASPNETCORE_MyKey=myvalue        # Simple key-value
set ASPNETCORE_Logging__LogLevel__Default=Debug  # Hierarchical key
```

Bash (macOS, Linux)

```
export ASPNETCORE_MyKey="myvalue"        # Simple key-value
export ASPNETCORE_Logging__LogLevel__Default="Debug"  # Hierarchical key
```

Example: Setting a Database Connection String

Let's say you want to set your database connection string using an environment variable. Here's how you would do it:

1. **Set the Environment Variable:**

   # In PowerShell or Bash

   $env:ASPNETCORE_ConnectionStrings__DefaultConnection =
   "Server=myServer;Database=myDb;User Id=myuser;Password=mypassword;"

2. # In Command Prompt (Windows)

   set
   ASPNETCORE_ConnectionStrings__DefaultConnection="Server=myServer;Database=
   myDb;User Id=myuser;Password=mypassword;"

3. **Access in Your Code:** You can then retrieve this connection string in your ASP.NET Core application as usual:

   var connectionString = _configuration.GetConnectionString("DefaultConnection");

Important Considerations

- **Prefix Customization:** You can change the default ASPNETCORE_ prefix using the AddEnvironmentVariables method. For example, configuration.AddEnvironmentVariables("CUSTOM_PREFIX_");
- **Case Sensitivity:** On Linux and macOS, environment variable names are case-sensitive.
- **Deployment:** When deploying your application, ensure that the appropriate environment variables are set on the target server.
- **Security:** While environment variables are more secure than hardcoding values, they might not be suitable for extremely sensitive secrets. In those cases, consider using a dedicated secret management solution like Azure Key Vault or HashiCorp Vault.

Custom JSON Files

While ASP.NET Core natively supports appsettings.json and environment-specific variations, there are scenarios where using custom JSON files for configuration might be advantageous:

- **Modularity:** You can organize settings into multiple files based on functional areas or components, making your configuration more manageable and easier to navigate.
- **Customization:** You can load custom JSON files conditionally, based on specific requirements or runtime decisions.
- **Separation of Concerns:** This approach allows you to keep default settings in appsettings.json while maintaining custom settings separately.

Adding Custom JSON Files as Configuration Sources

1. **Create the File:** Create a JSON file with your custom configuration settings. Let's call it customsettings.json:

```
{
  "CustomSettings": {
    "APIKey": "your_api_key",
    "FeatureEnabled": true,
    "NotificationSettings": {
      "EmailEnabled": true,
      "SMSEnabled": false
    }
  }
}
```

2. **Add to Configuration:** In your Program.cs, use the AddJsonFile extension method to include your custom JSON file:

```
var builder = WebApplication.CreateBuilder(args);

var configuration = builder.Configuration;


// ... (other configuration sources) ...


// Add the custom JSON file:

configuration.AddJsonFile("customsettings.json", optional: true, reloadOnChange: true);
```

var app = builder.Build();

// ... (rest of the application) ...

- **optional: true:** Set this to true if the file might not exist (e.g., in certain environments).
- **reloadOnChange: true:** Enables automatic reloading of the configuration if the file changes.

Accessing Custom Configuration Values

You can access values from your custom JSON file using the same mechanisms as you would for appsettings.json:

// Option 1: Directly using IConfiguration

var apiKey = configuration["CustomSettings:APIKey"];

var featureEnabled = configuration.GetValue<bool>("CustomSettings:FeatureEnabled");

// Option 2: Options Pattern

var notificationSettings = configuration.GetSection("CustomSettings:NotificationSettings").Get<NotificationSettings>();

Best Practices

- **Naming:** Choose descriptive and meaningful names for your custom JSON files.
- **Organization:** Structure your custom configuration files with sections and subsections to enhance readability and maintainability.
- **Environment-Specific Overlays:** Create environment-specific versions of your custom files (e.g., customsettings.Development.json) to override settings in different environments.
- **Secrets Management:** Store sensitive information (API keys, passwords) in a secure store like Azure Key Vault or User Secrets.
- **Error Handling:** Handle potential errors, such as missing or invalid configuration files, gracefully.
- **Strong Typing with Options:** Strongly recommend using Options Pattern for type safety and better code structure.

Example: Options Pattern with Custom JSON File

```
// CustomSettings.cs (Options Class)
public class CustomSettings
{
    public string APIKey { get; set; }
    public bool FeatureEnabled { get; set; }
    public NotificationSettings NotificationSettings { get; set; }
}

// ... (other options classes if needed) ...

// Program.cs
builder.Services.Configure<CustomSettings>(configuration.GetSection("CustomSettings"));

// In your controller or service
public class MyController : Controller
{
    private readonly CustomSettings _settings;

    public MyController(IOptions<CustomSettings> settings)
    {
        _settings = settings.Value;
    }
}
```

HttpClient

The HttpClient class is a powerful and versatile tool in the .NET ecosystem for interacting with web-based resources over the HTTP protocol. You use it to send requests (GET, POST, PUT, DELETE, etc.) to APIs and retrieve responses containing data in various formats (JSON, XML, HTML).

Key Features of HttpClient

- **Sending Requests:** Craft and send HTTP requests to any URL.
- **Receiving Responses:** Process the server's response (status code, headers, body content).
- **Async Operations:** Designed for asynchronous programming, allowing your application to perform other tasks while waiting for network responses.
- **Customization:** Configure request headers, timeouts, authentication, and more.

Using HttpClient in ASP.NET Core

While you can create and manage HttpClient instances directly, ASP.NET Core offers a more robust approach through the IHttpClientFactory interface. The factory handles the following for you:

- **Connection Pooling:** Manages a pool of HTTP connections, optimizing performance and preventing socket exhaustion.
- **Lifetime Management:** Ensures proper disposal of HttpClient instances to avoid resource leaks.
- **Named Clients:** Lets you define and configure named clients for different APIs, each with its own settings (base address, headers, etc.).

Integrating HttpClient with Your Stock App

Let's analyze how your stock application uses HttpClient and IHttpClientFactory:

1. **FinnhubService:**
    - **Injection:** The constructor injects IHttpClientFactory to create HttpClient instances.
    - **Request Building:** The GetStockPriceQuote method constructs an HttpRequestMessage object, specifying the URL (including the Finnhub API token) and the HTTP method (GET).
    - **Sending the Request:** It uses httpClient.SendAsync to send the request asynchronously.
    - **Response Processing:** It reads the response content as a stream and deserializes the JSON data into a dictionary.
    - **Error Handling:** It checks for errors in the response and throws exceptions accordingly.

2. **HomeController:**
    - **Injection:** It injects both the FinnhubService and IOptions<TradingOptions> for configuration.
    - **Data Fetching:** The Index action calls _finnhubService.GetStockPriceQuote to get stock data.
    - **Model Creation:** It maps the retrieved data to a Stock model object.
    - **View Rendering:** The Stock model is passed to the view for display.

Code Breakdown

- **IFinnhubService:** Defines an interface for the Finnhub service, allowing for different implementations if needed.
- **FinnhubService:** Implements the interface and uses HttpClient to interact with the Finnhub API.
- **TradingOptions:** A class to hold configuration options for the default stock symbol (read from appsettings.json).
- **Stock:** A model class to represent the stock data.
- **HomeController:** The controller that fetches stock data and renders the view.

Best Practices

- **IHttpClientFactory:** Always use IHttpClientFactory instead of directly creating HttpClient instances to benefit from connection pooling and proper lifetime management.
- **Named Clients:** For multiple APIs, use named clients (_httpClientFactory.CreateClient("name");) to configure different settings for each API.
- **Error Handling:** Handle exceptions that might occur during HTTP requests, such as network errors or invalid responses.
- **Resilience:** Consider using Polly or other libraries to implement retries and circuit breaker patterns for increased resilience in the face of transient errors.

Key Points to Remember

- **Purpose:** Provide named configurations to tailor your app's behavior for different scenarios (development, staging, production, etc.).

- **Environment Variable:**
    - ASPNETCORE_ENVIRONMENT is the key environment variable.
    - Its value determines the active environment.

- **Setting the Environment:**
    - **launchSettings.json (Development):** Set within the environmentVariables section of a profile.
    - **System Environment Variables:** Set directly on your machine (persistent).
    - **Command Line:** Use --environment or -e flag when running the app (e.g., dotnet run --environment Staging).
    - **Azure App Service:** In the Azure portal, under Configuration > Application settings.

- **IWebHostEnvironment Interface:**
    - Use it in your code to access environment information (e.g., EnvironmentName, WebRootPath).
    - Inject it into your controllers or middleware:

```
private readonly IWebHostEnvironment _env;


public MyController(IWebHostEnvironment env)
{
   _env = env;
}
```

- **Environment-Specific Configuration:**
    - Create files like appsettings.Development.json, appsettings.Staging.json, etc.
    - ASP.NET Core automatically loads the appropriate file based on the environment.
    - Override base settings in appsettings.json.

- **Conditional Configuration (In Program.cs):**
    - Use if (app.Environment.IsDevelopment()) or similar methods to apply settings or middleware based on the environment.

```
if (app.Environment.IsDevelopment())
{
   app.UseDeveloperExceptionPage();
}
```

- **Default Environments:**
    - Development: Default for local development.
    - Staging: Typically used for pre-production testing.
    - Production: The live environment.

- **Custom Environments:** You can define and use your own environment names.

- **Best Practices:**
    - **Separate Configurations:** Keep environment-specific settings in separate files.
    - **Tailor Middleware:** Use different middleware pipelines for different environments (e.g., enable DeveloperExceptionPage only in development).
    - **Logging:** Adjust logging levels based on the environment.
    - **Feature Flags:** Use environment variables to toggle features on/off.

Interview Tips

- **Explain the Why:** Be able to articulate the reasons for using environments (configuration, security, flexibility).
- **Configuration:** Show how you would use appsettings.{Environment}.json files to manage environment-specific settings.
- **Middleware:** Explain how you would customize middleware pipelines based on the environment.
- **Deployment:** Discuss how you would set the environment variable when deploying to different servers.