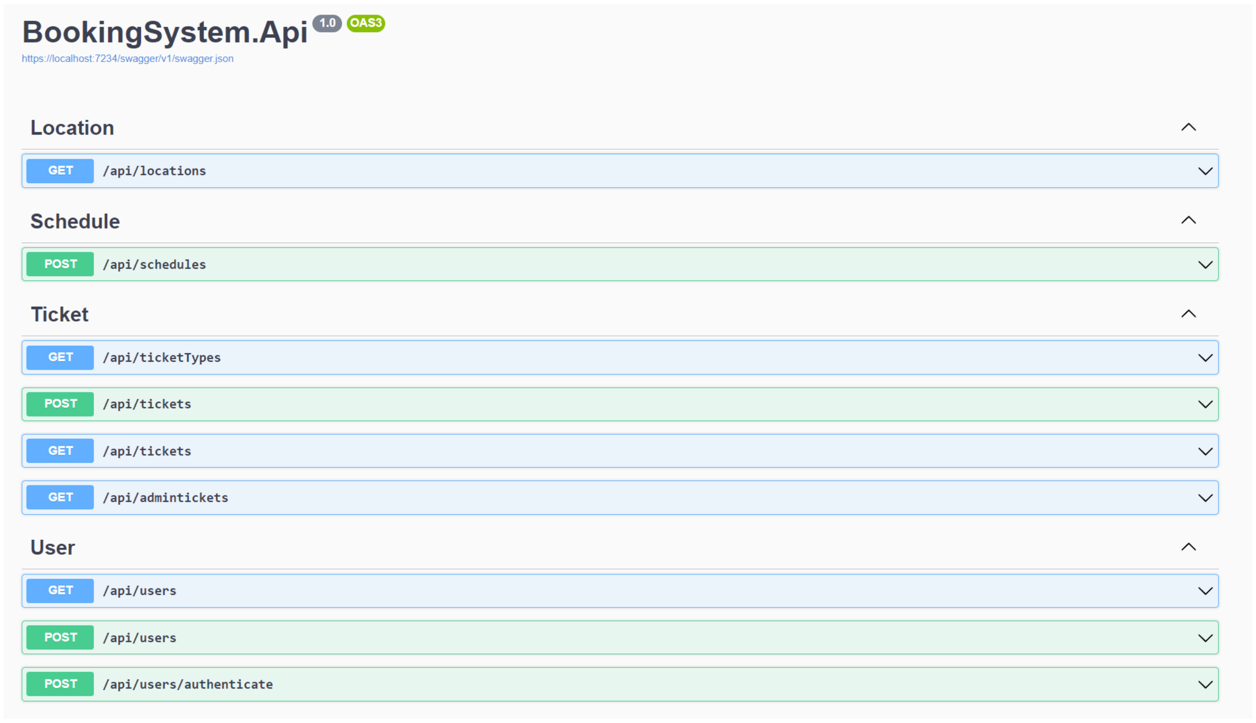
INTRODUCTION

CODING TASK: Booking System

This is a small application to manage bookings for trains, buses, and planes that will allow the user to Register, Login, Search Bookings, Book Ticket, Generate Confirm Ticket XML, View Ticket History, View All Users Bookings, and Download User Bookings.

It provides below REST endpoints:



BookingSystem.Api 1.0 OAS3

https://localhost:7234/swagger/v1/swagger.json

Location

GET /api/locations

Schedule

POST /api/schedules

Ticket

GET /api/ticketTypes

POST /api/tickets

GET /api/tickets

GET /api/admintickets

User

GET /api/users

POST /api/users

POST /api/users/authenticate

The project has been created using the **.Net Core Web API** (**Dependency Injection — Auto Mapper — Repository Pattern – Entity Framework In-Memory**), **Angular 16+ 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
- 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.

- ☐ 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 https://localhost:7234/swagger/index.html**.** From this URL we can directly access the endpoints to create and request a user and tickets.

- ☐ GitHub Code Repository: https://github.com/jhinganankita/booking

- ☐ 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/account/login.

WEB API ARCHITECTURE

Client (Web browser/Mobile/Other Micro-service) will request/create a user that will interact with an API layer that acts as a frontend for clients. The 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 be used 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.

- **NUnit Test:** I have used the NUnit test framework for unit testing. I have used In-Memory to test the API calls. I have created unit tests for LocationService, ScheduleService, and RouteService. All test cases are not taken into account as this is a basic application.

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 'BookingSystem'' solution in Visual Studio 2022, it will look like the below image:

The above image includes Api, dal, Services, and UnitTest. All of these are briefly explained below.

API: Open the TicketController, it's available in the Controller folder under the API. We have injected services named ILogger, ITicketTypeService, ITicketService, and IUserService. Also, it includes three functions “Add, Update, GetById, Get, and Delete”. In the Add function, we are getting the TicketsDto model from the request, and we are passing it to the ITicketService acting as a service.

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

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. **DTO object:** This is an 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.

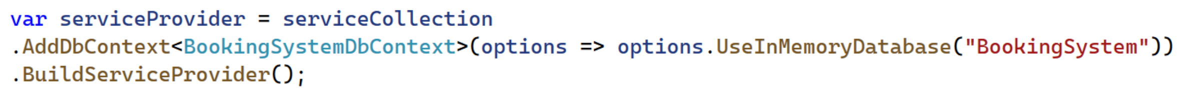
DAL: In the data access layer, we have used the generic repository. The generic IRepository interface, which will handle most of the tasks. Below is the code sample for the interface.

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. In the GET method, you can return IQueryable<T> which is a better approach. Below is the code sample for the generic repository.

Dependency Injection

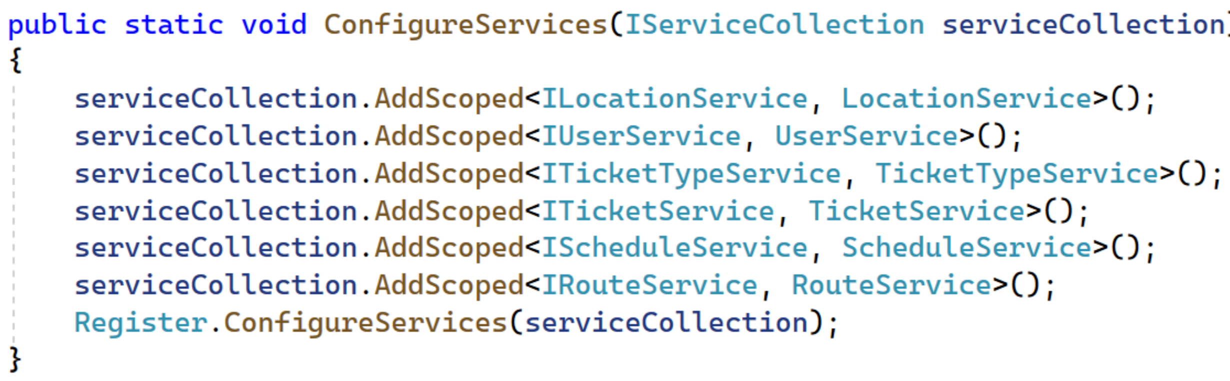
Open the Startup.cs file present in the API project. In this, we have called a class Registration.ConfigureServices. This class is available in the services layer, where the services layer registers services interfaces with the services implementation.

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



```
var serviceProvider = serviceCollection
.AddDbContext<BookingSystemDbContext>(options => options.UseInMemoryDatabase("BookingSystem"))
.BuildServiceProvider();
```

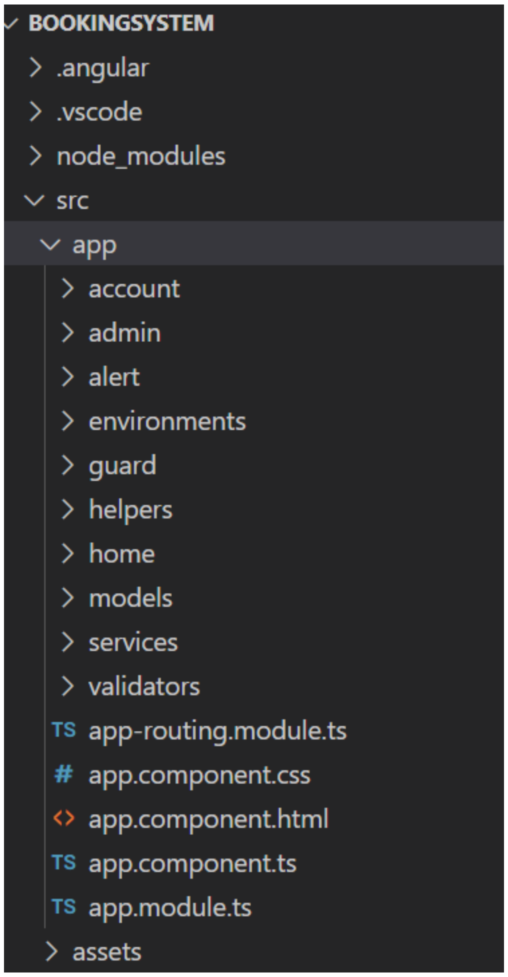
To register a service, we use the below syntax. Interface name and service name.



```
public static void ConfigureServices(IServiceCollection serviceCollection)
{
    serviceCollection.AddScoped<ILocationService, LocationService>();
    serviceCollection.AddScoped<IUserService, UserService>();
    serviceCollection.AddScoped<ITicketTypeService, TicketTypeService>();
    serviceCollection.AddScoped<ITicketService, TicketService>();
    serviceCollection.AddScoped<IScheduleService, ScheduleService>();
    serviceCollection.AddScoped<IRouteService, RouteService>();
    Register.ConfigureServices(serviceCollection);
}
```

FRONTEND ARCHITECTURE

Below is the application folder structure.



```
BOOKINGSYSTEM
  > .angular
  > .vscode
  > node_modules
  v src
    v app
      > account
      > admin
      > alert
      > environments
      > guard
      > helpers
      > home
      > models
      > services
      > validators
      TS app-routing.module.ts
      #  app.component.css
      <> app.component.html
      TS app.component.ts
      TS app.module.ts
    > assets
```

Account: It contains Layout, Login, and Register components. In addition, it contains account folder routing settings. This folder handles login and registration functionality. Currently, I am using Integer/ Number as UserId but we can enhance it to GUID.

Admin: It contains Admin components which shows all users and their ticket bookings. It is accessed by the Admin role. Currently, I am not using JSON Web Token (JWT) based authentication because of the timeline.

Alert: It handles alerts in the account module.

Environments: It handles the WEB API URL information and setup.

Guard: It contains configuration for restricting Admin role to Admin URLs.

Helpers: It contains common classes. Currently, it has a Class for AuthGuard for Admin URLs.

Helpers: It contains components for user functions like Booking, ConfirmBooking, Main-Nav, TicketHIstory, and Home.

Models: It contains model classes.

Services: It contains service classes used to handle different API calls.

Validators: It contains validation classes used in templates/UI for validation purposes.

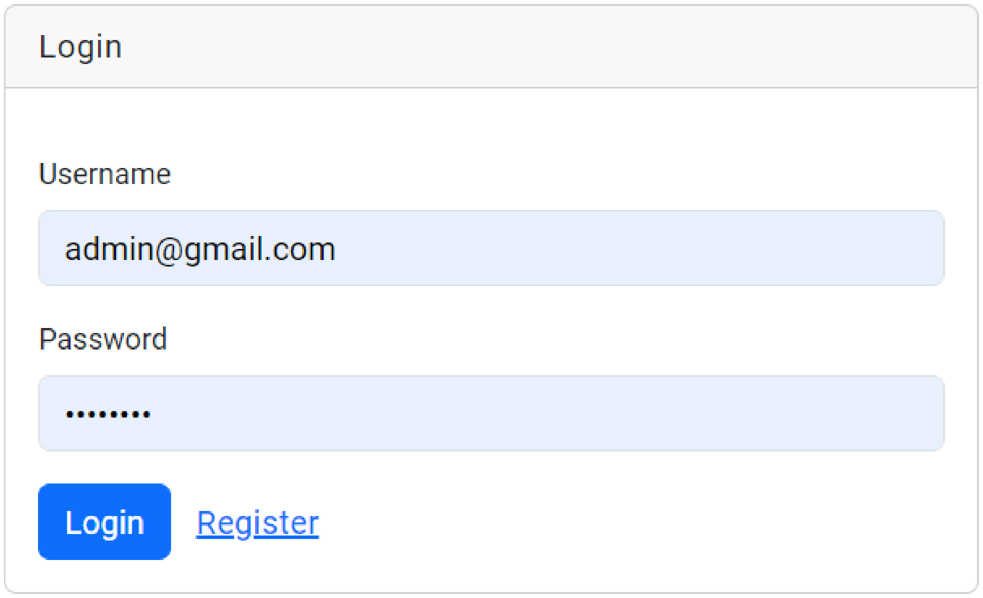
Assets: It contains any other styles, documents, etc.

WEBSITE OVERVIEW

This application has three screens: **Login, Register, Admin Dashboard, Booking Dashboard (Search Bookings), Book Ticket Modal, and Booking History**. All screens are briefly described below.

LOGIN

The screen contains a login form having Username (Email) and Password. Passwords must be between 8 - 20 characters long. Currently, passwords are not encrypted.


Login

Username

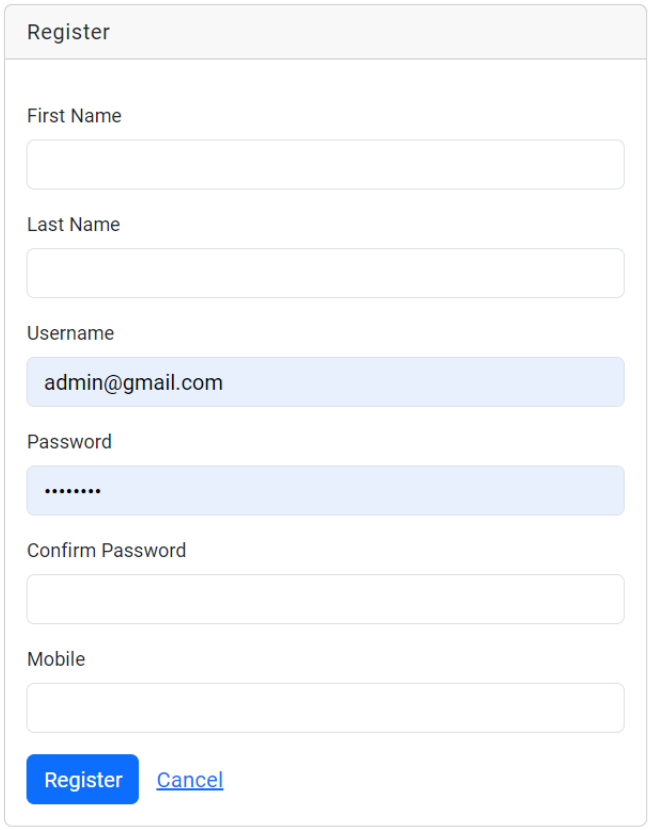
admin@gmail.com

Password

•••••••••

REGISTER

This screen has six text boxes and all the fields have required validation.. Passwords must be between 8 - 20 characters long.



Register

First Name

Last Name

Username

admin@gmail.com

Password

••••••••

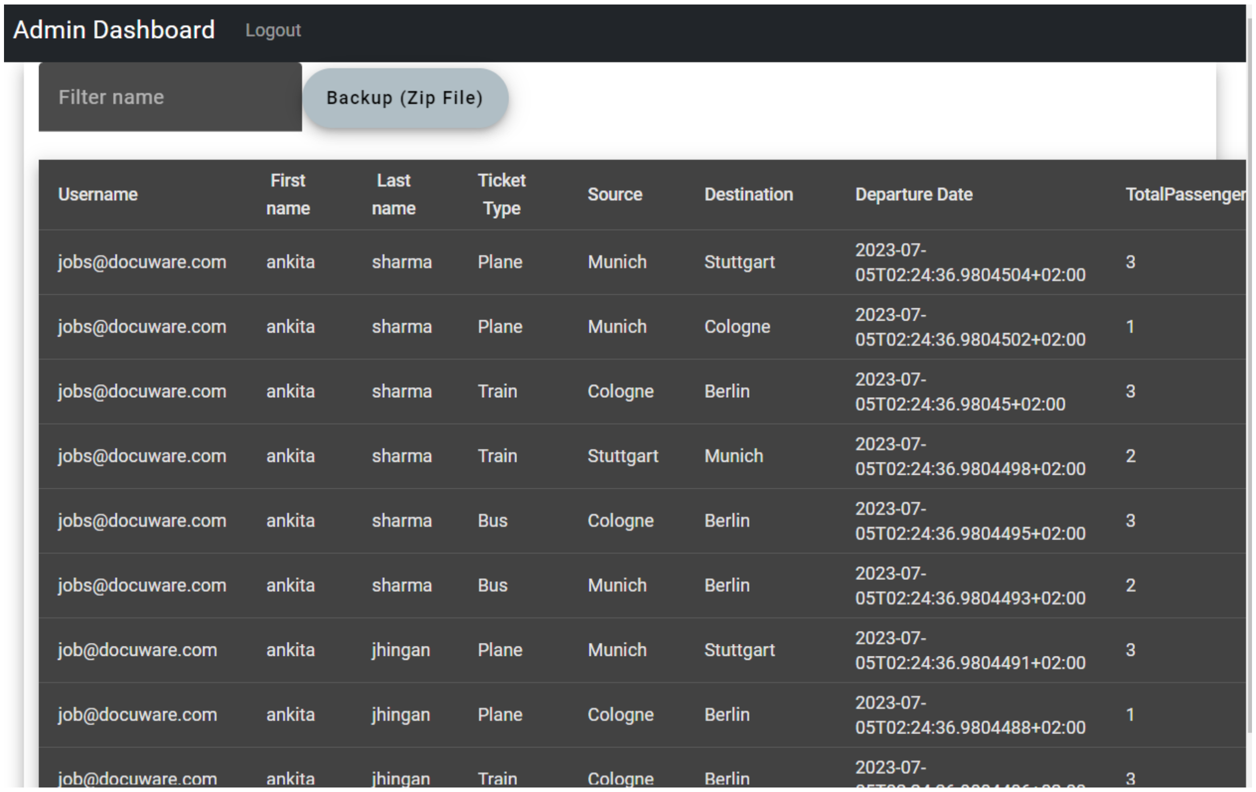
Confirm Password

Mobile

Register Cancel

ADMIN DASHBOARD

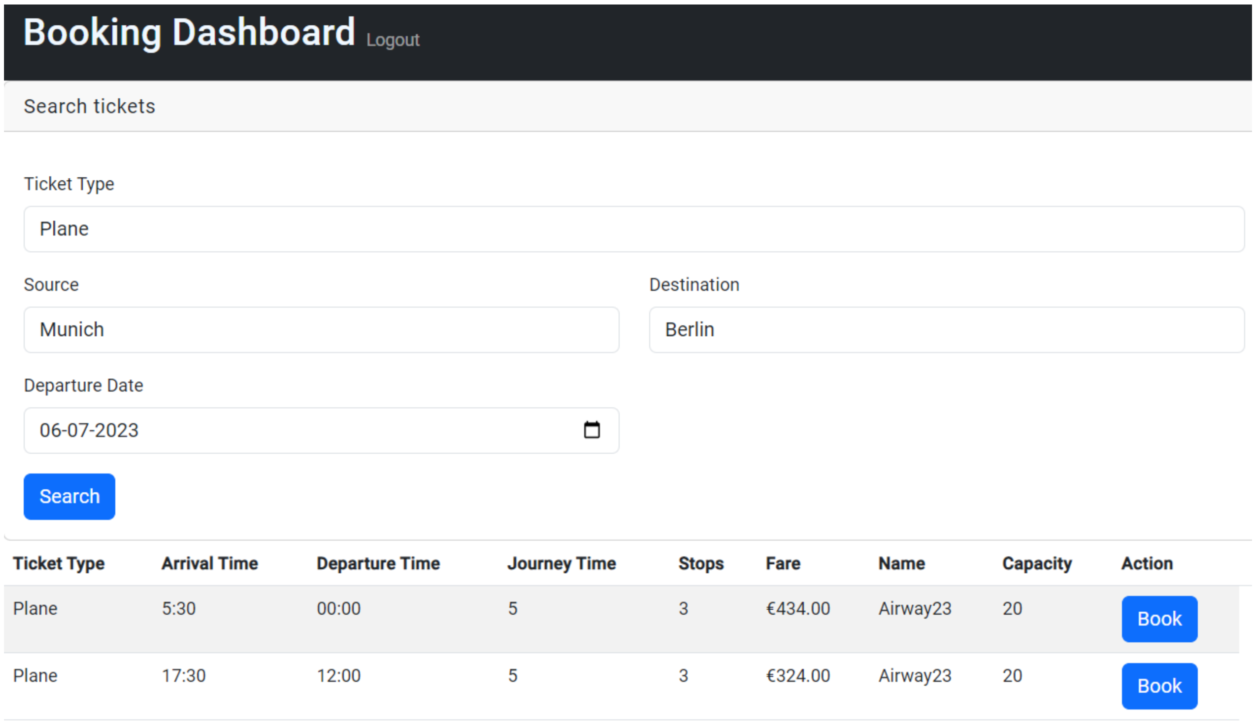
The screen shows all users' data and their tickets. You can download the folder structure of user tickets by button **“Backup (Zip File)”.** You can filter the records based on all table columns.



Username	First name	Last name	Ticket Type	Source	Destination	Departure Date	TotalPassenger
jobs@docuware.com	ankita	sharma	Plane	Munich	Stuttgart	2023-07-05T02:24:36.9804504+02:00	3
jobs@docuware.com	ankita	sharma	Plane	Munich	Cologne	2023-07-05T02:24:36.9804502+02:00	1
jobs@docuware.com	ankita	sharma	Train	Cologne	Berlin	2023-07-05T02:24:36.98045+02:00	3
jobs@docuware.com	ankita	sharma	Train	Stuttgart	Munich	2023-07-05T02:24:36.9804498+02:00	2
jobs@docuware.com	ankita	sharma	Bus	Cologne	Berlin	2023-07-05T02:24:36.9804495+02:00	3
jobs@docuware.com	ankita	sharma	Bus	Munich	Berlin	2023-07-05T02:24:36.9804493+02:00	2
job@docuware.com	ankita	jhingan	Plane	Munich	Stuttgart	2023-07-05T02:24:36.9804491+02:00	3
job@docuware.com	ankita	jhingan	Plane	Cologne	Berlin	2023-07-05T02:24:36.9804488+02:00	1
job@docuware.com	ankita	jhingan	Train	Cologne	Berlin	2023-07-	3

BOOKING DASHBOARD

The screen shows ticket search where you can search schedules depending upon **Ticket Type : Bus, Train, and Plane.** After searching schedules, you can book a ticket from a button in the Actions column of the schedule table.



Booking Dashboard Logout

Search tickets

Ticket Type: Plane

Source: Munich

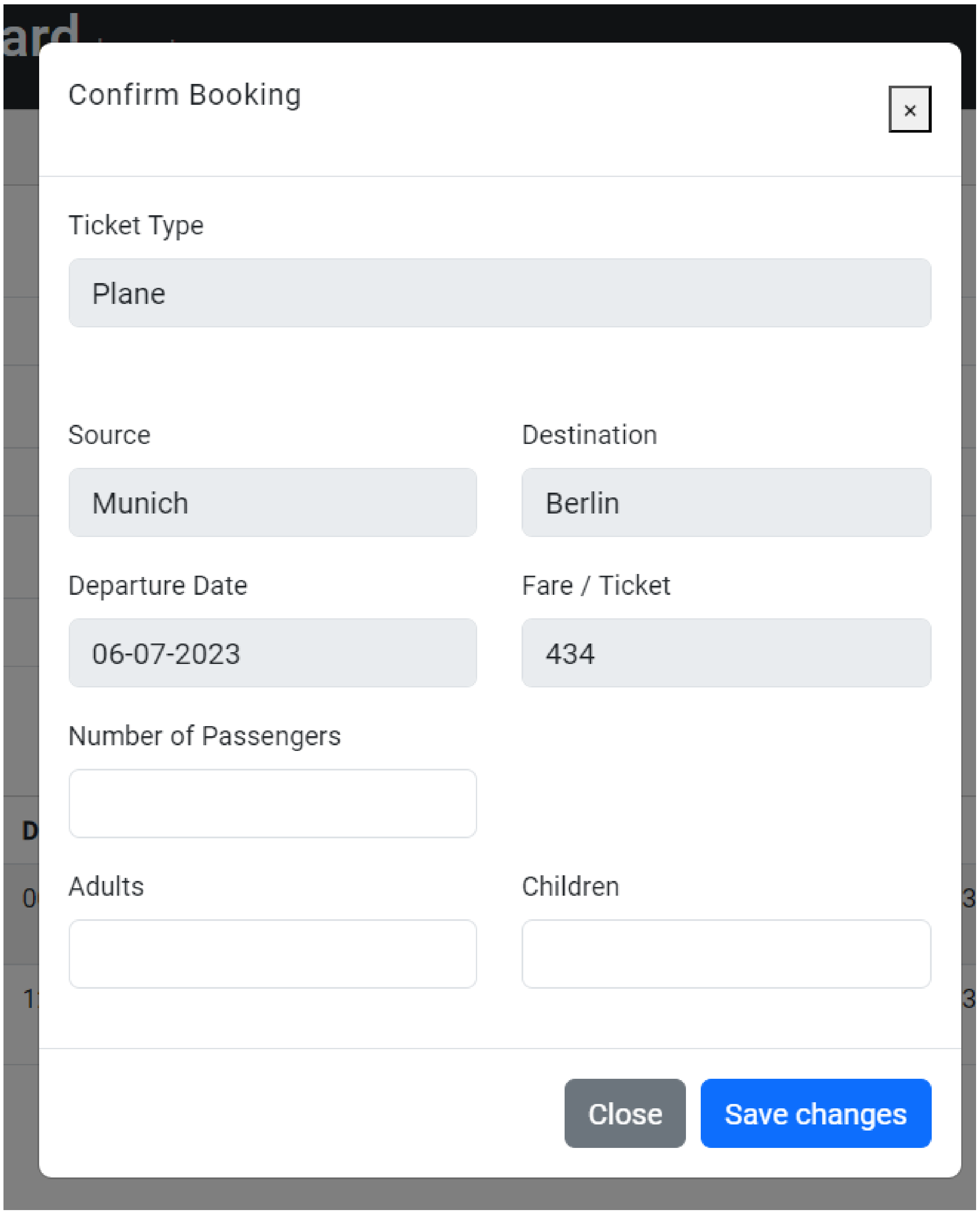
Destination: Berlin

Departure Date: 06-07-2023

Ticket Type	Arrival Time	Departure Time	Journey Time	Stops	Fare	Name	Capacity	Action
Plane	5:30	00:00	5	3	€434.00	Airway23	20	Book
Plane	17:30	12:00	5	3	€324.00	Airway23	20	Book

CONFIRM BOOKING

The popup shows a book ticket form where mostly fields are pre populated. You can add Total Passengers, Adult, and Children. On save changes, a xml file is generated to show booking details.

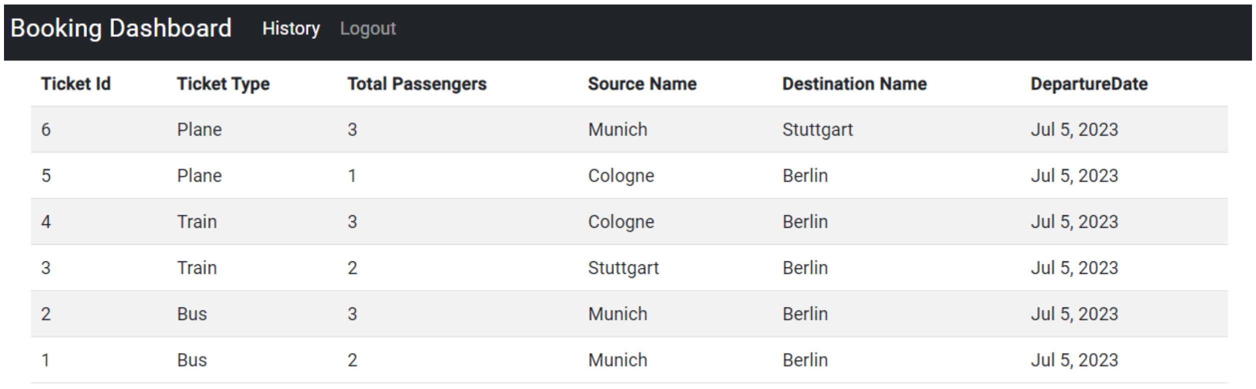


Confirm Booking

Ticket Type	
Plane	
Source	Destination
Munich	Berlin
Departure Date	Fare / Ticket
06-07-2023	434
Number of Passengers	
Adults	Children

BOOKING HISTORY

This screen is opened by clicking the History tab on the menu bar.



Ticket Id	Ticket Type	Total Passengers	Source Name	Destination Name	DepartureDate
6	Plane	3	Munich	Stuttgart	Jul 5, 2023
5	Plane	1	Cologne	Berlin	Jul 5, 2023
4	Train	3	Cologne	Berlin	Jul 5, 2023
3	Train	2	Stuttgart	Berlin	Jul 5, 2023
2	Bus	3	Munich	Berlin	Jul 5, 2023
1	Bus	2	Munich	Berlin	Jul 5, 2023

IMPROVEMENTS

Below are some of the improvements that we can implement to make the product more efficient and effective. But for this demo, I am mainly focusing on the structure.

- We can create unit test cases in angular.
- We can implement NUnit test cases for all services.
- We can add CI/CD pipeline for the project which will trigger the test cases before it deploys the build on the staging environment.
- We can add more columns to schedules and booking forms.
- We can add more strict validations, GUID UserId, JWT token based authentication and authorization.