Pre-Requisite for this Course

Link to download .NET 7

<https://dotnet.microsoft.com/en-us/download>

Node JS

It’s a javascript runtime to run our reactjs application inside our development server.

NVM

It’s a node version manager

If u use NVM, you can switch between nodejs version very easily.

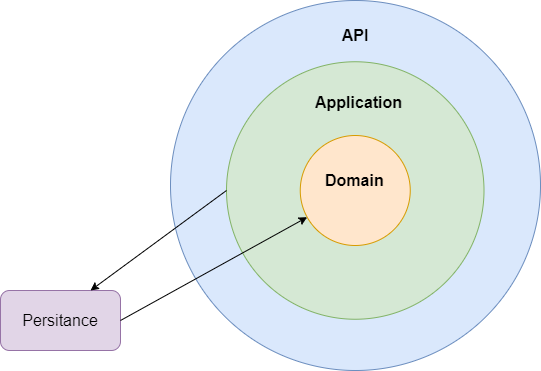
Visual Studio Code – Integrated Development Environment

Post Man – For testing our API

Resources has been forked from [TryCatchLearn/Reactivities](https://github.com/TryCatchLearn/Reactivities) to [MadOnProgramming](https://github.com/MadOnProgramming)/[.NetCoreWithReact](https://github.com/MadOnProgramming/.NetCoreWithReact)

**Clean architecture**

Here we are going to implement clean architecture. I am listing the layer here from the inner most ring to outer most ring. Outer depends on the immediate inner layer but the inner layer doesn’t depends on the outer layer.



* Domain(Entities for our Business model) – *inner most*
* Application layer (contains business login of our application)
* API (Responsible for handling http request and response) – *outer most*

**Creating a .net core project and walking through the skeleton**

Dotnet CLI commands

Dotnet –info

* To get the information about the installed sdk and runtime in our system

Dotnet new list

* To see the list of template that we can create

Dotnet new sln -n YourSolutionName

* Creates solution

Dotnet new webapi -n YourProjectName

* Creates webapi project

Dotnet new classlib -n YourProjectName

* Creates class library project

Dotnet sln add youcsproject\_file

* Adds project to the solution

Dotnet add reference csproject\_file

* Adds one project as a reference to another

**VS code command**

After opening the solution folder in vscode, open command pallete Ctrl+Shift+P

And select ‘Generate assets for build and debug’

This will generate .vscode folder to ur folder

**API Project template**

Properties\launchSettings.json

Keep only the necessary settings and change others.

Program.cs

- It’s the entry point in this application

- Creates a builder from WebApplication

- Region for adding services to the container(builder), these services can be injected while we are using it.

- Build app from the builder

- Region for adding middleware to the application pipeline

- Run the application

Some properties in .csproj

<Nullable> - In .net 6, by default string is required, You can explicitly enable or disable through this property

<ImplicitUsings>

**Starting our implementation**

Start by creating entity classes under domain project.

In our case, our project going to revolve aroung entity called Activity. So, I am creating a class called activity

*public* *class* Activity

    {

*public* Guid Id { *get*; *set*; }

*public* *string* Title { *get*; *set*; }

*public* *string* Description { *get*; *set*; }

*public* *string* Category { *get*; *set*; }

*public* DateTime Date{*get*;*set*;}

*public* *string* Venue { *get*; *set*; }

*public* *string* City { *get*; *set*; }

    }

Note: always create a property with a shortcut of typing ‘*prop’* and type enter

Always name the id property with ‘id’ instead of giving name like ‘ActivityId’. Beacause, entity framework couldn’t recognize this as key property. Unless, u explicitly specify as key property.

**Adding Entity framework to our project**

Here we are not going to use sql server. Instead, we are going to use entity framework with sql lite db.

Add nuget package “Microsoft.EntityFrameworkcore.Sqllite” to our persistent project.

Then add our dbcontext class like below

*using* Domain;

*using* Microsoft.EntityFrameworkCore;

*namespace* Persistence

{

*public* *class* DataContext : DbContext

    {

*public* DataContext(DbContextOptions *options*) : *base*(*options*)

        {

        }

*public* DbSet<Activity> Activities { *get*; *set*; }

    }

}

Then add services to our webapi project program.cs , inorder to use our datacontext class

Through dependency injection.

builder.Services.AddDbContext<DataContext>(*opt* => {

*opt*.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection"));

});

So, we have successfully integrated entityframework core within our application with sqllite.

**Entity framework migrations**

Next we are going install dotnet-ef tool

Get list of dotnet tools installed by the following command

**Dotnet tool list -g**

Here -g – represents global

If dotnet-ef is not available,then install through the following command

**Dotnet tool install –global dotnet-ef –version 7.0.0**

dotnet-ef command to add migration

dotnet-ef migrations add <<MigrationName>> -s <<.\Folder\StartupProject.csproj>> -p <<.\Folder\dbcontextHoldingProject.csproj>>

through this we have added our migration to our code first database

now, we want to create and apply migrations to our actual db(here I am going to use sqllite in my development environment).

Instead of handling this through dotnet-ef command, I am going to do it using following code

#region doing database migrations

//creating scope for our services

*using* *var* scope = app.Services.CreateScope();

//get the service collection from the scope

*var* services = scope.ServiceProvider;

*try*

{

//get the required datacontext service from the service collection

*var* dataContext = services.GetRequiredService<DataContext>();

//do the actual migration through this method

    dataContext.Database.Migrate();

}

*catch*(Exception ex)

{

//get the logger from the required service

*var* logger = services.GetRequiredService<ILogger<Program>>();

//log incase of any error

    logger.LogError(ex,"An error occured during database migration");

}

#endregion

app.Run();

Add the above codes just before the app.Run();

I have explained the code through the comments above each line.

Finally run the code through ***dotnet watch*** command and see the logs in the terminal for what happened as everything is logged because of the setting ***Microsoft.AspNetCore:Information*** in the ***appSettings.Development.json.***

If u want to see the content of the db after migration, u can see through the Ctrl+Shift+p

**Sqllite:Open the database**

Then in the sqllite explorer, of the left side panel, u can see the contents.

**Seeding the database(Sample data)**

Add a class called seed.cs in the persistence project and add a async static method called ‘SeedData’ to load the database with the sample data.

*public* *async* *static* Task SeedData(DataContext *context*)

        {

*if*(*context*.Activities.Any())

*return*;

*var* activities = *new* List<Activity>

            {

*new* Activity

                {

                    Title = "Past Activity 1",

                    Date = DateTime.UtcNow.AddMonths(-2),

                    Description = "Activity 2 months ago",

                    Category = "drinks",

                    City = "London",

                    Venue = "Pub",

                },

*new* Activity

                {

                    Title = "Past Activity 2",

                    Date = DateTime.UtcNow.AddMonths(-1),

                    Description = "Activity 1 month ago",

                    Category = "culture",

                    City = "Paris",

                    Venue = "Louvre",

                },

            };

*await* *context*.Activities.AddRangeAsync(activities);

*await* *context*.SaveChangesAsync();

        }

Call this method in program.cs after calling the migrate method like this

*try*

{

*var* context = services.GetRequiredService<DataContext>();

*await* context.Database.MigrateAsync();

*await* Seed.SeedData(context);

}

This will load the sample data to the tables.

After the change use the command called ***dotnet watch –no-hot-reload*** which disables hot reload and thus inturn removes lot of headaches.

After this open the db and check the table which will contain our sample data.

**Adding API Controller**

We are going to add two api controller cs,

one is BaseApiController which is inherited from ControllerBase and going to be base class all our other controller api(s).

second one is ActivityController which inturn contains two httpget endpoint called *api/activity/all – to get all activities*  & api/activity/{id} – to get single activity

These are the coding for those cs files

*using* Microsoft.AspNetCore.Mvc;

*namespace* API.Controllers

{

    [ApiController]

    [Route("api/[controller]")]

*public* *class* BaseApiController : ControllerBase

    {

    }

}

*using* Domain;

*using* Microsoft.AspNetCore.Mvc;

*using* Microsoft.EntityFrameworkCore;

*using* Persistence;

*namespace* API.Controllers

{

*public* *class* ActivityController : BaseApiController

    {

*public* DataContext \_context { *get*; }

*public* ActivityController(DataContext *context*)

        {

            \_context = *context*;

        }

        [HttpGet]

        [Route("all")]

*public* *async* Task<ActionResult<List<Activity>>> GetActivities()

        {

*return* *await* \_context.Activities.ToListAsync();

        }

        [HttpGet]

        [Route("{id}")]

*public* *async* Task<ActionResult<Activity>> GetActivity(Guid *id*)

        {

*return* *await* \_context.Activities.FindAsync(*id*);

        }

    }

}

Then I am going to add gitignore file through dotnet cli command

***Dotnet new gitignore***

After that commit the files to the git repository and pushed to git.

**Creating React application in our project**

We are going to use vite to create our react application and vite going to act as a build tool and development server in our case.

Command to create our react project template

**Npm create vite@latest**

Which will ask for project name and other information

Once project is created,

**Cd app\_directory**

**Npm install** --> to install dependencies and packages

**Npm run dev** --> here dev is our vite

**After setting up our react project**

* I am going to create some edits in app.tsx to call the api we have created to list all the activities using semantic-ui react component and styling framework.
* Read the documentation from semantic-ui react app and install it through

npm install semantic-ui-react@3.0.0-beta.0 semantic-ui-css

* and import css in main.tsx above importing index.css

import 'semantic-ui-css/semantic.min.css'

* after that, make the following changes and run to see changes in styling

import { useEffect, useState } from 'react'

import './App.css'

import axios from 'axios';

import { Header, List } from 'semantic-ui-react';

*function* App() {

  // state - usestate hook

  // activities - state variable that holds data

  // SetActivities - method to set the state variable

  // [] - initial value of the stat

*const* [activities, SetActivities] = useState([]);

  //used to implement effect after loading component

  //useEffect hook will be called after loading our component

  useEffect(() *=>* {

    axios.get("http://localhost:5000/api/activity/all")

      .then((*response*) *=>* {

        console.log(*response*);

        SetActivities(*response*.data);

      });

  }, []);

  return (

    <div>

      {/\* Header, List are semantic-ui react components \*/}

      <*Header* as='h2' icon='users' content='My Activities' />

      <*List*>

        {activities.map((*activity*: *any*) *=>* (

          <*List.Item* key={*activity*.id}>

            {*activity*.title}

          </*List.Item*>

        ))}

      </*List*>

    </div>

  )

}

export default App

**Next Section : CRUD application using CQRS & Mediator patter**

**Clean Architecture**A diagram of a clean architecture

Description automatically generated

This is a high level architectural pattern for software design satisfies soc(separation of concern) so that applications are easily scalable, testable & maintainable etc.,

**Mediator Pattern**

A diagram of a mediator

Description automatically generated

This is a low level behavioural design pattern that makes our application loosely coupled with each other so that layers can be easily testable, upgradable etc. The communication between layer is handled by mediator.

**CQRS – Command Query Responsibility Segregation**

This is a architecture pattern is to separate the responsibility for handling commands(create,update & delete) and queries(read).

By doing so, we can optimize and scale these responsibilities independently. For example, you might have different storage mechanisms, databases or even servers for handling commands and queries.

Command – changes state of an application (creating,updating or deleting data).

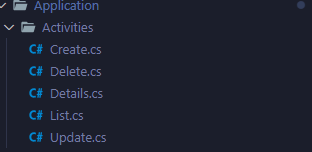
Query – retrieve data but don’t modify it.

A diagram of a database

Description automatically generated

**Applying MediatR with CQRS in our Application layer**

* Install MediatR package in our application project. This is going to be mediator for communication between our api project and application project.
* Inorder to apply CQRS orchestration in our application project, we have to create separate class for each use case (create,delete,edit,list,details) under a folder with name corresponding to the domain entity name its going to handle.



After making necessary changes in our application and api controllers for using mediatr and CQRS.

Our controller look like this

*./API/BaseApiController.cs*

|  |
| --- |
| [ApiController]      [Route("api/[controller]")]  *public* *class* BaseApiController : ControllerBase      {  *private* IMediator \_mediator;  *protected* IMediator Mediator => \_mediator ??= HttpContext.RequestServices.GetService<IMediator>();      } |

*./API/ActivityController.cs*

|  |
| --- |
| *using* Application.Activities;  *using* Domain;  *using* MediatR;  *using* Microsoft.AspNetCore.Mvc;  *namespace* API.Controllers  {  *public* *class* ActivityController : BaseApiController      {  */// <summary>*  */// To list all activities*  */// </summary>*  */// <returns></returns>*          [HttpGet]  *public* *async* Task<ActionResult<List<Activity>>> GetActivities()          {  *return* *await* Mediator.Send(*new* List.Query());          }  */// <summary>*  */// To get single activity*  */// </summary>*  */// <param name="id"></param>*  */// <returns></returns>*          [HttpGet]          [Route("{id}")]  *public* *async* Task<ActionResult<Activity>> GetActivity(Guid *id*)          {  *return* *await* Mediator.Send(*new* Details.Query { Id = *id* });          }  */// <summary>*  */// Creates an activity*  */// </summary>*  */// <param name="activity"></param>*  */// <returns></returns>*          [HttpPost]  *public* *async* Task<IActionResult> Create(Activity *activity*)          {  *await* Mediator.Send(*new* Create.Command { Activity = *activity* });  *return* Ok();          }  */// <summary>*  */// Update an activity*  */// </summary>*  */// <param name="id"></param>*  */// <param name="activity"></param>*  */// <returns></returns>*          [HttpPut]          [Route("Update")]  *public* *async* Task<IActionResult> Update(Guid *id*, Activity *activity*)          {  *await* Mediator.Send(  *new* Update.Command                  {                      Id = *id*,                      Activity = *activity*                  });  *return* Ok();          }  */// <summary>*  */// Delete an activity*  */// </summary>*  */// <param name="id"></param>*  */// <returns></returns>*          [HttpDelete]          [Route("{id}")]  *public* *async* Task<IActionResult> Delete(Guid *id*)          {  *await* Mediator.Send(*new* Delete.Command { Id = *id* });  *return* Ok();          }      }  } |

Here i am going to specify the sample for one Query & one Command usecase

*./Application/Activities/List.cs*

|  |
| --- |
| *using* System;  *using* System.Collections.Generic;  *using* System.Linq;  *using* System.Threading.Tasks;  *using* Domain;  *using* MediatR;  *using* Microsoft.EntityFrameworkCore;  *using* Persistence;  *namespace* Application.Activities  {  *//Usecase - get the list of activities*  *public* *class* List      {  *//query operation class*  *//returns list of activity(this is our domain object)*  *public* *class* Query : IRequest<List<Activity>>          { } *// empty as query doesn't have any parameter*  *//handler class*  *public* *class* Handler : IRequestHandler<Query, List<Activity>>          {  *public* DataContext \_context { *get*; }  *public* Handler(DataContext *context*)              {                  \_context = *context*;              }  *//handle method*  *public* *async* Task<List<Activity>> Handle(Query *request*, CancellationToken *cancellationToken*)              {  *return* *await* \_context.Activities.ToListAsync<Activity>();              }          }      }  } |

*./Application/Activities/Delete.cs*

|  |
| --- |
| *using* System;  *using* System.Collections.Generic;  *using* System.Linq;  *using* System.Threading.Tasks;  *using* Domain;  *using* MediatR;  *using* Persistence;  *namespace* Application.Activities  {  *public* *class* Create      {  *public* *class* Command : IRequest          {  *public* Activity Activity;          }  *public* *class* Handler : IRequestHandler<Command>          {  *private* *readonly* DataContext \_dataContext;  *public* Handler(DataContext *dataContext*)              {                  \_dataContext = *dataContext*;              }  *public* *async* Task Handle(Command *request*, CancellationToken *cancellationToken*)              {                  \_dataContext.Activities.Add(*request*.Activity);  *await* \_dataContext.SaveChangesAsync();              }          }      }  } |

**Automapper**

* Automapper maps properties of one object to other.
* While updating an entity, we need to assign properties from the request to the matched entity that we take from the domain.
* If we manually write the code assign every property, no of lines of code will be more and more cluttered. So we are going to use automapper for this functionality and going to inject it.
* Package Name : ***Automapper.Extensions.Microsoft.DependencyInjection***
* Create a folder name ‘*Core’* under the application project and write your mapping profiles there

|  |
| --- |
| * *using* System; * *using* System.Collections.Generic; * *using* System.Linq; * *using* System.Threading.Tasks; * *using* AutoMapper; * *using* Domain; * *namespace* Application.Core * { * */// <summary>* * *///Mapping profiles for automapper functionality* * *///which will be registered to IOC container in program.cs* * *///and injected to usecase cs files under application project* * */// </summary>* * *public* *class* MappingProfiles : Profile * { * *public* MappingProfiles() * { * CreateMap<Activity,Activity>(); * } * } * } |

Then in the program.cs, we have to register the automapper service

builder.Services.AddAutoMapper(*typeof*(MappingProfiles));

or

builder.Services.AddAutoMapper(*typeof*(MappingProfiles).Assembly);

then in the usecase ./Activities/edit or update.cs

|  |
| --- |
| *public* *async* Task Handle(Command *request*, CancellationToken *cancellationToken*)              {                  Activity matchedActivity = \_context.Activities.Find(*request*.Activity.Id);  *// matchedActivity.Title = request.Activity.Title;*  *// matchedActivity.Description = request.Activity.Description;*  *// matchedActivity.Category = request.Activity.Category;*  *// matchedActivity.Date = request.Activity.Date;*  *// matchedActivity.Venue = request.Activity.Venue;*  *// matchedActivity.City = request.Activity.City;*                  \_mapper.Map(*request*.Activity,matchedActivity);  *await* \_context.SaveChangesAsync();              } |

Here \_***mapper.Map(request.Activity,matchedActivity);***  maps activity from the request to the matched activity

**Cleaning up the program.cs with the extension class**

* Here we are going to move the services we have added to an extension class for better maintainability and to avoid code cluttering
* Create a folder structure ***./API/ApplicationServiceExtensions.cs***
* and move the service code added by us and then call this extension method from program.cs

|  |
| --- |
| *using* Application.Activities;  *using* Application.Core;  *using* Microsoft.EntityFrameworkCore;  *using* Persistence;  *namespace* API.Extensions  {  *public* *static* *class* ApplicationServiceExtensions      {  *public* *static* IServiceCollection AddApplicationServices(*this* IServiceCollection *services*,IConfiguration *config*)          {  *//register service for db context*  *services*.AddDbContext<DataContext>(*opt* =>              {  *//database configuration*  *opt*.UseSqlite(*config*.GetConnectionString("DefaultConnection"));              });  *//register service for applying CORS*  *services*.AddCors(*opt* =>              {  *opt*.AddPolicy("MyCorsPolicy", *bui* =>                  {  *bui*.AllowAnyHeader()                          .AllowAnyMethod()                          .WithOrigins("http://localhost:5001");                  });              });  *//register MediatR service*  *services*.AddMediatR(*conf* =>              {  *//we need to say where it can find handlers from*  *conf*.RegisterServicesFromAssembly(*typeof*(List.Handler).Assembly);              });  *//register automapper and their mapping profile*  *services*.AddAutoMapper(*typeof*(MappingProfiles).Assembly);  *return* *services*;          }      }  } |

**Cancellation Token**

While we send a request to server to process it, Service will process request until it is completed. Even though if cancel the request, server will be still processing the request until it is completed.

Here comes the cancellation token.

Whenever we cancel the request, cancellation token will be passed. We have handle in our code to stop processing our request, when cancel request has been made in order to avoid free up the resource and unwanted loading of the server.

For example:-

***In controller’s action method(endpoint)***

|  |
| --- |
| [HttpGet]  *public* *async* Task<ActionResult<List<Activity>>> GetActivities(CancellationToken *ct*)  {  *return* *await* Mediator.Send(*new* List.Query(),*ct*);  } |
|  |

***In UseCase’s Handler method***

|  |
| --- |
| *public* *async* Task<List<Activity>> Handle(Query *request*, CancellationToken *cancellationToken*)              {  *try*                  {  *for*(*int* i =1;i<20;i++)                      {  *//throw error if cancelled*  *cancellationToken*.ThrowIfCancellationRequested();  *//mock code for long running process*  *await* Task.Delay(2000,*cancellationToken*);  *//to log whether Nth task is completed*                          \_logger.LogInformation($"Task {i} is completed");                      }                  }  *catch*                  {                      \_logger.LogInformation("Task has been cancelled");                  }  *return* *await* \_context.Activities.ToListAsync<Activity>();              } |

**Debugging code in VSCode**

1. Ctrl+Shift+p => .NET:Generate assests for build and debug =>this will create .vscode folder
2. In .vscode/launch.json , two profiles will be there by default
   1. .NET Core Launch(Web) – to debug while ***dotnet run***
   2. .NET Core Attach – to debug while while ***dotnet watch***

**Changing our code in App.tsx into exact type specific**

In our app.tsx, we have used :any in place of calling map function.

Now we are changing it to the strongly typed by changing the following code

For this we have to change in 3 places,

We have to define our type called Activity in Activity.ts

*export* type Activity = {

    id: string

    title: string

    description: string

    category: string

    date: string

    venue: string

    city: string

  }

Specify the type of the model in UseState<T>

const [activities, SetActivities] = useState<Activity[]>([]);

Specify the type axios method

  useEffect(() => {

    axios.get<Activity[]>("http://localhost:5000/api/activity")

      .then((*response*:any) => {

        console.log(*response*);

        SetActivities(*response*.data);

      });

  }, []);

Then, we will automatically get the intellisense in our map function while using object notatation for our type Activity.