INTRODUCTION

CODING TASK: Calendar

This is a small application to see Appointments based on different months. It provides two REST endpoints:

1. REST endpoint (GET) (api/month): This endpoint retrieves the months data from the

database (**Entity framework In-Memory**).

2. REST endpoint (GET) (api/month/{monthid}/appointment): This endpoint receives a month

ID and returns the appointments for that particular month.

The project has been created using the **.Net Core Web API** (**Dependency Injection — Auto Mapper — Repository Pattern – Entity Framework In-Memory**), **Angular 10+ as frontend.**

.Net Core Web API is a framework for building HTTP-based services that can be consumed by different clients, it includes web browsers, mobile applications, etc.

Built With

Below are the listed platforms, languages, and patterns used in the project.

Platform:

- .Net Core
- GitHub
- Angular

Technical Specifications:

- C#
- Entity Framework
- Repository Design Pattern
- Auto Mapper
- Dependency Injection
- Angular

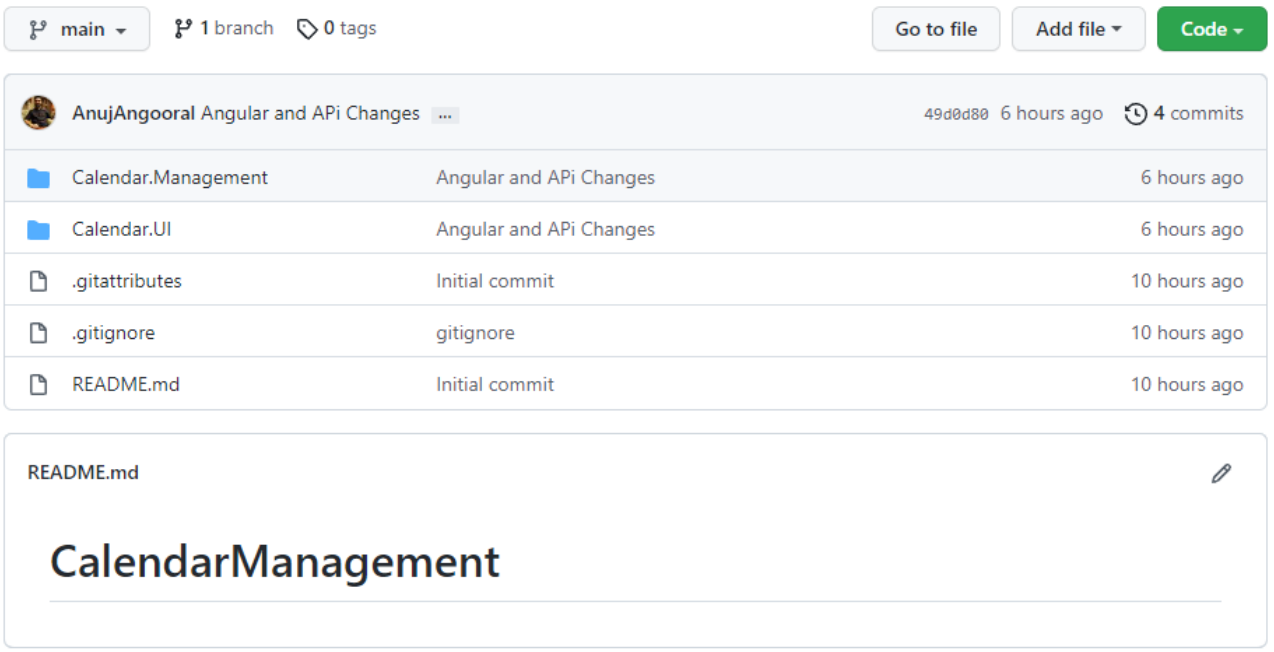
Installation

To get a local copy up and running follow these simple steps.

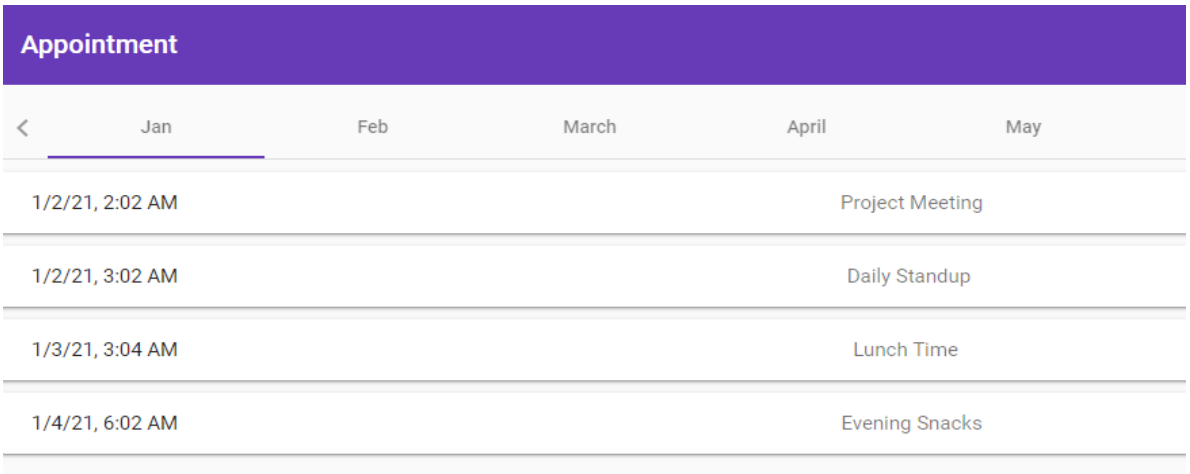
Visual Studio Code: Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Linux, macOS, and Windows. It comes with built-in support for JavaScript, TypeScript, and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and runtimes (such as .NET and Unity). Download and install from here.

Installation Steps → click here

GitHub Code Repository:
https://github.com/AnujAngooral/CalendarManagement



Frontend URL: We have used **Angular** as a frontend. Angular is a platform for building mobile and desktop web applications. After installation and code setup we can check the application on the local machine. Our local URL will be something similar to http://localhost:4200/.
NOTE: Default angular picks the 4200 port, but you can change it through command.



WEB API ARCHITECTURE

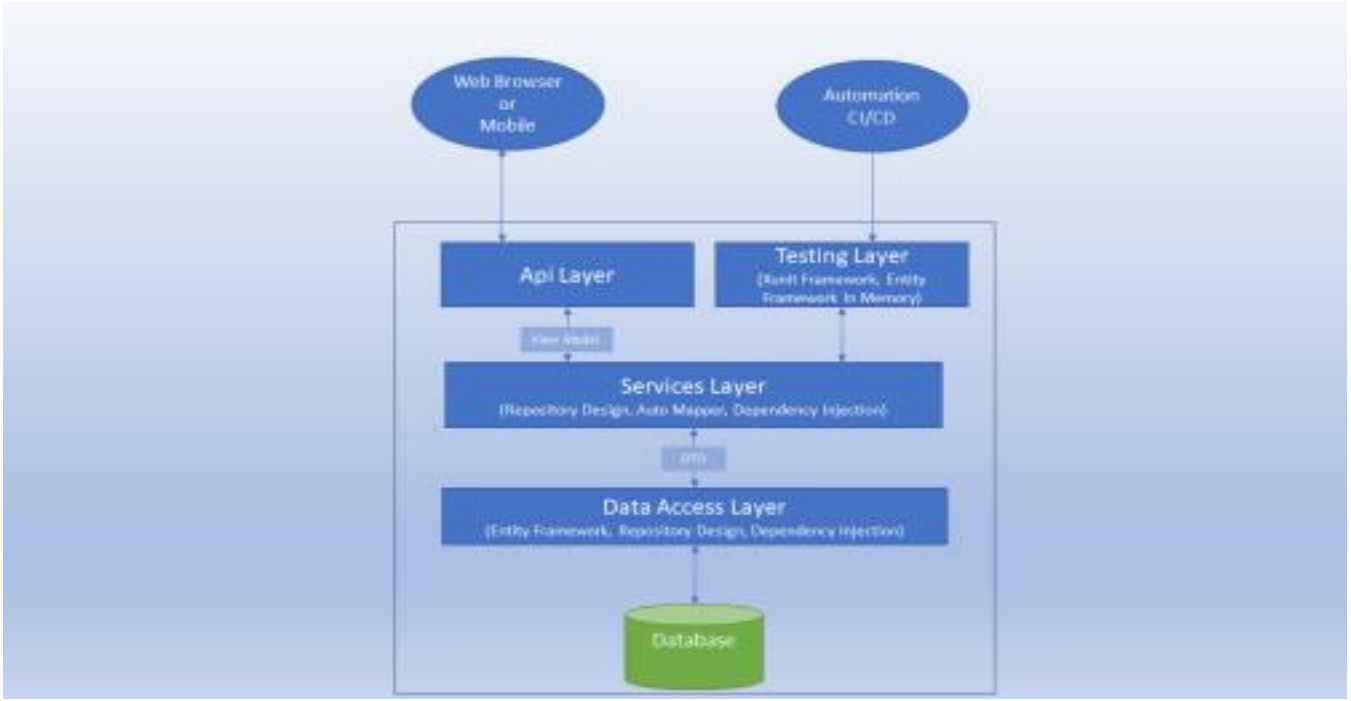
Client (Web browser/Mobile/Other Micro-service) will request a appointment based on month that will interact with an API layer that acts as a frontend for clients. API layer will further interact with the Service layer which is explained below. Web API is divided into different layers. Below is the diagram which shows the project structure.

- **API Layer**: It will use to handle the request and send back the appropriate responses. This layer doesn't know about the service layer functionality, its main function is to pass the request to the service layer and handle any Global Exception. Authentication and authorization will fall under this layer.

- **Service Layer:** The main role of this layer is to have the business logic as well as to convert the ViewModel object to DTO (Data Transfer Object) and vice versa. You can have a private validation method to validate the ViewModel object and take necessary actions on it.

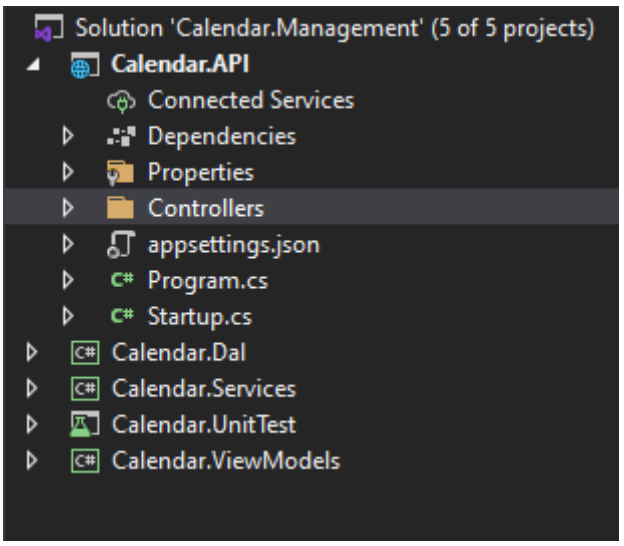
- **DAL:** It is known as a Data Access Layer; it is used to communicate with the Databases. It could be SQL Server, MySQL, NoSQL, Redis Cache, etc. We are using Entity Framework **In-Memory** to fetch and post data.

- **Integration Test:** I have used the NUnit test framework for Unit testing. I have used In-Memory to test the Business Logic of the application. In the test project, we have used the DI to initialise the service and repository.



INTEGRATION

In this section, we will discuss different layers implemented in the project mentioned in our architecture section. Firstly, to view the project you can open the 'Calendar.Management' solution in Visual Studio 2019, it will look like the below image:



The above image includes API, DAL, Services, ViewModel, and IntegrationTest. All of these are briefly explained below.

API: Open the AppointmentController, it's available in the Controller folder under the API. We have

injected two services named ILogger and IAppointmentService. Also, it includes one functions “GetByMonthId”. In this function, we are getting the monthId from the request, and we are passing it to the IAppointmentService which is acting as a service. To enhance the project we can add Authentication and Authorization in the controller. But for this demo, I am mainly focusing on the structure. Below is the code sample for the Get function.

```
/// <summary>
/// This method is use to return the collection of appointment based on the methodid.
/// </summary>
/// <param name="monthId">The id of the month for which you want to fetch the appointme
/// <returns>Returns the collection of appointment. In case of any error, 500 internal
[HttpGet]
[Route("month/{monthId}/appointments")]
0 references | Anuj Angooral, 9 hours ago | 1 author, 1 change
public async Task<IActionResult> GetByMonthId(int monthId)
{
    _logger.LogInformation("About to call appointment service to get appointments based
    var result = await _appointmentService.GetByMonthIdAsync(monthId);

    if (result.IsSuccess)
        return Ok(result.Appointments);

    return StatusCode(500, "Internal Server Error");

}
```

Services: Let's see how this is implemented in the Services Layer. In this layer, I have added the Automapper NuGet package. This is to convert ViewModel to DTO and vice versa. In this function, we are converting the Viewmodel to DTO and passing it to Data Access Layer to Add and Save the object in the database (In-Memory).

Note: In the response, we are sending back three properties.

API Response: Below points will explain the responses returned from the API.

1. **IsSuccess:** Which tells the controller/API layer everything is good and data has been saved.
2. **Appointment:** This is an appointment object, that we are sending back to the API layer, so further it can send back to the client.
3. **ErrorMessage:** If IsSuccess is false, then we will populate this property with the error.

Below is the code sample for the Get method.

```
/// <summary>
/// This method takes the month id and returns the collection of appointments.
/// </summary>
/// <param name="monthId">Month id for which you want to get appointments</param>
/// <returns>Returns collection of appointment and issuccess true. In case of
/// any error, errormessage is returned with issuccess false</returns>
3 references | 0/1 passing | Anuj Angooral, 9 hours ago | 1 author, 1 change
public async Task<(bool IsSuccess, IEnumerable<AppointmentViewModel> Appointments, string ErrorMessage)>
{
    try
    {
        // Fetch appointment from the database based on monthid
        // convert dto to viewmodel
        var appointments = _mapper.Map<IEnumerable<AppointmentViewModel>>
            (await _appointmentRepository.GetAppointmentsByMonth(monthId));
        return (true, appointments, null);
    }
    catch (Exception ex)
    {
        _logger.LogError($"Error: {ex.Message} | {ex.StackTrace}");
        return (false, null, ex.Message);
    }
}
```

DAL: In the data access layer, we have used the generic repository. The generic IRepository interface, which will handle most of the tasks. Below is the code sample for the interface.

```
public interface IRepository<T>
{

    public  Task AddAsync(T entity);

    public Task<T> GetAsync(Expression<Func<T, bool>> filter);

    public Task<IEnumerable<T>> GetAsync();


    Task DeleteAsync(Expression<Func<T, bool>> filter);
    1 reference | Anuj Angooral, 9 hours ago | 1 author, 1 change
    public Task CommitAsync();

}
```

This interface is implemented in the generic Repository Class. We have injected the Dbcontext into the repository constructor. This layer has a generic DbSet which is used to pass/create the instance of any Entity(DTO). In the GET method, you can return IQueryable<T> which is a better approach. Below is the code sample for the generic repository.

```
/// Abstract generic class which handles common functions for all the repositories.
/// </summary>
/// <typeparam name="T">Entity</typeparam>
5 references | Anuj Angooral, 9 hours ago | 1 author, 1 change
public abstract class Repository<T> : IRepository<T> where T : class
{
    internal CalendarDbContext dbContext;
    internal DbSet<T> dbSet;

    2 references | Anuj Angooral, 9 hours ago | 1 author, 1 change
    public Repository(CalendarDbContext context)
    {
        this.dbContext = context;
        this.dbSet = context.Set<T>();
    }
    /// <summary>
    /// Add an entity to database
    /// </summary>
    /// <param name="entity">Entity to store in the database</param>
    /// <returns>Newly created entity</returns>
    1 reference | Anuj Angooral, 9 hours ago | 1 author, 1 change
    public async Task AddAsync(T entity)
    {
        await dbContext.AddAsync(entity);
    }

    /// <summary>
    /// Get the entity based on the filter
    /// </summary>
    /// <param name="filter">pass the filter like id==1, etc</param>
    /// <returns>first matched entity</returns>
    1 reference | Anuj Angooral, 9 hours ago | 1 author, 1 change
    public async Task<T> GetAsync(Expression<Func<T, bool>> filter)
    {
        return await dbSet.FirstOrDefaultAsync(filter);
    }
```

Dependency Injection and Automapper

Open the Startup.cs file present in the API project. In this, we have called a class Registration.ConfigureServices. This class is available in the services layer, where the services layer registers services interfaces with the services implementation as well as the auto mapper.

The below line requests the Entity framework to use **In-Memory Database** with the name CalendarManagement.

```
services.AddDbContext<CalendarDbContext>(opt =>
                                        opt.UseInMemoryDatabase("CalendarManagement"));
```

To register a service, we use the below syntax. Interface name and service name.

```
services.AddScoped<IMonthRepository, MonthRepository>();
services.AddScoped<IAppointmentRepository, AppointmentRepository>();
```

Automapper Configuration: To configure the automapper, we are using the below code.

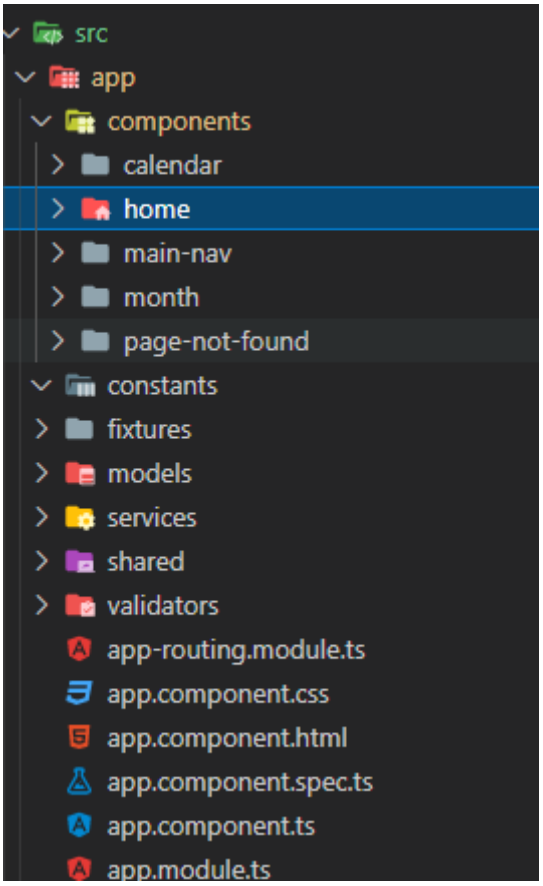
```
var mapperConfig = new MapperConfiguration(mc =>
{
    mc.AddProfile(new MappingProfile());
});

IMapper mapper = mapperConfig.CreateMapper();
services.AddSingleton(mapper);
```

FRONTEND ARCHITECTURE

I have divided the application into folders structure. Below are the different folders explained.

Component: This is the core part of the application, it contains the month, calendar, page not found, home, and navigation component.

1) Page not found component is called when a user enters a URL that is not registered in angular or is an invalid URL.
2) The Main Nav component is used to display the menu bar, right now I am displaying Home, About and Calendar on the left panel.
3) Home Component gives an overview of the project.
4) Calendar Component is used to display the appointments month wise. You can select different month and based on the month, we fetch the appointments from the API and display them in the same component.



Services: We have multiple services to interact with different components. Below are the services explained.

1) HttpInterceptor: The main function of this service is to add authentication tokens, change requests, and responses. I have implemented this handler to handle the errors and log into the console and display using the Matsnackbar service.

2) Notify Service: It is a wrapper over the Matsnackbar service, to change the color of the error and success message. I have implemented this service all over the project to display messages on the UI.

3) Appointment Service: This is the main service to interact with the Rest API. All the requests are send from the component to service and then further from service to Rest API. It has HttpClient DI in the constructor and the base URL of the API is coming from the environment file.

4) Month Service: This service is use to fetch the data related to month.

IMPROVEMENTS

Below are some of the improvements that we can implement to make the product more efficient and effective.

- We can create a unit test case for every function in angular.
- We can implement integration-type test cases in Web API.
- We can add CI/CD pipeline for the project which will trigger the test cases before it deploys the build on the staging environment.