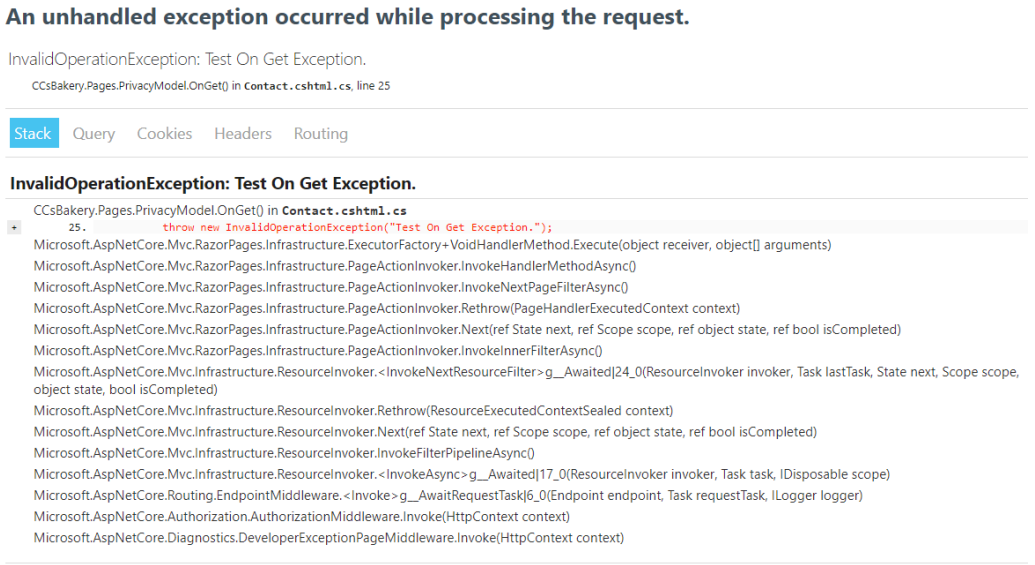
codecademy

ASP.NET: Middleware

Understanding ASP.NET Developer Exception Pages

`UseDeveloperExceptionPage()` is a method which provides an exception page specifically designed for developers. The method should be placed before any middleware components that are catching exceptions. Some of the information displayed includes the stack trace, any query string parameters, cookies, headers, and routing information.



Understanding ASP.NET UseExceptionHandler Component

The `UseExceptionHandler()` component can be used in production environments to catch and log errors and route users to a general error page without exposing sensitive details about the application. The `UseExceptionHandler()` component has several overloads, but a simple way of using it is to pass in the name of the page (as a string) that should display if an exception is thrown.

```
if (env.IsDevelopment())
  app.UseDeveloperExceptionPage();
else
  app.UseExceptionHandler("/Error");
```

Redirecting Requests with ASP.NET Middleware

The `UseHttpsRedirection()` method is the middleware component used to capture HTTP requests and redirect them to the more secure HTTPS protocol. Since the redirection is done in the app configuration, the user never even has to know the redirection occurred.

Enabling ASP.NET Endpoints

`UseRouting()`, must be called to compare the HTTP request with the available endpoints and decide which is the best match. `UseEndpoints()` then calls `MapRazorPages()` with a lambda expression to register the endpoint, and then executes the selected delegate that matches our HTTP request.

```
app.UseRouting();
app.UseEndpoints(endpoints => {
  endpoints.MapRazorPages();
});
```

Understanding the ASP.NET UseStaticFiles Component

Static files make up an important part of the application — they often contain HTML, CSS, and JavaScript code that controls how the application looks and behaves. The `UseStaticFiles()` method is available to ensure static file content is rendered alongside the HTML for our web applications.

Understanding the ASP.NET UseAuthorization Component

The `UseAuthorization()` component checks the user's request against their authorization status. If the authorization check passes, this component will pass the request to the next component in the pipeline. Otherwise, it will short-circuit the pipeline and either present the user with a login page or an error.

Understanding ASP.NET middleware

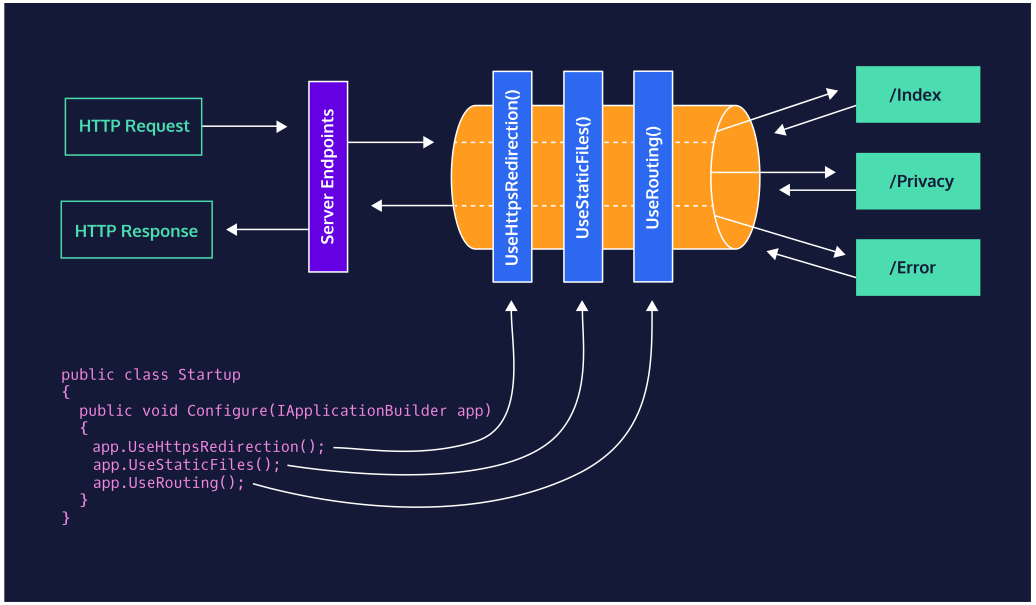
Web applications use a series of components to route each HTTP request to its destination and then return an appropriate response to the user. This series of components is organized in a pipeline which is collectively known as *middleware*.

Adding ASP.NET Middleware Components

The `Configure()` method is called from the `Startup` class. `Configure()` calls various `app.UseX()` methods to build the middleware pipeline.

Ordering ASP.NET Middleware

Middleware components are called in sequence. The order in which components are called is very important and is determined by where the component appears in the `Startup.Configure()` method.



Accessing Built-in ASP.NET Components

The `IApplicationBuilder` interface defines the built-in middleware methods. These methods are defined by using the format `UseX()` with `X` describing the action performed by the method.

```
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
```

Adding custom middleware components

The `IApplicationBuilder` interface provides a generic `Use()` method which can be used to process custom middleware components and call the next component in the pipeline. Multiple middleware components can be chained together with the `Use()` method, which accepts two parameters, the HTTP context and the next middleware component to be called.

```
app.Use(async (context, next) => {
  await context.Response.WriteAsync("Custom middleware!");
  await next();
});
```

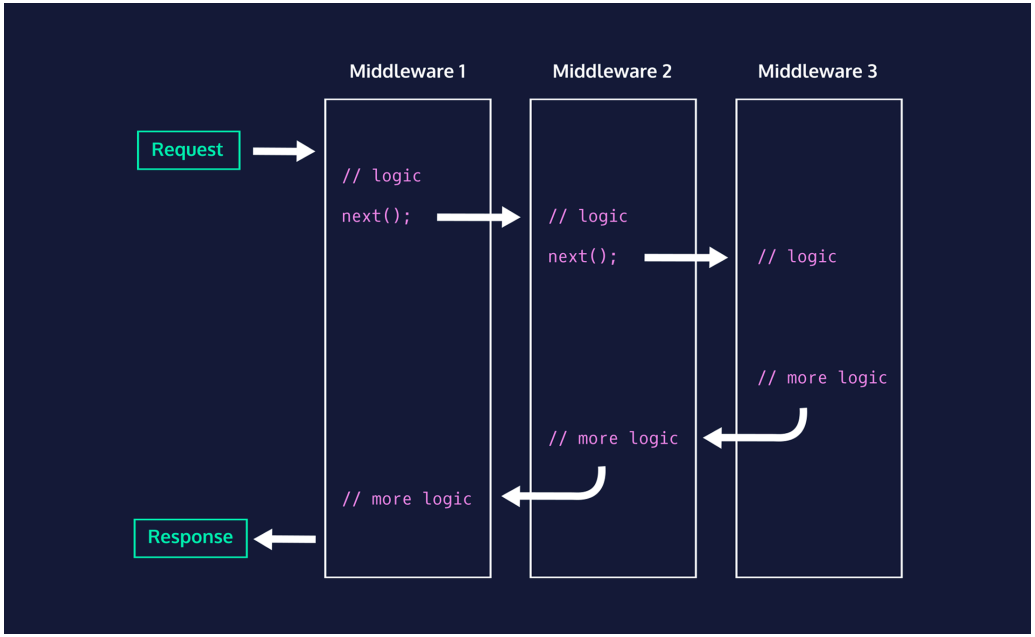
Adding terminal ASP.NET middleware

`IApplicationBuilder.Run()` is a terminal middleware component which begins returning the HTTP response. `Run()` can be added to the end of the request pipeline since any delegates after this component will not be executed.

```
app.Run(async (context) => {
  await context.Response.WriteAsync("Terminal middleware!");
});
```

Understanding ASP.NET Nested Structure

Proper ordering of middleware components is important. The middleware request pipeline has a nested structure where each component can perform operations before and after the next component.



Print Share