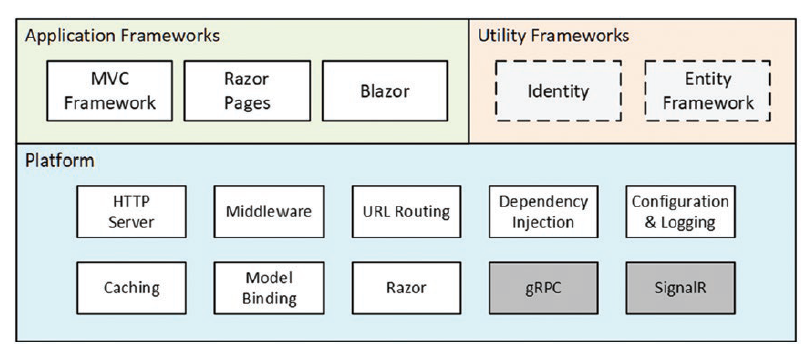
**Dot Net Core**

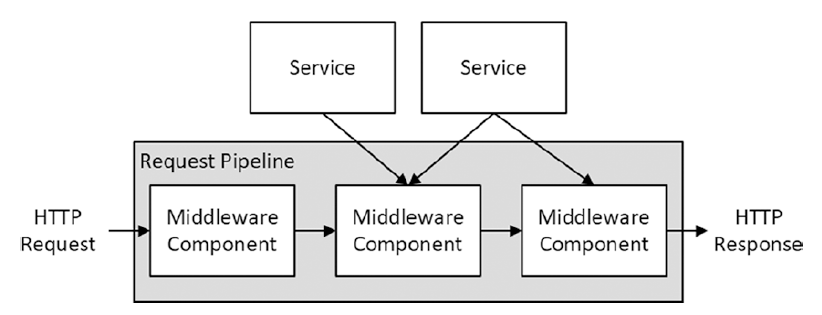
****

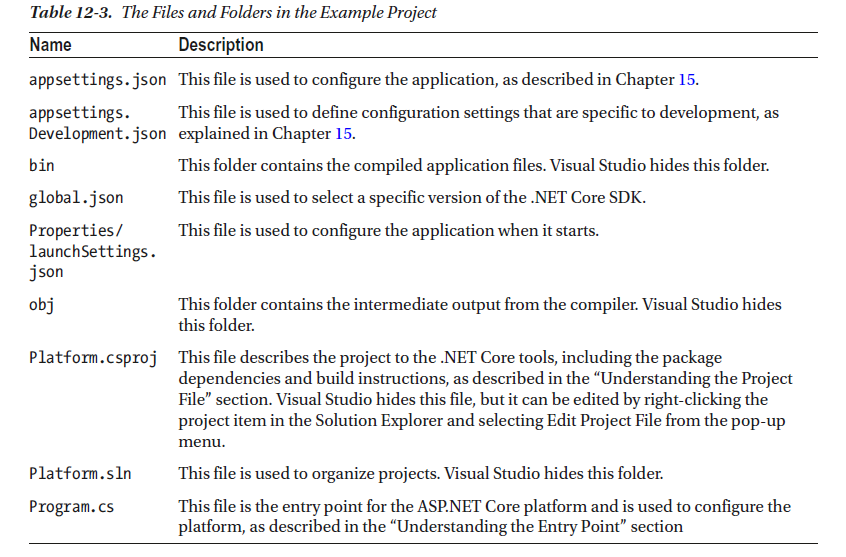
**ASP.NET Core MVC**

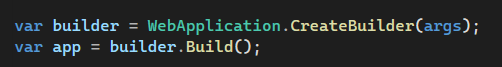
* When the project is created, a file named launchSettings.json is created in the Properties folder and it is this file that determines which HTTP port ASP.NET Core will use to listen for HTTP requests
* dotnet watch **🡪** commands helps us run and watch dotnet applications.
* When the form posts back from HTML and if there is no ‘POST’ method in the controller, it will be directed to the usual GET call.

**ASP.NET Core Platform**

* Three important focus elements of Asp.net Core platform are request pipeline, middleware and services.
* Middleware components are arranged in a chain called *Request Pipeline*.
* The purpose of Asp.Net core platform is accept incoming HTTP requests and send them responses. Once HTTP requested is received Asp.Net core platform creates an object that represents HTTP request and sends an object that represents HTTP response. This object once created goes through the chain of middleware components. The first middleware component inspects the request object and modifies the response. This object will be sent to next middleware component and so on. Once the request is out of this request pipeline Asp.Net Core returns the response.



* Asp.Net Core project structure: 

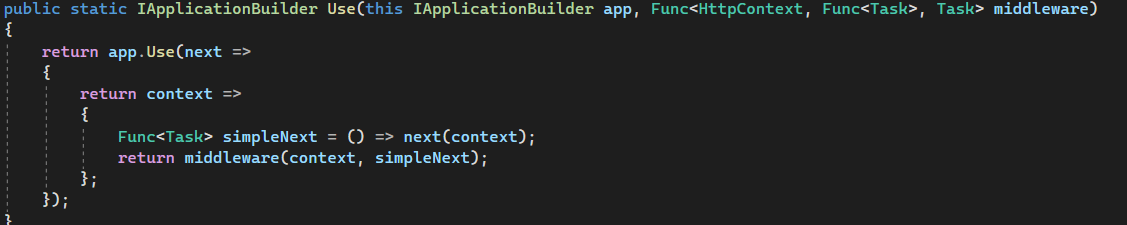


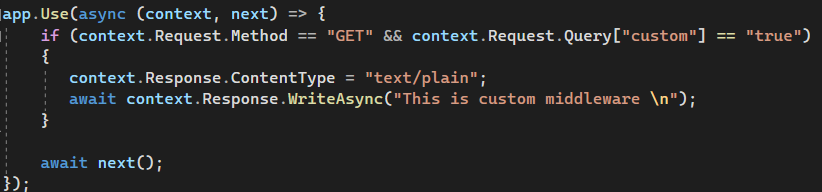
The line ***WebApplication.CreateBuilder(args)*** does the following:

* Sets up basic features of ASP.NET Core Platform required for creating services responsible for logging & configuration.
* Creates a HTTP Server called Kestrel to receive HTTP requests.
* The result from ***CreateBuilder*** is a WebApplicationBuilder object that is used to register additional services.
* **var app = builder.Build()** returns a **WebApplication** objectthat is used to register middleware components.

**Middleware:**

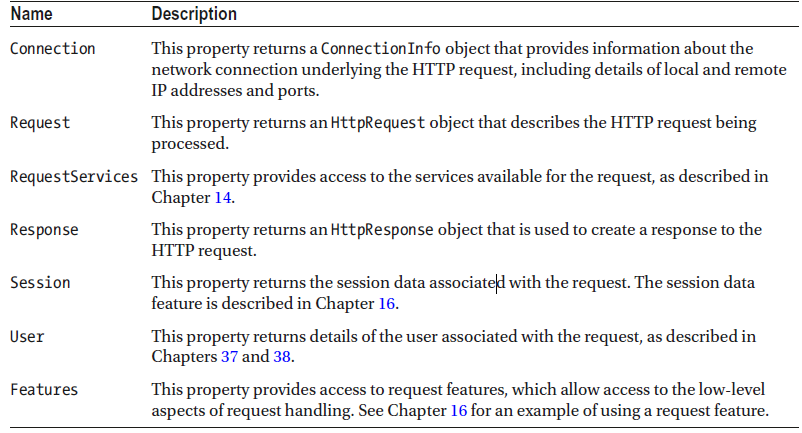
* The ‘Use’ method registers a middleware component that is typically expressed as a lambda function that receives each request as it passes through the pipeline and modifies the response and passed the execution to ‘next’ middleware.

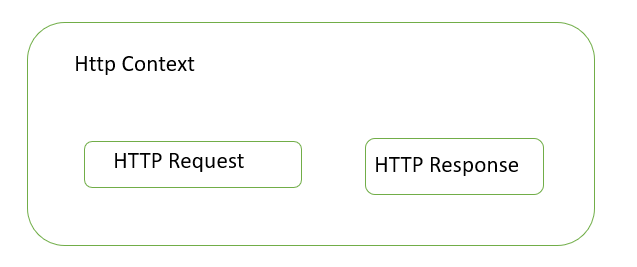
****

****

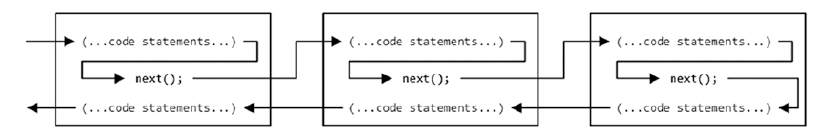
****

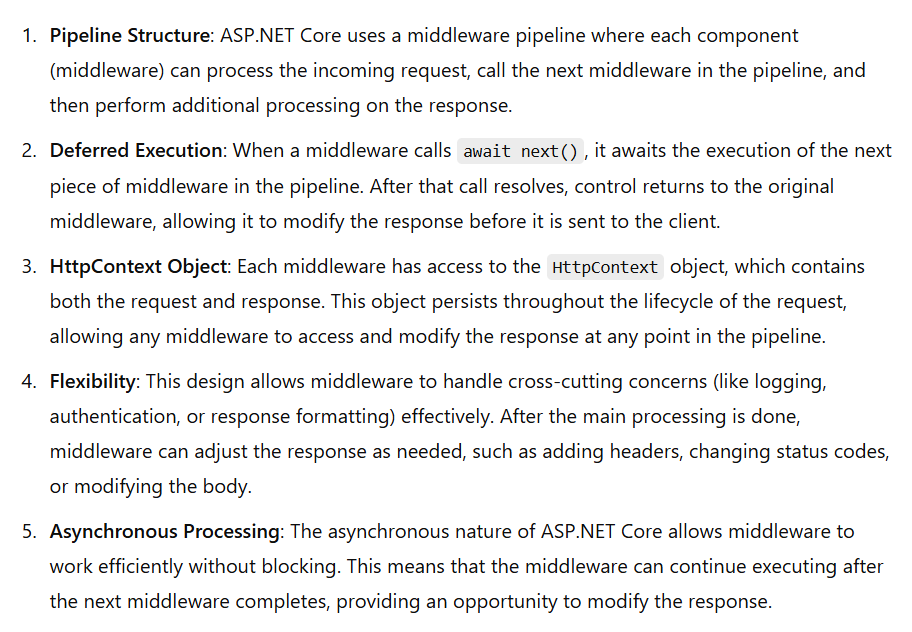
The context here is HttpContext and it contains following members:



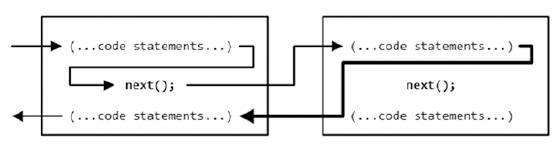


Middleware classes receive a RequestDelegate as a constructor parameter, which is used to forward the request to the next component in the pipeline. The Invoke method is called by ASP.NET Core when a request is received and receives an HttpContext object that provides access to the request and response, using the same classes that lambda function middleware receives. The RequestDelegate returns a Task, which allows it to work asynchronously.

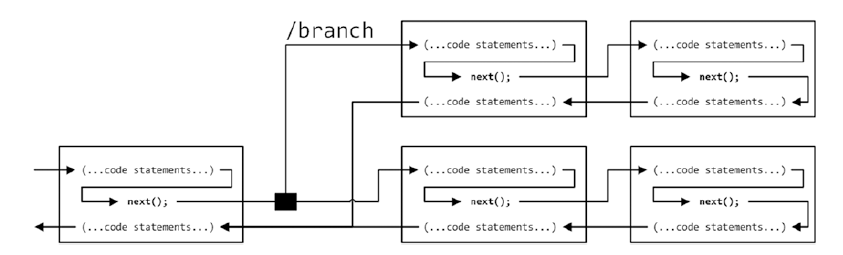




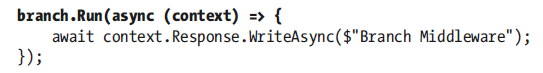
Components that only write response or if they are meant to be end of middleware components they will not call ‘next()’ method. These components don’t pass request to next components and these are called as ‘short circuit components’ like below:



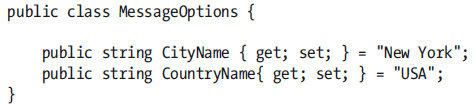
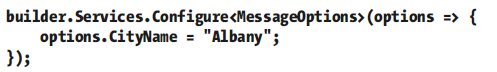
**Map 🡪** This method creates a section of pipeline deviating from original request pipeline.



**Run 🡪** This method can be used to create a terminal middleware and we can use it as convenience.

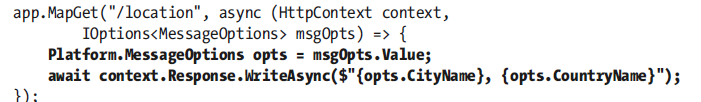


**Configuring Middleware with Options Pattern:**

1. Create an options class 🡪 
2. Asp.Net Core platform will create an new object of ‘MessageOptions’ and it will pass it to ‘Configure’ method 

The configure method looks like this : 

It adds ‘MessageOptions’ object as a service to the service collection.

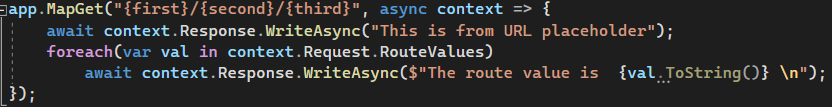
1. Once this service is created we can use it like below: 

**URL Routing:**

**Problems with Custom Middleware:** Each middleware component checks each incoming request and checks if the incoming URL matches it not. Hence this repeats the same process by each middleware component. It is far from ideal can be very difficult to maintain.

To overcome this problem, we can use URL routing.

URL Routing 🡪 It has middleware components and endpoints. Middleware components inspect incoming request URLs for their matching endpoints. The mapping between endpoints and middleware components are expressed in as *route*. The routing middleware components inspects the URLs, matches the endpoints in a process calling as *routing*.

Segment Variables (or route parameters) 🡪 

When processing requests the middleware components gives score for each route upon the incoming request. The route with *lowest score* gets the chance to handle the request. Literal segments are given preference over segment variables and segment variables with constraints are given preference over those without them.

If two routes produce the same score then ambiguous route exception will be thrown.

**Default Values:** 

**Optional Segment: **

**Catch All 🡪 **

**Constraint Segment: **

** 🡪** The first segment should match exactly 3 ‘alphabet’ characters and second segment should be bool.

**Regex Constraint 🡪**  The route will match only those URLs with two segments. The first segment must be capital, and the second segment must be uk, france, or monaco**.**

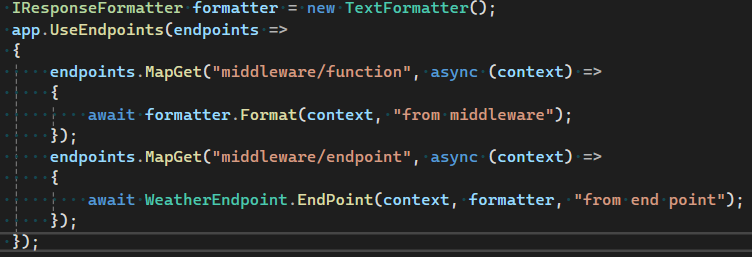
**FallBack route 🡪** This route will used if all the routes have failed to process the incoming request so that we can use this route as fallback route. A white background with black text

Description automatically generated

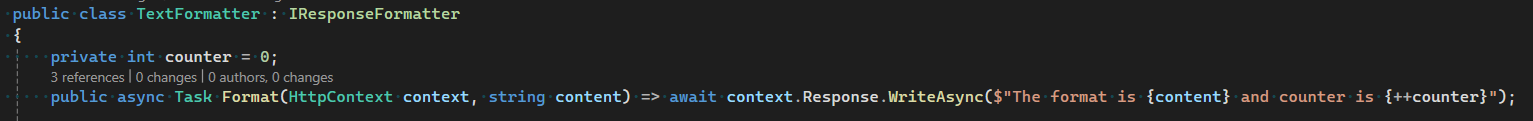
**Dependency Injection**:

* IServiceProvider 🡪 gets the service object in an endpoint
* ActivatorUtilities 🡪 Instantiating a class that has constructor dependencies.

Let’s say if we have formatter class being used in both WeatherEndpoint and also in inside the end point like below



And with text formatter like below:

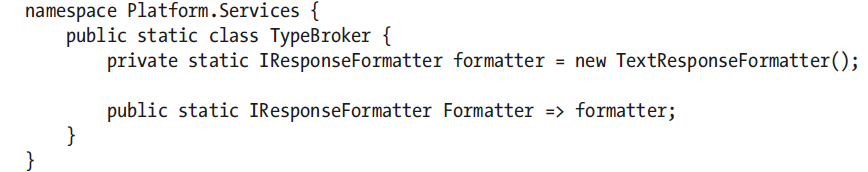


It’s clear that we will have a single counter object that will be shared between both.

We can also use a singleton object to do so.

But if we create a singleton like above then we have to change the formatter to (TextFormatter.Singleton()) at every place it is used. Hence to avoid it we use type broker pattern.

A type broker is a static class that handles the instantiation of objects so that if we are change the instantiation we will change it in type broker class instead of all places where it is used.

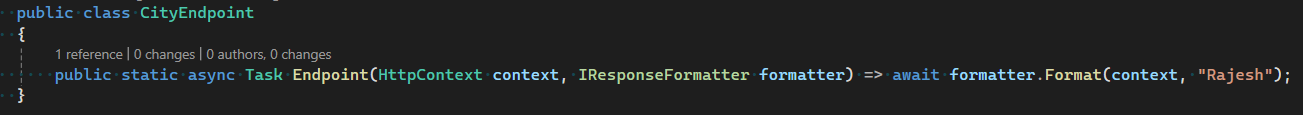


**Dependency Injection:**

When services are defined as below:



And it’s used as





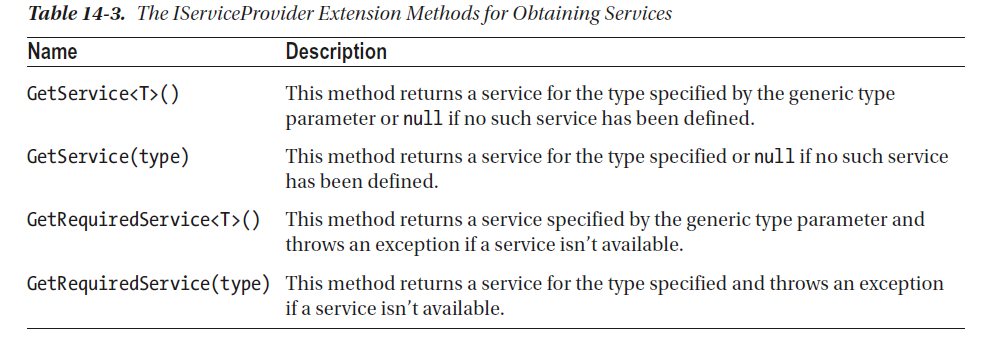
The Asp.net platform understands that it will be access the ‘Endpoint’ when ‘city’ route is mapped and it observes that there is dependency on IResponseFormatter and finds to check if there is any dependency resolved for IResponseFormatter and an object is created for it’s usage.

It’s important to remember that ‘CityEndpoint’ doesn’t know which class resolves ‘IResponseFormatter’ , it just knows that only receives an object that conforms to the interface. Secondly it doesn’t know how this dependency is going to get resolved.

IServiceProvider 🡪 used to get a service in an endpoint

HttpContext.RequestServices property returns an object that implements the IServiceProvider interface that provides access to all the services that have been resolved in program.cs

Microsoft.Extensions.DependencyInjection 🡪 This interface contains all the methods needed for the IServiceProvider interface that allows individual services to be obtained.



The drawback of HttpRequest.RequestServices is that service must be resolved for every request that is routed via the endpoint.

So, another approach is to use the method offered by IEndpointRouteBuilder interface as below:

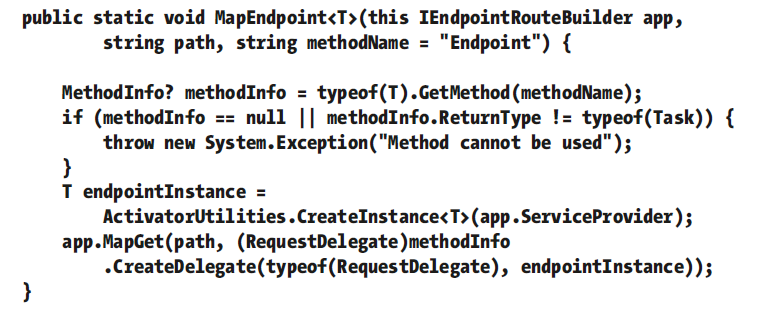
A screen shot of a computer code

Description automatically generated

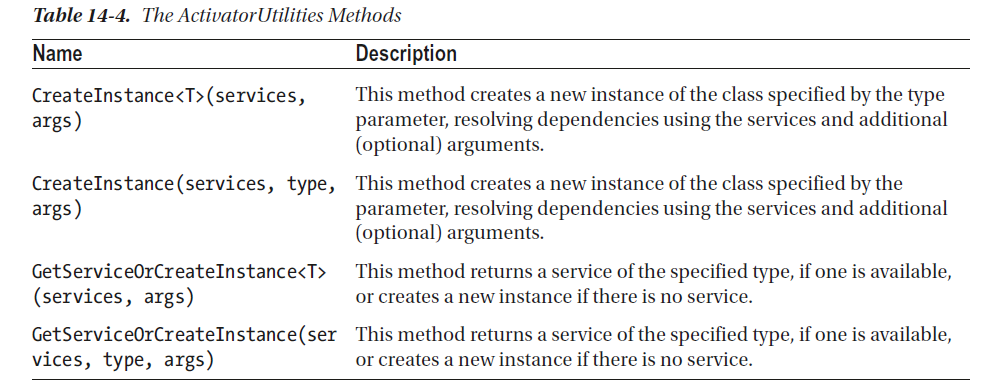
The interface IEndpointRouteBuilder defines a property ‘ServiceProvider’ that returns IServiceProvider object through which services can be obtained.

**ActivatorUtilities** class present in *Microsoft.Extensions.DependencyInjection* namespace provides methods instantiating classes that have dependencies declared through their constructor.

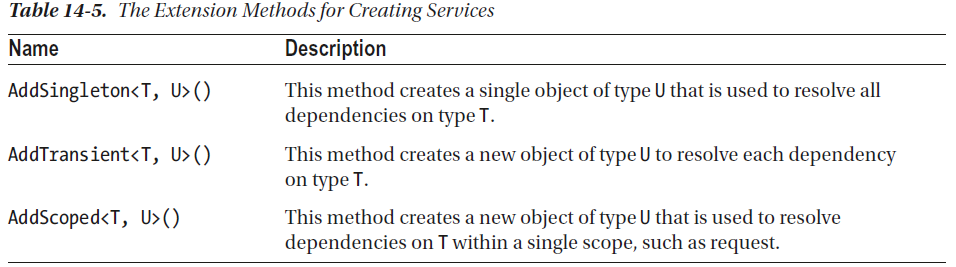
T endpointinstance = ActivatorUtilities.CreateInstance<T>(app.ServiceProvider)



Above method resolves the constructor dependencies using services through IServiceProvider object.



**Dependency Life Cycle**



AddTransient 🡪 Creates a new instance of implementation class for every dependency.

New objects are created only when dependencies are resolved NOT when services are used.

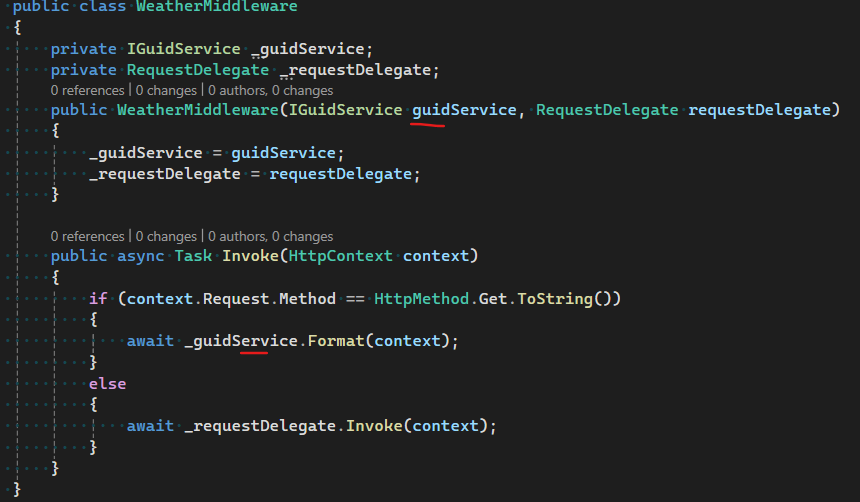
The components and endpoints have their dependencies resolved ONLY when application starts and all the top level statements in the Program.cs execute. Each dependency receives a separate object which is then REUSED for every request that is processed.

But if we are using below code , it’s going to resolve every time when this code gets called as app.ServiceProvider resolves dependencies every time below code runs.

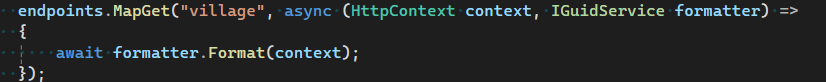
A close-up of words

Description automatically generated

Example: In WeatherMiddleware class when it’s object is created guidService will be resolved and it will NOT resolved every time we call ‘Invoke’ method even if we refresh the page.

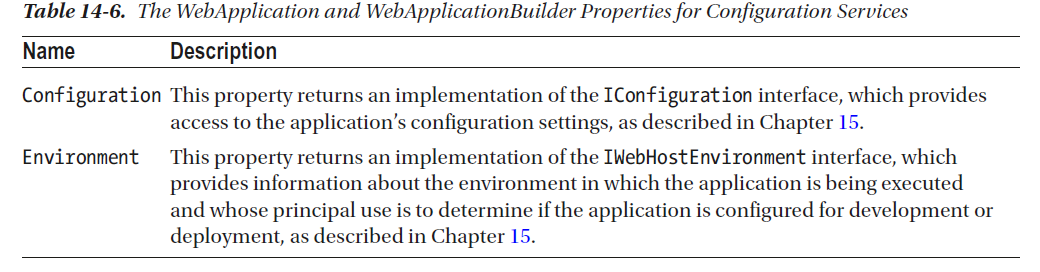


Where as in below case every time the below endpoint gets called ‘formatter’ will be resolved.



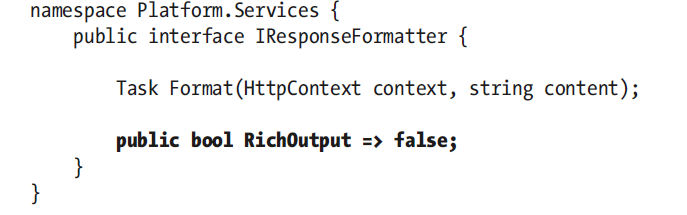
A common requirement can be to use application’s configuration settings while defining to alter a set of services that are created in Program.cs file. But configuration is presented as a service and services can’t be accessed until builder.Build() method is invoked.

To fix this WebApplication and WebApplicationBuilder classes define properties that provide access to built-in services that provide access to application configuration:

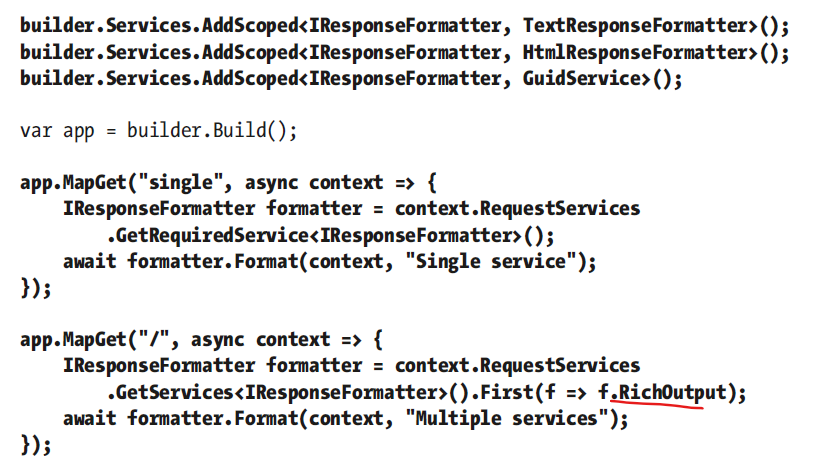


Creating Services with Multiple implementations:

Suppose we have an interface like this:



Some classes can implement this interface with ‘RichOutput’ set as true and we can resolve their dependency on the fly like shown below:



**Logging**

* Asp.Net core provides logging service that can be used to record messages for monitoring and diagnosing problems.
* Log messages are sent to logging providers that can send, store and process these messages. The three built in providers that are enabled by default : console, debug and EventSource.
  + Console 🡪 The console logging provider forwards messages to Console.
  + Debug 🡪 Debug logging provider forwards messages so they can processed by System.Diagnostics.Debug class.
  + EventSource 🡪 It’s provider forwards messages for event tracing tools like PerfView.

**Using Cookies**

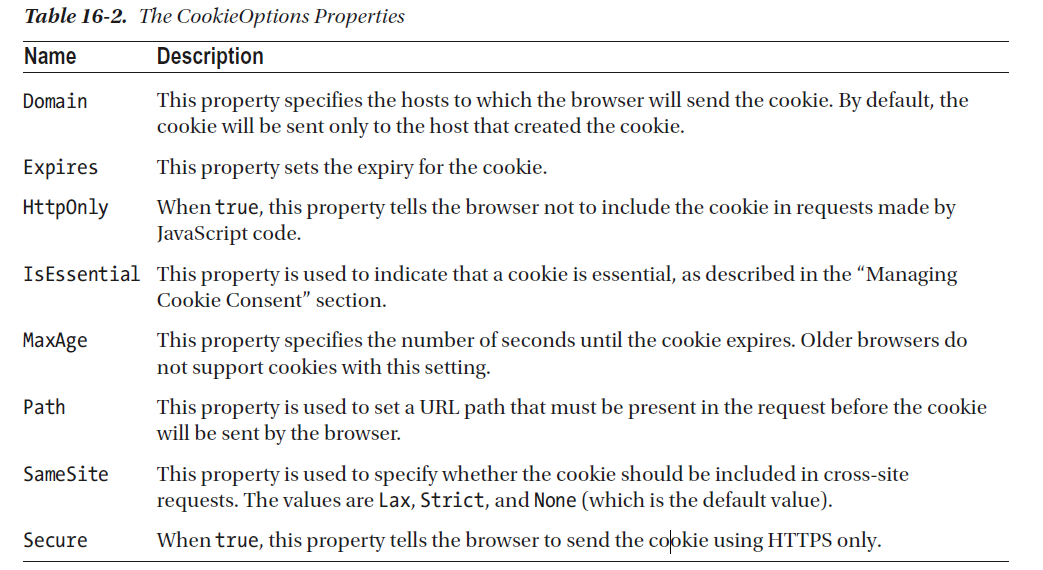
* Cookies are small text that are sent by the server as part of HttpResponse.

Ex : context.Response.Cookies.Append(“key”,value, new CookieOptions { Max = TimeSpan.FromMinutes(10) });

* Once browser receives them it will re-send them back to the server as part of HttpRequest.

Ex: context.Request.Cookies[“key”]

* Cookies are important for Web applications as they allow features to be developed that span a series of HTTP requests, each of which can be identified by cookies sent by browser to server.



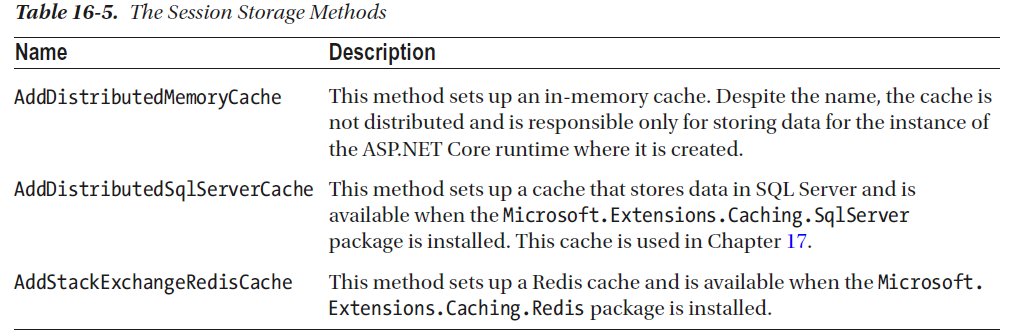
Cookie Consent:

The EU General Data Protection Regulation (GDPR) requires the user’s consent before non essential cookies can be used. Asp.Net core provides support for obtaining consent and preventing non essential cookies from being sent to the browser when consent has been granted.

Managing Cookie Consent:

**Use Sessions**

* The problem with cookies is that as they are stored on the client they can easily be manipulated and used to alter the behaviour of the web application.
* A better approach is to use Asp.Net Core Session feature that adds a session related cookie in the response sent to browser first and browser sends back this session cookie so that data associated with the session cookie can be stored in the server. Session values are stored in server not on client.
* When session cookie (identifier for that session) is received then session middleware retrieves it’s relevant session stored server data and makes it available for other parts of the middleware or application through HttpContext object.

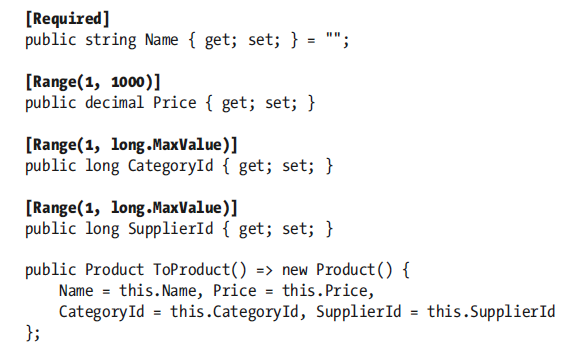
****

**Enabling MVC**

* builder.Services.AddControllers() 🡪 add services to web service controllers needed for MVC framework and app.MapControllers() 🡪 defines the routes that will allow controllers to handle requests.
* In .Net 6 , we can use builder.services.AddControllersWithViews() for adding features to controllers (controllers without [ApiController] attribute)

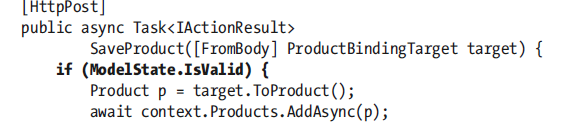
**Data Validation**

* The properties can be validated using **System.ComponentModel.DataAnnotations** namespace

****

For example: [Required] attribute here suggests that this property is needed from client and null can be also assigned when there is no value in the request.

Only objects that contains valid properties are considered as valid with their model state updated as valid.

A black text on a white background

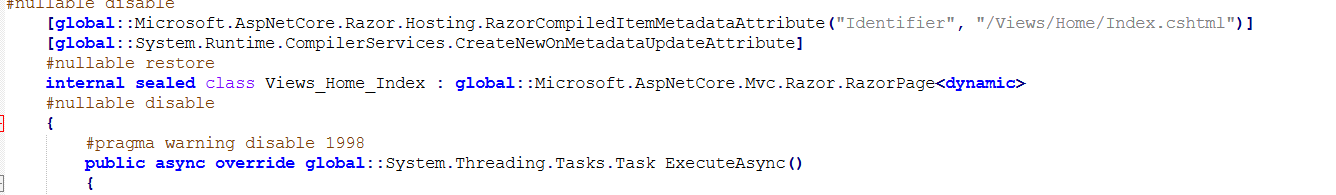
Description automatically generated

Else the action method should return a ‘BadRequest’.

**Controllers & Views**

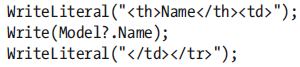
* The MVC framework assumes that all the public methods in the controller are action methods and that action methods supports all HTTP methods. So, if we don’t want to define any method as action method then we should declare it either as private or use ‘NonAction’ attribute for the action method.
* All the views are compiled in DLLs but if we mention like below in the csproj file 

then all the compiled files will be present in obj/debug/…/generated/ with every class implementing RazorPage<T> like below

****

* Responses from views can be cached by applying ‘ResponseCache’ attribute to the action methods.
* The expressions in the view are translated using Write method that takes a string value and encodes it so that it can safely included in HTML document. The method WriteLiteral also takes string parameter but this parameter value will not be encoded as this method is used to pass on the static HTML regions of the view.

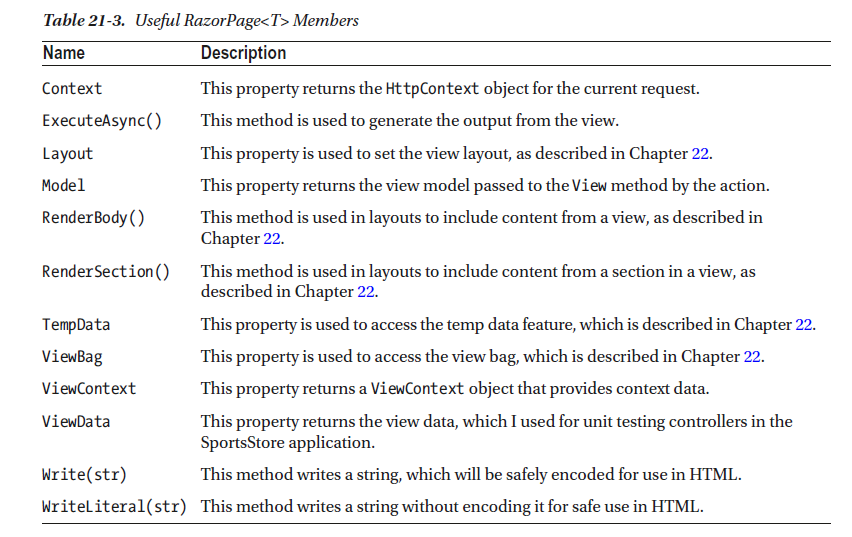
For example:  This is converted into a series of C# statements like these in the ExecuteAsync method.



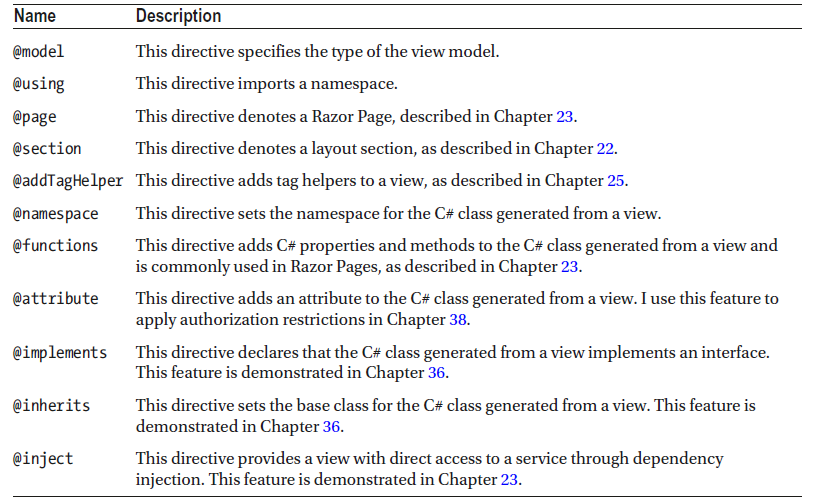
When ExecuteAsync() method is invoked the results are written as a combination of static HTML and expressions contained in the view into HTML like this 🡪 

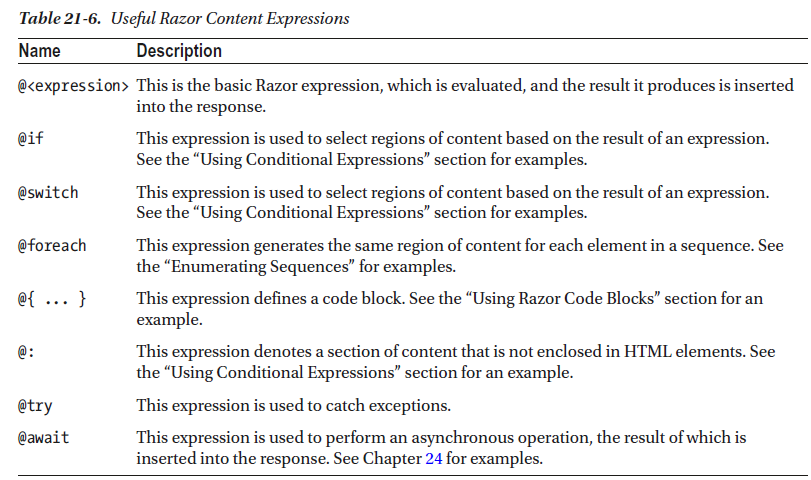
View Imports File 🡪 By default all the types that are referenced in the razor page should be qualified with a namespace. Instead of specifying namespaces in individual razor cshtmls we can mention the name space directly in view imports file.

Any view that starts with an \_ (underscore) is meant not to be delivered to the user.

****

**Directives:** These are expressions that give instructions to the razor view engine.





ViewBag 🡪 It is inherited from ‘Controller’ class that returns ‘dynamic’ object. It is best used to transfer small amount of supplementary data from controller to view without having to create new view model classes.

TempData 🡪 It allows a controller to store the data from one request to another which is useful in performing redirections. It is usually stored in cookie unless session data is enabled when it is stored as session data. Temp data values are marked for deletion once they are read and removed once request has been processed.

We can maintain \_ViewStart.cshtml for every Views/[controller] folder as Razor engine will give preference to this file than centralized one.

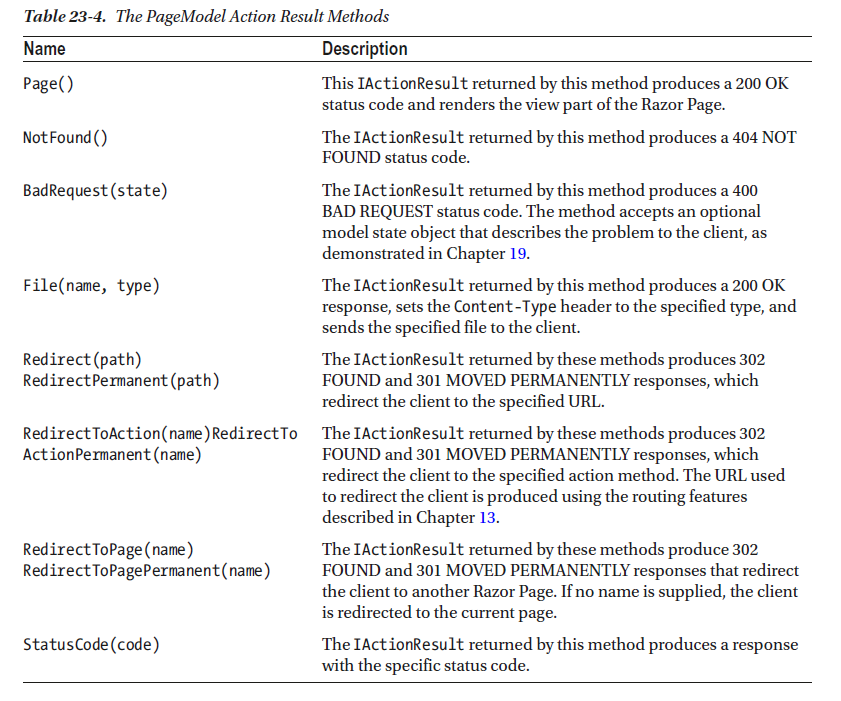
We can have a code block @{ } where we can define Layout. If we don’t want any Layout we can define Layout = null.

Layout Sections 🡪 Razor engine allows us to have sections and we define the @section <sectionName> { } block in the view cshtml page and then define @RenderSection(“sectionName”) in the layout cshtml.

@Html.Raw(..) 🡪 Encodes the HTML content

**Razor Pages**

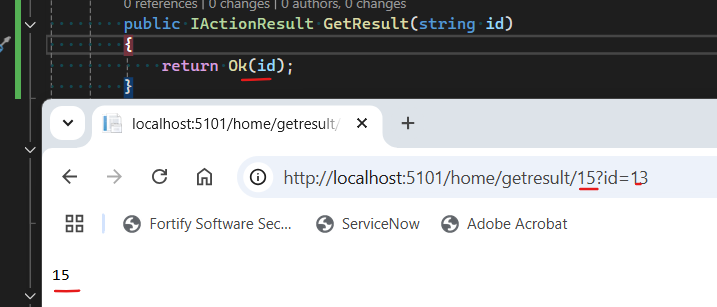
* Razor pages can be faster than MVC framework and they associate a single view with the class that provides features and they use file-based routing system to match URLs.
* Razor pages are less flexible than MVC framework and they can only generate HTML responses. They are no good for REST services.
* builder.services.AddRazorPages() 🡪 adds services to the DI container, app.MapRazorPages() 🡪 adds middleware needs to be added for razor pages.

****

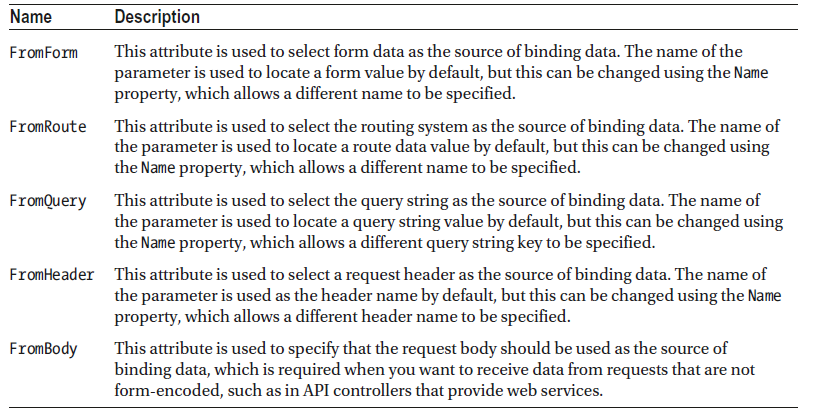
**Model Binding:**

* It acts as a bridge between HTTP request and action methods or page handler methods. Model binding binds the values from model binders, components that are responsible for providing data values from one part of the HTTP request or application. These model binders gets their values from:
  + Form Data
  + Response Body (only for controllers decorated with ApiController)
  + Routing segment variables or Route Data
  + Query Strings

The search for the data checks in above places until a suitable value is found in above defined order. Once data is found the searching for data variables stops. In below example we can see that the value of ‘id’ will be 15 instead of 13 as route segment variables has 15 so id value as 15 will be picked up.



* Simple Data types 🡪 If any item of data can be parsed from string then it is called Simple Data Type. Example: numeric values, decimal values, datetime values.
* Model Binding Source 🡪 As discussed we have four data sources, Form data, Response Body (only in case of web service controllers), Route data & query strings. But what if we want to other data sources then we will need to define explicitly using below binding source attributes:

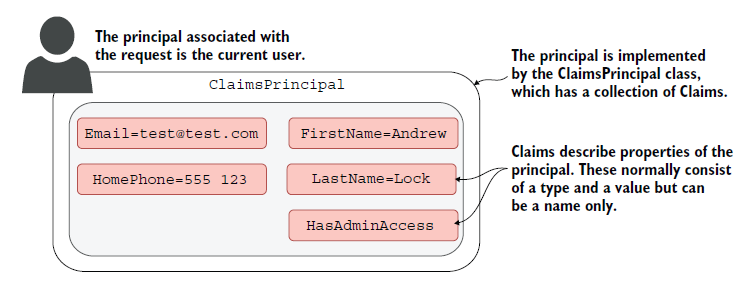


**Authentication & Authorization:**

* Authentication 🡪 The process to identity who you are
* Authorization 🡪 The process to identify what you want to do or what you can access

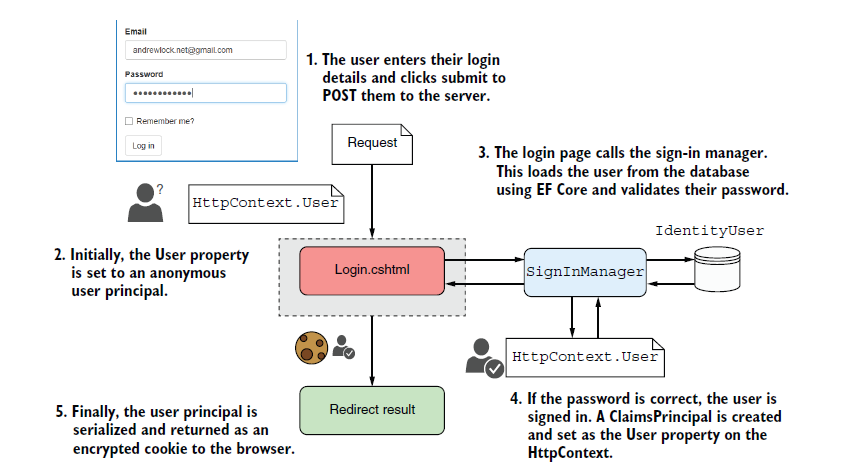
**Users and Claims:**

* HTML Server and Kestrel are going to create an *HTTPContext* object for every incoming request.
* HttpContext.User principal gives all the information related to the user. In Asp.Net Core, claims are implemented using *ClaimsPrincipal* class

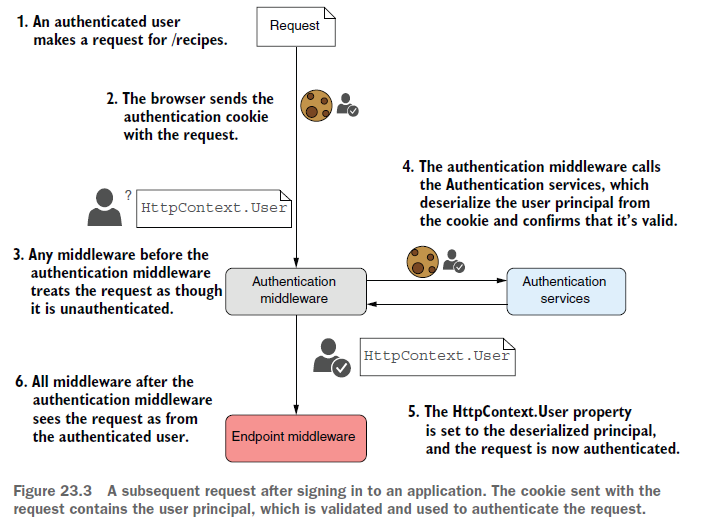
****

*A claim is a single piece of information about principal that contains claim type and optional value.*

A claim can also be indirectly related to permissions and authorizations.



Once user gets authenticated for subsequent requests the browser sends the encrypted cookie along with the request which gets verified at the server. After verification *principal* for the user in HttpContext object will be assigned.



**How OpenID Connect Works:**

1. **User Requests Authentication**: The user tries to access a resource that requires authentication.
2. **Redirect to Identity Provider**: The application redirects the user to an identity provider's authentication endpoint.
3. **User Authenticates**: The user enters their credentials on the IdP.
4. **Authorization Code**: Upon successful authentication, the IdP redirects the user back to the application with an authorization code.
5. **Token Exchange**: The application uses the authorization code to request tokens (access token, identity token) from the IdP.
6. **Access Protected Resources**: The application can now use the access token to access protected resources and verify the identity token to authenticate the user.

**Example Use Case:**

In a web application, you might implement OpenID Connect to allow users to log in using their Google or Microsoft accounts. When users click "Log in with Google," they are redirected to Google's authentication page. After logging in, Google sends them back to your application with an identity token containing user information.

**Without Identity Provider:**

* Without identity provider we will need to have a mechanism to store user’s private date like storing passwords (by hashing them using one way algorithm), store claims or roles, provide MFA, protect against brute force attacks etc.,

**Asp.Net Core Identity**

* By default it uses EF Core to store user details in the database.

