ASP.Net Core 8+ Reference Docs

Main contents

[**1. HTTP [MVC vs API]** 3](#_Toc189428897)

[**2. Middleware [MVC vs API]** 21](#_Toc189428898)

[**3. Routing [MVC vs API]** 37](#_Toc189428899)

[**4. Controllers and IActionResult [MVC]** 59](#_Toc189428900)

[**5. Model Binding and Validations [MVC vs API]** 82](#_Toc189428901)

[**6. Razor Views [MVC]** 119](#_Toc189428902)

[**7. Layout Views [MVC]** 156](#_Toc189428903)

[**8. Partial Views [MVC]** 170](#_Toc189428904)

[**9. View Components [MVC]** 181](#_Toc189428905)

[**10. DI [MVC vs API]** 195](#_Toc189428906)

[**11. Environments [MVC vs API]** 223](#_Toc189428907)

[**12. Configuration [MVC vs API]** 239](#_Toc189428908)

[**13. xUnit [MVC vs API]** 270](#_Toc189428909)

[**14. CRUD Operations [MVC]** 270](#_Toc189428910)

[**15. Tag Helpers [MVC]** 270](#_Toc189428911)

[**16. Entity Framework core [MVC vs API]** 270](#_Toc189428912)

[**17. Advanced Unit Testing – Moq and Repository Pattern [MVC vs API]** 270](#_Toc189428913)

[**18. Logging and Serilog [MVC vs API]** 271](#_Toc189428914)

[**19. Filter [MVC vs API]** 271](#_Toc189428915)

[**20. Error handling [MVC vs API]** 271](#_Toc189428916)

[**21. SOLID Principles [MVC vs API]** 271](#_Toc189428917)

[**22. Clean Architecture [MVC vs API]** 271](#_Toc189428918)

[**23. Identity , Authorization, Security [MVC vs API]** 271](#_Toc189428919)

[**24. Asp.net Core Web API [API]** 271](#_Toc189428920)

[**25. Swagger and OpenAPI [API]** 271](#_Toc189428921)

[**26. Angular and CORS [API]** 271](#_Toc189428922)

[**27. JWT and Web API Authentication [API]** 271](#_Toc189428923)

[**28. Minimal API [API]** 271](#_Toc189428924)

# **1. HTTP [MVC vs API]**

**Overview:**

* HTTP (Hypertext Transfer Protocol) is a protocol used for transmitting hypertext (e.g., HTML) over the internet.
* It operates on a client-server model, where the client (usually a web browser) makes requests to a server, which then responds with the requested resources or error messages.
* **Stateless Protocol (Trạng thái)**: Each HTTP request is independent of others; the server does not retain information from previous requests.

**Request/Response Model:**

* **Client Request**: The client sends an HTTP request to the server.
* **Server Response**: The server processes the request and sends back an HTTP response.

HTTP Server

**Definition:**

* An HTTP server is software that handles HTTP requests from clients and serves back responses. It processes incoming requests, executes the necessary logic (e.g., accessing a database, generating HTML), and returns the appropriate response.

**Examples:**

* Apache HTTP Server, Nginx, Microsoft IIS, Kestrel (used with ASP.NET Core).

**Kestrel:**

* Kestrel is a cross-platform web server included with ASP.NET Core.
* It is lightweight, high-performance, and suitable for running both internal and public-facing web applications.

Request and Response Flow with Kestrel

1. **Client Sends Request:**
   * The client (e.g., web browser) sends an HTTP request to the server.
2. **Kestrel Receives Request:**
   * Kestrel receives the request and passes it through the ASP.NET Core middleware pipeline.
3. **Request Processing:**
   * Middleware components process the request and eventually pass it to the application’s request handling logic.
4. **Generate Response:**
   * The application generates an HTTP response and sends it back through the middleware pipeline.
5. **Kestrel Sends Response:**
   * Kestrel sends the HTTP response back to the client.

How Browsers Use HTTP

* Browsers use HTTP to request resources such as HTML documents, images, CSS files, and JavaScript files from servers.
* When a user enters a URL or clicks a link, the browser sends an HTTP request to the server, which then responds with the requested resource.

Observing HTTP Requests and Responses in Chrome Dev Tools

1. **Open Chrome Dev Tools:**
   * Press F12 or Ctrl+Shift+I (or Cmd+Option+I on Mac) to open Chrome Dev Tools.
2. **Navigate to the Network Tab:**
   * Click on the Network tab to view HTTP requests and responses.
3. **Inspect a Request:**
   * Click on any request in the list to see detailed information:
     + **Headers**: View request and response headers.
     + **Preview/Response**: View the response body.
     + **Timing**: See the timing details of the request.

HTTP Response Message Format

**Response Message Format:**

* **Start Line**: Contains the HTTP version, status code, and status message.
* **Headers**: Key-value pairs providing information about the response.
* **Body**: Optional, contains the actual data (e.g., HTML, JSON).

**Example:**

1. HTTP/1.1 200 OK
2. Content-Type: text/html
3. Content-Length: 137
5. <html>
6. <body>
7. <h1>Hello, World!</h1>
8. </body>
9. </html>

**Commonly Used Response Headers:**

* Content-Type: Specifies the media type of the resource.
* Content-Length: The size of the response body in bytes.
* Server: Provides information about the server handling the request.
* Set-Cookie: Sets cookies to be stored by the client.
* Cache-Control: Directives for caching mechanisms in both requests and responses.

**Default Response Headers in Kestrel**

* Content-Type: Typically defaults to text/html or application/json depending on the content being served.
* Server: Indicates the server software (e.g., Kestrel).
* Date: The date and time when the response was generated.

**HTTP Status Codes**

**Overview:**

* Status codes are issued by the server in response to the client's request to indicate the result of the request.
* Categories include:
  + **1xx Informational**: Request received, continuing process.
  + **2xx Success**: The request was successfully received, understood, and accepted.
  + **3xx Redirection**: Further action needs to be taken in order to complete the request.
  + **4xx Client Error**: The request contains bad syntax or cannot be fulfilled.
  + **5xx Server Error**: The server failed to fulfill an apparently valid request.

**Common Status Codes:**

* 200 OK: The request succeeded.
* 201 Created: The request succeeded and a new resource was created.
* 204 No Content: The server successfully processed the request, but is not returning any content.
* 400 Bad Request: The server could not understand the request due to invalid syntax.
* 401 Unauthorized: Authentication is required.
* 403 Forbidden: The client does not have access rights to the content.
* 404 Not Found: The server cannot find the requested resource.
* 500 Internal Server Error: The server encountered an unexpected condition.
* 502 Bad Gateway: The server was acting as a gateway or proxy and received an invalid response from the upstream server.
* 503 Service Unavailable: The server is not ready to handle the request.

Setting Status Codes and Response Headers in ASP.NET Core

**Example Code 1:**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
4. app.Run(async (HttpContext context) =>
5. {
6. context.Response.Headers["MyKey"] = "my value";
7. context.Response.Headers["Server"] = "My server";
8. context.Response.Headers["Content-Type"] = "text/html";
9. await context.Response.WriteAsync("<h1>Hello</h1>");
10. await context.Response.WriteAsync("<h2>World</h2>");
11. });
13. app.Run();

**Explanation:**

* context.Response.Headers["MyKey"] = "my value";: Adds a custom header to the response.
* context.Response.Headers["Server"] = "My server";: Modifies the Server header.
* context.Response.Headers["Content-Type"] = "text/html";: Sets the Content-Type header to text/html.
* await context.Response.WriteAsync("<h1>Hello</h1>");: Writes the first part of the response body.
* await context.Response.WriteAsync("<h2>World</h2>");: Writes the second part of the response body.

**Example Code 2:**

1. csharpCopy codevar builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
4. app.Run(async (HttpContext context) =>
5. {
6. if (1 == 1)
7. {
8. context.Response.StatusCode = 200;
9. }
10. else
11. {
12. context.Response.StatusCode = 400;
13. }
14. await context.Response.WriteAsync("Hello");
15. await context.Response.WriteAsync(" World");
16. });
18. app.Run();

**Explanation:**

* context.Response.StatusCode = 200;: Sets the status code to 200 OK.
* context.Response.StatusCode = 400;: Sets the status code to 400 Bad Request (this line won't be executed due to the condition).
* await context.Response.WriteAsync("Hello");: Writes the first part of the response body.
* await context.Response.WriteAsync(" World");: Writes the second part of the response body.

Summary

* **HTTP Protocol**: A fundamental protocol for web communication, following a request/response model and operating statelessly.
* **HTTP Server**: Software that processes HTTP requests and responses, such as Kestrel.
* **Request/Response Flow**: From client request to server response, involving middleware processing in Kestrel.
* **Browser Usage**: Browsers request resources via HTTP, which are then processed and rendered.
* **Dev Tools**: Chrome Dev Tools can inspect HTTP traffic in detail.
* **Message Format**: HTTP requests and responses consist of a start line, headers, and an optional body.
* **Headers**: Key-value pairs providing additional information about requests and responses.
* **Status Codes**: Indicate the result of HTTP requests, categorized into informational, success, redirection, client error, and server error codes.
* **Setting Status Codes and Headers**: ASP.NET Core allows customization of responses using code, enabling setting of status codes and headers as demonstrated.

HTTP Requests

In the world of web applications, an HTTP request is a client's way of saying, "Hey server, I need something." This "something" could be a web page, an image, data from a database, or the result of some server-side calculation. The client, typically a web browser, sends this request to the server, which processes it and returns a response.

**Anatomy of an HTTP Request**

An HTTP request consists of several parts:

1. **Start Line:** This is the first line of the request, and it contains three crucial pieces of information:
   * **Method:** This indicates the action the client wants the server to perform. Common methods include:
     + GET: Retrieve data from the server.
     + POST: Submit data to the server (e.g., form data).
     + PUT: Update an existing resource on the server.
     + DELETE: Remove a resource from the server.
   * **Request URI (Uniform Resource Identifier):** This is the path to the resource on the server that the client is requesting.
   * **HTTP Version:** This specifies the version of the HTTP protocol being used (e.g., HTTP/1.1 or HTTP/2).
2. **Headers:** These provide additional information about the request, such as:
   * User-Agent: The client's browser or application.
   * Accept: The types of content the client can understand (e.g., HTML, JSON).
   * Host: The domain name of the server.
   * Content-Type: The type of data being sent in the request body (if any).
   * Authorization: Credentials for authentication (if required).
3. **Empty Line:** This separates the headers from the body of the request.
4. **Body (Optional):** This part of the request contains data that the client is sending to the server. For example, a POST request might include form data or JSON data.

**Query Strings: Passing Parameters in URLs**

A query string is a way to pass parameters to a server within the URL itself. It starts with a question mark (?) and follows the path in the URL. Each parameter is a key-value pair, separated by an equals sign (=), and multiple parameters are separated by ampersands (&).

**Example:**

1. https://example.com/products?category=electronics&brand=apple

In this example, category=electronics and brand=apple are parameters being passed to the server.

**The Request Object in ASP.NET Core**

ASP.NET Core provides a HttpRequest object that gives you access to all the information within an incoming request. This object has properties like:

* Method: The HTTP method (GET, POST, etc.).
* Path: The URI path requested by the client.
* Query: A collection of query string parameters.
* Headers: A collection of request headers.
* Body: A stream representing the request body (if present).

**Code 1: Displaying Request Path and Method**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
4. app.Run(async (HttpContext context) =>
5. {
6. string path = context.Request.Path;
7. string method = context.Request.Method;
9. context.Response.Headers["Content-type"] = "text/html";
10. await context.Response.WriteAsync($"<p>{path}</p>");
11. await context.Response.WriteAsync($"<p>{method}</p>");
12. });
14. app.Run();

This code defines a simple middleware component (using app.Run) that:

1. Extracts the Path and Method from the Request object.
2. Sets the Content-type response header to text/html.
3. Writes the extracted path and method into the response body as HTML paragraphs.

**Code 2: Handling GET Requests with Query Parameters**

1. app.Run(async (HttpContext context) =>
2. {
3. context.Response.Headers["Content-type"] = "text/html";
4. if (context.Request.Method == "GET")
5. {
6. if (context.Request.Query.ContainsKey("id"))
7. {
8. string id = context.Request.Query["id"];
9. await context.Response.WriteAsync($"<p>{id}</p>");
10. }
11. }
12. });

This code focuses on GET requests:

1. It sets the Content-type response header.
2. It checks if the request method is GET.
3. If so, it checks if a query parameter named "id" exists.
4. If found, it extracts the value of the "id" parameter and displays it.

**Code 3: Extracting the User-Agent Header**

1. app.Run(async (HttpContext context) =>
2. {
3. context.Response.Headers["Content-type"] = "text/html";
4. if (context.Request.Headers.ContainsKey("User-Agent"))
5. {
6. string userAgent = context.Request.Headers["User-Agent"];
7. await context.Response.WriteAsync($"<p>{userAgent}</p>");
8. }
9. });

This code:

1. Sets the Content-type response header.
2. Checks if the User-Agent header is present in the request.
3. If found, it extracts the value of the User-Agent header and displays it, indicating the client's browser or application.

**Summary about HTTP Request:**

HTTP requests are the messages sent from clients (like web browsers) to servers to request resources or actions. They consist of a start line (method, URI, HTTP version), headers (additional information), an empty line, and an optional body containing data. Query strings are used to pass parameters within URLs.

ASP.NET Core provides the HttpRequest object to access request details. The example codes demonstrated:

1. Displaying the requested path and HTTP method.
2. Handling GET requests and extracting query parameter values.
3. Retrieving and displaying the User-Agent header from a request.

HTTP Methods

**GET: Retrieving Data**

The GET method is primarily designed for fetching data from a server. Think of it as asking the server for a specific resource, like a webpage, an image, or some data from a database. Here's what characterizes GET requests:

1. **Data in the URL:** Parameters are appended to the URL as a query string. This makes the request parameters visible in the browser's address bar.
2. **Limited Data Size:** The size of data that can be sent in a GET request is restricted due to limitations in URL lengths (browsers and servers might have different limits).
3. **Idempotent:** GET requests are considered idempotent. This means you can make the same GET request multiple times, and it should have the same effect as making it once (assuming the underlying data hasn't changed).
4. **Caching:** GET requests can be cached, meaning that if a client requests the same resource again, the browser might serve the previously retrieved response from its cache, improving performance.
5. **Security:** GET requests are generally less secure than POST requests because the data is visible in the URL. Avoid using GET for sensitive information like passwords or credit card numbers.

**Example GET Request:**

1. GET /products?category=electronics&brand=apple HTTP/1.1
2. Host: example.com

**POST: Submitting Data**

The POST method is primarily used for submitting data to the server for processing. This data is typically included in the body of the request and is not visible in the URL. Here's how POST requests differ from GET:

1. **Data in the Body:** Data is sent in the request body, making it more suitable for sending large amounts of data or sensitive information.
2. **Not Idempotent:** POST requests are not idempotent. Repeated POST requests might result in different outcomes (e.g., creating multiple resources or triggering actions multiple times).
3. **Not Cachable:** POST requests are generally not cached, as they often result in changes on the server.
4. **Security:** POST requests are considered more secure than GET requests because the data is not exposed in the URL. However, they are still susceptible to attacks like cross-site request forgery (CSRF), which requires additional security measures.

**Example POST Request:**

1. POST /login HTTP/1.1
2. Host: example.com
3. Content-Type: application/x-www-form-urlencoded
5. username=john&password=secret

**Choosing Between GET and POST**

* **Use GET when:**
  + You are retrieving data from the server.
  + You want the request to be bookmarkable.
  + The data being sent is small and non-sensitive.
* **Use POST when:**
  + You are submitting data to the server for processing.
  + The request might cause changes on the server.
  + You are sending sensitive data or large amounts of data.

Postman

Postman is a versatile API development and testing tool. It allows you to easily craft HTTP requests, send them to your ASP.NET Core application (or any API), and inspect the responses. It's a fantastic way to debug, experiment, and explore your API endpoints.

**Installation**

1. **Download:** Head to the official Postman website (<https://www.postman.com/downloads/>) and download the version suitable for your operating system (Windows, macOS, Linux).
2. **Install:** Follow the on-screen instructions to install Postman. The process is usually straightforward.

**Usage: Making Requests to Your ASP.NET Core App**

Let's say your ASP.NET Core application is running locally at https://localhost:7070 and has an endpoint /api/products. Here's how to use Postman:

1. **Launch Postman:** Open the Postman application.
2. **Create a New Request:**
   * Click on the "New" button in the top left corner.
   * Choose "Request" from the options.
3. **Set the Request Method and URL:**
   * In the request builder, select the appropriate HTTP method (GET, POST, PUT, DELETE, etc.) from the dropdown.
   * Enter the full URL of your ASP.NET Core endpoint (e.g., https://localhost:7070/api/products) in the address bar.
4. **(Optional) Add Headers:**
   * If your endpoint requires specific headers (like Content-Type), click on the "Headers" tab and add them as key-value pairs.
5. **(Optional) Add Request Body:**
   * If you are sending data with the request (e.g., JSON data for a POST request), click on the "Body" tab.
   * Choose the format (e.g., "raw" for JSON) and enter your data.
6. **Send the Request:**
   * Click the "Send" button.
7. **Inspect the Response:**
   * The response from your ASP.NET Core application will appear in the lower part of Postman. You'll see:
     + The status code (200 OK, 404 Not Found, etc.)
     + Response headers
     + The response body (if any)

Summary

**HTTP (Hypertext Transfer Protocol):**

* **Foundation of the Web:** HTTP is the protocol that powers the World Wide Web. It defines how clients (browsers, apps) and servers communicate.
* **Request-Response Cycle:** Communication follows a request-response model. The client sends a request, and the server sends back a response.
* **Stateless:** HTTP is stateless, meaning each request is independent. Servers don't inherently remember past interactions.
* **Methods:** HTTP methods define actions (GET, POST, PUT, DELETE, etc.).
* **Versions:** HTTP/1.1 and HTTP/2 are the most commonly used versions.

**HTTP Requests:**

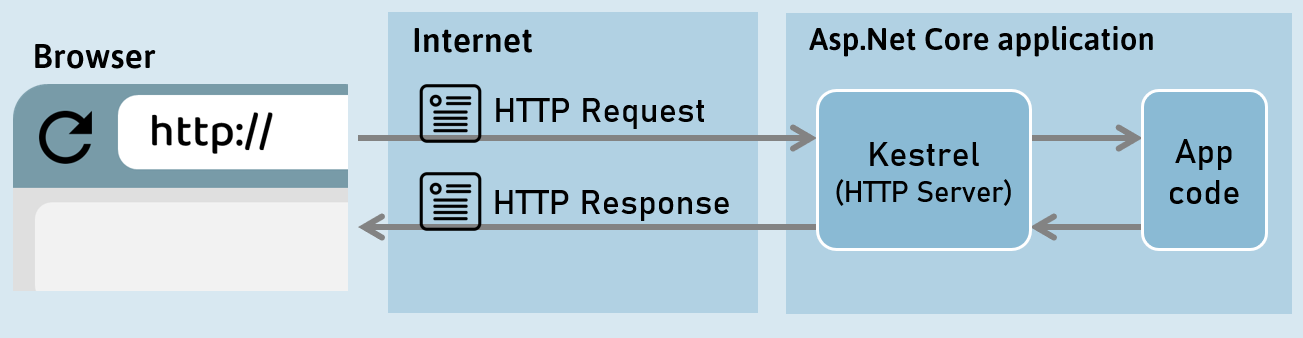
* **Purpose:** Initiate communication, asking for a resource or action from the server.
* **Structure:** Start line (method, URI, version), headers, empty line, optional body.
* **Methods:**
  + GET: Fetch data, idempotent, cachable.
  + POST: Submit data, not idempotent, not typically cached.
  + PUT, DELETE: Update and delete resources, respectively.
* **Headers:** Provide metadata like content type, user agent, authentication.
* **Body:** Used to send data with POST, PUT, etc.

**HTTP Responses:**

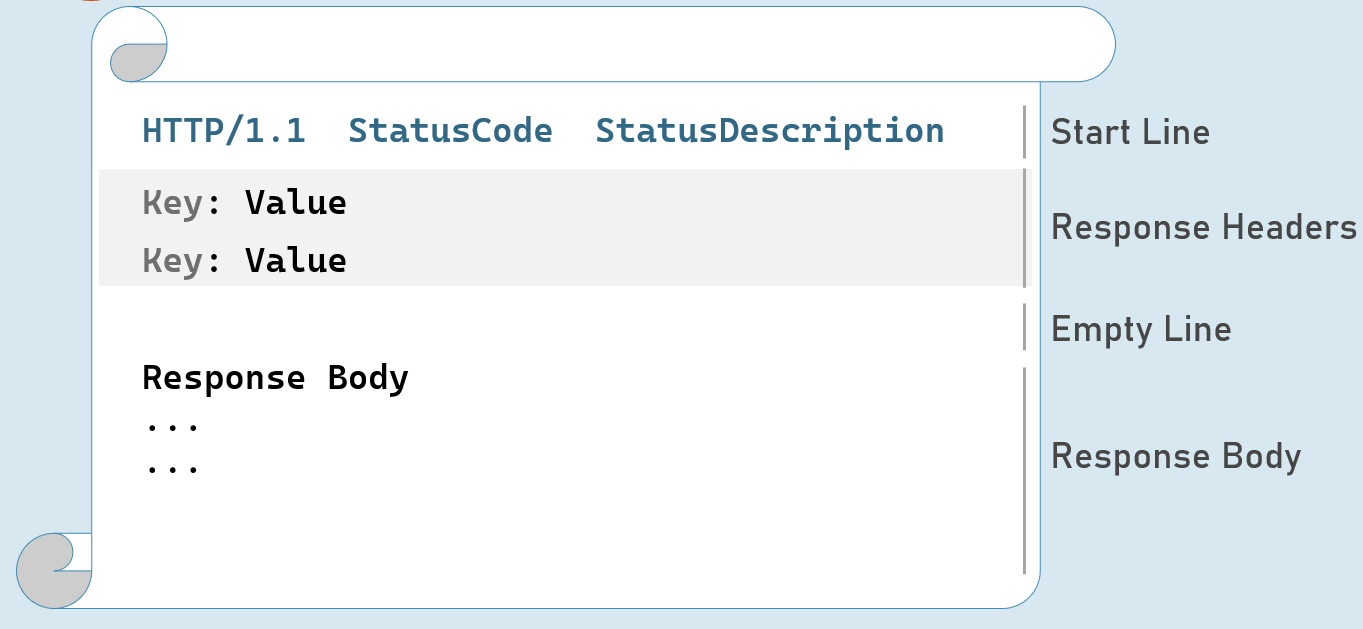
* **Purpose:** Server's reply to a request.
* **Structure:** Start line (version, status code, reason phrase), headers, empty line, optional body.
* **Status Codes:** Three-digit codes indicate the outcome (200 OK, 404 Not Found, 500 Internal Server Error).
* **Headers:** Provide metadata about the response (content type, length, caching).
* **Body:** Contains the requested data (HTML, JSON, etc.) or error messages.

HTTP is an application-protocol that defines set of rules to send request from browser to server and send response from server to browser.

Initially developed by Tim Berners Lee, later standardized by IETF (Internet Engineering Task Force) and W3C (World Wide Web Consortium)



HTTP Response



Response Start Line

Includes HTTP version, status code and status description.

**HTTP Version:** 1/1 | 2 | 3

**Status Code:** 101 | 200 | 302 | 400 | 401 | 404 | 500

**Status Description:**Switching Protocols | OK | Found | Bad Request | Unauthorized | Not Found | Internal Server Error

HTTP Response Status Codes

**1xx | Informational**

101           Switching Protocols

**2xx | Success**

200          OK

**3xx | Redirection**

302          Found

304          Not Modified

**4xx | Client error**

400           Bad Request

401            Unauthorized

404           Not Found

**5xx | Server error**

500           Internal Server Error

HTTP Response Headers

**Date**

Date and time of the response. e.g: Tue, 15 Nov 1994 08:12:31 GMT

**Server**

Name of the server.

e.g: Server=Kestrel

**Content-Type**

MIME type of response body.

e.g: text/plain, text/html, application/json, application/xml etc.

**Content-Length**

Length (bytes) of response body.

e.g: 100

**Cache-Control**

Indicates number of seconds that the response can be cached at the browser.

e.g: max-age=60

**Set-Cookie**

Contains cookies to send to browser.

e.g: x=10

**Access-Control-Allow-Origin**

Used to enable CORS (Cross-Origin-Resource-Sharing)

e.g: Access-Control-Allow-Origin: http://www.example.com

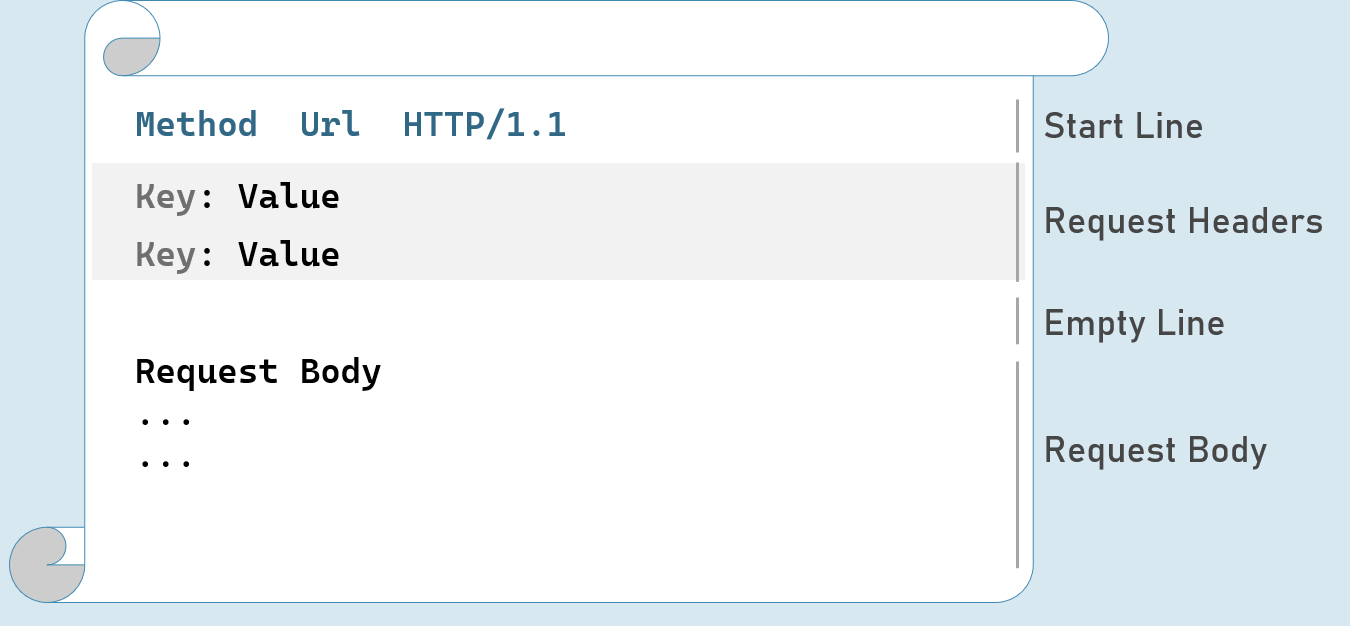
**Location**

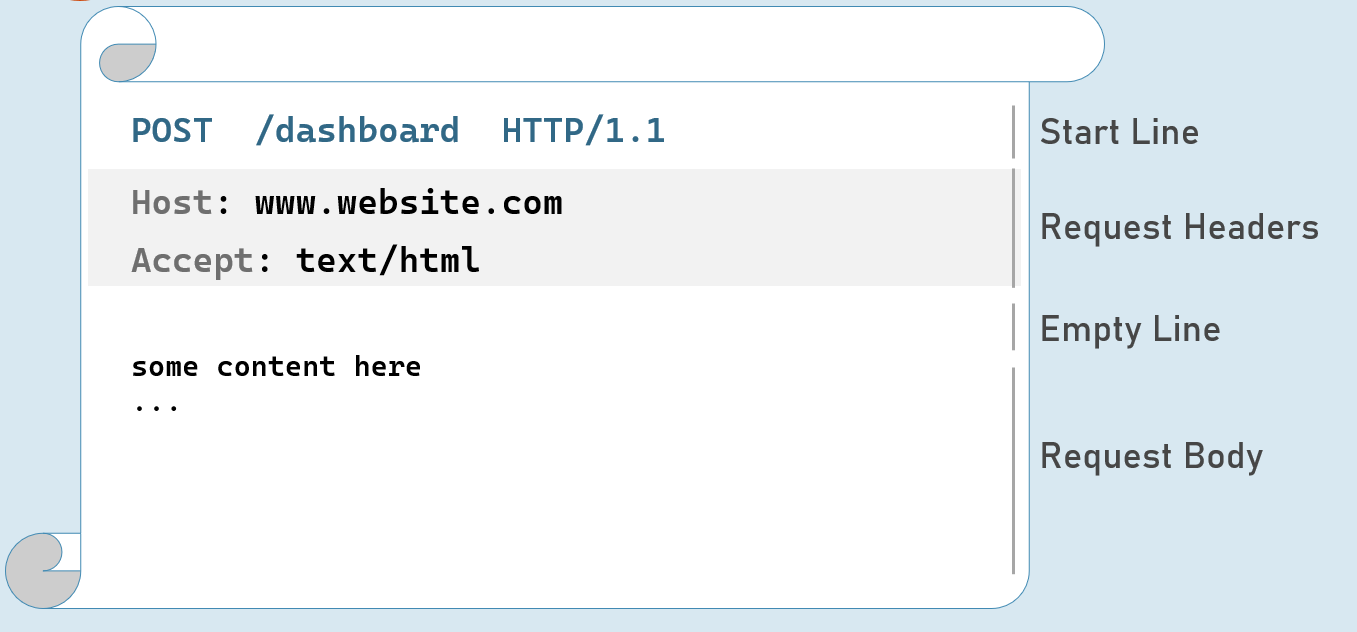
Contains url to redirect.

e.g: http://www.example-redirect.com

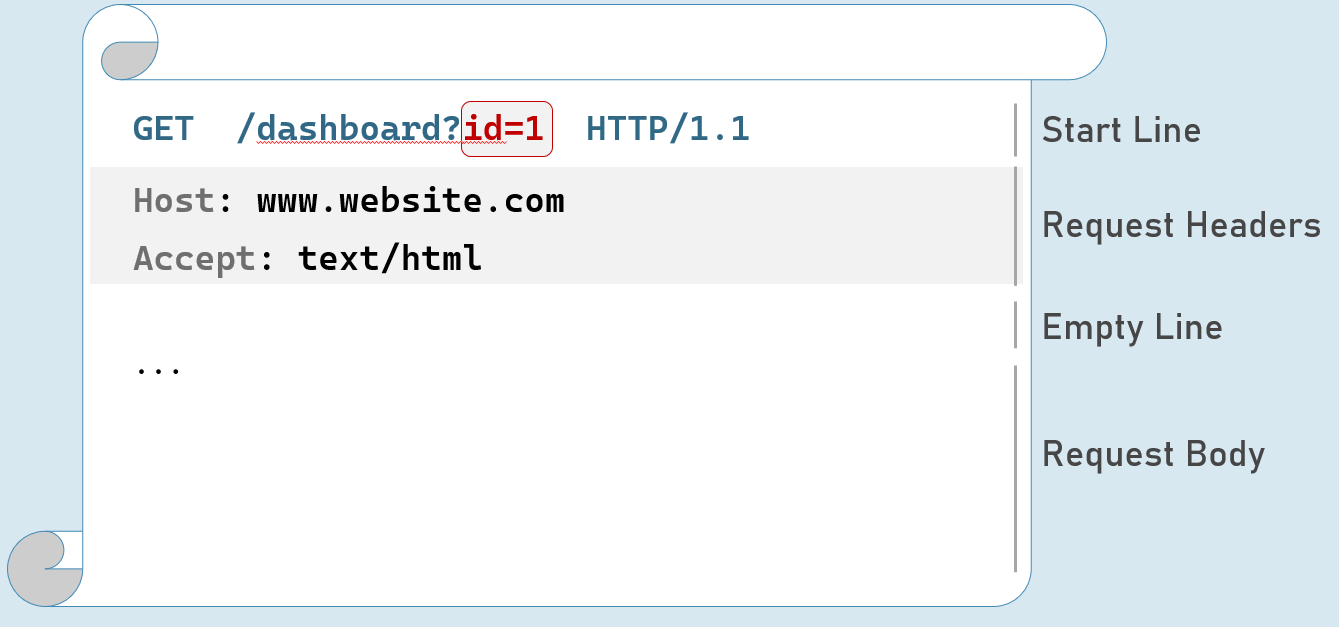
Further reading: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

HTTP Request





**HTTP Request - with Query String**



HTTP Request Headers

**Accept**

Represents MIME type of response content to be accepted by the client. e.g: text/html

**Accept-Language**

Represents natural language of response content to be accepted by the client. e.g: en-US

**Content-Type**

MIME type of request body.

Eg: text/x-www-form-urlencoded, application/json, application/xml, multipart/form-data

**Content-Length**

Length (bytes) of request body.

e.g: 100

**Date**

Date and time of request.

Eg: Tue, 15 Nov 1994 08:12:31 GMT

**Host**

Server domain name.

Eg: www.example.com

**User-Agent**

Browser (client) details.

Eg: Mozilla/5.0 Firefox/12.0

**Cookie**

Contains cookies to send to server.

Eg: x=100

Further reading: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

HTTP Request Methods

**GET**

Requests to retrieve information (page, entity object or a static file).

**Post**

Sends an entity object to server; generally, it will be inserted into the database.

**Put**

Sends an entity object to server; generally updates all properties (full-update) it in the database.

**Patch**

Sends an entity object to server; generally updates few properties (partial-update) it in the database.

**Delete**

Requests to delete an entity in the database.

HTTP Get [vs] Post

**Get:**

* Used to retrieve data from server.
* Parameters will be in the request url (as query string only).
* Can send limited number of characters only to server. Max: 2048 characters
* Used mostly as a default method of request for retrieving page, static files etc.
* Can be cached by browsers / search engines.

**Post:**

* Used to insert data into server
* Parameters will be in the request body (as query string, json, xml or form-data).
* Can send unlimited data to server.
* Mostly used for form submission / XHR calls
* Can't be cached by browsers / search engines.

# **2. Middleware [MVC vs API]**

At its core, middleware in ASP.NET Core is a series of components that form a pipeline through which every HTTP request and response flows. Each middleware component can:

1. **Examine** the incoming request.
2. **Modify** the request or response (if needed).
3. **Invoke** the next middleware in the pipeline or short-circuit the process and generate a response itself.

This pipeline allows you to modularize your application's logic and add features like authentication, logging, error handling, routing, and more in a clean and maintainable way.

**Middleware Chain (Request Pipeline)**

Imagine the request pipeline as a series of connected pipes. Each piece of middleware is like a valve in this pipeline, allowing you to control the flow of information and apply specific operations at different stages. The order you register your middleware matters, as they are executed sequentially.

**app.Use vs. app.Run**

These two methods are fundamental for adding middleware to your pipeline, but they have key differences:

* **app.Use(async (context, next) => { ... })**
  + **Non-Terminal Middleware:** This type of middleware typically performs some action and then calls the next delegate to pass control to the next middleware in the pipeline.
  + **Can Modify Request/Response:** It can change the request or response before passing it along.
  + **Examples:** Authentication, logging, custom headers, etc.
* **app.Run(async (context) => { ... })**
  + **Terminal Middleware:** This middleware doesn't call next; it ends the pipeline and generates the response itself.
  + **Often Used for the Final Response:** It's commonly used for handling requests that don't need further processing (e.g., returning a simple message).
  + **Can't Modify Request:** Since it's the end of the line, it cannot modify the request before passing it on.

**Code 1: The Consequence of Multiple app.Run Calls**

1. app.Run(async (HttpContext context) => {
2. await context.Response.WriteAsync("Hello");
3. });
5. app.Run(async (HttpContext context) => {
6. await context.Response.WriteAsync("Hello again");
7. });
9. app.Run();

In this code, only the first app.Run middleware will be executed. It terminates the pipeline by writing "Hello" to the response, and the subsequent app.Run (which would write "Hello again") never gets a chance to run.

**Code 2: Chaining Middleware with app.Use and app.Run**

1. //middlware 1
2. app.Use(async (context, next) => {
3. await context.Response.WriteAsync("Hello ");
4. await next(context);
5. });
7. //middleware 2
8. app.Use(async (context, next) => {
9. await context.Response.WriteAsync("Hello again ");
10. await next(context);
11. });
13. //middleware 3
14. app.Run(async (HttpContext context) => {
15. await context.Response.WriteAsync("Hello again");
16. });

This code demonstrates a correct way to chain middleware.

1. The first app.Use writes "Hello " to the response and then calls next to pass control to the next middleware.
2. The second app.Use writes "Hello again " and also calls next.
3. The final app.Run (which is terminal) writes "Hello again" and ends the pipeline. The result would be output of "Hello Hello again Hello again".

**Key Points to Remember**

* **Middleware Order is Crucial:** The order in which you register middleware matters, as they are executed in sequence.
* **Use app.Use for Non-Terminal Actions:** Use it for tasks like authentication, logging, or modifying headers/bodies.
* **Use app.Run to Terminate the Pipeline:** Employ it when you want to generate the final response.
* **Short-Circuiting:** Middleware can choose to short-circuit the pipeline (not call next) and return a response early if needed.

Custom Middleware in ASP.NET Cor

While ASP.NET Core provides a plethora of built-in middleware components, sometimes you need to create your own to address specific requirements unique to your application. Custom middleware allows you to:

* **Encapsulate logic:** Bundle related operations (e.g., logging, security checks, custom headers) into a reusable component.
* **Customize behavior:** Tailor the request/response pipeline to precisely match your application's needs.
* **Improve code organization:** Keep your middleware code clean and maintainable.

**Anatomy of a Custom Middleware Class**

1. **Implement IMiddleware:** This interface requires a single method: InvokeAsync(HttpContext context, RequestDelegate next). This is the heart of your middleware's logic.
2. **InvokeAsync or Invoke Method:**
   * context: The HttpContext provides access to the request and response objects.
   * next: The RequestDelegate allows you to call the next middleware in the pipeline.

**Code Explanation**

Let's dissect the code you provided:

1. // MyCustomMiddleware.cs
2. namespace MiddlewareExample.CustomMiddleware
3. {
4. public class MyCustomMiddleware : IMiddleware
5. {
6. public async Task InvokeAsync(HttpContext context, RequestDelegate next)
7. {
8. await context.Response.WriteAsync("My Custom Middleware - Starts\n");
9. await next(context);
10. await context.Response.WriteAsync("My Custom Middleware - Ends\n");
11. }
12. }
14. // Extension method for easy registration
15. public static class CustomMiddlewareExtension
16. {
17. public static IApplicationBuilder UseMyCustomMiddleware(this IApplicationBuilder app)
18. {
19. return app.UseMiddleware<MyCustomMiddleware>();
20. }
21. }
22. }

* **MyCustomMiddleware Class:** This class implements IMiddleware. Its InvokeAsync method:
  + Writes "My Custom Middleware - Starts" to the response.
  + Calls next(context) to invoke the next middleware in the pipeline.
  + Writes "My Custom Middleware - Ends" to the response after the next middleware has finished.
* **CustomMiddlewareExtension Class:** This provides a convenient extension method UseMyCustomMiddleware to register your middleware in the Startup.Configure method.

1. // Program.cs (or Startup.cs)
2. using MiddlewareExample.CustomMiddleware;
4. // ...
6. builder.Services.AddTransient<MyCustomMiddleware>(); // Register as transient
8. app.Use(async (HttpContext context, RequestDelegate next) => {
9. await context.Response.WriteAsync("From Midleware 1\n");
10. await next(context);
11. });
13. app.UseMyCustomMiddleware(); // Use the extension method
15. app.Run(async (HttpContext context) => {
16. await context.Response.WriteAsync("From Middleware 3\n");
17. });

**How It Works**

1. **Registration:** You register MyCustomMiddleware as a transient service so that ASP.NET Core can create instances of it when needed.
2. **Pipeline Integration:** The app.UseMyCustomMiddleware() extension method seamlessly adds your middleware to the pipeline.
3. **Execution Order:** Middleware components are executed in the order they are added to the pipeline. In this case, the order would be Middleware 1, MyCustomMiddleware, then Middleware 3.

**Output**

When you run the application, you'll see the following output in your browser:

1. From Midleware 1
2. My Custom Middleware - Starts
3. From Middleware 3
4. My Custom Middleware - Ends

This clearly demonstrates the flow of execution through the middleware chain.

Custom Conventional Middleware

ASP.NET Core middleware comes in two flavors: conventional and factory-based. Conventional middleware, as shown in your example, is a simple yet powerful way to encapsulate custom logic for processing HTTP requests and responses.

**Key Characteristics**

* **Class-Based:** Conventional middleware is implemented as a class.
* **Constructor Injection:** It receives dependencies (if any) through its constructor.
* **Invoke Method:** This is the heart of the middleware, containing the logic that handles each request.
* **RequestDelegate:** The Invoke method takes a RequestDelegate parameter (\_next in your example). This delegate represents the next middleware in the pipeline.
* **Flexibility:** You have full control over the request and response objects within the Invoke method.

**Code Breakdown: HelloCustomMiddleware**

1. // HelloCustomMiddleware.cs
2. public class HelloCustomMiddleware
3. {
4. private readonly RequestDelegate \_next;
5. public HelloCustomMiddleware(RequestDelegate next)
6. {
7. \_next = next;
8. }
10. public async Task Invoke(HttpContext httpContext)
11. {
12. if (httpContext.Request.Query.ContainsKey("firstname") &&
13. httpContext.Request.Query.ContainsKey("lastname"))
14. {
15. string fullName = httpContext.Request.Query["firstname"] + " " + httpContext.Request.Query["lastname"];
16. await httpContext.Response.WriteAsync(fullName);
17. }
18. await \_next(httpContext);
19. }
20. }
22. // Extension method for easy registration
23. public static class HelloCustomModdleExtensions
24. {
25. public static IApplicationBuilder UseHelloCustomMiddleware(this IApplicationBuilder builder)
26. {
27. return builder.UseMiddleware<HelloCustomMiddleware>();
28. }
29. }

Let's analyze each part:

1. **Constructor:** The constructor receives the RequestDelegate, which is stored for later use to invoke the next middleware in the pipeline.
2. **Invoke Method:**
   * It checks if the query string contains both "firstname" and "lastname" parameters.
   * If so, it combines the values into a fullName string and writes it to the response.
   * **Crucially:** It calls await \_next(httpContext); to continue the middleware chain. This line ensures that the request is passed on to subsequent middleware components, even if a full name is generated.
   * By design, any code after this line, such as the comment "//after logic", would not execute for requests containing both "firstname" and "lastname", as the await \_next(httpContext); line immediately transfers control to the next middleware in the pipeline.
3. **UseHelloCustomMiddleware Extension:** This extension method simplifies the registration process by hiding the details of instantiating and using your custom middleware class.

**Program.cs (or Startup.cs): Using the Middleware**

1. // ... other middleware ...
2. app.UseMyCustomMiddleware();
3. app.UseHelloCustomMiddleware();
4. // ...

**How It Works**

1. When a request arrives, ASP.NET Core traverses the middleware pipeline.
2. It reaches HelloCustomMiddleware, which checks for the specific query parameters.
3. If the parameters are present, the middleware generates a personalized greeting.
4. Regardless of whether it generates the greeting, the middleware calls next(context) to pass the request along to the next middleware component in the pipeline.

**Key Points**

* **Simplicity:** Conventional middleware is easy to write and understand.
* **Control:** You have fine-grained control over how the request is processed and how the response is generated.
* **Extension Methods:** Use extension methods to make middleware registration clean and readable.

The Ideal Order of Middleware Pipeline

1. **Exception/Error Handling:**
   * **Purpose:** Catches and handles exceptions that occur anywhere in the pipeline.
   * **Examples:** UseExceptionHandler, UseDeveloperExceptionPage (for development environments).
2. **HTTPS Redirection:**
   * **Purpose:** Redirects HTTP requests to HTTPS for security.
   * **Example:** UseHttpsRedirection.
3. **Static Files:**
   * **Purpose:** Serves static files like images, CSS, and JavaScript directly to the client.
   * **Example:** UseStaticFiles.
4. **Routing:**
   * **Purpose:** Matches incoming requests to specific endpoints based on their URLs.
   * **Examples:** UseRouting, UseEndpoints.
5. **CORS (Cross-Origin Resource Sharing):**
   * **Purpose:** Enables secure cross-origin requests from different domains.
   * **Example:** UseCors.
6. **Authentication:**
   * **Purpose:** Verifies user identities and establishes a user principal.
   * **Example:** UseAuthentication.
7. **Authorization:**
   * **Purpose:** Determines whether a user is allowed to access a particular resource or perform a certain action.
   * **Example:** UseAuthorization.
8. **Custom Middleware:**
   * **Purpose:** Your application-specific middleware components to handle tasks like logging, feature flags, etc.

**Reasoning Behind the Order**

* **Early Exception Handling:** Catching exceptions early prevents them from propagating and causing further issues down the pipeline.
* **Security First:** HTTPS redirection, authentication, and authorization are essential for securing your application.
* **Performance Optimization:** Static files, response caching, and compression are placed early to optimize the response generation process.
* **Routing as a Foundation:** Routing determines how requests are handled by your application's core logic.
* **CORS for Flexibility:** CORS allows your application to be consumed by a wider range of clients.
* **Custom Middleware:** Your custom middleware can be placed strategically within the pipeline to apply logic at the appropriate stage.

**Flexibility and Exceptions**

While this is the general recommended order, there might be exceptions based on your application's specific needs. For instance:

* **Health Checks:** You might want to place health check middleware very early in the pipeline to quickly determine the application's status without executing other middleware components.
* **Specialized Middleware:** Some middleware components may have specific ordering requirements documented by their providers.

**Example (Program.cs or Startup.cs):**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
4. if (app.Environment.IsDevelopment())
5. {
6. app.UseDeveloperExceptionPage();
7. }
9. app.UseHttpsRedirection();
10. app.UseStaticFiles();
11. app.UseRouting();
12. app.UseAuthentication();
13. app.UseAuthorization();
15. // ... your custom middleware ...
17. app.UseEndpoints(endpoints =>
18. {
19. endpoints.MapControllers(); // Or MapRazorPages(), MapGet(), etc.
20. });

By adhering to this recommended order, you'll create a well-structured and efficient ASP.NET Core application that's easier to maintain, debug, and secure.

UseWhen()

UseWhen() is a powerful extension method in ASP.NET Core's IApplicationBuilder interface. It allows you to conditionally add middleware to your request pipeline based on a predicate (a condition). This means you can create dynamic pipelines where specific middleware components are executed only when certain conditions are met.

**Syntax**

1. app.UseWhen(
2. context => /\* Your condition here \*/,
3. app => /\* Middleware configuration for the branch \*/
4. );

* **context:** The HttpContext object representing the current request.
* **Predicate (Condition):** A function that takes the HttpContext and returns true if the middleware branch should be executed, false otherwise.
* **Middleware Configuration:** An action that configures the middleware components that should be executed if the condition is true. This is where you use app.Use(), app.Run(), or other middleware registration methods.

**How UseWhen() Works**

1. **Predicate Evaluation:** When a request comes in, the UseWhen() method first evaluates the predicate function against the HttpContext.
2. **Branching (if true):** If the predicate returns true, the middleware branch specified in the configuration action is executed. The request flows through this branch, potentially undergoing modifications or generating a response.
3. **Rejoining the Main Pipeline:** After the branch is executed (or skipped if the predicate was false), the request flow rejoins the main pipeline, continuing with the next middleware components registered after the UseWhen() call.

**Code Example: Explained**

1. app.UseWhen(
2. context => context.Request.Query.ContainsKey("username"),
3. app => {
4. app.Use(async (context, next) =>
5. {
6. await context.Response.WriteAsync("Hello from Middleware branch");
7. await next();
8. });
9. });
11. app.Run(async context =>
12. {
13. await context.Response.WriteAsync("Hello from middleware at main chain");
14. });

* **Condition:** The predicate context.Request.Query.ContainsKey("username") checks if the query string contains a parameter named "username".
* **Branch Middleware:** If the "username" parameter is present, the branch middleware is executed. It writes "Hello from Middleware branch" to the response and then calls next to allow the rest of the pipeline to continue.
* **Main Pipeline:** The final app.Run middleware is part of the main pipeline. It writes "Hello from middleware at main chain" to the response.

**Output**

* If the request contains the "username" query parameter (e.g., /path?username=John), the output will be:
  1. Hello from Middleware branch
  2. Hello from middleware at main chain
* If the request does not contain the "username" parameter (e.g., /path), the output will be:
  1. Hello from middleware at main chain

**When to Use UseWhen()**

* **Conditional Features:** Enable or disable certain features based on the request (e.g., logging only for certain users, applying caching rules based on query parameters).
* **Dynamic Pipelines:** Create pipelines that adapt to different requests (e.g., different authentication middleware for specific routes).
* **A/B Testing:** Route a subset of users through alternative middleware branches for experimentation.
* **Debugging and Diagnostics:** Apply diagnostic middleware only in development environments.

Key Points to Remember:

**Conceptual Understanding:**

1. **The Pipeline:** Middleware forms a pipeline for HTTP requests and responses. Each component can inspect, modify, or terminate the flow.
2. **Order Matters:** Middleware is executed in the order it's registered. Think carefully about the sequence.
3. **Types of Middleware:**
   * **Built-in:** ASP.NET Core offers middleware for authentication, routing, static files, etc.
   * **Custom:** You can create your own to add specific logic to your app.

**app.Use vs. app.Run:**

1. **app.Use:** For non-terminal middleware. It calls next to pass control to the next component.
2. **app.Run:** For terminal middleware. It ends the pipeline and generates a response.

**Custom Middleware:**

1. **Two Ways:**
   * **Conventional:** Class-based, using the Invoke method and constructor injection.
   * **Factory-Based:** Uses a delegate to create the middleware instance.
2. **Benefits:** Encapsulates logic, improves code organization, and allows you to tailor your application's behavior.

**Recommended Order:** (Not strict, but a good guideline)

1. Exception Handling
2. HTTPS Redirection
3. Static Files
4. Routing
5. CORS
6. Authentication
7. Authorization
8. Custom Middleware
9. MVC/Razor Pages/Minimal APIs

**Bonus Points:**

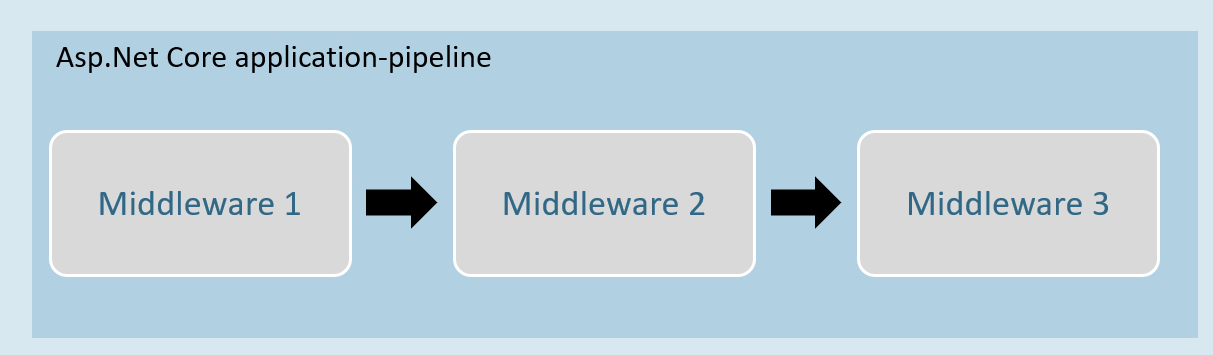
* **Short-Circuiting:** Middleware can choose not to call next and return a response early.
* **UseWhen:** Conditionally add middleware branches based on request criteria.
* **Middleware Ordering Flexibility:** Understand the reasons behind the recommended order, but also know when to deviate from it based on your application's specific requirements.

Introduction to Middleware

Middleware is a component that is assembled into the application pipeline to handle requests and responses.

Middlewares are chained one-after-other and execute in the same sequence how they're added.





Middleware can be a request delegate (anonymous method or lambda expression) [or] a class.

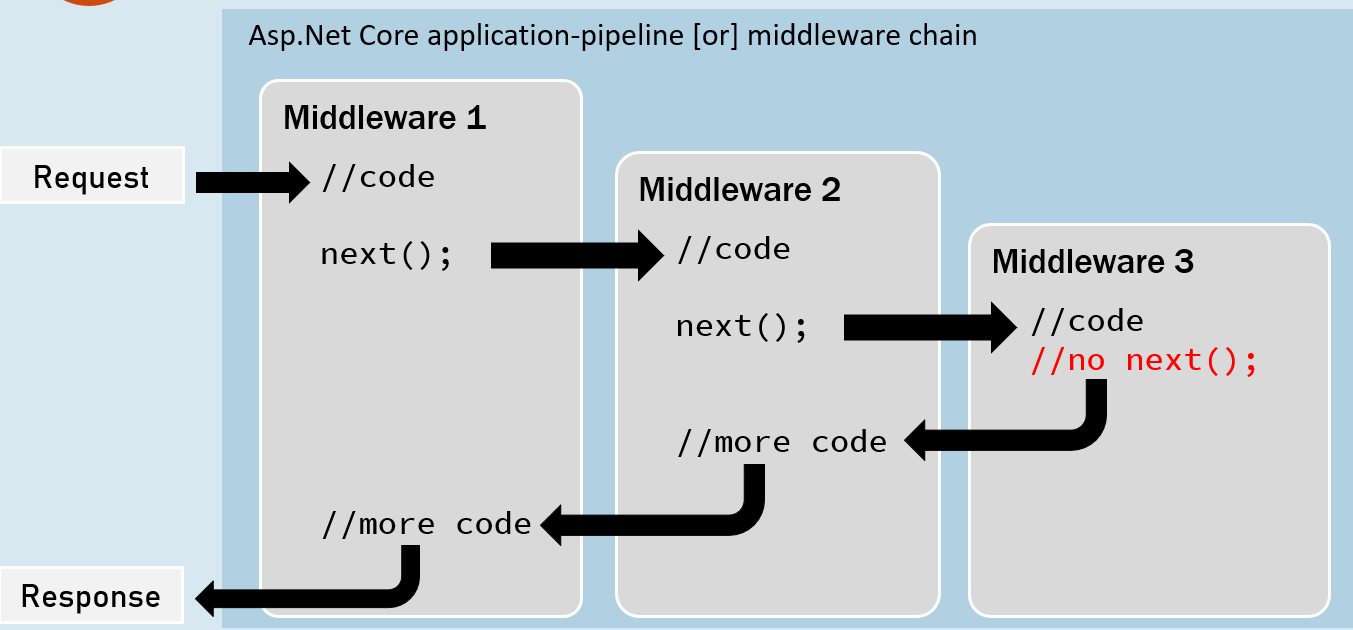
Middleware - Run

**app.Run( )**

1. app.Run(async (HttpContext context) =>
2. {
3. //code
4. });

The extension method called “Run” is used to execute a terminating / short-circuiting middleware that doesn’t forward the request to the next middleware.

Middleware Chain



**app.Use( )**

1. app.Use(async (HttpContext context, RequestDelegate next) =>
2. {
3. //before logic
4. await next(context);
5. //after logic
6. });

The extension method called “Use” is used to execute a non-terminating / short-circuiting middleware that may / may not forward the request to the next middleware.

Middleware Class

Middleware class is used to separate the middleware logic from a lambda expression to a separate / reusable class.

1. class MiddlewareClassName : IMiddleware
2. {
3. public async Task InvokeAsync(HttpContext context, RequestDelegate next)
4. {
5. //before logic
6. await next(context);
7. //after logic
8. }
9. }

app.UseMiddleware<MiddlewareClassName>();

Middleware Extensions

1. class MiddlewareClassName : IMiddleware
2. {
3. public async Task InvokeAsync(HttpContext context,RequestDelegate next)
4. {
5. //before logic
6. await next(context);
7. //after logic
8. }
9. });

Middleware extension method is used to invoke the middleware with a single method call.

1. static class ClassName
2. {
3. public static IApplicationBuilder ExtensionMethodName(this IApplicationBuilder app)
4. {
5. return app.UseMiddleware<MiddlewareClassName>();
6. }
7. }

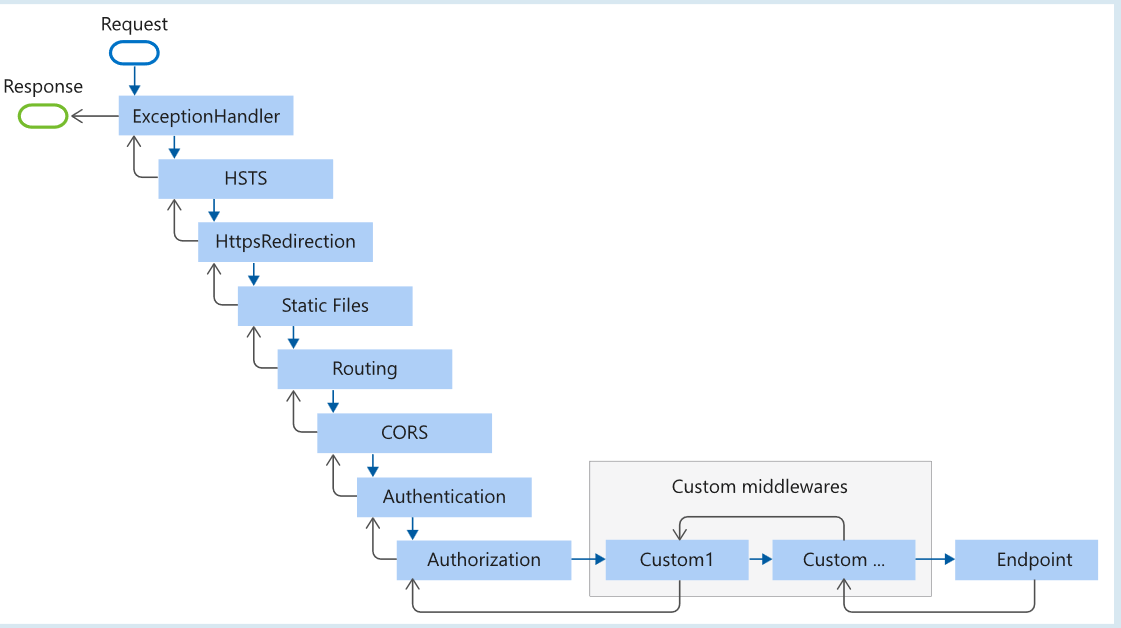
app.ExtensionMethodName();

Conventional Middleware

1. class MiddlewareClassName
2. {
3. private readonly RequestDelegate \_next;
5. public MiddlewareClassName(RequestDelegate next)
6. {
7. \_next = next;
8. }
10. public async Task InvokeAsync(HttpContext context)
11. {
12. //before logic
13. await \_next(context);
14. //after logic
15. }
16. });
17. static class ClassName
18. {
19. public static IApplicationBuilder ExtensionMethodName(this IApplicationBuilder app)
20. {
21. return app.UseMiddleware<MiddlewareClassName>();
22. }
23. }

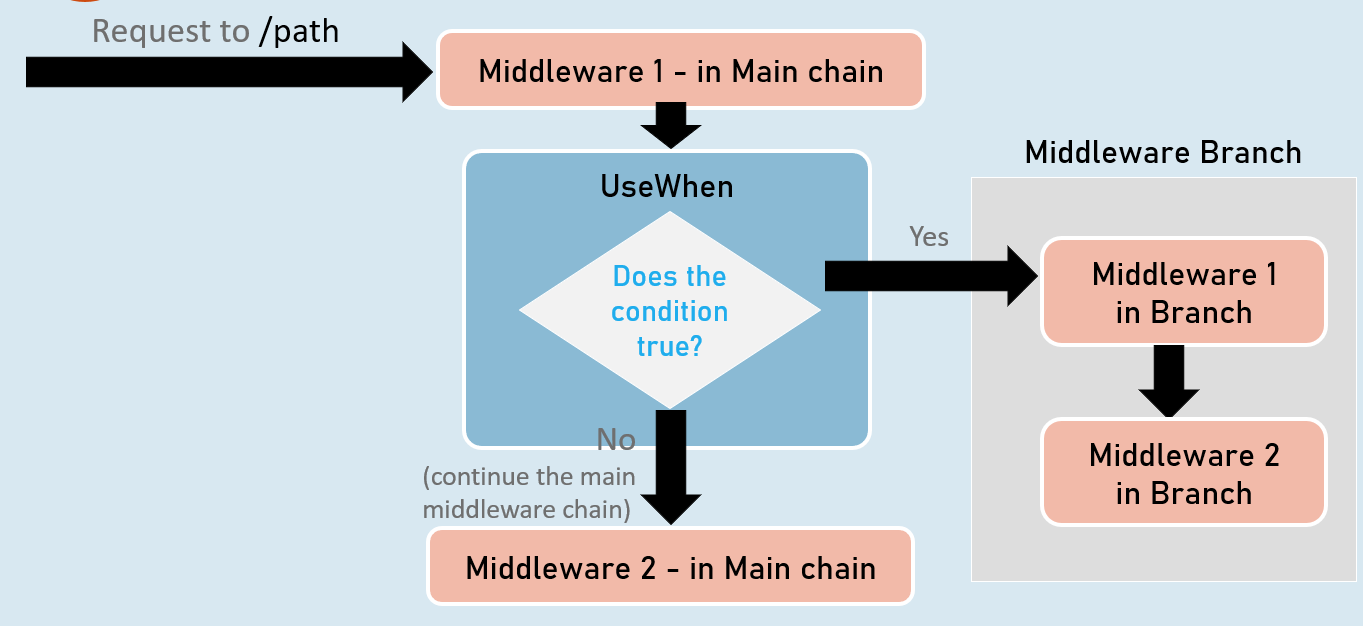
app.ExtensionMethodName();

The Right Order of Middleware



1. app.UseExceptionHandler("/Error");
2. app.UseHsts();
3. app.UseHttpsRedirection();
4. app.UseStaticFiles();
5. app.UseRouting();
6. app.UseCors();
7. app.UseAuthentication();
8. app.UseAuthorization();
9. app.UseSession();
10. app.MapControllers();
11. //add your custom middlewares
12. app.Run();

Middleware - UseWhen



**app.UseWhen( )**

1. app.UseWhen(
2. context => { return boolean; },
3. app =>
4. {
5. //add your middlewares
6. }
7. );

The extension method called “UseWhen” is used to execute a branch of middleware only when the specified condition is true.

# **3. Routing [MVC vs API]**

Routing

At its heart, routing is the mechanism that ASP.NET Core uses to match incoming HTTP requests to specific endpoints (e.g., controller actions, Razor Pages, or minimal API handlers) within your application. This allows you to define clean and meaningful URLs that clearly indicate the resources or actions being requested.

**How Routing Works in ASP.NET Core**

1. **Endpoint Registration:** You define endpoints (routes) within your application, specifying:
   * The URL pattern (e.g., /products, /api/orders/{id}).
   * The HTTP method(s) the endpoint handles (GET, POST, etc.).
   * The code to execute when the endpoint is matched (RequestDelegate).
2. **Request Matching (Middleware):**
   * The UseRouting middleware component is added to the pipeline.
   * When a request arrives, UseRouting analyzes the incoming URL and HTTP method.
   * It compares the URL against your registered endpoints to find the best match.
3. **Endpoint Execution (Middleware):**
   * The UseEndpoints middleware component is added to the pipeline, following UseRouting.
   * If UseRouting found a matching endpoint, UseEndpoints executes the code (the RequestDelegate) associated with that endpoint.

UseRouting vs. UseEndpoints

* **UseRouting:**
  + It's responsible for **route matching** - finding the right endpoint for a given request.
  + It adds route data to the HttpContext, which subsequent middleware can use to make decisions.
  + It **must** come before UseEndpoints.
* **UseEndpoints:**
  + It's responsible for **endpoint execution** - invoking the code (the delegate) associated with the matched endpoint.
  + It also lets you configure the endpoints (e.g., define policies, filters) using lambda expressions.

Map\* Methods: Creating Endpoints

ASP.NET Core provides a family of Map\* extension methods on the IEndpointRouteBuilder interface that simplify endpoint creation:

* MapGet: Creates an endpoint that only handles GET requests.
* MapPost: Creates an endpoint that only handles POST requests.
* MapPut, MapDelete: Create endpoints for PUT and DELETE requests, respectively.
* MapMethods: Creates an endpoint that handles multiple HTTP methods.
* MapControllerRoute, MapAreaControllerRoute: Used for configuring MVC/Razor Pages controllers.
* MapFallbackToFile: Used to specify a default file to serve when no other endpoint matches.

**Code: Detailed Explanation**

1. //enable routing
2. app.UseRouting();
4. //creating endpoints
5. app.UseEndpoints(endpoints =>
6. {
7. //add your endpoints here
8. endpoints.MapGet("map1", async (context) => {
9. await context.Response.WriteAsync("In Map 1");
10. });
12. endpoints.MapPost("map2", async (context) => {
13. await context.Response.WriteAsync("In Map 2");
14. });
15. });
17. app.Run(async context => {
18. await context.Response.WriteAsync($"Request received at {context.Request.Path}");
19. });
20. **app.UseRouting();:** This line activates routing middleware. It sets up the machinery to analyze incoming requests and match them against your defined endpoints.
21. **app.UseEndpoints(endpoints => { ... });:** This lambda expression configures the endpoints of your application:
    * endpoints.MapGet("map1", ...);: Registers a GET endpoint that responds to the path "/map1" with the text "In Map 1".
    * endpoints.MapPost("map2", ...);: Registers a POST endpoint for the path "/map2", responding with "In Map 2".
22. **app.Run(async context => { ... });:** This is a fallback terminal middleware. If no other endpoint matches the request (e.g., if you visit "/map3"), it will execute this code, writing the requested path to the response.

GetEndpoint()

In ASP.NET Core, the GetEndpoint() method is a powerful tool for retrieving information about the specific endpoint that was selected to handle an incoming HTTP request. This method is an extension method available on the HttpContext object.

* **Purpose:** It allows you to access details about the matched endpoint, such as its display name, route pattern, metadata, and more.
* **When to Use It:** You typically use GetEndpoint() within middleware components to make decisions based on the selected endpoint or to extract information that's relevant to your custom logic.
* **Middleware Placement:** The GetEndpoint() method will return a valid Endpoint object **only after** the UseRouting middleware has executed and successfully matched the request to an endpoint.

Code:

1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
4. // Middleware 1: Before Routing
5. app.Use(async (context, next) =>
6. {
7. Microsoft.AspNetCore.Http.Endpoint? endPoint = context.GetEndpoint();
8. if (endPoint != null)
9. {
10. await context.Response.WriteAsync($"Endpoint: {endPoint.DisplayName}\n");
11. }
12. await next(context);
13. });
15. // Enable Routing Middleware
16. app.UseRouting();
18. // Middleware 2: After Routing
19. app.Use(async (context, next) =>
20. {
21. Microsoft.AspNetCore.Http.Endpoint? endPoint = context.GetEndpoint();
22. if (endPoint != null)
23. {
24. await context.Response.WriteAsync($"Endpoint: {endPoint.DisplayName}\n");
25. }
26. await next(context);
27. });
29. // Creating Endpoints
30. app.UseEndpoints(endpoints =>
31. {
32. endpoints.MapGet("map1", async (context) =>
33. {
34. await context.Response.WriteAsync("In Map 1");
35. });
37. endpoints.MapPost("map2", async (context) =>
38. {
39. await context.Response.WriteAsync("In Map 2");
40. });
41. });
43. // Fallback Middleware
44. app.Run(async context =>
45. {
46. await context.Response.WriteAsync($"Request received at {context.Request.Path}");
47. });
49. app.Run();

Let's analyze the code step-by-step:h

1. **Middleware 1 (Before Routing):**
   * Here, GetEndpoint() will return null because routing hasn't happened yet. The request hasn't been matched to any specific endpoint.
2. **app.UseRouting();**
   * This enables the routing middleware, which is responsible for matching the request to an endpoint.
3. **Middleware 2 (After Routing):**
   * Now, GetEndpoint() will return the matched endpoint object (if a match was found). You can access its DisplayName (or other properties) to get information about the selected endpoint.

* For a GET request to "/map1", the display name would be "map1".
* For a POST request to "/map2", the display name would be "map2".
* For any other path, the display name would be null (since the fallback middleware handles those cases).

1. **Endpoint Creation:**
   * The app.UseEndpoints section defines your endpoints (routes).
2. **Fallback Middleware:**
   * This middleware handles requests that didn't match any defined endpoints. It simply writes the requested path to the response.

Route Parameters

Route parameters are placeholders within your URL patterns that capture values from incoming requests. These values can then be used within your endpoint handlers to customize the response or perform specific actions.

**Types of Route Parameters**

1. **Required Parameters:**
   * **Syntax:** Enclosed in curly braces {}.
   * **Behavior:** Must be provided in the URL for the route to match. If not present, the request won't match this endpoint.
   * **Example:** /products/{id} (The id parameter is required).
2. **Optional Parameters:**
   * **Syntax:** Enclosed in curly braces {} and followed by a question mark ?.
   * **Behavior:** Can be omitted from the URL. If not present, the parameter's value will be null.
   * **Example:** /products/details/{id?} (The id parameter is optional).
3. **Parameters with Default Values:**
   * **Syntax:** Enclosed in curly braces {}, followed by an equals sign =, and then the default value.
   * **Behavior:** If not provided in the URL, the parameter will take the specified default value.
   * **Example:** /employee/profile/{EmployeeName=harsha} (The EmployeeName parameter defaults to "harsha").

Code:

1. // ... (UseRouting and other middleware) ...
3. app.UseEndpoints(endpoints =>
4. {
5. // Required Parameters
6. endpoints.Map("files/{filename}.{extension}", async context =>
7. {
8. string? fileName = Convert.ToString(context.Request.RouteValues["filename"]);
9. string? extension = Convert.ToString(context.Request.RouteValues["extension"]);
11. await context.Response.WriteAsync($"In files - {fileName} - {extension}");
12. });
14. // Default Parameter
15. endpoints.Map("employee/profile/{EmployeeName=harsha}", async context =>
16. {
17. string? employeeName = Convert.ToString(context.Request.RouteValues["employeename"]);
18. await context.Response.WriteAsync($"In Employee profile - {employeeName}");
19. });
21. // Optional Parameter
22. endpoints.Map("products/details/{id?}", async context => {
23. if (context.Request.RouteValues.ContainsKey("id"))
24. {
25. int id = Convert.ToInt32(context.Request.RouteValues["id"]);
26. await context.Response.WriteAsync($"Products details - {id}");
27. }
28. else
29. {
30. await context.Response.WriteAsync($"Products details - id is not supplied");
31. }
32. });
33. });
35. // ... (Fallback middleware) ...
36. **Required Parameters Example:**
    * The route files/{filename}.{extension} expects both filename and extension to be present in the URL (e.g., /files/sample.txt).
    * The endpoint handler extracts these values from context.Request.RouteValues and uses them in the response.
37. **Default Parameter Example:**
    * The route employee/profile/{EmployeeName=harsha} has a default value for EmployeeName.
    * If you visit /employee/profile, the response will be "In Employee profile - harsha".
    * If you visit /employee/profile/john, the response will be "In Employee profile - john".
38. **Optional Parameter Example:**
    * The route products/details/{id?} allows the id parameter to be omitted.
    * If you visit /products/details/123, it will show the product details for ID 123.
    * If you visit /products/details, it will indicate that the ID was not provided.

Route Constraints

Route constraints are an essential tool in ASP.NET Core routing that allows you to add extra validation to your route parameters. They define rules that restrict the values a parameter can accept, helping you filter out invalid requests before they reach your endpoint handlers.

**Why Use Route Constraints?**

* **Enhanced Validation:** Ensure that only requests with valid parameter values are handled.
* **Improved Security:** Prevent malicious input by rejecting requests with potentially harmful values.
* **Cleaner Code:** Avoid cluttering your endpoint handlers with validation logic.
* **Explicit Routing:** Make your routes more self-documenting and easier to understand.

Common Route Constraints

ASP.NET Core provides a variety of built-in route constraints:

* **int:** Requires the parameter value to be an integer.
* **bool:** Requires the parameter value to be a boolean (true or false).
* **datetime:** Requires the parameter value to be a valid date and time string.
* **decimal, double, float, long:** Require the parameter value to be of the specified numeric type.
* **guid:** Requires the parameter value to be a valid GUID (Globally Unique Identifier).
* **alpha:** Requires the parameter value to consist only of alphabetic characters (a-z, A-Z).
* **regex:** Requires the parameter value to match a regular expression pattern.
* **length:** Requires the parameter value to have a specific length or within a specified range.
* **min, max, range:** Require the parameter value to be greater than or equal to the minimum (min), less than or equal to the maximum (max), or within a specific range (range).

Code

1. // ... (UseRouting and other middleware) ...
3. app.UseEndpoints(endpoints =>
4. {
5. // ... (other endpoints) ...
7. // Alphabetic and Length Constraint
8. endpoints.Map("employee/profile/{EmployeeName:length(4,7):alpha=harsha}", async context =>
9. {
10. // ...
11. });

14. // Integer, Range, and Optional Constraint
15. endpoints.Map("products/details/{id:int:range(1,1000)?}", async context => {
16. // ...
17. });
19. // DateTime Constraint
20. endpoints.Map("daily-digest-report/{reportdate:datetime}", async context =>
21. {
22. // ...
23. });
25. // GUID Constraint
26. endpoints.Map("cities/{cityid:guid}", async context =>
27. {
28. // ...
29. });
31. // Int, Min, Regex Constraint
32. endpoints.Map("sales-report/{year:int:min(1900)}/{month:regex(^(apr|jul|oct|jan)$)}", async context =>
33. {
34. // ...
35. });
36. });
38. // ... (Fallback middleware) ...
39. **Alphabetic and Length Constraint:** /employee/profile/{EmployeeName:length(4,7):alpha=harsha}: Ensures EmployeeName is 4-7 characters long and consists only of alphabetic characters. If not supplied, it defaults to "harsha".
40. **Integer, Range, and Optional Constraint:** /products/details/{id:int:range(1,1000)?}: Requires id to be an integer between 1 and 1000. The question mark makes it optional.
41. **DateTime Constraint:** /daily-digest-report/{reportdate:datetime}: Requires reportdate to be a valid date-time string.
42. **GUID Constraint:** /cities/{cityid:guid}: Requires cityid to be a valid GUID.
43. **Integer, Min, and Regex Constraint:** /sales-report/{year:int:min(1900)}/{month:regex(^(apr|jul|oct|jan)$)}: Requires year to be an integer greater than or equal to 1900, and month to be one of the specified values (apr, jul, oct, jan).

Custom Route Constraint Classes

While ASP.NET Core offers a variety of built-in route constraints, sometimes your application requires more specialized validation rules. Custom route constraint classes allow you to define your own criteria for determining whether a parameter value is valid.

**Key Requirements**

1. **Implement IRouteConstraint:** Create a class that implements the IRouteConstraint interface.
2. **Match Method:** Implement the Match method, which will contain your custom validation logic. This method receives several parameters:
   * httpContext: The current HttpContext.
   * route: The IRouter object associated with the route.
   * routeKey: The name of the route parameter being validated.
   * values: A dictionary containing the route values.
   * routeDirection: Indicates whether the route is being matched for an incoming request or for generating a URL.
3. **Return true or false:** The Match method must return true if the parameter value is valid according to your constraint, and false otherwise.

Code

1. // MonthsCustomConstraint.cs
2. public class MonthsCustomConstraint : IRouteConstraint
3. {
4. public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)
5. {
6. // Check if the parameter value exists
7. if (!values.ContainsKey(routeKey))
8. {
9. return false; // Not a match
10. }
12. Regex regex = new Regex("^(apr|jul|oct|jan)$");
13. string? monthValue = Convert.ToString(values[routeKey]);
15. if (regex.IsMatch(monthValue))
16. {
17. return true; // It's a match
18. }
19. return false; // Not a match
20. }
21. }

Let's break this down:

1. **Implementation of IRouteConstraint:** The MonthsCustomConstraint class clearly implements this interface, signaling that it's a custom route constraint.
2. **Match Method:**
   * It first checks if the values dictionary contains the route parameter being validated (routeKey). If not, it's an immediate mismatch, and false is returned.
   * A regular expression (^(apr|jul|oct|jan)$) is used to define the valid month values.
   * The value associated with the routeKey is retrieved from the values dictionary and converted to a string.
   * The Regex.IsMatch method tests whether the retrieved value matches the allowed month pattern.
   * Returns true if the value matches, and false otherwise.

**Using the Custom Constraint**

1. // ... (in your endpoint configuration) ...
2. endpoints.Map("sales-report/{year:int:min(1900)}/{month:months}", async context =>
3. {
4. // ... your endpoint handler logic ...
5. });

* Notice the :months constraint after the month parameter. This indicates that the value for month should be validated against the MonthsCustomConstraint class.

Endpoint Selection

When a request arrives at your ASP.NET Core application, the routing middleware (UseRouting) analyzes the URL and HTTP method. It then compares this information against the collection of endpoints you've defined using methods like MapGet, MapPost, etc. The goal is to find the most suitable endpoint to handle the request.

However, what happens when multiple endpoints seem like potential matches? ASP.NET Core employs a well-defined algorithm to determine the winning endpoint.

**Endpoint Selection Algorithm**

1. **Precedence:**
   * **Explicit Matches:** Endpoints defined with more specific patterns (e.g., /products/{id}) take precedence over those with broader patterns (e.g., /products).
   * **Order of Registration:** If multiple endpoints with equally specific patterns could match, the endpoint that was registered *first* wins.
2. **HTTP Method:**
   * **Exact Match:** If the request method (GET, POST, etc.) exactly matches the method specified for an endpoint, that endpoint is preferred.
3. **Route Constraints:**
   * **More Specific Constraints:** Endpoints with more restrictive route constraints (e.g., id:int:range(1,100) vs. id:int) are favored.
4. **Catch-All (Fallback):**
   * If no other endpoint matches, and you have a catch-all endpoint (defined using MapFallback), it will be selected.

**Order of Precedence: A Visual Summary**

1. Explicit Match with Exact HTTP Method and More Specific Route Constraints
2. Explicit Match with Exact HTTP Method and Less Specific Route Constraints
3. Explicit Match with Any HTTP Method and More Specific Route Constraints
4. Explicit Match with Any HTTP Method and Less Specific Route Constraints
5. Order of Registration (if specificity is equal)
6. Catch-All Endpoint (if no other match is found)

**Practical Implications and Tips**

* **Mind Your Order:** Be mindful of the order in which you register your endpoints, especially if they have similar patterns.
* **Specificity Wins:** Define your routes as specifically as possible to avoid ambiguity.
* **Route Constraints:** Use route constraints to narrow down the valid values for parameters.
* **Catch-All with Caution:** Catch-all endpoints can be useful, but use them sparingly to avoid unintended matches.
* **Endpoint Metadata:** Explore the Endpoint object's metadata for insights into why a particular endpoint was selected.

Code

1. app.UseEndpoints(endpoints =>
2. {
3. endpoints.MapGet("/products/{id:int}", GetProductById); // Most specific
4. endpoints.MapGet("/products", GetAllProducts); // Less specific
5. endpoints.MapGet("/{path?}", CatchAllHandler); // Catch-all
6. });

In this example:

* /products/123 will match the first endpoint (GetProductById).
* /products will match the second endpoint (GetAllProducts).
* /anything-else will match the catch-all endpoint (CatchAllHandler).

**Resolving Ambiguity**

If the routing system cannot definitively determine the best match, you'll encounter an AmbiguousMatchException. This exception signals that you need to refine your route definitions or registration order to eliminate the conflict.

Static Files in ASP.NET Core

Static files are the assets that make up the visual presentation and functionality of your web application:

* **HTML Files:** The structure of your web pages.
* **CSS Stylesheets:** The styling and appearance of your content.
* **JavaScript Files:** The interactive elements and logic of your application.
* **Images:** Visual elements that enhance the user experience.

ASP.NET Core provides the UseStaticFiles() middleware component to efficiently serve these static files directly to the browser without requiring any server-side processing.

**WebRoot: The Default Location**

The WebRoot property in ASP.NET Core specifies the default directory from which static files are served. By default, this directory is named "wwwroot" and is located at the root of your project. However, you can customize this location if needed.

**UseStaticFiles() Middleware: Enabling Static File Serving**

* **Basic Usage:** Calling app.UseStaticFiles(); with no arguments will serve static files from the default WebRoot directory.
* **Customization:** You can customize the behavior of UseStaticFiles() by passing a StaticFileOptions object:
  + FileProvider: Specify a different file provider (e.g., PhysicalFileProvider) to serve files from a custom location.
  + RequestPath: Configure the base URL path for your static files (e.g., /static).
  + ContentTypeProvider: Customize how content types are determined for different file extensions.
  + OnPrepareResponse: Perform additional actions on the response before it's sent to the client.

Code

1. using Microsoft.Extensions.FileProviders;
3. // ...
5. var builder = WebApplication.CreateBuilder(new WebApplicationOptions()
6. {
7. WebRootPath = "myroot"
8. });
9. var app = builder.Build();
11. // Serve from the specified WebRoot ("myroot" in this case)
12. app.UseStaticFiles();
14. // Serve from a custom directory ("mywebroot") located within the project's ContentRootPath
15. app.UseStaticFiles(new StaticFileOptions()
16. {
17. FileProvider = new PhysicalFileProvider(
18. Path.Combine(builder.Environment.ContentRootPath, "mywebroot")
19. )
20. });
21. // ... (rest of your middleware and endpoints) ...

**Explanation**

1. **Custom WebRoot:** The WebRootPath property in WebApplicationOptions is set to "myroot", making "myroot" the default location for static files served by the first app.UseStaticFiles().
2. **Default Static Files:** The initial app.UseStaticFiles(); call serves files directly from the "myroot" directory. For instance, a request to /styles.css would look for a file named styles.css within "myroot".
3. **Custom Static Files Location:** The second app.UseStaticFiles call configures a PhysicalFileProvider to serve files from a custom location: "mywebroot". This directory is located within the application's ContentRootPath (the project's root folder).

**Important Considerations**

* **Security:** Always be cautious about the files you expose as static content. Avoid placing sensitive information in your WebRoot or custom directories.
* **Performance:** Consider using caching and compression techniques to optimize the delivery of static files.
* **Content Security Policy (CSP):** Implement a CSP to mitigate cross-site scripting (XSS) attacks that could exploit your static files.

By effectively managing your static files and utilizing the UseStaticFiles() middleware, you can enhance your ASP.NET Core application's performance and user experience.

KeyPoints to remember:

**Routing**

* **Purpose:** Matches incoming HTTP requests to specific endpoints (controllers, Razor Pages, minimal APIs) in your application.
* **Middleware:** UseRouting and UseEndpoints are essential middleware components for routing.
  + UseRouting: Analyzes the request URL and matches it to an endpoint.
  + UseEndpoints: Executes the matched endpoint's code.
* **Map\* Methods:** Used to define endpoints for different HTTP methods (e.g., MapGet, MapPost, MapControllerRoute).

**Endpoint Selection Order**

* **Specificity:** More specific routes (with more parameters or constraints) take precedence over less specific ones.
* **Registration Order:** If multiple routes are equally specific, the one registered first wins.
* **HTTP Method:** Routes with an exact method match are preferred.
* **Route Constraints:** Routes with more restrictive constraints are favored.
* **Catch-All:** A fallback endpoint handles unmatched requests.

**Route Parameters**

* **Types:** Required ({id}), optional ({id?}), default value ({id=123}).
* **Access:** Parameter values are accessed through context.Request.RouteValues.

**Route Constraints**

* **Purpose:** Restrict the allowed values for route parameters.
* **Built-in:** int, bool, datetime, guid, regex, length, min, max, range, etc.
* **Custom:** Create classes implementing IRouteConstraint to define your own validation logic.

**GetEndpoint()**

* **Purpose:** Retrieves information about the matched endpoint.
* **Usage:** Call context.GetEndpoint() within middleware **after** UseRouting.
* **Information:** Access endpoint properties like DisplayName, route pattern, and metadata.

**Static Files**

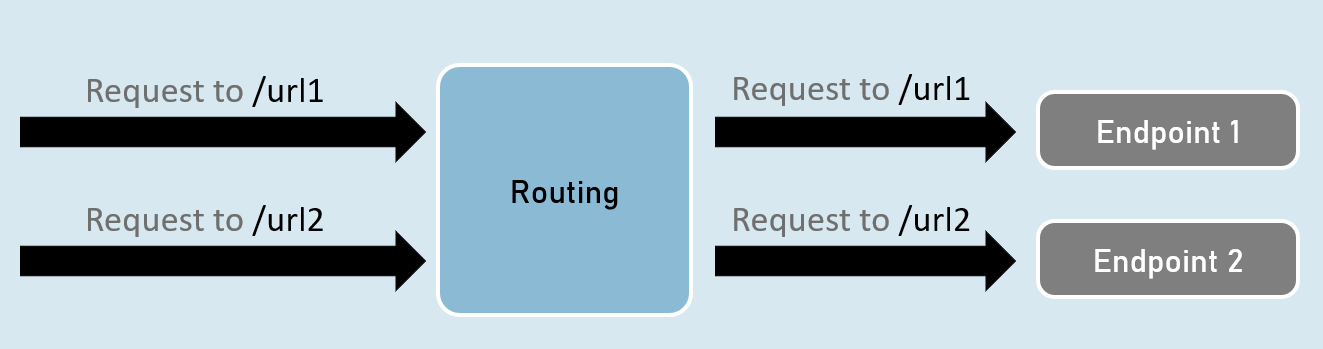
* **WebRoot:** The default directory from which static files are served (usually "wwwroot").
* **UseStaticFiles():** Middleware for serving static files (HTML, CSS, JavaScript, images).
* **Customization:** Use StaticFileOptions to change the file provider, request path, or other settings.

**Key Interview Tips**

* **Explain the Flow:** Clearly articulate how a request flows through the routing middleware and how endpoints are selected.
* **Code Examples:** Be prepared to write code snippets demonstrating endpoint registration, parameter usage, and constraint application.
* **Troubleshooting:** Explain how you would diagnose and fix common routing issues (e.g., 404 errors, ambiguous matches).
* **Best Practices:** Discuss how to design clean, maintainable, and secure routes.

Introduction to Routing

Routing is a process of matching incoming HTTP requests by checking the HTTP method and url; and then invoking corresponding endpoints.



Routing - UseRouting and UseEndPoints

**UseRouting( )**

app.UseRouting();

Enables routing and selects an appropriate end point based on the url path and HTTP method.

**UseEndPoints( )**

1. app.UseEndPoints(endpoints =>
2. {
3. endpoints.Map(…);
4. endpoints.MapGet(…);
5. endpoints.MapPost(…);
6. );

Executes the appropriate endpoint based on the endpoint selected by the above UseRouting() method.

Map, MapGet, MapPost

**endpoints.Map( )**

1. endpoints.Map("path", async (HttpContext context) =>
2. {
3. //code
4. });

Executes the endpoint when a HTTP request's url path begins with the specified path.

**endpoints.MapGet( )**

1. endpoints.MapGet("path", async (HttpContext context) =>
2. {
3. //code
4. });

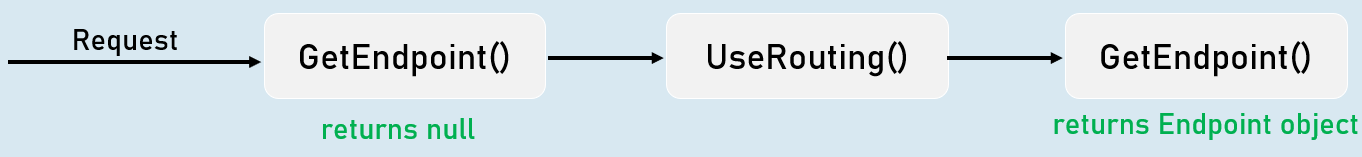
Executes the endpoint when a HTTP GET request's url path begins with the specified path.

**endpoints.MapPost( )**

1. endpoints.MapPost("path", async (HttpContext context) =>
2. {
3. //code
4. });

Executes the endpoint when a HTTP POST request's url path begins with the specified path.

GetEndpoint( )



context.GetEndpoint();

Returns an instance of Microsoft.AspNetCore.Http.Endpoint type, which represents an endpoint.

That instance contains two important properties: DisplayName, RequestDelegate.

Route Parameters

**"{parameter}"**

A route parameter can match with any value.



**Default Route Parameters**

"{parameter=default\_value}"

A route parameter with default value matches with any value.

It also matches with empty value. In this case, the default value will be considered into the parameter.

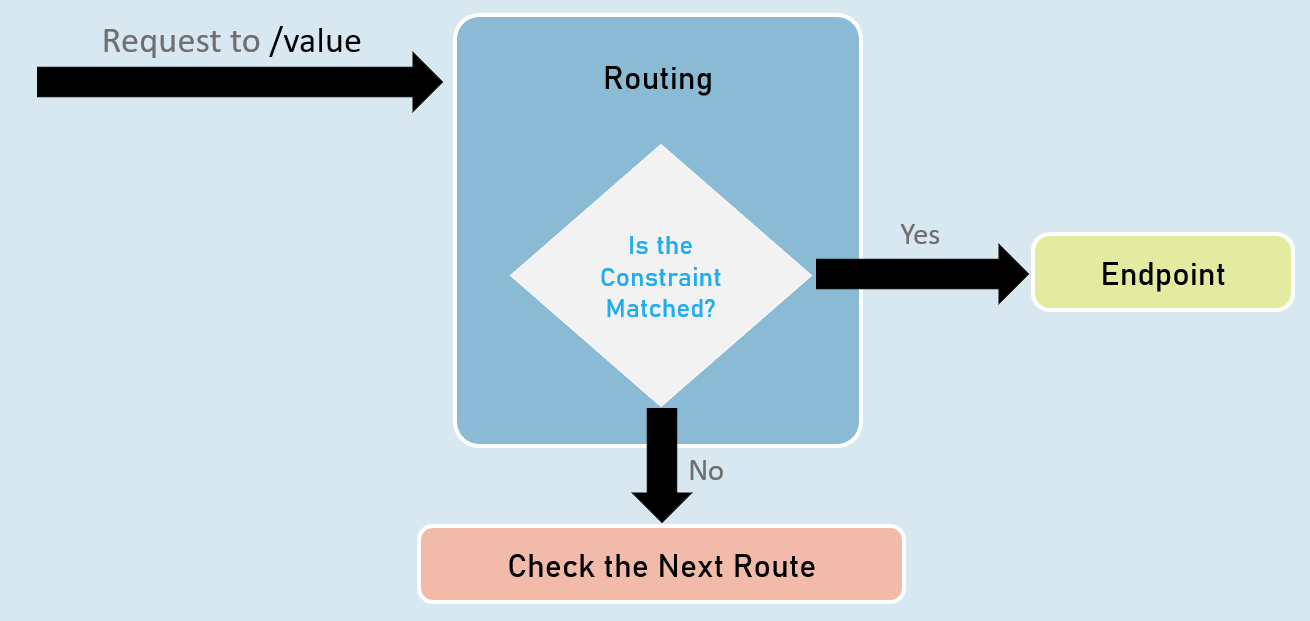
**Optional Route Parameters**

"{parameter?}"

"?" indicates an optional parameter.

That means, it matches with a value or empty value also.

Route Constraints



**Route Parameter with Constraint:**

"{parameter:constraint}"

A route parameter that has a constraint can match with a value that satisfies the given constraint.

**Multiple Constraints**

"{parameter:constraint1:constraint2}"

A route parameter can have more than one constraint, separated with colon ( : ).

**int**

Matches with any integer.

Eg: {id:int} matches with 123456789, -123456789

**bool**

Matches with true or false. Case-insensitive.

Eg: {active:bool} matches with true, false, TRUE, FALSE

**datetime**

Matches a valid DateTime value with formats "yyyy-MM-dd hh:mm:ss tt" and "MM/dd/yyyy hh:mm:ss tt".

Eg: {id:datetime} matches with 2030-01-01%2011:59%20pm

Note: '%20' is equal to space.

**decimal**

Matches with a valid decimal value.

Eg: {price:decimal} matches with 49.99, -1, 0.01

**long**

Matches a valid long value.

Eg: {id:long} matches with 123456789, -123456789

**guid**

Matches with a valid Guid value (Globally Unique Identifier - A hexadecimal number that is universally unique).

Eg: {id:guid} matches with 123E4567-E89B-12D3-A456-426652340000

**minlength(value)**

Matches with a string that has at least specified number of characters.

Eg: {username:minlength(4)} matches with John, Allen, William

**maxlength(value)**

Matches with a string that has less than or equal to the specified number of characters.

Eg: {username:maxlength(7)} matches with John, Allen, William

**length(min,max)**

Matches with a string that has number of characters between given minimum and maximum length (both numbers including).

Eg: {username:length(4, 7)} matches with John, Allen, William

**length(value)**

Matches with a string that has exactly specified number of characters.

Eg: {tin:length(9)} matches with 987654321

**min(value)**

Matches with an integer value greater than or equal to the specified value.

Eg: {age:min(18)} matches with 18, 19, 100

**max(value)**

Matches with an integer value less than or equal to the specified value.

Eg: {age:max(100)} matches with -1, 1, 18, 100

**range(min,max)**

Matches with an integer value between the specified minimum and maximum values (both numbers including).

Eg: {age:range(18,100)} matches with 18, 19, 99, 100

**alpha**

Matches with a string that contains only alphabets (A-Z) and (a-z).

Eg: {username:alpha} matches with rick, william

**regex(expression)**

Matches with a string that matches with the specified regular expression.

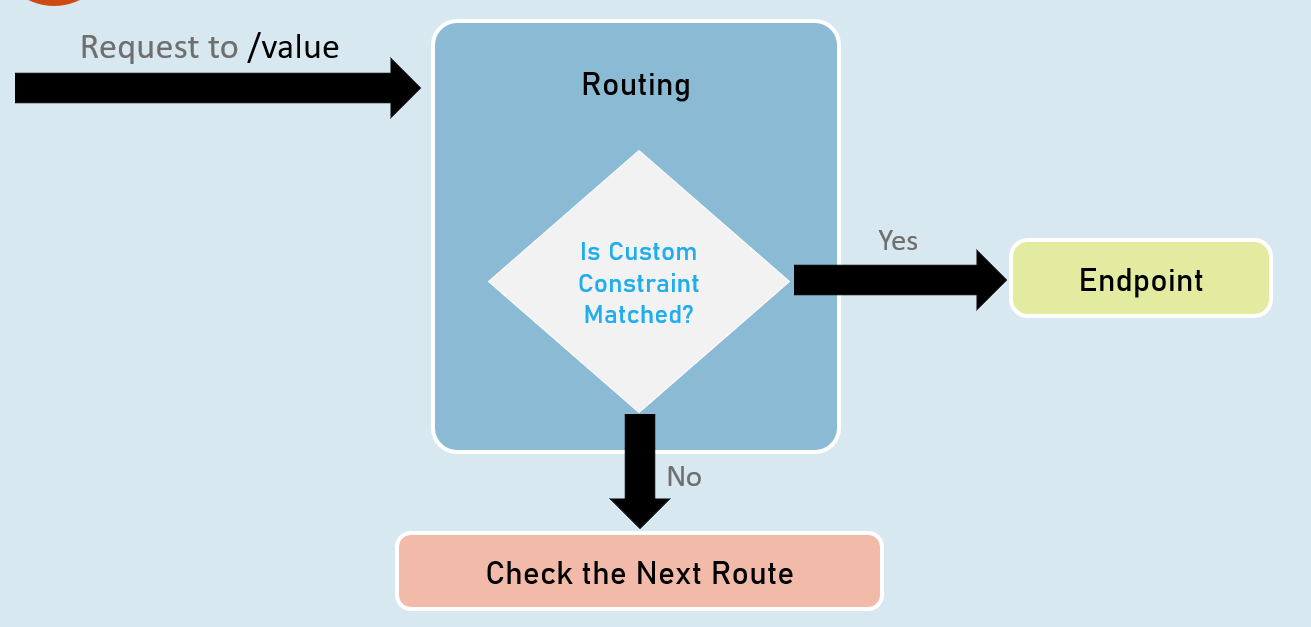
Eg 1: {age:regex(^[0-9]{2}$)} matches with any two-digit number, such as 10, 11, 98, 99

Eg 2: {age:regex(^\d{3}-\d{3}$)} matches with any three-digit number, then hyphen, and then three-digit number, such as 123-456

Custom Route Constraint Classes

Custom Route Constraint Class

1. public class ClassName : IRouteConstraint
2. {
3. public bool Match(HttpContext? HttpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)
4. {
5. //return true or false
6. }
7. }
8. builder.Services.AddRouting(options =>
9. {
10. options.ConstraintMap.Add("name", typeof(ClassName));
11. }); //adding the custom constraint to routing



Endpoint Selection Order

Top is highest precedence (will be evaluated first)

**1:**URL template with more segments.

Eg: "a/b/c/d" is higher than "a/b/c".

**2:**URL template with literal text has more precedence than a parameter segment.

Eg: "a/b" is higher than "a/{parameter}".

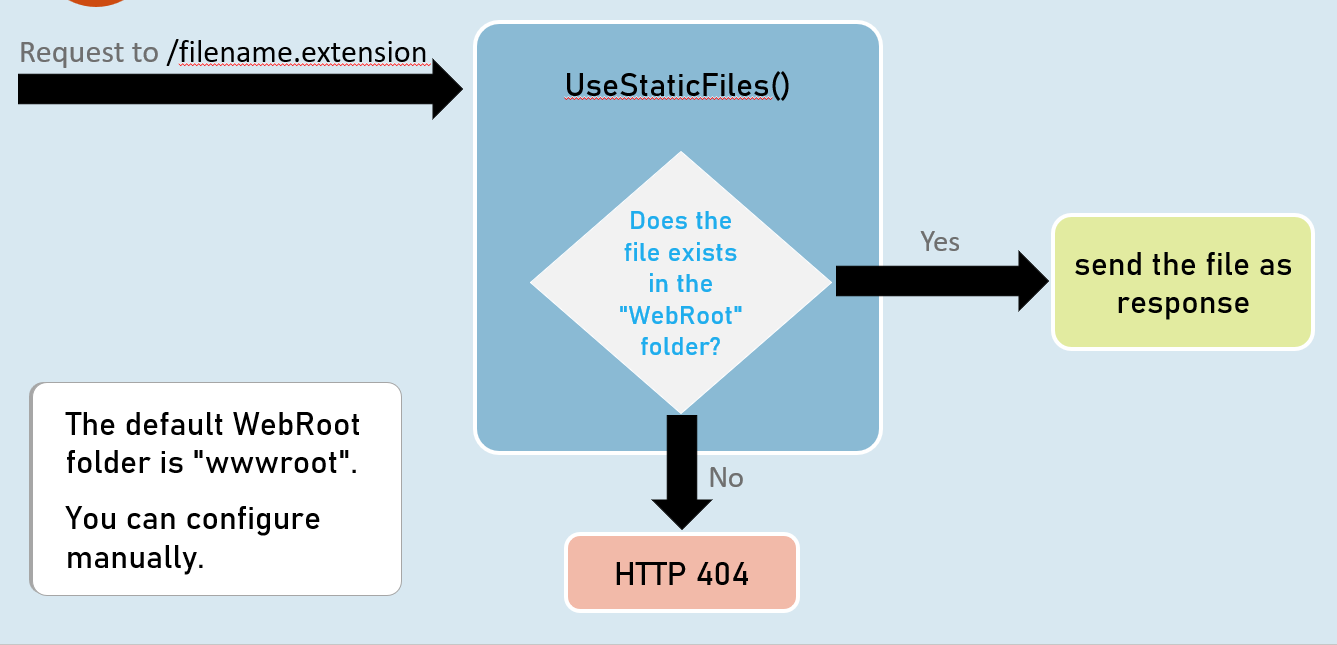
**3:**URL template that has a parameter segment with constraints has more precedence than a parameter segment without constraints.

Eg: "a/b:int" is higher than "a/b".

**4:**Catch-all parameters (\*\*).

Eg: "a/{b}" is higher than "a/\*\*".

WebRoot



# **4. Controllers and IActionResult [MVC]**

Controllers and Action Methods

In the Model-View-Controller (MVC) architectural pattern, controllers serve as the orchestrators of your web application. They handle incoming HTTP requests, interact with the model (your data layer), and select the appropriate view to render the response back to the user.

* **Controllers:** Classes that group related action methods and typically reside in the Controllers folder in your project.
* **Action Methods:** Public methods within a controller that handle specific requests (e.g., displaying a page, processing form data).

**Purpose**

* **Organize Logic:** Controllers provide a logical grouping for actions that work on the same type of data or functionality.
* **Handle Requests:** They are responsible for processing requests, retrieving necessary data, and preparing a response.
* **Select Views:** Controllers often choose the appropriate view to render, passing data (the model) to the view for presentation.

**Syntax and Conventions**

* **Class Naming:** Controller class names should end with "Controller" (e.g., HomeController, ProductsController).
* **Inheritance:** Controllers inherit from the Controller base class (or ControllerBase for API controllers).
* **Action Method Naming:** Action methods can have any valid C# method name.
* **Return Types:** Action methods can return various types, including:
  + IActionResult: A common interface that allows you to return different result types (views, content, redirects, etc.).
  + string, int, etc.: For API controllers, you might return raw data.

Attribute Routing

Attribute routing allows you to define routes directly on your controller classes and action methods using attributes:

* **[Route] Attribute:** Specifies the base route template for the controller or action.
* [HttpGet], [HttpPost], etc.: Indicate the HTTP method(s) the action should handle.

**Controller Responsibilities**

* **Request Handling:** Process incoming requests and extract relevant data (from route parameters, query strings, or the request body).
* **Model Interaction:** Retrieve data from your model (database, services) or update the model based on the request.
* **View Selection:** Determine which view should be rendered and provide the necessary model data to the view.
* **Error Handling:** Handle errors gracefully and return appropriate responses.

Code

1. // HomeController.cs
2. namespace ControllersExample.Controllers
3. {
4. [Controller] // Marks the class as a controller
5. public class HomeController
6. {
7. [Route("home")] // Routes for this action
8. [Route("/")]
9. public string Index()
10. {
11. return "Hello from Index";
12. }
14. [Route("about")]
15. public string About()
16. {
17. return "Hello from About";
18. }
20. [Route("contact-us/{mobile:regex(^\\d{10}$)}")] // Route with constraint
21. public string Contact()
22. {
23. return "Hello from Contact";
24. }
25. }
26. }
28. // Program.cs (or Startup.cs)
29. var builder = WebApplication.CreateBuilder(args);
30. builder.Services.AddControllers(); // Enables MVC controllers
32. var app = builder.Build();
33. app.UseRouting();
34. app.MapControllers(); // Connects controllers to the routing system
35. app.Run();

* **HomeController:** This is your controller class.
* **Index, About, Contact:** These are action methods within the controller, each with a corresponding route.
* **[Route] Attributes:** Define the routes for each action method.
* **[Controller] Attribute:** Marks the class as a controller, making it discoverable by the framework.
* **builder.Services.AddControllers();:** Registers MVC services and makes controllers available for dependency injection.
* **app.MapControllers();:** Connects the routing system to your controllers, enabling them to handle requests.

ContentResult

In the realm of ASP.NET Core MVC, action methods often return different types of results: views (HTML), JSON data, or file streams. The ContentResult class caters to a specific need: returning raw content directly to the client, without the overhead of rendering a full view. This content could be plain text, XML, JSON, CSV, or any other format you specify.

**Why Use ContentResult?**

* **Flexibility:** You have complete control over the content you send and the Content-Type header, allowing you to tailor the response to specific client requirements.
* **Lightweight:** ContentResult is efficient because it doesn't involve complex view rendering.
* **Directness:** Ideal for scenarios where you want to return simple text messages, API responses, or custom content formats.

The ContentResult class provides the following key properties to shape your response:

1. **Content:** This is where you set the actual content that you want to send back to the client. It could be a simple string, a serialized object, or any data you want to transmit.
2. **ContentType:** This property is crucial. It specifies the MIME type (Multipurpose Internet Mail Extensions) of the content. The MIME type tells the client how to interpret the data you are sending. Here are some common examples:
   * text/plain: Plain text
   * text/html: HTML content
   * application/json: JSON data
   * text/csv: CSV data
   * application/xml: XML data
3. **StatusCode (Optional):** You can optionally set the HTTP status code of the response (e.g., 200 OK, 404 Not Found). If not specified, it defaults to 200 OK.

**Creating a ContentResult**

You have a couple of options for creating a ContentResult in your action methods:

1. **Instantiating ContentResult:**
   1. return new ContentResult()
   2. {
   3. Content = "Hello from Index",
   4. ContentType = "text/plain"
   5. };
2. **Using the Content() Helper Method:**
   1. return Content("Hello from Index", "text/plain");

The Content() method is a shortcut provided by the Controller base class to conveniently create a ContentResult.

Code

1. // HomeController.cs (modified)
2. [Route("home")]
3. [Route("/")]
4. public ContentResult Index()
5. {
6. return Content("<h1>Welcome</h1> <h2>Hello from Index</h2>", "text/html");
7. }

In this modified Index action:

1. **HTML Content:** The content being returned is an HTML string containing heading tags.
2. **Content Type:** The ContentType is set to "text/html", instructing the browser to render the response as HTML.
3. **Client Experience:** When a user navigates to the /home or / route, they will see a webpage with a formatted heading "Welcome" and a subheading "Hello from Index."

While returning raw strings directly from action methods is often convenient, using ContentResult gives you explicit control over the ContentType header, which is vital for ensuring the client correctly interprets the response data.

JsonResult

The JsonResult class in ASP.NET Core MVC is your go-to tool when you need to return structured data in JSON (JavaScript Object Notation) format from your controller actions. JSON has become the de facto standard for data exchange in web APIs and modern web applications due to its simplicity, readability, and wide support across platforms and languages.

**Why Use JsonResult?**

* **Standardized Format:** JSON is a well-established format for representing structured data, making it ideal for communication between web applications and APIs.
* **Serialization:** ASP.NET Core seamlessly serializes your objects into JSON, saving you from manual formatting.
* **Content Type:** JsonResult automatically sets the Content-Type header to application/json, ensuring that the client (e.g., a browser or another application) correctly interprets the response.
* **API-Friendly:** Perfect for building RESTful APIs or returning data for client-side JavaScript to consume.

**Creating a JsonResult**

Similar to ContentResult, you have a couple of convenient ways to create a JsonResult in your action methods:

1. **Instantiating JsonResult:**
   1. return new JsonResult(person);

Here, you pass the object (e.g., person) that you want to serialize into JSON directly to the JsonResult constructor.

1. **Using the Json() Helper Method:**
   1. return Json(person);

The Json() method is a shorthand provided by the Controller base class, making it even easier to create a JsonResult.

Code

1. // HomeController.cs
2. [Route("person")]
3. public JsonResult Person()
4. {
5. Person person = new Person()
6. {
7. Id = Guid.NewGuid(),
8. FirstName = "James",
9. LastName = "Smith",
10. Age = 25
11. };
13. return Json(person);
14. }

In this modified Person action:

1. **Person Object:** A Person object is created with some sample data (including a unique ID).
2. **JSON Serialization:** The Json(person) call serializes the person object into a JSON string.
3. **Response:** The resulting JSON string is returned as a JsonResult, with the Content-Type header automatically set to application/json.

**Output:**

The response sent to the client would look like this:

1. {
2. "id": "123e4567-e89b-12d3-a456-426614174000",
3. "firstName": "James",
4. "lastName": "Smith",
5. "age": 25
6. }

File Results

In ASP.NET Core MVC, file results are action results designed to serve files to the client. They are particularly useful when you want your application to deliver files like PDFs, images, documents, or other binary content.

**Types of File Results**

1. **VirtualFileResult:**
   * **Purpose:** Serves a file from the application's web root directory (wwwroot by default) or a virtual path.
   * **Parameters:**
     + virtualPath: The path to the file within the web root or the virtual path.
     + contentType: The MIME type of the file (e.g., application/pdf).
   * **Usage:**
     + return new VirtualFileResult("/sample.pdf", "application/pdf");
     + return File("/sample.pdf", "application/pdf"); (Shorthand version)
   * **Benefits:** Provides security by restricting file access to the web root or configured virtual paths.
2. **PhysicalFileResult:**
   * **Purpose:** Serves a file from an absolute file path on the server's file system.
   * **Parameters:**
     + physicalPath: The absolute path to the file.
     + contentType: The MIME type of the file.
   * **Usage:**
     + return new PhysicalFileResult(@"c:\aspnetcore\sample.pdf", "application/pdf");
     + return PhysicalFile(@"c:\aspnetcore\sample.pdf", "application/pdf"); (Shorthand version)
   * **Benefits:** Allows serving files from locations outside the web root, but requires careful handling due to potential security risks.
3. **FileContentResult:**
   * **Purpose:** Serves a file from an in-memory byte array.
   * **Parameters:**
     + fileContents: The file contents as a byte array.
     + contentType: The MIME type of the file.
   * **Usage:**
     + byte[] bytes = System.IO.File.ReadAllBytes(@"c:\aspnetcore\sample.pdf");
     + return new FileContentResult(bytes, "application/pdf");
     + return File(bytes, "application/pdf"); (Shorthand version)
   * **Benefits:** Useful for dynamically generated files or when you don't want to expose the file's actual path.

Code

1. // HomeController.cs
2. [Route("file-download")]
3. public VirtualFileResult FileDownload()
4. {
5. return File("/sample.pdf", "application/pdf"); // Serves from wwwroot
6. }
8. [Route("file-download2")]
9. public PhysicalFileResult FileDownload2()
10. {
11. return PhysicalFile(@"c:\aspnetcore\sample.pdf", "application/pdf"); // Full path
12. }
14. [Route("file-download3")]
15. public FileContentResult FileDownload3()
16. {
17. byte[] bytes = System.IO.File.ReadAllBytes(@"c:\aspnetcore\sample.pdf");
18. return File(bytes, "application/pdf"); // In-memory bytes
19. }

**Key Considerations**

* **Security:** Be extremely cautious when using PhysicalFileResult to prevent unauthorized access to your server's file system. Validate paths rigorously and avoid exposing sensitive information.
* **Performance:** Consider caching file results to improve performance, especially for larger files or frequently requested content.
* **Content Disposition:** Use the FileDownloadName property of the file result to suggest a filename for the browser when the user downloads the file.

**Choosing the Right File Result**

* **VirtualFileResult:** When the file resides within your web root and you don't need to expose its absolute path.
* **PhysicalFileResult:** When you need to serve a file from an arbitrary location on the server's file system (use with caution).
* **FileContentResult:** When you have the file content in memory (e.g., dynamically generated) or when you don't want to reveal the file's actual path.

IActionResult

The IActionResult interface is a core concept in ASP.NET Core MVC. It serves as the return type for action methods in your controllers, providing flexibility and enabling you to return different types of responses depending on the context of the request.

Essentially, it's a contract that defines a single method:

1. Task ExecuteResultAsync(ActionContext context);

This method is responsible for executing the specific logic associated with the action result, generating the appropriate HTTP response that's sent back to the client.

Action Result Types

Here's a breakdown of some of the most important action result types derived from IActionResult:

* **ContentResult:** Returns a string as raw content (text, HTML, XML, etc.).
  + Example: return Content("Hello from Index", "text/plain");
* **EmptyResult:** Represents an empty response (204 No Content).
  + Example: return new EmptyResult();
* **FileResult:** Used to send files to the client (PDF, images, etc.). This is a base class for several more specific file result types.
  + VirtualFileResult: Serves a file from the web root or a virtual path.
  + PhysicalFileResult: Serves a file from a physical path on the server.
  + FileContentResult: Serves a file from an in-memory byte array.
* **JsonResult:** Serializes an object into JSON format and sends it as the response.
  + Example: return Json(new { message = "Success" });
* **RedirectResult:** Redirects the user to a different URL.
  + Example: return Redirect("/home");
* **RedirectToActionResult:** Redirects to a specific action method in a controller.
  + Example: return RedirectToAction("Index", "Home");
* **ViewResult:** Renders a view, typically an HTML page, with optional model data.
  + Example: return View("Index", model);
* **PartialViewResult:** Renders a partial view (a reusable portion of a view).
  + Example: return PartialView("\_ProductCard", product);
* **StatusCodeResult:** Returns a specific HTTP status code with an optional message.
  + Example: return StatusCode(404, "Resource not found");
* **BadRequestResult:** Shorthand for returning a 400 Bad Request response.
* **NotFoundResult:** Shorthand for returning a 404 Not Found response.
* **OkResult:** Shorthand for returning a 200 OK response.

Code

1. // HomeController.cs
2. [Route("book")]
3. public IActionResult Index()
4. {
5. // Book id should be applied
6. if (!Request.Query.ContainsKey("bookid"))
7. {
8. Response.StatusCode = 400; // Setting status code manually
9. return Content("Book id is not supplied");
10. }
12. // ... other validation checks ...
14. // If all checks pass
15. return File("/sample.pdf", "application/pdf");
16. }

In this action method:

1. **Validation:** The code performs several validation checks on the bookid query parameter:
   * It checks if the parameter exists.
   * It checks if the parameter value is not null or empty.
   * It checks if the parameter value is within a valid range (1-1000).
   * It checks if the isloggedin query parameter is true.
2. **Error Responses:** If any validation fails, a ContentResult is returned with an appropriate error message and a 400 Bad Request or 401 Unauthorized status code.
3. **Successful Response:** If all validation passes, a FileResult is returned, serving the sample.pdf file from the web root.

**Notes**

* **Flexibility:** IActionResult allows you to return different types of responses based on the logic in your action.

Status Code Results

In web communication, it's crucial to inform the client about the outcome of their request. Status codes provide a standardized way to convey this information. ASP.NET Core MVC offers a range of action results designed specifically to return these status codes along with optional messages.

**Common Status Code Results**

* **OkResult:** Indicates a successful request (HTTP 200).
* **BadRequestResult:** Indicates a client error (HTTP 400). Often used for invalid input.
* **NotFoundResult:** Indicates that the requested resource was not found (HTTP 404).
* **UnauthorizedResult:** Indicates that the request requires authentication (HTTP 401).
* **ForbiddenResult:** Indicates that the user is not authorized to access the resource (HTTP 403).
* **StatusCodeResult:** Allows you to return any arbitrary HTTP status code.

Using Status Code Results

1. **Direct Instantiation:**
2. return new BadRequestResult(); // Returns HTTP 400
3. return new NotFoundResult(); // Returns HTTP 404
4. **Helper Methods:**
5. return BadRequest(); // Returns HTTP 400
6. return NotFound(); // Returns HTTP 404
7. return Unauthorized(); // Returns HTTP 401
8. return StatusCode(403); // Returns HTTP 403
9. **With Messages:**
10. return BadRequest("Invalid input data");
11. return NotFound("Resource not found");

These helper methods are more concise and expressive than directly instantiating the result objects.

Code

1. // HomeController.cs
2. [Route("book")]
3. public IActionResult Index()
4. {
5. // ... (validation checks similar to the previous example) ...
7. if (bookId <= 0)
8. {
9. return BadRequest("Book id can't be less than or equal to zero");
10. }
12. // Note the use of NotFound here
13. if (bookId > 1000)
14. {
15. return NotFound("Book id can't be greater than 1000");
16. }
18. if (Convert.ToBoolean(Request.Query["isloggedin"]) == false)
19. {
20. return StatusCode(401); // Customizable status code
21. }
23. return File("/sample.pdf", "application/json");
24. }

In this refined example, the validation logic remains the same. However, we've made the following changes:

1. **Specific Status Codes:**
   * We use BadRequest() for invalid input (e.g., bookId less than or equal to zero).
   * We use NotFound() when the bookId is out of the valid range (greater than 1000), as it could imply the requested book doesn't exist.
2. **Customizable Status Code:**
   * For the authentication failure case (isloggedin is false), we use StatusCode(401) to return the standard 401 Unauthorized status code. You could also use return Unauthorized(); as a shortcut.

**Notes**

* **Inform the Client:** Status codes are essential for communicating the outcome of a request to the client.
* **Standard Codes:** Use the standard HTTP status codes whenever possible for consistency and interoperability.
* **Helper Methods:** Leverage the helper methods (BadRequest, NotFound, etc.) for cleaner and more expressive code.
* **Customization:** The StatusCode result allows you to return any HTTP status code you need, but use it judiciously.
* **Beyond Validation:** Status codes are not just for validation; use them to signal the result of any action in your API.

Redirect Results

Redirect results are action results in ASP.NET Core MVC that instruct the client's browser to navigate to a new URL. This is commonly used after actions like form submissions, logins, or other operations where you want to transition the user to a different page.

**Types of Redirect Results**

1. **RedirectResult:**
   * **Purpose:** Redirects to a specified URL (either absolute or relative).
   * **Parameters:**
     + url: The URL to redirect to.
     + permanent: A boolean indicating whether the redirect is permanent (301 Moved Permanently) or temporary (302 Found). Defaults to false (temporary).
   * **Usage:**
     + return Redirect("/home"); (Temporary)
     + return RedirectPermanent("/home"); (Permanent)
2. **RedirectToActionResult:**
   * **Purpose:** Redirects to a specific action method within a controller.
   * **Parameters:**
     + actionName: The name of the action method.
     + controllerName: The name of the controller (optional, defaults to the current controller).
     + routeValues: An object containing route values to pass to the action (optional).
     + permanent: A boolean indicating whether the redirect is permanent (301) or temporary (302).
   * **Usage:**
     + return RedirectToAction("Index"); (Temporary, same controller)
     + return RedirectToAction("Details", "Products", new { id = 123 }); (Temporary, with route values)
     + return RedirectToActionPermanent("About"); (Permanent)
3. **LocalRedirectResult:**
   * **Purpose:** Redirects to a local URL within the same application.
   * **Parameters:**
     + localUrl: The local URL to redirect to.
     + permanent: A boolean indicating whether the redirect is permanent (301) or temporary (302).
   * **Usage:**
     + return LocalRedirect("/products/details/456"); (Temporary)
     + return LocalRedirectPermanent("/about"); (Permanent)

Code

1. // HomeController.cs
2. [Route("bookstore")]
3. public IActionResult Index()
4. {
5. // ... validation logic (same as previous example) ...
7. // Conditional Redirects
8. if (someConditionIsTrue)
9. {
10. return RedirectToAction("Books", "Store", new { id = bookId }); // Temporary, to a different action
11. }
12. else
13. {
14. return LocalRedirectPermanent($"store/books/{bookId}"); // Permanent, local redirect
15. }
17. // ... other redirect examples ...
18. }

**Explanation of Redirect Types**

* **302 Found (RedirectResult or RedirectToActionResult with permanent: false):**
  + The standard temporary redirect. Tells the browser to fetch the new resource, but future requests should still use the original URL.
* **301 Moved Permanently (RedirectResult, RedirectToActionResult, or LocalRedirectResult with permanent: true):**
  + Indicates the resource has been permanently moved. The browser should update its bookmarks/links and future requests should use the new URL.
* **LocalRedirectResult:**
  + Specifically for redirects within the same application. Helps prevent open redirects, where a malicious actor could trick your site into redirecting to an external, harmful site.

**Choosing the Right Redirect**

* **External vs. Internal:** Use RedirectResult for external URLs and LocalRedirectResult for internal URLs.
* **Temporary vs. Permanent:** Use 301 for permanent moves, 302 for temporary ones (e.g., after form submission).
* **Action-Specific:** Use RedirectToActionResult when you want to redirect to a specific action within your application.
* **Safety:** Prefer LocalRedirectResult over RedirectResult for internal redirects to protect against open redirect attacks.

Key Points to Remember:

**1. Controllers**

* **Purpose:**
  + Handle HTTP requests.
  + Interact with the model (data layer).
  + Select appropriate views for rendering responses.
* **Naming:** End with "Controller" (e.g., HomeController).
* **Inheritance:** Inherit from Controller (or ControllerBase for APIs).
* **Action Methods:** Public methods within controllers that handle specific requests.
* **Attribute Routing:** Use [Route], [HttpGet], [HttpPost], etc., to define routes.

**2. IActionResult**

* **Purpose:** Flexible return type for action methods, enabling various response types.
* **Types:**
  + **Content-Based:**
    - ContentResult: Raw content (text, HTML, JSON, etc.).
    - JsonResult: Serialized JSON data.
    - FileResult (and subtypes): Files (PDF, images, etc.).
  + **Redirection:**
    - RedirectResult: Redirect to any URL.
    - RedirectToActionResult: Redirect to a specific action within your app.
    - LocalRedirectResult: Redirect to a local URL within the same app.
  + **Status Codes:**
    - StatusCodeResult: Any arbitrary HTTP status code.
    - BadRequestResult, NotFoundResult, UnauthorizedResult, etc.: Specific status codes.
  + **Views:**
    - ViewResult: Render a full view.
    - PartialViewResult: Render a partial view.

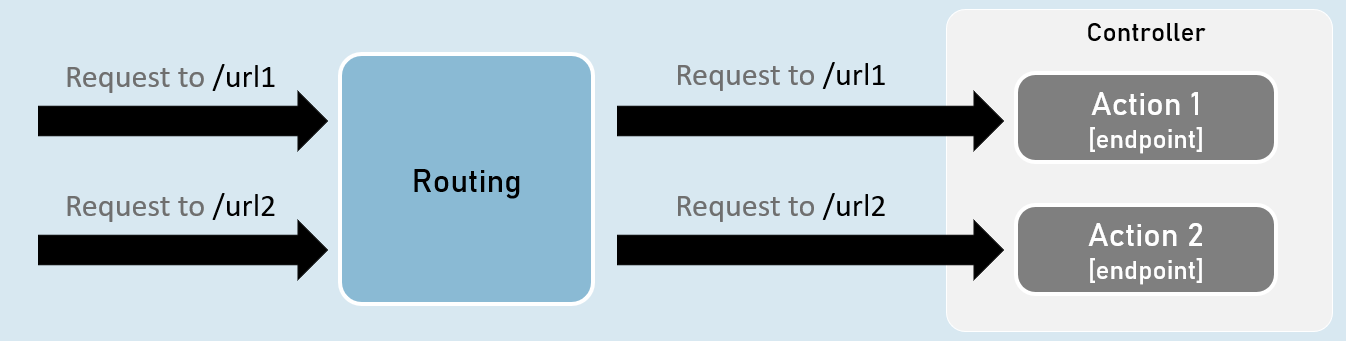
**Key Interview Tips**

* **Understand MVC:** Be able to explain the roles of models, views, and controllers.
* **Choosing Action Results:** Explain why you would choose one action result type over another based on the desired outcome.
* **Status Codes:** Know the common HTTP status codes and their meanings (200 OK, 404 Not Found, etc.).
* **Attribute Routing:** Demonstrate your ability to define routes using attributes.

Introduction to Controllers

Controller is a class that is used to group-up a set of actions (*or action methods*).

Action methods do perform certain operation when a request is received & returns the result (response).



**Creating Controllers**

Controllers should be either or both:

* The class name should be suffixed with "Controller". Eg: HomeController
* The [Controller] attribute is applied to the same class or to its base class.

**Controller**

1. [Controller]
2. class ClassNameController
3. {
4. //action methods here
5. }

**Optional:**

* Is a public class.
* Inherited from Microsoft.AspNetCore.Mvc.Controller.

**Enable 'routing' in controllers**

**AddControllers( )**

builder.Services.AddControllers();

Adds all controllers as services in the IServiceCollection.

So that, they can be accesed when a specific endpoint needs it.

**MapControllers()**

app.MapControllers();

Adds all action methods as endpoints.

So that, no need of using UseEndPoints() method for adding action methods as end points.

**Responsibilities of Controllers**

**Reading requests**

Extracting data values from request such as query string parameters, request body, request cookies, request headers etc.

**Invoking models**

Calling business logic methods.

Generally business operations are available as 'services'.

**Validation**

Validate incoming request details (query string parameters, request body, request cookies, request headers etc.)

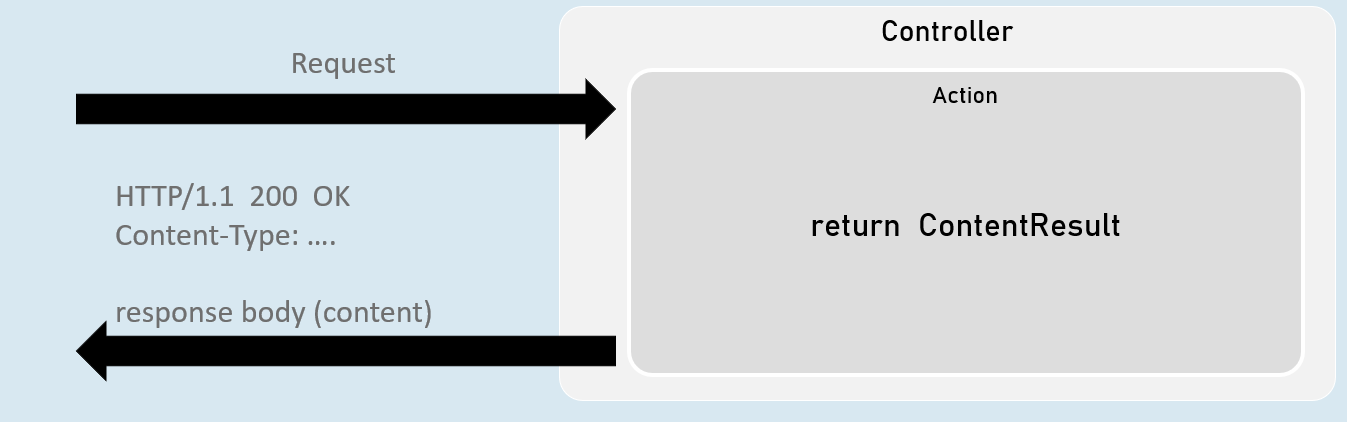
**Preparing Response**

Choosing what kind of response has to be sent to the client & also preparing the response (*action result*).

ContentResult

ContentResult can represent any type of response, based on the specified MIME type.

MIME type represents type of the content such as text/plain, text/html, application/json, application, xml, application/pdf etc.



return new ContentResult() { Content = "content", ContentType = "content type" };

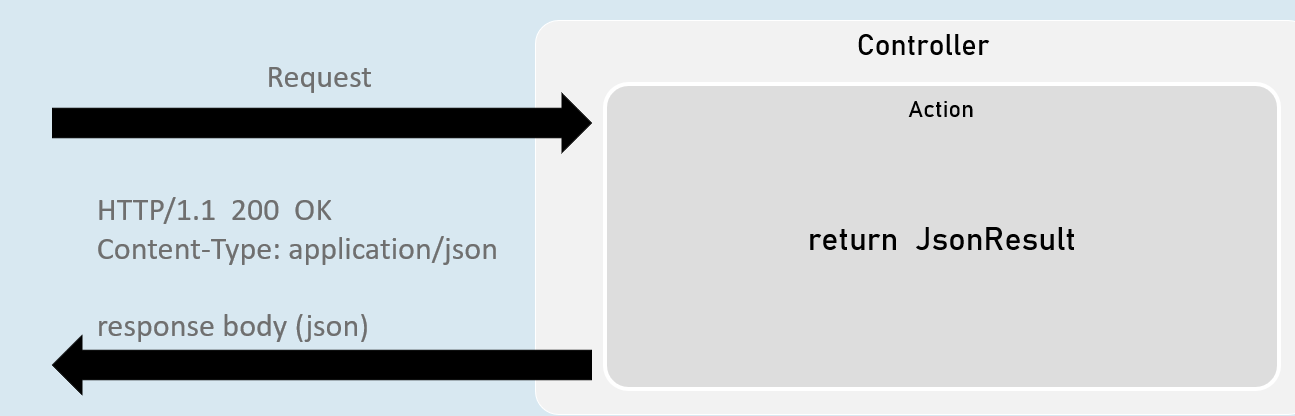
[or]

return Content("content", "content type");

JsonResult

JsonResult can represent an object in JavaScript Object Notation (JSON) format.

Eg: { "firstName": "James", "lastName": "Smith", "age": 25 }



return new JsonResult(your\_object);

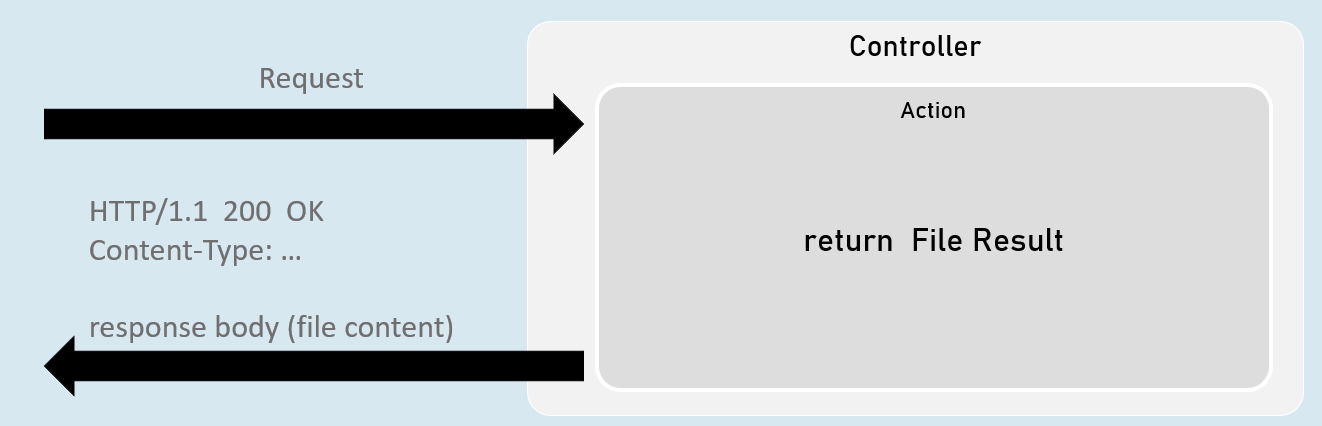
[or]

return Json(your\_object);

File Results

File result sends the content of a file as response.

Eg: pdf file, txt file, exe file, zip file etc.



**VirtualFileResult**

return new VirtualFileResult("file relative path", "content type");

//or

return File("file relative path", "content type");

Represents a file within the WebRoot ('wwwroot' by default) folder.

Used when the file is present in the WebRoot folder.

**PhysicalFileResult**

Represents a file that is not necessarily part of the project folder.

Used when the file is present outside the WebRoot folder.

return new PhysicalFileResult("file absolute path", "content type");

//or

return PhysicalFile("file absolute path", "content type");

**FileContentResult**

Represents a file from the byte[ ].

Used when a part of the file or byte[ ] from other data source has to be sent as response.

return new FileContentResult(byte\_array, "content type");

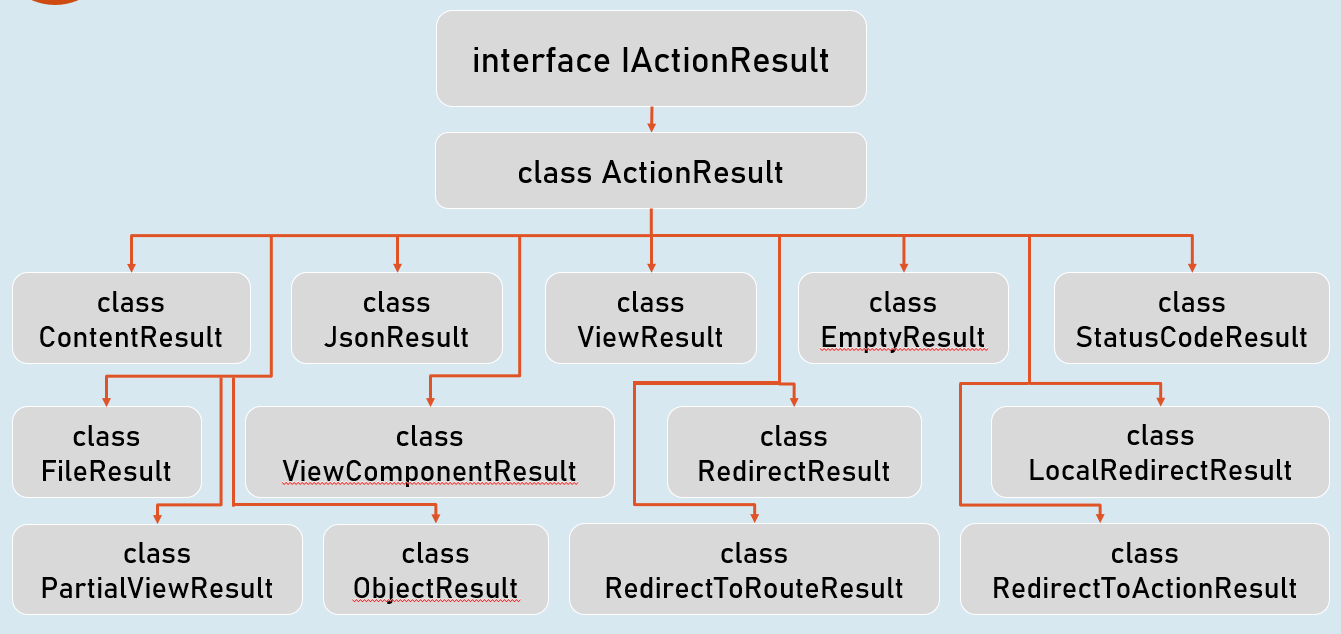
//or

return File(byte\_array, "content type");

IActionResult

It is the parent interface for all action result classes such as ContentResult, JsonResult, RedirectResult, StatusCodeResult, ViewResult etc.

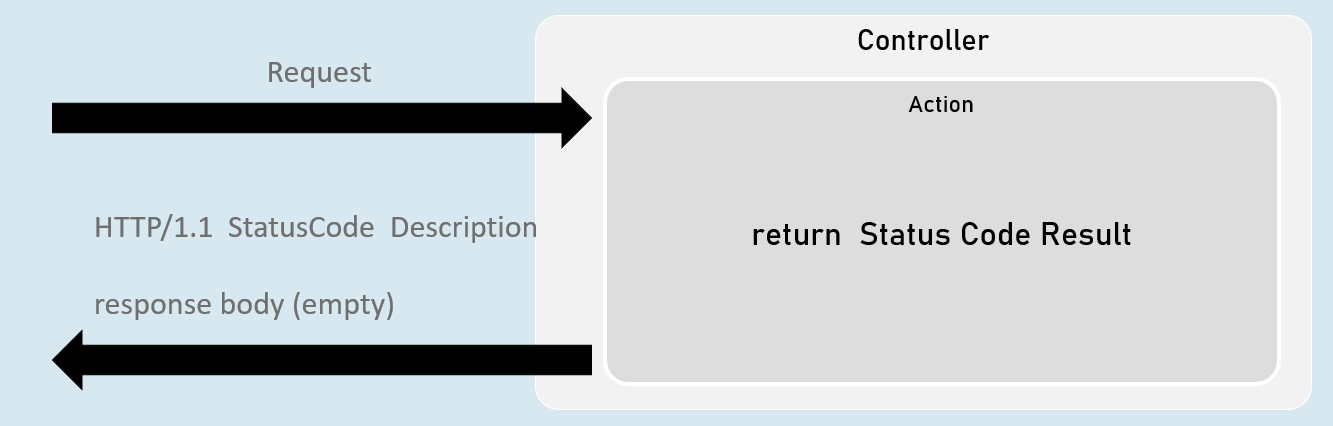
By mentioning the return type as IActionResult, you can return either of the subtypes of IActionResult



Status Code Results

Status code result sends an empty response with specified status code.

Eg: 200, 400, 401, 404, 500 etc.



**StatusCodeResult**

return new StatusCodeResult(status\_code);

**UnauthorizedResult**

return new UnauthorizedResult();

BadRequestResult

return new BadRequestResult();

**NotFoundResult**

return new NotFoundResult();

**StatusCodeResult**

* Represents response with the specified status code.
* Used when you would like to send a specific HTTP status code as response.

return new StatusCodeResult(status\_code);

//or

return StatusCode(status\_code);

**UnauthoriziedResult**

* Represents response with HTTP status code '401 Unauthorized'.
* Used when the user is unauthorized (not signed in).

return new UnauthorizedResult();

//or

return Unauthorized();

**BadRequestResult**

* Represents response with HTTP status code '400 Bad Request'.
* Used when the request values are invalid (validation error).

return new BadRequestResult();

//or

return BadRequest();

**NotFoundResult**

* Represents response with HTTP status code '404 Not Found'.
* Used when the requested information is not available at server.

return new NotFoundResult();

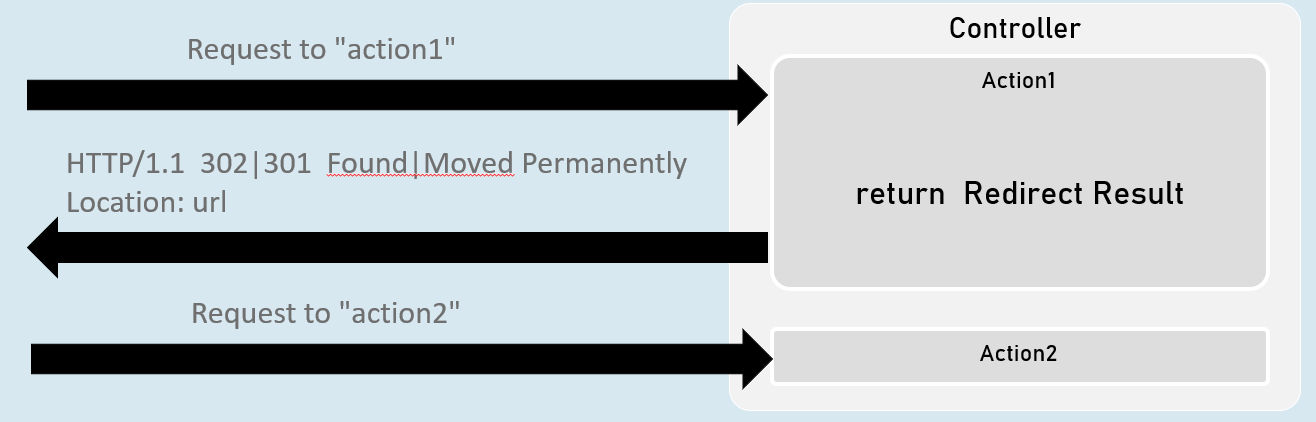
//or

return NotFound();

Redirect Results

Redirect result sends either HTTP 302 or 301 response to the browser, in order to redirect to a specific action or url.

Eg: redirecting from 'action1' to 'action2'.



**RedirectToActionResult**

return new RedirectToActionResult("action", "controller", new { route\_values }, permanent);

**LocalRedirectResult**

return new LocalRedirectResult("local\_url", permanent);

**RedirectResult**

return new RedirectResult("url", permanent);

RedirectToActionResult

Represents response for redirecting from the current action method to another action method, based on action name and controller name.

**302 - Found**

return new RedirectToActionResult("action", "controller", new { route\_values });

//or

return RedirectToAction("action", "controller", new { route\_values });

**301 - Moved Permanently**

return new RedirectToActionResult("action", "controller", new { route\_values }, true);

//or

return RedirectToActionPermanent("action", "controller", new { route\_values });

LocalRedirectResult

•Represents response for redirecting from the current action method to another action method, based on the specified url.

**302 - Found**

return new LocalRedirectResult("url");

//or

return LocalRedirect("url);

**301 - Moved Permanently**

return new LocalRedirectResult("url", true);

//or

return LocalRedirectPermanent("url");

RedirectResult

Represents response for redirecting from the current action method to any other url (either within the same web application or other web application).

**302 - Found**

return new RedirectResult("url");

//or

return Redirect("url);

**301 - Moved Permanently**

return new RedirectResult("url", true);

//or

return RedirectPermanent("url");

# **5. Model Binding and Validations [MVC vs API]**

Model Binding

Model binding is a powerful feature in ASP.NET Core MVC that automates the process of extracting data from various parts of an HTTP request (form data, route values, query strings) and converting it into strongly typed C# objects that you can use directly in your action methods.

**When Model Binding Executes**

Model binding takes place **after** routing has determined which action method to invoke. The model binding system examines the parameters of the selected action method and tries to populate them with values from the incoming request.

**Order of Model Binding**

ASP.NET Core model binding follows a specific order when looking for data sources:

1. **Form Data (POST requests):** Values submitted through HTML forms.
2. **Route Data:** Values extracted from the URL route template (e.g., /products/{id}).
3. **Query String:** Values appended to the URL after a question mark (?).

**Parts of Model Binding**

* **Form Data:** Typically used for submitting data from HTML forms using POST requests.
* **Route Data:** Values captured from the URL segments defined in your route templates.
* **Query String:** Parameters passed in the URL after the question mark (?).

**Query Strings in Detail**

* **Purpose:** To pass parameters to your application through the URL.
* **Syntax:** ?key1=value1&key2=value2 (multiple key-value pairs separated by ampersands).
* **Usage:** Useful for filtering, sorting, and pagination.
* **Example:** /products?category=electronics&sort=price\_desc

**Best Practices (Query Strings):**

* **Limit Sensitive Data:** Avoid passing sensitive information like passwords or credit card details in query strings.
* **Sanitize Input:** Always sanitize and validate query string values to prevent security vulnerabilities.
* **Keep It Simple:** Use clear and meaningful parameter names. Avoid excessively long query strings.
* **Encoding:** Properly encode special characters in query string values.

**Things to Avoid (Query Strings):**

* **Sensitive Data:** Never pass sensitive data like passwords or authentication tokens in query strings.
* **Complex Objects:** Avoid passing complex objects as query strings due to URL length limitations.
* **Overuse:** Don't overload URLs with too many query parameters.

**Route Data in Detail**

* **Purpose:** Capture dynamic values from the URL based on the route template.
* **Syntax:** /products/{id}, where id is a route parameter.
* **Usage:** Essential for RESTful APIs and for creating clean, readable URLs.
* **Example:** /products/12345 (12345 would be the value of the id parameter).

**Best Practices (Route Data):**

* **Choose Clear Names:** Use descriptive names for route parameters.
* **Constraints:** Apply route constraints (e.g., int, guid) to ensure valid data types.
* **Custom Constraints:** Create custom constraints for more complex validation.

Code

1. // HomeController.cs
2. [Route("bookstore/{bookid?}/{isloggedin?}")] // Route with optional parameters
3. public IActionResult Index(int? bookid, bool? isloggedin)
4. {
5. // ... validation logic (same as previous example) ...
6. return Content($"Book id: {bookid}, isloggedin: {isloggedin}", "text/plain");
7. }

In this action method:

* bookid and isloggedin parameters are automatically bound from the query string and route data.

Code

1. // StoreController.cs
2. [Route("store/books/{id}")] // Route with a required parameter
3. public IActionResult Books()
4. {
5. int id = Convert.ToInt32(Request.RouteValues["id"]);
6. return Content($"<h1>Book Store {id}</h1>", "text/html");
7. }

In this action method:

* id is a required parameter.
* id is bound from the route data (Request.RouteValues

[FromQuery] and [FromRoute]

While ASP.NET Core's model binding automatically tries to match action method parameters to different parts of the request (form data, route values, query strings), you can use the [FromQuery] and [FromRoute] attributes to explicitly tell the model binder where to look for specific values.

**[FromQuery]**

* **Purpose:** Instructs the model binder to extract the parameter value from the query string.
* **Usage:** Apply this attribute to action method parameters that you expect to receive values from the query string portion of the URL (the part after the "?").
* **Example:**

1. public IActionResult Index([FromQuery] int page) { ... }

In this example, the page parameter would be bound to the value of the page query parameter in the URL (e.g., /products?page=3).

**[FromRoute]**

* **Purpose:** Instructs the model binder to extract the parameter value from the route data.
* **Usage:** Apply this attribute to action method parameters that you expect to receive values from the route template of the URL.
* **Example:**

1. [Route("products/{id}")]
2. public IActionResult Details([FromRoute] int id) { ... }

In this example, the id parameter would be bound to the value of the id segment in the URL (e.g., /products/123).

Code

1. // HomeController.cs
2. [Route("bookstore/{bookid?}/{isloggedin?}")] // Route with optional parameters
3. public IActionResult Index([FromQuery] int? bookid, [FromRoute] bool? isloggedin)
4. {
5. // ... (rest of the validation and response logic) ...
6. }

In this code:

* **bookid (FromQuery):** The model binder will attempt to retrieve the bookid value exclusively from the query string. If it's not present in the query string, it will be set to null due to the nullable type (int?).
* **isloggedin (FromRoute):** The model binder will specifically look for the isloggedin value in the route data. If the parameter is not present in the route, it will default to null due to the nullable boolean type (bool?).

**Combined Binding:**

By using both [FromQuery] and [FromRoute] on different parameters in the same action method, you can effectively bind values from both the query string and route data simultaneously.

**Notes**

* **Explicit Binding:** Use [FromQuery] and [FromRoute] for explicit control over where the model binder gets values for your action method parameters.
* **Flexibility:** You can combine both attributes in the same action to bind from multiple sources.
* **Default Behavior:** Even without these attributes, ASP.NET Core's model binding will try to intelligently determine the binding source. However, using these attributes makes your code more explicit and less prone to unexpected behavior.
* **Type Conversion:** The model binder automatically attempts to convert values to the appropriate data types for your action method parameters.

Model Classes

In ASP.NET Core MVC, model classes are the foundation for representing the data your application works with. They typically mirror the structure of your data, whether it comes from a database, an API, or other sources.

* **Purpose:**
  + **Structure:** Provide a well-defined structure for your data, including properties that correspond to the fields or attributes of your data entities.
  + **Validation:** Enforce data validation rules using attributes like [Required], [StringLength], and [Range].
  + **Organization:** Keep your application's data logic organized and maintainable.
* **Example Model Class:**

1. // Book.cs (Model)
2. namespace IActionResultExample.Models
3. {
4. public class Book
5. {
6. public int? BookId { get; set; }
7. public string? Author { get; set; }
9. public override string ToString() // For easy display in this example
10. {
11. return $"Book object - Book id: {BookId}, Author: {Author}";
12. }
13. }
14. }

Model Binding with Model Classes

Model binding with model classes simplifies the process of populating your model objects with data from incoming HTTP requests. Instead of manually extracting values from query strings, route data, or form data, you can directly use the model class as a parameter in your action method.

* **How it Works:**
  1. **Action Parameter:** Declare an action method parameter of your model class type.
  2. **Model Binding:** The model binder automatically maps incoming request data to the properties of your model class based on their names.
  3. **Attribute Usage:** You can use attributes like [FromQuery], [FromRoute], and [FromBody] to specify where the model binder should look for the data for each property.

Code

1. // HomeController.cs
2. [Route("bookstore/{bookid?}/{isloggedin?}")]
3. //Url: /bookstore/1/false?bookid=20&isloggedin=true&author=harsha
4. public IActionResult Index([FromQuery] int? bookid, [FromRoute] bool? isloggedin, Book book)
5. {
6. // ... validation and response logic ...
7. return Content($"Book id: {bookid}, Book: {book}", "text/plain");
8. }

In this code:

1. **Model Class Parameter:** The action method Index has a parameter named book.
2. **[FromQuery] Attribute:** The BookId property of the Book class has the [FromQuery] attribute, indicating that its value should be retrieved from the query string.
3. **Automatic Binding:** When a request like /bookstore/1/false?bookid=20&isloggedin=true&author=harsha comes in:
   * bookid (int?) will be 20 (from the query string, due to [FromQuery]).
   * isloggedin (bool?) will be true (from the route data, due to [FromRoute]).
   * book.Author (string?) will be "harsha" (from the query string, because no attribute was specified for the Author property so it defaults to looking in the query string).

**Notes**

* **Simplified Code:** Model binding reduces boilerplate code for extracting data from requests.
* **Strong Typing:** You work with strongly typed model objects in your actions.
* **Clear Intent:** Attributes like [FromQuery], [FromRoute], and [FromBody] make your code more explicit.
* **Automatic Conversion:** The model binder tries to convert request data to match the types of your model properties.
* **Complex Types:** You can bind complex objects from JSON or XML data in request bodies (using [FromBody]).
* **Validation:** Leverage model validation attributes to ensure data integrity.

URL Encoding

URL encoding (or percent-encoding) is a mechanism to encode special characters in URLs that are not allowed in their raw form. This encoding is essential to ensure that URLs are transmitted correctly and that the server interprets them correctly.

* **Special Characters:** Characters like spaces, ampersands (&), question marks (?), and non-ASCII characters need to be encoded.
* **Encoding Format:** Special characters are replaced with a percent sign (%) followed by two hexadecimal digits representing their ASCII code.
* **Example:** A space is encoded as %20, and an ampersand is encoded as %26.

**Content Types for Form Submission**

1. **application/x-www-form-urlencoded:**
   * **Purpose:** The default encoding for HTML forms. It encodes form data as key-value pairs separated by ampersands (&) and with equal signs (=) between keys and values. Spaces are converted to plus signs (+).
   * **Usage:** Suitable for simple forms with text data.
   * **Limitations:** Not efficient for large amounts of data or binary data (like file uploads).
2. **multipart/form-data:**
   * **Purpose:** Designed for submitting forms with files or large amounts of data. Each form field is sent as a separate part, with its own content type and headers.
   * **Usage:** Essential for file uploads.
   * **Benefits:** Handles binary data efficiently and can support larger payloads.
3. **form-data:**
   * **Purpose:** A newer and more flexible format for form submissions that can handle both simple and complex data, including files.
   * **Usage:** Offers a more modern alternative to multipart/form-data.
   * **Benefits:** Similar to multipart/form-data but with a more streamlined structure.

**Notes**

* **URL Encoding:** Essential for ensuring that URLs are properly formed and interpreted.
* **Form Submission:**
  + application/x-www-form-urlencoded: Default for simple forms.
  + multipart/form-data or form-data: Required for file uploads and larger payloads.
* **Model Binding:** ASP.NET Core MVC automatically handles binding form data (submitted via POST) to your model classes based on the Content-Type header.

Model Validation

Model validation is the process of verifying that the data submitted to your ASP.NET Core MVC application meets your defined criteria. This prevents invalid or malicious data from entering your system and helps maintain the integrity of your application's data.

**Why Model Validation Matters**

* **Security:** Protects against common attacks like SQL injection, cross-site scripting (XSS), and overposting.
* **Data Integrity:** Ensures that the data stored in your database or used in your application logic is valid.
* **User Experience:** Provides immediate feedback to users, guiding them to correct input errors.

**Best Practices**

1. **Validate on Both Sides:** Validate data both on the client-side (using JavaScript) for immediate feedback and on the server-side for security (as client-side validation can be bypassed).
2. **Use Data Annotations:** Leverage the built-in data annotation attributes provided by the System.ComponentModel.DataAnnotations namespace to express validation rules concisely.
3. **Custom Validation Attributes:** Create custom validation attributes for more complex or domain-specific rules.
4. **Model State:** Always check the ModelState.IsValid property in your controller actions before processing the data. If it's invalid, return an appropriate error response.
5. **Display Error Messages:** Clearly display error messages to the user, indicating which fields are invalid and why.

**Essential Data Annotations**

Here are some of the most commonly used data annotation attributes:

* **[Required]:** The field must not be null or empty.
* **[StringLength]:** Restricts the maximum or minimum length of a string.
* **[Range]:** Specifies a numeric range within which the value must fall.
* **[RegularExpression]:** Validates the value against a regular expression pattern.
* **[EmailAddress]:** Verifies that the value is a valid email address format.
* **[Compare]:** Compares the value of one property to another (e.g., password confirmation).
* **[Phone]:** Validates a phone number format.
* **[Url]:** Validates a URL format.

Model State in Controllers

The ModelState object in your controllers is crucial for validation. It tracks the validation state of your model after the model binder has attempted to populate it from the request.

* **ModelState.IsValid:** A boolean property that indicates whether all validation rules passed (true) or if there were any errors (false).
* **ModelState.AddModelError:** Manually add a model error for a specific property.
* **Error Messages:** Retrieve error messages associated with specific properties.

Code

1. // Person.cs (Model)
2. public class Person
3. {
4. // ... other properties
6. [Required(ErrorMessage = "{0} can't be blank")]
7. [Compare("Password", ErrorMessage = "{0} and {1} do not match")]
8. [Display(Name = "Re-enter Password")]
9. public string? ConfirmPassword { get; set; }
11. // ... other properties
12. }
14. // (In your controller action)
15. public IActionResult Create(Person person)
16. {
17. if (!ModelState.IsValid)
18. {
19. return View(person); // Return to the view with validation errors
20. }
22. // Model is valid, proceed with saving data
23. }

In this code:

1. **Data Annotations:** The Person model uses data annotations to enforce validation rules.
2. **Model State Check:** The controller action checks ModelState.IsValid. If false, the original view is re-rendered with the model object containing validation errors, allowing the user to correct them.
3. **Error Display:** The view typically uses the @Html.ValidationSummary() and @Html.ValidationMessageFor() helper methods to display error messages to the user.

Custom Validation with ValidationAttribute

While ASP.NET Core's built-in validation attributes cover a wide range of scenarios, you'll inevitably encounter validation rules specific to your application's business logic. Custom validation attributes, derived from the ValidationAttribute class, empower you to create these tailored validations.

**Key Steps**

1. **Inherit from ValidationAttribute:** Create a class that inherits from ValidationAttribute.
2. **Override IsValid:** The core of your custom validation logic lies in the IsValid method. This method receives the value to be validated and a ValidationContext object (containing additional information about the model).
3. **Return ValidationResult:**
   * If the value is valid, return ValidationResult.Success.
   * If the value is invalid, return a new ValidationResult object with your custom error message.

Code

1. public class DateRangeValidatorAttribute : ValidationAttribute
2. {
3. public string OtherPropertyName { get; set; }
5. // Constructor
6. public DateRangeValidatorAttribute(string otherPropertyName)
7. {
8. OtherPropertyName = otherPropertyName;
9. }
11. protected override ValidationResult? IsValid(object? value, ValidationContext validationContext)
12. {
13. if (value != null)
14. {
15. // Get the "to\_date"
16. DateTime toDate = Convert.ToDateTime(value);
18. // Get the "from\_date"
19. var otherProperty = validationContext.ObjectType.GetProperty(OtherPropertyName);
21. if (otherProperty != null)
22. {
23. DateTime fromDate = Convert.ToDateTime(otherProperty.GetValue(validationContext.ObjectInstance));
25. if (fromDate > toDate)
26. {
27. return new ValidationResult(ErrorMessage, new string[] { OtherPropertyName, validationContext.MemberName }); // Indicate the specific properties involved in the error
28. }
29. else
30. {
31. return ValidationResult.Success;
32. }
33. }
34. return null; // Return null if otherProperty is null
35. }
36. return null; // Return null if value is null
37. }
38. }

* **Purpose:** Ensures that a date (e.g., ToDate) is not earlier than another date (FromDate).
* **OtherPropertyName:** Specifies the name of the property to compare against (in this case, FromDate).
* **IsValid:**
  + It retrieves the values of both properties using reflection.
  + It compares the dates and returns an error message if toDate is earlier than fromDate.
  + The error message includes the names of both properties, providing clear feedback to the user.

Code

1. public class MinimumYearValidatorAttribute : ValidationAttribute
2. {
3. public int MinimumYear { get; set; } = 2000;
4. public string DefaultErrorMessage { get; set; } = "Year should not be less than {0}";
6. // ... (constructors) ...
8. protected override ValidationResult? IsValid(object? value, ValidationContext validationContext)
9. {
10. if (value != null)
11. {
12. DateTime date = (DateTime)value;
13. if (date.Year >= MinimumYear)
14. {
15. return ValidationResult.Success;
16. }
17. else
18. {
19. return new ValidationResult(string.Format(ErrorMessage ?? DefaultErrorMessage, MinimumYear)); // Use custom or default error message
20. }
21. }
23. return null;
24. }
25. }

* **Purpose:** Ensures that a date (e.g., DateOfBirth) is not earlier than a specified year.
* **MinimumYear:** Sets the minimum allowed year (defaulting to 2000).
* **DefaultErrorMessage:** Provides a default error message if a custom message isn't provided.
* **IsValid:**
  + It checks if the year of the given date is greater than or equal to the minimum year.

IValidatableObject

While data annotations ([Required], [StringLength], etc.) provide a concise way to define validation rules on individual model properties, the IValidatableObject interface allows you to perform more complex, model-level validation logic that spans multiple properties or depends on the entire model's state.

**Notes**

* **Interface:** IValidatableObject is an interface with a single method: Validate(ValidationContext context).
* **Validate Method:** This method is called by the model binder after individual property-level validations (data annotations) have been checked.
* **Yielding Errors:** Within the Validate method, you can yield ValidationResult objects for any errors you find. This allows you to report multiple errors for the entire model at once.
* **Model State Integration:** The errors you yield are automatically added to the ModelState object, making them available for error display in your views.

**When to Use IValidatableObject**

* **Cross-Property Validation:** When validation logic depends on the values of multiple properties (e.g., "Start Date" must be before "End Date").
* **Complex Business Rules:** When your validation rules involve complex logic or database lookups.
* **Customizable Errors:** When you want more control over the error messages displayed to the user.

Code

1. // Person.cs (Model)
2. public class Person : IValidatableObject
3. {
4. // ... (properties with data annotations) ...
6. public DateTime? DateOfBirth { get; set; }
7. public int? Age { get; set; } // New property
9. // ... (other properties and methods) ...
11. public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
12. {
13. if (DateOfBirth.HasValue == false && Age.HasValue == false)
14. {
15. yield return new ValidationResult("Either of Date of Birth or Age must be supplied", new[] { nameof(Age) }); // Yield an error
16. }
17. }
18. }

In this example:

1. **Implementation of IValidatableObject:** The Person class now implements IValidatableObject.
2. **Validate Method:**
   * It checks if either DateOfBirth or Age is provided. If neither is present, it yields a ValidationResult indicating that at least one of these properties must be supplied.
   * Notice how the error message is specifically associated with the Age property using new[] { nameof(Age) }. This helps target the error message to the correct field in your view.
3. **Model State Update:** When you use this model in your controller, the validation errors from the Validate method will automatically be added to the ModelState, and you can check ModelState.IsValid to determine if the model is valid.

**Notes**

* **Model-Level Validation:** IValidatableObject is ideal for validation logic that goes beyond individual properties.
* **Yielding Errors:** Use the yield return statement to return multiple validation errors from the Validate method.
* **Error Targeting:** Associate error messages with specific properties for clear user feedback.
* **Integration with Data Annotations:** IValidatableObject works in conjunction with data annotations, providing a comprehensive validation approach.

[Bind] and [BindNever] Attributes

Model binding is powerful, but sometimes you want more granular control over which properties get populated from incoming request data. This is where [Bind] and [BindNever] come in.

**[Bind] Attribute**

* **Purpose:** Explicitly include specific properties for model binding.
* **Usage:** Apply this attribute to your action method parameter (e.g., the model class) and provide a list of property names as arguments.
* **Example:**

1. [HttpPost]
2. public IActionResult Create([Bind("Title", "Description")] Product product)
3. {
4. // Only the Title and Description properties will be bound from the request.
5. }

In this example, even if the incoming request contains data for other properties of the Product class (like Price or Category), they will be ignored during model binding.

**[BindNever] Attribute**

* **Purpose:** Exclude specific properties from model binding.
* **Usage:** Apply this attribute directly to model properties that you never want to be bound from the request.
* **Example:**

1. public class Product
2. {
3. // ... other properties
5. [BindNever]
6. public DateTime CreatedAt { get; set; } // Never bind from request
7. }

In this example, the CreatedAt property will always retain its default value, regardless of whether the incoming request contains data for it.

Code

1. // Person.cs (Model)
2. public class Person : IValidatableObject
3. {
4. // ... (other properties)
6. [BindNever] // This property will not be bound during model binding
7. public DateTime? DateOfBirth { get; set; }
9. // ... (other properties and methods) ...
10. }
12. // HomeController.cs
13. [Route("register")]
14. public IActionResult Index(Person person)
15. {
16. // ... (validation and response logic) ...
17. }

In this code:

1. **DateOfBirth (BindNever):** The [BindNever] attribute on the DateOfBirth property tells the model binder to completely ignore any data for this property coming from the request. Even if the incoming request contains a value for DateOfBirth, it won't be assigned to the model property.

**Notes**

* **Security:** [BindNever] is a valuable tool for preventing overposting attacks, where an attacker tries to submit data for properties that shouldn't be modifiable from a client.
* **Explicit Control:** [Bind] and [BindNever] give you precise control over which model properties are populated from incoming requests.
* **Default Behavior:** Without these attributes, the model binder attempts to bind all public properties of your model class.
* **Complex Types:** You can use [Bind] on nested complex properties to specify which properties within those objects should be bound.

[FromBody] Attribute

The [FromBody] attribute is a crucial tool in ASP.NET Core MVC's model binding arsenal, designed to handle scenarios where the incoming data is contained within the body of an HTTP request. This is especially common when working with APIs and modern web applications that frequently exchange data in formats like JSON or XML.

**How It Works**

1. **Request Body Identification:** When a request arrives, the model binding middleware examines the Content-Type header to determine the format of the data in the request body. Typically, this is either application/json for JSON data or application/xml for XML data.
2. **Input Formatter Selection:** Based on the Content-Type, the middleware selects an appropriate input formatter. Input formatters are responsible for deserializing the raw request body into a format that the model binder can understand.
3. **Model Binding:** The model binder takes the deserialized data and attempts to map it to the properties of your model class. This mapping is typically done based on property names, but you can customize it using various model binding attributes.
4. **Validation:** After binding, the model undergoes validation to ensure it adheres to the rules defined by data annotations or custom validation logic.

**Benefits of [FromBody]**

* **Complex Data Handling:** Easily bind complex objects with nested properties from JSON or XML payloads.
* **Separation of Concerns:** The input formatter handles deserialization, keeping your controller actions clean.
* **API-Friendly:** Aligns well with RESTful API practices, where data is often transmitted in the request body.

Code

1. // HomeController.cs
2. [Route("register")]
3. // Example JSON: { "PersonName": "William", "Email": "william@example.com", "Phone": "123456", "Password": "william123", "ConfirmPassword": "william123" }
4. public IActionResult Index([FromBody] Person person)
5. {
6. if (!ModelState.IsValid)
7. {
8. // ... (handle validation errors) ...
9. }
11. return Content($"{person}");
12. }

In this code:

1. **[FromBody] Attribute:** The [FromBody] attribute on the person parameter instructs the model binder to look for the data in the request body.
2. **JSON Deserialization:** If the request has a Content-Type of application/json, the built-in JSON input formatter will deserialize the JSON data in the request body into a Person object.
3. **Model Validation:** The ModelState.IsValid check ensures the deserialized Person object meets your validation criteria.
4. **Successful Response:** If the model is valid, the Content result returns a string representation of the Person object.

**Important Considerations**

* **Single [FromBody] Parameter:** ASP.NET Core model binding allows only one parameter per action method to be decorated with [FromBody]. This is because the request body is typically a single stream of data.
* **Content-Type:** The Content-Type header of the request must match the expected format (e.g., application/json) for the correct input formatter to be used.
* **Security:** Always validate and sanitize data from the request body to protect against vulnerabilities like overposting and injection attacks.

Input Formatters

Input formatters are specialized components in ASP.NET Core MVC responsible for deserializing data from the body of HTTP requests. When a request arrives with a payload, the input formatter decodes this data into a format (e.g., C# objects, collections) that your action methods can readily work with.

**How Input Formatters Work**

1. **Content Negotiation:** The model binding process begins with content negotiation, where ASP.NET Core examines the Content-Type header of the request to determine the format of the incoming data (e.g., JSON, XML).
2. **Input Formatter Selection:** Based on the Content-Type, ASP.NET Core selects an appropriate input formatter that knows how to handle that specific data format.
3. **Deserialization:** The selected input formatter deserializes the raw data from the request body into C# objects, collections, or other supported types.
4. **Model Binding:** The deserialized data is then passed to the model binder, which populates the parameters of your action method.

**Common Input Formatters**

* **NewtonsoftJsonInputFormatter:** Handles JSON (JavaScript Object Notation) data using the popular Newtonsoft.Json library.
* **SystemTextJsonInputFormatter:** Handles JSON data using the built-in System.Text.Json serializer.
* **XmlSerializerInputFormatter:** Handles XML (Extensible Markup Language) data using the XmlSerializer.

**Configuring Input Formatters**

1. **Default Formatters:** ASP.NET Core MVC includes NewtonsoftJsonInputFormatter as a default formatter.
2. **Additional Formatters:** You can add support for other formatters (like XmlSerializerInputFormatter) by explicitly registering them in your application's startup configuration.

Code

1. var builder = WebApplication.CreateBuilder(args);
2. builder.Services.AddControllers().AddXmlSerializerFormatters();
3. // ... other configuration ...

In this code, the .AddXmlSerializerFormatters() extension method registers the XmlSerializerInputFormatter, enabling your application to handle requests with an application/xml content type.

**Using Input Formatters**

* **Implicit Binding:** If the Content-Type header of the request matches a supported format, the corresponding input formatter is automatically used. You don't need to explicitly specify which formatter to use in your action method.
* **Explicit Binding ([FromBody]):** You can use the [FromBody] attribute on an action method parameter to explicitly tell the model binder to look for the data in the request body. This is often used when you have complex objects that need to be deserialized.

**Important Considerations**

* **Content Negotiation:** The success of model binding depends on the client sending a valid Content-Type header that your application supports.
* **Error Handling:** Handle potential deserialization errors gracefully. If the input formatter cannot parse the request body, return an appropriate error response (e.g., 400 Bad Request).
* **Security:** Always validate and sanitize data deserialized from the request body to protect against security vulnerabilities.
* **Custom Input Formatters:** For highly specialized scenarios or custom data formats, you can create your own input formatters by implementing the IInputFormatter interface.

Custom Model Binders

While ASP.NET Core's default model binder is quite versatile, it might not always meet your specific needs. This is where custom model binders step in, allowing you to precisely define how data is extracted from incoming requests and mapped onto your model properties.

**Purpose**

* **Flexibility:** Handle complex or custom data formats that the default model binder doesn't understand.
* **Custom Logic:** Implement specific business rules or data transformations during binding.
* **Complete Control:** Take full control over the binding process, from parsing the raw data to populating your model object.

**Implementing IModelBinder**

To create a custom model binder, you implement the IModelBinder interface:

1. public interface IModelBinder
2. {
3. Task BindModelAsync(ModelBindingContext bindingContext);
4. }

The core of your custom logic resides in the BindModelAsync method, where you:

1. Retrieve raw data from the bindingContext.ValueProvider.
2. Parse and validate the data according to your requirements.
3. Create an instance of your model class and populate its properties.
4. Set the bindingContext.Result to ModelBindingResult.Success(yourModelInstance).

Code

1. // PersonModelBinder.cs
2. public class PersonModelBinder : IModelBinder
3. {
4. public Task BindModelAsync(ModelBindingContext bindingContext)
5. {
6. Person person = new Person();
8. // ... (Logic to extract and populate properties from the ValueProvider) ...
10. bindingContext.Result = ModelBindingResult.Success(person);
11. return Task.CompletedTask;
12. }
13. }

In this example, PersonModelBinder:

1. Creates a new Person object.
2. Extracts values for different properties (e.g., PersonName, Email, Phone) from the ValueProvider (which can access form data, query string, route data, etc.).
3. Performs some basic transformations (concatenating FirstName and LastName).
4. Sets the model binding result to indicate success and return the populated Person object.

Model Binder Providers

To inform ASP.NET Core that you want to use your custom model binder for a specific type, you create a model binder provider. This provider implements the IModelBinderProvider interface.

Code

1. // (This code is not provided in your original request, but it's a common way to register a custom model binder)
3. public class PersonBinderProvider : IModelBinderProvider
4. {
5. public IModelBinder? GetBinder(ModelBinderProviderContext context)
6. {
7. if (context.Metadata.ModelType == typeof(Person))
8. {
9. return new BinderTypeModelBinder(typeof(PersonModelBinder));
10. }
11. return null;
12. }
13. }

This provider checks if the model type is Person, and if so, it returns an instance of your PersonModelBinder.

**Registration and Usage**

1. // Program.cs (or Startup.cs)
2. builder.Services.AddControllers(options => {
3. options.ModelBinderProviders.Insert(0, new PersonBinderProvider());
4. });

By inserting your PersonBinderProvider at index 0, you ensure it takes precedence over the default model binders.

**Sample Request Data (Postman)**

To test this, you can use Postman to send a POST request to your controller action with the following JSON body:

JSON

1. {
2. "FirstName": "John",
3. "LastName": "Doe",
4. "Email": "john.doe@example.com",
5. "Phone": "1234567890",
6. "Password": "password123",
7. "ConfirmPassword": "password123",
8. "Price": 59.99,
9. "DateOfBirth": "2000-01-01"
10. }

**Important Considerations**

* **Complexity:** Custom model binders can become complex, so use them judiciously when the default behavior is insufficient.
* **Testability:** Write unit tests for your custom model binders to ensure they function correctly in different scenarios.
* **Performance:** Be mindful of performance when implementing complex parsing or validation logic in your binder.
* **Error Handling:** Handle potential exceptions during data extraction and validation to provide informative error responses.
* **Alternative Approaches:** In some cases, using a custom input formatter in conjunction with a simpler model binder might be a more suitable approach.

Collection Binding

ASP.NET Core's model binding isn't limited to simple properties; it can gracefully handle collections like lists and arrays within your model classes. This is especially useful when dealing with forms that allow users to input multiple values for a single field (e.g., selecting multiple interests from a checkbox list) or when working with data from APIs that naturally return collections.

**How Collection Binding Works**

1. **Collection Property in the Model:** Your model class should have a property that's a collection type (e.g., List<T>, T[]).
2. **Naming Convention:** The incoming request parameters should follow a specific naming convention to indicate which values belong to the collection.
3. **Model Binder Magic:** The model binder automatically recognizes the naming convention and populates the collection property accordingly.

**Naming Conventions for Collection Binding**

* **Indexed:** items[0], items[1], items[2], ... (Used for lists and arrays)
* **Same Name:** items, items, items, ... (Used for collections like ICollection<T>)

Code

1. // Person.cs (Model)
2. public class Person
3. {
4. // ... other properties
5. public List<string?> Tags { get; set; } = new List<string?>(); // Collection property
6. }
8. // HomeController.cs
9. public IActionResult Index(Person person)
10. {
11. // ... validation and response logic ...
12. return Content($"Person: {person}, Tags: {string.Join(",", person.Tags)}", "text/plain");
13. }

**Sample Request Data (Postman)**

To test this, you can send the following JSON request data to your register endpoint using Postman:

JSON

1. {
2. "PersonName": "Alice",
3. "Email": "alice@example.com",
4. "Phone": "1234567890",
5. "Password": "alicepassword",
6. "ConfirmPassword": "alicepassword",
7. "Price": 59.99,
8. "DateOfBirth": "1995-03-15",
9. "Tags": ["music", "reading", "coding"]
10. }

**Response** The response should look like:

1. Person object - Person name: Alice, Email: alice@example.com, Phone: 1234567890, Password: alicepassword, Confirm Password: alicepassword, Price: 59.99, Tags: music,reading,coding

**Detailed Explanation**

1. **Collection Property:** The Person model has a Tags property, which is a List<string?>.
2. **JSON Data:** The request body includes a Tags array with string values.
3. **Model Binding:** The model binder automatically recognizes the Tags array in the JSON data and populates the Person.Tags list with the corresponding values.

**Notes**

* **Naming:** Follow the correct naming convention (indexed or same name) for your collection parameters in the request data.
* **Flexibility:** You can bind to a variety of collection types, including lists, arrays, and custom collections that implement ICollection<T>.
* **Validation:** Apply validation attributes to your collection properties (e.g., [Required], [MaxLength]) to ensure data integrity.
* **Custom Model Binders:** If you have complex collection binding scenarios, you can create custom model binders to handle them.

[FromHeader] Attribute

In ASP.NET Core MVC, the [FromHeader] attribute is used to instruct the model binder to fetch values for action method parameters directly from HTTP request headers. HTTP headers are key-value pairs that provide metadata about the request, such as the client's browser type (User-Agent), accepted content types (Accept), and authorization tokens.

**How [FromHeader] Works**

1. **Header Identification:** When a request arrives, ASP.NET Core's model binding system identifies action parameters marked with the [FromHeader] attribute.
2. **Header Extraction:** It then examines the request headers to locate the headers that match the names specified in the [FromHeader] attribute.
3. **Value Assignment:** If the matching header is found, its value is assigned to the corresponding action parameter. If the header is not present or its value cannot be converted to the parameter's type, the model state will be marked as invalid.

**Why Use [FromHeader]?**

* **Access to Metadata:** HTTP headers contain valuable information about the client, request, and the data being transmitted.
* **Custom Parameters:** You can define custom headers to pass additional data to your API.
* **Security:** Headers are often used for transmitting authentication tokens and other security-related information.
* **Content Negotiation:** Headers like Accept are used to determine the preferred format for the response (e.g., JSON, XML).

Code

1. // HomeController.cs
2. [Route("register")]
3. public IActionResult Index(Person person, [FromHeader(Name = "User-Agent")] string UserAgent)
4. {
5. // ... (model validation logic) ...
7. return Content($"{person}, {UserAgent}"); // Include User-Agent in the response
8. }

In this code:

1. **UserAgent Parameter:** The action method now includes a UserAgent parameter with the [FromHeader] attribute. The Name property of the attribute is set to "User-Agent," indicating that this parameter should be bound to the value of the User-Agent header.
2. **Header Extraction:** When a request is made to the /register endpoint, the model binder will look for the User-Agent header in the request and assign its value to the UserAgent parameter.
3. **Response:** The Content result now includes both the person's information (from the request body) and the value of the User-Agent header in the response.

**Sample Request Data (Postman)**

To test this, you would send a POST request to the /register endpoint using Postman, with the same JSON body as before, but this time, you would also need to add a User-Agent header in the Headers tab with a value like:

1. Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3

**Important Considerations**

* **Case-Insensitivity:** Header names are case-insensitive, so you can use [FromHeader(Name = "user-agent")] or [FromHeader(Name = "USER-AGENT")].
* **Multiple Headers:** You can use [FromHeader] on multiple parameters to bind values from different headers.
* **Default Values:** If a header is not present in the request, you can specify a default value for the parameter using the ? operator (e.g., string? UserAgent).
* **Alternative:** If you need to access multiple headers or have more complex header parsing logic, consider using Request.Headers directly in your action method.

Key Points to Remember

**1. Model Binding: Bridging HTTP and C#**

* **Purpose:** Automatically maps data from HTTP requests (form data, route values, query strings, headers, body) to action method parameters or model properties.
* **Benefits:** Reduces boilerplate code, provides strong typing, and simplifies data handling in actions.
* **Process:**
  1. **Request Analysis:** Inspects the request's content type and method.
  2. **Value Provider:** Creates a value provider to access data from different sources.
  3. **Model Binder Selection:** Chooses the appropriate model binder based on the parameter type and attributes.
  4. **Property Mapping:** Maps values from the value provider to model properties based on name matching and attributes.

**2. Model Validation: Ensuring Data Integrity**

* **Purpose:** Ensures that data submitted to your application meets predefined criteria before processing.
* **Why It Matters:** Enhances security, maintains data integrity, and improves user experience.
* **Approaches:**
  + **Data Annotations:** Use attributes like [Required], [StringLength], [Range], etc. (from System.ComponentModel.DataAnnotations) to decorate model properties.
  + **IValidatableObject:** Implement this interface to perform custom model-level validation logic.
  + **Custom Validation Attributes:** Create your own attributes inheriting from ValidationAttribute for more complex rules.

**3. Model State:**

* **Centralized Validation:** The ModelState object tracks the validation state of your model after binding.
* **ModelState.IsValid:** A boolean property indicating whether the model is valid or contains errors.
* **ModelState.AddModelError:** Add custom error messages to the ModelState.

**4. Attributes: Fine-Tuning Model Binding and Validation**

* **[FromQuery]:** Binds parameters from the query string.
* **[FromRoute]:** Binds parameters from the route data (URL segments).
* **[FromBody]:** Binds complex objects from the request body (JSON, XML).
* **[FromHeader]:** Binds parameters from HTTP headers.
* **[Bind]:** Explicitly includes specific properties for binding.
* **[BindNever]:** Excludes specific properties from binding (prevents overposting).

**5. Custom Model Binders**

* **Purpose:** Create your own logic to extract and map data to models when the default behavior is insufficient.
* **IModelBinder Interface:** Implement this interface to define your custom binding logic.
* **ModelBindingContext:** This context object provides access to the value providers, model metadata, and other relevant information.

**Additional Tips**

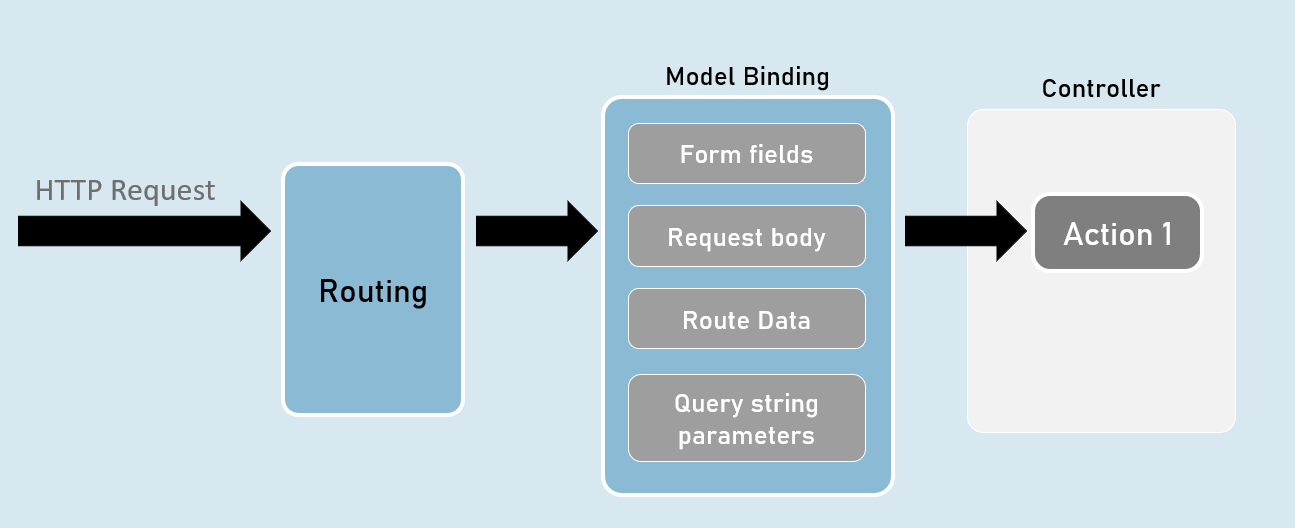
* **Default Model Binder:** Understand the default model binding behavior and when you need customization.
* **Input Formatters:** Know how input formatters work to deserialize request bodies in different formats (JSON, XML).
* **Collection Binding:** Be familiar with how to bind collections (lists, arrays) using proper naming conventions.
* **Error Handling:** Always check ModelState.IsValid in your actions and handle invalid model states gracefully.
* **Security:** Prioritize security by validating and sanitizing input data to prevent attacks.

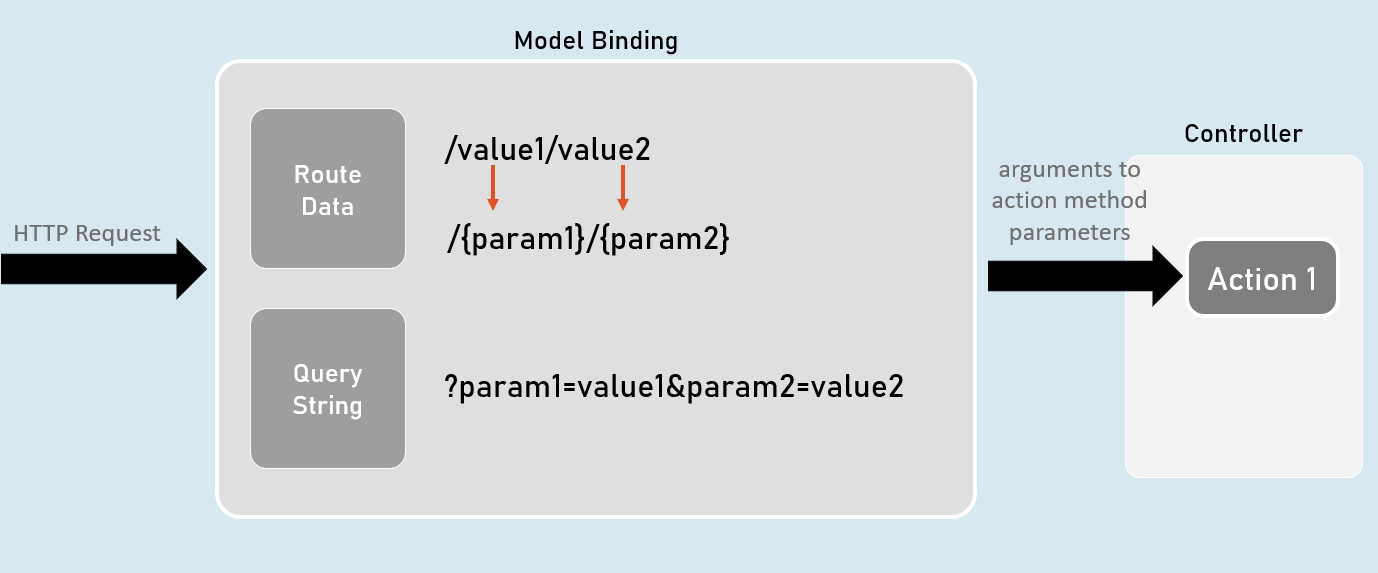
Model Binding

Model Binding is a feature of asp.net core that reads values from http requests and pass them as arguments to the action method.

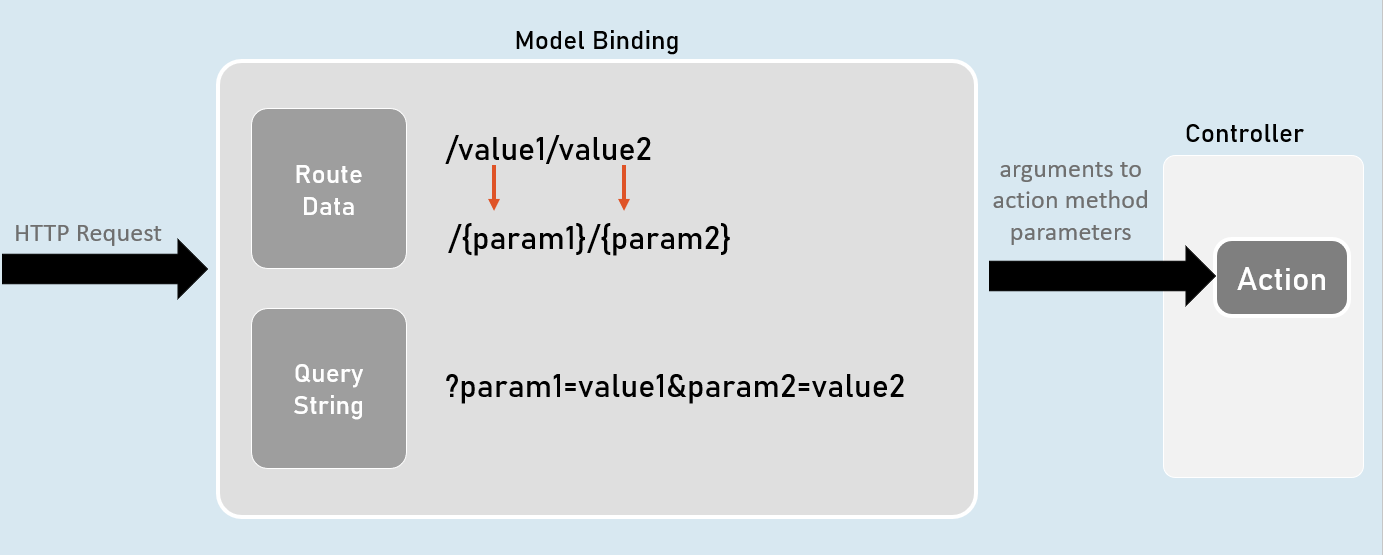


QueryString vs RouteData





[FromQuery] and [FromRoute]

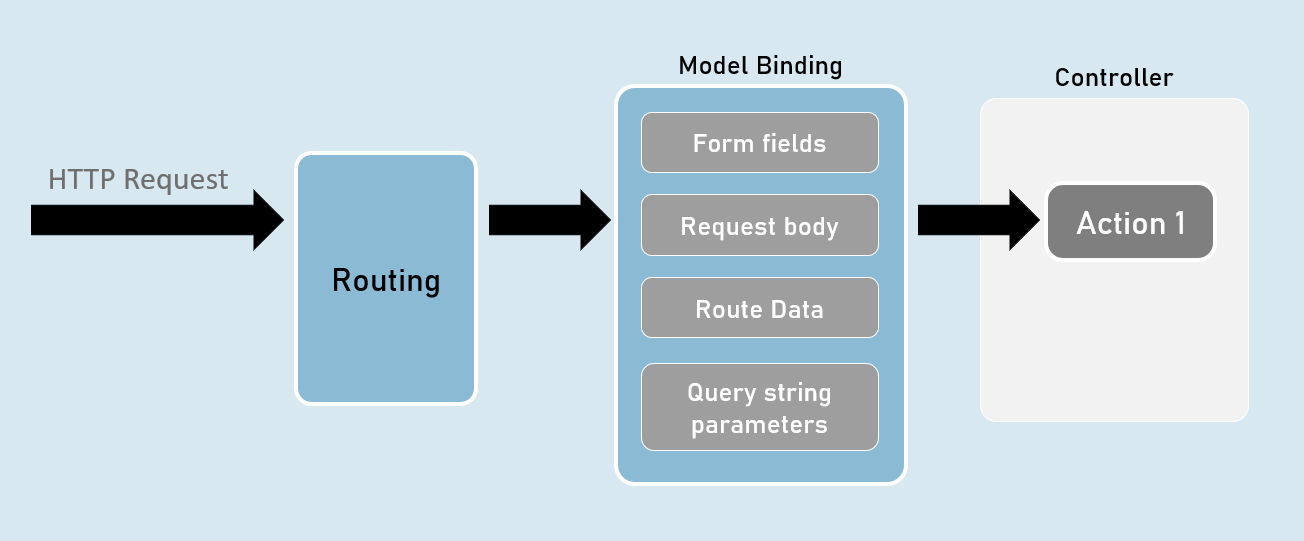


[FromQuery]

1. //gets the value from query string only
2. public IActionResult ActionMethodName( [FromQuery] type parameter)
3. {
4. }

[FromRoute]

1. //gets the value from route parameters only
2. public IActionResult ActionMethodName( [FromRoute] type parameter)
3. {
4. }



Models

Model is a class that represents structure of data (as properties) that you would like to receive from the request and/or send to the response.

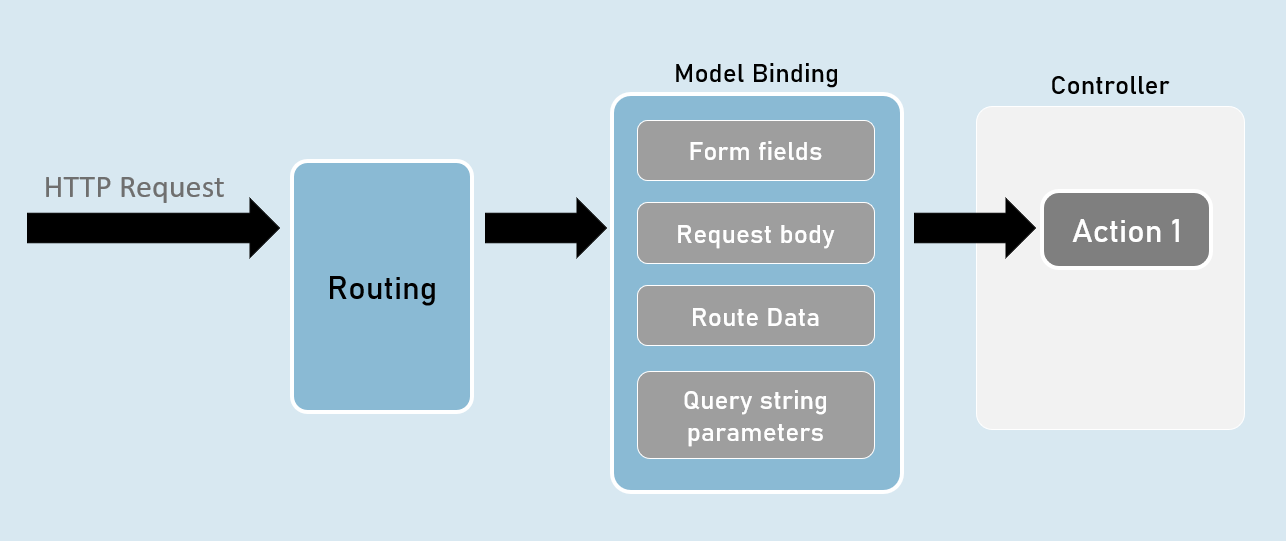
Also known as POCO (Plain Old CLR Objects).



Model

1. class ClassName
2. {
3. public type PropertyName { get; set; }
4. }

form-urlencoded and form-data



form-urlencoded (default)

**Request Headers**

Content-Type: application/x-www-form-urlencoded

**Request Body**

param1=value1&param2=value2

form-data

**Request Headers**

Content-Type: multipart/form-data

**Request Body**

--------------------------d74496d66958873e

Content-Disposition: form-data; name="param1"

value1

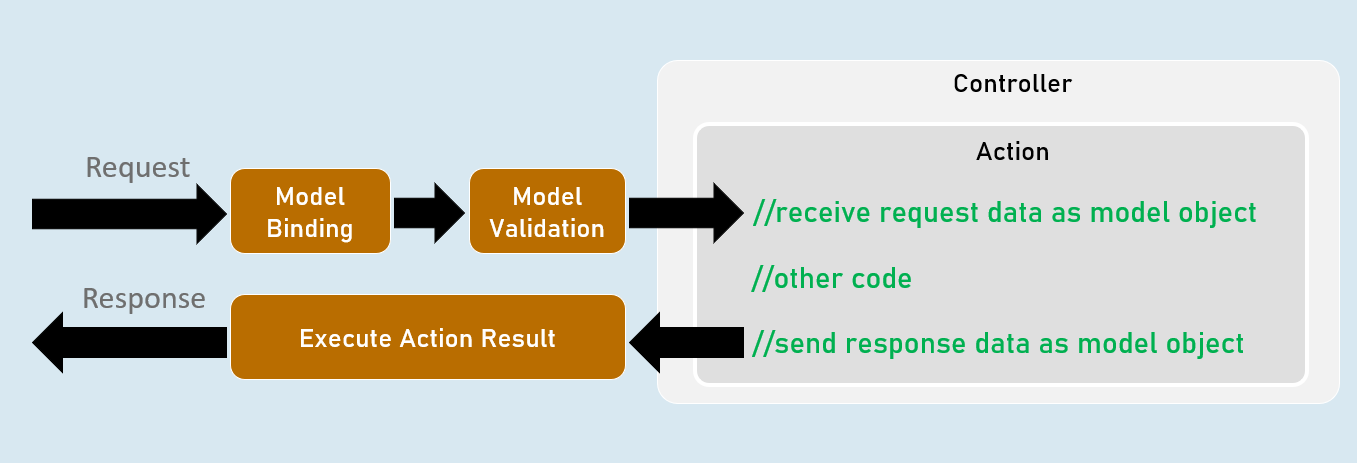
--------------------------d74496d66958873e

Content-Disposition: form-data; name="param2"

value2

Model Validation

1. class ClassName
2. {
3. [Attribute] //applies validation rule on this property
4. public type PropertyName { get; set; }
5. }



ModelState

**IsValid**

Specifies whether there is at least one validation error or not (true or false).

**Values**

Contains each model property value with corresponding "Errors" property that contains list of validation errors of that model property.

**ErrorCount**

Returns number of errors.

Model Validation

**[Required(ErrorMessage = "value")]**

Specifies that the property value is required (can't be blank or empty).

**[StringLength(int maximumLength, MinimumLength = value, ErrorMessage = "value")]**

Specifies minimum and maximum length (number of characters) allowed in the string.

**[Range(int minimum, int maximum, ErrorMessage = "value")]**

Specifies minimum and maximum numerical value allowed.

**[RegularExpression(string pattern, ErrorMessage = "value")]**

Specifies the valid pattern (regular expression).

**[EmailAddress(ErrorMessage = "value")]**

Specifies that the value should be a valid email address.

**[Phone(ErrorMessage = "value")]**

Specifies that the value should be a valid phone number).

Eg: (999)-999-9999 or 9876543210

**[Compare(string otherProperty, ErrorMessage = "value")]**

Specifies that the values of current property and other property should be same.

**[Url(ErrorMessage = "value")]**

Specifies that the value should be a valid url (website address).

Eg: http://www.example.com

**[ValidateNever]**

Specifies that the property should not be validated (excludes the property from model validation).

Custom Validations

1. class ClassName : ValidationAttribute
2. {
3. public override ValidationResult? IsValid(object? value, ValidationContext validationContext)
4. {
5. //return ValidationResult.Success;
6. //[or] return new ValidationResult("error message");
7. }
8. }

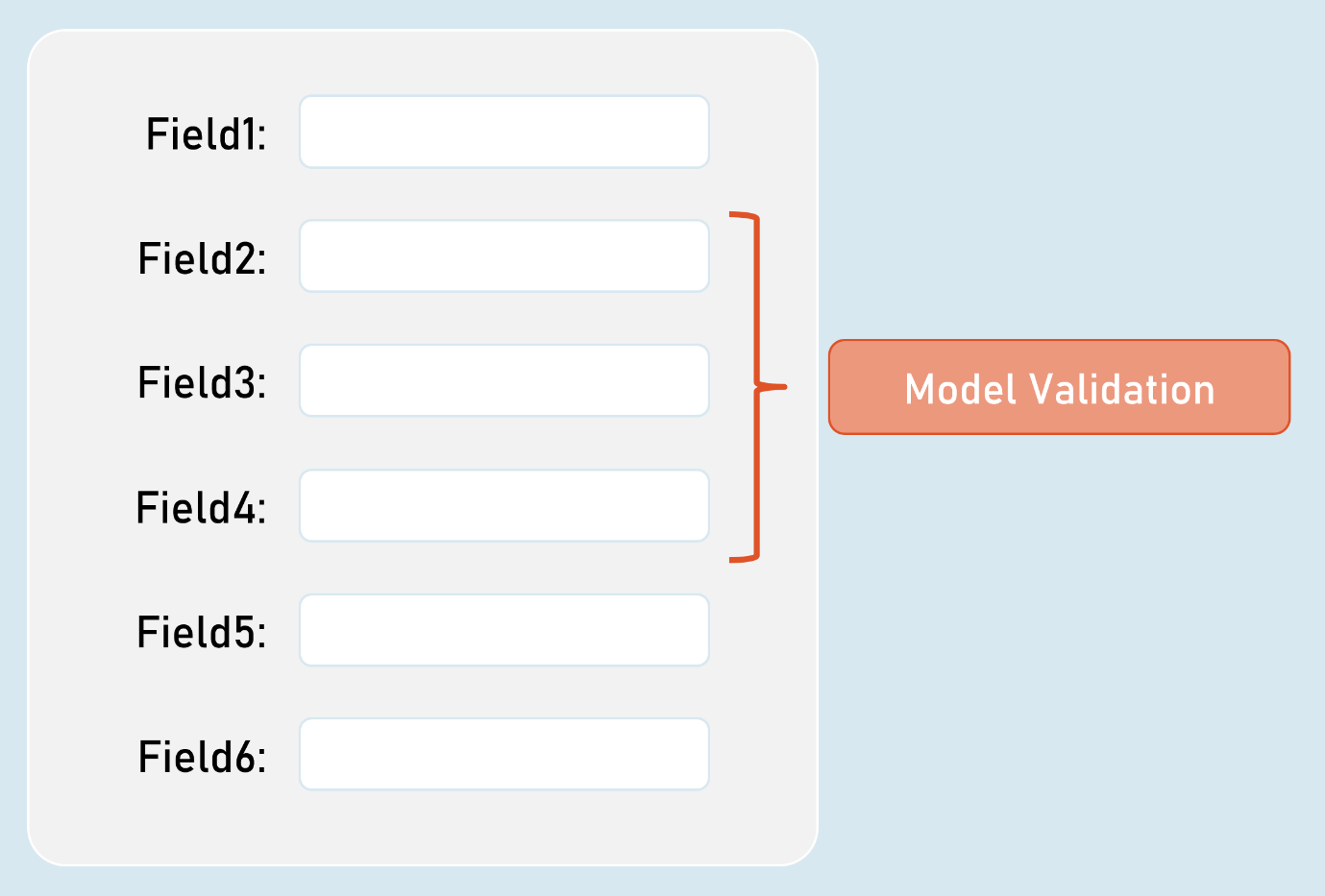
**ValidationAttribute**

* Base class for all validation attributes such as RequiredAttribute, RegularExpressionAttribute, RangeAttribute, StringLengthAttribute, CompareAttribute etc.
* Provides properties such as ErrorMessage & methods such as Validate(), IsValid() etc.

**ValidationContext**

* Acts as a parameter for "IsValid()" method of custom validation attribute classes.
* Provides properties such as ObjectType, ObjectInstance.

Custom Validations with Multiple Properties



IValidatableObject

1. class ClassName : IValidatableObject
2. {
3. //model properties here
5. public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
6. {
7. if (condition)
8. {
9. yield return new ValidationResult("error message");
10. }
11. }
12. }

* Base class for model classes with validation.
* Provides a method called Validate() to define class level validation logic.
* The Validate() method executes after validating all property-level validations are executed; but doesn't execute if at least one property-level validations result error.

**ValidationContext**

* Acts as a parameter for "Validate()" method of model classes with IValidatableObject.
* Provides properties such as ObjectType, ObjectInstance.

[Bind] and [BindNever]

**[Bind]**

1. class ClassNameController
2. {
3. public IActionResult ActionMethodName( [Bind(nameof(ClassName.PropertyName), nameof(ClassName.PropertyName) )] ClassName parameterName)
4. {
5. }
6. }

* [Bind] attribute specifies that only the specified properties should be included in model binding.
* Prevents over-posting (post values into unexpected properties) especially in 'Create' scenarios.

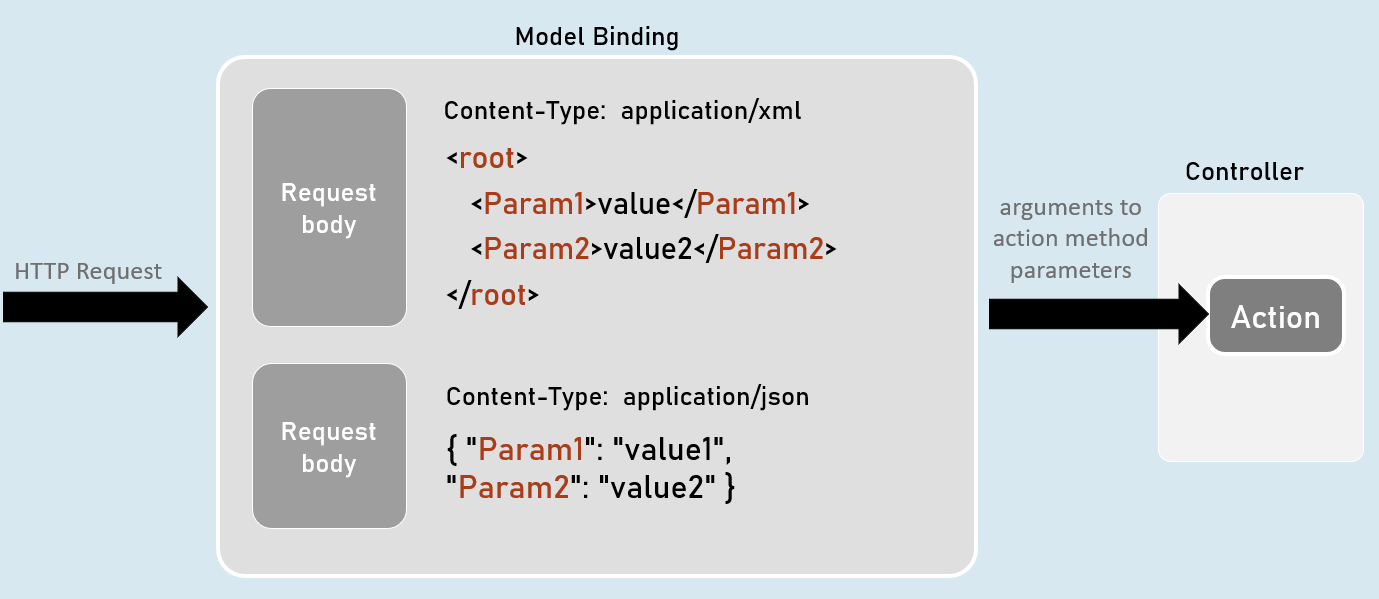
**[BindNever]**

1. class ModelClassName
2. {
3. [BindNever]
4. public type PropertyName { get; set; }
5. }

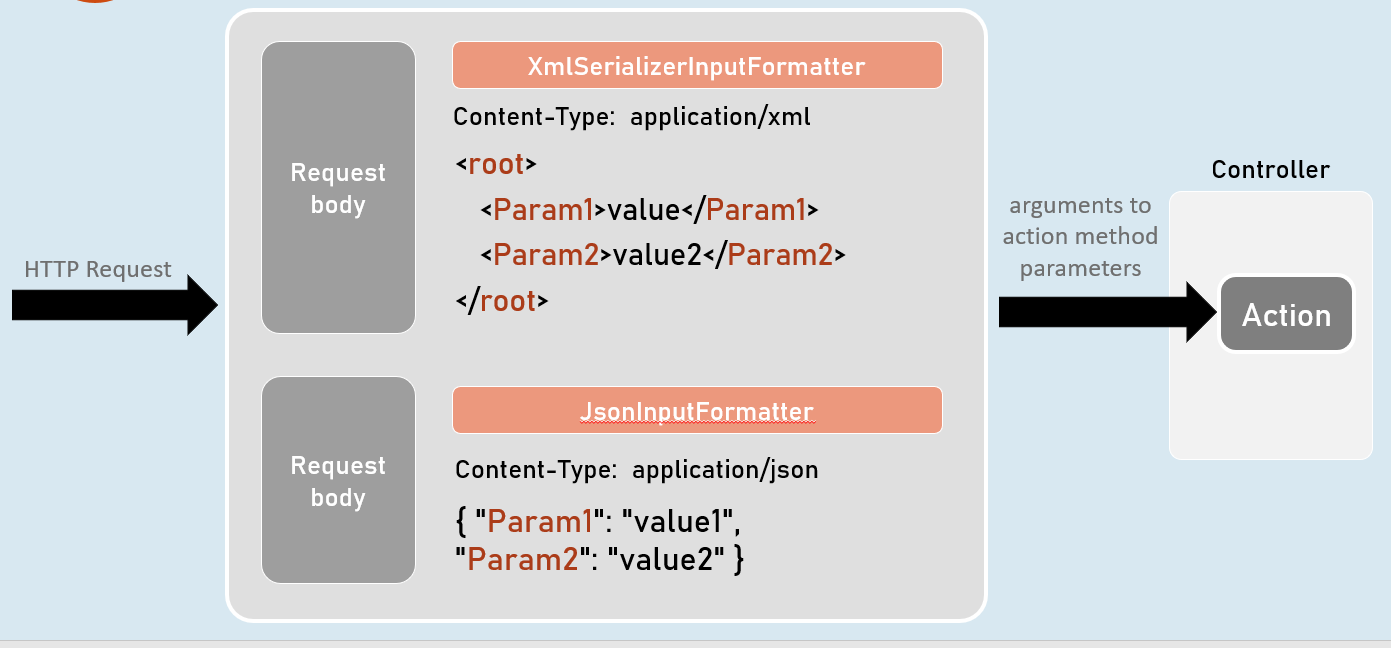
* [BindNever] attribute specifies that the specified property should NOT be included in model binding.
* Useful when you have fewer properties to eliminate from model binding.

[FromBody]

1. //enables the input formatters to read data from request body (as JSON or XML or custom) only
2. public IActionResult ActionMethodName( [FromBody] type parameter)
3. {
4. }



Input Formatters



Custom Model Binders

**Custom Model Binder**

1. class ClassName : IModelBinder
2. {
3. public Task BindModelAsync(ModelBindingContext bindingContext)
4. {
5. //gets value from request
6. bindingContext.ValueProvider.GetValue("FirstName");
8. //returns model object after reading data from the request
9. bindingContext.Result = ModelBindingResult.Success(your\_object);
10. }
11. }

**IModelBinder**

* Base interface for all custom model binders.
* Provides a method called BindModelAsync, to define logic for binding (reading) data from the request and creating a model object that has be received as parameter in the action method.

**ModelBindingContext**

* Acts as a parameter for "BindModelAsync()" method of custom model binder classes.
* Provides properties such as HttpContext, ModelState, ValueProvider, Result etc..

Custom Model Binder Providers

1. class ClassName : IModelBinderProvider
2. {
3. public IModelBinder GetBinder(ModelBinderProviderContext providerContext)
4. {
5. //returns type of custom model binder class to be invoked
6. return new BinderTypeModelBinder(typeof(YourModelBinderClassName));
7. }
8. }

**IModelBinderProvider**

* Base interface for all custom model binder providers.
* Provides a method called GetBinder, to return the type of custom model binder class.

**ModelBinderProviderContext**

* Acts as a parameter for "GetBinder()" method of custom model binder provider classes.
* Provides properties such as BindingInfo, Services etc.

# **6. Razor Views [MVC]**

MVC Architecture

Model-View-Controller (MVC) is a design pattern that organizes your application into three distinct components, each with a specific responsibility:

1. **Model:**
   * Represents the data and business logic of your application.
   * Encapsulates data access, validation rules, and any core business operations.
   * **Types of Models:**
     + **Business Model (Domain Model):** Represents the real-world entities and relationships of your application domain (e.g., Product, Customer, Order).
     + **View Model:** A tailored model specifically designed to provide data to a view. It might aggregate data from multiple business models or contain properties for UI-specific concerns.
2. **View:**
   * Renders the user interface (UI) based on the data provided by the controller.
   * Typically, views are HTML templates with embedded server-side code (Razor syntax in ASP.NET Core) that dynamically generate the content.
3. **Controller:**
   * Acts as the intermediary between the model and the view.
   * Receives user input (from requests), interacts with the model to perform actions or retrieve data, and then selects the appropriate view to present the results.

Execution Flow of a Request in MVC

1. **Routing:** The incoming request is processed by the routing middleware, which determines the appropriate controller and action method to handle it based on the URL pattern and HTTP method.
2. **Model Binding:** If the action method has parameters, the model binder extracts data from the request (query string, route values, form data, body) and attempts to convert it into the types expected by the parameters.
3. **Model Validation:** The model binder validates the bound data using data annotations and any custom validation logic you've implemented. If validation fails, errors are added to the ModelState object.
4. **Controller Action Execution:** If the model is valid, the controller action method executes its logic, which might involve:
   * Interacting with business models to retrieve or update data.
   * Performing calculations or other business operations.
   * Preparing a view model to pass data to the view.
5. **View Selection:** The controller selects the appropriate view to render and passes the view model to it.
6. **View Rendering:** The view engine (Razor) generates the HTML output based on the view template and the data in the view model.
7. **Response:** The rendered HTML is sent back to the client as the HTTP response.

Responsibilities of MVC Components

* **Controller:**
  + Handles incoming requests and routes them to the correct action methods.
  + Interacts with the model to fetch or modify data.
  + Selects the appropriate view and passes necessary data to it.
  + Handles errors and redirects.
* **Model (Business Model):**
  + Encapsulates the business logic and data access of the application.
  + Represents the core entities and relationships in your domain.
  + Defines validation rules for ensuring data integrity.
* **View Model:**
  + Tailored for a specific view, containing only the data required by that view.
  + Simplifies data presentation in the view and avoids exposing unnecessary details.
* **View:**
  + Renders the user interface based on the data provided by the controller.
  + Uses Razor syntax to combine HTML markup with C# code for dynamic content generation.

Benefits and Goals of MVC

* **Separation of Concerns (SoC):** The clear division of responsibilities makes code more organized, maintainable, and testable.
* **Testability:** Individual components (models, views, controllers) can be tested in isolation.
* **Reusability:** Views and models can be reused in different contexts.
* **Parallel Development:** Different team members can work on different parts of the application simultaneously.
* **Maintainability:** Changes to one component are less likely to affect others.
* **Extensibility:** New features can be added more easily due to the modular structure.
* **Scalability:** MVC architecture lends itself well to scaling as the application grows.

The MVC pattern provides a structured and organized approach to building complex web applications, facilitating collaboration, code reusability, and long-term maintainability. Understanding these core concepts will make you a more effective ASP.NET Core developer.

Views

In ASP.NET Core MVC, views are responsible for rendering the user interface (UI) that your application presents to the user. They are typically HTML templates with embedded Razor syntax, which is a powerful templating engine that allows you to seamlessly blend HTML markup with C# code.

**Notes**

* **Dynamic Content:** Views can generate dynamic content based on data passed to them from the controller (the model). This allows you to display information fetched from databases, user inputs, and other sources.
* **Razor Syntax:** Views use Razor syntax, which starts with @ symbols, to embed C# code within the HTML. This code can perform tasks like looping through data collections, conditional rendering, and accessing model properties.

AddControllersWithViews() Method

This extension method is used to register the necessary services for MVC (models, views, controllers) with the dependency injection container. It's a shortcut for adding multiple services at once, including:

* **Controller Discovery:** Automatically discovers controller classes in your project.
* **View Engine:** Configures Razor as the view engine.
* **Model Binding:** Sets up model binding for handling form submissions.
* **Validation:** Enables model validation.

ViewResult

The ViewResult class is an action result type in ASP.NET Core MVC that represents a view to be rendered. When a controller action returns a ViewResult, the MVC framework locates the corresponding view file, passes the model data (if any) to it, and renders the view's content as HTML.

**Default View Locations**

ASP.NET Core MVC follows a convention for determining where to find view files:

* **By Convention:** By default, the view engine looks for views in the Views/[ControllerName]/[ActionName].cshtml path. For example, the Index action method in the HomeController would look for a view file at Views/Home/Index.cshtml.
* **Overriding with ViewName:** You can explicitly specify the view name using the ViewName property of the ViewResult or the View() helper method.
* **Shared Views:** Shared views are stored in the Views/Shared folder and can be used by multiple controllers.

Code

1. // Program.cs
2. builder.Services.AddControllersWithViews(); // Enables MVC features
4. // ...
5. // HomeController.cs
6. [Route("home")]
7. public IActionResult Index()
8. {
9. return View(); // Renders Views/Home/Index.cshtml
10. }
12. // Views/Home/Index.cshtml
13. <!DOCTYPE html>
14. <html>
15. <head>
16. <title>Asp.Net Core App</title>
17. </head>
18. <body>
19. Welcome
20. </body>
21. </html>
22. **Enabling MVC:** The AddControllersWithViews() method configures the application for MVC, including view support.
23. **Controller Action:** The Index action method in HomeController returns a ViewResult without specifying a view name.
24. **View Location:** Since no view name is explicitly provided, the MVC framework follows the convention and looks for the view at Views/Home/Index.cshtml.
25. **View Content:** The Index.cshtml view contains simple HTML that will be rendered as the response.

Razor View Engine

Razor is the default view engine in ASP.NET Core MVC. It provides a concise and elegant way to create dynamic web pages by combining C# code with HTML markup.

**Key Razor Syntax Elements**

1. **Code Blocks (@{...}):**
   * Purpose: Enclose multi-line C# code statements.
   * Use Cases:
     + Declaring variables
     + Defining complex logic
     + Executing database queries or other operations
2. **Expressions (@variable or @method()):**
   * Purpose: Embed C# expressions directly into the HTML output.
   * Use Cases:
     + Displaying values from variables or model properties
     + Calling helper methods or functions
3. **Literals (@: or <text>):**
   * Purpose: Output raw text without HTML encoding.
   * Use Cases:
     + Displaying plain text, HTML snippets, or values that might contain HTML characters

Control Flow in Razor Views

1. **@if, @else, @elseif:**
   * Purpose: Conditional rendering of HTML blocks based on C# conditions.
   * Example:
2. @if (person.DateOfBirth.HasValue)
3. {
4. <p>Age: @(Math.Round((DateTime.Now - person.DateOfBirth).Value.TotalDays / 365.25)) years old</p>
5. }
6. else
7. {
8. <p>Date of birth is unknown</p>
9. }
10. **@switch:**
    * Purpose: Choose one of several blocks of code to execute based on the value of an expression.
    * Example:
11. @switch (person.PersonGender)
12. {
13. case Gender.Male:
14. <p>November 19 is International Men's Day</p>
15. break;
16. case Gender.Female:
17. <p>March 8 is International Women's Day</p>
18. break;
19. // ... other cases ...
20. }
21. **@foreach:**
    * Purpose: Iterate over a collection (e.g., a list or array) and render HTML for each item.
    * Example:
22. @foreach (var person in people)
23. {
24. <div>@person.Name, @person.PersonGender</div>
25. }
26. **@for:**
    * Purpose: Execute a code block a specific number of times.
    * Example:
27. @for (int i = 0; i < 5; i++)
28. {
29. <p>Iteration: @i</p>
30. }

**Notes**

* **Razor Syntax:** Familiarize yourself with Razor syntax (code blocks, expressions, literals) for embedding C# code in your views.
* **Control Flow:** Understand how to use if, else, switch, foreach, and for to control the flow of logic and create dynamic views.
* **Model Binding:** Views often work with models passed from the controller. Use Razor expressions to access and display model properties.

Local Functions in Razor Views

Local functions are C# functions defined within the scope of a Razor code block (a code block enclosed in @{ ... }). They allow you to encapsulate reusable logic directly in your views, making your view code more modular, organized, and easier to read.

**Syntax and Features**

* **Declaration:** Local functions are declared using the standard C# method syntax, typically within an @functions { ... } block.
* **Scope:** They can only be called within the code block or section where they are defined.
* **Parameters and Return Types:** Local functions can take parameters and return values, just like regular C# methods.
* **Accessibility:** They are implicitly private to the view.
* **Access to View Data:** Local functions can access variables and model properties declared within the same code block.

**Why Use Local Functions?**

* **Encapsulation:** Group related logic into self-contained functions, improving code readability.
* **Readability:** Break down complex code into smaller, more manageable chunks.
* **Reusability:** Call local functions multiple times within the same view, avoiding code duplication.
* **Cleaner Views:** Reduce the amount of inline C# code scattered throughout your HTML markup.

Code

1. @using ViewsExample.Models
3. @{
4. string appTitle = "Asp.Net Core Demo App";
5. List<Person> people = new List<Person>()
6. {
7. // ... (person data)
8. };
9. }
11. @functions {
12. double? GetAge(DateTime? dateOfBirth)
13. {
14. if (dateOfBirth is not null)
15. {
16. return Math.Round((DateTime.Now - dateOfBirth.Value).TotalDays / 365.25);
17. }
18. else
19. {
20. return null;
21. }
22. }
24. int x = 10; // Example of a local variable
26. string Name { get; set; } = "Hello name"; // Example of a local property
27. }
29. <!DOCTYPE html>
30. <html>
31. <head>
32. <title>@appTitle</title>
33. <meta charset="utf-8" />
34. </head>
35. <body>
36. <h1>Welcome to @Name</h1>
37. @for (int i = 0; i < 2; i++)
38. {
39. Person person = people[i];
40. <div>
41. @person.Name
42. <span>, </span>
43. <span>@person.PersonGender</span>
44. @if (person.DateOfBirth != null)
45. {
46. <span>@person.DateOfBirth.Value.ToString("MM/dd/yyyy")</span>
47. <span>@GetAge(person.DateOfBirth) years old</span>
48. }
49. </div>
50. }
51. </body>
52. </html>

In this code:

1. **Local Function:** GetAge(DateTime? dateOfBirth) calculates the age of a person based on their date of birth.
2. **Local Variable:** int x = 10; This is not used in the view but demonstrates how to declare local variables within a Razor code block.
3. **Local Property:** string Name { get; set; } = "Hello name"; This is used to set the title of the page.

The GetAge function is called within the @foreach loop to display the age of each person if their date of birth is available.

**Important Considerations**

* **Overuse:** Avoid overusing local functions. They're great for encapsulation, but too many can make your views harder to follow.
* **Alternatives:** Consider helper methods or view components for more complex or reusable logic.

Html.Raw()

In Razor views, the default behavior is to automatically encode HTML content to prevent potential security vulnerabilities like cross-site scripting (XSS) attacks. However, sometimes you have legitimate HTML content (e.g., from a rich text editor or a trusted source) that you want to render as-is, without encoding. This is where Html.Raw() comes in.

**Purpose**

* **Bypass HTML Encoding:** Html.Raw() prevents Razor from encoding the HTML content you provide, allowing it to be rendered directly in the browser.
* **Displaying Trusted HTML:** Use it when you trust the source of the HTML content and want to preserve its formatting and structure.
* **Dynamic Content Generation:** Html.Raw() can be used to dynamically insert HTML fragments into your views based on data or logic.

**Syntax**

1. @Html.Raw(htmlContent)

where htmlContent is a string variable or expression containing the HTML you want to render without encoding.

**Important Considerations**

* **Security Risk:** **Use Html.Raw() with extreme caution.** If the htmlContent originates from user input or an untrusted source, it could lead to XSS vulnerabilities. Always sanitize and validate user-generated content before rendering it with Html.Raw().
* **Content Security Policy (CSP):** Consider implementing a CSP to further mitigate XSS risks. A CSP defines a set of rules that govern how the browser handles external scripts, styles, and other resources.

Code

1. @{
2. // ... (other variables and logic) ...
3. string alertMessage = $"<script>alert('{people.Count} people found')</script>";
4. }
6. <!DOCTYPE html>
7. <html>
8. <head>
9. </head>
10. <body>
11. @Html.Raw(alertMessage)
12. <h1>Welcome</h1>
13. @for (int i = 0; i < 2; i++)
14. {
15. // ... (rest of the view code) ...
16. }
17. </body>
18. </html>

In this code:

1. **Raw HTML String:** The variable alertMessage contains a string of HTML that includes a <script> tag intended to display an alert message with the number of people found.
2. **Html.Raw() Usage:** The @Html.Raw(alertMessage) line tells Razor to render the contents of alertMessage directly into the HTML output without encoding. This will result in the following HTML in the output:
3. <script>alert('3 people found')</script>
4. **Client-Side Execution:** When the browser parses this HTML, it will execute the JavaScript code inside the <script> tag, triggering an alert box.
5. **No sanitization:** In this example, since the number of people is not user input, there is no risk of XSS attack. However, if the content of the alert message is coming from user input, it is essential to santize it before displaying it to prevent XSS attacks.

**Notes**

* **Trust but Verify:** Only use Html.Raw() when you're absolutely sure the content is safe and from a trusted source.
* **Security Risks:** Be aware of the potential for XSS vulnerabilities when rendering raw HTML.
* **Alternative Approaches:** If you need to inject dynamic HTML content, consider safer alternatives like partial views or view components.
* **Content Security Policy (CSP):** Implement a CSP to add an extra layer of protection against XSS attacks.

ViewData and ViewBag

Both ViewData and ViewBag are mechanisms in ASP.NET Core MVC for passing data from your controller actions to your views. While they serve the same purpose, they differ in their implementation and syntax.

* **ViewData (Dictionary-Based):**
  + It's a dictionary-like object (ViewDataDictionary) that stores key-value pairs.
  + Keys are strings, and values can be of any type.
  + Access data using string keys: ViewData["keyName"].
  + Requires casting when retrieving non-string values.
* **ViewBag (Dynamic):**
  + It's a dynamic wrapper around the ViewData dictionary.
  + Allows you to access data using dot notation: ViewBag.keyName.
  + No explicit casting is needed for non-string values.

**Availability in Controllers and Views**

Both ViewData and ViewBag are accessible in:

* **Controllers:** You set values in the controller action and they are passed to the view.
* **Views:** You retrieve values from the ViewData dictionary or the ViewBag dynamic object in the view to render content.

**Code of ViewData:**

1. // HomeController.cs (Controller)
2. ViewData["appTitle"] = "Asp.Net Core Demo App";
3. ViewData["people"] = people;
5. // Index.cshtml (View)
6. <title>@ViewData["appTitle"]</title>
7. List<Person>? people = (List<Person>?)ViewData["people"];

**Code of ViewBag:**

1. // HomeController.cs (Controller)
2. ViewBag.appTitle = "Asp.Net Core Demo App";
3. ViewBag.people = people;
4. // Index.cshtml (View)
5. <title>@ViewBag.appTitle</title>
6. @foreach (Person person in ViewBag.people)

**Best Practices**

* **Choose One:** It's generally recommended to pick either ViewData or ViewBag and use it consistently throughout your project to maintain a clear and unified approach.
* **Strong Typing with View Models:** While ViewData and ViewBag offer flexibility, they lack compile-time type safety. For larger applications, prefer strongly typed view models to pass data to views.
* **Limited Scope:** Understand that ViewData and ViewBag data exists only for the duration of the current request and response cycle. It's not meant for persisting data between requests.
* **Meaningful Keys:** Use descriptive and meaningful keys for your data to improve code readability.

**Things to Avoid**

* **Overuse:** Don't overstuff ViewData or ViewBag with excessive data. Keep it concise and focused on the specific information needed by the view.
* **Magic Strings:** Avoid hardcoding strings for keys. Use constants or enums for better maintainability.
* **Sensitive Data:** Never pass sensitive data like passwords or authentication tokens through ViewData or ViewBag.

Strongly Typed Views

In ASP.NET Core MVC, a strongly typed view is a view that is associated with a specific model class. This means that the view has direct access to the properties and methods of that model, providing compile-time type checking and IntelliSense support within the view.

**The @model Directive**

The @model directive is used at the top of a view to specify the model type for that view. For example, @model Person indicates that the view expects to work with data of the Person class.

**When to Use Strongly Typed Views**

* **Complex Data:** When your view needs to display or interact with complex data structures that have multiple properties or relationships.
* **Type Safety:** When you want to catch type-related errors during development, rather than at runtime.
* **IntelliSense:** When you want the full power of Visual Studio's IntelliSense to help you code faster and with fewer errors.
* **Refactoring:** When you need to change the model, strongly typed views make it easier to update the corresponding views automatically.

**Best Practices**

* **Use View Models:** Instead of directly passing your business models to views, create specialized view models tailored to the data needed by each view. This helps keep your views clean and focused.
* **Naming Conventions:** Follow standard naming conventions for your view models (e.g., ProductViewModel, OrderDetailsViewModel).
* **Keep Views Simple:** Views should primarily focus on presentation logic. Avoid complex business logic within views.
* **Partial Views:** Leverage partial views for reusable components, passing view models to them as needed.
* **Leverage Tag Helpers:** Explore the use of Tag Helpers, which provide a more HTML-friendly way to interact with your models in views.

Code:

**Controller (HomeController.cs)**

1. // ...
2. [Route("home")]
3. [Route("/")]
4. public IActionResult Index()
5. {
6. List<Person> people = new List<Person>()
7. {
8. new Person() { Name = "John", DateOfBirth = DateTime.Parse("2000-05-06"), PersonGender = Gender.Male},
9. new Person() { Name = "Linda", DateOfBirth = DateTime.Parse("2005-01-09"), PersonGender = Gender.Female},
10. new Person() { Name = "Susan", DateOfBirth = null, PersonGender = Gender.Other}
11. };
13. return View("Index", people); // Pass the 'people' list as the model
14. }
16. [Route("person-details/{name}")]
17. public IActionResult Details(string? name)
18. {
19. // ... (Retrieve the person based on the name) ...
20. Person? matchingPerson = people.Where(temp => temp.Name == name).FirstOrDefault();
21. return View(matchingPerson); // Pass a single 'Person' object as the model
22. }

Both action methods pass the model(s) to the view using return View(model).

**View (Index.cshtml)**

1. @using ViewsExample.Models
2. @model IEnumerable<Person> // Indicates that the model is a collection of Person objects
4. <!DOCTYPE html>
5. <html>
6. <head>
7. <title>@ViewBag.appTitle</title>
8. </head>
9. <body>
10. <div class="page-content">
11. <h1>Persons</h1>
13. @foreach (Person person in Model)
14. {
15. <div class="box float-left w-50">
16. <h3>@person.Name</h3>
17. <table class="table w-100">
18. <tbody>
19. <tr>
20. <td>Gender</td>
21. <td>@person.PersonGender</td>
22. </tr>
23. <tr>
24. <td>Date of Birth</td>
25. <td>@person.DateOfBirth?.ToString("MM/dd/yyyy")</td>
26. </tr>
27. </tbody>
28. </table>
29. <a href="/person-details/@person.Name">Details</a>
30. </div>
31. }
32. </div>
34. </body>
35. </html>

This view iterates over the Model (which is a list of Person objects) using a foreach loop. Inside the loop, it accesses the properties of each Person object directly (e.g., @person.Name, @person.PersonGender) thanks to strong typing.

**View (Details.cshtml)**

1. @using ViewsExample.Models
2. @model Person // Indicates that the model is a single Person object
4. <!DOCTYPE html>
5. <html>
6. <head>
7. <title>Person Details</title>
8. </head>
9. <body>
10. <div class="page-content">
11. <h1>Person Details</h1>
12. <div class="box">
13. <h3>@Model.Name</h3>
14. <table class="table w-100">
15. <tbody>
16. <tr>
17. <td>Gender</td>
18. <td>@Model.PersonGender</td>
19. </tr>
20. <tr>
21. <td>Date of Birth</td>
22. <td>@Model.DateOfBirth?.ToString("MM/dd/yyyy")</td>
23. </tr>
24. </tbody>
25. </table>
26. </div>
27. <a href="/home">Back to persons</a>
28. </div>
29. </body>
30. </html>

This view displays the details of a single Person object. Notice that you can directly access properties like @Model.Name without any casting or dictionary lookups.

Strongly Typed Views with Multiple Models

When your view requires data from more than one model class, the most common and recommended approach is to create a *view model*. A view model is a custom class that encapsulates all the data necessary for a specific view, aggregating properties from multiple models or providing additional properties tailored for the view's presentation needs.

**Why Use View Models?**

* **Encapsulation:** Keeps your view logic organized and prevents your views from becoming tightly coupled to your underlying business models.
* **Flexibility:** Allows you to combine data from different sources (multiple models, configuration settings, etc.) into a single object for the view.
* **Type Safety:** Strongly typed views with view models offer compile-time type checking and IntelliSense, improving development efficiency.
* **Data Shaping:** You can create properties in the view model specifically designed for how you want to display data in the view, such as formatted strings or calculated values.

**Creating and Using View Models**

1. **Define the View Model:** Create a new class in your Models folder to represent the view model. This class will have properties to hold the data from the various models your view needs.
2. **Populate in the Controller:** In your controller action, retrieve the data from your different models and use them to create an instance of your view model.
3. **Pass to the View:** Pass the populated view model object to the view using the View method.
4. **Access in the View:** In your view, use the @model directive at the top to specify the type of your view model. You can then access the view model's properties using Model.<PropertyName>.

**Code Example: PersonAndProductWrapperModel**

1. // PersonAndProductWrapperModel.cs (View Model)
2. public class PersonAndProductWrapperModel
3. {
4. public Person PersonData { get; set; }
5. public Product ProductData { get; set; }
6. }
8. // HomeController.cs (Controller)
9. public IActionResult SomeAction()
10. {
11. Person person = GetPersonData(); // Fetch person data
12. Product product = GetProductData(); // Fetch product data
14. var viewModel = new PersonAndProductWrapperModel
15. {
16. PersonData = person,
17. ProductData = product
18. };
20. return View(viewModel);
21. }

**View (PersonAndProduct.cshtml)**

1. @using ViewsExample.Models
2. @model PersonAndProductWrapperModel
4. <!DOCTYPE html>
5. <html>
6. <head>
7. </head>
8. <body>
9. <div class="page-content">
10. <h1>Person and Product</h1>
12. <div class="box w-100">
13. <h3>Person</h3>
14. <table class="table w-50">
15. <tbody>
16. <tr>
17. <td>Person Name</td>
18. <td>@Model.PersonData.Name</td>
19. </tr>
20. <tr>
21. <td>Gender</td>
22. <td>@Model.PersonData.PersonGender</td>
23. </tr>
24. </tbody>
25. </table>
26. </div>
28. <div class="box w-100">
29. <h3>Product</h3>
30. <table class="table w-50">
31. <tbody>
32. <tr>
33. <td>Product Id</td>
34. <td>@Model.ProductData.ProductId</td>
35. </tr>
36. <tr>
37. <td>Product Name</td>
38. <td>@Model.ProductData.ProductName</td>
39. </tr>
40. </tbody>
41. </table>
42. </div>
43. </div>
44. </body>
45. </html>

**Key Points:**

* **Organization:** View models help keep your views focused on presentation and your controllers focused on data retrieval and processing.
* **Maintainability:** When your underlying models change, you only need to update your view model, not every view that uses that data.
* **Flexibility:** You can include additional properties in your view model that aren't part of your original models. This could be formatting options, computed values, or flags for controlling the view's behavior.

\_ViewImports.cshtml

The \_ViewImports.cshtml file is a special file in ASP.NET Core MVC that allows you to centralize common directives and settings that apply to multiple views within your application. This file typically resides in the Views folder at the root of your project, but you can also place it within subfolders to apply the settings to views within those specific folders.

**What Can You Put in \_ViewImports.cshtml?**

* **@using Directives:** Import namespaces that you frequently use in your views. This eliminates the need to include these directives in every individual view.
* **@addTagHelper Directives:** Make Tag Helpers available to your views. Tag Helpers are server-side code snippets that generate or modify HTML elements in a more intuitive way than traditional Razor syntax.
* **@inject Directives:** Inject services into your views, making them accessible for use in your Razor code.
* **@model Directive (Optional):** Specify a default model type for all views in the directory or subdirectories (not recommended for complex projects).
* **@layout Directive (Optional):** Set a default layout for all views in the directory or subdirectories.

**How It Works**

When your application processes a request for a view, it looks for a \_ViewImports.cshtml file in the following order:

1. **The same directory as the view:** If found, the directives and settings in this file are applied.
2. **Parent directories:** It then checks the parent directory of the view, and so on, up to the root Views folder.
3. **Root Views folder:** Finally, it checks the \_ViewImports.cshtml file in the root Views folder.

Directives in a child folder's \_ViewImports.cshtml file can override settings inherited from parent directories.

**Code Example: \_ViewImports.cshtml**

1. @using System.Collections.Generic // Import the generic collections namespace
2. @using YourProject.Models // Import your project's models namespace
3. @addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers // Add built-in ASP.NET Core Tag Helpers

**Benefits of Using \_ViewImports.cshtml**

* **Reduced Redundancy:** Avoid repeating the same directives in multiple views.
* **Consistent Configuration:** Easily apply common settings across your views.
* **Improved Readability:** Keep your individual view files cleaner and more focused on their specific content.

**Code of View:**

1. @model YourProject.Models.Product
2. <!DOCTYPE html>
3. <html>
4. <head>
5. <title>Product Details</title>
6. </head>
7. <body>
8. <h1>@Model.Name</h1>
9. </body>
10. </html>

In this example, the @model directive is not required in the view because it's already specified in the \_ViewImports.cshtml file. Similarly, you don't need to include the @using directive for your model's namespace.

**Important Considerations**

* **Scoping:** Directives in \_ViewImports.cshtml are scoped to the directory and its subdirectories.
* **Overriding:** Settings in a child folder's \_ViewImports.cshtml file override those in parent folders.
* **Flexibility:** Use multiple \_ViewImports.cshtml files in different folders to manage settings at a more granular level.

**Key Points to Remember**

* **Centralization:** Use \_ViewImports.cshtml to centralize common view directives and settings.
* **DRY Principle:** Adhere to the Don't Repeat Yourself (DRY) principle by avoiding redundant code.
* **Scoping and Overriding:** Understand how the cascading nature of \_ViewImports.cshtml files works.

Shared Views

In ASP.NET Core MVC, a shared view is a view file (.cshtml) that is not tied to a specific controller or action. These views typically reside in the Views/Shared folder and are designed to be reused across different controllers or even multiple projects.

**Why Use Shared Views?**

* **Reduce Duplication:** Avoid writing the same code repeatedly for common UI elements across your application.
* **Consistent UI:** Maintain a unified look and feel across different pages and sections of your site.
* **Simplified Maintenance:** Update the shared view once, and the changes automatically apply to all views that use it.

**How Shared Views Work**

1. **Location:** By default, ASP.NET Core MVC looks for shared views in the Views/Shared folder. You can also create subfolders within Views/Shared to further organize your shared views.
2. **Naming:** Name your shared views in a way that reflects their purpose or content. Common examples include:
   * \_Layout.cshtml: The main layout for your application's pages.
   * \_PartialName.cshtml: Smaller, reusable components (partial views) that can be embedded in other views.
3. **Rendering:** You can render a shared view using the following approaches:
   * **PartialView():** Renders a partial view.
   * **View():** Can be used to render a shared view directly, but it's more common to use PartialView() for partial views and reserve View() for full pages.

**Code Example: Shared View All.cshtml**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>All Products</title>
5. <meta charset="UTF-8" />
6. <link href="~/StyleSheet.css" rel="stylesheet" />
7. </head>
8. <body>
9. <div class="page-content">
10. <h1>All Products</h1>
11. </div>
12. </body>
13. </html>

This shared view (All.cshtml) can be reused by multiple controllers to display the "All Products" page with the same structure and styling.

**Code Example: Controllers Using the Shared View**

1. // HomeController.cs
2. [Route("home/all-products")]
3. public IActionResult All()
4. {
5. return View(); // Will look for Views/Home/All.cshtml first, then Views/Shared/All.cshtml
6. }
8. // ProductsController.cs
9. [Route("products/all")]
10. public IActionResult All()
11. {
12. return View(); // Will look for Views/Products/All.cshtml first, then Views/Shared/All.cshtml
13. }

In both HomeController and ProductsController, the All action method returns a ViewResult. ASP.NET Core will first look for a view named All.cshtml within the specific controller's view folder (e.g., Views/Home or Views/Products). If it doesn't find it there, it will look for it in the Views/Shared folder.

**Important Considerations**

* **Naming Convention:** Prefix the names of partial views with an underscore (e.g., \_ProductCard.cshtml) to distinguish them from full views.
* **Data Passing:** Use view models to pass data to shared views, ensuring that the view has access to all the information it needs to render correctly.
* **Flexibility:** You can override shared views within specific controllers if you need to customize them for a particular use case.

Shared views are a cornerstone of maintainable and scalable ASP.NET Core MVC applications. By leveraging their reusability, you can significantly streamline your development process and ensure a consistent user experience across your website or application.

Key Points to Remember

**1. Views**

* **Purpose:** Render the user interface (UI) of your MVC application.
* **Razor View Engine:** Default engine for combining HTML markup with C# code.
* **View File Location:** Typically found in the Views folder, organized by controller.
* **View Selection:** Controllers return ViewResult or other action results to render views.
* **Model Binding:** Views often receive data from controllers (the model) for dynamic rendering.

**2. Razor Syntax Essentials**

* **Code Blocks (@{...}):** For multi-line C# code.
* **Expressions (@variable or @method()):** Embed C# expressions.
* **Literals (@: or <text>):** Output raw text (no HTML encoding).
* **Control Flow:** Use @if, @else, @switch, @foreach, @for for conditional and iterative logic.
* **Comments:** @\*...\*@ for server-side comments.

**3. Local Functions**

* **Purpose:** Encapsulate reusable C# functions within Razor views.
* **Syntax:** Define within @functions { ... }.
* **Benefits:** Improved organization, readability, and reusability within a view.

**4. Html.Raw()**

* **Purpose:** Renders raw HTML content without encoding.
* **Warning:** Use with caution due to potential XSS vulnerabilities.
* **Alternatives:** Consider partial views or view components for dynamic HTML generation.

**5. ViewData and ViewBag**

* **Purpose:** Pass data from controllers to views.
* **ViewData:** Dictionary-based (ViewData["key"]).
* **ViewBag:** Dynamic wrapper around ViewData (ViewBag.key).
* **Recommendation:** Prefer strongly typed views or view models for better type safety and maintainability.

**6. Strongly Typed Views**

* **Purpose:** Associate a view with a specific model class using the @model directive.
* **Benefits:** Type safety, IntelliSense, and improved maintainability.
* **View Models:** Combine data from multiple models into a single view model class.

**7. \_ViewImports.cshtml**

* **Purpose:** Centralize common @using, @addTagHelper, @inject, and other directives.
* **Location:** In the root Views folder or subfolders.
* **Benefits:** Reduce redundancy, ensure consistency, and improve readability.

**8. Shared Views**

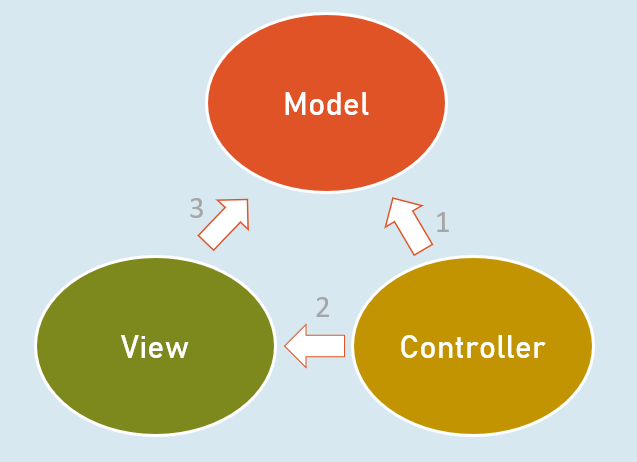
* **Purpose:** Reusable view components stored in the Views/Shared folder.
* **Types:**
  + **Layout Pages:** Define the overall page structure.
  + **Partial Views:** Reusable chunks of UI elements.
* **Rendering:** Use PartialView() to render partial views, View() for full views.

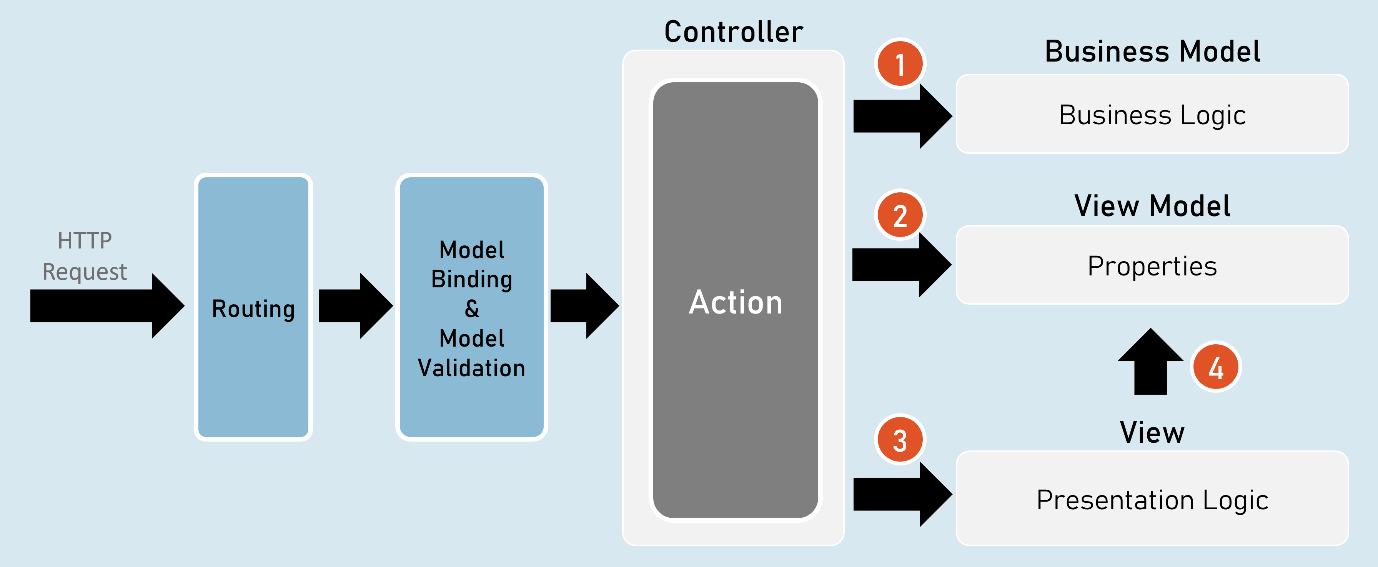
**Interview Tips:**

* **Razor Syntax:** Be ready to write examples of different Razor syntax elements and explain their use cases.
* **View Logic:** Discuss when it's appropriate to use local functions or view models in your views.
* **Security:** Emphasize the importance of security (input validation, avoiding Html.Raw() for untrusted content) when working with views.
* **Best Practices:** Demonstrate your knowledge of using view models, shared views, and \_ViewImports.cshtml for cleaner and more maintainable code.

Model-View-Controller (MVC) Pattern

"Model-View-Controller" (MVC) is an architectural pattern that separates application code into three main components: Models, Views and Controllers.





1. Controller invokes Business Model.
2. Controller creates object of View Model.
3. Controller invokes View.
4. View accesses View Model.

Responsibilities of Model-View-Controller

**Controller**

* Receives HTTP request data.
* Invoke business model to execute business logic.

**Business Model**

* Receives input data from the controller.
* Performs business operations such as retrieving / inserting data from database.
* Sends data of the database back to the controller.

**Controller**

* Creates object of ViewModel and files data into its properties.
* Selects a view & invokes it & also passes the object of ViewModel to the view.

**View**

* Receives the object of ViewModel from the controller.
* Accesses properties of ViewModel to render data in html code.
* After the view renders, the rendered view result will be sent as response.

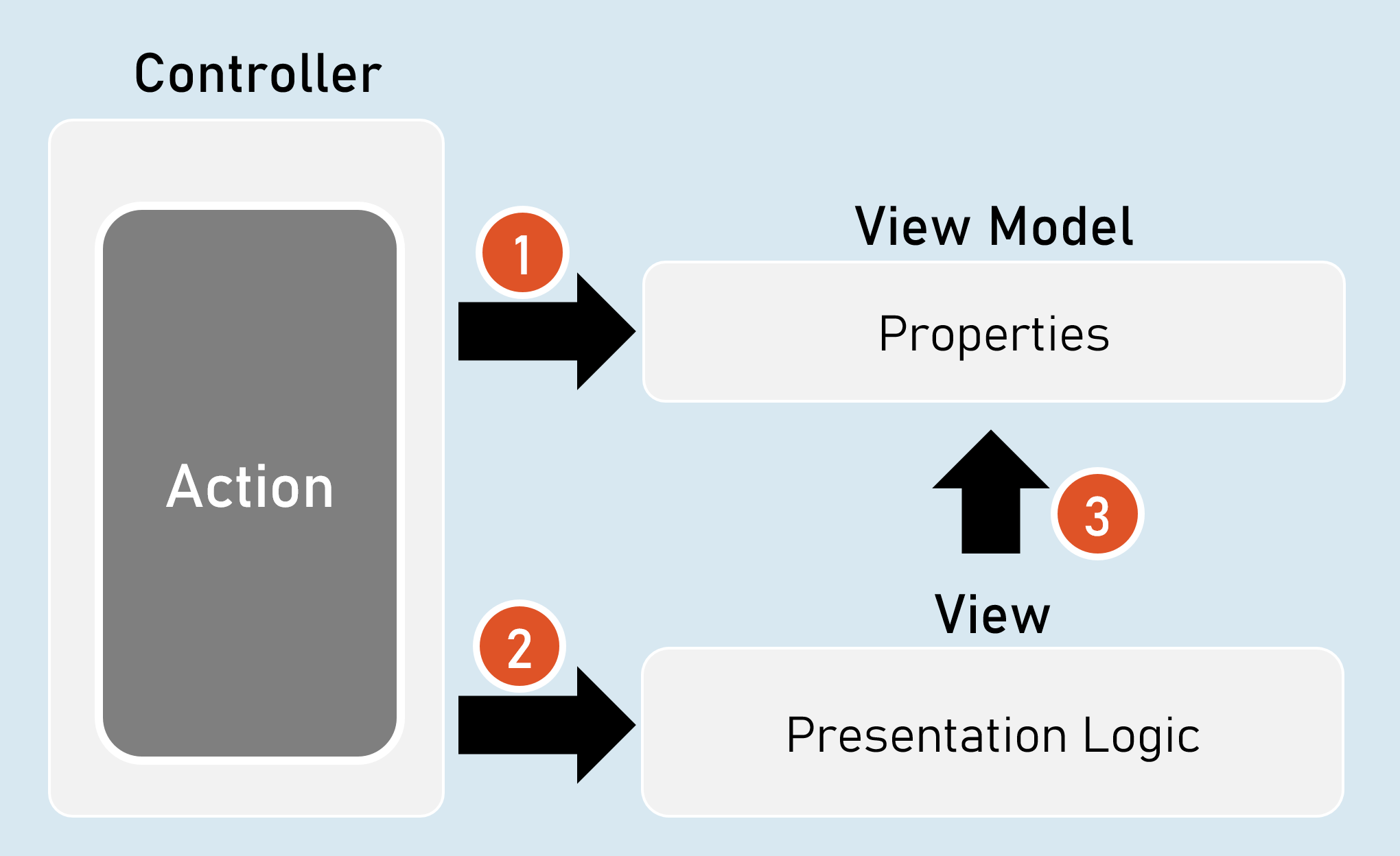
Benefits / Goals of MVC architectural pattern

* Clean separation of concerns
* Each component (model, view and controller) performs single responsibility.
* Identifying and fixing errors will be easy.
* Each component (model, view and controller) can be developed independently.
* In practical, both view and controller depend on the model.
* Model doesn't depend on neither view nor the controller.
* This is one of the key benefits of the 'clean separation'.
* This separation allows the model to be built and tested independently.
* Unit testing each individual component is easier.

Views

View is a web page (.cshtml) that is responsible for containing presentation logic that merges data along with static design code (HTML).

* Controller creates an object of ViewModel and fills data in its properties.
* Controller selects an appropriate view and invokes the same view & supplies object of ViewModel to the View.
* View access the ViewModel.



* View contains HTML markup with Razor markup (C# code in view to render dynamic content).
* Razor is the view engine that defines syntax to write C# code in the view. @ is the syntax of Razor syntax.
* View is NOT supposed to have lots of C# code. Any code written in the view should relate to presenting the content (presentation logic).
* View should neither directly call the business model, nor call the controller's action methods. But it can send requests to controllers.

Razor View Engine

**Razor Code Block**

1. @{
3. C# / html code here
5. }

Razor code block is a C# code block that contains one or more lines of C# code that can contain any statements and local functions.

**Razor Expressions**

1. @Expression
2. --or--
3. @(Expression)

Razor expression is a C# expression (accessing a field, property or method call) that returns a value.

**Razor - If**

1. @if (condition) {
2. C# / html code here
3. }

**Razor - if…else**

1. @if (condition) {
2. C# / html code here
3. }
4. else {
5. C# / html code here
6. }

Else…if and nested-if also supported.

**Razor - Switch**

1. @switch (variable) {
2. case value1: C# / html code here; break;
3. case value2: C# / html code here; break;
4. default: C# / html code here; break;
5. }

**Razor - foreach**

1. @foreach (var variable in collection ) {
2. C# / html code here
3. }

**Razor - for**

1. @for (initialization; condition; iteration) {
2. C# / html code here
3. }

**Razor - Literal**

1. @{
2. @: static text
3. }

**Razor - Literal**

<text>static text</text>

**Razor - Local Functions**

1. @{
2. return\_type method\_name(arguments) {
3. C# / html code here
4. }
5. }

The local functions are callable within the same view.

Razor - Members

**Razor - Methods, Properties, Fields**

1. @functions {
2. return\_type method\_name(arguments) {
3. C# / html code here
4. }
6. data\_type field\_name;
8. data\_type property\_name
9. {
10. set { … }
11. get { … }
12. }
13. }

The members of razor view can be accessible within the same view.

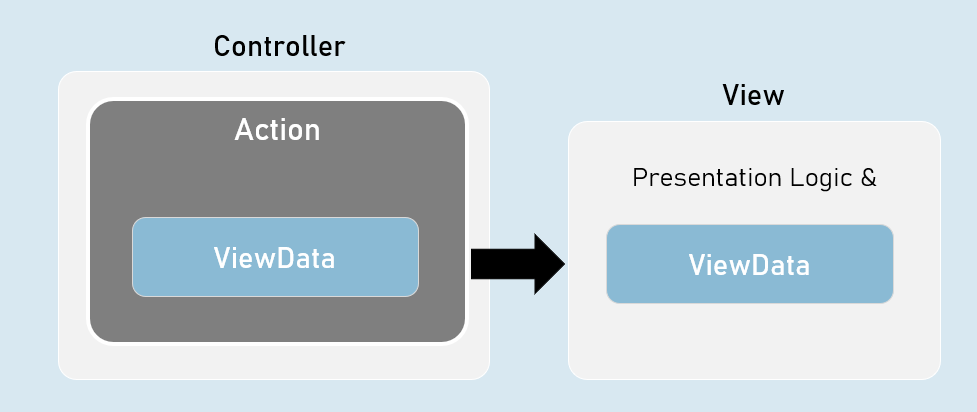
**Html.Raw( )**

1. @{
2. string variable = "html code";
3. }
5. @Html.Raw(variable) //prints the html markup without encoding (converting html tags into plain text)

ViewData

ViewData is a dictionary object that is automatically created up on receiving a request and will be automatically deleted before sending response to the client.

It is mainly used to send data from controller to view.



ViewData is a property of Microsoft.AspNetCore.Mvc.Controller class and Microsoft.AspNetCore.Mvc.Razor.RazorPage class.

It is of Microsoft.AspNet.Mvc.ViewFeatures.ViewDataDictionary type.

1. namespace Microsoft.AspNetCore.Mvc
2. {
3. public abstract class Controller : ControllerBase
4. {
5. public ViewDataDictionary ViewData { get; set; }
6. }
7. }

* It is derived from IDictionary<KeyValuePair<string, object>> type.
* That means, it acts as a dictionary of key/value pairs.
* Key is of string type.
* Value is of object type.

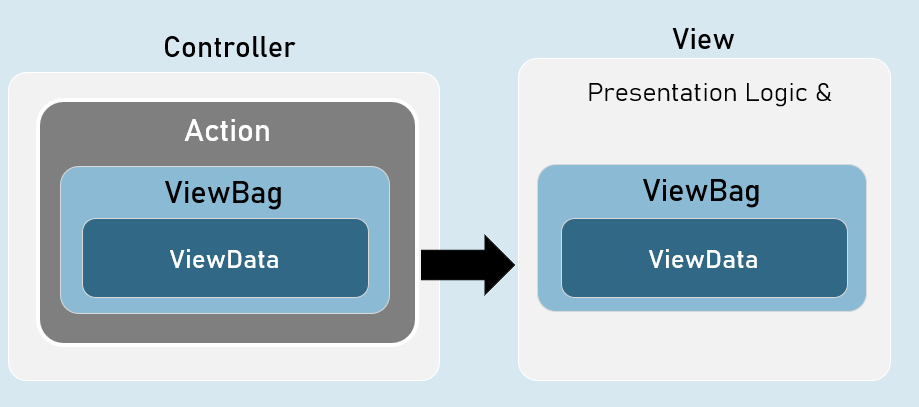
ViewData - Properties and Methods

* int Count { get; set; } //gets the number of elements.
* [string Key] //Gets or sets an element.
* Add(string key, object value) //Adds a new element.
* ContainsKey(string key) //Determines whether the specified key exists or not.
* Clear() //Clears (removes) all elements.

ViewBag

ViewBag is a property of Controller and View, that is used to access the ViewData easily.

ViewBag is 'dynamic' type.



ViewBag is a property of Microsoft.AspNetCore.Mvc.Controller class and Microsoft.AspNetCore.Mvc.Razor.RazorPageBase class.

It is of dynamic type.

1. namespace Microsoft.AspNetCore.Mvc
2. {
3. public abstract class Controller : ControllerBase
4. {
5. public dynamic ViewBag { get; set; }
6. }
7. }

The 'dynamic' type similar to 'var' keyword.

But, it checks the data type and at run time, rather than at compilation time.

If you try to access a non-existing property in the ViewBag, it returns null.

[string Key] //Gets or sets an element.

Benefits of 'ViewBag' over ViewData

ViewBag's syntax is easier to access its properties than ViewData.

Eg: ViewBag.property [vs] ViewData["key"]

You need NOT type-cast the values while reading it.

Eg: ViewBag.object\_name.property

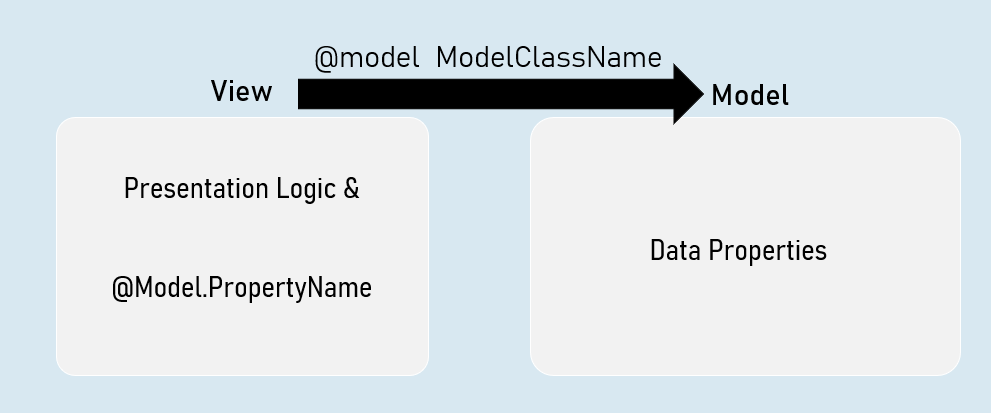
[vs]

(ViewData["key"] as ClassName).Property

Strongly Typed Views

Strongly Typed View is a view that is bound to a specified model class.

It is mainly used to access the model object / model collection easily in the view.



Benefits of Strongly Typed Views

* You will get Intellisense while accessing model properties in strongly typed views, since the type of model class was mentioned at @model directive.
* Property names are compile-time checked; and shown as errors in case of misspelled / non-existing properties in strongly typed views.
* You will have only one model per one view in strongly typed views.
* Easy to identify which model is being accessed in the view.

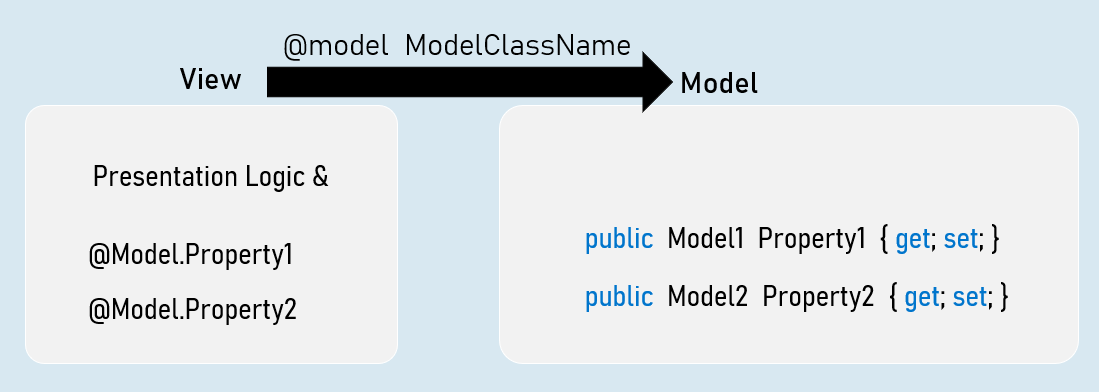
Helper methods in Controller to invoke a View

* return View( ); //View name is the same name as the current action method.
* return View(object Model ); //View name is the same name as the current action method & the view can be a strongly-typed view to receive the supplied model object.
* return View(string ViewName); //View name is explicitly specified.
* return View(string ViewName, object Model ); //View name is explicitly specified & the view can be a strongly-typed view to receive the supplied model object.

Strongly Typed Views

Strongly Typed View can be bound to a single model directly.

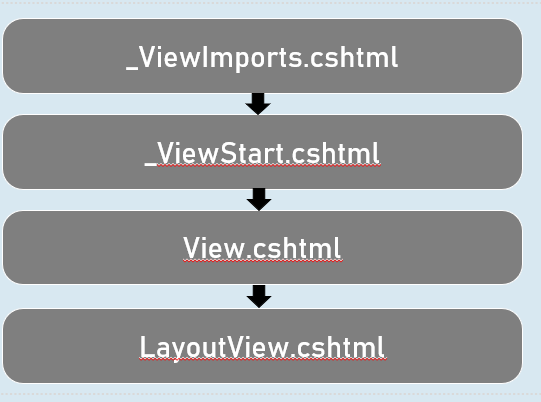
But that model class can have reference to objects of other model classes.

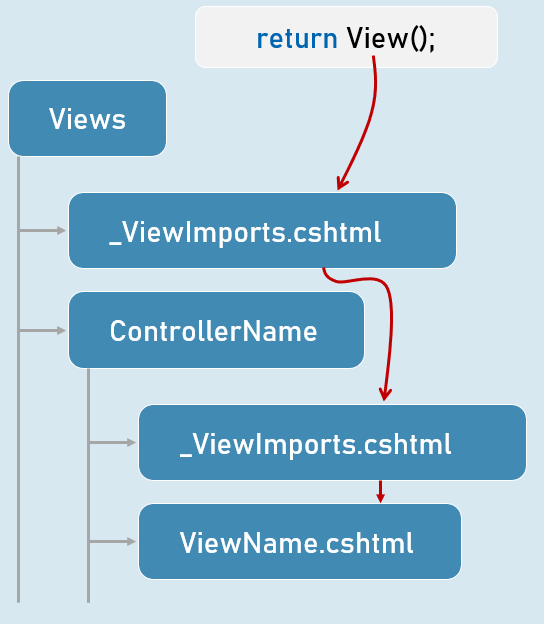


ViewImports.cshtml

ViewImports.cshtml is a special file in the "Views" folder or its subfolder, which executes automatically before execution of a view.

It is mainly used to import common namespaces that are to imported in a view.





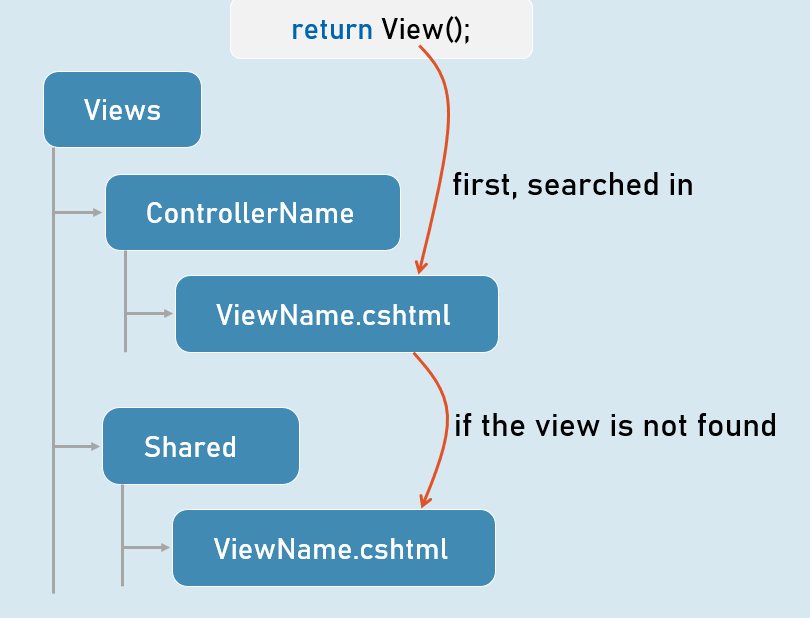
Shared Views

Shared views are placed in "Shared" folder in "Views" folder.

They are accessible from any controller, if the view is NOT present in the "Views\ControllerName" folder.



View Resolution



# **7. Layout Views [MVC]**

Layout Views

In ASP.NET Core MVC, a layout view is a master template that defines the common structure and elements that you want to share across multiple pages within your application. This could include headers, navigation bars, footers, sidebars, and other common UI components.

**Why Use Layout Views?**

* **Reusability:** Avoid repeating common HTML, CSS, and JavaScript code in every view.
* **Consistency:** Maintain a uniform look and feel across your entire application.
* **Simplified Maintenance:** Make updates to the layout view once, and the changes automatically apply to all views that use it.
* **Improved Organization:** Separate the structure of your pages from their specific content.

**How Layout Views Work**

1. **The \_Layout.cshtml File:** The main layout view is typically named \_Layout.cshtml and placed in the Views/Shared folder. You can also have multiple layout files if needed.
2. **RenderBody() Method:** The layout view contains a special Razor method called RenderBody(). This is a placeholder where the content of the specific view (e.g., Index.cshtml, About.cshtml) will be inserted.
3. **Specifying the Layout:** In your individual views, you use the Layout property to indicate which layout view to use. This can be done in two ways:
   * **Implicitly (Convention-based):** If you don't explicitly specify a layout, ASP.NET Core MVC automatically uses the \_Layout.cshtml file in the Views/Shared folder.
   * **Explicitly:** Use the Layout property at the top of your view:
     1. @{
     2. Layout = "~/Views/Shared/\_Layout.cshtml";
     3. }

**Code Example**

**\_Layout.cshtml (Layout View)**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>@ViewData["Title"]</title>
5. <link href="~/StyleSheet.css" rel="stylesheet" />
6. </head>
7. <body>
8. <div class="container">
9. <div class="navbar">
10. <div class="navbar-brand">AspNet Core App</div>
11. <ul>
12. <li><a href="/">Home</a></li>
13. </ul>
14. </div>
16. <div class="page-content">
17. @RenderBody()
18. </div>
19. </div>
20. </body>
21. </html>

* **@ViewData["Title"]:** Dynamically sets the title of the page. This value will be provided by the individual views.
* **RenderBody():** This placeholder will be replaced by the content of the specific view being rendered.

**Index.cshtml (Content View)**

1. @{
2. Layout = "~/Views/Shared/\_Layout.cshtml"; // Explicitly specifies the layout
3. ViewData["Title"] = "Home";
4. }
6. <h1>Home</h1>
7. <p>Welcome to home page</p>

* **Layout Property:** Sets the layout for this view.
* **ViewData["Title"]:** Sets the value for the Title property, which will be used in the layout view.

**About.cshtml (Content View)**

1. @{
2. Layout = "~/Views/Shared/\_Layout.cshtml";
3. ViewData["Title"] = "About";
4. }
6. <h1>About</h1>
7. <p>About company</p>

This view follows the same structure, setting the title to "About".

**Controller (HomeController.cs)**

The controller actions simply return the View() result, which triggers the view rendering process. Because the layout is not defined within the action, the default \_Layout.cshtml is used.

**Notes**

* **Structure and Consistency:** Layout views help you establish a consistent page structure throughout your application.
* **DRY Principle:** Don't Repeat Yourself (DRY) by avoiding duplication of common elements in your views.
* **RenderBody():** This method is essential for inserting the specific view's content into the layout.
* **Layout Property:** Use it to explicitly specify or override the layout for a particular view.
* **Partial Views:** You can use partial views within layout views to further break down your UI into reusable components.
* **Sections:** The RenderSection method allows you to define placeholders in the layout view and have individual views provide content for them.

\_ViewStart.cshtml

The \_ViewStart.cshtml file is a special file in ASP.NET Core MVC that allows you to configure common settings that apply to all views within a specific directory and its subdirectories. This file typically resides in the same folder as your views, but its effects cascade down to child folders.

**What Can You Put in \_ViewStart.cshtml?**

The primary purpose of \_ViewStart.cshtml is to specify the default layout for your views. You can also use it to set other common properties for your views, but using it for the layout is the most common practice.

* **Layout Property:** Sets the default layout for all views in the directory and its subdirectories.
  1. @{
  2. Layout = "\_Layout"; // Sets the default layout to \_Layout.cshtml
  3. }

You can specify the full path to the layout view if needed.

* **Other Properties:** While less common, you could set other properties of the View object within \_ViewStart.cshtml. However, it's generally recommended to keep the main purpose of this file to define the default layout.

**How It Works**

When ASP.NET Core MVC processes a request for a view, it looks for a \_ViewStart.cshtml file in the following order:

1. **The same directory as the view:** If found, the layout specified in this file is applied to the view.
2. **Parent directories:** It then checks parent directories of the view, recursively, until it reaches the root Views folder.
3. **Root Views folder:** Finally, it checks the \_ViewStart.cshtml file in the root Views folder. If a layout is specified there, it will be used as the default for any view that hasn't had a layout explicitly set.

**Code Example:**

Imagine you have the following file structure:

1. Views/
2. \_ViewStart.cshtml (Layout = "\_Layout")
3. Home/
4. Index.cshtml
5. About.cshtml
6. Products/
7. \_ViewStart.cshtml (Layout = "\_ProductLayout")
8. Index.cshtml
9. Details.cshtml

* The Home/Index.cshtml and Home/About.cshtml views will use \_Layout.cshtml as their layout (inherited from the root Views folder's \_ViewStart.cshtml).
* The Products/Index.cshtml and Products/Details.cshtml views will use \_ProductLayout.cshtml as their layout (specified in their own directory's \_ViewStart.cshtml).

**Benefits of Using \_ViewStart.cshtml**

* **DRY (Don't Repeat Yourself):** Avoids repeating the layout specification in every view file.
* **Centralized Configuration:** Makes it easy to change the default layout for an entire section of your views.
* **Maintainability:** Simplifies updating your application's layout without modifying individual views.

**Important Considerations**

* **Override with Layout Property:** You can still override the default layout in individual views by explicitly setting the Layout property.
* **Cascading Behavior:** Understand the cascading behavior when you have multiple \_ViewStart.cshtml files in different directories.
* **Scoping:** Be aware that \_ViewStart.cshtml affects views within the same directory and its subdirectories.

Dynamic Layout Views

While the \_ViewStart.cshtml file is excellent for setting default layouts, sometimes you need more flexibility to choose different layouts based on specific conditions or logic within your views. This is where dynamic layout selection comes into play.

**The Layout Property: Your Key to Flexibility**

The Layout property, which you typically set in the \_ViewStart.cshtml file, can also be set directly within individual views. This allows you to dynamically change the layout used for rendering a particular view based on the data or context of the request.

**How to Apply Dynamic Layouts**

1. **Conditional Logic:** Use conditional statements (if, else, switch) in your Razor view to determine which layout to apply based on specific criteria.
2. **Set Layout Property:** Within the conditional blocks, assign the appropriate layout path to the Layout property.
3. **Render the View:** The MVC framework will use the dynamically assigned layout to render the final page.

**Code Example: Dynamic Layout Based on ViewBag.ProductID**

1. // Views/Products/Search.cshtml
3. @{
4. ViewData["Title"] = "Search Products";
5. if (ViewBag.ProductID != null)
6. {
7. Layout = "~/Views/Shared/\_ProductsLayout.cshtml";
8. }
9. }
11. <h1>Search Products</h1>
12. <p>search details here</p>

**Explanation**

1. **Checking Condition:** The code first checks if ViewBag.ProductID is not null. This suggests that the view is being rendered in the context of a specific product (since the product ID is available).
2. **Setting Layout:** If ViewBag.ProductID is not null, the Layout property is set to ~/Views/Shared/\_ProductsLayout.cshtml. This means that the view will use a different layout than the default one potentially specified in the \_ViewStart.cshtml.
3. **Default Layout:** If ViewBag.ProductID is null, the view will fall back to the default layout, which is either specified in the \_ViewStart.cshtml file or the application's global configuration.

**Additional Considerations**

* **Flexibility:** You can use any condition you need to determine the appropriate layout. This could involve checking model properties, user roles, configuration settings, or other dynamic values.
* **Performance:** If you have complex layout selection logic, consider moving it to your controller or a helper method to keep your views clean.
* **Maintainability:** Organize your layouts clearly, and document the conditions under which each layout is used.

**Important Considerations**

* **View Model:** The preferred way to pass data from the controller to the view is by using a strongly typed view model. ViewBag is not type-safe and could lead to run-time errors if not used with care.
* **Error Handling:** Handle the case where ViewBag.ProductID might not be set correctly or could be null when you expect it to have a value. This could help avoid unexpected runtime errors.

**Example Usage in Controller (ProductsController.cs)**

1. public IActionResult Search(int? productId)
2. {
3. // ... (retrieve search results and potentially set ViewBag.ProductID)...
5. ViewBag.ProductID = productId;
6. return View();
7. }

By dynamically applying different layouts, you can create a more tailored and engaging user experience, adapting the structure and presentation of your views to match the specific context of each request. Let me know if you have any other questions!

Layout View Sections

In ASP.NET Core MVC, layout view sections allow you to define specific placeholders or areas within your layout view where content from individual views can be inserted. This provides a powerful way to control the placement and organization of content within your page's overall structure.

**Key Concepts**

* **@RenderSection() in Layout:**
  + **Purpose:** Defines a placeholder (section) in the layout view where content can be inserted from specific views.
  + **Syntax:**
    1. @RenderSection("sectionName", required: true/false)
    2. sectionName: A unique name for the section.
    3. required: A boolean flag indicating whether the section is required (default is true). If true, views using this layout must provide content for this section; otherwise, an exception is thrown.
* **@section in Content Views:**
  + **Purpose:** Provides the content to be inserted into the corresponding section in the layout view.
  + **Syntax:**
    1. @section sectionName {
    2. ... }

**Code Example**

**\_Layout.cshtml (Layout View)**

1. <div class="footer-content">
2. @RenderSection("footer\_section", false) // Optional footer section
3. </div>

**@RenderSection("footer\_section", false):** This line defines a section named footer\_section. The false argument indicates that this section is *optional*. Views using this layout can choose whether or not to provide content for it.

**Views/Home/Contact.cshtml (Content View)**

1. @section footer\_section
2. {
3. <p>Contact support: 98348734873984734</p>
4. }

**@section footer\_section { ... }:** This code block defines the content that will be rendered in the footer\_section of the layout view.

**How It Works**

1. When ASP.NET Core renders a view that uses the \_Layout.cshtml layout, it encounters the @RenderSection("footer\_section", false) line.
2. If the content view (e.g., Contact.cshtml) provides a @section footer\_section block, that content is inserted into the layout at the @RenderSection location.
3. If the content view does not provide a @section footer\_section block, and the section is marked as optional (false), no content is rendered in that section. If the section is required (true), an error will be thrown.

**Notes**

* **Flexibility:** Sections let you dynamically control the placement of content in your layout.
* **Reusability:** You can reuse the same layout with different content views, each providing their own section content.
* **Organization:** Keep your layout code clean and focused by breaking down the page into logical sections.
* **Optional vs. Required:** Use the required flag in @RenderSection wisely to control whether views must provide content for a section.
* **Default Content:** You can provide default content within the @RenderSection block in the layout view, which will be rendered if the content view doesn't override it.

Nested Layout Views

In ASP.NET Core MVC, nested layout views allow you to create hierarchical layouts where one layout view inherits from another. This creates a structure where a child layout can define additional content or override specific sections of its parent layout, while still inheriting the overall structure and common elements.

**Why Use Nested Layouts?**

* **Enhanced Reusability:** Achieve even greater reusability by creating multiple layers of layout views. This is particularly useful for large applications with distinct sections or modules that share some common elements but also have unique layout requirements.
* **Fine-Grained Control:** Customize specific areas of a layout without duplicating large chunks of code. You can create a base layout for the entire application and then have specialized layouts for different sections, such as the header, navigation, or footer.
* **Maintainability:** Manage and update layout elements more efficiently by making changes at the appropriate level in the layout hierarchy.

**Implementing Nested Layouts**

1. **Parent Layout:** Create a parent layout view that defines the main structure and common elements. Typically, this is your \_Layout.cshtml file in the Views/Shared folder.
2. **Child Layout:** Create a child layout view that inherits from the parent layout. In the child layout, you can:
   * Add new sections or content.
   * Override sections defined in the parent layout using @section.
   * Set the Layout property to the parent layout's path.
3. **Content View:** In your content views, you don't need to explicitly specify the child layout. It will automatically inherit from the parent layout if you don't set a Layout property.

**Code Example: In-Depth**

**\_MasterLayout.cshtml (Parent Layout)**

1. <!DOCTYPE html>
2. <html>
3. <head>
4. </head>
5. <body>
6. <div class="container">
7. <div class="header">
8. ASP.NET Core Demo App
9. </div>
10. </div>
11. <div>
12. @RenderBody()
13. </div>
14. </body>
15. </html>

**RenderBody():** The key placeholder where the child layout's content will be inserted.

**\_Layout.cshtml (Child Layout)**

1. @{
2. Layout = "~/Views/Shared/\_MasterLayout.cshtml"; // Inherit from \_MasterLayout.cshtml
3. }
5. <!DOCTYPE html>
6. <html>
7. <head>
8. </head>
9. <body>
10. <div class="container">
11. <div class="navbar">
12. </div>
14. <div class="page-content">
15. @RenderBody()
16. </div>
18. <div class="footer-content">
19. @RenderSection("footer\_section", false)
20. </div>
21. </div>
22. </body>
23. </html>

* **Layout = "~/Views/Shared/\_MasterLayout.cshtml":** Specifies the parent layout.
* **Additional Content:** The child layout adds the navbar, page content area, and footer section.
* **RenderBody():** Inherits this placeholder from the parent layout, so the content view's content will ultimately be rendered here.

**\_ViewStart.cshtml**

1. @{
2. Layout = "~/Views/Shared/\_Layout.cshtml";
3. }

This sets the default layout for all views in the application to be \_Layout.cshtml.

**How the Nesting Works**

1. When a view is rendered, ASP.NET Core first looks for a \_ViewStart.cshtml file.
2. \_ViewStart.cshtml specifies \_Layout.cshtml as the default layout.
3. \_Layout.cshtml, in turn, specifies \_MasterLayout.cshtml as its parent.
4. The content view is rendered within the @RenderBody() section of \_Layout.cshtml.
5. The combined content of \_Layout.cshtml is then rendered within the @RenderBody() section of \_MasterLayout.cshtml.

**Notes**

* **Flexibility:** Nested layouts provide flexibility to create complex and modular UI structures.
* **Inheritance:** Child layouts inherit the structure and content of their parent layouts.
* **Overriding:** Child layouts can override or extend sections defined in the parent layout.
* **Maintainability:** Makes it easier to update and manage the overall layout of your application.

Key points to remember

**Layout Views: Mastering Page Structure**

* **Purpose:** Define a common template for the structure and shared elements of your web pages.
* **Benefits:**
  + **Reusability:** Avoid code duplication.
  + **Consistency:** Maintain a uniform look and feel across pages.
  + **Maintainability:** Update the layout once, and changes apply everywhere.
* **\_Layout.cshtml:** The main layout file, usually placed in the Views/Shared folder.
* **RenderBody():** Placeholder in the layout view where the specific view's content is inserted.
* **Layout Property:**
  + Used in content views to specify which layout to use.
  + Can be set dynamically based on conditions in the view.
  + If not set, the default \_Layout.cshtml is used.

**Sections**

* **Purpose:** Define placeholders in the layout for content from specific views.
* **@RenderSection("sectionName", required: true/false):** Declares a section in the layout.
* **@section sectionName { ... }:** Provides content for the section in the content view.
* **Optional vs. Required:** Control whether a section is mandatory or optional using the required flag.

**Nested Layouts**

* **Purpose:** Create hierarchical layouts where one layout inherits from another.
* **Benefits:** Increased reusability and fine-grained control over layout customization.
* **Implementation:**
  + Set the Layout property in the child layout to point to the parent layout.
  + Use @RenderBody() in both layouts.

**\_ViewStart.cshtml**

* **Purpose:** Set the default layout for all views in a directory and its subdirectories.
* **Location:** Typically placed in the Views folder or in specific subfolders.
* **Layout Property:** Used within \_ViewStart.cshtml to specify the default layout file.

**Additional Tips**

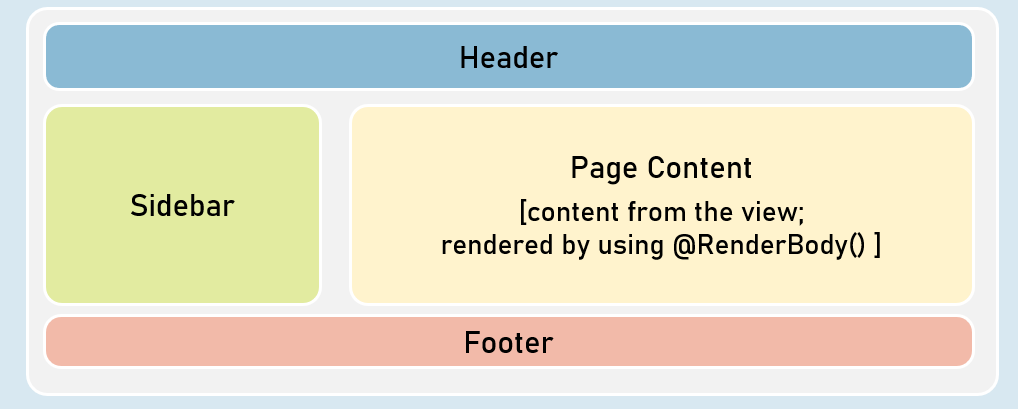
* **View Models:** Pass data to layout views using strongly typed view models.
* **Partial Views:** Break down complex layouts into smaller, reusable partial views.
* **Sections for Flexibility:** Use sections to make your layouts adaptable to different content views.
* **Error Handling:** Handle potential errors (e.g., missing sections) gracefully.

**Interview Tips**

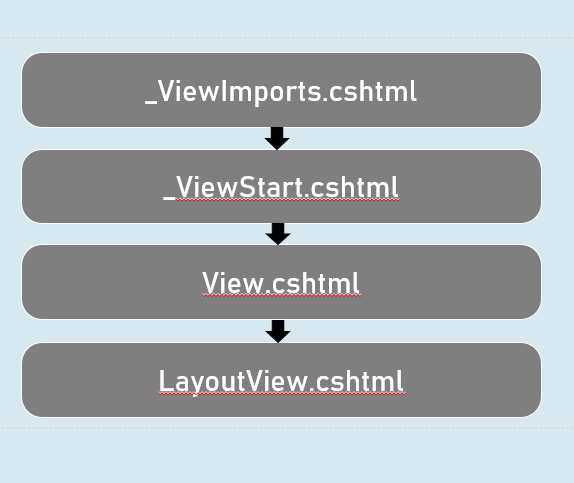
* **Explain the Hierarchy:** Clearly articulate how nested layouts and \_ViewStart.cshtml work together to define page structure.
* **Code Examples:** Be prepared to write code snippets demonstrating the use of layout views, sections, and dynamic layout selection.
* **Best Practices:** Discuss how to use layout views to create maintainable and reusable code.
* **Troubleshooting:** Explain how you would debug issues related to layout views (e.g., incorrect layout rendering).

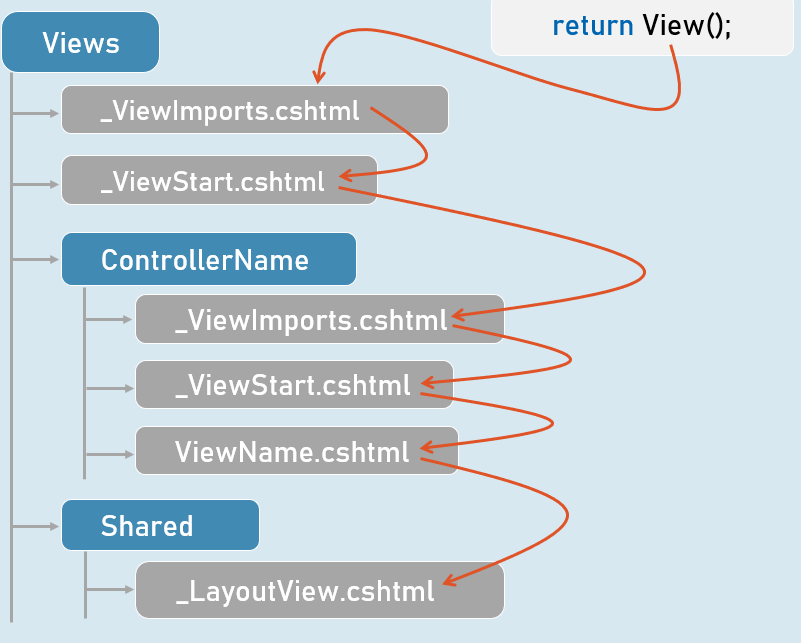
Layout Views

Layout View is a web page (.cshtml) that is responsible for containing presentation logic template (commonly the html template with header, sidebar, footer etc.)



Order of Views Execution





Layout Views

* The @RenderBody() method presents only in layout view to represent the place where exactly the content from the view has to be rendered.
* The "Layout" property of the view specifies path of the layout view.
* It can be dynamically set in the view.
* Both View and Layout View shares the same ViewData object.
* So it is possible to send data from view to layout, since the view executes first.
* The css files / js files imported in layout view will be applicable to view also, because the content of view will be merged into the layout view at run time.

# **8. Partial Views [MVC]**

Partial Views

In ASP.NET Core MVC, a partial view is a reusable chunk of Razor markup (.cshtml) that can be embedded within other views. They are designed to encapsulate specific UI elements, such as lists, forms, or widgets, allowing you to avoid repeating code and maintain a more organized structure.

**Why Use Partial Views?**

* **Reusability:** A single partial view can be used in multiple views, saving development time and reducing the potential for inconsistencies.
* **Modularity:** Partial views help break down complex views into smaller, more manageable components, making your code easier to read and maintain.
* **Dynamic Content:** You can pass data to partial views, making them dynamic and adaptable to different contexts.

**Notes**

* **Naming Convention:** Partial views are typically named with a leading underscore (e.g., \_ListPartialView.cshtml). This convention helps distinguish them from full views.
* **Location:** By default, partial views are located in the Views/Shared folder or in the same folder as the view that uses them. You can override the default location by specifying the full path.
* **Rendering:** You can render partial views in your main views using:
  + **Tag Helpers:** <partial name="\_ListPartialView" /> or <partial name="\_ListPartialView" model="yourModel" />
  + **HTML Helper:** @await Html.PartialAsync("\_ListPartialView") or @await Html.PartialAsync("\_ListPartialView", yourModel)

**Code Example**

**\_ListPartialView.cshtml (Partial View)**

1. <div class="list-container">
2. @{
3. //ViewBag.ListTitle = "Updated"; // Not recommended
4. }
6. <h3>@ViewBag.ListTitle</h3>
7. <ul class="list">
8. @foreach (string item in ViewBag.ListItems)
9. {
10. <li>@item</li>
11. }
12. </ul>
13. </div>

**ViewBag:** This example uses ViewBag to pass the ListTitle and ListItems data to the partial view. However, it is generally recommended to use a strongly typed model to pass data to partial views for better type safety and maintainability.

**Views/Home/Index.cshtml (Main View)**

1. <h1>Home</h1>
3. <partial name="\_ListPartialView" />
5. @{
6. var myViewData = new ViewDataDictionary(ViewData); // Create a new ViewDataDictionary
7. myViewData["ListTitle"] = "Countries";
8. myViewData["ListItems"] = new List<string>() { "USA", "Canada", "Japan", "Germany", "India" };
9. }
10. <div class="box">
11. <partial name="\_ListPartialView" view-data="myViewData" />
12. </div>
14. <h3>ListTitle in View: @ViewData["ListTitle"]</h3>

This main view renders the \_ListPartialView twice:

1. **First Rendering:**
   * The @ViewBag.ListTitle value in the partial view will be null (as it wasn't explicitly set before the partial view is rendered).
   * The @ViewBag.ListItems will also be null and the loop will not execute.
   * A new ViewDataDictionary named myViewData is created.
2. **Second Rendering:**
   * The myViewData dictionary is used to pass the ListTitle ("Countries") and ListItems (a list of countries) to the partial view.
   * The partial view will display the list of countries because ViewBag.ListItems is not null.
   * After the partial view has rendered, the ViewData["ListTitle"] still refers to the original value ("Asp.Net Core Demo App") set in the controller. This is because the myViewData dictionary is a separate instance of ViewDataDictionary.

**Views/Home/About.cshtml (Main View)**

1. <h1>About</h1>
3. <h2>About company here</h2>
4. <p>Lorem Ipsum is simply dummy text...</p>
5. @{
6. await Html.RenderPartialAsync("\_ListPartialView");
7. }

This view also renders \_ListPartialView using the Html.RenderPartialAsync method. The result is the same as the first render in Index.cshtml.

**Notes**

* **Reusability:** Partial views are essential for DRY (Don't Repeat Yourself) development in your views.
* **Flexibility:** You can pass data to partial views to make them dynamic.
* **Rendering Options:** Use tag helpers or HTML helpers to render partial views.
* **Data Sharing:** Ensure that your partial views have access to the necessary data using view models.
* **Organization:** Partial views contribute to a well-structured and maintainable codebase.
* **View Reusability:** The same partial view can be rendered multiple times on the same page, each time with a different set of data.

Strongly Typed Partial Views

A strongly typed partial view, just like a strongly typed view, is associated with a specific model class using the @model directive. This means that the partial view has direct access to the properties and methods of that model, providing compile-time type checking and IntelliSense support within the partial view.

**Benefits of Strongly Typed Partial Views**

* **Type Safety:** Catches errors during development due to mismatched data types or property names.
* **IntelliSense:** Provides autocompletion and suggestions for model properties, leading to faster and more accurate coding.
* **Refactoring:** Makes it easier to update partial views when your model changes.

**How to Use Strongly Typed Partial Views**

1. **Create a Model Class:** Define a class that represents the data structure you want to pass to your partial view.
2. **Use @model in the Partial View:** Add the @model directive at the top of your partial view, specifying the model class: @model YourNamespace.YourModel.
3. **Pass the Model:** When rendering the partial view from your main view, pass an instance of your model class using the model attribute of the <partial> tag helper or the model parameter of the Html.PartialAsync method.

**Code Example**

**ListModel.cs (Model)**

1. namespace PartialViewsExample.Models
2. {
3. public class ListModel
4. {
5. public string ListTitle { get; set; } = "";
6. public List<string> ListItems { get; set; } = new List<string>();
7. }
8. }

This model will be used to represent the data for the list that's rendered in the partial view.

**\_ListPartialView.cshtml (Strongly Typed Partial View)**

1. @model ListModel // Specify the model type
3. <div class="list-container">
4. <h3>@Model.ListTitle</h3>
5. <ul class="list">
6. @foreach (string item in Model.ListItems)
7. {
8. <li>@item</li>
9. }
10. </ul>
11. </div>

* **@model ListModel:** This directive indicates that this partial view expects a model of type ListModel.
* **Access Model Properties:** You can directly access properties of the model object using @Model.PropertyName.

**Index.cshtml (Main View)**

1. @using PartialViewsExample.Models
3. <h1>Home</h1>
5. @{
6. ListModel listModel = new ListModel();
7. listModel.ListTitle = "Countries";
8. listModel.ListItems = new List<string>() { "USA", "Canada", "Japan", "Germany", "India" };
9. }
10. <partial name="\_ListPartialView" model="listModel" />

* **Creating the Model:** An instance of ListModel is created and populated with data.
* **Passing the Model:** The model attribute in the <partial> tag helper is used to pass the listModel to the \_ListPartialView partial view.

**About.cshtml (Main View)**

1. @{
2. ListModel listModel = new ListModel();
3. listModel.ListTitle = "Programming Languages";
4. listModel.ListItems = new List<string>()
5. {
6. "Java",
7. "C#",
8. "Python"
9. };
11. await Html.RenderPartialAsync("\_ListPartialView", listModel);
12. }

Same as the Index.cshtml, but using the Html.RenderPartialAsync HTML helper.

**Notes**

* **Strongly Typed:** Use @model to specify the type of data expected by the partial view.
* **Pass the Model:** Pass an instance of your model class when rendering the partial view.
* **Type Safety and IntelliSense:** Benefit from compile-time checking and code completion when working with model properties.
* **Best Practice:** Strongly typed partial views promote clean, maintainable, and less error-prone code.

PartialViewResult

In ASP.NET Core MVC, a PartialViewResult is a specific type of ActionResult designed for returning partial views from your controller actions. Partial views, as you know, are reusable chunks of Razor markup (.cshtml) that can be embedded within other views. However, instead of being rendered as a full page, they're intended to be a fragment of HTML that can be inserted into a larger view.

**Why Use PartialViewResult?**

* **Dynamic Content Delivery:** You can use PartialViewResult to load content dynamically into your main view using AJAX or other client-side techniques.
* **Separation of Concerns:** This result type helps keep your controller actions focused on returning data and leaves the rendering of that data to the view.
* **Simplified Testing:** Since PartialViewResult doesn't involve rendering a complete page, it's often easier to unit test your action methods that return partial views.

**Creating a PartialViewResult**

In your controller actions, you can return a PartialViewResult in a couple of ways:

1. **Direct Instantiation:**
2. return new PartialViewResult
3. {
4. ViewName = "\_ListPartialView", // Name of the partial view
5. ViewData = new ViewDataDictionary(ViewData, model) // Pass the model data
6. };
7. **Using the PartialView() Helper Method:**
8. return PartialView("\_ListPartialView", model); // Much simpler way

The PartialView() method is a shorthand provided by the Controller base class for conveniently creating a PartialViewResult.

**Code Example**

**Views/Home/Index.cshtml (Main View)**

1. <button class="button button-blue-back" type="button" id="button-load">Load Programming Languages</button>
2. <div class="programming-languages-content">
4. </div>
6. <script>
7. document.querySelector("#button-load").addEventListener("click", async function() {
8. var response = await fetch("programming-languages");
9. var languages = await response.text();
10. document.querySelector(".programming-languages-content").innerHTML = languages;
11. });
12. </script>

This view contains a button, a div, and some JavaScript to load the partial view's content when the user clicks a button using fetch function.

**HomeController.cs (Controller)**

1. [Route("programming-languages")]
2. public IActionResult ProgrammingLanguages()
3. {
4. ListModel listModel = new ListModel() {
5. ListTitle = "Programming Languages List",
6. ListItems = new List<string>() { "Python", "C#", "Go" }
7. };
9. return PartialView("\_ListPartialView", listModel);
10. }

In this action method:

1. **Data Preparation:** A ListModel object is created and populated with data for the list (title and items).
2. **Partial View Returned:** The PartialView() method is used to return a PartialViewResult. The method takes two arguments:
   * "\_ListPartialView": The name of the partial view file (located in Views/Shared by default).
   * listModel: The model data to pass to the partial view.
   * When this action method is called via the URL /programming-languages by the javascript code on the page, it returns the partial view along with the data that is then injected into the div with class programming-languages-content by the javascript.

**Important Points:**

* **AJAX and Dynamic Loading:** PartialViewResult is frequently used in conjunction with AJAX to load content dynamically without full page refreshes.
* **View Model (Best Practice):** Always try to use a view model to pass data to your partial views for better type safety and maintainability.
* **Caching:** Consider using output caching on your partial views to improve performance if they render data that doesn't change frequently.

Key Points to Remember

**Partial Views: Reusable View Components**

* **Purpose:** Encapsulate reusable UI elements or chunks of Razor markup (.cshtml) to avoid repetition.
* **Benefits:**
  + Increased code reusability
  + Improved code organization and maintainability
  + Dynamic content generation
* **Naming Convention:** Prefixed with an underscore (e.g., \_PartialName.cshtml).
* **Location:** Typically in Views/Shared or alongside the main view.
* **Rendering Options:**
  + **Tag Helpers:** <partial name="\_PartialName" /> or <partial name="\_PartialName" model="yourModel" />
  + **HTML Helper:** @await Html.PartialAsync("\_PartialName", model)

**ViewData in Partial Views**

* **Purpose:** Pass data to partial views from the parent view or controller.
* **Usage:**
  + In the parent view or controller: ViewData["key"] = value
  + In the partial view: @ViewData["key"]
* **Caveat:** ViewData is not strongly typed; be careful with type casting and null checks.

**Strongly Typed Partial Views**

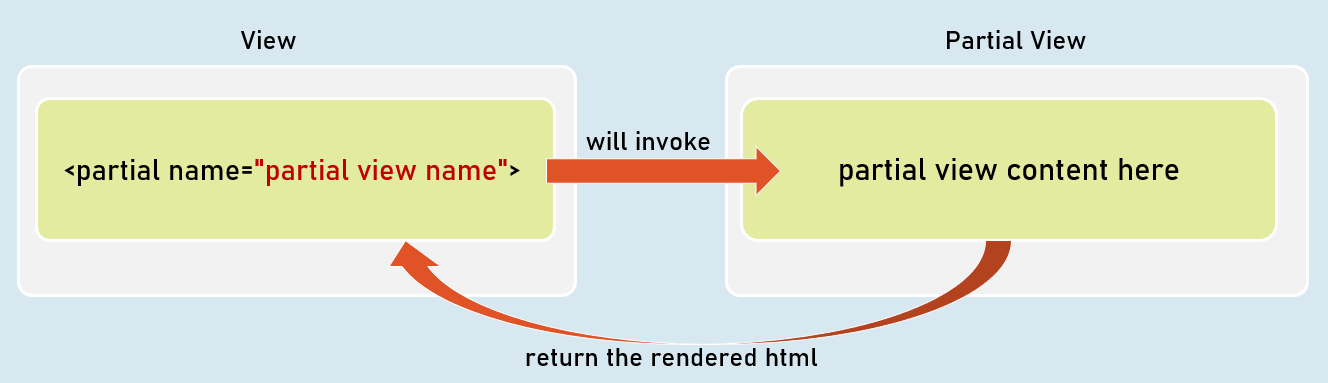
* **Purpose:** Associate a partial view with a specific model class using the @model directive.
* **Benefits:**
  + Type safety: Catches errors during development.
  + IntelliSense: Code completion for model properties.
  + Maintainability: Easier updates when models change.
* **How to Use:**
  + Create a model class.
  + Use @model YourModel in the partial view.
  + Pass a model instance when rendering: <partial model="yourModelInstance" /> or @await Html.PartialAsync("\_PartialName", yourModelInstance)

**PartialViewResult**

* **Purpose:** Return a partial view from a controller action (often for AJAX requests).
* **Creation:**
  + return PartialView("\_PartialName", model); (preferred)
  + return new PartialViewResult { ViewName = "\_PartialName", ViewData = ... };

Partial Views

Partial view is a razor markup file (.cshtml) that can't be invoked individually from the controller; but can be invoked from any view within the same web application.



Invoking Partial Views

<partial name="partial view name" />

Returns the content to the parent view.

@await Html.PartialAsync("partial view name")

Returns the content to the parent view.

@{ await Html.RenderPartialAsync("partial view name"); }

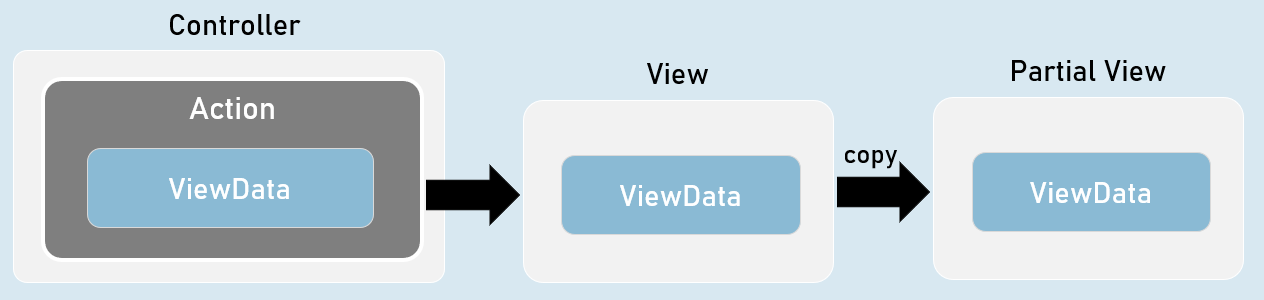
Streams the content to the browser.

Partial Views with ViewData

When partial view is invoked, it receives a *copy* of the parent view's ViewData object.

So, any changes made in the ViewData in the partial view, do NOT effect the ViewData of the parent view.

Optionally, you can supply a custom ViewData object to the partial view, if you don't want the partial view to access the entire ViewData of the parent view.



**Invoking Partial Views with View Data**

@{ await Html.RenderPartialAsync("partial view name", ViewData); }

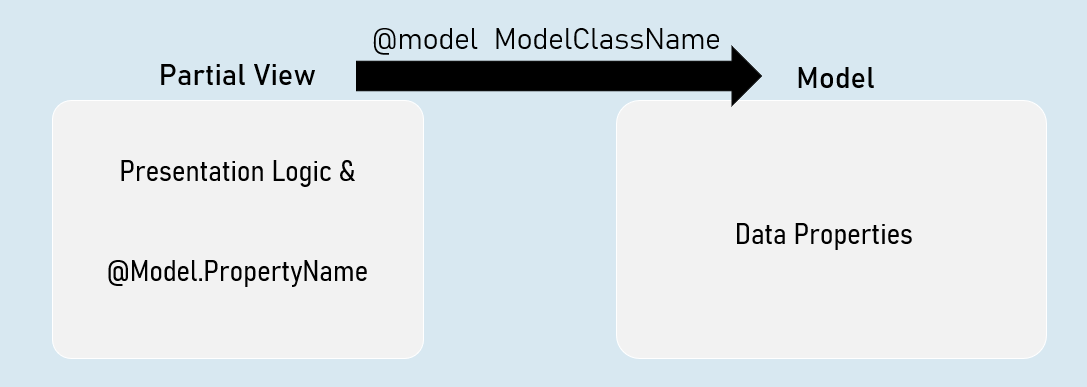
-- or --

<partial name="partial view name" view-data="ViewData" />

Strongly Typed Partial Views

Strongly Typed Partial View is a partial view that is bound to a specified model class.

So, it gets all the benefits of a strongly typed view.



**Invoking Strongly-Typed Partial View**

@{ await Html.RenderPartialAsync("partial view name", Model); }

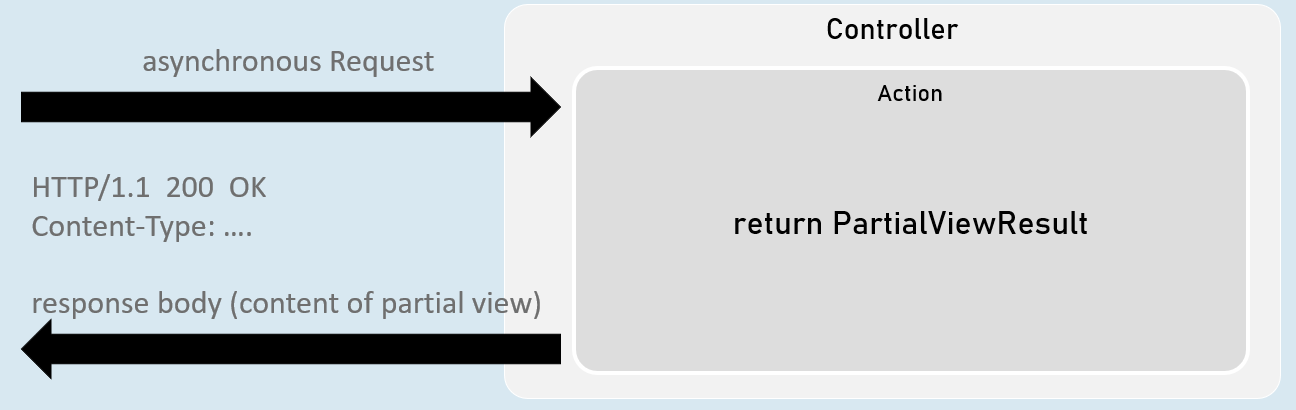
-- or --

<partial name="partial view name" model="Model" />

PartialViewResult

PartialViewResult can represent the content of a partial .

Generally useful to fetch partial view's content into the browser, by making an asynchronous request (XMLHttpRequest / fetch request) from the browser.



return new PartialViewResult() { ViewName = "partial view name", Model = model };

[or]

return PartialView("partial view name", model);

# **9. View Components [MVC]**

View Components

View Components are self-contained, reusable UI building blocks in ASP.NET Core MVC. They are designed to encapsulate rendering logic that's more complex than what you'd typically put in a partial view but doesn't warrant the complexity of a full controller and view.

**Purpose**

* **Encapsulate Complexity:** Group related UI rendering logic into a cohesive unit.
* **Reusability:** Use View Components across multiple views to avoid code duplication.
* **Testability:** Easier to unit test View Components due to their self-contained nature.
* **Rendering Logic:** Ideal for dynamic widgets, navigation menus, login forms, shopping cart summaries, or any UI element that involves fetching data or performing logic before rendering.

**Best Practices**

* **Naming:** View Component classes should end with the suffix ViewComponent (e.g., ProductListViewComponent).
* **Structure:** Place View Components in the ViewComponents folder. The associated view file should be placed in Views/Shared/Components/{ViewComponent Name}/{View Name}.cshtml.
* **Data:** Pass data to the View Component using a strongly typed model or view model.
* **Asynchronous:** Use asynchronous methods (InvokeAsync) to avoid blocking threads when performing data access or other I/O operations.
* **Simplicity:** Keep the View Component's logic focused on rendering the UI element. Avoid complex business logic within the component.

**Things to Avoid**

* **Overuse:** Don't use View Components for simple UI elements that can be handled by partial views.
* **Tight Coupling:** Avoid tightly coupling View Components to specific controllers or actions. Make them reusable across different parts of your application.
* **Complex Logic:** View Components should not be responsible for handling business logic. Instead, delegate that to your models or services.
* **Direct Database Access:** Avoid directly accessing your database from within a View Component. Use services for data access.

**When to Use View Components**

* **Complex UI Elements:** When a UI element requires more complex rendering logic than a simple partial view.
* **Data-Driven Elements:** When you need to fetch data or perform some computation before rendering the UI element.
* **Reusable Widgets:** When you want to create a reusable widget that can be used in different parts of your application.

**How to Implement View Components**

1. **Create a View Component Class:** Derive from ViewComponent and implement an Invoke or InvokeAsync method.
2. **Create a View:** Create a Razor view file (.cshtml) within the Views/Shared/Components/{ViewComponent Name} folder.
3. **Invoke in Your View:** Use the @await Component.InvokeAsync("ViewComponent Name", arguments) helper in your main view to render the View Component.

Code

1. // GridViewComponent.cs (View Component)
2. public class GridViewComponent : ViewComponent
3. {
4. public async Task<IViewComponentResult> InvokeAsync()
5. {
6. // Data for the view component (could come from a database, service, etc.)
7. PersonGridModel model = new PersonGridModel()
8. {
9. GridTitle = "Persons List",
10. Persons = new List<Person>()
11. {
12. new Person() { PersonName = "John", JobTitle = "Manager" },
13. // more people
14. }
15. };
17. // ViewBag is not ideal; prefer using a strongly typed model
18. ViewData["Grid"] = model;
20. return View("Sample"); // This will look for the view in Views/Shared/Components/Grid/Sample.cshtml
21. }
22. }
23. // Views/Home/Index.cshtml (Main View)
24. @await Component.InvokeAsync("Grid") // Invokes the "Grid" view component
25. // Views/Home/About.cshtml
26. @await Component.InvokeAsync("Grid")
28. <vc:grid></vc:grid> // alternative syntax (not preferred, as it is not strongly typed)

**Explanation:**

* The GridViewComponent class is a view component that fetches a list of people and passes it to the "Sample" partial view along with a grid title using ViewBag.
* Views/Home/Index.cshtml and Views/Home/About.cshtml invoke the Grid view component using @await Component.InvokeAsync("Grid") (or, less ideally, the vc:grid tag helper), rendering the person list in a grid format.

**Notes**

* **Purpose:** Encapsulate complex, reusable UI rendering logic.
* **Naming:** ViewComponent classes end with ViewComponent.
* **Location:** ViewComponents folder for classes, Views/Shared/Components/{ViewComponent Name} for views.
* **Data Passing:** Use strongly typed models or view models (preferable over ViewBag).
* **Rendering:** @await Component.InvokeAsync("ViewComponent Name", arguments)

Strongly Typed View Components

Just like strongly typed views, strongly typed view components are associated with a specific model class using the @model directive. This model class, often referred to as a "view model," provides a structured way to pass data from the view component to its corresponding view. The primary benefit is enhanced type safety and improved code maintainability.

**Benefits of Strongly Typed View Components**

* **Type Safety:** Catches errors during development due to mismatched data types or property names, as the Razor engine enforces type checking based on your model class.
* **IntelliSense:** Enhances productivity by providing code completion and suggestions for model properties directly within the Razor view. This leads to fewer typos and faster development.
* **Refactoring:** Simplifies the process of updating views when your model changes. Since the view is strongly typed, renaming or modifying properties in the model will automatically update the references in the view.
* **Clarity and Readability:** Makes your code more self-documenting by clearly defining the data structure expected by the view component.

**How to Implement Strongly Typed View Components**

1. **Create a View Model:** Define a class that represents the data you want to pass to your view component. This class should contain all the necessary properties that the view component will use to render its output.
2. **Use @model in the View:** In your view component's Razor view file (e.g., Default.cshtml), use the @model directive to specify the view model class.
3. **Return the Model from InvokeAsync:** In your view component's InvokeAsync method, create an instance of your view model, populate it with the required data, and return it using View(model).

Code

**PersonGridModel.cs (View Model)**

1. public class PersonGridModel
2. {
3. public string GridTitle { get; set; }
4. public List<Person> Persons { get; set; }
5. }

This model will be used to represent the data passed from the view component to the view.

**GridViewComponent.cs**

1. // ViewComponent
2. public class GridViewComponent : ViewComponent
3. {
4. public async Task<IViewComponentResult> InvokeAsync()
5. {
6. PersonGridModel personGridModel = new PersonGridModel()
7. {
8. GridTitle = "Persons List",
9. Persons = new List<Person>() {
10. new Person() { PersonName = "John", JobTitle = "Manager" },
11. new Person() { PersonName = "Jones", JobTitle = "Asst. Manager" },
12. new Person() { PersonName = "William", JobTitle = "Clerk" },
13. }
14. };
15. return View("Sample", personGridModel);
16. }
17. }

The method now creates a PersonGridModel object, populates it, and passes it as the second argument of the View() method.

**Views/Shared/Components/Grid/Sample.cshtml**

1. @model PersonGridModel
3. <div class="box">
4. <h3>@Model.GridTitle</h3>
5. <table class="table w-100">
6. <thead>
7. <tr>
8. <th>Sl. No</th>
9. <th>Name</th>
10. </tr>
11. </thead>
12. <tbody>
13. @foreach (Person person in Model.Persons)
14. {
15. <tr>
16. <td>@person.PersonName</td>
17. <td>@person.JobTitle</td>
18. </tr>
19. }
20. </tbody>
21. </table>
22. </div>

* **@model PersonGridModel:** This specifies that the view is strongly typed and expects an object of type PersonGridModel.
* **Accessing Properties:** The view directly accesses properties of the Model object like Model.GridTitle and Model.Persons.

**Notes**

* **Strongly Typed:** Use @model in the view component's view file to specify the view model type.
* **Pass the Model:** Pass an instance of your view model class when returning the view from the InvokeAsync method: return View("ViewName", model);
* **Naming:** View components should be named with the suffix "ViewComponent" (e.g., ProductListViewComponent).
* **Location:** View component classes go in a ViewComponents folder, and their associated views go in Views/Shared/Components/{ViewComponent Name}/.

View Components with Parameters

While view components can function without parameters, passing parameters to them significantly enhances their flexibility and reusability. Parameters allow you to customize the behavior and output of a view component based on data provided by the calling view or controller. This makes view components more dynamic and adaptable to different scenarios within your application.

**How to Pass Parameters to View Components**

1. **Define Parameters in InvokeAsync:** In your view component class, define the parameters that you want to receive within the InvokeAsync (or Invoke) method. These parameters will become part of the method's signature.
2. **Pass Arguments When Invoking:** When invoking the view component using @await Component.InvokeAsync("ViewComponent Name", arguments) or the <vc:grid> tag helper, pass an anonymous object containing the parameter values. The property names in the anonymous object must match the parameter names in the InvokeAsync method.

Code

**GridViewComponent.cs**

1. public class GridViewComponent : ViewComponent
2. {
3. public async Task<IViewComponentResult> InvokeAsync(PersonGridModel grid)
4. {
5. return View("Sample", grid);
6. }
7. }

**Parameter:** The InvokeAsync method now takes a parameter of type PersonGridModel. This means the view component expects to receive a model containing grid data when it's invoked.

**Views/Home/Index.cshtml**

1. @{
2. PersonGridModel personGridModel = new PersonGridModel() { /\* ... (data initialization) ... \*/ };
3. }
5. @await Component.InvokeAsync("Grid", new { grid = personGridModel })
7. @{
8. PersonGridModel personGridModel2 = new PersonGridModel() { /\* ... (different data initialization) ... \*/ };
9. }
10. @await Component.InvokeAsync("Grid", new { grid = personGridModel2 })

Two instances of PersonGridModel are created and passed as an argument to the view component when invoking it.

**Views/Home/About.cshtml**

1. @{
2. PersonGridModel personGridModel = new PersonGridModel() { /\* ... (data initialization) ... \*/ };
3. }
5. <vc:grid grid="personGridModel"></vc:grid>

* **Tag Helper Usage:** The <vc:grid> tag helper is used here to invoke the view component. The grid attribute is set to the personGridModel, passing the model data to the view component.

**Benefits of Strongly Typed View Components with Parameters**

* **Type Safety:** Ensures that you pass the correct data types to the view component, preventing runtime errors due to type mismatches.
* **IntelliSense:** You get code completion suggestions for parameter names and model properties, making development faster and more efficient.
* **Flexibility:** Customize the view component's behavior based on the provided parameters, making it more adaptable to different scenarios.
* **Maintainability:** Easier to refactor and modify the view component and its usage in your views.

**Notes**

* **Define Parameters:** Clearly define the parameters your view component expects in its InvokeAsync method.
* **Pass Parameters:** When invoking the view component, pass an anonymous object with properties matching the parameter names and types.
* **Strongly Typed:** Use a view model (like PersonGridModel) to pass structured data to your view component and benefit from type safety and IntelliSense.

ViewComponentResult

While view components are primarily invoked from within views, the ViewComponentResult class allows you to directly return a rendered view component from a controller action. This provides a powerful way to integrate view components with your MVC controller logic and leverage their reusable rendering capabilities.

**Purpose**

* **Dynamic View Component Loading:** Use ViewComponentResult to load view components on demand based on the controller's logic or data. This is particularly useful for rendering complex UI elements that depend on specific data or conditions.
* **Separation of Concerns:** Keep your controller actions focused on handling requests and data retrieval, while delegating the rendering of specific UI components to view components.
* **API-like Endpoints:** Create API-like endpoints that return HTML fragments (view components) instead of JSON or XML data. This can be helpful for building hybrid applications where you need to combine server-side rendering with client-side JavaScript.

**Creating a ViewComponentResult**

In your controller actions, you can return a ViewComponentResult using the ViewComponent() helper method:

1. return ViewComponent("ViewComponent Name", arguments);

* **ViewComponent Name:** The name of the view component you want to invoke. This should match the class name of your view component (without the "ViewComponent" suffix).
* **arguments (Optional):** An anonymous object containing the parameters you want to pass to the view component's InvokeAsync method.

Code

1. // HomeController.cs (Controller)
2. [Route("friends-list")]
3. public IActionResult LoadFriendsList()
4. {
5. PersonGridModel personGridModel = new PersonGridModel()
6. {
7. GridTitle = "Friends",
8. Persons = new List<Person>()
9. {
10. // ... (list of friends) ...
11. }
12. };
14. return ViewComponent("Grid", new { grid = personGridModel });
15. }

In this code:

1. **Data Preparation:** The personGridModel object is created and populated with data for the list of friends.
2. **ViewComponentResult:** The ViewComponent() method is used to return a ViewComponentResult. It takes two arguments:
   * "Grid": This indicates that we want to invoke the GridViewComponent.
   * new { grid = personGridModel }: This anonymous object passes the personGridModel as the grid parameter to the InvokeAsync method of the GridViewComponent.

**How It Works**

1. **Request:** A request to /friends-list triggers the LoadFriendsList action.
2. **View Component Invocation:** The ViewComponentResult returned from the action causes ASP.NET Core MVC to locate and invoke the GridViewComponent.
3. **View Component Execution:** The InvokeAsync method within the GridViewComponent receives the personGridModel and uses it to render the "Sample" partial view.
4. **Response:** The rendered output of the GridViewComponent (the HTML for the grid of friends) is returned as the HTTP response.

**Notes**

* **Purpose:** Render view components directly from controller actions.
* **Flexibility:** Load view components dynamically based on controller logic.
* **API-Style Endpoints:** Useful for creating endpoints that return HTML fragments.
* **Syntax:** Use ViewComponent("ViewComponent Name", arguments) to create a ViewComponentResult.
* **Parameters:** Pass parameters to the view component using an anonymous object.

Key Points to Remember

**1. View Components: Modular UI Components**

* **Purpose:** Encapsulate reusable UI rendering logic more complex than partial views.
* **Benefits:**
  + Modularization and reusability
  + Separation of concerns (UI logic from controllers)
  + Improved testability
* **Structure:**
  + Class: Inherits from ViewComponent (e.g., ProductListViewComponent).
  + View: Razor file in Views/Shared/Components/{ViewComponent Name}/{View Name}.cshtml.

**2. Methods**

* **Invoke or InvokeAsync:** The main method for your component's logic.
  + Returns IViewComponentResult.
  + Can take parameters for customization.
  + Use async for asynchronous operations.

**3. Invoking View Components**

* **In Views:**
  + @await Component.InvokeAsync("ViewComponent Name", arguments)
  + <vc:view-component-name></vc:view-component-name> (Tag Helper syntax, less preferred)
* **In Controllers:**
  + return ViewComponent("ViewComponent Name", arguments);

**4. Strongly Typed View Components**

* **Purpose:** Pass data to the view using a strongly typed model.
* **Benefits:** Type safety, IntelliSense, better maintainability.
* **How to Use:**
  1. Create a view model class.
  2. Use @model YourViewModel in the view component's view file.
  3. Return the view model from InvokeAsync: return View(model);

**5. Passing Parameters**

* **InvokeAsync Parameters:** Define parameters in the InvokeAsync method signature.
* **Invocation Arguments:** Pass arguments as an anonymous object when invoking the component.

**Example:**

1. // ViewComponent
2. public async Task<IViewComponentResult> InvokeAsync(int categoryId)
3. {
4. var products = \_productService.GetProductsByCategory(categoryId);
5. return View(products);
6. }
8. // View
9. @await Component.InvokeAsync("ProductList", new { categoryId = 5 })

**7. Best Practices**

* **Naming:** Use the suffix "ViewComponent" for class names.
* **Folder Structure:**
  + View components in the ViewComponents folder.
  + Views in Views/Shared/Components/{ViewComponent Name}/.
* **Strongly Typed:** Use strongly typed view models.
* **Async Operations:** Use InvokeAsync for asynchronous tasks.
* **Limit Logic:** Keep logic focused on UI rendering, not business operations.
* **Avoid Direct Database Access:** Use services for data access.

**8. Things to Avoid**

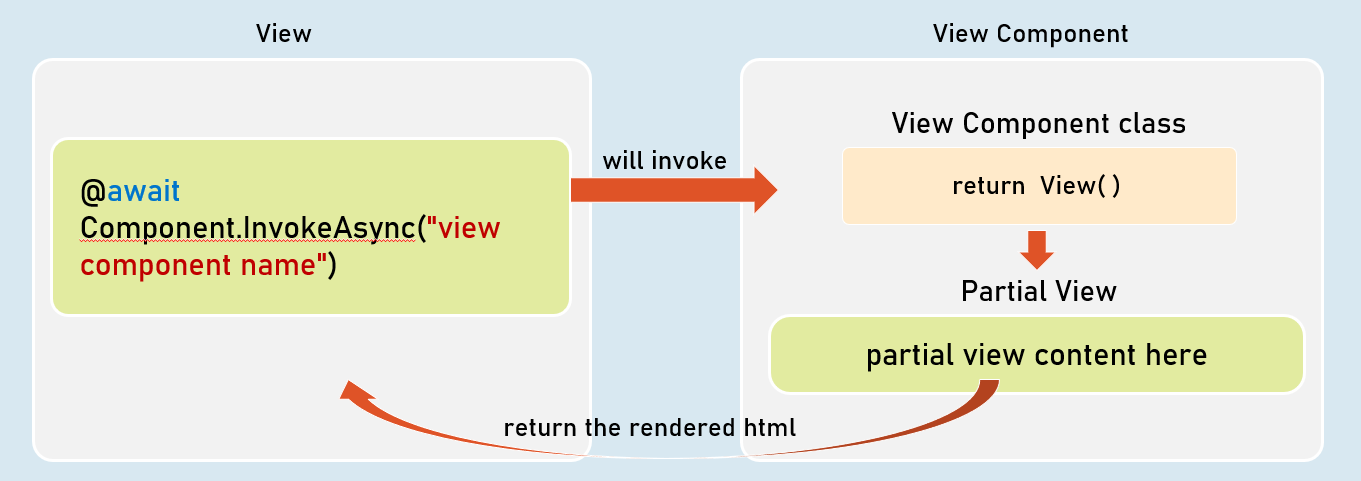
* **Overuse:** Don't use for simple UI elements.
* **Tight Coupling:** Keep them independent of specific controllers.
* **Complex Logic:** Avoid extensive business logic within components.

**9. When to Use**

* **Complex UI Elements:** When a partial view is not enough.
* **Data-Driven Elements:** Dynamic content based on data/logic.
* **Reusable Widgets:** To create reusable UI components.

View Components

View Component is a combination of a class (derived from Microsoft.AspNetCore.ViewComponent) that supplied data, and a partial view to render that data.



**Invoking View Component**

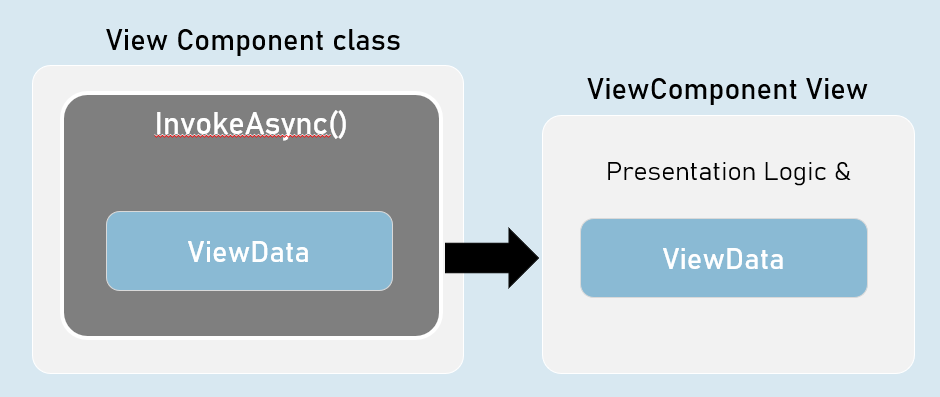
1. @await Component.InvokeAsync("view component name");
2. --or--
3. <vc:view-component-name />

**View Components**

* View component renders a chunk rather than a whole response.
* Includes the same separation-of-concerns and testability benefits found with a controller and view.
* Should be either suffixed with the word "ViewComponent" or should have [ViewComponent] attribute.
* Optionally, it can inherit from System.AspNetCore.Mvc.ViewComponent.

View Components with ViewData

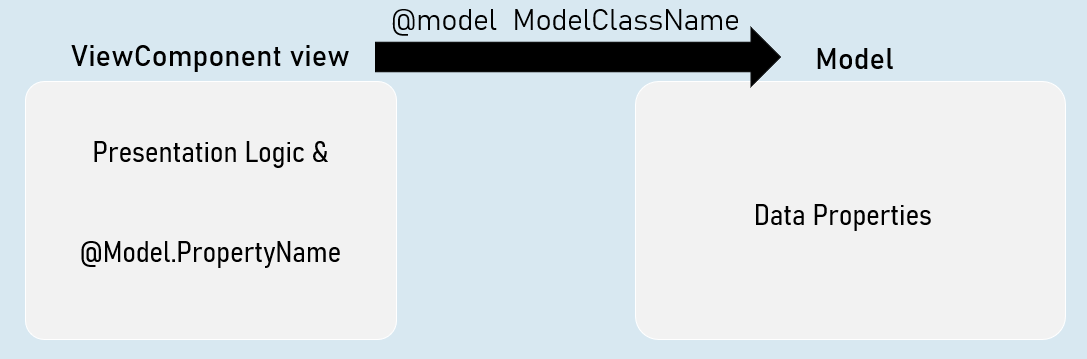
The ViewComponent class can share ViewData object to the ViewComponent view.



Strongly Typed ViewComponent

Strongly Typed ViewComponent's view is tightly bound to a specified model class.

So, it gets all the benefits of a strongly typed view.

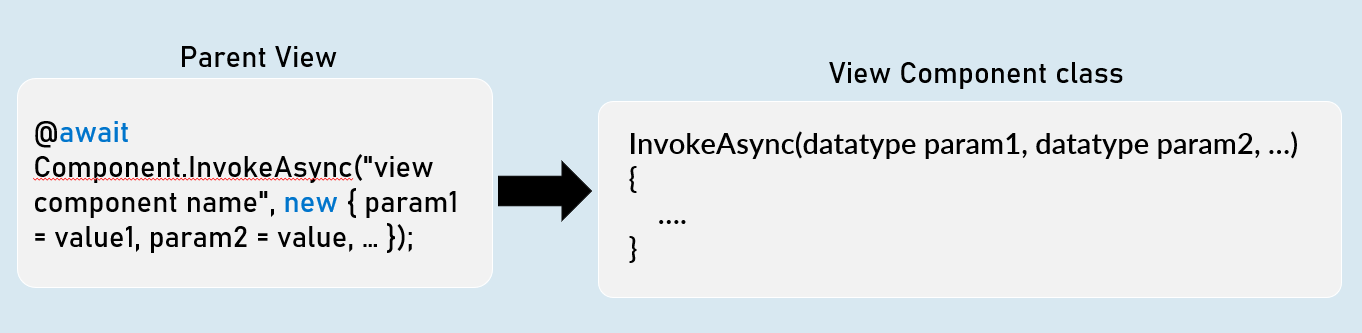


ViewComponents with Parameters

You can supply one or more parameters to the view component class.

The parameters are received by InvokeAsync method of the view component class.

All the parameters of view component are mandatory (must supply a value).



Invoking ViewComponent with parameters

@await Component.InvokeAsync("view component name", new { param = value });

-- or --

<vc:view-component-name param="value" />

ViewComponentResult

ViewComponent can represent the content of a view component .

Generally useful to fetch view component's content into the browser, by making an asynchronous request (XMLHttpRequest / fetch request) from the browser.



1. return new ViewComponentResult() { ViewName = "view component name", Arguments = new { param1 = value, param2 = value } };
2. [or]
3. return ViewComponent("view component name", new { param1 = value, param2 = value } });

# **10. DI [MVC vs API]**

Services

In ASP.NET Core MVC, services are classes responsible for implementing the core business logic of your application. They are designed to be reusable, self-contained, and independent of specific controllers or views. Services are the backbone of your application, handling tasks like data access, calculations, communication with external systems, and any other operations that involve the "how" of your application's functionality.

**Key Purposes of Services**

1. **Encapsulation of Business Logic:** Services provide a clean way to encapsulate complex operations and keep them separate from your presentation layer (controllers and views).
2. **Reusability:** A single service can be used by multiple controllers, promoting DRY (Don't Repeat Yourself) principles and making your code more maintainable.
3. **Testability:** Services can be easily unit tested in isolation, allowing you to verify the correctness of your business logic without the overhead of running the entire application.
4. **Dependency Injection (DI):** Services are typically registered in the DI container, making them easily accessible to controllers and other components within your application.

**Typical Responsibilities of Services**

* **Data Access:** Communicating with databases or other data sources to fetch, insert, update, or delete data.
* **Business Rules:** Implementing the rules that govern how your application behaves (e.g., validation, calculations, transformations).
* **Integration:** Interacting with external systems or APIs.
* **Notifications:** Sending emails, SMS messages, or other notifications.
* **Logging:** Recording events and errors for troubleshooting and analysis.

Code

1. // CitiesService.cs (Service)
2. namespace Services
3. {
4. public class CitiesService
5. {
6. private List<string> \_cities;
8. // Constructor
9. public CitiesService()
10. {
11. \_cities = new List<string>() { "London", "Paris", "New York", "Tokyo", "Rome" };
12. }
14. public List<string> GetCities()
15. {
16. return \_cities;
17. }
18. }
19. }
21. // HomeController.cs (Controller)
22. public class HomeController : Controller
23. {
24. private readonly CitiesService \_citiesService;
26. // Constructor (injecting the service)
27. public HomeController()
28. {
29. \_citiesService = new CitiesService();
30. }
32. [Route("/")]
33. public IActionResult Index()
34. {
35. List<string> cities = \_citiesService.GetCities();
36. return View(cities); // Pass the data to the view
37. }
38. }

Note that in this code example, there is no dependency injection being used. In real-world projects, it's good practice to register your services with ASP.NET Core's built-in dependency injection container and then have them injected into your controllers (or other components) through the constructor.

**Explanation**

1. **CitiesService Class:** This class represents a simple service that holds a list of city names and provides a GetCities method to retrieve them.
2. **HomeController Class:**
   * **Dependency:** It has a dependency on the CitiesService class.
   * **Instantiation:** In this simplified example, the CitiesService object is created directly within the controller's constructor.
   * **Action Method:** The Index action method calls the GetCities method of the \_citiesService to retrieve the list of cities and then passes this data to the view.

**Best Practices**

* **Single Responsibility Principle (SRP):** Design your services to have a single responsibility to keep them focused and maintainable.
* **Dependency Injection:** Use dependency injection to manage service lifetimes and dependencies, making your code loosely coupled and easier to test.
* **Interface-Based Programming:** Define interfaces for your services to create abstraction layers and facilitate testing with mocks.
* **Clear Naming:** Use descriptive names for your services and their methods to make your code self-documenting.
* **Testing:** Write unit tests for your services to ensure that your business logic works correctly in isolation.

**Key Points to Remember**

* **Encapsulation:** Services encapsulate the "how" of your application logic, separating it from the presentation layer.
* **Reusability:** Services can be reused across multiple controllers.
* **Testability:** Services are designed to be easily unit tested.
* **Dependency Injection:** Services are typically managed by the DI container and injected into controllers.

Dependency Inversion Principle (DIP)

DIP is a design principle that promotes loosely coupled software architecture. It states that:

1. **High-level modules should not depend on low-level modules.** Both should depend on abstractions.
2. **Abstractions should not depend on details.** Details should depend on abstractions.

In simpler terms:

* Instead of tightly coupling your classes by having them depend on concrete implementations, they should depend on abstractions (interfaces or abstract classes).
* This allows you to easily swap out implementations without changing the higher-level code.

**Inversion of Control (IoC): Shifting Responsibility**

IoC is a broad principle that involves transferring the control of object creation and management from your application code to a framework or container. Instead of your classes explicitly creating their dependencies, they receive them from an external source. This external source is often a DI container.

**Dependency Injection (DI): The Practical Tool**

DI is a specific implementation of the IoC principle. It involves supplying dependencies to a class from outside the class itself. The most common way to do this in ASP.NET Core is through constructor injection, but there are also other techniques like property injection and method injection.

**Benefits of DIP, IoC, and DI**

* **Loose Coupling:** Reduces the direct dependencies between classes, making them easier to change and test independently.
* **Flexibility:** You can easily swap out different implementations of dependencies without affecting the consuming class.
* **Testability:** Unit testing becomes much easier, as you can provide mock dependencies to isolate the code under test.
* **Maintainability:** Code becomes more modular, easier to understand, and less prone to ripple effects from changes.

Code

1. // ServiceContracts (Interface)
2. namespace ServiceContracts
3. {
4. public interface ICitiesService // Abstraction of CitiesService
5. {
6. List<string> GetCities();
7. }
8. }
10. // Services (Implementation)
11. namespace Services
12. {
13. public class CitiesService : ICitiesService // CitiesService depends on the ICitiesService abstraction
14. {
15. // ... (Implementation of GetCities) ...
16. }
17. }

The interface ICitiesService defines the abstraction for a service that can retrieve a list of cities. The class CitiesService provides the concrete implementation, but it depends on the ICitiesService interface, not on a concrete class.

Code

1. // Program.cs (or Startup.cs)
2. builder.Services.Add(new ServiceDescriptor(
3. typeof(ICitiesService), // Interface to register
4. typeof(CitiesService), // Concrete implementation
5. ServiceLifetime.Transient // Lifetime of the service (more on this later)
6. ));
8. // HomeController.cs (Controller)
9. public class HomeController : Controller
10. {
11. private readonly ICitiesService \_citiesService; // Dependency on the interface
13. // Constructor injection
14. public HomeController(ICitiesService citiesService)
15. {
16. \_citiesService = citiesService;
17. }
19. // ... (Action methods) ...
20. }

In this code:

1. **Service Registration:** The CitiesService is registered in the DI container using the Add method. The ServiceDescriptor specifies:
   * The interface type (ICitiesService) that other components will request.
   * The concrete implementation type (CitiesService) that the container will create.
   * The lifetime of the service (ServiceLifetime.Transient means a new instance is created for each request).
2. **Constructor Injection:** The HomeController constructor has a parameter of type ICitiesService. This means the DI container will automatically provide an instance of CitiesService when the controller is created.

**Notes**

* **Abstractions:** Focus on designing interfaces or abstract classes to represent your dependencies.
* **Loose Coupling:** Your classes should depend on abstractions, not concrete implementations.
* **Dependency Injection:** Use DI containers (like the one built into ASP.NET Core) to manage and resolve dependencies.
* **Service Lifetimes:** Understand the different service lifetimes (Transient, Scoped, Singleton) and choose the right one for each service.

Scenario - A Light Switch and Light Bulb

* **Without DIP/IoC/DI:**
  + Imagine a traditional light switch directly wired to a specific light bulb. If you want to change the light bulb to a different type, you might need to rewire the switch, as it's tightly coupled to the original bulb. This is analogous to tightly coupled code, where classes depend directly on specific implementations of other classes.
* **With DIP/IoC/DI:**
  + Now, imagine a standard electrical outlet and a plug. The outlet represents an interface (an abstraction), while the plug represents a class that implements this interface. You can plug any compatible device (light bulb, fan, etc.) into the outlet, and it will work. This is the essence of DIP - depending on abstractions, not concrete implementations.
  + The "inversion of control" comes in because the outlet (interface) dictates the shape of the plug (implementation), not the other way around.
  + "Dependency injection" happens when you plug a device into the outlet. The outlet doesn't create the device; it simply receives it and allows it to function.

**Visual Representation**

1. Without DIP/IoC/DI:
3. Light Switch -----> Specific Light Bulb
4. (Tightly Coupled)
6. With DIP/IoC/DI:
8. Outlet (Interface) <--- Plug (Implementation)
9. |
10. |
11. Light Bulb, Fan, etc.

**Code Analogy**

1. // Without DIP
2. class LightSwitch
3. {
4. private SpecificLightBulb \_bulb = new SpecificLightBulb();
6. public void TurnOn()
7. {
8. \_bulb.Illuminate();
9. }
10. }
12. // With DIP
13. interface ILight
14. {
15. void Illuminate();
16. }
18. class LightBulb : ILight { /\* ... \*/ }
20. class LightSwitch
21. {
22. private ILight \_light;
24. public LightSwitch(ILight light) // Dependency injection
25. {
26. \_light = light;
27. }
29. public void TurnOn()
30. {
31. \_light.Illuminate();
32. }
33. }

In the DIP version:

* LightSwitch depends on the ILight interface, not a specific bulb.
* The LightBulb class implements ILight.
* The LightSwitch constructor takes an ILight parameter (dependency injection). Now, you can pass in any object that implements ILight (a different type of bulb, a fan, etc.), and the LightSwitch will work with it.

**Key Points**

* **DIP:** Depend on abstractions (interfaces) to make your code more flexible and maintainable.
* **IoC:** Let a framework or container manage object creation and dependencies.
* **DI:** Implement IoC by having dependencies provided to your classes (often through constructors).
* **Benefits:** Achieve loose coupling, flexibility, testability, and maintainability in your code.

Service Lifetimes

When you register a service in the DI container, you specify its lifetime. This determines how the DI container creates and manages instances of that service throughout your application's execution.

**Three Main Lifetime Options**

1. **Transient:**
   * **Creation:** A new instance is created each time the service is requested (injected).
   * **Lifetime:** The instance lives only as long as it's needed to fulfill the current request.
   * **Usage:** Ideal for lightweight, stateless services where each request requires a fresh instance.
   * **Example:** Database context, logger, helper classes.
2. **Scoped:**
   * **Creation:** A single instance is created per HTTP request (or scope) within your application.
   * **Lifetime:** The instance is shared throughout the request and disposed of when the request ends.
   * **Usage:** The most common lifetime for web applications. Ensures consistency within a request while avoiding long-lived objects.
   * **Example:** User-specific data, transaction handling, shopping carts.
3. **Singleton:**
   * **Creation:** A single instance is created for the entire lifetime of your application.
   * **Lifetime:** The instance is shared across all requests and components.
   * **Usage:** Suitable for stateless services, caches, background tasks, or configurations that you want to load once and share globally.
   * **Example:** Application-wide configuration settings, shared caches, singleton design pattern implementations.

**Choosing the Right Lifetime**

The lifetime you choose for a service depends on its purpose and how you intend to use it:

* **State:** If your service holds state that needs to be unique per request, use Scoped. If the state needs to be shared globally, use Singleton. If state is irrelevant, Transient is often sufficient.
* **Resource Usage:** Singleton services consume memory for the entire application lifetime, so use them judiciously.
* **Concurrency:** Be mindful of concurrency issues when using singleton services in multi-threaded environments.

**Registration Examples**

1. // Startup.cs (or Program.cs)
2. builder.Services.AddTransient<ITransientService, TransientService>();
3. builder.Services.AddScoped<IScopedService, ScopedService>();
4. builder.Services.AddSingleton<ISingletonService, SingletonService>();

**Lifetime Best Practices**

* **Prefer Scoped for Web Apps:** In most cases, Scoped is the recommended lifetime for services in web applications.
* **Avoid Captive Dependencies:** Don't inject a shorter-lived service (e.g., Transient) into a longer-lived one (e.g., Singleton). This can lead to unexpected behavior and memory leaks.
* **Consider Thread Safety:** If you use a singleton service, ensure it's thread-safe if it will be accessed concurrently.

Dependency Injection Techniques in ASP.NET Core

ASP.NET Core provides a robust built-in dependency injection (DI) container that allows you to inject services into your application's components (controllers, middleware, view components, etc.). Here are the primary ways you can inject dependencies:

1. **Constructor Injection (Most Common):**
   * **Mechanism:** Dependencies are passed as parameters to the class's constructor.
   * **Benefits:**
     + Easy to understand and use.
     + Encourages loose coupling and testability.
     + Ensures that required dependencies are available before the class is used.
   * **Example:**
   * public class ProductsController : Controller
   * {
   * private readonly IProductService \_productService;
   * public ProductsController(IProductService productService)
   * {
   * \_productService = productService;
   * }
   * }
2. **Property Injection (Less Common):**
   * **Mechanism:** Dependencies are assigned to public properties with a [FromServices] attribute.
   * **Benefits:**
     + Can be useful when you have optional dependencies or want to avoid constructor clutter.
     + Allows for lazy loading of dependencies.
   * **Example:**
   * public class MyMiddleware
   * {
   * [FromServices]
   * public ILogger<MyMiddleware> Logger { get; set; }
   * }
3. **Method Injection (Least Common):**
   * **Mechanism:** Dependencies are passed as parameters to individual methods.
   * **Benefits:**
     + Provides fine-grained control over when dependencies are resolved.
     + Can be useful in cases where you only need a dependency within a specific method.
   * **Example:**
   * public IActionResult Index([FromServices] IUserService userService)
   * {
   * // ... use the userService within this method
   * }
   * Use this injection technique if you require a service in one or few actions.
4. **Action Method Injection:**

* **Mechanism:** Injects services directly into action methods as parameters.
* **Benefits:**
  + Simplifies dependency management within specific actions.
  + Useful for scenarios where a dependency is needed only in a particular action method.
* **Example:**

1. public IActionResult MyAction([FromServices] IMyService service)
2. {
3. // ... use the service within this action
4. }

**Choosing the Right Injection Technique**

* **Constructor Injection:** The recommended and most common approach for mandatory dependencies.
* **Property Injection:** Use for optional dependencies or when constructor injection is cumbersome.
* **Method Injection:** Consider this for dependencies that are only needed within specific methods or for finer control over dependency resolution.
* **Action Method Injection:** Ideal for scenarios where a dependency is required only within a specific action method.

**Key Points to Remember**

* **Loose Coupling:** Regardless of the injection type, the core principle of DI is to achieve loose coupling between components.
* **Dependency Inversion Principle (DIP):** Ensure that your classes depend on abstractions (interfaces) rather than concrete implementations.
* **Dependency Injection Container:** ASP.NET Core's built-in DI container handles the registration and resolution of services.
* **Service Lifetimes:** Understand the different service lifetimes (Transient, Scoped, Singleton) and choose the appropriate one for each dependency.

Best Practices of DI

**1. Constructor Injection as the Default**

* **Why:** Constructor injection is the most straightforward and reliable way to inject dependencies. It ensures that a class has all its required dependencies before it can be used, promoting object validity.
* **How:** Declare all the necessary dependencies as constructor parameters.

1. public class ProductService : IProductService
2. {
3. private readonly IProductRepository \_productRepository;
4. private readonly ILogger<ProductService> \_logger;
6. public ProductService(IProductRepository productRepository, ILogger<ProductService> logger)
7. {
8. \_productRepository = productRepository;
9. \_logger = logger;
10. }
11. }

**2. Use Interfaces for Dependencies**

* **Why:** Interfaces promote loose coupling, enabling you to easily swap implementations during testing or when using different environments.
* **How:** Define interfaces for your services and have your classes depend on the interfaces, not concrete implementations.

C#

1. public interface IProductRepository { /\* ... \*/ }
2. public class ProductRepository : IProductRepository { /\* ... \*/ }

**3. Avoid Service Locator Anti-Pattern**

* **Why:** The Service Locator pattern involves directly accessing the DI container from within your classes (e.g., using IServiceProvider.GetService()). This tightly couples your code to the DI container and makes testing harder.
* **How:** Instead, have the DI container inject dependencies directly into your classes.

**4. Register Dependencies at the Composition Root**

* **Why:** The composition root (typically the Program.cs or Startup.cs file) is where you should configure your DI container. This centralizes dependency registration and makes it easier to manage and understand your application's structure.
* **How:** Use the IServiceCollection.Add\* methods (e.g., AddTransient, AddScoped, AddSingleton) to register your services and their lifetimes.

**5. Choose the Appropriate Service Lifetime**

* **Transient:** A new instance is created each time the service is requested.
* **Scoped:** A single instance is created per request.
* **Singleton:** A single instance is created for the entire application lifetime.
* **How:** Carefully consider the nature of your service (stateful vs. stateless) and its usage patterns to choose the right lifetime.

**6. Avoid Captive Dependencies**

* **Why:** A captive dependency occurs when you inject a shorter-lived service (e.g., Transient) into a longer-lived service (e.g., Singleton). This can lead to unexpected behavior and memory leaks.
* **How:** Ensure that your service lifetimes are compatible and that you don't inadvertently capture a transient instance within a singleton.

**7. Use Decorators to Add Cross-Cutting Concerns**

* **Why:** Decorators wrap existing services and allow you to add additional behavior (e.g., logging, caching) without modifying the original service.
* **How:** Implement the same interface as the service you want to decorate and inject the original service into the decorator.

**8. Leverage Options Pattern for Configuration**

* **Why:** The Options pattern provides a strongly typed way to access configuration settings in your services.
* **How:** Create classes that represent your configuration sections and use the IOptions interface to inject them.

**9. Consider Pure DI for Testability**

* **Why:** Pure DI (avoiding the IServiceProvider altogether) makes your classes more testable, as you can easily provide mock dependencies during unit testing.
* **How:** Design your classes so that all their dependencies are passed through the constructor or other injection points.

**10. Don't Overuse DI**

* **Why:** Dependency injection is a powerful tool, but it should be used judiciously. Overusing it can lead to complex object graphs and make code harder to reason about.
* **How:** Don't inject every single class in your application. Use DI for services and components with clear dependencies and where you need flexibility and testability.

Autofac

While ASP.NET Core has a built-in dependency injection (DI) container, Autofac is a popular third-party IoC container known for its flexibility, advanced features, and customization options. It seamlessly integrates with ASP.NET Core, providing you with more powerful tools to manage your dependencies.

**Key Advantages of Autofac**

* **Flexibility:** Offers a wider range of component lifetime scopes and registration options compared to the built-in container.
* **Customization:** Provides more fine-grained control over how dependencies are resolved and managed.
* **Advanced Features:** Supports features like module-based registration, property injection, assembly scanning, and interception (for cross-cutting concerns).
* **Performance:** Generally considered to have a good performance profile.

**Integrating Autofac with ASP.NET Core**

1. **Install Package:** Add the Autofac.Extensions.DependencyInjection NuGet package to your project.
2. **Configure Container:** In your Program.cs (or Startup.cs in older versions), replace the default service provider factory with Autofac's:
   1. builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());
3. **Register Services:** Use the builder.Host.ConfigureContainer<ContainerBuilder> method to access Autofac's ContainerBuilder and register your services: C#
   1. builder.Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
   2. {
   3. // Your Autofac registration logic here
   4. });

Code

1. // Program.cs
2. // ... other imports ...
4. builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()); // Use Autofac
5. builder.Services.AddControllersWithViews(); // Add MVC services
7. builder.Host.ConfigureContainer<ContainerBuilder>(containerBuilder =>
8. {
9. containerBuilder.RegisterType<CitiesService>().As<ICitiesService>().InstancePerLifetimeScope(); // Register CitiesService as Scoped
10. });
12. var app = builder.Build();
13. // ... the rest of the code ...

In this code:

1. **UseServiceProviderFactory:** This line tells ASP.NET Core to use Autofac as the service provider.
2. **AddControllersWithViews:** This registers the necessary services for MVC (models, views, controllers).
3. **ConfigureContainer:** This lambda expression gives you access to Autofac's ContainerBuilder.
4. **RegisterType<CitiesService>().As<ICitiesService>().InstancePerLifetimeScope();:** This registers the CitiesService class as an implementation of the ICitiesService interface with a scoped lifetime.

**Autofac Registration Methods**

* **RegisterType<T>():** Registers a specific type.
* **As<T>():** Specifies the interface or base type that the registered type should be resolved as.
* **Lifetime Scopes:**
  + InstancePerDependency() (equivalent to Transient)
  + InstancePerLifetimeScope() (equivalent to Scoped)
  + SingleInstance() (equivalent to Singleton)

**Notes**

* **Why Autofac?**
  + More flexibility and control over dependency resolution.
  + Additional features (modules, property injection, etc.).
* **Integration:** Replace the default service provider factory with Autofac's.
* **Registration:** Use Autofac's syntax (RegisterType, As, lifetime scopes) within the ConfigureContainer lambda.
* **Familiar Concepts:** The underlying concepts of DI (abstractions, lifetimes) remain the same, just with different syntax.

Service Scope

In ASP.NET Core DI, a service scope is a logical boundary that defines the lifetime of services registered as *Scoped*. When a scope is created, the DI container instantiates any scoped services that are required within that scope. These scoped service instances are then shared across all components within that scope, ensuring consistency and avoiding unnecessary object creation.

**How Service Scopes Work in ASP.NET Core**

1. **Request Scope (Default):** In ASP.NET Core web applications, the most common scope is the *request scope*. A new scope is automatically created at the beginning of each HTTP request. All scoped services are resolved from this request scope and remain alive throughout the entire request-response cycle. Once the request is processed, the scope is disposed, and all scoped services within it are also disposed.
2. **Explicitly Creating Scopes:** You can also create custom scopes manually. This is useful in scenarios where you need a scoped lifetime for operations that don't directly correspond to an HTTP request (e.g., background tasks, unit testing). You can create a scope using the IServiceProvider.CreateScope() method.

Code

1. using (var scope = provider.CreateScope())
2. {
3. var scopedService = scope.ServiceProvider.GetRequiredService<IScopedService>();
4. // Use the scopedService within this scope
5. }

**Lifetime of Scoped Services**

* **Creation:** A new instance of a scoped service is created the first time it's requested within a scope.
* **Sharing:** Subsequent requests for the same scoped service within the same scope will receive the same instance.
* **Disposal:** When the scope is disposed (e.g., at the end of an HTTP request), all scoped services within that scope are also disposed.

**Benefits of Service Scopes**

* **State Management:** Scoped services are perfect for managing state that needs to persist throughout a request but should not leak across different requests.
* **Efficient Resource Usage:** Scopes ensure that you don't create unnecessary instances of services, leading to better memory management.
* **Consistency:** Scoped services provide a consistent view of data and state within a single request.

**Common Scenarios for Scoped Services**

* **Database Contexts (EF Core):** A new database context instance is usually created per request to ensure data isolation and avoid concurrency issues.
* **User-Specific Data:** Services holding data specific to the current user (e.g., shopping cart) are often scoped to the request.
* **Logging with Context:** If you need to log information with request-specific context, a scoped logger is beneficial.
* **Transactions:** If you need to maintain transactional integrity within a request, you can use a scoped service to manage the transaction.

**Important Considerations**

* **Avoid Captive Dependencies:** Be cautious of injecting a scoped service into a singleton service. This can lead to unexpected behavior and memory leaks because the scoped service will be held alive for the entire application lifetime.
* **Explicit Disposal:** When you create custom scopes, remember to dispose them properly using a using statement or by manually calling Dispose() on the IServiceScope object.

Key Points to Remember

**Dependency Inversion Principle (DIP)**

* **Core Idea:** High-level modules shouldn't depend on low-level modules; both should depend on abstractions (interfaces/abstract classes).
* **Goal:** Loose coupling, flexibility, testability.

**Inversion of Control (IoC)**

* **Core Idea:** Transfer control of object creation and management from your code to a framework or container (e.g., the DI container).
* **Goal:** Decoupling, simplified configuration, improved testability.

**Dependency Injection (DI)**

* **Core Idea:** Dependencies are provided (injected) into a class from an external source (usually a DI container).
* **Types in ASP.NET Core:**
  + **Constructor Injection:** Dependencies are passed as constructor parameters (most common).
  + **Property Injection:** Dependencies are assigned to properties with the [FromServices] attribute.
  + **Method Injection:** Dependencies are passed as method parameters.
* **Benefits:**
  + Loose coupling
  + Flexibility
  + Testability
  + Maintainability

**Service Lifetimes in ASP.NET Core DI**

* **Transient:** A new instance created each time a service is requested.
* **Scoped:** A single instance created per request/scope.
* **Singleton:** A single instance created once and shared throughout the application's lifetime.

**Autofac: A Powerful IoC Container**

* **Purpose:** Alternative to the built-in ASP.NET Core DI container.
* **Benefits:**
  + More flexible component registration and lifetime options
  + Advanced features (modules, property injection, interception)
  + Good performance
* **Integration:**
  + Install Autofac.Extensions.DependencyInjection.
  + Use builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory()) in Program.cs.
  + Register services in builder.Host.ConfigureContainer<ContainerBuilder>.

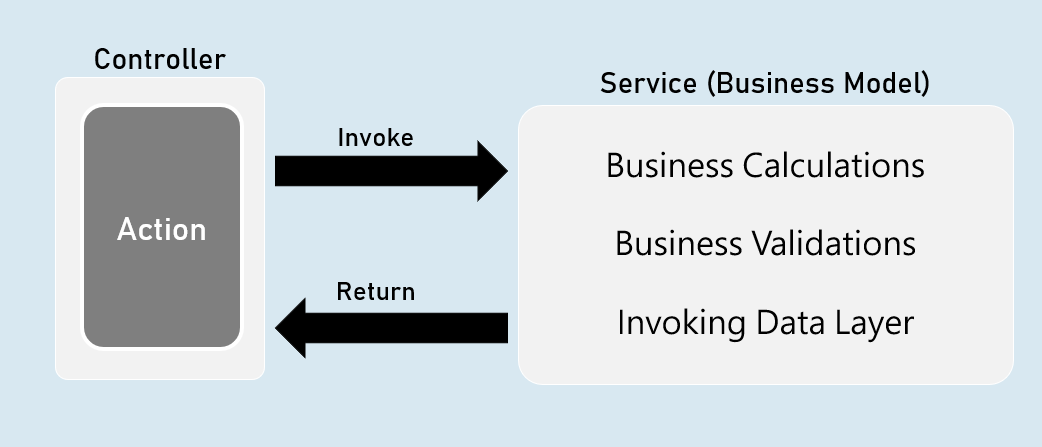
**Autofac Registration**

* containerBuilder.RegisterType<T>().As<TInterface>().InstancePerLifetimeScope();
  + Equivalent to services.AddScoped<TInterface, T>().
* InstancePerDependency() (Transient), SingleInstance() (Singleton)

**Interview Tips**

* **Conceptual Understanding:** Be able to explain the principles of DIP, IoC, and DI and how they relate to each other.
* **Practical Application:** Demonstrate your ability to choose the right lifetime for a given service and explain the implications of each choice.
* **Best Practices:** Discuss the advantages of using interfaces and constructor injection.
* **Autofac:** Highlight the benefits of using Autofac over the built-in container and showcase your knowledge of its registration syntax.
* **Troubleshooting:** Explain how you would diagnose common DI issues (e.g., circular dependencies, incorrect lifetimes).

Services



'Service' is a class that contains business logic such as business calculations, business validations that are specific to the domain of the client's business.

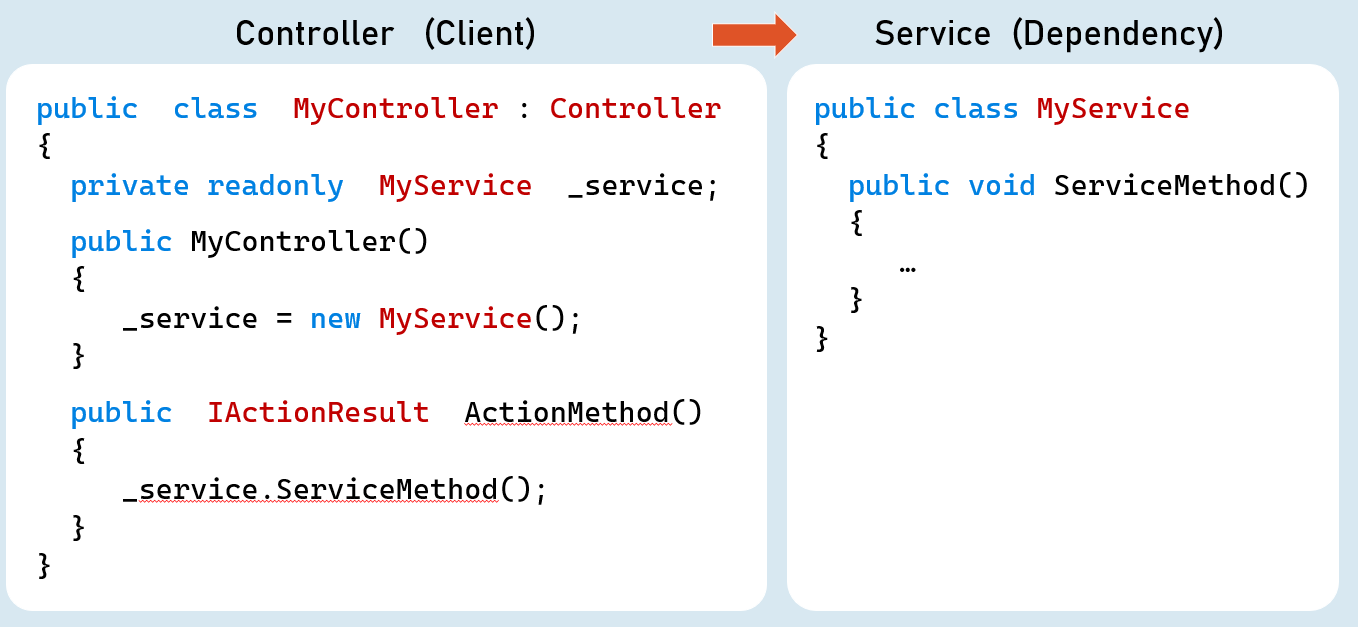
Service is an abstraction layer (middle layer) between presentation layer (or application layer) and data layer.

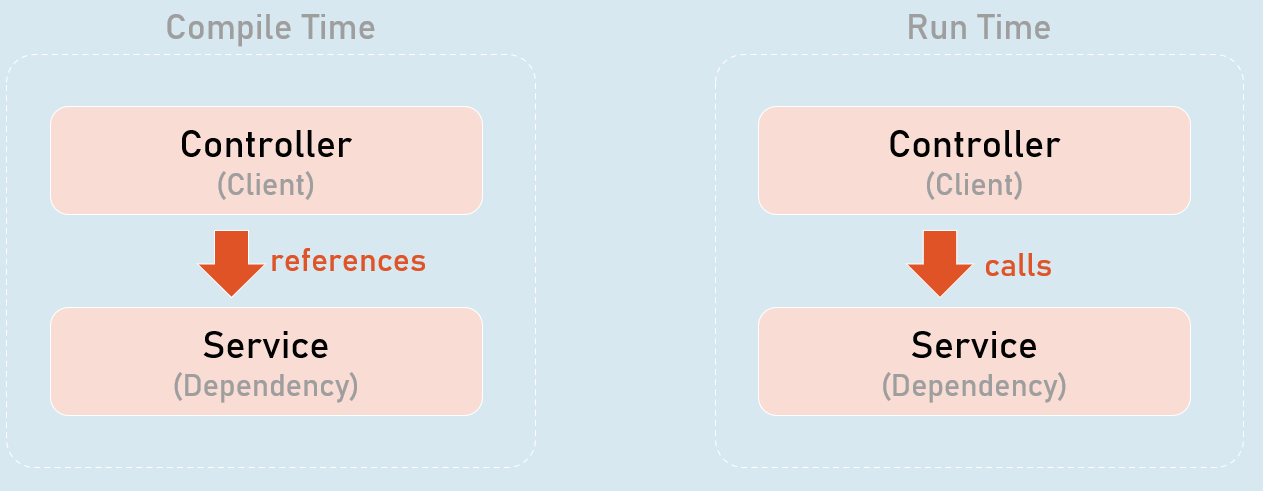
It makes the business logic separated from presentation layer and data layer.

It makes the business logic to be unit testable easily.

Will be invoked by controller.

Direct Dependency





Higher-level modules depend on lower-level modules.

Dependency Problem

Higher-level modules depend on lower-level modules.

* Means, both are tightly-coupled.
* The developer of higher-level module SHOULD WAIT until the completion of development of lower-level module.
* Requires much code changes in to interchange an alternative lower-level module.
* Any changes made in the lower-level module effects changes in the higher-level module.
* Difficult to test a single module without effecting / testing the other module.

Dependency Inversion Principle

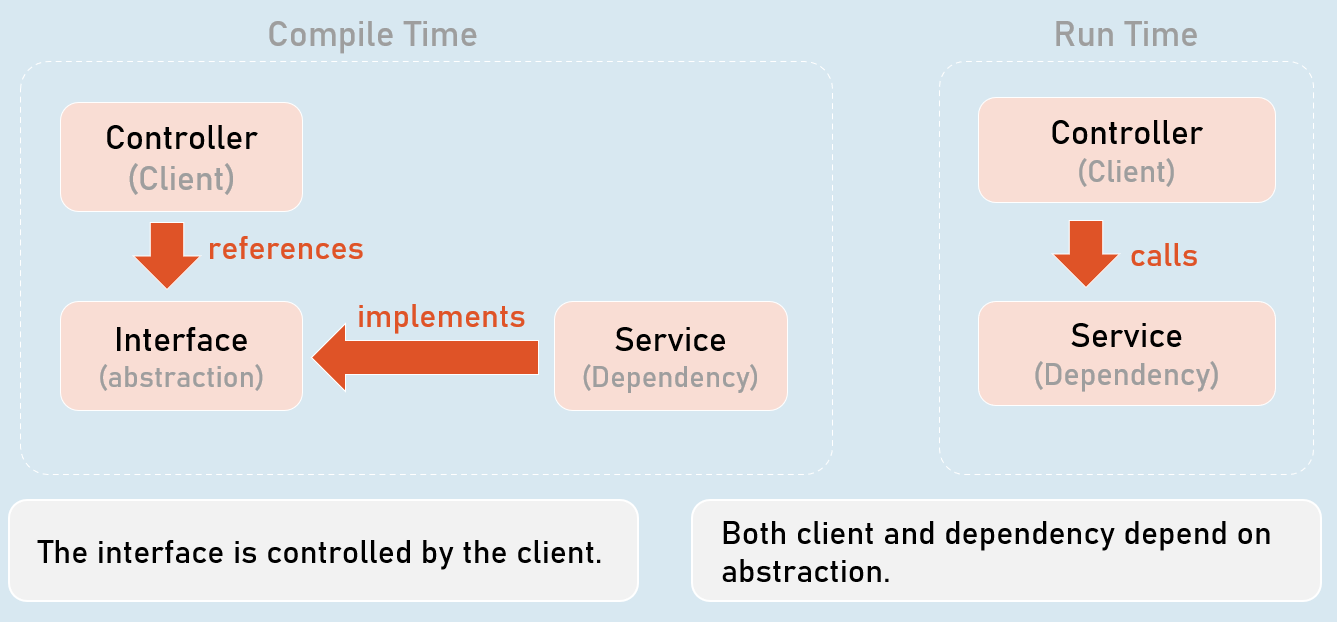
Dependency Inversion Principle (DIP) is a design principle (guideline), which is a solution for the dependency problem.

"The higher-level modules (clients) SHOULD NOT depend on low-level modules (dependencies).

Both should depend on abstractions (interfaces or abstract class)."

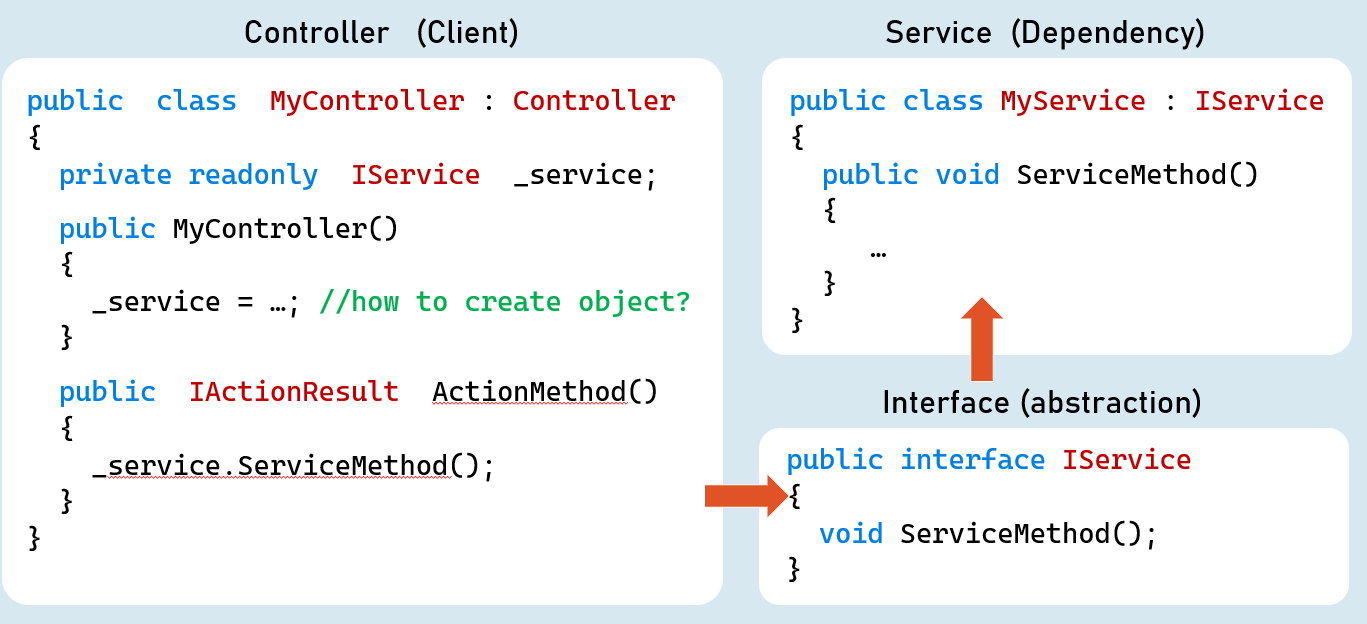
"Abstractions should not depend on details (both client and dependency).

Details (both client and dependency) should depend on abstractions."



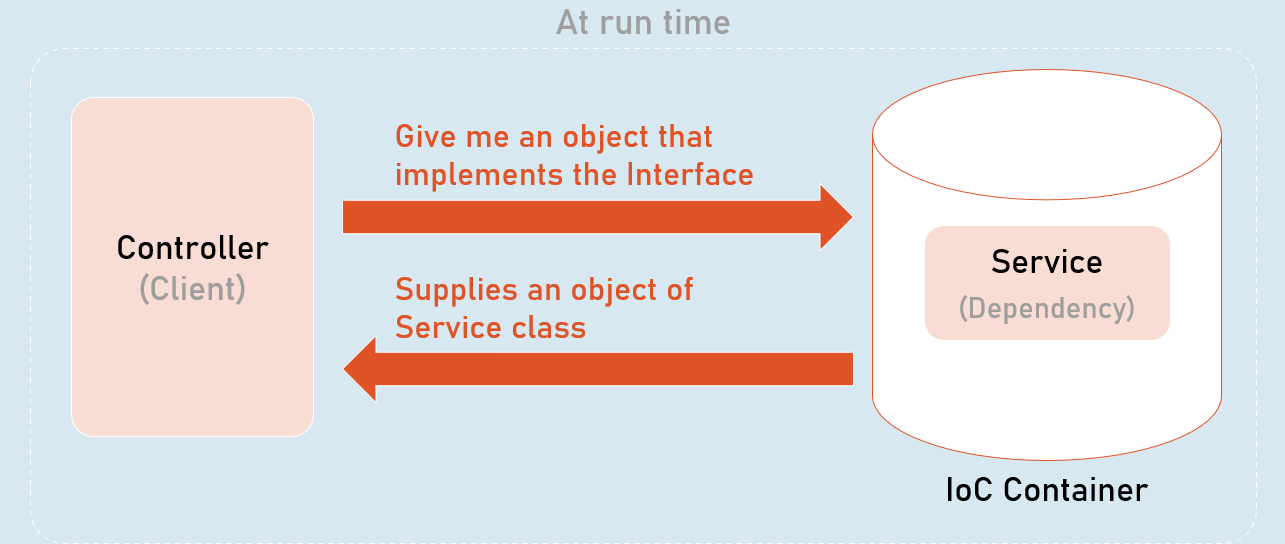
The interface is controlled by the client.

Both client and dependency depend on abstraction.



Inversion of Control (IoC)

* Inversion of Control (IoC) is a design pattern (reusable solution for a common problem), which suggests "IoC container" for implementation of Dependency Inversion Principle (DIP).
* It inverses the control by shifting the control to IoC container.
* "Don't call us, we will call you" pattern.
* It can be implemented by other design patterns such as events, service locator, dependency injection etc.

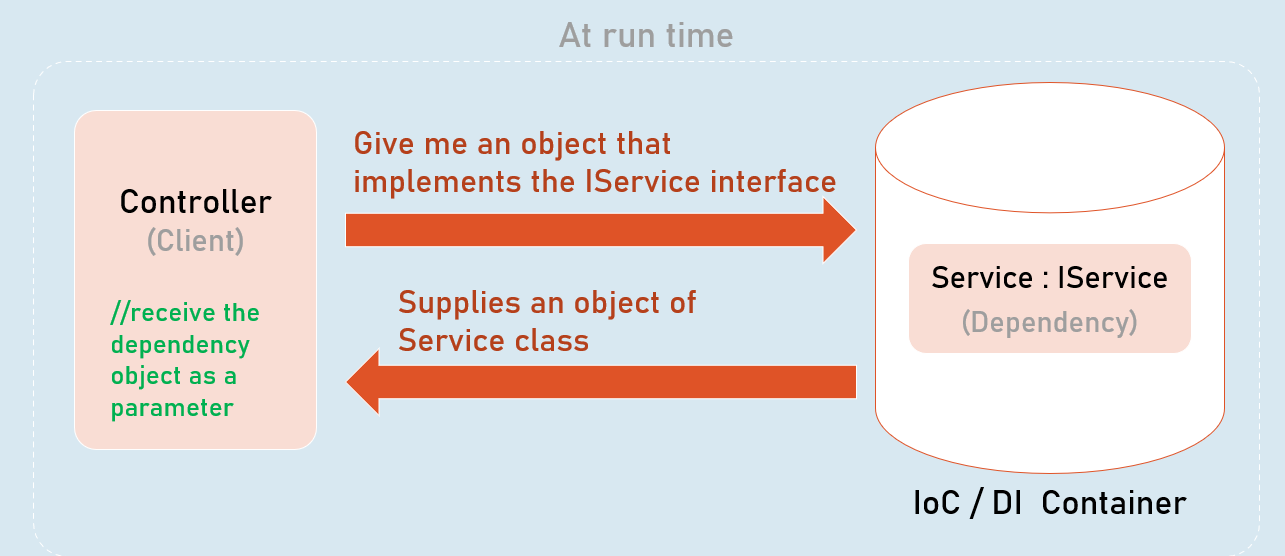


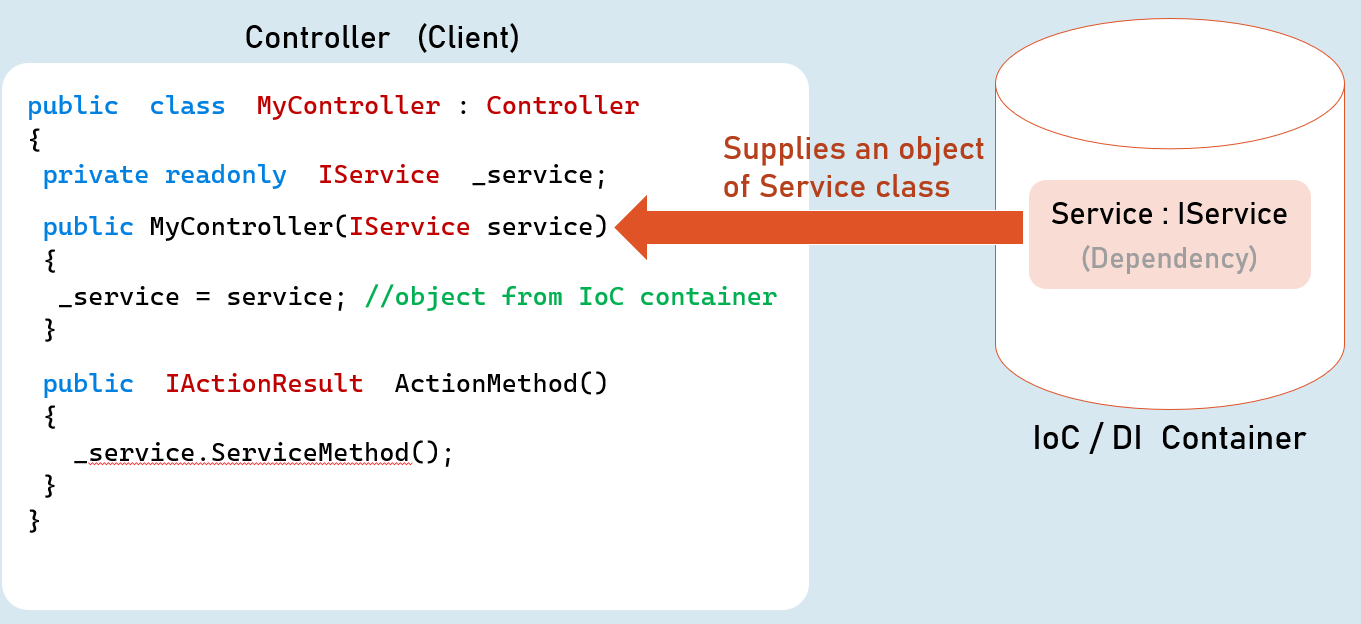
All dependencies should be added into the IServiceCollection (acts as IoC container).

1. builder.Services.Add(
2. new ServiceDescriptor(
3. typeof (Interface),
4. typeof (Service)
5. ServiceLifetime.LifeTime //Transient, Scoped, Singleton
6. )
7. );

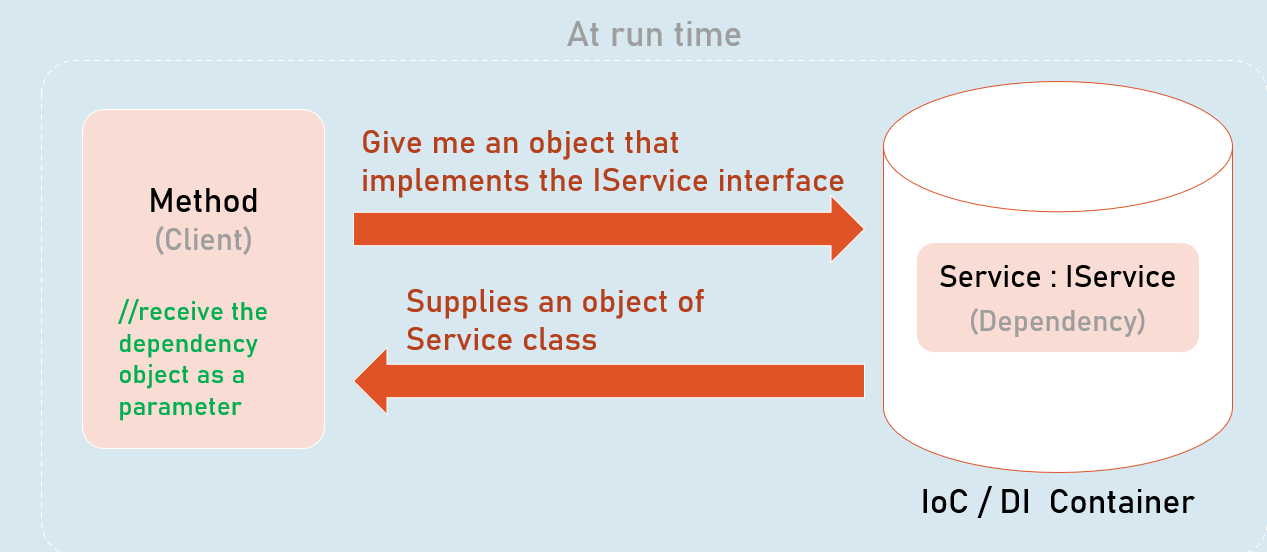
Dependency Injection (DI)

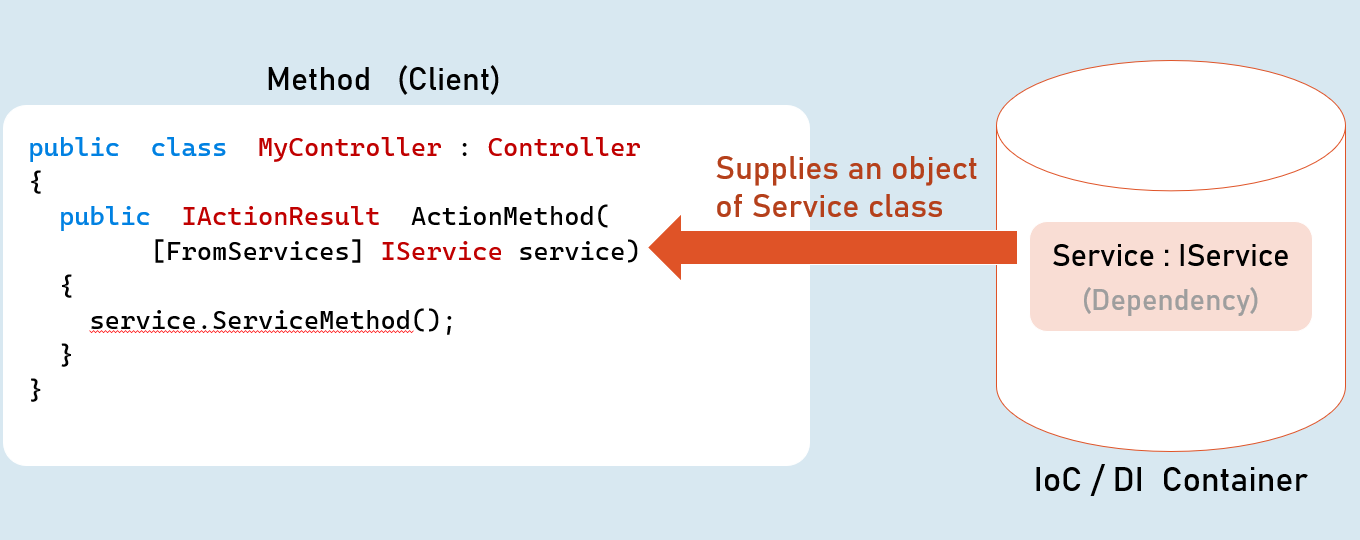
* Dependency injection (DI) is a design pattern, which is a technique for achieving "Inversion of Control (IoC)" between clients and their dependencies.
* It allows you to inject (supply) a concrete implementation object of a low-level component into a high-level component.
* The client class receives the dependency object as a parameter either in the constructor or in a method.





Method Injection



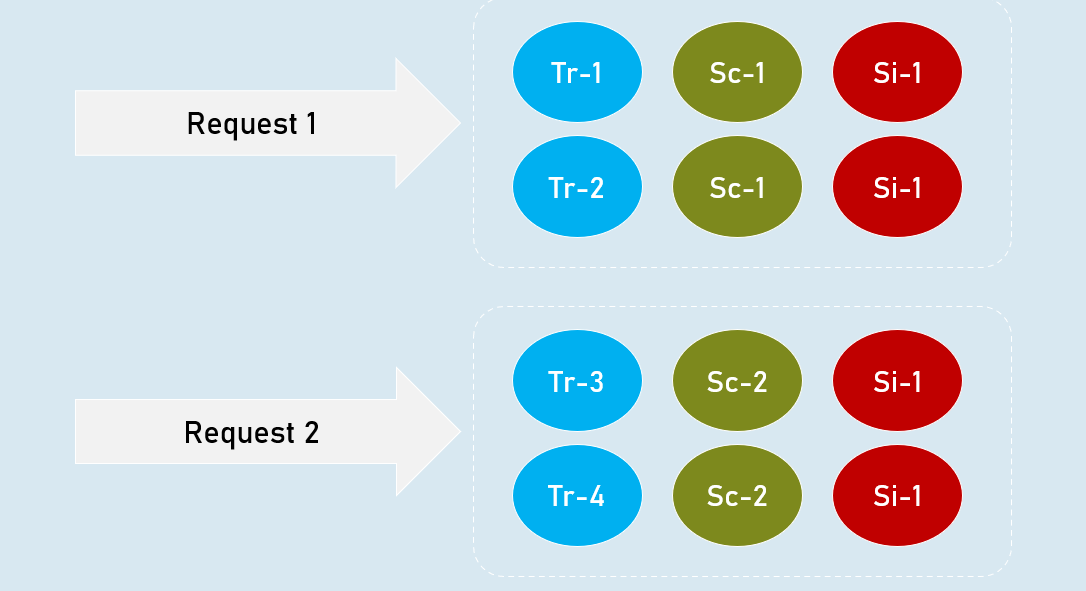


Service Lifetime

(Transient, Scoped, Singleton)

A service lifetime indicates when a new object of the service has to be created by the IoC / DI container.

1. **Transient:**Per injection
2. **Scoped:**Per scope (browser request)
3. **Singleton:**For entire application lifetime.



**Transient**

Transient lifetime service objects are created each time when they are injected.

Service instances are disposed at the end of the scope (usually, a browser request)

**Scoped**

Scoped lifetime service objects are created once per a scope (usually, a browser request).

Service instances are disposed at the end of the scope (usually, a browser request).

**Singleton**

Singleton lifetime service objects are created for the first time when the are requested.

Service instances are disposed at application shutdown.

**Transient**

builder.Services.AddTransient<IService, Service>(); //Transient Service

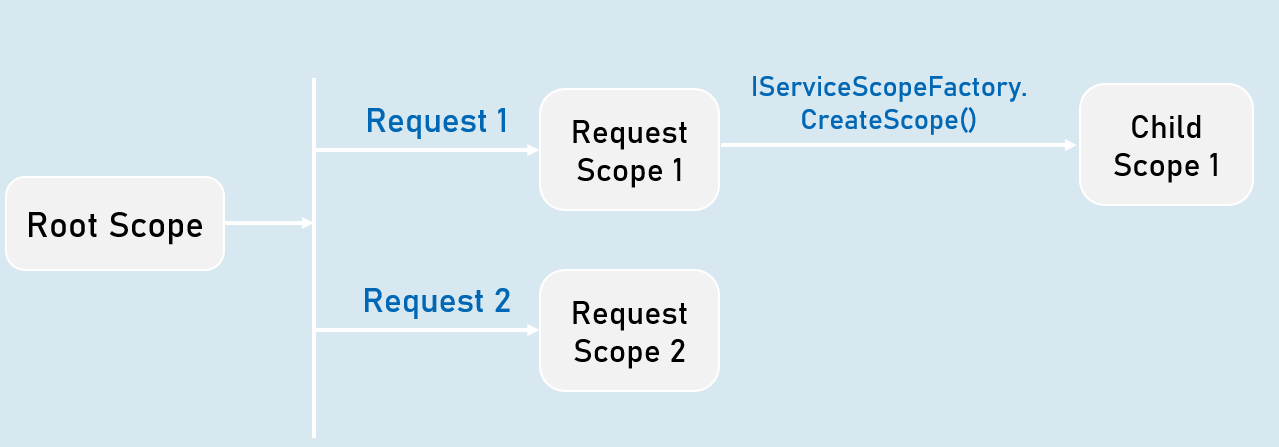
**Scoped**

builder.Services.AddScoped<IService, Service>(); //Scoped Service

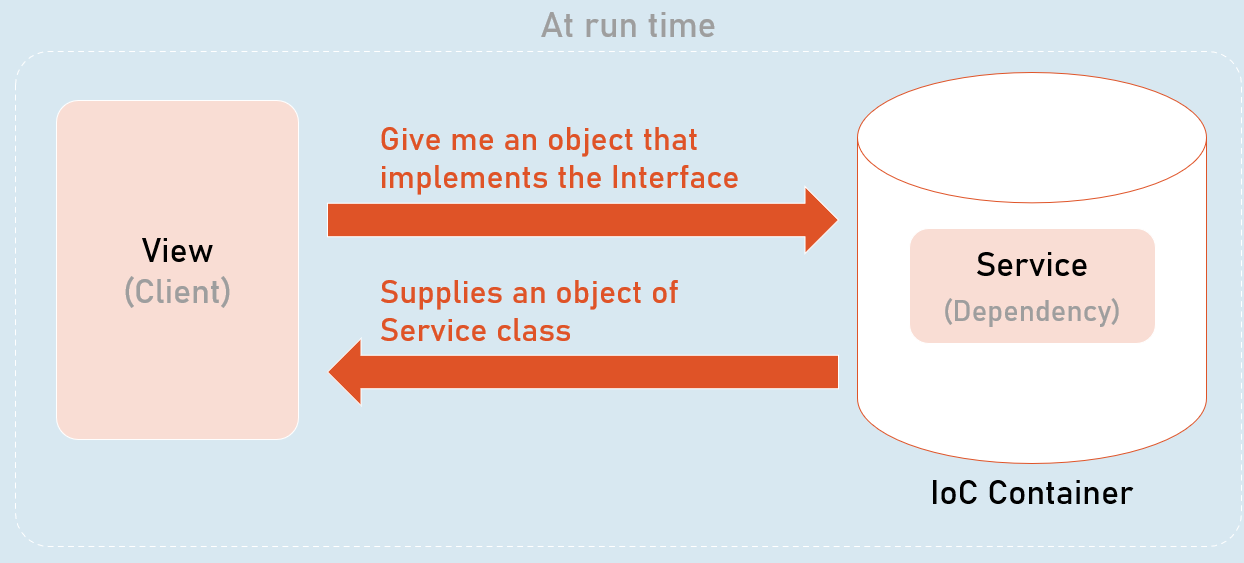
**Singleton**

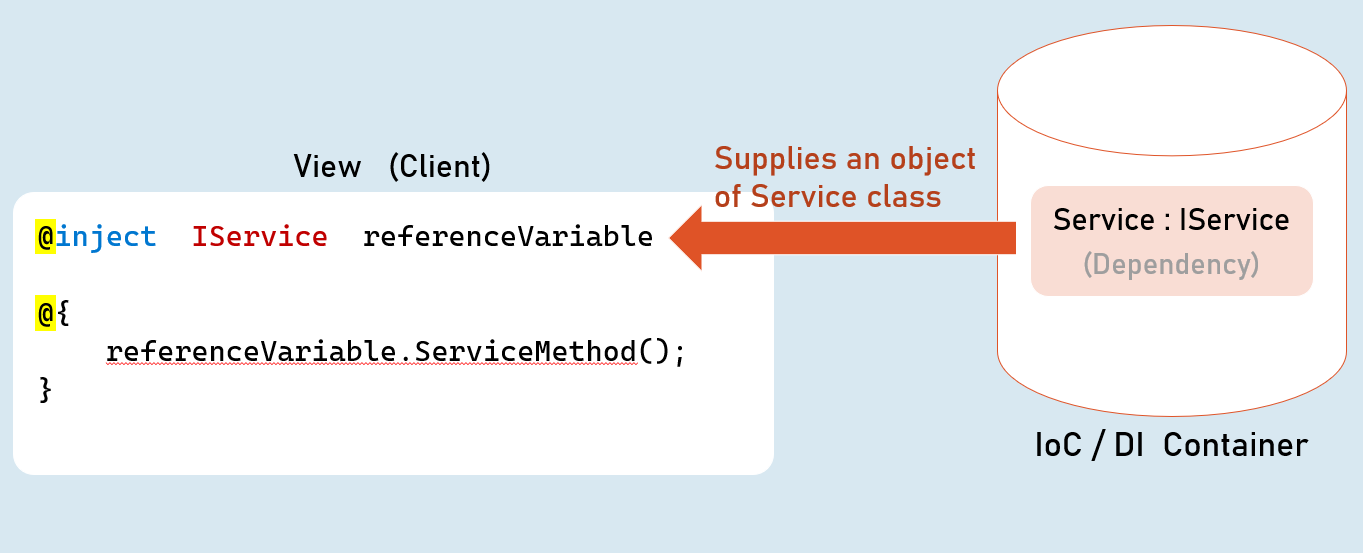
builder.Services.AddSingleton<IService, Service>(); //Singleton Service

Service Scope



View Injection





Best Practices in DI

**Global state in services**

Avoid using **static classes**to store some data globally for all users / all requests.

You may use **Singleton** services for simple scenarios / simple amount of data. In this case, prefer ConcurrentDictionary instead of Dictionary, which better handles concurrent access via multiple threads.

Alternatively, prefer to use **Distributed Cache / Redis** for any significant amount of data or complex scenarios.

**Request state in services**

Don't use scoped services to share data among services within the same request, because they are NOT thread-safe.

Use **HttpContext.Items**instead.

**Service Locator Pattern**

Avoid using service locator pattern, without creating a child scope, because it will be harder to know about dependencies of a class.

For example, don't invoke **GetService()** in the default scope that is created when a new request is received.

But you can use the **IServiceScopeFactory.ServiceProvider. GetService()** within a child scope.

**Calling Dispose() method**

Don't invoke the Dispose() method manually for the services injected via DI.

The IoC container automatically invoke Dispose(), at the end of its scope.

**Captive Dependencies**

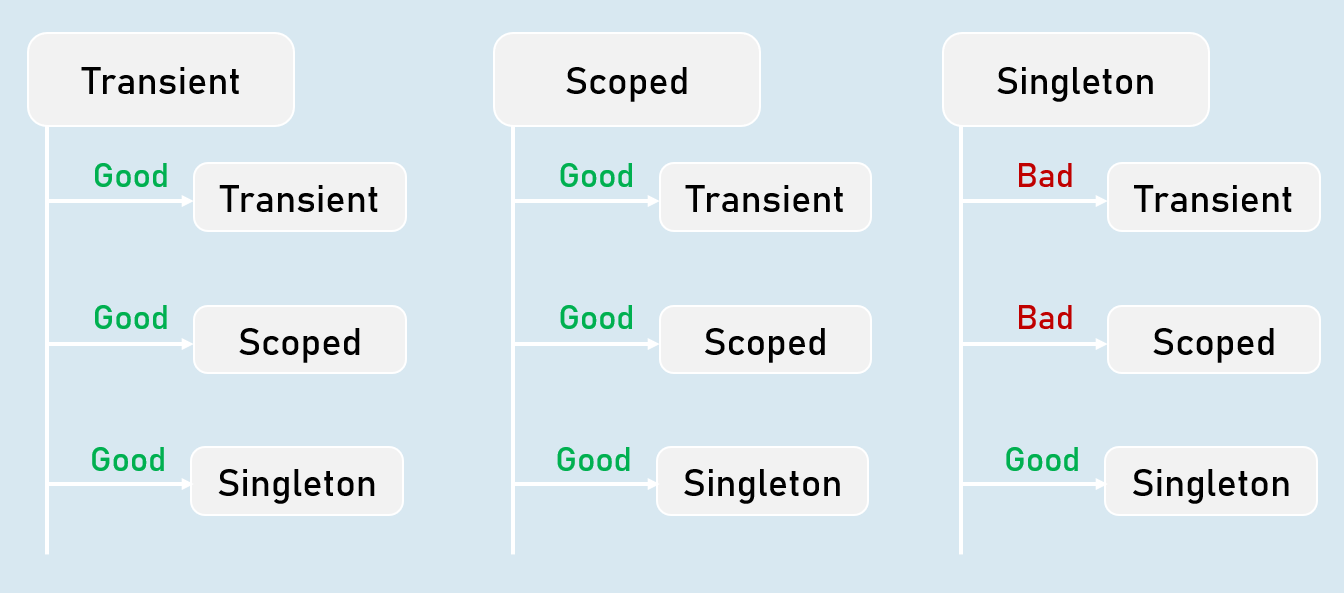
Don't inject scoped or transient services in singleton services.

Because, in this case, transient or scoped services act as singleton services, inside of singleton service.

**Storing reference of service instance**

Don't hold the reference of a resolved service object.

It may cause memory leaks and you may have access to a disposed service object.



Autofac

* Autofac is another IoC container library for .Net Core.
* Means, both are tightly-coupled.
* Microsoft.Extensions.DependencyInjection [vs] Autofac
* https://autofac.readthedocs.io/en/latest/getting-started/index.html

**Microsoft.Extensions.DependencyInjection**

* Built-in IoC container in asp.net core
* Lifetimes: Transient, Scoped, Singleton
* Metadata for services: Not supported
* Decorators: Not supported

**Autofac**

* Alternative to the Microsoft.Extensions
* Lifetimes: InstancePerDependency, InstancePerLifetimeScope, SingleInstance, InstancePerOwned, InstancePerMatchingLifetimeScope
* Metadata for services: Supported
* Decorators: Supported

# **11. Environments [MVC vs API]**

ASP.NET Core Environments

In ASP.NET Core, environments are named configurations that allow you to tailor your application's behavior to different deployment scenarios. This helps you manage settings, configurations, and middleware pipelines that are specific to development, testing, staging, or production environments.

**Common Environments**

* **Development:** Your local development environment. It's where you build and test your application.
* **Staging:** A pre-production environment that closely mirrors your production setup. You use it for final testing and validation.
* **Production:** Your live environment where users interact with your application.

**Setting the Environment**

ASP.NET Core reads the environment from the ASPNETCORE\_ENVIRONMENT environment variable when your application starts. The value of this variable determines the active environment.

**How to Set the Environment**

* **launchSettings.json:** For Visual Studio, you can set the ASPNETCORE\_ENVIRONMENT variable in the launchSettings.json file within your project's Properties folder.
* **Environment Variables:** Set the ASPNETCORE\_ENVIRONMENT variable directly in your system's environment variables.
* **Command Line:** When running your application from the command line, you can set the environment variable using the --environment or -e flag:Bash
  1. dotnet run --environment Staging

**Using Environments in Program.cs**

1. **Retrieving the Environment:**
   1. var builder = WebApplication.CreateBuilder(args);
   2. var environment = builder.Environment;

The environment object gives you access to the current environment's name and other properties.

1. **Conditional Configuration:** You can use conditional logic based on the environment name to configure different settings or middleware.

C#

* 1. if (app.Environment.IsDevelopment())
  2. {
  3. app.UseDeveloperExceptionPage(); // Use a detailed error page in development
  4. }
  5. else
  6. {
  7. app.UseExceptionHandler("/Error"); // Use a generic error page in production
  8. }

1. **Environment-Specific Configuration Files:**
   1. You can create environment-specific configuration files like appsettings.Development.json, appsettings.Staging.json, and appsettings.Production.json.
   2. ASP.NET Core automatically loads the appropriate configuration file based on the current environment.
   3. Use these files to store settings that vary between environments, such as database connection strings or API keys.
   4. These files override the settings in the appsettings.json.

**Best Practices**

* **Environment-Specific Configuration:** Separate your configuration into environment-specific files to avoid exposing sensitive data (like production database credentials) in your development environment.
* **Middleware Pipelines:** Tailor your middleware pipelines for each environment. For example, use UseDeveloperExceptionPage in development but UseExceptionHandler in production.
* **Logging:** Configure different logging levels and targets for different environments (e.g., more verbose logging in development).
* **Feature Flags:** Use environment variables or configuration values to toggle features on or off depending on the environment.

**Example (Program.cs)**

1. var builder = WebApplication.CreateBuilder(args);
2. var app = builder.Build();
4. if (app.Environment.IsDevelopment())
5. {
6. // Development-specific configuration
7. }
8. else if (app.Environment.IsStaging())
9. {
10. // Staging-specific configuration
11. }
12. else // Production
13. {
14. // Production-specific configuration
15. }
17. // ... Rest of your application setup ...

**Notes**

* **Flexibility:** Environments allow you to easily adapt your application to different scenarios.
* **Configuration:** Use environment-specific configuration files (appsettings.{Environment}.json) for organization.
* **Middleware:** Customize middleware pipelines based on the environment.
* **Best Practices:** Follow the best practices mentioned above to ensure a smooth deployment process and optimal behavior in each environment.

Understanding launchSettings.json

This file is primarily used by Visual Studio to configure how your ASP.NET Core application launches during development. It contains settings for different profiles (e.g., IIS Express, ProjectName) and provides a convenient way to set environment variables without modifying your system's global environment variables.

**Location**

You'll find launchSettings.json in the Properties folder within your project's root directory.

**Structure**

1. {
2. "iisSettings": { ... }, // Settings for IIS Express (if used)
3. "profiles": {
4. "IIS Express": { ... }, // Configuration for IIS Express profile
5. "YourProjectName": { // Configuration for running the project directly
6. "commandName": "Project",
7. "dotnetRunMessages": "true",
8. "launchBrowser": true,
9. "applicationUrl": "https://localhost:7272;http://localhost:5248", // URLs to launch
10. "environmentVariables": {
11. "ASPNETCORE\_ENVIRONMENT": "Development" // Setting the environment
12. }
13. }
14. }
15. }

**Setting the ASPNETCORE\_ENVIRONMENT Variable**

Within the environmentVariables section of the desired profile (e.g., "YourProjectName"), you can set the ASPNETCORE\_ENVIRONMENT variable to one of the standard values:

* **Development:** For local development and debugging.
* **Staging:** For pre-production testing.
* **Production:** For the live environment.

You can also use a custom environment name if needed.

**Example: Setting the Development Environment**

1. "environmentVariables": {
2. "ASPNETCORE\_ENVIRONMENT": "Development"
3. }

**How It Works**

When you launch your application from Visual Studio using a specific profile, the environmentVariables settings are applied to the running process. This ensures that your application reads the correct value for ASPNETCORE\_ENVIRONMENT, which in turn influences which configuration settings are loaded (from appsettings.json, appsettings.Development.json, etc.) and which middleware pipelines are used.

**Important Considerations**

* **Environment-Specific Configuration Files:** Remember that you'll still need to create environment-specific configuration files (e.g., appsettings.Development.json) to store settings that vary between environments. launchSettings.json only sets the environment variable.
* **Local Development:** The launchSettings.json file is primarily for local development with Visual Studio. When you deploy your application to a server, you'll typically set the ASPNETCORE\_ENVIRONMENT variable through the hosting environment's configuration (e.g., in the web server's configuration file or environment variables).
* **Multiple Profiles:** launchSettings.json can contain multiple profiles, each with its own set of environment variables. This allows you to easily switch between different configurations during development.

Developer Exception Page

The Developer Exception Page is a powerful tool in ASP.NET Core for diagnosing exceptions during development. It provides a detailed view of the exception, including:

* Stack trace
* Request details (headers, query string, cookies)
* Routing information
* Configuration settings

This information is invaluable for identifying and fixing issues quickly.

**Environment-Specific Behavior**

* **Development:** The Developer Exception Page is enabled by default in the Development environment. This makes sense because during development, you want as much information as possible to help you troubleshoot.
* **Production and Other Environments:** In production or other non-development environments, this page is typically disabled due to security concerns. Exposing detailed exception information to the public could reveal vulnerabilities or sensitive details about your application's internal workings.

**IWebHostEnvironment Interface: Accessing Environment Information**

The IWebHostEnvironment interface gives you access to information about the hosting environment of your ASP.NET Core application. It includes properties like:

* **EnvironmentName:** The name of the current environment (Development, Staging, Production, or a custom name).
* **WebRootPath:** The path to the application's web root directory.
* **ContentRootPath:** The path to the application's content root directory.

**Using IWebHostEnvironment and app.Environment**

1. // HomeController.cs
2. public class HomeController : Controller
3. {
4. private readonly IWebHostEnvironment \_webHostEnvironment; // Injected
6. public HomeController(IWebHostEnvironment webHostEnvironment)
7. {
8. \_webHostEnvironment = webHostEnvironment;
9. }
11. [Route("/")]
12. public IActionResult Index()
13. {
14. ViewBag.CurrentEnviornment = \_webHostEnvironment.EnvironmentName;
15. return View();
16. }
17. }

In this code, the IWebHostEnvironment is injected into the HomeController. The current environment name (\_webHostEnvironment.EnvironmentName) is then assigned to ViewBag.CurrentEnvironment and sent to the view to display.

**Enabling the Developer Exception Page in Specific Environments**

1. // Program.cs
2. if (app.Environment.IsDevelopment() || app.Environment.IsStaging() || app.Environment.IsEnvironment("Beta"))
3. {
4. app.UseDeveloperExceptionPage();
5. }

In this code snippet, the Developer Exception Page is only enabled if the environment is Development, Staging, or a custom environment named "Beta". You can use app.Environment.IsDevelopment() etc. because they are just shorthands for app.Environment.EnvironmentName == "Development". This can be helpful in scenarios where you want to include staging in your development processes.

**Notes**

* **Purpose:** The Developer Exception Page provides detailed error information during development.
* **Environments:** It's enabled by default in Development, but you can customize its behavior based on other environments.
* **IWebHostEnvironment:** Use this interface to access environment information within your controllers or middleware.
* **Security:** Always disable the Developer Exception Page in production to avoid exposing sensitive details.
* **Custom Error Pages:** For production, use app.UseExceptionHandler to create custom error pages that provide a user-friendly message without revealing internal information.

<environment> Tag Helper

The <environment> tag helper is a versatile tool that allows you to include or exclude specific content in your views depending on the current environment your ASP.NET Core application is running in.

This is particularly useful for scenarios where you want to:

* **Include Development Resources:** Load unminified CSS or JavaScript files during development to facilitate debugging and testing.
* **Optimize for Production:** Load minified and bundled assets in production to improve performance.
* **Display Environment-Specific Content:** Show different messages, warnings, or features based on the environment.

**Syntax**

1. <environment include="Environment1,Environment2,...">
2. Content to render if the environment matches any of the included environments
3. </environment>
5. <environment exclude="Environment1,Environment2,...">
6. Content to render if the environment does NOT match any of the excluded environments
7. </environment>

* **include:** A comma-separated list of environment names for which the content should be rendered.
* **exclude:** A comma-separated list of environment names for which the content should **not** be rendered.

**Environment Names**

* **Standard:** Development, Staging, Production.
* **Custom:** You can also define and use your own custom environment names.

**How It Works**

1. **Environment Check:** The <environment> tag helper reads the value of the ASPNETCORE\_ENVIRONMENT environment variable to determine the current environment.
2. **Conditional Rendering:** Based on the include or exclude attributes and the current environment, it either renders or skips the content within the tag helper.

**Code Examples**

1. <environment include="Development">
2. <link rel="stylesheet" href="~/css/site.css" />
3. </environment>
4. <environment exclude="Development">
5. <link rel="stylesheet" href="~/css/site.min.css" />
6. </environment>

In this example:

* The unminified site.css file is loaded only in the Development environment.
* The minified site.min.css file is loaded in all other environments.

**Notes**

* **Flexibility:** Easily adapt your views to different environments without complex conditional logic.
* **Performance Optimization:** Serve optimized assets in production while retaining flexibility in development.
* **Environment-Specific Content:** Display warnings, messages, or debug tools only when needed.

**Best Practices**

* **Use for Static Assets:** Primarily leverage the <environment> tag helper for including or excluding static files (CSS, JavaScript) based on the environment.
* **Avoid Complex Logic:** Keep the content within <environment> tags relatively simple. If you need more complex logic, consider using a partial view or a view component.
* **Custom Environments:** If you need more than the standard environments, define and use your own.

Set the Environment from the Terminal

* **Flexibility:** Setting the environment variable directly from the terminal allows you to easily switch between different environments (development, staging, production) without modifying configuration files or IDE settings.
* **Automation:** This approach is easily scriptable, enabling you to automate deployment processes and seamlessly change configurations for different environments.
* **Non-Windows Environments:** If you're working on macOS or Linux, the terminal is the primary way to manage environment variables.

**Setting ASPNETCORE\_ENVIRONMENT in PowerShell**

* **Windows, macOS, Linux:**

1. $env:ASPNETCORE\_ENVIRONMENT = "Development" # Set to Development
2. $env:ASPNETCORE\_ENVIRONMENT = "Production" # Set to Production

* **Scope:** In PowerShell, environment variables set with $env: are typically limited to the current session. To make them persistent, you need to modify the system or user environment variables (see below).

**Setting ASPNETCORE\_ENVIRONMENT in Command Prompt**

* **Windows:**

1. set ASPNETCORE\_ENVIRONMENT=Development # Set to Development
2. set ASPNETCORE\_ENVIRONMENT=Production # Set to Production

**Scope:** By default, variables set with the set command are temporary and only apply to the current command prompt session. To make them persistent, use the /M switch:

1. setx ASPNETCORE\_ENVIRONMENT Development /M # Set for the user account (persistent)

* **macOS and Linux (bash):**

1. export ASPNETCORE\_ENVIRONMENT=Development # Set to Development
2. export ASPNETCORE\_ENVIRONMENT=Production # Set to Production

**Making Environment Variables Persistent**

* **Windows (System Properties):**
  1. Right-click on "This PC" and select "Properties".
  2. Click on "Advanced system settings".
  3. Click the "Environment Variables" button.
  4. Under "System variables" (or "User variables" for a specific user), click "New".
  5. Enter ASPNETCORE\_ENVIRONMENT as the variable name and the desired environment as the value.
* **macOS and Linux (Shell Configuration Files):**
  1. Edit your shell's configuration file (.bashrc, .zshrc, etc.) and add the following line (replacing Development with your desired environment):Bash
     1. export ASPNETCORE\_ENVIRONMENT=Development
  2. After saving the file, run source ~/.bashrc (or the appropriate command for your shell) to reload the configuration.

**Important Considerations**

* **Overriding:** When multiple ways of setting the environment are used, the most specific one takes precedence. For example, a value set in the terminal will override the value in launchSettings.json.
* **Case-Sensitivity (Linux/macOS):** Environment variable names are case-sensitive on Linux and macOS. Be sure to use the correct capitalization (ASPNETCORE\_ENVIRONMENT).
* **Environment-Specific Configuration Files:** Even after setting the environment variable, ensure you have the corresponding appsettings.{Environment}.json files in your project to load the correct settings for that environment.

**Example: Running Your App with Different Environments**

1. # PowerShell
2. $env:ASPNETCORE\_ENVIRONMENT = "Development"
3. dotnet run
5. # Command Prompt (Windows)
6. set ASPNETCORE\_ENVIRONMENT=Production
7. dotnet run
9. # bash (macOS/Linux)
10. export ASPNETCORE\_ENVIRONMENT=Staging
11. dotnet run

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).
* **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.

Introduction to Environments

An environment represents is a system in which the application is deployed and executed.

**Development**

The environment, where the developer makes changes in the code, commits code to the source control.

**Staging**

The environment, where the application runs on a server, from which other developers and quality controllers access the application.

**Production**

The environment, where the real end-users access the application.

Shortly, it's where the application "live" to the audience.

Environment Setting

**Set Environment in launchSettings.json**

in launchSettings.json

1. {
2. "profiles":
3. {
4. "profileName":
5. {
6. "environmentVariables":
7. {
8. "DOTNET\_ENVIRONMENT": "EnvironmentNameHere",
9. "ASPNETCORE\_ENVIRONMENT": "EnvironmentNameHere"
10. }
11. }
12. }
13. }

**Access Environment in Program.cs**

app.Environment

IWebHostEnvironment

**EnvironmentName**

Gets or sets name of the environment.

By default it reads the value from either DOTNET\_ENVIRONMENT or ASPNETCORE\_ENVIRONMENT.

**ContentRootPath**

Gets or sets absolute path of the application folder.

**IsDevelopment()**

Returns Boolean true, if the current environment name is "Development".

**IsStaging()**

Returns Boolean true, if the current environment name is "Staging".

**IsProduction()**

Returns Boolean true, if the current environment name is "Production".

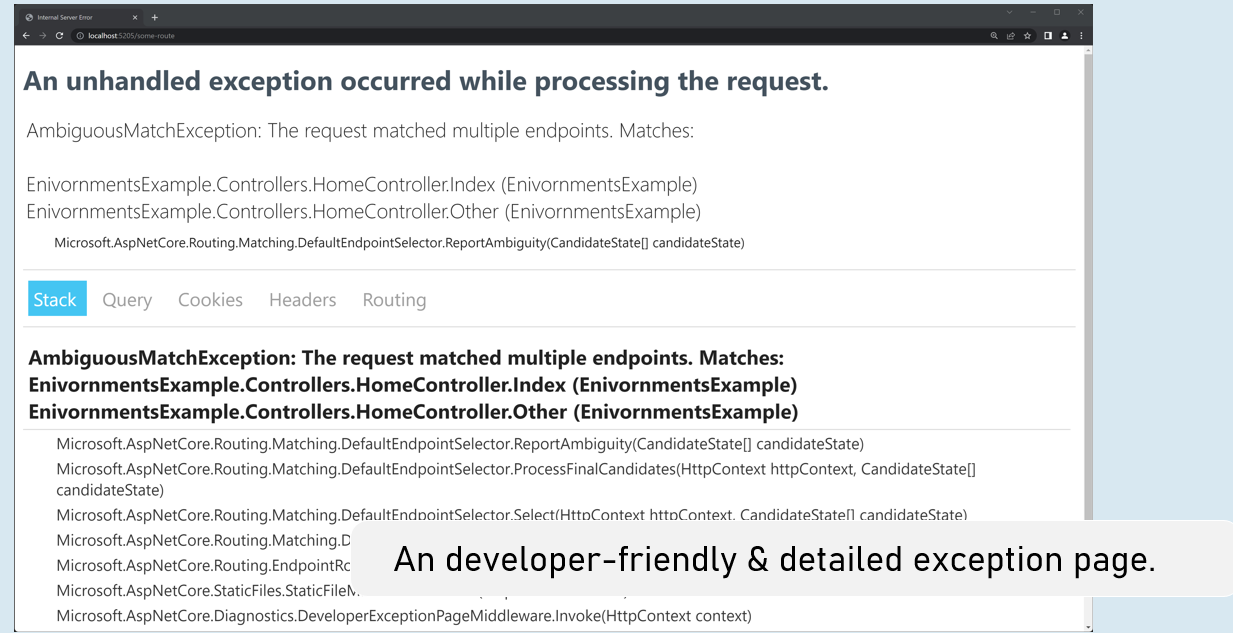
**IsEnvironment(string environmentName)**

Returns Boolean true, if the current environment name matches with the specified environment.

Access Environment in Controller and other classes

1. using Microsoft.AspNetCore.Mvc;
2. using Microsoft.AspNetCore.Hosting;
4. public class ControllerName : Controller
5. {
6. private readonly IWebHostEnvironment \_webHost;
8. public ControllerName(IWebHostEnvironment webHost)
9. {
10. \_webHost = webHost;
11. }
12. }

Developer Exception Page

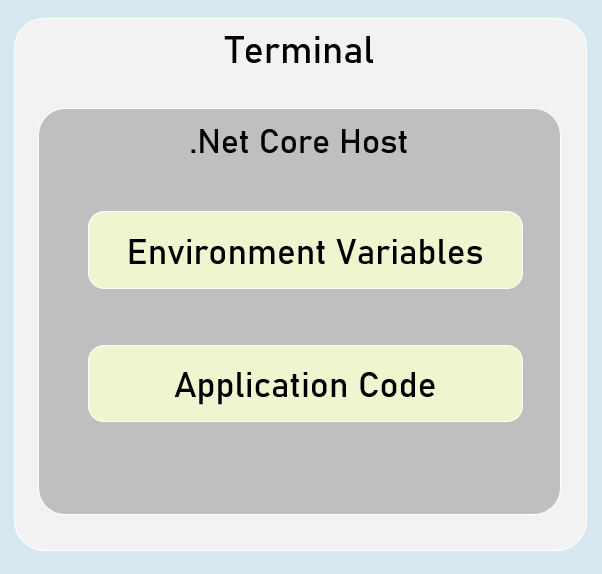


Enable developer exception page

in Program.cs

1. if (app.Environment.IsDevelopment()
2. {
3. app.UseDeveloperExceptionPage();
4. }

Process-Level Environment



The environment variables are stored & accessible within the same process only.

Setting Environment Variables in Process

in "Windows PowerShell" / "Developer PowerShell in VS"

$Env:Environment="EnvironmentName"

dotnet run --no-launch-profile

<environment> tag helper

**include**

1. <environment include="Environment1,Environment2">
2. html content here
3. </environment>

It renders the content only when the current environment name matches with either of the specified environment names in the "include" property.

**exclude**

1. <environment exclude="Environment1,Environment2">
2. html content here
3. </environment>

It renders the content only when the current environment name doesn't match with either of the specified environment names in the "exclude" property.

# **12. Configuration [MVC vs API]**

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

1. var builder = WebApplication.CreateBuilder(args);
2. var configuration = builder.Configuration;
4. // Add configuration sources in the desired order of precedence (last added wins)
5. configuration.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true);
6. configuration.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);
7. configuration.AddEnvironmentVariables();
8. configuration.AddUserSecrets<Program>(); // For development secrets
9. // ... other sources ...
10. AddJsonFile: Loads configuration from JSON files.
11. AddEnvironmentVariables: Loads configuration from environment variables.
12. 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:**

1. public class HomeController : Controller
2. {
3. private readonly IConfiguration \_configuration; // Field to store IConfiguration
5. public HomeController(IConfiguration configuration)
6. {
7. \_configuration = configuration;
8. }
10. public IActionResult Index()
11. {
12. var myKeyValue = \_configuration["MyKey"]; // Access configuration value
13. return View();
14. }
15. }

* **In Services:**

1. public class EmailService : IEmailService
2. {
3. private readonly IConfiguration \_configuration;
5. public EmailService(IConfiguration configuration)
6. {
7. \_configuration = configuration;
8. }
10. public void SendEmail(string to, string subject, string body)
11. {
12. var smtpServer = \_configuration["Email:SmtpServer"]; // Use configuration for email settings
13. // ... (email sending logic)
14. }
15. }

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**

1. // MyOptions.cs
2. public class MyOptions
3. {
4. public string Option1 { get; set; }
5. public int Option2 { get; set; }
6. }
8. // Program.cs (or Startup.cs)
9. builder.Services.Configure<MyOptions>(builder.Configuration.GetSection("MyOptions"));
11. // MyService.cs
12. public class MyService : IMyService
13. {
14. private readonly IOptions<MyOptions> \_options;
16. public MyService(IOptions<MyOptions> options)
17. {
18. \_options = options;
19. }
21. public void DoSomething()
22. {
23. var option1Value = \_options.Value.Option1;
24. // ...
25. }
26. }

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):**

1. {
2. "ConnectionStrings": {
3. "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDatabase;Trusted\_Connection=True;"
4. },
5. "Logging": {
6. "LogLevel": {
7. "Default": "Information",
8. "Microsoft.AspNetCore": "Warning"
9. }
10. },
11. "Inventory": {
12. "StockAlertThreshold": 20,
13. "WarehouseLocations": [
14. "New York",
15. "London",
16. "Tokyo"
17. ]
18. }
19. }

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**

1. var connectionString = \_configuration.GetConnectionString("DefaultConnection");
3. var logLevel = \_configuration.GetValue<string>("Logging:LogLevel:Default");
5. // Using IConfigurationSection:
6. var inventorySection = \_configuration.GetSection("Inventory");
7. var stockAlertThreshold = inventorySection.GetValue<int>("StockAlertThreshold");
9. // Get an array
10. 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**

1. // InventoryOptions.cs
2. public class InventoryOptions
3. {
4. public int StockAlertThreshold { get; set; }
5. public string[] WarehouseLocations { get; set; }
6. }
8. // Program.cs (or Startup.cs)
9. builder.Services.Configure<InventoryOptions>(builder.Configuration.GetSection("Inventory"));
11. // In your service or controller
12. public class InventoryService : IInventoryService
13. {
14. private readonly InventoryOptions \_options;
16. public InventoryService(IOptions<InventoryOptions> options)
17. {
18. \_options = options.Value;
19. }
21. // ... use \_options.StockAlertThreshold and \_options.WarehouseLocations
22. }

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.
2. public class EmailOptions
3. {
4. public string SmtpServer { get; set; } = string.Empty;
5. public int SmtpPort { get; set; } = 25;
6. public string SenderEmail { get; set; } = string.Empty;
7. public string SenderPassword { get; set; } = string.Empty;
8. }
9. **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:
10. 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.

1. **Inject IOptions<T>:** Inject the IOptions<T> interface into your controllers or services to access the bound options:
2. public class EmailService : IEmailService
3. {
4. private readonly EmailOptions \_options;
6. public EmailService(IOptions<EmailOptions> options)
7. {
8. \_options = options.Value;
9. }
11. // ... use \_options.SmtpServer, \_options.SmtpPort, etc. ...
12. }

**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**

1. // Program.cs (or Startup.cs)
2. var emailOptions = new EmailOptions();
3. builder.Configuration.GetSection("Email").Bind(emailOptions);
4. 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**

1. // appsettings.json
2. {
3. "ConnectionStrings": {
4. "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDatabaseDev;Trusted\_Connection=True;"
5. }
6. }
8. // appsettings.Production.json
9. {
10. "ConnectionStrings": {
11. "DefaultConnection": "Server=myprodserver;Database=MyDatabaseProd;User Id=myuser;Password=mypassword;"
12. }
13. }

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

**Code Example: GetSection() and GetValue()**

1. var connectionString = \_configuration.GetConnectionString("DefaultConnection");
3. 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:
   1. 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.

1. **Set a Secret:** Use the set command to store a secret:
   1. dotnet user-secrets set "MySecretName" "MySecretValue"

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

1. **List Secrets (Optional):**
   1. dotnet user-secrets list

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

1. **Remove a Secret (Optional):**
   1. dotnet user-secrets remove "MySecretName"

**Accessing User Secrets in Your Code**

1. var builder = WebApplication.CreateBuilder(args);
2. var configuration = builder.Configuration;
4. // In Program.cs (or Startup.cs):
5. if (builder.Environment.IsDevelopment())
6. {
7. configuration.AddUserSecrets<Program>();
8. }

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:

1. 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):**

1. 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)**

1. $env:ASPNETCORE\_MyKey = "myvalue" # Simple key-value
2. $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)**

1. set ASPNETCORE\_MyKey=myvalue # Simple key-value
2. set ASPNETCORE\_Logging\_\_LogLevel\_\_Default=Debug # Hierarchical key

**Bash (macOS, Linux)**

1. export ASPNETCORE\_MyKey="myvalue" # Simple key-value
2. 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:**
   1. # In PowerShell
   2. $env:ASPNETCORE\_ConnectionStrings\_\_DefaultConnection = "Server=myServer;Database=myDb;Trusted\_Connection=True;"
   4. # In Command Prompt (Windows)
   5. set ASPNETCORE\_ConnectionStrings\_\_DefaultConnection="Server=myServer;Database=myDb;Trusted\_Connection=True;"
   7. # In Bash (macOS/Linux)
   8. export ASPNETCORE\_ConnectionStrings\_\_DefaultConnection="Server=myServer;Database=myDb;Trusted\_Connection=True;"

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

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

1. $env:ASPNETCORE\_MyKey = "myvalue" # Simple key-value
2. $env:ASPNETCORE\_Logging\_\_LogLevel\_\_Default = "Debug" # Hierarchical key

**Command Prompt (Windows)**

1. set ASPNETCORE\_MyKey=myvalue # Simple key-value
2. set ASPNETCORE\_Logging\_\_LogLevel\_\_Default=Debug # Hierarchical key

**Bash (macOS, Linux)**

1. export ASPNETCORE\_MyKey="myvalue" # Simple key-value
2. 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:**
   1. # In PowerShell or Bash
   2. $env:ASPNETCORE\_ConnectionStrings\_\_DefaultConnection = "Server=myServer;Database=myDb;User Id=myuser;Password=mypassword;"
   4. # In Command Prompt (Windows)
   5. set ASPNETCORE\_ConnectionStrings\_\_DefaultConnection="Server=myServer;Database=myDb;User Id=myuser;Password=mypassword;"
2. **Access in Your Code:** You can then retrieve this connection string in your ASP.NET Core application as usual:
   1. 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:
2. {
3. "CustomSettings": {
4. "APIKey": "your\_api\_key",
5. "FeatureEnabled": true,
6. "NotificationSettings": {
7. "EmailEnabled": true,
8. "SMSEnabled": false
9. }
10. }
11. }
12. **Add to Configuration:** In your Program.cs, use the AddJsonFile extension method to include your custom JSON file:
13. var builder = WebApplication.CreateBuilder(args);
14. var configuration = builder.Configuration;
16. // ... (other configuration sources) ...
18. // Add the custom JSON file:
19. configuration.AddJsonFile("customsettings.json", optional: true, reloadOnChange: true);
21. var app = builder.Build();
22. // ... (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:

1. // Option 1: Directly using IConfiguration
2. var apiKey = configuration["CustomSettings:APIKey"];
3. var featureEnabled = configuration.GetValue<bool>("CustomSettings:FeatureEnabled");
5. // Option 2: Options Pattern
6. 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**

1. // CustomSettings.cs (Options Class)
2. public class CustomSettings
3. {
4. public string APIKey { get; set; }
5. public bool FeatureEnabled { get; set; }
6. public NotificationSettings NotificationSettings { get; set; }
7. }
9. // ... (other options classes if needed) ...
11. // Program.cs
12. builder.Services.Configure<CustomSettings>(configuration.GetSection("CustomSettings"));
14. // In your controller or service
15. public class MyController : Controller
16. {
17. private readonly CustomSettings \_settings;
19. public MyController(IOptions<CustomSettings> settings)
20. {
21. \_settings = settings.Value;
22. }
23. }

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.

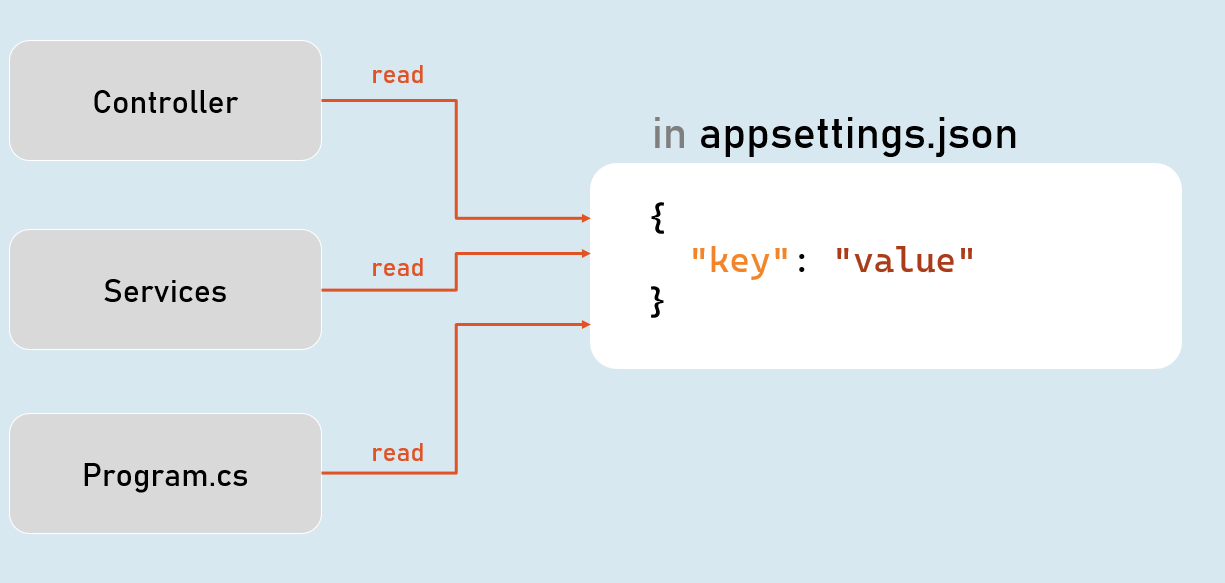
**Interview Tips**

* **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.

**Configuration Settings**

**Configuration (or configuration settings) are the constant key/value pairs that are set at a common location and can be read from anywhere in the same application.**

**Examples: connection strings, Client ID & API keys to make REST-API calls, Domain names, Constant email addresses etc.**

****

**Configuration Sources**

1. **appsettings.json**
2. **Environment Variables**
3. **File Configuration (JSON, INI or XML files)**
4. **In-Memory Configuration**
5. **Secret Manager**

**Access Configuration**

**in Program.cs:**

**app.Configuration**

**IConfiguration**

**[string key]**

**Gets or sets configuration value at the specified key.**

**GetValue<T>(string key, object defaultValue)**

**Gets the configuration value at the specified key; returns the default value if the key doesn't exists.**

**IConfiguration in Controller**

**in Controller and other classes**

1. **using Microsoft.AspNetCore.Mvc;**
2. **using Microsoft.Extensions.Configuration;**
4. **public class ControllerName : Controller**
5. **{**
6. **private readonly IConfiguration \_configuration;**
8. **public ControllerName(IConfiguration configuration)**
9. **{**
10. **\_configuration = configuration;**
11. **}**
12. **}**

**Hierarchical Configuration**

**in appsettings.json**

1. **{**
2. **"MasterKey":**
3. **{**
4. **"Key1": "value"**
5. **"Key2": "value"**
6. **}**
7. **}**

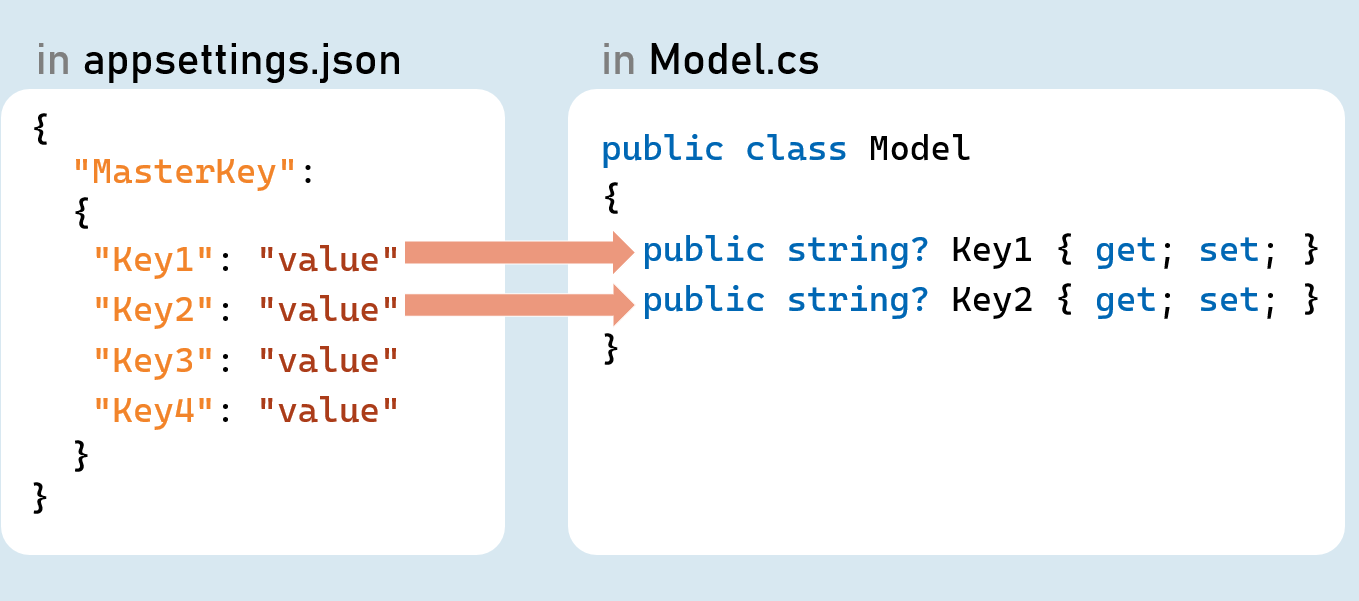
**to read configuration**

**Configuration["MasterKey:Key1"]**

**IConfiguration.GetSection(string key)**

**Returns an IConfigurationSection based on the specified key.**

**Options Pattern**

****

**Options pattern uses custom classes to specify what configuration settings are to be loaded into properties.**

**Examples: Reading the specific connections strings out of many configuration settings.**

**The option class should be a non-abstract class with a public parameterless constructor.**

**Public read-write properties are bound.**

**Fields are not bound.**

**IConfiguration.GetSection(string key)**

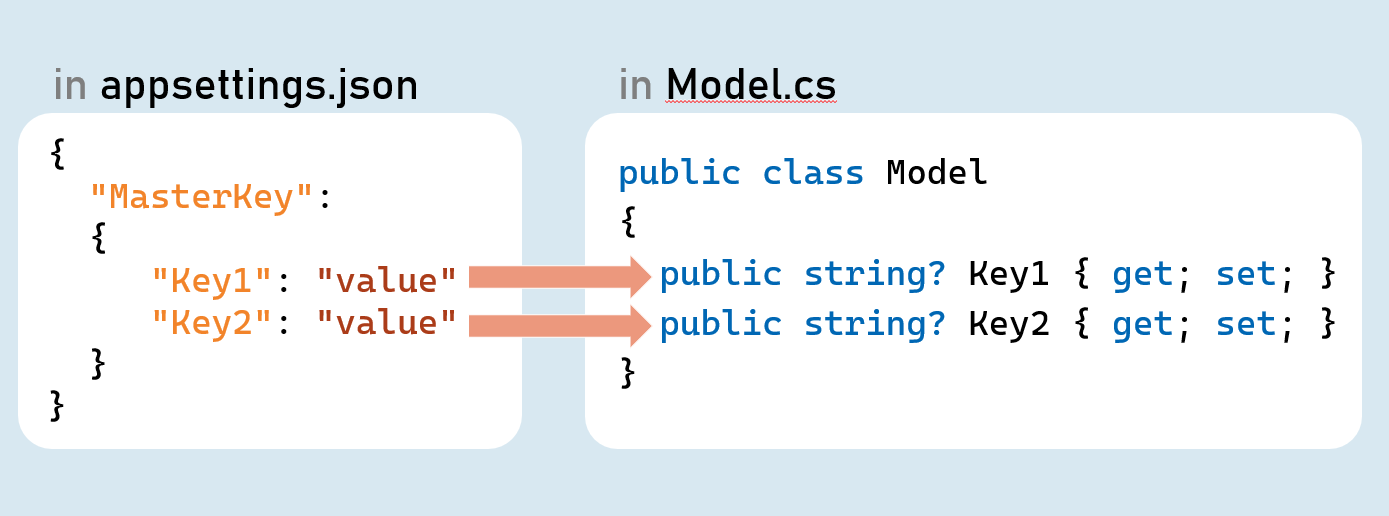
**Returns an IConfigurationSection based on the specified key.**

**IConfiguration.Bind(object instance) and IConfiguration.Get<T>()**

**Binds (loads) configuration key/value pairs into a new object of the specified type.**

**Configuration as Service**

**Inject Configuration as Service**

****

**Add Configuration as Service**

**in Program.cs:**

**builder.Services.Configure<Model>(builder.Configuration.GetSection("MasterKey"));**

**Inject Configuration as Service in Controller in Controller and other classes**

1. **using Microsoft.AspNetCore.Mvc;**
2. **using Microsoft.Extensions.Options;**
4. **public class ControllerName : Controller**
5. **{**
6. **private readonly Model \_options;**
8. **public ControllerName(IOptions<Model> options)**
9. **{**
10. **\_options = options.Value;**
11. **}**
12. **}**

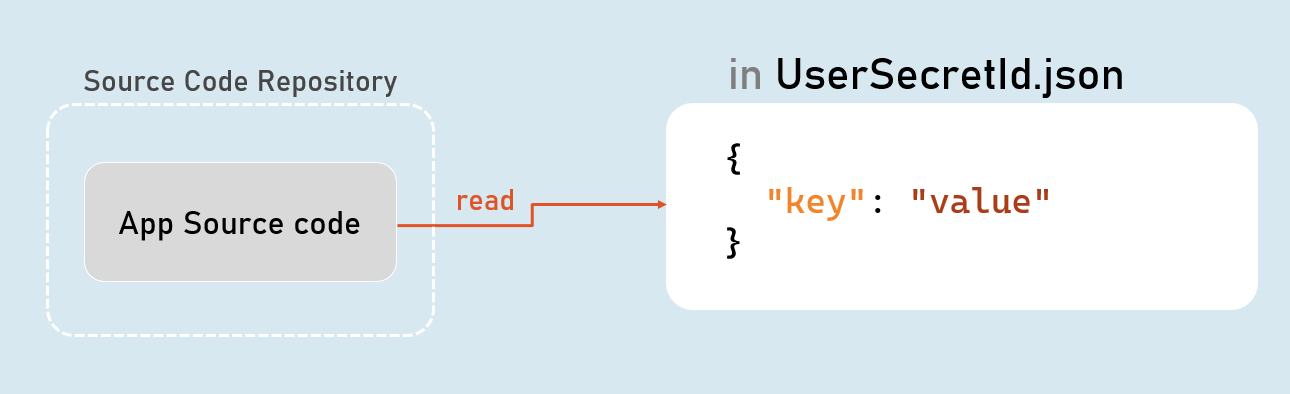
**Environment Specific Configuration**

**Order of Precedence of Configuration Sources**

****

**Secrets Manager**

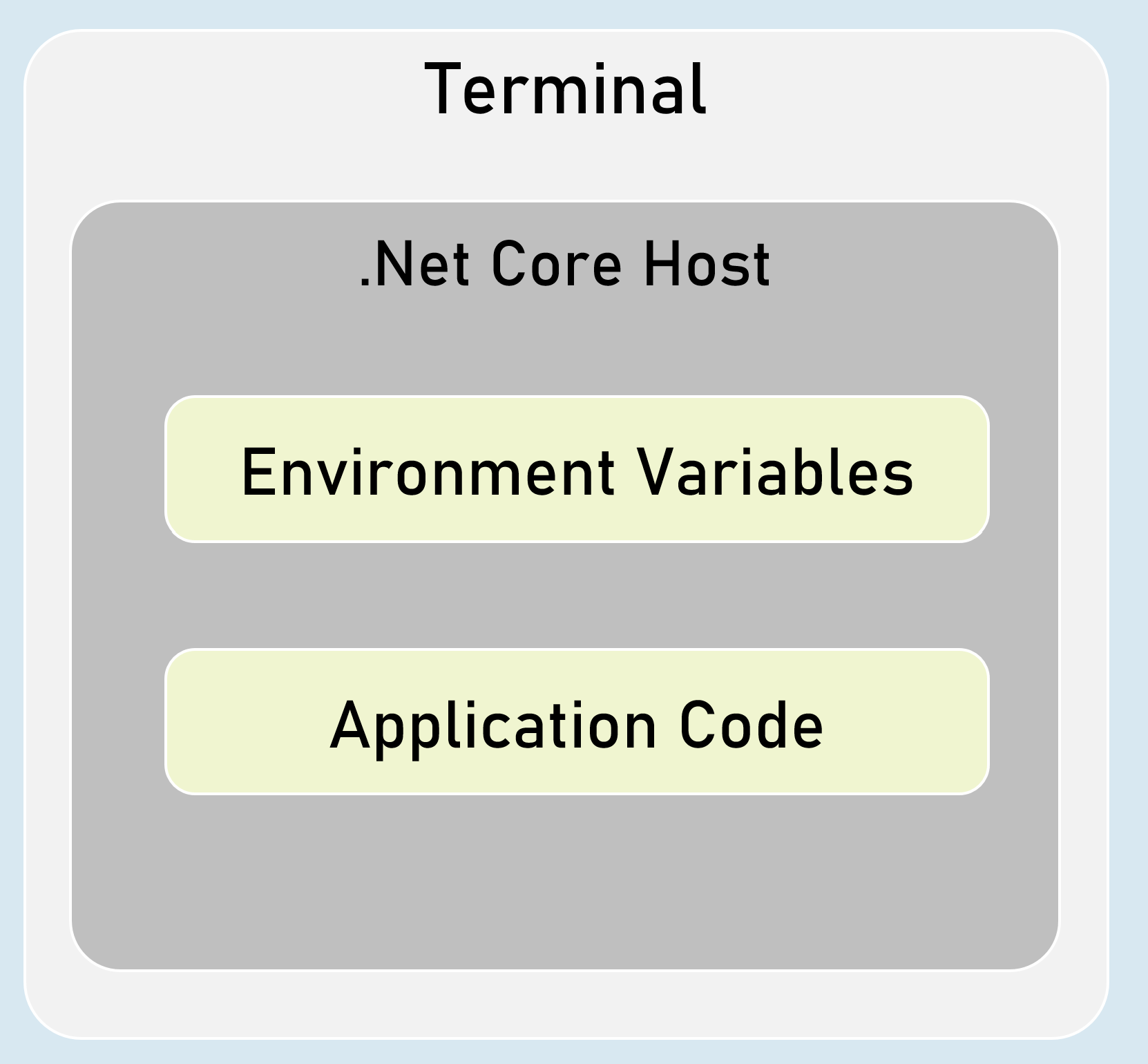
**The 'secrets manager ' stores the user secrets (sensitive configuration data) in a separate location on the developer machine.**

****

**Enable Secrets Manager in "Windows PowerShell" / "Developer PowerShell in VS"**

1. **dotnet user-secrets init**
2. **dotnet user-secrets set "Key" "Value"**
3. **dotnet user-secrets list**

**Environment Variables Configuration**

****

**You can set configuration values as in-process environment variables.**

**Set Configuration as Environment Variables**

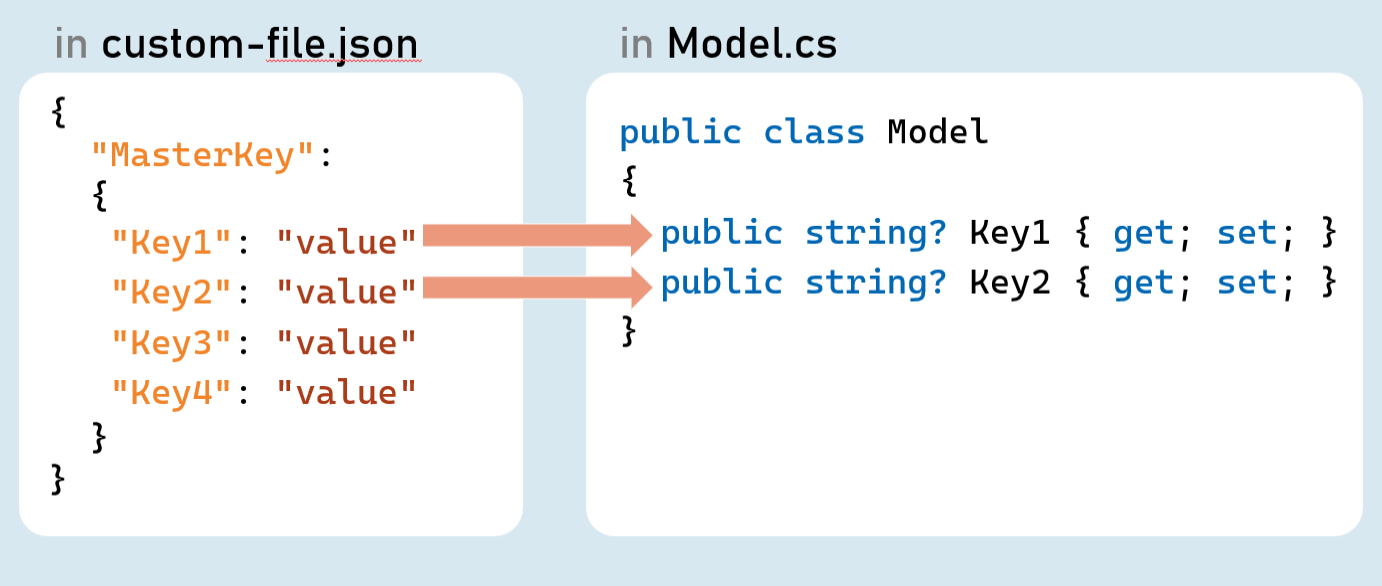
**in "Windows PowerShell" / "Developer PowerShell in VS":**

1. **$Env:ParentKey\_\_ChildKey="value"**
2. **dotnet run --no-launch-profile**

**It is one of the most secured way of setting-up sensitive values in configuration.**

**\_\_ (underscore and underscore) is the separator between parent key and child key.**

**Custom Json Configuration**

****

**Add Custom Json file as Configuration Source**

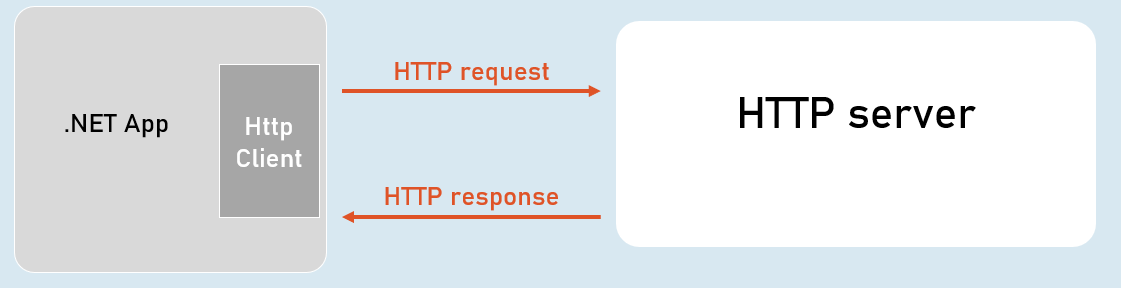
**in Program.cs:**

1. **builder.Host.ConfigureAppConfiguration( (hostingContext, config) => {**
2. **config.AddJsonFile("filename.json", optional: true, reloadOnChange: true);**
3. **});**

**Http Client**

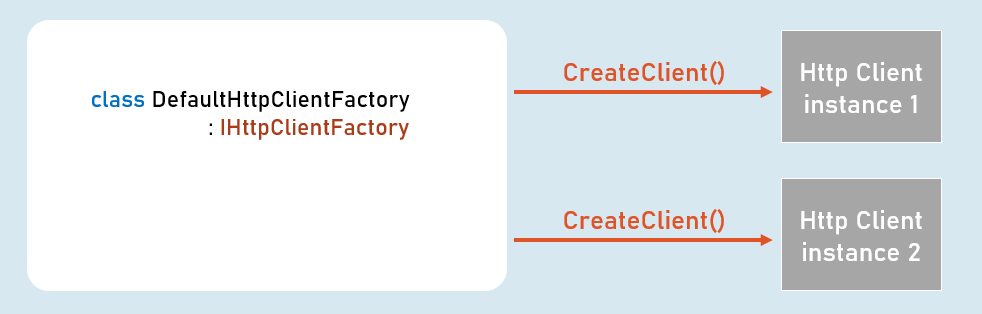
**HttpClient is a class for sending HTTP requests to a specific HTTP resource (using its URL) and receiving HTTP responses from the same.**

**Examples: Making a request to a third-party weather API, ChatGPT etc.**

****

**IHttpClientFactory**

**IHttpClientFactory is an interface that provides a method called CreateClient() that creates a new instance of HttpClient class and also automatically disposes the same instance (closes the connection) immediately after usage.**

****

**HttpClient**

**Properties**

* **BaseAddress**
* **DefaultRequestHeaders**

**Methods**

* **GetAsync()**
* **PostAsync()**
* **PutAsync()**
* **DeleteAsync()**

# **13. xUnit [MVC vs API]**

# **14. CRUD Operations [MVC]**

# **15. Tag Helpers [MVC]**

# **16. Entity Framework core [MVC vs API]**

# **17. Advanced Unit Testing – Moq and Repository Pattern [MVC vs API]**

# **18. Logging and Serilog [MVC vs API]**

# **19. Filter [MVC vs API]**

# **20. Error handling [MVC vs API]**

# **21. SOLID Principles [MVC vs API]**

# **22. Clean Architecture [MVC vs API]**

# **23. Identity , Authorization, Security [MVC vs API]**

# **24. Asp.net Core Web API [API]**

# **25. Swagger and OpenAPI [API]**

# **26. Angular and CORS [API]**

# **27. JWT and Web API Authentication [API]**

# **28. Minimal API [API]**