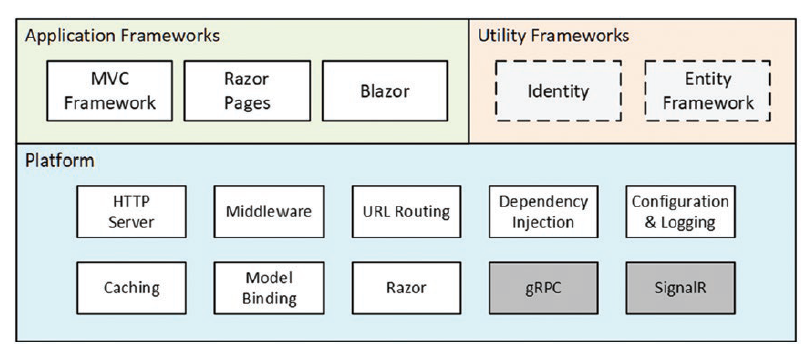
**Dot Net Core**

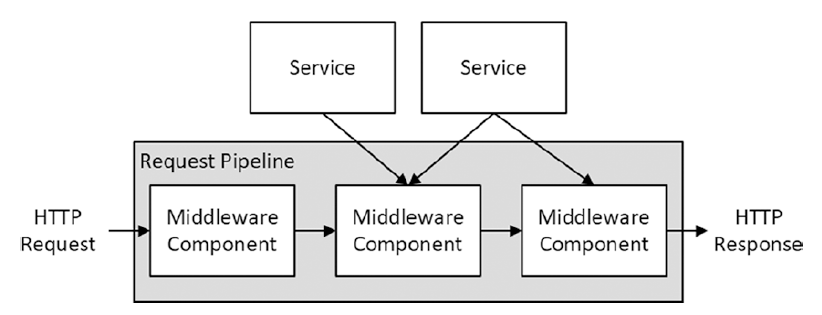
****

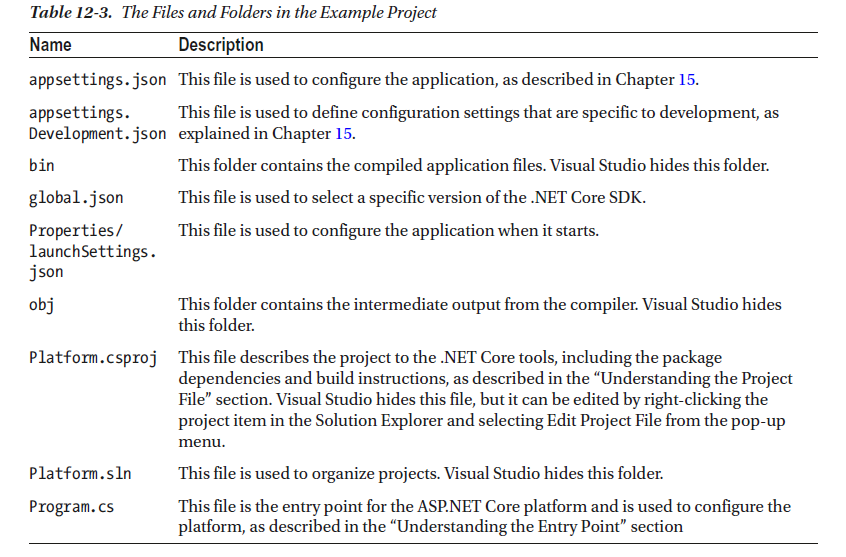
**ASP.NET Core MVC**

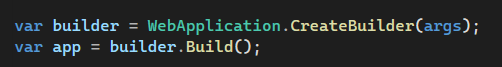
* When the project is created, a file named launchSettings.json is created in the Properties folder and it is this file that determines which HTTP port ASP.NET Core will use to listen for HTTP requests
* dotnet watch **🡪** commands helps us run and watch dotnet applications.
* When the form posts back 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: 

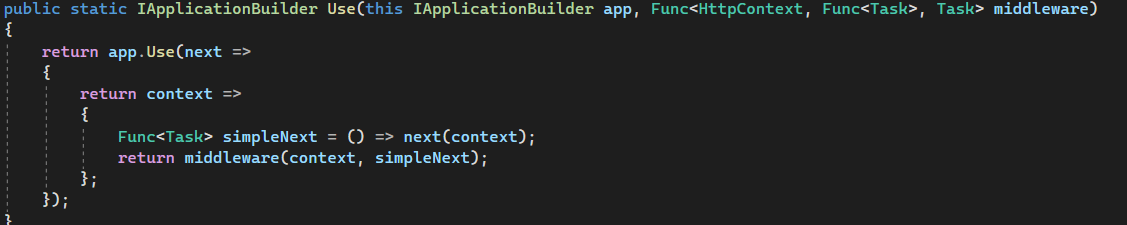


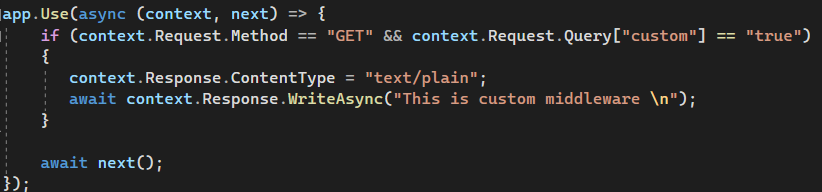
The line ***WebApplication.CreateBuilder(args)*** does the following:

* Sets up basic features of ASP.NET Core Platform required for creating services responsible for logging & configuration.
* Creates a HTTP Server called Kernel 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:**

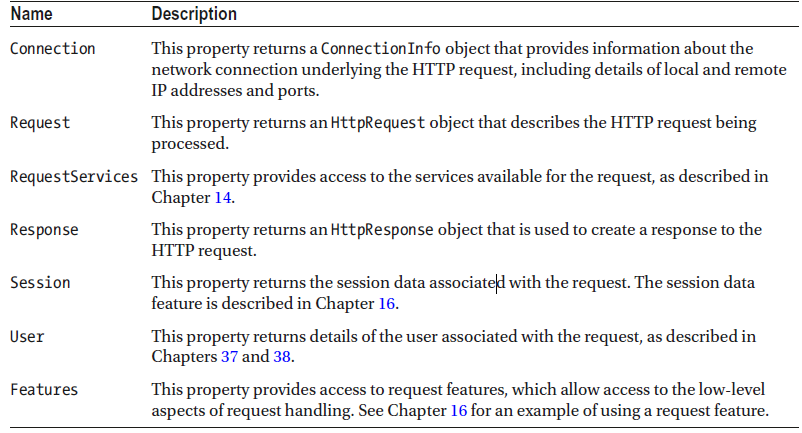
* The ‘Use’ method registers a middleware component that is typically expressed as a lambda function that receives each request as it passes through the pipeline and modifies the response and passes the execution to ‘next’ middleware.

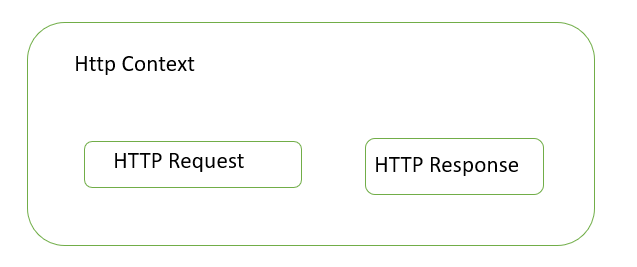
****

****

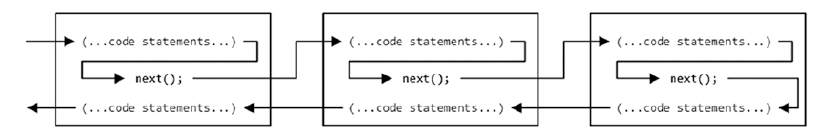
****

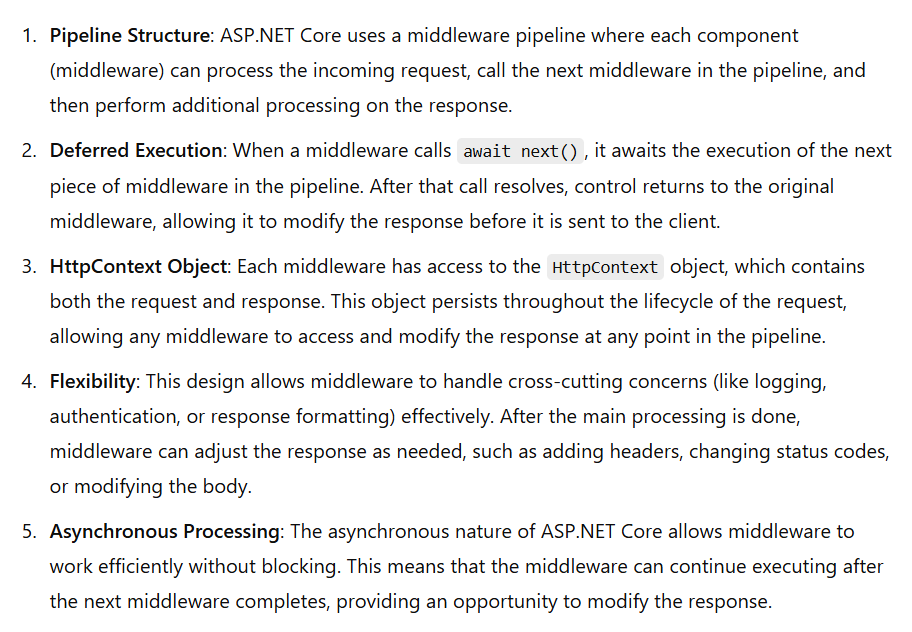
The context here is HttpContext and it contains following members:



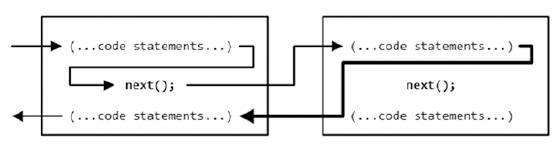


Middleware classes receive a RequestDelegate as a constructor parameter, which is used to forward the request to the next component in the pipeline. The Invoke method is called by ASP.NET Core when a request is received and receives an HttpContext object that provides access to the request and response,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

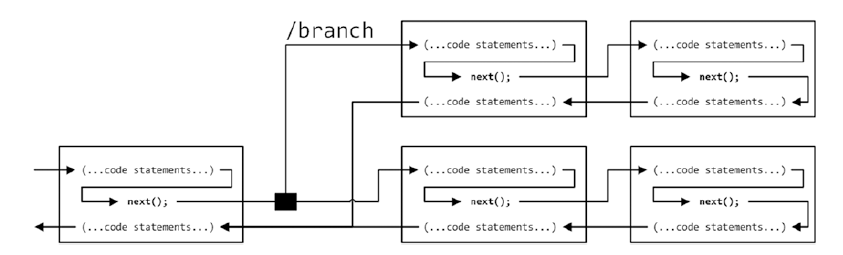




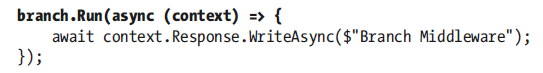
Components that only write response or if they are meant to be end of middleware components they will not call ‘next()’ method. These components don’t pass request to next components and these are called as ‘short circuit components’ like below:



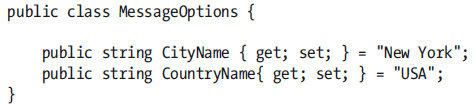
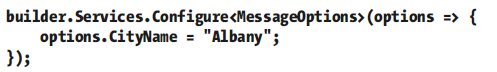
**Map 🡪** This method creates a section of pipeline deviating from original request pipeline.



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

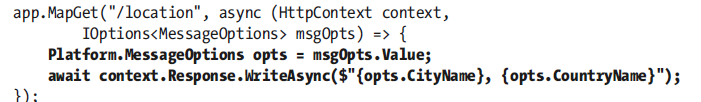


**Configuring Middleware with Options Pattern:**

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.

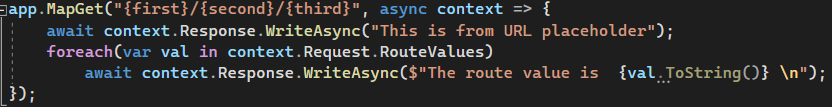
1. Once this service is created we can use it like below: 

**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

**UseRouting():** This sets up middleware components required for the application. This does the following:

* It sets up the routing Table with defined routes.
* It allows application to look for incoming requests to match appropriate endpoints that match route patterns
* It adds routedata to the HTTPContext for middlewares or endpoints to use.

UseEndPoints() 🡪 This is used to configure endpoints that handle the incoming request based on route pattern.

**Note**: Asp.Net Core implements UseRouting() and UseEndPoints() into the request pipeline so that methods like Map, MapGet, MapPut, MapPost, MapDelete can be used automatically (without specifying ‘UseRouting()’ )on the **WebApplication** class object returned WebApplication.CreateBuilder().

**Dependency Injection:**

Most projects have features that need to be used in different parts of the application, which are known as *services.* The object that resolves the dependency is created outside the function that uses it, it is said to be injected and hence this process is called *dependency injection*.

Services are registered using extension methods given by IServiceCollection, an implementation of which is obtained by using WebApplicationBuilder.Services property.

During request pipeline when ASP.NET Core platform reaches this statement:

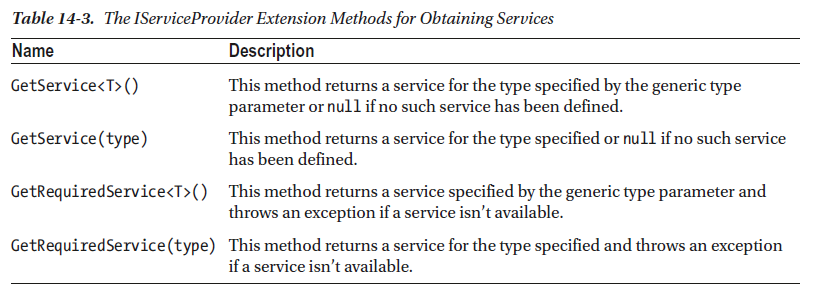


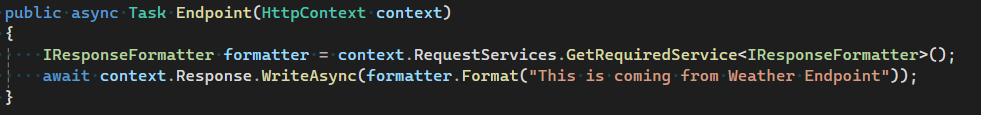
The platform understands that it needs to create an instance for ‘WeatherMiddleware’ class and inspects the constructor. The dependency of an interface or class will be identified. Then it looks for the object of that dependency in the services object (of WebApplicationBuilder.Services) and the dependency is resolved.

There is no need for WeatherMiddleware class to know how these dependencies are resolved.

**Resolving Service’s Dependency**

* HttpContext.RequestServices 🡪 This property returns an object that implements *IServiceProvider* interface that provides access to the services that are configured in program.cs. The Microsoft.Extensions.DependencyInjection namespace contains extension methods for the *IServiceProvider* interface that allow individual services to be obtained



Example: 

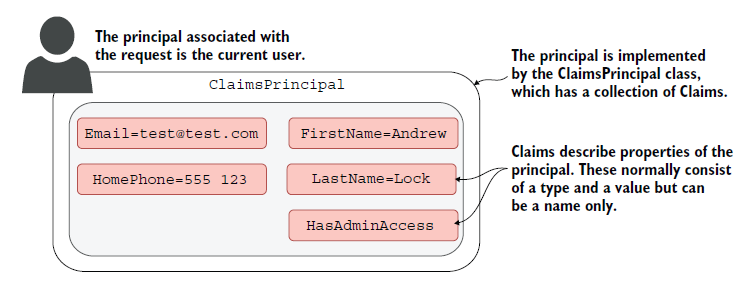
AddSingleTon 🡪 This tells that a single object should be used to satisfy all the demands of the service.

**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

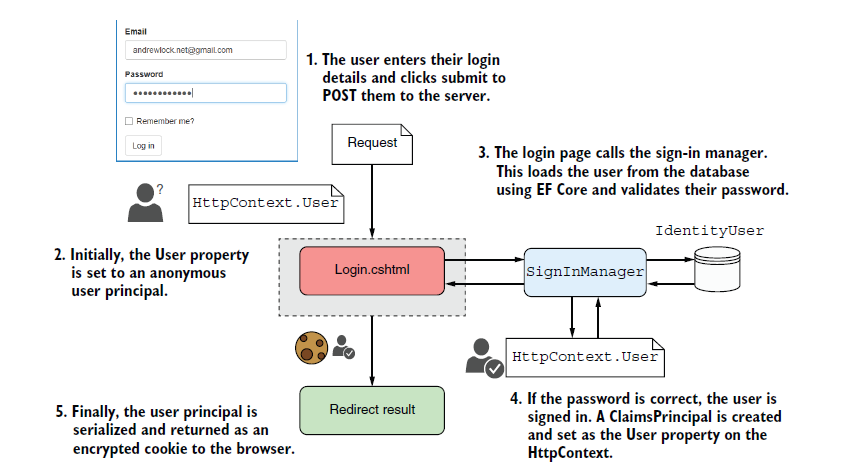
**Users and Claims:**

* 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

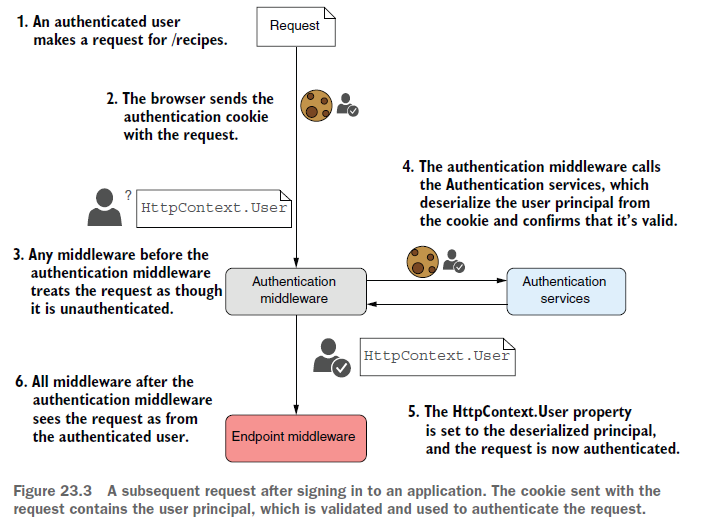
****

*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.

