INTRODUCTION

CODING TASK: SHIP

This is a small application to manage ships that will allow the user to perform CRUD (Create, Read, Update & Delete) operations on a ship. Each ship must have a name (string), length (in meters), width (in meters), and code (a string with a format of **AAAA-1111-A1** where A is any character from the Latin alphabet and 1 is a number from 0 to 9). It provides two REST endpoints:

1. REST endpoint (POST) (api/ship): This endpoint receives a ship type (request body), saves it to a database (**Entity framework In-Memory**) with a unique ID, and returns a newly created ship endpoint (response body).

2. REST endpoint (GET) (api/ship/<ID>): This endpoint receives a ship ID and returns the ship data (response body).

3. REST endpoint (GET) (api/ship): This endpoint returns all the ship's data (response body).

4. REST endpoint (PUT) (api/ship/<ID>): This endpoint receives a ship (request body), updates it to a database (In-Memory), and returns the updated ship data (response body).

5. REST endpoint (DELETE) (api/ship/<ID>): This endpoint receives a ship ID, deletes it from a database (In-Memory), and returns the success code.

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, and Docker as a deployment platform.**

.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
- Docker
- 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.

Docker for Desktop: Docker Desktop for Windows is Docker designed to run on Windows 10. It is a native Windows application that provides an easy-to-use development environment for building, shipping, and running dockerized apps. Download and install from here.

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.

Swagger UI: There is another option to test the API endpoints without any installation. The Swagger UI is an open-source project to visually render documentation for an API defined with the OpenAPI (Swagger) Specification. Swagger UI lets you visualize and interact with the API's resources without having any of the implementation logic in place, making it easy for back-end implementation and client-side consumption. We have installed this tool, therefore, there is no need for any further installation. For localhost, the Swagger URL will be something like http://localhost:5005/swagger/index.html. From this URL we can directly access the endpoints to create and request a ship.

shipmanagement.api 1.0 OAS3

/swagger/v1/swagger.json

Error

GET /error

Ship

POST /api/ship

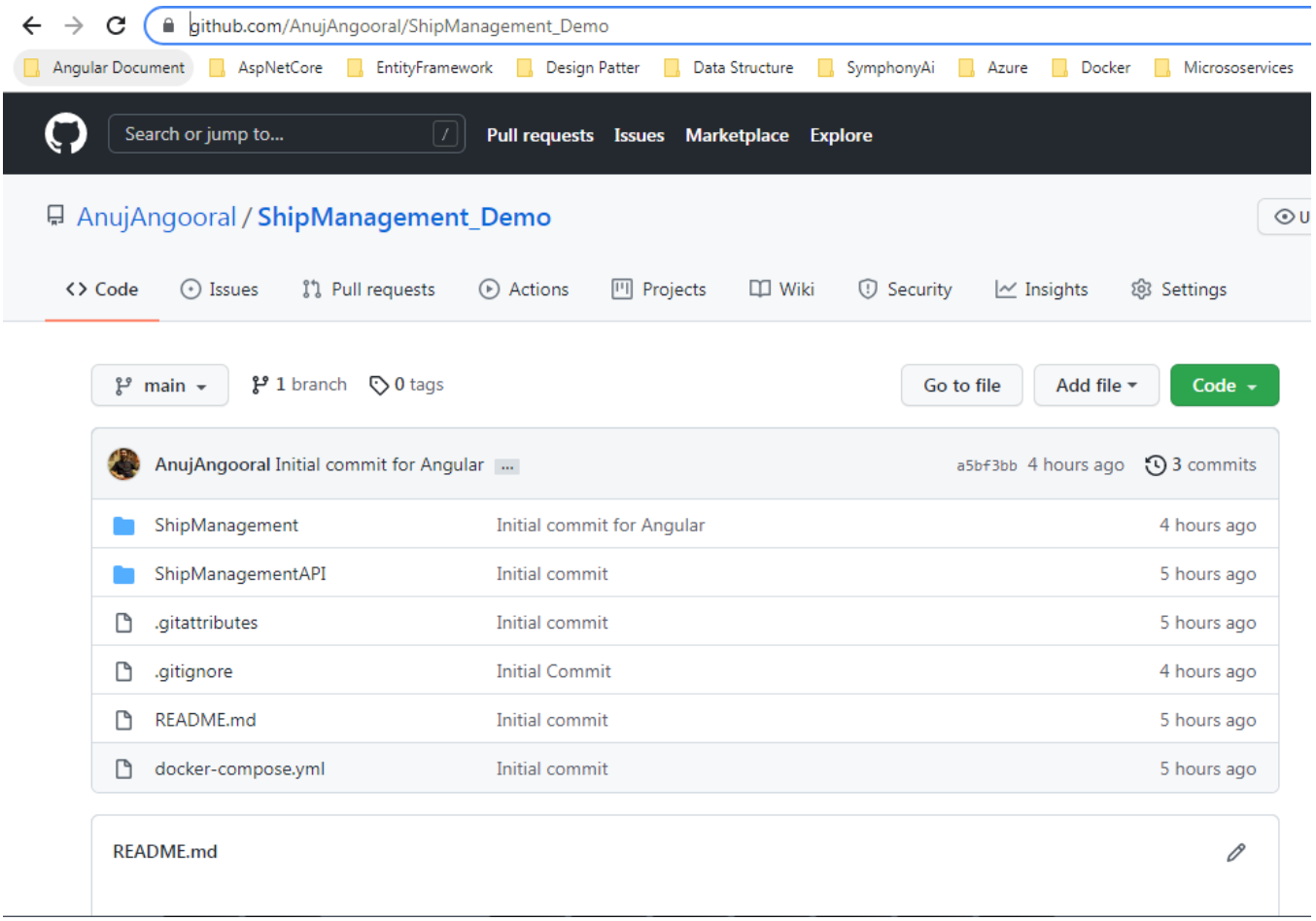
GET /api/ship

PUT /api/ship/{id}

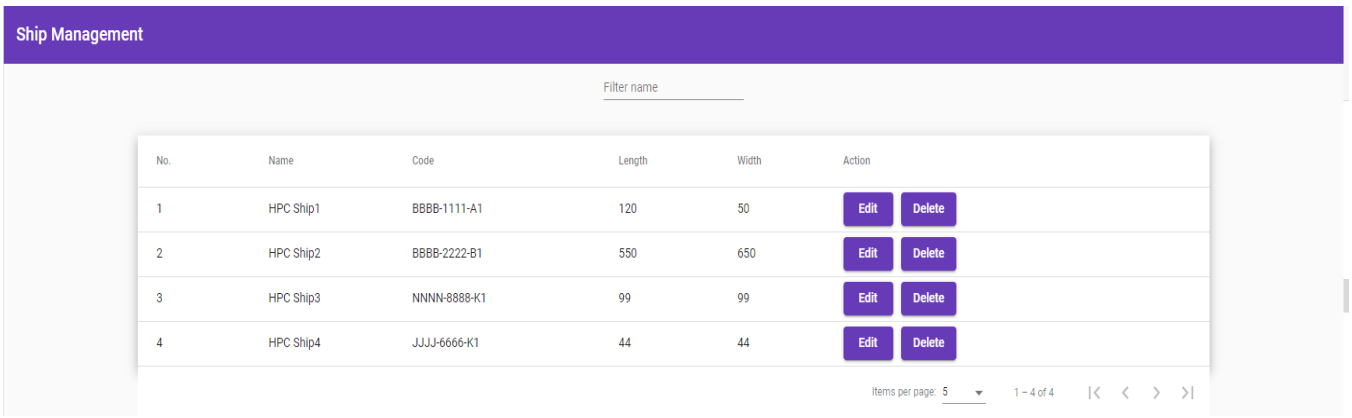
GET /api/ship/{id}

DELETE /api/ship/{id}

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



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:3000/.



WEB API ARCHITECTURE

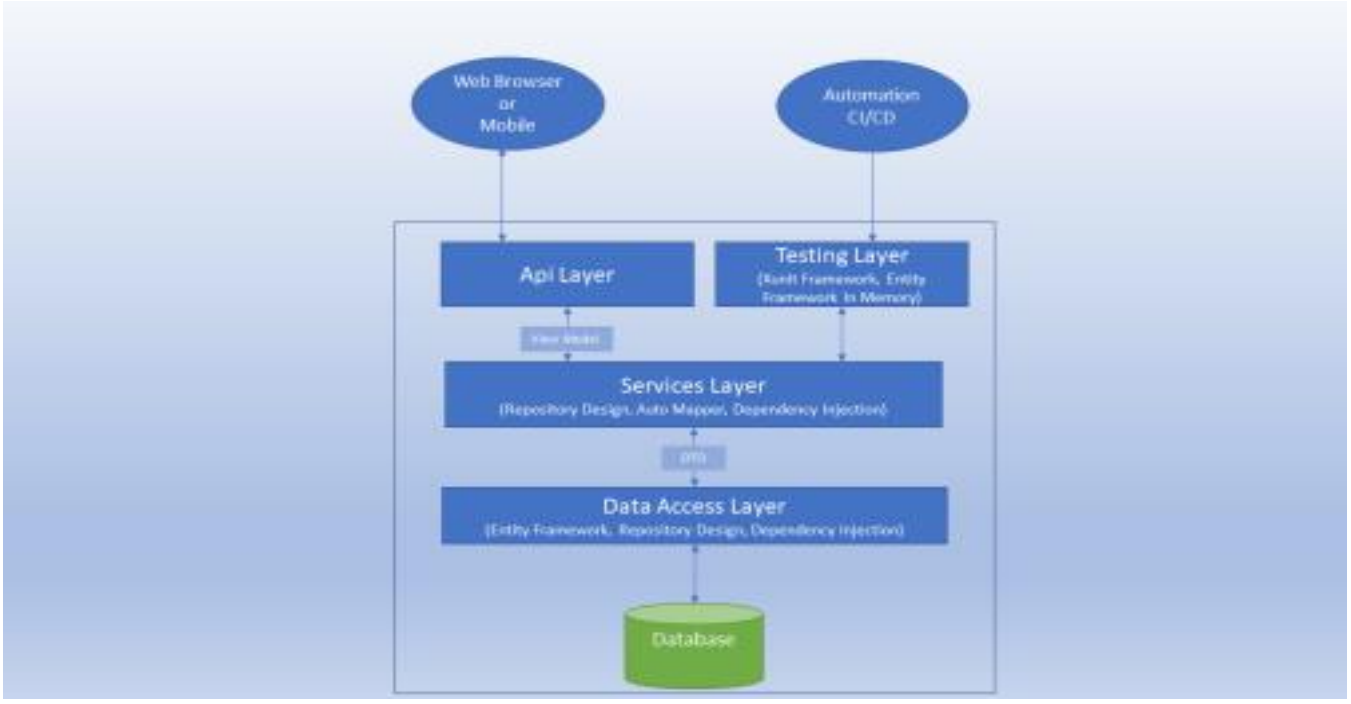
Client (Web browser/Mobile/Other Micro-service) will request/create a ship that will interact with an API layer that acts as a frontend for clients. API layer will further interact with the Service layer which is explained below. Web API is divided into different layers. Below is the diagram which shows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

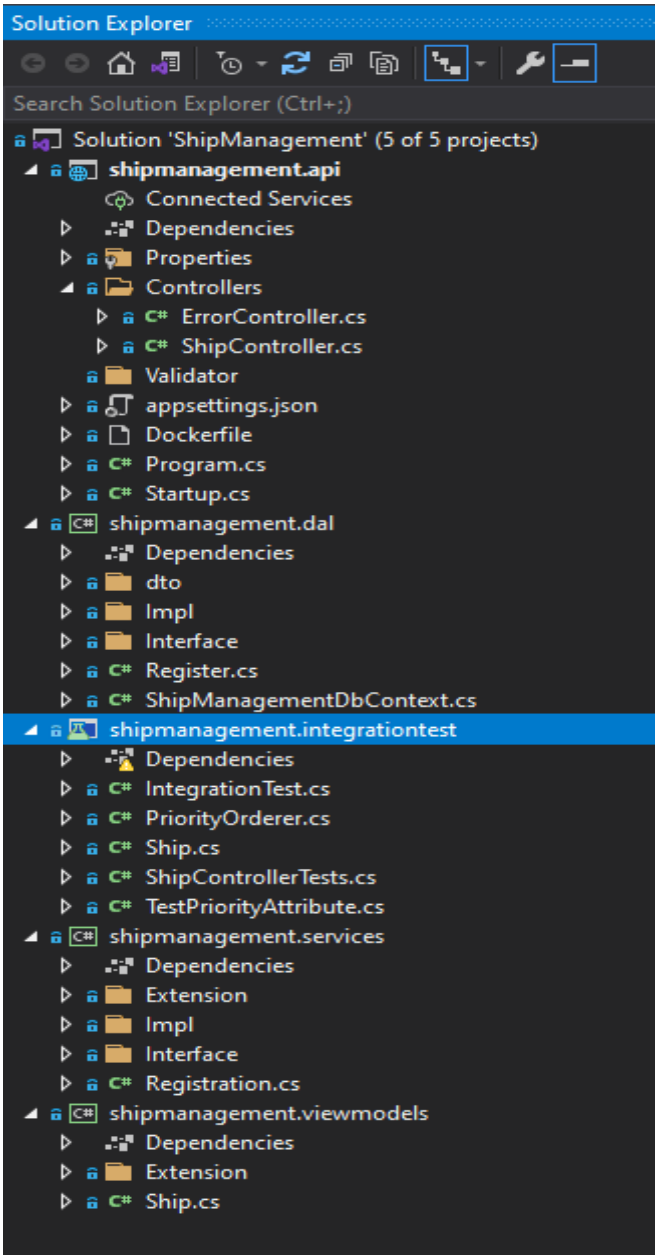
- **Integration Test:** I have used the XUnit test framework for integration testing. I have used In-Memory to test the API calls. In the test project, we have created an HTTP client to send the request to API and get back the response and used the response to Assert the test cases.



Note: The testing layer shown in the diagram is not implemented in the project as this is a basic application but this layer can be implemented in enhancement.

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 'ShipManagement' 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 ShipController, it's available in the Controller folder under the API. We have injected two services named ILogger and IShipService. Also, it includes three functions “Add, Update, GetById, Get, and Delete”. In the Add function, we are getting the ship model from the request, and we are passing it to the IShipService 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 Add function.

```
[HttpPost]
[Route("ship")]
[ProducesResponseType(typeof(Ship), (int)System.Net.HttpStatusCode.Created)]
0 references
public async Task<IActionResult> Add([FromBody] Ship ship)
{
    if (!ModelState.IsValid)
    {
        _logger.LogError("Error: Invalid ship");
        return BadRequest("Invalid ship");
    }

    var result = await IShipService.AddAsync(ship);
    if (result.IsSuccess)
        return CreatedAtRoute("get", new { id = result.Ship.Id }, result.Ship);

    return BadRequest(result.ErrorMessage);
}
```

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. **Ship:** This is a ship 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 AddAsync method.

```
public async Task<(bool IsSuccess, Ship Ship, string ErrorMessage)> AddAsync(shipmanagement.viewmodels.Ship ship)
{
    try
    {
        _logger.LogInformation("About to convert ship view model to ship dto");

        var isValid = await ship.RemoveSpace()    // RemoveSpace---> Extension method to remove the space
              .ValidateShipNameAndCode(_shipRepository); // Extension method to check if the name already exists in the database
        if (!isValid)
            throw new Exception("Duplicate ship name or code");

        var shipDTO = _mapper.Map<shipmanagement.dal.dto.Ship>(ship);
        await _shipRepository.AddAsync(shipDTO); // add the ship object into the database
        await _shipRepository.CommitAsync(); // commit once everything is done.

        _logger.LogDebug("Ship data has been commited into the database.");

        ship.Id = shipDTO.Id;
        return (true, ship, 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>
{
    2 references
    public  Task AddAsync(T entity);
    5 references
    public Task<T> GetAsync(Expression<Func<T, bool>> filt
    2 references
    public Task<IEnumerable<T>> GetAsync();

    2 references
    Task DeleteAsync(Expression<Func<T, bool>> filter);
    4 references
    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.

```
public abstract class Repository<T> : IRepository<T> where T : class
{
    internal ShipManagementDbContext dbContext;
    internal DbSet<T> dbSet;

    1 reference
    public Repository(ShipManagementDbContext 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>
    2 references
    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>
    5 references
    public async Task<T> GetAsync(Expression<Func<T, bool>> filter)
    {
        return await dbSet.FirstOrDefaultAsync(filter);
    }


2 references
public Task DeleteAsync(Expression<Func<T, bool>> filter)
{
    var itemToDelete = dbSet.First(filter);
    dbContext.Remove(itemToDelete);
    return Task.CompletedTask;
}
/// <summary>
/// Get collection of entities
/// </summary>
/// <returns>collection of entities</returns>
2 references
public async Task<IEnumerable<T>> GetAsync()
{
    return await dbSet.ToListAsync();
}

/// <summary>
/// Save changes on entity
/// </summary>
/// <returns></returns>
4 references
public async Task CommitAsync()
{
    await dbContext.SaveChangesAsync();
}
```

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 ShipManagement.

```
services.AddDbContext<ShipManagementDbContext>(opt =>
                                        opt.UseInMemoryDatabase("ShipManagement"));
```

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

```
services.AddScoped<IShipRepository, ShipRepository>();
```

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

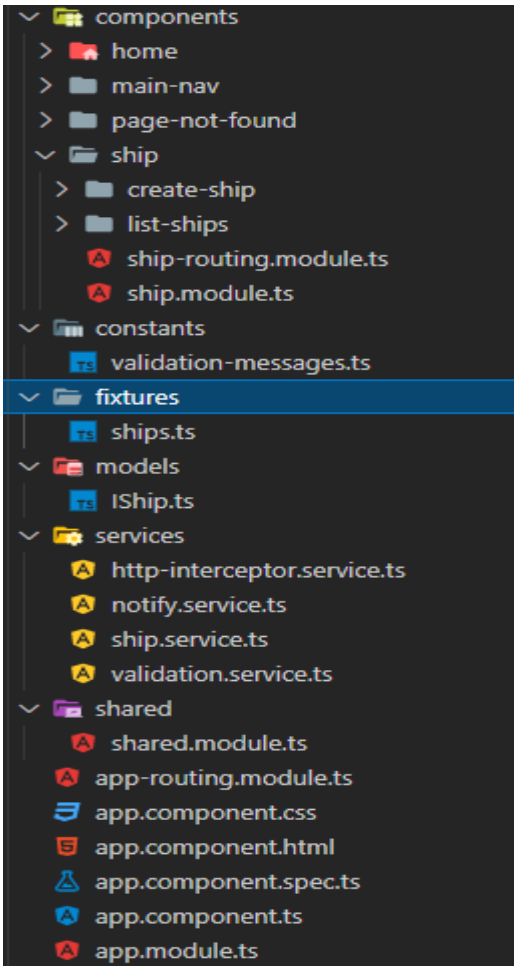
```
var mapperConfig = new MapperConfiguration(mc =>
{
mc.AddProfile(new MappingProfile());
});
IMapper mapper = mapperConfig.CreateMapper();
services.AddSingleton(mapper);
```

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 ship, 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 Add Ship and Ship Management on the left panel. On click the different link, it will display add form or list of ships accordingly.
3) Home Component gives an overview of the project.
4) Create Ship Component is used to create and edit the ship. Different validations are implemented in different fields. Code will take a specific format "AAAA-1111-B1". Name and code cannot be repeated, they are unique throughout the application.
5) List Ship Component shows the list of ships stored in the memory. It fetches the list from the web API and displays it in an angular table. It does support the filtering and sorting over the table.



Note: The module ship is created to support the lazy loading ship component in the future.

Shared Module: The shared module mainly contains the things which we want to export. I have included multiple modules that will be used by other modules. This helps in removing the redundancy of the code.

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) Ship Service: This is the main service to interact with the Rest API. All the requests and 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) Validation Service: It's a generic service to whom we pass the Form group that we want to validate and in return, it sends back the key-value pair of the property and error message if any.

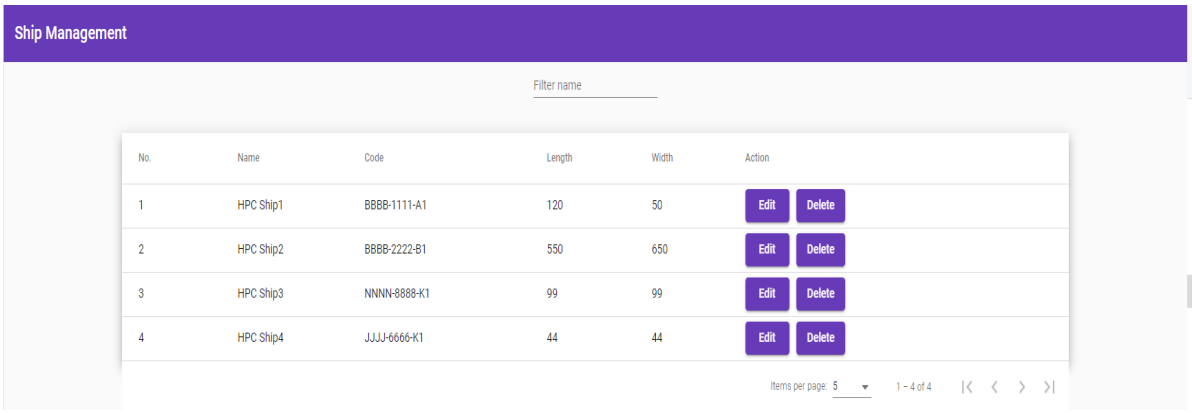
5) Unit Testing: Jasmine and karma framework are used for testing the functionality of the components. I have mocked the service to return the dummy data in case of interaction for service.

WEBSITE OVERVIEW

This application has three screens: **Ship Management, Add Ship, and Edit Ship**. All screens are briefly described below.

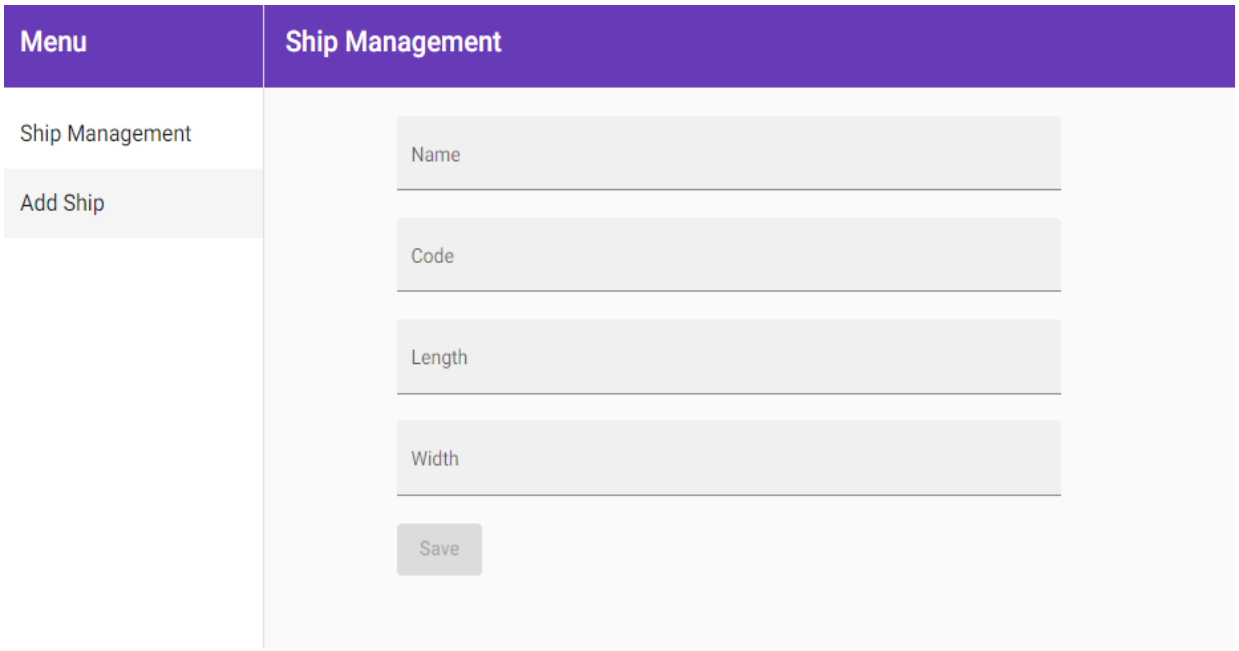
SHIP MANAGEMENT

The screen contains details of all the ships in the system. We can perform **edit, delete, search, sort, and pagination operations** from this screen. You can open add and edit form from this screen. Below is the screen for ship management.



ADD SCREEN

The ship screen has four text boxes to insert the value and all the fields have required validation. Length and Width only accept the numeric data i.e. 2 or 2.22. Code field only accepts a particular format of data i.e. "**AABB-1111-C1**". The **name and code field have unique validation** which works on the change event of the textbox.



EDIT SCREEN

The screen is similar to the Add screen with prepopulated values associated with the record. We can change values and hit save to update the values.

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.
- Change docker configuration so it uses the server's name while making the request from angular to web API.