IntelliCare Outpatient Management System

Introduction

This document provides detailed documentation for the IntelliCare – AI-Driven Outpatient Management System. This module allows patients to register, log in securely using OTP, and manage their profile including medical history and insurance details. The system is built using Angular for the frontend and .NET Core with Clean Architecture for the backend.

Concepts Implemented:

1. JWT
2. Custom Identity
3. Rate Limiting
4. Logging
5. Signal R

External Services Implemented:

6. Hangfire
7. Email Service(SMTP)
8. Twilio(whatsapp service)
9. OTP verification

Architecture Overview

The backend follows a Clean Architecture pattern with four layers:
1. IntelliCare.API – Contains controllers, middlewares, and configuration files like appsettings.json.
2. IntelliCare.Domain – Defines core entities and enums.
3. IntelliCare.Application – Contains DTOs, interfaces, and service logic.
4. IntelliCare.Infrastructure – Implements repositories, seeders, and database migrations.

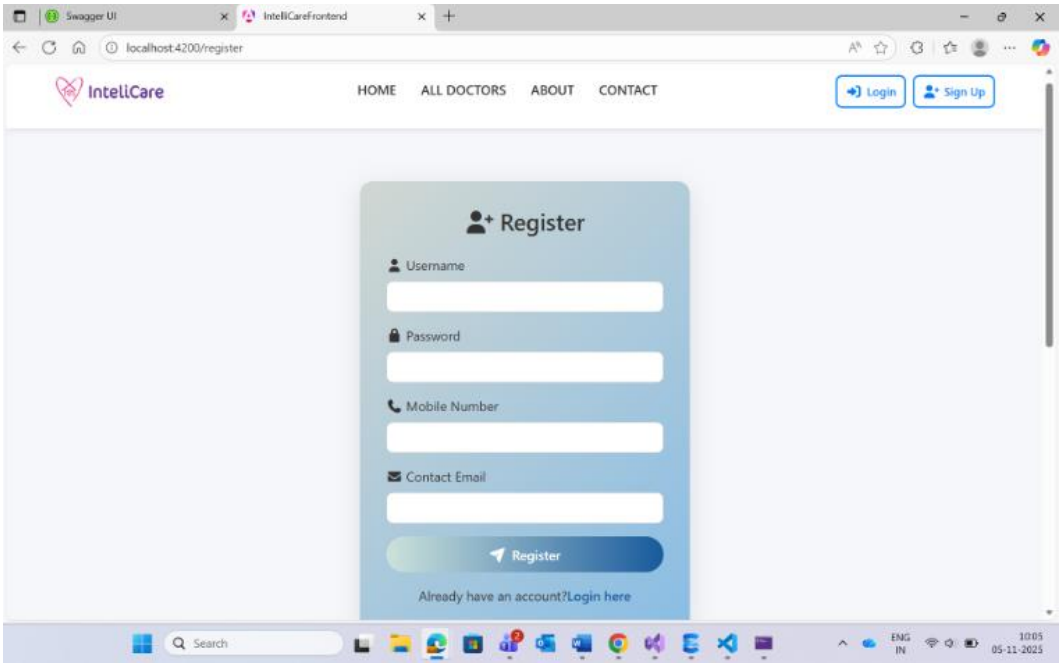
The frontend is developed using Angular and communicates with the backend via RESTful APIs.

1.Patient Registration & Profile Management Module

User Interface and Functionalities

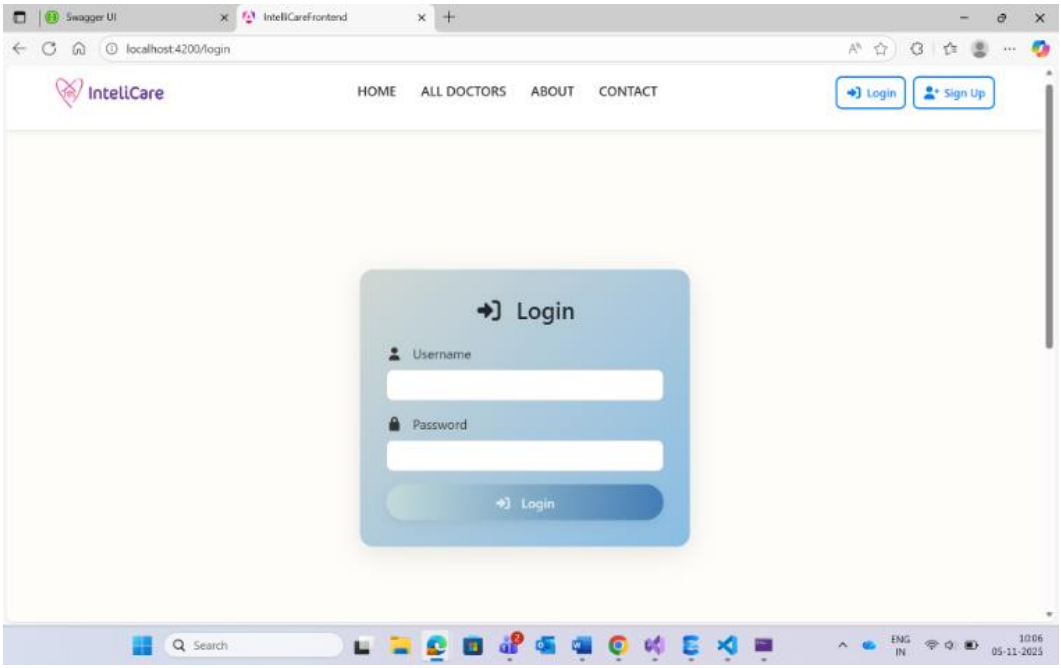
Registration Page

The registration page allows new patients to create an account by providing a username, password, mobile number, and contact email. The form includes validation rules for each field to ensure data integrity.



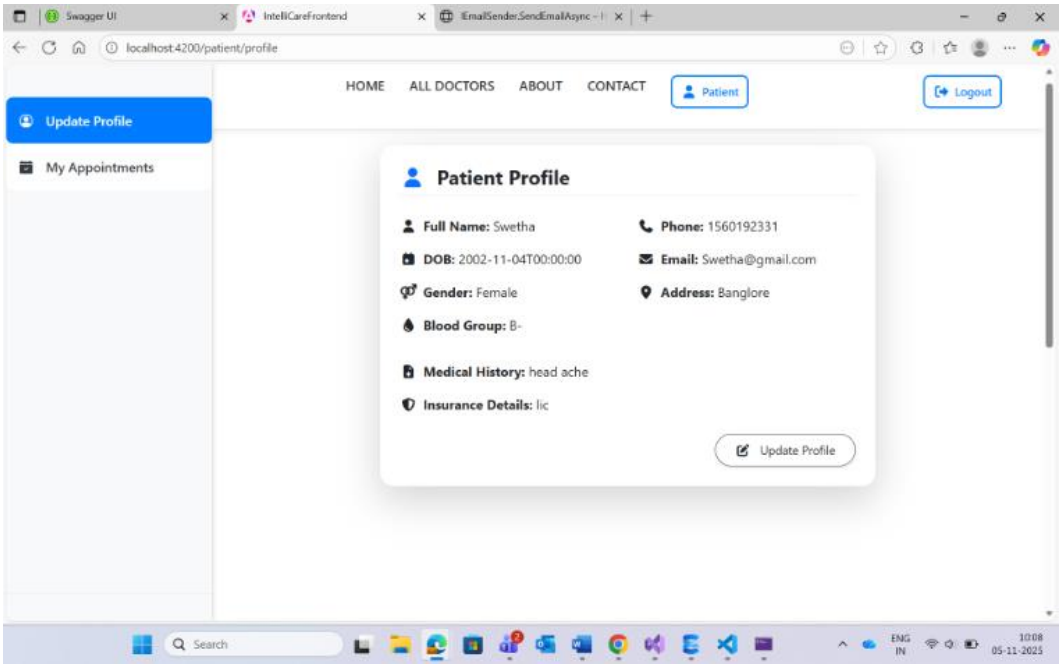
Login Page

The login page enables patients to log in using their registered username and password. It includes form validation and displays feedback for incorrect inputs.



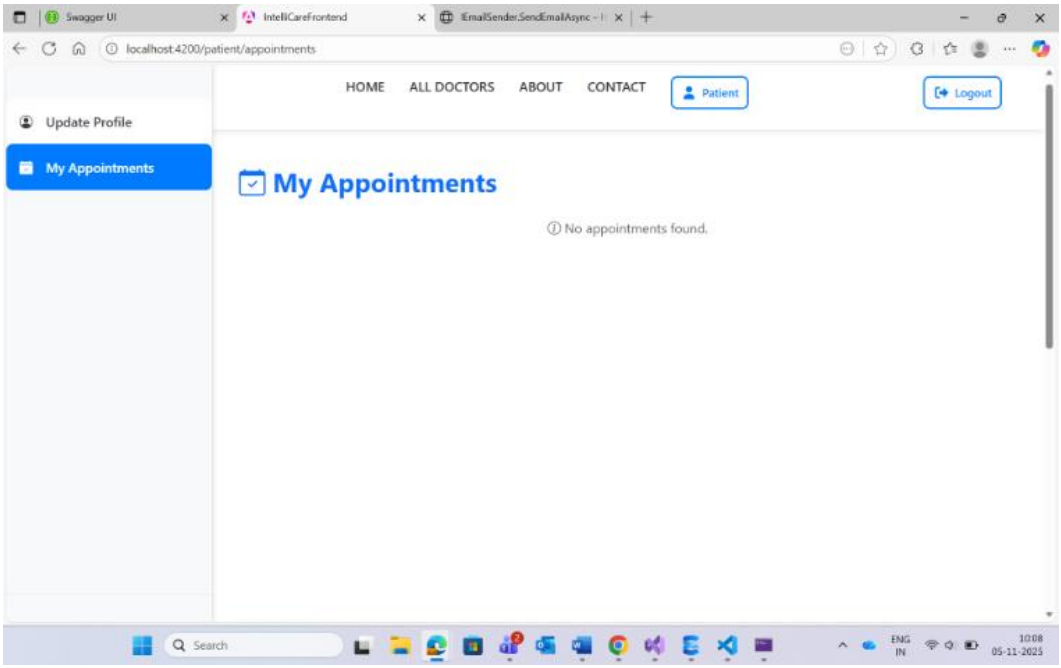
Patient Profile Page

After logging in, patients can view and update their profile information including personal details, medical history, and insurance information.



My Appointments Page

This page displays the list of appointments scheduled by the patient. It helps patients keep track of their upcoming consultations.



Frontend Implementation

The frontend is developed using Angular. It includes services and components for handling user registration, login, OTP verification, and profile management.

AuthService

This service handles user authentication tasks such as registering a new patient, logging in, verifying OTP, and managing authentication tokens in local storage.

```
// Register Patient
registerPatient(dto: RegisterUserDto): Observable<any> {
 return this.http.post(`${this.baseUrl}/register-patient`, dto).pipe(
  catchError(this.handleError)
 );
}

// Login
login(dto: LoginRequestDto): Observable<any> {
 return this.http.post(`${this.baseUrl}/login`, dto).pipe(
  catchError(this.handleError)
 );
}

// Verify OTP
verifyOtp(dto: OtpVerifyDto): Observable<any> {
 return this.http.post(`${this.baseUrl}/verify-otp`, dto).pipe(
  catchError(this.handleError)
 );
}
```

```
// Store token and user
storeAuthData(token: string, user: any): void {
 localStorage.setItem('token', token);
 localStorage.setItem('user', JSON.stringify(user));
}
```

PatientService

This service is responsible for fetching and updating patient profile data. It includes methods to get patient details by ID or username, update profile, and delete the account.

```
// Get patient by username with JWT
getPatientByUsername(username: string): Observable<any> {
 const token = localStorage.getItem('token');
 const headers = { Authorization: `Bearer ${token}` };
 return this.http.get(`${this.baseUrl}/by-username/${username}`, { headers });
}

// Complete profile
completeProfile(dto: CreatePatientDto): Observable<any> {
 return this.http.post(`${this.baseUrl}/complete-profile`, dto);
}
```

Backend Implementation

The backend is built using ASP.NET Core and follows Clean Architecture. It includes controllers, services, repositories, and domain models to handle business logic and data persistence.

UserController

Handles endpoints for patient registration, login, OTP verification, and admin/doctor creation. It validates input and delegates business logic to the UserService.

```
[HttpPost("register-patient")]
public async Task<IActionResult> RegisterPatient([FromBody] RegisterUserDto dto) {
   await _service.RegisterAsync(dto);
   return Ok(new { message = "Patient account created successfully." });
}

[HttpPost("login")]
public async Task<IActionResult> Login([FromBody] LoginRequestDto dto) {
   var user = await _service.LoginAsync(dto);
   return Ok(user);
```

```
}

[HttpPost("verify-otp")]
public async Task<IActionResult> VerifyOtp([FromBody] OtpVerifyDto dto) {
   var result = await _service.VerifyOtpAndGenerateTokenAsync(dto.Username,
dto.OTPCode);
   return Ok(result);
}
```

PatientController

Manages patient-specific operations such as viewing and updating profile, completing profile, and deleting account. It ensures role-based access control using authorization attributes.

```
[HttpPost("complete-profile")]
public async Task<IActionResult> CompleteProfile([FromBody] CreatePatientDto dto) {
   var result = await _userService.CompletePatientProfileAsync(dto);
   return Ok(new { message = "Patient profile completed successfully." });
}
```

UserService

Implements business logic for user registration, login, OTP verification, and profile completion. It interacts with repositories to persist and retrieve user data.

```
  public async Task RegisterAsync(RegisterUserDto dto) {
     if (await _repo.GetByUsernameAsync(dto.Username) != null)
        throw new InvalidOperationException("Username already exists.");

     var user = new User {
        Username = dto.Username,
        PasswordHash = Hash(dto.Password),
        RoleName = UserRole.Patient,
        MobileNumber = dto.MobileNumber,
        ContactEmail = dto.ContactEmail
     };

     await _repo.CreateAsync(user);
  }

  public string GenerateJwtToken(User user) {
     var claims = new List<Claim> {
        new Claim("username", user.Username),
        new Claim("role", user.RoleName.ToString())
     };
```

```
    var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:SecretKey"]));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken(claims: claims, expires:
DateTime.UtcNow.AddHours(3), signingCredentials: creds);
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Repositories

Repositories like UserRepository and PatientRepository handle database operations using Entity Framework Core. They provide methods to create, read, update, and delete user and patient records.

Data Flow

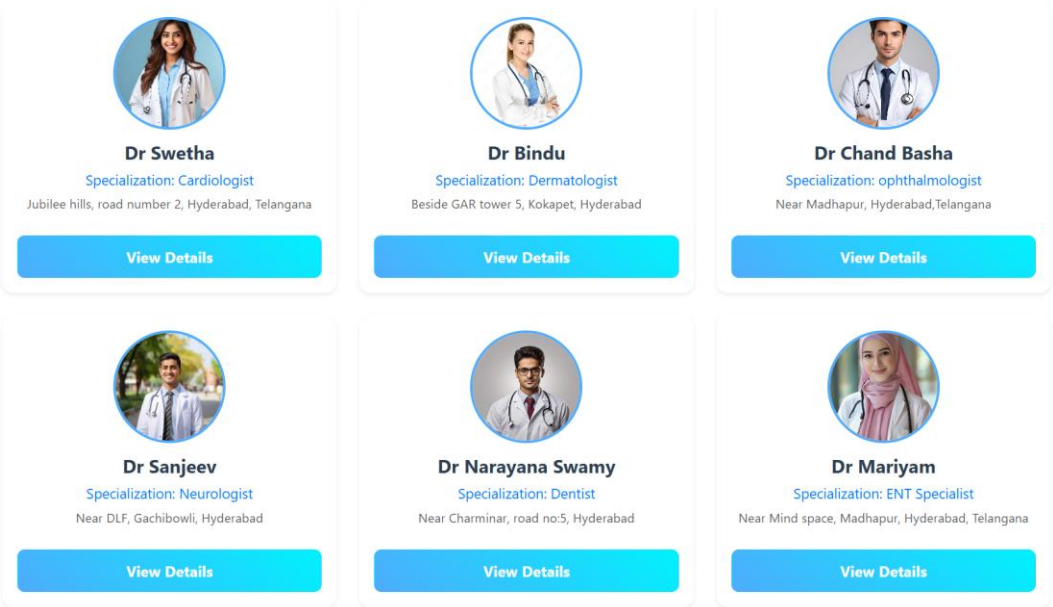
The following steps describe the data flow in the Patient Registration & Profile Management Module:

1. The user fills out the registration form in the Angular frontend.
2. The form data is sent to the backend API via AuthService.
3. The UserController receives the request and calls UserService to handle registration.
4. UserService validates the data and stores it using UserRepository.
5. Upon successful registration, the user can log in and complete their profile.
6. Profile data is submitted via PatientService and stored using PatientRepository.
7. JWT tokens are generated and stored in local storage for session management.

2. APPOINTMENT AND QUEUE MANAGEMENT MODULE

This module is designed to streamline healthcare operations by managing doctor schedules, patient appointments, cancel and reschedule the appointment and real-time queue tracking.

It ensures efficient resource utilization, reduces patient wait times, and improves overall service delivery.



When the patient login to the dashboard and navigates to all doctors the page looks like this and when the patient click view slots the below end point will call and show the available slots for the specific doctor.

Backend Code:

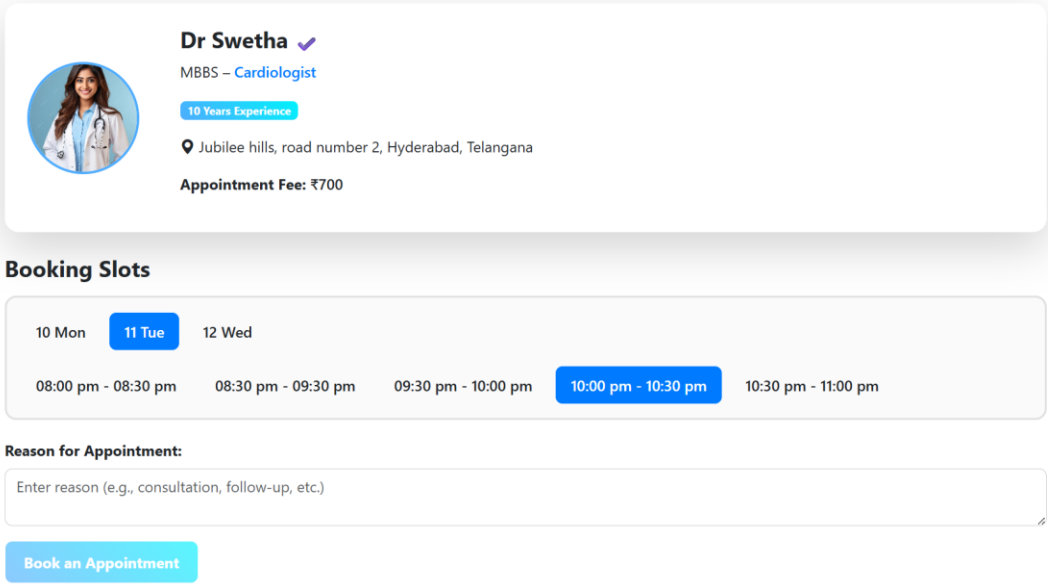
Doctor availability Endpoint:

This feature allows patients and administrators to check the availability of a specific doctor before booking an appointment. It ensures that patients can select suitable time slots based on the doctor's current schedule. This endpoint checks the doctor availability to choose the slot for the patient.

Code:

```
[Authorize(Roles="Patient,Admin")]
[HttpGet("DoctorAvailability")]
public async Task<IActionResult> GetAvailability([FromQuery]int
doctorId)
{
    try
    {
        var slots = await
_service.GetDoctorAvailabilityAsync(doctorId);
        return Ok(slots);
    }
    catch(Exception ex)
```

```
    {
        return StatusCode(500,new{message="Unable to fetch
availability. Please try again later."});
    }
}
```



When the patient select the available slot and give reason the Book an Appointment button will be enabled and the below endpoint will be called and appointment is booked successfully toast will be displayed.

BookAppointment Endpoint:

Access Control

- Role required: `Patient`
- Secured via `[Authorize]` attribute

Functionality

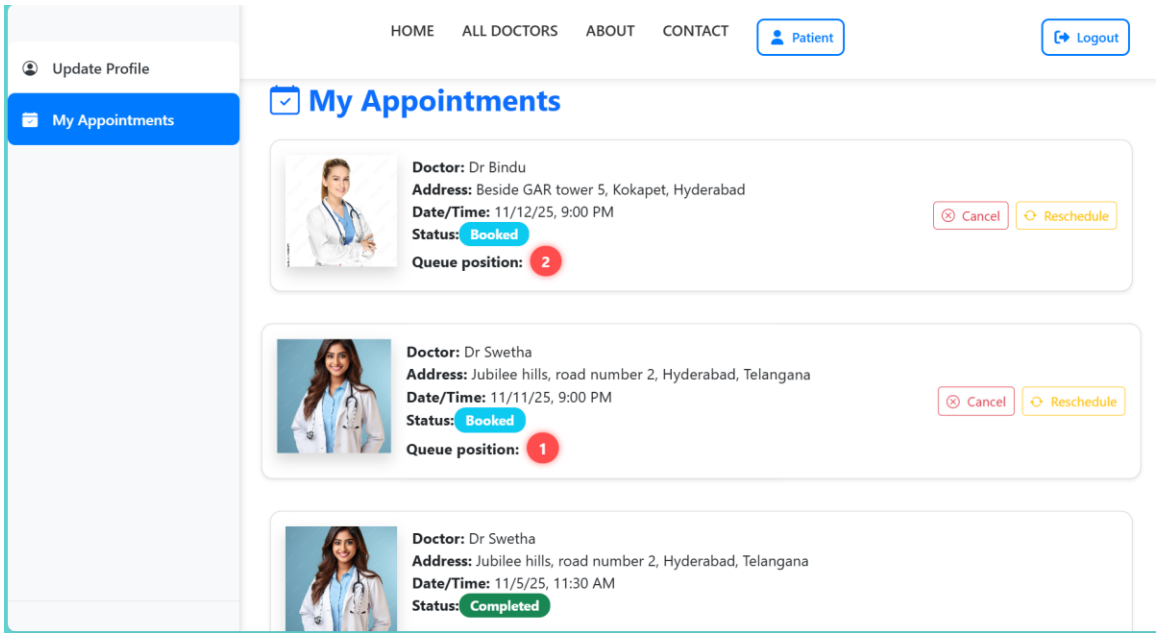
- Accepts appointment details via `BookAppointmentDto`.
- Validates input and checks doctor availability.
- Returns booking confirmation with:
    - Appointment ID
    - Fee details
    - Payment requirement status
    - Success message

```
[Authorize(Roles="Patient")]
[HttpPost("bookAppointment")]
```

```
public async Task<IActionResult> Book([FromBody]BookAppointmentDto dto)
{
    if(!ModelState.IsValid)
        return BadRequest(new{message="Invalid appointment data."});

    try
    {
        var result = await _service.BookAsync(dto);
        return Ok(new
        {
            message=result.Message,
            fee=result.Fee,
            paymentRequired=result.PaymentRequired,
            appointmentID=result.AppointmentID
        });
    }
    catch(ArgumentException ex)
    {
        return BadRequest(new{message="Invalid input: "+ex.Message});
    }
    catch(InvalidOperationException ex)
    {
        return Conflict(new{message=ex.Message});
    }
    catch(UnauthorizedAccessException ex)
    {
        return Unauthorized(new{message=ex.Message});
    }
    catch(Exception ex)
    {
        return StatusCode(500,new{message="Unexpected error occurred.
Please contact support."});
    }
}
```

After Booking an appointment the booked appointment will be displayed under the My appointments



Patients can cancel and reschedule their appointments by using the below endpoints

Cancellation :

Access Control

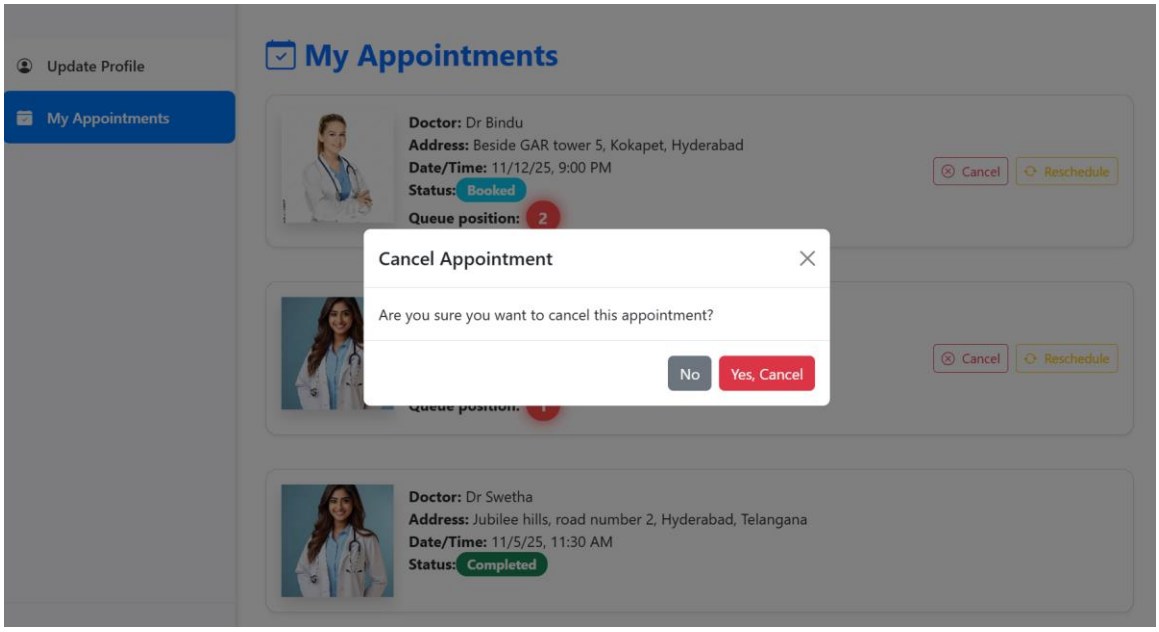
- Roles allowed: `Patient`, `Doctor`
- Secured via `[Authorize]` attribute

Functionality

- Accepts cancellation request via `CancelAppointmentDto`.
- Validates user permissions and appointment status.
- Returns confirmation message on successful cancellation.

```
[Authorize(Roles="Patient,Doctor")]
[HttpPost("cancelAppointment")]
public async Task<IActionResult> Cancel([FromBody]CancelAppointmentDto
dto)
{
    try
    {
        await _service.CancelAsync(dto);
        return Ok("Appointment cancelled");
    }
    catch(UnauthorizedAccessException ex)
    {
        return Unauthorized(new{message=ex.Message});
    }
    catch(InvalidOperationException ex)
    {
```

```
        return Conflict(new{message=ex.Message});
    }
    catch(Exception ex)
    {
        return StatusCode(500,new{message="Unexpected error occurred.
Please contact support."});
    }
}
```



Rescheduling Endpoint:

Access Control

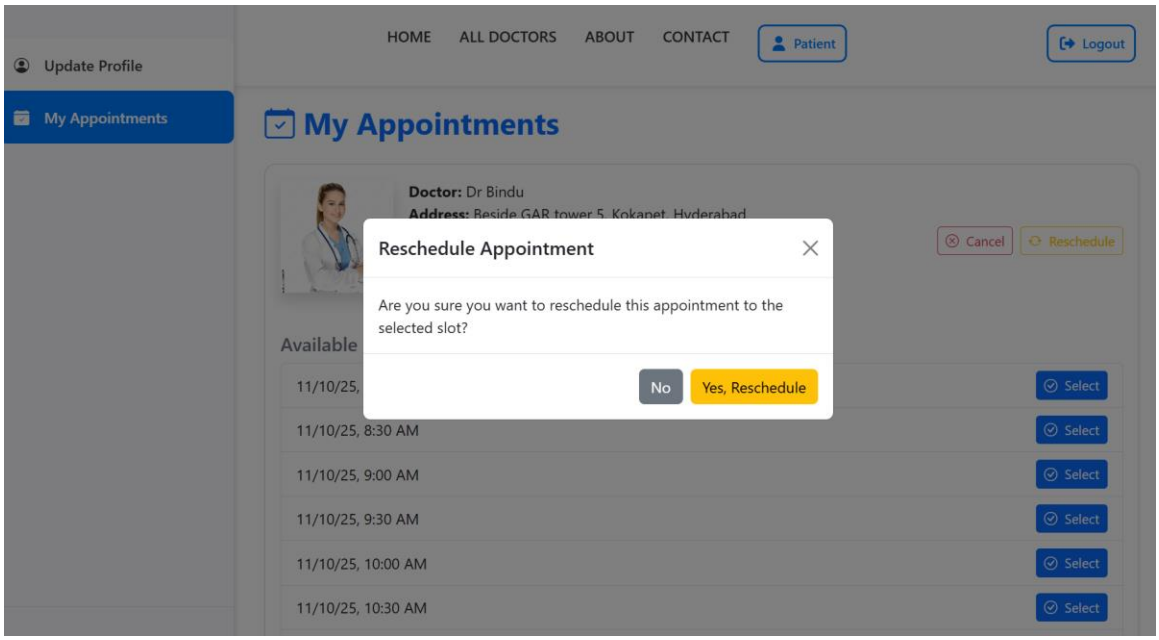
- Role required: `Patient`
- Secured via `[Authorize]` attribute

Functionality

- Accepts rescheduling request via `RescheduleAppointmentDto`.
- Validates input and checks slot availability.
- Updates appointment details and confirms the change.

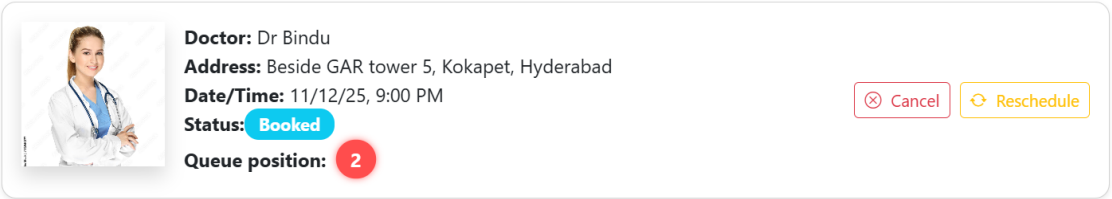
```
[Authorize(Roles="Patient")]
[HttpPost("rescheduleAppointment")]
public async Task<IActionResult>
Reschedule([FromBody]RescheduleAppointmentDto dto)
{
    if(!ModelState.IsValid)
        return BadRequest(new{message="Invalid reschedule request."});
    try
    {
        await _service.RescheduleAsync(dto);
```

```
        return Ok("Appointment rescheduled");
    }
    catch(ArgumentException ex)
    {
        return BadRequest(new{message="Invalid input: "+ex.Message});
    }
    catch(InvalidOperationException ex)
    {
        return Conflict(new{message=ex.Message});
    }
    catch(UnauthorizedAccessException ex)
    {
        return Unauthorized(new{message=ex.Message});
    }
    catch(Exception ex)
    {
        return StatusCode(500,new{message="Unexpected error occurred.
Please contact support."});
    }
}
```



Queue Position Tracking:

To track their queue position for the particular appointment we use the below endpoint



Queue tracking endpoint:

Allows patients to view the current appointment queue for a specific doctor on a selected date. This helps patients estimate wait times and plan their visit accordingly.

Endpoint

```
GET /api/Report/queue/{doctorId}?date=YYYY-MM-DD
```

Code:

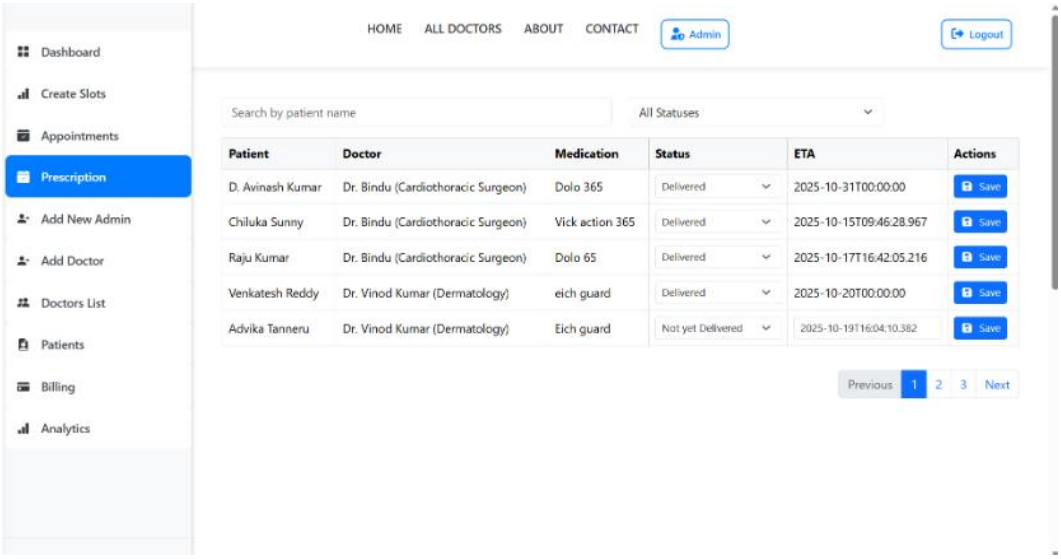
```
[Authorize(Roles="Patient")]
[HttpGet("queue/{doctorId}")]
public async Task<IActionResult> GetQueue(int
doctorId,[FromQuery]DateTime date)
{
    if(doctorId<=0)
        return BadRequest(new{message="Invalid doctor ID. Please
provide a valid doctor."});
    if(date.Date<DateTime.Today)
        return BadRequest(new{message="Date cannot be in the past.
Please select today or a future date."});
    try
    {
        var queue=await _service.GetQueueAsync(doctorId,date);
        if(queue==null||!queue.Any())
            return NotFound(new{message="No appointments found for the
selected doctor and date."});
        return Ok(queue);
    }
    catch(InvalidOperationException ex)
    {
        return BadRequest(new{message=ex.Message});
    }
    catch(Exception ex)
    {
        return StatusCode(500,new{message="Something went wrong while
fetching the queue. Please try again later."});
    }
}
```

3. Consultation & Prescription Module

User Interface and Functionalities

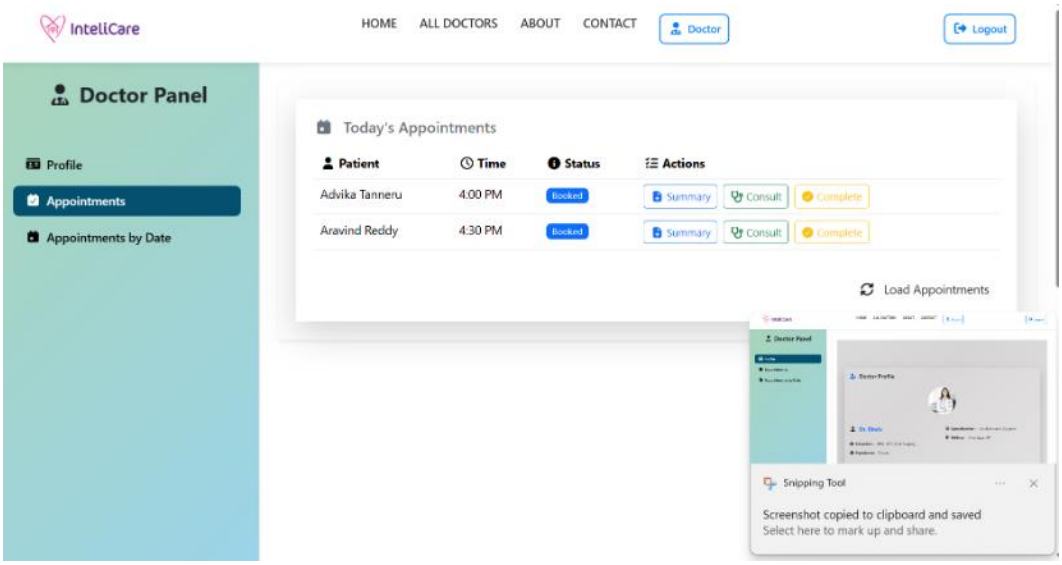
Record Consultation Page

This page allows doctors to record consultation details for a patient. The form includes fields for notes, diagnosis, and medication. Once submitted, an e-prescription is generated and displayed below the form.



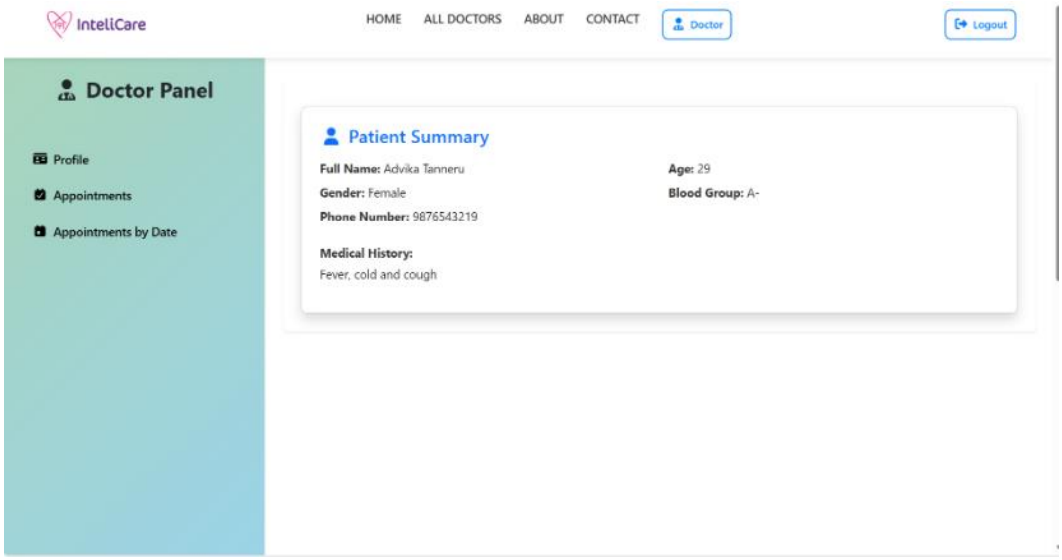
Patient Summary Page

This page displays a summary of the patient's details including name, age, gender, and medical history. It helps doctors quickly review patient information before recording a consultation.



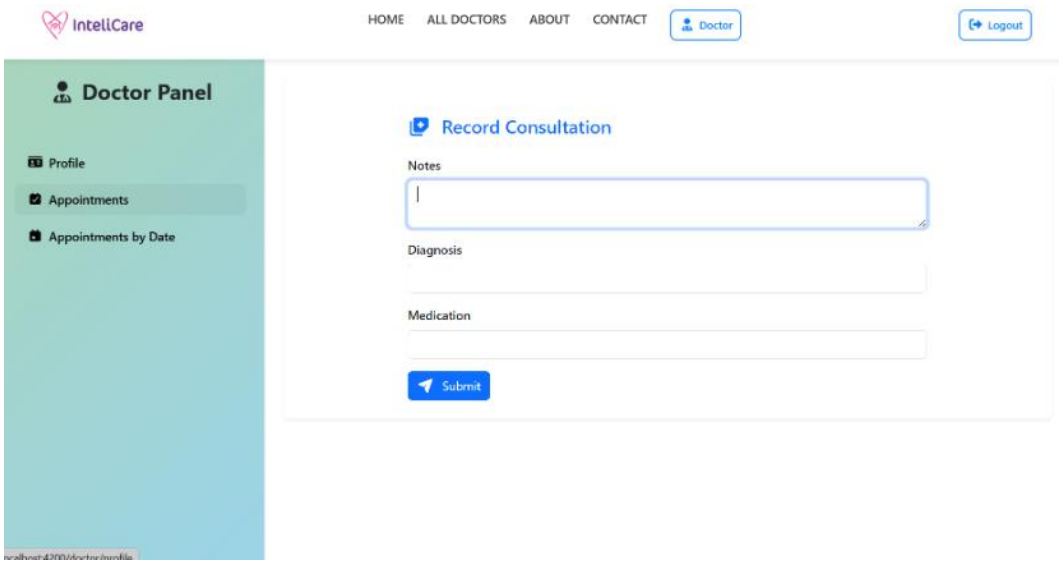
Appointments List Page

This page shows a list of appointments scheduled for the doctor. Each appointment includes patient name, time, and status. Doctors can click on an appointment to begin consultation.



Prescription Management Page

This page is used by the admin to manage prescriptions. It includes filters for patient name and pharmacy status. Admins can update the delivery status and ETA for each prescription.



Frontend Implementation

ConsultationService

This Angular service handles consultation-related operations. It includes methods to record a new consultation and fetch prescription details.

Main Methods:

- recordConsultation(dto): Sends consultation data to backend.
- getPrescription(appointmentId): Retrieves prescription for a given appointment.

PrescriptionService

This service manages prescription-related operations. It allows fetching all prescriptions and updating pharmacy status.

Main Methods:

- getPrescriptionByAppointmentId(id): Fetches prescription by appointment ID.
- updatePrescriptionStatus(id, dto): Updates pharmacy status and ETA.
- getAllPrescriptions(): Retrieves all prescriptions for admin view.

Backend Implementation

ConsultationController

This controller handles API endpoints for consultation and prescription management. It includes endpoints for recording consultations, fetching prescriptions, updating pharmacy status, and retrieving all prescriptions.

Main Endpoints:

- POST /api/Consultation: Records a new consultation and generates prescription.
- GET /api/Consultation/{appointmentId}: Fetches prescription for appointment.
- PATCH /api/Consultation/PrescriptionStatus/{id}: Updates pharmacy status.
- GET /api/Consultation/AllPrescriptions: Retrieves all prescriptions.

ConsultationService

This service implements business logic for consultation and prescription. It interacts with repositories to store and retrieve data.

Main Methods:

- RecordNewConsultationAsync(dto): Validates appointment and stores consultation record.
- GenerateEPrescriptionAsync(appointmentId): Builds prescription DTO from clinical record.
- UpdatePrescriptionStatusAsync(id, status, eta): Updates pharmacy status and delivery ETA.
- GetAllPrescriptionsAsync(): Aggregates all prescriptions with patient and doctor details.

Data Flow

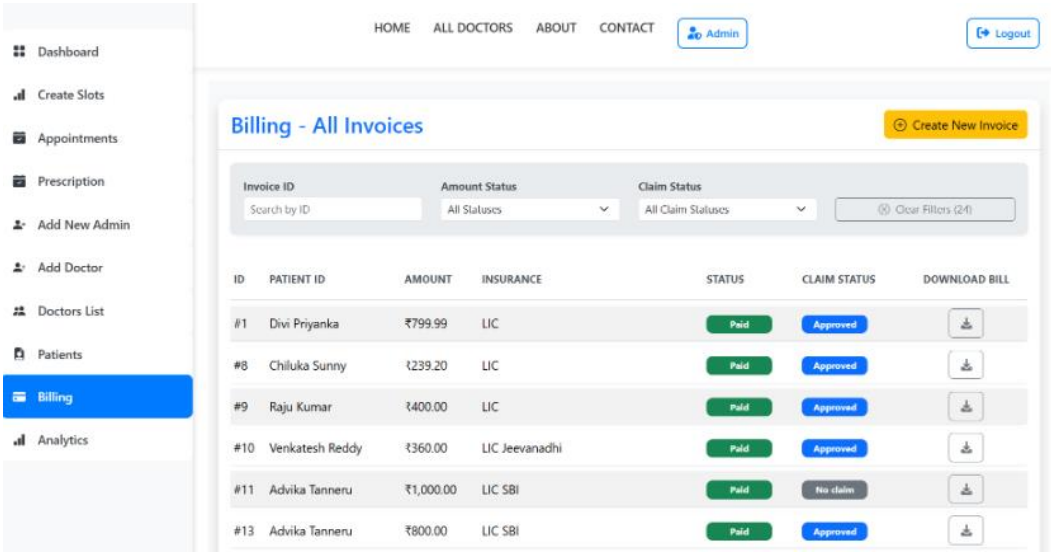
1. Doctor selects an appointment and fills out the consultation form.
2. Angular frontend sends the consultation data to backend via ConsultationService.
3. Backend validates and stores the consultation using ConsultationService and repositories.
4. A prescription is generated and returned to the frontend.
5. Admin views prescriptions and updates pharmacy status and ETA.
6. Patients can view their prescriptions and track delivery status.

4. Billing & Insurance Claims Module

User Interface and Functionalities

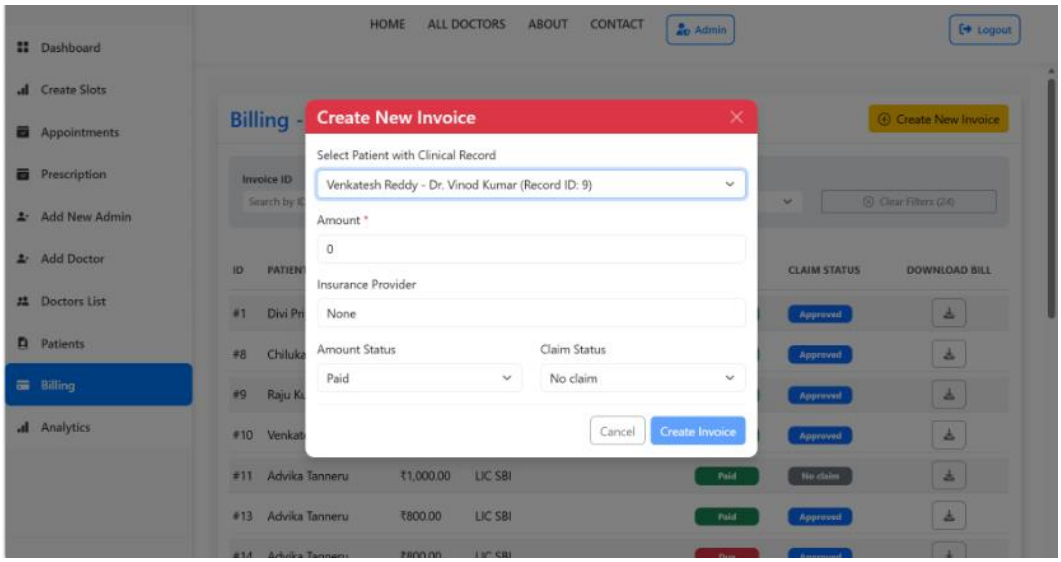
Billing Page – All Invoices

Displays all invoices in a tabular format with filters for Invoice ID, Amount Status, and Claim Status. Users can download invoices and create new ones.



Create Invoice Modal

Allows admins to create a new invoice by selecting a patient, entering amount, insurance provider, and setting payment and claim status.



Frontend Implementation

The frontend is developed using Angular. It includes services and components for handling invoice listing, filtering, downloading, and creation.

BillingService

This service handles all API calls related to invoices and patients.
Key methods:
- getAllInvoices(): Fetch all invoices.
- getInvoiceById(id): Fetch invoice by ID.
- getInvoicePdf(invoiceId): Download invoice PDF.
- getByStatus(status): Filter invoices by payment status.
- getByClaimStatus(claimStatus): Filter invoices by claim status.
- createInvoice(dto): Create a new invoice.
- getAllPatients(): Fetch patients for dropdown.

Backend Implementation

The backend is built using ASP.NET Core and follows Clean Architecture. It includes controllers, services, repositories, and domain models to handle business logic and data persistence.

InvoiceController

Handles endpoints for:
- Fetching all invoices.
- Fetching invoice by ID.
- Downloading invoice PDF.
- Filtering invoices by status and claim status.
- Creating new invoices.

Implements role-based access control for admin users.

Data Flow

1. Admin navigates to Billing page in Angular frontend.
2. BillingComponent calls BillingService.getAllInvoices() to load data.
3. API request hits InvoiceController in backend.
4. Controller delegates to service layer for business logic.
5. Data is fetched from database via repository and returned to frontend.
6. For creating invoice:
   - Modal form submits data via BillingService.createInvoice().
   - Backend validates and stores invoice in database.
7. Download invoice triggers getInvoicePdf() which returns a PDF blob.

5.Analytics & Operational Reporting Module Documentation

The **Analytics & Operational Reporting Module** provides hospital administrators with actionable insights into patient flow, doctor utilization, and revenue trends. It supports the generation, retrieval, and management of operational and predictive reports via secure API endpoints.

IntelliCare Analytics & Operational Reporting Module

Purpose

This module enables hospital administrators to generate and manage reports that provide insights into:

- **Revenue trends**
- **Doctor utilization**
- **Patient flow**
- **Predictive analytics**

Revenue Trends

Purpose:

To analyze the hospital's income over a specific period.

Key Metrics:

- Total revenue generated
- Number of billed appointments
- Average revenue per appointment
- Revenue per doctor or department

Use Case: Hospital administrators can track financial performance, identify high-performing doctors, and plan budgets or investments.

2. Doctor Utilization

Purpose:

To measure how effectively doctors are being utilized based on their appointment load.

Key Metrics:

- Number of appointments per doctor
- Average consultation time
- Idle time vs active time
- Utilization percentage

Use Case: Helps optimize doctor schedules, balance workloads, and improve patient care efficiency.

⚙ Service Logic: `ReportService`

1. GenerateReportAsync

- Validates input:
  - Report type
  - Doctor ID existence
  - Non-empty metrics and detailed data
- Delegates to specific report calculators:
  - `CalculateRevenueAsync`
  - `CalculateUtilizationAsync`
  - `CalculatePatientFlowAsync`
  - `CalculatePredictiveAnalysisAsync`
- Serializes metrics and detailed data to JSON
- Saves report to database via `SupportData` entity
- Returns `ReportSummaryDto` with generated `ReportID`

2. GetReportSummaryAsync

- Fetches report by ID
- Maps entity to DTO
- Deserializes JSON fields:
  - `MetricsJson` → `List<MetricDto>`
  - `DetailedDataJson` → `object`

3. GetAllReportSummariesAsync

- Retrieves all reports
- Maps each to `ReportSummaryDto`

4. GetReportDetailAsync

- Fetches full report detail by ID
- Returns `ReportDetailDto` with raw JSON metrics

5. DeleteReportAsync

- Deletes report by ID
- Returns success status