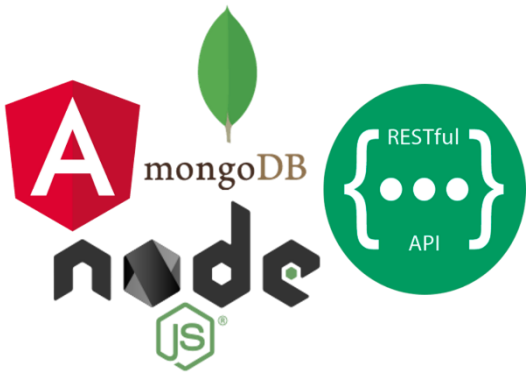
PRACTICE MATERIALS

Business Web Development



Instructions:

- Practical exercises are divided into many Modules
- Each Module is designed for a duration of 3 or 5 practice hours in class with the guidance of the Supervisor.
- Depending on the number of lessons allotted, each week can do many Modules.
- Students must do all the exercises in the Modules in the respective week. Students who have not completed their work assignments are responsible for continuing to do them at home.
- Exercises marked with an asterisk (*) are advanced exercises for good students.

Supervisor

Tran Duy Thanh, PhD

INDEX

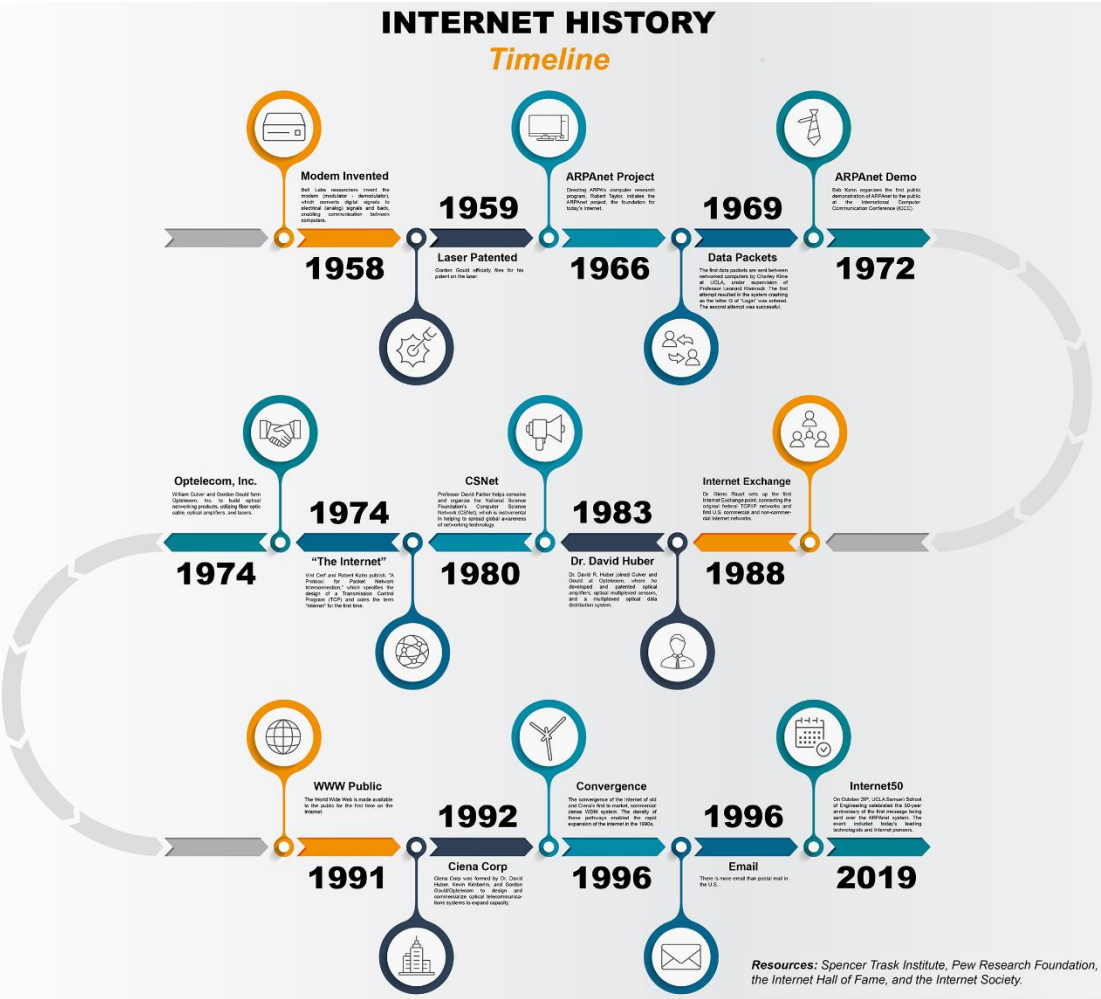
Module 1: Business Web Development Overview

Practical knowledge content:

+ Review basic concepts of Network and Internet
+ Exploiting Internet resources
+ The process of steps to create a Website
+ Distinguish Website, Webpage, Server, WebServer
+ Distinguish Hosting-Domain, how to register
+ Understand the process of deploying and putting the Website into operation

Exercise 1: History of the Internet

Let's present the development history of the Internet



Exercise 2: Server vs Client

Distinguish between Server and Client. Are desktops and laptops used every day called Servers or not? Why?

Exercise 3: HTTP vs HTTPS

Present and compare HTTP and HTTPS protocols.

Exercise 4: Static Web and Dynamic Web

Distinguish between Static Web and Dynamic Web.

Exercise 5: Hosting vs Domain

Let's distinguish Hosting and Domain. What types of domains are there? Based on where to choose to buy the optimal Hosting for the Website system?. If the Website specializes in e-commerce, what popular types should you buy: .org or .edu or .com or .net or .info or .com.vn or .vn ... (or put more self-suggestion)

Exercise 6: Server- Webserver-website

Distinguish Server, WebServer, Website, Webpage, Web Browser. The People said that a Server should install many WebServers to save the system right or wrong? Why?

Exercise 7: Structure of URL

Let's show how to find the IP address of a domain.

Given URL with the following structure:

https://tranduythanh.com/category/angularjs/?id=1

Please indicate the components: Protocol, domain, parameter in the above link

Exercise 8: The process of creating a Website

Search information from the Internet, then present at least two different ideas about: The step-by-step process for creating a Website.

Module 2: Hypertext Markup Language (HTML)

Practical knowledge content:

+ Use Visual Studio tools to create files, Web projects
+ Configure IIS Webserver, configure Hosting + Domain Free
+ Understand the structure of an HTML document
+ Practice Basic tags in HTML
+ Practice HTML List Tags
+ Practice Page Link Cards
+ How to create a table in HTML
+ How to use Frame, Iframe, Frameset
+ How to use Marquee
+ How to use Multimedia

Exercise 9: Using Visual Studio to create a Website

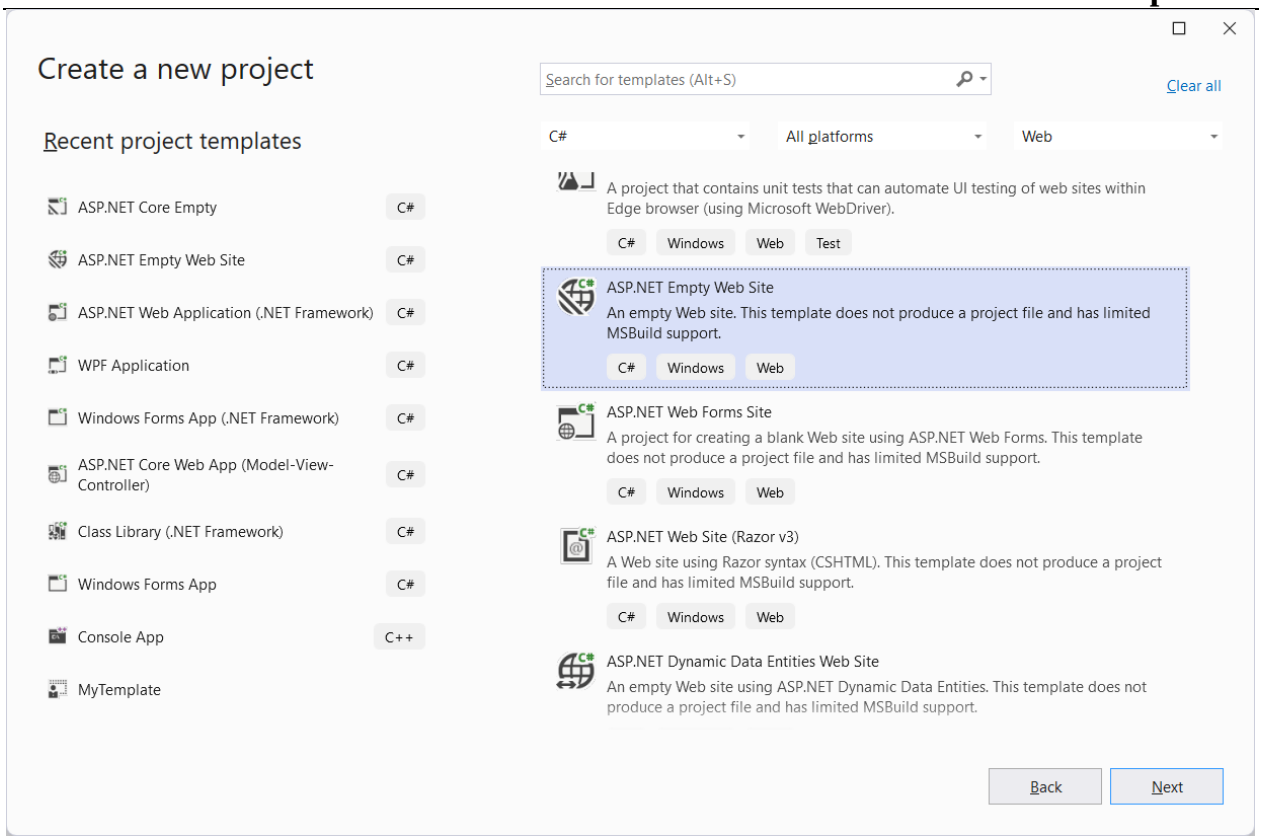
Requirement:

Using Visual Studio Tools to create a Web Project, how to create and run an HTML File in Visual Studio.

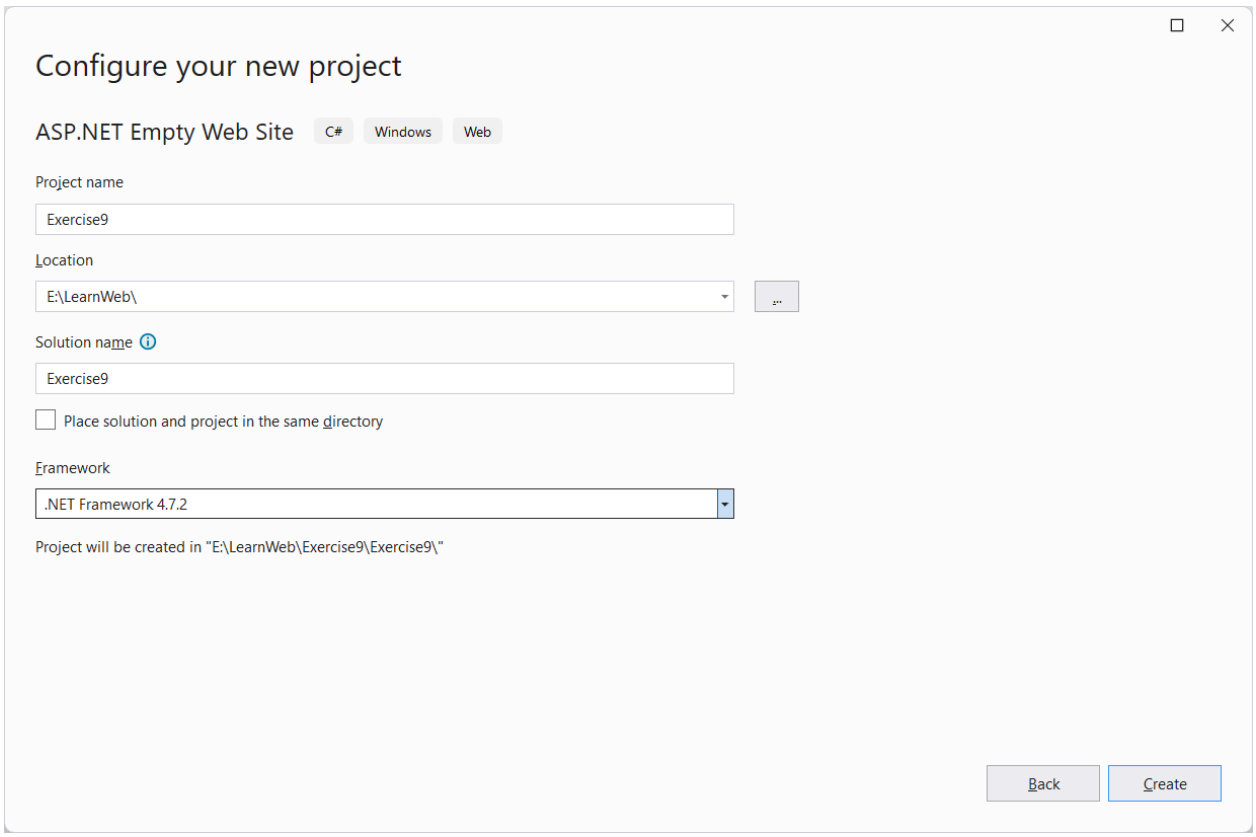
Instructions:

+ **Step 1:** Start Microsoft Visual Studio
+ **Step 2:** Select ASP.NET Empty Website template

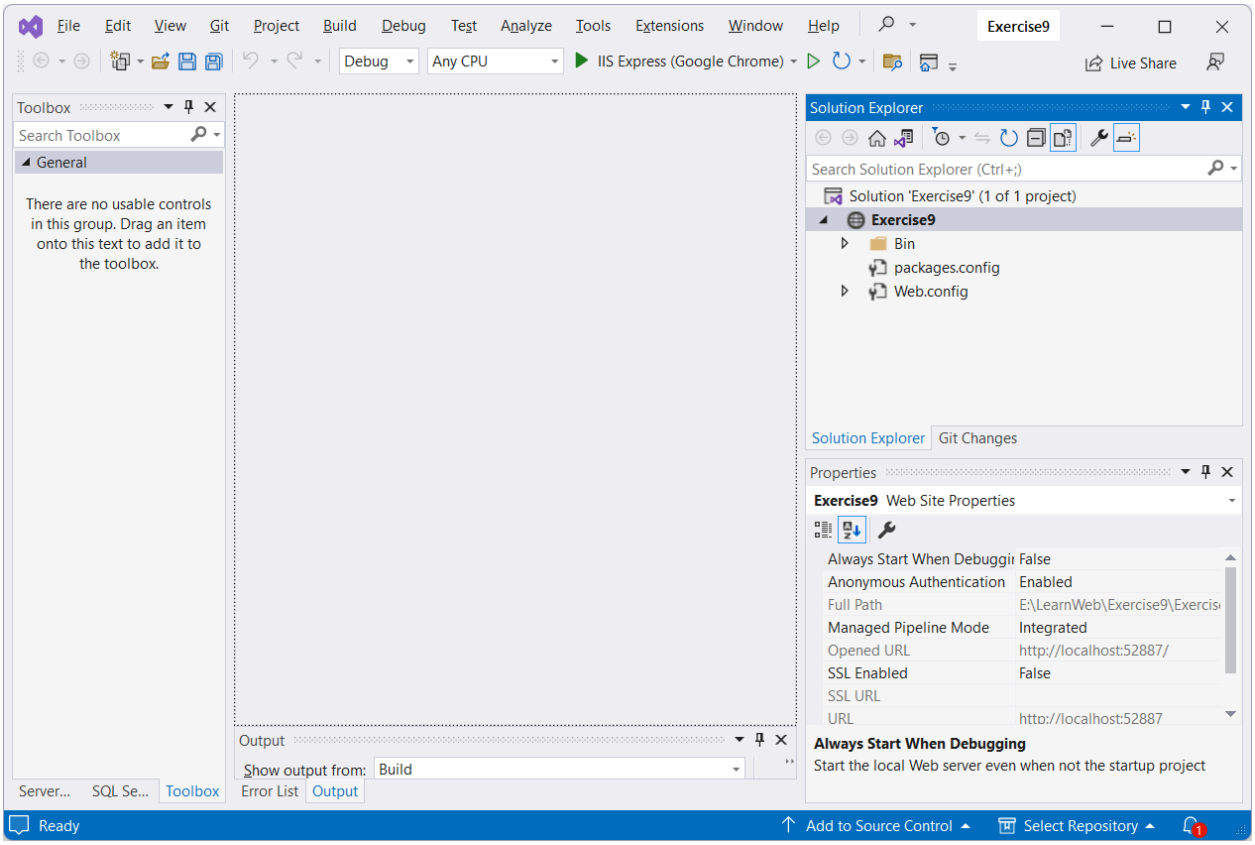
Then click Next button.



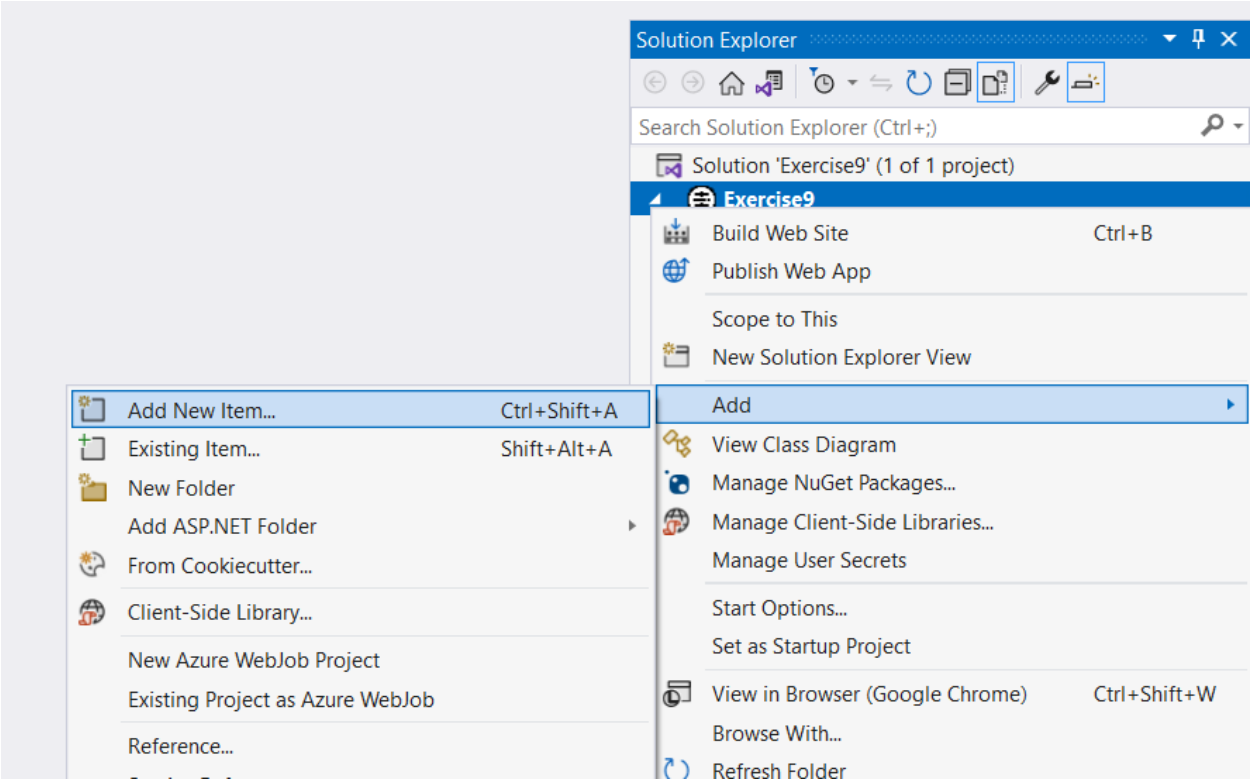
Step 3: Project name, enter “Exercise9”
Location: place to save the project
Framework: select lasted framework
Then click Create



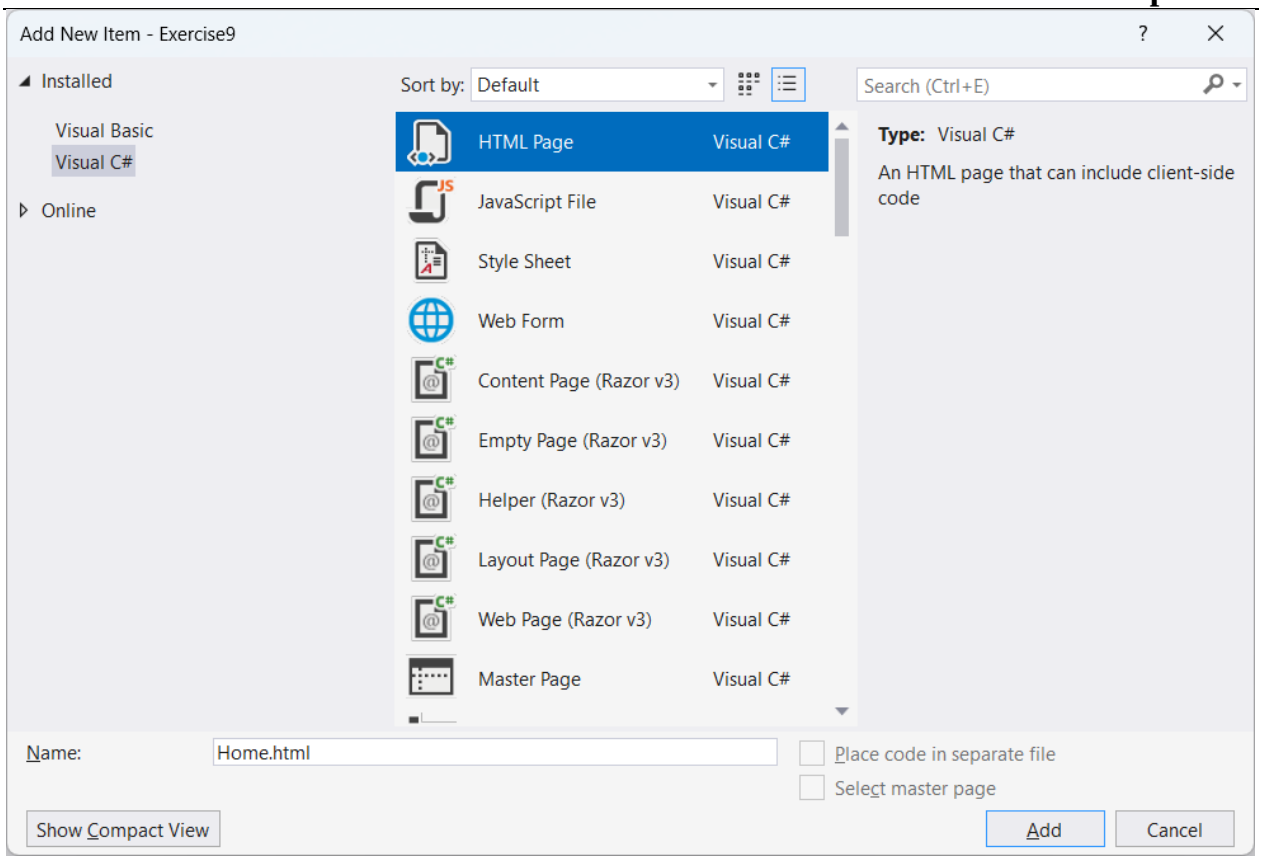
After click create button, the Project Exercise9 is created like figure below:



Step 4: Right click on the Exercise9, then choose Add/ choose Add new Item

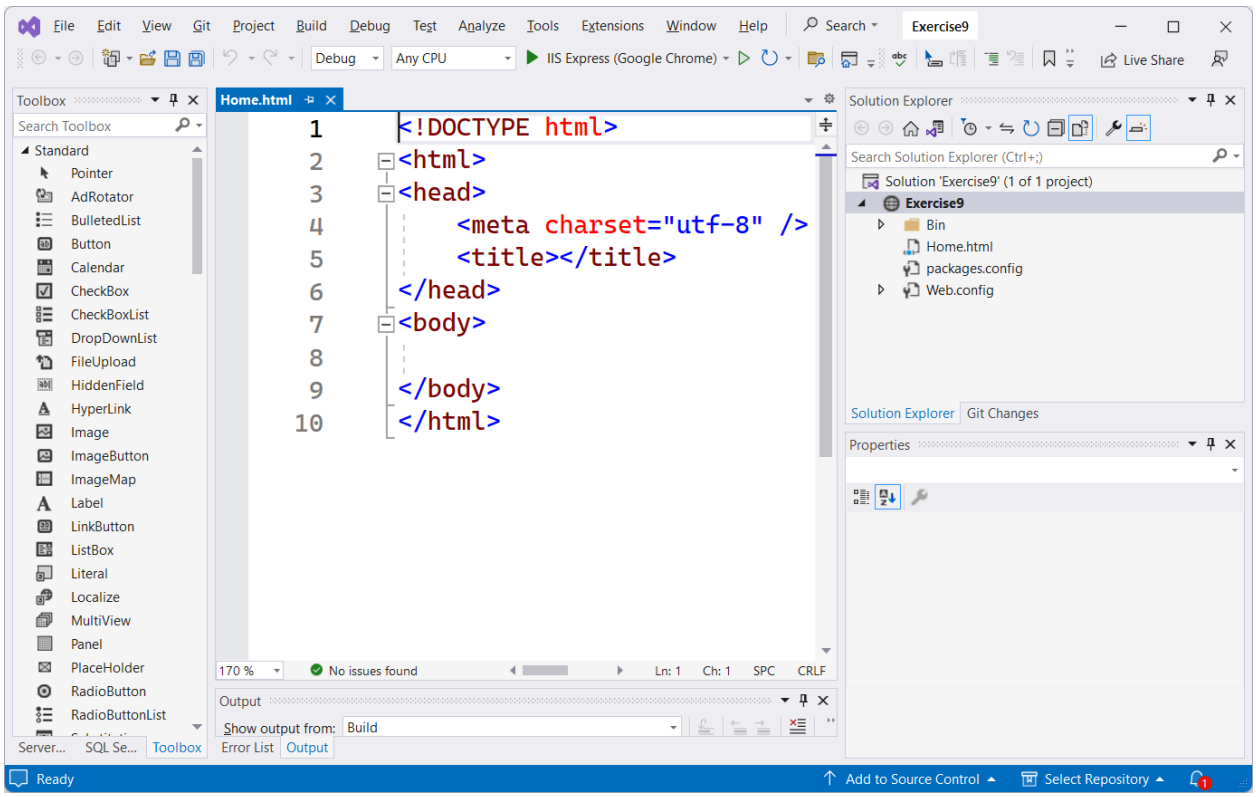


After that the screen add new item should be displayed:



Name: Change to “Home.html” then click Add button

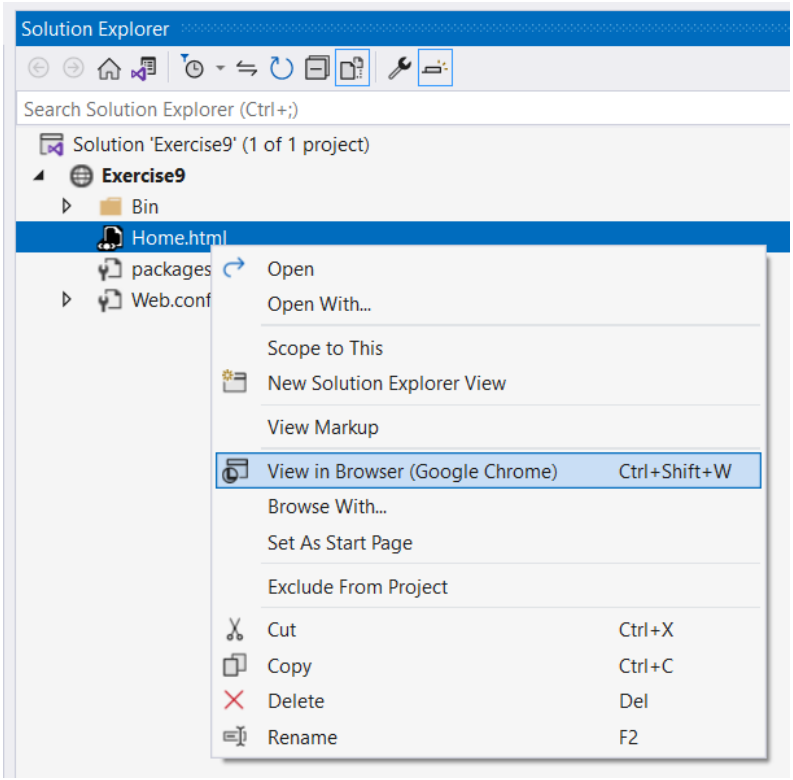
Then, the Screen of Home.html will be displayed as below:



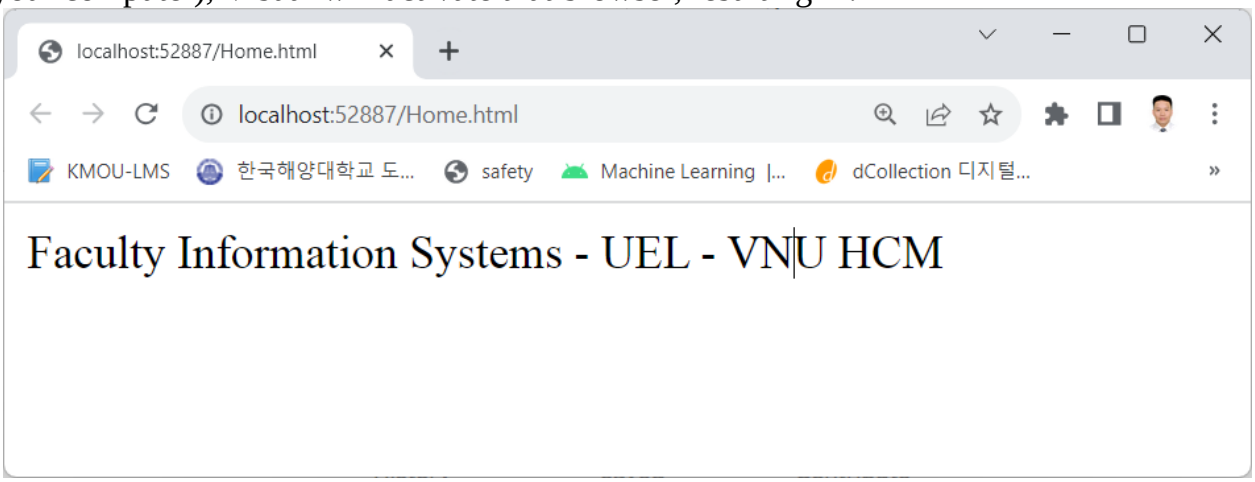
In the Body tag, we write more content:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    Faculty Information Systems - UEL - VNU HCM
</body>
</html>
```

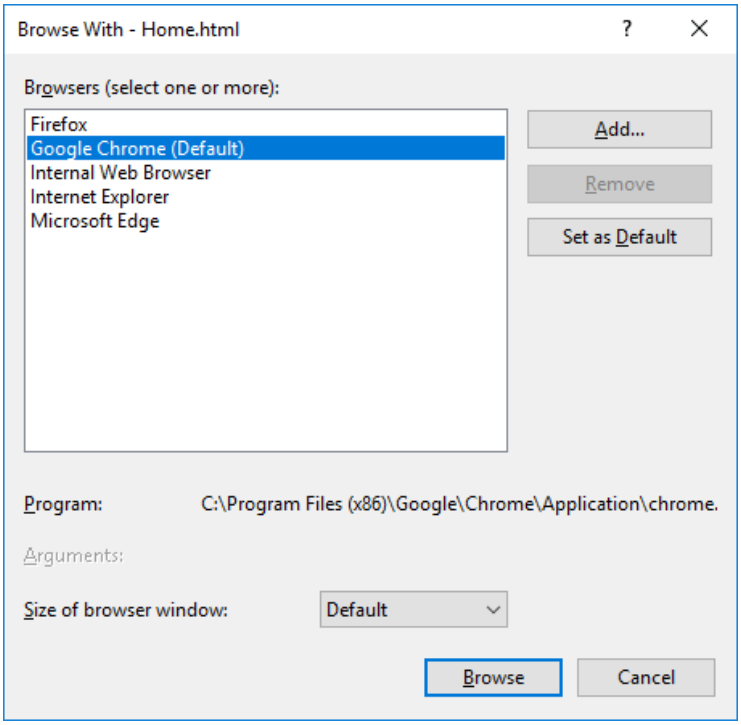
Step 5: To run the Website in Visual studio we have many ways, the simplest way is to right-click on the HTML file you want to run View In Browser or Browser With:



If you choose View In Browser (depending on the browser you are running by default in your computer), Visual will activate that browser, resulting in:



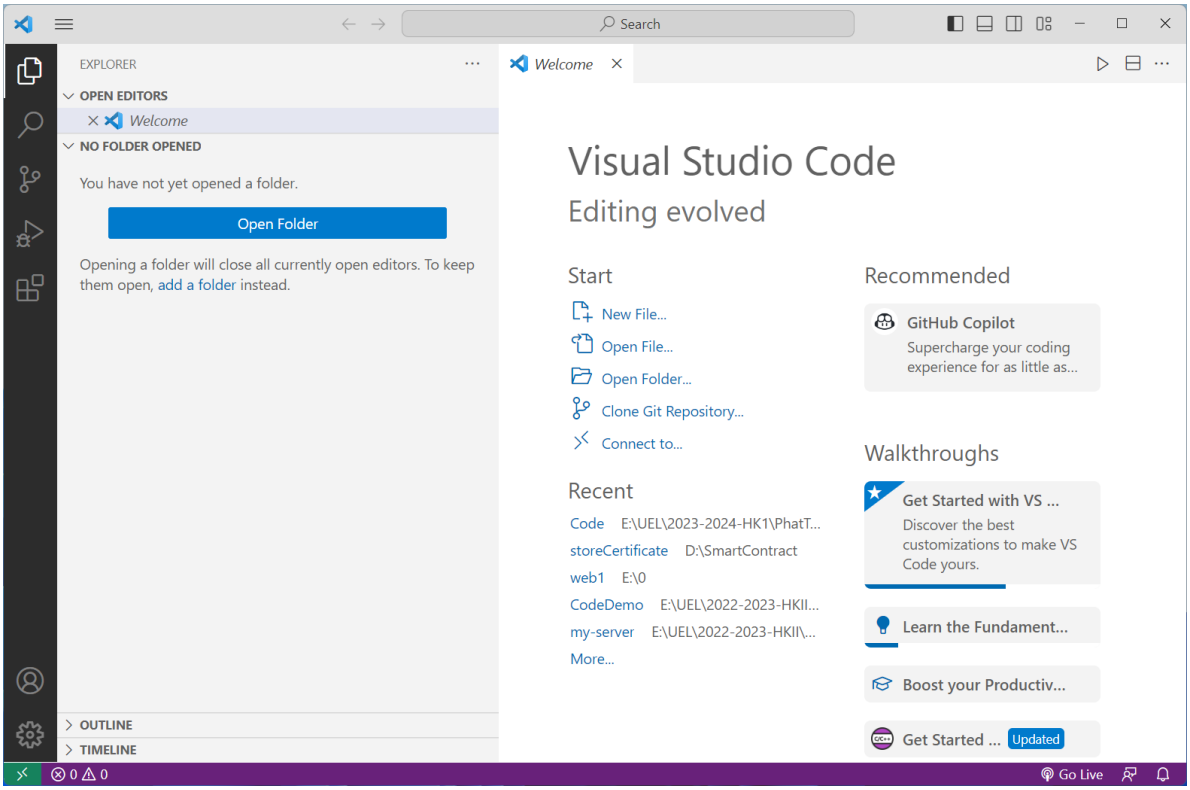
If you choose Browse With: Visual Studio will display a window asking you to choose another browser:



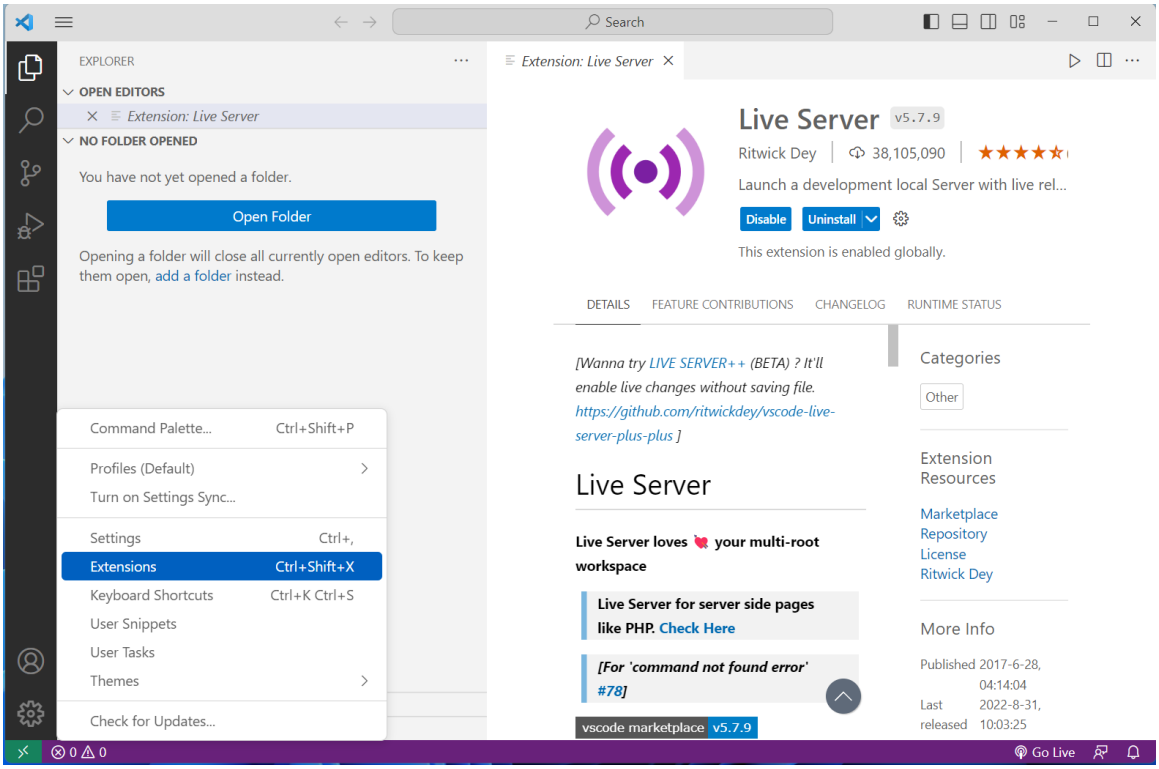
For example, the screen above, the computer has 5 browsers, we want the browser to run by default, select it and click "Set as Default", if you want to run this Website, click the Browse button at the bottom of the window.

Now, I would like to present writing HTML project on Visual Studio Code.

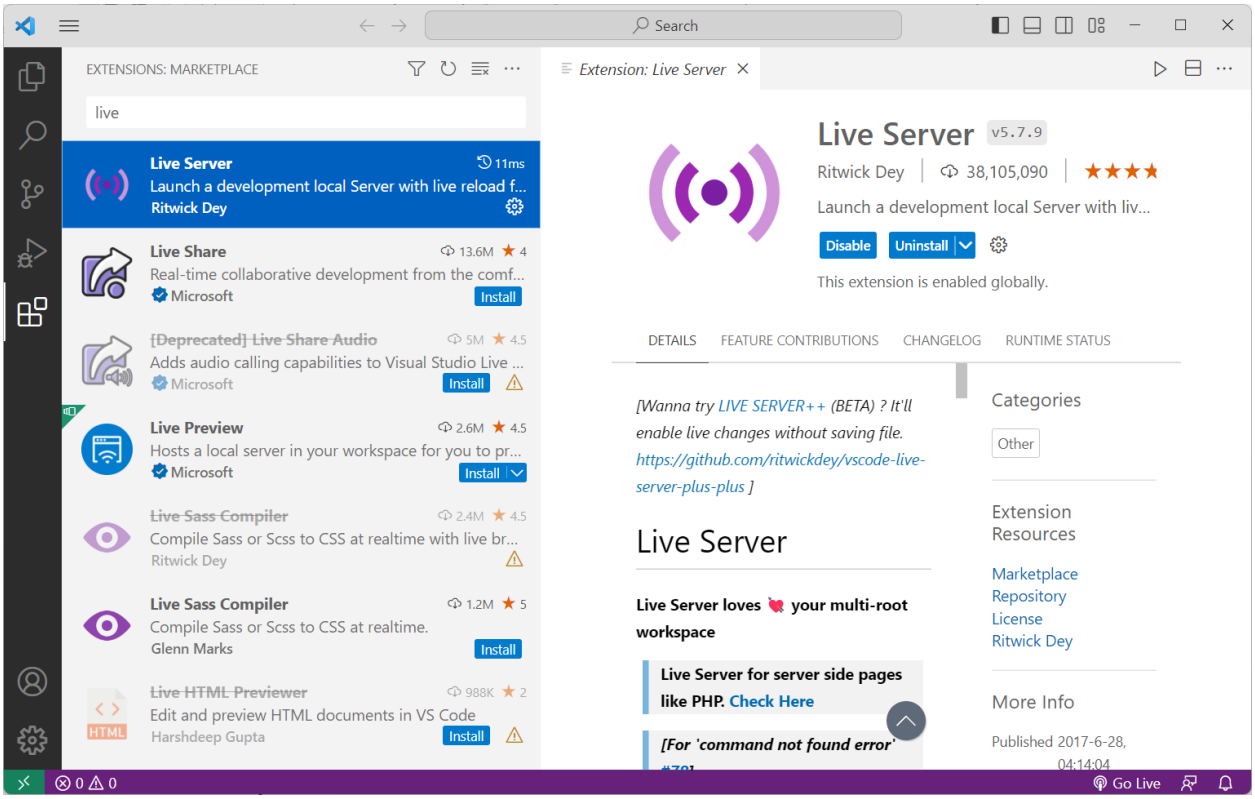
Step 1: Visit the link to download and setup this software:
https://code.visualstudio.com/



Step 2: Install Live Server extension

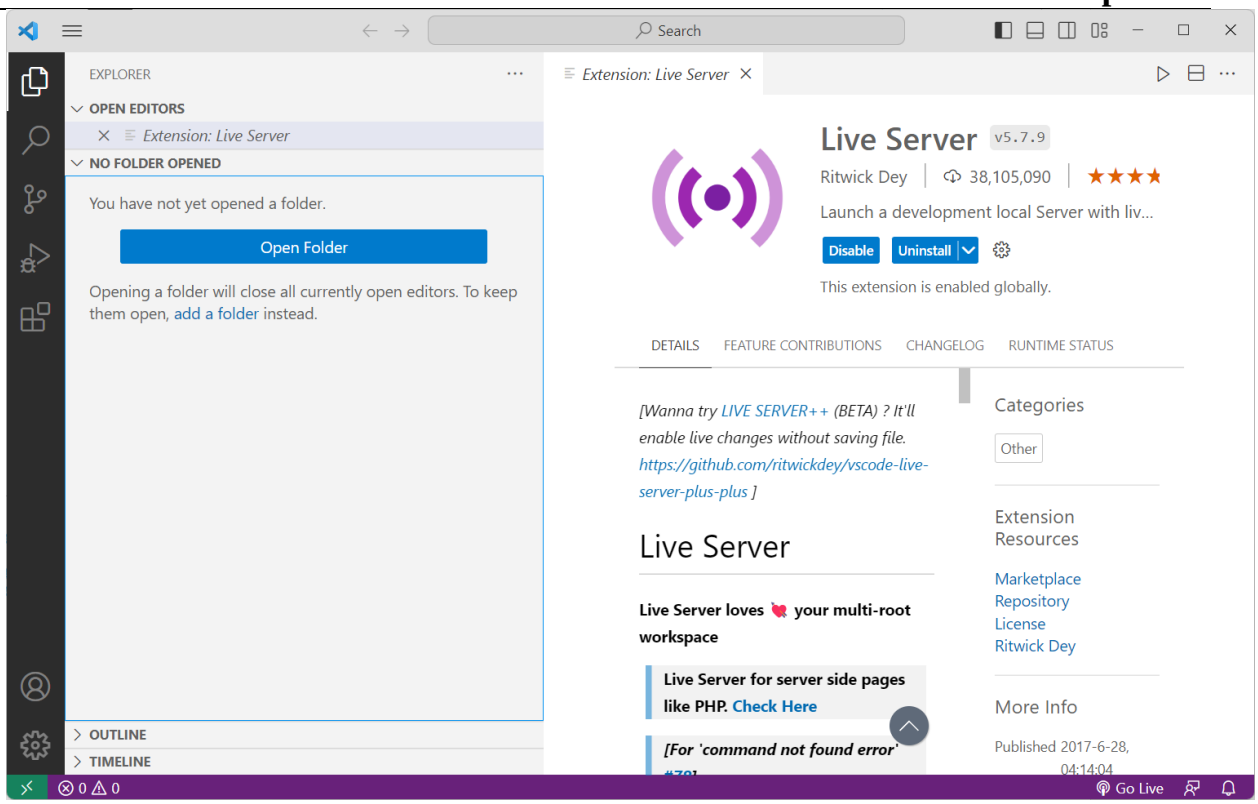


Click on the "Manage" button in the left-bottom screen, then choose "Extensions"
Then search the "**Live Server**" and click install:

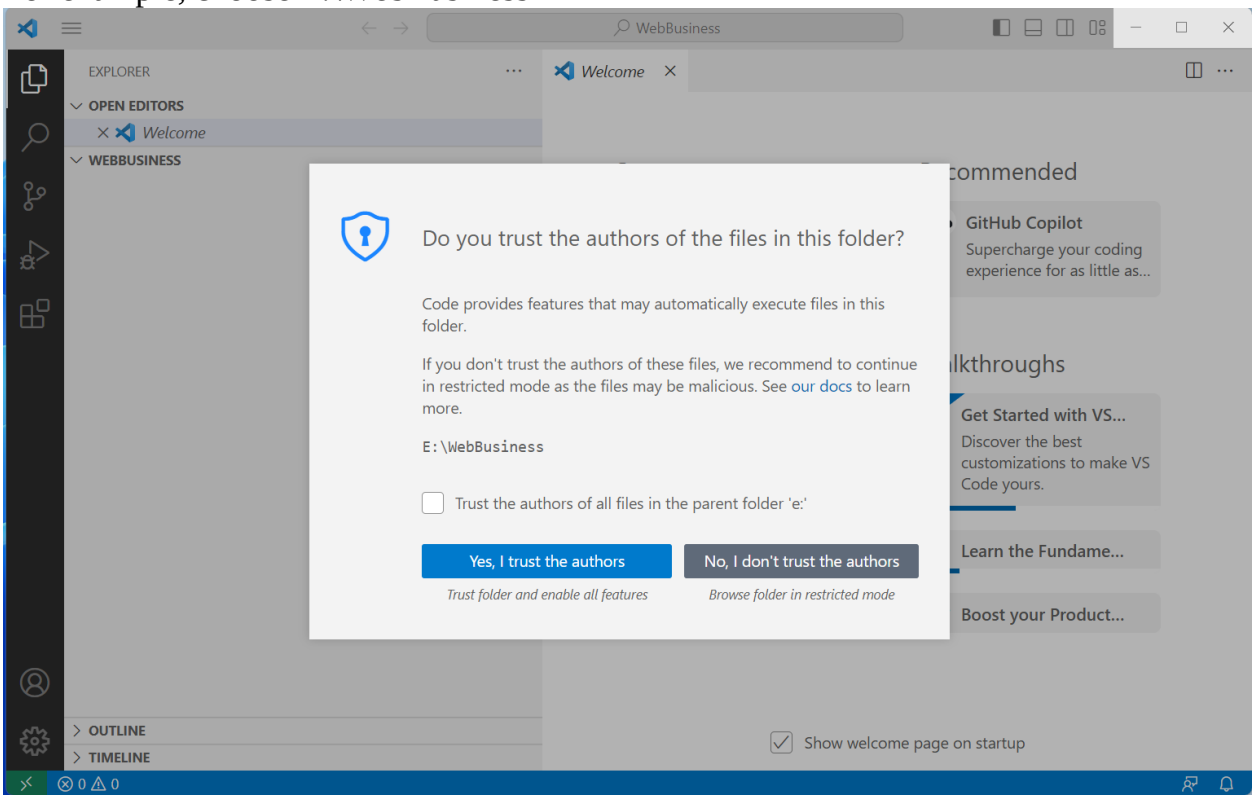


Step 3: Create a Folder/Project for HTML

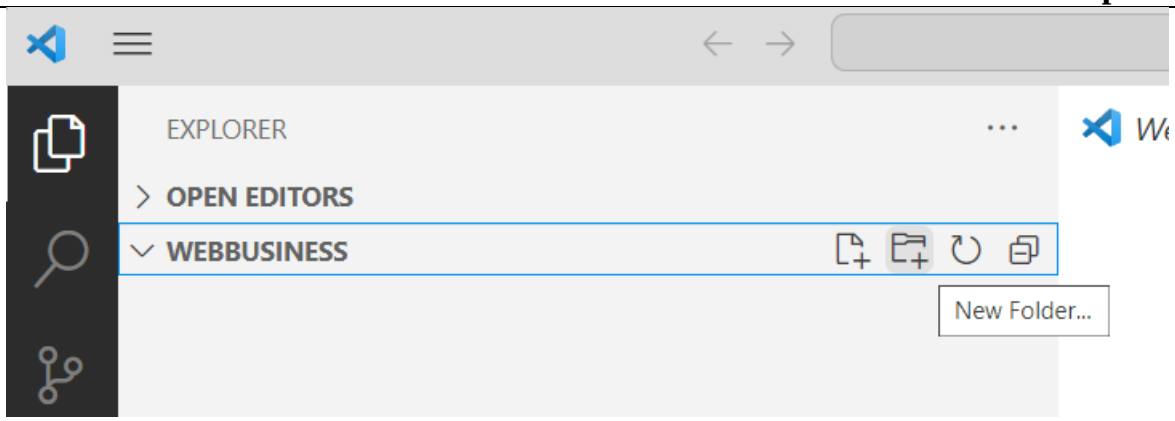
Come back to the Explorer screen:



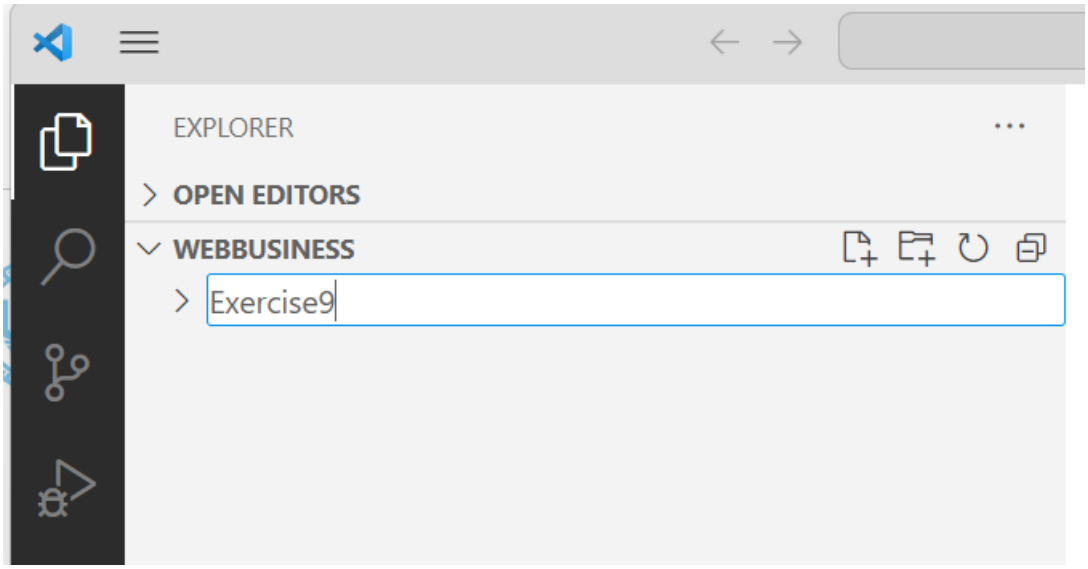
Click “Open Folder” button to set the folder for editors.
For example, choose E:\WebBusiness



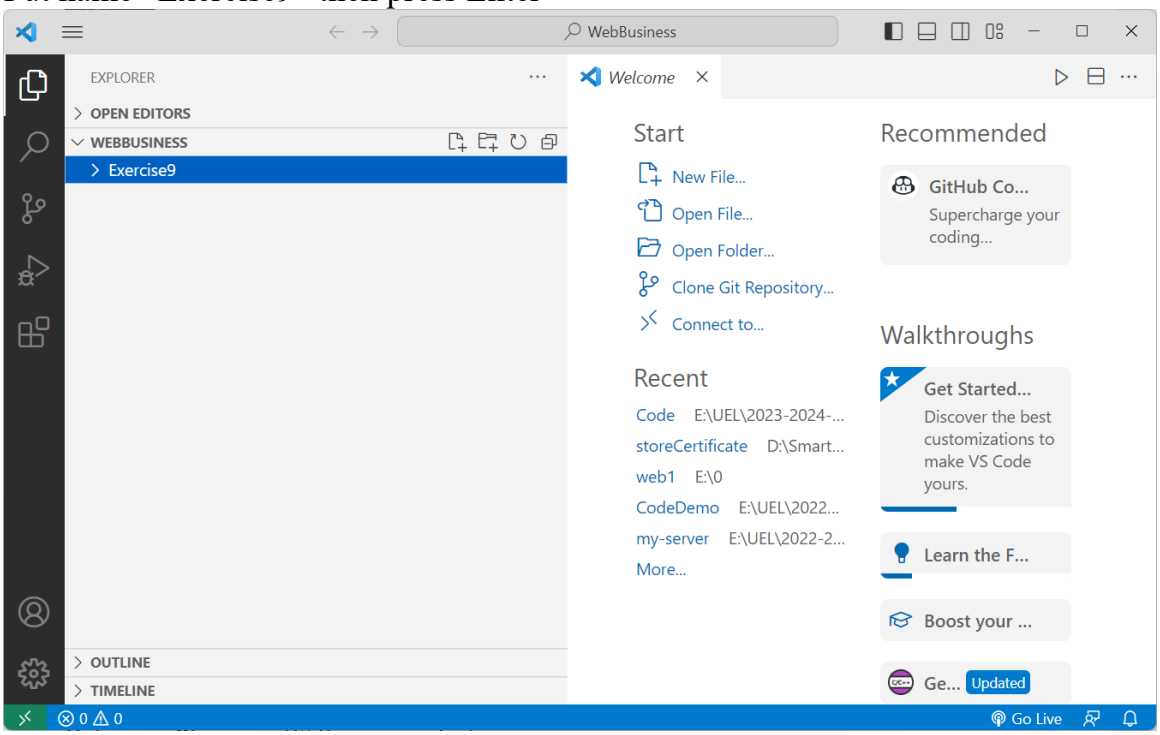
Click “Yes I trust the authors” if the screen below is popup.



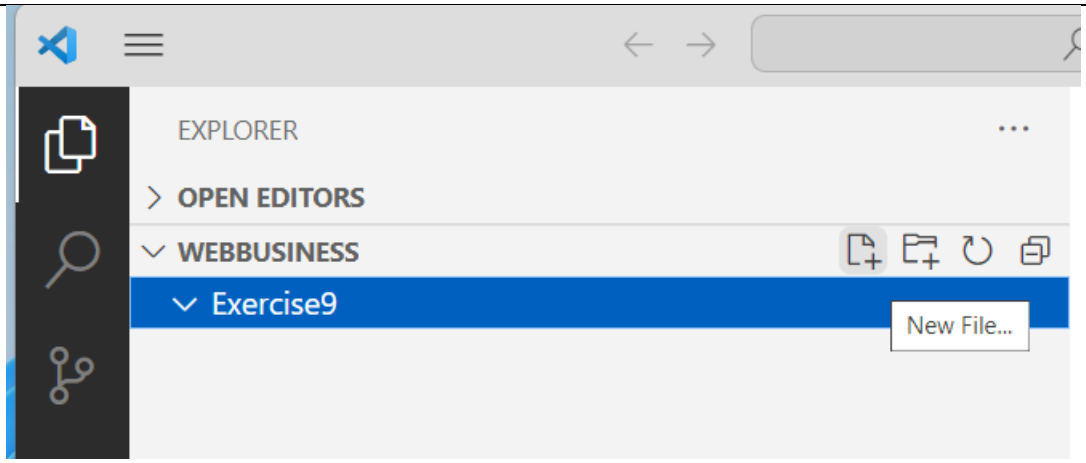
Then click New Folder button as picture above.



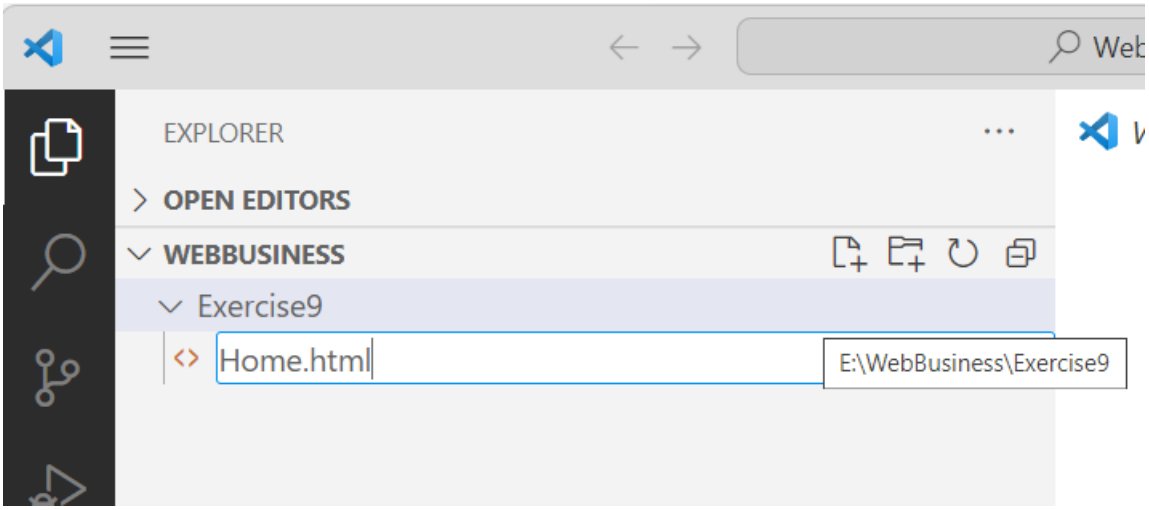
Put name “Exercise9” then press Enter



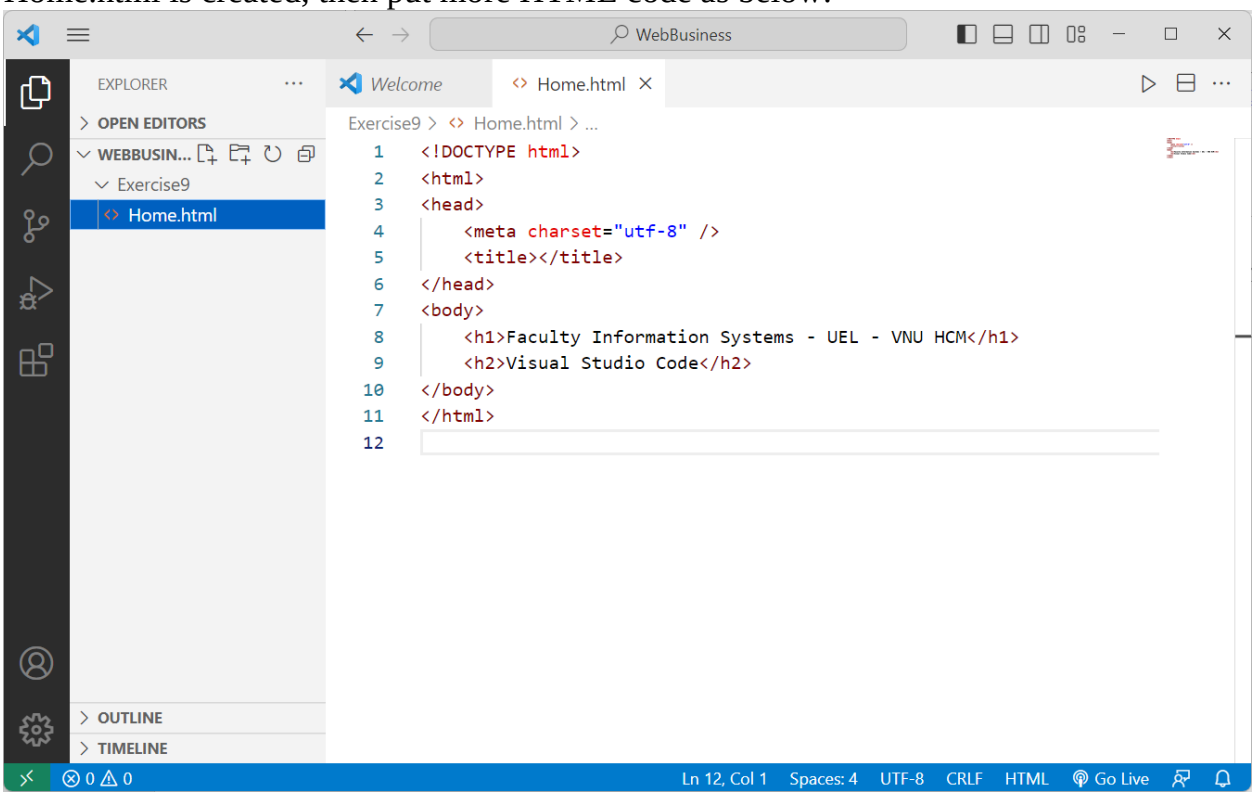
Create a new file (HTML file) in Exercise9, click button to create file:



Click button “New File…”

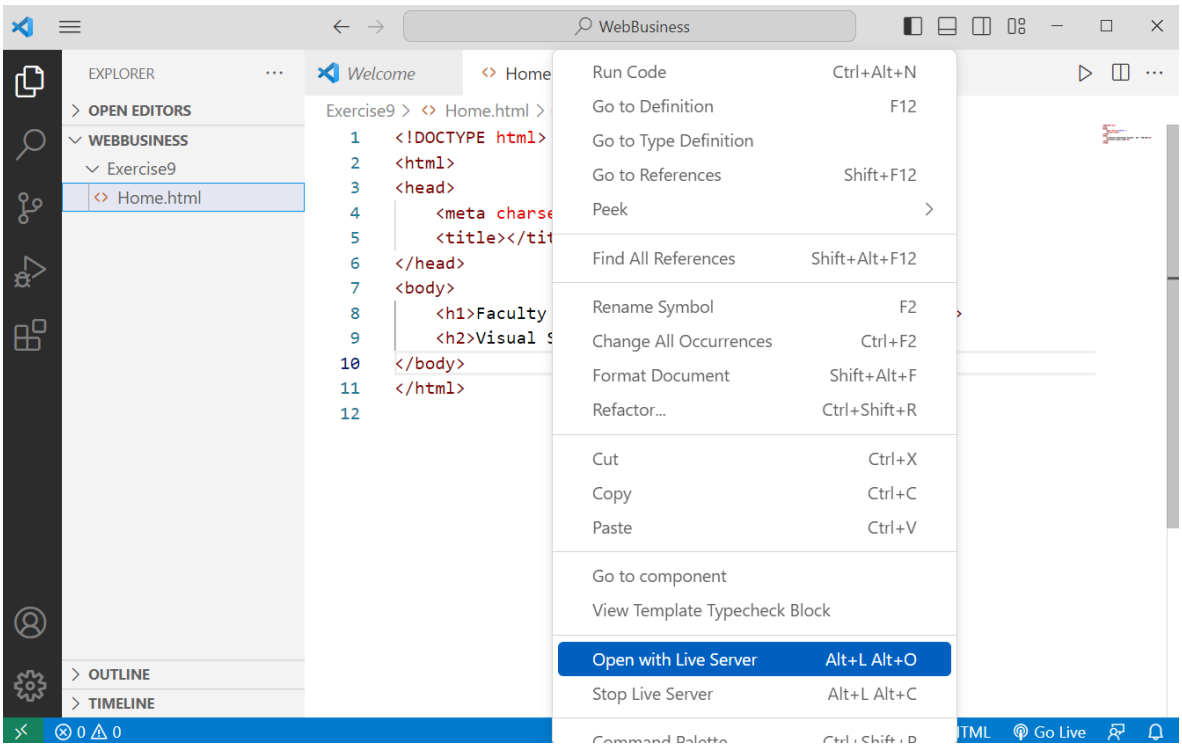


Enter “Home.html” then press Enter
Home.html is created, then put more HTML code as below:



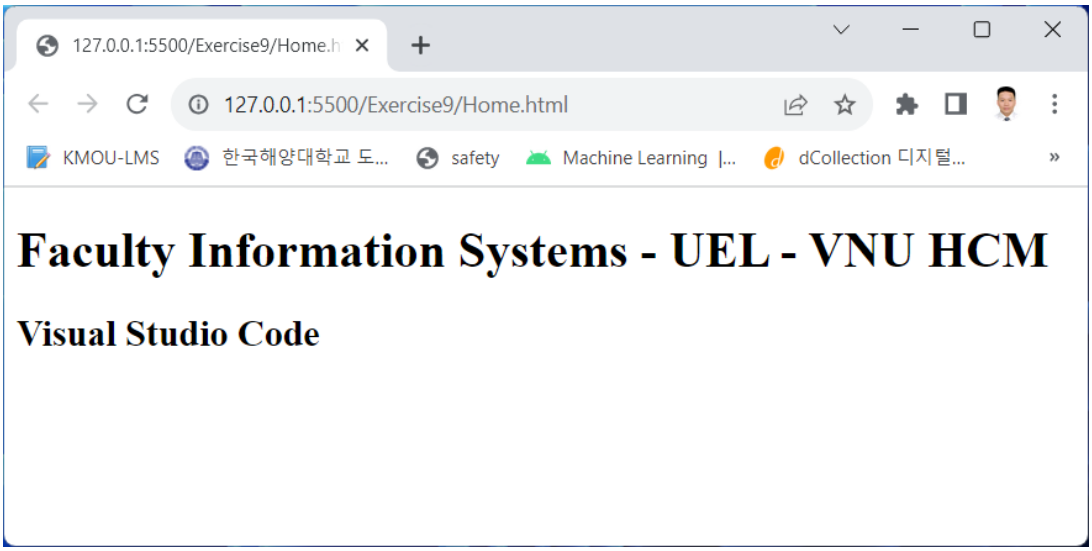
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h1>Faculty Information Systems - UEL - VNU HCM</h1>
    <h2>Visual Studio Code</h2>
</body>
</html>
```

Step 4: Run the HTML project



Right click on the "**Home.html**" then click "**Open with Live Server**"

The result:



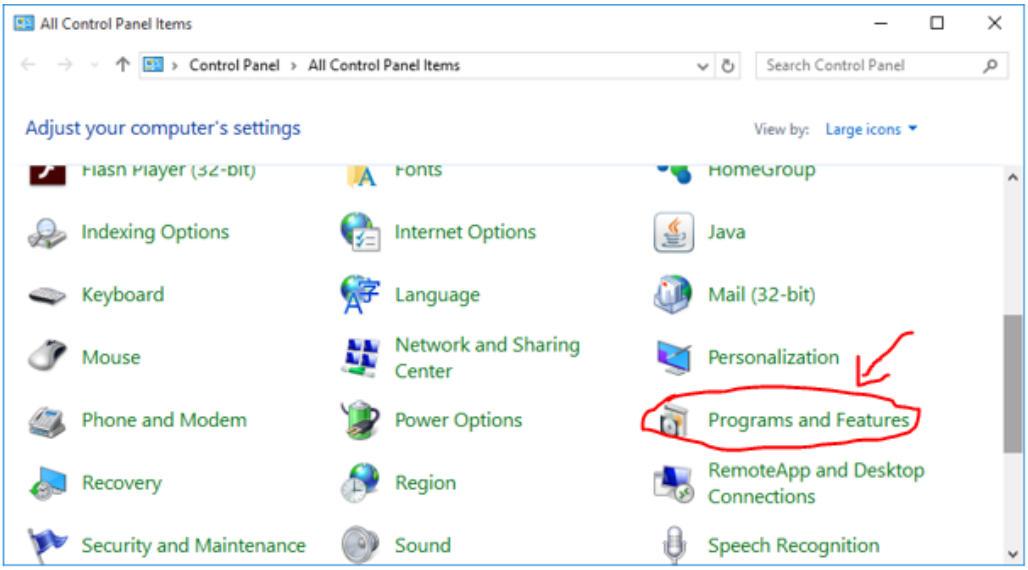
Exercise 10: Configure IIS Web Server

Requirement:

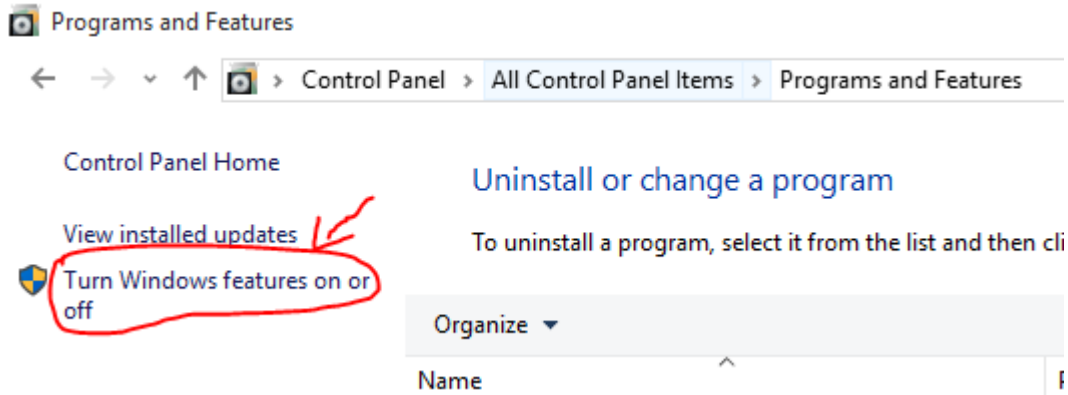
Let's configure IIS Webserver and deploy exercise 1 to WebServer so that this Website can be run on LAN.

Instructions:

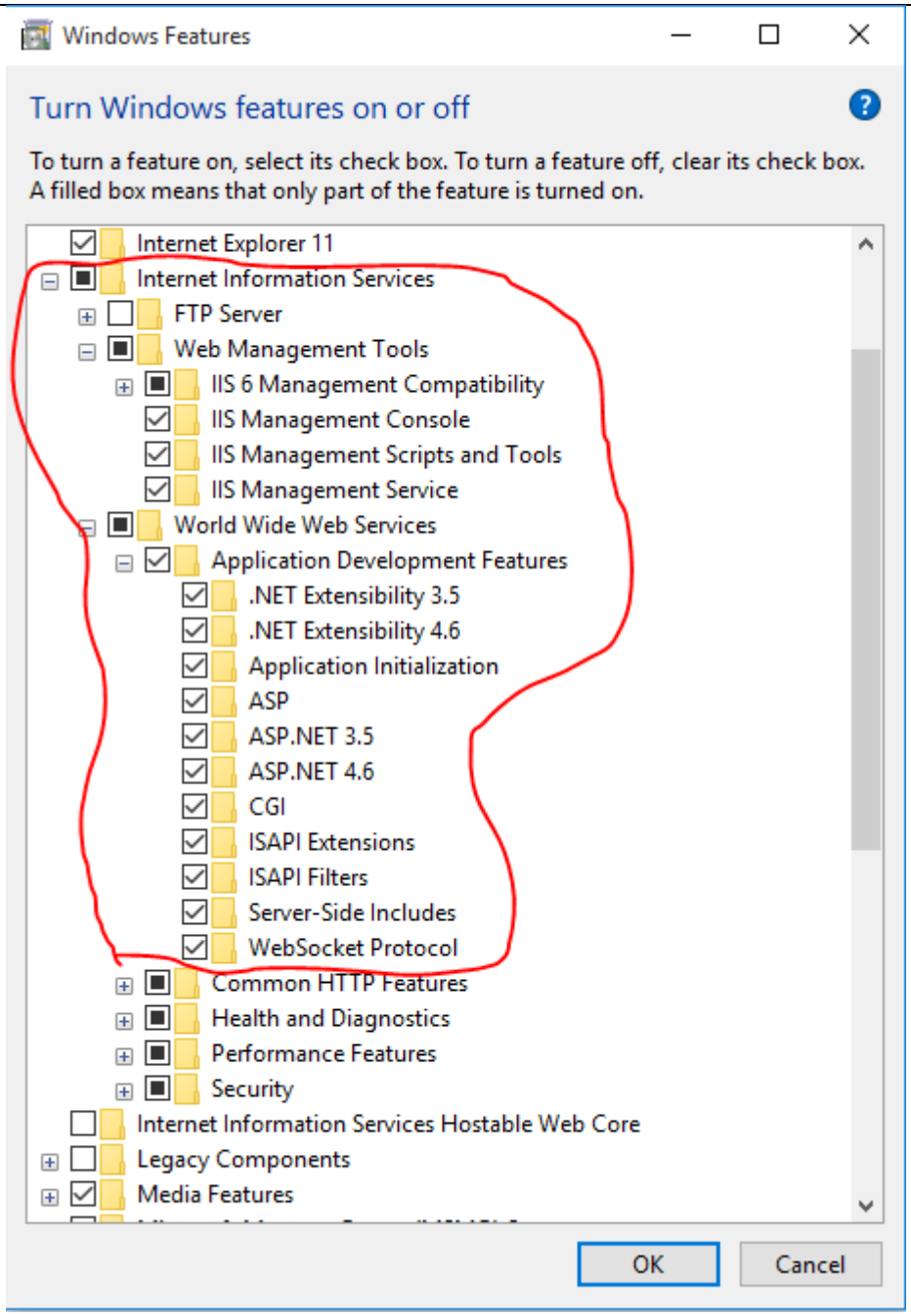
Step 1: Go to Control panel and select Programs and Features



Step 2: Select Turn Windows features on or off:

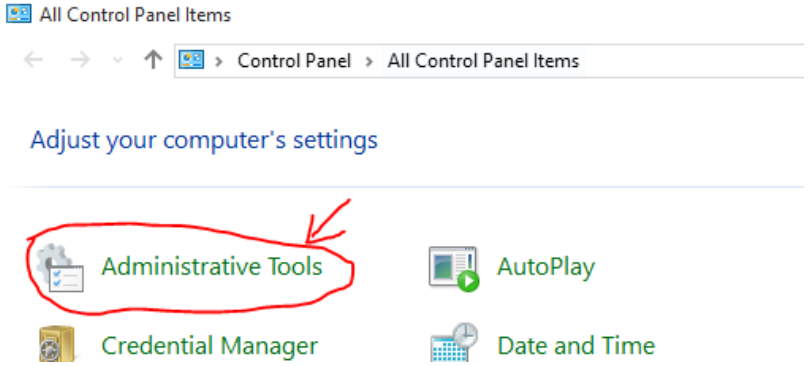


Step 3: Make the selection to install IIS Web Server according to the screen below:

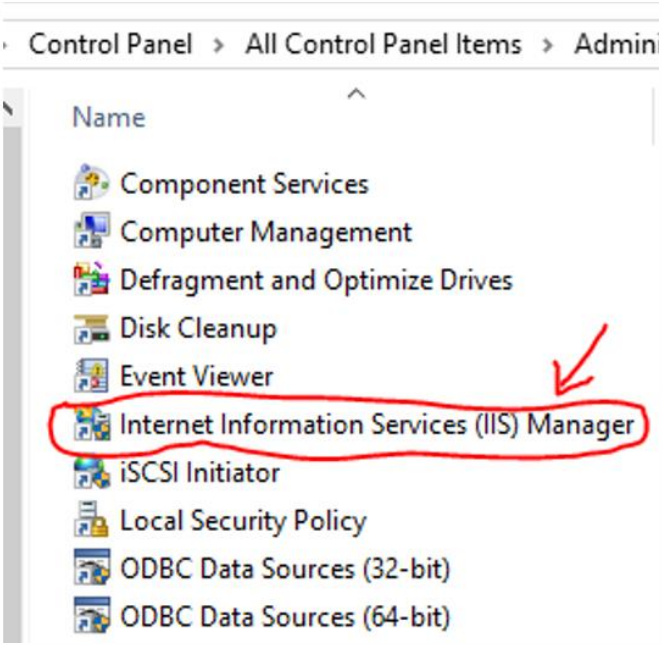


After selecting like the red circled frame, click OK to proceed with the installation, depending on the machine, the program can install from 5-10 minutes.

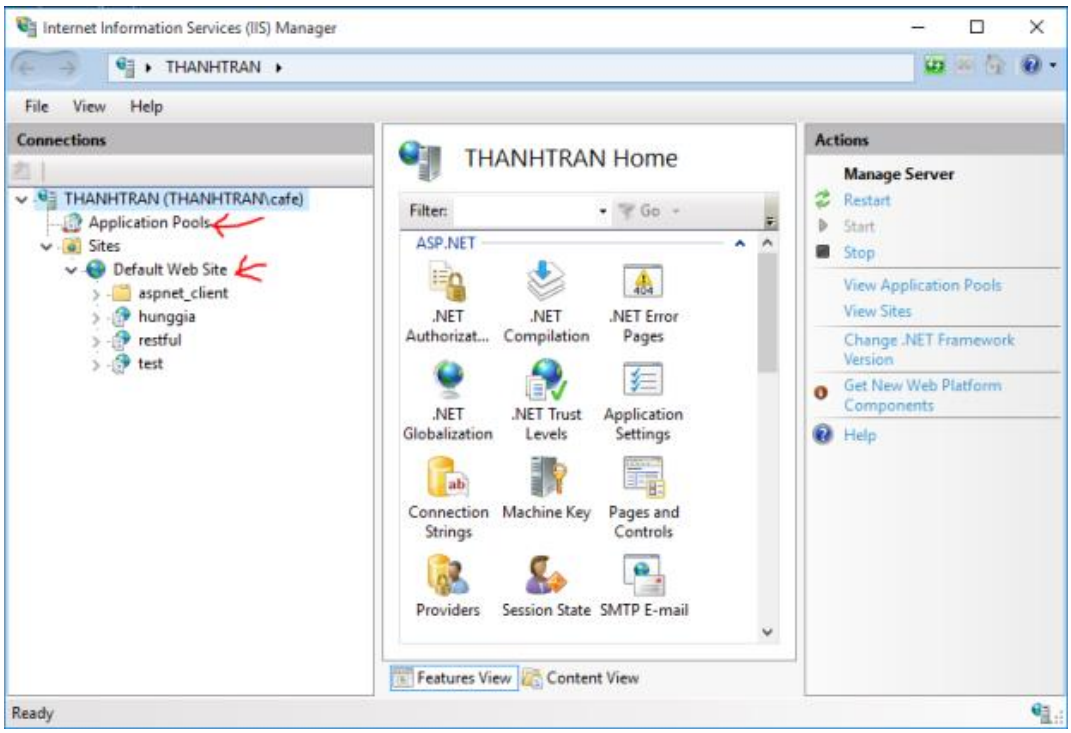
Step 4: Using IIS Web Server: After installing IIS Web Server, you return to the Control panel screen, you will see Administrative Tools:



You click on the Administrative Tools icon, we have the next screen:



You select "**Internet Information Services (IIS) Manager**" as shown above:



In the admin screen you see there are 2 areas: Application Pools and Default Web site.

Step 4.1 Configure Application Pools:

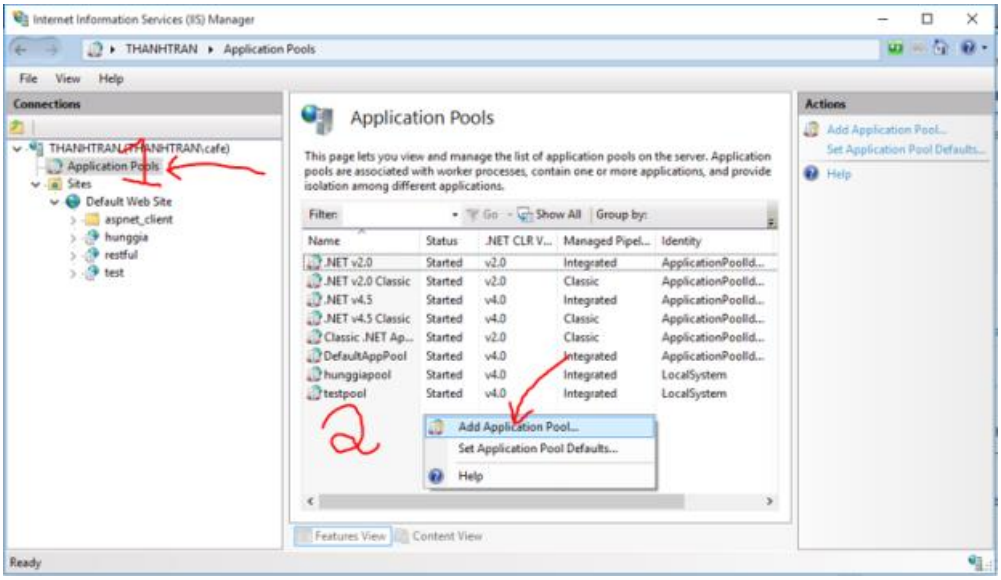
What is Application Pool?

The Application Pool can contain one or more applications and allows us to configure the level between different web applications. For example, if you want to isolate all web

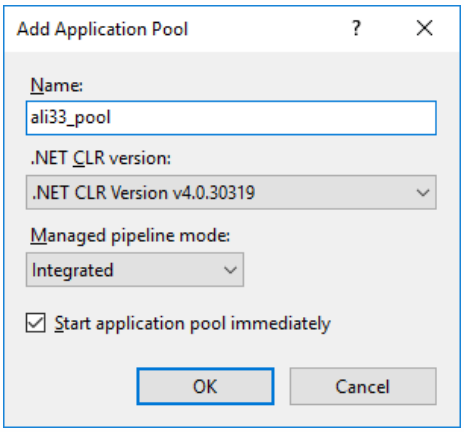
applications running on the same machine, you can do this by creating a separate Application Pool for each web application and placing them in the respective Application Pool. Because each Application Pool runs in its own process, failures in the Application Pool will not affect applications running in other Application Pools. Deploying applications in the Application Pool is the main advantage of IIS during isolated work because you can customize the Application Pool to achieve the level of application isolation you need.

As you configure the Application Pool for optimal availability, you should also consider how to configure the Application Pool to secure the application. For example, you may need to create a separate Application Pool for applications that require a high level of security, while allowing applications that require a lower level of security to share the same Application Pool.

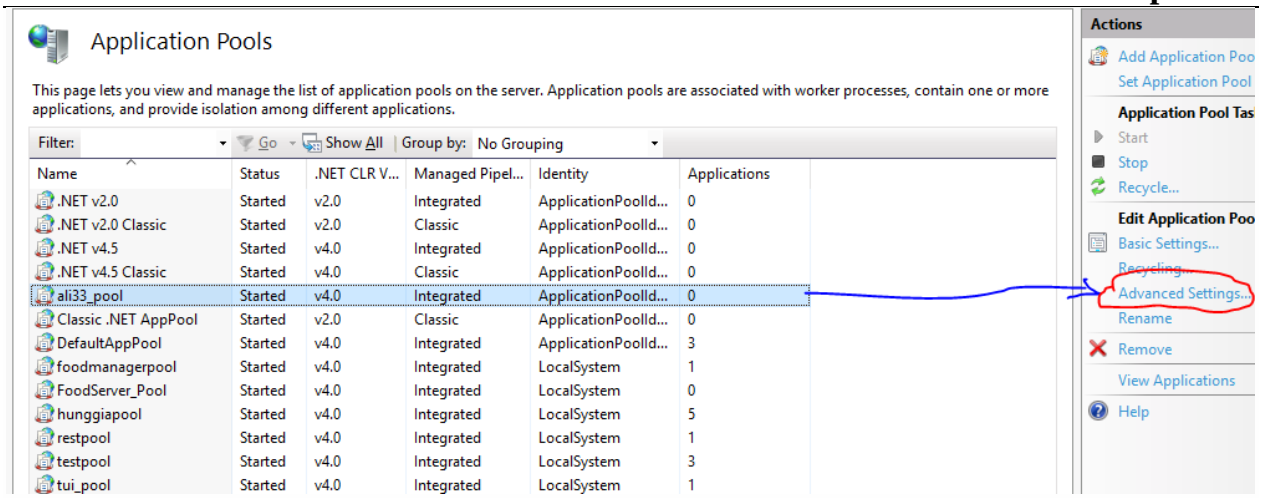
You right-click on the **Application Pools** screen / select **Add Application Pool**... as shown below:



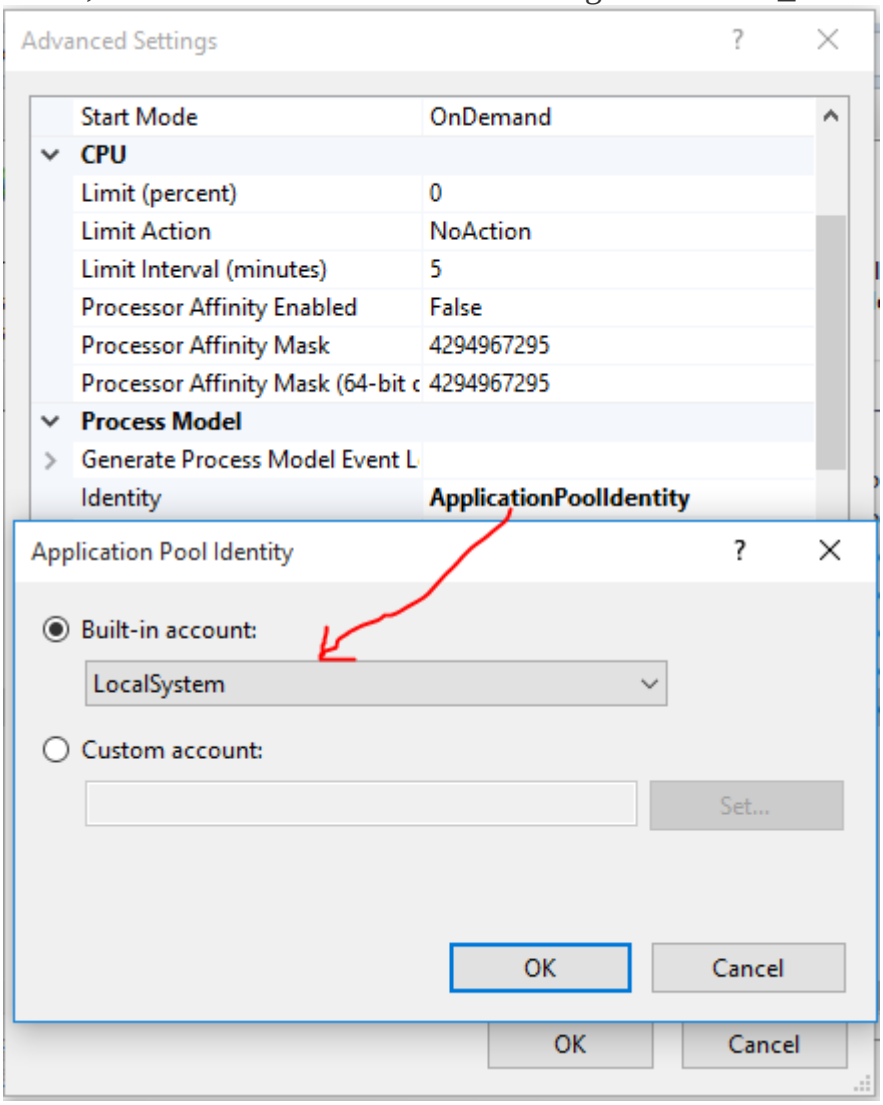
The screen to create a new Application Pool is displayed, enter "ali33_Pool" and then click OK:



After clicking OK, you observe:



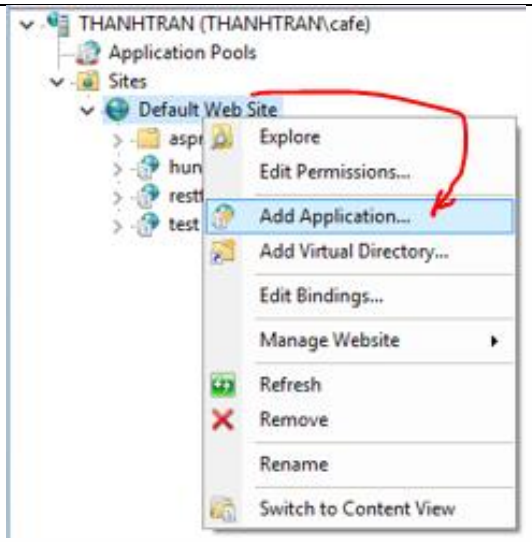
In the above screen, continue to select Advanced Settings… for ali33_Pool:



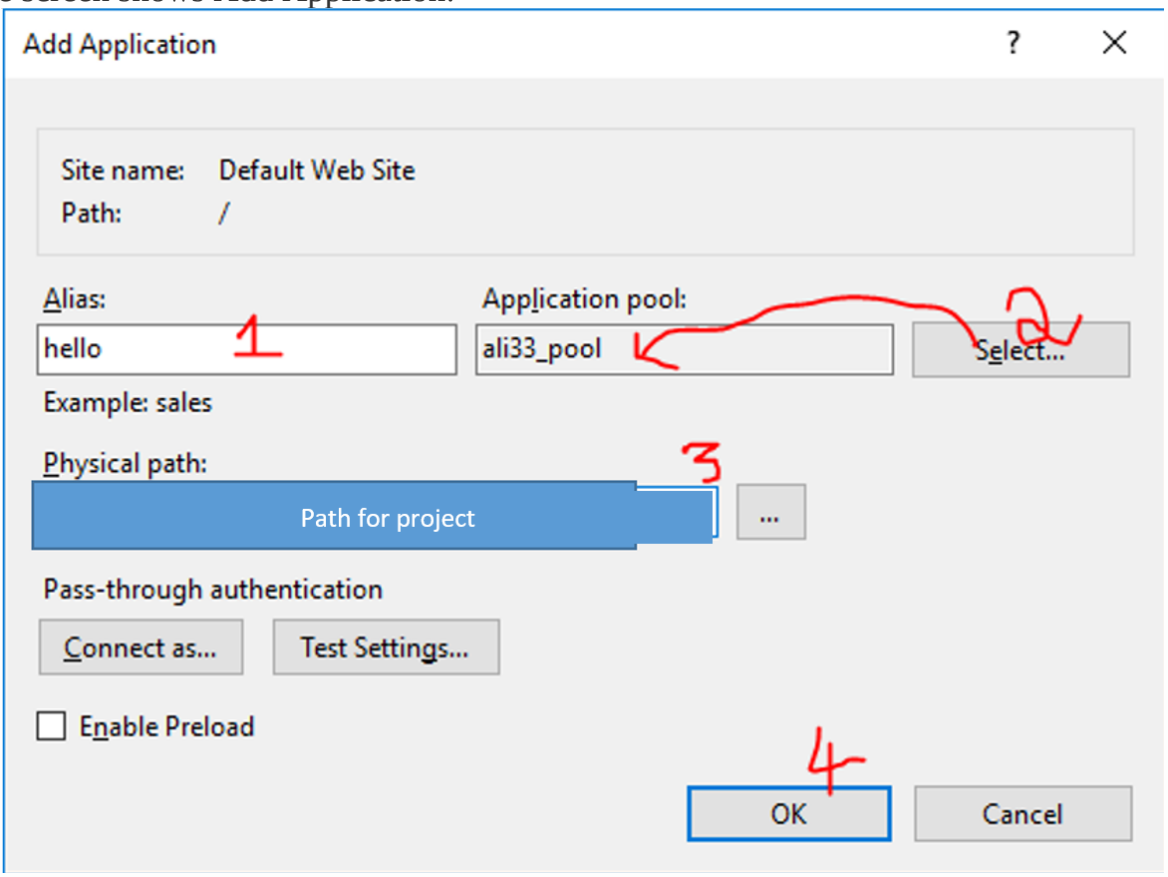
You need to set the **ApplicationPoolIdentity** via **LocalSystem**.

Step 4.2: Configure WebService on IIS Server

Right-click **Default Web Site** / select **Add Application**…



The screen shows Add Application:



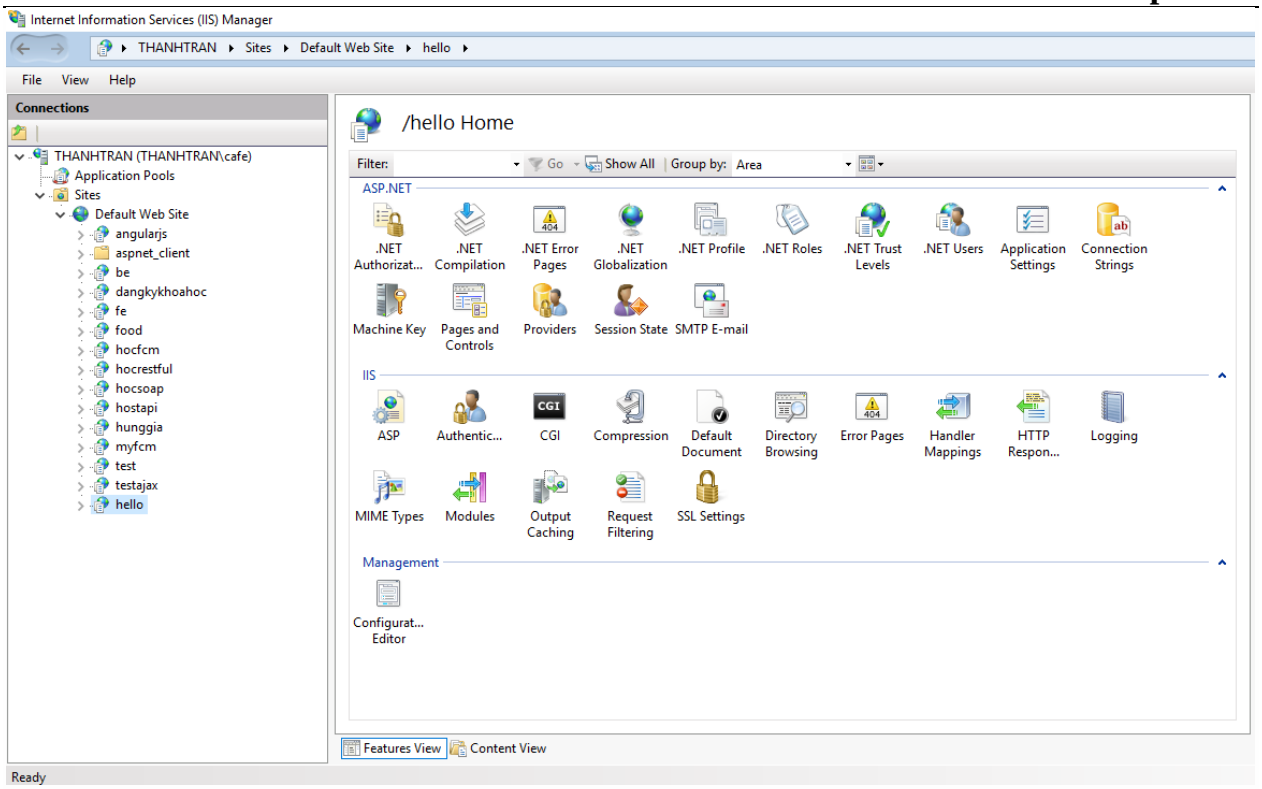
Section **Alias** (1): Name the Website, here we put hello

Application pool (2): Press the Select… button and select the correct ali33_pool

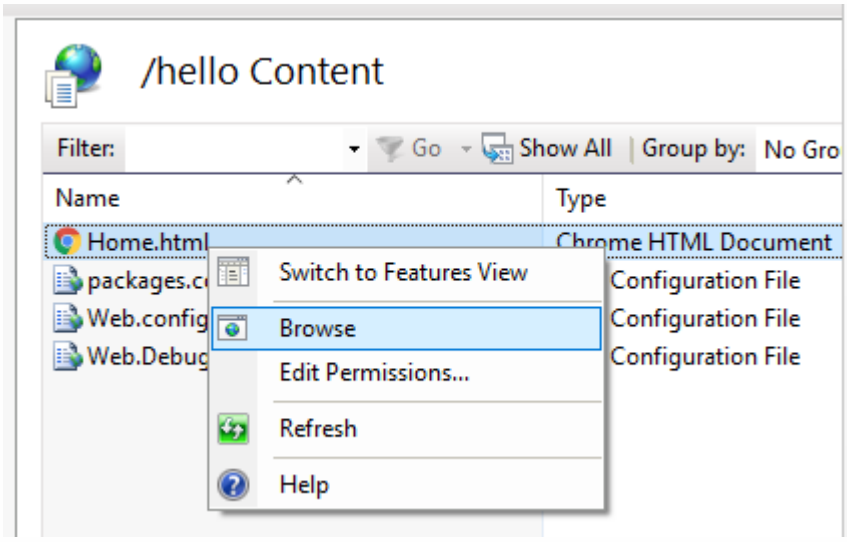
Physical path (3): Point to the path containing the source code of the project

Then click OK to create.

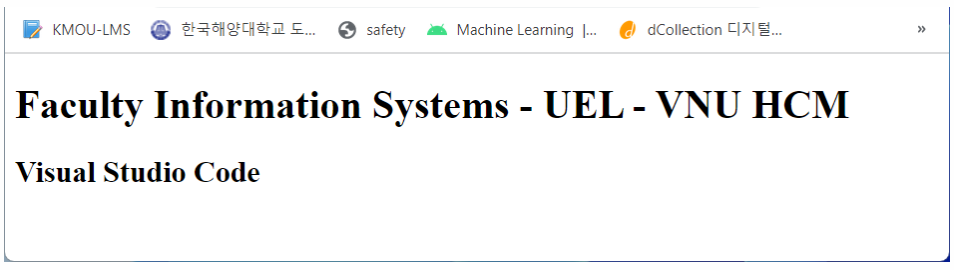
You observe the results:



We select hello in the Default Web Site group ➔select the Content View Tab (see below):

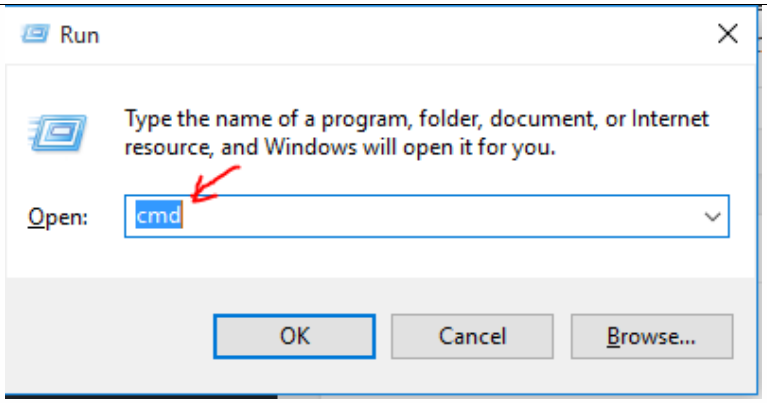


To run a page, right-click it and select Browse:

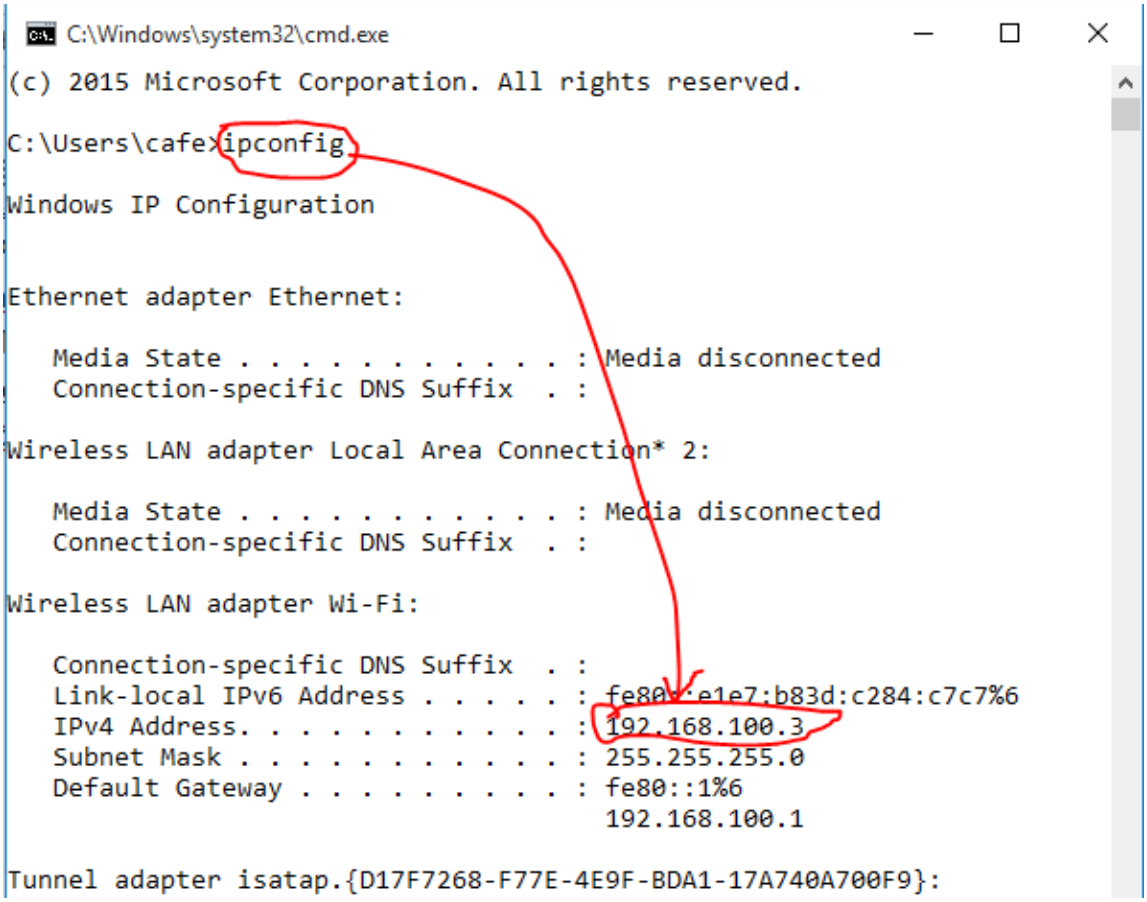


Step 4.3: How to get the IP address of my local machine:

– Press the key combination with Windows + R icon to open the Run window:



– Type **cmd** command, command line screen displays, continue to type **ipconfig** command



Above We see the IP address 192.168.100.3 ➔depending on your machine it will be different
You replace localhost (or 127.0.0.1) to 192.168.100.3 , then other computers will access your machine according to this IP address.

Step 4.4: How to Configure Default Home Page for Website

We remember that a Website has many pages, must have a default page as the homepage when the Website is operated➔we need to configure. The default IIS Web Server takes the Sites with the following names as Home Pages:

Default.htm

Default.asp

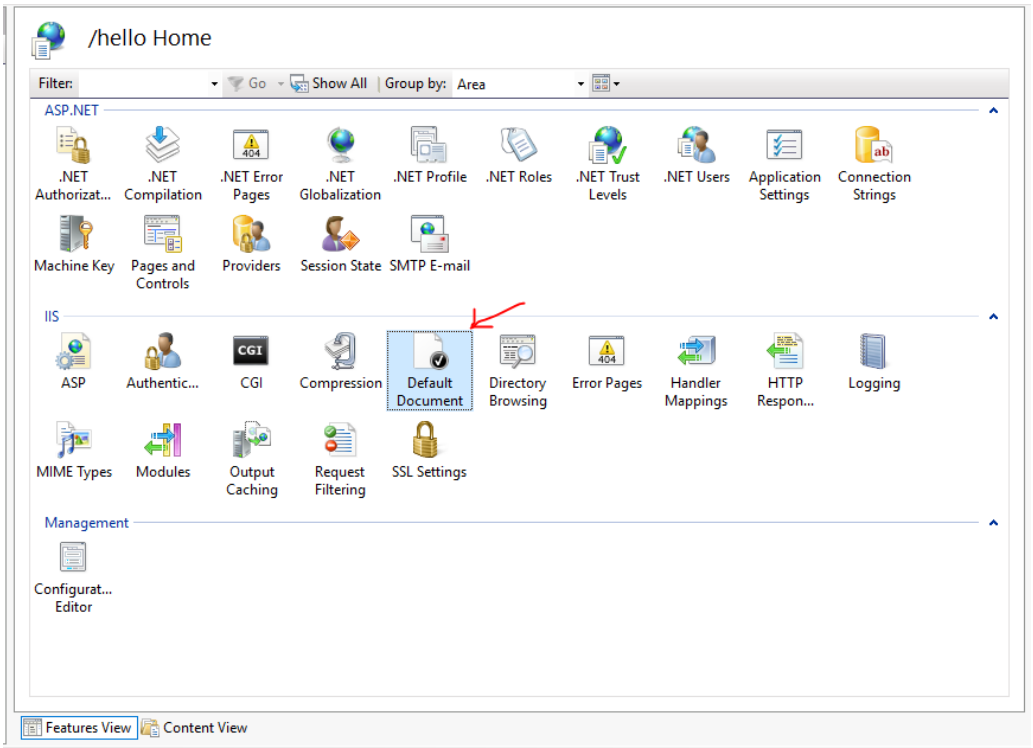
Default.aspx
index.htm
index.html
iisstart.htm

Therefore, if in the Website system, IIS does not find any Web site name with the same name above, running it will give an error that cannot see the homepage, we have 2 solutions to make the homepage:

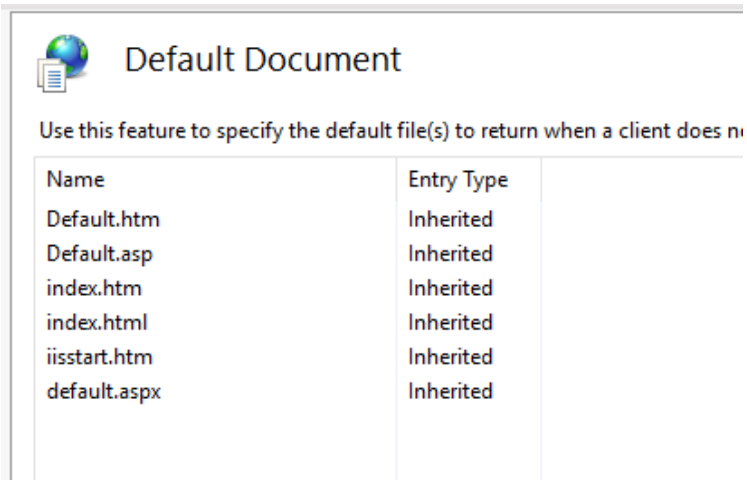
Solution 1: Have a Page with the same name as the default files above

Solution 2: Specify a page with any name as the homepage

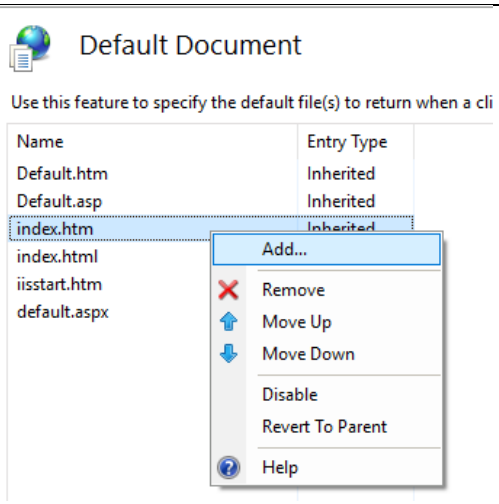
We will handle solution 2➔Hello on the right choose Default Document:



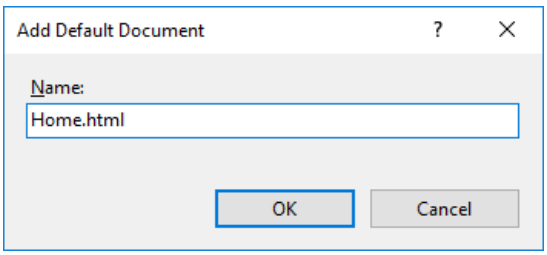
You see the Default Document list displayed as below:



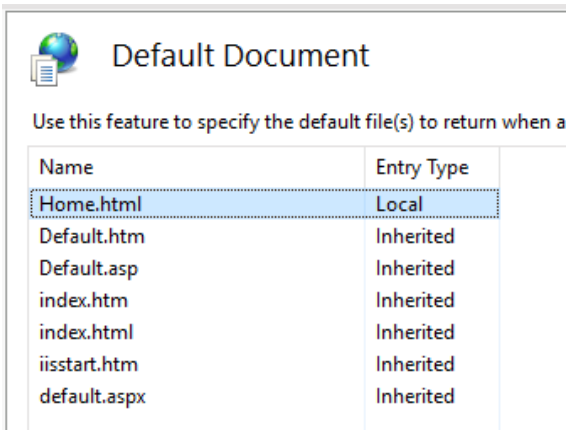
Right-click and select Add:



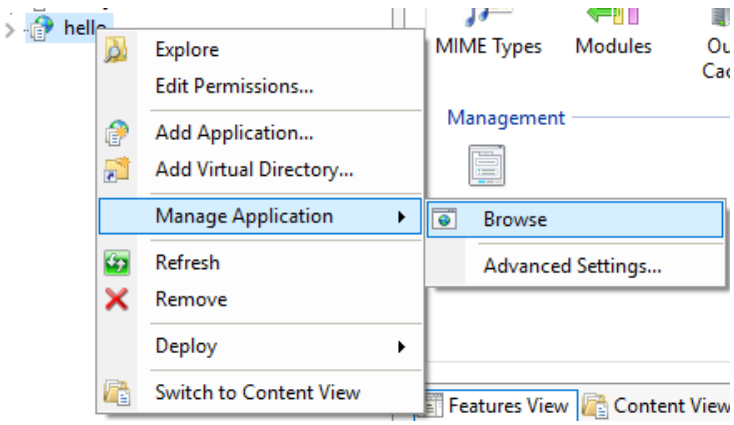
Then type the name of the Web site that you want to set as the homepage:



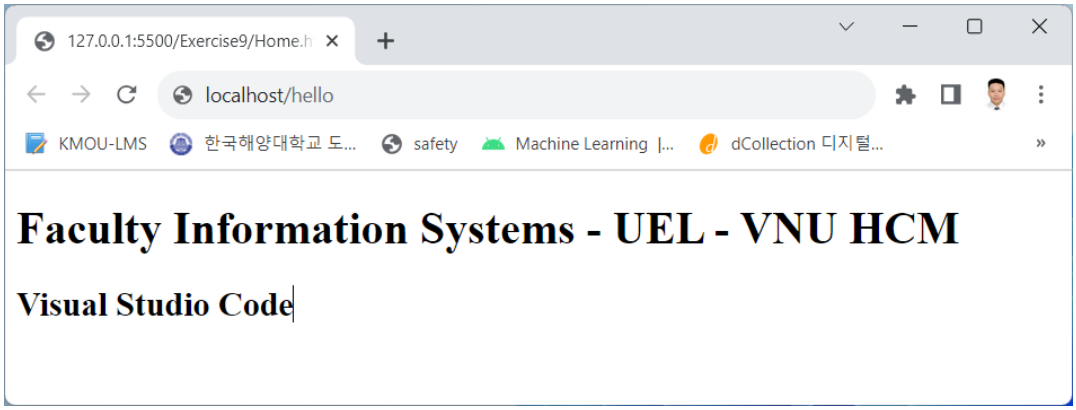
Click OK to set up:



After successful setup, right-click hello / select Manage Application / select Browse:



Result:



Note: We no longer see the Home.html page displayed in the address bar (remember that if you don't set the homepage, IIS will not find it when you run a Website on IIS, it will give an error). Setting up the Home Page is mandatory, for example, if you have a Login.html page, if you want this page to display first, you must configure it as Home.

Exercise 11: *Configure Hosting + Domain Free*

Requirement:

Let's deploy exercise 9 to any free Hosting introduced in the class so that any place in the world can access the Website:

https://azure.microsoft.com (email education)

https://somee.com

https://freevnn.com

https://www.000webhost.com

https://www.freehosting.com

https://www.awardspace.com/free-hosting

https://vhost.vn/hosting/free-hosting

Instructions:

Students can find out for themselves, If using Somee.com, you can refer here:

https://tranduythanh.com/2014/02/25/bai-44-cach-tao-webservice/

Exercise 12: Configure NAT port for Internet Modem at home (*)

Requirement:

Learn how to NAT Port for Internet Modem at home to turn a personal computer into a machine with a Web Server that can be accessed anywhere in the world: Go to IIS, Remote Desktop…

Instructions:

- Check which company your network modem belongs to, because each company will have a different NAT port.
- Learn Dynamic DNS (https://www.noip.com)
- Learn the software Update Client for DynDN
- (https://www.noip.com/download?page=win)

➔ Note that when the configuration is successful, our computer can be illegally penetrated anywhere in the world, so we must be careful about information security. The computer we leave at home, but we can be anywhere in the world, we can Remotely access the machine, use it normally as if we were holding a computer in our hands. If you work in a small and medium-sized company, have a moderate server, and to save a lot of costs on setting up a server (there are data that cannot be ordered), one can use this solution to operate data mining in the company.

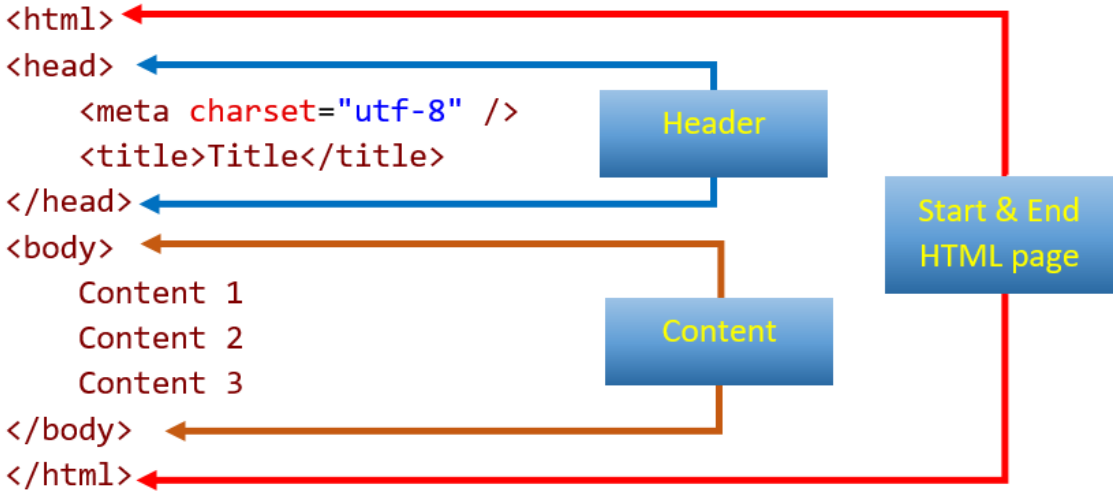
Exercise 13: HTML file structure

Requirement:

Describe the structure and meaning of the main components of an HTML Static Web Page. Explain opening and closing tags.

Instructions:

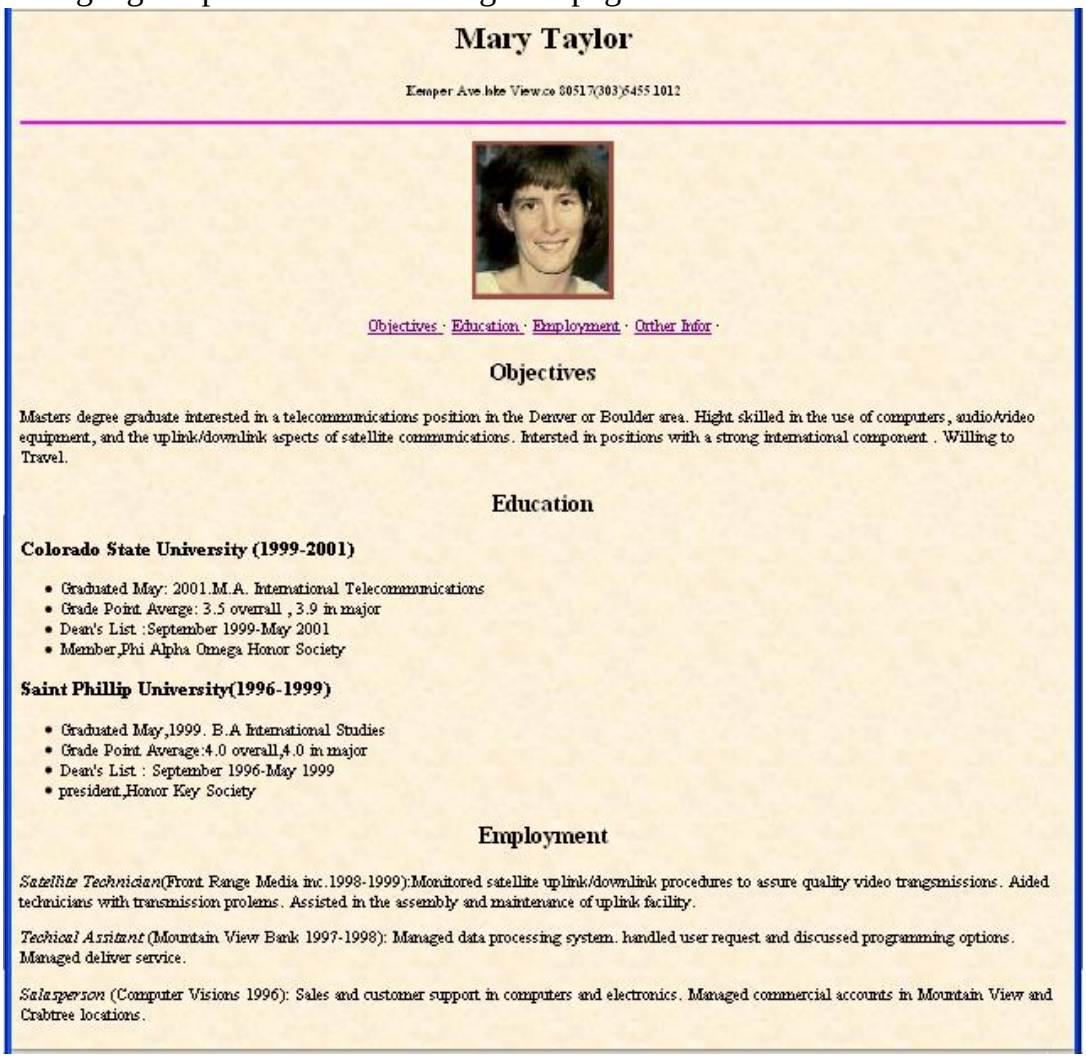
See the figure below to show the structure of an HTML Static Web page.



Exercise 14: Resume

Requirement:

Create a **Resume.html** web page, using the <Hn>, <UL>, <HR> , <IMG>, <A> tags and the formatting tags to present the following web page:



Mary Taylor

Kemper Ave lake View,co 80517(303)6455 1012

Objectives · Education · Employment · Orther Infor ·

Objectives

Masters degree graduate interested in a telecommunications position in the Denver or Boulder area. Hight skilled in the use of computers, audio/video equipment, and the uplink/downlink aspects of satellite communications. Intersted in positions with a strong international component . Willing to Travel.

Education

Colorado State University (1999-2001)

- Graduated May: 2001.M.A. International Telecommunications
- Grade Point Averge: 3.5 overrall , 3.9 in major
- Dean's List :September 1999-May 2001
- Member,Phi Alpha Omega Honor Society

Saint Phillip University(1996-1999)

- Graduated May,1999. B.A International Studies
- Grade Point Average:4.0 overall,4.0 in major
- Dean's List : September 1996-May 1999
- president,Honor Key Society

Employment

Satellite Technician(Front Range Media inc.1998-1999):Monitored satellite uplink/downlink procedures to assure quality video trangsmissions. Aided technicians with transmission prolems. Assisted in the assembly and maintenance of uplink facility.

Techical Assitant (Mountain View Bank 1997-1998): Managed data processing system. handled user request and discussed programming options. Managed deliver service.

Salesperson (Computer Visions 1996): Sales and customer support in computers and electronics. Managed commercial accounts in Mountain View and Crabtree locations.

Instructions:

Image file and attached data in folder Exercise14.

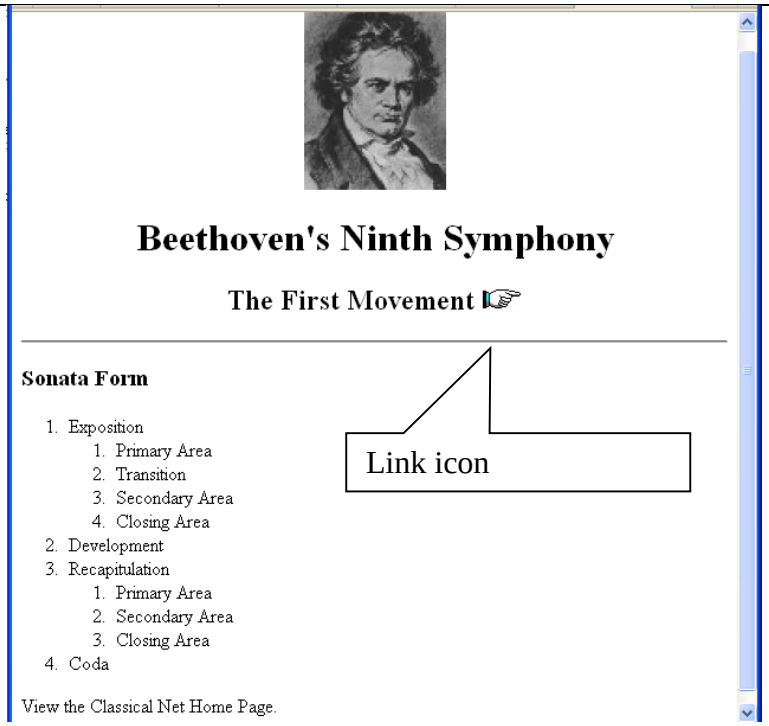
http://teacher.uelstore.com/businesswebdevelopment/ex14.zip

In addition, students can use the image as a background for the page.

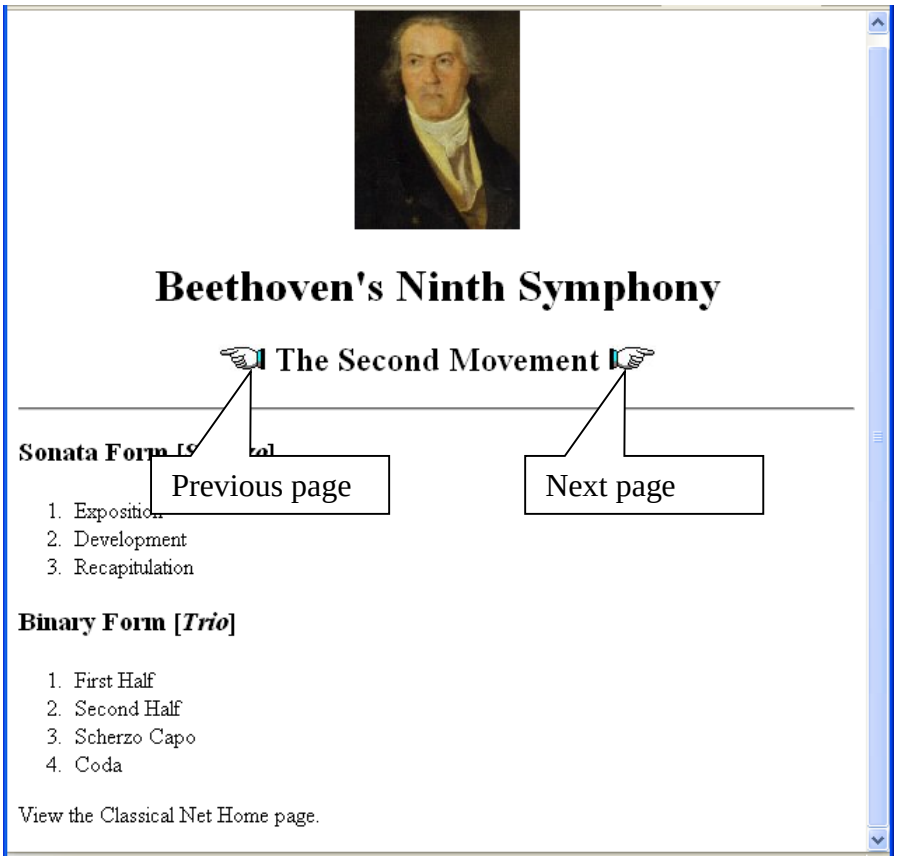
Exercise 15: Beethoven

Requirement:

Create a Beethoven website consisting of 4 linking pages, each page will use an image to link to the next page and the page before it. Use the <IMG>, <HR>, <OL>, <Hn>, <A> tags to display the web pages in the website, choose the background image for the pages yourself.

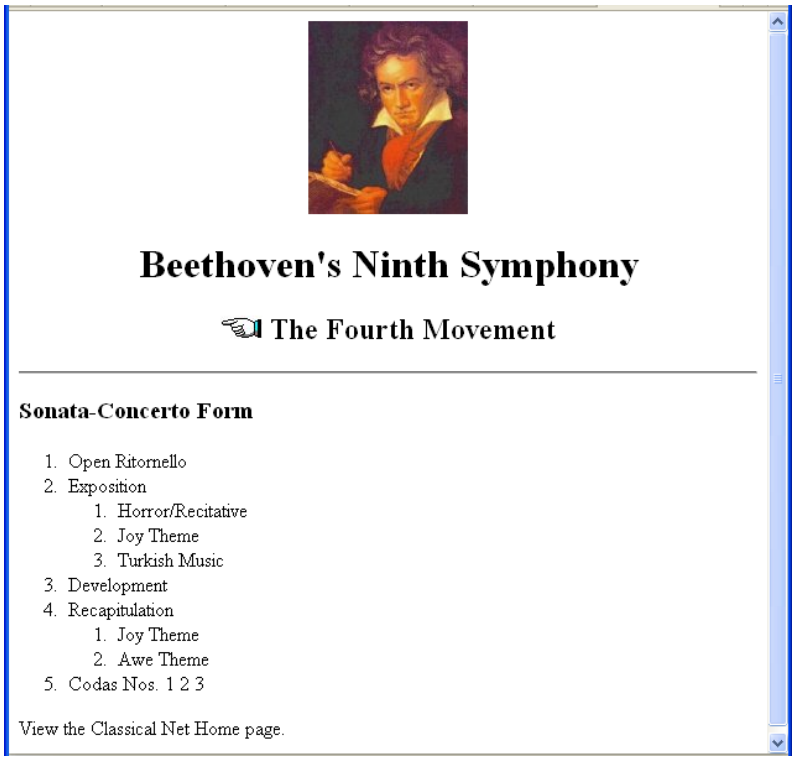


FirtMovement.html



SecondMovement.html

ThirdMovement.html



FourthMovement.html

Instructions:

Image file and attached data in folder Exercise15.

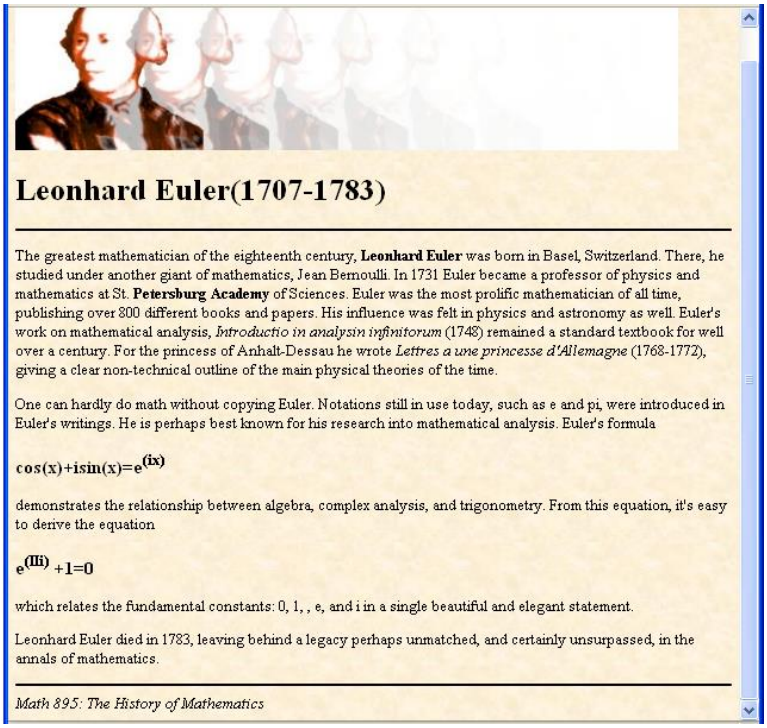
http://teacher.uelstore.com/businesswebdevelopment/ex15.zip

Students manually separate the data in the file to display it as required. In addition, students can use images as background for the page. The index finger icon finds itself online.

Exercise 16: Euler

Requirement:

Design the **Euler.html** website as shown below:



Instructions:

Image file and attached data in folder Exercise16.

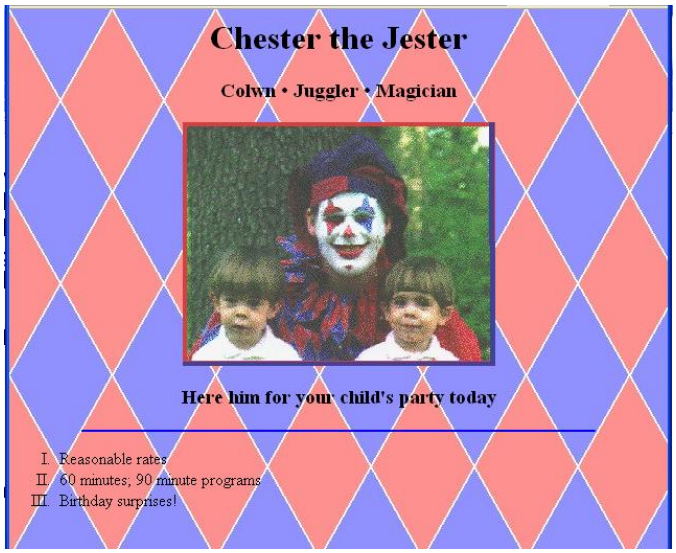
http://teacher.uelstore.com/businesswebdevelopment/ex16.zip

In addition, students can use the image as a background for the page.

Exercise 17: Chester

Requirement:

Design the **Chester.html** website as shown below:



Instructions:

Image file and attached data in folder Exercise17.

http://teacher.uelstore.com/businesswebdevelopment/ex17.zip

In addition, students can use the image as a background for the page.

Exercise 18: Resume (more)

Requirement:

Open the **Resume.html** page again in Exercise 14 and add the following content at the bottom of the page, then create links for the items:

- **References** ➔open page **Reference.html.**
- **Comments on my work** ➔open page **Comments.html.**
- **Go to Colorado State** ➔open page **www.colostate.edu**

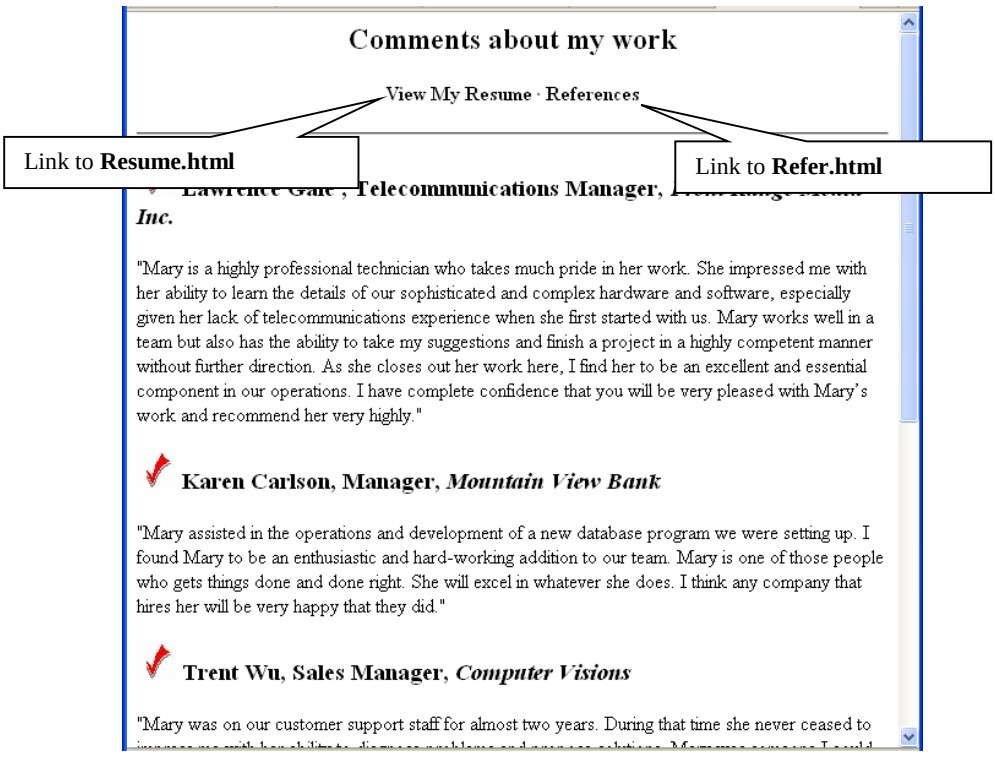
Other Information

- References
- Comments on my work
- Go to Colorado State

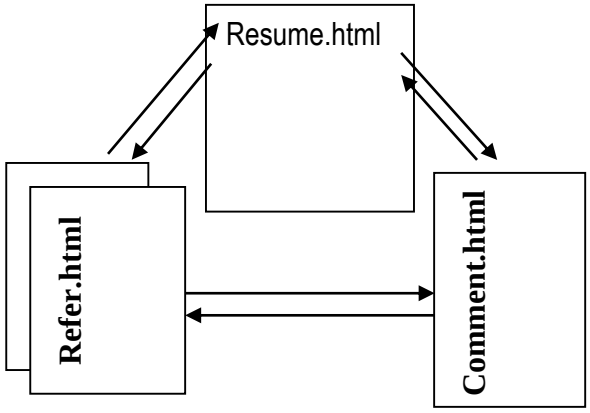
Interested?

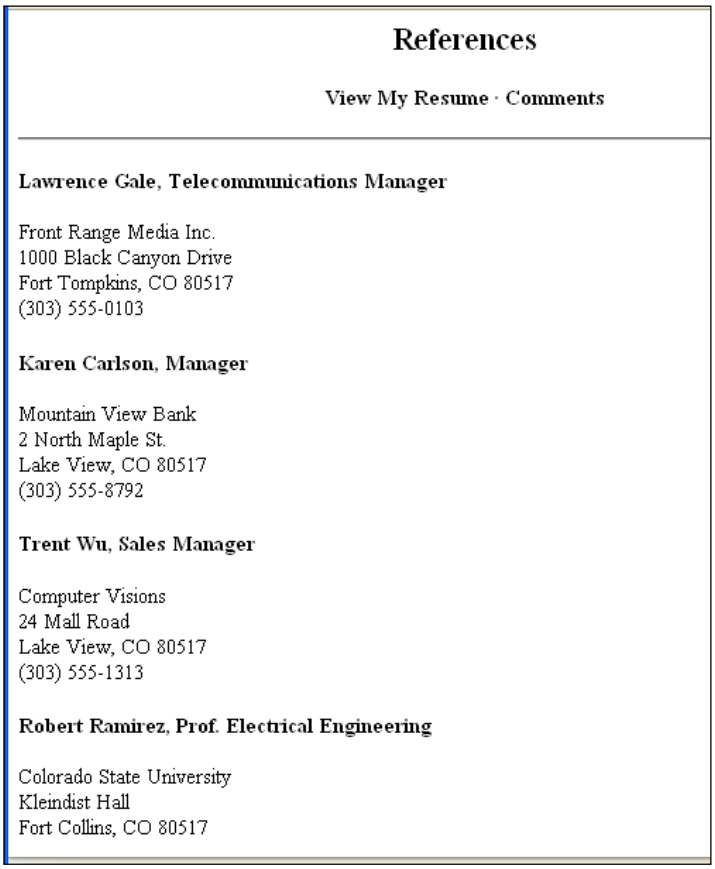
Contact Mary Taylor at mtaylor@tt.gr.csu.edu

The contents of the **Refer.html** and **Comments.html** pages are as follows:



Create links between 3 pages according to the model:





References

View My Resume · Comments

Lawrence Gale, Telecommunications Manager

Front Range Media Inc.
1000 Black Canyon Drive
Fort Tompkins, CO 80517
(303) 555-0103

Karen Carlson, Manager

Mountain View Bank
2 North Maple St.
Lake View, CO 80517
(303) 555-8792

Trent Wu, Sales Manager

Computer Visions
24 Mall Road
Lake View, CO 80517
(303) 555-1313

Robert Ramirez, Prof. Electrical Engineering

Colorado State University
Kleindist Hall
Fort Collins, CO 80517

Open **Comments.html** page, create 2 more links to **Resume.html** page and **Refer.html** page:

- View My Resume link to **Resume.html**
- References link to **Refer.html**

Open Refer.html page, create 2 more links to **Resume.html** page and **Comment.html** page:

Instructions:

Image file and attached data in folder Exercise18.

http://teacher.uelstore.com/businesswebdevelopment/ex18.zip

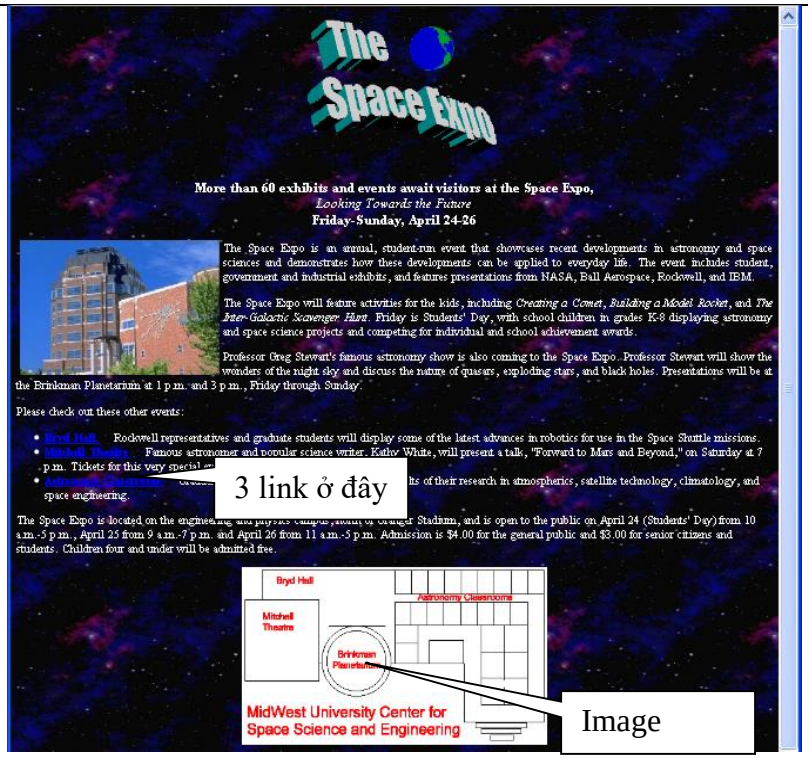
In addition, students can use images as background for the page.

Exercise 19: Website Expo

Requirement:

Create the **Expo.html** website as shown below:

- Bryd Hall (Text and Image Maps) links to Bryd.html
- Mitchell Theater (Text and Image Maps) link to Mitchell.html
- Astronomy Classrooms (Text and Image Maps) link to Brinkman.html



Instructions:

Image files, Bryd.html, Mitchell.html, Brinkman.html and attached data in folder Exercise 19.

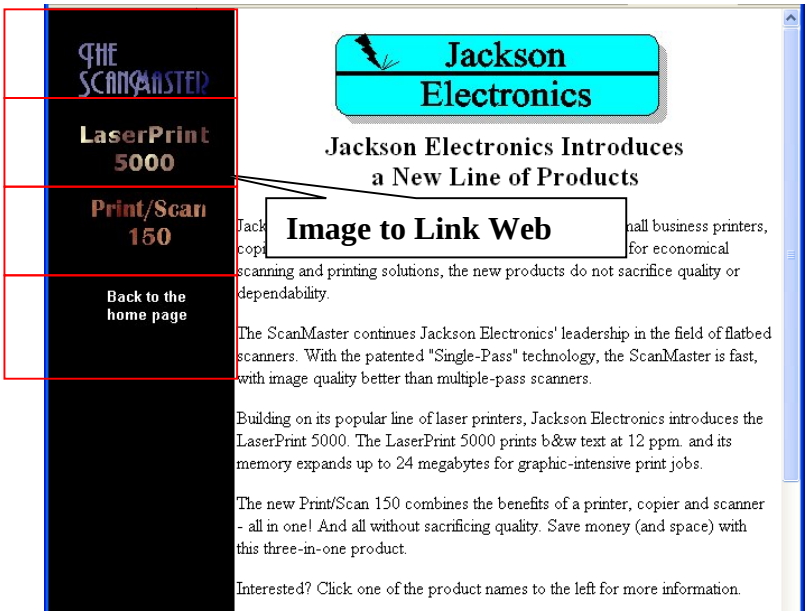
http://teacher.uelstore.com/businesswebdevelopment/ex19.zip

In addition, students can use the image as a background for the page.

Exercise 20: Website Jaction

Requirement:

Create **Jaction.html** website as shown below:



Instructions:

Image files, Scantext.htm, PStext.htm, Printext.htm and attached data in folder Exercise 20.

http://teacher.uelstore.com/businesswebdevelopment/ex20.zip

In addition, students can use the image as a background for the page.

Exercise 21: Website SFSF

Requirement:

Create **SFSF** website as shown below:

- Image map1: link to **Forsttxt.htm**
- Image map2: link to **Charntxt.htm**
- Image map3: link to **Unwintxt.htm**

Instructions:

Image files, Forsttxt.htm, Charntxt.htm, Unwintxt.htm and attached data in folder Exercise 21.

http://teacher.uelstore.com/businesswebdevelopment/ex21.zip

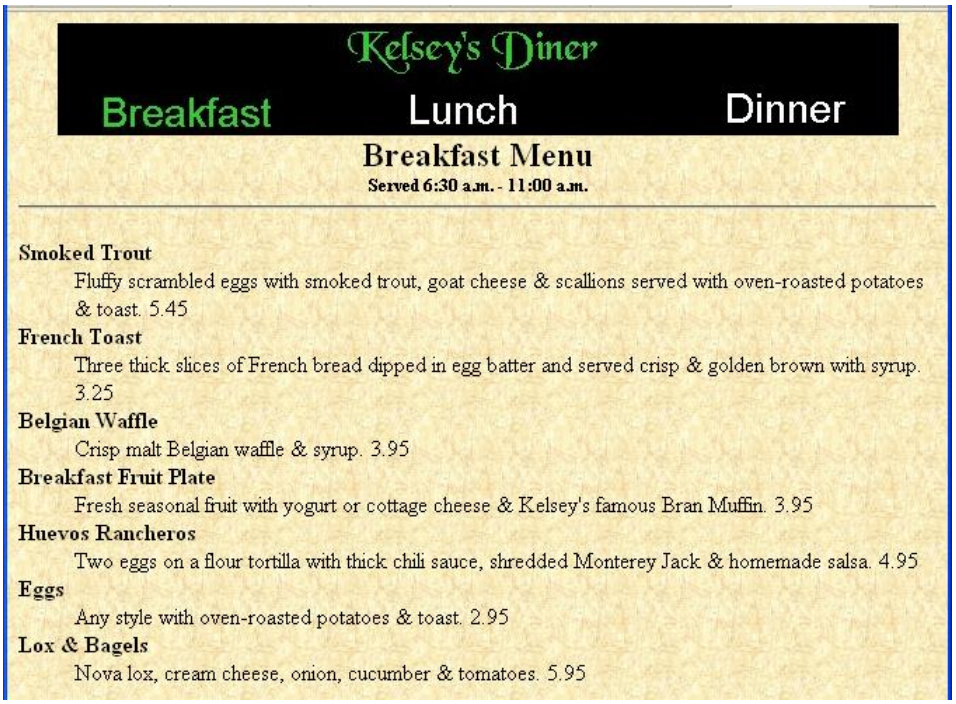
In addition, students can use the image as a background for the page.

Exercise 22: Web Menu

Requirement:

Create a Web menu for Kelsey's Dinner as described below:

- Breakfast points to **Breakfast.html**
- Lunch points to **Lunch.html**
- Dinner points to **Dinner.html**



Instructions:

Image file and attached data in folder Exercise 22.

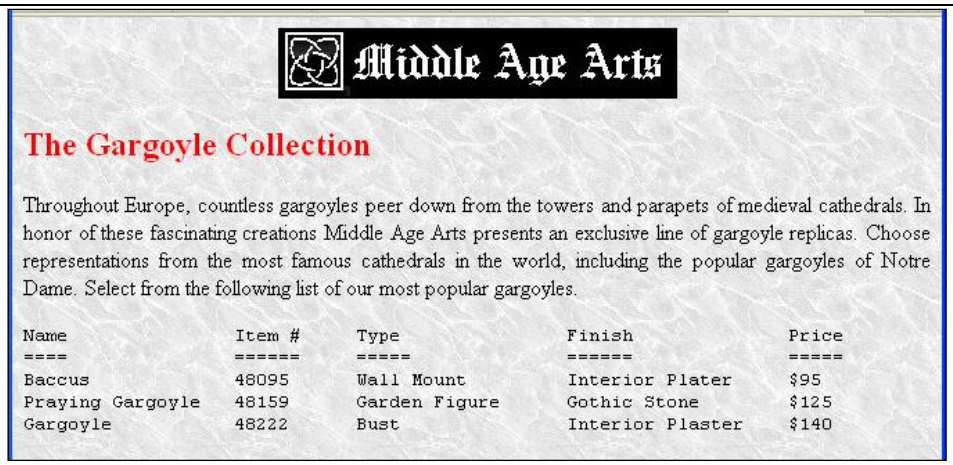
http://teacher.uelstore.com/businesswebdevelopment/ex22.zip

In addition, students can use the image as a background for the page.

Exercise 23: Table – Gargoyle Collection

Requirement:

Create a **MAU.html** web page containing a Text Table using the <pre> tag and the spacebar key, with the interface described below.:



Instructions:

Image file and attached data in folder Exercise 23.

http://teacher.uelstore.com/businesswebdevelopment/ex23.zip

In addition, students can use the image as a background for the page.

Exercise 24: Table – Gargoyle Products

Requirement:

Create a **MAA.html** container web page with the interface as described below:

Instructions:

Image file and attached data in folder Exercise 24.

http://teacher.uelstore.com/businesswebdevelopment/ex24.zip

In addition, students can use the image as a background for the page.

Exercise 25: *Table Gargoyle (*)*

Requirement:

Design the website The **Gargoyle.html**, using nested <Table> and learned tags: <A>, <IMG>, <Hn>, <UL> as described below:

Instructions:

Image file and attached data in folder Exercise 25.

http://teacher.uelstore.com/businesswebdevelopment/ex25.zip

In addition, students can use the image as a background for the page.

Exercise 26: Table Dunston

Requirement:

Create a **Dunston.html** site, using a nested <table>. As described below:

Instructions:

Image file and attached data in folder Exercise 26.

http://teacher.uelstore.com/businesswebdevelopment/ex26.zip

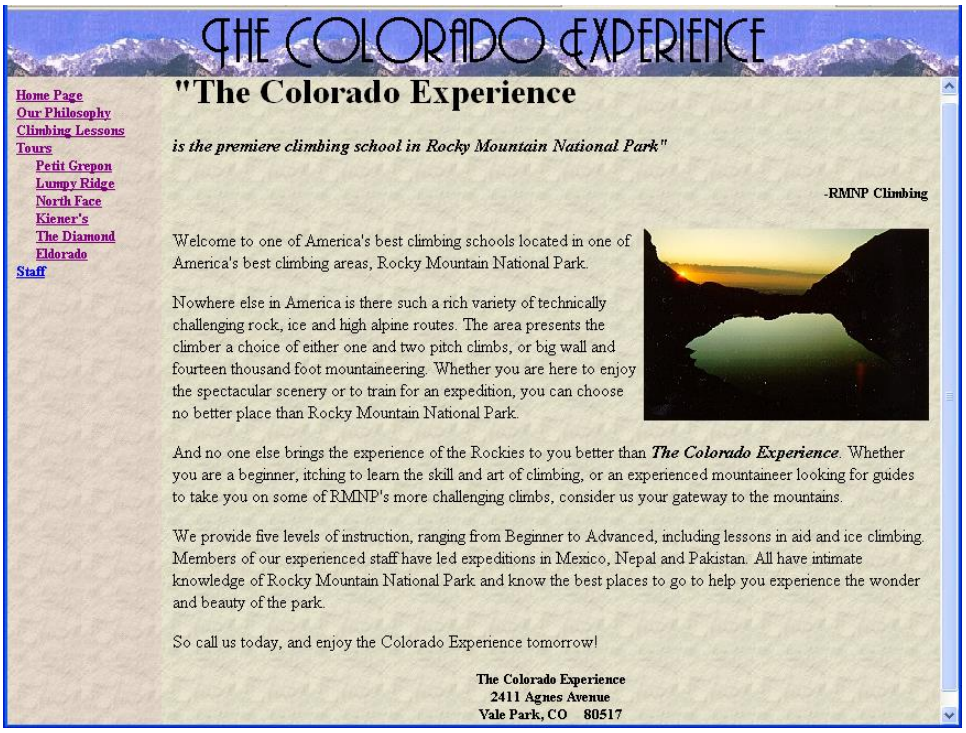
In addition, students can use the image as a background for the page.

Exercise 27: Frame

Requirement:

Create a **Colorado.html** website using a frameset of the form:

Logo.html	
Link.html	Hompage.html Our Philosophy.html Climbing Lessons.html Petit Grepon.html Lumpy Ridge.html North Face.html Kiener's.html The Diamond.html Eldorado.html



Instructions:

Image file and attached data in folder Exercise 27.

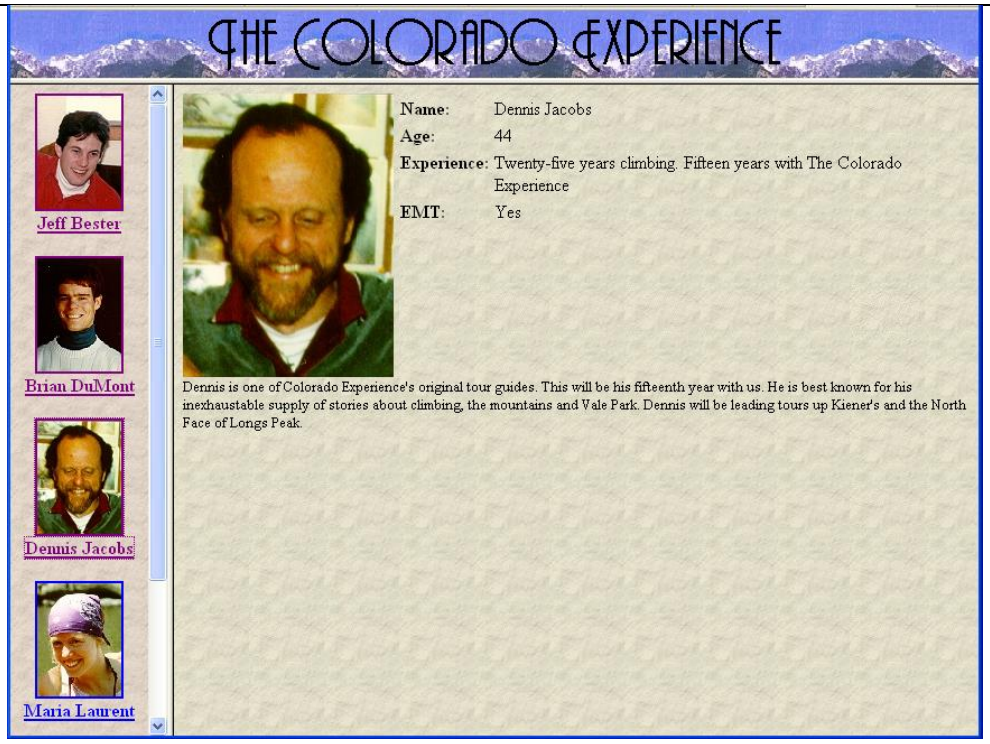
http://teacher.uelstore.com/businesswebdevelopment/ex27.zip

In addition, students can use the image as a background for the page.

Exercise 28: Frameset

Requirement:

Design a website using a frameset with the interface described below:



Instructions:

Image file and attached data in folder Exercise 28.

http://teacher.uelstore.com/businesswebdevelopment/ex28.zip

In addition, students can use the image as a background for the page.

Exercise 29: Frameset – Image maps

Requirement:

Website design using Frameset and image maps, Frameset has the form

Logo	4 pages of links
Image Maps	

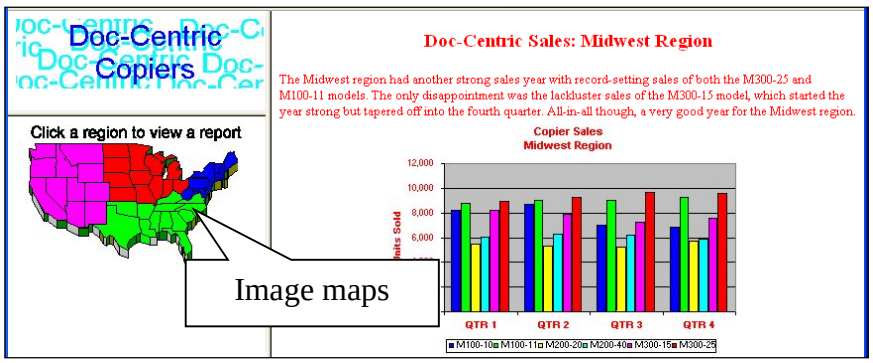


Image maps include 4 areas with 4 colors, when clicking on each area will link to a page with the corresponding color in the image maps.

Instructions:

Image file and attached data in folder Exercise 29.

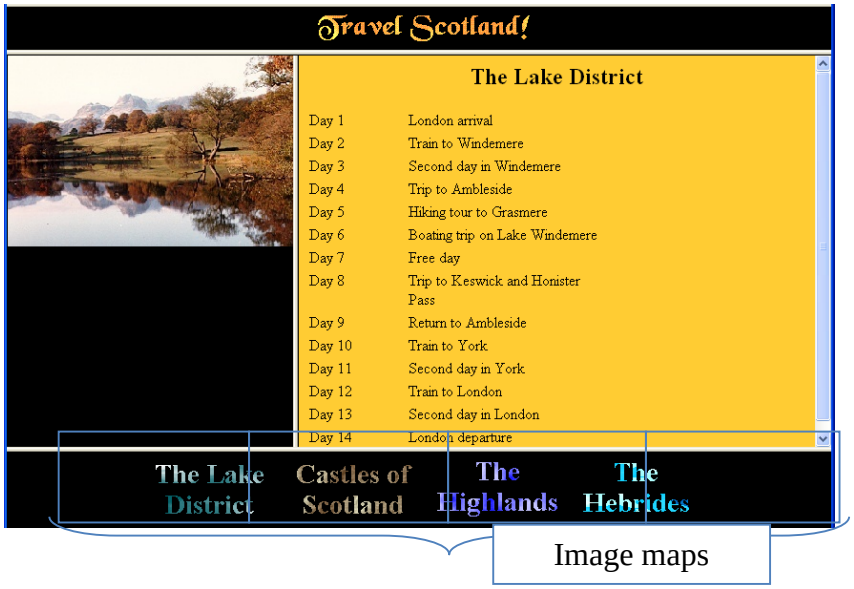
http://teacher.uelstore.com/businesswebdevelopment/ex29.zip

In addition, students can use the image as a background for the page.

Exercise 30: Frameset – Image maps

Requirement:

Design **Travel Scotland.html** website, using Frameset and image maps. Interface as described below:



Instructions:

Image file and attached data in folder Exercise 30.

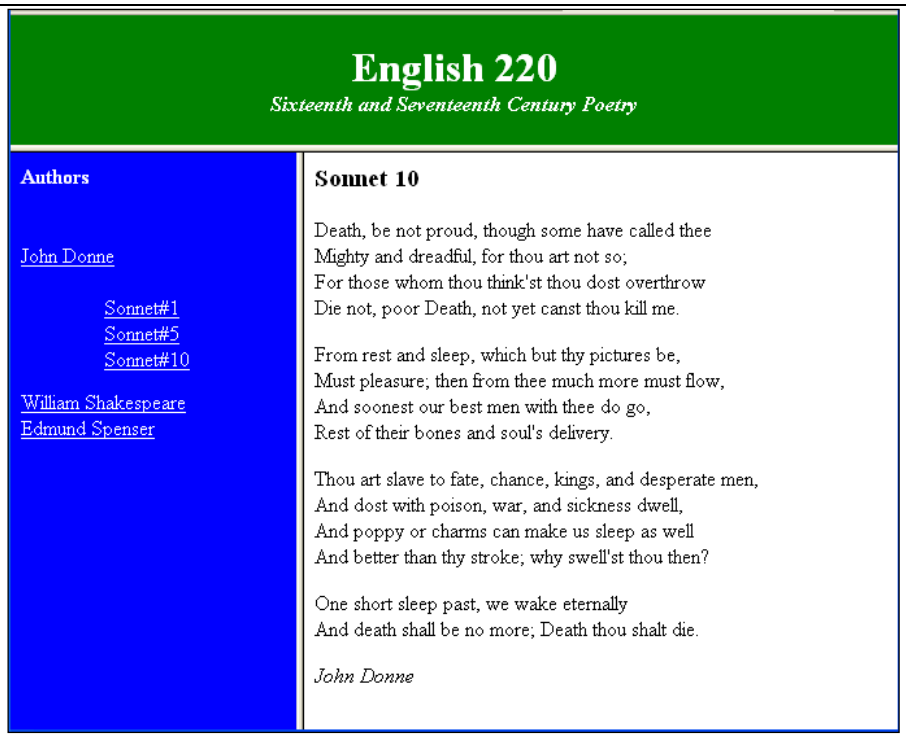
http://teacher.uelstore.com/businesswebdevelopment/ex30.zip

In addition, students can use the image as a background for the page.

Exercise 31: Frameset (*)

Requirement:

Design Sonnet website using Frameset. Graphic User Interface as shown below:



Instructions:

Image file and attached data in folder Exercise 31.

http://teacher.uelstore.com/businesswebdevelopment/ex31.zip

In addition, students can use the image as a background for the page.

Exercise 32:Marquee, Multimedia

Requirement:

Use the Marquee to let the words "Faculty of Information Systems" run left and right on the Web screen. Insert a **song** as **background** music for this page.

Instructions:

Watch the Lesson Slide about Marquee and Multimedia.

Module 3: Cascading Style Sheets (CSS)

Practical knowledge content:

+ Understand the Concept of DHTML

+ Understand what CSS means, how it works, CSS types and rules for creating CSS in HTML

+ Know the details of commonly used elements in CSS

+ Create box model

+ Understand the concepts of Margin, Padding

+ How to use Display, Position, Floating

Exercise 33: Internal Style

Requirement:

Use Internal Style to design the following Website:

Instructions:

Image file and attached data in folder Exercise 33

http://teacher.uelstore.com/businesswebdevelopment/ex33.zip

```
<style type = "text/css">
        body{
             background-image: url('hinh/camel.gif');
            background-position:  right bottom;
            background-repeat: no-repeat;
            background-attachment: fixed;
        }
        p{
             font-size: 12pt;
            color: #aa5588;
            text-align: justify;
            font-family: arial, sans-serif;
          }
        strong{
                    color:#0000FF;
                    font-family: verdana;
          }
       </style>
```

Exercise 34: Inline Style

Requirement:

Use Inline Style to design the Website below:

Instructions:

Image file and attached data in folder Exercise 34.

http://teacher.uelstore.com/businesswebdevelopment/ex34.zip

```
<h1>Inline Styles</h1>
<p><strong>Style Properties:</strong> text, font, background, margin, padding</p>
<hr />
<h1 style="color: #AAAAAA">Vietnam Information</h1>
<p style="font-family: verdana; font-size=14pt;">
<b>Geography</b><br />
```

```
<p style="text-align: left; font-size: 14pt; background-color: #CCCCCC; padding-left: 1cm">
 <b style="color: #0000FF">Topography</b><br />
```

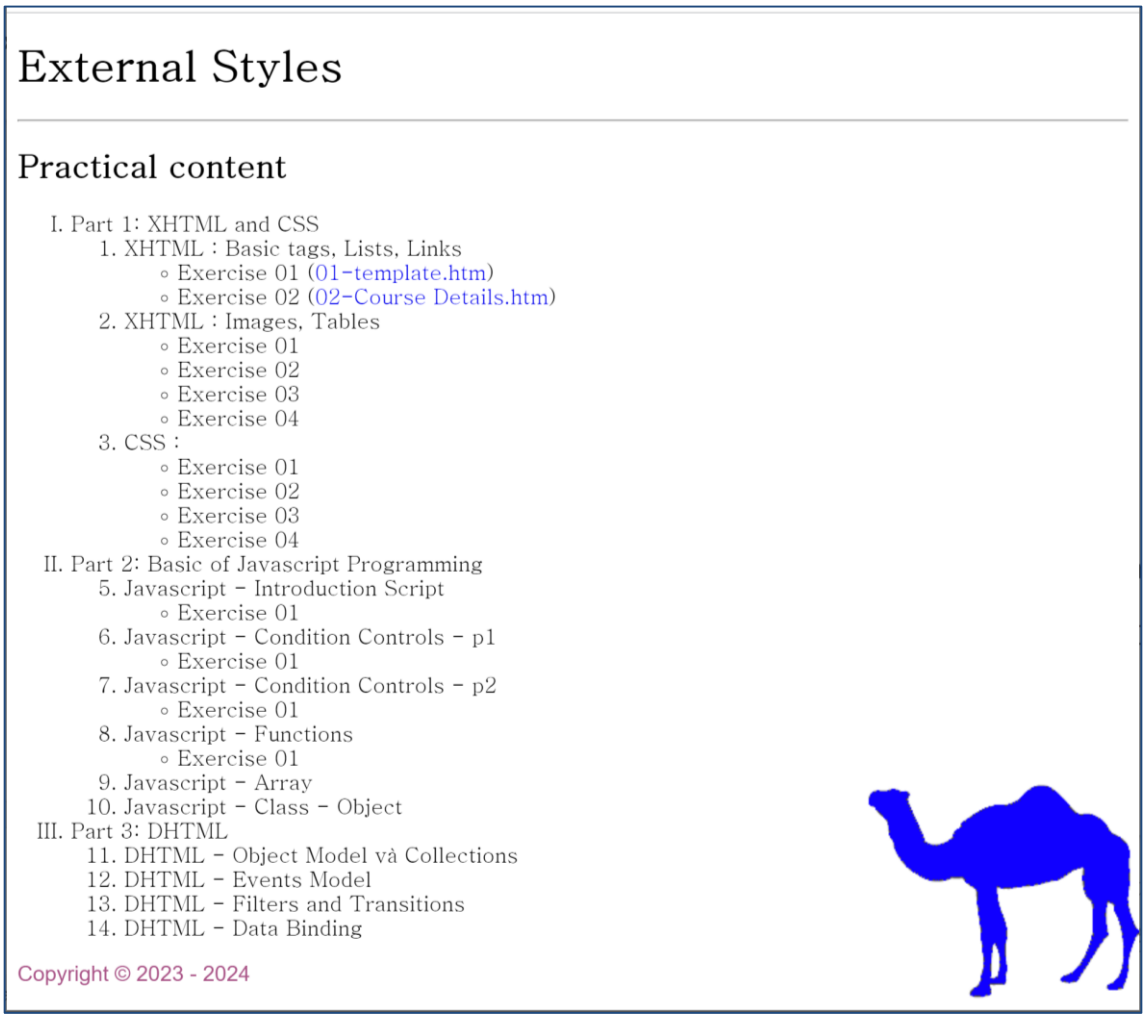
```
<p style="margin-left: 2cm; background-image:url('hinh/camel.gif')">
<b style="color: #FF00FF">People</b><br />
```

```
<p style="margin-right:1cm; margin-left:1cm; text-align: justify; background-color:
#CCCCCC">
<b style="color: #00FFFF; text-decoration: underline">Religion</b><br />
```

Exercise 35: External Style

Requirement:

Use External Style to design the following Website:



Instructions:

Image file and attached data in folder Exercise 35.

http://teacher.uelstore.com/businesswebdevelopment/ex35.zip

Create a file named **mainstyle.css**, with the following code:

```
body  { background-image: url('hinh/camel.gif');
            background-position:  right bottom;
            background-repeat: no-repeat;
            background-attachment: fixed;
            }
      p     { font-size: 12pt;
            color: #aa5588;
            text-align: justify;
            font-family: arial, sans-serif;
            }
        ol {list-style-type: upper-roman;
            }
        ol ol{list-style-type: decimal;
             }
        ol ol ul{
                  list-style-type: circle;
                  }
        .list{
                  font-family:verdana;
        }
        a{
                  text-decoration: none
        }
        a:hover {
            text-decoration: underline;
            color: red;
            background-color: #ccffcc
        }
```

Exercise 36: Box Menu

Requirement:

Design the **part1.html** page containing the menu box as shown below, using Internal CSS:

Instructions:

Image file and attached data in folder Exercise 36

http://teacher.uelstore.com/businesswebdevelopment/ex36.zip

Create a file named mainstyle.css, with the following code:

```
h3{
height:50px;
width:100%;
background:url("hinh/h3.png") no-repeat 0 100%;
margin : 0;
padding : 0;
}
ul
{
margin : 0;
padding : 10px 10px 25px 45px;
background:url("hinh/ul.png") no-repeat 0 100%;
list-style-image:url("hinh/li.png");
}
/* unvisited link */
a:link {
    color: red;
}
/* visited link */
a:visited {
    color: green;
}
/* mouse over link */
a:hover {
    color: hotpink;
          padding-left:10px;
}
/* selected link */
a:active {
    color: blue;
}
```

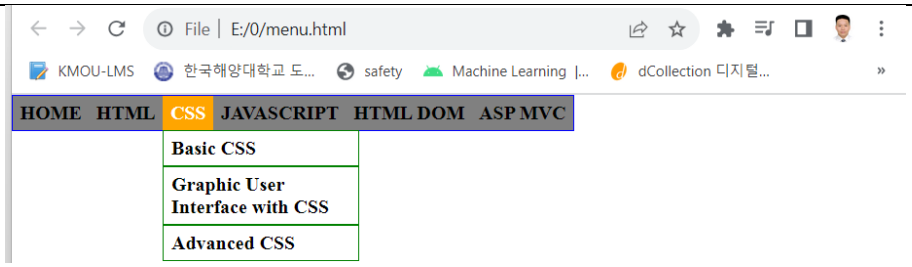
Exercise 37: Menu for Website

Requirement:

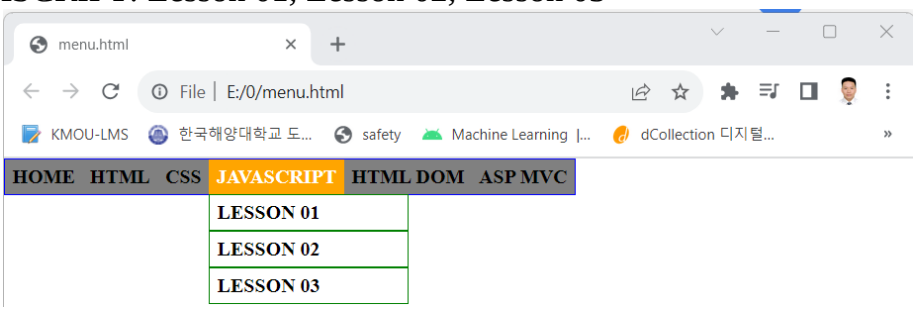
Design the **home.html** page, using external CSS as shown below:

Main menu: Home, HTML, CSS, JAVASCRIPT, HTML DOM, ASP

CSS Menu: Basic CSS, Layout with CSS, Advanced CSS



Menu JAVASCRIPT: Lesson 01, Lesson 02, Lesson 03



Instructions:

Image file and attached data in folder Exercise 37.

http://teacher.uelstore.com/businesswebdevelopment/ex37.zip

Create a file named **mainstyle.css**, with the following code:

```
*{
  margin:0;
  padding:0;
}
#container
{ border:1px solid blue;
background:gray;
float:left;
top: 60px;
text-indent:0px;
}
#container ul{
float:left;
list-style:none
}
#container ul li{
position:relative;
float:left;
}
#container ul li a
{
display:block;
color:black;
font-weight:bold;
text-decoration:none;
padding:.3em 6px;
}
#container li a:hover
{color:white;
```

```
background:orange;
}
#container li ul
{position:absolute;
display:none;
width:10em
}
#container li:hover ul {
   display:block;
}
#container ul li ul li{
  width:100%;
  border:1px solid green
}
```

Exercise 38: Four Seasons

Requirement:

Design **four-seasons.html** page, using Internal CSS as shown below.

SPRING
In the days of December, when heaven and earth are bustling in the love song of the season, we often walk together through the crowded streets of Tet holiday. The street these days is not as shabby as when the new winter comes, the street is also fresh and colorful in bright colors to wait for spring to come.

SUMMER
Summer rain comes and goes suddenly bringing cool, clear air to erase the burning summer sun. Memories flood back. The first rain of the season brings bountiful crops, but sometimes takes away the rice fields of the austere farmers.

AUTUMN
When autumn comes, the gentle warmth of autumn will gradually replace the sweltering summer sun. The coming of autumn is not only a transformation of nature, but it also brings with it many changes of the human soul, a new look, a new emotion and possibly a new beginning.

WINTER
Winter is the season suitable for late love. Season of mistakes and worries. The season when myrtle stands hard in the cold rain but cannot bloom. It is also the season to cherish the buds. Spring begins with the cold to numb from the end of winter.

Instructions:

Create a Tag Style to format some attributes for the p tag

width: 45%;
margin-top:...;
margin-left:...;
float:left;

Enter data for paragraph 1 normally at the end of the paragraph press Enter to see the result. Similarly enter for the next paragraph.

Create a formatting class for the title

font-size: ...;
font-weight: ...;
color: #00F;

Create a unique background color format ID for Four seasons

background-color:...;
color:...;

Image file and attached data in folder Exercise 38.

http://teacher.uelstore.com/businesswebdevelopment/ex38.zip

Exercise 39: Table with external CSS

Requirement:

Design the **fantastable.html** page as shown below, using external CSS:

Company	Contact	Country
Alfreds Futterkiste	Maria Anders	Germany
Berglunds snabbköp	Christina Berglund	Sweden
Centro comercial Moctezuma	Francisco Chang	Mexico
Ernst Handel	Roland Mendel	Austria
Island Trading	Helen Bennett	UK
Königlich Essen	Philip Cramer	Germany
Laughing Bacchus Winecellars	Yoshi Tannamuri	Canada
Magazzini Alimentari Riuniti	Giovanni Rovelli	Italy
North/South	Simon Crowther	UK
Paris spécialités	Marie Bertrand	France

Instructions:

```
#customers{
font-family:....
width:100%;
border-collapse:collapse;
}
#customers td, #customers th
{
font-size:...
border:1px solid #98bf21;
padding:3px 7px 2px 7px;
}
 #customers th
{
font-size:1.1em;
text-align:left;
padding-top:5px;
padding-bottom:4px;
background-color:#A7C942;
color:#ffffff;
}
#customers tr.alt td
{
Color:...
background-color:...
}
```

Exercise 40: Table with external CSS

Requirement:

Design the **tablecss.html** page as shown below, using external CSS:

Instructions:

padding:...

float:...

margin-bottom:...

margin-left:...

border: ...

Exercise 41: Vietnam Airline Website – external CSS

Requirement:

Design **vnairlines.html** page as shown below, Using External CSS:

Instructions:

Image file and attached data in folder Exercise 41.

http://teacher.uelstore.com/businesswebdevelopment/ex41.zip

In addition, students can use the image as a background for the page.

Exercise 42: Maxwell Scientific

Requirement:

Use style sheet to design website with Layout similar to illustration:

Instructions:

– Data and images in folder Exercise 42.

http://teacher.uelstore.com/businesswebdevelopment/ex42.zip

– style sheet file:

```
BODY {
        color:green;
        background: white url(Draft.jpg) no-repeat fixed center center
        }
H1, H2, H3, H4, H5, H6 {
        font-family: Arial, Helvetica, sans-serif
        }
ADDRESS {
        font-size: 0.9em;
        font-style:normal;
        text-align:center;
        text-transform:uppercase
        }
BLOCKQUOTE{
        background-color:silver
        }
UL {list-style:circle url(Apple.jpg) outside}
UL B {
        color:rgb(155,0,0)
        }
A {
        font-size: 0.9em;
        color-green
        }
A:hover {
        color:red;
        text-transform:uppercase;
        font-weight:bold
        }
.Special {
        color: rgb(153,102,6);
        font-weight:bold
        }
DIV.Article {
        padding: 0.5em;
        border-style: solid;
        border-width: 2px;
        background-color: rgb(252,221,163);
        width: 250px;
        float:right
        }
```

Exercise 43: *StuffShop*

Requirement:

Use the style sheet to design the **StuffShop** website as shown below.

Instructions:

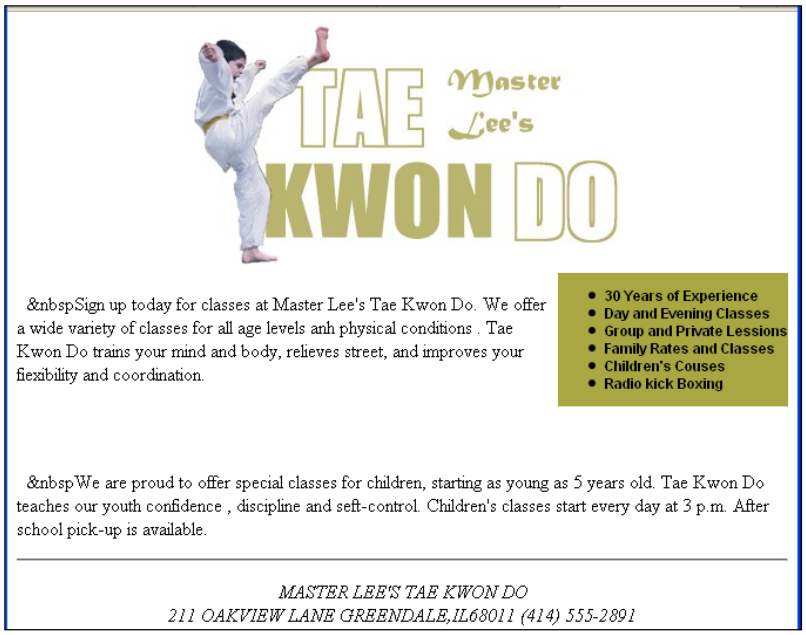
- Data and images in folder Exercise 43.

http://teacher.uelstore.com/businesswebdevelopment/ex43.zip

Exercise 44: M.Lee's

Requirement:

Use the style sheet to design **M.Lee's** website as shown below:



Instructions:

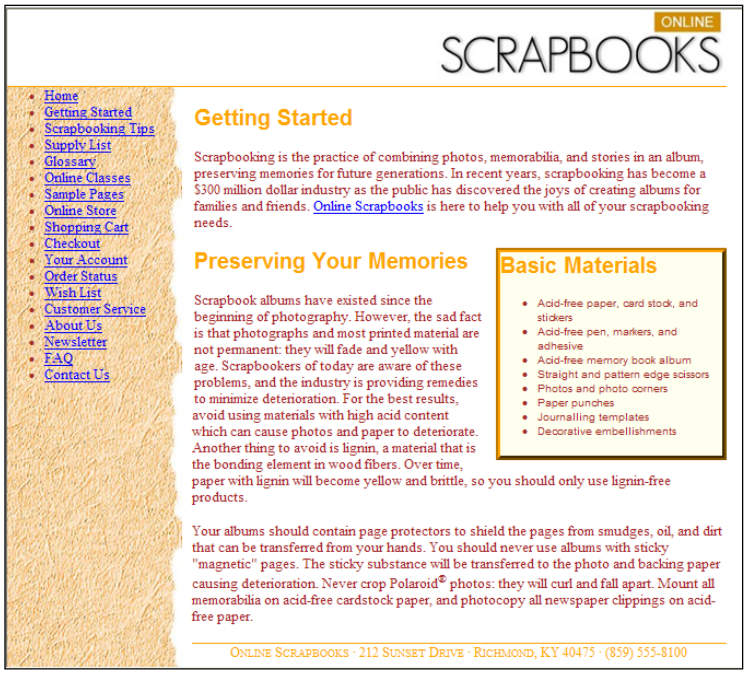
Image file and attached data in folder Exercise 44.

http://teacher.uelstore.com/businesswebdevelopment/ex44.zip

Exercise 45: ScrapBooks

Requirement:

Use the style sheet to design the **ScrapBooks** website as shown below:



Instructions:

Image file and attached data in folder Exercise 45.

http://teacher.uelstore.com/businesswebdevelopment/ex45.zip

Exercise 46: Mount Rainier News

Requirement:

Use a style sheet to design the website as shown below, students design it themselves:

Instructions:

Image file and attached data in folder Exercise 46.
http://teacher.uelstore.com/businesswebdevelopment/ex46.zip

Exercise 47: Table with Style-trigger

Requirement:

Use a style sheet to design the website as shown below, students design it themselves:

#	Student ID	Student Name
1	123456	Albert Einstein
2	321232	Niels Bohr
3	987544	Stephen Hawking
4	359687	Isaac Newton
5	574878	Nikola Tesla
Designed by John		

-requires separating the structure of the table into 3 distinct parts: thead, tbody, tfoot
-thead is the title row
-tbody is 5 rows of data
-tfoot is the last row (footer)
-CSS format as above.
-Also note that the odd line is colored whitesmoke, the even line is colored pink (and must be automatically colored, when a new line is added it must automatically recognize which is even and odd lines to fill the background). Move the mouse to any line, that line will have a yellow background and the mouse pointer will look like an index finger.

Instructions:

Table structure Students handle themselves, style structure (use internal or external style to do).

```
<style type="text/css">
        thead,tfoot
        {
            background-color:blue;
            color:white;
        }
        tbody tr:nth-child(2n+1)
        {
            background-color:whitesmoke
        }
        tbody tr:nth-child(2n) {
            background-color: pink
        }
        tbody tr:hover
        {
            background-color:yellow;
            cursor:pointer
        }
</style>
```

Module 4: Form and Controls

Practical knowledge content:

+ Understand the meaning and function of Form

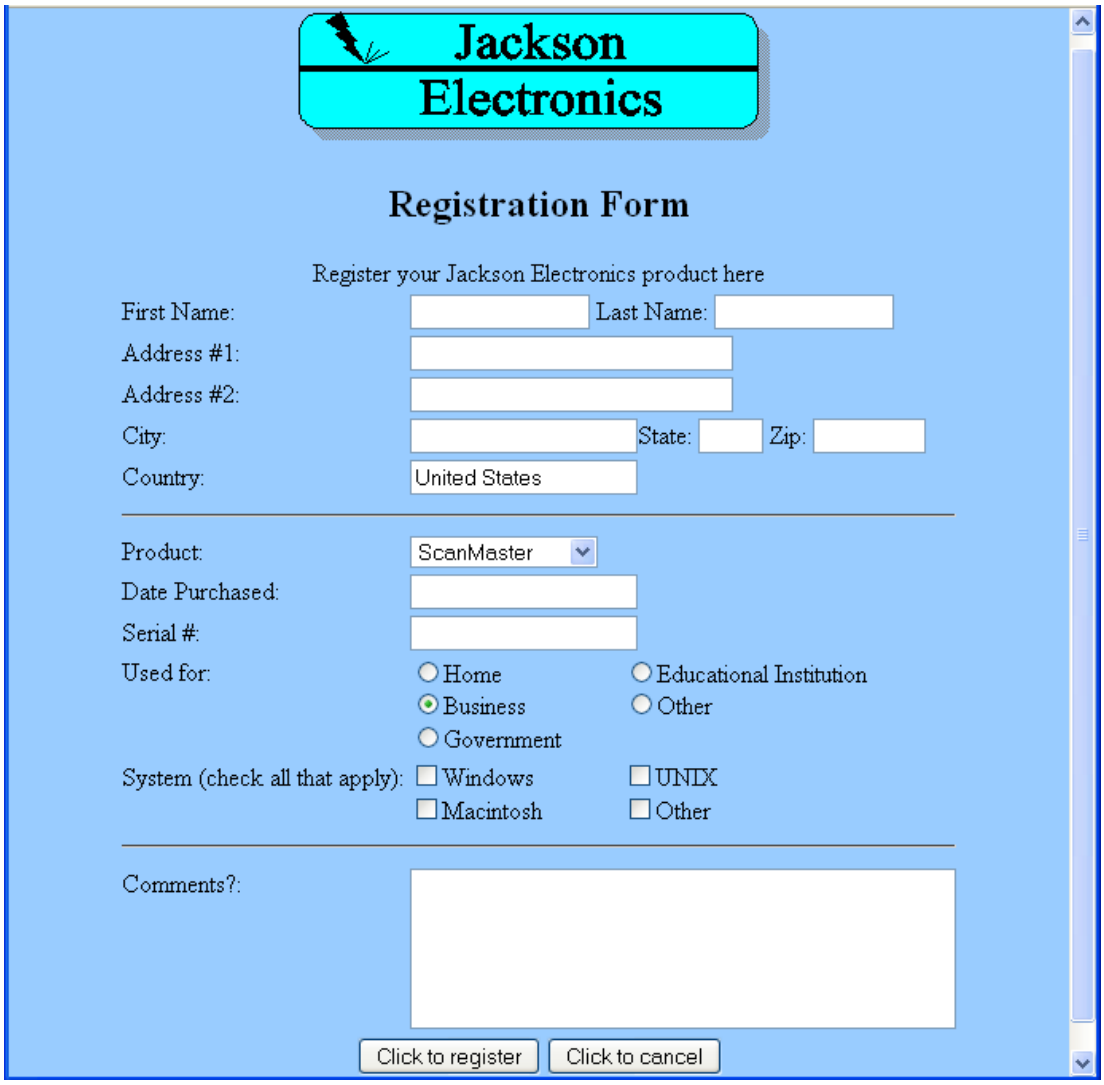
+ Distinguish and practice GET/POST/PUT/DELETE methods

+ Understand and use controls on Form: Label, Textbox, Button, Combo, ListBox, TextArea ….

Exercise 48: Register

Requirement:

Create **Register.html** page, Graphic User Interface as described below:



Click to register➔alert for Registration
Click to cancel➔Reset data on Form

Instructions:

Image file and attached data in folder Exercise 48.

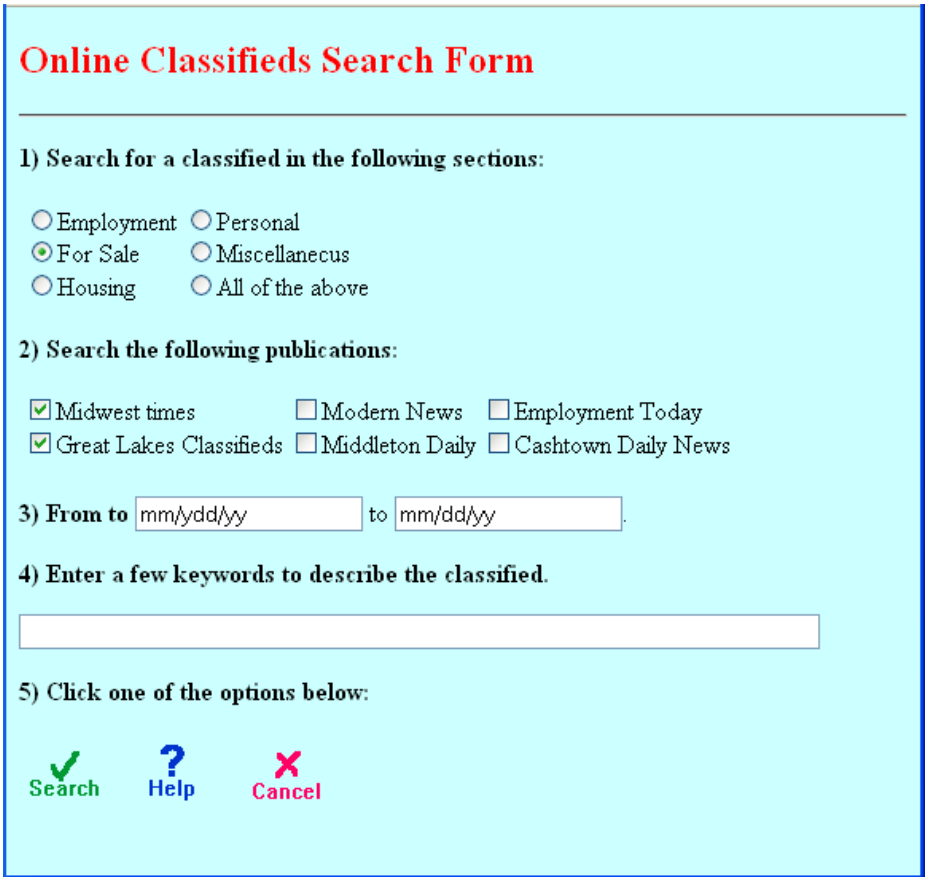
http://teacher.uelstore.com/businesswebdevelopment/ex48.zip

In addition, students can use the image as a background for the page.

Exercise 49: Online Classifield

Requirement:

Design the **Online Classifield** form page as required as below:



Simulation:

Search➔Search simulation

Help d➔isplays 1 help page(Student optional)

Cancel➔clear data on Form

Instructions:

Image file and attached data in folder Exercise 49.

http://teacher.uelstore.com/businesswebdevelopment/ex49.zip

In addition, students can use the image as a background for the page.

Exercise 50: Travel Expense Report

Requirement:

Design the form page as shown below:

Submit ➔Alert for submit

Reset➔Reset the data on Form

Instructions:

Image file and attached data in folder Exercise 50.

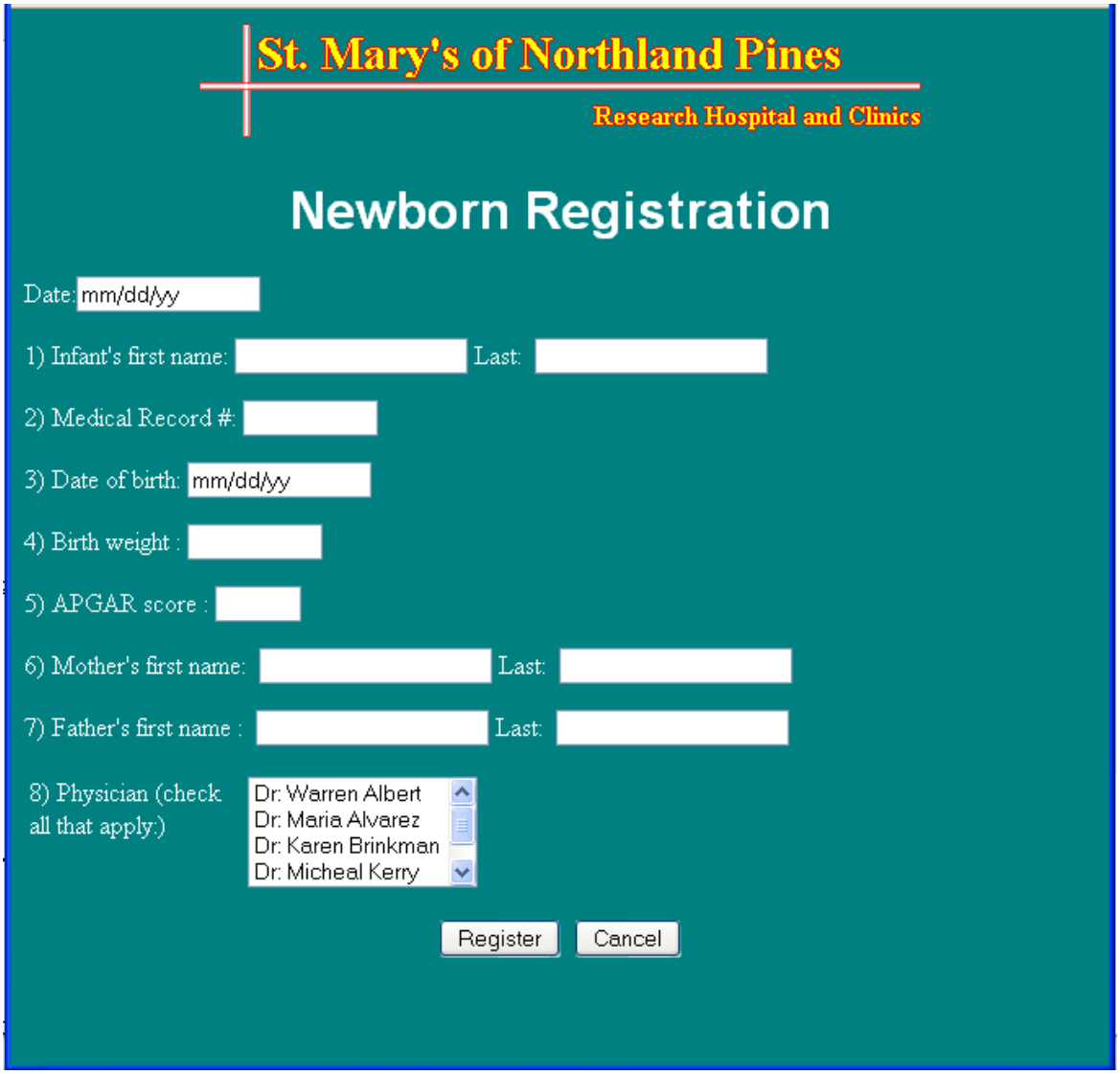
http://teacher.uelstore.com/businesswebdevelopment/ex50.zip

In addition, students can use the image as a background for the page.

Exercise 51: Registration

Requirement:

Design the **Registration** form page, Graphic User Interface as shown below:



Instructions:

Image file and attached data in Exercise 51.

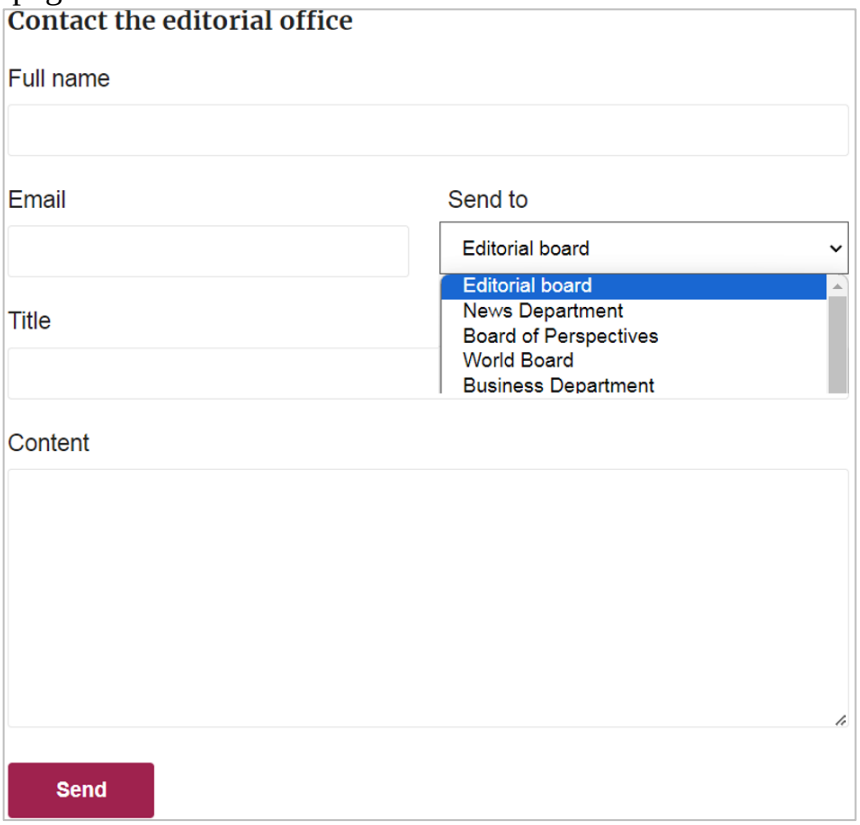
http://teacher.uelstore.com/businesswebdevelopment/ex51.zip

In addition, students can use the image as a background for the page.

Exercise 52: Contact - GET

Requirement:

Create contact page as shown below:



When clicking send message➔use GET method to transfer all data in Contact page to a new page (Student designed it by themselves)..

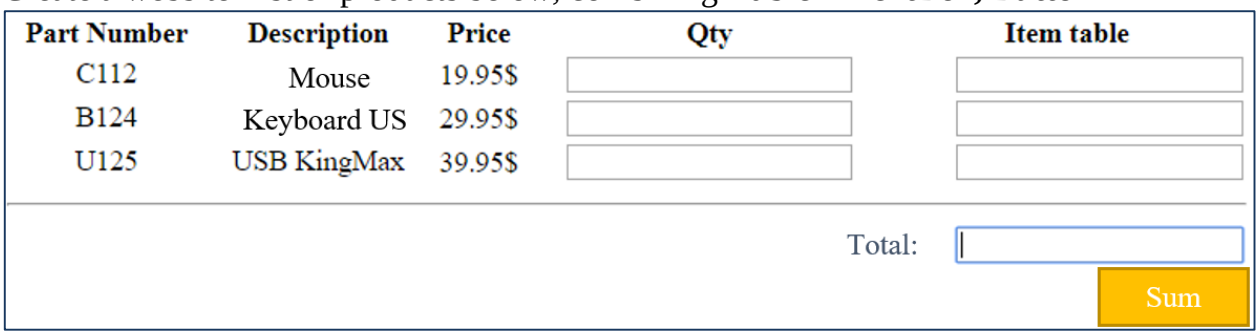
Instructions:

Refer to the lesson of getting data in theory class

Exercise 53: Product-GET

Requirement:

Create a website List of products below, combining **Table + TextBox, Button**



Use the GET Method to process the Sum button=>send all the data through a new HTML page (Students make up this HTML page by themselves)

Instructions:

Refer to the lesson of getting data in theory class

Module 5: Javascript

Content of practical knowledge:

+ Understanding JavaScript language in web design and programming

+ Know how to use the rules and syntax of the HTML language to support the interface design and handling design of the website

+ Know how to associate JavaScript language, JavaScript functions with HTML tags

+ Mastering the rules and techniques of string processing with Javascript

+ Build and use object classes in JavaScript

Exercise 54: Math function

Requirement:

Create a website with Table and Textbox, Button to calculate MAX, MIN, sin, cos… As described below:

a:	30
b:	45
c:	90
Result:	-0.9880316240928618

MAX | MIN | sin(a) | Cos(a) | a^b

Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

Exercise 55: The quadratic equation

Requirement:

Write a WebSite to solve quadratic equations as described below:

The quadratic equation	
a:	1
b:	3
c:	-4
Result:	x1=-4 ; x2=1
	Find solution Clear

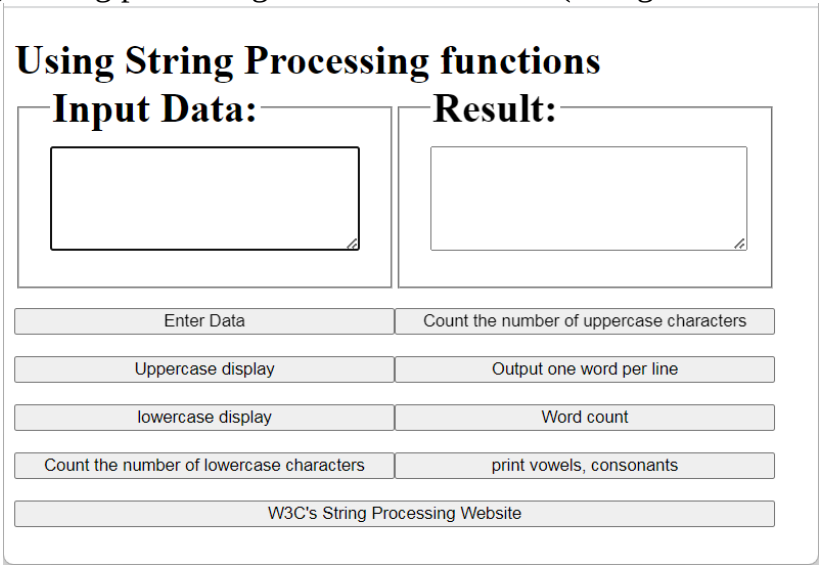
Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

Exercise 56: String processing

Requirement:

Create web page String processing As described below (Using tables and controls):



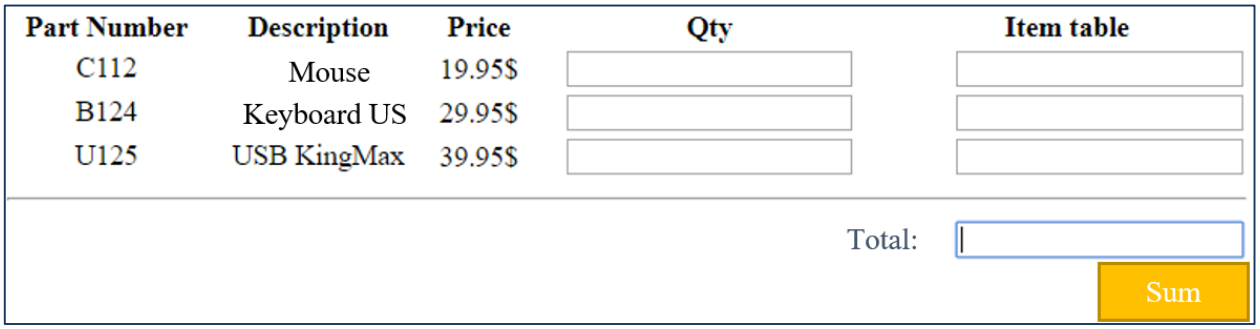
Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

Exercise 57: Handle the text change event

Requirement:

Create a website with Tables and controls as described below:



- The user enters the quantity in the Textboxes the quantity in each line➔The box into money will automatically calculate (Price * quantity). Data validity check required
- Click on "Sum"➔ display the total amount in the Total box

Instructions:

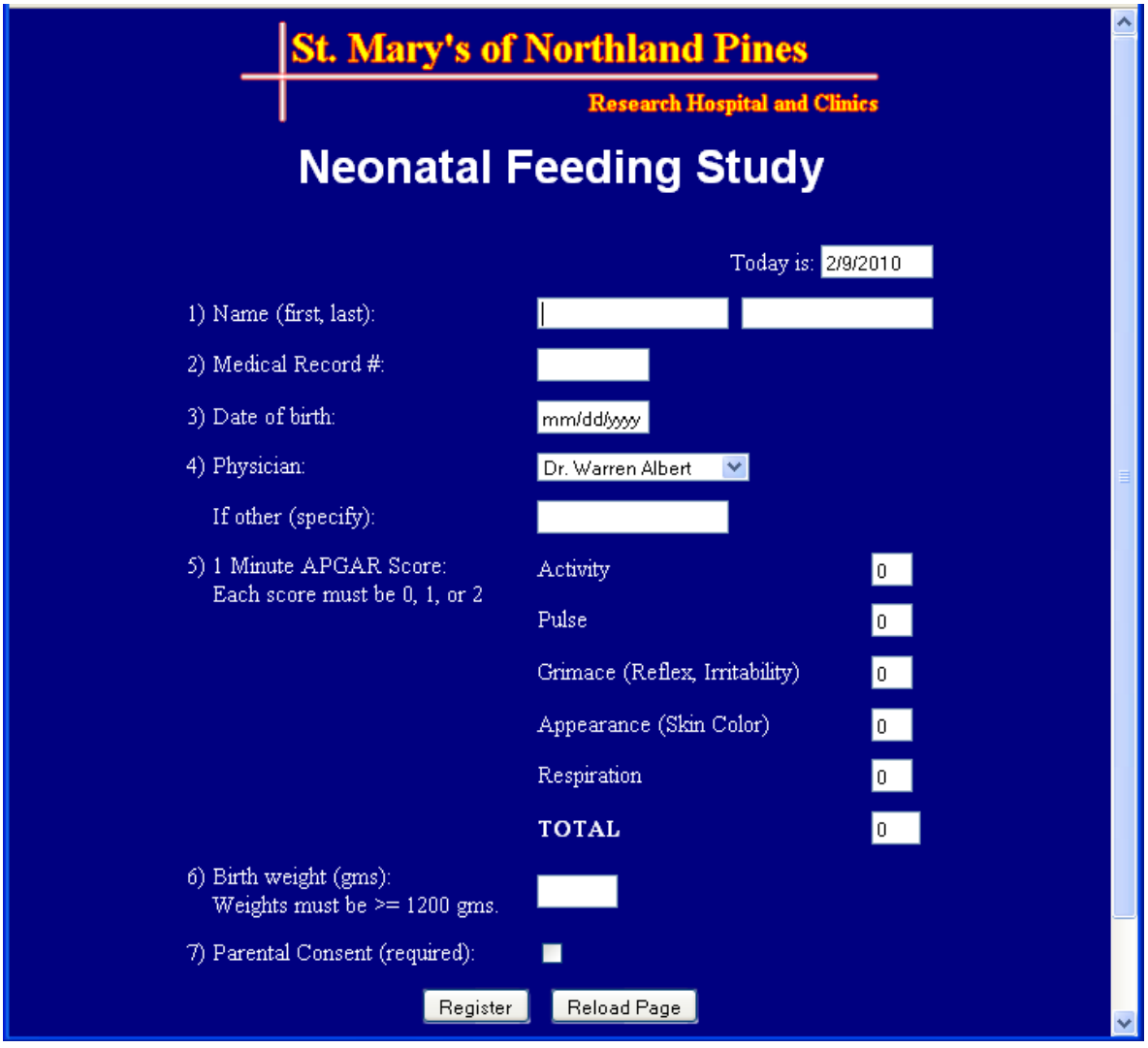
Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

Exercise 58: Neonatal Feeding Study

Requirement:

Design the form page as shown, with the following requirements:

- When loading the form, Today is will display the current date in the format **d/m/yyyy** and the apostrophe jumps to the name.
- In the Physician section, if the user chooses other, when pointing out, a message will appear asking to enter Name.
- Enter the value in the Activity, Pulse, Grimace, Appearance and Respiration boxes, the program checks the input value from 0=>2, if it is wrong, the program displays a message to re-enter. When entering the value in each cell, the Total cell will calculate the total value of the entered cells.



Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

Image file and attached data in folder Exercise 58.

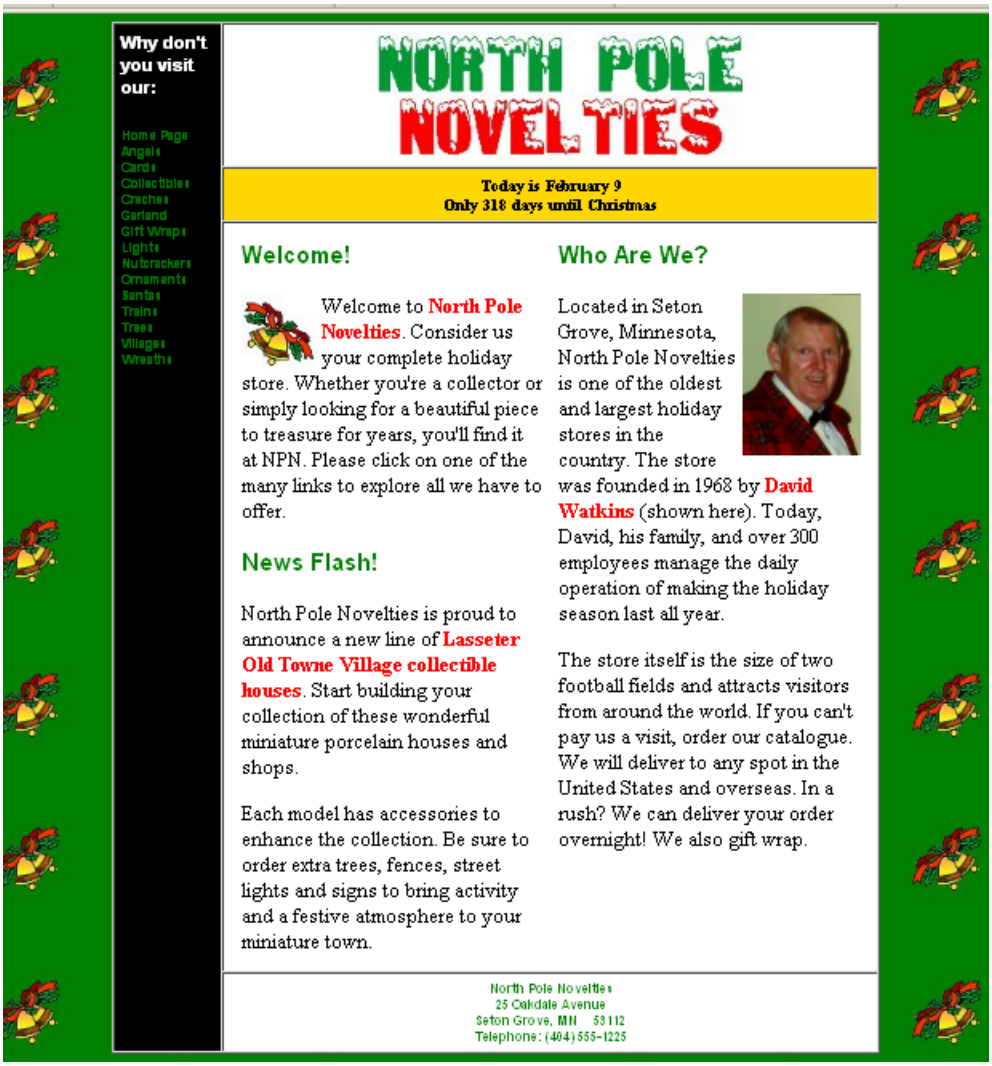
http://teacher.uelstore.com/businesswebdevelopment/ex58.zip

Exercise 59: North Pole Novel Ties

Requirement:

Design the **NPN.htm** website with the following requirements:

- Use the style sheet to format the headers, and the links do not have underlined, when hovering the mouse, the links turn red.
- Use Javascript to write a function to calculate the number of days remaining from the current day of the machine to Christmas (look at the yellow background).



Instructions:

Image file and attached data in folder Exercise 59.

http://teacher.uelstore.com/businesswebdevelopment/ex59.zip

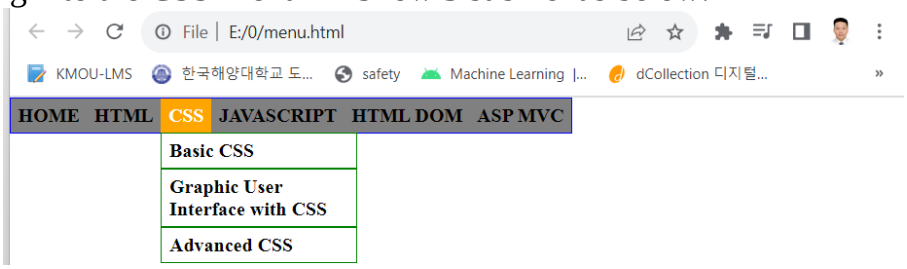
In addition, students can use the image as a background for the page.

Exercise 60: Menu

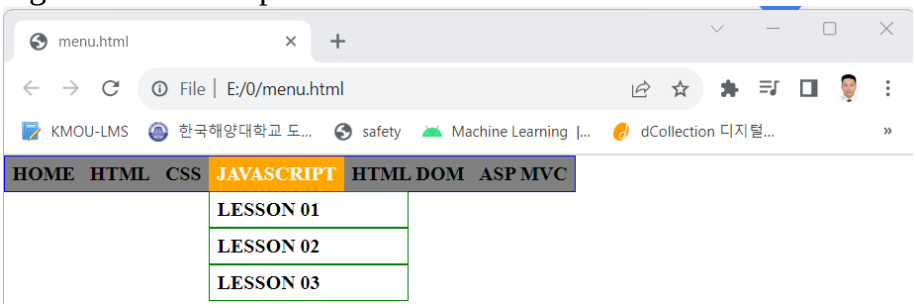
Requirement:

Design a website with Menu with below requirements, combining **CSS and Javascript**:

When moving into the CSS Menu ➔ Show 3 submenus below:



When scrolling to the JavaScript Menu ➔ Show 3 submenus below



Moving to any Menu item changes the background color to Orange as shown in the picture.

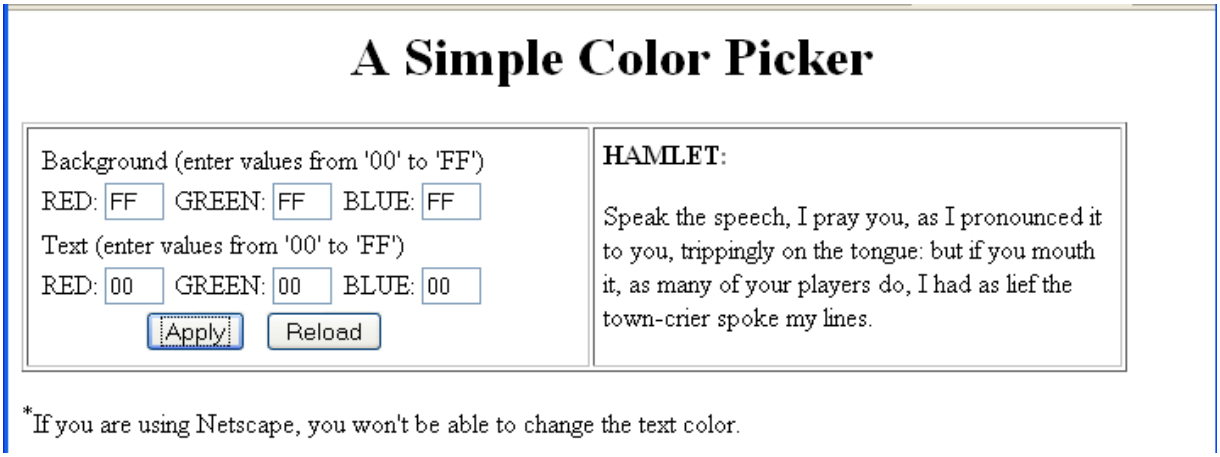
Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle. And refer to more on Google.

Exercise 61: Color Picker

Requirement:

Design the **Color Picker** page as shown below:



Clicking Apply will apply the color to the right content

Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle. And refer to more on Google.

Module 6: XML - DOM and JSon

Practical knowledge content:

+ Understanding HTML DOM in web design
+ Understand the preeminent features of HTML DOM combined with JavaScript
+ Applying the dynamics of HTML DOM in designing and building websites
+ Understand and analyze XML document structure
+ Understand how AJAX works in data retrieval
+ Know the JSON data structure
+ Declaring and exploiting JSon

Exercise 62: Table Mouse Trigger

Requirement:

Write a command to move the mouse on any row in the Table, then fill that row with yellow, move the mouse out, then fill it with green.

#	Student ID	Student Name
1	123456	Albert Einstein
2	321232	Niels Bohr
3	987544	Stephen Hawking
4	359687	Isaac Newton
5	574878	Nikola Tesla
Designed by John		

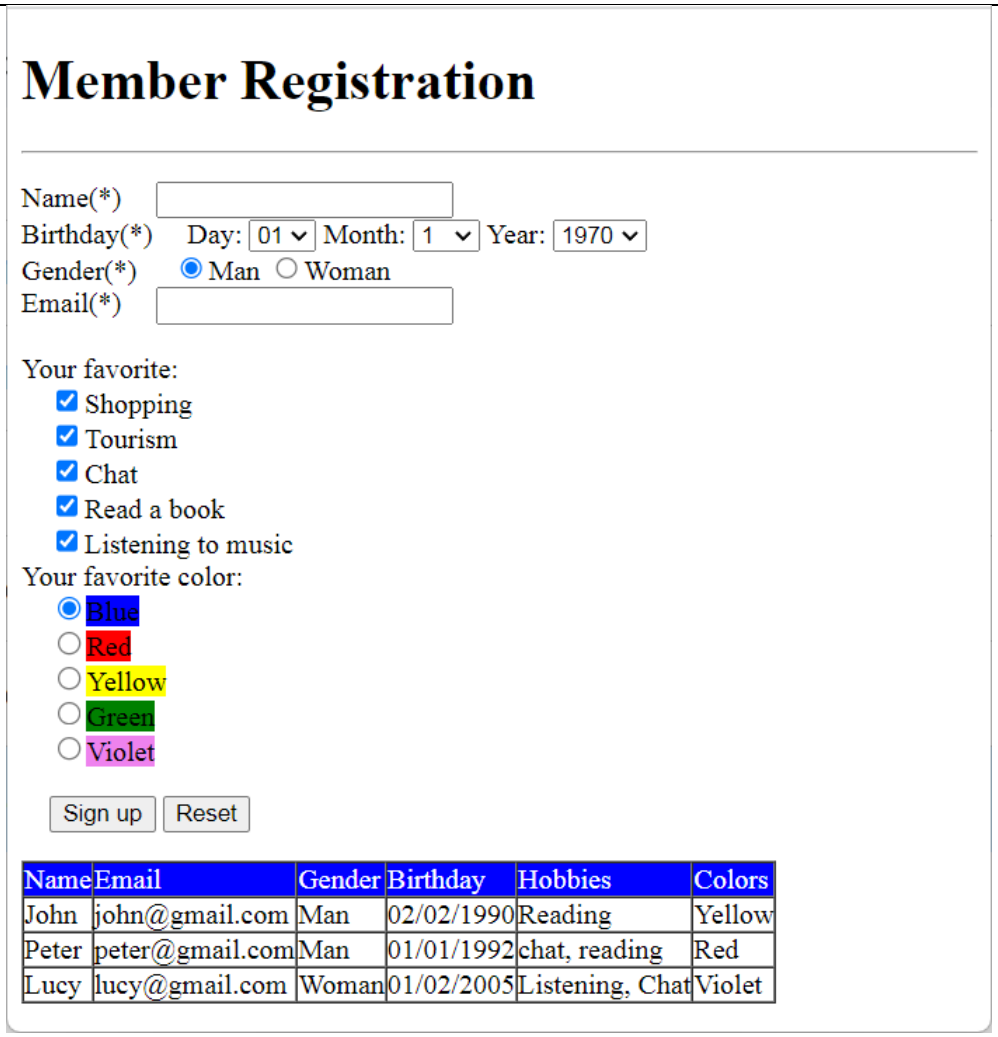
Instructions:

Using Javascript to handle TR, each TR will have 2 **Mouse Over** and **Mouse Leave** events, each function should write its own function to handle.

Exercise 63: Member Registration

Requirement:

Create a **Member Registration** website, using <table> and controls on Forms and objects in JavaScript as described below:



- **Date, month, year of birth:** Write javascript to automatically upload droplist(combo), use loops to load data for control.
- **When pressing register button:** Check Name cannot be left blank, Email must be valid. All information entered on the Form will be displayed in the Table below
- **Click Next button:** Delete the old data on the Form, focus on the Name box.
- Add **mouse over** and **mouse out** events for rows in Table. When moving the mouse in, the background is yellow, moving the mouse to the background is white.

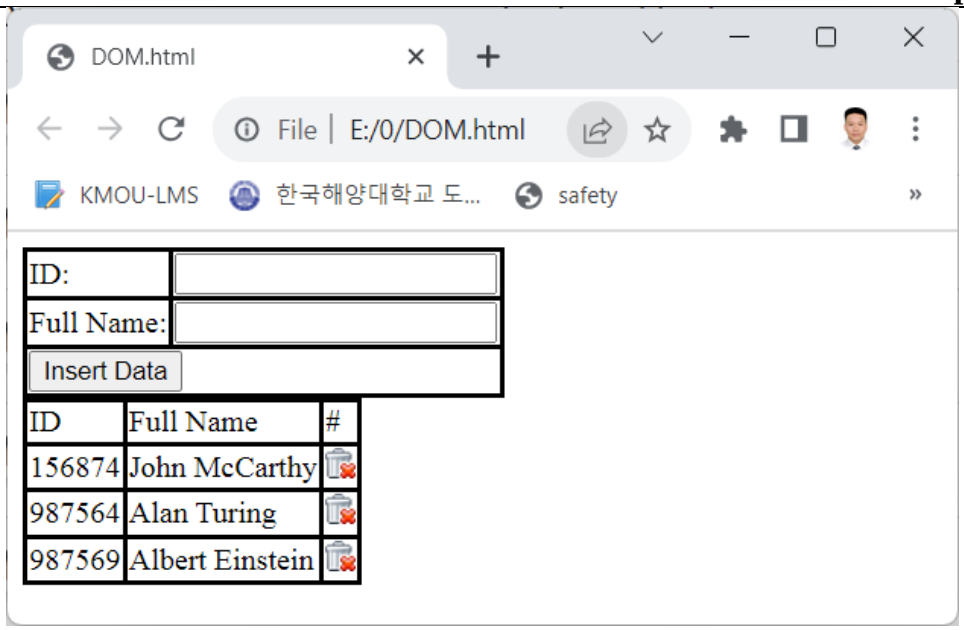
Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

Exercise 64: Dynamic Table

Requirement:

Design Interactive Table Website as shown below:



- Click "Insert Data"➔using HTML DOM to create new TR with ID and Full Name information and then insert into Table.
- Click "Remove Icon" ➔Remove TR from the Table.

Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle.

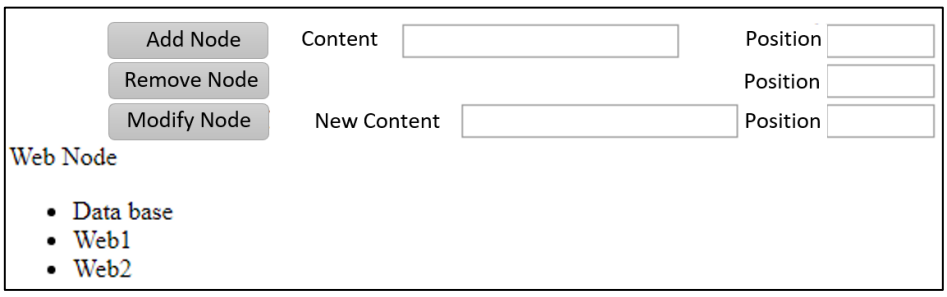
```
<html>
<head>
    <style type="text/css">
        table, td {
            border: solid;
            border-collapse: collapse
        }
        img {
            cursor: pointer
        }
    </style>
    <script type="text/javascript">
        function insertLast()
        {
            tbl = document.getElementById("tbl")

            tr = document.createElement("tr")
            td1 = document.createElement("td")
            td2 = document.createElement("td")
            td3 = document.createElement("td")
            id = document.getElementById("id").value
            name = document.getElementById("name").value
            td1.innerText = id
            td2.innerText = name
            td3.innerHTML = "<img src='deletetopic.png'
onclick='processDelete(this)'/>"
            tr.appendChild(td1)
            tr.appendChild(td2)
            tr.appendChild(td3)
            tbl.appendChild(tr)
        }
        function processDelete(img) {
            tbl = document.getElementById("tbl")
            vtDelete=img.parentNode.parentNode.rowIndex
```

```
            tbl.deleteRow(vtDelete)
        }
    </script>
</head>
<body>
    <table>
        <tr>
            <td>ID:</td>
            <td><input type="text" id="id" name="id" /></td>
        </tr>
        <tr>
            <td>Full Name:</td>
            <td><input type="text" id="name" name="name" /></td>
        </tr>
        <tr>
            <td colspan="2">
                <button type="button" onclick="insertLast()">Insert Data</button>
            </td>
        </tr>
    </table>
    <table id="tbl">
        <tr>
            <td>ID</td>
            <td>Full Name</td>
            <td>#</td>
        </tr>
        <tr>
            <td>156874</td>
            <td>John McCarthy</td>
            <td>
                <img src="deletetopic.png" onclick="processDelete(this)" />
            </td>
        </tr>
        <tr id="idtr2">
            <td>987564</td>
            <td>Alan Turing</td>
            <td><img src="deletetopic.png" onclick="processDelete(this)" /></td>
        </tr>
        <tr>
            <td>987569</td>
            <td>Albert Einstein</td>
            <td><img src="deletetopic.png" onclick="processDelete(this)" /></td>
        </tr>
    </table>
</body>
</html>
```

Exercise 65: Dynamic List

Requirement:

Design Web Node with content as illustrated:



- **Add Node**: Add a Node with content to the position in the Web Node (the div contains the li)
- **Delete Node**: Delete Node at any position in li
- **Modify Node**: Replace old content in place with new content

Instructions:

Using Javascript to handle buttons in Web pages, each function writes its own function to handle. Use insertBefore, appendChild, removeChild, replaceChild...

Exercise 66: XML-Dom Parser

Requirement:

In the HTML file there is an XML string with the following content:

```
<student>
        <id>987456</id>
        <name>Marie Curie </name>
        <birthday>7/11/1867</ birthday >
        <gender>Woman</ gender >
</student>
```

Use DOMParser to read Student ID, Full Name, Date of Birth, Gender and display it on Table:

Student Information	
Student ID:	987456
Student Name:	Marie Curie
Birthday:	7/11/1867
Gender:	Woman

- Add mouse movement event to the row: Blue background, white text
- Add event to move mouse out of row: white background, black text

Instructions:

Read the DOMParser lecture in slides

Exercise 67: XML – DOM Parser (*)

Requirement:

In the HTML file there is an XML string with the following content:

```
<students>
    <student>
        <id>987456</id>
        <name> Marie Curie </name>
        <birthday>7/11/1867</birthday>
        <gender> Woman </gender>
     </student>
    <student>
        <id>987561</id>
        <name> Einstein </name>
        <birthday>14/3/1879</birthday>
        <gender>Man</gender>
     </student>
    <student>
        <id>985467</id>
        <name> Leó Szilárd </name>
        <birthday>11/2/1898</birthday>
        <gender> Man </gender>
     </student>
  …..
</students>
```

Use DOMParser to read Student ID, Full Name, Date of Birth, Gender and display it on Table:

List of Students			
Student ID	**Student Name**	**Birthday**	**Gender**
987456	Marie Curie	7/11/1867	Woman
987561	Einstein	14/3/1879	Man
985467	Leó Szilárd	11/2/1898	Man

- Add event click on column header: sort ascending or descending
- Add mouse movement event for the row: yellow background
- Add event to move mouse out of row: white background
- Add event click on the row➔ opens a detail page, for example, clicking on the row you have "Marie Curie" will display the detail page as follows:

Student Information	
Student ID:	987456
Student Name:	Marie Curie
Birthday:	7/11/1867
Gender:	Woman

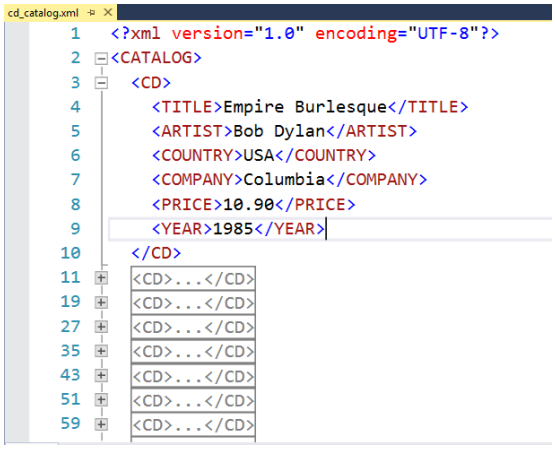
Instructions:

See how to use **DOMParser** in lecture slides

Exercise 68: XML - AJAX

Requirement:

Given an XML data file with content:



Download files here:

http://teacher.uelstore.com/businesswebdevelopment/ex68.zip

Use AJAX to read the **cd_catalog.xml** file displayed on the Table as shown:

Get my CD collection

Artist	Title
Bob Dylan	Empire Burlesque
Bonnie Tyler	Hide your heart
Dolly Parton	Greatest Hits
Gary Moore	Still got the blues
Eros Ramazzotti	Eros
Bee Gees	One night only
Dr.Hook	Sylvias Mother
Rod Stewart	Maggie May
Andrea Bocelli	Romanza
Percy Sledge	When a man loves a woman
Savage Rose	Black angel
Many	1999 Grammy Nominees
Kenny Rogers	For the good times

Instructions:

See instructions on how to use AJAX in lecture slides.

Exercise 69: XML – AJAX (*)

Requirement:

For XML File with structure and data as below:

XMLFile.xml

```xml
<?xml version="1.0" encoding="utf-8" ?>
<employees>
    <employee id="1" title="Architect">
        <name>John</name>
        <phone>0987456689</phone>
    </employee>
    <employee id="2" title="Engineer">
        <name>Peter</name>
        <phone>0948762123</phone>
    </employee>
    <employee id="3" title="Teacher">
        <name>Cayse</name>
        <phone>0916563214</phone>
    </employee>
    <employee id="4" title="Architect">
        <name>Tom</name>
        <phone>0974123214</phone>
    </employee>
    <employee id="5" title="Engineer">
        <name>Daisy</name>
        <phone>0967854123</phone>
    </employee>
</employees>
```

Use AJAX to read data:

- The title attribute of the Employee=>groups do not overlap and display on Droplist
- Select data in Droplist => Display Employee List by Title in Droplist (use Table to display):

Title:	Architect ▾	
Employee ID	**Employee Name**	**Phone**
1	John	0987456689
4	Tom	0974123214

Instructions:

See instructions on how to use AJAX in lecture slides, google search.

Exercise 70: JSonObject vs JSonArray

Requirement:

Present the structure of JsonObject, JsonArray. Compare the difference between XML and JSON.

Instructions:

+ Read theory in Slide and learn more on Google

Exercise 71: Display User Information

Requirement:

Given a user object in the format Json Object as below (data in HTML file):

```
<script>
    var user = {
       "id": "100005823642721",
       "first_name": "Duy Thanh",
       "gender": "male",
       "last_name": "Trần",
       "link": "https://www.facebook.com/duythanhcse",
       "locale": "en_US",
       "birthday": "20/12/1961",
       "name": "Duy Thanh Trần",
       "username": "duythanhcse"
    }
  </script>
```

Display User's details on Table.

Instructions:

+ See instructions in the lesson Slide about JSon

Exercise 72: List of products

Requirement:

Given a JsonArray with the following structure and data (data in HTML file):

```
"Products": [
            {
               "ProductID": "p123",
               "ProductName": "DELL Inspiron 14NR",
               "Quantity": 100,
               "UnitPrice": 150000
            },
            {
               "ProductID": "p456",
               "ProductName": "HP Inspiron 113",
               "Quantity": 130,
               "UnitPrice": 140000
            },
            …
                ]
```

Let's display a list of Products on Table:

List of Products			
Product ID	**Product Name**	**Quantity**	**UnitPricec**
p123	DELL Inspiron 14NR	100	150000
p456	HP Inspiron 113	130	140000

Exercise 73: List of products

Requirement:

Use JsonArray to display the following data on the Table (data in the HTML file):

```
var products = [
        {
           code: "sp1",
           name: "Cocacola",
           price: 15000
        },
        {
           code: "sp2",
           name: "Pepsi",
           price: 25000
        },
        {
           code: "sp3",
           name: "Sting",
           price: 17000
        },
    ]
```

Product Code	Product Name	UnitPricec
sp1	Cocacola	15000
sp2	Pepsi	25000
sp3	Sting	17000

Instructions:

+ See the instructions in the Lesson Slide about Json

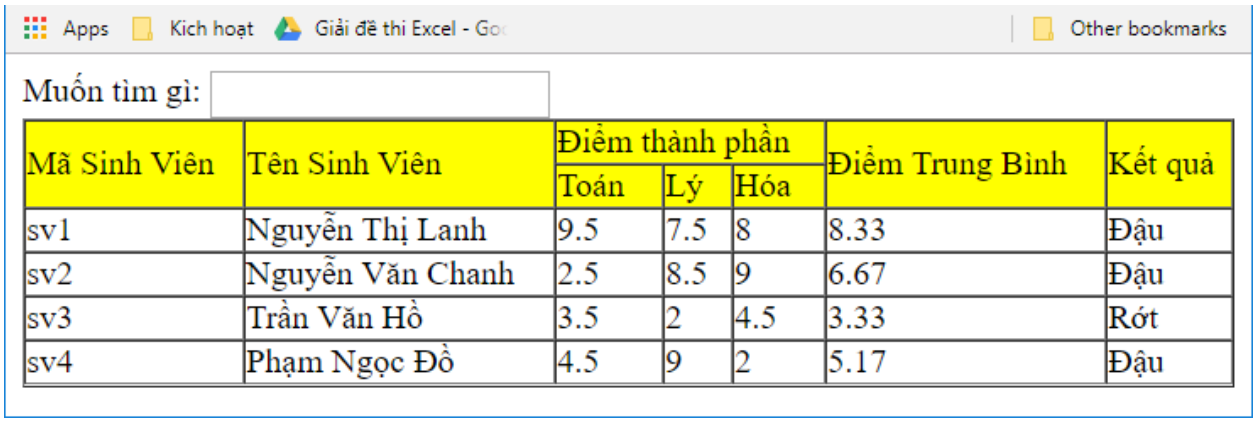
Exercise 74: Show Score - JSon-AJAX

Requirement:

Using JSon to display Student's Score data on the Table, requiring AJAX to read the data (data is outside the HTML file):

This **database-students.json** data file is stored in the directory Exercise 74

http://teacher.uelstore.com/businesswebdevelopment/ex74.zip



Mã Sinh Viên	Tên Sinh Viên	Điểm thành phần			Điểm Trung Bình	Kết quả
		Toán	Lý	Hóa		
sv1	Nguyễn Thị Lanh	9.5	7.5	8	8.33	Đậu
sv2	Nguyễn Văn Chanh	2.5	8.5	9	6.67	Đậu
sv3	Trần Văn Hồ	3.5	2	4.5	3.33	Rớt
sv4	Phạm Ngọc Đồ	4.5	9	2	5.17	Đậu

Instructions:

+ Read the instructions in the Lesson Slide about Json

Exercise 75: Show Subject Book - JsonArray

Requirement:

Given the structure and data of a Json in HTML as follows:

```
sinhvien={
        Ma:1,
        Ten:'Trần Duy Thanh',
        Sachs:
        [
            {Ma:'S1',Ten:'Hồng Lâu Mộng',Trang:100},
            {Ma:'S2',Ten:'Tây Du Ký',Trang:200},
            {Ma:'S3',Ten:'Tam Quốc Chí',Trang:90},
            {Ma:'S4',Ten:'Bích Huyết Kiếm',Trang:70},
            {Ma:'S5',Ten:'Anh Hùng Xạ Điêu',Trang:1000},
            {Ma:'S6',Ten:'Thần Điêu Đại Hiệp',Trang:500},
            {Ma:'S7',Ten:'Tần Thủy Hoàng',Trang:600},
            {Ma:'S8',Ten:'Chiến Quốc',Trang:400},
            {Ma:'S9',Ten:'Hán Sở Tranh Hùng',Trang:300},
            {Ma:'S10',Ten:'Bảo Đao',Trang:700},
        ],
        ChiTiet:function(){
            sv=$scope.sinhvien
            return sv.Ma +" "+sv.Ten
        }
    }
```

Please read the data and display it on the Table as shown below:

Nhập Mã:	
Nhập Tên:	
Mã và Tên:	

Danh sách Sách đang mượn:

Mã Sách	Tên Sách	Số Trang
S4	Bích Huyết Kiếm	70
S3	Tam Quốc Chí	90
S1	Hồng Lâu Mộng	100
S2	Tây Du Ký	200
S9	Hán Sở Tranh Hùng	300
S8	Chiến Quốc	400
S6	Thần Điêu Đại Hiệp	500
S7	Tần Thủy Hoàng	600
S10	Bảo Đao	700
S5	Anh Hùng Xạ Điêu	1000

Instructions:

+ Read the instructions in the Lesson Slide about Json, AngularJS, Google search

Module 7: AngularJS – Binding - Component

Practical knowledge content:

+ Configure and install NodeJS, AngularJS

+ Practice AngularJS commands: Create and run projects, create components, services

+ Learn the component structure of an AngularJS Project

+ Binding programming: Binding, Binding Property, Binding Style, Binding Event, Binding Two Way

+ Type Script programming

+ Programming built-in directives: ngIf, ngSwitch, ngFor

+ Component Programming

Exercise 76: Angular's component architecture

Requirement:

Describe the Angular's component architecture, explaining the function of each component (e.g. App, Module, Component, Service...).

Exercise 77: Install and program Angular

Requirement:

Describe the steps of installing AngularJS

which **commands** to create project (name my-app)? run project?, create component?, create service?

Exercise 78: Binding

Requirement:

Create a Component named “MyComponent”. File Type Script (.ts) contains a variable named “myVar” with value “Hello Angular”, write binding commands to display data on interface (html) in 2 ways:

- Direct access to the variable "myVar"

- Write a function getMyVar() in (.ts) that returns the value of "myVar"

File (html) retrieves and displays the value of "myVar" in (.ts) as follows:

- Display original data of variable "myVar"
- Display all capital letters of "myVar"
- Display all lowercase letters of "myVar"

Exercise 79: Binding Property

Requirement:

Create a "BindingPropertyComponent", Type Script (.ts) containing the following Property (Attribute/Variable):

```
export class BindingPropertyComponent {
  public name:string="Trần Duy Thanh"
  public email:string="thanhtd@uel.edu.vn"
  public nameid:string="nameid"
  public emailid:string="emailid"
  public isDisabled:boolean=true
}
```

HTML uses two ways: create HTML structure in .ts and HTML structure in HTML file to bind these attributes. Use the [] syntax to bind for **name** and {{}} to bind for **email**. The **isDisabled** property will be set for Email, students change true or false to observe the results.

Exercise 80: Binding Class

Requirement:

Create a "BindingClassComponent".

The file "binding-class.component.css" defines the following classes:

```
.text-success{
    color:darkgreen
}
.text-error{
    color:darkred
}
.text-primary{
```

```
    color:navy
}
.text-normal{
    font-style: italic;
}
```

File "binding-class.component.ts" write the following code:

```
export class BindingClassComponent {
  public success="text-success"
  public error="text-error"
  public primary="text-primary"
  public normal="text-normal"
  public multiClass={
    "text-primary":true,
    "text-normal":true,
    "text-error":true
  }
}
```

Request the file "binding-class.component.html": creates 5 <h1> tags using [] syntax to bind classes, creates 5 <h2> tags using {{}} syntax to bind classes. The content displayed in each <h1> and <h2> is arbitrary.

Then define a class ".text-complexity" with font format **Bold**, *Italic*, Font "Cambria", font size 18px, font color Blue. Proceed to define variable names in .ts and binding in html to use this ".text-complexity" class.

Exercise 81: Binding Style

Requirement:

Create a "BindingStyleComponent" component.

The Type Script (.ts) definition is as follows:

```
export class BindingStyleComponent {
  public isError:boolean=false
  public textStyle={
    color:'darkorange',
    fontSize:'26px'
  }
}
```

In the html binding style follow the following syntax:

```
<h1 style="color:red;">Trần Duy Thanh</h1>
<h1 [style.color]="isError? 'red':'darkgreen'">Trần Phạm Thanh Trà</h1>
```

```
<h1 [style]="{'color':'purple','font-size':'26px'}">Trần Phạm Mẫn Nhi</h1>
<h1 [style]="textStyle">Phạm Thị Xuân Diệu</h1>
```

Run and observe the results.

Expanded: Added style for auto-caps all text. Assign **binding** and check the result.

Exercise 82: Binding Event

Requirement:

Create a "Binding Event Component" component.

The Typescript file "binding-event.component.ts" defines two events:

```
export class BindingEventComponent {
  onClick(event:any){
    alert(event.pointerId)
  }
  onSubmit(value:string){
    alert(value)
  }
}
```

The file "binding-event.component.html" defines and binds the event:

```
<p>binding-event works!</p>
<button (click)="onClick($event)">Click Me</button>
Name:
<input type="text" #myName>
<button (click)="onSubmit(myName.value)">Submit</button>
```

Run the component and observe the result.

Expand: Edit the interface for users to enter any 2 numbers a, b and 5 buttons:

Plus button: handle the total output a+b

Subtract button: handle output effectively a-b

Multiply Button: handle output product a*b

Divide button: handle the output of the quotient a/b, note handling the denominator =0

Button reset: Delete data on the interface HTML.

Run the component to check the results after editing.

Exercise 83: Binding Two-Way

Requirement:

Create a "BindingTwoWayComponent" component.

Write coding for "binding-two-way.component.ts" file:

```
export class BindingTwoWayComponent {
  public name:string=''
  public address:string=''
  setDefaultAddress(){
    this.address='13 Hung Vuong street'
  }
}
```

Write HTML code for "binding-two-way.component.html" file:

```
<p>binding-two-way works!</p>
<p>Name: <input [(ngModel)]="name" type="text"></p>
<p>Your name:{{name}}</p>
<p>Address:<input [(ngModel)]="address" type="text"></p>
<p>Your address:{{address}}</p>
<button (click)="setDefaultAddress()">Set default</button>
```

Run the component and observe the result.

Expand: put more email, phone, process binding and re-run component, observe the result after changing.

Exercise 84: Binding Two-Way, model for Quadratic Equation

Requirement:

Create a component with HTML structure as below:

The quadratic equation	
a:	1
b:	3
c:	-4
Result:	x1=-4 ; x2=1
	Find solution Clear

- Use Binding Two-way for mapping variables in Type Script and Html.

- Create a new class with name "Quadratic", and there are 3 variables a, b,c. And one method "findSolution()" to process solutions for Quadratic.
- When clicking on "Find solution" button then using **binding** way in Type Script to receive the input parameter from HTML user interface. And then we use "Quadratic" class to find the solutions, then show the results into the HTML.
- Clicking "Clear" button then reset all data for controls HTML.

***Exercise 85:* Binding Two-Way, model for Lunar Year (*)**

Requirement:

Create a component as below:

Users select Day, Month, Year from the Graphic User Interface (use select/ dropdownlist control). Clicking on "Chuyển đổi" button, then convert Calendar year to Lunar year.

Students must use **binding machanism** to load data into dropdownlist. Write "LunarYear" class with 3 parameters (day, month, calendar year) and "findLunarYearDetail()" method to return the detailed information into HTML.

Instructions:

Google search to find the algorithm about converting Calendar year to Lunar year, and binding machanism to load data into dropdownlist.

Coding example for Binding machanism:

```
days=["1","2","3"]
<select class="form-select" id="days" name="days">
  <option *ngFor="let day of days">{{day}}</option>
</select>
```

Module 8: AngularJS – Service - Routing

Practical knowledge content:

+ Programming built-in directives: ngIf, ngSwitch, ngFor
+ Service Programming
+ Programming Routing
+ And other practical skills

Exercise 86: Json Object Model - Product

Requirement:

Create a Component to display the product as shown below. **JsonObject** and the **binding** mechanism must be used. Declared data in Type Script.

Product ID:	123
Product Name:	Thuốc Trị Xàm
Price:	300

Exercise 87: Json Array Model - Product

Requirement:

Create a Component to display the product list as shown below. Requires the use of **JsonArray** and the **binding** mechanism. Declared data in Type Script.

Product ID	Product Name	Unit Price	Picture
p1	Coca	100	
p2	Pepsi	300	
p3	Sting	200	

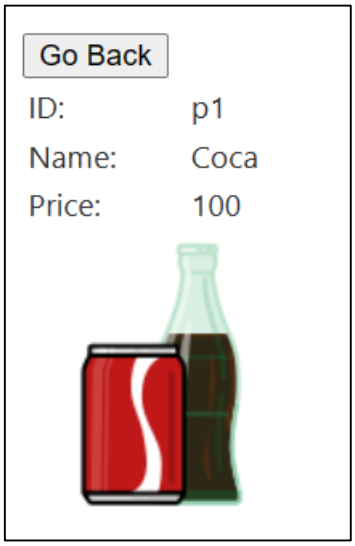
Exercise 88: *Json Array Model – Product Event (*)*

Requirement:

Create a Component to display the product list as shown below. Requires the use of **JsonArray** and the **binding** mechanism. Declared data in Service.

Product ID	Product Name	Unit Price	Picture	#
p1	Coca	100		Details
p2	Pepsi	300		Details
p3	Sting	200		Details

Clicking on **Details** link, web will display the details of that product:



Pressing "Go Back", web will return to the product list screen.

Instructions:

- Create ProductService:

```
export class ProductService {
  productsImage=[
    {"ProductId":"p1","ProductName":"Coca","Price":100,"Image":"assets/h1.png"},
    {"ProductId":"p2","ProductName":"Pepsi","Price":300,"Image":"assets/h2.png"},
    {"ProductId":"p3","ProductName":"Sting","Price":200,"Image":"assets/h3.png"},
  ]
  constructor() { }
  getProductsWithImages()
  {
    return this.productsImage
  }
  getProductDetail(id:any){

    return this.productsImage.find(x=>x.ProductId==id)
  }
}
```

- The images copied to assets
- Create component "service-product-image-event":

+ Write coding for "service-product-image-event.component.ts":

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { ProductService } from '../product.service';

@Component({
  selector: 'app-service-product-image-event',
  templateUrl: './service-product-image-event.component.html',
  styleUrls: ['./service-product-image-event.component.css']
```

```
})
export class ServiceProductImageEventComponent {
  public products:any
  constructor(pservice: ProductService,private router:Router){
    this.products=pservice.getProductsWithImages()
  }
  viewDetail(f:any)
  {
    this.router.navigate(['service-product-image-event',f.ProductId])
  }
}
```

+Write coding for “service-product-image-event.component.html”:

```
<p>service-product-image-event works!</p>
<table border="1">
    <tr>
        <td>Product ID</td>
        <td>Product Name</td>
        <td>UnitPrice</td>
        <td>Picture</td>
        <td>#</td>
    </tr>
    <tbody *ngFor="let p of products">
        <tr>
            <td>{{p.ProductId}}</td>
            <td>{{p.ProductName}}</td>
            <td>{{p.Price}}</td>
            <td>
                <img src="{{p.Image}}">
            </td>
            <td>
                <a style="cursor:pointer;" (click)="viewDetail(p)" title="Click
here to view detail">Details</a>
            </td>
        </tr>
    </tbody>
</table>
```

- Create component “service-product-image-event”:

+ Write coding for “service-product-image-event-detail.component.ts”:

```
import { Component } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { ProductService } from '../product.service';

@Component({
  selector: 'app-service-product-image-event-detail',
  templateUrl: './service-product-image-event-detail.component.html',
  styleUrls: ['./service-product-image-event-detail.component.css']
})
```

```
export class ServiceProductImageEventDetailComponent {
  selectedProduct:any
  constructor(private activateRoute:ActivatedRoute,private _fs:ProductService,
private router:Router)
  {
    activateRoute.paramMap.subscribe(
      (param)=>{
        let id=param.get('id')

        if(id!=null)
        {
          this.selectedProduct=_fs.getProductDetail(id)
        }
      }
    )
  }
  goBack(){
    this.router.navigate(['service-product-image-event'])
  }
}
```

+ Write coding for "service-product-image-event-detail.component.html":

```
<p>service-product-image-event-detail works!</p>
<button (click)="goBack()">Go Back</button>
<table>
    <tr>
        <td>ID:</td>
        <td>{{selectedProduct.ProductId}}</td>
    </tr>
    <tr>
        <td>Name:</td>
        <td>{{selectedProduct.ProductName}}</td>
    </tr>
    <tr>
        <td>Price:</td>
        <td>{{selectedProduct.Price}}</td>
    </tr>
    <tr>
        <td colspan="2">
            <img [src]="selectedProduct.Image"/>
        </td>
    </tr>
</table>
```

- Configure the Routing:

```
{path:'service-product-image-event',
component:ServiceProductImageEventComponent},
{path:'service-product-image-event/:id',
component:ServiceProductImageEventDetailComponent},
```

Exercise 89: Json Array Model – Product - Catalog

Requirement:

The **CatalogService** struct is shown as below:

```
export class CatalogService {
  datas=[
    {"Cateid":"cate1","CateName":"nuoc ngot",
      "Products":[
        {"ProductId":"p1","ProductName":"Coca","Price":100,
"Image":"assets/h1.png"},
        {"ProductId":"p2","ProductName":"Pepsi","Price":300,
"Image":"assets/h2.png"},
        {"ProductId":"p3","ProductName":"Sting","Price":200,
"Image":"assets/h3.png"},
      ]
    },
    {"Cateid":"cate2","CateName":"Bia",
      "Products":[
        {"ProductId":"p4","ProductName":"Heleiken","Price":500,
"Image":"assets/h4.png"},
        {"ProductId":"p5","ProductName":"333","Price":400,
"Image":"assets/h5.png"},
        {"ProductId":"p6","ProductName":"Sai Gon","Price":600,
"Image":"assets/h6.png"},
      ]
    },
  ]

  constructor() { }
  getCategories()
  {
    return this.datas
  }
}
```

- Create component and use nested **ngFor** to display categories and products as shown below:

Ma Danh Muc	Ten Danh Muc
cate1	nuoc ngot

Ma San Pham	Ten San Pham	Gia San Pham	Picture
p1	Coca	100	
p2	Pepsi	300	
p3	Sting	200	

cate2	Bia

Ma San Pham	Ten San Pham	Gia San Pham	Picture
p4	Heleiken	500	
p5	333	400	
p6	Sai Gon	600	

Exercise 90: *Json Array Model – Product– Http Service(*)*

Requirement:

Create a json database in “assets/data/products.json” with sample structure:

```
[
    {"ProductId":"p1","ProductName":"Coca","Price":100,"Image":"assets/h1.png"},
    {"ProductId":"p2","ProductName":"Pepsi","Price":300,"Image":"assets/h2.png"},
    {"ProductId":"p3","ProductName":"Sting","Price":200,"Image":"assets/h3.png"}
]
```

Let's use the **Http Service mechanism** to display the data as shown:

Product ID	Product Name	Unit Price	Picture
p1	Coca	100	
p2	Pepsi	300	
p3	Sting	200	

Instructions:

-Create **IProduct** Interface:

```
export interface IProduct{
  ProductId:string,
  ProductName:string,
  Price:number,
  Image:string
 }
```

(All properties can be left blank):

```
export interface IProduct{


 }
```

-Create "ProductHttpService":

```
export class ProductHttpService {

  private _url:string="./assets/data/products.json";
  constructor(private _http: HttpClient) { }

  getProducts():Observable<IProduct[]>{
    return this._http.get<IProduct[]>(this._url)
  }
}
```

-Create component "service-product-http":

+ File "service-product-http.component.ts":

```
export class ServiceProductHttpComponent {
  products:any;
  constructor(private _service: ProductHttpService){
    this._service.getProducts().subscribe({
      next:(data)=>{this.products=data}
    })
  }
}
```

+ File "service-product-http.component.html":

```
<p>service-product-http works!</p>
<table border="1">
    <tr>
        <td>Product ID</td>
        <td>Product Name</td>
        <td>UnitPrice</td>
```

```
        <td>Picture</td>
    </tr>
    <tbody>
        <tr  *ngFor="let p of products">
            <td>{{p.ProductId}}</td>
            <td>{{p.ProductName}}</td>
            <td>{{p.Price}} </td>
            <td>
                <img src="{{p.Image}}" width="50px" height="50px">
            </td>
        </tr>
    </tbody>
</table>
```

Exercise 91: *Json Array Model – Product– Http Service Handle Error (*)*

Requirement:

Similar to exercise 90, however, exercise 91 handles an exception that occurs.

For example, the database does not exist..

Instructions:

-Add "ProductHttpService", and then intentionally correct the data path:

```
export class ProductHttpService {

  private _url:string="./assets/data/productsXXX.json";
  constructor(private _http: HttpClient) { }

  getProducts():Observable<IProduct[]>{
    return this._http.get<IProduct[]>(this._url)
  }
  getProductsHandleError()
  {
    return this._http.get<IProduct[]>(this._url)
      .pipe(retry(3),
      catchError(this.handleError))
  }

  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }

}
```

-Create component "service-product-http-handle-error":

+ File "service-product-http-handle-error.component.ts":

```
export class ServiceProductHttpHandleErrorComponent {
  products:any
  errMessage:string=''
  constructor(_service:ProductHttpService){
    _service.getProductsHandleError().subscribe({
      next:(data)=>{this.products=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

+ File "service-product-http-handle-error.component.html":

```
<p>service-product-http-handle-error works!</p>
{{errMessage}}
<table border="1">
    <tr>
        <td>Product ID</td>
        <td>Product Name</td>
        <td>UnitPrice</td>
        <td>Picture</td>
    </tr>
    <tbody>
        <tr  *ngFor="let p of products">
            <td>{{p.ProductId}}</td>
            <td>{{p.ProductName}}</td>
            <td>{{p.Price}}</td>
            <td>
                <img src="{{p.Image}}" width="50px" height="50px">
            </td>
        </tr>
    </tbody>
</table>
```

Running the website we have the results (in case we have corrected the database path wrong):

service-product-http-handle-error works!

Error: Http failure response for http://localhost:4200/assets/data/products1.json: 404 Not Found

Product ID Product Name Unitprice Picture

If we correct it, we have the same result as in exercise 90.

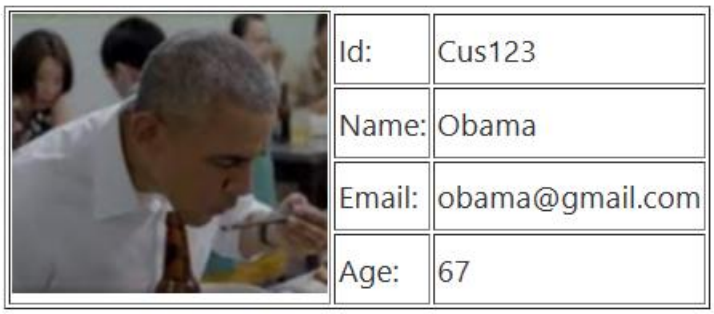
Exercise 92: Json Object Model – Customer – Service

Requirement:

Create a Component to display the Customer as shown below. Requires the use of **JsonObject** and the **binding** mechanism. Declared data in Type Script.

```
customer={
    "Id":"Cus123",
    "Name":"Obama",
    "Email":"obama@gmail.com",
    "Age":67,
    "Image":"assets/avatars/obama-avatar.png"
  }
```

HTML Interface when running the component:



Exercise 93: Json Array Model – Group Customers (*)

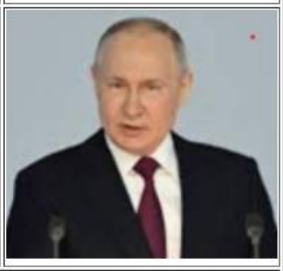
Requirement:

Create a Json file to save the list of Customers in "assets/data/customers.json" with the sample structure as below:

```
[
    {"CustomerTypeId":1,"CustomterTypeName":"VIP",
    "Customers":[{
      "Id":"Cus123",
      "Name":"Obama",
      "Email":"obama@gmail.com",
      "Age":67,
      "Image":"assets/avatars/obama-avatar.png"
    },
    {
      "Id":"Cus456",
      "Name":"Kim jong Un",
      "Email":"unun@gmail.com",
      "Age":38,
```

```
      "Image":"assets/avatars/unun-avatar.png"
    }
    ,
    {
      "Id":"Cus789",
      "Name":"Putin",
      "Email":"putin@gmail.com",
      "Age":77,
      "Image":"assets/avatars/putin-avatar.png"
    }]
    },
    {"CustomerTypeId":2,"CustomterTypeName":"Normal",
    "Customers":[{
      "Id":"Cus000",
      "Name":"Hồ Cẩm Đào",
      "Email":"hodao@gmail.com",
      "Age":16,
      "Image":"assets/avatars/hodao-avatar.png"
    },
    {
      "Id":"Cus111",
      "Name":"Tap Can Binh",
      "Email":"binhbinh@gmail.com",
      "Age":67,
      "Image":"assets/avatars/binhbinh-avatar.png"
    },
    ]
    },
  ]
```


Applying the exercise 89 (displaying group data) and exercise 90 (using http service) to do this exercise.

Id	Name	Email	Age	#
1 - VIP				
Cus123	Obama	obama@gmail.com	67	
Cus456	Kim jong Un	unun@gmail.com	38	
Cus789	Putin	putin@gmail.com	77	
2 - Normal				
Cus000	Hồ Cẩm Đào	hodao@gmail.com	16	
Cus111	Tap Can Binh	binhbinh@gmail.com	67	

Exercise 94: *Configuration Routing*

Roughly understanding Routing is page navigation, the example we see:

http://localhost:4200/products

http://localhost:4200/customers

http://localhost:4200/login

When accessing the above links and it navigates to the defined content pages, it is known as Routing.

To navigate the Page we use Routing technique, the steps are as follows:

Step 1: Create Components, example ProductComponent, ListProductComponent, ServiceProductComponent, …

Then we want to type the following links after opening these components:

http://localhost:4200/product => Open ProductComponent

http://localhost:4200/list-product => Open ListProductComponent

http://localhost:4200/service-product => Open ServiceProductComponent

Step 2: Modify the "app-routing.module.ts" file

Declare path and component directives for routes array as below:

```
const routes: Routes = [
  {path:'product',component:ProductComponent},
  {path:'list-product',component:ListProductComponent},
  {path:'service-product',component:ServiceProductComponent},
];
```

At the end of this file continue to declare the variable "RoutingComponent":

```
export const RoutingComponent=[
ProductComponent,
ListProductComponent,
ServiceProductComponent,
]
```

Step 3: Modify the file "app.module.ts":

The "**declarations**" tag adds the RoutingComponent that we declared in **step 2**

```
@NgModule({
  declarations: [
    AppComponent,
    RoutingComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

At this point, the program understands the Routing already.

Step 4: (This step is not necessary)
Suppose in the application (home page) we want to navigate to the Routing that we have declared, then do the following:

Open file "app.component.html"

```
<table>
  <tr>
   <td>
    <input type="button" value="Product" [routerLink]="'product'">
   </td>
    <td>
<input type="button" value="List-Product" [routerLink]="'list-product'">
</td>
    <td>
<input type="button" value="Service-Product" [routerLink]="'service-product'">
</td>
  </tr>
</table>
```

Exercise 95: Routing for Page Not Found

To navigate through Page Not found when the user makes a wrong choice, configure the last line in routes:

```
const routes: Routes = [
  {path:'product',component:ProductComponent},
  {path:'list-product',component:ListProductComponent},
  {path:'service-product',component:ServiceProductComponent},
  {path:"**",component:PageNotFoundComponent},
];
```

Module 9: AngularJS - Form

Practical knowledge content:

+ Understand the meaning and function of Form, Reactive Form

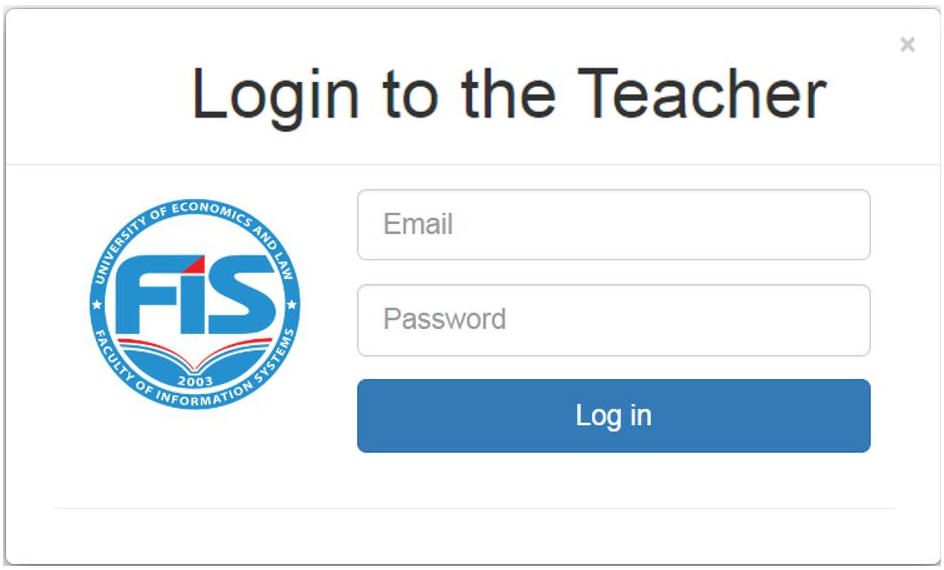
+ Using FormsModule, ngModel

+ Binding Data, Validation

+ Understand and use the controls on the Form: Label, Textbox, Button, Combo, ListBox, TextArea....

Exercise 96: Login Screen

Requirement:
Design the login interface similar to below:

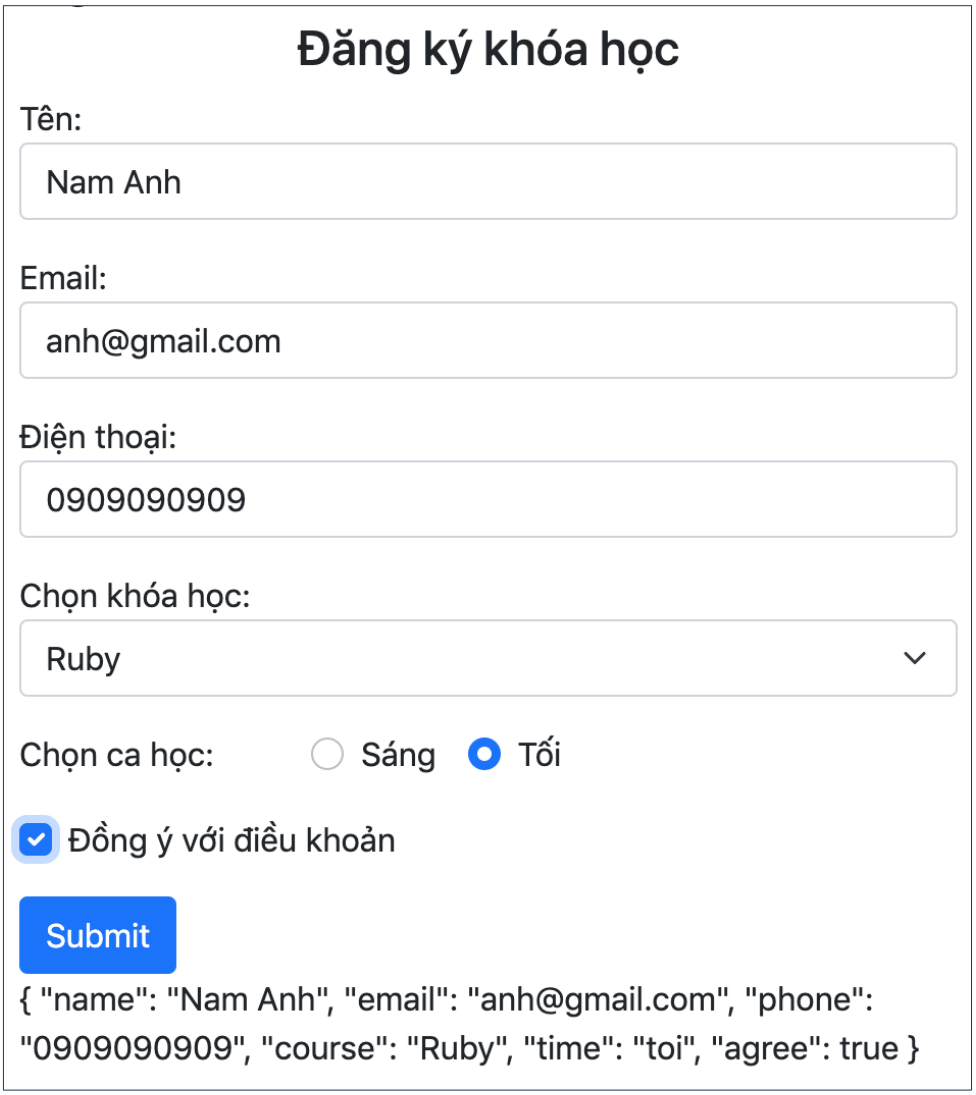


- Requires creating a model with email and password attributes
- When the "Login" button is pressed, the JSonObject as string of information and user input is displayed below the "Login" button (p tag, red letter).
- **Additional 1:** Added validation for invalid email, password at least 5 characters
- **Additional 2:** Add simulation, if you enter the correct email and password, navigate to any component.
- **Additional 3:** Add Checkbox button (save login information), search keyword "Angular Local Storage". If checkbox is checked before, the next time it will reload the login information when running the login screen again.

Exercise 97: Course Registration Form

Requirement:

Design the Course Registration screen as shown below:



- Requires a model class declaration to **bind** data
- When you click Submit, the JSonObject as string information will be displayed right below the Submit button.

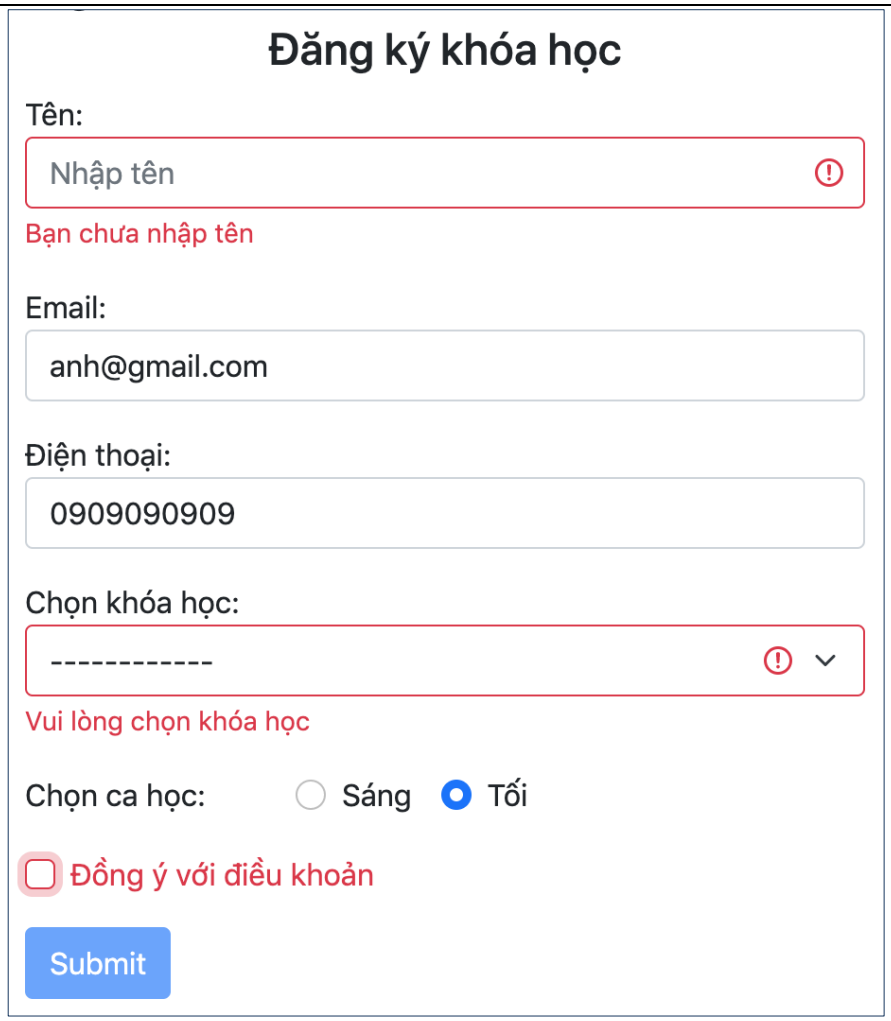
Instructions:

- Using **ReactiveFormsModule**

Exercise 98: Course Registration Form - Validation

Requirement:

Expand the Course Registration Form exercise, add Validation



When you press the "**Submit**" button:

- If the name is empty, it will give an error
- If the course is not selected, an error will be reported
- If you don't click on "agree to terms" you will get an error

Instructions:

Exercise 99: Mathematics

Requirement:

Create a website with Table and Textbox, Button to calculate MAX, MIN, sin, cos... As described below:

a:	30
b:	45
c:	90
Result:	-0.9880316240928618

MAX | MIN | sin(a) | Cos(a) | a^b

- Show Error (red background) of Textbox if the numeric data format is wrong.

Module 10: AngularJS invoke External Restful API

Practical knowledge content:

+ Practice CSS
+ Embed Bootstrap
+ Practice Service
+ Practice Components
+ Practice HTTP Call external restful API
+ Practice Http Handling Error
+ Handling CORS Policy

Exercise 100: Interactive External Restful API- Dong A Bank

Requirement:

DONG A Bank provides the following Restful API to retrieve the exchange rates of several countries:

https://www.dongabank.com.vn/exchange/export

Below is the data of the API:


```
({"items":[{"type":"GBP","imageurl":"https:\/\/www.dongabank.com.vn\/images\/
flag\/GBP.gif","muatienmat":"28050","muack":"28170","bantienmat":"28760","ban
ck":"28710"},{"type":"AUD","imageurl":"https:\/\/www.dongabank.com.vn\/images
\/flag\/AUD.gif","muatienmat":"15800","muack":"15890","bantienmat":"16230","b
anck":"16190"},{"type":"CAD","imageurl":"https:\/\/www.dongabank.com.vn\/imag
es\/flag\/CAD.gif","muatienmat":"17220","muack":"17320","bantienmat":"17690",
"banck":"17650"},{"type":"CHF","imageurl":"https:\/\/www.dongabank.com.vn\/im
ages\/flag\/CHF.gif","muatienmat":"22610","muack":"25000","bantienmat":"23080
","banck":"25500"},{"type":"CNY","imageurl":"https:\/\/www.dongabank.com.vn\/
images\/flag\/CNY.gif","muatienmat":"3000","muack":"3000","bantienmat":"3500"
,"banck":"3500"},{"type":"EUR","imageurl":"https:\/\/www.dongabank.com.vn\/im
ages\/flag\/EUR.gif","muatienmat":"24860","muack":"24970","bantienmat":"25490
","banck":"25440"},{"type":"HKD","imageurl":"https:\/\/www.dongabank.com.vn\/
images\/flag\/HKD.gif","muatienmat":"2410","muack":"2900","bantienmat":"2920"
,"banck":"3060"},{"type":"XAU","imageurl":"https:\/\/www.dongabank.com.vn\/im
ages\/flag\/XAU.gif","muatienmat":"6620000","muack":"6655000","bantienmat":"6
690000","banck":"6655000"},{"type":"USD","imageurl":"https:\/\/www.dongabank.
com.vn\/images\/flag\/USD.gif","muatienmat":"23590","muack":"23590","bantienm
at":"23920","banck":"23870"},{"type":"THB","imageurl":"https:\/\/www.dongaban
k.com.vn\/images\/flag\/THB.gif","muatienmat":"610","muack":"670","bantienmat
":"700","banck":"690"},{"type":"SGD","imageurl":"https:\/\/www.dongabank.com.
vn\/images\/flag\/SGD.gif","muatienmat":"17310","muack":"17470","bantienmat":
"17820","banck":"17800"},{"type":"PNJ_DAB","imageurl":"https:\/\/www.dongaban
k.com.vn\/images\/flag\/PNJ_DAB.gif","muatienmat":"5330000","muack":"5330000"
```

```
,"bantienmat":"5430000","banck":"5430000"},{"type":"NZD","imageurl":"https:\/
\/www.dongabank.com.vn\/images\/flag\/NZD.gif","muatienmat":"","muack":"14650
","bantienmat":"","banck":"15060"},{"type":"JPY","imageurl":"https:\/\/www.do
ngabank.com.vn\/images\/flag\/JPY.gif","muatienmat":"169","muack":"172.4","ba
ntienmat":"176.1","banck":"175.6"}]})
```


Ask Student to use HTTP and handling error to call this API and display the Graphic User Interface as shown below:

Dong A Bank

Type	ImageURL	MuaTienMat	MuaCK	BanTienMat	BanCK
CAD		17220	17320	17690	17650
XAU		6620000	6655000	6690000	6655000
AUD		15800	15890	16230	16190
CHF		22610	25000	23080	25500
CNY		3000	3000	3500	3500
EUR		24860	24970	25490	25440
GBP		28050	28170	28760	28710
HKD		2410	2900	2920	3060
JPY		169	172.4	176.1	175.6
NZD			14650		15060
PNJ_DAB		5330000	5330000	5430000	5430000
SGD		17310	17470	17820	17800
THB		610	670	700	690
USD		23590	23590	23920	23870

Detailed instructions:

As learned, Json data has 2 types: JsonObject **{}** and JsonArray**[]**. However, the data sent by Dong A Bank is surrounded by round brackets **( )**. So We need to load the data and remove the brackets ( ) so that it becomes JSonObject (wrapped by curly braces {}).

Observe that “items” contains array of JSonObject:

{"type","imageurl","muatienmat","muack","bantienmat","banck"}

So we need to create 2 interfaces (or 2 classes) to model and map data.

Step 1: Create modeling classes for data

Create file "DongABankItem.ts" and declare interface as below:

```
export interface IDongABankItem
{
    type:string,
    imageurl:string,
    muatienmat:string,
    muack:string,
    bantienmat:string,
    banck:string,
}
```

Continue to create the file "DongABankData.ts" and declare the interface as below:

```
import { IDongABankItem } from "./DongABankItem";

export interface IDongABankData
{
    items:Array<IDongABankItem>
}
```

So with the declaration of the above two interfaces, the program will **map** the **Json** data sent by **Dong A Bank** into an **object-oriented** model.

The most important thing is that we have to understand the correct Json data structure that any API returns, and the correct understanding of the Json structure to declare the correct class models.

Attributes declared in the interface or in the class We should copy + paste from the data returned by the API to avoid typos (must be correct in both uppercase and lowercase).

Step 2: Service declaration

Create a Service named "DongABankService" now the file "dong-abank.service.ts" will be created, we proceed to coding for this file.

```
import { JsonPipe } from '@angular/common';
import { HttpClient, HttpErrorResponse, HttpHeaders } from
'@angular/common/http';
import { Injectable } from '@angular/core';
import { catchError, map, Observable, retry, throwError } from 'rxjs';
import { IDongABankData } from './interfaces/DongABankData';
import { IDongABankItem } from './interfaces/DongABankItem';

@Injectable({
  providedIn: 'root'
})
export class DongABankService {

  private _url:string="/exchange/export"
  constructor(private _http: HttpClient) { }
  getDongABankData()
```

```
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-
8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>(this._url,requestOptions).pipe(
      map(res=>JSON.parse(res.slice(1,-1)) as IDongABankData),
      retry(3),
      catchError(this.handleError))
  }

  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }
}
```

res.slice(1,-1) removes the parentheses at the beginning and at the end of the data returned by the API

JSON.parse(res.slice(1,-1)) returns the data to JSon

JSON.parse(res.slice(1,-1)) as IDongABankData models Json into an object-oriented model.

handleError helps to check for errors during API invocation

Step 3: Create Component

Create "DongABankComponent" component, now we code in the file "dong-abank.component.ts" as follows:

```
import { Component } from '@angular/core';
import { DongABankService } from '../dong-abank.service';
@Component({
  selector: 'app-dong-abank',
  templateUrl: './dong-abank.component.html',
  styleUrls: ['./dong-abank.component.css']
})
export class DongABankComponent {
  data:any
  errMessage:string=''
  constructor(_service:DongABankService){
    _service.getDongABankData().subscribe({
      next:(data)=>{
        this.data=data
      },
      error:(err)=>{
        this.errMessage=err}
    })
  }
}
```

Continue coding the html in the file "dong-abank.component.html":

```
<h3 class="text-center text-white bg-dark p-3">Dong A Bank</h3>
{{errMessage}}
<div class="container">
    <table class="table table -hover">
        <thead>
            <tr>
                <th>Type</th>
                <th>ImageURL</th>
                <th>MuaTienMat</th>
                <th>MuaCK</th>
                <th>BanTienMat</th>
                <th>BanCK</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let item of data.items">
                <td>{{item.type}}</td>
                <td><img src="{{item.imageurl}}"></td>
                <td>{{item.muatienmat}}</td>
                <td>{{item.muack}}</td>
                <td>{{item.bantienmat}}</td>
                <td>{{item.banck}}</td>
            </tr>
        </tbody>
    </table>
</div>
```

Step 4: Handling CORS Policy

Attention: when calling external restful API normally we will get the same error:

```
Access to XMLHttpRequest at 'https://www.dongabank.com.vn/e localhost/:1
xchange/export' from origin 'http://localhost:4200' has been blocked by
CORS policy: No 'Access-Control-Allow-Origin' header is present on the
requested resource.
```

So we need to handle CORS Policy Handling as follows:

- Create file src/**proxy.conf.json** and declare the syntax

Since usually a project can connect many different APIs (arrays), we declare the contents of this proxy file as an array for convenience.:

```
[
    {
      "context": ["/exchange"],
      "target": "https://www.dongabank.com.vn",
      "secure": true,
      "changeOrigin":true,
      "logLevel":"debug"
    }
 ]
```

"context" reference to the API we want to access

"target" is the domain where the API is configured

Other parameters are set as shown above. When calling many other APIs, just Copy + paste the **{}** structure into many, then adjust the context and target.

In case if the project only has access to a single API, we can declare the Object type as below for "**proxy.conf.json**" (but it's probably rare, since a project often accesses many APIs) :

```
{
    "/exchange":{
        "target":"https://www.dongabank.com.vn",
        "secure":true,
        "changeOrigin":true,
        "logLevel":"debug"
    }
}
```

After declaring the file "proxy.conf.json", we proceed to configure "angular.json":

Add red frame.

```
"serve": {
          "builder": "@angular-devkit/build-angular:dev-server",
          "options": {
            "browserTarget": "my-app:build",
            "proxyConfig": "src/proxy.conf.json"
          },
          "configurations": {
            "production": {
              "browserTarget": "my-app:build:production"
```

```
            },
            "development": {
              "browserTarget": "my-app:build:development"
            }
          },
          "defaultConfiguration": "development"
        },
```

"**browserTarget**" reference to the name of the application is created, for example "**my-app:build**"

"**proxyConfig**" reference to the proxy file is defined, for example "**src/proxy.conf.json**"

Step 5: Install BootStrap – Jquery (not needed if not used)

Execute command:

```
npm install --save bootstrap jquery
```

Continue configuring "angular.json":

```
12        "architect": {
13          "build": {
14            "builder": "@angular-devkit/build-angular:browser",
15            "options": {
16              "outputPath": "dist/my-app",
17              "index": "src/index.html",
18              "main": "src/main.ts",
19              "polyfills": [
20                "zone.js"
21              ],
22              "tsConfig": "tsconfig.app.json",
23              "assets": [
24                "src/favicon.ico",
25                "src/assets"
26              ],
27              "styles": [
28                "src/styles.css",
29                "node_modules/bootstrap/dist/css/bootstrap.min.css"
30              ],
31              "scripts": [
32                "node_modules/jquery/dist/jquery.min.js",
33                "node_modules/bootstrap/dist/js/bootstrap.min.js"
34              ]
35            },
```

Add red frame.

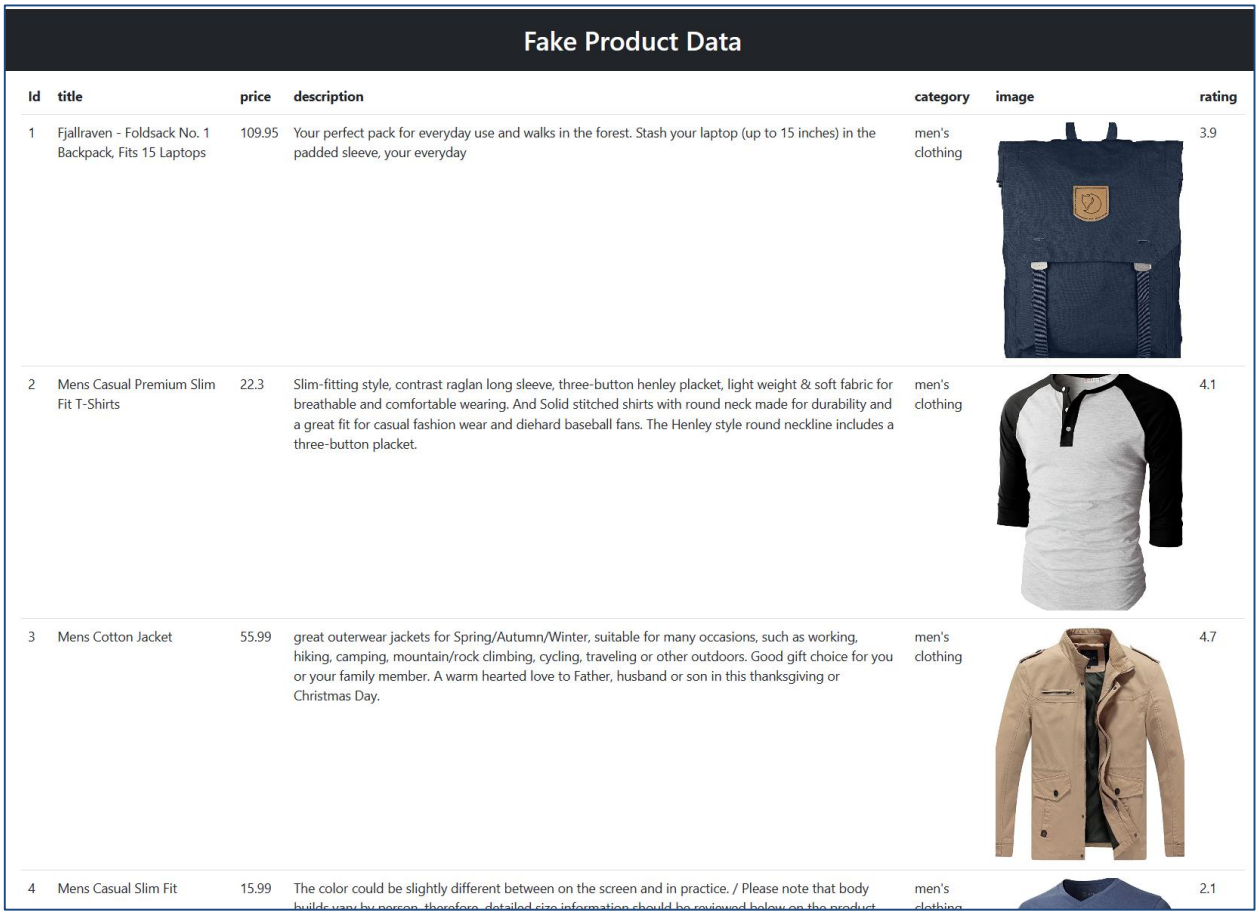
```
"styles": [
            "src/styles.css",
            "node_modules/bootstrap/dist/css/bootstrap.min.css"
          ],
          "scripts": [
            "node_modules/jquery/dist/jquery.min.js",
            "node_modules/bootstrap/dist/js/bootstrap.min.js"
          ]
```

Run the program and we get the desired result.

Exercise 101: Interaction External Restful API- Product Service 1

Requirement:

Students do the same as the API call exercise of Dong A bank, load the product list from the API: https://fakestoreapi.com/products displays the interface as follows:



Fake Product Data

Id	title	price	description	category	image	rating
1	Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops	109.95	Your perfect pack for everyday use and walks in the forest. Stash your laptop (up to 15 inches) in the padded sleeve, your everyday	men's clothing		3.9
2	Mens Casual Premium Slim Fit T-Shirts	22.3	Slim-fitting style, contrast raglan long sleeve, three-button henley placket, light weight & soft fabric for breathable and comfortable wearing. And Solid stitched shirts with round neck made for durability and a great fit for casual fashion wear and diehard baseball fans. The Henley style round neckline includes a three-button placket.	men's clothing		4.1
3	Mens Cotton Jacket	55.99	great outerwear jackets for Spring/Autumn/Winter, suitable for many occasions, such as working, hiking, camping, mountain/rock climbing, cycling, traveling or other outdoors. Good gift choice for you or your family member. A warm hearted love to Father, husband or son in this thanksgiving or Christmas Day.	men's clothing		4.7
4	Mens Casual Slim Fit	15.99	The color could be slightly different between on the screen and in practice. / Please note that body builds vary by person, therefore, detailed size information should be reviewed below on the product	men's clothing		2.1

Detailed instructions:

Observe the data returned by the API https://fakestoreapi.com/products:


```
[{"id":1,"title":"Fjallraven - Foldsack No. 1 Backpack, Fits 15
Laptops","price":109.95,"description":"Your perfect pack for everyday use and walks in the
forest. Stash your laptop (up to 15 inches) in the padded sleeve, your
everyday","category":"men's clothing","image":"https://fakestoreapi.com/img/81fPKd-
2AYL._AC_SL1500_.jpg","rating":{"rate":3.9,"count":120}},{"id":2,"title":"Mens Casual Premium
Slim Fit T-Shirts ","price":22.3,"description":"Slim-fitting style, contrast raglan long sleeve,
three-button henley placket, light weight & soft fabric for breathable and comfortable wearing.
And Solid stitched shirts with round neck made for durability and a great fit for casual fashion
wear and diehard baseball fans. The Henley style round neckline includes a three-button
placket.","category":"men's clothing","image":"https://fakestoreapi.com/img/71-
3HjGNDUL._AC_SY879._SX._UX._SY._UY_.jpg","rating":{"rate":4.1,"count":259}},{"id":3,"title":"Mens
Cotton Jacket","price":55.99,"description":"great outerwear jackets for Spring/Autumn/Winter,
suitable for many occasions, such as working, hiking, camping, mountain/rock climbing, cycling,
traveling or other outdoors. Good gift choice for you or your family member. A warm hearted love
to Father, husband or son in this thanksgiving or Christmas Day.","category":"men's
clothing","image":"https://fakestoreapi.com/img/71li-
ujtlUL._AC_UX679_.jpg","rating":{"rate":4.7,"count":500}},{"id":4,"title":"Mens Casual Slim
Fit","price":15.99,"description":"The color could be slightly different between on the screen and
in practice. / Please note that body builds vary by person, therefore, detailed size information
should be reviewed below on the product description.","category":"men's
clothing","image":"https://fakestoreapi.com/img/71YXzeOuslL._AC_UY879_.jpg","rating":{"rate":2.1,
"count":430}},{"id":5,"title":"John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain
Bracelet","price":695,"description":"From our Legends Collection, the Naga was inspired by the
```

mythical water dragon that protects the ocean's pearl. Wear facing inward to be bestowed with love and abundance, or outward for protection.","category":"jewelery","image":"https://fakestoreapi.com/img/71pWzhdJNwL._AC_UL640_QL65_ML3_.jpg","rating":{"rate":4.6,"count":400}},{"id":6,"title":"Solid Gold Petite Micropave ","price":168,"description":"Satisfaction Guaranteed. Return or exchange any order within 30 days.Designed and sold by Hafeez Center in the United States. Satisfaction Guaranteed. Return or exchange any order within 30 days.","category":"jewelery","image":"https://fakestoreapi.com/img/61sbMiUnoGL._AC_UL640_QL65_ML3_.jpg","rating":{"rate":3.9,"count":70}},{"id":7,"title":"White Gold Plated Princess","price":9.99,"description":"Classic Created Wedding Engagement Solitaire Diamond Promise Ring for Her. Gifts to spoil your love more for Engagement, Wedding, Anniversary, Valentine's Day...","category":"jewelery","image":"https://fakestoreapi.com/img/71YAIFU48IL._AC_UL640_QL65_ML3_.jpg","rating":{"rate":3,"count":400}},{"id":8,"title":"Pierced Owl Rose Gold Plated Stainless Steel Double","price":10.99,"description":"Rose Gold Plated Double Flared Tunnel Plug Earrings. Made of 316L Stainless Steel","category":"jewelery","image":"https://fakestoreapi.com/img/51UDEzMJVpL._AC_UL640_QL65_ML3_.jpg","rating":{"rate":1.9,"count":100}},{"id":9,"title":"WD 2TB Elements Portable External Hard Drive - USB 3.0 ","price":64,"description":"USB 3.0 and USB 2.0 Compatibility Fast data transfers Improve PC Performance High Capacity; Compatibility Formatted NTFS for Windows 10, Windows 8.1, Windows 7; Reformatting may be required for other operating systems; Compatibility may vary depending on user’s hardware configuration and operating system","category":"electronics","image":"https://fakestoreapi.com/img/61IBBVJvSDL._AC_SY879_.jpg","rating":{"rate":3.3,"count":203}},{"id":10,"title":"SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s","price":109,"description":"Easy upgrade for faster boot up, shutdown, application load and response (As compared to 5400 RPM SATA 2.5” hard drive; Based on published specifications and internal benchmarking tests using PCMark vantage scores) Boosts burst write performance, making it ideal for typical PC workloads The perfect balance of performance and reliability Read/write speeds of up to 535MB/s/450MB/s (Based on internal testing; Performance may vary depending upon drive capacity, host device, OS and application.)","category":"electronics","image":"https://fakestoreapi.com/img/61U7T1koQqL._AC_SX679_.jpg","rating":{"rate":2.9,"count":470}},{"id":11,"title":"Silicon Power 256GB SSD 3D NAND A55 SLC Cache Performance Boost SATA III 2.5","price":109,"description":"3D NAND flash are applied to deliver high transfer speeds Remarkable transfer speeds that enable faster bootup and improved overall system performance. The advanced SLC Cache Technology allows performance boost and longer lifespan 7mm slim design suitable for Ultrabooks and Ultra-slim notebooks. Supports TRIM command, Garbage Collection technology, RAID, and ECC (Error Checking & Correction) to provide the optimized performance and enhanced reliability.","category":"electronics","image":"https://fakestoreapi.com/img/71kWymZ+c+L._AC_SX679_.jpg","rating":{"rate":4.8,"count":319}},{"id":12,"title":"WD 4TB Gaming Drive Works with Playstation 4 Portable External Hard Drive","price":114,"description":"Expand your PS4 gaming experience, Play anywhere Fast and easy, setup Sleek design with high capacity, 3-year manufacturer's limited warranty","category":"electronics","image":"https://fakestoreapi.com/img/61mtL65D4cL._AC_SX679_.jpg","rating":{"rate":4.8,"count":400}},{"id":13,"title":"Acer SB220Q bi 21.5 inches Full HD (1920 x 1080) IPS Ultra-Thin","price":599,"description":"21. 5 inches Full HD (1920 x 1080) widescreen IPS display And Radeon free Sync technology. No compatibility for VESA Mount Refresh Rate: 75Hz - Using HDMI port Zero-frame design | ultra-thin | 4ms response time | IPS panel Aspect ratio - 16: 9. Color Supported - 16. 7 million colors. Brightness - 250 nit Tilt angle -5 degree to 15 degree. Horizontal viewing angle-178 degree. Vertical viewing angle-178 degree 75 hertz","category":"electronics","image":"https://fakestoreapi.com/img/81QpkIctqPL._AC_SX679_.jpg","rating":{"rate":2.9,"count":250}},{"id":14,"title":"Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED ","price":999.99,"description":"49 INCH SUPER ULTRAWIDE 32:9 CURVED GAMING MONITOR with dual 27 inch screen side by side QUANTUM DOT (QLED) TECHNOLOGY, HDR support and factory calibration provides stunningly realistic and accurate color and contrast 144HZ HIGH REFRESH RATE and 1ms ultra fast response time work to eliminate motion blur, ghosting, and reduce input lag","category":"electronics","image":"https://fakestoreapi.com/img/81Zt42ioCgL._AC_SX679_.jpg","rating":{"rate":2.2,"count":140}},{"id":15,"title":"BIYLACLESEN Women's 3-in-1 Snowboard Jacket Winter Coats","price":56.99,"description":"Note:The Jackets is US standard size, Please choose size as your usual wear Material: 100% Polyester; Detachable Liner Fabric: Warm Fleece. Detachable Functional Liner: Skin Friendly, Lightweigt and Warm.Stand Collar Liner jacket, keep you warm in cold weather. Zippered Pockets: 2 Zippered Hand Pockets, 2 Zippered Pockets on Chest (enough to keep cards or keys)and 1 Hidden Pocket Inside.Zippered Hand Pockets and Hidden Pocket keep your things secure. Humanized Design: Adjustable and Detachable Hood and Adjustable cuff to prevent the wind and water,for a comfortable fit. 3 in 1 Detachable Design provide more convenience, you can separate the coat and inner as needed, or wear it together. It is suitable for different season and help you adapt to different climates","category":"women's clothing","image":"https://fakestoreapi.com/img/51Y5NI-I5jL._AC_UX679_.jpg","rating":{"rate":2.6,"count":235}},{"id":16,"title":"Lock and Love Women's Removable Hooded Faux Leather Moto Biker Jacket","price":29.95,"description":"100% POLYURETHANE(shell) 100% POLYESTER(lining) 75% POLYESTER 25% COTTON (SWEATER), Faux leather material for style and comfort / 2 pockets of front, 2-For-One Hooded denim style faux leather jacket, Button detail on waist / Detail stitching at sides, HAND WASH ONLY / DO NOT BLEACH / LINE DRY / DO NOT IRON","category":"women's clothing","image":"https://fakestoreapi.com/img/81XH0e8fefL._AC_UY879_.jpg","rating":{"rate":2.9,"count":340}},{"id":17,"title":"Rain Jacket Women Windbreaker Striped Climbing Raincoats","price":39.99,"description":"Lightweight perfet for trip or casual wear---Long sleeve with hooded, adjustable drawstring waist design. Button and zipper front closure raincoat, fully stripes Lined and The Raincoat has 2 side pockets are a good size to hold all kinds of things, it covers the hips, and the hood is generous but doesn't overdo it.Attached Cotton Lined Hood with

```
Adjustable Drawstrings give it a real styled look.","category":"women's
clothing","image":"https://fakestoreapi.com/img/71HblAHs5xL._AC_UY879_-
2.jpg","rating":{"rate":3.8,"count":679}},{"id":18,"title":"MBJ Women's Solid Short Sleeve Boat
Neck V ","price":9.85,"description":"95% RAYON 5% SPANDEX, Made in USA or Imported, Do Not
Bleach, Lightweight fabric with great stretch for comfort, Ribbed on sleeves and neckline /
Double stitching on bottom hem","category":"women's
clothing","image":"https://fakestoreapi.com/img/71z3kpMAYsL._AC_UY879_.jpg","rating":{"rate":4.7,
"count":130}},{"id":19,"title":"Opna Women's Short Sleeve
Moisture","price":7.95,"description":"100% Polyester, Machine wash, 100% cationic polyester
interlock, Machine Wash & Pre Shrunk for a Great Fit, Lightweight, roomy and highly breathable
with moisture wicking fabric which helps to keep moisture away, Soft Lightweight Fabric with
comfortable V-neck collar and a slimmer fit, delivers a sleek, more feminine silhouette and Added
Comfort","category":"women's
clothing","image":"https://fakestoreapi.com/img/51eg55uWmdL._AC_UX679_.jpg","rating":{"rate":4.5,
"count":146}},{"id":20,"title":"DANVOUY Womens T Shirt Casual Cotton
Short","price":12.99,"description":"95%Cotton,5%Spandex, Features: Casual, Short Sleeve, Letter
Print,V-Neck,Fashion Tees, The fabric is soft and has some stretch., Occasion:
Casual/Office/Beach/School/Home/Street. Season: Spring,Summer,Autumn,Winter.","category":"women's
clothing","image":"https://fakestoreapi.com/img/61pHAEJ4NML._AC_UX679_.jpg","rating":{"rate":3.6,
"count":145}}]
```


Looking at the above structure, it is simpler than the data of Dong A Bank. It is an array of objects and from the beginning it was true to the structure of Json Data.

Step 1: Create model interfaces/classes

For "rating":{"rate":3.6,"count":145} need an interface, name the file "**FakeProductRating.ts**" and declare the interface "IFakeProductRating" with content:

```
export interface IFakeProductRating{
    rate:number,
    count:number
}
```

For external structure, create file "**FakeProduct.ts**" and declare interface "IFakeProduct" with content:

```
import {IFakeProductRating } from "./FakeProductRating";

export interface IFakeProduct{
   id:number,
   title:string,
   price:number,
   description:string,
   category:string,
   image:string    ,
   rate:IFakeProductRating
}
```

Step 2: Create "FakeProductService" Service, write code in "**fake-product.service.ts**" file as follows:

```
import { HttpClient, HttpErrorResponse, HttpHeaders } from
'@angular/common/http';
import { Injectable } from '@angular/core';
import { catchError, map, Observable, retry, throwError } from 'rxjs';
import { IFakeProduct } from './interfaces/FakeProduct';
@Injectable({
  providedIn: 'root'
})
export class FakeProductService {
  private _url:string="/products"
  constructor(private _http: HttpClient) { }
  getFakeProductData():Observable<any> {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>(this._url,requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IFakeProduct>),
        retry(3),
        catchError(this.handleError))
  }
  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }
}
```

Step 3: Create Component

Create "**FakeProductComponent**" component, now we code in "**fake-product.component.ts**" file as follows:

```
import { Component, OnInit } from '@angular/core';
import { FakeProductService } from '../fake-product.service';
@Component({
  selector: 'app-fake-product',
  templateUrl: './fake-product.component.html',
  styleUrls: ['./fake-product.component.css']
})
export class FakeProductComponent {
  data:any
  errMessage:string=''
  constructor(_service:FakeProductService){
    _service.getFakeProductData().subscribe({
      next:(data)=>{ this.data=data},
      error:(err)=>{
        this.errMessage=err
      }
    })
  }
}
```

Edit the HTML in the file "fake-product.component.html":

```
<p>binding-style works!</p>
<table>
    <thead>
        <tr>
            <th>Id</th>
            <th>title</th>
            <th>price</th>
            <th>description</th>
            <th>category</th>
            <th>image</th>
            <th>rating</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of data">
            <td>{{item.id}}</td>
            <td>{{item.title}}</td>
            <td>{{item.price}}</td>
            <td>{{item.description}}</td>
            <td>{{item.category}}</td>
            <td><img src="{{item.image}}" width="250" height="250" ></td>
            <td>{{item.rating.rate}}</td>
        </tr>
    </tbody>
</table>
```

Step 4: Process CORS Policy

The declaration and configuration are similar to the API call of Dong A Bank. We edit (or create new) "**proxy.conf.json**". As explained in the previous post, because in a project can connect many APIs, the content of this file is as follows:

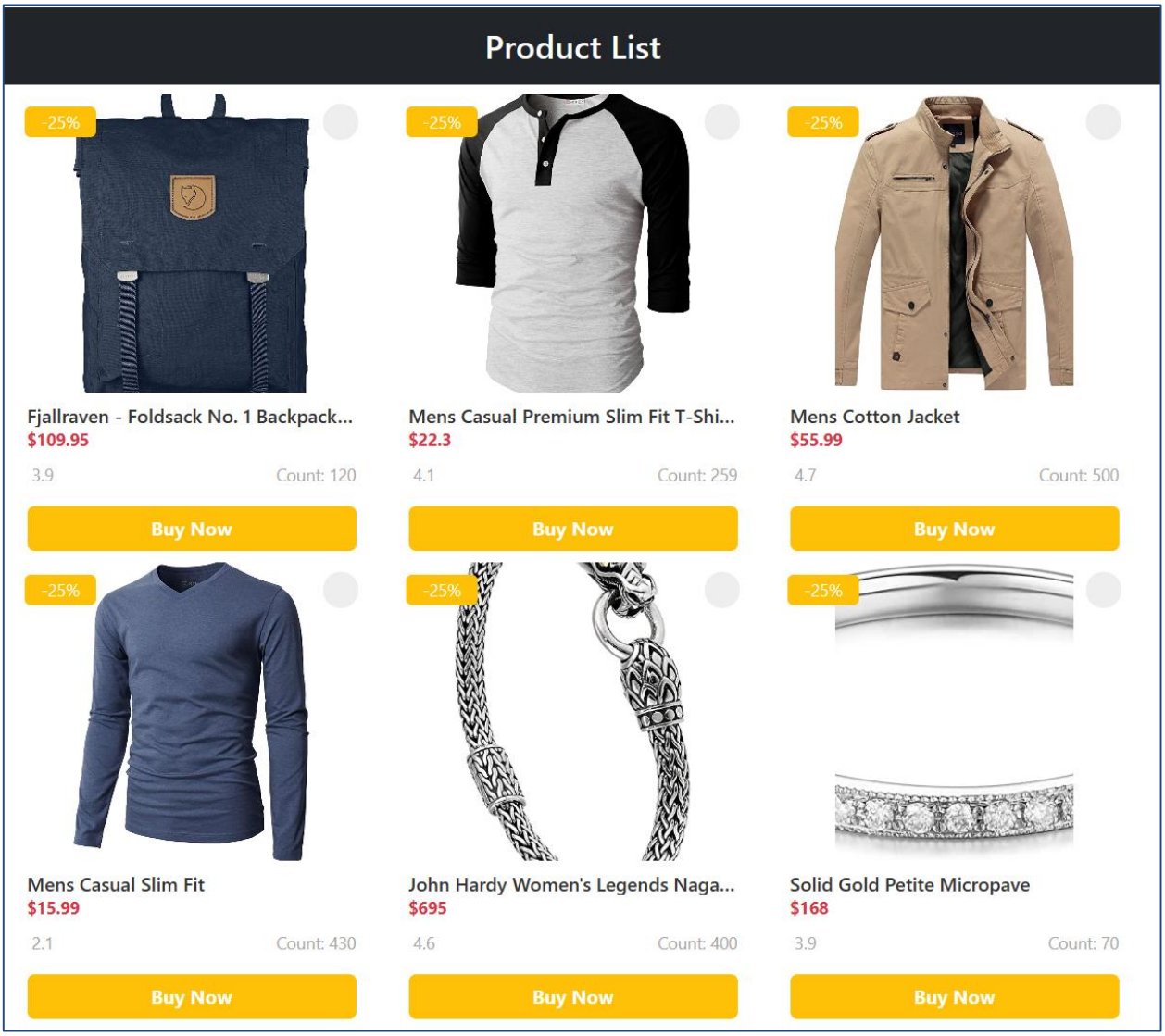
```
[
    {
      "context": ["/exchange"],
      "target": "https://www.dongabank.com.vn",
      "secure": true,
      "changeOrigin":true,
      "logLevel":"debug"
    },
    {
        "context": ["/products"],
        "target": "https://fakestoreapi.com",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
     }
 ]
```

Run the application and get the desired result.

Exercise 102: Interaction External Restful API- Product Service 2

Requirement:

Continue to improve the post https://fakestoreapi.com/products above. Specifically calibrate the resulting interface as shown below:



Detailed instructions:

We do all the same steps as in the previous tutorial. Just add 2 main tasks:

- The first is to install BootStrap JQuery (review step 5 of the previous exercise)
- Then edit "**fake-product.component.html**"

```
<p>fake-product works!</p>
<p>{{errMessage}}</p>
<h3 class="text-center text-white bg-dark p-3">Product List</h3>
<div class="container">
    <div class="row">
```

```
        <div class="col-md-4" *ngFor="let p of data">
            <div class="card">
                <div class="img-container">
                    <div class="d-flex justify-content-between align-items-center
p-2 first">
                        <span class="percent">-25%</span>
                        <span class="wishlist"><i class="bi bi-heart-
fill"></i></span>
                    </div>
                    <div class="text-center">
                        <img src="{{p.image}}" class="img-scale" >
                    </div>
                </div>
                <div class="product-detail-container">
                    <div class="d-flex justify-content-between align-items-
center">
                        <h6 class="mb-0 text-truncate">{{p.title}}</h6>
                    </div>
                    <div>
                        <span class="text-danger fw-bold">${{p.price}}</span>
                    </div>
                    <div class="d-flex justify-content-between align-items-center
mt-2">
                        <div class="ratings"><i class="bi bi-star-fill me-1"></i>
<span>{{p.rating.rate}}</span> </div>
                        <div class="count">Count:
{{p.rating.count}}</div>
                    </div>
                    <div class="mt-3"> <button class="btn btn-warning d-block w-
100 text-white fw-bold">Buy Now</button> </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

File CSS “fake-product.component.css”:

```
@import
url("https://fonts.googleapis.com/css2?family=Poppins:wght@100;200;300;400;500;600;700;800&dis
play=swap");
body {
    background-color: #d6d2d2;
    font-family: "Poppins", sans-serif;
    font-weight: 300
}

.card {
    border: none;
    border-radius: 10px
}

.card:hover {
    box-shadow: 5px 6px 6px 2px #cccecf;
    /* transform: scale(1.1) */
}
```

```
.percent {
    padding: 5px;
    background-color: #ffc107;
    border-radius: 5px;
    color: #fff;
    font-size: 14px;
    height: 25px;
    width: 60px;
    display: flex;
    justify-content: center;
    align-items: center;
    cursor: pointer
}

.wishlist {
    height: 30px;
    width: 30px;
    display: flex;
    justify-content: center;
    align-items: center;
    border-radius: 50%;
    background-color: #eee;
    padding: 10px;
    cursor: pointer
}

.img-container {
    position: relative
}

.img-container .first {
    position: absolute;
    width: 100%
}

.img-container img {
    border-top-left-radius: 5px;
    border-top-right-radius: 5px
}

.img-scale {
    width: 200px;
    height: 250px;
    object-fit: cover
}

.product-detail-container {
    padding: 10px
}

.ratings i {
    color: #a9a6a6
}

.ratings span {
    color: #a9a6a6
}

.count {
    color: #a9a6a6;
}

label.radio {
    cursor: pointer
}

label.radio input {
    position: absolute;
    top: 0;
```

```
    left: 0;
    visibility: hidden;
    pointer-events: none
}
label.radio span {
    height: 25px;
    width: 25px;
    display: flex;
    justify-content: center;
    align-items: center;
    border: 2px solid #dc3545;
    color: #dc3545;
    font-size: 10px;
    border-radius: 50%;
    text-transform: uppercase
}
label.radio input:checked+span {
    border-color: #dc3545;
    background-color: #dc3545;
    color: #fff
}
```

Run the application and get the desired result.

Exercise 103: *Coindesk API- Bitcoin Price Index in Real-time*

Requirement:

Use AngularJS to display real-time Bitcoin Price Index (BPI) according to the following public API:

https://api.coindesk.com/v1/bpi/currentprice.json


```
{"time":{"updated":"Mar 12, 2023 10:43:00 UTC","updatedISO":"2023-03-
12T10:43:00+00:00","updateduk":"Mar 12, 2023 at 10:43
GMT"},"disclaimer":"This data was produced from the CoinDesk Bitcoin Price
Index (USD). Non-USD currency data converted using hourly conversion rate
from
openexchangerates.org","chartName":"Bitcoin","bpi":{"USD":{"code":"USD","symb
ol":"&#36;","rate":"20,520.4346","description":"United States
Dollar","rate_float":20520.4346},"GBP":{"code":"GBP","symbol":"&pound;","rate
":"17,146.7110","description":"British Pound
Sterling","rate_float":17146.711},"EUR":{"code":"EUR","symbol":"&euro;","rate
":"19,989.8993","description":"Euro","rate_float":19989.8993}}}
```


Instructions:

Exercise 104: *Get List of Public APIs*

Requirement:

Write a program to display the list of public APIs at the following link https://api.publicapis.org/entries, layout the appropriate interface.

Instructions:

Exercise 105: *Predict the gender of a person based on their name - API.*

Requirement:

Design website to use public API:

https://api.genderize.io/?name=hoa

This API receive names and predicts gender. The interface needs a name input box and a gender check button.

Instructions:

Exercise 106: *Get US public data API*

Requirement:

For public US API:

https://datausa.io/api/data?drilldowns=Nation&measures=Population

Write a program to display data on the interface, request the creation of a model structure and appropriate interface layout.

Instructions:

Exercise 107: *Random dog images API*

Requirement:

For public API: https://dog.ceo/api/breeds/image/random

This API will randomly display data that is the image link of a Dog, write a program to display this random image on the interface (each time the display button is clicked, call the image in this random API).


```
{"message":"https:\/\/images.dog.ceo\/breeds\/poodle-
miniature\/n02113712_9682.jpg","status":"success"}
```


Instructions:

Module 11: Restful API with NodeJS and ExpressJS

Practical knowledge content:

+ Practice Model-View-Controller (MVC) model
+ Install and use ExpressJS
+ Configure Webserver
+ Create Restful APIs: GET, POST, PUT, DELETE
+ Use Postman to test Restful API

Exercise 108: Explain in detail the operation model of MVC

Requirement:

Explain in detail the operation model of MVC, the meaning of each component.

Exercise 109: API and Restful API

Requirement:

What is API? Explain the meaning of Restful API

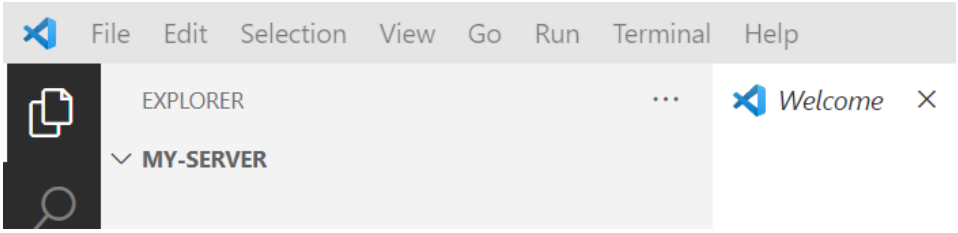
Exercise 110: Create and execute Project Restful API

Requirement:

Create a Project named "my-server" to run on Web Server platform, using NodeJS and ExpressJS. Illustration of default API outputting "Hello Restful API" message

Instructions:

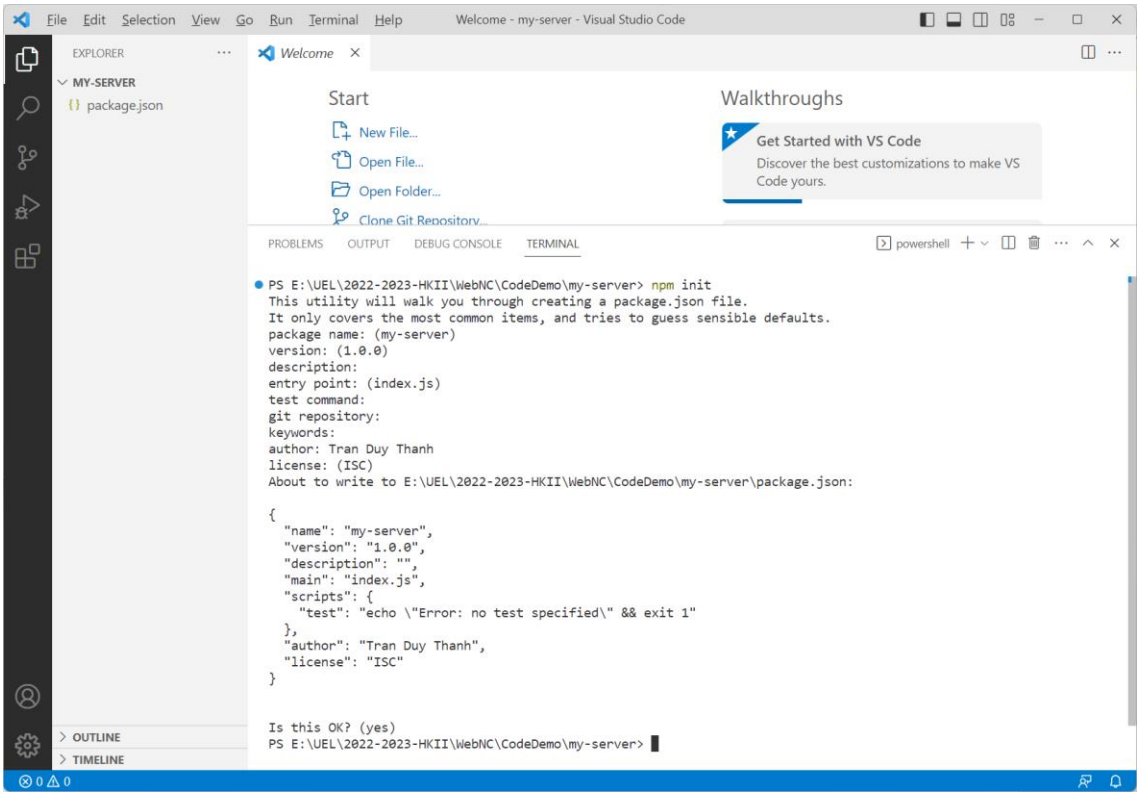
Step 1: Create a folder "**my-server**" and use Visual Studio Code to open project:



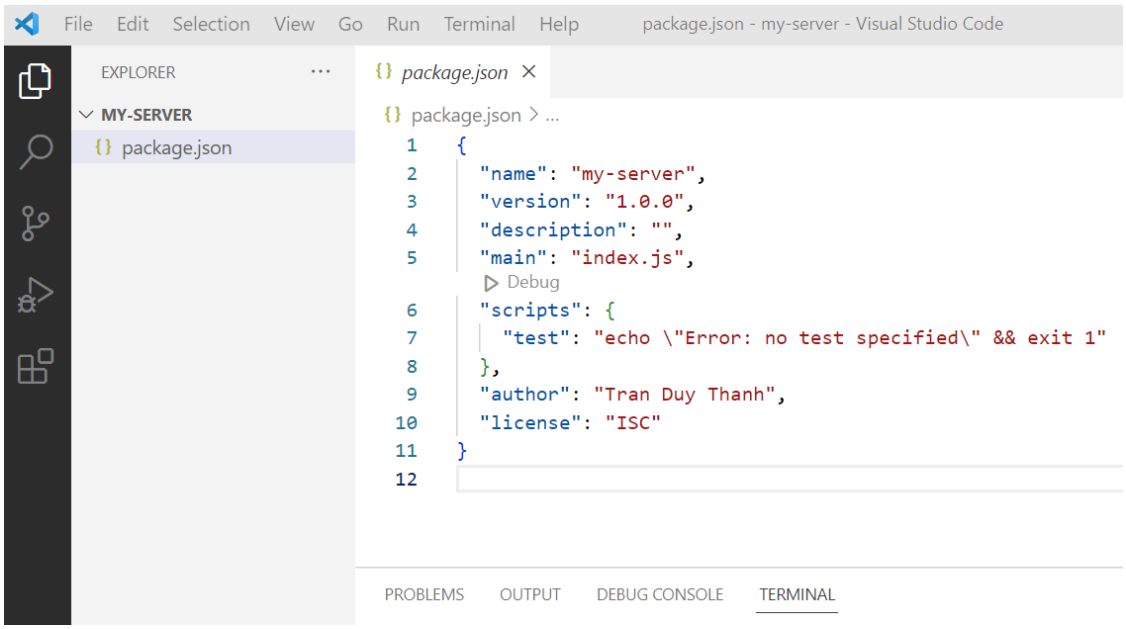
Step 2: Then use terminal in Visual Studio Code or command line prompt to write commands, this tutorial uses Terminal:

npm init

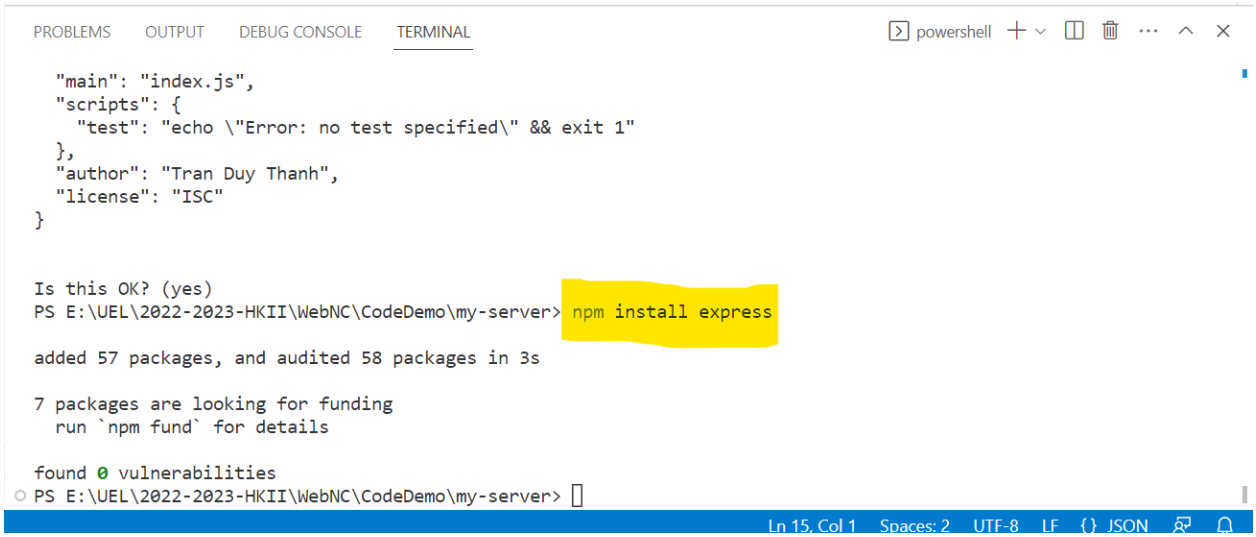
Illustrate the process of running this code:



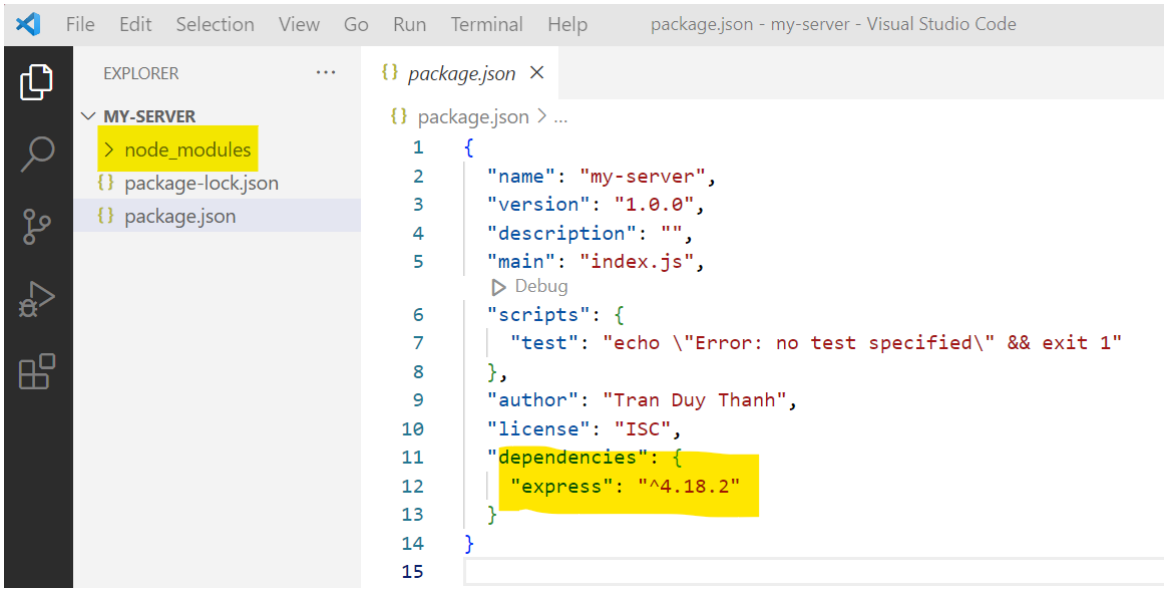
After typing **npm init**, press Enter repeatedly (you can provide information at each command prompt). Once done it will generate a file **package.json**.



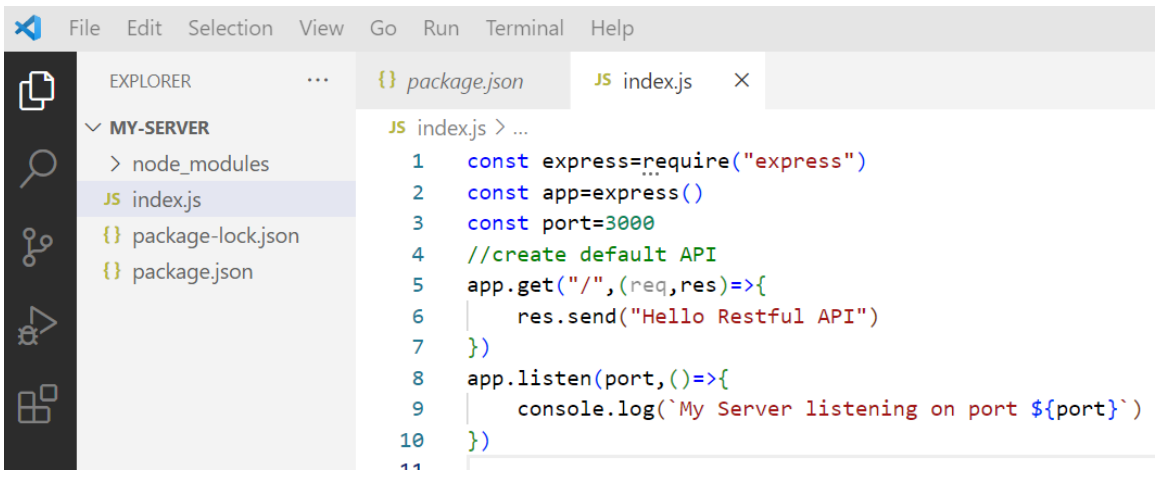
Step 3: execute command: **npm install express** to install ExpressJS libraries



Once done, express library is installed into Project:



Step 4: Create a file "**index.js**" to create default API as required, coding as below:



The code above runs the Web service (Restful API) on port **3000**, with the default API outputting the "Hello Restful API" data when it is invoked.

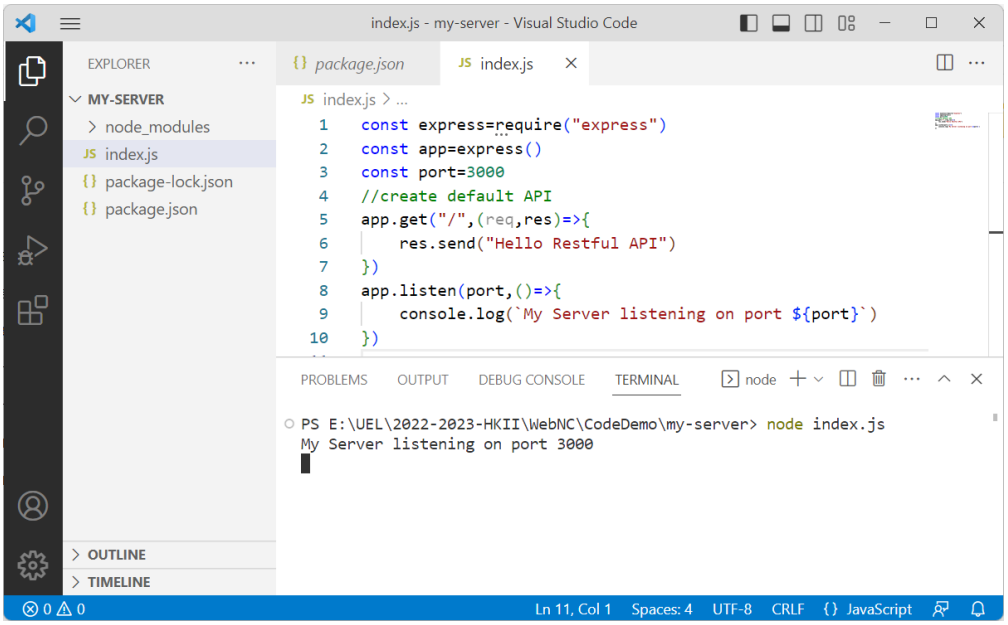
req (the variable name representing the Request), is the object to send data from the Client to the Server.

res (the variable name representing Response), is the object to send data from the Server to the Client.

The above code writes: **res.send("Hello Restful API")** ie when calling this web service by default, it returns the string "Hello Restful API".

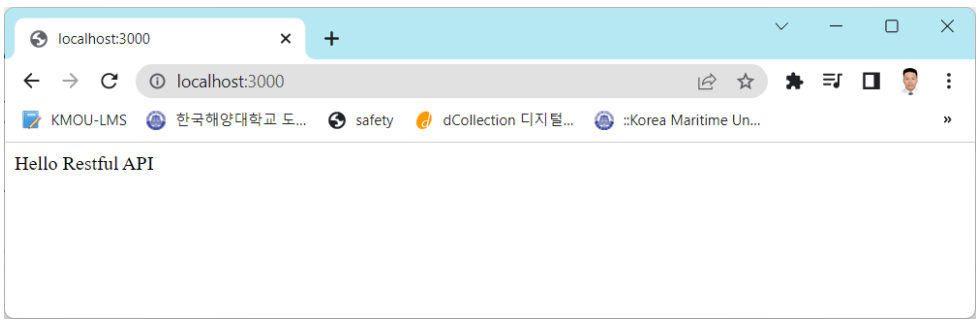
Step 5: Launch Web Server:

call command **node index.js**



At this time, the listen function will be executed, if the Web Server is successfully initialized, the command on line 9 will execute, specifically here we see the message "My Server listening on port 3000".

Open a browser to see the results:



Exercise 111: Install nodemon for Web Server Project

Requirement:

Usually we have to change the code continuously in the project, every time we change, we have to rerun the Web server for this project, leading to time-consuming and unnecessary. The nodemon technique helps to restart the server when changing the code in the project (**node-mon /nəʊd mʌn/**).

Instructions:

Step 1: Execute the command

npm i --save-dev nodemon

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server> npm i --save-dev nodemon

added 32 packages, and audited 92 packages in 3s

10 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server>
```

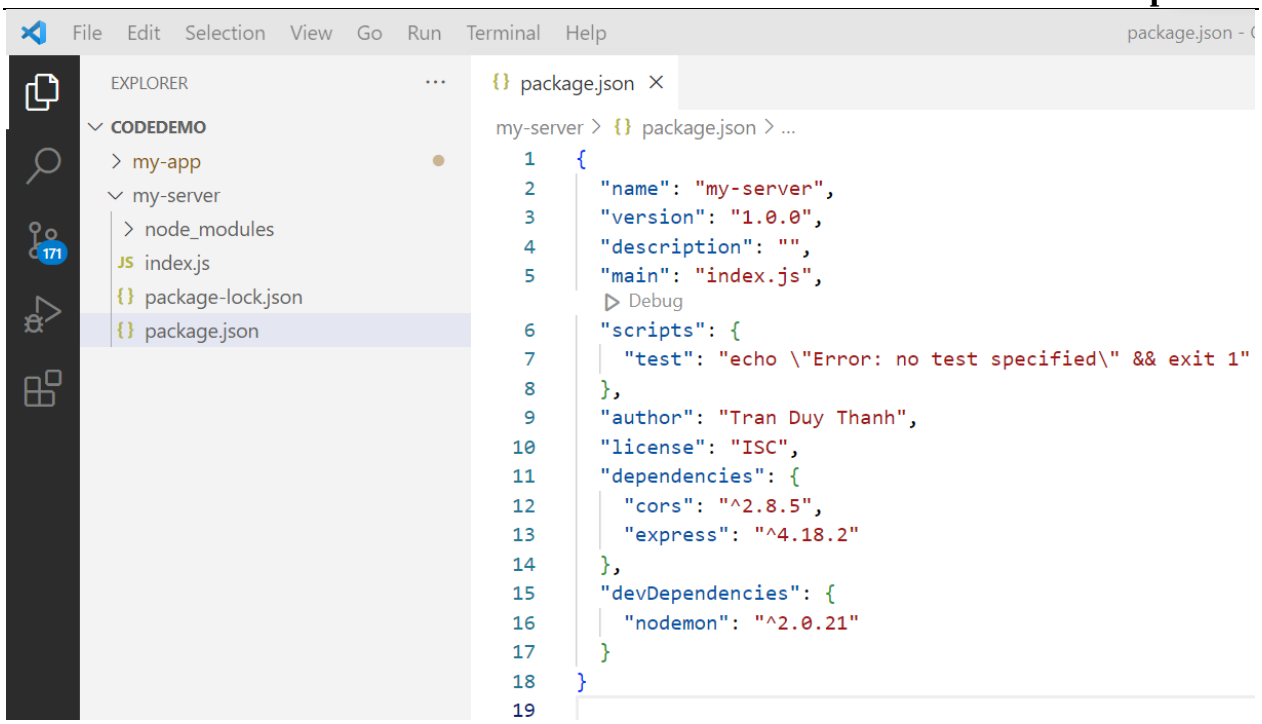
Step 2: Execute the command

npm i -g nodemon

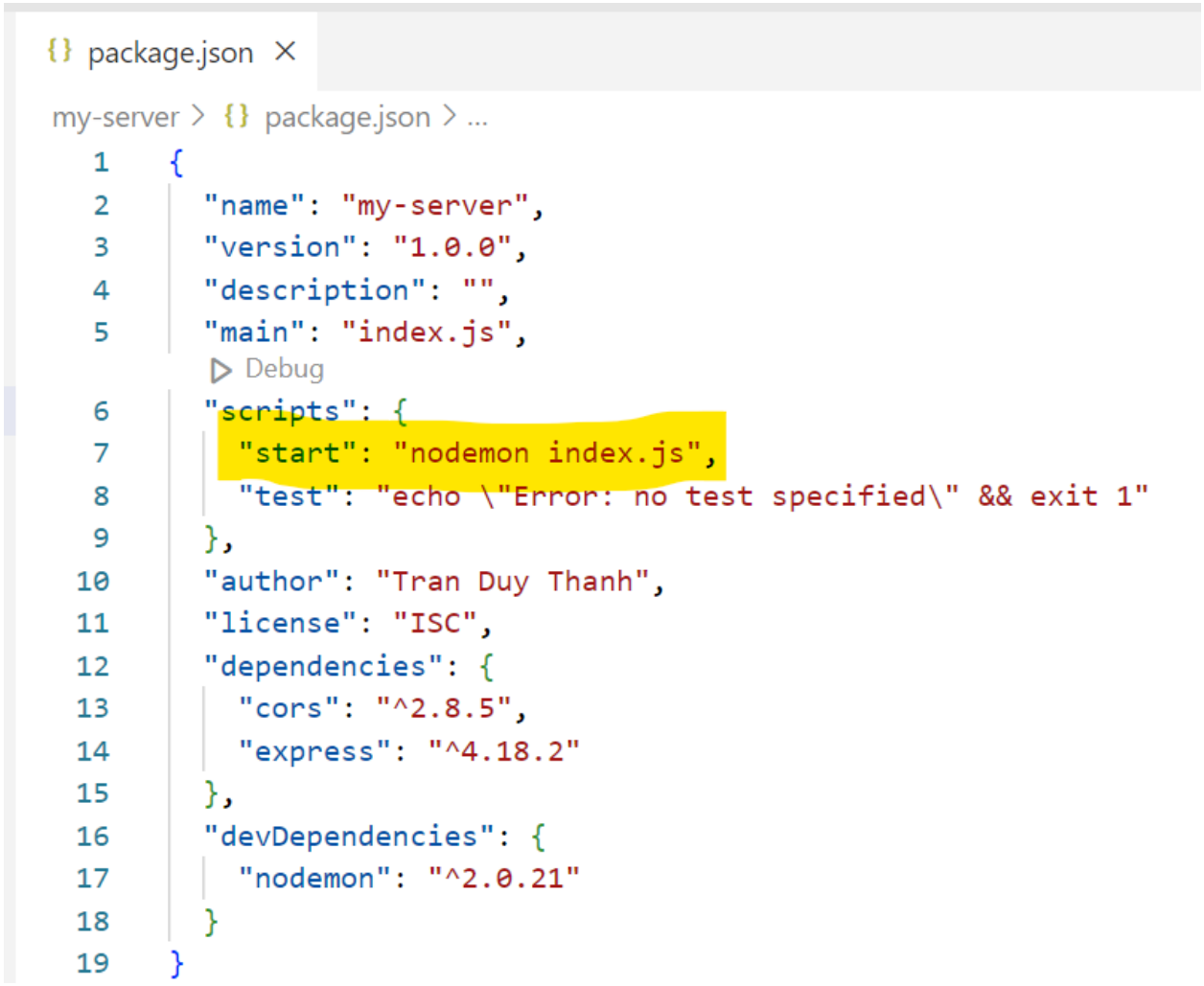
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server> npm i -g nodemon

changed 32 packages in 284ms

3 packages are looking for funding
  run `npm fund` for details
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server>
```

Step 3: Modify "package.json"

After installing the nodemon commands, the file "**package.json**" will look like the screenshot below:



Put more command “start”: “nodemon index.js” into “scripts” tag



Step 4: Re-execute the Web Server run command for this API.

Instead of writing

node index.js

Then we write:

npm start

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

[nodemon] 2.0.21
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
My Server listening on port 3000
```

So the **Server** has been started at **port 3000** with nodemon mechanism to automatically restart when the source code changes, for example if we change any source code:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

[nodemon] 2.0.21
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
My Server listening on port 3000
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
My Server listening on port 3000
```

Exercise 112: Install morgan for Web Server Project

Requirement:

Also we can use **morgan** command to view http request logger

Instructions:

Step 1: Execute command

npm i --save-dev morgan

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server> npm i --save-dev morgan

added 5 packages, and audited 97 packages in 4s

10 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server>
```

File "**package.json**" will be updated:

package.json | index.js

my-server > package.json > ...

```
{
  "name": "my-server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  Debug
  "scripts": {
    "start": "nodemon index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Tran Duy Thanh",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.18.2"
  },
  "devDependencies": {
    "morgan": "^1.10.0",
    "nodemon": "^2.0.21"
  }
}
```

Step 2: Modify **index.js** to use **morgan**

```
const express=require("express")
const app=express()
const port=3000
const morgan=require("morgan")
```

```
app.use(morgan("combined"))
//create default API
app.get("/",(req,res)=>{
    res.send("Hello Restful API")
})
app.listen(port,()=>{
    console.log(`My Server listening on port ${port}`)
})
```

Step 3: Launch the **Server** and we have the **http request logger** output as shown below:

Exercise 113: *Create HTTP GET – List of Book*

Requirement:

In the **index.js** file, continue to create a list of Books, write an API to get this list.

Instructions:

Suppose in the file **index.js** we add the declaration of the list of Books as below:

```
let  database=[
    {"BookId":"b1","BookName":"Kỹ thuật lập trình cơ bản",
"Price":70,"Image":"b1.png"},
    {"BookId":"b2","BookName":"Kỹ thuật lập trình nâng cao",
"Price":100,"Image":"b2.png"},
    {"BookId":"b3","BookName":"Máy học cơ bản","Price":200,"Image":"b3.png"},
    {"BookId":"b4","BookName":"Máy học nâng cao","Price":300,"Image":"b4.png"},
    {"BookId":"b5","BookName":"Lập trình Robot cơ bản","Price":250,"Image":"b5.png"},
  ]
```

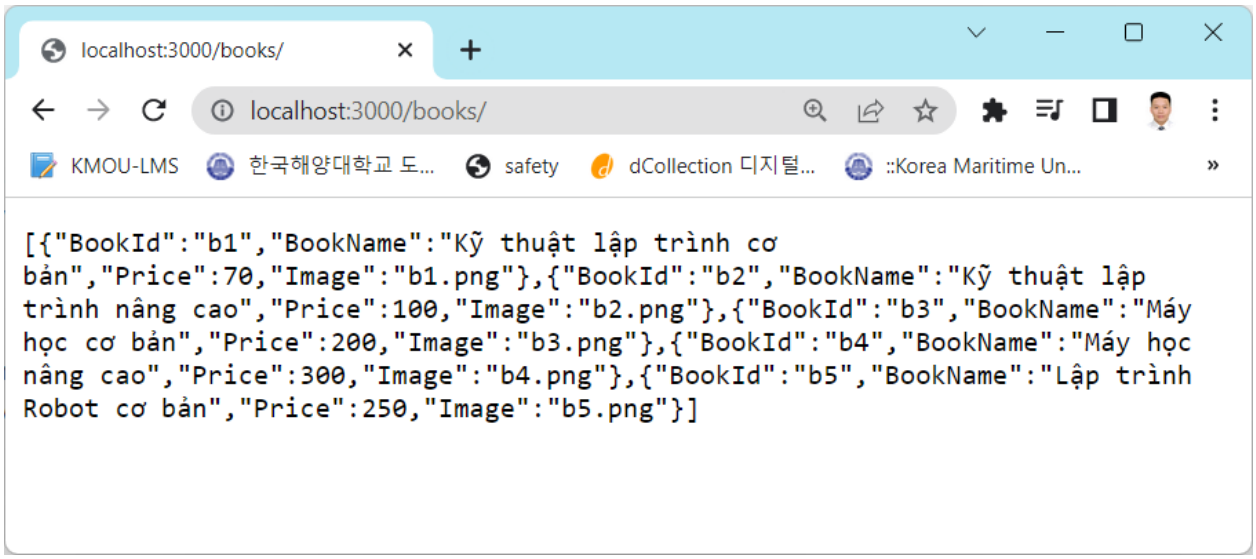
To create an API to get a list of Books, we usually declare the pattern "**/books**" as follows:

```
app.get("/books",(req,res)=>{
    res.send(database)
})
```

The full code of the program (**index.js**) after adding this API:

```
const express=require("express")
const app=express()
const port=3000
const morgan=require("morgan")
app.use(morgan("combined"))

app.get("/",(req,res)=>{
    res.send("Hello Restful API")
})
app.listen(port,()=>{
    console.log(`My Server listening on port ${port}`)
})
let  database=[
    {"BookId":"b1","BookName":"Kỹ thuật lập trình cơ
bản","Price":70,"Image":"b1.png"},
    {"BookId":"b2","BookName":"Kỹ thuật lập trình nâng
cao","Price":100,"Image":"b2.png"},
    {"BookId":"b3","BookName":"Máy học cơ bản","Price":200,"Image":"b3.png"},
    {"BookId":"b4","BookName":"Máy học nâng cao","Price":300,"Image":"b4.png"},
    {"BookId":"b5","BookName":"Lập trình Robot cơ
bản","Price":250,"Image":"b5.png"},
  ]
app.get("/books",(req,res)=>{
    res.send(database)
})
```

Proceed to run the command again: **node index.js** (or if you have run **npm start** before, the system will automatically restart, we don't have to run it again) and call the api books we will have the result:



Exercise 114: Invoke HTTP GET – List of Book

Requirement:

From the **books API** created above. Write a program that invokes this API in AngularJS.

Assume that in **assets/book_images** there are 5 images with the same name as in the server returned. HTML Interface after calling the API books:

BookId	BookName	Price	Image
b1	Kỹ thuật lập trình cơ bản	70	
b2	Kỹ thuật lập trình nâng cao	100	
b3	Máy học cơ bản	200	
b4	Máy học nâng cao	300	
b5	Lập trình Robot cơ bản	250	

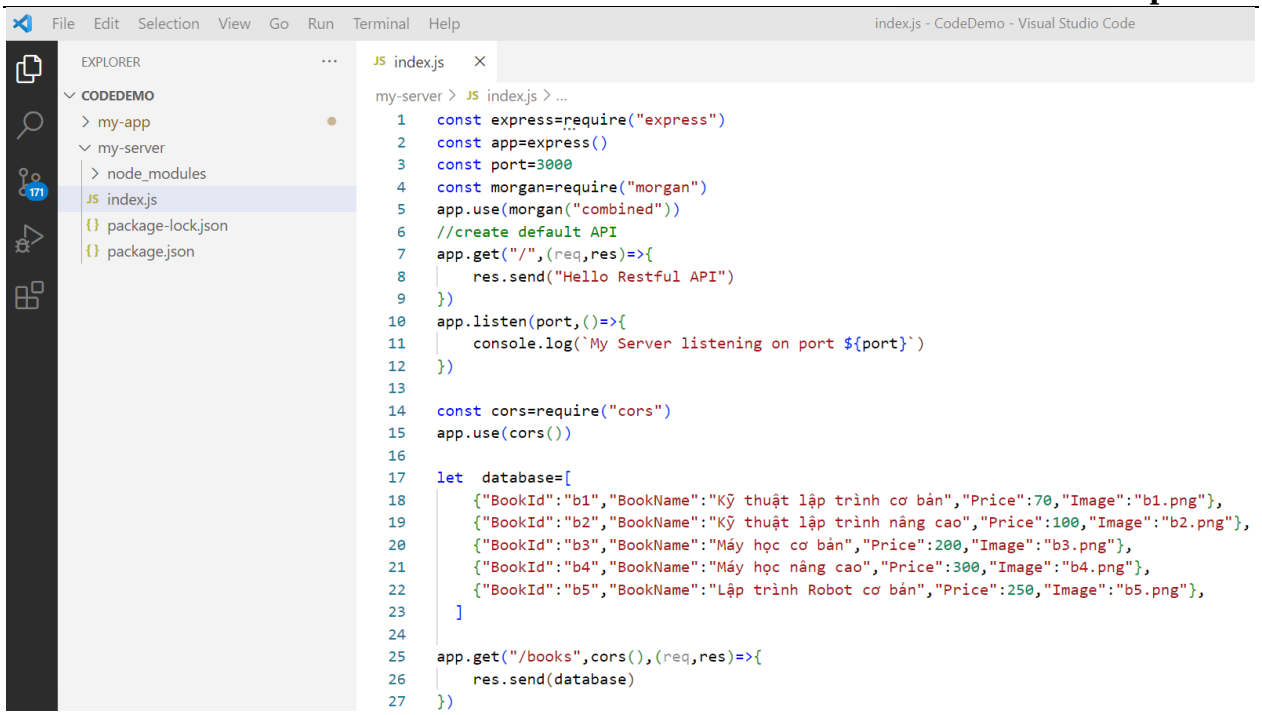
Instructions:

*In the project "**my-server**". Normally, we will encounter a CORS Policy error, so on the Server we add the following commands:

Step 1: Install the command cors(Cross-Origin Resource Sharing) for Web server that has api books.

npm i cors

Step 2: Add command related to **cors**, last code of **index.j**s:



Command lines 14, 15 declare the cors library and enable cors

Command line 25 adds cors() to the books API

* In the project "**my-app**":

Step 1: Create **IBook** interface in **Book.ts** file

```
export interface IBook{
    BookId:string,
    BookName:string,
    Price:number,
    Image:string
}
```

Step 2: Create a service named "BookAPI", in the file "book-api.service.ts" write the following commands:

```
import { HttpClient, HttpErrorResponse, HttpHeaders } from
'@angular/common/http';
import { Injectable } from '@angular/core';
import { catchError, map, Observable, retry, throwError } from 'rxjs';
import { IBook } from './interfaces/Book';

@Injectable({
  providedIn: 'root'
})
export class BookAPIService {

  constructor(private _http: HttpClient) { }

  getBooks():Observable<any>
```

```
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-
8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/books",requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }

  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }
}
```

Step 3: Configure "**proxy.conf.json**", continue adding at the end of the file:

```
[
    {
      "context": ["/exchange"],
      "target": "https://www.dongabank.com.vn",
      "secure": true,
      "changeOrigin":true,
      "logLevel":"debug"
    },
    {
        "context": ["/products"],
        "target": "https://fakestoreapi.com",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    },
    {
        "context": ["/books"],
        "target": "http://localhost:3000",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    }
 ]
```

Step 4: Create a "**Books**" component.

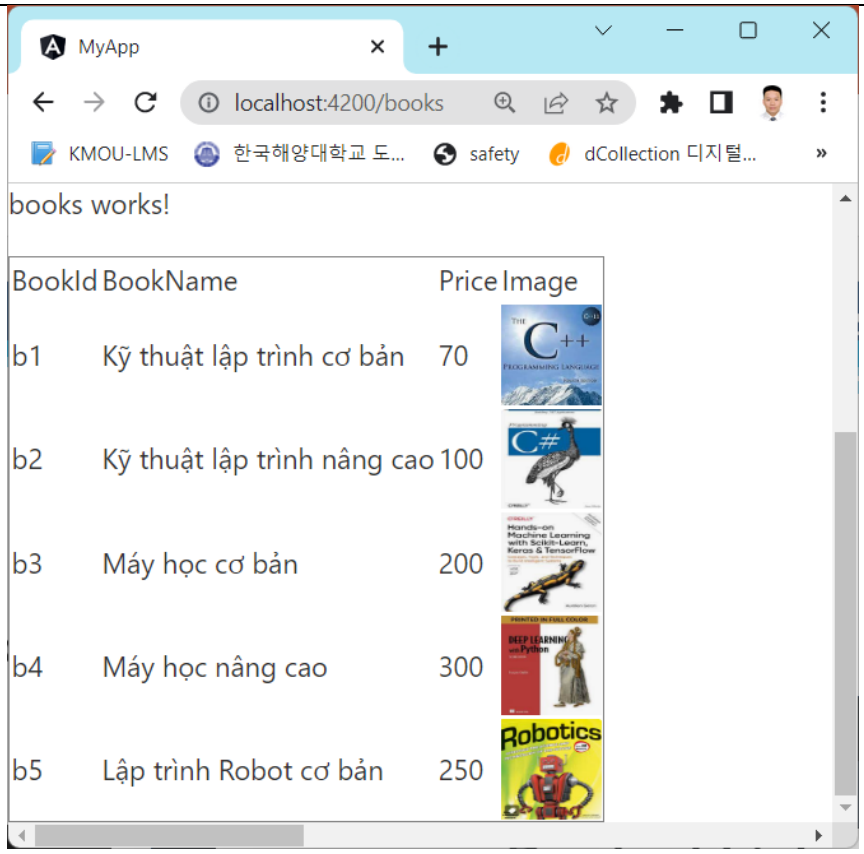
In the file "**books.component.ts**" add the command:

```
import { Component } from '@angular/core';
import { BookAPIService } from '../book-api.service';

@Component({
  selector: 'app-books',
  templateUrl: './books.component.html',
  styleUrls: ['./books.component.css']
})
export class BooksComponent {
  books:any;
  errMessage:string=''
  constructor(private _service: BookAPIService){
    this._service.getBooks().subscribe({
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

In the file "**books.component.html**" add the command:

```
 <p>books works!</p>
{{errMessage}}
<table border="1">
    <tr>
        <td>BookId</td>
        <td>BookName</td>
        <td>Price</td>
        <td>Image</td>
    </tr>
    <tbody>
        <tr  *ngFor="let book of books">
            <td>{{book.BookId}}</td>
            <td>{{book.BookName}}</td>
            <td>{{book.Price}} </td>
            <td>
                <img src="assets/book_images/{{book.Image}}" width="50px"
height="50px">
            </td>
        </tr>
    </tbody>
</table>
```

Step 5: Declare Routing or refer to it directly in "**app.component.html**", run the application "**my-app**" we have the results:



So the data in the books API built in "**my-server**" has been successfully loaded to the front end Angularjs "**my-app**".

Exercise 115: Create HTTP GET – a Book

Requirement:

Build an API to get the details of a Book when the BookId is known. For example:

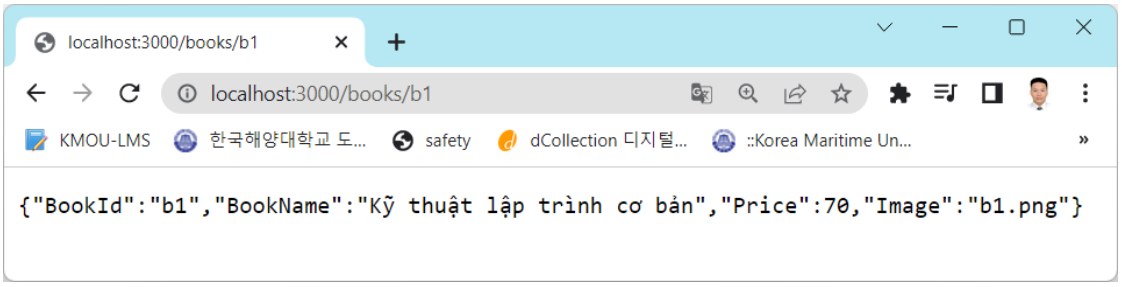
http://localhost:3000/books/b1

This API will return details of Book with BookId=b1

http://localhost:3000/books/b2

This API will return details of Book with BookId=b2

for example, when entering b1, the JsonObject of Book will give the result:



Instructions:

In "**index.js**" file, add the following command at the end of the file:

```
app.get("/books/:id",cors(),(req,res)=>{
    id=req.params["id"]
    let p=database.find(x=>x.BookId==id)
    res.send(p)
})
```

Full code:

```
const express=require("express")
const app=express()
const port=3000
const morgan=require("morgan")
app.use(morgan("combined"))
//create default API
app.get("/",(req,res)=>{
    res.send("Hello Restful API")
})
app.listen(port,()=>{
    console.log(`My Server listening on port ${port}`)
})

const cors=require("cors")
app.use(cors())

let  database=[
    {"BookId":"b1","BookName":"Kỹ thuật lập trình cơ
bản","Price":70,"Image":"b1.png"},
    {"BookId":"b2","BookName":"Kỹ thuật lập trình nâng
cao","Price":100,"Image":"b2.png"},
    {"BookId":"b3","BookName":"Máy học cơ bản","Price":200,"Image":"b3.png"},
    {"BookId":"b4","BookName":"Máy học nâng cao","Price":300,"Image":"b4.png"},
    {"BookId":"b5","BookName":"Lập trình Robot cơ
bản","Price":250,"Image":"b5.png"},
  ]

app.get("/books",cors(),(req,res)=>{
    res.send(database)
})
app.get("/books/:id",cors(),(req,res)=>{
    id=req.params["id"]
    let p=database.find(x=>x.BookId==id)
    res.send(p)
})
```

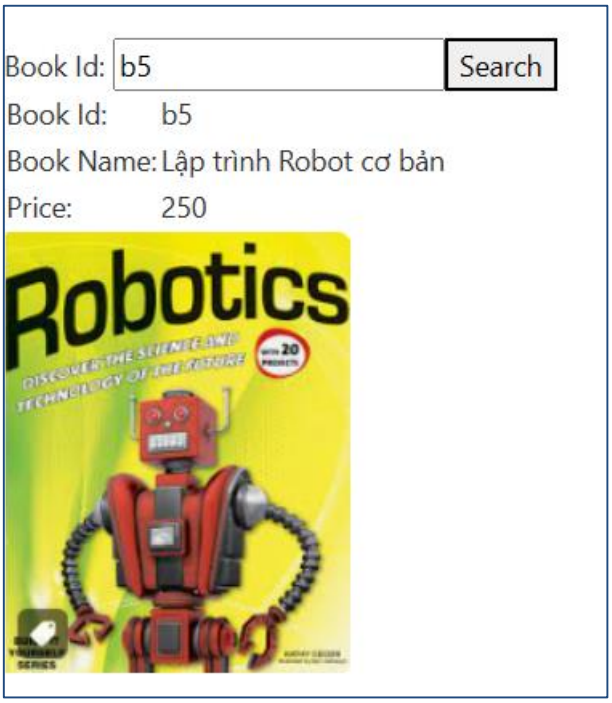
Attention for the syntax:

```
"/books/:id"
```

Exercise 116: Invoke HTTP GET – a Book

Requirement:

Build an interface inside AngularJS, the software allows users to enter BookId to search for details of this Book by calling the Get a Book API in the previous exercise.



Instructions:

Step 1: Open the file "**book-api.service.ts**" again to edit the source code for the "BookAPIService" service class.

```
getBook(bookId:string):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/books/"+bookId,requestOptions).pipe(
        map(res=>JSON.parse(res) as IBook),
        retry(3),
        catchError(this.handleError))
  }
```

Full code of this Service after update function getBook(bookId:string)

```
import { HttpClient, HttpErrorResponse, HttpHeaders } from
'@angular/common/http';
import { Injectable } from '@angular/core';
```

```
import { catchError, map, Observable, retry, throwError } from 'rxjs';
import { IBook } from './interfaces/Book';

@Injectable({
  providedIn: 'root'
})
export class BookAPIService {

  constructor(private _http: HttpClient) { }

  getBooks():Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-
8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/books",requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }

  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }

  getBook(bookId:string):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-
8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/books/"+bookId,requestOptions).pipe(
        map(res=>JSON.parse(res) as IBook),
        retry(3),
        catchError(this.handleError))
  }
}
```

Step 2: Create component "**BookDetail**" to display detailed information of Book. Adding code to File "**book-detail.component.ts**":

```
import { Component } from '@angular/core';
import { BookAPIService } from '../book-api.service';

@Component({
  selector: 'app-book-detail',
  templateUrl: './book-detail.component.html',
  styleUrls: ['./book-detail.component.css']
})
export class BookDetailComponent {
  book:any;
  errMessage:string=''
  constructor(private _service: BookAPIService){
  }
  searchBook(bookId:string)
  {
    this._service.getBook(bookId).subscribe({
      next:(data)=>{this.book=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

Add code for File "**book-detail.component.html**":

```
<p>book-detail works!</p>
<p>{{errMessage}}</p>
Book Id: <input type="text" #bookId> <button
(click)="searchBook(bookId.value)">Search</button>
<table>
    <tr>
        <td>Book Id:</td>
        <td>{{book.BookId}}</td>
    </tr>
    <tr>
        <td>Book Name:</td>
        <td>{{book.BookName}}</td>
    </tr>
    <tr>
        <td>Price:</td>
        <td>{{book.Price}} </td>
    </tr>
    <tr>
        <td colspan="2">
            <img src="assets/book_images/{{book.Image}}">
        </td>
    </tr>
</table>
```

Declare the routing or reference this component in "**app.component.html**" for testing. Run the application, we have the required result.

Exercise 117: Tạo HTTP POST – Create a Book

Requirement:

Write more API to add a new Book.

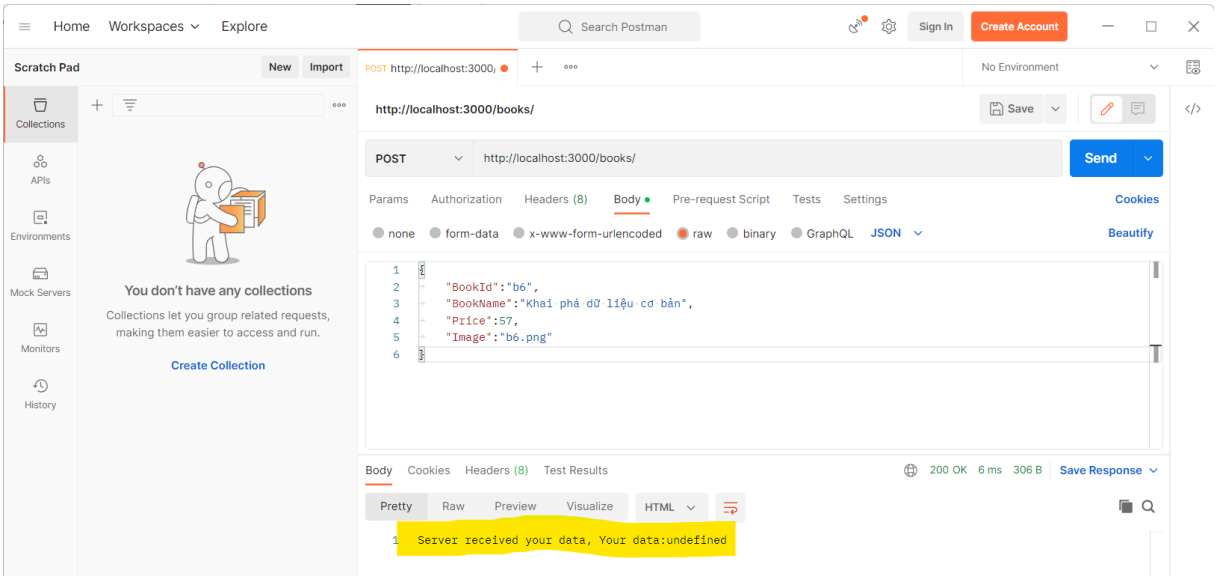
Instructions:

Step 1: In the project "my-server", continue to add the following code at the end of the file "index.js"

```
app.post("/books",cors(),(req,res)=>{
    console.log(req.body)
    res.send("Server received your data, Your data:"+req.body)
})
```

Step 2: Install Postman and use it to test methods: POST, PUT, DELETE (because only GET can be tested in the browser). Note that if the Postman test is successful, then the API commands in "my-server" are correctly. This tool is very convenient, it helps us to test the API before using it. There is also the Swagger tool (https://swagger.io). In the content of this course we will use Postman. Download Postman here: https://www.postman.com/downloads

Test api books with POST method, configure as below, press **send** button.



Apparently the API books (method POST) call was successful, but the body was not retrieved (undefined).

Step 3: Install body-parser for "**my-server**"

npm i body-parser

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server> npm i body-parser

added 2 packages, changed 2 packages, and audited 99 packages in 1s

10 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server>
```

After installing the library, proceed to edit **index.js**:

```
const bodyParser=require("body-parser")
app.use(bodyParser.json())
```

Step 4: Edit the API Post for a book

```
app.post("/books",cors(),(req,res)=>{
    //put json book into database
    database.push(req.body);
    //send message to client(send all database to client)
    res.send(database)
})
```

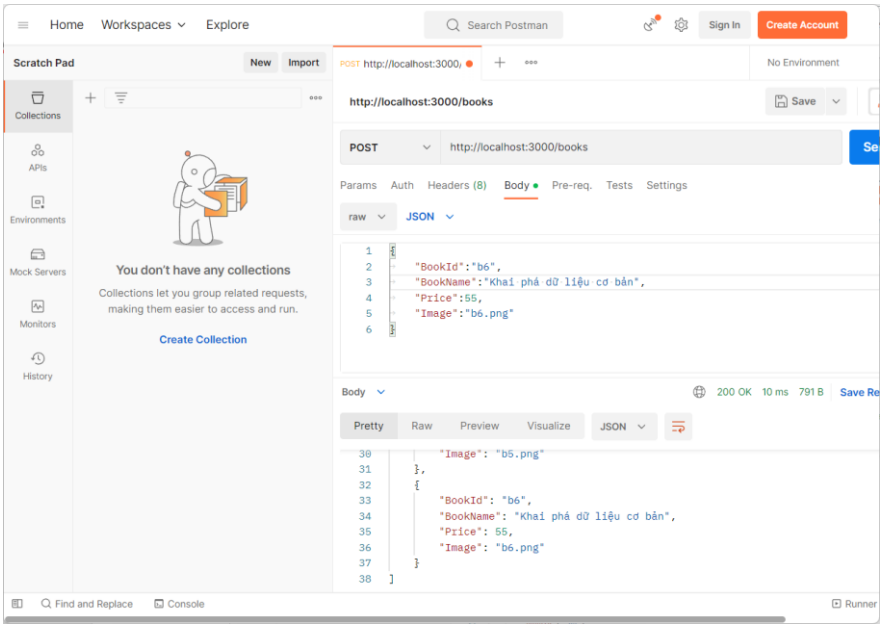
With the above command, every time POST 1 Book to the server, it will be saved to the database. We can then call GET books to see all the latest data. This is an illustration of adding a new object to the Server, it will be lost when restarting the server. Therefore, we often have to save to the Database, so the program will use MongoDB to store, which will be guided in the following modules.

Full code of **index.js** after adding POST API 1 Book:

```
const express=require("express")
const app=express()
const port=3000
const morgan=require("morgan")
app.use(morgan("combined"))
const bodyParser=require("body-parser")
app.use(bodyParser.json())
//create default API
app.get("/",(req,res)=>{
    res.send("Hello Restful API")
})
app.listen(port,()=>{
    console.log(`My Server listening on port ${port}`)
})
```

```
const cors=require("cors")
app.use(cors())

let  database=[
    {"BookId":"b1","BookName":"Kỹ thuật lập trình cơ
bản","Price":70,"Image":"b1.png"},
    {"BookId":"b2","BookName":"Kỹ thuật lập trình nâng
cao","Price":100,"Image":"b2.png"},
    {"BookId":"b3","BookName":"Máy học cơ bản","Price":200,"Image":"b3.png"},
    {"BookId":"b4","BookName":"Máy học nâng cao","Price":300,"Image":"b4.png"},
    {"BookId":"b5","BookName":"Lập trình Robot cơ bản","Price":250,"Image":"b5.png"},
  ]
app.get("/books",cors(),(req,res)=>{
    res.send(database)
})
app.get("/books/:id",cors(),(req,res)=>{
    id=req.params["id"]
    let p=database.find(x=>x.BookId==id)
    res.send(p)
})
app.post("/books",cors(),(req,res)=>{
    //put json book into database
    database.push(req.body);
    //send message to client(send all database to client)
    res.send(database)
})
```

Step 5: Retest using Postman



Select **method** as **POST**, **body** tag select **raw/json** we enter structured data:

```
{
    "BookId":"b6",
    "BookName":"Khai phá dữ liệu cơ bản",
    "Price":55,
    "Image":"b6.png"
}
```

Then press the "Send" button, it will invoke the Post a Book API, if successful it will also return all the latest data (for us to test), of course we just need the software to return true/ false is enough. In this example the document returns the entire Book for easy viewing by students. Students can test again using the GET all Books API presented in the previous lesson. For example, just click again to view Books on the Web interface, it will reload the latest data:

books works!

BookId	BookName	Price	Image
b1	Kỹ thuật lập trình cơ bản	70	
b2	Kỹ thuật lập trình nâng cao	100	
b3	Máy học cơ bản	200	
b4	Máy học nâng cao	300	
b5	Lập trình Robot cơ bản	250	
b6	Khai phá dữ liệu cơ bản	55	

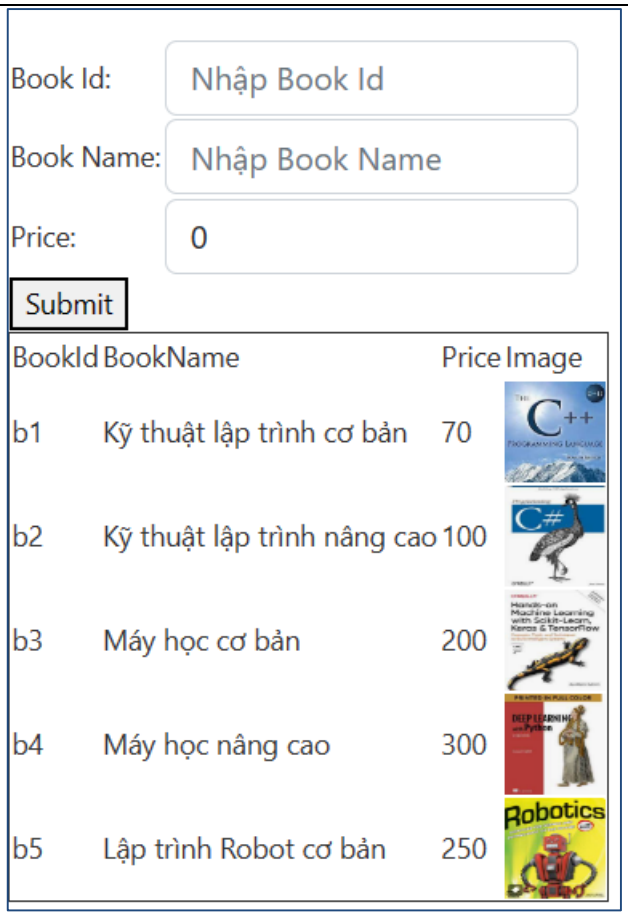
The data has been displayed for the book with code **{b6}**, the image we have not processed, through the MongoDB section we will process the image.

Exercise 118: *Invoke HTTP POST – Create a Book*

Requirement:

Design the Front End interface (**my-app**) to insert Book data and then call the API in the previous exercise to create a new Book:

- Default startup screen will download all Books in Server (Students can reuse **BooksComponent** or repeat API call to get all Books)
- Every time successfully submit 1 Book, the screen will update the list again.



Instructions:

Step 1: In the file "**Book.ts**", add the class **Book**:

```
export interface IBook{
    BookId:string,
    BookName:string,
    Price:number,
    Image:string
}
export class Book{
    constructor(
        public BookId:string="",
        public BookName:string="",
        public Price:number=0,
        public Image:string="")
    {}
}
```

Step 2: In the file "**book-api.service.ts**", Add the script for **BookAPIService**:

```
postBook(aBook:any):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-
Type","application/json;charset=utf-8")
```

```
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return
this._http.post<any>("/books",JSON.stringify(aBook),requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }
```

Because in "**my-server**" we receive json, so "**content-type**" is set to "**application/json**".

aBook is a model (Book) received from the interface, it needs to be converted to Json as String format by the JSON.stringify(aBook) command to send the Server command (HTTP POST API). Full Code of BookAPIService after modifying:

```
import { HttpClient, HttpErrorResponse, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { catchError, map, Observable, retry, throwError } from 'rxjs';
import { IBook } from './interfaces/Book';

@Injectable({
  providedIn: 'root'
})
export class BookAPIService {

  constructor(private _http: HttpClient) { }

  getBooks():Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/books",requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }

  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }

  getBook(bookId:string):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-8")
```

```
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/books/"+bookId,requestOptions).pipe(
        map(res=>JSON.parse(res) as IBook),
        retry(3),
        catchError(this.handleError))
  }
  postBook(aBook:any):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","application/json;charset=utf-
8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.post<any>("/books",JSON.stringify(aBook),requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }
}
```

Step 3: Create one more **BookNewComponent**.

Edit the code in the file "**book-new.component.ts**":

```
import { Component } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { BookAPIService } from '../book-api.service';
import { Book } from '../interfaces/Book';

@Component({
  selector: 'app-book-new',
  templateUrl: './book-new.component.html',
  styleUrls: ['./book-new.component.css']
})
export class BookNewComponent {
  book=new Book();
  books:any
  errMessage:string=''
  constructor(private _service: BookAPIService){
    this._service.getBooks().subscribe({
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
  postBook()
  {
    this._service.postBook(this.book).subscribe({
```

```
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

-The **book variable** is a variable used to map by ngModel. When the user enters data on the interface, it will automatically map the attributes and input fields.

-The **books variable** to load all books on the server to serve to display the list again.

-To default when opening the interface, the Book list is displayed, in the constructor we call the **HTTP GET books API** (here, if we **reuse** the **BookComponent**, there is no need to call it again)

-The **postBook()** function will call the **POST Book API** to push data from the Interface to the Server for storage. If successful, it will return all the latest data, then we update the list.

Edit the code in the file "**book-new.component.html**":

```
<p>book-new works!</p>
<p>{{errMessage}}</p>
<table>
    <tr>
        <td>Book Id:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel" name="BookId"
id="BookId" placeholder="Enter Book Id"
            [(ngModel)]="book.BookId"/>
        </td>
    </tr>
    <tr>
        <td>Book Name:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel"
name="BookName" id="BookName" placeholder="Enter Book Name"
            [(ngModel)]="book.BookName"/>

        </td>
    </tr>
    <tr>
        <td>Price:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel"
name="BookPrice" id="BookPrice" placeholder="Enter Price"
            [(ngModel)]="book.Price"/>
        </td>
    </tr>
    <tr>
```

```
        <td colspan="2">
            <button (click)="postBook()">Submit</button>
        </td>
    </tr>
</table>
<table border="1">
    <tr>
        <td>BookId</td>
        <td>BookName</td>
        <td>Price</td>
        <td>Image</td>
    </tr>
    <tbody>
        <tr  *ngFor="let book of books">
            <td>{{book.BookId}}</td>
            <td>{{book.BookName}}</td>
            <td>{{book.Price}} </td>
            <td>
                <img src="assets/book_images/{{book.Image}}" width="50px"
height="50px">
            </td>
        </tr>
    </tbody>
</table>
```

The first table is used to input the details of the Book, using **[(ngModel)]** for binding.

The second table is used to display the list of Books retrieved from the server. If we reuse the component, we can replace this table with a single directive:

```
<div app-books></div>
```

Configure routing or refer directly to "**app.component.html**" to run this component, we will have the result as the exercise requested.

Exercise 119: Create HTTP PUT – Update a Book

Requirement:

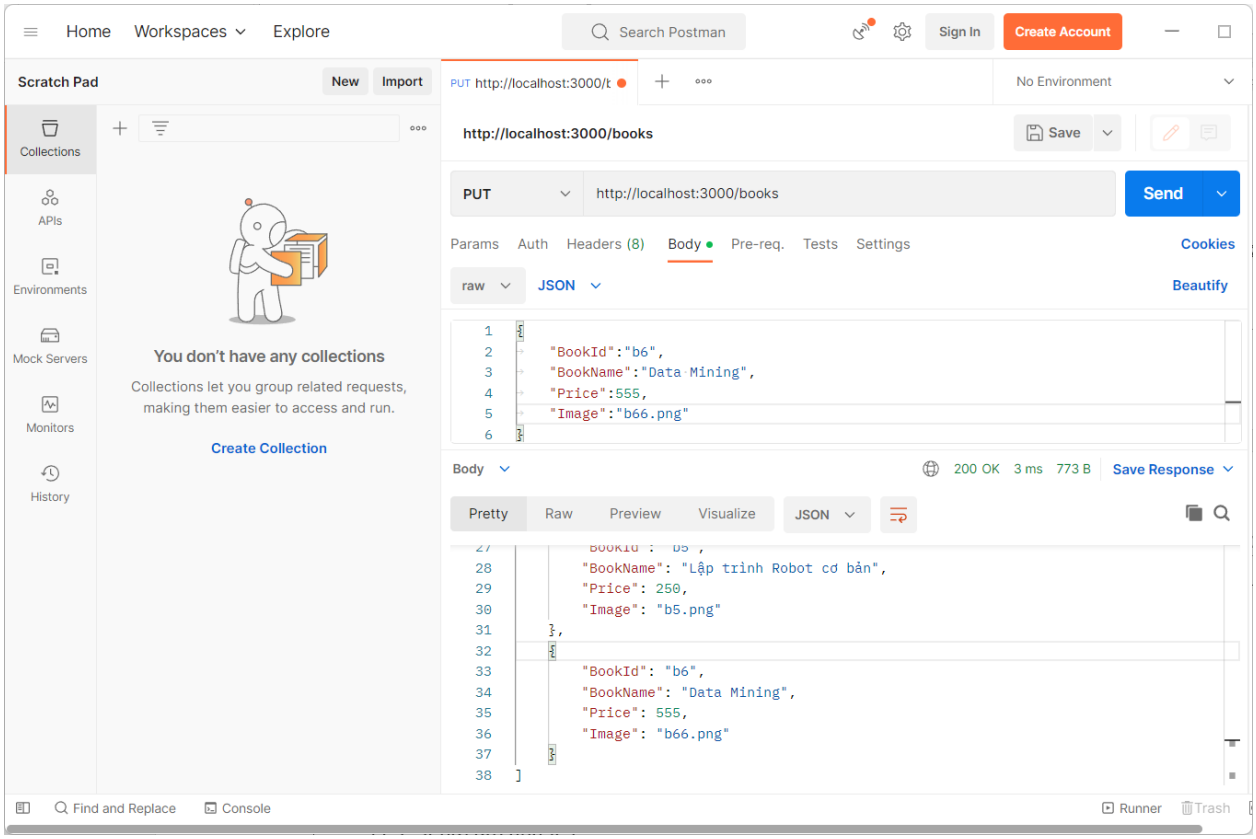
Write a **Restful API HTTP PUT** to **update** a Book's information when the **BookId** is known.

Instructions:

Step 1: Open **index.js** file in project "**my-server**", add Restful API update book as below:

```
app.put("/books",cors(),(req,res)=>{
    book=database.find(x=>x.BookId==req.body.BookId)
    if(book!=null)
    {
        book.BookName=req.body.BookName
        book.Price=req.body.Price
        book.Image=req.body.Image
    }
    res.send(database)
})
```

Step 2: Test **Postman**, to make sure the API works before plugging into the Front End for programming:



-Select method as **PUT**
-Select **raw/JSON** in Body tag
-The body tag enters the Book Json structure to send to the Server

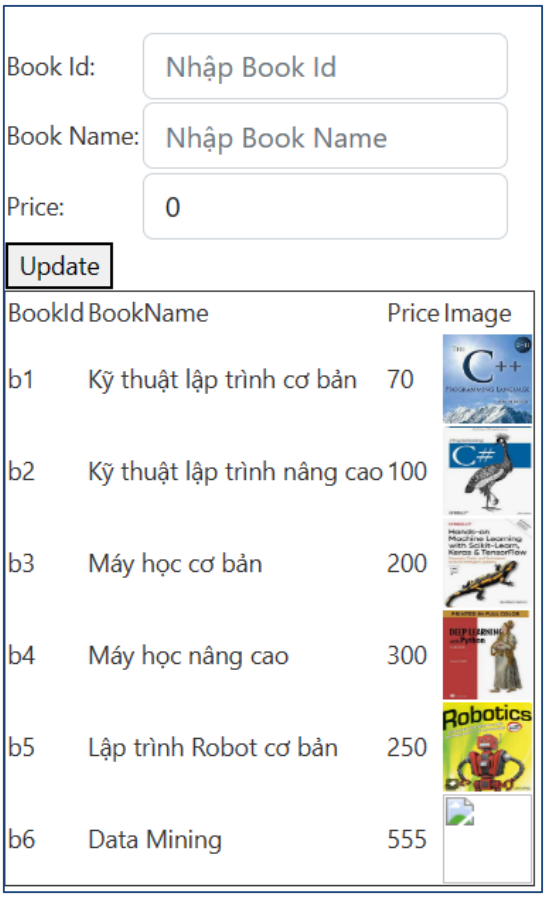
```
{
    "BookId":"b6",
    "BookName":"Data Mining",
    "Price":555,
    "Image":"b66.png"
}
```

- Click the "**send**" button to see that the server has successfully edited the data when it returns the latest list of data with the updated information.

Exercise 120: Invoke HTTP PUT – Update a Book

Requirement:

The HTML design is similar to the HTTP POST call, but this case is to submit data for editing:



Instructions:

Step 1:

Open the file "**book-api.service.ts**" to add the **putBook** function to the **BookAPIService** class.

```
  putBook(aBook:any):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-
Type","application/json;charset=utf-8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return
this._http.put<any>("/books",JSON.stringify(aBook),requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }
```

Step 2:

Create one more component **BookUpdate**.
File "**book-update.component.ts**" edit the code:

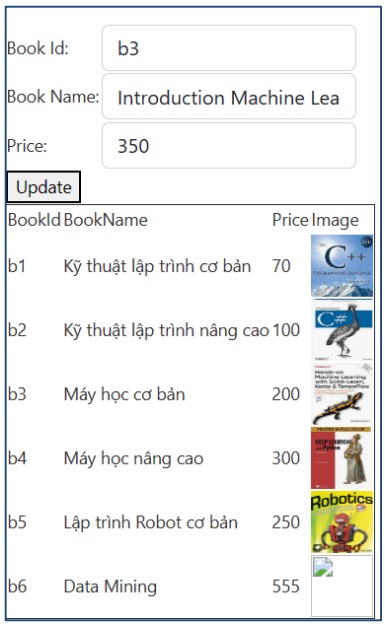
```
export class BookUpdateComponent {
  book=new Book();
  books:any
  errMessage:string=''
  constructor(private _service: BookAPIService){
    this._service.getBooks().subscribe({
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
  putBook()
  {
    this._service.putBook(this.book).subscribe({
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
  }
```

File "**book-update.component.html**" edit the code:

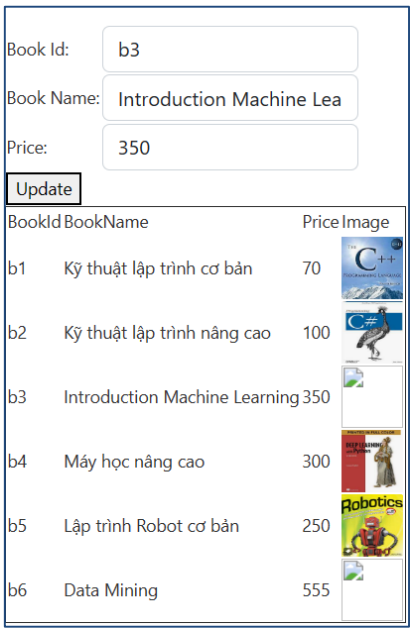
```
<p>book-update works!</p>
<p>{{errMessage}}</p>
<table>
    <tr>
        <td>Book Id:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel" name="BookId"
id="BookId" placeholder="Enter Book Id"
            [(ngModel)]="book.BookId"/>
        </td>
    </tr>
    <tr>
        <td>Book Name:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel"
name="BookName" id="BookName" placeholder="Enter Book Name"
            [(ngModel)]="book.BookName"/>

        </td>
    </tr>
    <tr>
        <td>Price:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel"
name="BookPrice" id="BookPrice" placeholder="Enter Price"
            [(ngModel)]="book.Price"/>
```

```
        </td>
    </tr>
    <tr>
        <td colspan="2">
            <button (click)="putBook()">Update</button>
        </td>
    </tr>
</table>
<table border="1">
    <tr>
        <td>BookId</td>
        <td>BookName</td>
        <td>Price</td>
        <td>Image</td>
    </tr>
    <tbody>
        <tr  *ngFor="let book of books">
            <td>{{book.BookId}}</td>
            <td>{{book.BookName}}</td>
            <td>{{book.Price}} </td>
            <td>
                <img src="assets/book_images/{{book.Image}}" width="50px"
height="50px">
            </td>
        </tr>
    </tbody>
</table>
```

Configure routing or refer directly to "app.component.html" to run this component, we will get the results as required..



Before modifying



After modifying

Exercise 121: Create HTTP DELETE– Remove a Book

Requirement:

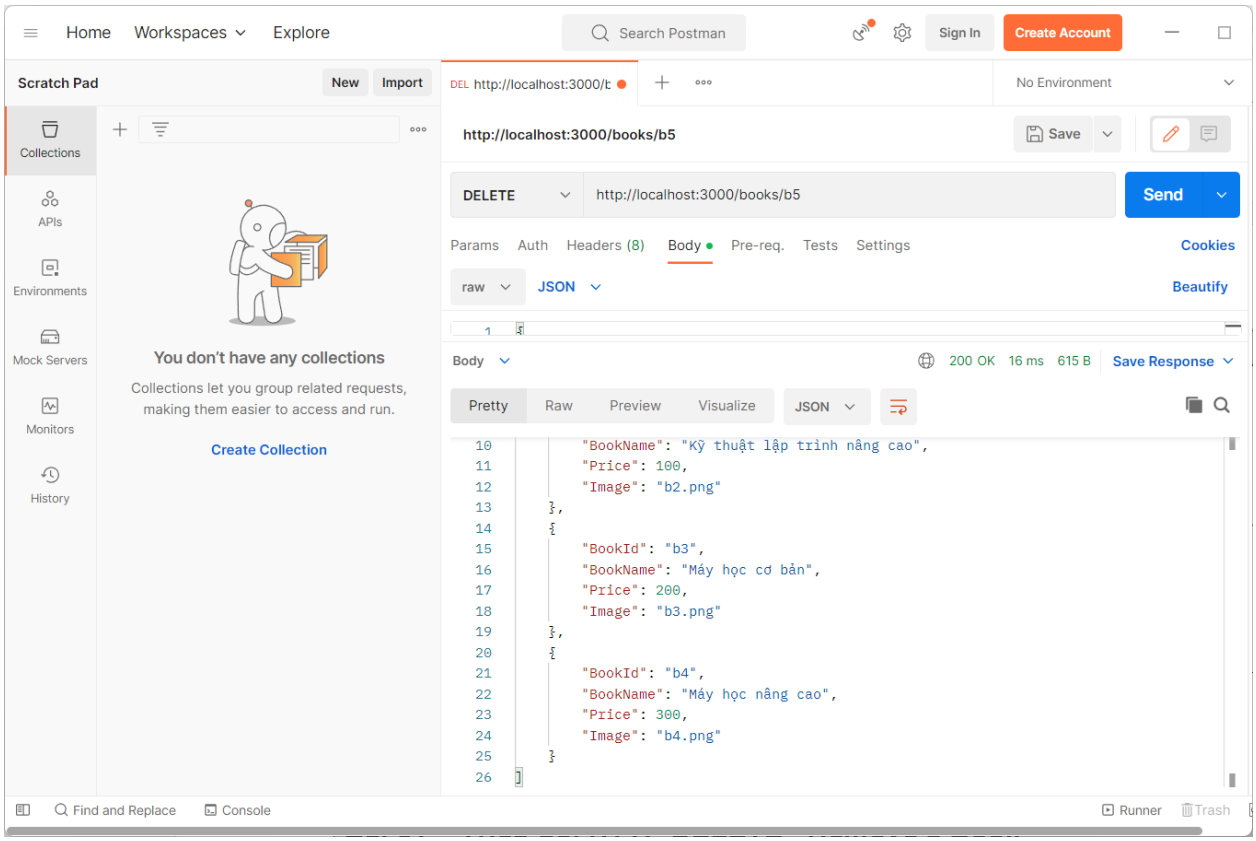
Write Restful API to **delete** 1 Book.

Instructions:

In the "**my-server**" project, edit **index.js**:

```
app.delete("/books/:id",cors(),(req,res)=>{
    id=req.params["id"]
    database = database.filter(x => x.BookId !== id);
    res.send(database)
})
```

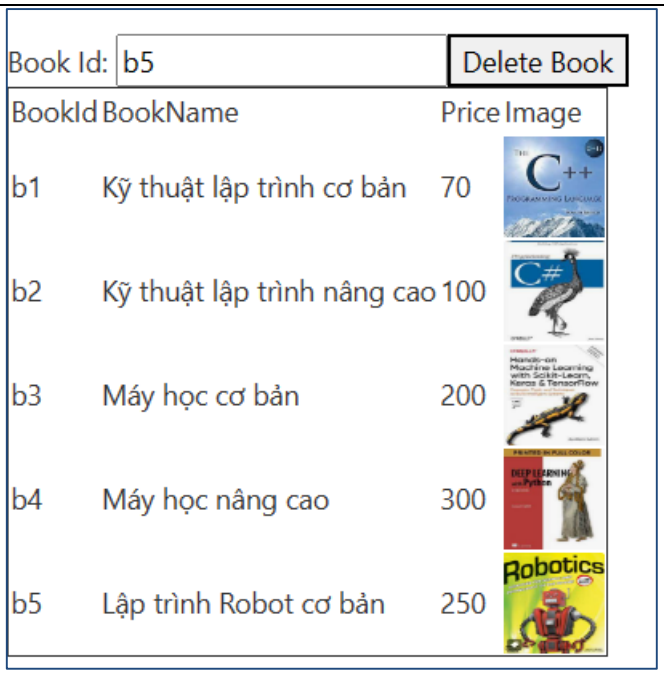
In Postman, select DELETE method, and the url = "**http://localhost:3000/books/b5**" Then press the Send button, the program will successfully delete the Book with BookId=b5.



Exercise 122: Invoke HTTP DELETE– Remove a Book

Requirement:

Design the HTML interface to **delete** the Book by calling the HTTP Delete API.



-By default, the Book list will be displayed on the HTML interface

-Enter the **Book Id**, press the **Delete Book button** to delete and update the list again if the deletion is successful

Instructions:

Step 1: Open the file "**book-api.service.ts**" and Add the script for **BookAPIService**:

```
deleteBook(bookId:string):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-
Type","application/json;charset=utf-8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.delete<any>("/books/"+bookId,requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<IBook>),
        retry(3),
        catchError(this.handleError))
  }
```

Step 2: Make more component BookDelete.

Added script for "**book-delete.component.ts**":

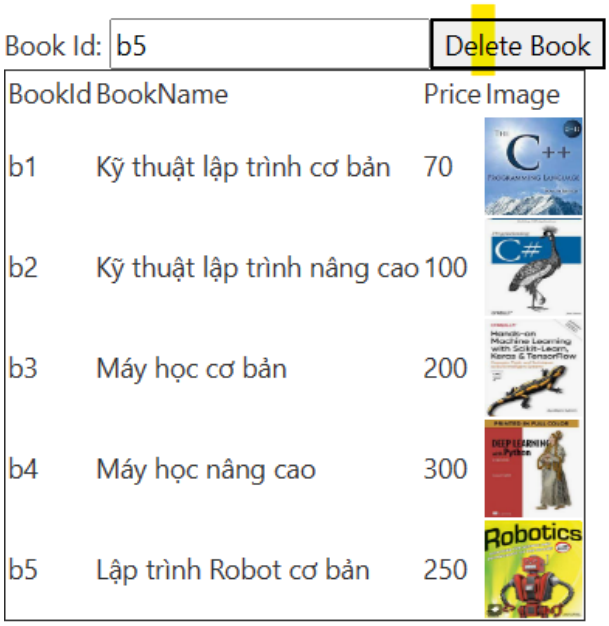
```
import { Component } from '@angular/core';
import { BookAPIService } from '../book-api.service';

@Component({
```

```
  selector: 'app-book-delete',
  templateUrl: './book-delete.component.html',
  styleUrls: ['./book-delete.component.css']
})
export class BookDeleteComponent {
  books:any
  errMessage:string=''
  constructor(private _service: BookAPIService){
    this._service.getBooks().subscribe({
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
  deleteBook(bookId:any)
  {
    this._service.deleteBook(bookId).subscribe({
      next:(data)=>{this.books=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

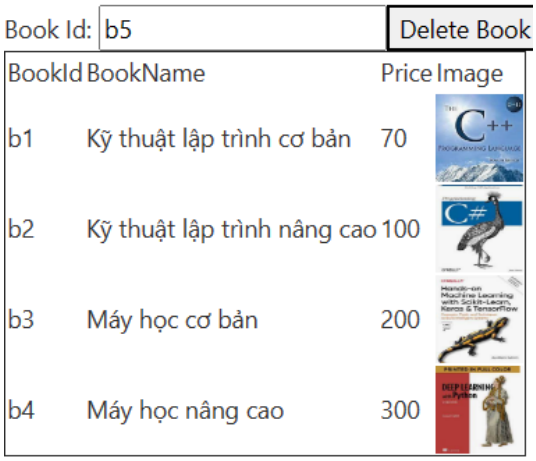
Added script for "**book-delete.component.html**":

```
<p>book-delete works!</p>
<p>{{errMessage}}</p>
Book Id: <input type="text" #bookId> <button
(click)="deleteBook(bookId.value)">Delete Book</button>
<table border="1">
    <tr>
        <td>BookId</td>
        <td>BookName</td>
        <td>Price</td>
        <td>Image</td>
    </tr>
    <tbody>
        <tr  *ngFor="let book of books">
            <td>{{book.BookId}}</td>
            <td>{{book.BookName}}</td>
            <td>{{book.Price}} </td>
            <td>
                <img src="assets/book_images/{{book.Image}}" width="50px"
height="50px">
            </td>
        </tr>
    </tbody>
</table>
```

Configure routing or refer directly to "**app.component.html**" to run this component, we will get the results as required.



Before deleting



After Deleting

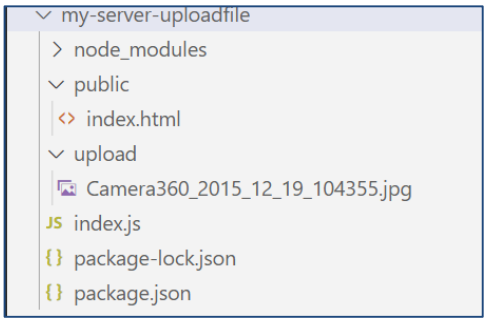
Exercise 123: Upload files to Server-1

Requirement:

Create a new **NodeJs** Project named "**my-server-uploadfile**", write the command to upload the file to the Server. In this case, the file loading screen is in the Server

Instructions:

Step 1: Create Project "**my-server-uploadfil**e" with structure as shown below:



- Create public and upload folders

Step 2: Install commands:

```
npm install express
npm install express-fileupload
npm i --save-dev nodemon
npm i --save-dev morgan
npm i body-parser
```

- After installing and configuring "**package.json**"

```
{
  "name": "my-server-uploadfile",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Tran Duy Thanh",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "express-fileupload": "^1.4.0"
  },
  "devDependencies": {
    "morgan": "^1.10.0",
    "nodemon": "^2.0.21"
  }
}
```

Step 3: Write code for **index.js**:

```
const express = require('express');
const fileUpload = require('express-fileupload');
const app = express();
const port = 3001;
const morgan=require("morgan")
app.use(morgan("combined"))
const bodyParser=require("body-parser")
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));

app.use(
    fileUpload({
        limits: {
            fileSize: 10000000,
        },
        abortOnLimit: true,
    })
);

// Add this line to serve our index.html page
app.use(express.static('public'));

app.get('/', (req, res) => {
    res.sendFile('index.html');
});
app.post('/upload', (req, res) => {
    // Get the file that was set to our field named "image"
    const { image } = req.files;
```

```
    // If no image submitted, exit
    if (!image) return res.sendStatus(400);

    // Move the uploaded image to our upload folder
    image.mv(__dirname + '/upload/' + image.name);

    // All good
    res.sendStatus(200);
});

app.listen(port, () => {
    console.log(`Example app listening on port ${port}`);
});
```

-Step 4: Create index.html in **public** folder

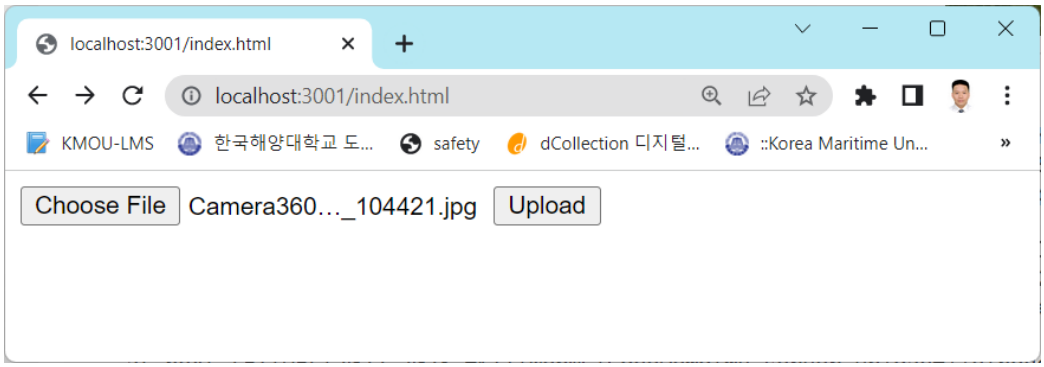
```
<!DOCTYPE html>
<form action="/upload" method="POST" enctype="multipart/form-data">
    <input type="file" name="image" />
    <button type="submit">Upload</button>
</form>
```

-Run Server Application: **npm start**

```
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server-uploadfile> npm start

> my-server-uploadfile@1.0.0 start
> nodemon index.js

[nodemon] 2.0.21
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Example app listening on port 3001
```

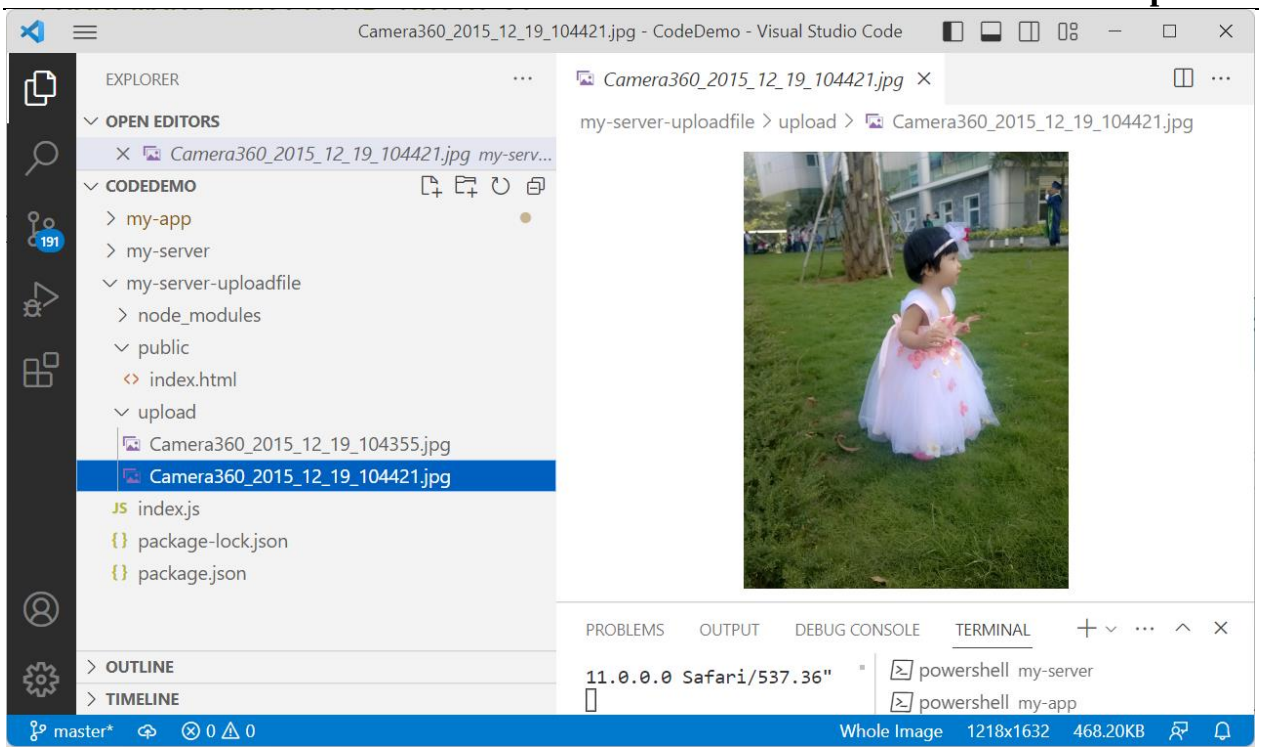
Run the browser:

http://localhost:3001/index.html



Click Choose File, select an image and then upload it

Result after successful upload:



Exercise 124: Upload files to Server-2

Requirement:

Reuse "**my-server-uploadfile**" as a standalone service for uploading images (and you don't need to reprogram the upload function), or you can move commands in your "**index.js**" project "**my-server-uploadfile**" to **"index.js**" of project "**my-server**".

Students apply this exercise to update images for Books in previous exercise.

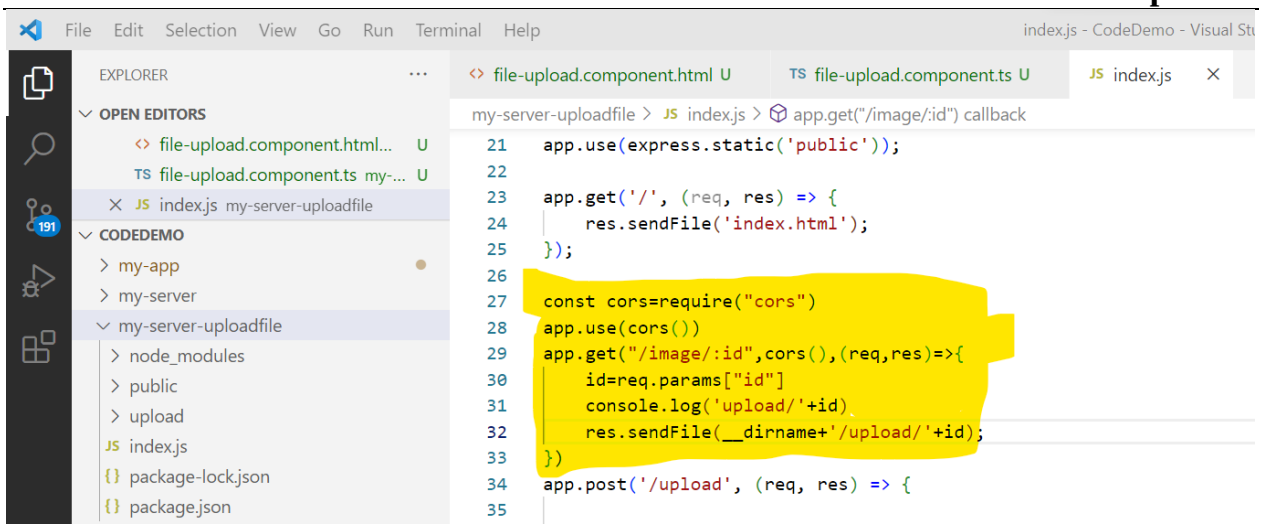
Once the image is uploaded to the Server, we already have a physical path stored on the Server (not in the assets folder), when loading the Json data returned to the HTML displaying the image, we just need to correct the price. The value of the "Image" property of the object points to the correct path of the image uploaded to the server.

Instructions:
In this exercise, the documentation will show you how to reuse "**my-server-uploadfile**" and add how to review uploaded images. Based on this exercise, Students can easily apply for the update of Books' images in previous exercise.

Step 1: Install more cors library for "**my-server-uploadfile**"

npm i cors

Step 2: Open the **index.js** file of the project "**my-server-uploadfile**" and write the Get API to download the image from the server to the client when knowing the image name to display it in HTML



For example, when there is a certain image name, in the client (AngularJS) in the components we will call:

<img src='http://localhost:3001/image/metaverse.png'>

It will automatically call the GET API to display the image details on the HTML interface

That is, if applied to save images for the book, we only need to save the image name when creating a new Book on the Server. In this example, **book.image= "metaverse.png"**:

Instead of the previous exercises write:

```
<img src="assets/book_images/{{book.Image}}">
```

Then change to:

```
<img src="'http://localhost:3001/image/{{book.Image}}">
```

Below is the full code of **index.js** in the server "**my-server-uploadfile**", note this Webserver is running on port 3001

```
const express = require('express');
const fileUpload = require('express-fileupload');
const app = express();
const port = 3001;
const morgan=require("morgan")
app.use(morgan("combined"))
const bodyParser=require("body-parser")
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));

app.use(
    fileUpload({
        limits: {
            fileSize: 10000000,
        },
        abortOnLimit: true,
    })
);
```

```
// Add this line to serve our index.html page
app.use(express.static('public'));

app.get('/', (req, res) => {
    res.sendFile('index.html');
});

const cors=require("cors")
app.use(cors())
app.get("/image/:id",cors(),(req,res)=>{
    id=req.params["id"]
    console.log('upload/'+id)
    res.sendFile(__dirname+'/upload/'+id);
})
app.post('/upload', (req, res) => {

    // Get the file that was set to our field named "image"
    const { image } = req.files;

    // If no image submitted, exit
    if (!image) return res.sendStatus(400);

    // Move the uploaded image to our upload folder
    image.mv(__dirname + '/upload/' + image.name);

    // All good
    res.sendStatus(200);
});

app.listen(port, () => {
    console.log(`Example app listening on port ${port}`);
});
```

Step 3: Open project "**my-app**" to reference using this Restful API in "**my-server-uploadfile**", so "**my-app**" has used many Restful API in different Web server (it is also true for reality).

- Open the file "**proxy.conf.json**" with additional declaration:

```
    {
        "context": ["/upload"],
        "target": "http://localhost:3001",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    }
```

Here is the full declaration of "**proxy.conf.json**":

```
[
    {
      "context": ["/exchange"],
      "target": "https://www.dongabank.com.vn",
      "secure": true,
      "changeOrigin":true,
      "logLevel":"debug"
    },
    {
        "context": ["/products"],
        "target": "https://fakestoreapi.com",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    },
    {
        "context": ["/books"],
        "target": "http://localhost:3000",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    },
    {
        "context": ["/upload"],
        "target": "http://localhost:3001",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    }
 ]
```

Step 4: Create a new component with the name "FileUpload".
-The code "**file-upload.component.html**" is as follows:

```
<input type="file" class="file-input"
       (change)="onFileSelected($event)" #fileUpload name="image">

<div class="file-upload">

   {{fileName || "No file uploaded yet."}}

    <button mat-mini-fab color="primary" class="upload-btn"
      (click)="fileUpload.click()">
        Upload
    </button>
</div>
```

The code of "**file-upload.component.ts**" is as follows:

```
import { HttpClient, HttpEventType } from '@angular/common/http';
import { Component, Input } from '@angular/core';
import { finalize, Subscription } from 'rxjs';
@Component({
  selector: 'app-file-upload',
  templateUrl: './file-upload.component.html',
  styleUrls: ['./file-upload.component.css']
})
export class FileUploadComponent {
  @Input()
  requiredFileType:any;
  fileName = '';
  uploadProgress:number=0;
  uploadSub: Subscription=new Subscription();

  constructor(private http: HttpClient) {}

  onFileSelected(event:any) {
      const file:File = event.target.files[0];
      if (file) {
          this.fileName = file.name;
          const formData = new FormData();
          formData.append("image", file);

          const upload$ = this.http.post("/upload", formData, {
              reportProgress: true,
              observe: 'events'
          })
          .pipe(
              finalize(() => this.reset())
          );
          this.uploadSub = upload$.subscribe(event => {
            if (event.type == HttpEventType.UploadProgress) {
              this.uploadProgress = Math.round(100 * (event.loaded /
event.total!));
            }
          })
      }
  }
cancelUpload() {
  this.uploadSub.unsubscribe();
  this.reset();
}
reset() {
  this.uploadProgress = 0;
  this.uploadSub = new Subscription();
}
 }
```

Configure Routing or directly reference "**app.component.html**" to run it will get the desired result. Students think to integrate this image into Book.

Exercise 125: Restful API self-study (*)

Instructions:

-Build Restful API providing functions: get all book information, view book details, create new book, edit book, delete book.

-Design the book information management screen as shown:

QUẢN LÝ THÔNG TIN SÁCH

Create New

Tensach	Giaban	Mota	Anhbia	Ngaycapnhat	Soluongton	MaCD	MaNXB	
Giáo trình Tin học cơ bản	26000.00	Nội dung của cuốn: Tin Học Cơ Bản Windows XP gồm có 7 chương: Chương 1: Một số vấn đề cơ bản. Chương 2: Sử dụng nhanh thanh công cụ và thanh thực đơn trong My Computer và Windows Explorer. Chương 3: Các thao tác trong windows XP. Chương 4: Các thiết lập trong Windows XP. Chương 5: Bảo trì máy tính. Chương 6: Các phím tắt Chương 7: Hỏi và đáp các thắc mắc. Xin trân trọng giới thiệu cuốn sách cùng bạn	THCB.jpg	25/10/2014 12:00:00 SA	120	7	1	Edit | Details | Delete
Giáo trình Cơ Sở Dữ Liệu Với Visual Basic 2005 Và ADO.NET 2.0	12000.00	Cuốn sách này gồm 3 phần sau: Phần 1: Xử lý văn bản trong Microsoft thiệu các nội dung sau: Chương 1: Căn bản về cơ sở dữ liệu. Chương 2: Các bộ kết nối và tương tác dữ liệu. Chương 3: Bộ chứa dữ liệu DataSet. Chương 4: Bộ điều hợp dữ liệu DataAdapter. Chương 5: Sử dụng các điều khiển ràng buộc dữ liệu. Chương 6: Tạo báo cáo với Crystal Report. Chương 7: ADO.NET và XML. Xin trân trọng giới thiệu cùng các bạn.	TH004.jpg	23/10/2013 12:00:00 SA	25	3	2	Edit | Details | Delete
Visual Basic 2005 Tập 3, Quyển 2: Lập Trình Web	20000.00	"Visual Basic 2005 Tập 3, Quyển 2: Lập Trình Web Với Cơ Sở Dữ Liệu" sẽ cung cấp kỹ thuật và hướng dẫn bạn: Tự học xây dựng ứng dụng Web quản lý CSDL toàn diện với ADO.NET 2.0 và ASP.NET. Khai thác các đối tượng và nguồn dữ liệu dành cho WebForm. Sử dụng các điều khiển dữ liệu đặc thù dành riêng cho ASP.NET và Web. Làm quen với những khái niệm xử lý dữ liệu hoàn toàn mới. Ràng buộc dữ liệu với các thành phần giao diện Web Forms. Thiết kế ứng dụng Web "Quản lý CD Shop" trực quan và thực tế. Cung cấp một kiến thức hoàn chỉnh về Web cho các bạn yêu thích ngôn ngữ Visual Basic và .NET Framework. Sách có kèm theo CD-ROM bài tập.	LTWeb2005.jpg	15/09/2014 12:00:00 SA	240	8	4	Edit | Details | Delete

+ By default, the program will call the GET all Books API to display.

+ Press the "**Create New**" button to display the screen to add books, when successfully added, it will return to the original list screen.

+ Pressing the "**Edit**" button displays the book information editing screen, successful editing will return to the original list screen.

+ Click the "**Details**" button to display detailed information of the selected book

+ Pressing the "**Delete**" button will confirm whether to delete or not, if the deletion is successful, reload the list after deleting.

- Students must design their own JSon structure for Book to meet the requirements of the exercise.

- Processing images for Book.

Module 12: Restful API with MongoDB

Practical knowledge content:

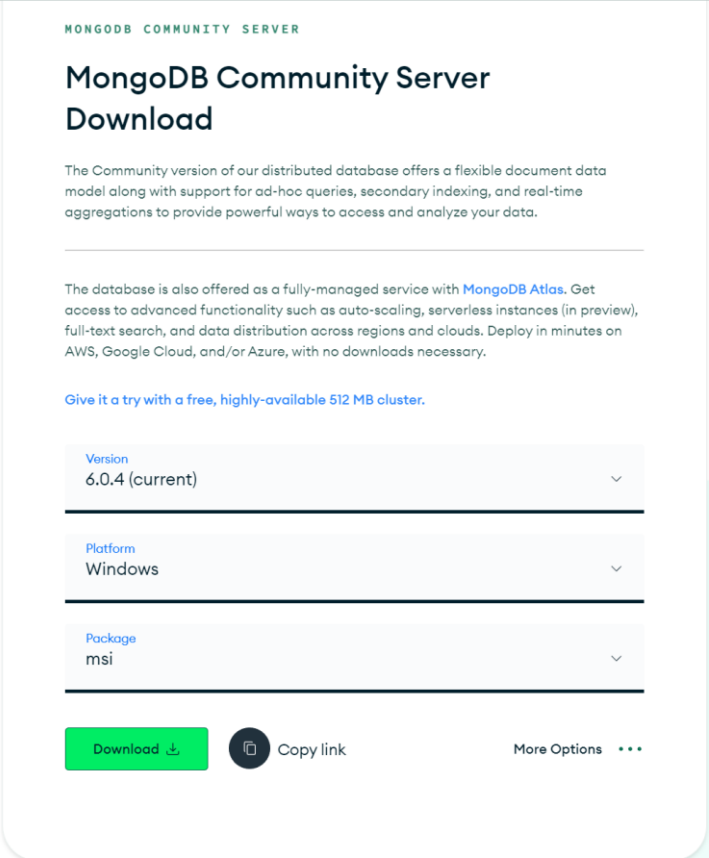
+ Strengthening knowledge and programming skills about Restful API
+ Using MongoDB and MongoDB Compass
+ MongoDB connection libraries
+ Create Restful API with MongoDB
+ Store images in MongoDB

Exercise 126: Install and use MongoDB

Requirement:

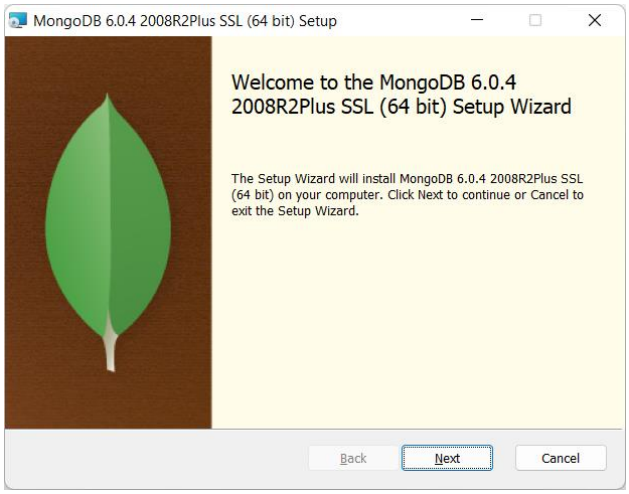
To download MongoDB, go to the link:

https://www.mongodb.com/try/download/community

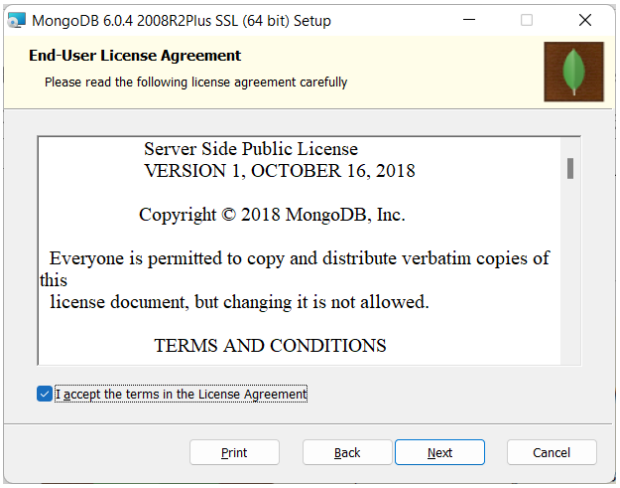


Click Download to get the file "mongodb-windows-x86_64-6.0.4-signed.msi"

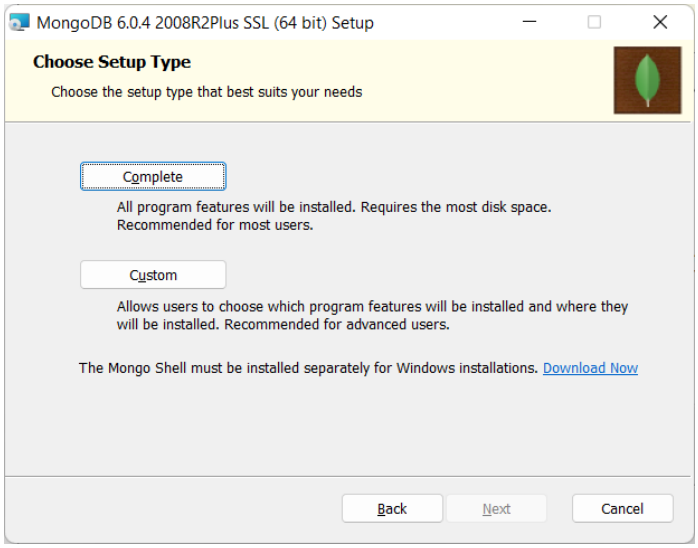
Click on the file to install



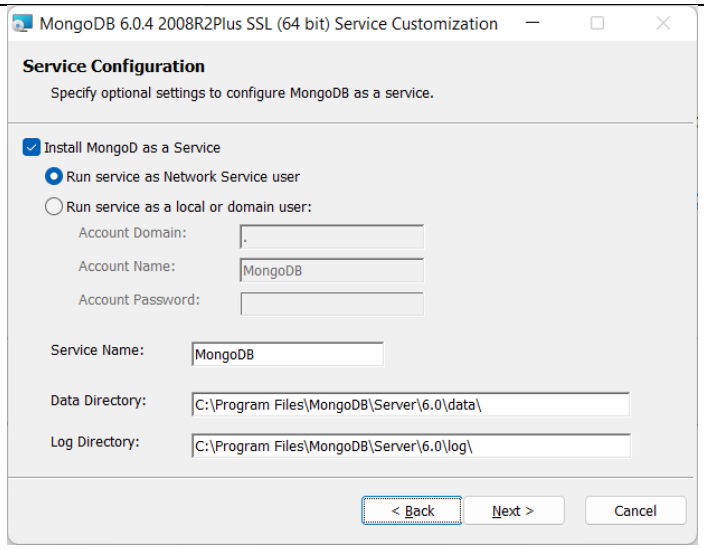
Click Next



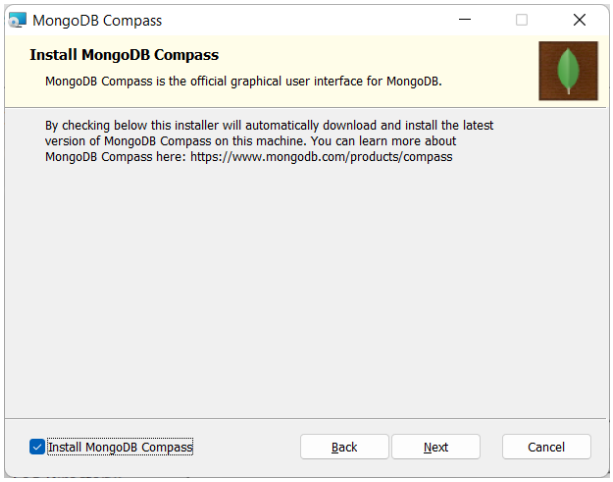
Checked “I accept the terms in the License Agreement” and click Next



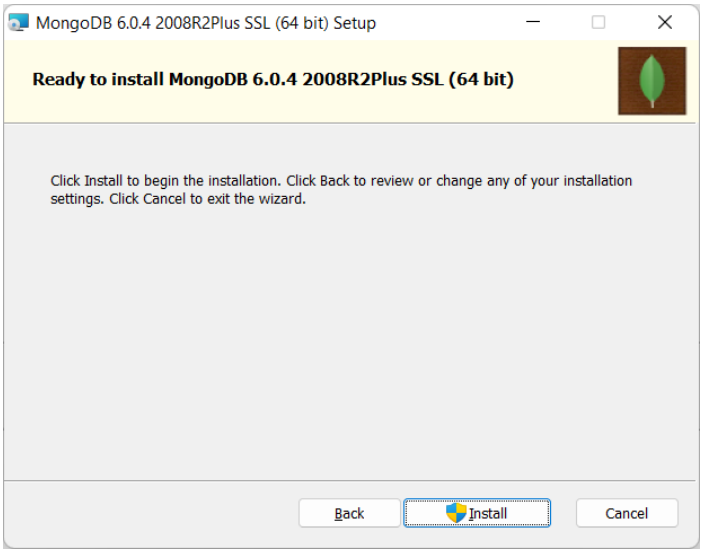
Press the complete button



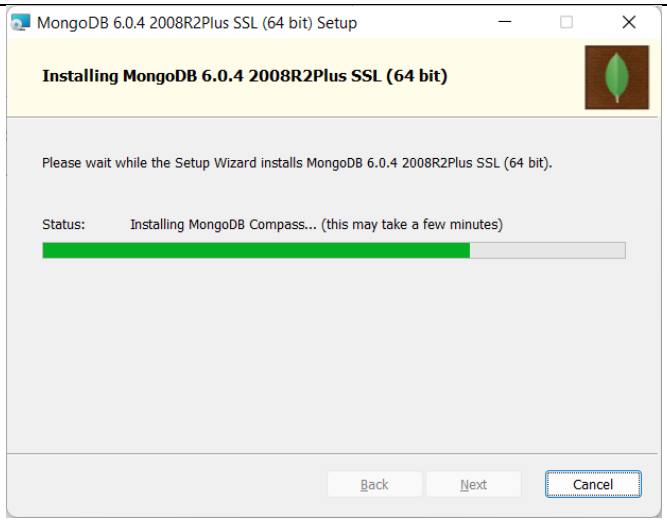
Set the default as shown above and click Next



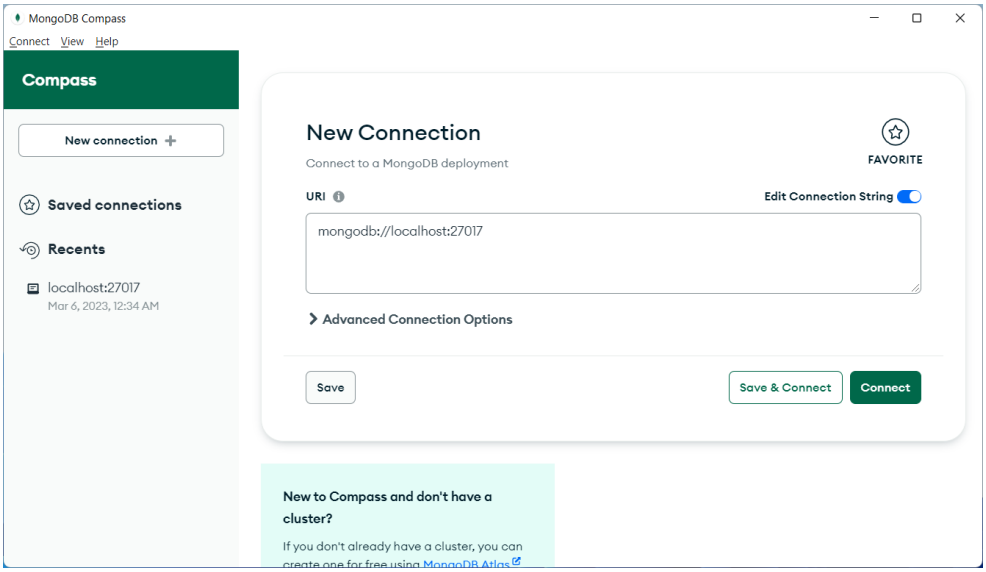
Checked “Install MongoDB Compass” and then click Next. Compass is a tool for us to manipulate MongoDB Server



Click Install



Wait until the system installs successfully. This is the MongoDB Compass screen when the installation is complete:

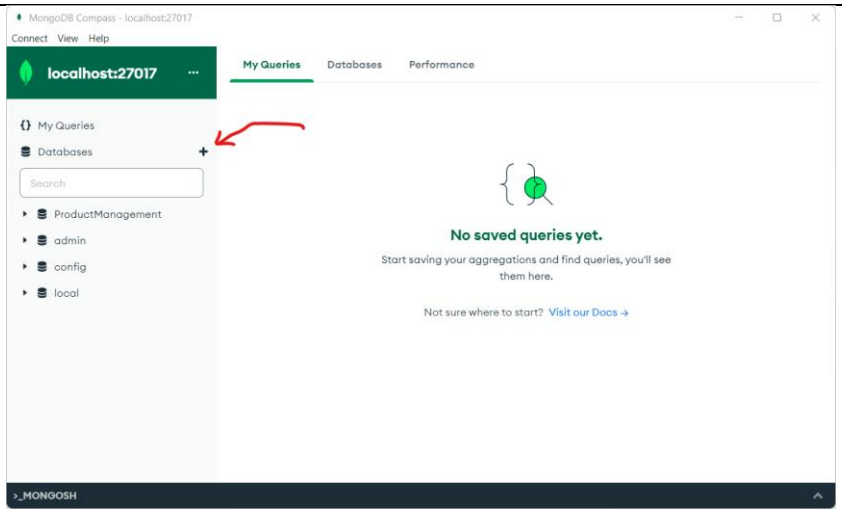


Connection path:

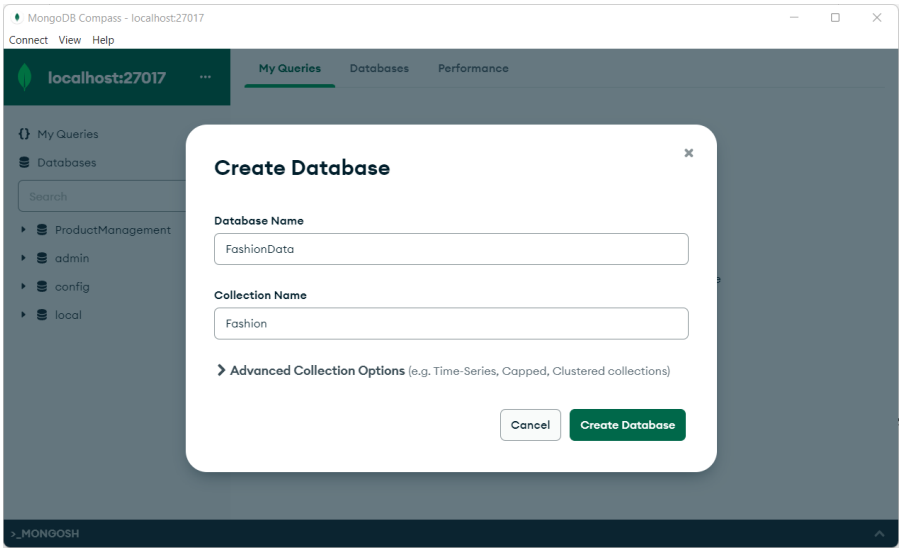
mongodb://localhost:27017

Sometimes during programming, maybe some libraries set default IPv6 if localhost. Thus the computer will not be able to program if it is IPv4. At this point we should replace **localhost** with **127.0.0.1**

Click connect to connect:



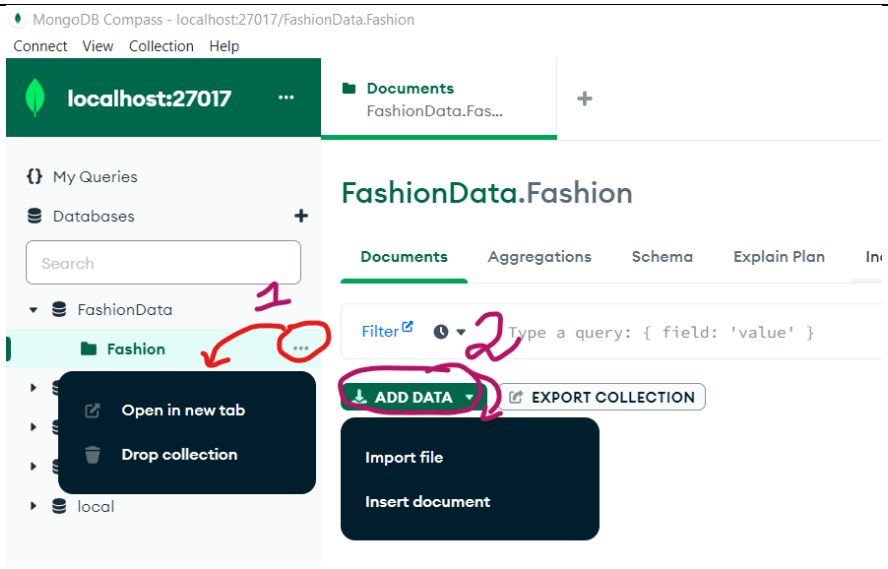
To create a Database, click on the "+" icon next to the word "Databases" and name the database as shown:



Database Name: **FashionData**

Collection Nam: **Fashion**

SQL		NoSQL
Database	➔	Database
Table	➔	Collection
Row	➔	Document
Column	➔	Field/Attribute



Item No. 1:

- "Open in new tab": Open a new tab to work

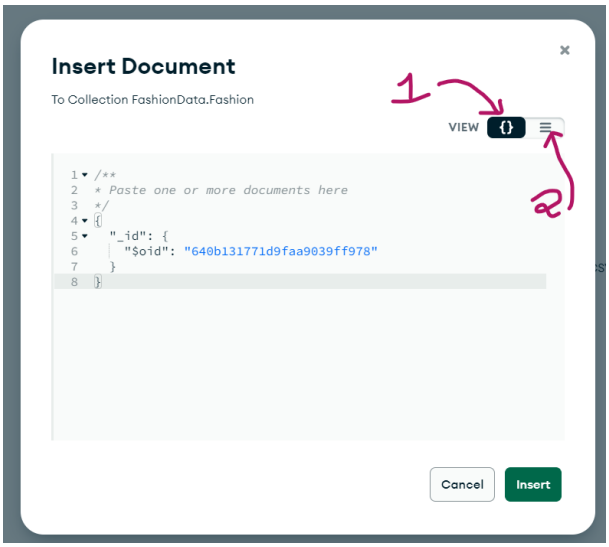
- “Drop collection”: Delete the currently selected collection

Item No. 2:

- “Import File”: Import data from a previously available file

- "Insert document": Add a new data, it is usually a JSon (we can create a complex structure, not only a JSonObject)

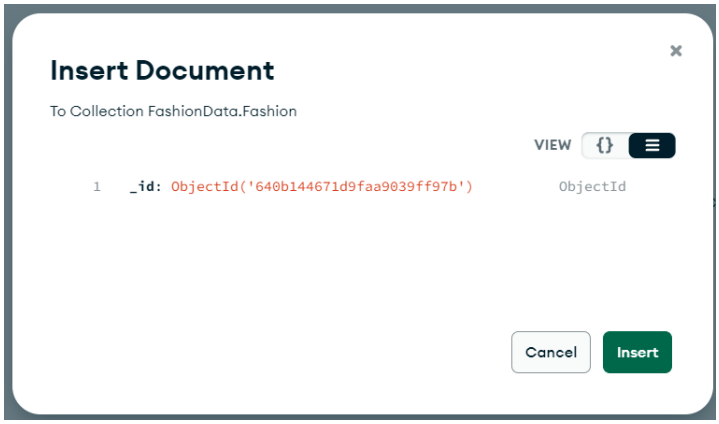
Now we choose “Insert document”:



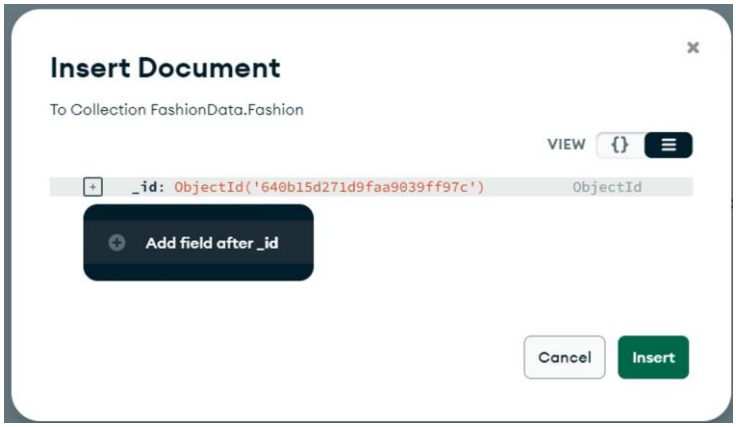
-The way to enter by number 1 is direct Json structure format

-The way to enter by number 2 is the Field Visualization format and for choosing the data type of each Field.

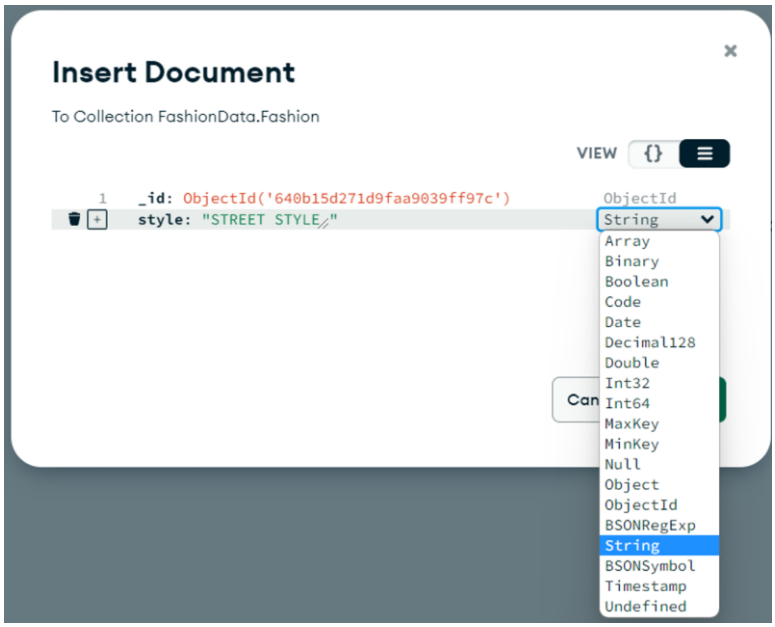
We choose the 2nd way to make it easy to design:



Move the mouse in front of any 1 Field, it shows a "+" sign, click on the "+" sign and select "**Add field after _id**" as shown below:



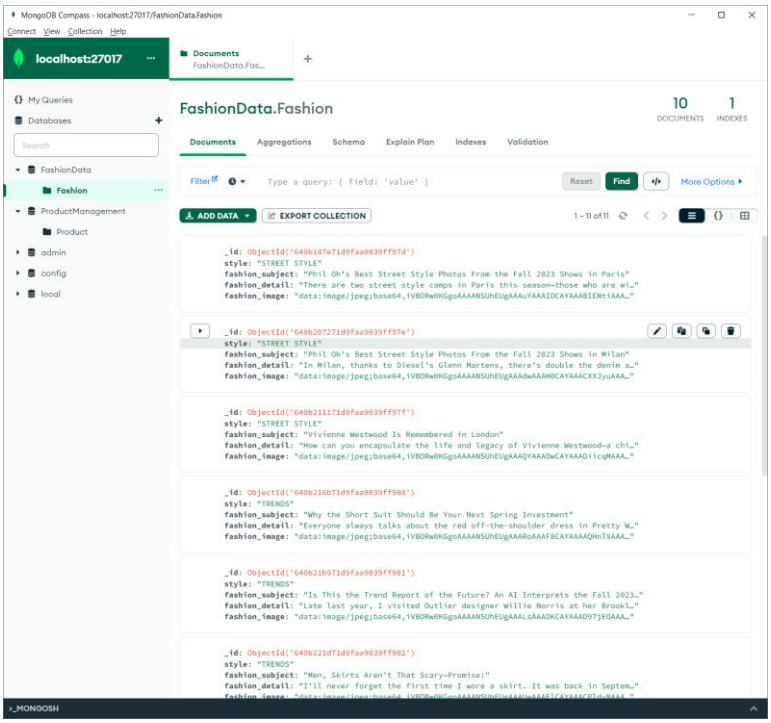
We create a Field named style and value "STREET STYLE" and data type is String as shown below:



Similarly, we continue to create some more Fields for this Collection Fashion:

Image is Base64 String. Use this tool to convert the image to a string and paste it into MongoDB: https://codebeautify.org/image-to-base64-converter

This is an illustration of creating structure and importing documents for the Fashion collection:



Students can download the exported data here:

http://teacher.uelstore.com/resource/Fashion.json

Then use the import collection function to save time, how to import:

-Step 1: First create a database named "FashionData"

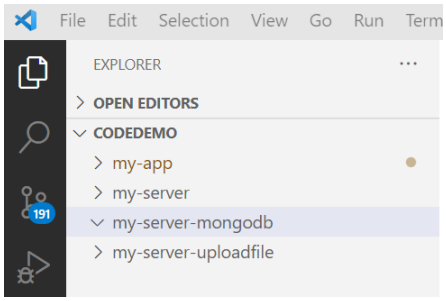
-Step 2: Then create a collection named “Fashion”

-Step 3: Finally select “Fashion” and import the file, point to “Fashion.json”

Exercise 127: Configure and Connect NodeJS to MongoDB

Requirement:

Create a peer directory "**my-server-mongo**db" to build Project for MongoDB Interactive Webserver.



Step 1: Install commands:

```
npm install express
npm install express-fileupload
npm install mongodb
npm i --save-dev nodemon
npm i --save-dev morgan
npm i body-parser
npm i cors
```

In addition, if we use **mongoose** in combination, we can download and install 2 more libraries:

```
npm i mongoose
npm install --save mongoclient
```

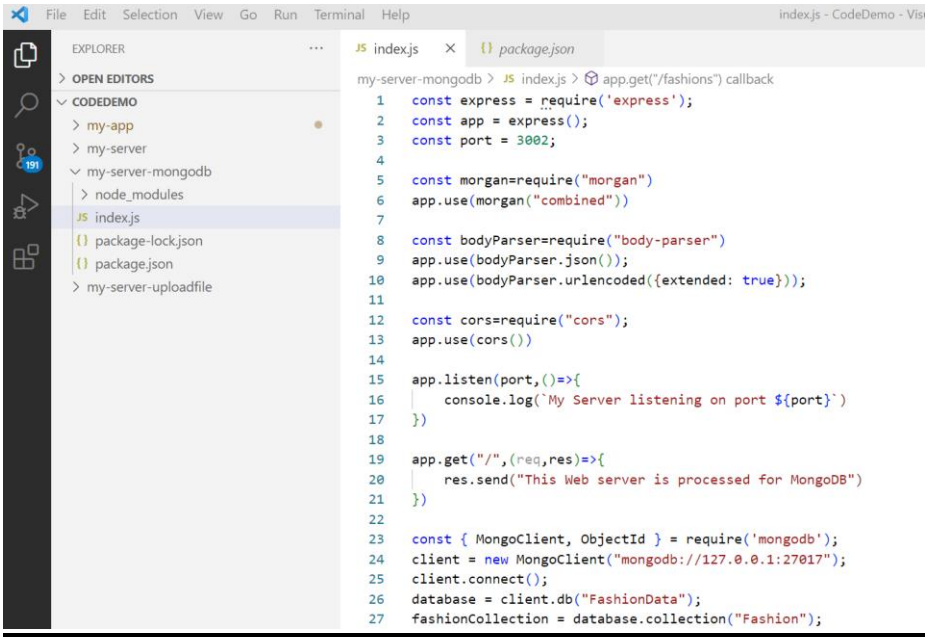
In this article, the documentation for using the library "mongodb"
File "package.json":

```
{
  "name": "my-server-mongodb",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Tran Duy Thanh",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.20.2",
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "express-fileupload": "^1.4.0",
    "mongodb": "^5.1.0"
```

```
  },
  "devDependencies": {
    "morgan": "^1.10.0",
    "nodemon": "^2.0.21"
  }
}
```

Step 2: Create **Index.js** file to write API Service in "**my-server-mongodb**"

Create an **index.js** file to configure some libraries and some functions as learned in previous lessons, running at Port **3002**:



Command **lines from 1 to 21** are the same configuration as learned in the previous lesson to handle Restful API related issues.

Command **lines 23 to 27** are for connecting mongoDB database:

-Line 23 calls the library **mongodb**

-Line 24 declares the connection string to the Server, replace localhost with 127.0.0.1 if the libraries take the computer's IPv6

-Line 25 calls the command to connect to Server

-Line 26 connects to the Database "FashionData" that we designed in the previous exercise.

-Line 27 retrieves the Fashion collection. If the database has many collections that we want to access, just add new declaration lines.

Below is the full code **index.js**:

```
const express = require('express');
const app = express();
const port = 3002;

const morgan=require("morgan")
app.use(morgan("combined"))

const bodyParser=require("body-parser")
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));

const cors=require("cors");
app.use(cors())

app.listen(port,()=>{
    console.log(`My Server listening on port ${port}`)
})

app.get("/",(req,res)=>{
    res.send("This Web server is processed for MongoDB")
})

const { MongoClient, ObjectId } = require('mongodb');
client = new MongoClient("mongodb://127.0.0.1:27017");
client.connect();
database = client.db("FashionData");
fashionCollection = database.collection("Fashion");
```

Step 3: Call the command **npm start** to run the Service

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL

> my-server-mongodb@1.0.0 start
> nodemon index.js

[nodemon] 2.0.21
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
My Server listening on port 3002
```

So now the program has run at **port 3002** and successfully connected to MongoDB

Exercise 128: Create and invoke API - HTTP GET – List Fashion

Requirement:

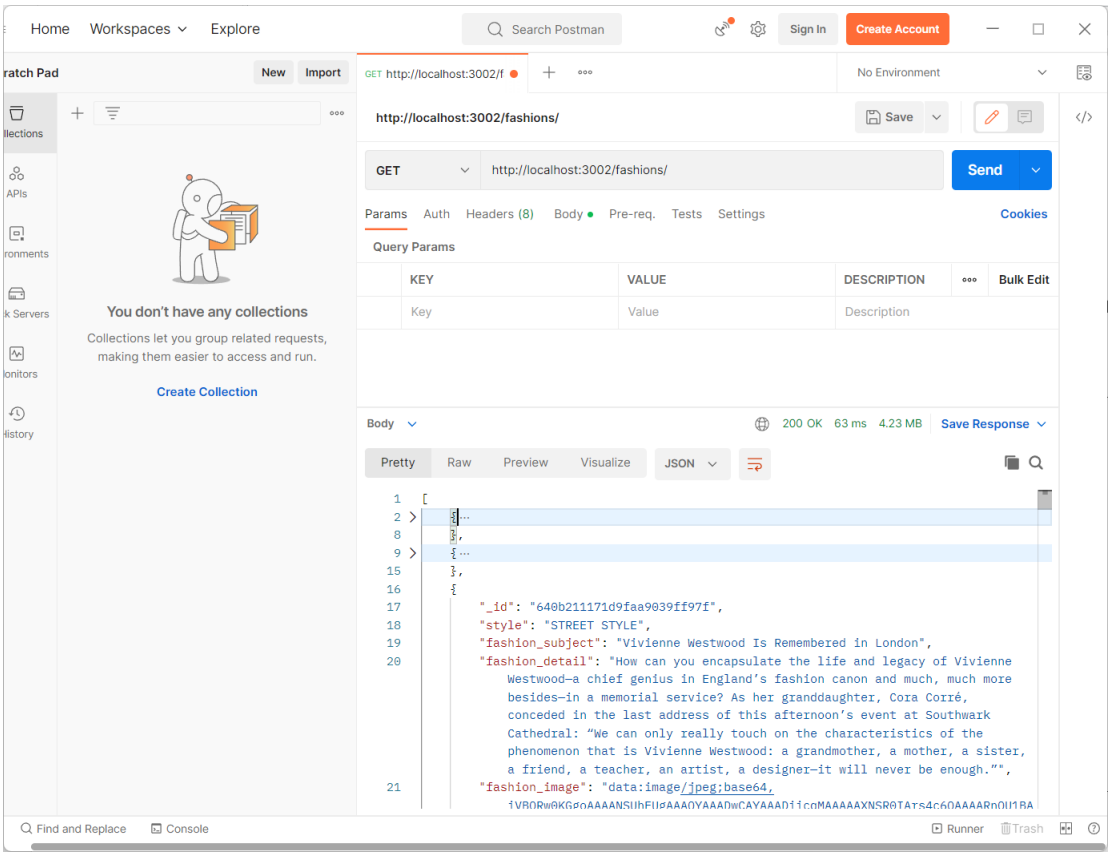
Continue to update the code for **index.js** in the project "**my-server-mongodb**", declare the API to take the entire Fashion and display this Fashion list on the interface in the "**my-app**" project.

Instructions:

Step 1: Go to "**my-server-mongodb**", add the following function at the end of the **index.js** file (additional behind the commands to connect to MongoDB):

```
app.get("/fashions",cors(),async (req,res)=>{
    const result = await fashionCollection.find({}).toArray();
    res.send(result)
    }
)
```

The above command queries the entire Fashion and returns a **JsonArray**. Since we have configured to automatically restart the server when the source code changes, just open Postman to test this function:



If you get the above result, you have successfully written API fashions.

Step 2: In "**my-app**", add a new model class Fashion in the file "**Fashion.ts**":

```
export class Fashion{

    constructor(

        public _id:any=null,

        public style:string="",

        public fashion_subject:string="",

        public fashion_detail:string="",

        public fashion_image:string=""){}

}
```

Step 3: In "my-app", add Service **FashionAPIService**.

Write the code for "**fashion-api.service.ts**":

```
import { HttpClient, HttpErrorResponse, HttpHeaders } from
'@angular/common/http';
import { Injectable } from '@angular/core';
import { catchError, map, Observable, retry, throwError } from 'rxjs';
import { Fashion } from './models/Fashion';

@Injectable({
  providedIn: 'root'
})
export class FashionAPIService {

  constructor(private _http: HttpClient) { }
  getFashions():Observable<any>
  {
    const headers=new HttpHeaders().set("Content-Type","text/plain;charset=utf-
8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return this._http.get<any>("/fashions",requestOptions).pipe(
        map(res=>JSON.parse(res) as Array<Fashion>),
        retry(3),
        catchError(this.handleError))
  }
  handleError(error:HttpErrorResponse){
    return throwError(()=>new Error(error.message))
  }
}
```

Step 4: In "**my-app**", add Component Fashion Component.

Write the code for "**fashion.component.ts**":

```
import { Component } from '@angular/core';
import { FashionAPIService } from '../fashion-api.service';

@Component({
  selector: 'app-fashion',
  templateUrl: './fashion.component.html',
  styleUrls: ['./fashion.component.css']
})
export class FashionComponent {
  fashions:any;
  errMessage:string=''
  constructor(public _service: FashionAPIService){
    this._service.getFashions().subscribe({
      next:(data)=>{this.fashions=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

Write the code for "**fashion.component.html**":

```
<p>fashion works!</p>
{{errMessage}}
<table border="1">
    <tr>
        <td>Id</td>
        <td>Fashion Subject</td>
        <td>Fashion Detail</td>
        <td>Image</td>
    </tr>
    <tbody>
        <tr  *ngFor="let fashion of fashions">
            <td>{{fashion._id}}</td>
            <td>{{fashion.fashion_subject}}</td>
            <td>{{fashion.fashion_detail}} </td>
            <td>
                <img src="{{fashion.fashion_image}}" width="100px" height="100px">
            </td>
        </tr>
    </tbody>
</table>
```

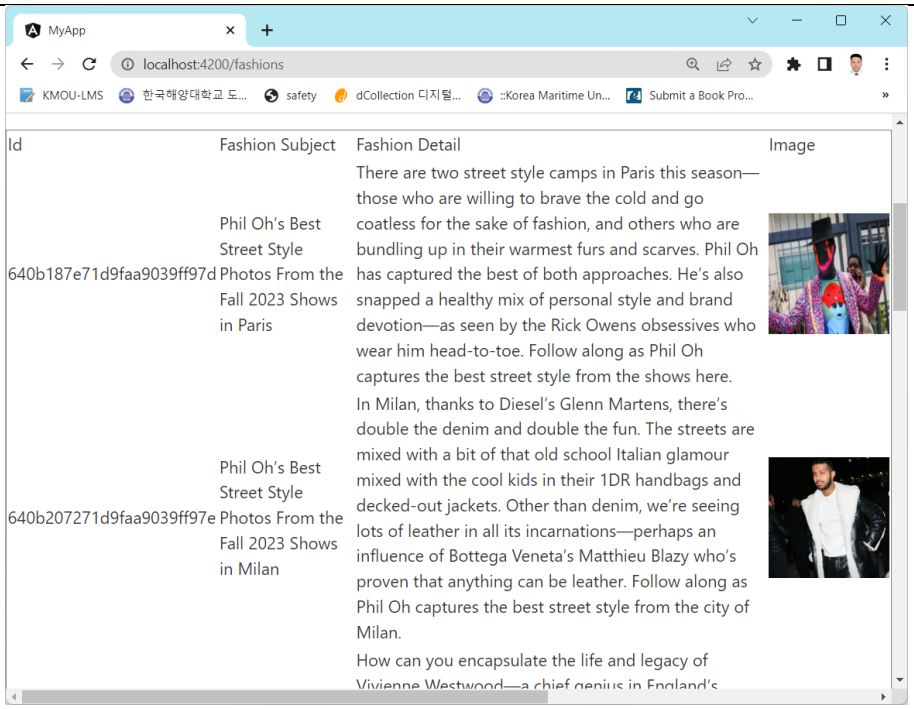
Step 5: In "**my-app**", add declaration for "**proxy.conf.json**":

```
 {
        "context": ["/fashions"],
        "target": "http://localhost:3002",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
 }
```

Full declaration code:

```
[
    {
      "context": ["/exchange"],
      "target": "https://www.dongabank.com.vn",
      "secure": true,
      "changeOrigin":true,
      "logLevel":"debug"
    },
    {
        "context": ["/products"],
        "target": "https://fakestoreapi.com",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    },
    {
        "context": ["/books"],
        "target": "http://localhost:3000",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    },
    {
        "context": ["/upload"],
        "target": "http://localhost:3001",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    }
    ,
    {
        "context": ["/fashions"],
        "target": "http://localhost:3002",
        "secure": true,
        "changeOrigin":true,
        "logLevel":"debug"
    }
 ]
```

Step 6: Declare routing or run this component directly in “app.component.html”, we have the desired result:



Note that the image is Base64string, so we use the syntax:

```
<img src="{{fashion.fashion_image}}" width="100px" height="100px">
```

So when creating a new API to save 1 Fashion, we convert the image to Base64String

Exercise 129: Create and invoke API - HTTP GET – A Fashion

Requirement:

-In "**my-server-mongodb**" update **index.js** to provide API to get 1 Fashion when Id is known

- In "**my-app**" call API to display Fashion details

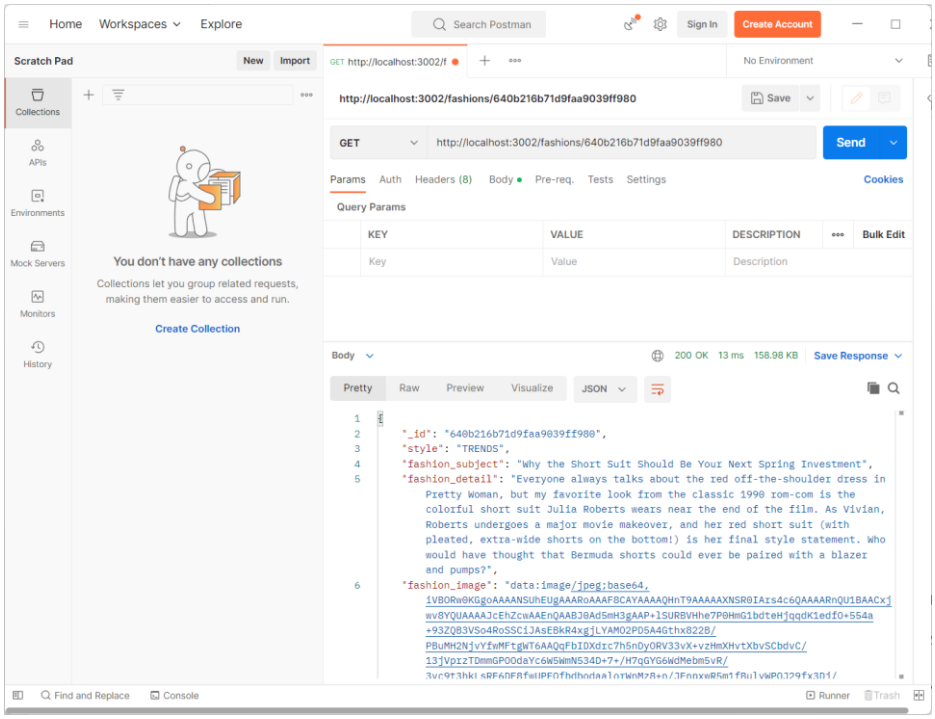
- Students think the HTML interface is suitable for calling API

Instructions:

file "**index.js**" in "**my-server-mongodb**" needs to add the following code at the end of the file:

```
app.get("/fashions/:id",cors(),async (req,res)=>{
    var o_id = new ObjectId(req.params["id"]);
    const result = await fashionCollection.find({_id:o_id}).toArray();
    res.send(result[0])
    }
)
```

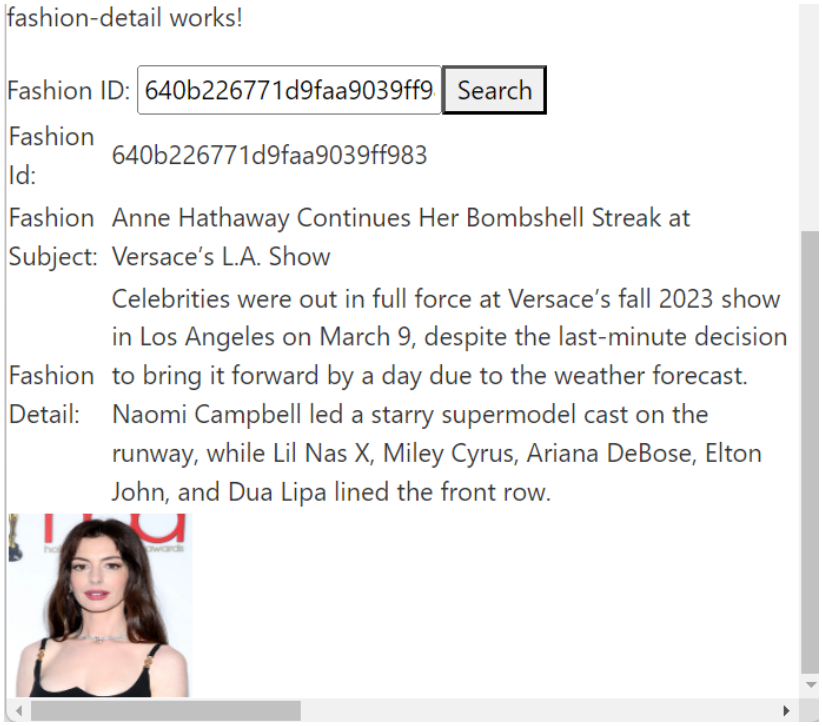
Use Postman to test:



The AngularJS Student section handles the same thing as the previous exercise, just like BookAPIService (the getBook(bookId:string) function) and BookDetailComponent:

-Step 1: Add function getFashion(id:string) in class FashionAPIService. This function calls the API Restful 1 Fashion

-Step 2: Add component "FashionDetailComponent", interface as below, however Students should arrange it aesthetically:



Exercise 130: Create and invoke API - HTTP POST – Create a Fashion

Requirement:

Build Restful API to add a new Fashion, use AngularJS to invoke this API, design appropriate interface.

Instructions:

Step 1: In "**my-server-mongodb**", add the Post a Fashion API in the **index.js** file (added at the end of the file):

```
app.post("/fashions",cors(),async(req,res)=>{
    //put json Fashion into database
    await fashionCollection.insertOne(req.body)
    //send message to client(send all database to client)
    res.send(req.body)
})
```

Note in case the amount sent to the Server (payload) is too large it will report an error, we need to handle this error by adding the following commands in front of the APIs, in this example is for 10mb payload, you can increase if you want:

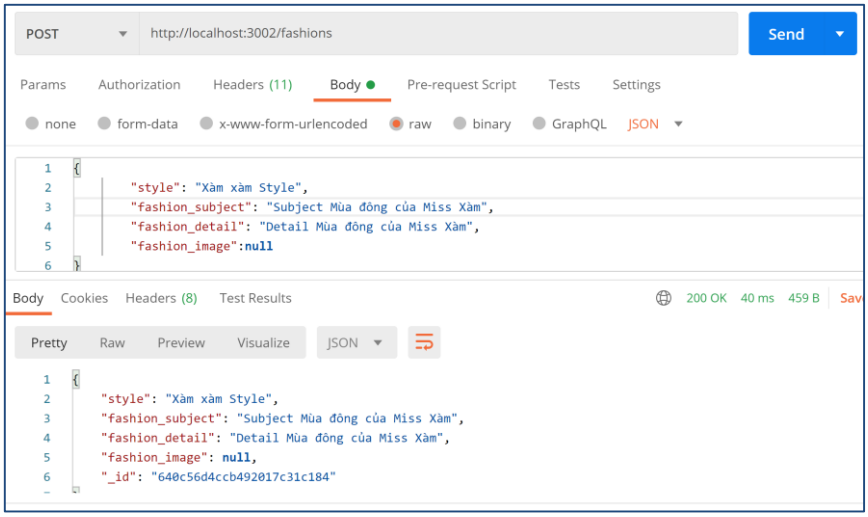
```
const bodyParser=require("body-parser")
app.use(bodyParser.json({ limit: '10mb' }));
app.use(bodyParser.urlencoded({ extended: true, limit: '10mb' }));
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ limit: '10mb' }));
app.use(express.json());
```

The full code of **index.js**:

```
const express = require('express');
const app = express();
const port = 3002;

const morgan=require("morgan")
app.use(morgan("combined"))

const bodyParser=require("body-parser")
app.use(bodyParser.json({ limit: '10mb' }));
app.use(bodyParser.urlencoded({ extended: true, limit: '10mb' }));

app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ limit: '10mb' }));
app.use(express.json());

const cors=require("cors");
```

```
app.use(cors())

app.listen(port,()=>{
    console.log(`My Server listening on port ${port}`)
})

app.get("/",(req,res)=>{
    res.send("This Web server is processed for MongoDB")
})

const { MongoClient, ObjectId } = require('mongodb');
client = new MongoClient("mongodb://127.0.0.1:27017");
client.connect();
database = client.db("FashionData");
fashionCollection = database.collection("Fashion");

app.get("/fashions",cors(),async (req,res)=>{
    const result = await fashionCollection.find({}).toArray();
    res.send(result)
    }
)
app.get("/fashions/:id",cors(),async (req,res)=>{
    var o_id = new ObjectId(req.params["id"]);
    const result = await fashionCollection.find({_id:o_id}).toArray();
    res.send(result[0])
    }
)
app.post("/fashions",cors(),async(req,res)=>{
    //put json Fashion into database
    await fashionCollection.insertOne(req.body)
    //send message to client(send all database to client)
    res.send(req.body)
})
```

Step 2: Use postman to test, if it succeeds go to step 3



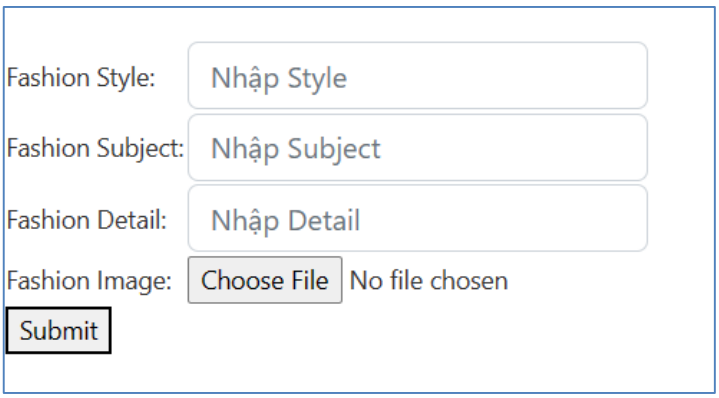
Note that the input is 1 Json without _id (because it is automatically generated), the output of the program will return an input structure with an additional _id along with its value. Such programming makes it easy for us to access the data we just entered through _id.

Input	Output
{ "style": "Xàm xàm Style", "fashion_subject": "Miss Xàm's Winter Subject ", "fashion_detail": "Detail Mùa đông của Miss Xàm", "fashion_image":null }	{ "style": "Xàm xàm Style", "fashion_subject": "Miss Xàm's Winter Subject", "fashion_detail": "Detail Mùa đông của Miss Xàm", "fashion_image": null, "_id": "640c56d4ccb492017c31c184" }

Step 3: In "my-app", add **postFashion** function in **FashionAPIService**

```
postFashion(aFashion:any):Observable<any>
  {
    const headers=new HttpHeaders().set("Content-
Type","application/json;charset=utf-8")
    const requestOptions:Object={
      headers:headers,
      responseType:"text"
    }
    return
this._http.post<any>("/fashions",JSON.stringify(aFashion),requestOptions).pipe(
        map(res=>JSON.parse(res) as Fashion),
        retry(3),
        catchError(this.handleError))
  }
```

Step 4: Add **FashionNewComponent** in "**my-app**", interface design and handle calling a new Fashion:



The file "**fashion-new.component.html**" design the interface as follows:

```
<p>{{errMessage}}</p>
<table>
    <tr>
```

```
        <td>Fashion Style:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel" name="style"
id="style" placeholder="Nhập Style"
            [(ngModel)]="fashion.style"/>

        </td>
    </tr>
    <tr>
        <td>Fashion Subject:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel"
name="fashion_subject" id="fashion_subject" placeholder="Nhập Subject"
            [(ngModel)]="fashion.fashion_subject"/>
        </td>
    </tr>
    <tr>
        <td>Fashion Detail:</td>
        <td>
            <input type="text" class="form-control" #name="ngModel"
name="fashion_detail" id="fashion_detail" placeholder="Nhập Detail"
            [(ngModel)]="fashion.fashion_detail"/>
        </td>
    </tr>
    <tr>
        <td>Fashion Image:</td>
        <td>
            <input type="file" class="file-input"
            #name="ngModel" name="fashion_image" id="fashion_image"
            [(ngModel)]="fashion.fashion_image"
            (change)="onFileSelected($event,fashion)"
            >
        </td>
    </tr>
     <tr>
        <td colspan="2">
            <button (click)="postFashion()">Submit</button>
        </td>
    </tr>
</table>
```

The file "**fashion-new.component.ts**" coding handles converting the physical image into **Base64String** and pushing it to the Server as follows:

```
import { Component } from '@angular/core';
import { from } from 'rxjs';
import { FashionAPIService } from '../fashion-api.service';
import { Fashion } from '../models/Fashion';

@Component({
  selector: 'app-fashion-new',
  templateUrl: './fashion-new.component.html',
  styleUrls: ['./fashion-new.component.css']
})
export class FashionNewComponent {
  fashion=new Fashion();
```

```
  errMessage:string=''
  constructor(private _service: FashionAPIService){

  }
  public setFashion(f:Fashion)
  {
    this.fashion=f
  }
  onFileSelected(event:any,fashion:Fashion) {
    let me = this;
    let file = event.target.files[0];

    let reader = new FileReader();
    reader.readAsDataURL(file);
    reader.onload = function () {
      fashion.fashion_image=reader.result!.toString()
    };
    reader.onerror = function (error) {
      console.log('Error: ', error);
    };
 }

  postFashion()
  {
    this._service.postFashion(this.fashion).subscribe({
      next:(data)=>{this.fashion=data},
      error:(err)=>{this.errMessage=err}
    })
  }
}
```

Configure routing or reference in **app.component.html** to run, we have the desired result.

Exercise 131: Create and invoke API - HTTP PUT – Update a Fashion

Requirement:

Build API to update Fashion. Note that this post is similar to the post to create a new API and add a Fashion.

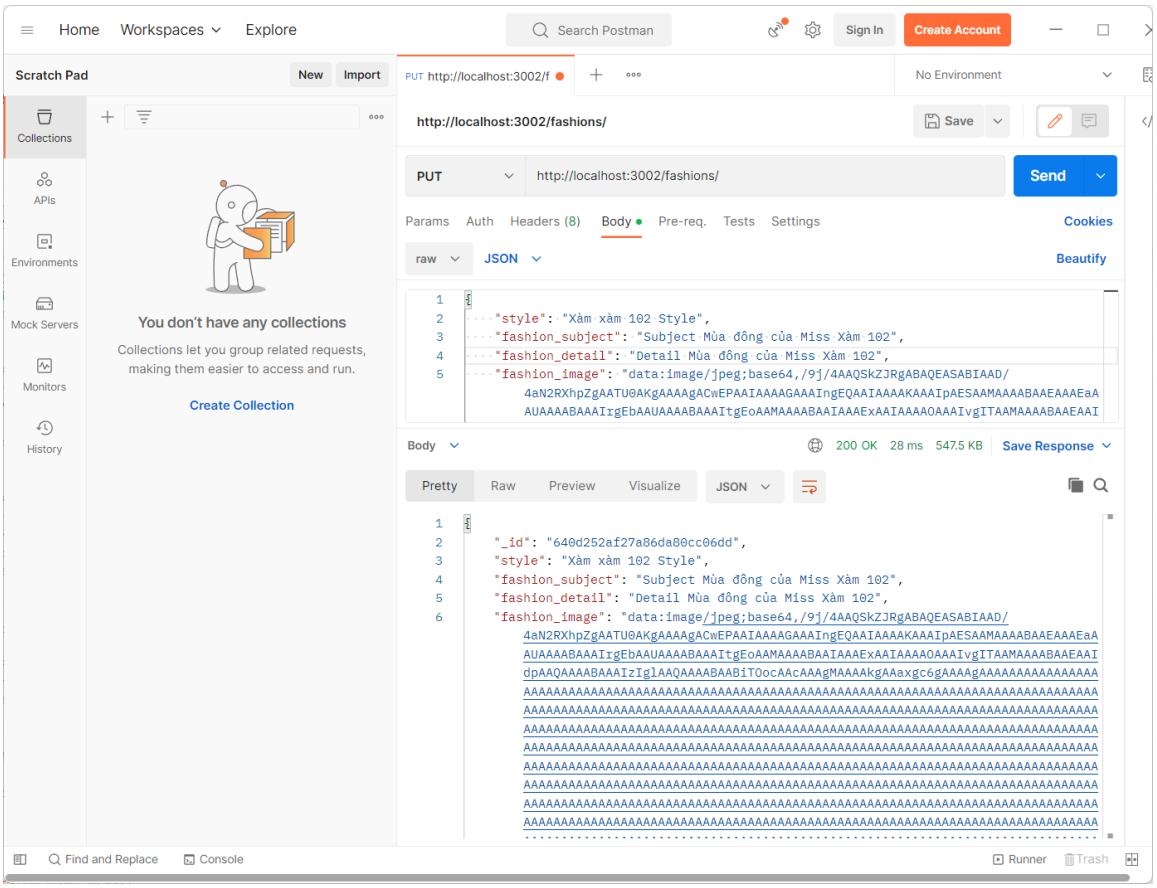
Instructions:

- Everything is similar to the exercise for adding a new Fashion.

- Update “my-server-mongodb”, in the index.js file add the following API:

```
app.put("/fashions",cors(),async(req,res)=>{

    //update json Fashion into database
    await fashionCollection.updateOne(
        {_id:new ObjectId(req.body._id)},//condition for update
        { $set: { //Field for updating
                style: req.body.style,
                fashion_subject:req.body.fashion_subject,
                fashion_detail:req.body.fashion_detail,
                fashion_image:req.body.fashion_image
            }
        }
        )
    //send Fahsion after updating
    var o_id = new ObjectId(req.body._id);
    const result = await fashionCollection.find({_id:o_id}).toArray();
    res.send(result[0])
})
```

The above function updates the information of a Fashion, then returns the updated Fashion. We use Postman to test to make sure the code is written correctly:



The HTML interfaces and services in "**my-app**" do the same as in the previous post, so the documentation does not provide details here (copy it and change the place to call the API).

Exercise 132: Create and invoke API - HTTP DELETE – Remove a Fashion

Requirement:

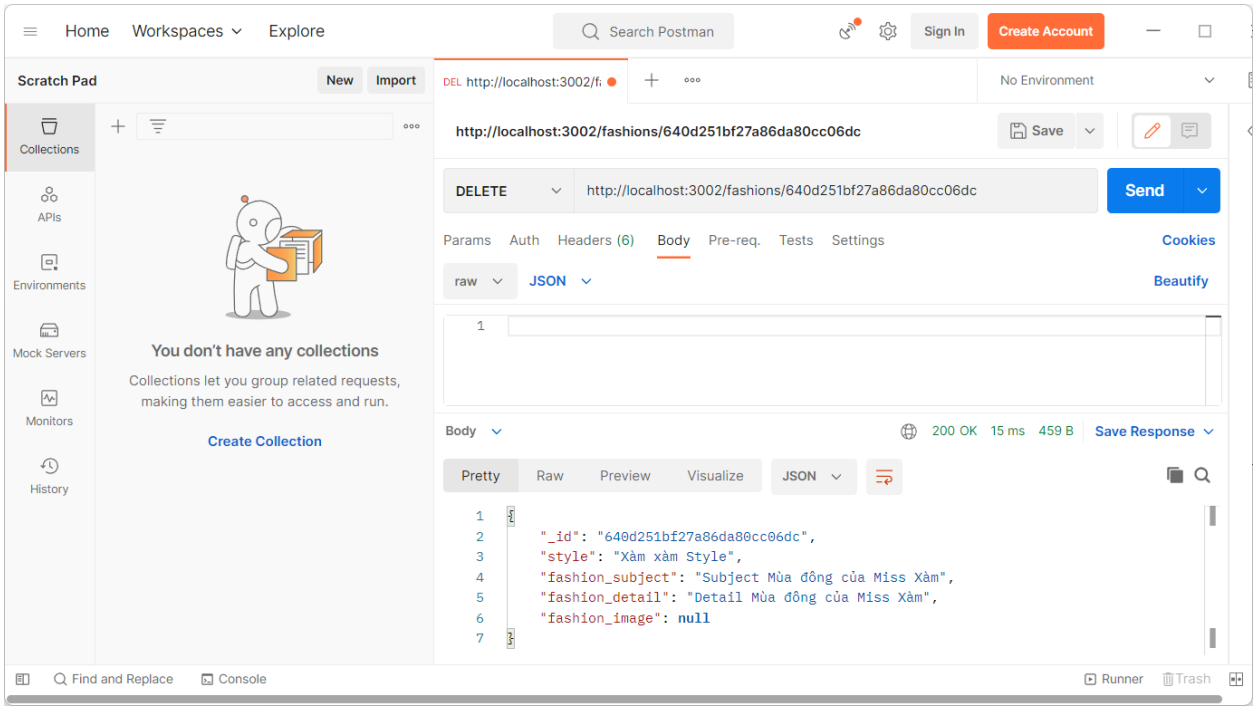
Write an API that deletes a Fashion when **_id** is known

Instructions:

In "**my-server-mongodb**" add **DELETE API** at the end of "**index.js**"

```
app.delete("/fashions/:id",cors(),async(req,res)=>{
    //find detail Fashion with id
    var o_id = new ObjectId(req.params["id"]);
    const result = await fashionCollection.find({_id:o_id}).toArray();
    //update json Fashion into database
    await fashionCollection.deleteOne(
        {_id:o_id}
        )
    //send Fahsion after remove
    res.send(result[0])
})
```

Test Postman to verify the code is correct, if the screen displays the results as below, it means that we have written the correct code (choose the DELETE method):



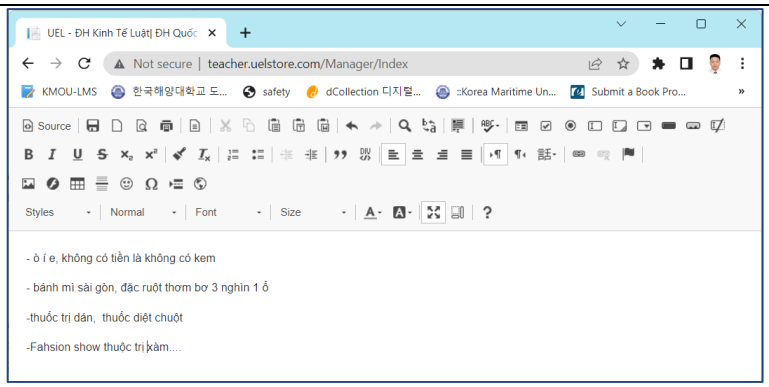
The "**my-app**" part is similar to **BookDeleteComponent** and **BookAPIService (deleteBook(bookId:string):Observable<any>).** The documentation does not detail this delete API call.

Exercise 133: Self-study exercises (*)

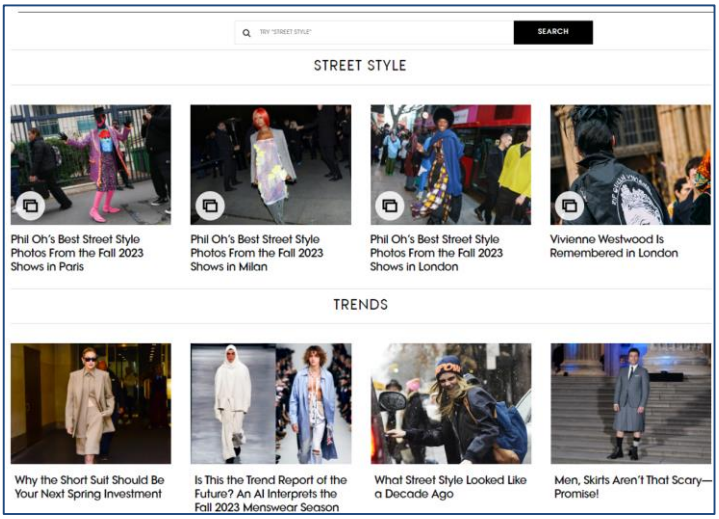
Requirement:

Apply the lessons in this Module to Build a Fashion Website as described:

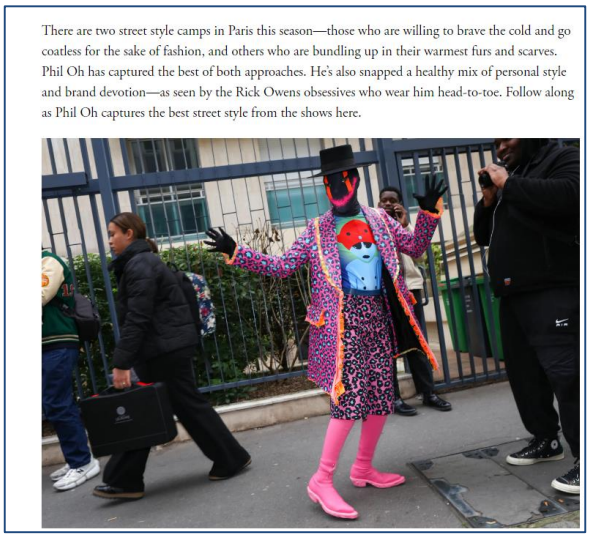
- MongoDB database named **FashionData**, Students need to deduce and propose a suitable data structure to design in MongoDB:
    - There is a single collection named Fashion which includes the following information: ObjectId (automatically generated by MongoDB), fashion title, fashion details, thumbnail, Fashion's Style, creation date.
    - Enter sample data about 3 Styles, each Style has about 3 to 5 fashions.
- Module Restful API (NodeJS, ExpressJS), including APIs (create a project named "**server-fashion**", port **4000**):
    - API returns the entire Fashion, sorted by creation date descending
    - API to filter the list of Fashions according to a certain Style
    - API returns a Fashion based on ObjectId (collectively id)
    - API to add a new Fashion
    - API to edit a Fashion
    - API deletes a Fashion based on id
- Admin module (AngularJs Front End) for system administration, including functions (create a project named "**admin-fashion**", port **4001**):
    - View the entire Fashion list, each row has a view detail button, an edit button, and a delete button (must confirm before deleting).
    - Also in the Fashion list view page there is a button Add a new Fashion.
    - When pressing the edit button or the add new button, a detail screen is displayed to enter information for Fashion, after successful saving, return to the Fashion list view screen, requiring **WYSIWYG** (What You See Is What You Get) **Html Editor** for the fashion detail field, Students find similar tools:



- Client module (AngularJs Front End) to demonstrate, including the following functions (create a project named "**client-fashion**" , port **4002**):
  - Display Fashion in the Style group as shown below, the avatar is the thumbnail attribute.



  - When clicking on each Fashion, it displays details, images and content if it is in the details field entered with **WYSIWYG**:



  - Search for Fashion by Style, users enter Style or choose from dropdownlist, then filter out Fashions of the selected Style.

Module 13: Cookie and Session

Practical knowledge content:

+ Understand the meaning of Cookies and Session
+ Programming Cookies to store client-side data
+ Session programming to store data on the Server side
+ Create cart with Session

Exercise 134: Explain the meaning of Cookie and Session

Requirement:

Please present the meaning of Cookie and Session in e-commerce website systems. Compare the differences in the mechanism of action of these two objects.

List some libraries that support Cookie and Session handling.

Instructions:

- Use **cookie-parser** library to demonstrate how to handle Cookies
- Use **express-session** library to illustrate how to handle Session

Exercise 135: Cookies Programming

Requirement:

Using cookie-parser illustrates the functions:

- Initialize Cookie object
- Add New Cookies
- Retrieve Cookies
- Delete Cookies

Instructions:

Step 1:

- Open project "**my-server-mongodb**"
- Then install **cookie-parser** by commands:

npm install --save cookie-parser

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server-mongodb> npm install --save cookie-parser

added 2 packages, removed 1 package, and audited 120 packages in 865ms

10 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server-mongodb>
```

Step 2:

Edit "**index.js**" for **Cookie** object **initialization** setting:

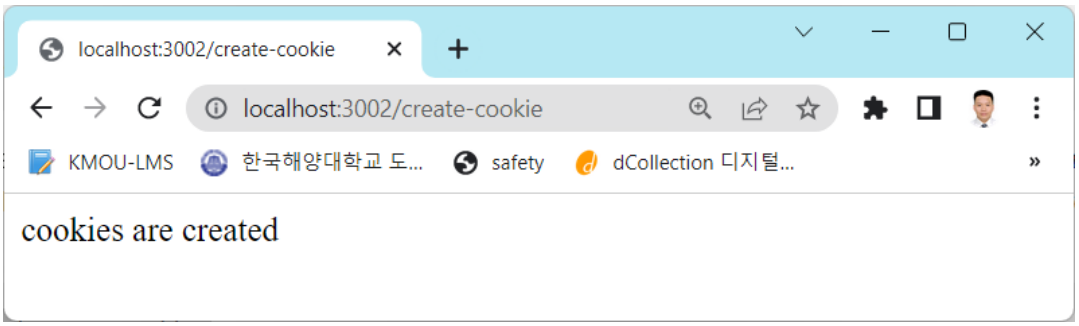
```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

Step 3:

Illustrate an API to create Cookies, single data and **JsonObject** data:

```
app.get("/create-cookie",cors(),(req,res)=>{
    res.cookie("username","tranduythanh")
    res.cookie("password","123456")
    account={"username":"tranduythanh",
            "password":"123456"}
    res.cookie("account",account)
    res.send("cookies are created")
})
```

Initialize the server and test the create-cookie api, we have the result:

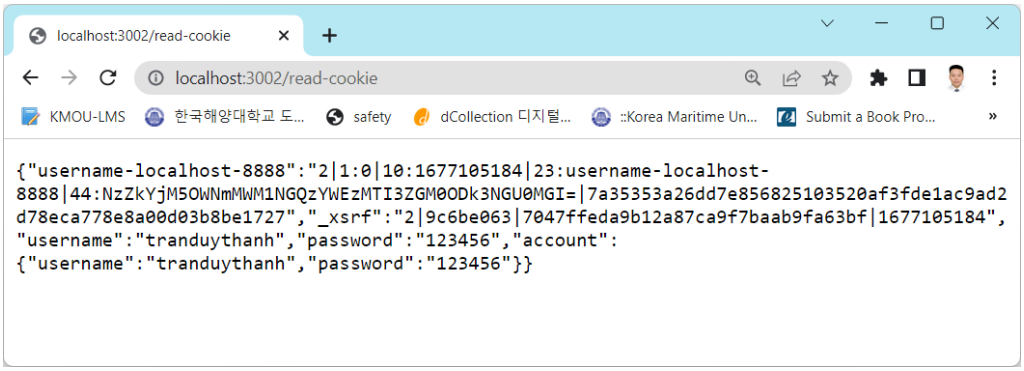


Step 4:

Illustrate an API to read Cookies:

```
app.get("/read-cookie",cors(),(req,res)=>{
    //cookie is stored in client, so we use req
    res.send(req.cookies)
})
```

Run the program and call the api **read-cookie**, we have the result:



Observing the result, it is clear that it is a **JsonObject**, we can easily access each Cookie that we have saved before.

Step 5:

Modify the **read-cookie api** to access and extract each cookie variable that we have saved (note the use of the **req** variable):

```
app.get("/read-cookie",cors(),(req,res)=>{
    //cookie is stored in client, so we use req
    username=req.cookies.username
    password=req.cookies.password
    account=req.cookies.account
    infor="username = "+username+"<br/>"
    infor+="password = "+password+"<br/>"
    infor+="account.username = "+account.username+"<br/>"
    infor+="account.password = "+account.password+"<br/>"
    res.send(infor)
})
```

Run the program and call the **api read-cookie** we have the result:

In addition, to create cookies with timeout (cookies with a period of time), we can code:

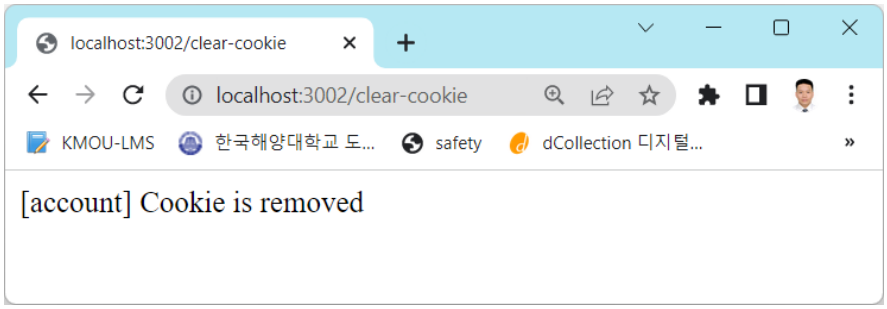
```
 //Expires after 360000 ms from the time it is set.
    res.cookie("infor_limit1", 'I am limited Cookie - way 1', {expire: 360000 +
Date.now()});
    res.cookie("infor_limit2", 'I am limited Cookie - way 2', {maxAge: 360000});
```

Step 6:

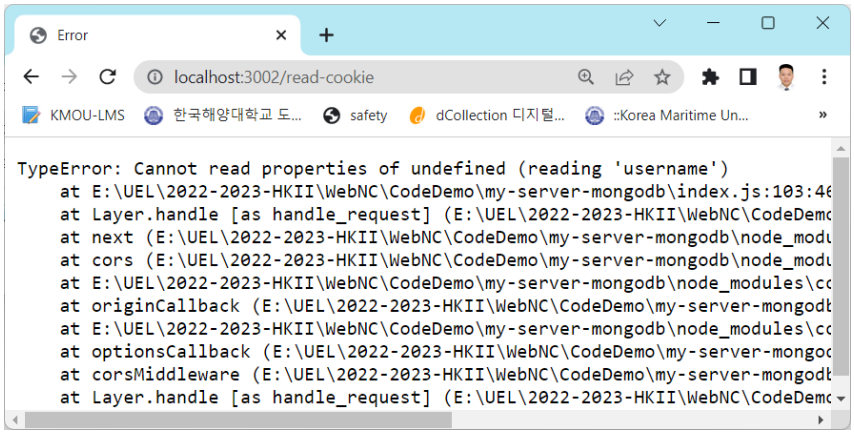
Add clear-cookie api to clear saved Cookies (note: use of **res** variable):

```
app.get("/clear-cookie",cors(),(req,res)=>{
    res.clearCookie("account")
    res.send("[account] Cookie is removed")
})
```

Running the api we get the result:



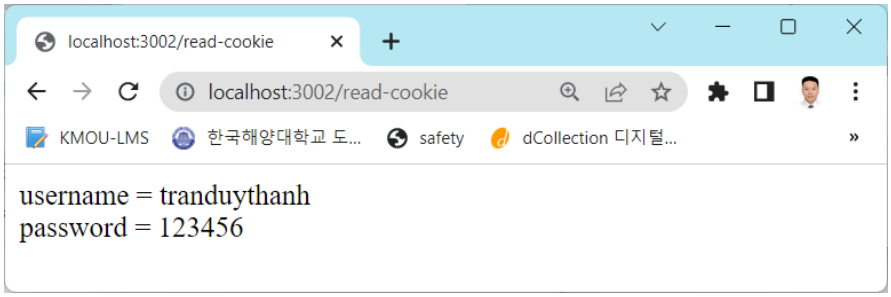
Note that after deleting the Cookie that we tried to access, it was null, and gave an error:



Therefore we need to check the Cookie read out, if it is null, do not process, edit the code:

```
app.get("/read-cookie",cors(),(req,res)=>{
    //cookie is stored in client, so we use req
    username=req.cookies.username
    password=req.cookies.password
    account=req.cookies.account
    infor="username = "+username+"<br/>"
    infor+="password = "+password+"<br/>"
    if(account!=null)
    {
        infor+="account.username = "+account.username+"<br/>"
        infor+="account.password = "+account.password+"<br/>"
    }
    res.send(infor)
})
```

Result after correcting for null check:



At this time, there is no longer an error, the program only displays the cookie of the username and password, and the account cookie has been deleted, it is null, so it is not displayed.

Note Cookies are saved on the client side (stored on the browser side), in any browser, cookies are saved according to that browser and do not understand other browsers. For example, if you run Chrome and save cookies on Chrome, Firefox or Microsoft Edge cannot access it. And remember that even if the Cookie programming is correct, if the user forbids all browsers from storing cookies, our code is meaningless.

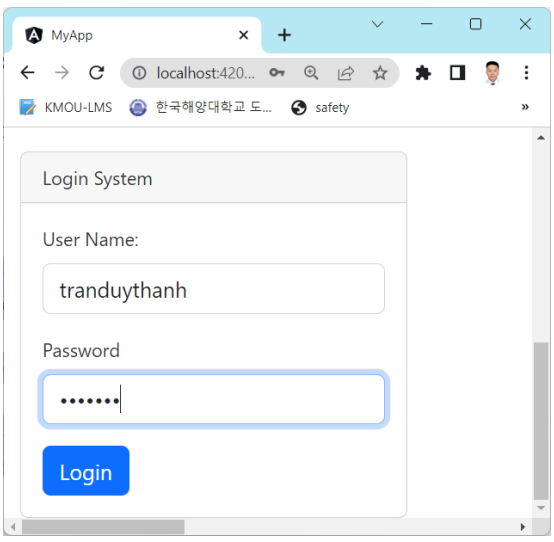
Exercise 136: Programming Cookies – Save login information

Requirement:

In the database "**FashionData**", add a collection named "**User**", it has 2 fields "**username**" and "**password**". Write the Restful API to login, if the login is successful, save the login information Cookie, the next time you open the browser, read the cookie and display this information in the user name and password input boxes.

Instructions:

Create LoginComponent, design the interface as shown below:



Here is the code for "**login.component.html**":

```
<div class="container mt-4">
    <div class="card" style="width: 18rem;">
        <div class="card-header">Login System</div>
        <div class="card-body">
            <form action="/auth/register" method="POST">
                <div class="mb-3">
                    <label for="name-reg" class="form-label">User Name:</label>
                    <input type="text" class="form-control" id="name-reg"
name="name">
                </div>
                <div class="mb-3">
                    <label for="password-reg" class="form-label">Password</label>
                    <input type="password" class="form-control" id="password-reg"
name="password">
                </div>

                <button type="submit" class="btn btn-primary">Login</button>
            </form>
        </div>
    </div>
</div>
```

Ask Students to apply the previous lessons to create a Collection "**User**", import some sample Documents, write a Restful API to log in the **index.js** file of the "**my-server-mongodb**" project. The login API should use the POST method. When the login is successful, save the Cookie, the next time the Login screen is opened, the cookie will be read and displayed on the HTML interface.

Exercise 137: Session Programming

Requirement:

Use **express-session** to demonstrate basic Session usage.

Instructions:

Step 1:

- Open project "**my-server-mongodb**"
- Then install **express-session** by commands:

```
npm install --save express-session
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server-mongodb> npm install --save express-session

added 4 packages, and audited 124 packages in 3s

10 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\UEL\2022-2023-HKII\WebNC\CodeDemo\my-server-mongodb>
```

Step 2:

Edit "**index.js**" to set Session object initialization:

```
var session = require('express-session');
app.use(session({secret: "Shh, its a secret!"}));
```

Step 3:

Illustrate an API to create Session visited

```
app.get("/contact",cors(),(req,res)=>{
    if(req.session.visited!=null)
    {
        req.session.visited++
        res.send("You visited this page "+req.session.visited +" times")
    }
    else
    {
        req.session.visited=1
        res.send("Welcome to this page for the first time!")
    }
})
```

Initialize the server and test the **contact** api, we have the result:

Pressing F5 many times on this page we see the value stored in the session is changed:

Note similar to Cookie, we can store any data structure for Session.

Exercise 138: Session Programming – Self-Study - Shopping Cart(*)

Requirement:

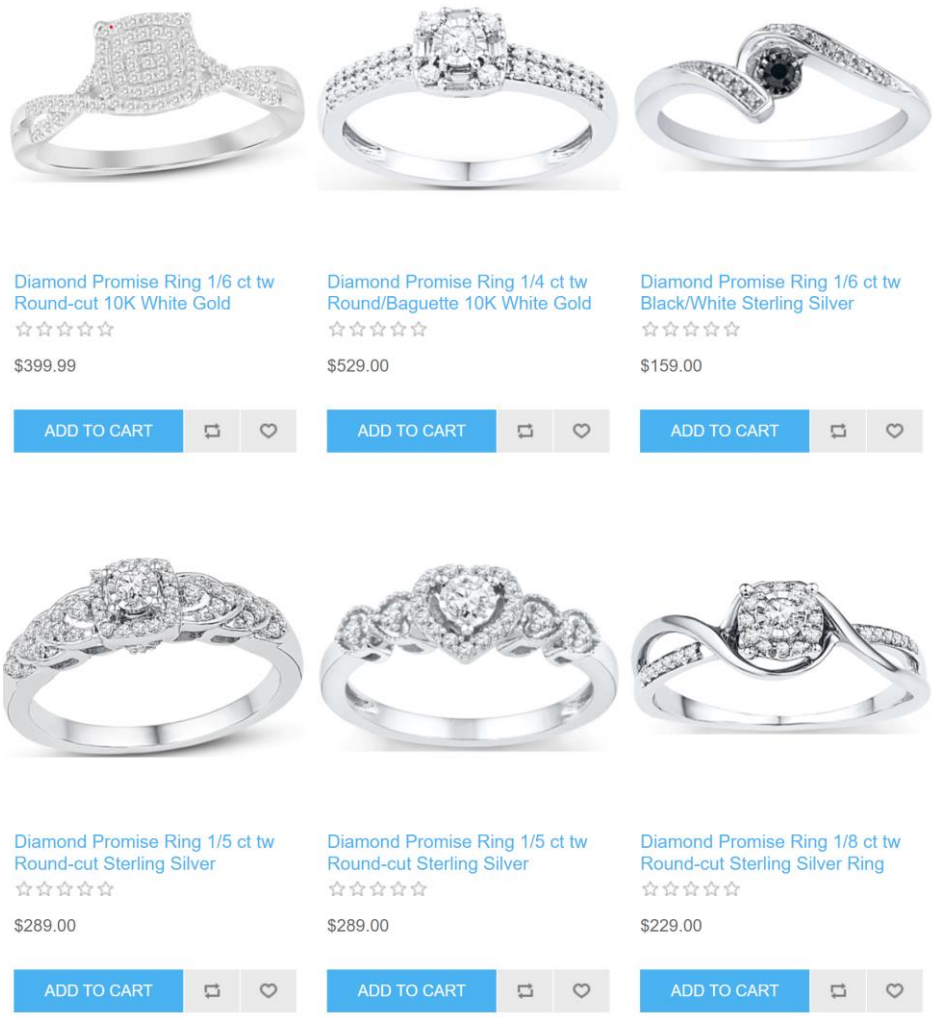
Using Session illustrates cart handling. In e-commerce sites, when the user selects the items and puts them in the cart (haven't actually purchased yet), we use Session to handle temporary storage. Or we still save it to the database in the clipboard to handle the case of unfinished orders: When placing an order, pause, or after placing an order at the payment stage, suspend... all operations we need to save. store to take care of customers (for example, customer X ordered 5 items but the status of the order is still incomplete, after a period of checking the system, the store owner should actively contact to push the order).

In this exercise, students are required to only use Session to temporarily store data of customer operations on the Server (Server memory). Only when the customer confirms the payment will the options included in the order be saved in the database. In this case, when shutting down the Website system, the customer's data stored in the session will usually be lost or after the timeout time it will also be lost. Ask Students to make and illustrate the following descriptions:

- In MongoDB create a collection of Product, the Fields depending on the design needs of the students to create it to their liking.
- Write APIs to handle displaying Product, let customers choose Product and save to Session (This session creates an object variable called **cart** to store JsonArray of Products selected by customer)
- Design interface for customers to choose products to put in the **cart**.
- Design the interface to display Product details in the **cart**

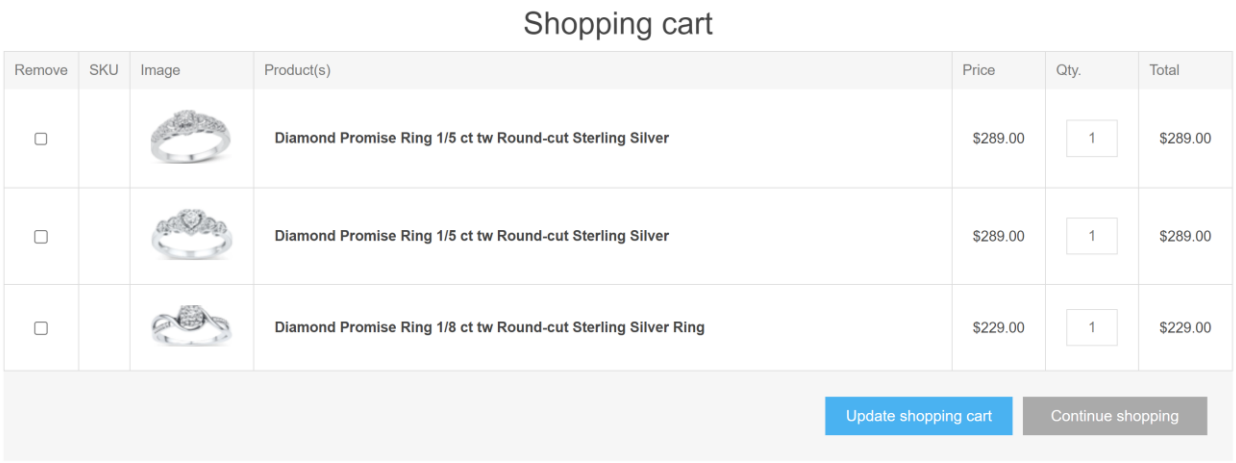
Instructions:

Example of Product display screen for customers to choose to buy:



Every time you click "**ADD TO CART**", the product will be added to the cart (variable **cart**).

When entering the cart view function, display the interface as below:



-Allow checked Product to remove from cart, update quantity ➔ click "Update shopping cart" to update cart.

-Want to buy more (return to Product screen) => click "Continue shopping"