* 1. **Rest Constraints**
     1. **Uniform Interface**
        + should be a uniform way of interacting with a given server irrespective of device or type of application.
     2. **Client-server**
        + REST application should have a client-server architecture.
        + A Client is someone who is requesting resources and are not concerned with data storage, which remains internal to each server.
        + A server is someone who holds the resources and are not concerned with the user interface or user state.
     3. **Stateless** 
        + It means that the necessary state to handle the request is contained within the request itself and server would not store anything related to the session.
        + In REST, the client must include all information for the server to fulfil the request whether as a part of query params, headers or URI.
     4. **Cacheable**
        + Every response should include whether the response is cacheable or not and for how much duration responses can be cached at the client side.
     5. **Layered system**
        + An application architecture needs to be composed of multiple layers.
        + Each layer doesn’t know anything about any layer other than that of immediate layer and there can be lot of intermediate servers between client and the end server.
     6. **Code on demand**
        + It is an optional feature. According to this, servers can also provide executable code to the client.

* 1. **Routing**
     1. C*ontroller* is a class that handles HTTP requests.
     2. Asp.net MVC uses Conventional routing by using **app.MapControllerRoute**
     3. Web api uses Attribute routing by using **app.MapControllers()**
     4. When the Web API framework receives a request, it routes the request to an action.
     5. To determine which action to invoke, the framework uses a *routing table*

varbuilder = WebApplication.CreateBuilder(args);  
Var app = builder.Build();  
app.MapGet("/", () => "Hello World!");

app.Run();

|  |  |  |
| --- | --- | --- |
| **Route Template** | **Example Matching URI** | **The request URI…** |
| hello | /hello | Only matches the single path /hello. |
| {Page=Home} | / | Matches and sets Page to Home. |
| {Page=Home} | /Contact | Matches and sets Page to Contact. |
| {controller}/{action}/{id?} | /Products/List | Maps to the Products controller and List action. |
| {controller}/{action}/{id?} | /Products/Details/123 | Maps to the Products controller and Details action with id set to 123. |
| {controller=Home}/{action=Index}/{id?} | / | Maps to the Home controller and Index method. id is ignored. |
| {controller=Home}/{action=Index}/{id?} | /Products | Maps to the Products controller and Index method. id is ignored. |
|  |  |  |

* 1. **Route constrains** - Route constraints let you restrict how the parameters in the route template are matched

|  |  |  |  |
| --- | --- | --- | --- |
| **constraint** | **Example** | **Example Matches** | **Notes** |
| int | {id:int} | 123456789, -123456789 | Matches any integer |
| bool | {active:bool} | true, FALSE | Matches true or false. Case-insensitive |
| datetime | {dob:datetime} | 2016-12-31, 2016-12-31 7:32pm | Matches a valid DateTime value in the invariant culture. See preceding warning. |
| decimal | {price:decimal} | 49.99, -1,000.01 | Matches a valid decimal value in the invariant culture. See preceding warning. |
| double | {weight:double} | 1.234, -1,001.01e8 | Matches a valid double value in the invariant culture. See preceding warning. |
| float | {weight:float} | 1.234, -1,001.01e8 | Matches a valid float value in the invariant culture. See preceding warning. |
| guid | {id:guid} | CD2C1638-1638-72D5-1638-DEADBEEF1638 | Matches a valid Guid value |
| long | {ticks:long} | 123456789, -123456789 | Matches a valid long value |
| minlength(value) | {username:minlength(4)} | Rick | String must be at least 4 characters |
| maxlength(value) | {filename:maxlength(8)} | MyFile | String must be no more than 8 characters |
| length(length) | {filename:length(12)} | somefile.txt | String must be exactly 12 characters long |
| length(min,max) | {filename:length(8,16)} | somefile.txt | String must be at least 8 and no more than 16 characters long |
| min(value) | {age:min(18)} | 19 | Integer value must be at least 18 |
| max(value) | {age:max(120)} | 91 | Integer value must be no more than 120 |
| range(min,max) | {age:range(18,120)} | 91 | Integer value must be at least 18 but no more than 120 |
| alpha | {name:alpha} | Rick | String must consist of one or more alphabetical characters, a-z and case-insensitive. |
| regex(expression) | {ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)} | 123-45-6789 | String must match the regular expression. See tips about defining a regular expression. |
| required | {name:required} | Rick | Used to enforce that a non-parameter value is present during URL generation |

* + 1. Example:

[Route("users/{id:int:min(1)}")]  
publicUser GetUserById(intid){ }

app.MapGet("{message:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}",  
 () => "Inline Regex Constraint Matched");

* + 1. Can create custom route constraint by using **IRouteConstraint**
    2. Route Names - **[Route("api/books/{id}", Name="GetBookById")]**
    3. Route Order - **[Route("pending", RouteOrder = 1)]**

* 1. **Http Verbs**
     1. [HttpGet]
     2. [HttpPut]
     3. [HttpPost]
     4. [HttpDelete]
     5. [HttpHead] - retrieving meta-information written in response headers without having to download the entire content
     6. [HttpOptions] - The HTTP OPTIONS method requests permitted communication options for a given URL or server. response will be like **Allow: OPTIONS, GET, HEAD, POST**
     7. [HttpPatch] - partial data that is to be updated without modifying the entire data
     8. To allow multiple HTTP verbs for an action, or to allow HTTP verbs other than GET, PUT, POST, DELETE, HEAD, OPTIONS, and PATCH, use the [AcceptVerbs] attribute, which takes a list of HTTP verbs.

[AcceptVerbs("GET", "HEAD")]  
 publicProduct FindProduct(id){ }

* + 1. [NonAction] - To prevent a method from getting invoked as an action
  1. **Dependency Injection**
     1. <https://learn.microsoft.com/en-us/aspnet/web-api/overview/advanced/dependency-injection>
     2. ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving **Inversion of Control (IoC)** between classes and their dependencies.

Example: **builder.Services.AddScoped<IMyDependency, MyDependency>();**

* + 1. Types

1. AddScoped -
   * + - Scoped objects are the same for a given request but differ across each new request.
       - A scoped lifetime indicates that services are created once per client request (connection)
       - Scoped services are disposed at the end of the request

1. AddTransient
   * + - Transient lifetime services are created each time they're requested from the service container
       - Transient services are disposed at the end of the request
       - This lifetime incurs per/request allocations, as services are resolved and constructed every time

1. AddSingleton
   * + - Singleton objects are the same for every request.
       - The first time they're requested.
       - By the developer, when providing an implementation instance directly to the container.
       - Every subsequent request of the service implementation from the dependency injection container uses the same instance.
     1. Keyed Services
        + Keyed services refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys

Example **builder.Services.AddKeyedSingleton<ICache, BigCache>("big");** -- big is the class name

* + - * AddKeyedScoped
      * AddKeyedTransient
      * AddKeyedSingleton
    1. Resolve a scoped service for a limited duration when the app starts

Example:

using (var scope = app.Services.CreateScope())

{

var db = scope.ServiceProvider.GetRequiredService<ShopContext>();

await db.Database.EnsureCreatedAsync();

}

* + 1. Disposal of services will happen by container itself based on type(scoped or transient or singleton)
  1. **Middleware & Request pipeline**
     1. Middleware is software that's assembled into an app pipeline to handle requests and responses.

Each component:

* + - * Chooses whether to pass the request to the next component in the pipeline.
      * Can perform work before and after the next component in the pipeline.
    1. Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.
    2. Request delegates are configured using **Run**, **Map**, **Next** and **Use** extension methods.
    3. The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other
    4. 
  1. **Extension methods**
     1. **Use()** Extension Method is used to insert a new Middleware component to the Request Processing Pipeline.
     2. **Next()** Extension Method is used to call the next middleware component in the request processing pipeline.
     3. **Map()** Extension Method is used to map the Middleware to a specific URL.
     4. **Run()** The first Run delegate is always terminal and terminates the pipeline.

Example

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.Use(async (context, next) =>

{

// Do work that can write to the Response.

await next.Invoke();

// Do logging or other work that doesn't write to the Response.

});

app.Run(async context =>

{

await context.Response.WriteAsync("Hello from 2nd delegate.");

});

app.Run();

호세c Files 
(띠에” rniddlewares 

* + 1. When a delegate doesn't pass a request to the next delegate, it's called short-circuiting the request pipeline
    2. Run delegates don't receive a next parameter. The first Run delegate is always terminal and terminates the pipeline
    3. Middleware order
  1. **Return Types**
     1. Specific type - Returns a primitive or complex data type, like string or custom object

[HttpGet("syncsale")]

public IEnumerable<Product> GetOnSaleProducts(){}

* + 1. IActionResult
       - Appropriate when multiple ActionResult return types are possible in an action.
       - The ActionResult types represent various HTTP status codes.

public IActionResult GetById\_IActionResult(int id)

{

var product = \_productContext.Products.Find(id);

return product == null ? NotFound() : Ok(product);

}

* + 1. ActionResult<T>
    2. The [ProducesResponseType] attribute's Type property can be excluded. For example, [ProducesResponseType(200, Type = typeof(Product))] is simplified to [ProducesResponseType(200)]. The action's expected return type is inferred from the T in ActionResult<T>.

* + 1. Implicit cast operators support the conversion of both T and ActionResult to ActionResult<T>. T converts to ObjectResult, which means return new ObjectResult(T); is simplified to return T;.

public ActionResult<IEnumerable<Product>> Get() =>

\_repository.GetProducts();

* + 1. HttpResults
    2. IResult

publicIResult GetById(intid)

* 1. **Content Negotiation**
     1. Content negotiation is the process in which the client and server agree to exchange data in a representational format when the server is allowed to exchange data in multiple representational formats.
     2. Like JSON, XML, HTML, and plain text.
     3. Content negotiation happens when a client specifies the media type it wants as a response to the request Accept header
     4. By default, ASP.NET Core Web API returns a JSON formatted result and it will ignore the browser Accept header
     5. Supported media types by default
        + application/json
        + text/json
        + text/plain
     6. The primary mechanism for content negotiation in HTTP are these request headers:
        + **Accept**: Which media types are acceptable for the response, such as "application/json," "application/xml," or a custom media type such as "application/vnd.example+xml"
        + **Accept-Charset**: Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
        + **Accept-Encoding**: Which content encodings are acceptable, such as gzip.
        + **Accept-Language**: The preferred natural language, such as "en-us".

* + 1. How Content Negotiation Works
       - First, the pipeline gets the IContentNegotiator service from the HttpConfiguration object. It also gets the list of media formatters from the HttpConfiguration.Formatters collection.
       - Next, the pipeline calls IContentNegotiator.Negotiate, passing in:
         1. The type of object to serialize
         2. The collection of media formatters
         3. The HTTP request
       - The Negotiate method returns two pieces of information:
         1. Which formatter to use
         2. The media type for the response
    2. Restricting media types in Content negotiation

builder.Services.AddControllers(options =>

{

options.RespectBrowserAcceptHeader = true;

options.ReturnHttpNotAcceptable = true;

}).AddXmlSerializerFormatters();

* 1. **Authentication and authorization** 
     1. OAuth/OpenID Conect
     2. <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/security?view=aspnetcore-8.0>
     3. <https://learn.microsoft.com/en-us/aspnet/core/migration/identity?view=aspnetcore-8.0> -- asp.net core
     4. Identity Server
  2. **Model binding**
     1. [FromBody]
        + Data from the body of the HTTP request, mostly POST/PUT
        + At most one parameter is allowed to read from the message body - only one FromBody parameter
     2. [FromUri] - Data from the route template
        + orders/{id}

* + 1. [FromQuery] - Data from the URL
       - orders/123?showHistory=true

* 1. **Model validation**
     1. We can use attributes from **using System.ComponentModel.DataAnnotations**;
     2. Like,

[Required]

public string Name { get; set; }

[Range(0, 999)]

public double Weight { get; set; }

* + 1. We can check whether the model is valid:

if (ModelState.IsValid) {

// Do something with the product (not shown).

return new HttpResponseMessage(HttpStatusCode.OK);

}

Else {

return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);

}

* 1. **Error handling**
     1. Use Exception handling middleware

if(!app.Environment.IsDevelopment())  
{app.UseExceptionHandler("/error");  
}

* + 1. Then add method to handle exception and

[Route("/error")]  
publicIActionResult HandleError()=> Problem();

* + 1. Ignore on Swagger/OpenAPI specification

[ApiExplorerSettings(IgnoreApi = true)]

* 1. **Versioning API**
     1. Packages

Asp.Versioning.Mvc

Asp.Versioning.Mvc.ApiExplorer

* + 1. In Program.cs

var apiVersioningBuilder = builder.Services.AddApiVersioning(o =>  
{  
 o.AssumeDefaultVersionWhenUnspecified = true;  
 o.DefaultApiVersion = new ApiVersion(1, 0);  
 o.ReportApiVersions = true;  
 o.ApiVersionReader = ApiVersionReader.Combine(  
 new QueryStringApiVersionReader("api-version"),  
 new HeaderApiVersionReader("X-Version"),  
 new MediaTypeApiVersionReader("ver"));  
}).AddApiExplorer(options =>

{

options.GroupNameFormat = "'v'VVV";

options.SubstituteApiVersionInUrl = true;

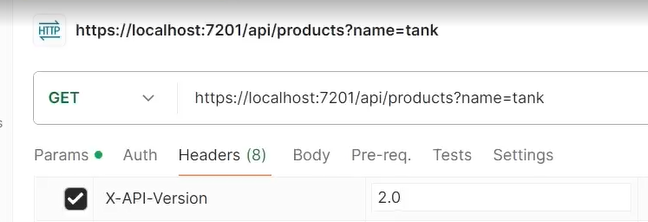
});

* + 1. HTTP Header
       - In Controller class

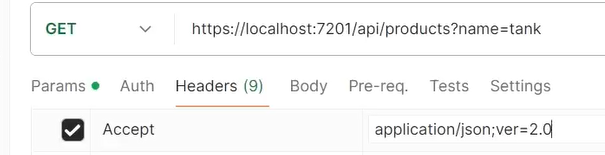
[ApiVersion("1.0")]

[Route("api/products")]

[ApiController]



* + - * Accept header



* + 1. URL
       - In Controller class

[ApiVersion("1.0")]

[Route("api/v{version:apiVersion}/products")]

[ApiController]

* + 1. <https://localhost:7136/api/v2/products>

* + 1. QueryString
       - <https://localhost:7136/api/products?api-version=2.0>

* + 1. For minimal API

**var apiVersionSet = app.NewApiVersionSet()**

**.HasApiVersion(new ApiVersion(1, 0))**

**.HasApiVersion(new ApiVersion(2, 0))**

**.ReportApiVersions()**

**.Build();**

app.MapGet("/products", async (ShopContext \_context) =>

{

return await \_context.Products.ToArrayAsync();

})**.WithApiVersionSet(apiVersionSet)**

**.MapToApiVersion(new ApiVersion(1, 0));**

* 1. **Securing API**
     1. Enforcing HTTPS
        + app.UseHttpsRedirection();
          1. Redirects HTTP requests to HTTPS, via HTTP status code 307(temporary redirect)
        + app.UseHsts();
          1. Uses HTTP Strict transport security to instruct client to use HTTPS from now on
        + For enforcing HTTPS for certain time

builder.Services.AddHsts(options =>

{

options.MaxAge = TimeSpan.FromSeconds(365);

});

* + 1. CORS – cross origin resource sharing
       - Origin header is automatically set in request by browser – **Access-Control-Allow-Origin**
       - CORS can be enabled for controller, actions or minimal API

[EnableCors]

* + - * CORS can be disabled

[DisableCors()]

* + - * Default policy in Program.cs

**builder.Services.AddCors(options =>**

**{**

**options.AddDefaultPolicy(builder =>**

**{**

**builder**

**.WithOrigins("https://localhost:7212", "http://localhost:5180", "http://localhost:4200")**

**.WithHeaders("X-API-Version");**

**});**

**});**

* + - * Then use the Cors

**app.UseCors();**

* 1. **Unit Testing**
  2. **Controllers and Actions**
  3. **EF core**
  4. **API design – architecture, best practices**
  5. **Asp.net core security**

Linked In learning

<https://www.linkedin.com/learning/building-web-apis-with-asp-dot-net-core-8/handling-errors?autoSkip=true&resume=false&u=2146489>

<https://www.linkedin.com/learning/advanced-web-apis-with-asp-dot-net-core-8/upskill-your-asp-dot-net-core-8-web-apis?u=2146489>