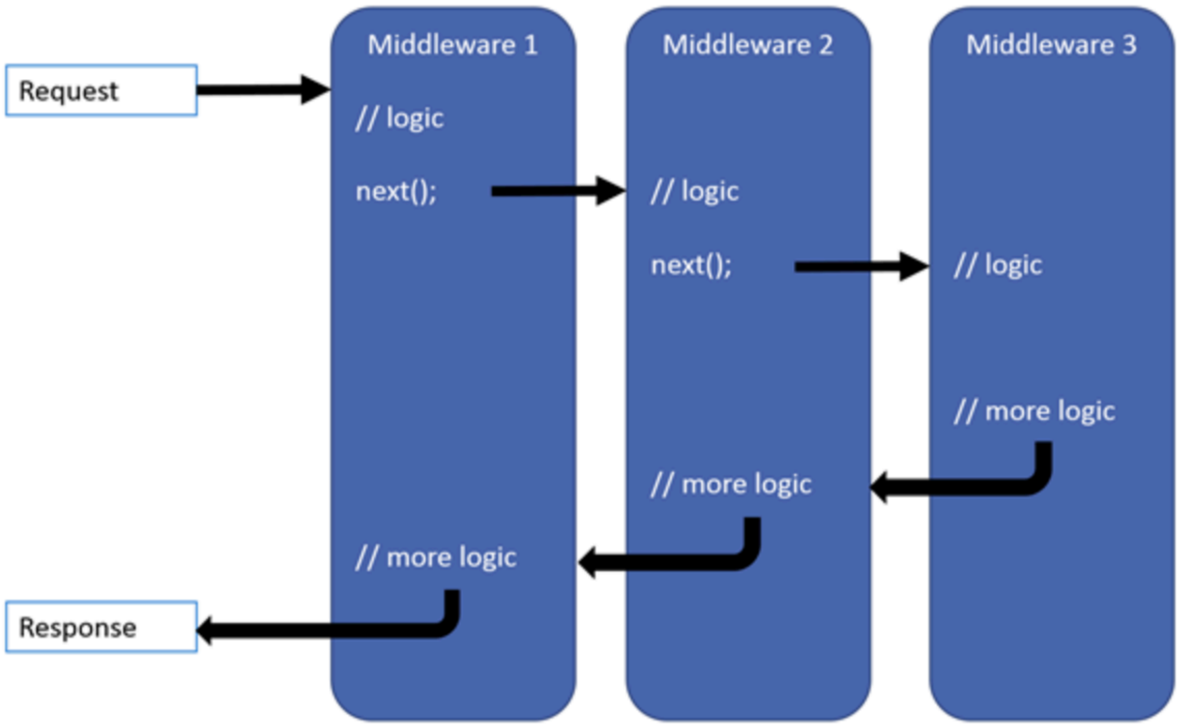
What is Middleware?
Middleware is software that lies between an operating system and the applications running on it. Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications. It's sometimes called plumbing, as it connects two applications together so data and databases can be easily passed between the "pipe." Using middleware allows users to perform such requests as submitting forms on a web browser, or allowing the web server to return dynamic web pages based on a user's profile.

What extension methods it uses?
The Run method is an extension method on IApplicationBuilder and accepts a parameter of RequestDelegate.
Use () extension method. It is similar to Run() method except that it includes next parameter to invoke next middleware in the sequence.

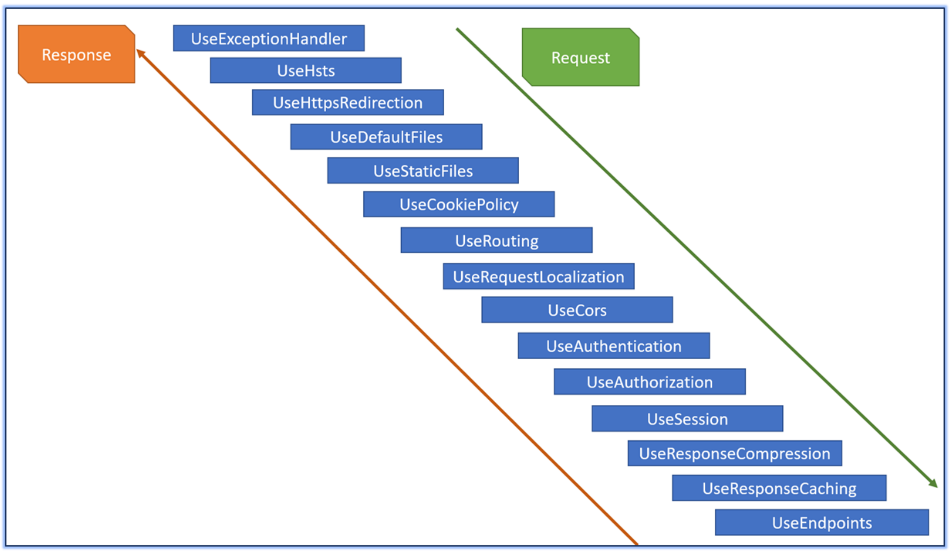
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   app.Use(async (context, next) =>
   {
      await context.Response.WriteAsync("Hello World From 1st Middleware!");

      await next();
   });

   app.Run(async (context) =>
   {
      await context.Response.WriteAsync("Hello World From 2nd Middleware");
   });
}
```

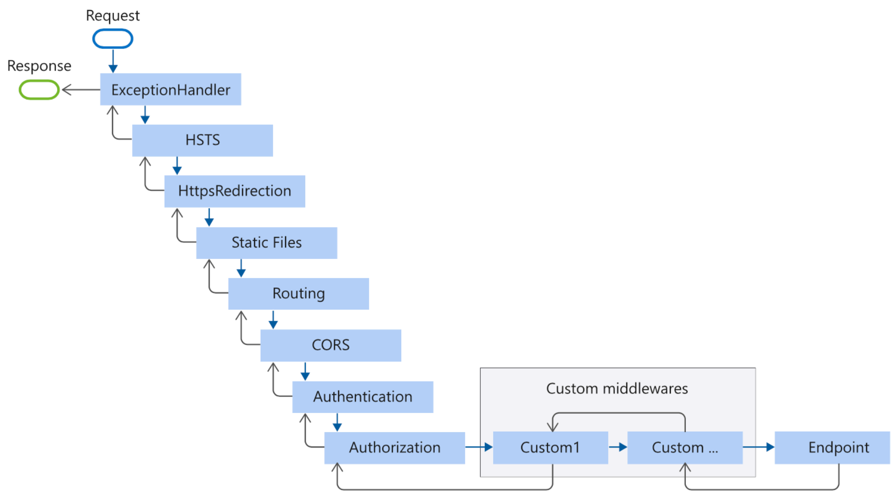
Middleware order



How do you call a delegate, when it doesn't pass a request to the next delegate?
When a delegate doesn't pass a request to the next delegate, it's called short-circuiting the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a terminal middleware by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their next.Invoke statements.

Request order





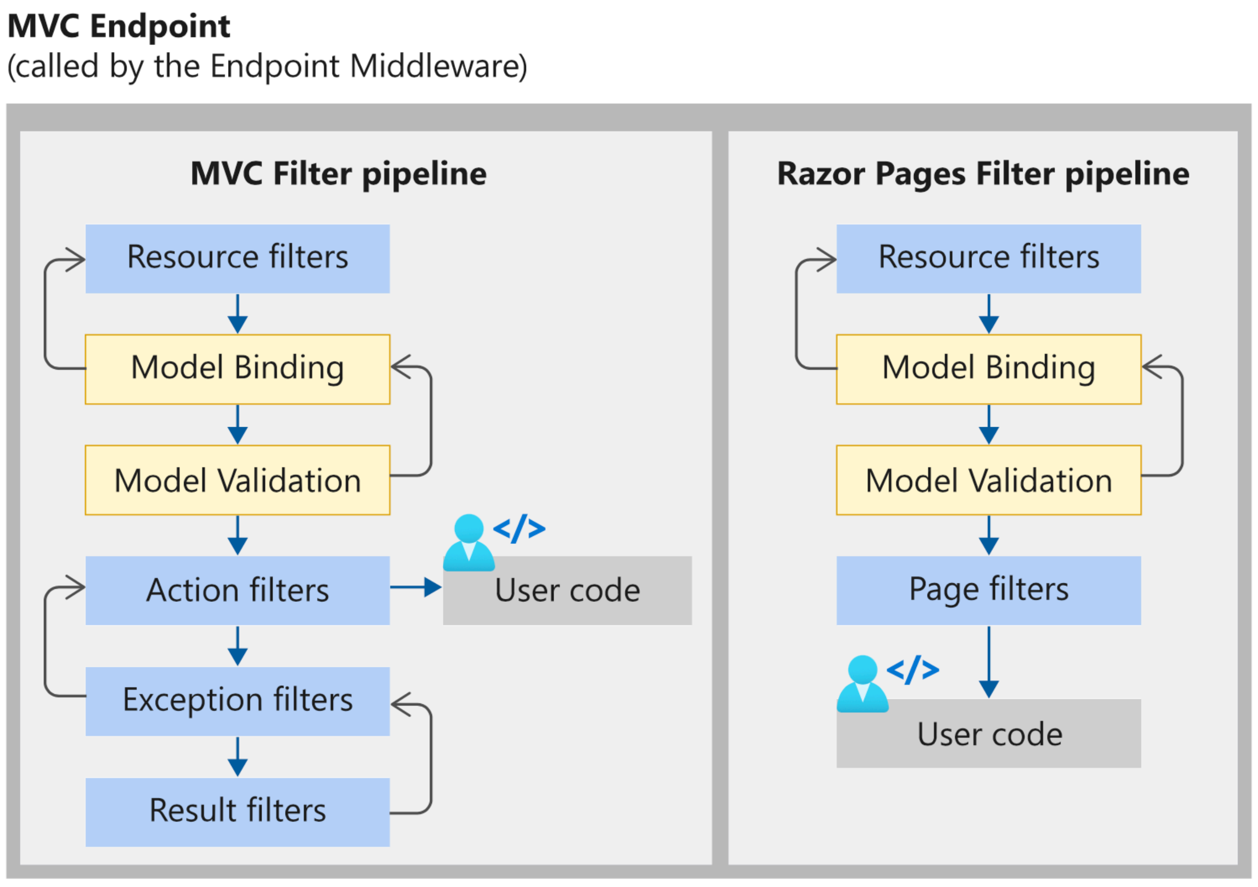
When a request is received, generally first middleware configured is exception middleware. This is because it can handle any exception from any other parts of request processing. It becomes global exception handler for the whole app. In development mode though, *DeveloperExceptionPage* middleware can be used instead of exception handler middleware.

Then *HSTS middleware* can be used to add the `Strict-Transport-Security` header. Optionally, *UseHttpsRedirection* might be used to redirect users to HTTPS endpoint.
Note that UseHttpsRedirection middleware is not really recommended with Web APIs because this middleware sends redirect status code to client and if the web API client is not a browser, then the client might respect the redirect status codes.
Default files middleware is to rewrite the URL if the endpoint is not specified. Again this is not a web API related middleware, it makes more sense to have this in MVC apps. Hence it can be skipped for Web APIs.
On the same lines, Static Files middleware is to serve static files (*js, CSS, images, etc*) for a web app. It is a terminal middleware. So, any static files request would not go via subsequent middleware pipeline. This middleware also does not make sense for Web APIs.
Cookie policy middleware conforms the app to the EU General Data Protection Regulation (GDPR) regulations.
Routing middleware (UseRouting) is to add the endpoint information to the request object. This information is later used by other middlewares. Some of those middlewares may include:

- CORS middleware for applying CORS policies
- Authentication middleware to check if the user is authenticated
- Authorization middleware to check if user is authorized to perform current operation

Session middleware is to enables and helps in maintaining server side session state. Generally this middleware might not be required as web APIs are generally stateless.
Routing middleware (UseEndpoints) is generally the last middleware in the pipeline which hands over the incoming request to appropriate endpoint (i.e. controller action) for processing. This middleware can use the patterns specified while configuring middleware to identify the endpoint.
After result is generated, response caching middleware can be used to store the processed response in cache. Optionally, response compression can also be used. Response compression middleware is generally not recommended if the "real host" (e.g. IIS or NGINX or apache) of .NET core app can support it. Response compression applied by actual web servers are generally more efficient than the middleware.

How does the order affect component performance?



The order that middleware components are added in the startup.configure method defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is **critical** for security, performance, and functionality

Explain *Developer Exception Page Middleware, UseExceptionHandler, UseHsts, UseHttpsRedirection, UseStaticFile, UseCookiePolicy, UseRouting, UseAuthentication, UseAuthorization, UseSession ja UseEndpoints with MapRazorPages, UseSpaStaticFiles?*

Developer Exception Page Middleware reports app runtime errors.
Exception Handler Middleware catches exceptions thrown in the following middlewares.
HTTP Strict Transport Security Protocol (HSTS) Middleware UseHsts adds the Strict-Transport-Security header.
HTTPS Redirection Middleware redirects HTTP requests to HTTPS.
Static File Middleware returns static files and short-circuits further request processing.
Cookie Policy Middleware conforms the app to the EU General Data Protection Regulation (GDPR) regulations.
Routing Middleware to route requests.
Authentication Middleware attempts to authenticate the user before they're allowed access to secure resources.
Authorization Middleware authorizes a user to access secure resources.
Session Middleware establishes and maintains session state. If the app uses session state, call session Middleware after Cookie Policy Middleware and before MVC Middleware.
Endpoint Routing Middleware UseEndpoints with MapRazorPages to add Razor Pages endpoints to the request pipeline.
UseSpaStaticFiles Configures the application to serve static files for a Single Page Application (SPA).

Choose 5 Built-in Middleware and describe them.

Health check - Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability. ASP.NET Core offers Health Checks Middleware and libraries for reporting the health of app infrastructure components.

Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for a variety of real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Request Localization - A multilingual website allows the site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures. Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas. Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable
2. Provide localized resources for the languages and cultures you support
3. Implement a strategy to select the language/culture for each request

Session - Provides support for managing user sessions.
HTTP is a stateless protocol. By default, HTTP requests are independent messages that don't retain user values. This article describes several approaches to preserve user data between requests.

WebSockets - Enables the WebSockets protocol.
This is a is a protocol that enables two-way persistent communication channels over TCP connections. It's used in apps that benefit from fast, real-time communication, such as chat, dashboard, and game apps.

URL Rewrite - Provides support for rewriting URLs and redirecting requests.
URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses aren't tightly linked. URL rewriting is valuable in several scenarios to:

- Move or replace server resources temporarily or permanently and maintain stable locators for those resources.
- Split request processing across different apps or across areas of one app.
- Remove, add, or reorganize URL segments on incoming requests.
- Optimize public URLs for Search Engine Optimization (SEO).
- Permit the use of friendly public URLs to help visitors predict the content returned by requesting a resource.
- Redirect insecure requests to secure endpoints.
- Prevent hotlinking, where an external site uses a hosted static asset on another site by linking the asset into its own content.