**.Net Core Documentation**

**.Net Core Overview**

* .NET Core is an open-source, cross-platform framework developed by Microsoft for building modern applications, including web, cloud, IoT, and mobile backends.
* It is a modular, lightweight, and high-performance successor to the .NET Framework.

**ASP.NET Core Project Structure**

**Program.cs**

* **Purpose: The entry point for the application. Configures the web host and logging services.**
* **Key Points:** 
  + **Calls Startup.cs to configure services and request handling.**
  + **Sets up the web server (e.g., Kestrel) and logging.**

**Startup.cs**

* **Purpose:** Configures application services and the middleware pipeline.
* **Key Points:**
  + **ConfigureServices**: Registers services like Dependency Injection (DI), authentication, database context, etc.
  + **Configure**: Defines the middleware pipeline (e.g., routing, CORS, error handling, etc.).

**launchSettings.json**

* **Purpose:** Defines environment-specific settings for local development.
* **Key Points:**
  + Configures different profiles for how the application will be run locally (e.g., with IIS Express or Kestrel).
  + Specifies environment variables, URLs, and application settings.

**appsettings.json**

* **Purpose:** Configures application-wide settings that are accessible through the IConfiguration interface.
* **Key Points:**
  + Stores settings such as database connection strings, application secrets, API keys, and other configuration values.
  + Supports hierarchical configuration values using JSON format, allowing for complex settings (e.g., connection strings, logging settings).
  + **Environment-Specific Configuration:** appsettings.{Environment}.json can be used to override settings for specific environments (like Development or Production).

**wwwroot Folder**

* **Purpose:** The wwwroot folder is the web root for static files in an ASP.NET Core application.
* **Key Points:**
  + Any static files such as images, CSS, JavaScript, and other assets are placed here.
  + Files placed in the wwwroot folder are publicly accessible via a URL (e.g., /images/logo.png).
  + The wwwroot folder is automatically served by the ASP.NET Core application, and its contents are accessible from the root of the web application.

**Middleware**

Middleware components are responsible for handling HTTP requests and responses. Middleware can perform tasks such as authentication, logging, error handling, static file serving, routing, etc. The **middleware pipeline** is a series of components that are invoked in a sequential order to handle a request.

The middleware pipeline is configured in the Configure method of the Startup.cs class. You add middleware components to the pipeline using methods such as UseMiddleware, UseStaticFiles, UseRouting, UseEndpoints, etc.

* **Error Handling**: UseDeveloperExceptionPage(), UseExceptionHandler()
* **Security**: UseHttpsRedirection(), UseHsts()
* **Static Files**: UseStaticFiles(), UseFileServer()
* **Authentication/Authorization**: UseAuthentication(), UseAuthorization()
* **Routing & Endpoints**: UseRouting(), UseEndpoints()
* **Caching & Performance**: UseResponseCaching(), UseCors()
* **State Management**: UseSession(), UseCookiePolicy()
* **Localization & Health**: UseRequestLocalization(), UseHealthChecks()

**Routing**

Routing in ASP.NET Core helps to map HTTP requests to specific endpoints (like controllers or Razor Pages) based on the URL pattern. The routing system in ASP.NET Core is flexible and supports both **conventional routing** (which uses route templates) and **attribute routing** (which uses annotations).

Routing is a core part of the ASP.NET Core **middleware pipeline** and typically requires the **UseRouting()** and **UseEndpoints()** middlewares to function correctly.

**Conventional Routing**

**Conventional routing** uses route templates defined globally (typically in Startup.cs) that map URLs to controller actions. It follows a predefined pattern, and the URL structure determines how requests are mapped to controllers and their actions.

**Conventional Routing Setup:**

1. **Route Template**: The route template defines the URL pattern used for routing. The most common format is {controller}/{action}/{id?}, where:
   * {controller} is the controller name.
   * {action} is the action method inside the controller.
   * {id?} is an optional parameter for passing values (e.g., an ID for a specific resource).
2. **Registering Conventional Routes**:
   * In the Configure method of Startup.cs, we use the UseRouting() and UseEndpoints() middlewares to configure conventional routes.

**Attribute Routing**

**Attribute routing** allows you to define routes directly on controller actions using attributes. It provides more flexibility, as you can specify custom URLs for each action or even parameterize the routes.

**Attribute Routing Setup:**

1. **Define Route on Actions**: You can decorate controller actions with Route attributes to specify the URL pattern for that action.
2. **Route Prefix**: You can define a route prefix on the controller level to avoid repeating part of the route on every action.
3. **Parameter Binding**: You can define route parameters, and ASP.NET Core will automatically bind those values to method parameters.
4. **Multiple Routes on a Single Action**: You can apply multiple routes to a single action by using multiple route attributes.

**Controller Initialization**

**Request Routing to a Controller in ASP.NET Core**

When a request is routed to a controller, the following steps occur:

1. **Controller Instantiation**
   * The framework creates an instance of the controller.
2. **Dependency Injection**
   * Any required dependencies are injected via **constructor injection**.
3. **Action Method Execution**
   * The appropriate action method is called based on the request route and parameters.
4. **Result Processing & Response**
   * The action method's result is processed and returned as an HTTP response.

**Dependency Injection**

Dependency Injection (DI) is a core design pattern used in ASP.NET Core to achieve **Inversion of Control (IoC)**, which is essential for creating maintainable and testable applications. DI allows services to be injected into components (like controllers or other services) rather than having those components create instances of their dependencies.

ASP.NET Core has a built-in **Inversion of Control (IoC)** container, and it automatically handles the DI for your application. Below is an overview of **Dependency Injection**, **the Built-in IoC Container**, and **Service Lifetime** in ASP.NET Core.

**Built-in IoC Container in ASP.NET Core**

ASP.NET Core provides a simple and powerful Inversion of Control (IoC) container that supports DI. This container is **configured in Startup.cs** during the application's setup phase.

**How to Register Services in DI Container:**

In the ConfigureServices method of Startup.cs, you register services using the IServiceCollection interface. The service is registered by specifying the interface (or base class) and the implementation (or derived class).

Services can be registered in three main lifetimes (Transient, Scoped, Singleton).

**Explanation of Service Lifetime**

* **Transient**:
  + A **new instance** of the service is created each time it is requested. This is suitable for lightweight, stateless services that don't hold data between requests.
  + **Example**: A service that processes calculations for every request or a service that generates a temporary report.
* **Scoped**:
  + A **single instance** of the service is created per **HTTP request** (or per operation if you're using the scope in a non-web context).
  + **Use Case**: When you want to have a unique instance of a service throughout a request, such as a database context (DbContext), but the instance should not persist between requests.
  + **Example**: A service that maintains state during a single user request (e.g., saving data to a database).
* **Singleton**:
  + A **single instance** of the service is created and shared throughout the entire application's lifetime.
  + **Use Case**: Services that are stateless and can be reused across multiple requests or components without having any side effects.
  + **Example**: Logging, caching services, or external API clients that do not require state management.

**Logging in ASP.Net Core**

ASP.NET Core provides a **logging framework** that helps you capture, store, and analyze log information. The built-in logging system can log to various destinations like the console, files, and external systems (e.g., databases, cloud-based log storage). It supports a range of log levels (e.g., Information, Debug, Error) and integrates with different log providers (e.g., Console, Debug, File, Serilog, NLog).

**Built-in logging providers:**

ASP.NET Core has built-in **log providers** that capture log output and send it to different destinations. Some of the built-in log providers include:

1. **Console**: Logs to the console output.
2. **Debug**: Logs to the debug output window in Visual Studio or other IDEs.
3. **EventSource**: Logs to Event Tracing for Windows (ETW).
4. **TraceSource**: Provides a custom source for logging (typically used in desktop applications).
5. **File** (via third-party providers): Logs to files, typically using external libraries like **Serilog**, **NLog**, or **log4net**..