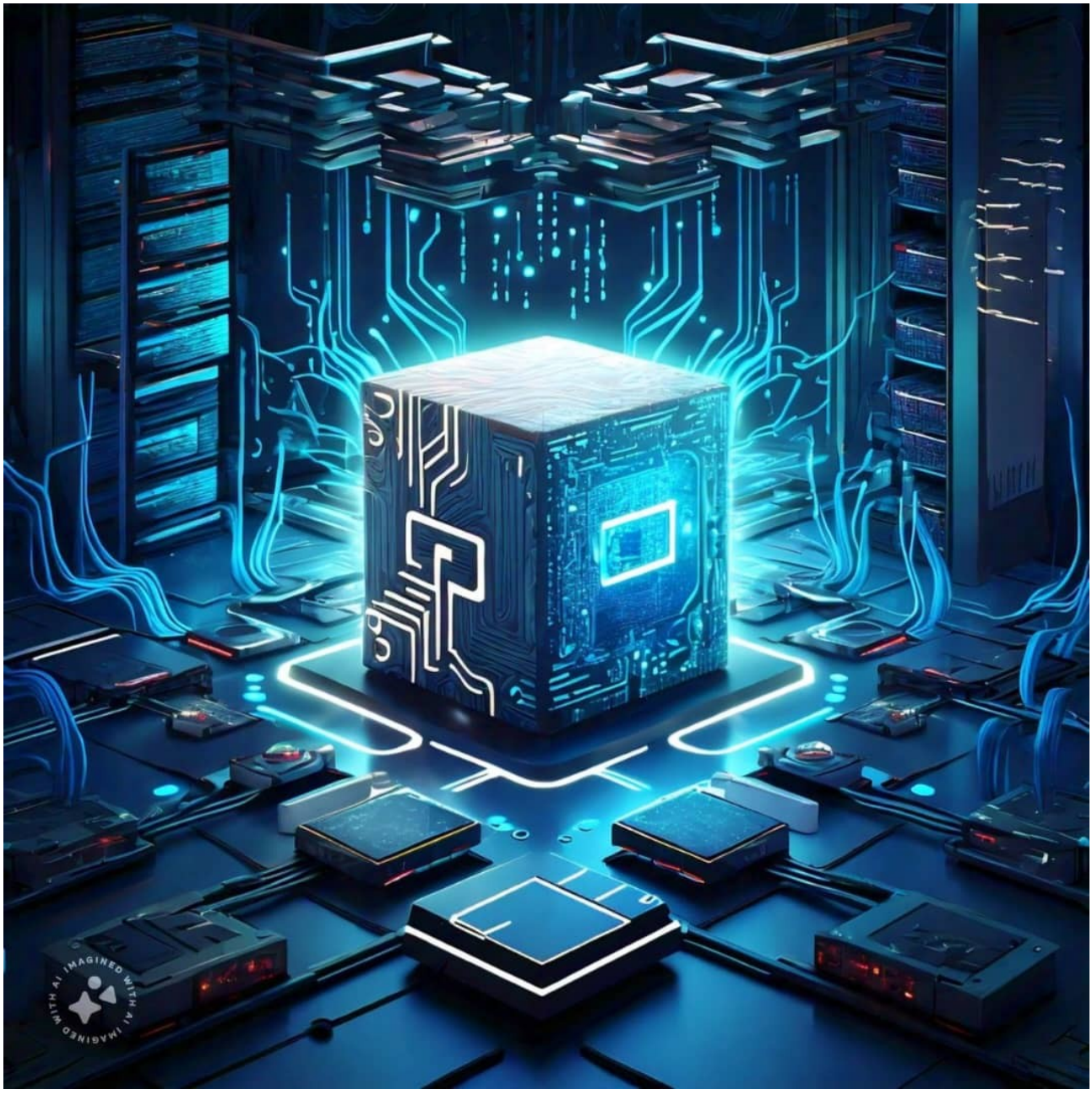
Hydot Backend Tutorial



Master 5 Popular Backend Frameworks Step By Step

(1) Dotnet mvc and Api
(2) Laravel mvc and Api
(3) Spring boot mvc and Api
(4) Django mvc and Api
(5) Express Node Api

DOTNET MVC

[Start]

From the terminal run this command **dotnet new mvc -o ProjectName**
Open the solution in VSCode **code ProjectName**

When Visual Studio Code requests that you add assets to build and debug the project, select **Yes**. If Visual Studio Code doesn't offer to add build and debug assets, select **View** > **Command Palette** and type ".NET" into the search box. From the list of commands, select the .NET: Generate Assets for Build and Debug command.

Visual Studio Code adds a .vscode folder with generated launch.json and tasks.json file

Trust the HTTPS development certificate by running the following command:
dotnet dev-certs https --trust

In vscode, press ctrl + f5 to run the app without debugging

[Application]

The typical flow of the application is Model View Controller(MVC) and each of them play a critical role
Model: It defines the data schema that will be used to write data into the database.
View: It defines the UI of the application
Controller: It correlate the Model and View Activity. It stores the application logic and process the users request, it then response to the user request with a view.

Ui logic belongs to the View, Input logic belongs to the Controller and the business logic belongs to the model

Create a new file name it HelloWorld.cs and paste this content into it

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController: Controller{

public string Index(){

return "Hello World";
}

public string Welcome(){
return "Welcome";
}



}
```

Every public method is a Http endpoint.

The url is https://localhost:7111/HelloWorld
https => The protocol used
localhost:7111 => The server and port running the application
HelloWorld => The controller Name, [HelloWorldController], inside this controller, we have the Index() method. This index method will be the initial method which will be executed when the user access the HelloWorld controller
The Index() is the base url.

https://localhost:7111/HelloWorld/Welcome
Now for this example, we are moving into the welcome method Welcome() to execute the code there. The controller name is HelloWorld and the specific method name is Welcome

/[Controller]/[ActionName]/[Parameters]

In the program.cs, the default route is

```
app.MapControllerRoute(
name: "default",
pattern: "{controller=Home}/{action=Index}/{id?}"
);
```

We can modify it so that the default route will be HelloWorldController

```
app.MapControllerRoute(
name: "default",
pattern: "{controller=HelloWorld}/{action=Index}/{id?}"
);
```

Parameters
From the welcome method in the HelloWorld controller add this

```
public string Welcome(string name, int numTimes = 1)
{
return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

In your browser do this https://localhost:7111/HelloWorld/Welcome?name=Rick&numtimes=4

Add A View

In the View folder, create a folder called HelloWorld. Note that the folder name in the View folder should match with the controller name.

If the controller name is MoneyController, the folder name should be Money and so on

In the HelloWorld folder, create an Index.cshtml which is a razor view file,

The code should be this

```
@{
ViewData["Title"] = "Index";
}
```

```
<h2>This is the index file</h2>

<p>Hello from the view template!</p>
```

Now in the HelloWorldController, update the Index()

```
public IActionResult Index(){

return View();
}
```

Now this code will do this, It will move into the View Folder > HelloWorld Folder > Index.cshtml

Layout

In the shared folder, there is a _Layout.cshtml file that will act as the root of the application. Now in this file, the section is divided into 3 main section

<header> </header>

<main> </main>

<footer> </footer>

The header contains the normal header of the application, it can be modified

```
<header>
<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
<div class="container-fluid">
<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>
<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
aria-expanded="false" aria-label="Toggle navigation">
<span class="navbar-toggler-icon"></span>
</button>
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
<ul class="navbar-nav flex-grow-1">
<li class="nav-item">
<a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
</li>
<li class="nav-item">
<a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
</li>
</ul>
</div>
</div>
</nav>
</header>
```

The main section will display all the route i will enter

```
<div class="container">
<main role="main" class="pb-3">
@RenderBody()
</main>
</div>
```

@RenderBody() => RenderBody is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the Privacy link, the *Views/HelloWorld/Index.cshtml* view is rendered inside the RenderBody method.

The Footer section will contain the normal footer.

```
<footer class="border-top footer text-muted">
<div class="container">
&copy; 2024 - MvcMovie - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
</div>
</footer>
```

Good, now the scripts resides in the wwwroot folder. The [~] sign means wwwroot folder

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
```

Okay, the Views/_ViewStart.cshtml contain this code

```
@{
Layout = "_Layout";
}
```

To display the title of the page, modify the Index.cshtml

```
@{
ViewData["Title"] = "Movie List";
}


<title>@ViewData["Title"]-Movie App</title>

<h2>This is the index file</h2>

<p>Hello from the view template!</p>
```

Passing Data from controller to view

Modify the Welcome() in the HelloWorld.cshtml controller

```
public IActionResult Welcome(string name, int numTimes)
{
ViewData["Message"] = "Hello, " + name;
ViewData["NumTimes"] = numTimes;

return View();
}
```

Now in the Views folder, create Welcome.cshtml and add this code

```
@{
ViewData["Title"] = "Welcome";
}

<title>@ViewData["Title"]</title>

<ul>
@for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
{
<li>@ViewData["Message"]</li>
}
</ul>
```

Now Visit this url https://localhost:7111/HelloWorld/Welcome?name=Rick&numtimes=4
It will print Rick 4 times

Data is taken from the URL and passed to the controller using the MVC model binder. The controller packages the data into a ViewData dictionary and passes that object to the view. The view then renders the data as HTML to the browser.

Add Models

We need to define poco (Plain old clr objects) class

```
using System.ComponentModel.DataAnnotations;
namespace MvcMovie.Models;

public class Movie{
public int Id { get; set; }
public string? Title { get; set; }
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }
public string? Genre { get; set; }
public decimal Price { get; set; }


}
```

Install the necessary nugget packages,

dotnet tool install --global dotnet-aspnet-codegenerator
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SQLite
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools

Create a database
mssql -u sa -p Password

Create a data folder and add the datacontext file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using MvcMovie.Models;
using Microsoft.EntityFrameworkCore;




namespace MvcMovie.Data
{
public class DataContext:DbContext
{
//Empty constructor
public DataContext(): base(){
}

//Database Connection String
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
base.OnConfiguring(optionsBuilder);
optionsBuilder.UseSqlServer("Server=localhost,1433;Database=learnDOTNETMvc;User=sa;Password=Hydo
tTech;TrustServerCertificate=true;");
}
//Data Set, where Project and User are models in the Model folder

//public DbSet<Attitude> Attitudes {get; set;}



}
}
```

Import it into the Program.cs

```
using Microsoft.EntityFrameworkCore;
using MvcMovie.Data;
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>();


// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
app.UseExceptionHandler("/Home/Error");
// The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
name: "default",
pattern: "{controller=Home}/{action=Index}/{id?}"
);

app.Run();
```

Run this to scaffold the project

```
dotnet aspnet-codegenerator controller -name MovieController -m Movie -dc
LearnDotnetMVC.Data.MyDatabase --relativeFolderPath Controllers --useDefaultLayout --
referenceScriptLibraries -sqlite
```

-m => The name of the model.
-dc => The Data context
--relativeFolderPath => The relative output folder path to create the files.
--useDefaultLayout => The default layout should be used for the views.
--referenceScriptLibraries => Adds _ValidationScriptsPartial to Edit and Create pages.
-sqlite => Flag to specify if DbContext should use SQLite instead of SQL Server.

In the program.cs, add this configuration

```
builder.Services.AddDbContext<LearnDotnetMVCcontext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("MyDatabase")));
```

In the appsetting.json, add the following configuration to connect to your database

```
{
"Logging": {
"LogLevel": {
"Default": "Information",
"Microsoft.AspNetCore": "Warning"
}
},
"AllowedHosts": "*",
"ConnectionStrings": {
"MyDatabase":
"Server=localhost,1433;Database=learnDOTNETMvc;User=sa;Password=HydotTech;TrustServerCertificate=
true;"
}
}
```

From Visual Studio, install the following nugget packages

- Microsoft.VisualStudio.Web.CodeGeneration.Design
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.Tools
- Microsoft.EntityFrameworkCore.SQLite

To migrate the table, run
dotnet ef migrations add InitialCreate
dotnet ef database update

Datacontext File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using LearnDotnetMVC.Models;

namespace LearnDotnetMVC.Data
{
public class LearnDotnetMVCcontext : DbContext
{
```

```
public LearnDotnetMVCcontext (DbContextOptions<LearnDotnetMVCcontext> options)
: base(options)
{
}

public DbSet<Movie> Movie { get; set; } = default!;
}
}
```

The DbSet is used to define the table name to be connected to the database and their respective model

MovieController

Inside the MovieController, the datacontext is injected into the constructor

```
private readonly LearnDotnetMVCcontext _context;

public MovieController(LearnDotnetMVCcontext context)
{
_context = context;
}
```

Inside the Details method

```
// GET: Movie/Details/5
public async Task<IActionResult> Details(int? id)
{
if (id == null)
{
return NotFound();
}

var movie = await _context.Movie
.FirstOrDefaultAsync(m => m.Id == id);
if (movie == null)
{
return NotFound();
}

return View(movie);
}
```

The var movie will query the database through `_context.Movie .FirstOrDefaultAsync(m => m.Id == id);` It will query the Movie table and search for a movie whose id is in the parameter `Details(int? id),` If there is no such movie id, it will return null, else it will return a view with the movie in a json format

The movie generated will then be passed into Details.cshtml in the /Views/Movie folder

```
@model LearnDotnetMVC.Models.Movie

@{
```

```
ViewData["Title"] = "Delete";
}

<h1>Delete</h1>

<h3>Are you sure you want to delete this?</h3>
<div>
<h4>Movie</h4>
<hr />
<dl class="row">
<dt class = "col-sm-2">
@Html.DisplayNameFor(model => model.Title)
</dt>
<dd class = "col-sm-10">
@Html.DisplayFor(model => model.Title)
</dd>
<dt class = "col-sm-2">
@Html.DisplayNameFor(model => model.ReleaseDate)
</dt>
<dd class = "col-sm-10">
@Html.DisplayFor(model => model.ReleaseDate)
</dd>
<dt class = "col-sm-2">
@Html.DisplayNameFor(model => model.Genre)
</dt>
<dd class = "col-sm-10">
@Html.DisplayFor(model => model.Genre)
</dd>
<dt class = "col-sm-2">
@Html.DisplayNameFor(model => model.Price)
</dt>
<dd class = "col-sm-10">
@Html.DisplayFor(model => model.Price)
</dd>
</dl>
<form asp-action="Delete">
<input type="hidden" asp-for="Id" />
<input type="submit" value="Delete" class="btn btn-danger" /> |
<a asp-action="Index">Back to List</a>
</form>
</div>
```

The @model statement at the top of the view file specifies the type of object that the view expects.

Simply put, the @model is coming from the controller and it has a type of LearnDotnetMVC.Models.Movie

The Index() method in the controller
public async Task<IActionResult> Index()

```
{
return View(await _context.Movie.ToListAsync());
}
```

It will generate a list and feed the index.cshtml

```
@model IEnumerable<LearnDotnetMVC.Models.Movie>

@{
ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
<a asp-action="Create">Create New</a>
</p>
<table class="table">
<thead>
<tr>
<th>
@Html.DisplayNameFor(model => model.Title)
</th>
<th>
@Html.DisplayNameFor(model => model.ReleaseDate)
</th>
<th>
@Html.DisplayNameFor(model => model.Genre)
</th>
<th>
@Html.DisplayNameFor(model => model.Price)
</th>
<th></th>
</tr>
</thead>
<tbody>
@foreach (var item in Model) {
<tr>
<td>
@Html.DisplayFor(modelItem => item.Title)
</td>
<td>
@Html.DisplayFor(modelItem => item.ReleaseDate)
</td>
<td>
@Html.DisplayFor(modelItem => item.Genre)
</td>
<td>
@Html.DisplayFor(modelItem => item.Price)
</td>
```

```
<td>
<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
<a asp-action="Details" asp-route-id="@item.Id">Details</a> |
<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
</td>
</tr>
}
</tbody>
</table>
```

@Html.DisplayNameFor(model => model.Title), It will display the field name as a label and an input field

@Html.DisplayFor(modelItem => item.Title), it will display only the content

The Rest

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using LearnDotnetMVC.Data;
using LearnDotnetMVC.Models;

namespace LearnDotnetMVC.Controllers
{
public class MovieController : Controller
{
private readonly LearnDotnetMVCcontext _context;

public MovieController(LearnDotnetMVCcontext context)
{
_context = context;
}

// GET: Movie
public async Task<IActionResult> Index()
{
return View(await _context.Movie.ToListAsync());
}

// GET: Movie/Details/5
public async Task<IActionResult> Details(int? id)
{
if (id == null)
{
return NotFound();
}
```

```
var movie = await _context.Movie
.FirstOrDefaultAsync(m => m.Id == id);
if (movie == null)
{
return NotFound();
}

return View(movie);
}

// GET: Movie/Create
public IActionResult Create()
{
return View();
}

// POST: Movie/Create
// To protect from overposting attacks, enable the specific properties you want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("Id,Title,ReleaseDate,Genre,Price")] Movie movie)
{
if (ModelState.IsValid)
{
_context.Add(movie);
await _context.SaveChangesAsync();
return RedirectToAction(nameof(Index));
}
return View(movie);
}

// GET: Movie/Edit/5
public async Task<IActionResult> Edit(int? id)
{
if (id == null)
{
return NotFound();
}

var movie = await _context.Movie.FindAsync(id);
if (movie == null)
{
return NotFound();
}
return View(movie);
}

// POST: Movie/Edit/5
```

```
// To protect from overposting attacks, enable the specific properties you want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("Id,Title,ReleaseDate,Genre,Price")] Movie movie)
{
if (id != movie.Id)
{
return NotFound();
}

if (ModelState.IsValid)
{
try
{
_context.Update(movie);
await _context.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException)
{
if (!MovieExists(movie.Id))
{
return NotFound();
}
else
{
throw;
}
}
return RedirectToAction(nameof(Index));
}
return View(movie);
}

// GET: Movie/Delete/5
public async Task<IActionResult> Delete(int? id)
{
if (id == null)
{
return NotFound();
}

var movie = await _context.Movie
.FirstOrDefaultAsync(m => m.Id == id);
if (movie == null)
{
return NotFound();
}

return View(movie);
```

```
}

// POST: Movie/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
var movie = await _context.Movie.FindAsync(id);
if (movie != null)
{
_context.Movie.Remove(movie);
}

await _context.SaveChangesAsync();
return RedirectToAction(nameof(Index));
}

private bool MovieExists(int id)
{
return _context.Movie.Any(e => e.Id == id);
}
}
}
```

The actions for each is defined

```
<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
<a asp-action="Details" asp-route-id="@item.Id">Details</a>
| <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
```

I will throw more light on subsequent Lectures from scratch

Search

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title!.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

Add this to the frontend

```
<form asp-controller="Movie" asp-action="Index">

 <p>
Title: <input type="text" name="SearchString" />
<input type="submit" value="Filter" />
 </p>

</form>
```

The code will go into the Movie controller and execute the Index method. The index method accept a searchString parameter, it will query the database comparing the title with the search string if there is some match, if yes it will return some data

```
public async Task<IActionResult> Index(string searchString)
{
if (_context.Movie == null)
{
return Problem("Entity set 'MvcMovieContext.Movie' is null.");
}

var movies = from m in _context.Movie
select m;

if (!String.IsNullOrEmpty(searchString))
{
movies = movies.Where(s => s.Title!.Contains(searchString));
}

return View(await movies.ToListAsync());
}
```

The movie list will then be pushed into the view to be looped and to be displayed

```
@foreach (var item in Model) {
<tr>
<td>
@Html.DisplayFor(modelItem => item.Title)
</td>
<td>
@Html.DisplayFor(modelItem => item.ReleaseDate)
</td>
<td>
@Html.DisplayFor(modelItem => item.Genre)
</td>
<td>
@Html.DisplayFor(modelItem => item.Price)
</td>
<td>
<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
<a asp-action="Details" asp-route-id="@item.Id">Details</a> |
<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
```

```
</td>
</tr>
}
```

Validation

In the poco table, we can add some input validation like this

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string? Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    [Required]
    [StringLength(30)]
    public string? Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string? Rating { get; set; }
}
```

For GET, there is no need to define a HTTP method name, but you need to do that for post

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{

    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
```

```
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    if (movie != null)
    {
        _context.Movie.Remove(movie);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Project Work Link
Not Yet

Express Js

With express.js, there is no need for additional configurations. You can get your api up and running in 30 seconds

Personally, I will use ExpressJS for building Realtime applications like chat systems, video calls, notifications, and other applications that needs speed. I will not use it for creating complex applications like Mall systems, Banking systems and others, because if not organized correctly, it can be messy real quick

Let start the tutorial

[Installation]

```
npm init -y
npm install express
```

Configure the Package.json to look like this

package.json

```
{
"name": "expressjs",
"version": "1.0.0",
"description": "",
"main": "index.mjs",
"scripts": {
```

```
"test": "echo \"Error: no test specified\" && exit 1",
"start":"nodemon ./src/index.mjs"
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
"express": "^4.19.2"
},
"devDependencies": {
"nodemon": "^3.1.0"
},
"type":"module"
}
```

Also create an **index.mjs** and paste this code in it

```
import express from "express"


const app = express()

const PORT = process.env.PORT|| 3000;


app.listen(PORT, ()=>{
console.log(`Server running on port ${PORT}`)
})
```

[Routing]

Routing is essential for the application navigation, without much a do, let start

GET REQUEST

```
import express from "express"

const app = express()

const PORT = process.env.PORT|| 3000;


app.get("/",(req,res)=>{
res.status(200).send({msg:"Kofi Money"})
})

const users = [
```

```
{
"id":1,
"fname":"Solo",
},

{
"id":2,
"fname":"Danso",
},

{
"id":3,
"fname":"Kofi",
},









]
app.get("/api/users",(req,res)=>{
res.status(200).send({users})
})



app.listen(PORT, ()=>{
console.log(`Server running on port ${PORT}`)
})
```

Route Parameters

The parameters are the values we additional parameters we add to the url
for Example
http://localhost:3000/api/users/2
2 is a parameter

```
import express from "express"


const app = express()

const PORT = process.env.PORT|| 3000;
```

```
app.get("/",(req,res)=>{
res.status(200).send({msg:"Kofi Money"})
})

const mockUsers = [
{
"id":1,
"fname":"Solo",
},

{
"id":2,
"fname":"Danso",
},

{
"id":3,
"fname":"Kofi",
},

]


app.get("/api/users",(req,res)=>{
res.status(200).send({msg:mockUsers})
})

app.get("/api/users/:id",(req, res)=>{
const parsedId = parseInt(req.params.id)

if(isNaN(parsedId)){
return res.status(400).send("Bad Request ")
}


const findUser = mockUsers.find((user)=>user.id==parsedId)
if(findUser ==null){
return res.status(400).send({msg:"User not found"})
}
else{
return res.status(200).send(findUser);
}
```

```
})



app.listen(PORT, ()=>{
console.log(`Server running on port ${PORT}`)
})
```

Query Selectors

Query parameters are the additional parameters added to the URL. They exist after the question mark (?)

For example
http://localhost:3000/api/users?search=Solo
{ search: 'Solo' } is the query parameters

For this
http://localhost:3000/api/users?filter=fname&value=Solo
from this Data

```
const mockUsers = [
{
"id":1,
"fname":"Solo",
},

{
"id":2,
"fname":"Danso",
},

{
"id":3,
"fname":"Kofi",
},

]
```

This code will filter the fname field for the value provided

```
app.get("/api/users",(req,res)=>{
//Deconstructing query from request object
```

```
const {query:{filter, value}} = req;
if(!filter && !value){
return res.send({mockUsers})
}
if(filter && value){
return res.send(
mockUsers.filter((data)=>data[filter].includes(value))
)
}

return res.send({mockUsers})

})
```

Install Thunder Client extension for vscode

POST Request

First, register your middleware, right at the top after the app constant is assigned to express

```
app.use(express.json());
```

Create a small post method

```
app.post("/api/users",(req,res)=>{
//Capture all the body from req
const {body} = req;

//Create a newUser Object
//id: get the index of the last object in the mockUser array
//Get the id and auto increase by 1

//...body means pass all the body defined in the frontend into it
const newUser = {
id: mockUsers[mockUsers.length-1].id+1,
...body
}

mockUsers.push(newUser)

res.status(201).send(mockUsers)
})
```

PUT Request

The key difference between a PUT and a PATCH request is,

PUT update the entire record, every single field
PATCH update just a portion of the record

```
app.put("/api/users/:id",(req,res)=>{

const {body, params:{id}, } = req;

const Id = parseInt(id)
if(isNaN(id)){
return res.status(400).send("Id is not a number")
}

//This will find the index of the object in the array where it id is equal to the id provided
const findUserIndex = mockUsers.findIndex( (data)=>data.id===Id)

if(findUserIndex === -1){
return res.status(404).send({msg:"User not found"})
}
/*
localhost:3001/api/users/2
return res.send(findUserIndex.toString())
the index is 1

return res.send(mockUsers[findUserIndex])
the result is
{
"id": 2,
"fname": "Danso"
}



*/

const old = mockUsers[findUserIndex]
mockUsers[findUserIndex] = {id:Id, ...body}

const wow = [{
"Old":old
},{"New":mockUsers[findUserIndex]}]


return res.send(wow)


})
```

The update data is
{
"username":"Solomon 1",
"displayName":"Solomon The CTO"
}

The Result
[
{
"Old": {
"id": 2,
"fname": "Danso"
}
},
{
"New": {
"id": 2,
"username": "Solomon 1",
"displayName": "Solomon The CTO"
}
}
]

PATCH Request

Patch is more useful than PUT because we update only specific fields

```
app.patch("/api/users/:id",(req,res)=>{

const {body, params:{id}, } = req;

const Id = parseInt(id)
if(isNaN(id)){
return res.status(400).send("Id is not a number")
}

//This will find the index of the object in the array where it id is equal to the id provided
const findUserIndex = mockUsers.findIndex( (data)=>data.id===Id)

if(findUserIndex === -1){
return res.status(404).send({msg:"User not found"})
}

mockUsers[findUserIndex] = {...mockUsers[findUserIndex], ...body};

/*
```

mockUsers[findUserIndex] = {...mockUsers[findUserIndex], ...body};
Explanation:

mockUsers[findUserIndex] => This will pick the user account based on the id provided
{...mockUsers[findUserIndex], ...body};

...mockUsers[findUserIndex] => This will hold the current values example {id:1, fname:Solo}
...body => This will hold the new values in the body of the request

Now this is what will happen, if the body contains a field, then the field of the old value will be updated

*/

return res.send(mockUsers[findUserIndex])

})

DELETE Request

app.delete("/api/users/:id",(req, res)=>{

const {params:{id} } = req;
const Id = parseInt(id)
const uIndex = mockUsers.findIndex((data)=>data.id===Id);

if(uIndex<0){
return res.send({msg:"User not found"})
}

if(isNaN(Id)){
return res.send({msg:"Imvalid Id"})
}
/*
Find the array of the element you want to delete
const uIndex = mockUsers.findIndex((data)=>data.id===Id);

Delete the array and make the count 1, otherwise it will delete all the
mockUsers.splice(uIndex,1)

*/
mockUsers.splice(uIndex,1)

```
return res.send(mockUsers)

})
```

MIDDLEWARE

Middleware is very essential, we can assign it to each route

For example
This middleware is enabled globally

```
const loggingMiddleware = (req,res,next) =>{
console.log(`${req.method} - ${req.url}`)

next();
}

app.use(loggingMiddleware)
```

In the terminal, the output will be
GET /api/users

For example
This middleware is only allowed on this endpoint

```
app.get("/api/users/:id",loggingMiddleware,(req, res)=>{
const parsedId = parseInt(req.params.id)

if(isNaN(parsedId)){
return res.status(400).send("Bad Request ")
}


const findUser = mockUsers.find((data)=>data.id==parsedId)
if(findUser ==null){
return res.status(400).send({msg:"User not found"})
}
else{
return res.status(200).send(findUser);
}


})
```

The middleware is passed as an argument,

The next() in the middleware basically means "Call the next function to be executed if the middleware code is done executing"

We can also customise the middleware for a particular endpoint

```
app.get("/",(req,res,next)=>{

console.log("Kofi Nkamkam")

},(req,res)=>{
res.status(200).send({msg:"Kofi Money"})
})
```

In this example, the next() is not called, Kofi Nkamkam will be console logged but the frontend will not respond (It will be stucked)

We can even add multiple middleware to one endpoint

```
app.get("/",
(req,res,next)=>{

console.log("First Middleware")
next()

},
(req,res,next)=>{

console.log("Second Middleware")
next()

},
(req,res,next)=>{

console.log("Third Middleware")
next()

},
(req,res)=>{
res.status(200).send({msg:"Kofi Money"})
})
```

Output
First Middleware
Second Middleware
Third Middleware

For Global Middlewares, make sure the middleware is defined right after you have initialised the app variable
Order matters

In the app.use() we can pass multiple middlewares

```
const loggingMiddleware = (req,res,next) =>{
console.log(`${req.method} - ${req.url}`)

next();
}

const signMiddleware = (req,res,next) =>{
console.log("Signup Middleware")
next();
}

const closeMiddleware = (req,res,next) =>{
console.log("Close Middleware")
next();
}

app.use(loggingMiddleware,signMiddleware,closeMiddleware)
```

Output
GET - /
Signup Middleware
Close Middleware

Practical Aspect of MiddleWare

Instead of defining the process step by step, we can create a middleware

```
const resolveIndexByUserId = (req, res, next) =>{

const {params:{id}, } = req;

const Id = parseInt(id)
if(isNaN(id)){
return res.status(400).send("Id is not a number")
}

const findUserIndex = mockUsers.findIndex( (data)=>data.id===Id)

if(findUserIndex === -1){
return res.status(404).send({msg:"User not found"})
}

//We can dynamically assign values to the req object, since we cannot pass data from one middleware to the other
```

```
req.findUserIndex = findUserIndex;
req.Id = Id;
next();

}
```

We add the variable to the req object so that it can be destructured and accessed in the other function

```
app.put("/api/users/:id",resolveIndexByUserId,(req,res)=>{

//Destructure the findUserIndex and Id from the request
const {body,findUserIndex,Id} = req;

const old = mockUsers[findUserIndex]
mockUsers[findUserIndex] = {id:Id, ...body}

const wow = [{
"Old":old
},{"New":mockUsers[findUserIndex]}]


return res.send(wow)


})
```

This is how the patch request will look

```
app.patch("/api/users/:id",resolveIndexByUserId ,(req,res)=>{

const {body,findUserIndex} = req;

mockUsers[findUserIndex] = {...mockUsers[findUserIndex], ...body};

return res.send( mockUsers[findUserIndex] )


})
```

Validation

Query Validation

```
import { query, validationResult } from "express-validator";

app.get("/api/users",

query("filter").
isString().
withMessage("Should be a string").
notEmpty().
withMessage("Filter cannot be empty").
isLength({min:3, max:10}).
withMessage("Must be between 3 to 10 characters ")

,

(req,res)=>{
//From express validator, import query to validate the query section of the code
//From the query, we are validating the filter field

const result = validationResult(req)
console.log(result)

//Deconstructing query from request object
const {query:{filter, value}} = req;
if(!filter && !value){
return res.send({mockUsers})
}
if(filter && value){
return res.send(
mockUsers.filter((data)=>data[filter].includes(value))
)
}

return res.send({mockUsers})
})
```

Body Request Validation

```
import { query, validationResult, body,matchedData } from "express-validator";


app.post("/api/users",

[
```

```
body("fname")
.notEmpty()
.withMessage("Username cannot be empty")
.isLength({min:5, max:32})
.withMessage("Must be betwen 5 to 32 characters")
.isString()
.withMessage("Must be a string")
]

,

(req,res)=>{

//This will hold all the errors after validation
const result = validationResult(req);

//If the errorResult is not empty throw the errors
//If empty means there are no errors from the validation
if(!result.isEmpty()){
return res.status(400).send({ errors: result.array() })
}


//const data will store only validated data, I will store only validated data
const data = matchedData(req)

const newUser = {
id: mockUsers[mockUsers.length-1].id+1,
...data
}

mockUsers.push(newUser)

res.status(201).send(mockUsers)
})
```

We can make the code more cleaner by creating Schemas

Define a schema and store this code
validationSchema.mjs

```
export const createUserValidation = {
fname: {
isLength:{
options:{min:5, max:32},
errorMessage:"Must be betwen 5 to 32 characters"
},
```

```
notEmpty:{
errorMessage: "Username cannot be empty"
},
isString:{
errorMessage:"Must be a string"
},


},

displayName: {
isLength:{
options:{min:5, max:32},
errorMessage:"Must be betwen 5 to 32 characters"
},

notEmpty:{
errorMessage: "Username cannot be empty"
},
isString:{
errorMessage:"Must be a string"
},


}









}
```

In the index.mjs

```
import { query, validationResult, body, matchedData,checkSchema } from "express-validator";
import { createUserValidation } from "./utils/validationSchemas.mjs";

app.post("/api/users",

checkSchema(createUserValidation)

,

(req,res)=>{
```

```
//This will hold all the errors after validation
const result = validationResult(req);

//If the errorResult is not empty throw the errors
//If empty means there are no errors from the validation
if(!result.isEmpty()){
return res.status(400).send({ errors: result.array() })
}


//const data will store only validated data, I will store only validated data
const data = matchedData(req)

const newUser = {
id: mockUsers[mockUsers.length-1].id+1,
...data
}

mockUsers.push(newUser)

res.status(201).send(mockUsers)
})
```

Express Router

Spring Boot

Hello and welcome to the ultimate spring boot tutorial, This tutorial is straight to the point, more code, less talking. If you are interested in the grammar portion of the framework, always check it up

Now let focus on the money aspect of the framework

Introduction

@Configuration declare a class as full configuration class and the @Bean is declared as a method in the class

example

```
@Configiuration
public class AppConfig{


@Bean
public PaymentService paymentService() {
return new PaymentServiceImpl(accountRepository());
}


@Bean
public AccountRepository accountRepository() {
return new JdbcAccountRepository (dataSource() );
}


@Bean("ds")
public DataSource dataSource() {
return (.....)
}


}
```

Bean Naming

If you don't specify a name for the bean, spring uses the method name as the bean name
So the bean naming will be paymentService, accountRepository and ds.

Dependecy Injenction

- Constructor injection
  @Service
  public class DefaultPaymentSystem{

  private final AccountRepository accountRepository;

  public DefaultPaymentSystem(AccountRepository accountRepositoryContext){

  this. accountRepository = accountRepositoryContext;

  }

  }

We can create 2 Bean and qualify i, so that when we are injecting, we can know which to inject

```
@Configuration
public class HDSS {

@Bean
@Qualifier("primary")
public Account primary(){
return new AccoutList();
}

@Bean
@Qualifier("secondary")
public Account secondary(){
return new AccoutList();
}


}
```