

Title: Developing a Full-Stack Healthcare Appointment Scheduling System

Table of Contents	
1. Introduction	
2. Project Structure	
3. Backend Development	
- Dependencies	
- Configuration	
- Model Classes	
- Repository Interfaces	
- Service Layer	
- Controller Classes	
4. Frontend Development	
- React Components	
- State Management	
- API Integration & Error Resolving	
5. Integration	
6. Summary	

1. Introduction:

This documentation outlines the step-by-step development of a full-stack healthcare appointment scheduling system. The backend is implemented using Java Spring Boot with MongoDB as the database, while the frontend is built using React. The system provides a secure and user-friendly interface for managing healthcare appointments, including features like CRUD operations, role-based access control, and API documentation using Swagger.

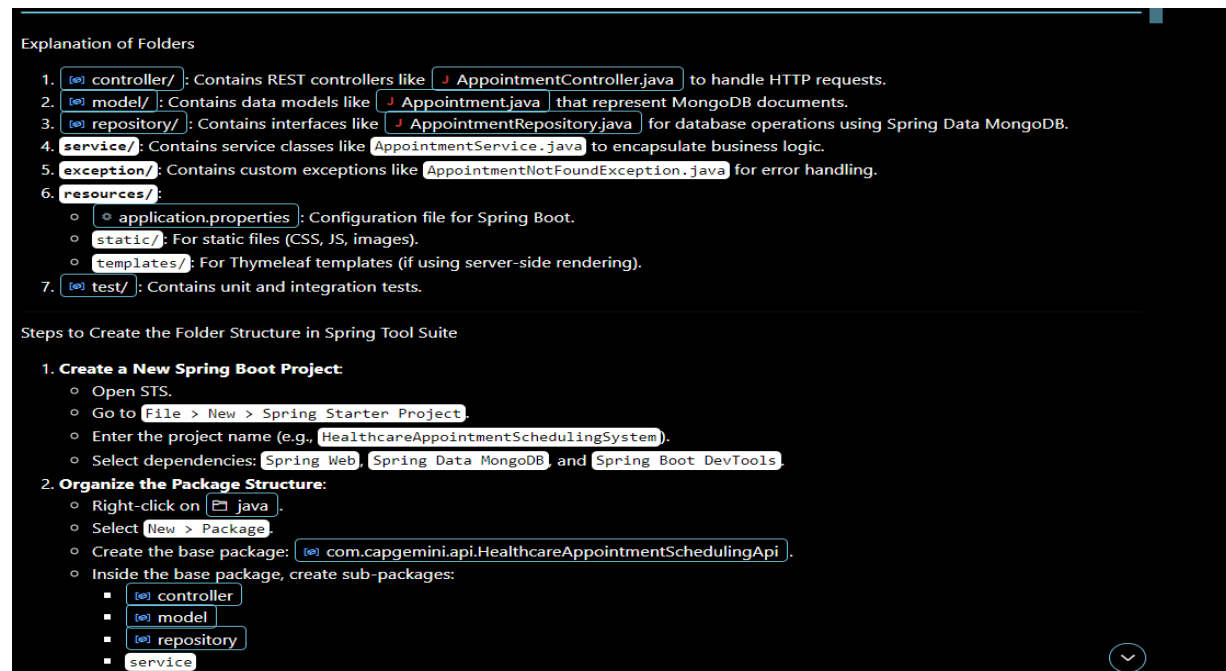
2. Project Structure: This is the project structure of the application we are creating.

Prompt 01:

"I am developing a full-stack application for a healthcare appointment scheduling system. The application will be built using Java Spring Boot for the backend and MongoDB as the database. I need guidance on the project structure and folder organization within Spring Tool Suite. Tasks: Provide a detailed project structure. Guide me in creating the folder structure in Spring Tool Suite."

Purpose: Copilot has provided me with the project structure and instructions on how to create a Spring Boot project, including the respective classes

Response:



The screenshot displays a code editor with a dark theme. It contains two sections of text. The first section, titled 'Explanation of Folders', lists seven folders and their purposes, with specific class names as examples. The second section, titled 'Steps to Create the Folder Structure in Spring Tool Suite', provides a two-step guide: '1. Create a New Spring Boot Project' and '2. Organize the Package Structure'. The second step includes a list of sub-packages to be created under a base package.

Explanation of Folders

1. `controller/`: Contains REST controllers like `AppointmentController.java` to handle HTTP requests.
2. `model/`: Contains data models like `Appointment.java` that represent MongoDB documents.
3. `repository/`: Contains interfaces like `AppointmentRepository.java` for database operations using Spring Data MongoDB.
4. `service/`: Contains service classes like `AppointmentService.java` to encapsulate business logic.
5. `exception/`: Contains custom exceptions like `AppointmentNotFoundException.java` for error handling.
6. `resources/`:
 - o `application.properties`: Configuration file for Spring Boot.
 - o `static/`: For static files (CSS, JS, images).
 - o `templates/`: For Thymeleaf templates (if using server-side rendering).
7. `test/`: Contains unit and integration tests.

Steps to Create the Folder Structure in Spring Tool Suite

1. **Create a New Spring Boot Project:**
 - o Open STS.
 - o Go to `File > New > Spring Starter Project`.
 - o Enter the project name (e.g., `HealthcareAppointmentSchedulingSystem`).
 - o Select dependencies: `Spring Web`, `Spring Data MongoDB`, and `Spring Boot DevTools`.
2. **Organize the Package Structure:**
 - o Right-click on `java`.
 - o Select `New > Package`.
 - o Create the base package: `com.capgemini.api.HealthcareAppointmentSchedulingApi`.
 - o Inside the base package, create sub-packages:
 - `controller`
 - `model`
 - `repository`
 - `service`

```

HealthcareAppointmentSchedulingApi/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   ├── com/
│   │   │   │   ├── capgemini/
│   │   │   │   │   ├── api/
│   │   │   │   │   │   HealthcareAppointmentSchedulingApi/
│   │   │   │   │   │       ├── controller/      # REST controllers
│   │   │   │   │   │       ├── service/         # Service layer
│   │   │   │   │   │       ├── repository/       # Data access layer
│   │   │   │   │   │       ├── model/           # Domain models
│   │   │   │   │   │       ├── config/          # Configuration classes
│   │   │   │   │   │       └── exception/        # Custom exception handling
│   │   │   │   ├── resources/
│   │   │   │   │   ├── application.properties    # Application configuration
│   │   │   │   │   ├── static/                  # Static files (e.g., CSS, JS)
│   │   │   │   │   └── templates/               # Thymeleaf templates (if applicable)
│   │   │   └── test/
│   │   │       ├── java/
│   │   │       │   ├── com/
│   │   │       │   │   ├── capgemini/
│   │   │       │   │   │   ├── api/
│   │   │       │   │   │   │   HealthcareAppointmentSchedulingApi/
│   │   │       │   │   │   │       └── tests/      # Unit and integration tests
│   │   │       └── .mvn/                          # Maven wrapper files
│   │   └── target/                               # Compiled output (ignored by Git)
│   └── .gitignore                                # Git ignore rules
├── HELP.md                                         # Project documentation
└── How to set up MongoDB connection?

```

I have created a project in my workspace. I am using the @workspace annotation so that Copilot can understand the context of the content in my workspace, which will help generate more relevant responses.

3. Backend Development: Here are the prompts that we for backend development .

Purpose To generate all the necessary class code along with the required dependencies.

Prompt 02:

@workspace Generate the complete code for setting up a Spring Boot application with MongoDB for a Healthcare Appointment Scheduling System.

Include the following:

Dependencies: List the necessary dependencies in the pom.xml file. Configuration:

Provide the configuration for connecting to a MongoDB database.

Model: Create a sample model class for managing healthcare appointments.

Repository: Create a repository interface for CRUD operations on healthcare appointments. Service: Create a service layer to handle the business logic for managing healthcare appointments.

Controller: Create a controller class to handle HTTP requests and responses for healthcare appointments.

Java documentation: Also generate java documentation of each cla

1. Dependencies (pom.xml)

```
<dependencies>
  <!-- Spring Boot Starter Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot Starter Data MongoDB -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>

  <!-- Spring Boot Starter Security -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <!-- Spring Boot Starter Test -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

3. Model (Appointment.java)

```
package com.capgemini.api.HealthcareAppointmentSchedulingApi.model;

import java.time.LocalDateTime;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "appointments")
public class Appointment {

    @Id
    private String id;
    private String patientName;
    private String doctorName;
    private LocalDateTime appointmentDate;
    private String reason;

    // Getters and Setters
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getPatientName() {
```

```

4. Repository (AppointmentRepository.java)

```java
// filepath: src/main/java/com/capgemini/api/HealthcareAppointmentSchedulingApi/repository/AppointmentRe
package com.capgemini.api.HealthcareAppointmentSchedulingApi.repository;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

import com.capgemini.api.HealthcareAppointmentSchedulingApi.model.Appointment;

@Repository
public interface AppointmentRepository extends MongoRepository<Appointment, String> {
}
```
5. Service (AppointmentService.java and AppointmentServiceImpl.java)

```

Copilot has provided the code for all the necessary classes required for the application.

Purpose:

To tailor the code according to the requirements..

Prompt 03:

@workspace you're doing great! Next, we will refine each class one by one as per our needs. I will specify my requirement:

Please modify the above-given controller class for managing healthcare appointments with the following format:

Security: Use @PreAuthorize for security.

API Documentation: Use @Operation for API documentation.

Response Documentation: Use @ApiResponse for response documentation.

HTTP Methods: Use @PostMapping for creating appointments, @GetMapping for retrieving appointments, and @PutMapping for updating appointments. Ensure ResponseEntity is used for responses and specify the URL in the @PutMapping and @GetMapping annotations.

Make sure you do not miss any instruction provided above.

Response:

```

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import io.swagger.v3.oas.annotations.responses.ApiResponses;
import io.swagger.v3.oas.annotations.tags.Tag;

@RestController
@RequestMapping("/api/appointments")
@Tag(name = "Healthcare-Appointment-Scheduling", description = "CRUD operations for Healthcare appointment")
public class AppointmentController {

    @Autowired
    private AppointmentService appointmentService;

    @Operation(summary = "Retrieve all appointments", description = "Fetches a list of all healthcare ap")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Successfully retrieved the list of appointment"),
        @ApiResponse(responseCode = "403", description = "Access denied")
    })
    @PreAuthorize("hasRole('ADMIN') or hasRole('USER')")
    @GetMapping
    public ResponseEntity<List<Appointment>> getAllAppointments() {
        List<Appointment> appointments = appointmentService.getAllAppointments();
        return ResponseEntity.ok(appointments);
    }

    @Operation(summary = "Create a new appointment", description = "Creates a new healthcare appointment")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "201", description = "Successfully created the appointment"),
        @ApiResponse(responseCode = "400", description = "Invalid input"),
        @ApiResponse(responseCode = "403", description = "Access denied")
    })

```

Copilot has provided me with customized code based on the instructions given in the prompt.

Purpose:

To tailor the code to meet our specific requirements.

Prompt 04:

@workspace Greate now reframe above-provided service layer for managing healthcare appointments with separate interface and implementation classes. Include detailed business logic.

```

public interface AppointmentService {

    /**
     * Retrieves all appointments.
     *
     * @return List of all appointments.
     */
    List<Appointment> getAllAppointments();

    /**
     * Retrieves an appointment by its ID.
     *
     * @param id The ID of the appointment.
     * @return Optional containing the appointment if found, or empty if not found.
     */
    Optional<Appointment> getAppointmentById(String id);

    /**
     * Creates a new appointment.
     *
     * @param appointment The appointment to create.
     * @return The created appointment.
     */
    Appointment createAppointment(Appointment appointment);

    /**
     * Updates an existing appointment.
     *
     * @param id The ID of the appointment to update.
     */

```



```

package com.capgemini.api.HealthcareAppointmentSchedulingApi.service;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.capgemini.api.HealthcareAppointmentSchedulingApi.model.Appointment;
import com.capgemini.api.HealthcareAppointmentSchedulingApi.repository.AppointmentRepository;

@Service
public class AppointmentServiceImpl implements AppointmentService {

    @Autowired
    private AppointmentRepository appointmentRepository;

    @Override
    public List<Appointment> getAllAppointments() {
        // Fetch all appointments from the database
        return appointmentRepository.findAll();
    }

    @Override
    public Optional<Appointment> getAppointmentById(String id) {
        // Fetch a specific appointment by ID
        return appointmentRepository.findById(id);
    }
}

```

It has provided me with customized code based on the instructions given in the prompt.

Purpose;

For integrating Swagger into my backend and setting up Swagger URLs

Prompt 05:

@workspace Great, the application is successfully running but I need login credentials to access above-provided swagger URL.

Set up Swagger for my Spring Boot application to view API documentation. Include the following steps:

Dependencies: Add the necessary Swagger dependencies to the pom.xml file. Configuration:

Create a configuration class for Swagger.

Security Configuration: Configure Spring Security to allow access to Swagger UI without requiring authentication.

Access: Provide the URL to access Swagger UI.

Also provide Swagger request bodies to perform CRUD operations on this application via Swagger.

Response:

1. Dependencies

Add the necessary Swagger dependencies to your `pom.xml` file. Since you already have the `springdoc-openapi-starter-webmvc-ui` dependency in your `pom.xml`, no additional dependencies are required.

Your `pom.xml` already includes:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.1.0</version>
</dependency>
```

2. Swagger Configuration

Create a configuration class for Swagger to customize the API documentation.

2. Swagger Configuration

Create a configuration class for Swagger to customize the API documentation.

```
package com.capgemini.api.HealthcareAppointmentSchedulingApi.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Info;

@Configuration
public class SwaggerConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .info(new Info()
                .title("Healthcare Appointment Scheduling API")
                .version("1.0")
                .description("API documentation for managing healthcare appointments"));
    }
}
```

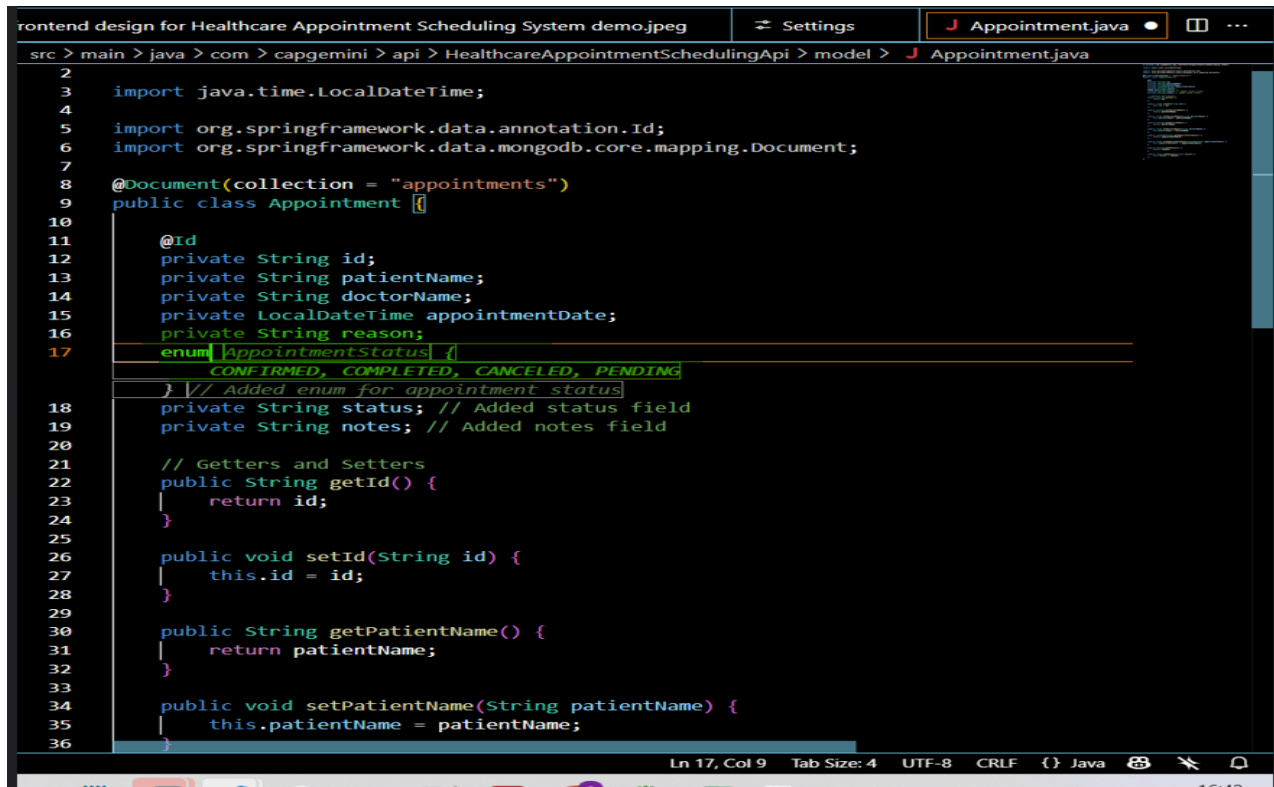
 Add Context...

Ask Copilot

Copilot has provided all the necessary code snippets for Swagger, including the required dependencies and request bodies.

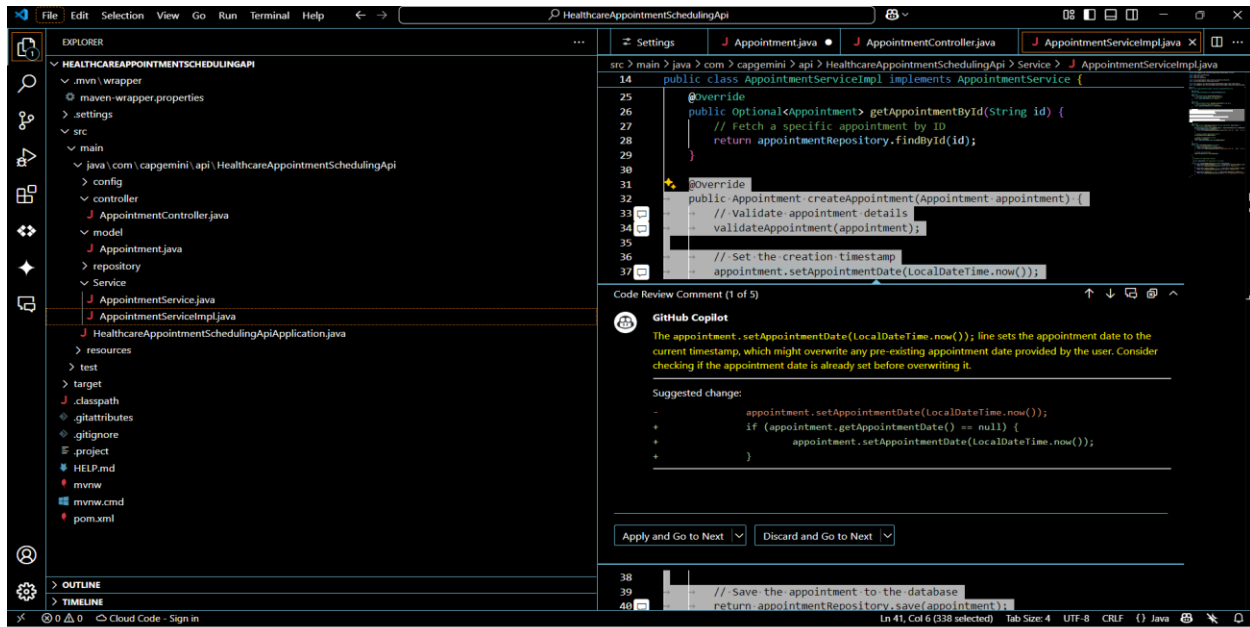
To customize our code and add snippets, we can utilize GitHub Copilot's latest feature, **"Next Edit Suggestion."** (NES).

For instance, I wanted to add a few fields to my model class, and Copilot's "Next Edit Suggestion" feature provided helpful code suggestions enhancing our coding efficiency and accuracy.



```
rontend design for Healthcare Appointment Scheduling System demo.jpeg  Settings Appointment.java
src > main > java > com > capgemini > api > HealthcareAppointmentSchedulingApi > model > Appointment.java
2
3 import java.time.LocalDateTime;
4
5 import org.springframework.data.annotation.Id;
6 import org.springframework.data.mongodb.core.mapping.Document;
7
8 @Document(collection = "appointments")
9 public class Appointment {
10
11     @Id
12     private String id;
13     private String patientName;
14     private String doctorName;
15     private LocalDateTime appointmentDate;
16     private String reason;
17     enum AppointmentStatus {
18         CONFIRMED, COMPLETED, CANCELED, PENDING
19     } // Added enum for appointment status
20     private String status; // Added status field
21     private String notes; // Added notes field
22
23     // Getters and Setters
24     public String getId() {
25         return id;
26     }
27
28     public void setId(String id) {
29         this.id = id;
30     }
31
32     public String getPatientName() {
33         return patientName;
34     }
35
36     public void setPatientName(String patientName) {
37         this.patientName = patientName;
38     }
39 }
```

To ensure the quality and maintainability of my code, I utilized GitHub Copilot's **"Code Comment & Review"** feature. This feature analyzed my code snippet and provided actionable suggestions for improvement. It helped me in **Improved Validation, Enhanced Maintainability, Auditability & Better Error Handling** in code.



Purpose: I aimed to improve the reusability of code, ensure cleaner code, and adhere to best practices. Utilizing the **Agent Mode feature of GitHub Copilot**, I refactored the validateAppointment method from the AppointmentServiceImpl class into a reusable utility class named AppointmentValidator.

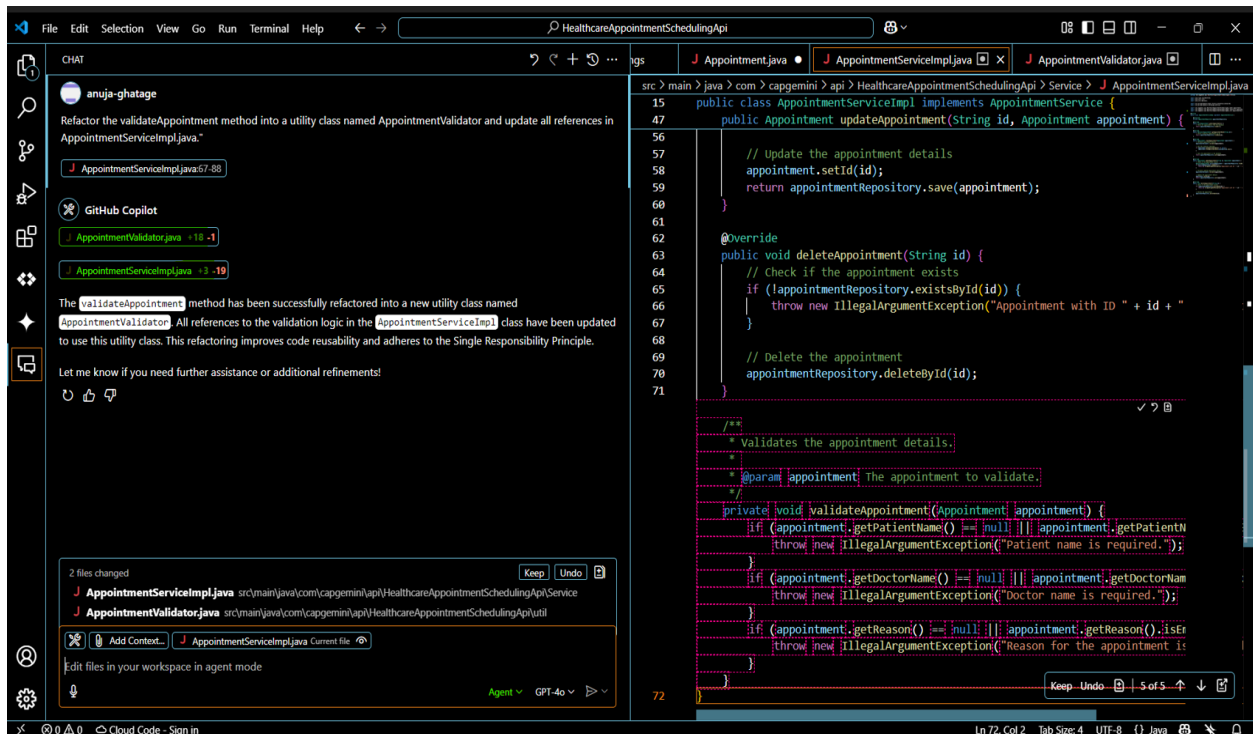
Prompt 05:

Refactor the validateAppointment method into a utility class named AppointmentValidator and update all references in AppointmentServiceImpl.java.

Response:

Agent Mode in Action:

1. Copilot analyzed the workspace and suggested creating a new utility class AppointmentValidator in the util package.
2. It moved the validation logic from AppointmentServiceImpl to the AppointmentValidator class as a static method.
3. Copilot then updated all references in the AppointmentServiceImpl class to use the new utility class



Purpose: Using **multi-edit feature** and its ability to streamline code updates across multiple files. By simply prompting the addition of a new attribute (cancellationReason) in the model class, the feature automatically applied consistent changes throughout the codebase—saving time, reducing manual effort, and minimizing the risk of errors.

Prompt 06:Add a field named as “cancellation Reason” to appointment model class and update the functions accordingly in application..

Response:

It added the field to the model and automatically updated the related code in other classes.

```

HealthcareAppointmentSchedulingApi
Appointment.java AppointmentServiceImpl.java Settings pom.xml AppointmentController.java AppointmentService.java
src > main > java > com > capgemini > api > HealthcareAppointmentSchedulingApi > model > Appointment.java
5 import org.springframework.data.annotation.Id;
6 import org.springframework.data.mongodb.core.mapping.Document;
7
8 @Document(collection = "appointments")
9 public class Appointment {
10
11     @Id
12     private String idNotGenerated; // ID not generated by the database
13     private String patientNameAlphabetical;
14     private String doctorName;
15     private LocalDateTime appointmentDate;
16     private String reason;
17     private String status; // String data type for status
18     private String notes; // String data type for notes
19     private double fees;
20     private String nurse;
21     private String cancellationReason; // Added field
22
23     // Getters and Setters
24     public String getId() {
25         return id;
26     }
27
28     public void setId(String id) {
29         this.id = id;
30     }
31
32     public String getPatientName() {
33         return patientName;
34     }
35
36     public void setPatientName(String patientName) {
37         this.patientName = patientName;
38     }
39
40     public String getDoctorName() {
41         return doctorName;
42     }
43
44     public void setDoctorName(String doctorName) {
45         this.doctorName = doctorName;
46     }
47
48     public LocalDateTime getAppointmentDate() {

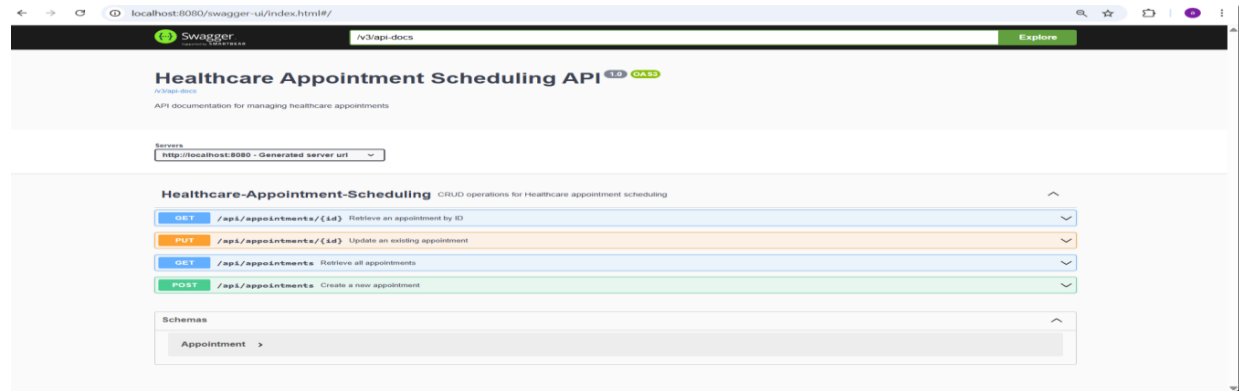
```

```

HealthcareAppointmentSchedulingApi
AppointmentRepository.java AppointmentServiceImpl.java Settings pom.xml AppointmentController.java AppointmentService.java
src > main > java > com > capgemini > api > HealthcareAppointmentSchedulingApi > controller > AppointmentController.java
29 public class AppointmentController {
67
68     @Operation(summary = "Update an existing appointment", description = "Updates the details of an existing healthcare a")
69     @ApiResponse(responseCode = "200", description = "Successfully updated the appointment"),
70     @ApiResponse(responseCode = "404", description = "Appointment not found"),
71     @ApiResponse(responseCode = "400", description = "Invalid input"),
72     @ApiResponse(responseCode = "403", description = "Access denied")
73     @PreAuthorize("hasRole('ADMIN')")
74     @PutMapping("/{id}")
75     public ResponseEntity<Appointment> updateAppointment(@PathVariable String id,
76     @RequestBody Appointment appointment) {
77         return appointmentService.getAppointmentsById(id).map(existingAppointment -> {
78             appointment.setId(id);
79             appointment.setFees(appointment.getFees());
80             appointment.setNurse(appointment.getNurse());
81             Appointment updatedAppointment = appointmentService.updateAppointment(id, appointment);
82             return ResponseEntity.ok(updatedAppointment);
83         }).orElse(ResponseEntity.notFound().build());
84
85     @Operation(summary = "Update cancellation reason for an appointment", description = "Updates the cancellation reason")
86     @ApiResponse(value = { @ApiResponse(responseCode = "200", description = "Successfully updated the cancellation reason"),
87         @ApiResponse(responseCode = "404", description = "Appointment not found"),
88         @ApiResponse(responseCode = "403", description = "Access denied") })
89     @PreAuthorize("hasRole('ADMIN')")
90     @PatchMapping("/{id}/cancellation-reason")
91     public ResponseEntity<Appointment> updateCancellationReason(@PathVariable String id,
92     @RequestParam String cancellationReason) {
93         Appointment updated = appointmentService.updateCancellationReason(id, cancellationReason);
94         if (updated != null) {
95             return ResponseEntity.ok(updated);
96         } else {
97             return ResponseEntity.notFound().build();
98         }
99     }
100 }

```

Below is an image showing how the backend looks (swagger view) after integrating the code provided by GitHub Copilot:



Prompts for developing frontend UI:

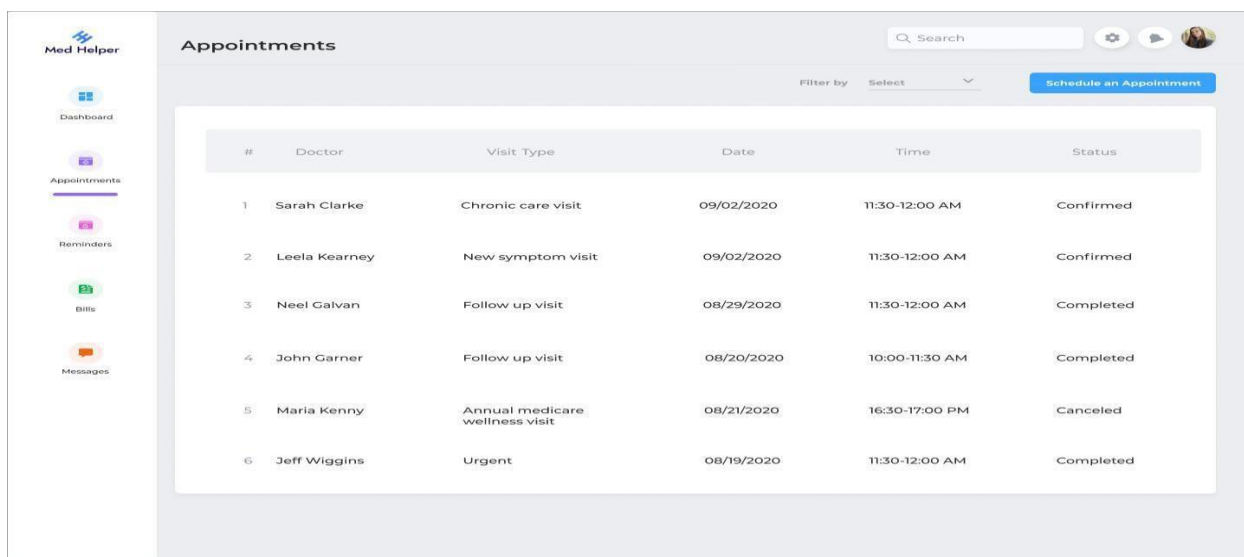
Using GitHub Copilot's Vision feature, we streamlined the front-end development process. By providing an image reference, Copilot generated the necessary React code for our application, significantly accelerating our workflow.

Purpose: To generate UI code.

Prompt 01:

@workspace Let's create the front end for this application. Go through my provided image and please provide me front-end code in React for the application.

Attached is an image for reference of Copilot:



#	Doctor	Visit Type	Date	Time	Status
1	Sarah Clarke	Chronic care visit	09/02/2020	11:30-12:00 AM	Confirmed
2	Leela Kearney	New symptom visit	09/02/2020	11:30-12:00 AM	Confirmed
3	Neel Galvan	Follow up visit	08/29/2020	11:30-12:00 AM	Completed
4	John Garner	Follow up visit	08/20/2020	10:00-11:30 AM	Completed
5	Maria Kenny	Annual medicare wellness visit	08/21/2020	16:30-17:00 PM	Canceled
6	Jeff Wiggins	Urgent	08/19/2020	11:30-12:00 AM	Completed

Response:

```
1. App.js
This is the main entry point of the React application.

import React from "react";
import Appointments from "../components/Appointments";
import "../App.css";

function App() {
  return (
    <div className="App">
      <Appointments />
    </div>
  );
}

export default App;
```

```
2. Appointments.js
This component displays the appointments table and the sidebar.

import React from "react";
import "../Appointments.css";

const Appointments = () => {
  // ...
}
```

 Add Context...

Copilot has provided all the necessary class code.

Purpose:

To implement the desired changes in the UI

Prompt 02:

@workspace Great, the UI looks good, but I have a few changes to suggest. Please update the provided code accordingly:

Task:

Center the appointment table.

Add icons to the sidebar to enhance its appearance and make it more intuitive. Additionally, could you provide me with the steps to add icons?

Response

Update AppointmentTable.css:

```
.appointment-table-container {
  display: flex;
  justify-content: center; /* Center horizontally */
  align-items: center; /* Center vertically */
  height: 100vh; /* Full viewport height */
}

.appointment-table {
  width: 80%; /* Adjust the width as needed */
  border-collapse: collapse;
  background-color: #ffffff;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
  border-radius: 8px;
  overflow: hidden;
}

.appointment-table th,
.appointment-table td {
  border: 1px solid #ddd;
  padding: 12px;
  text-align: left;
}

.appointment-table th {
  background-color: #f4f4f4;
```

Update AppointmentTable.js:

Wrap the `<table>` element in a container div with the class `appointment-table-container`:

```
import React, { useEffect, useState } from "react";
import axios from "axios";
import "../AppointmentTable.css";

function AppointmentTable() {
  const [appointments, setAppointments] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    axios
      .get("http://localhost:8080/api/appointments")
      .then((response) => {
        setAppointments(response.data);
        setLoading(false);
      })
      .catch((error) => {
        console.error("Error fetching appointments:", error);
        setLoading(false);
      });
  }, []);

  if (loading) {
    return <p>loading appointments...</p>;
```

Purpose:

To integrate the UI with the backend.

Prompt 03:

@workspace Great, the UI is working perfectly! Now, I need to integrate this UI with my backend service. Could you provide me with the code and steps to achieve this integration?

Response:

The backend must allow requests from the React frontend. You already have a `CorsConfig` class in your backend. Ensure it is correctly configured to allow requests from `http://localhost:3000` (the default React development server).

Here's your existing `CorsConfig` class for reference:

```
package com.capgemini.api.HealthcareAppointmentsSchedulingApi.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**") // Allow all endpoints
                    .allowedOrigins("http://localhost:3000") // Allow requests from the React frontend
                    .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS") // Allow specific HTTP methods
                    .allowedHeaders("*") // Allow all headers
                    .allowCredentials(true); // Allow cookies
            }
        };
    }
}
```

3. Create an Axios Instance

Create a reusable Axios instance to simplify API calls.

Create `api.js`:

```
import axios from "axios";

const api = axios.create({
  baseURL: "http://localhost:8080/api", // Base URL of your backend
  headers: {
    "Content-Type": "application/json",
  },
});

export default api;
```

Copilot has also provided other updated classes in its response.

Purpose:

I was facing errors, so I provided the error message to Copilot to help resolve them.

Prompt 04:

@workspace im getting below error in browser Compiled with problems:

x

```
ERROR in ./src/components/AppointmentTable.js 6:0-25
Module not found: Error: Can't resolve '../api' in
'C:\Users\AANILGHA\healthcare_appointment_schedule\src\components' ERROR in
./src/components/Header.js 6:0-25
Module not found: Error: Can't resolve '../api' in
'C:\Users\AANILGHA\healthcare_appointment_schedule\src\components'
```

Response:

Copilot has provided multiple solutions to fix the above error:

```
import axios from "axios";

const api = axios.create({
  baseURL: "http://localhost:8080/api", // Base URL of your backend
  headers: {
    "Content-Type": "application/json",
  },
});

export default api;
```

2. Verify File Location

Ensure the `api.js` file is located in the `src` directory, not in any subdirectory. The structure should look like this:

```
src/
├── api.js
├── components/
│   ├── AppointmentTable.js
│   └── Header.js
├── App.js
└── index.js
```

3. Correct Import Statements

In both `AppointmentTable.js` and `Header.js`, ensure the import statement for `api.js` is correct:

`AppointmentTable.js`

4. Restart the Development Server

After making the changes, restart your React development server to ensure the changes are applied:

```
npm start
```

5. Check for Typos

Ensure there are no typos in the file name (`api.js`) or the import paths. File names in JavaScript are case-sensitive.

6. Clear Cache (Optional)

If the issue persists, clear the cache and rebuild the project:

```
npm cache clean --force
npm start
```

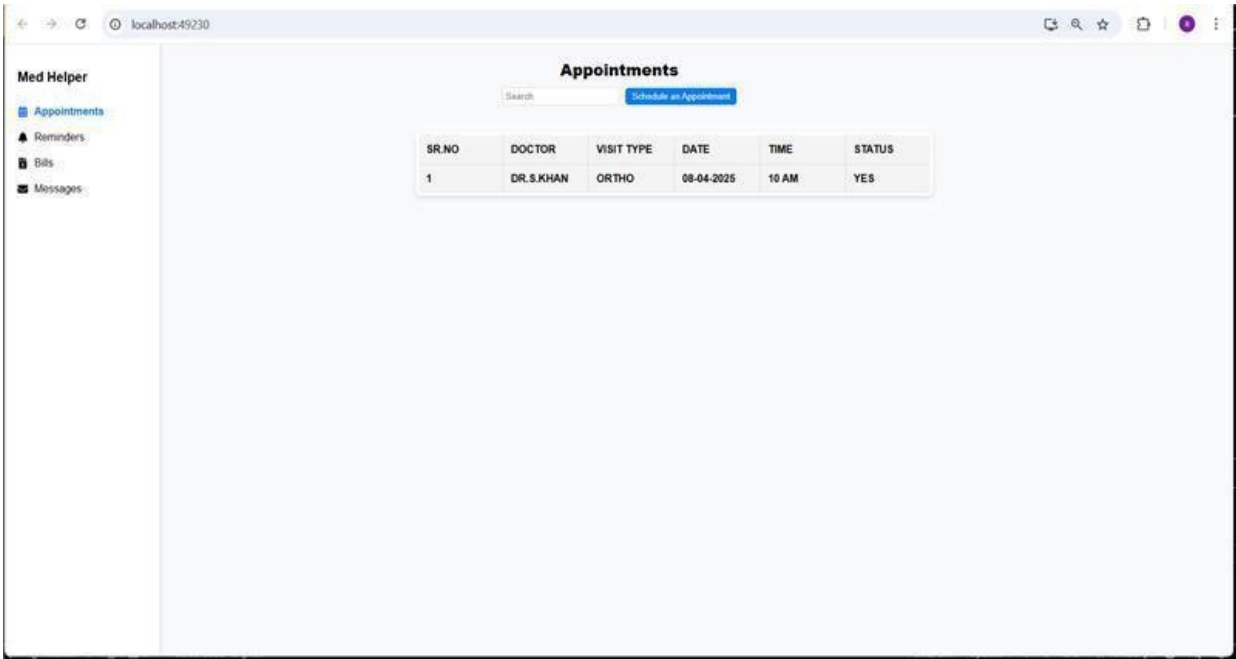
Summary

- Ensure `api.js` exists in the `src` directory.
- Verify the import paths in `AppointmentTable.js` and `Header.js`.
- Restart the development server after making changes.

Let me know if you need further assistance!

👍 🗨

Below is an image showing how the UI looks after implementing the code generated by GitHub Copilot:



Summary

This document outlines the development of a full-stack healthcare appointment scheduling system, utilizing Java Spring Boot for the backend and MongoDB as the database, with React for the frontend. The project successfully implemented CRUD operations, role-based access control, and API documentation using Swagger.

Key achievements include:

- Setting up the backend with Spring Boot and MongoDB.
- Developing the frontend with React components and state management.
- Integrating the frontend with the backend services.
- Resolving errors and refining the code based on specific requirements

Technologies used in this project include Java Spring Boot, MongoDB, React, and Swagger.

Despite facing challenges such as configuration issues and integration errors, the project was successfully completed with the help of GitHub Copilot.

Future work may involve further enhancements to the UI, additional security features, and scalability improvements to handle larger datasets.