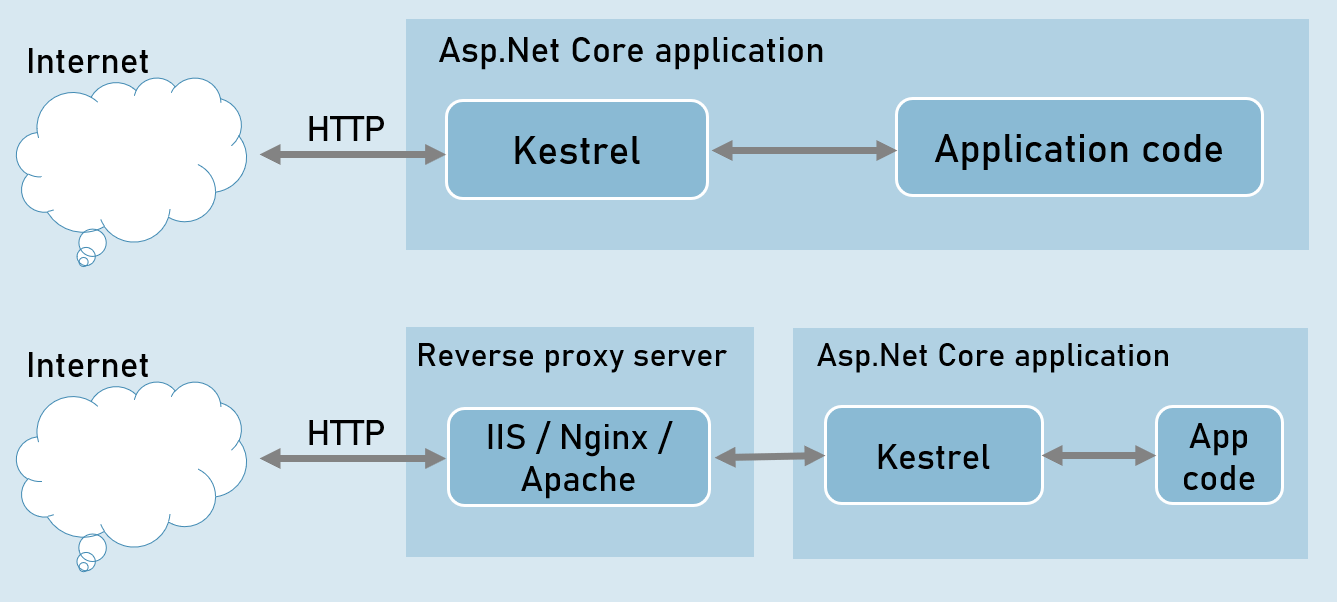
Kestrel and Other Servers

**Application Servers**

* Kestrel

**Reverse Proxy Servers**

* IIS
* Nginx
* Apache



**Benefits of Reverse Proxy Servers**

* Load Balancing
* Caching
* URL Rewriting
* Decompressing the requests
* Authentication
* Decryption of SSL Certificates

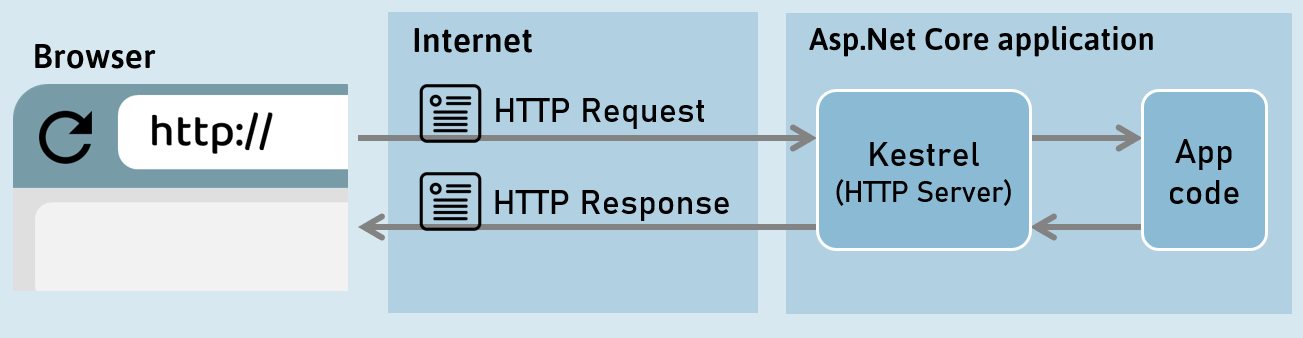
**IIS express**

* HTTP access logs
* Port sharing
* Windows authentication
* Management console
* Process activation
* Configuration API
* Request filters
* HTTP redirect rules

Introduction to HTTP

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 (Http to Https)

**2xx | Success**

200          OK

**3xx | Redirection**

302          Found

304          Not Modified (File not modifies as it in the cache)

**4xx | Client error**

400           Bad Request (Support Course Id not provided, )

401            Unauthorized

404           Not Found (Url not found)

**5xx | Server error**

500           Internal Server Error

HTTP Response Headers

**Date**

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

**Server**

Name of the server.

Ex: Server=Kestrel

**Content-Type**

MIME type of response body.

Ex: text/plain, text/html, application/json, application/xml etc.

**Content-Length**

Length (bytes) of response body.

Ex: 100

**Cache-Control**

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

Ex: max-age=60

**Set-Cookie**

Contains cookies to send to browser.

Ex: x=10

**Access-Control-Allow-Origin**

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

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

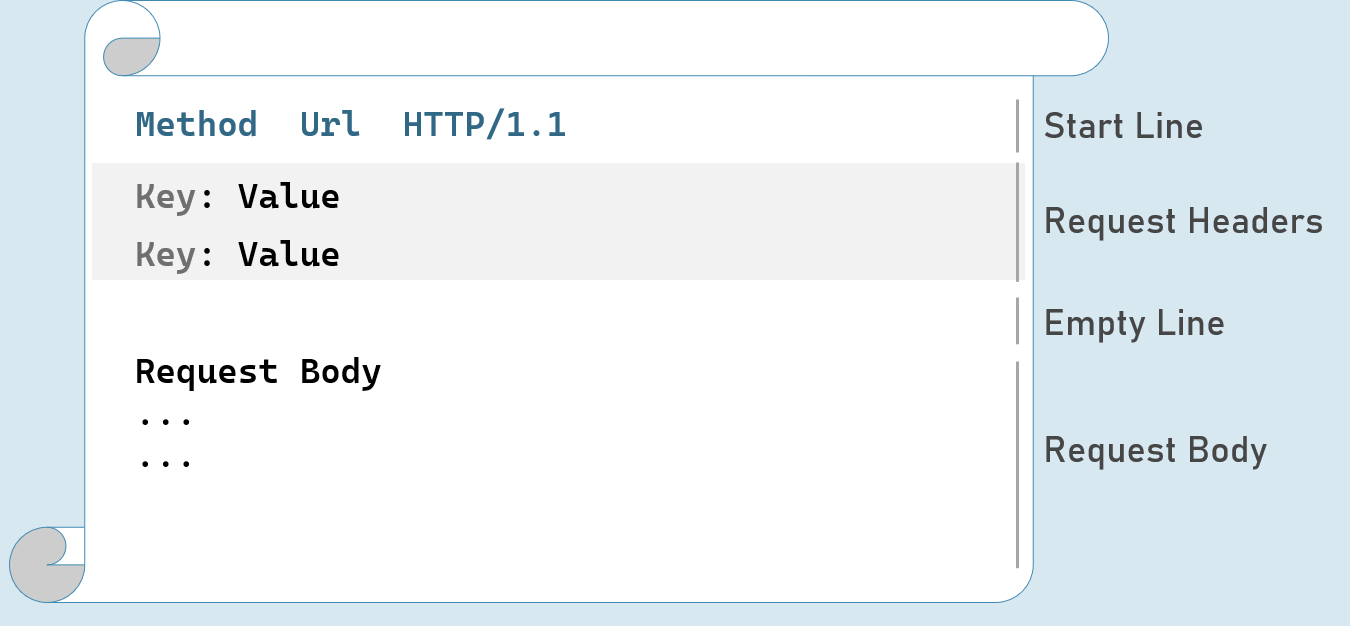
**Location**

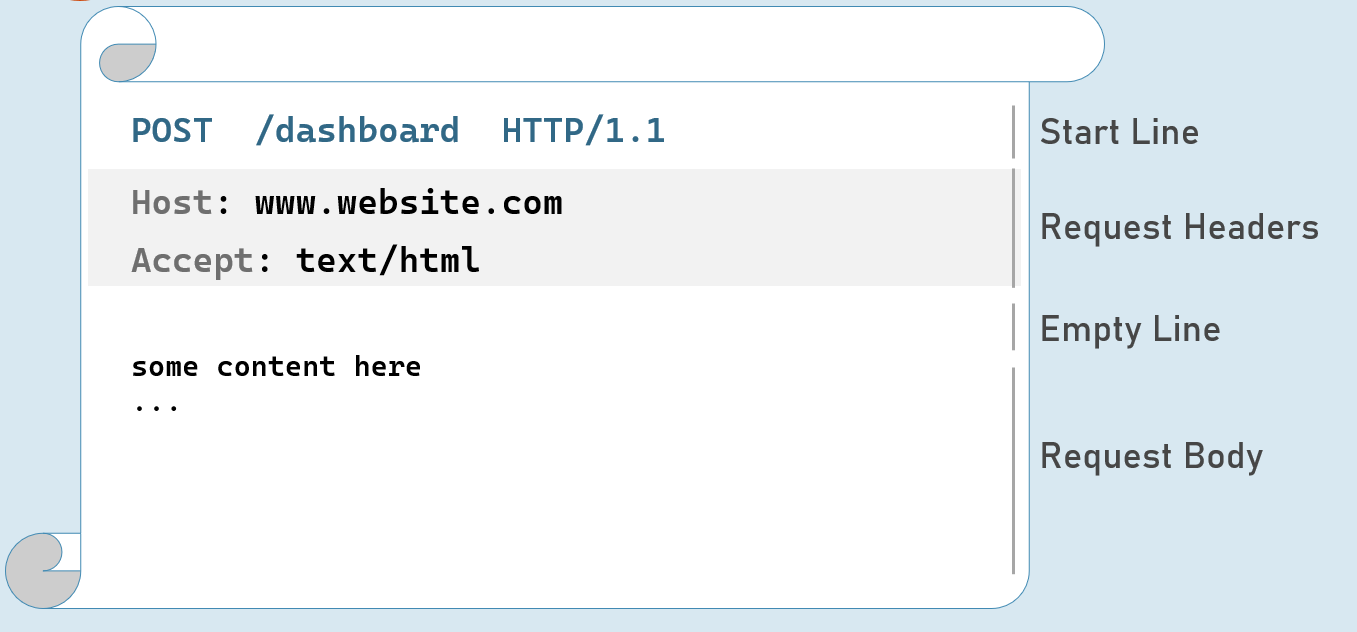
Contains URL to redirect.

Ex: http://www.example-redirect.com

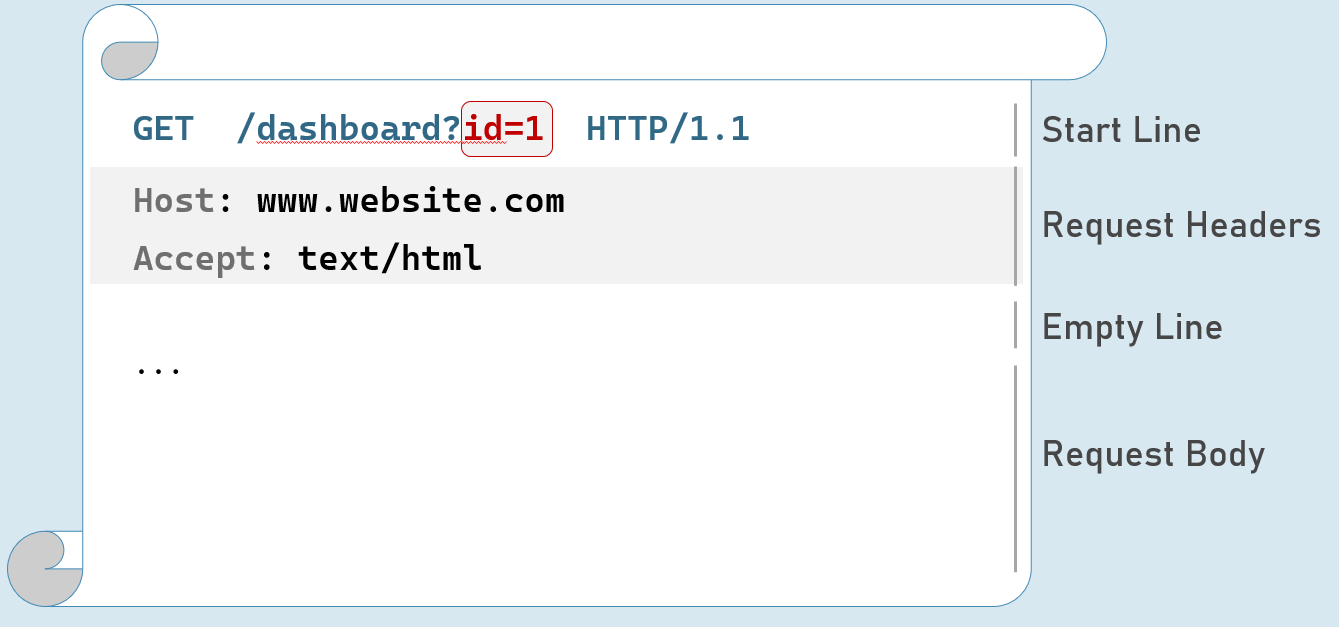
Further reading: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

HTTP Request





**HTTP Request - with Query String**



HTTP Request Headers

**Accept**

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

**Accept-Language**

Represents natural language of response content to be accepted by the client. Ex: 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.

Ex: 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

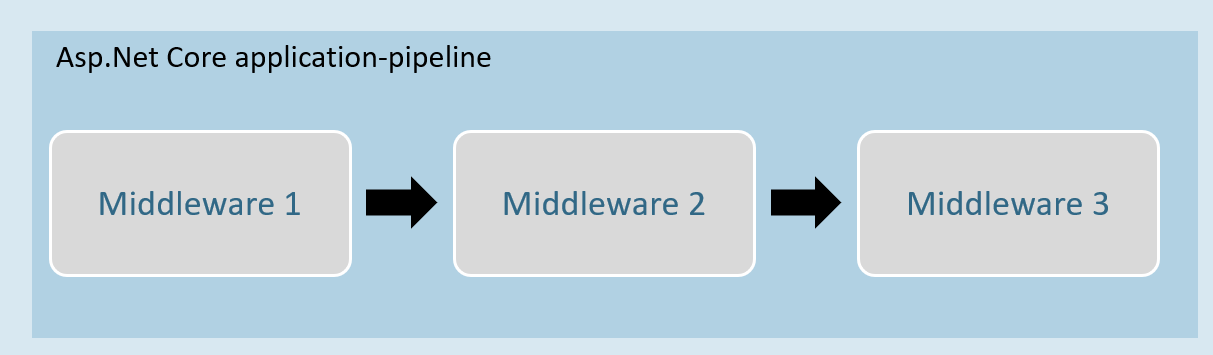
Can't be cached by browsers / search engines.

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( )**

app.Run(async (HttpContext context) =>

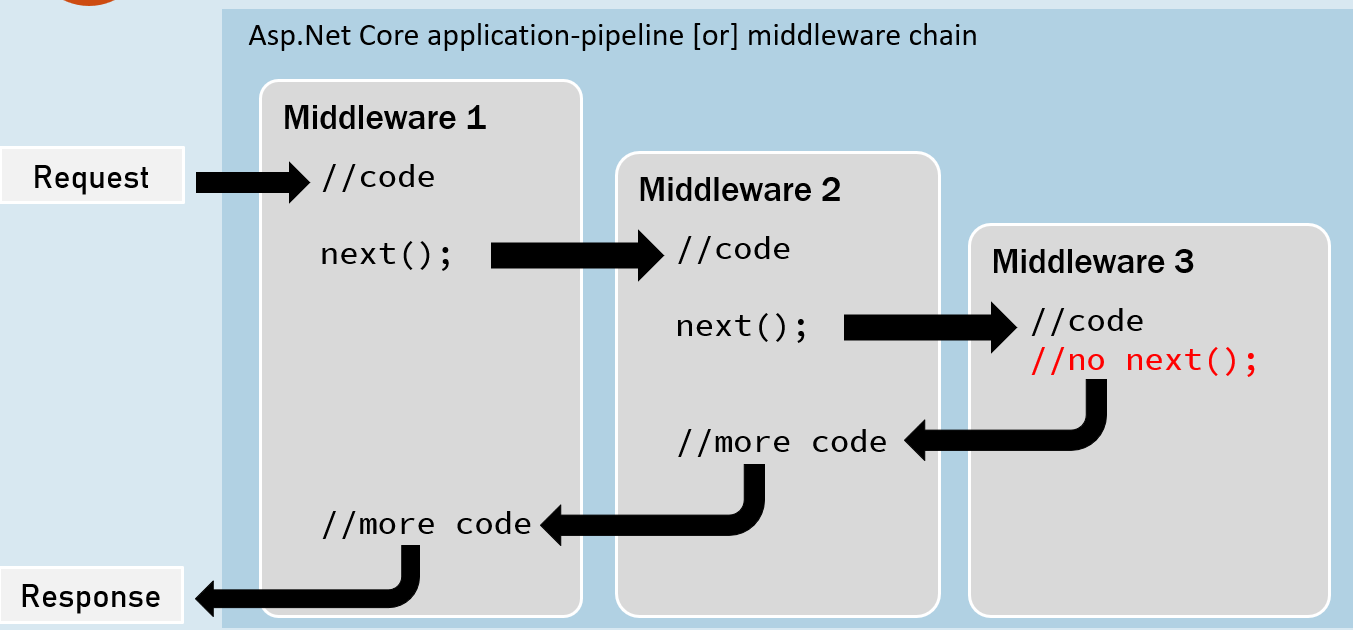
{

//code

});

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( )**

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

{

//before logic

await next(context);

//after logic

});

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.

class MiddlewareClassName : IMiddleware

{

public async Task InvokeAsync(HttpContext context, RequestDelegate next)

{

//before logic

await next(context);

//after logic

}

}

app.UseMiddleware<MiddlewareClassName>();

**Middleware Extensions**

class MiddlewareClassName : IMiddleware

{

public async Task InvokeAsync(HttpContext context,RequestDelegate next)

{

//before logic

await next(context);

//after logic

}

});

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

static class ClassName

{

public static IApplicationBuilder ExtensionMethodName(this IApplicationBuilder app)

{

return app.UseMiddleware<MiddlewareClassName>();

}

}

app.ExtensionMethodName();

**Conventional Middleware**

class MiddlewareClassName

{

private readonly RequestDelegate \_next;

public MiddlewareClassName(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

//before logic

await \_next(context);

//after logic

}

});

static class ClassName

{

public static IApplicationBuilder ExtensionMethodName(this IApplicationBuilder app)

{

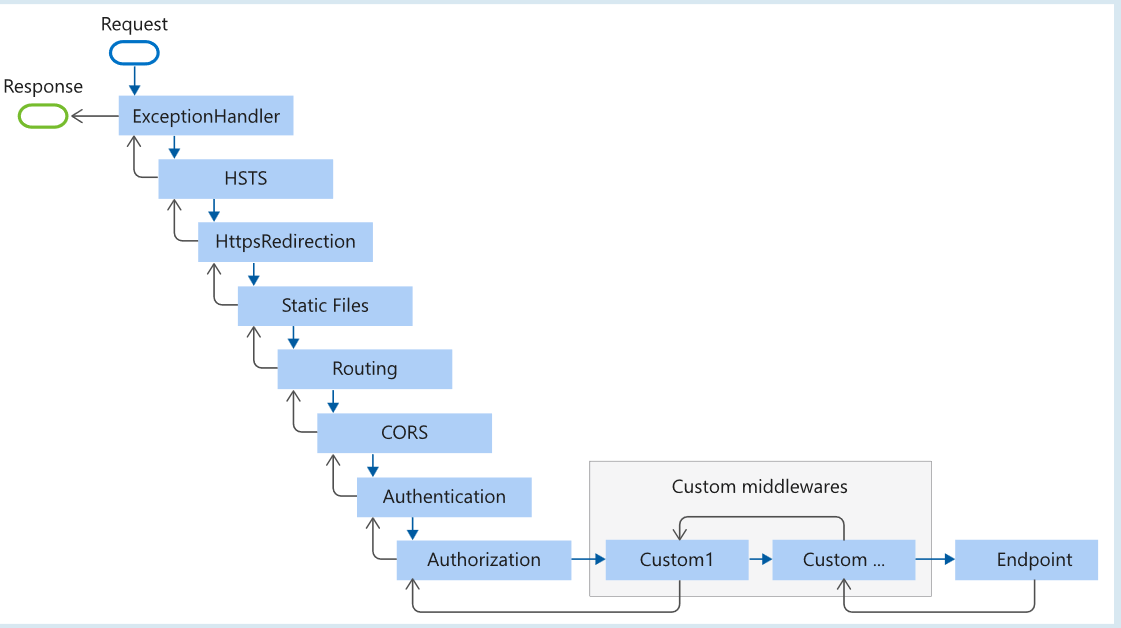
return app.UseMiddleware<MiddlewareClassName>();

}

}

app.ExtensionMethodName();

The Right Order of Middleware



app.UseExceptionHandler("/Error");

app.UseHsts();

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseCors();

app.UseAuthentication();

app.UseAuthorization();

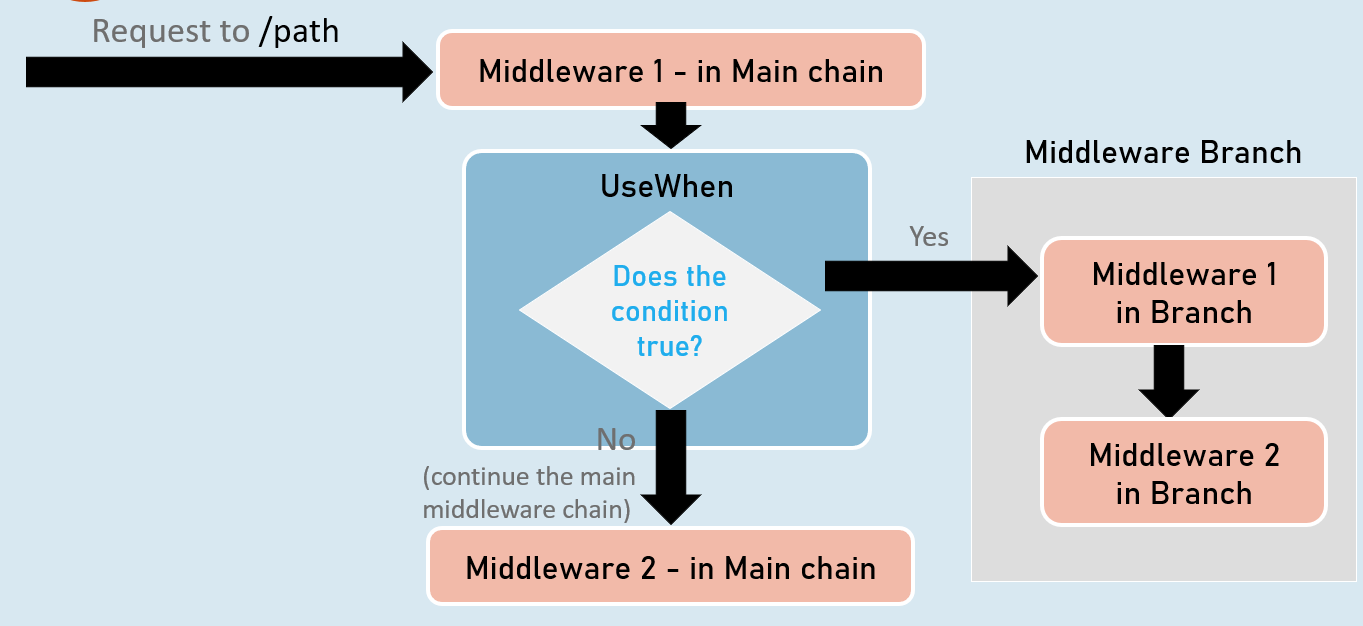
app.UseSession();

app.MapControllers();

//add your custom middlewares

app.Run();

**Middleware - UseWhen**



app.UseWhen( )

app.UseWhen(

context => { return boolean; },

app =>

{

//add your middlewares

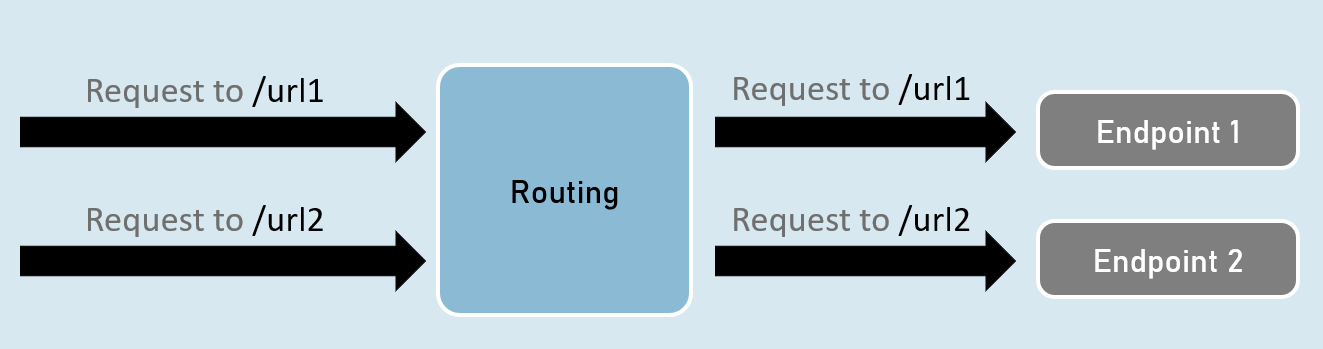
}

);

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

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( )**

app.UseEndPoints(endpoints =>

{

endpoints.Map(…);

endpoints.MapGet(…);

endpoints.MapPost(…);

);

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

**Map, MapGet, MapPost**

**endpoints.Map( )**

endpoints.Map("path", async (HttpContext context) =>

{

//code

});

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

**endpoints.MapGet( )**

endpoints.MapGet("path", async (HttpContext context) =>

{

//code

});

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

**endpoints.MapPost( )**

endpoints.MapPost("path", async (HttpContext context) =>

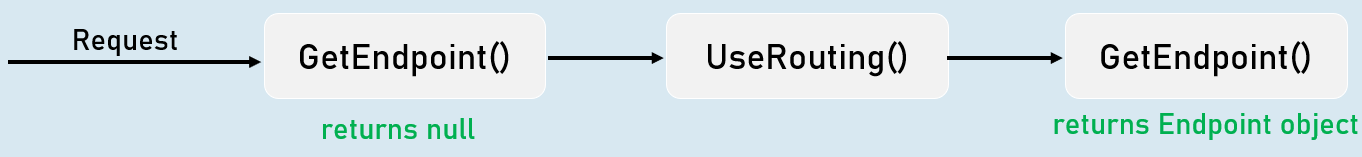
{

//code

});

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.

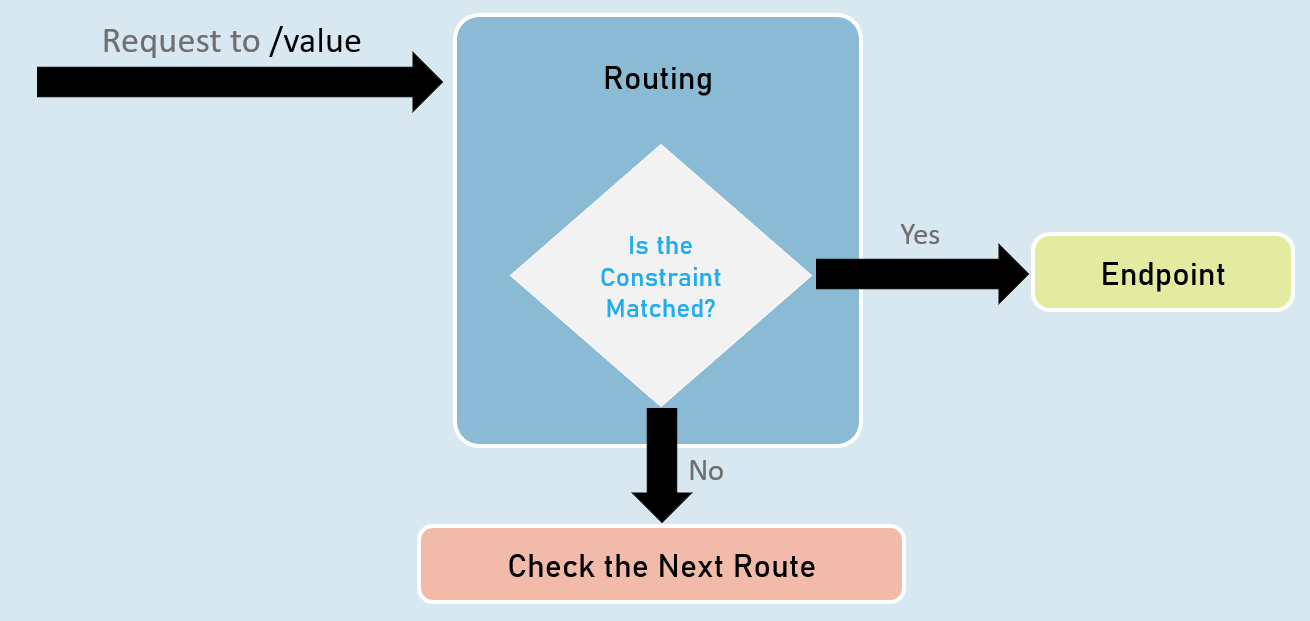
**Optional Route Parameters**

"{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

public class ClassName : IRouteConstraint

{

public bool Match(HttpContext? HttpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)

{

//return true or false

}

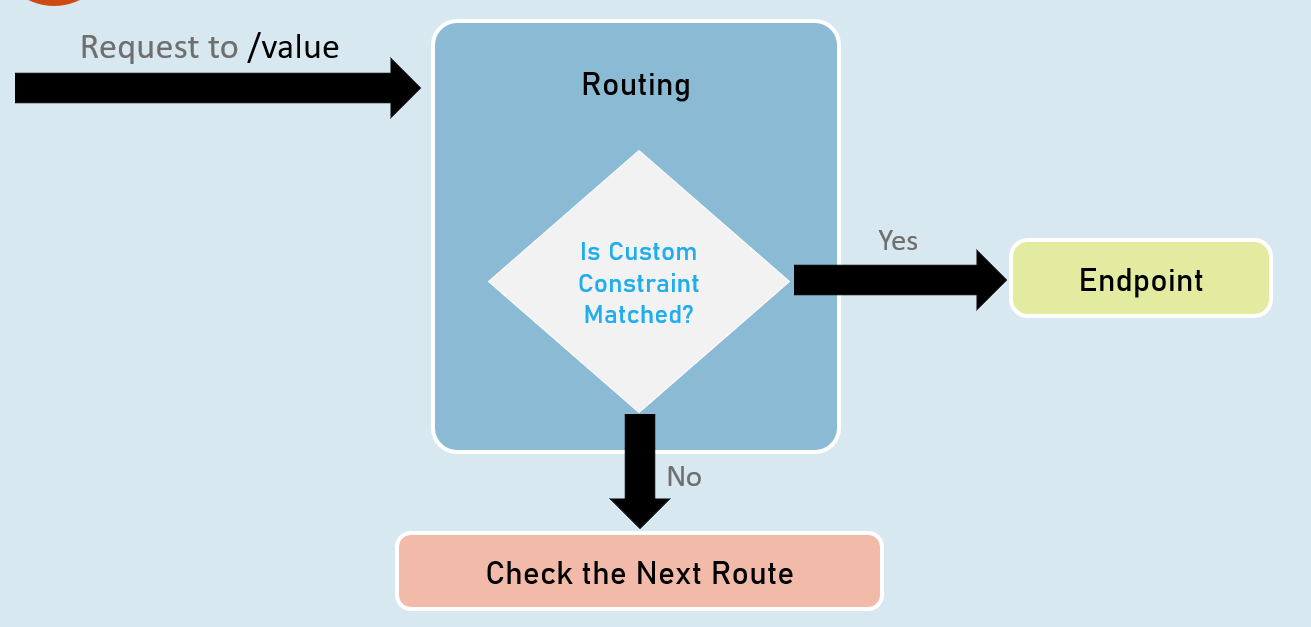
}

builder.Services.AddRouting(options =>

{

options.ConstraintMap.Add("name", typeof(ClassName));

}); //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}".

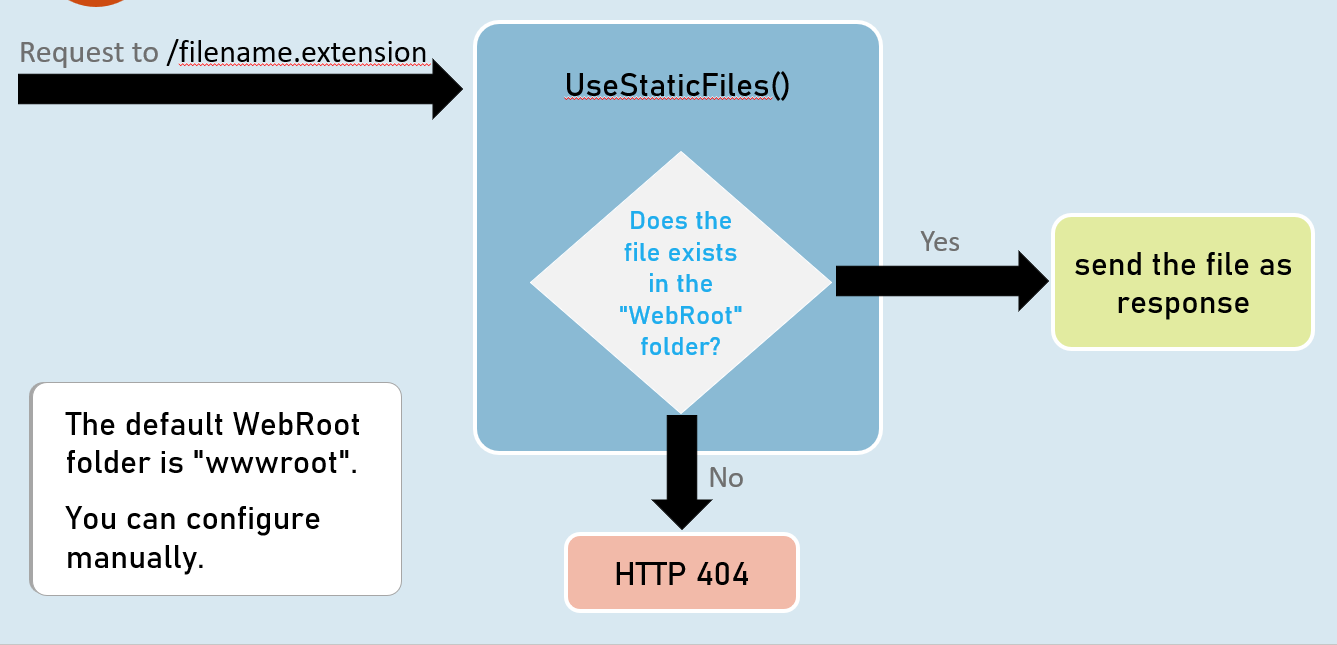
**3:**URL template that has a parameter segment with constraints has more precedence than a parameter segment without constraints.

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

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

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

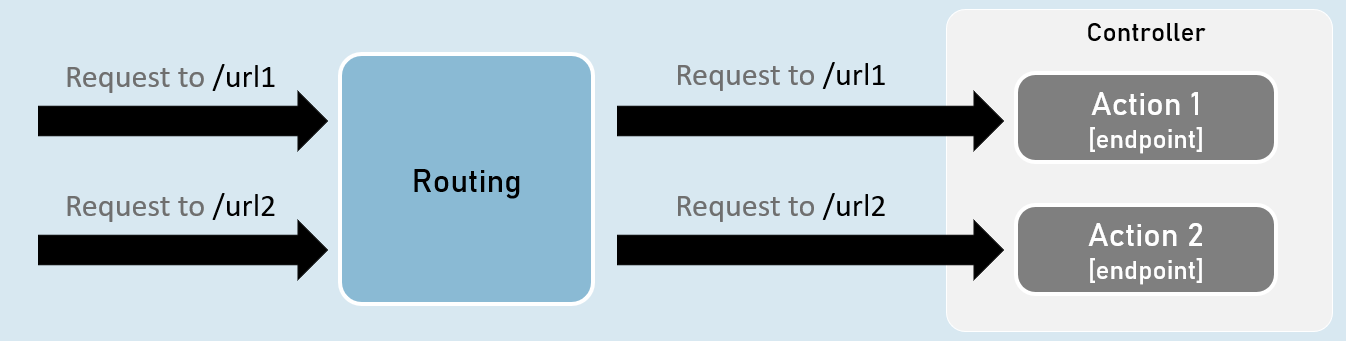
WebRoot



**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**

[Controller]

class ClassNameController

{

//action methods here

}

**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.)

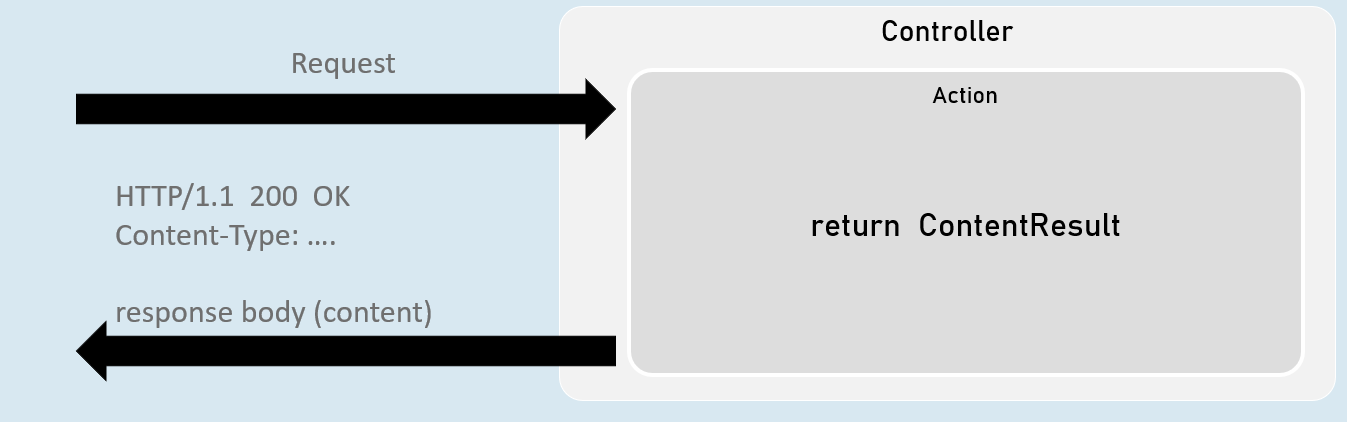
**Preparing Response**

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" };

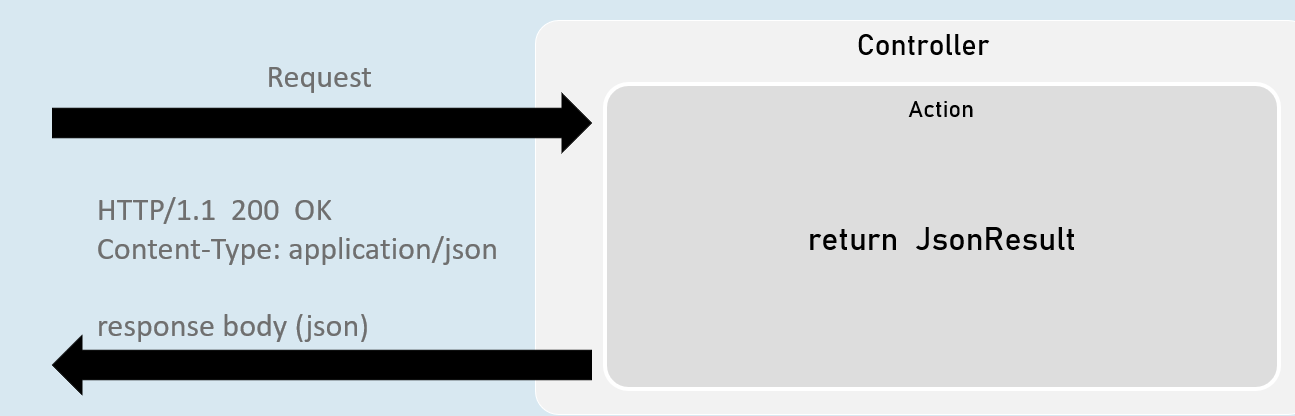
[or]

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);

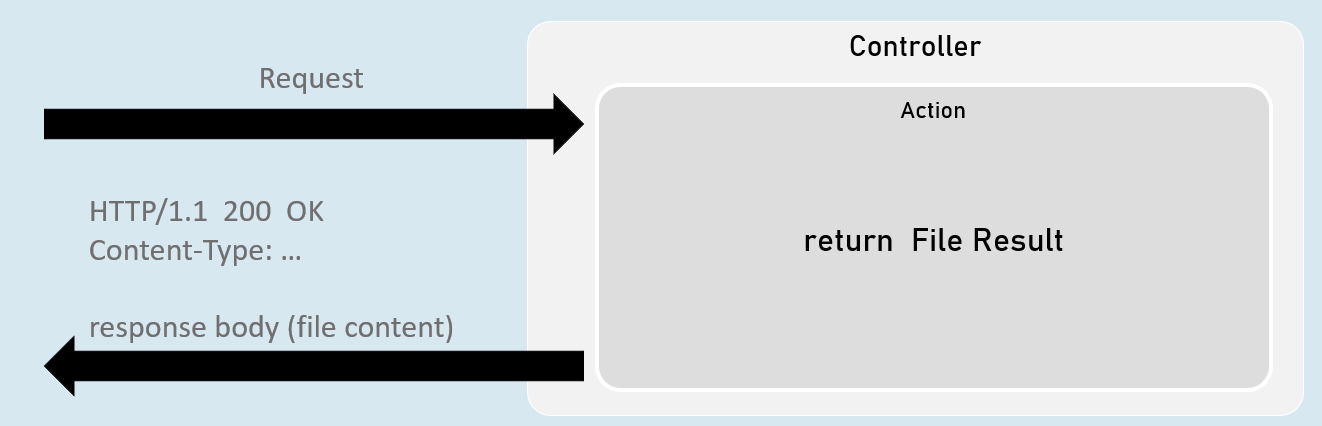
[or]

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");

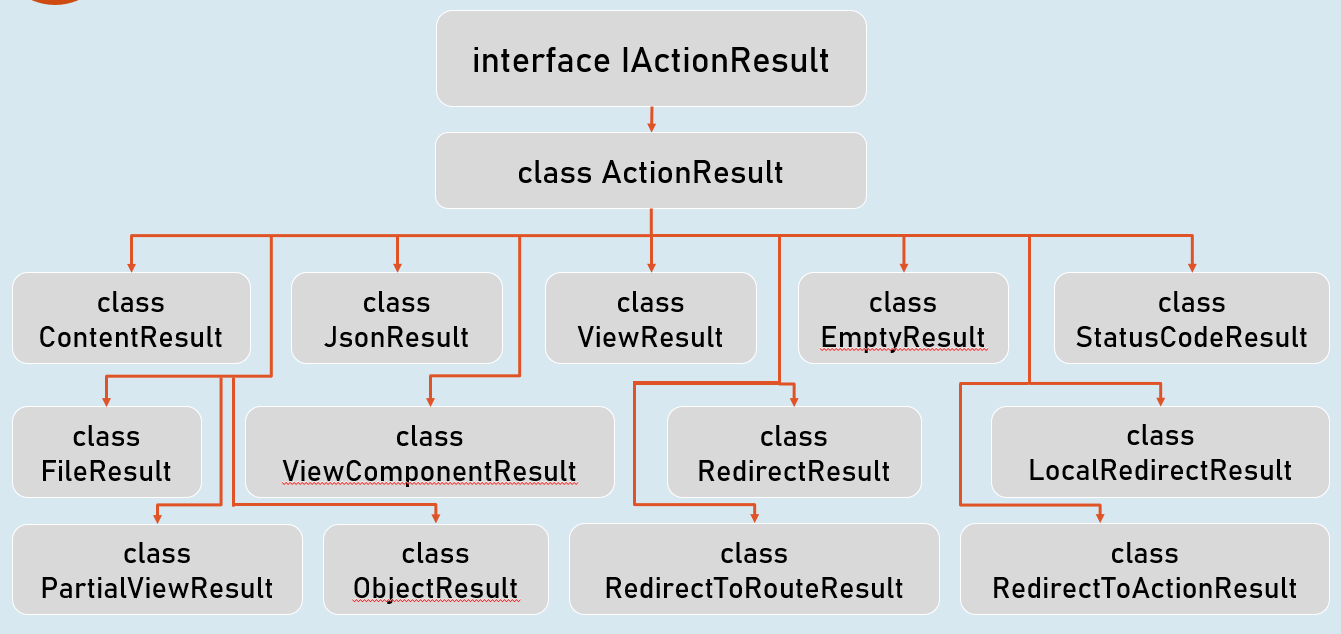
//or

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.

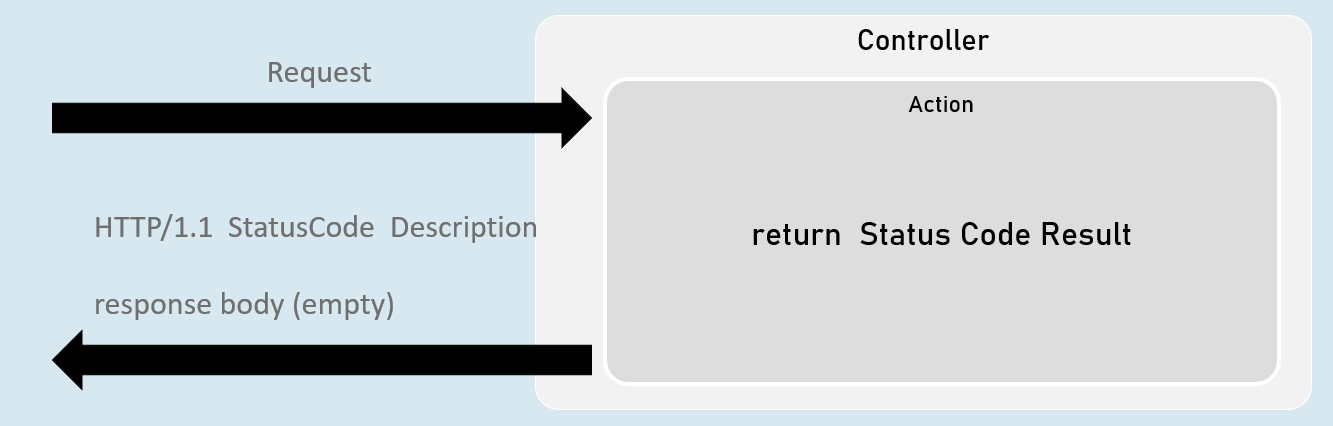
By mentioning the return type as IActionResult, you can return either of the subtypes of IActionResult



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();

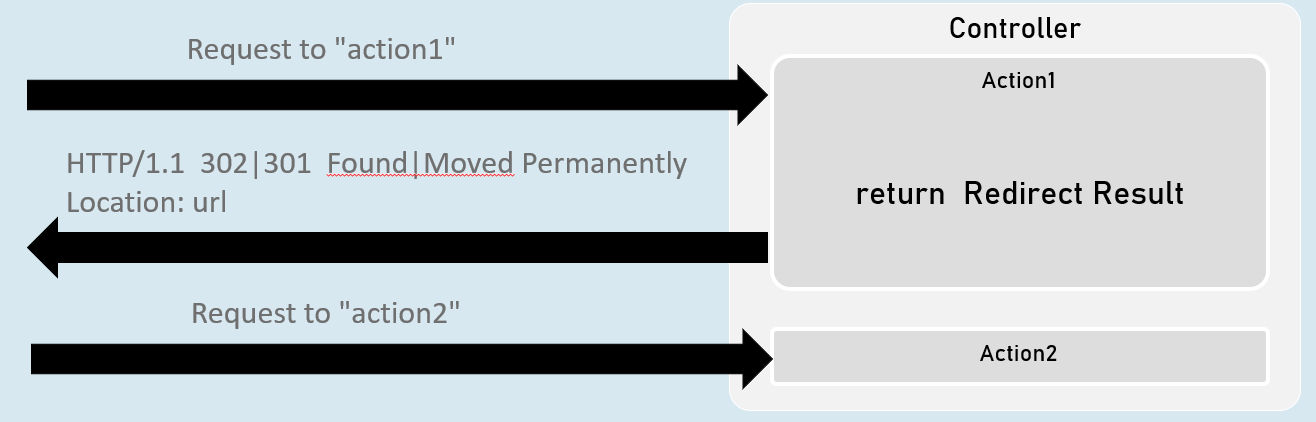
//or

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

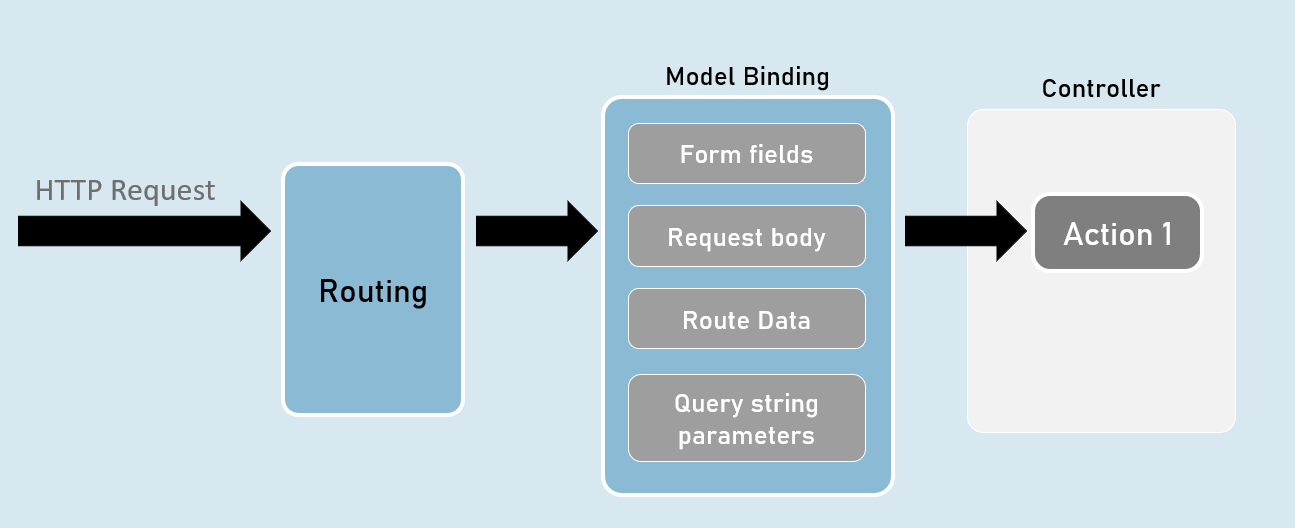
return RedirectPermanent("url");

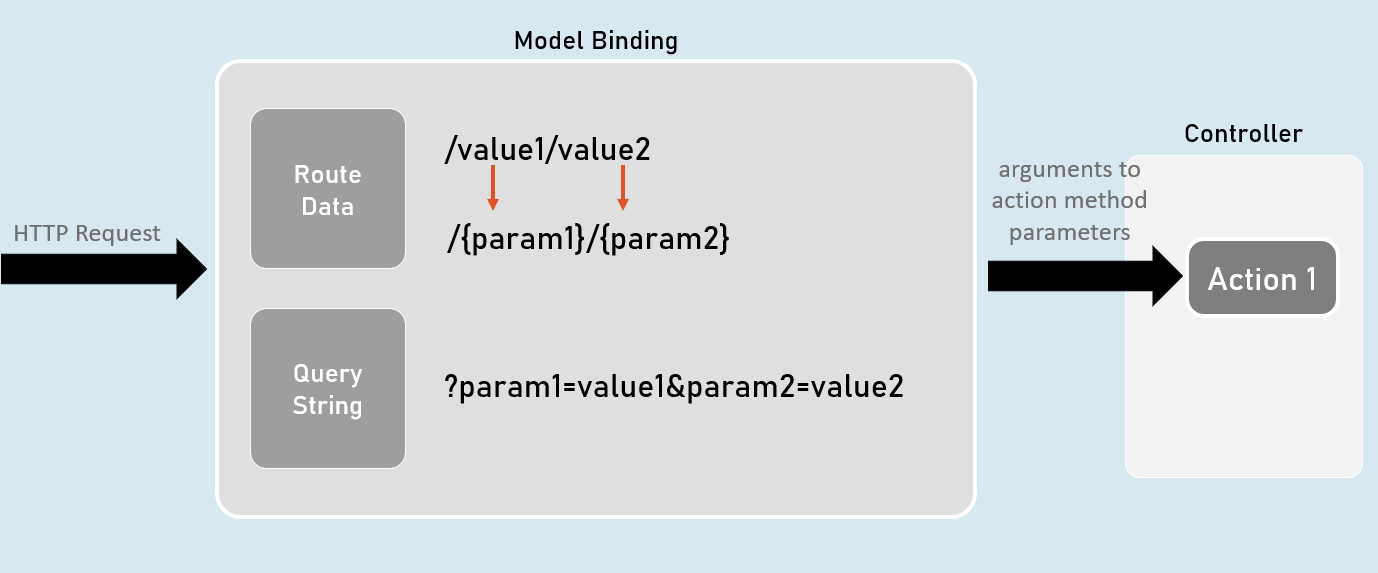
Model Binding

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

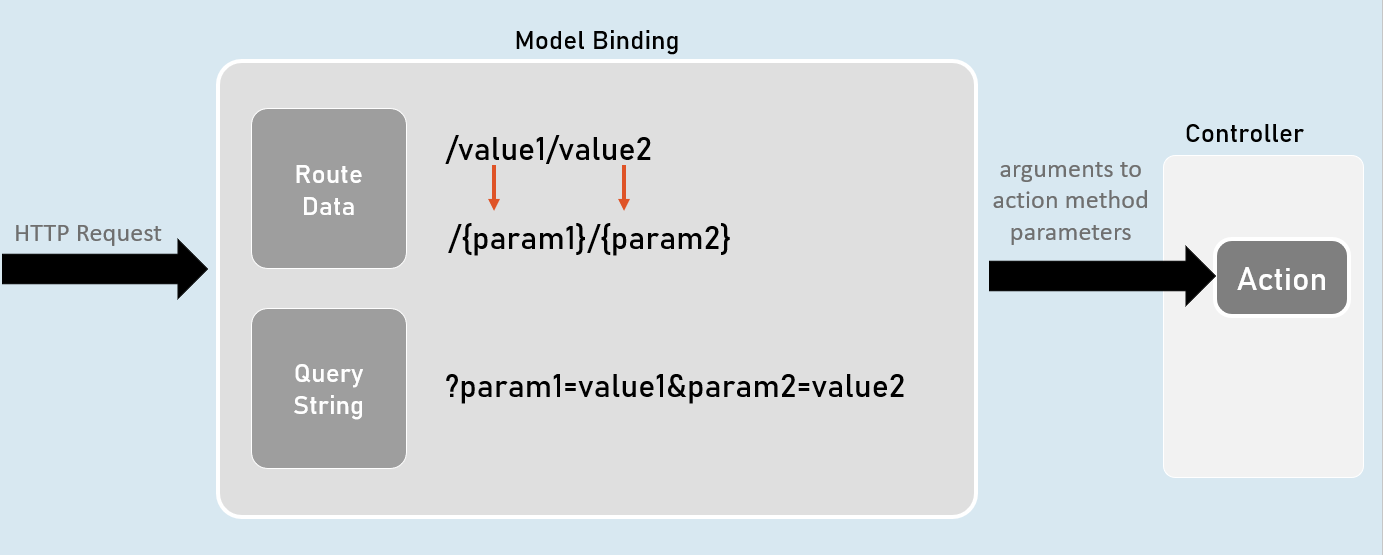


QueryString vs RouteData





[FromQuery] and [FromRoute]



[FromQuery]

//gets the value from query string only

public IActionResult ActionMethodName( [FromQuery] type parameter)

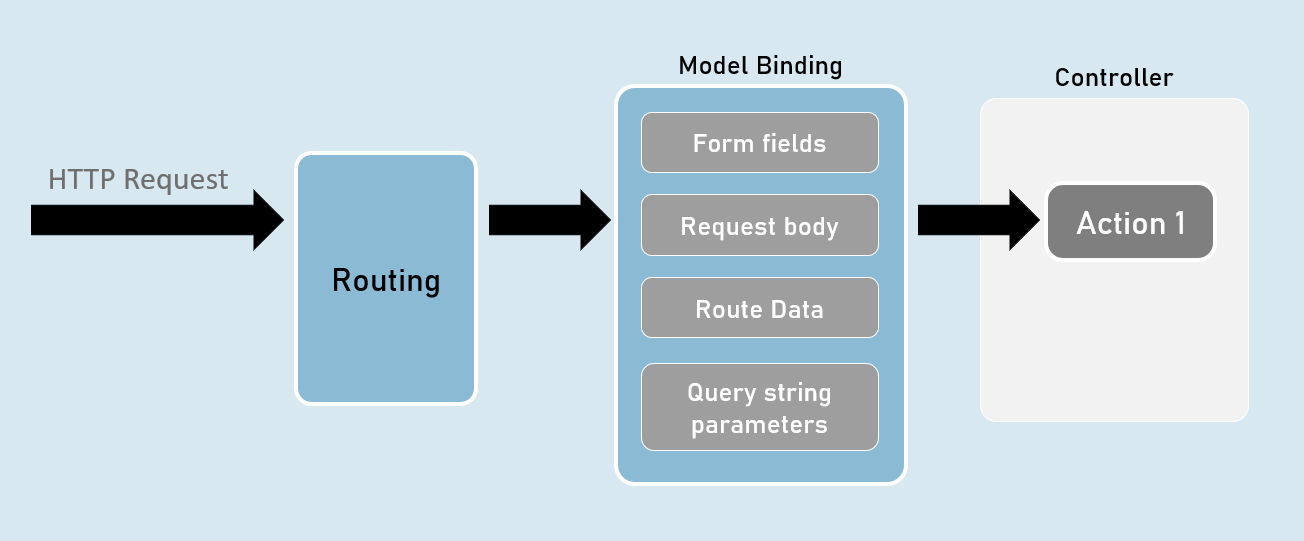
{}

[FromRoute]

//gets the value from route parameters only

public IActionResult ActionMethodName( [FromRoute] type parameter)

{}



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

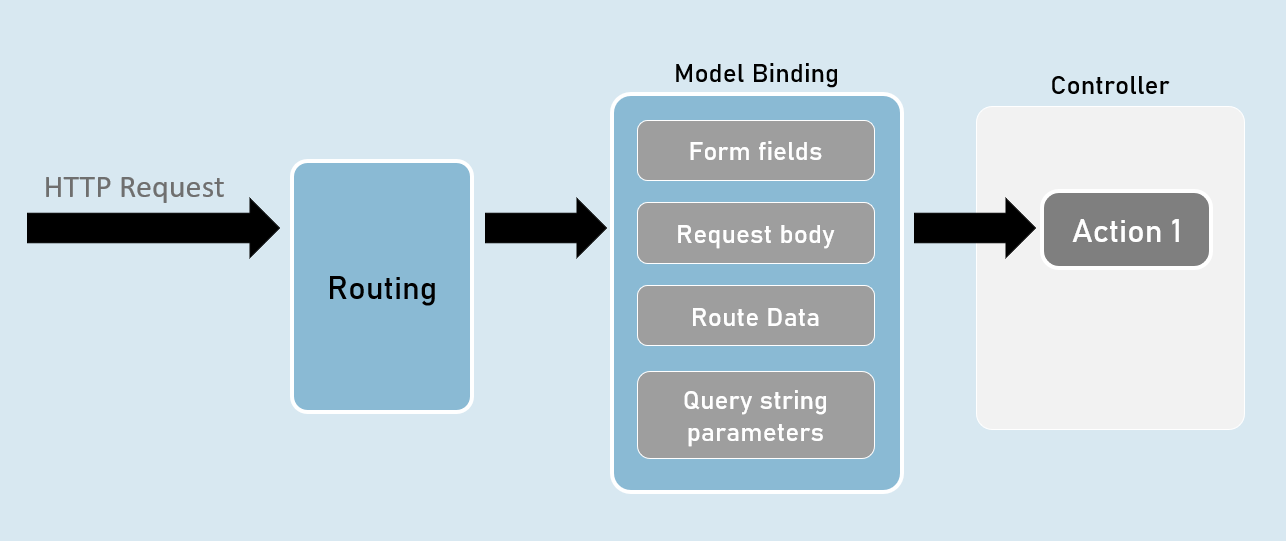
class ClassName

{

public type PropertyName { get; set; }

}

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

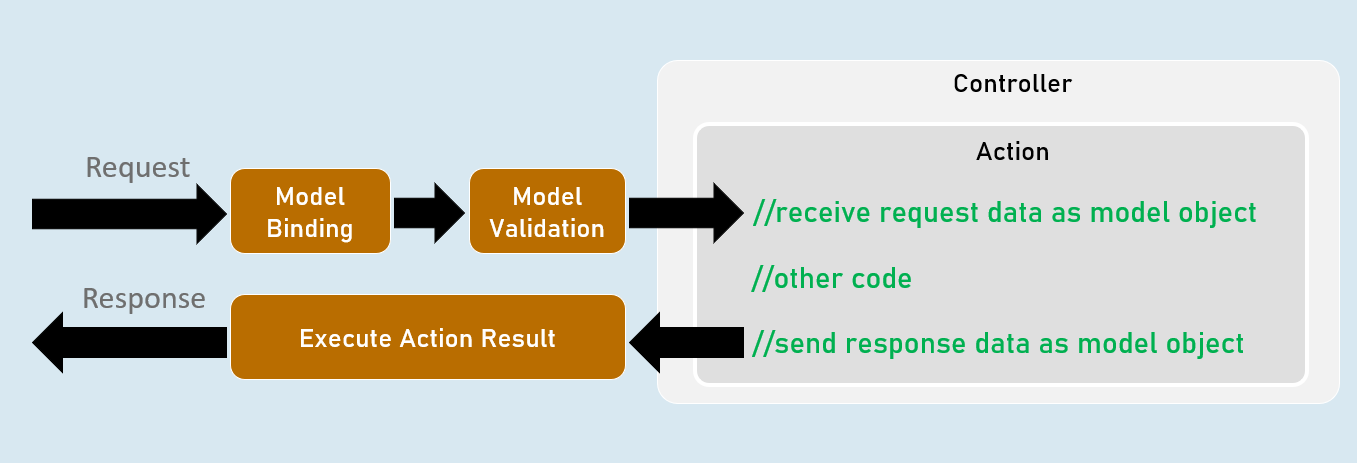
class ClassName

{

[Attribute] //applies validation rule on this property

public type PropertyName { get; set; }

}



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

class ClassName : ValidationAttribute

{

public override ValidationResult? IsValid(object? value, ValidationContext validationContext)

{

//return ValidationResult.Success;

//[or] return new ValidationResult("error message");

}

}

ValidationAttribute

Base class for all validation attributes such as RequiredAttribute, RegularExpressionAttribute, RangeAttribute, StringLengthAttribute, CompareAttribute etc.

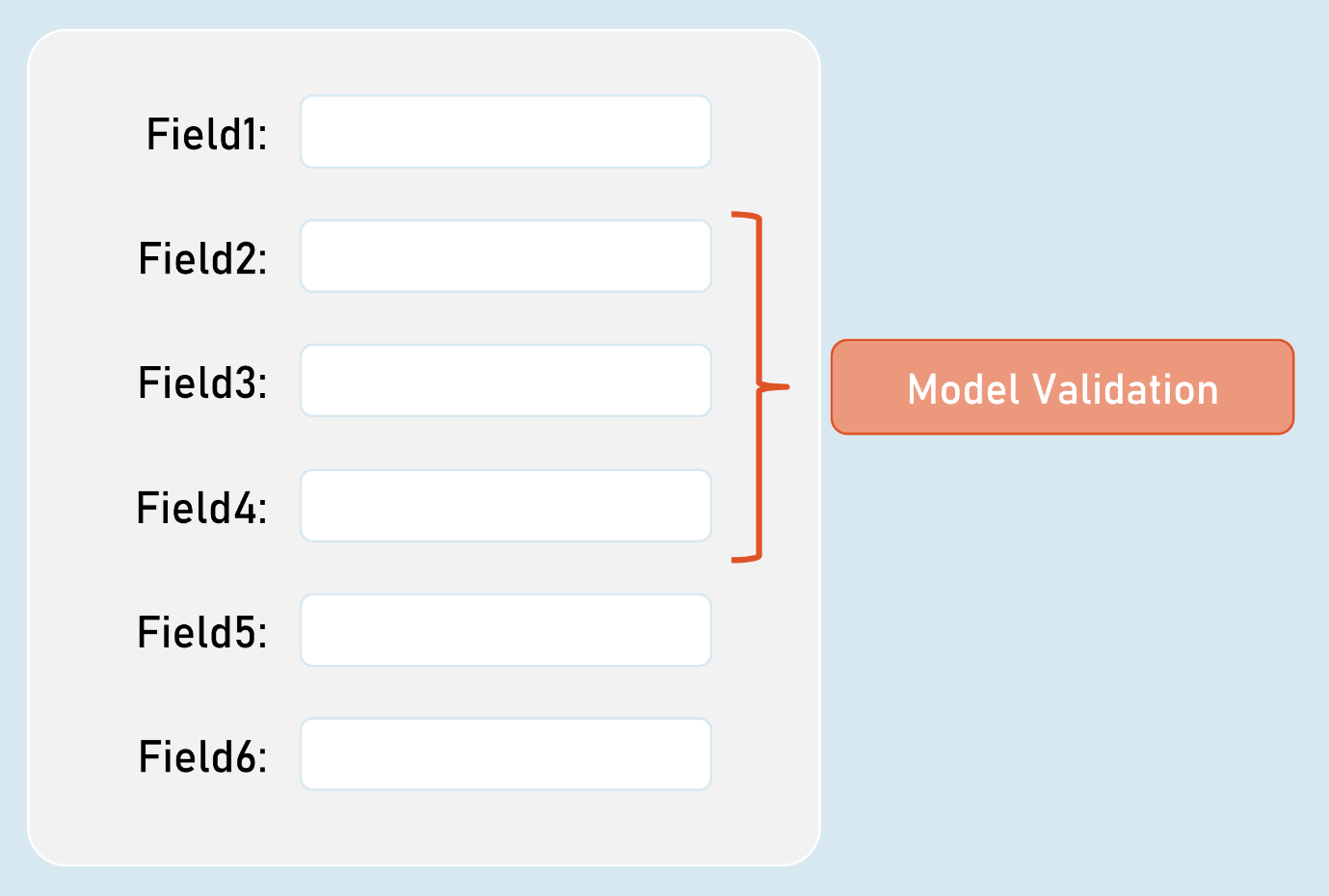
Provides properties such as ErrorMessage & methods such as Validate(), IsValid() etc.

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

class ClassName : IValidatableObject

{

//model properties here

public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)

{

if (condition)

{

yield return new ValidationResult("error message");

}

}

}

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]

class ClassNameController

{

public IActionResult ActionMethodName( [Bind(nameof(ClassName.PropertyName), nameof(ClassName.PropertyName) )] ClassName parameterName)

{

}

}

[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]

class ModelClassName

{

[BindNever]

public type PropertyName { get; set; }

}

[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.

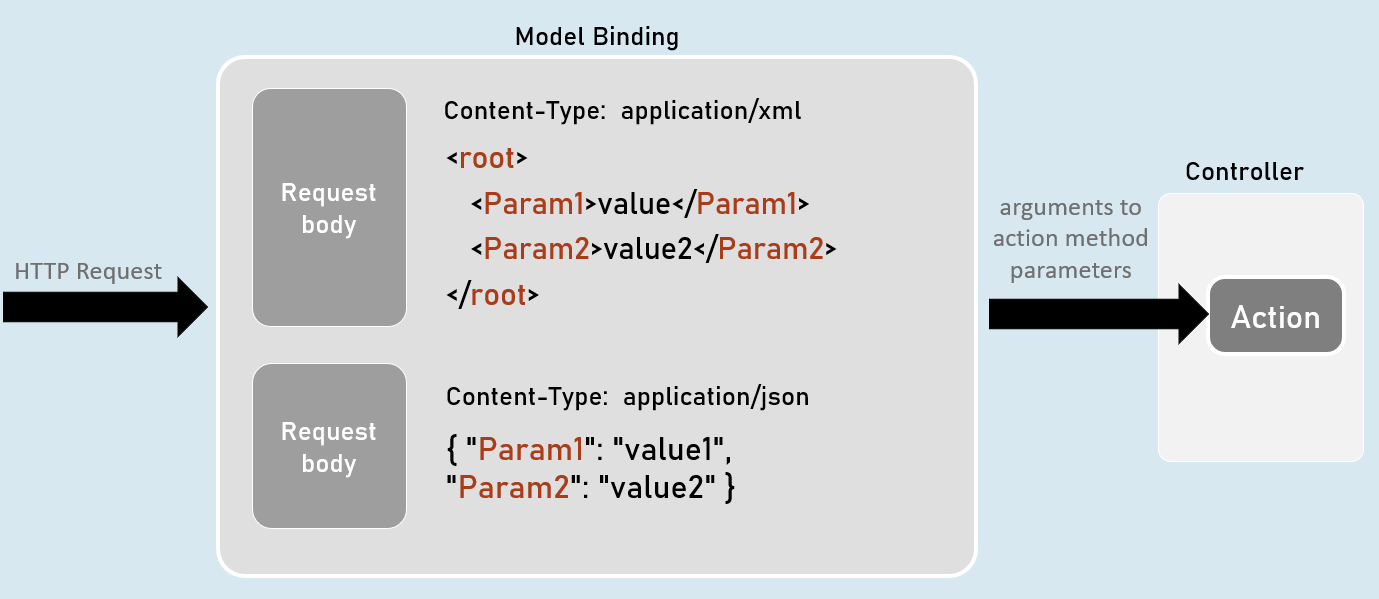
[FromBody]

//enables the input formatters to read data from request body (as JSON or XML or custom) only

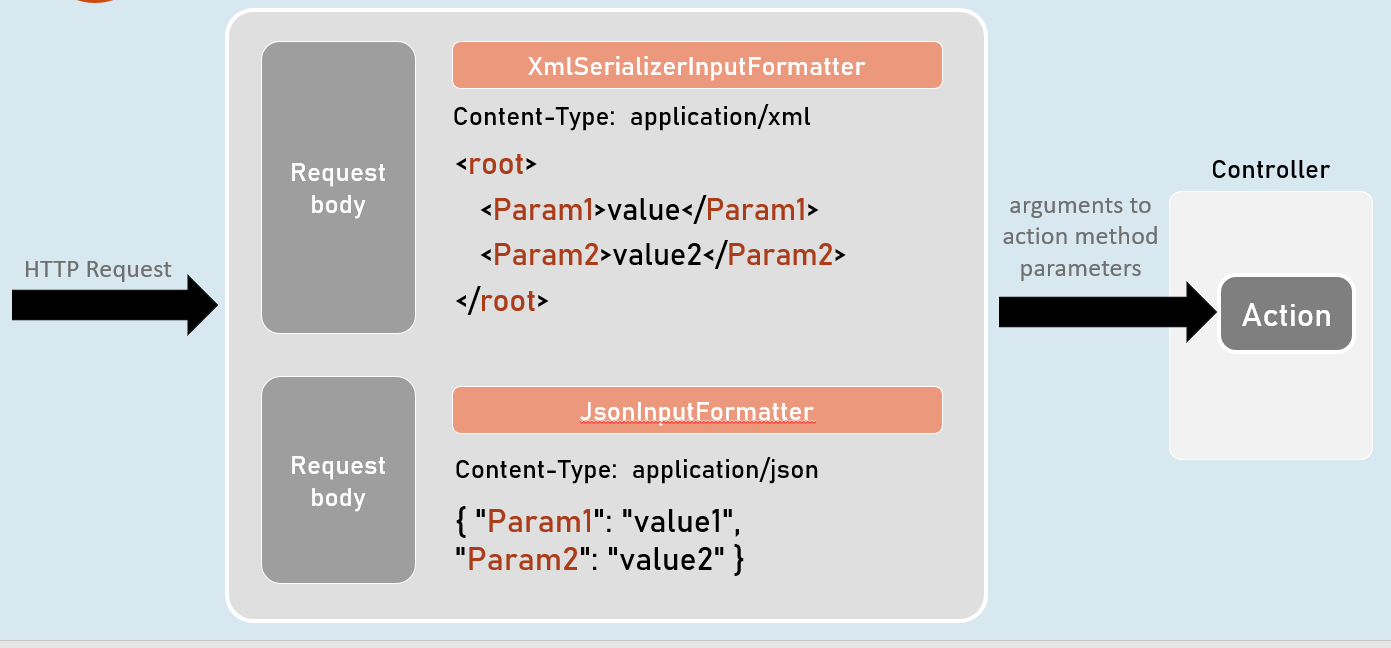
public IActionResult ActionMethodName( [FromBody] type parameter)

{

}



Input Formatters



Custom Model Binders

Custom Model Binder

class ClassName : IModelBinder

{

public Task BindModelAsync(ModelBindingContext bindingContext)

{

//gets value from request

bindingContext.ValueProvider.GetValue("FirstName");

//returns model object after reading data from the request

bindingContext.Result = ModelBindingResult.Success(your\_object);

}

}

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

class ClassName : IModelBinderProvider

{

public IModelBinder GetBinder(ModelBinderProviderContext providerContext)

{

//returns type of custom model binder class to be invoked

return new BinderTypeModelBinder(typeof(YourModelBinderClassName));

}

}

IModelBinderProvider

Base interface for all custom model binder providers.

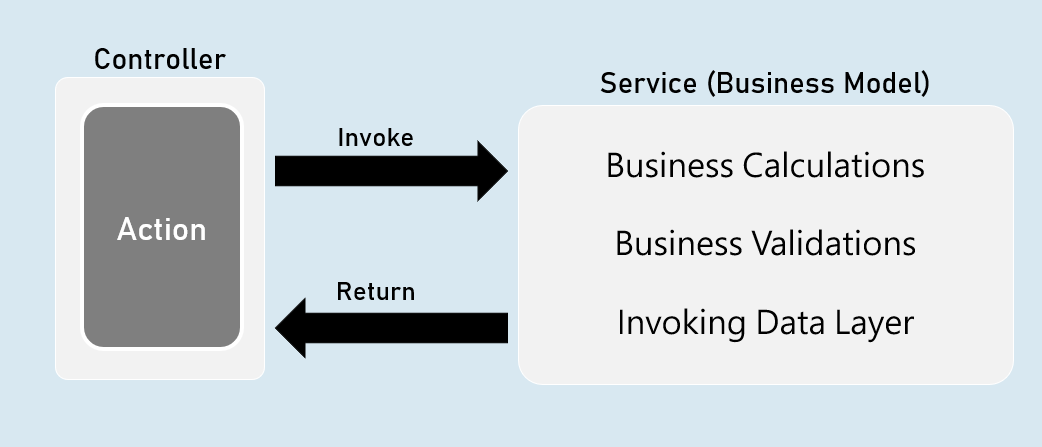
Provides a method called GetBinder, to return the type of custom model binder class.

ModelBinderProviderContext

Acts as a parameter for "GetBinder()" method of custom model binder provider classes.

Provides properties such as BindingInfo, Services etc.

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.

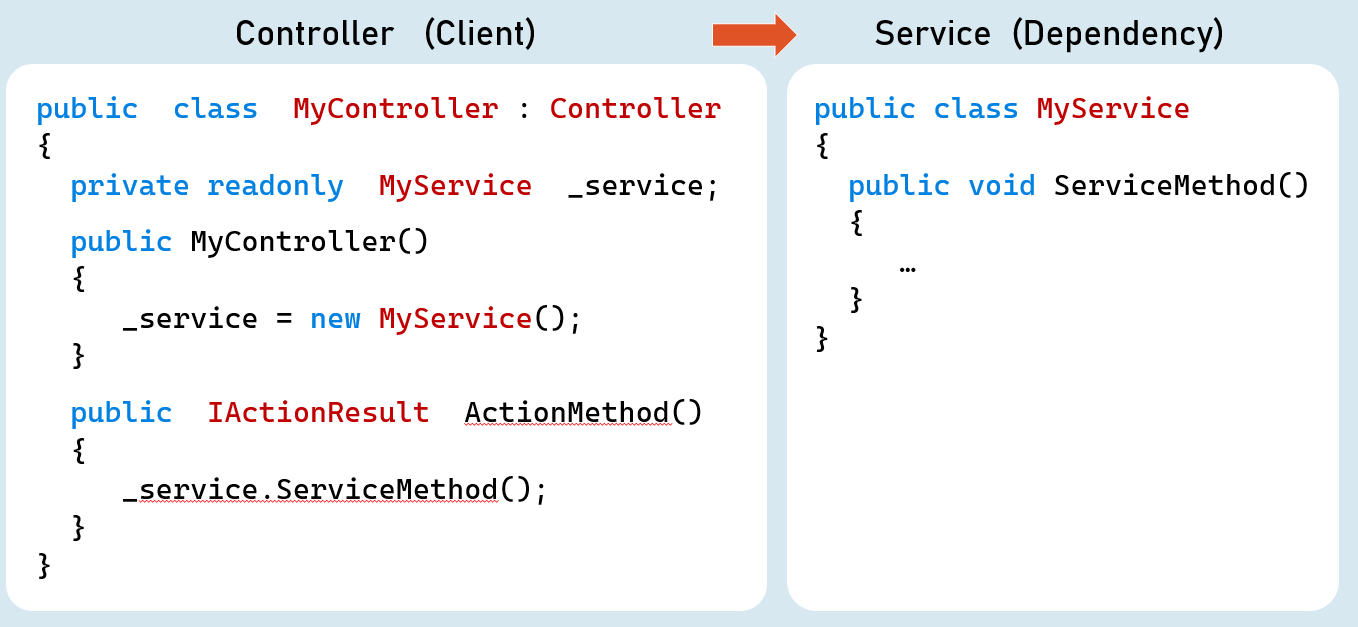
Service is an abstraction layer (middle layer) between presentation layer (or application layer) and data layer.

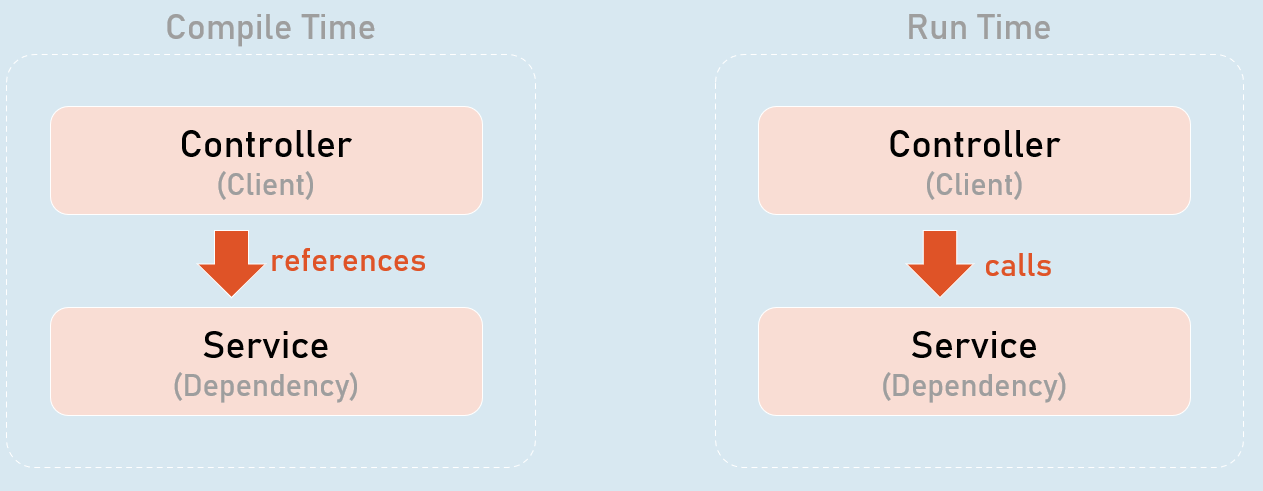
It makes the business logic separated from presentation layer and data layer.

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

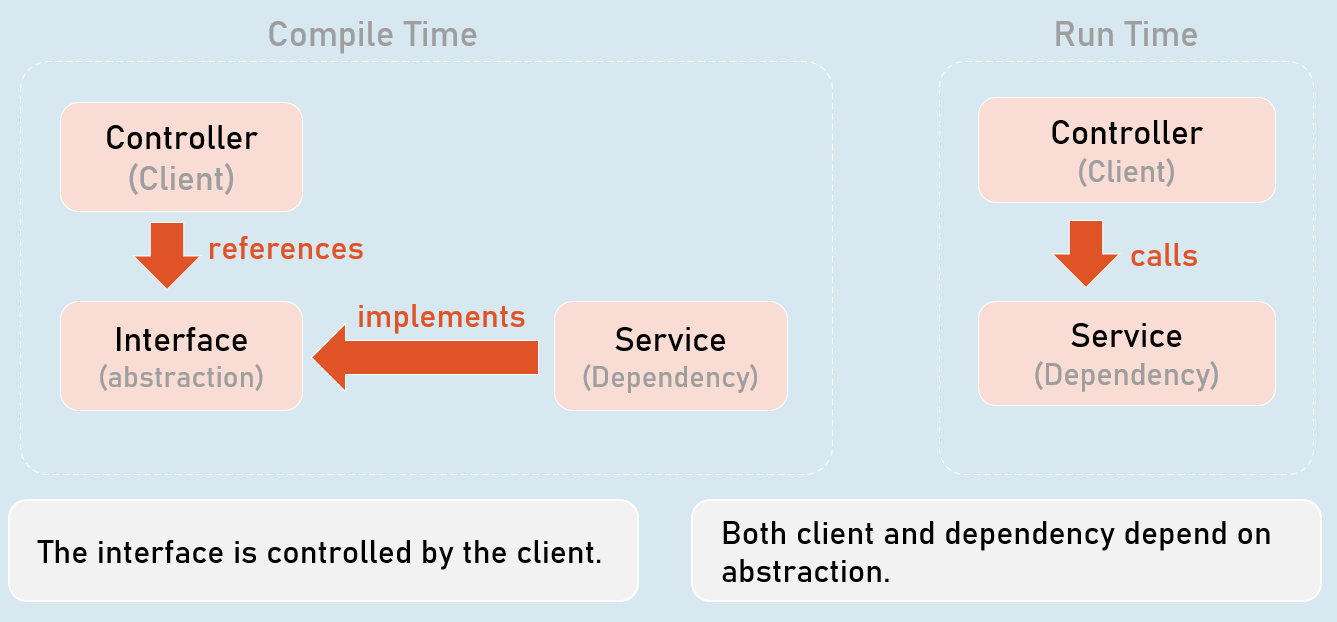
Dependency Inversion Principle (DIP) is a design principle (guideline), which is a solution for the dependency problem.

"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.



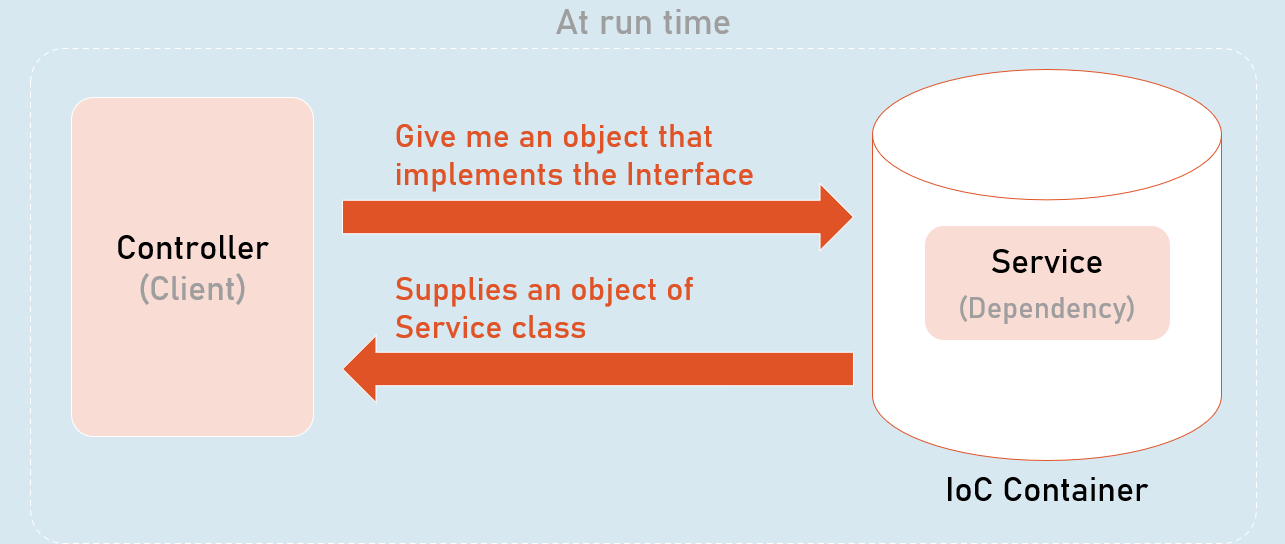
Inversion of Control (IoC)

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).

builder.Services.Add(

new ServiceDescriptor(

typeof (Interface),

typeof (Service)

ServiceLifetime.LifeTime //Transient, Scoped, Singleton

)

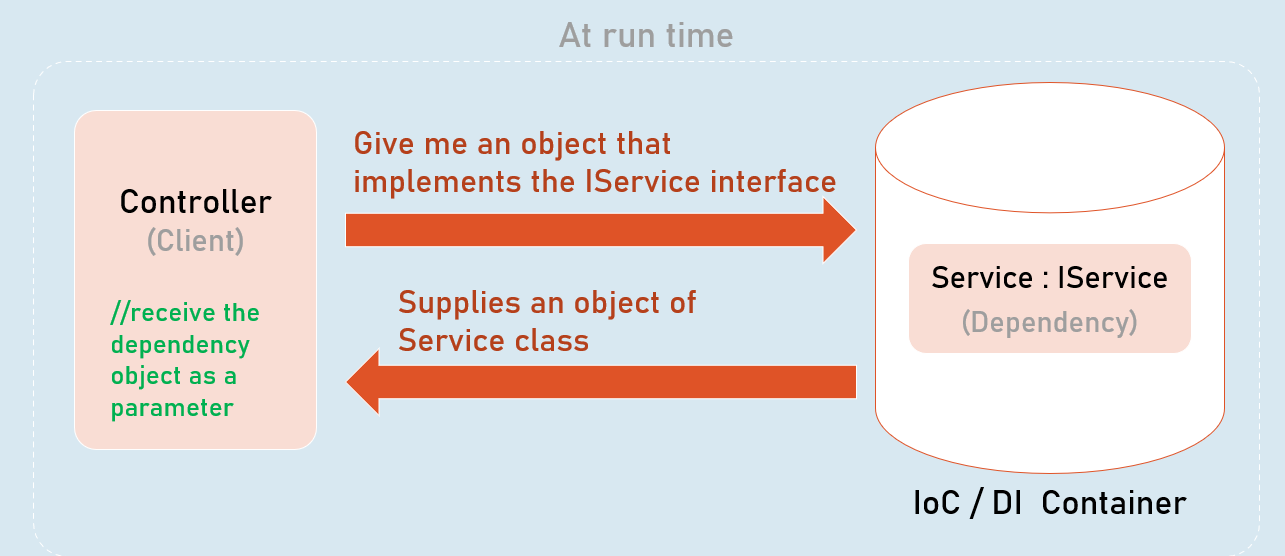
);

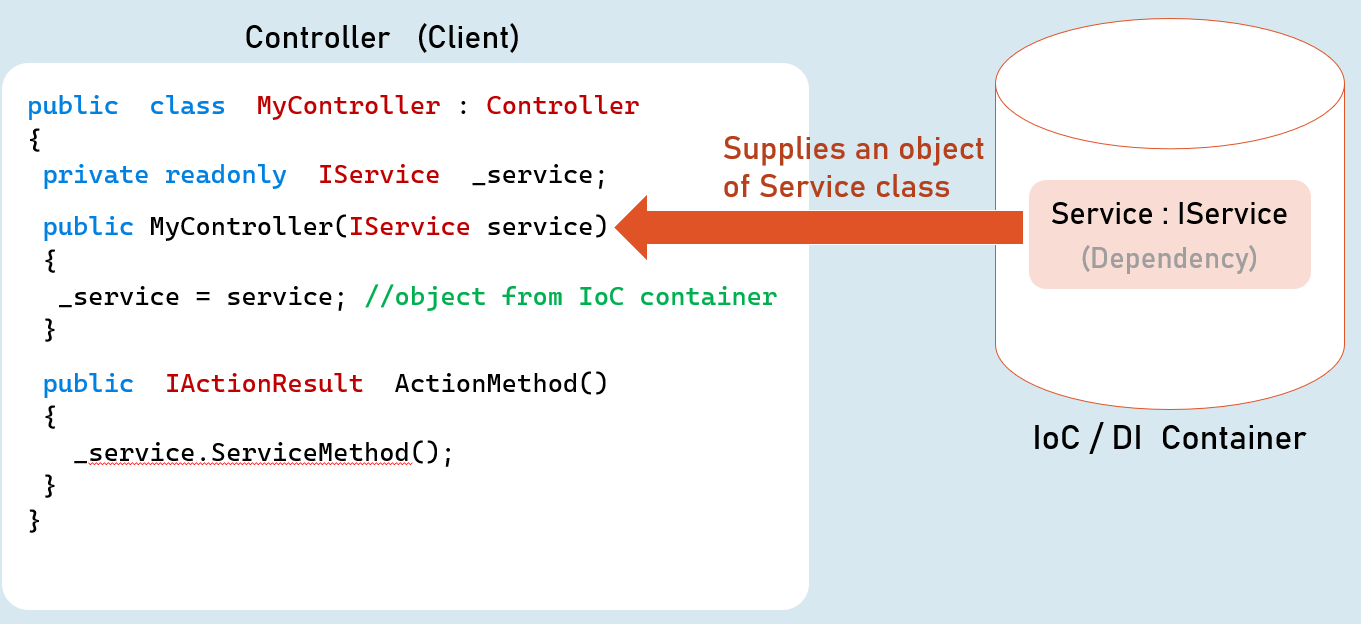
Dependency Injection (DI)

Dependency injection (DI) is a design pattern, which is a technique for achieving "Inversion of Control (IoC)" between clients and their dependencies.

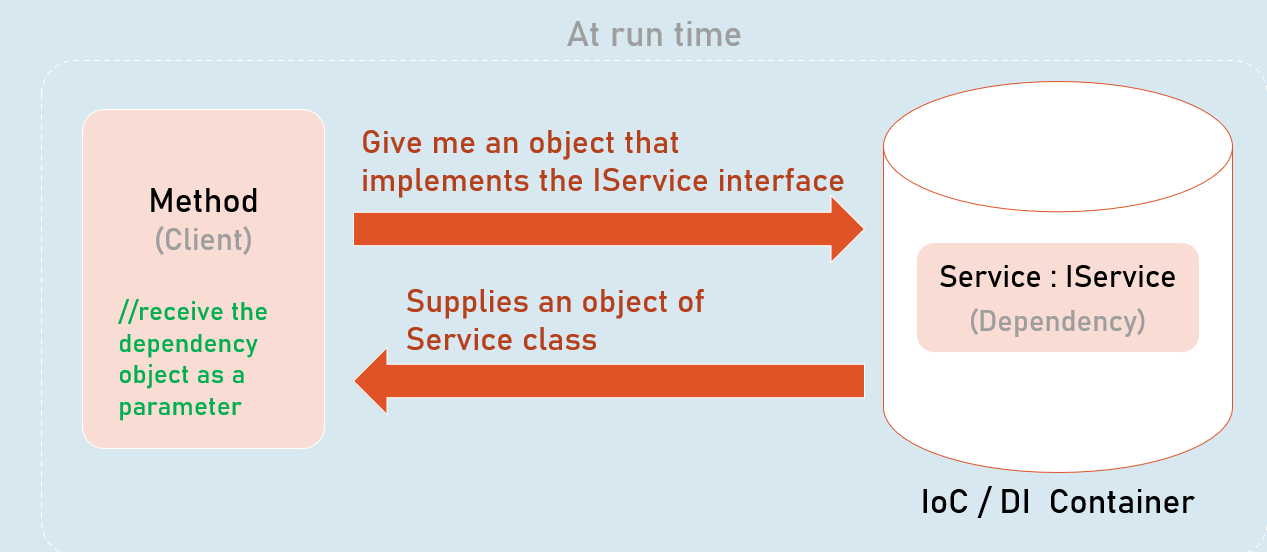
It allows you to inject (supply) a concrete implementation object of a low-level component into a high-level component.

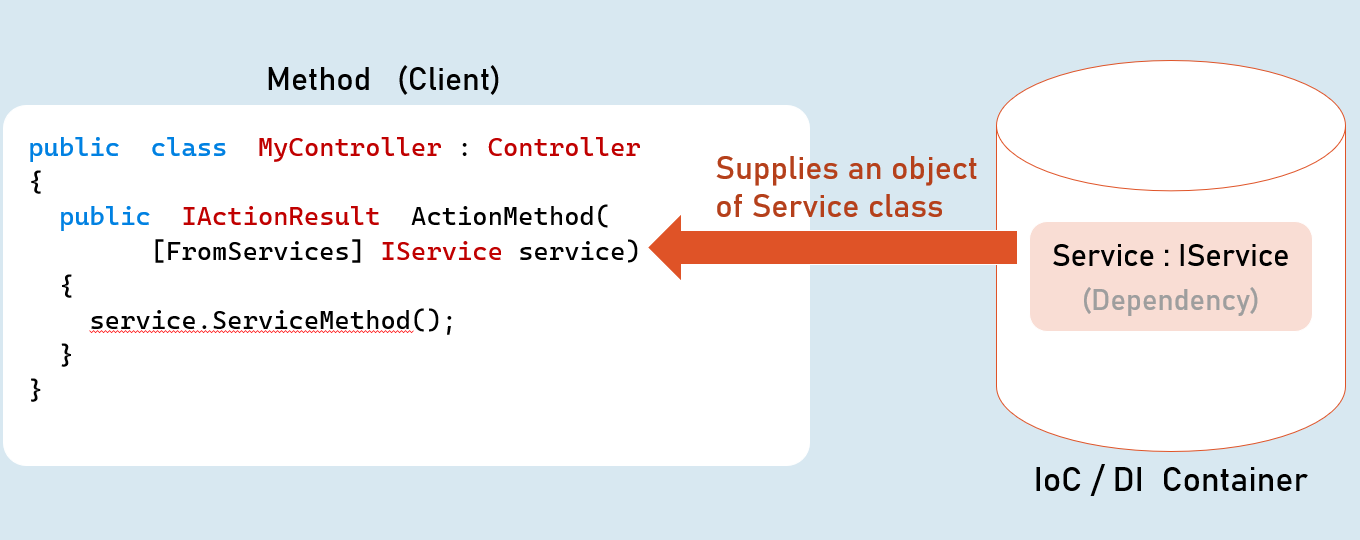
The client class receives the dependency object as a parameter either in the constructor or in a method.





Method Injection





Service Lifetime

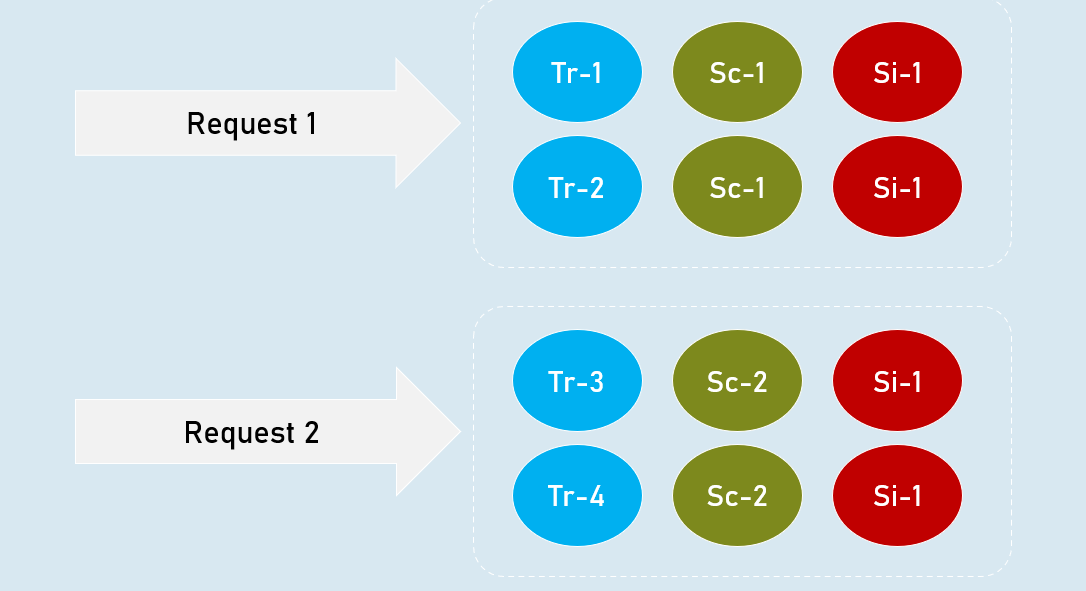
(Transient, Scoped, Singleton)

A service lifetime indicates when a new object of the service has to be created by the IoC / DI container.

**Transient:**Per injection

**Scoped:**Per scope (browser request)

**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

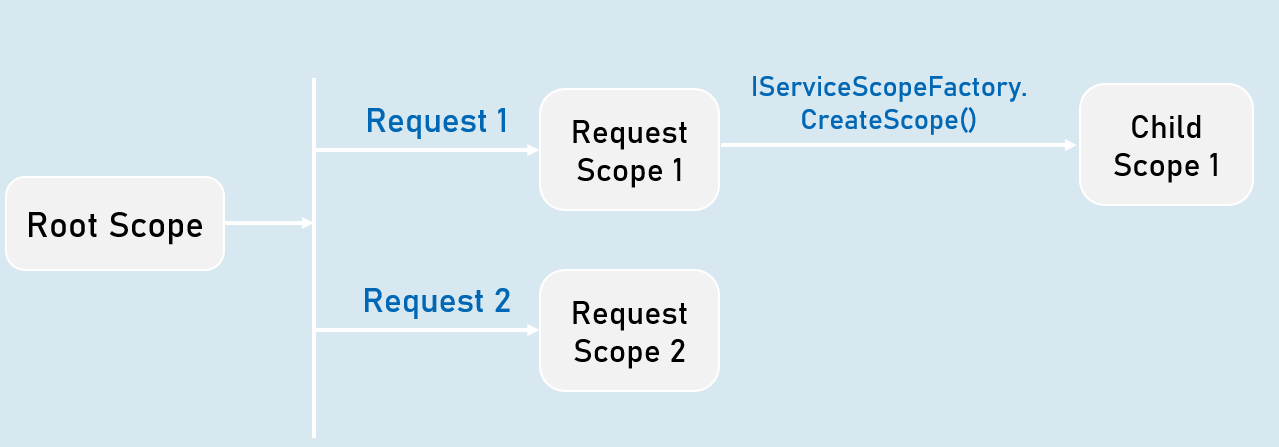
Scoped

builder.Services.AddScoped<IService, Service>(); //Scoped Service

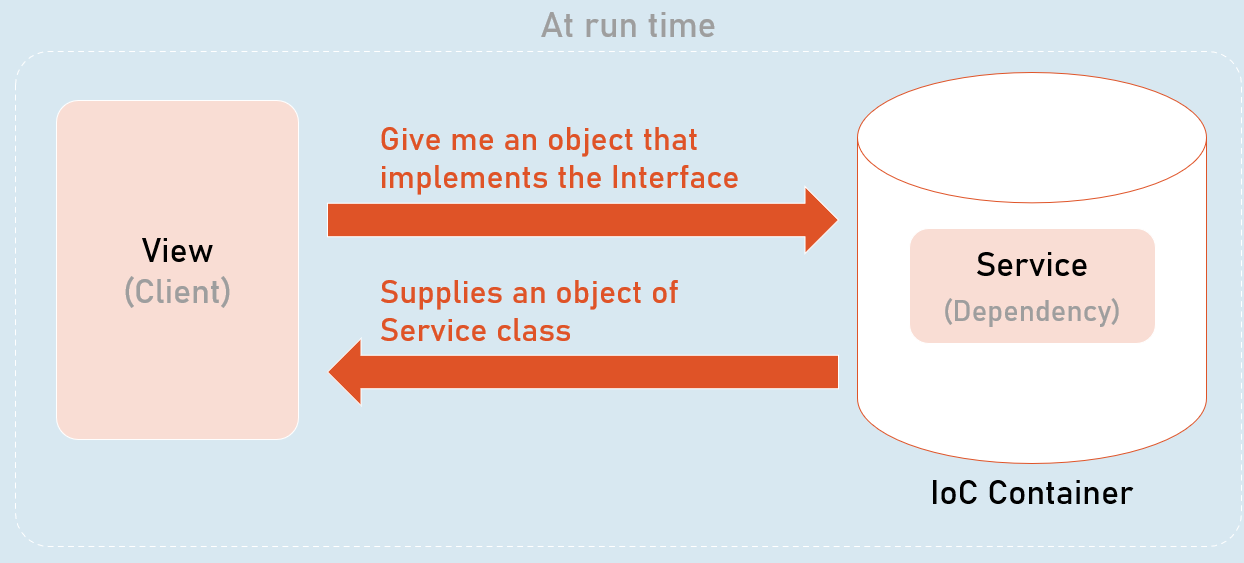
Singleton

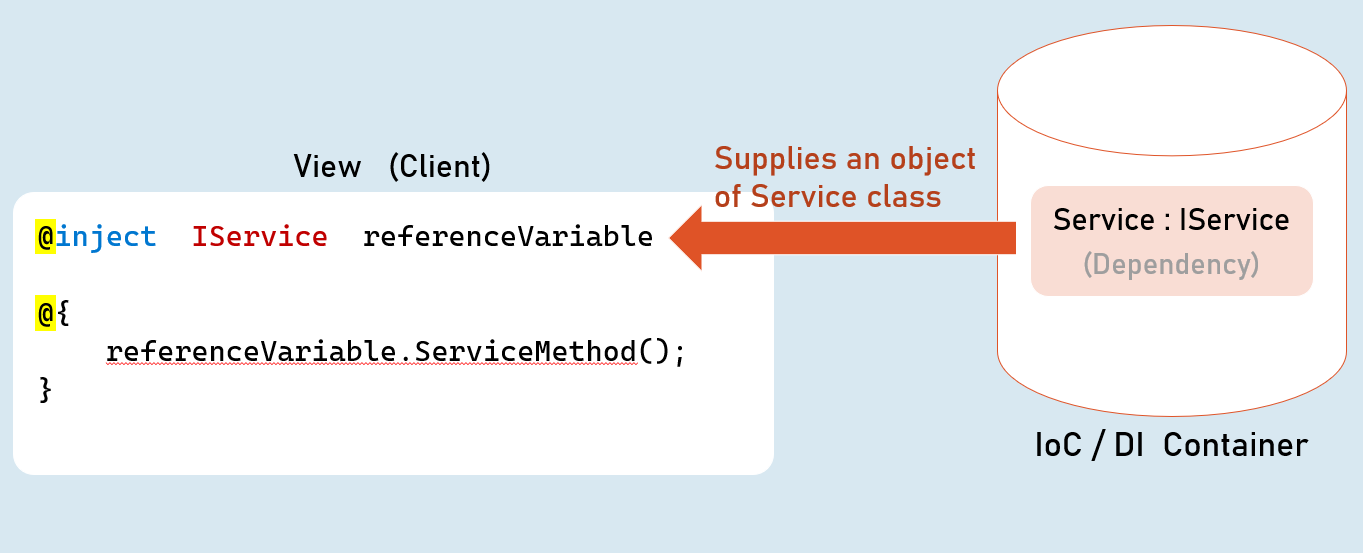
builder.Services.AddSingleton<IService, Service>(); //Singleton Service

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**

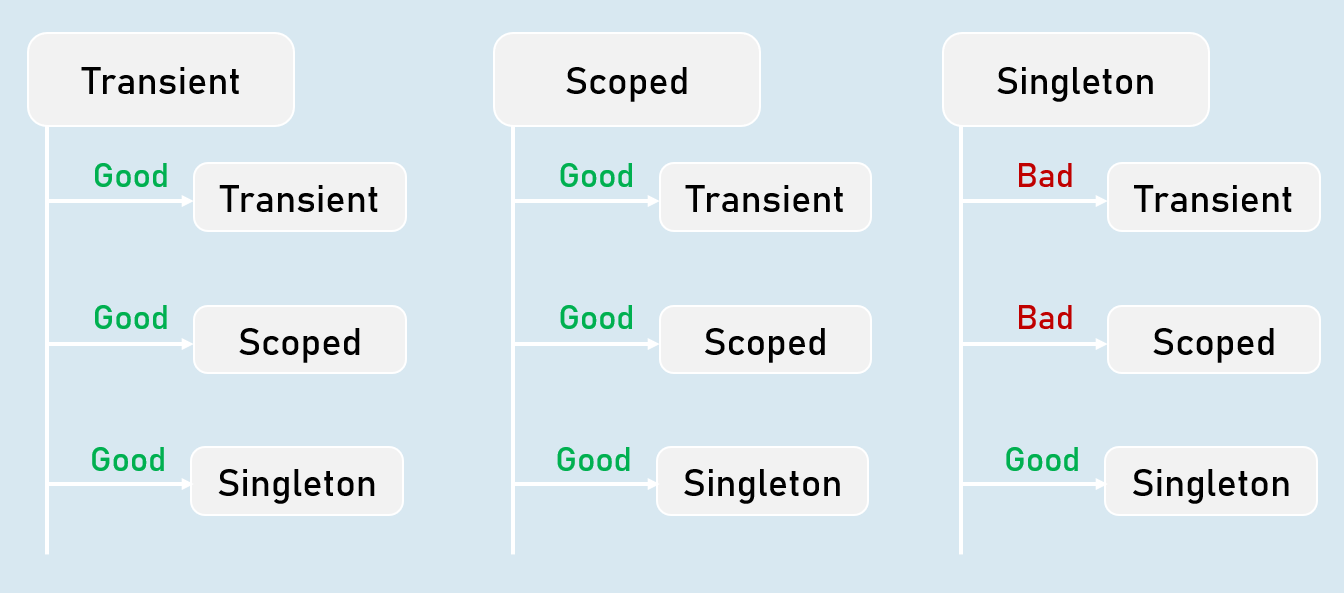
Don't inject scoped or transient services in singleton services.

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

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".

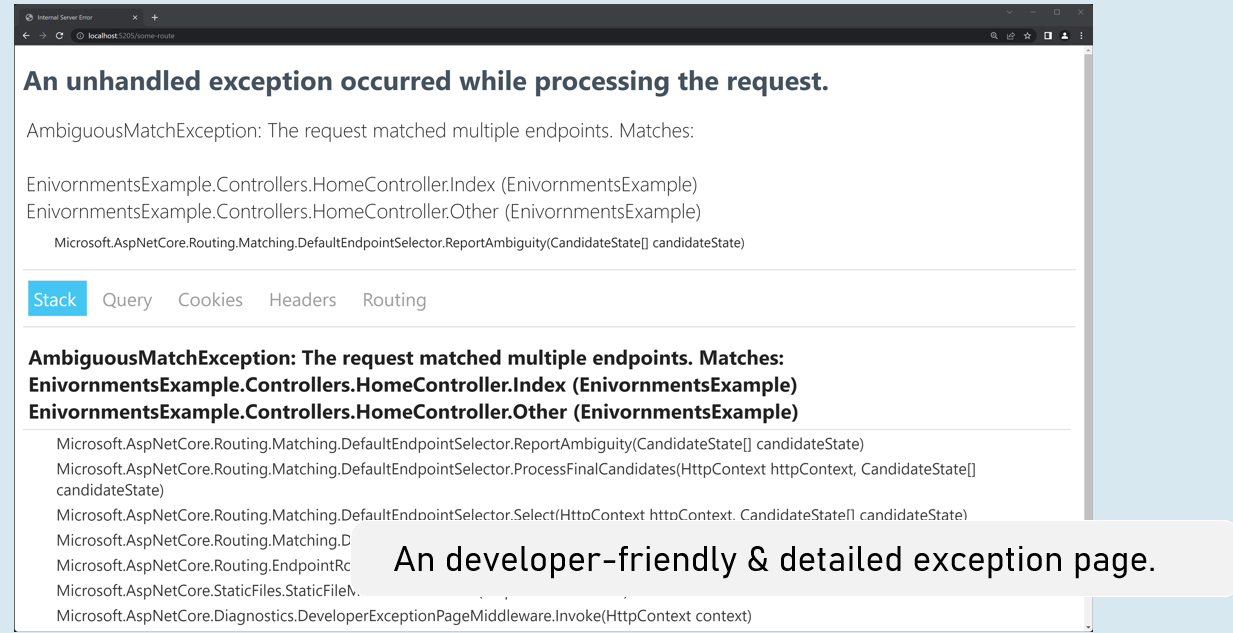
**IsEnvironment(string environmentName)**

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

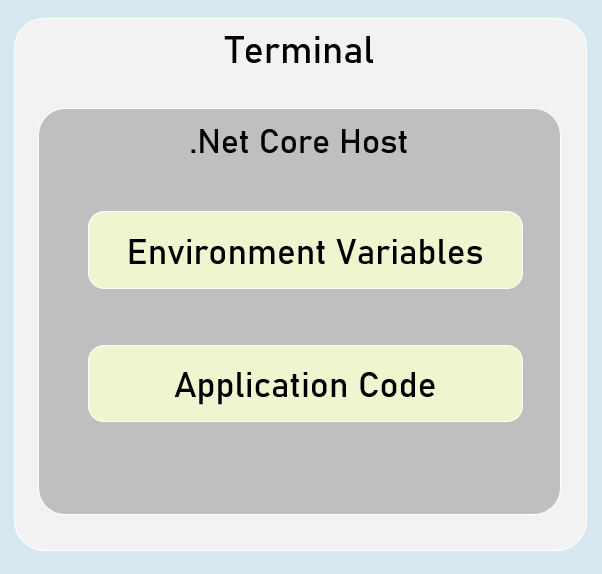


Enable developer exception page

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**

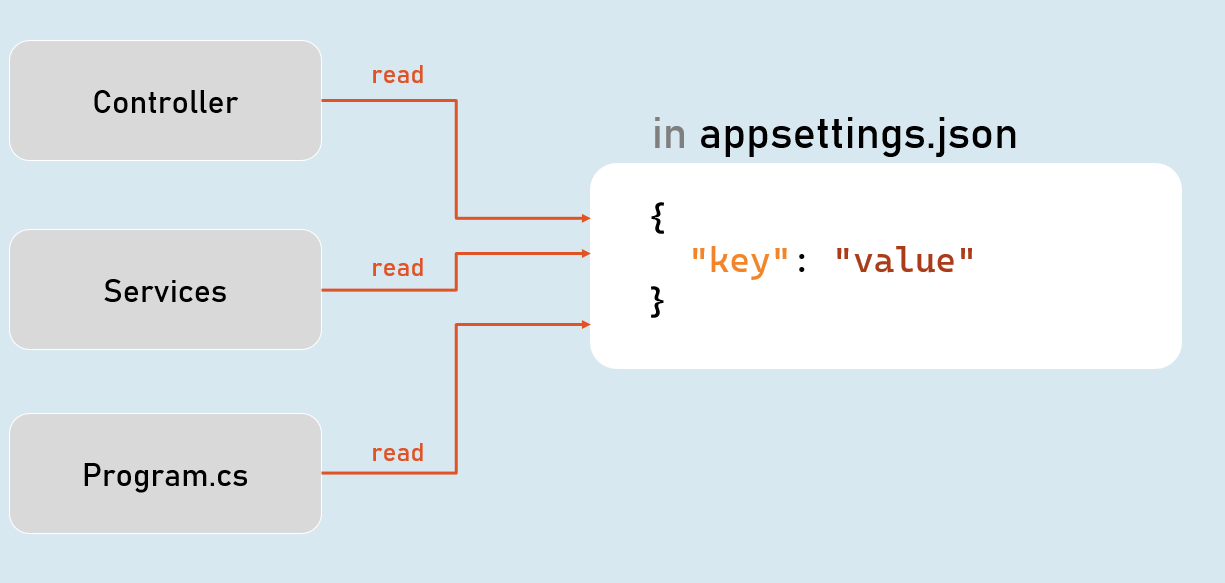
1. <environment exclude="Environment1,Environment2">
2. html content here
3. </environment>

It renders the content only when the current environment name doesn't match with either of the specified environment names in the "exclude" property.

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. }

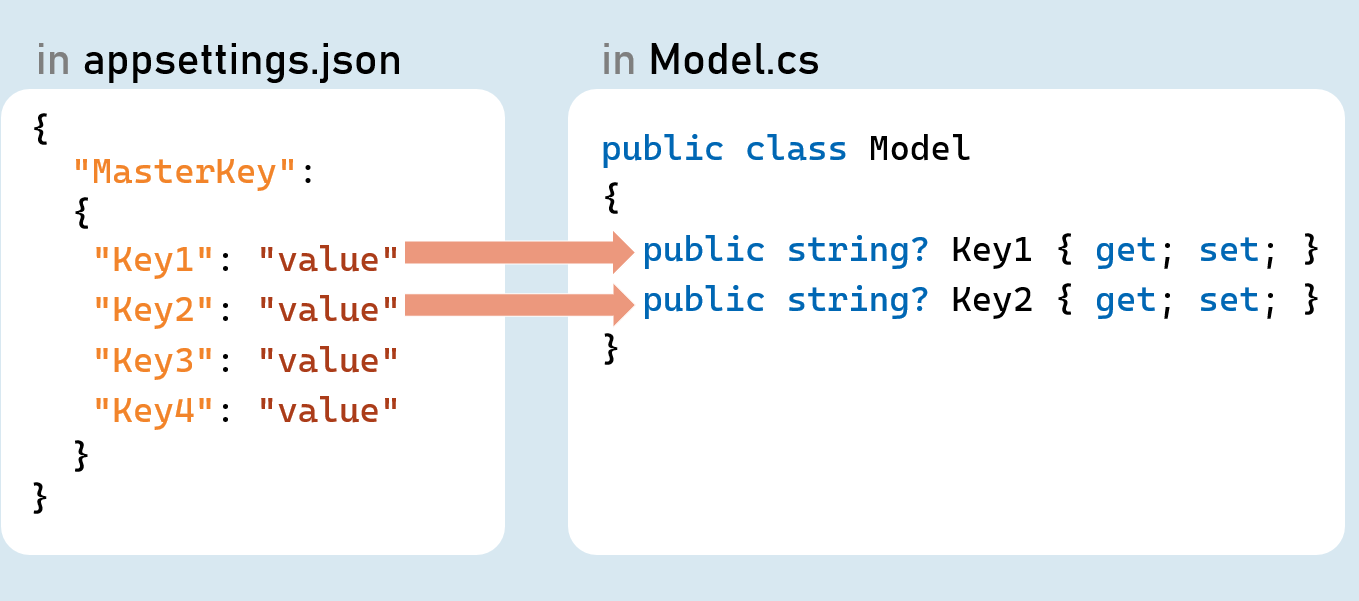
**to read configuration**

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)**

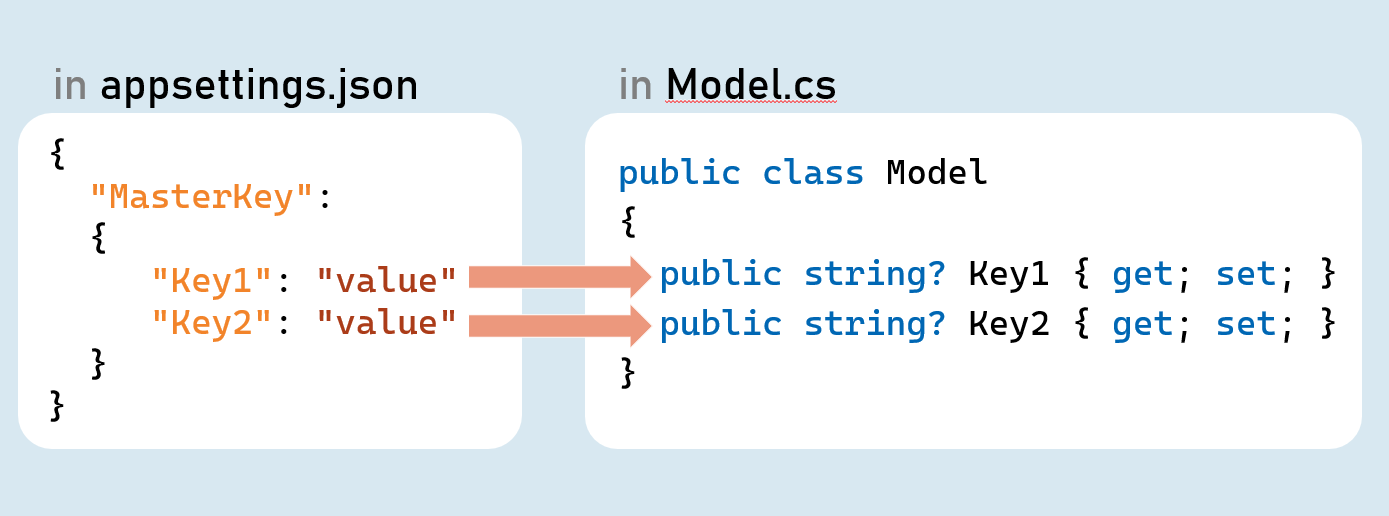
Returns an IConfigurationSection based on the specified key.

**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. }

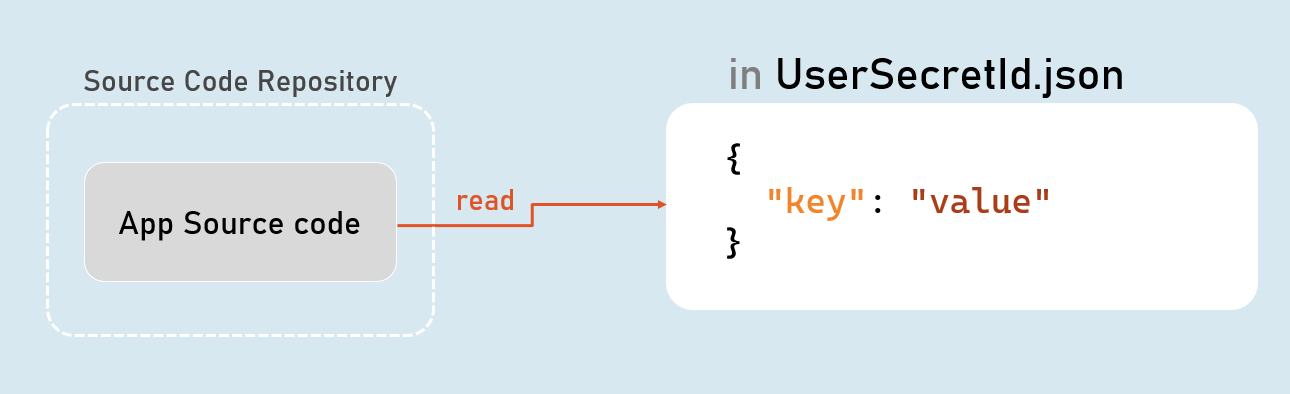
Environment Specific Configuration

Order of Precedence of Configuration Sources



Secrets Manager

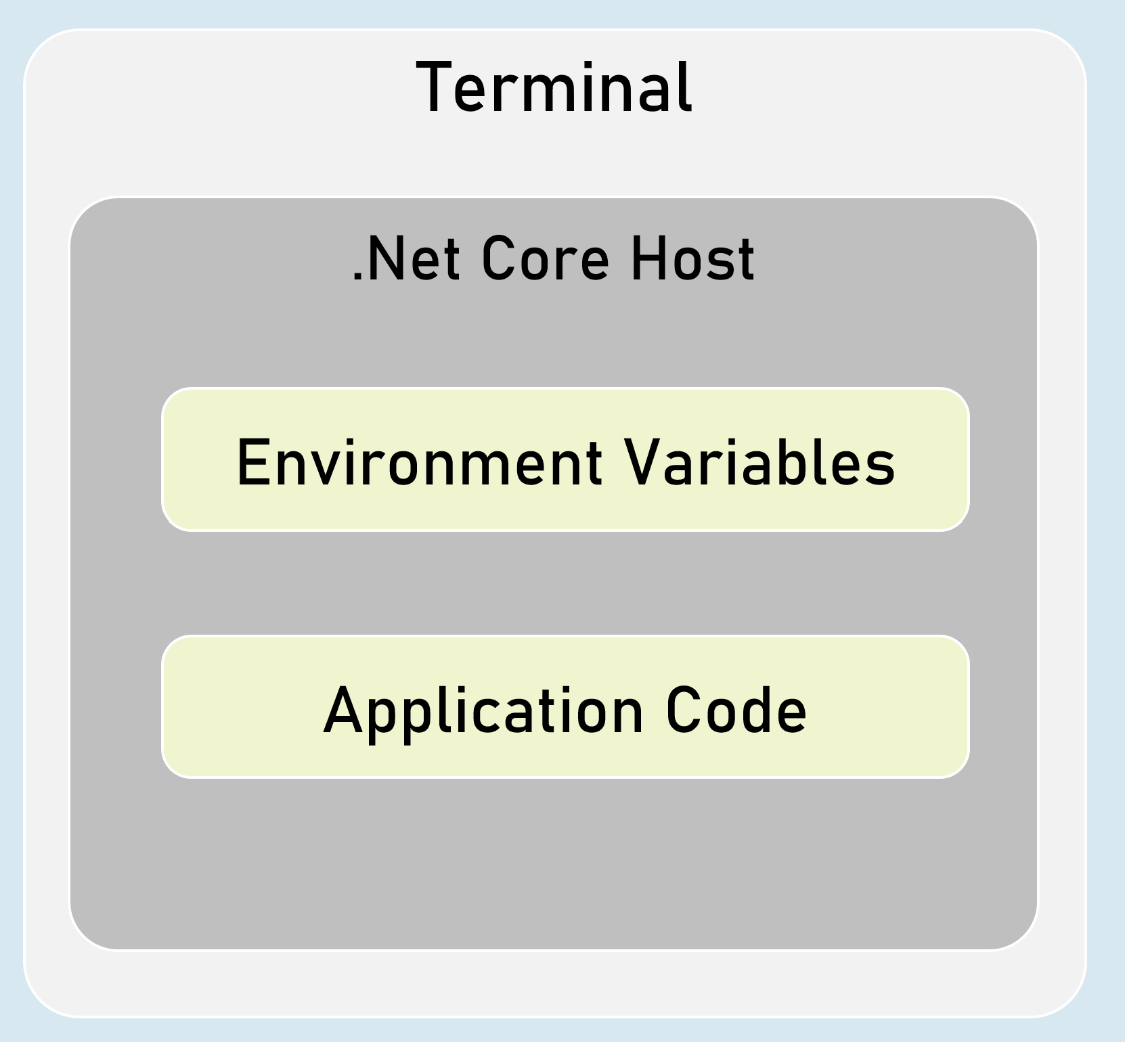
The 'secrets manager ' stores the user secrets (sensitive configuration data) in a separate location on the developer machine.



**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**

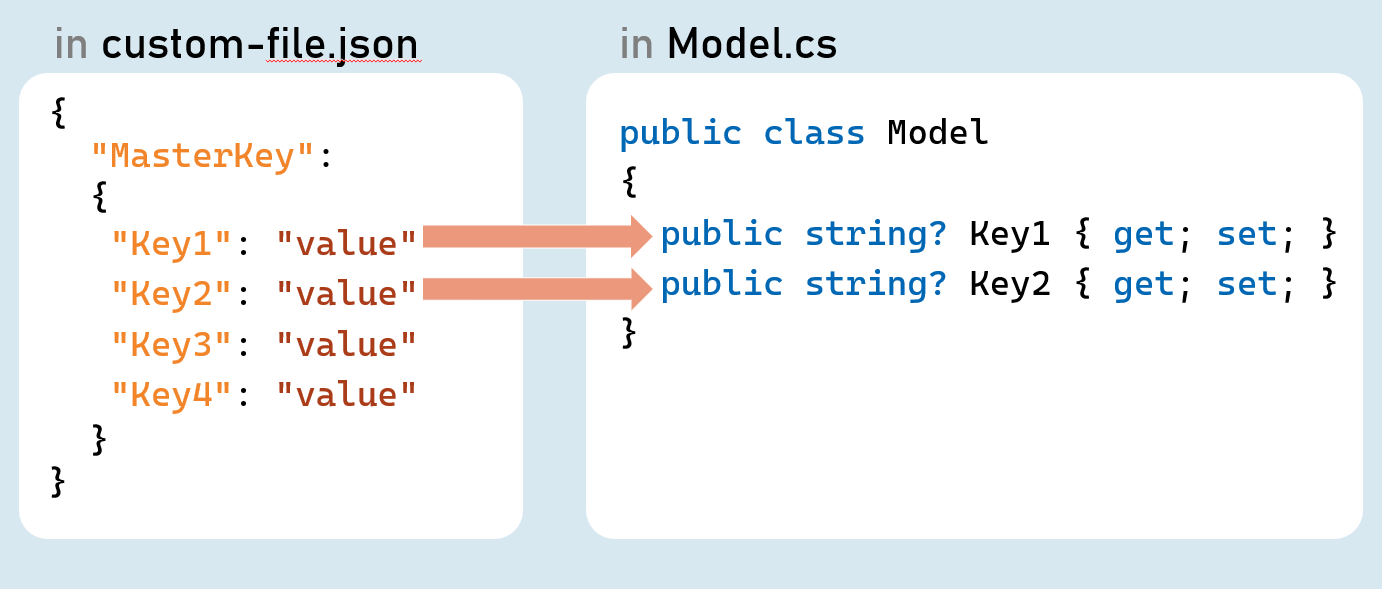
in "Windows PowerShell" / "Developer PowerShell in VS":

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**

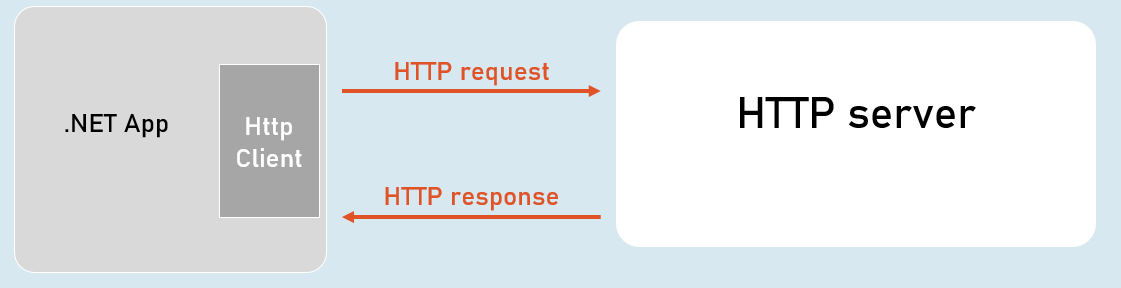
in Program.cs:

1. builder.Host.ConfigureAppConfiguration( (hostingContext, config) => {
2. config.AddJsonFile("filename.json", optional: true, reloadOnChange: true);
3. });

Http Client

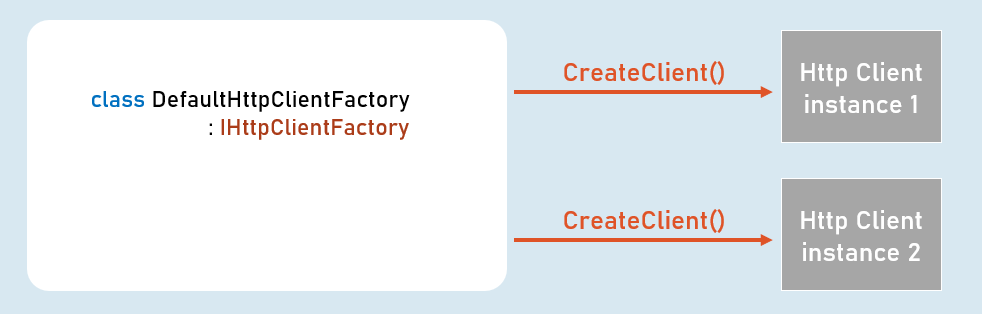
HttpClient is a class for sending HTTP requests to a specific HTTP resource (using its URL) and receiving HTTP responses from the same.

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()