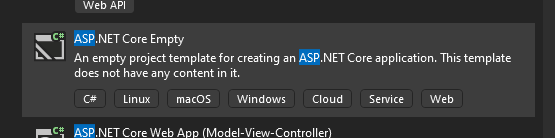
# Giới thiệu về ASP.Net Core

* ASP là 1 framework mã nguồn mở của .net cho phép chúng ta tạ ra những ứng dụng web, những ứng dụng chạy trên máy chủ để các client kết nối đến, truy vấn và lấy thông tin VD như website, web service,…

## Kiến thức cần thiết để thao tác với asp.net

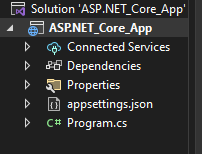
1. C#
   1. DI (Dependency Injection)
   2. Delegate
   3. Lambda
   4. Attribute
   5. Linq
   6. Asynchronous
   7. HTTP
   8. ADO.NET/Entity framework
2. HTML
3. CSS
4. JavaScript (JQuery)
5. SQL
   1. Query
   2. MS SQL Server / MySQL

## Tạo ứng dụng ASP.NET đầu tiên

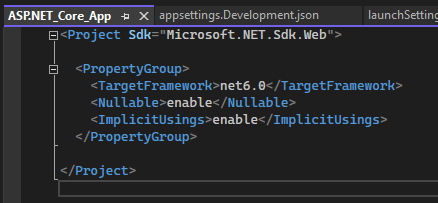


Tôi đang sử dụng .net 6.0

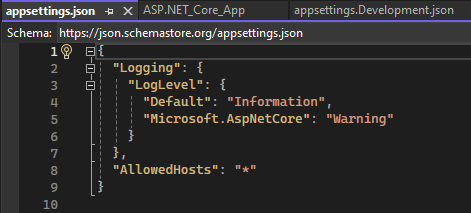
File project dự án như sau:



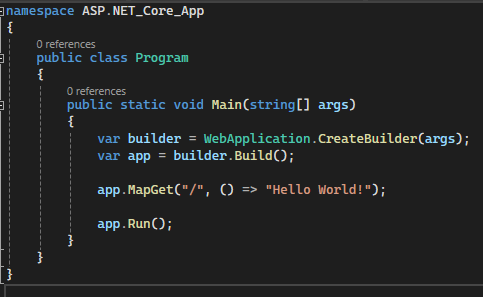
Trong file project của dự án:

Project được cấu hình sử dụng SDK là NET.Sdk.Web, tức là nó đã thíc hợp vào đây tất cả các thư viện để tạo ra ứng dụng web

ứng dụng dựa trên nền tảng net6.0

File appsetting.json

Chứa thông số cấu hình mà khi ứng dụng chạy nó sẽ được nạp vào



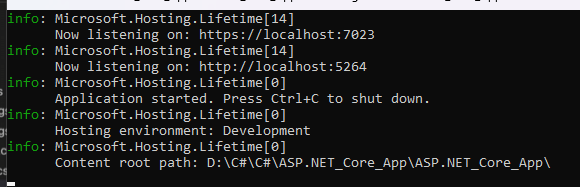
Program.cs

Có phương thức Main có nhiệm vụ cấu hình và tạo ra host và sau đó chạy host đó

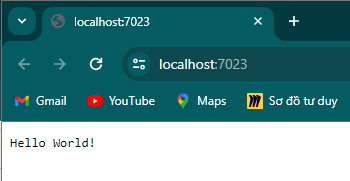
Và khi ứng dụng hoạt động host sẽ lắng nghe trên cổng nào đó, để các client kết nối với giao thức http đến nó nhằm truy vấn lấy các thông tin

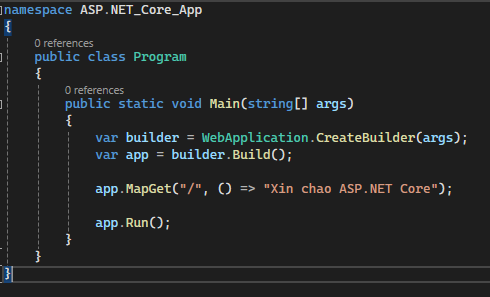
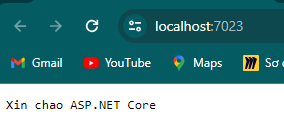
Nhiệm vụ của ta là trả về cho các client các thông tin thông qua các http message

Chạy dự án:

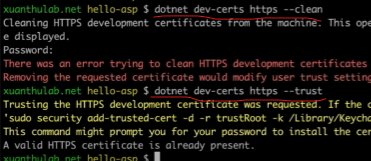


Kết quả:



Để cấu hình SSL tại localhost nếu cần



## Điều kiện để tạo ra 1 ứng dụng ASP.NET

1. Tạo 1 Host object được triển khai từ interface IHost
   1. Dependency Injection (ID): IserviceProvider (ServiceCollection)
   2. Logging (ILogging)
   3. Cofiguration
   4. IHostedService => StartAsync : Run HTTP Server (Kestrel Http)

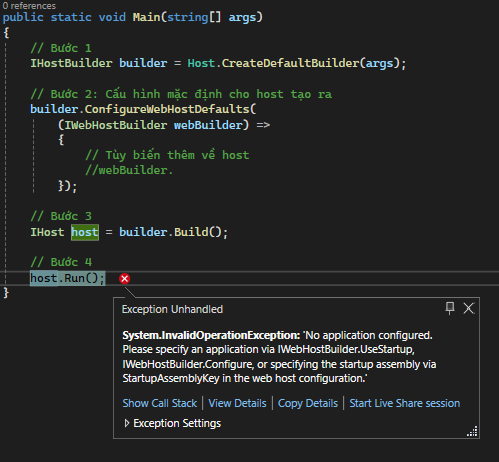
Để tạo 1 Host ta cần tạo ra 1 đối tượng mà từ đối tượng đấy sinh ra host đó là 1 đối tượng triển khai từ **IHostBuilder**

**Bước 1:**  Tạo **IHostBuilder**

**Bước 2:** Dùng **IHostBuilder** để thiết lập các cấu hình, đăng ký các dịch vụ. Chúng ta làm việc này bằng cách gọi phương thức **ConfigureWebHostDefault()** của **IHostBuilder**

**Bước 3:** Gọi phương thức **IHostBuilder**.**Build**() của **IHostBuilder** phương thức này sẽ tạo ra cho chúng ta một đối tượng **Host**

**Bước 4:** gọi phương thức **Host.Run()** để chạy host, phương thức run được gọi nó sẽ gọi đồng thời phương thức StartAsync của IhostedService



Khi chúng ta chạy host phát sinh exception vì khi host này khởi chạy nghĩa là máy chủ mặc định khởi chạy, khi nó chạy nó sẽ lắng nghe các yêu cầu Http gửi đến, vấn đề là khi nhận được http message gửi đến thì các request đấy phải chuyển cho đoạn code nào xử lý. Thì ở đây ta chưa cấu hình đoạn này

Trong ứng dụng Asp.Net khi nhận được các request thì các request đó phải được chuyển xử lý và đi qua các đoạn code để cuối cùng trả về các response

Những thành phần mà request phải đi qua gọi là **pipeline**

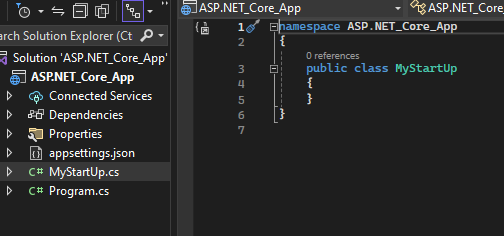
Pipiline là một chuỗi middleware

Giờ ta sẽ xây dựng pipeline thông qua phương thức: **ConfigureWebHostDefaults**

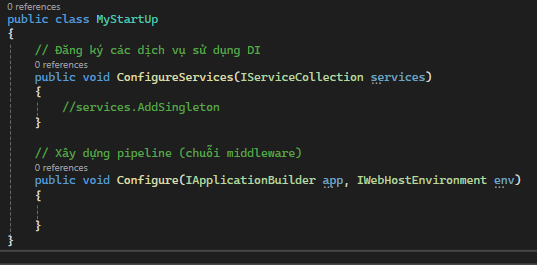


NameClassStartUp: là lớp sử dụng để xây dựng lên pipeline

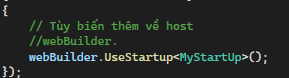
Ta sẽ tạo class startup này:



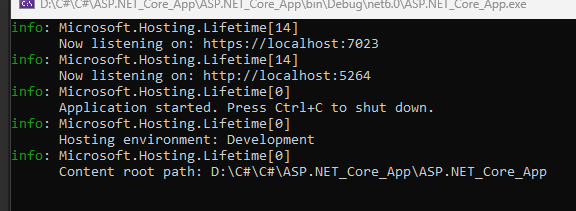
Để class MyStartUp này sử dụng được UseStartUp thì bên trong nó ta phải khai báo có tối thiểu 2 phương thức



Lúc này ta có thể sử dụng:

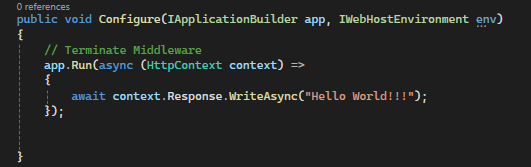


Chạy lại ứng dụng:

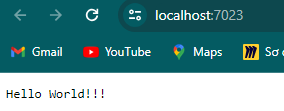


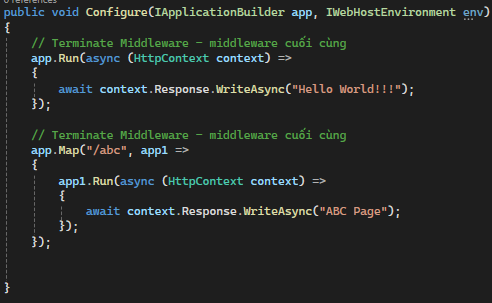
Ứng dụng đã chạy và đang lắng nghe ở local 7023

## Sử dụng middleware xây dựng pipeline

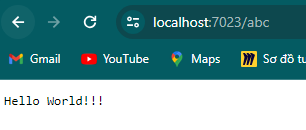


Kết quả:



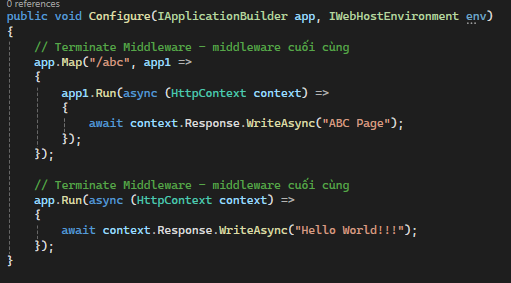


Kết quả:

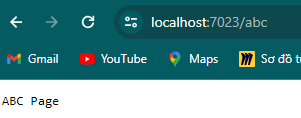


ở đây do ta có 2 middleware cuối nên khi chương trình chạy đến middleware đầu tiên thỏa mãn nó sẽ chạy code và bỏ qua những middleware còn lại

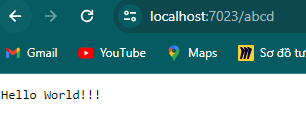
Ta sẽ sửa lại như sau:



Kết quả:



còn các đại chỉ khác sẽ đều trả về



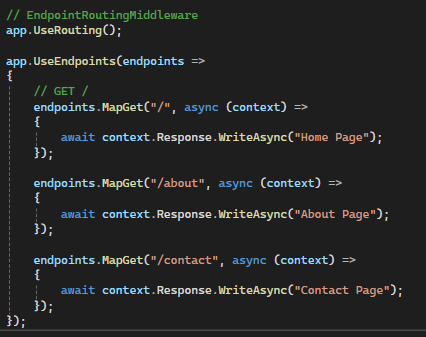
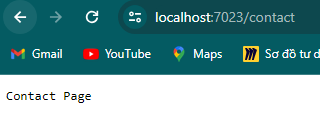
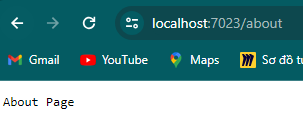
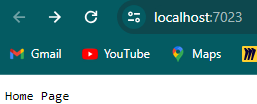
Trong .net thường sử dụng 1 middleware nữa là Routing dùng để điều hướng request và tạo ra các endpoint

Thực tế rất ít khi ta sử dụng endpoint được viết trực tiếp từ đối tượng **IApplicationBuilder**

Để sử dụng middleware routing ta sử dụng phương thức 

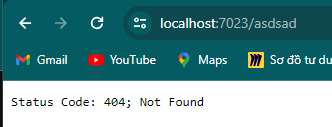
Khi request đi qua middleware này nó sẽ phân tích địa chỉ truy cập và điều hướng request đó đi theo 1 luồng tới một endpoint nhất định

Sau khi có UseRouting(), ta phải định nghĩa các Endpoint của nó bằng phương thức UseEndpoint()

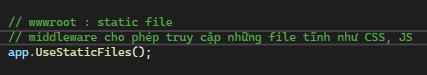


## Xây dựng trang 404



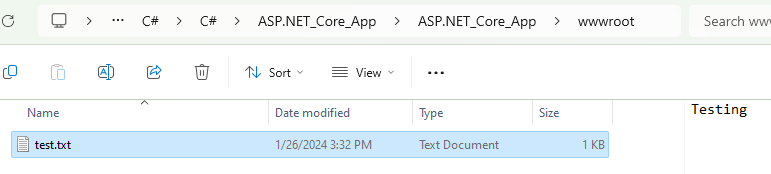


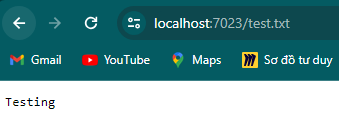
## Static file



Mặc định sẽ public file tĩnh trong folder wwwroot

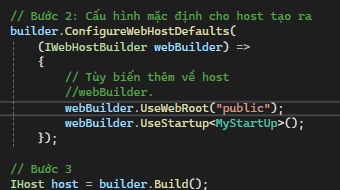
Giả sử có file txt như sau:





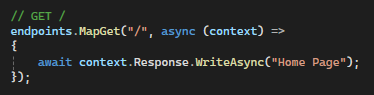
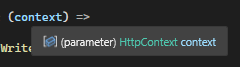
Lưu ý: khi sử dụng staticMiddleware chúng ta đảm bảo chương trình sẽ đi qua middleware này trước khi đi qua routing bằng cách đưa nó lên trên đầu.

Chúng ta có thể đổi thư mục mặc định wwwroot thành thư mục khác bằng cách :



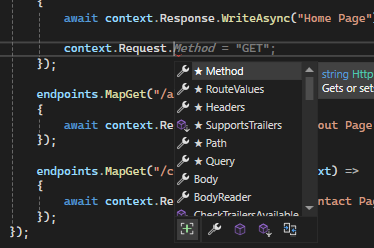
## Đối tượng context

Trong EndpointRoutingMiddleware

Chúng ta có một đối tượng là http context, đối tượng này chứa tất cả thông tin về request, yêu cầu cảu client gửi đến server, đồng thời cũng chứa đối tượng biểu diễn http message response mà sẽ trả về cho client

Thông tin về request thế hiện cho thuộc tính:



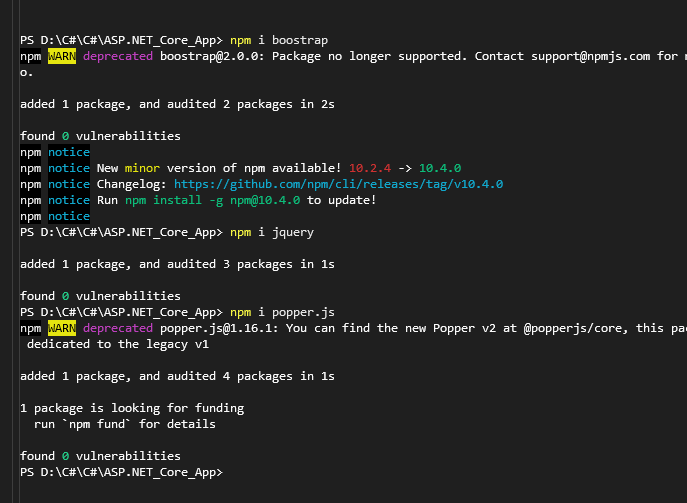
## Tích hợp thư viện JS/CSS

Sử dụng chương trình quản lý gói của node (npm) vì vậy ta cần cài đặt nodejs

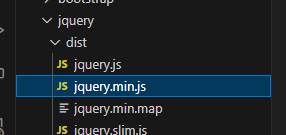
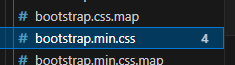
Sau khi cài đặt nodejs



Ta thực hiện cài đặt bootstrap, jquery và popper.js

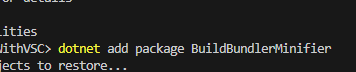


Sau khi tải xong các package, chúng ta chỉ cần sử dụng các file đã đóng gói là

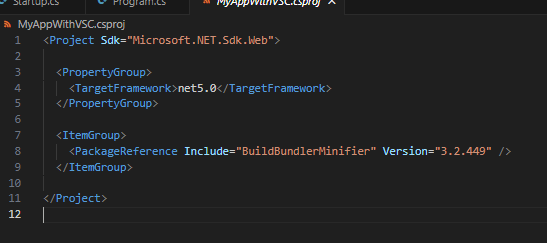
 

Việc chúng ta cần làm đó là lấy được fiel này copy sang thư mục static để sử dụng

Để tự động hóa quá trình copy như vậy ta có thể sử dụng 1 package của .net là **BuildBundlerMinifier**

Ta thực hiện cài đặt package: 

Sau khi cài đặt dự án của ta sẽ có thêm package này:



Tạo file bundleconfig.json ở thư mục gốc và cập nhật vào nó nội dung

[

{

"outputFileName": "wwwroot/css/bootstrap.min.css",

"inputFiles": [

"node\_modules/bootstrap/dist/css/bootstrap.min.css"

]

},

{

"outputFileName": "wwwroot/js/bootstrap.min.js",

"inputFiles": [

"node\_modules/bootstrap/dist/js/bootstrap.min.js"

]

},

{

"outputFileName": "wwwroot/js/jquery.min.js",

"inputFiles": [

"node\_modules/jquery/dist/jquery.min.js"

]

},

{

"outputFileName": "wwwroot/js/popper.min.js",

"inputFiles": [

"node\_modules/popper.js/dist/popper.min.js"

],

"minify": {

"enabled": false,

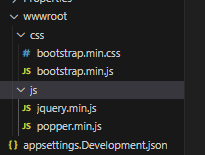
"renameLocals": true

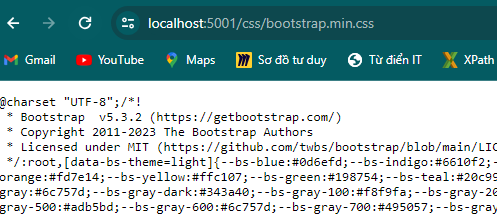
}

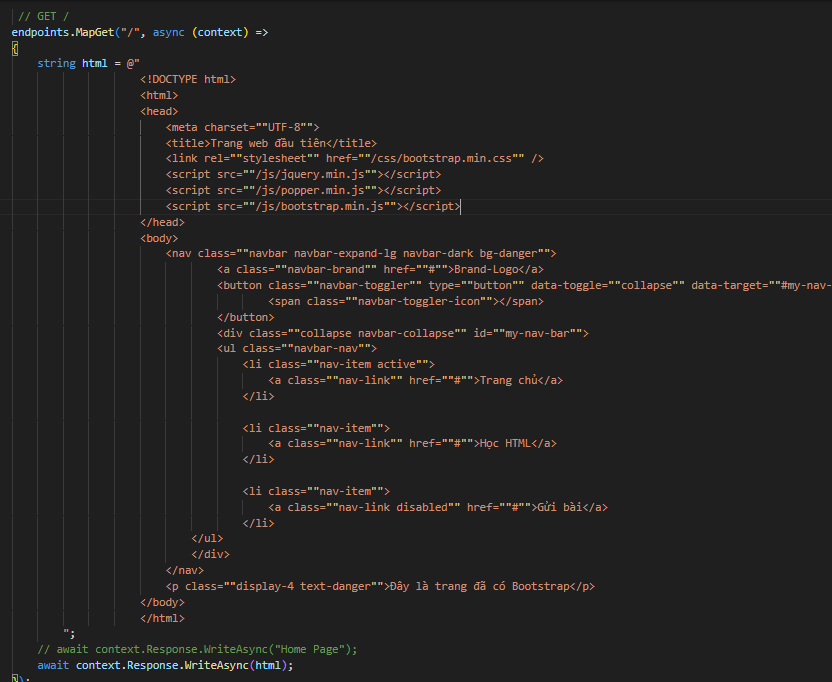
}

]

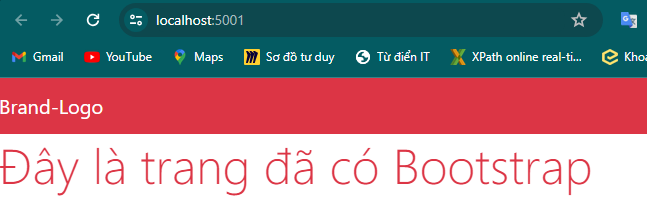
Sau khi chạy lệnh dotnet build chương trình sẽ sao chép file như sau:

 khi đó ta có thể truy cập





Kết quả:

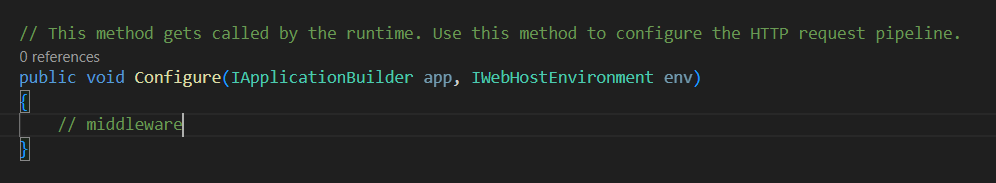


# Middleware

## Pipeline

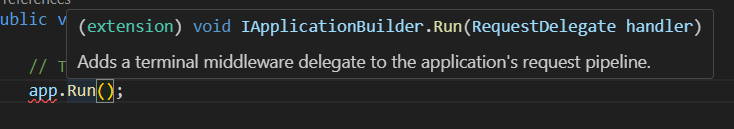


Pineline gọi tới Startup.cs là nơi cấu hình các middleware  
Trong file Startup.cs Middleware được cấu hình ở trong phương thức Configure

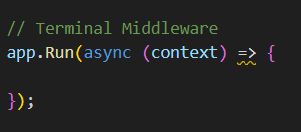
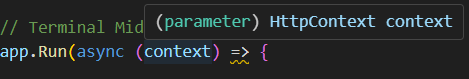


Khởi tạo 1 middleware Sử dụng phương thức Run của đối tượng **IApplicationBuilder.**

Phương thức này nhận tahm số là 1 RequestDelegate là 1 tác vụ

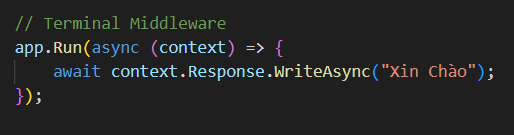


Do RequestDelegate là 1 tác vụ nên ta cần sử dụng async

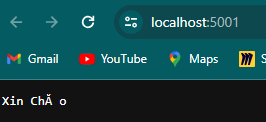
Delegate này nhận tham số là 1 đối tượng HttpContext

Đối tượng context này chứa cả Request và Response

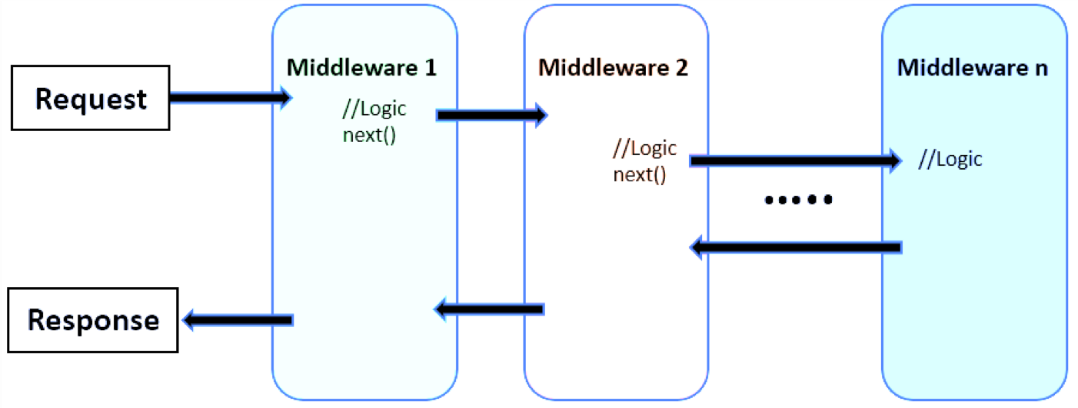
Lúc này Pipe chỉ có duy nhất 1 middleware, vì vậy mọi truy vấn đều đi qua middleware này



Kết quả:

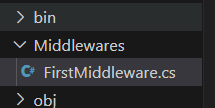
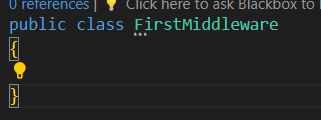


## Middleware

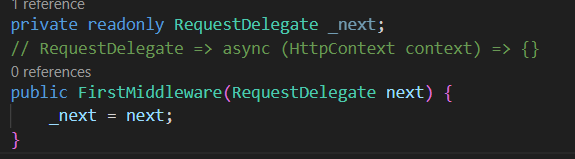
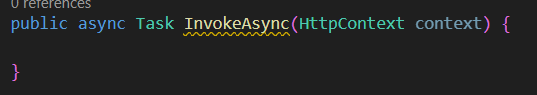
* Là 1 module của ứng dụng
* Có chức năng nhận truy vấn, xử lý các truy vấn, trả về response
* Trong ứng dụng có thể có nhiều middleware
* Các middleware kết nối với nhau hình thành lên Pipeline
* Thông thường các middleware sau khi nhận request xử lý và chuyển request và response cho middleware phía sau lần lượt sau đó sẽ trả ngược kết quả về các middleware phía trước
* Nếu 1 middleware sau khi xử lý nhưng không gọi middleware tiếp theo thì gọi là terminal middleware

## Tạo middleware

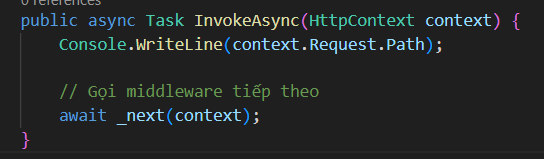
Ta sẽ tạo 1 folder chứa middleware

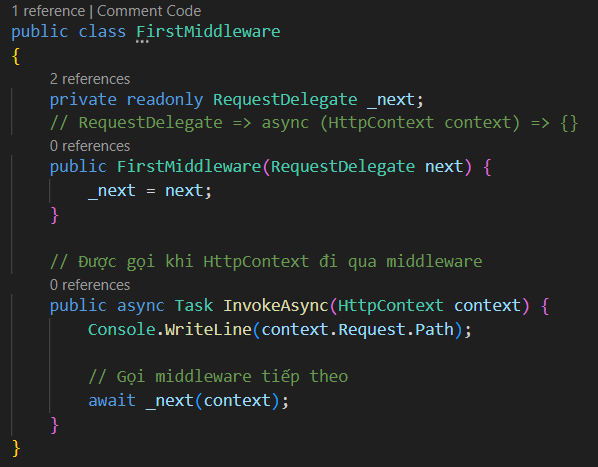
 

Để một class là một middleware ta cần khai báo như sau:

* Phương thức khởi tạo phải nhận tham số khởi tạo là RequestDelegate
  + Tham số này là tham chiếu để thi hành middleware phía sau trong pipeline
  + Tham số này có dạng async (HttpContext context) => {}
  + Vì là 1 phương thức bất đồng bộ trả về 1 task nên cần ghi lại vào 1 biến
  + 
* Phương thức bất đồng bộ **InvokeAsync** trả về Task và nhận tham số là 1 HttpContext
  + Được gọi khi HttpContext đi qua middleware trong pipeline
  + Nhiệm vụ của middleware sẽ được xử lý tỏng phương thức này
  + 

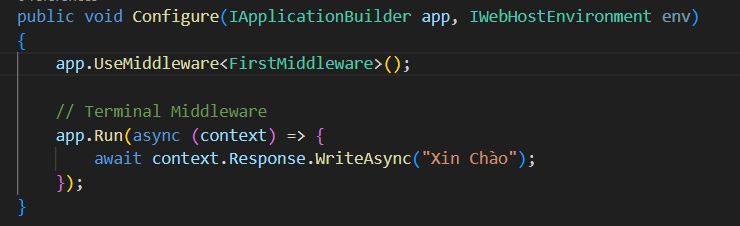
Ở đây ta làm 1 nhiệm vụ đơn giản là hiển thị đại chỉ URL của request và sau đó gọi đến middleware tiếp theo



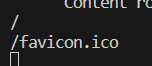
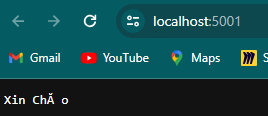


1 lớp khai báo như này được gọi là 1 middleware

Đăng ký lớp này vào pipeline ta làm như sau:



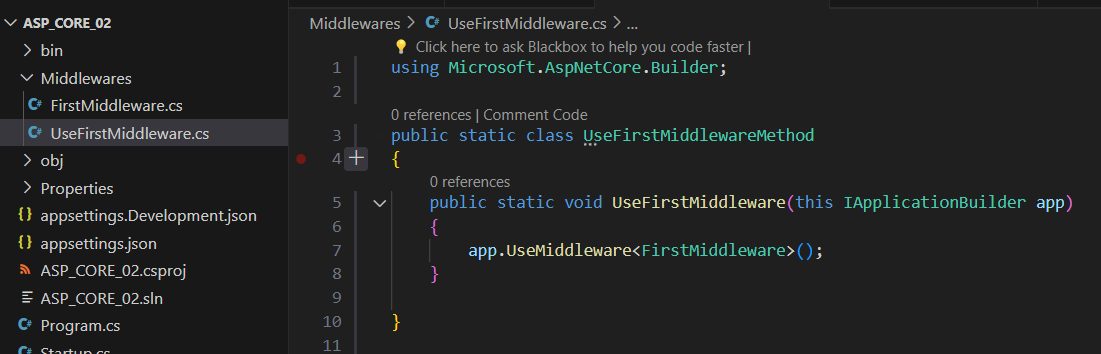
Kết quả:

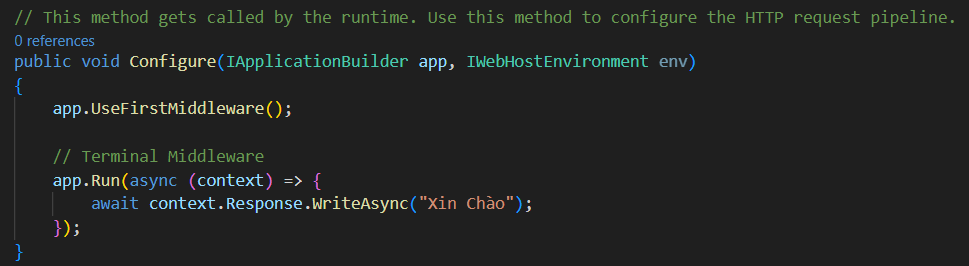
Lúc này pipeline của chúng ta có dạng như sau:

FirstMiddleware -> Terminal middleware

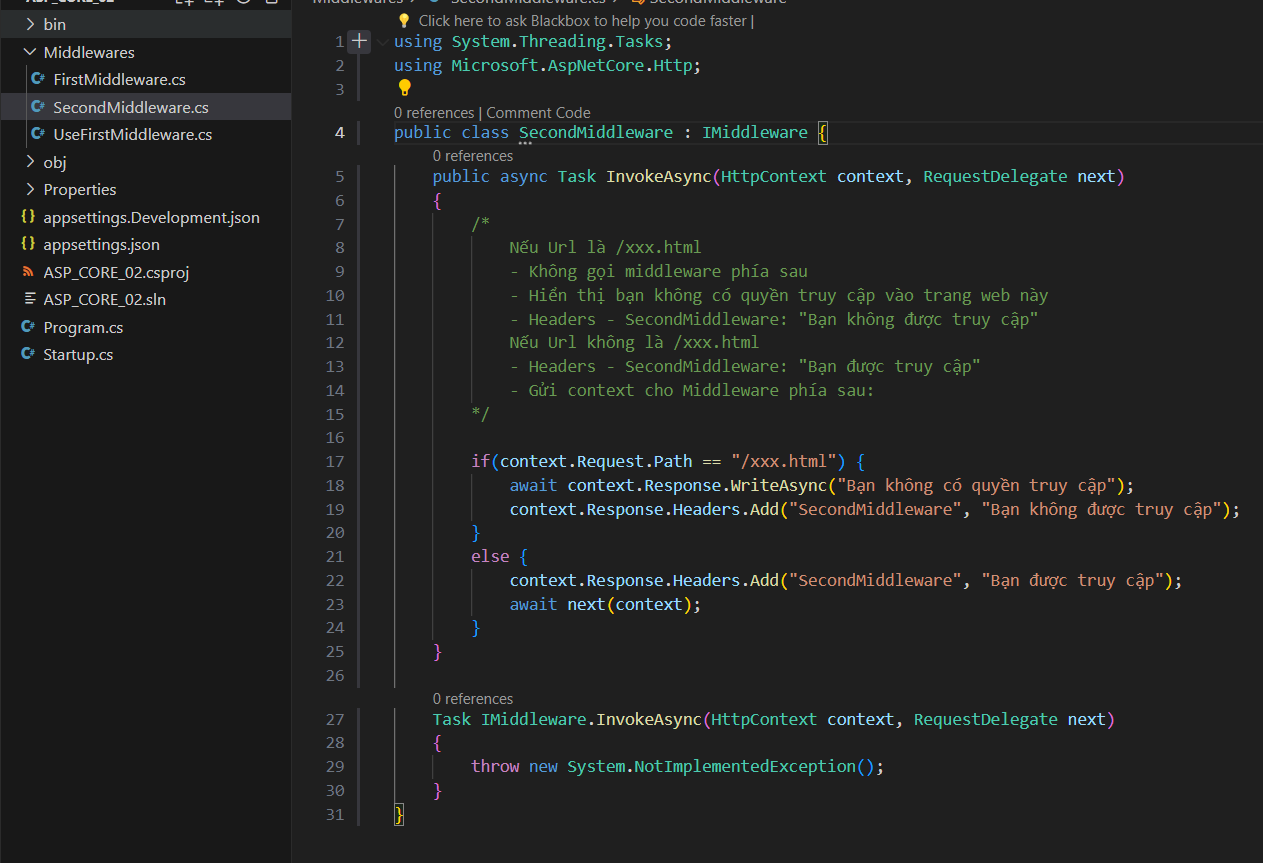
Xây dựng phương thức mở rộng cho **IApplicationBuilder** để thêm middleware



Sau khi xây dựng xong phương thức mở rộng ta có thể sửa lại code như sau:



Cách thứ 2 sử dụng middleware là sử dụng như 1 interface

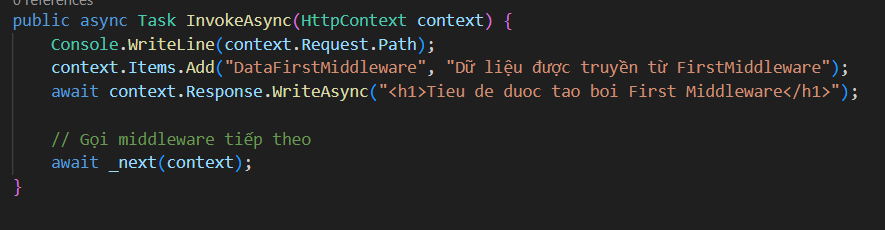


Ở đây có 1 lưu ý như sau:

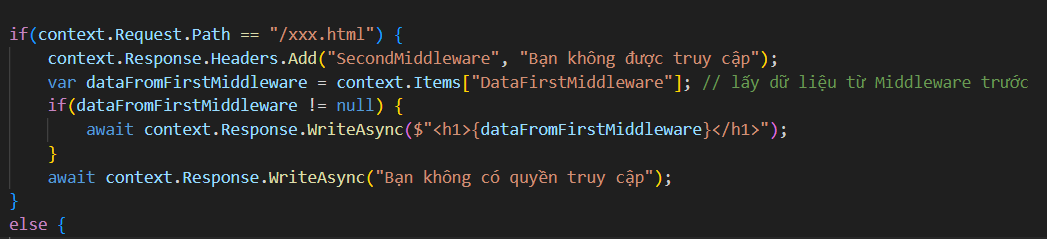
* Khi 1 response đã thiết lập nội dung mà sau đó tiếp tục thiết lập header thì sẽ phát sinh lỗi
* Vì vậy luôn luôn thiết lập Header trước khi thiết lập nội dung content
* Điều này áp dụng trên các middleware theo như code bên trên ta cũng đã set Header cho FirstMiddleware nên vì vậy nếu SecondMiddleware lại set Header thì sữ phát sinh lỗi vì vậy trong Middleware đầu tiên ta sẽ bỏ phần set Header

Để truyền dữ liệu giữa các Middleware ta làm như sau:

ở FirstMiddleware

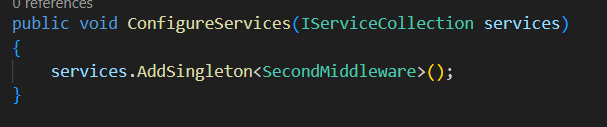


Ở SecondMiddleware:



Đối với Middleware được khai báo từ Imiddleware thì bao giờ cũng phải khia báo cho nó với 1 service

Tại phương thức **ConfigureServices**



Sau khi đăng ký service này ta có thẻ đưa nó vào pipeline:



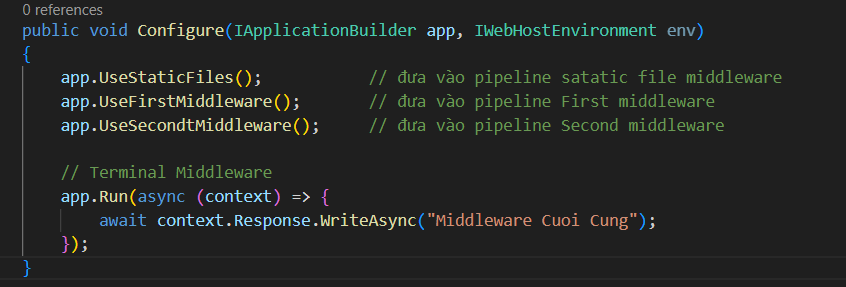
Hoặc chúng ta có thể mở rộng phương thức như trường hợp 1

## StaticFileMiddleware

Là middleware cho phép truy cập file tĩnh

Những file tĩnh được lưu mặc định trong folder wwwroot

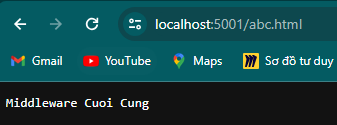
Để đưa StaticFileMiddleware vào pipeline ta làm như sau:



Lúc này trong pipeline các middleware sẽ đi theo thứ tự như sau:

StaticFileMiddleware -> FirstMiddleware -> SecondMiddleware -> Terminal Middleware

Lúc này nếu ta truy cập

Chương trình sẽ kiểm tra xem có file abc.html trong folder wwwroot hay không, nếu không nó sẽ tiếp tục đi qua các middleware tiếp theo

Nếu tồn tại file abc.hmtl thì sẽ trả về file abc.html mà không đi qua các middleware còn lại

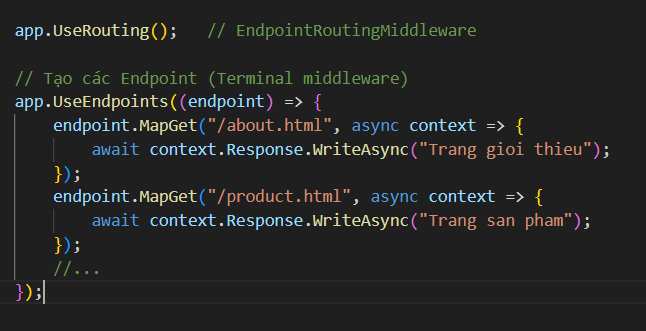
## EndpointMiddleware (Routing URL)

* EndpointMiddleware là 1 terminal middleware
* Nó điều hướng những truy vấn theo địa chỉ url đến các terminal middleware là các endpoint

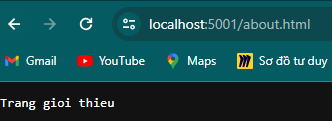
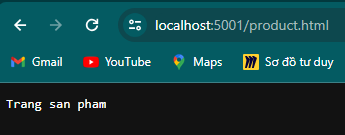


Khi HttpContext đi tới EndpointRoutingMiddleware, nó sẽ phân tích địa chỉ truy cập url của request đồng thời kết hợp các phương thức truy vấn như GET, POST, PUT,… thì nó sẽ điều hướng HttpContext đến những Endpoint mà những endpoint đó sẽ được định nghĩa ở phía sau

Tạo ra các endpoint được sử dụng bởi EndpointRoutingMiddleware ta làm như sau:



Kết quả:

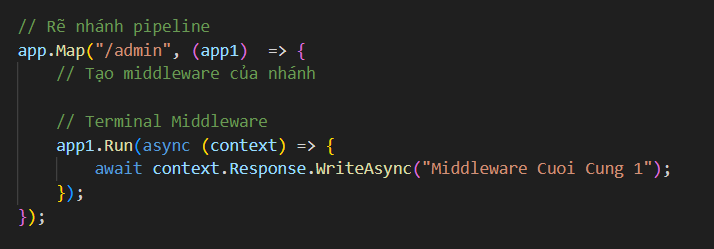
 

Lúc này trong pipeline các middleware sẽ đi theo thứ tự như sau:

StaticFileMiddleware -> FirstMiddleware -> SecondMiddleware -> EndpointRoutingMiddleware -> Terminal Middleware

Khi context đi qua EndpointRoutingMiddleware nó kiểm tra địa chỉ truy vấn trùng với endpoint nó sẽ kết thúc tại endpoint tương ứng

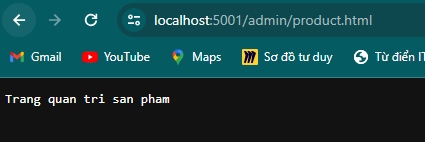
## Rẽ nhánh pipeline với IApplicationBuilder.Map

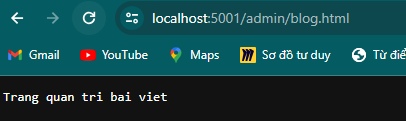


Trong nhánh con này ta có thể đưa vào các middleware tương tự như nhánh chính



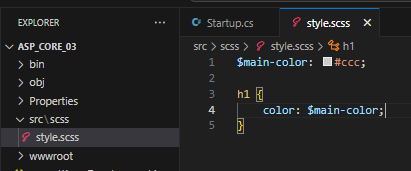
Kết quả:





# Webpack

Ta tạo 1 folder src chứa folder scss trong đó chứa file style.scss với nội dung như sau:



Bây giờ ta muốn biên dịch file scss này ra thành css rồi lưu trữ lại trong thư mục wwwroot, chúng ta sẽ sử dụng webpack với cách làm như sau:

## Cài đặt webpack

Webpack là 1 package của node.js vì vậy ta cần cài đặt nhu sau:

npm init -y # tạo file package.json cho dự án

npm **i** -D webpack webpack-cli # cà**i** đặt Webpack

npm **i** node-sass postcss-loader postcss-preset-env # cà**i** đặt các gó**i** để làm việc vớ**i** SCSS

npm **i** sass-loader css-loader cssnano # cà**i** đặt các gó**i** để làm việc vớ**i** SCSS, CSS

npm **i** mini-css-extract-plugin cross-env file-loader # cà**i** đặt các gó**i** để làm việc vớ**i** SCSS

npm install copy-webpack-plugin # cà**i** đặt plugin copy file cho Webpack

npm install npm-watch # package giám sát file thay đổ**i**

npm install bootstrap # cà**i** đặt **th**ư viện bootstrap

npm install jquery # cà**i** đặt Jquery

npm install popper.js # **th**ư viện cần cho bootstrap

Sau khi cài đặt:

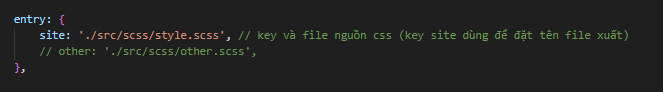
Điều tiếp theo cần làm đó là:

* Copy jquery.min.js từ package jquery ra thư mục wwwroot/js
* Copy popper.min.js từ package popper.js ra thư mục wwwroot/js
* Copy bootstrap.min.js từ package bootstrap ra thư mục wwwroot/js
* Biên dịch file src/scss/style.scss thành file wwww/css/style.min.css (đã gộp cả CSS của Bootstrap)

Để làm được điều này việc đầu tiên trong thư mục gốc của dự án ta tạo ra 1 file **webpack.config.js**

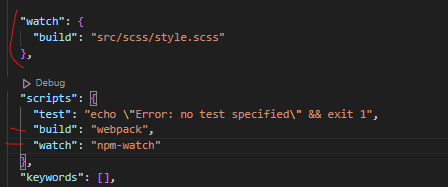
Trong file này ta copy code trong đường dẫn sau: [**https://gist.github.com/xuanthulabnet/ce1f473658567c09f277362694c90157**](https://gist.github.com/xuanthulabnet/ce1f473658567c09f277362694c90157)

Nội dung file này khởi tạo những đối tượng để webpack làm việc



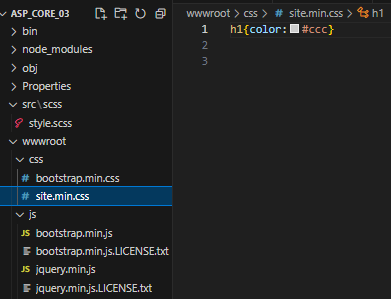


Sau đó ta cần cấu hình trong package.json

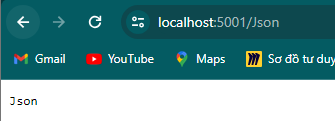
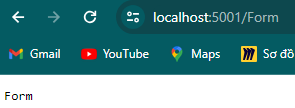


Để chạy chuong trình ta chạy lệnh:



Kết quả:

## Tạo các enpoint

Ta sẽ tạo các 5 endpoint với những chức năng khác nhau

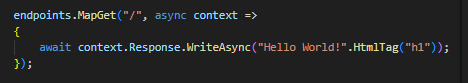
|  |  |
| --- | --- |
| **URL** | **Chức năng** |
| /RequestInfo | Đọc và hiển thị các thông tin vê request truy cập |
| /Encoding | Demo tính năng encoding dữ liệu khi kết xuất HTML |
| /Cookies | Demo – Đọc và ghi cookie |
| /Json | Demo trả về sữ liệu JSON |
| /Form | Demo Hiển thị form HTML xử lý đọc thông tin từ form, kể cả xử lý upload file |

Để thuận tiện hơn trong code ta sẽ sử dụng tạm thời 1 thư viện HtmlHelper

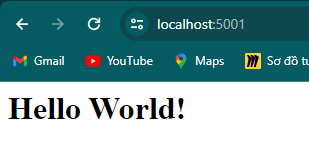
Tạo file HtmlHelper.cs và thêm đoạn code sau:

<https://github.com/xuanthulabnet/learn-cs-netcore/blob/master/ASP_NET_CORE/03.RequestResponse/HtmlHelper.cs>

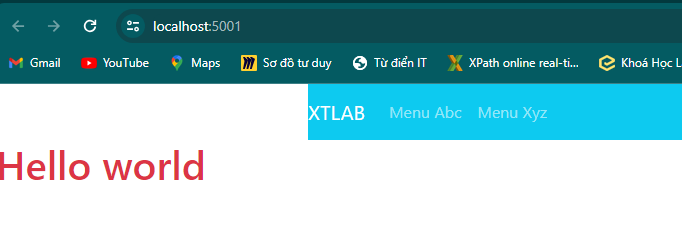
Sau khi thêm ta có thể viết như sau:

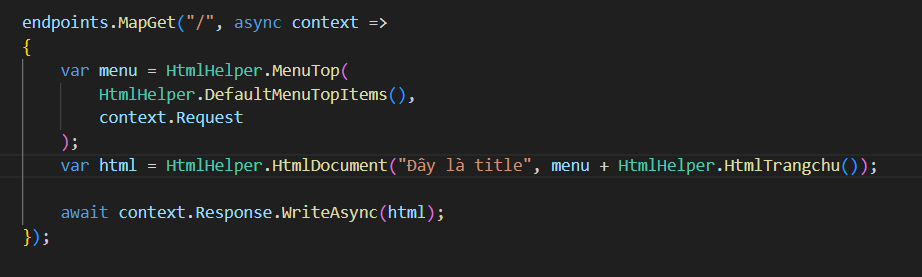


Kết quả:





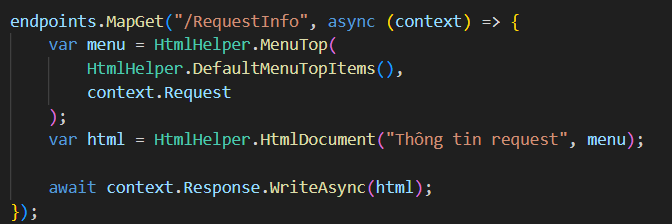


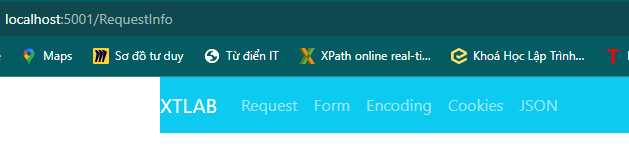




## Đọc các thông tin của Request

Tạo trang /RequestInfo

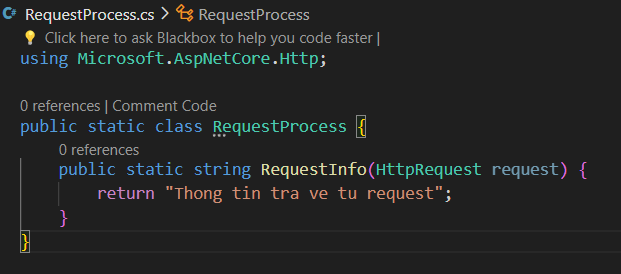
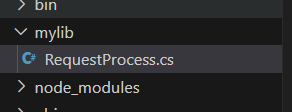


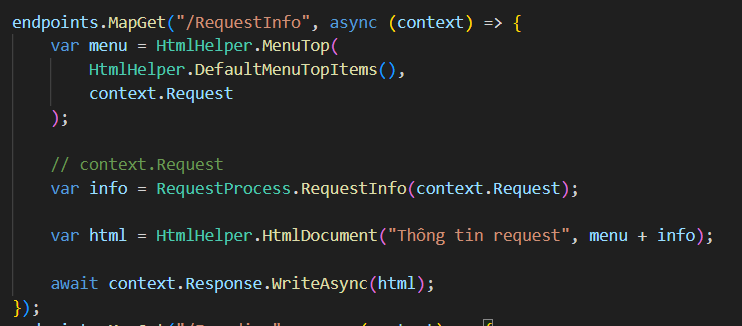


Khi 1 yêu cầu truy vấn gửi tới ứng dụng thì những thông tin của truy vấn được lưu trong đối tượng **context.Request**

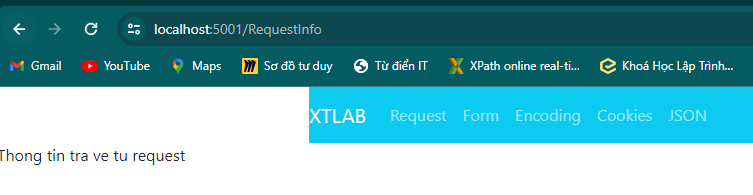
Chúng ta sẽ truyền đối tướng request này cho 1 phương thức mà tại đó chúng ta sẽ đọc những thông tin có được từ request

Chúng ta sẽ tạo 1 lớp tĩnh để xử lý request





Kết quả:



Trong phương thức RequestInfo ta sẽ khai báo các phần để lấy thông tin từ request

using System.Linq;

using System.Text;

using Microsoft.AspNetCore.Http;

public static class RequestProcess

{

    // Đọc các thông tin cơ bản của Request

    // Trả về HTML trình  bày các thông tin đó

    public static string RequestInfo(HttpRequest request)

    {

        // sử dụng đối tượng StringBuilder để xây dựng chuỗi

        var sb = new StringBuilder();

        // Lấy http scheme (http|https)

        var scheme = request.Scheme;

        sb.Append(("scheme".td() + scheme.td()).tr());

        // HOST Header

        var host = (request.Host.HasValue ? request.Host.Value : "no host");

        sb.Append(("host".td() + host.td()).tr());

        // Lấy pathbase (URL Path - cho Map)

        var pathbase = request.PathBase.ToString();

        sb.Append(("pathbase".td() + pathbase.td()).tr());

        // Lấy Path (URL Path)

        var path = request.Path.ToString();

        sb.Append(("path".td() + path.td()).tr());

        // Lấy chuỗi query của URL

        var QueryString = request.QueryString.HasValue ? request.QueryString.Value : "no query string";

        sb.Append(("QueryString".td() + QueryString.td()).tr());

        // Lấy phương thức

        var method = request.Method;

        sb.Append(("Method".td() + method.td()).tr());

        // Lấy giao thức

        var Protocol = request.Protocol;

        sb.Append(("Protocol".td() + Protocol.td()).tr());

        // Lấy ContentType

        var ContentType = request.ContentType;

        sb.Append(("ContentType".td() + ContentType.td()).tr());

        // Lấy danh sách các Header và giá trị  của nó, dùng Linq để lấy

        // Header gửi đến lưu trong thuộc tính Header  kiểu Dictionary

        var listheaderString = request.Headers.Select((header) => $"{header.Key}: {header.Value}".HtmlTag("li"));

        var headerhmtl = string.Join("", listheaderString).HtmlTag("ul"); // nối danh sách thành 1

        sb.Append(("Header".td() + headerhmtl.td()).tr());

        // Lấy danh sách các Header và giá trị  của nó, dùng Linq để lấy

        var listcokie = request.Cookies.Select((header) => $"{header.Key}: {header.Value}".HtmlTag("li"));

        var cockiesHtml = string.Join("", listcokie).HtmlTag("ul");

        sb.Append(("Cookies".td() + cockiesHtml.td()).tr());

        // Lấy tên và giá trí query

        var listquery = request.Query.Select((header) => $"{header.Key}: {header.Value}".HtmlTag("li"));

        var queryhtml = string.Join("", listquery).HtmlTag("ul");

        sb.Append(("Các Query".td() + queryhtml.td()).tr());

        //Kiểm tra thử query tên abc có không

        Microsoft.Extensions.Primitives.StringValues abc;

        bool existabc = request.Query.TryGetValue("abc", out abc);

        string queryVal = existabc ? abc.FirstOrDefault() : "không có giá trị";

        sb.Append(("abc query".td() + queryVal.ToString().td()).tr());

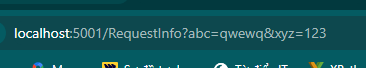
        string info = "Thông tin Request".HtmlTag("h2") + sb.ToString().HtmlTag("table", "table table-sm table-bordered");

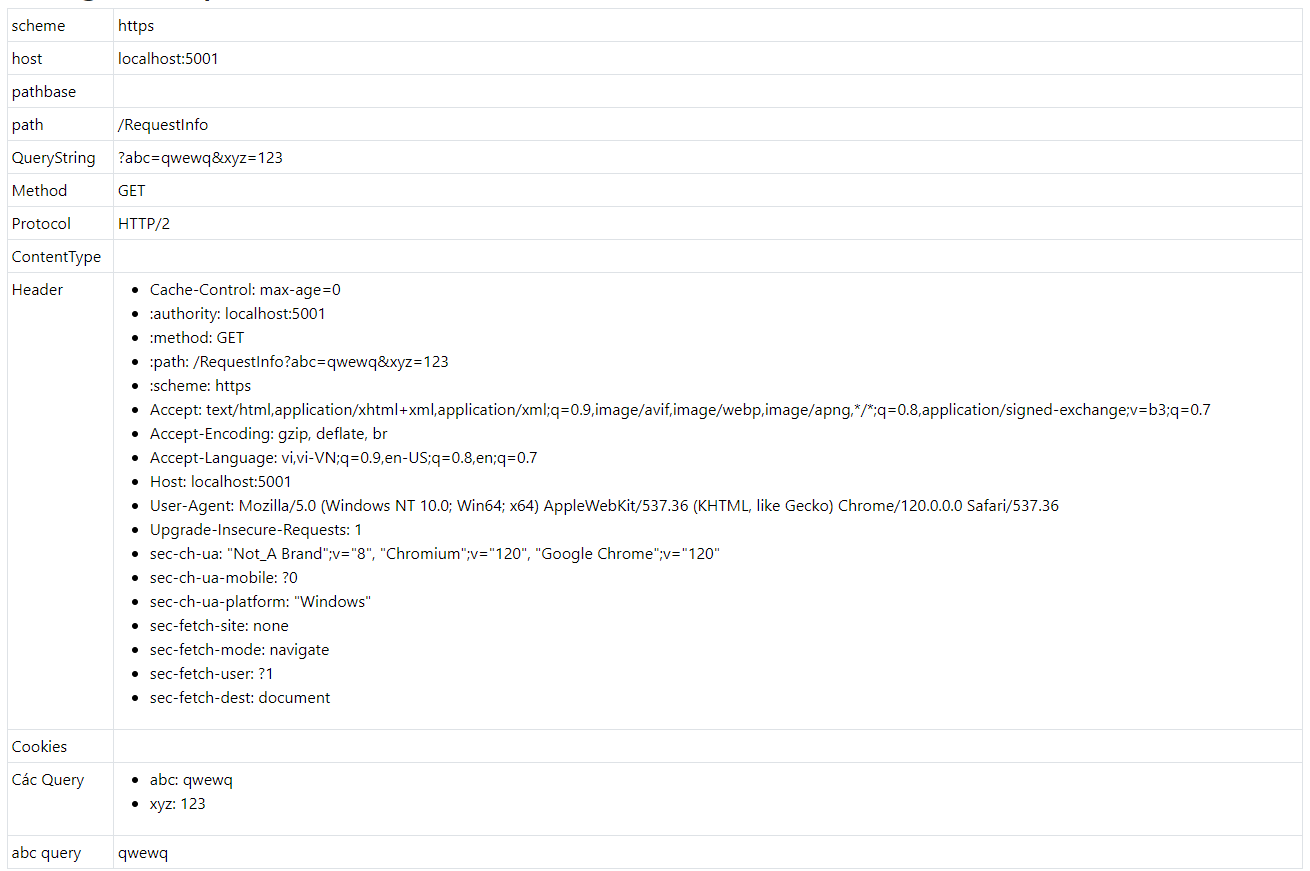
        return info;

    }

}

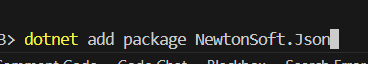
Kết quả:

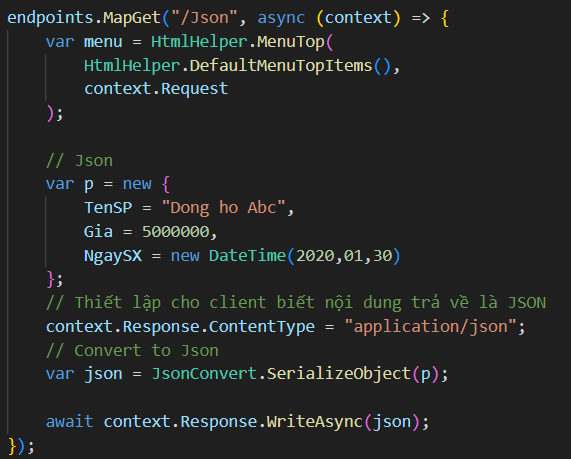




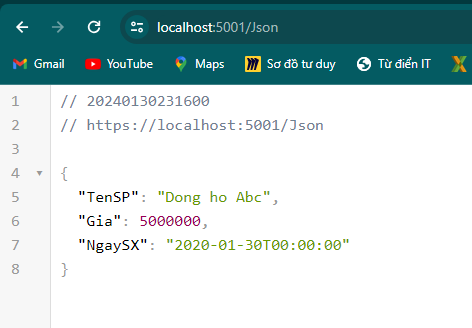
## Thiết lập Response trả về là 1 JSON

Để dự án có thể sử dụng được JSON ta cần tích hợp vào dự án thư viện



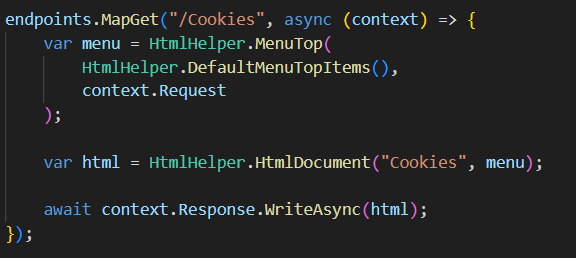


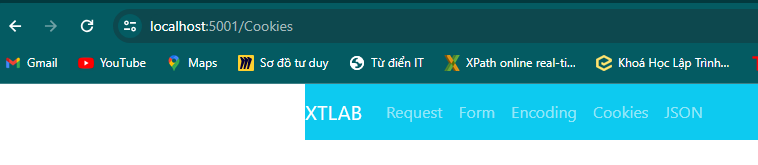
Kết quả:



## Cookies

* Cookie là một dữ liệu nhỏ do server gửi về cho client, browser
* Khi browser có được cookie nó sẽ lưu trữ lại để lần truy vấn sau nó sẽ gửi cookie này lại cho server
* Thường sử dụng cookie để lưu trữ các phiên làm việc
  + Ví dụ khi user đã đăng nhập sẽ gửi 1 cookie về server thì lần tới khi user đó truy cập thì cookie đó được gửi lên phiên làm việc đó được phục hồi và nó biết user đã đăng nhập, không cần yêu cầu đăng nhập nữa

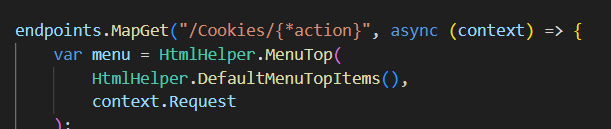




Trong endpoint này chúng ta sẽ thiết lập 2 trường hợp truy cập

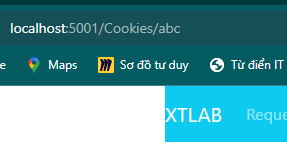
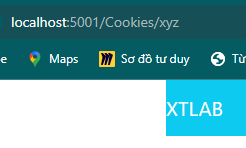
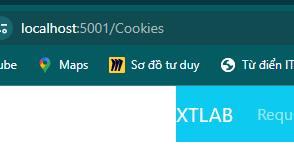
* Truy cập với địa chỉ /Cookies/write
* Truy cập với địa chỉ /Cookies/read

Chúng ta có thể thiết lập chuỗi truy vấn với phần đầu là địa chỉ cookies, phần sau là một chuỗi nào đó có thể có tên ví dụ ta để là action

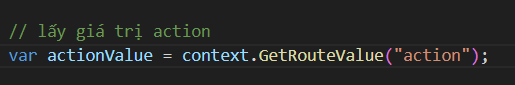


Ta để dấu \* vào trước tên để có thể truy cập àm không có chuỗi action đằng sau cũng được

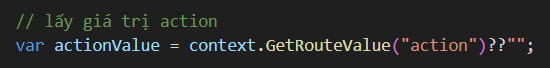
Kết quả:

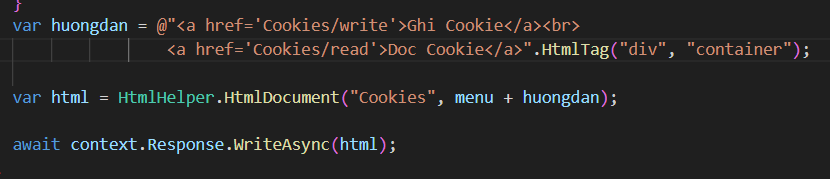
Khi ta cấu hình như vậy ta có thể đọc được giá trị action



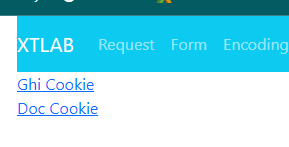
Trong trường hợp truy cập với /Cookies thì action = null nên ta có thể cho giá trị amwjc định trong trường hợp này là 1 chuỗi rỗng:



Ta thêm 2 thẻ a cho 2 trường hợp:



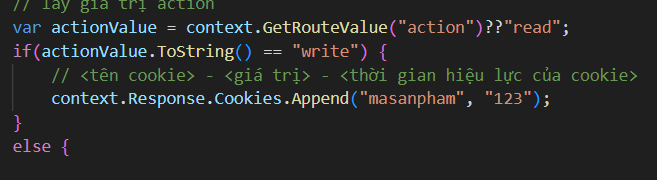
Kết quả:



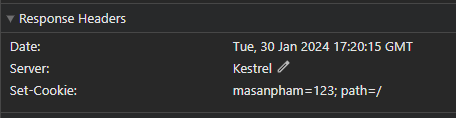
Giờ ta sẽ xử lý trường hợp ghi cookie

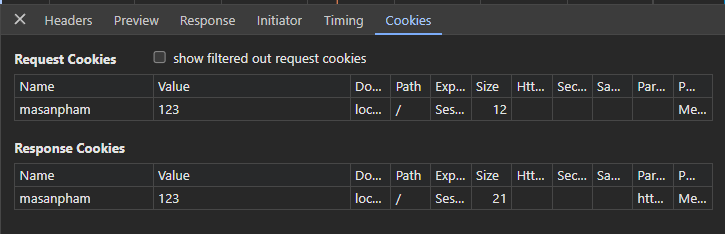
### Ghi cookies

Một cookie thường có dạng <tên cookie> - <giá trị> - <thời gian hiệu lực cảu cookie>

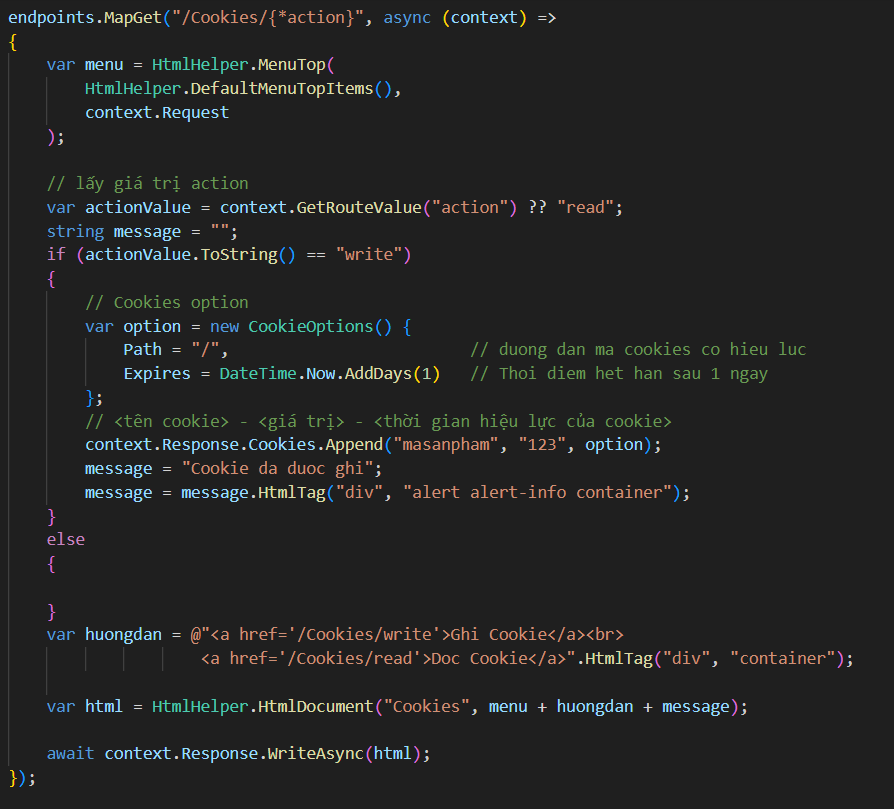


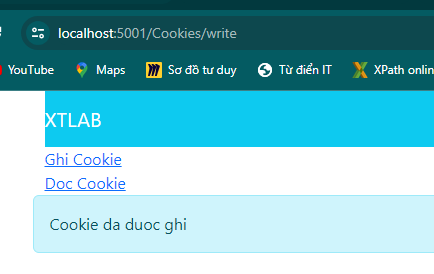
Kết quả:

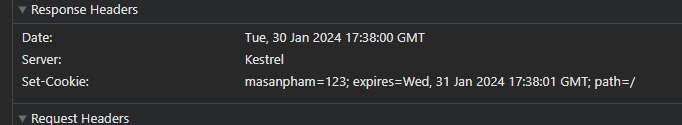
 



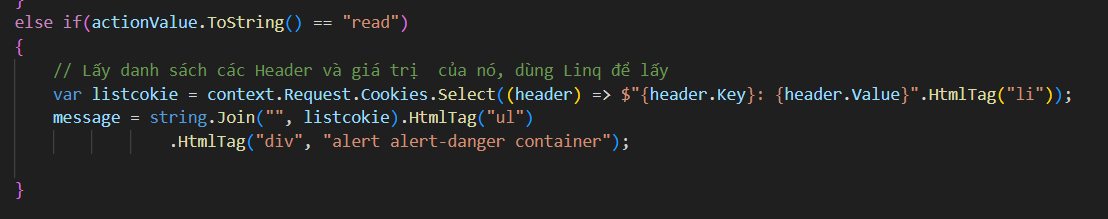
Ta có thể thêm 1 số hiển thị và option cho cookies



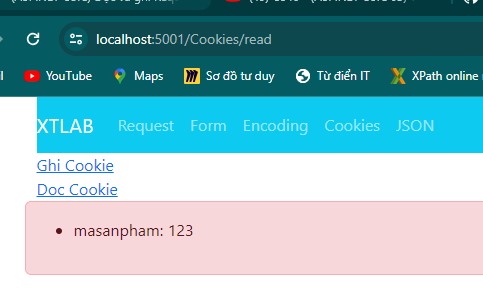




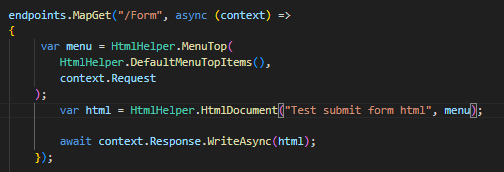
### Đọc Cookies



Kết quả:

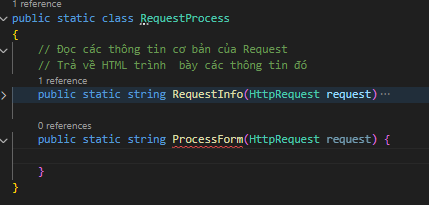


## Submit dữ liệu form



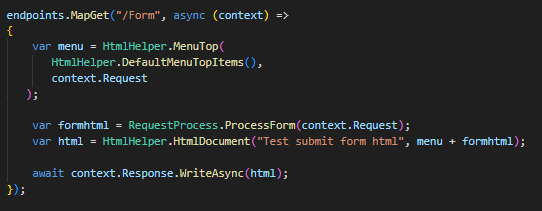
Khi form submit thì ta sẽ chuyển reqest cho một phương thức xử lý

Trong mylib/RequestProcess.cs ta tạo phương thức xử lý submit form:



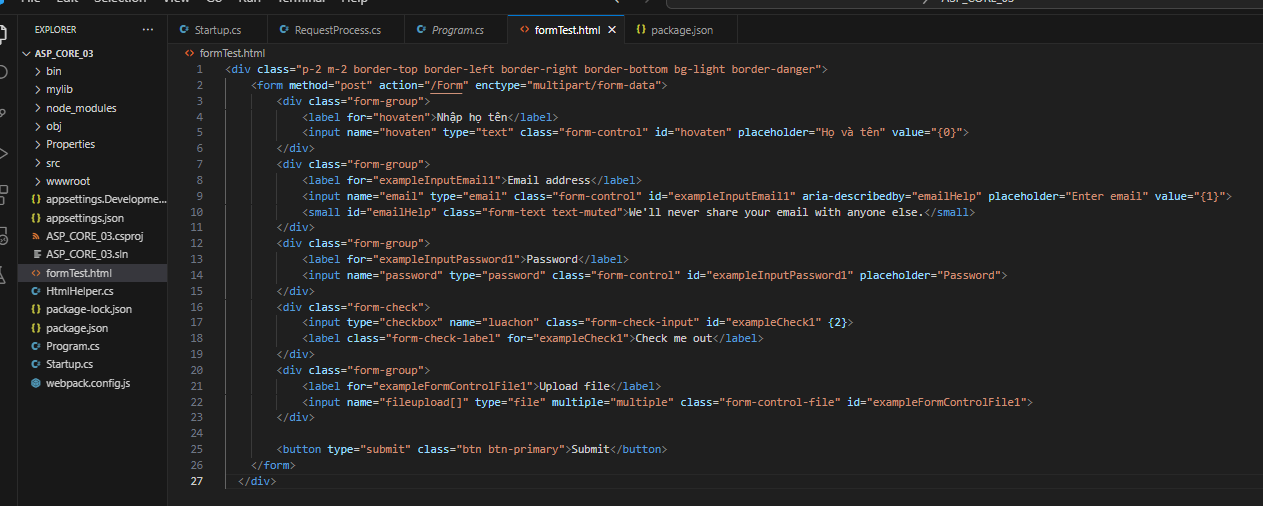
Phương thức này nhận tham số là HttpRequest và trả về chuỗi html

Tiếp đó trong Startup.cs:

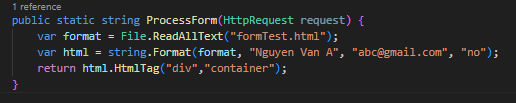


Để cho nhanh ta sẽ tạo 1 file html chứa form

Link form mẫu: <https://github.com/xuanthulabnet/learn-cs-netcore/blob/master/ASP_NET_CORE/03.RequestResponse/formtest.html>



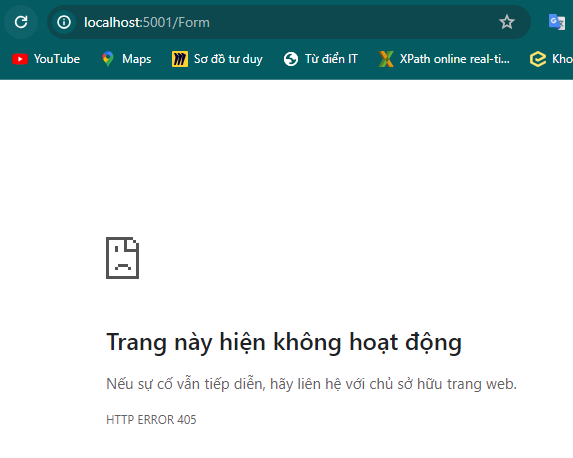
Tiếp đó trong mylib/RequestProcess.cs



Kết quả:

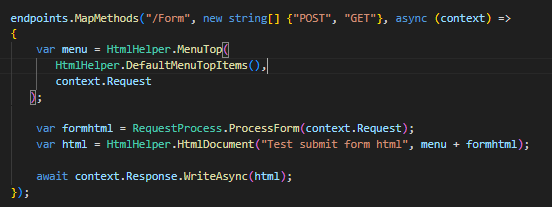


Khi ta nhấn submit thì sẽ nhận được kết quả:



Vì trang này là /Form nhưng với method là Post ta chưa cấu hình cho đường dẫn này

1 là ta có thể tạo thêm 1 phương thức endpoint là MapPost tuy nhiên ta có thể sử dụng 1 phương thức mà không phải tạo ra 2 endpoint như sau:

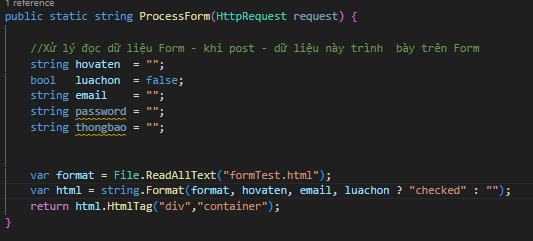


Khi đó ta submit thì sẽ đều chạy vào endpoint này

Khi ta submit dữ liệu thì ứng dụng sẽ nhận được và nó sẽ xử lý trong ProcessForm

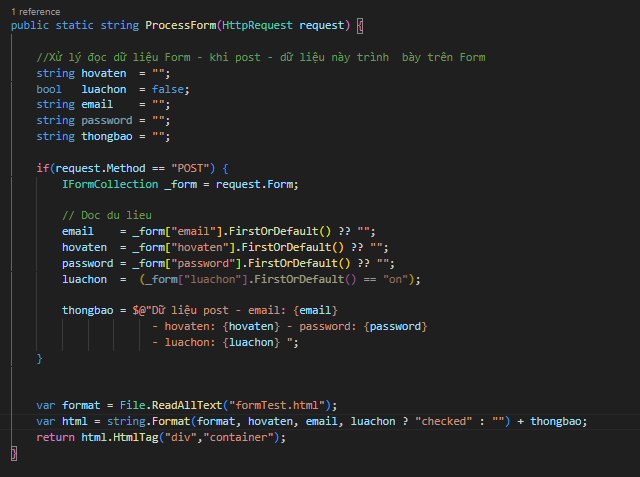
Để đọc được những dữ liệu gửi đến ta làm như sau:

Đầu tiên khai báo các trường dữ liệu tưng ứng với form

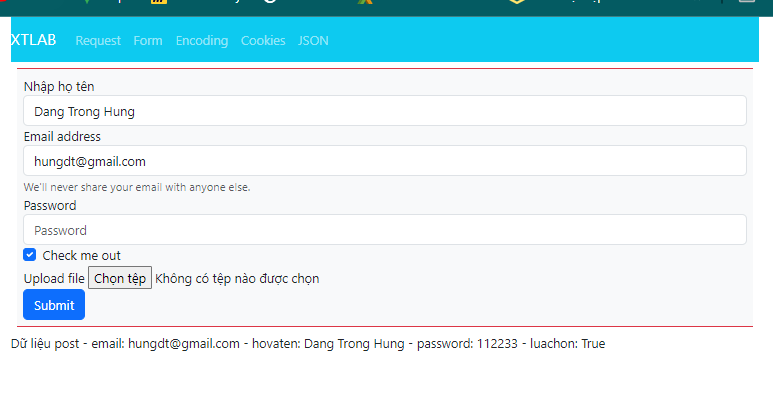


Những dữ liệu Html Form khi submit lên thì nó nằm trong thuộc tính của request là **request.Form**

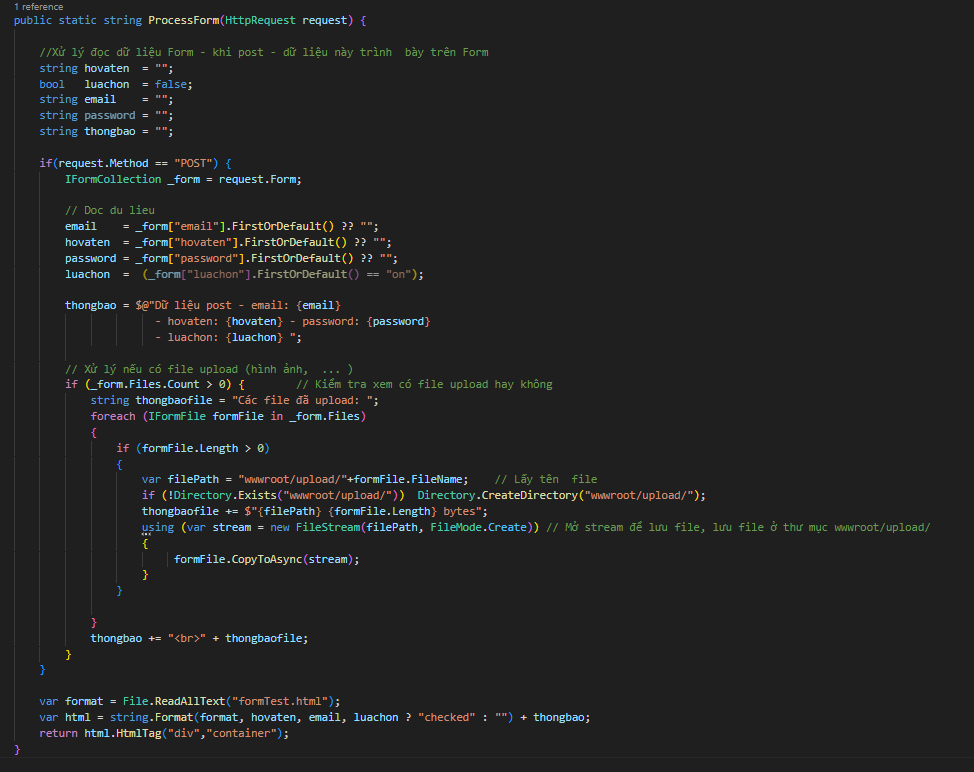
Lưu ý 1 điều chúng ta chỉ đọc nội dung form khi phương thức gửi đến là POST vì vậy ta cần kiểm tra trước khi sử dụng



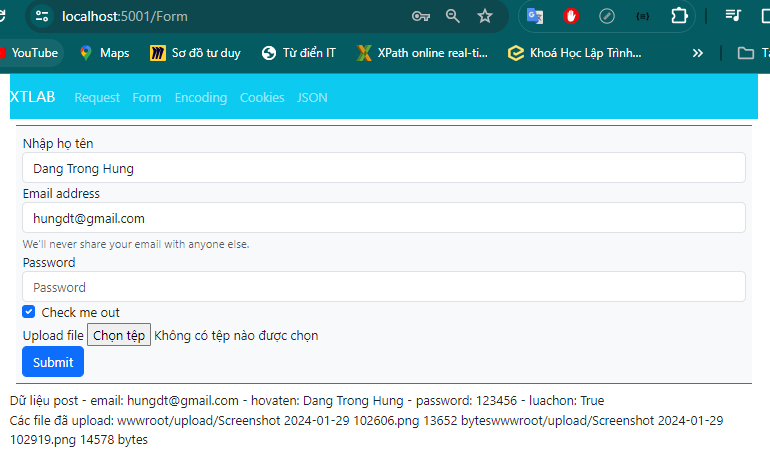
Kết quả:

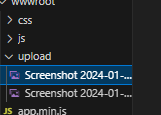


## Upload File



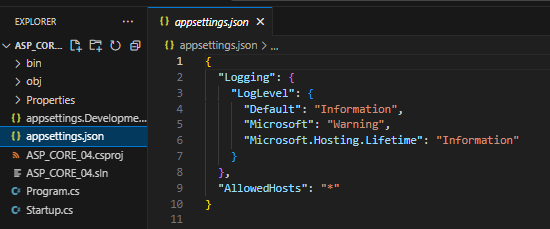
Kết quả:





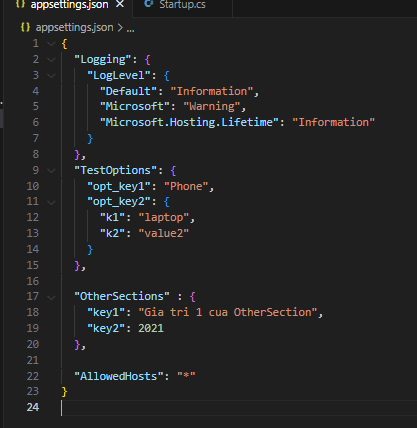
# Cấu hình ứng dụng và kỹ thuật inject các serveice

Trong ứng dụng ASP.NET các cấu hình của nó được lưu trữ trong file appsetting.json



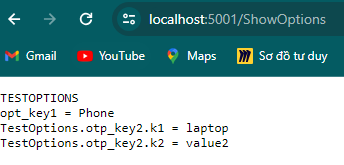
Ta thấy có 1 Section cấu hình Logging

Các cấu hình khác ta cũng tạ ra các Section tương tự

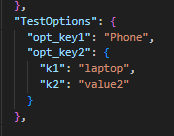


## Sử dụng Iconfiguration để lấy ra các sections

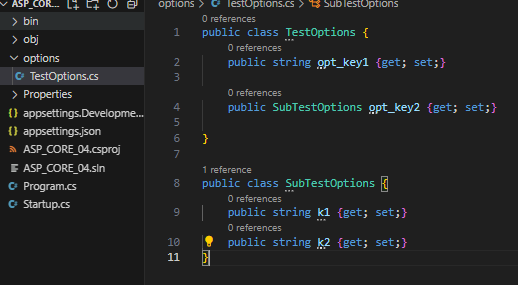




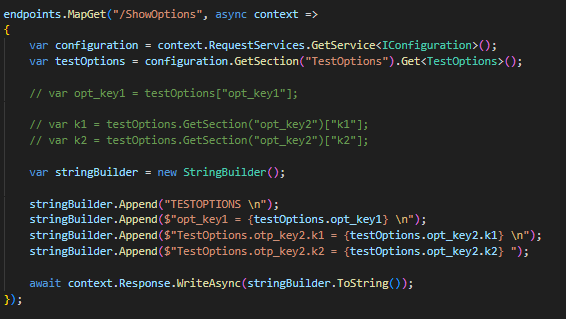
## Chuyển đổi các Section lấy ra thành object

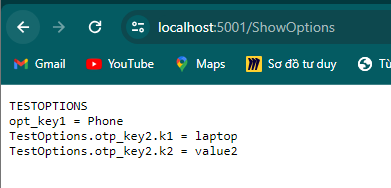


Trước khi chuyển đổi ta cần có 1 class tương tự với section về các trường



Lúc này ta có thể tạo object từ secsion thông qua class vừa tạo



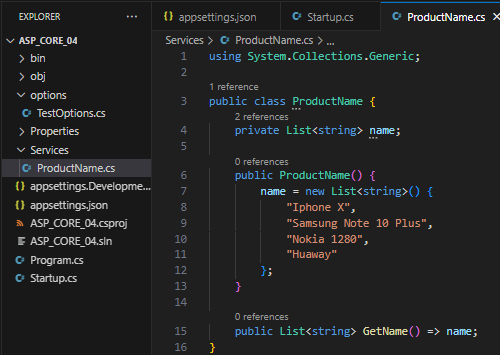


## Dịch vụ và cách đăng ký dịch vụ trong .NET

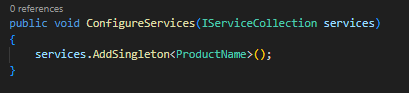
Trong ngữ cảnh của ASP.NET Core hoặc nhiều framework và thư viện khác, "service" thường là một thành phần cung cấp các chức năng hoặc dịch vụ cụ thể. Một service có thể thực hiện một loạt các công việc, chẳng hạn như xử lý logic kinh doanh, quản lý trạng thái ứng dụng, truy cập cơ sở dữ liệu, ghi log, và nhiều chức năng khác.

Trong ngữ cảnh của ASP.NET Core và Dependency Injection, một service thường được đăng ký với Dependency Injection container để có thể được chia sẻ và sử dụng trong ứng dụng. Dependency Injection giúp giảm sự phụ thuộc giữa các thành phần của ứng dụng và làm cho mã nguồn trở nên dễ bảo trì hơn.

Giả sử ta có 1 service sau:

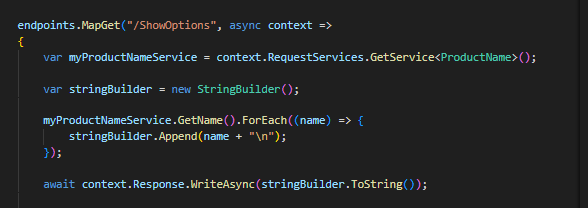


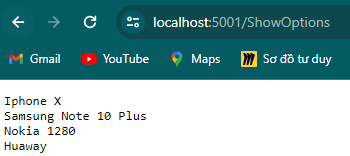
Khi ta có service ta có thể đăng ký vào DI của hệ thống tại phương thức ConfigureService



Ở đây ta tạo ra kiểu Singleton tức là chỉ tạo 1 ProductName cho tất cả các truy vấn

Khi đăng ký xong service ta có thể lấy ra bằng ServiceProvider hoặc chúng ta có thể inject và các service khác





# Sesstion

Sesstion là dữ liệu của phiên làm việc

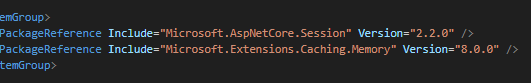
Khi chúng ta sử dụng section thì dữ liệu của phiên làm việc được lưu trữ lại và chúng chia sẻ giữa các lần truy vấn

VD: Sesstion lưu trữ thông tin người dùng đăng nhập, Sesstion trong Giỏ hàng

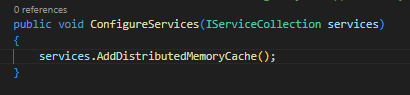
## Kích hoạt section

dotnet **add** package Microsoft.AspNetCore.Session

dotnet **add** package Microsoft.Extensions.Caching.Memory



Để kích hoạt section ta phải thêm vào service section trong ConfigureService



Sau đó trong pipeline ta cần thêm vào 1 middleware

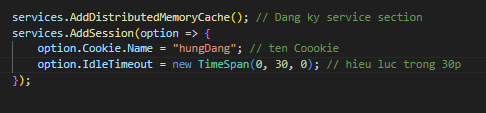


Khi httpCotext đi qua Sesstion Middleware thì dữ liệu của phiên làm việc sẽ được phục hồi vào 1 thuộc tính của HttpContext. Thuộc tính này có thể truy cập qua HttpContext thông qua thuộc tính Sesstion

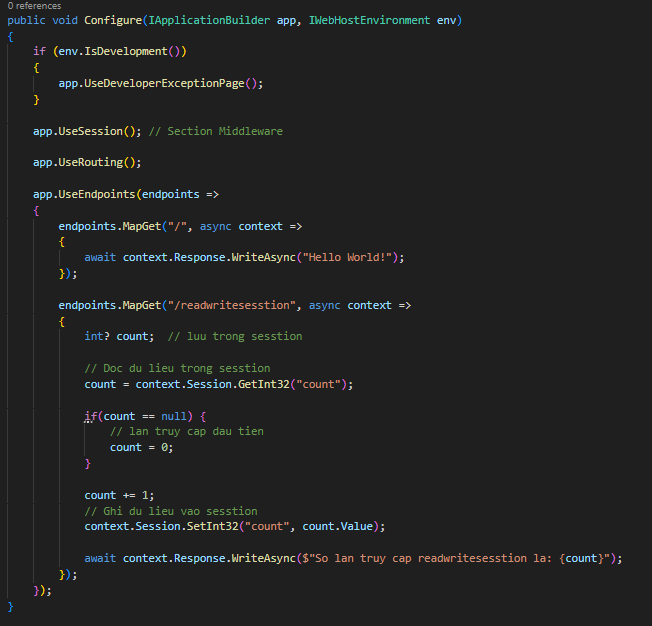
Một số thuộc tính của Section

|  |  |
| --- | --- |
| **Member** | **Ý nghĩa** |
| ID | Thuộc tính lấy ID của Session, ID này có gửi về lưu ở Cookie |
| Clear() | Xóa bỏ các giá trị lưu trong Session |
| Set(String key, Byte[]) | Lưu mảng byte vào Session với key chỉ ra |
| TryGetValue(String key, Byte[]) | Lấy dữ liệu lưu trong key (trả về false là có thành công) |
| SetString(String key, String s) | Lưu chuỗi vào key |
| GetString(String key) | Lấy chuỗi lưu trong key |
| SetInt32(String key, Int32 val) | Lưu số kiểu int32 vào key |
| GetInt32(String key) | Lấy số int32 trong key |

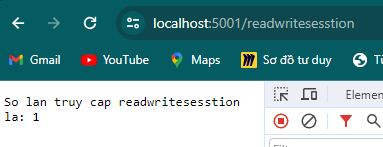
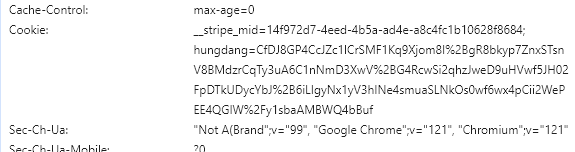
Thông thường Sesstion Middleware sẽ gửi về cho Client 1 cookies chứa id của Sesstion. Trình duyệt lưu trữ lại cookie này thì lần truy vấn tiếp theo nó sẽ gửi cookie và căn cứ vào cookie thì section sẽ phục hồi lại dữ liệu mà nó đang lưu trữ trong bộ nhớ. Dữ liệu đươc phục hồi nằm trong thuộc tính Section của context

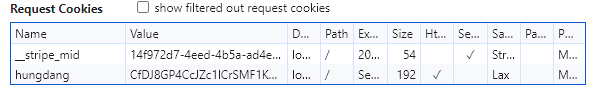


Nếu quá thời gian hiệu lực thì nếu cookie gửi lên thì dữ liệu của sesstion vẫn không được phục hồi và bị xóa đi và khởi tạo một sesstion mới

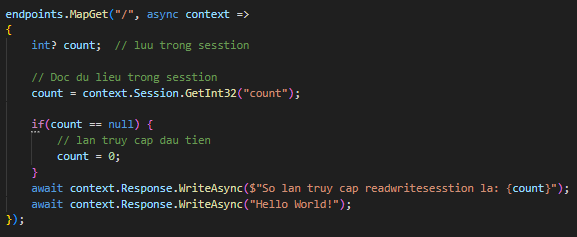


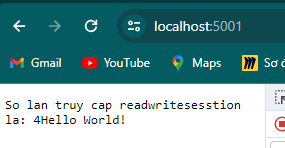
Kết quả:



Dữ liệu trong sesstion không chỉ chia sẻ cho 1 url ở những lần truy vấn khác nhau mà nó còn chia sẻ cho những url khác nhau, ví dụ như ở địa chỉ / ta cũng có thể phục hồi sesstion mà được thiết lập trong / readwritesesstion





## Sử dụng SQL Server lưu Sesstion

# Gửi Mail trong .NET Core