

Mental Health Management Software

COMP 400

VASU KHANNA: 260831350

Table of Contents

Abstract	3
Introduction	3
Use Case Diagram.....	3
Wireframing.....	4
Storyboarding and Domain Model	8
Entity Relationship	8
UML Diagram (class diagram)	11
Sequence diagram.....	12
Environment Setup	13
Software Engineering Techniques	15
Observer Pattern	15
Dependency injection.....	16
Singleton pattern	17
Testing	17
End to End Testing.....	17
Unit Testing	18
Learning	19
What would you do differently	20
What would you do again	20
Things to fix.....	20
Features to add	20
References.....	20

Abstract

One reason for choosing this mental health clinic management software for a Comp 400 project is the current lack of a similar system at the university. The current method of booking appointments by phone is not efficient and causes inconvenience for patients and staff. The software will improve the appointment booking process and manage patient records and prescriptions more effectively. The software application helps healthcare facilities manage and organize their daily operations, and it's growing in demand due to the complexity and competitiveness of the healthcare industry. Additionally, as a student at the university, I have personally experienced the inefficiencies and inconvenience of the current appointment booking process and saw the need for a more streamlined system. This personal experience further solidified my decision to choose this mental health clinic management software as a topic for my Comp 400 project.

Introduction

This report will focus on the software design of a mental health clinic management software, which is specifically designed to streamline operations and provide better care for patients. The report will use, Use case diagrams to address the functional requirements, a class diagram to show the system structure including classes, attributes, relationships, and procedures. A sequence diagram will provide an overview of the functional requirements for the primary use cases, and an entity-relationship diagram will show the relationships between different entities in the database. Additionally, the report will discuss the engineering techniques and testing techniques used, and their advantages in implementing a healthcare facility.

Use Case Diagram



Wireframing

① Login Page / Register

A hand-drawn sketch of a login/register form. The form is enclosed in a rectangular border. Inside the border, there are two rectangular input fields stacked vertically. To the left of the top input field is the label "EMAIL" with an arrow pointing to the field. To the left of the bottom input field is the label "PASSWORD" with an arrow pointing to the field. Below the input fields, there are two buttons. The top button is an oval containing the word "Login", with the label "Login Button" and an arrow pointing to it. The bottom button is a rectangle containing the word "Register", with the label "Register Button" and an arrow pointing to it.

Register

button ← Register

input text

Book Appointment

button

View Doctors

ID	Name	Gender	Specialty	Specialty	Action
101	John	M	Cardiologist	Cardiologist	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

Only Admin will be able to view and action

View Patients

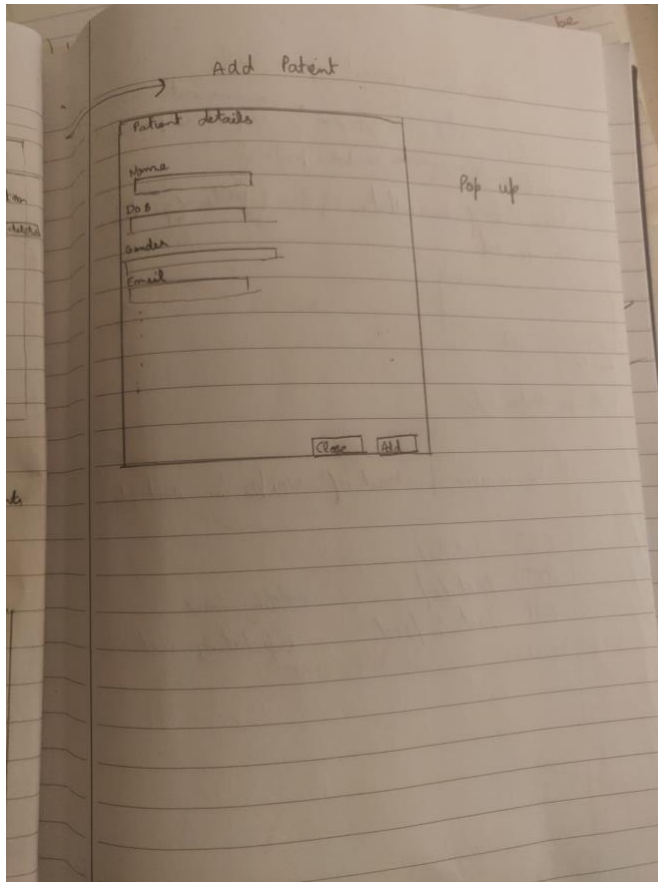
MH							
Book Appointment				Add Patient		Logout	
Patient Id	Name	Gender	Email	Language	Medical History	Action	
0A1	John M	Male	john@gmail.com	English		Add	Delete

→ Only admin and doctors can add patients and edit and delete patients

Upcoming appointments

MH						
Book Appointment				Logout		
Appointment Id	Doctor	Patient Id	Date	Time	Action	
0A1	John M	0A1	10-01-2021	10:00	Delete	

only Admin can see all appointments, doctor and patients can only see their appointments



Storyboarding and Domain Model

1. The user (patient or doctor) opens the hospital management application and is presented with a login screen.
2. After successfully logging in, the user is taken to their upcoming appointment where they can view their upcoming appointments.
3. The user can navigate to various sections of the application, such as patient management, doctor management, and book appointment system.
4. In the patient management section, the doctor/admin can view a list of patients, add new patients, and update patient information. They can also view patients' medical history.
5. In the doctor management section, the user can view a list of doctors, admin can add new doctor, and update doctor information and delete doctors.
6. The doctor can assign patients prescription and update patient medical records.
7. The user can logout from the application when they're done.

Entity Relationship

Entity: Patient

- Patient ID (primary key)
- Name

- Age
- Gender
- Phone number
- Email
- Languages

Entity: Doctor

- Doctor ID (primary key)
- Name
- Specialization
- Phone number
- Certifications
- Languages
- Email

Entity: Clinic

- Clinic ID (primary key)
- Name
- Address

Entity: Appointment

- Appointment ID (primary key)
- Patient ID (foreign key)
- Doctor ID (foreign key)
- Clinic ID (foreign key)
- Date
- Time
- Patient Email

Entity: Billing

- Billing ID (primary key)
- Appointment ID (foreign key)
- Amount

Entity: Prescription

- Prescription ID (primary key)
- Appointment ID (foreign key)
- Medicine
- Dosage

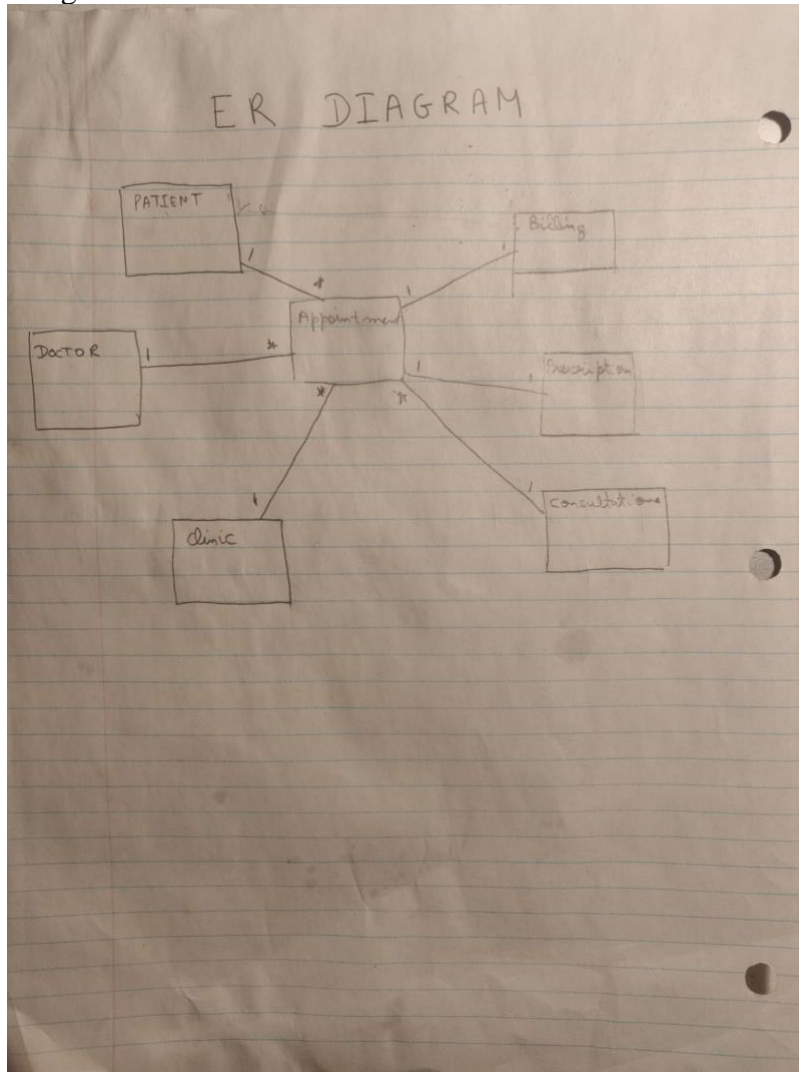
Entity: Consultation

- Consultation ID (primary key)
- Appointment ID (foreign key)
- Notes

Relationships:

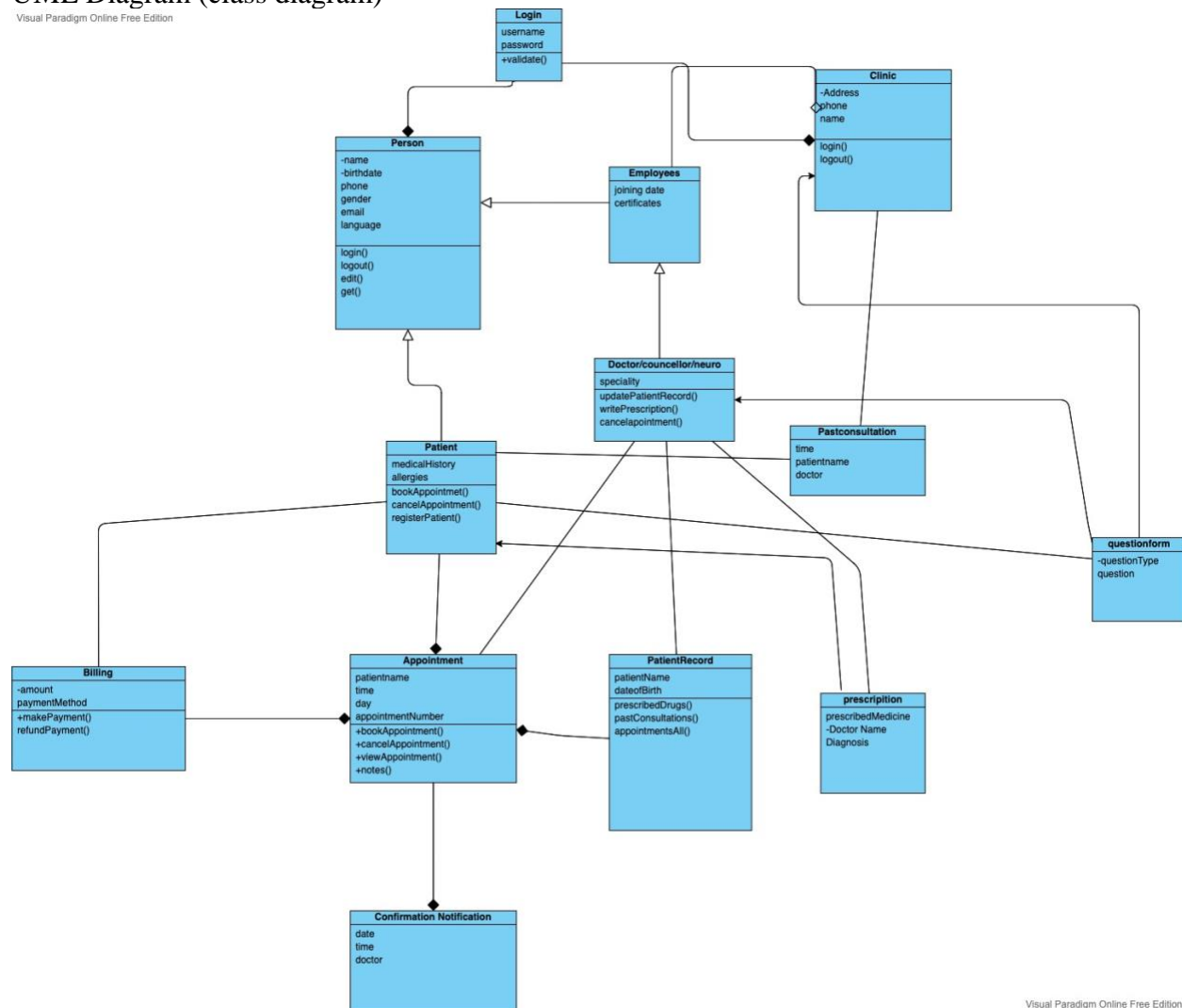
- Patient can have many appointments
- Doctor can have many appointments
- Clinic can have many appointments
- Appointment can have one patient, one doctor, and one clinic
- Appointment can have one billing and one prescription
- consultations can have many appointments

Diagram



UML Diagram (class diagram)

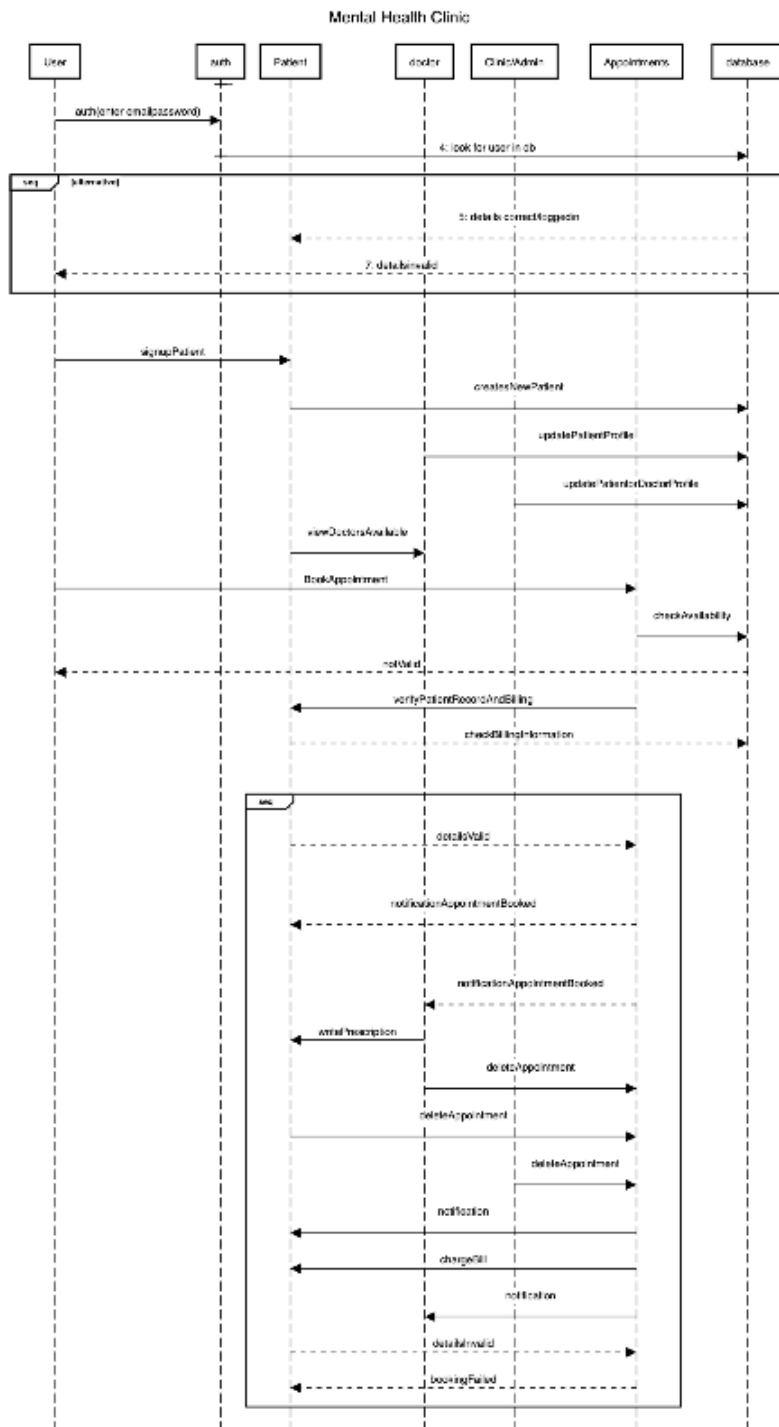
Visual Paradigm Online Free Edition



Visual Paradigm Online Free Edition

The person class and clinic have an composite relationship with login class. Employee class and Patient class extends/inherits person class. Doctor extends employee class. Appointment has a composite relationship with Billing class, confirmationNotification class, Patient Class i.e it is not possible to have an appointment without a patient. (check)Appointment should also have a composite relationship with doctor because it shouldn't be possible to book an appointment without a doctor. Doctor has an association with Prescription and Prescription has an association with Patient i.e doctor can write prescriptions to patient. Doctor has an association with patient record class i.e doctor can view patient record.

Sequence diagram



- 1) A patient visits the website and logs in or creates a new account if he hasn't already created one.
- 2) On a successful login the patient is redirected to home page where he can book an appointment.
- 3) The patient selects the doctor from the list of available doctors and selects an available date and time slot for an appointment.
- 4) On a successful booking of an appointment the patient and the doctor selected get notified.
 - On a successful booking of an appointmentClient:
 - Sends a booking request to the serverServer:
 - Validates the booking request and if booking is available. It saves the booking in the database and sends a confirmation message to the clientClient:
 - Receives the confirmation message
 - Displays a success message to the user
 - Updates the calendar display to reflect the new booking
- 5) The appointment is added to the doctor's schedule and the patient receives a confirmation email about their upcoming appointments
- 6) The patient and doctor both can view their upcoming appointments, and have an option to cancel it.
 - If the patient cancels the appointment, the system sends a notification to the doctor about the cancellation.
 - If the doctor cancels the appointment, the system sends a notification to the patient about the cancellation.
 - If the appointment is successfully cancelled, the system updates the availability of the doctor and the upcoming appointments.
- 7) The patient and doctors have an option to update their own profile. The clinic/admin has an option to add/delete/update any doctors profile or patients' profile.
- 8) Clinic/admin can view/delete any appointment whereas doctor and patients can view and delete their appointments only.
- 9) The doctor sees the patient and documents the visit in the hospital's electronic medical record system.
- 10) After the consultation, the patient is given any necessary prescriptions and the appointment is marked as completed in the system.
- 11) The patient can then log back into the website to view their updated appointment history and any prescribed medications.
- 12) After the appointment the bill gets added to the profile.
- 13) The patient pays for any fees associated with the visit (e.g., co-payments, deductibles) and schedules any follow-up appointments if needed.

Environment Setup

Tech Stack: Express.js, Angular, and Node.js, Mocha, Git, GitHub

Express.js: It is a popular framework for building web servers and APIs in JavaScript, commonly used with Node.js.

Angular: It is a JavaScript framework for building web applications, developed by Google.

Node.js: It is a JavaScript runtime built on Chrome's V8 JavaScript engine, allows to run JavaScript on the server-side.

Mocha: It is a JavaScript test framework that runs on Node.js and can be used to test both synchronous and asynchronous code.

Database: Apple File System

Apple File System is a proprietary file system developed and deployed by Apple Inc. for macOS Sierra and later. I used a simple CSV file to do the storage for my application's backend.

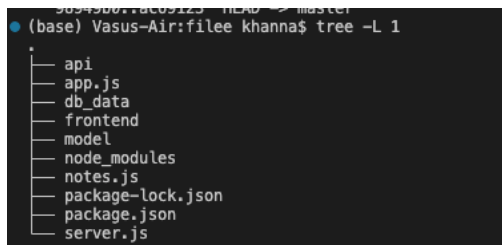
A CSV file was chosen as database option because

- Simplicity: CSV files are simple to understand and work with, making them an easy choice for prototyping.
- Cost effective
- Ease of use: It is very easy to read and write data from a CSV file, as it is just a plain text file with rows and columns.
- Compatibility: CSV files are compatible with a wide range of software, including spreadsheet programs like Microsoft Excel, Google Sheets, and more.
- Portability: CSV files can be easily shared and transferred between different systems, making them a convenient choice for prototyping.
- Flexibility: CSV files are very flexible and can be easily modified to add or remove data as needed during the prototyping process

IDE: Visual studio code

Repo: https://github.com/Vasukhanna007/comp400_MH

Directory Structure

A terminal window showing the output of the 'tree -L 1' command. The output lists the following files and directories: api, app.js, db_data, frontend, model, node_modules, notes.js, package-lock.json, package.json, and server.js.

```
(base) Vasus-Air:filee khanna$ tree -L 1
.
├── api
├── app.js
├── db_data
├── frontend
├── model
├── node_modules
├── notes.js
├── package-lock.json
├── package.json
└── server.js
```

```

19 directories, 69 files
(base) Vasus-Air:filee khanna$ tree -I node_modules frontend
frontend
├── README.md
├── angular.json
├── db.json
├── package-lock.json
├── package.json
├── src
│   ├── app
│   │   ├── app-routing.module.ts
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.component.ts
│   │   ├── app.module.ts
│   │   ├── appointment
│   │   │   ├── appointment.component.css
│   │   │   ├── appointment.component.html
│   │   │   ├── appointment.component.spec.ts
│   │   │   ├── appointment.component.ts
│   │   │   └── appointment.model.ts
│   │   ├── appointment-list
│   │   │   ├── appointment-list.component.css
│   │   │   ├── appointment-list.component.html
│   │   │   ├── appointment-list.component.spec.ts
│   │   │   └── appointment-list.component.ts
│   │   ├── doctor
│   │   │   ├── doctor.component.css
│   │   │   ├── doctor.component.html
│   │   │   ├── doctor.component.spec.ts
│   │   │   ├── doctor.component.ts
│   │   │   └── doctor.model.ts
│   │   ├── employee-dashboard
│   │   │   ├── employee-dashboard.component.css
│   │   │   ├── employee-dashboard.component.html
│   │   │   ├── employee-dashboard.component.spec.ts
│   │   │   ├── employee-dashboard.component.ts
│   │   │   └── employee-dashboard.model.ts
│   │   ├── home
│   │   │   ├── home.component.css
│   │   │   ├── home.component.html
│   │   │   ├── home.component.spec.ts
│   │   │   └── home.component.ts
│   │   ├── login
│   │   │   ├── login.component.css
│   │   │   ├── login.component.html
│   │   │   ├── login.component.spec.ts
│   │   │   └── login.component.ts
│   │   ├── shared
│   │   │   ├── api.service.spec.ts
│   │   │   └── api.service.ts
│   │   ├── signup
│   │   │   ├── signup.component.css
│   │   │   ├── signup.component.html
│   │   │   ├── signup.component.spec.ts
│   │   │   └── signup.component.ts
│   ├── assets
│   │   ├── favicon.ico
│   │   ├── index.html
│   │   ├── main.ts
│   │   └── styles.css
│   ├── tsconfig.app.json
│   ├── tsconfig.json
│   ├── tsconfig.spec.json
│   └── webpack.config.js
└── 11 directories, 52 files

```

We can see that the project is structured in a modular way, with different feature modules separated into different folders like appointment, appointment-list, doctor, employee-dashboard, home, login, shared, signup. Each of these folders contains the source files for a specific feature of the application. This way of structuring the application is referred as modular design pattern. The application uses component-based architecture which is the standard architecture pattern in Angular where each component defines a specific functionality, and the application is built by composing those components together. The application makes use of the Model-View-Controller (MVC) design pattern. where the model represents the data and business logic, the view represents the user interface, and the controller handles the communication between the model and the view. This pattern is commonly used in web development.

Software Engineering Techniques

Observer Pattern

The observer pattern is a design pattern that allows an object to be notified of changes to another object. In Angular, this pattern is implemented using the RxJS library, which provides a powerful tool for working with asynchronous data streams.

The subscribe method is a key aspect of this pattern, and it is used to register an observer (a function or object) that will be notified of changes to a data stream. observer pattern is behavioral software engineering pattern

```
import { catchError, from, Observable, of } from 'rxjs';
```

```
selectedDoctor: any; // selected doctor from the dropdown  
allDoctors$: Observable<any>;
```

```
console.log(this.timeSlot);  
this.allDoctors$ = this.api.getDoctor().pipe(  
  catchError(err => of({}))
```

The variable allDoctors\$ is an observable because it is being set to the return value of the this.api.getDoctor() method, which is also an observable. An observable is a way to handle asynchronous data streams in Angular. It allows to subscribe to updates from a data source and receive notifications whenever new data is available.

In this example, the allDoctors\$ variable is being used to hold the data stream coming from the this.api.getDoctor() method, which is making an HTTP call to retrieve a list of doctors from a backend server.

Dependency injection

Dependency injection (DI) is a technique used in software development to manage the dependencies between objects. In Angular, DI is used to provide a way for components and services to access the resources they need, such as services and other objects, without having to create or manage them directly.

Angular uses a powerful and flexible DI system to manage the dependencies between components and services. The system is based on the concept of injectors, which are responsible for creating and providing instances of objects. Dependency Injection is a software design pattern that falls under the category of "creational design patterns"

```
import { Injectable } from '@angular/core';  
import { HttpClient, HttpClientModule } from '@angular/common/http'  
import { map } from 'rxjs/operators'
```

```
@Injectable({  
  providedIn: 'root'  
})  
export class ApiService {  
  email!: string;  
  admin: boolean = false;  
  doctor: boolean=false;
```

```
  constructor(private http: HttpClient ) {} }
```


Then we inject an ApiService into all of the components that are subscribing to the subject in order to make any necessary API calls or perform any other actions based on the updated state.

```
constructor(private api: ApiService,private formbuilder: FormBuilder, private http: HttpClient){}
```

Singleton pattern

Singleton pattern is used in the API service of the application. The API service is responsible for handling all the HTTP requests to the backend server, such as retrieving and updating data. By using the Singleton pattern, it can ensure that there is only one instance of the API service across the entire application. This is important because it allows me to share the same instance of the service among all the different components that need to use it, instead of creating multiple instances of the service. This ensures that all the components are using the same data and that any changes made to the service's state are reflected in all the components. In addition, using the Singleton pattern also helps to keep track of the state of the service, and also ensures that the service is only instantiated once. This can improve the performance of the application and reduces the memory footprint. The API service is injected into different components using Angular's dependency injection mechanism. This allows me to easily use the service in any component by simply injecting it into the constructor. This way, any component that needs to make an API call or access data from the backend can easily do so by using the API service. Singleton is creational design pattern.

Testing

End to End Testing

The experiment or test case or use-case to prove your project has completed successfully. A test case where a patient logs in to the website and books an appointment with a doctor. This test case could include the following steps:

Login/Signup Test

- 1) The patient or doctor visits the website and logs in
 - The patient is only able to login if email and password is correct and doctor checkbox is unchecked, otherwise fails and gets an error message
 - The doctor is only able to login if the login information is correct and doctor checkbox is checked.
- 2) The patient can create a new account if he doesn't already have an account.

Appointment Booking Test

- 3) The patient searches for a doctor and selects one from the list of available doctors.
- 4) The patient selects a date and time for the appointment and confirms the booking.

- 5) The system sends a confirmation email to the patient and updates the doctor's schedule to reflect the new appointment.
 - The patient and doctor both can view only their upcoming appointments and have an option to cancel only their appointments.
 - The patient and doctor both receive a notification about the upcoming appointment or cancellation.
- 6) Once appointment date and time has passed it cannot be cancelled.
- 7) Application should be able to perform this test case multiple times with different patients and doctors, and verify that the system correctly handles bookings, cancellations, and updates to the coming appointments.

Update Profile Test

- 8) Patients and Clinic/Admin should have option to update patient profile. A patient should only be able to update their profile whereas a clinic should be able to update or delete any profile.

Prescription Test

- 9) Doctor should be able to add diagnosis and prescription which should be viewable by patient and clinic, it should be stored in the records and viewable by other doctors if they are assigned this patient.

Billing Test

- 10) The system should correctly calculate and records the charges for a patient's visit or treatment. This could include checking that insurance coverage is applied correctly and that the final amount charged to the patient is accurate.
- 11) It should also be able to verify if the billing information is correctly recorded and stored in the system, and that reports, and invoices can be generated as needed.

Unit Testing

Backend

```

describe('Router GET /', () => {
  it('should return an array of JSON objects', (done) => {
    chai.request('http://localhost:3001/patient')
      .get('/')
      .end((err, res) => {
        expect(err).to.be.null;
        expect(res).to.have.status(200);
        expect(res.body).to.be.an('array');
        for (let i = 0; i < res.body.length; i++) {
          console.log("i am here", res.body[i])
          expect(res.body[i]).to.be.an('object');
        }
        done();
      });
  });
});

```

Given below is example of some of the end points for which tests were written using Mocha

1. GET /doctors: This endpoint should return a list of all doctors in the hospital. It should be accessible to authorized users such as administrators, doctors, and other staff members.
2. GET /appointments: This endpoint should return a list of all appointments scheduled in the hospital. It should be accessible to authorized users such as administrators, doctors, and other staff members.
3. GET /patients: This endpoint should return a list of all patients that are registered in the hospital. It should be accessible to authorized users such as administrators, doctors and other staff members.
4. POST /appointments: This endpoint should allow authorized users such as patients or staff to create an appointment by sending a POST request with the relevant details such as date, time, patient, and doctor.
5. DELETE /appointments/:id : This endpoint should allow authorized users such as patients or staff to delete an existing appointment by sending a DELETE request with the id of the appointment.
6. DELETE /patients/:id: This endpoint should allow authorized users such as patients or staff to delete an existing patient's information by sending a DELETE request with the id of the patient.

Learning

What would you do differently

- Focus more on error handling and write more robust testing.
- Not use CSV files as database because it requires a lot more error handling and is inefficient to read file as files become large.

What would you do again

I would use a SQL or NoSQL database if I had to make this application again, as it would help eliminate a lot of code that is currently required for working with a CSV file. For example, using a database would allow for built-in methods such as `findByIdAndDelete()` instead of having to write custom code for it. Currently, I am using a CSV file as my database but using a proper database would make the application more efficient and less prone to errors.

Things to fix

- When a user refreshes the page, the environment variables are destroyed, causing the application to not know what access (admin, doctor, or patient) to grant to the user.
- The application should move from a CSV database to a proper database, as the use of a CSV file can be resource-intensive and may lead to unexpected errors if the data provided is incorrect.
- The application should include better error handling and more robust testing, such as writing integration tests to ensure that different parts of the application are working correctly together.
- The application should include a better user interface for a better user experience.

Features to add

- To Add Billing, Prescription, sending notification, video calling features to the application
- Better error handling and robust testing.
- To make it accessible and inclusive for users with disabilities and diverse needs

References

Merritt, Kamarin, and Shichao Zhao. "Software Design and Development of an Appointment Booking System: A Design Study." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020, pp. 887-892.

Robillard, Martin P. Introduction to Software Design with Java
Maple.ca(inspiration)

