MỤC LỤC

Healthcare Service Requirements Specification

Họ tên: Phạm Huy Hòa

MSV: B21DCCN381

1. Requirements

This document outlines the detailed requirements for a comprehensive healthcare service system designed to streamline operations for hospitals, clinics, and associated stakeholders. The system aims to enhance patient care, improve operational efficiency, and ensure secure and seamless interactions among patients, healthcare providers, administrators, pharmacists, insurance providers, and laboratory technicians.

1.1. Actors

The system supports the following primary actors, each with specific roles and responsibilities:

1.1.1 Patient

- **Description**: Individuals seeking medical services.
- **Responsibilities**:
  - Register and maintain a personal account.
  - Book, modify, or cancel appointments with doctors.
  - Access and review personal medical records, including diagnoses, treatment plans, and test results.
  - View and manage prescriptions, including requesting refills.
  - Make payments for medical services and track billing history.
  - Receive notifications (email/SMS) for appointment reminders, test results, and prescription updates.

1.1.2 Doctor

- **Description**: Licensed medical professionals providing diagnoses and treatments.
- **Responsibilities**:
  - Manage personal availability and appointment schedules.
  - Access patient medical histories, including past diagnoses, treatments, and test results.
  - Diagnose conditions and prescribe medications or treatments.
  - Order laboratory tests and review results.
  - Document medical reports and update patient records.
  - Communicate with patients and other healthcare staff via secure messaging.

1.1.3 Nurse

- **Description**: Healthcare professionals assisting doctors and managing patient care.
- **Responsibilities**:

- Record and update patient vitals (e.g., blood pressure, temperature).
- Assist doctors during consultations and procedures.
- Administer medications or treatments as prescribed.
- Monitor patient progress and update medical records.
- Communicate care updates to doctors and patients.

1.1.4 *Administrator*

- **Description**: Staff responsible for system and operational management.
- **Responsibilities**:
  - Manage user accounts (patients, doctors, nurses, pharmacists, etc.) and assign roles/permissions.
  - Create and update schedules for doctors and nurses.
  - Oversee billing processes and coordinate with insurance providers.
  - Monitor system performance and ensure compliance with healthcare regulations (e.g., HIPAA).
  - Generate reports on system usage, financials, and operational metrics.

1.1.5 *Pharmacist*

- **Description**: Professionals managing medication dispensing and pharmacy operations.
- **Responsibilities**:
  - Verify prescriptions for accuracy and authenticity.
  - Dispense medications to patients and provide usage instructions.
  - Manage pharmacy inventory, including stock levels and reordering.
  - Process payments for medications and coordinate with insurance providers.
  - Update prescription statuses in the system.

1.1.6 *Insurance Provider*

- **Description**: Entities verifying insurance coverage and processing claims.
- **Responsibilities**:
  - Verify patient insurance eligibility and coverage details.
  - Process claims for medical services, tests, and medications.
  - Communicate claim statuses to patients and administrators.
  - Integrate with the billing system for seamless payment processing.

1.1.7 *Laboratory Technician*

- **Description**: Professionals conducting diagnostic tests and managing results.
- **Responsibilities**:
  - Perform medical tests (e.g., blood tests, imaging) as ordered by doctors.
  - Upload test results to the system and ensure accuracy.
  - Notify doctors and patients of test completion and results availability.
  - Maintain laboratory equipment and ensure compliance with safety standards.

1.2 Functions with respect to actors

This section details the system functions that each actor interacts with to perform their responsibilities. These functions are designed to support seamless workflows, secure data handling, and efficient communication across the healthcare service system. Each actor's functions are mapped to the system's core capabilities, including user authentication, electronic health records (EHR) management, appointment scheduling, billing, prescription management, laboratory operations, and notifications.

1.2.1 Patient Functions

Patients interact with the system to manage their healthcare needs, access information, and communicate with providers. The system provides an intuitive patient portal (web and mobile) to facilitate these interactions.

- **Account Management**:
  - **Register**: Create an account by providing personal details (e.g., name, date of birth, contact information, insurance details) and setting up secure credentials.
  - **Login/Logout**: Authenticate using username/password or multi-factor authentication (MFA); secure session management with automatic logout after inactivity.
  - **Profile Updates**: Modify personal information, update insurance details, or reset passwords via a secure interface.
- **Appointment Management**:
  - **Book Appointment**: Browse doctor availability by specialty, date, or time; select and confirm appointments with automated calendar integration.
  - **Modify/Cancel Appointment**: Edit or cancel existing appointments, with real-time updates to doctor schedules and confirmation notifications.
  - **Appointment History**: View past and upcoming appointments, including details like doctor name, date, and purpose.
- **Medical Records Access**:
  - **View Records**: Access a secure, searchable view of medical history, including diagnoses, treatment plans, test results, and doctor notes.
  - **Download/Share**: Export records as PDFs or share them securely with other healthcare providers.
  - **Request Corrections**: Submit requests to correct inaccuracies in medical records, with audit trails for changes.
- **Prescription Management**:
  - **View Prescriptions**: Access active and past prescriptions, including medication names, dosages, and instructions.
  - **Request Refills**: Submit refill requests, which are routed to doctors for

approval and pharmacies for fulfillment.
    - **Track Status**: Receive updates on prescription approvals, refills, or denials via notifications.
- **Billing and Payments**:
    - **View Bills**: Access itemized bills for services, tests, and medications, with details on insurance coverage and out-of-pocket costs.

    - **Make Payments**: Pay bills securely using credit cards, bank transfers, or insurance-linked accounts through integrated payment gateways.
    - **Payment History**: Track past payments, download receipts, and view outstanding balances.
- **Notifications and Communication**:
    - **Receive Alerts**: Get email/SMS notifications for appointment reminders, test result availability, prescription updates, and billing alerts.
    - **Secure Messaging**: Communicate with doctors or nurses for follow-up questions or clarifications, with encryption for privacy.

1.2.2 Doctor Functions

Doctors use the system to manage patient care, access clinical data, and coordinate with other healthcare staff. The system provides a professional dashboard tailored to clinical workflows.

- **Schedule Management**:
    - **Set Availability**: Define available time slots for consultations, with options for recurring or ad-hoc schedules.
    - **View Schedule**: Access daily/weekly appointment calendars, including patient details and visit purposes.
    - **Receive Updates**: Get notifications for new bookings, cancellations, or schedule conflicts.
- **Patient Record Access**:
    - **Retrieve History**: View comprehensive patient records, including past diagnoses, medications, allergies, and test results, with filters for quick navigation.
    - **Update Records**: Add consultation notes, diagnoses, or treatment plans, with version control and audit logs.
    - **Share Records**: Securely share relevant records with specialists or other providers within the system.
- **Diagnosis and Treatment**:
    - **Record Diagnoses**: Document patient conditions based on consultations, vitals, and test results, using standardized medical codes (e.g., ICD-10).
    - **Prescribe Medications**: Create electronic prescriptions, with checks for drug interactions, allergies, and dosage guidelines.
    - **Order Treatments**: Recommend therapies or procedures, with instructions for nurses or other staff.
- **Laboratory Test Management**:
    - **Order Tests**: Request diagnostic tests (e.g., blood work, imaging) with specific instructions and priority levels.
    - **Track Progress**: Monitor test statuses and receive notifications when results are uploaded.

    - **Review Results**: Analyze test reports, with tools to highlight abnormal findings or compare with historical data.
- **Reporting and Documentation**:
    - **Write Reports**: Create detailed medical reports for consultations, referrals, or insurance purposes.

    - **Export Reports**: Generate PDFs or share reports securely with patients or other providers.
    - **Sign Off**: Digitally sign records and prescriptions to ensure authenticity.
- **Communication**:
    - **Secure Messaging**: Exchange messages with patients, nurses, or pharmacists for care coordination.
    - **Notifications**: Receive alerts for test results, patient inquiries, or schedule changes.

1.2.3 *Nurse Functions*

Nurses interact with the system to support patient care and assist doctors, focusing on real-time updates and care coordination.

- **Vital Signs Management**:
    - **Record Vitals**: Enter patient vitals (e.g., blood pressure, heart rate, temperature) during visits, with validation for data accuracy.
    - **Update Vitals**: Modify or append vital records as patient conditions change.
    - **View Trends**: Access historical vital data to monitor patient progress.
- **Care Administration**:
    - **Administer Medications**: Follow prescription instructions to dispense medications, with documentation of administration times and dosages.
    - **Perform Procedures**: Assist with minor procedures (e.g., wound dressing, injections) and record details.
    - **Monitor Patients**: Track patient conditions during hospital stays or outpatient visits, updating records accordingly.
- **Record Updates**:
    - **Document Care**: Add notes on patient interactions, care provided, or observations for doctor review.
    - **Flag Issues**: Highlight urgent patient conditions or anomalies for doctor attention.
- **Communication**:
    - **Collaborate with Doctors**: Share updates on patient status or care needs via secure messaging.
    - **Notify Patients**: Inform patients of care instructions or follow-up requirements.
    - **Receive Alerts**: Get notifications for new prescriptions, test orders, or schedule changes.

1.2.4 *Administrator Functions*

Administrators oversee system operations, user management, and compliance, using a dedicated administrative interface.

- **User and Role Management**:

- **Create Accounts**: Set up accounts for patients, doctors, nurses, pharmacists, and other staff, assigning appropriate roles.
- **Manage Permissions**: Define access levels based on roles (e.g., read-only for nurses, full access for doctors).
- **Deactivate Accounts**: Suspend or remove accounts for inactive or unauthorized users, with audit trails.

- **Schedule Management**:
  - **Create Schedules**: Assign shifts for doctors and nurses, ensuring adequate coverage.
  - **Update Schedules**: Modify schedules in response to staff availability or emergencies.
  - **Notify Staff**: Send schedule updates via email/SMS to relevant personnel.
- **Billing and Insurance Oversight**:
  - **Monitor Billing**: Review generated bills for accuracy and completeness.
  - **Coordinate Claims**: Submit insurance claims and track statuses, resolving discrepancies with providers.
  - **Handle Refunds**: Process refunds or adjustments for billing errors, with documentation.
- **System Monitoring and Compliance**:
  - **Track Performance**: Monitor system uptime, response times, and error rates.
  - **Ensure Compliance**: Audit user activities and data access to comply with regulations (e.g., HIPAA, GDPR).
  - **Generate Reports**: Produce reports on system usage, financial performance, or staff productivity.
- **Communication**:
  - **Notify Users**: Send system-wide alerts for maintenance, updates, or policy changes.
  - **Respond to Inquiries**: Address user issues related to access, billing, or functionality.

1.2.5 Pharmacist Functions

Pharmacists manage prescription fulfillment and inventory, interacting with a pharmacy-specific module.

- **Prescription Management**:
  - **Verify Prescriptions**: Validate prescriptions for authenticity, accuracy, and patient eligibility.
  - **Dispense Medications**: Fulfill prescriptions, providing patients with medications and usage instructions.
  - **Update Status**: Mark prescriptions as dispensed, refilled, or denied, with notifications to patients and doctors.
- **Inventory Management**:
  - **Track Stock**: Monitor medication stock levels, with alerts for low inventory.
  - **Reorder Supplies**: Initiate purchase orders for restocking, with approval workflows.
  - **Manage Expirations**: Record and remove expired medications, updating inventory records.
- **Payment Processing**:
  - **Generate Bills**: Create invoices for dispensed medications, factoring in insurance coverage.

    - **Accept Payments**: Process payments via cash, card, or insurance claims, issuing receipts.
    - **Track Transactions**: Maintain records of all pharmacy transactions for auditing.
- **Communication**:
    - **Notify Patients**: Inform patients of prescription readiness or issues (e.g., stock shortages).
    - **Coordinate with Doctors**: Clarify prescription details or request modifications.
    - **Receive Alerts**: Get notifications for new prescriptions or refill requests.

1.2.6 *Insurance Provider Functions*

Insurance providers integrate with the system to verify coverage and process claims, primarily through automated APIs.

- **Eligibility Verification**:
  - **Check Coverage**: Validate patient insurance status and coverage details in real-time.
  - **Update Records**: Sync insurance data with patient profiles to ensure accuracy.
- **Claims Processing**:
  - **Receive Claims**: Accept electronic claims for services, tests, and medications from the billing module.
  - **Process Claims**: Evaluate claims based on coverage policies, approving or denying with justifications.
  - **Issue Payments**: Transfer approved claim amounts to the hospital or pharmacy, with transaction records.
- **Status Communication**:
  - **Notify Administrators**: Share claim statuses (e.g., approved, denied, pending) for billing reconciliation.
  - **Inform Patients**: Provide claim updates via the patient portal or notifications.
- **Integration**:
  - **API Connectivity**: Use secure APIs to integrate with the billing and patient management modules.
  - **Data Security**: Ensure encrypted data exchange to protect patient information.

1.2.7 *Laboratory Technician Functions*

Laboratory technicians manage diagnostic tests and results, using a lab-specific module integrated with the EHR system.

- **Test Management**:
  - **Receive Orders**: Access doctor-ordered tests with detailed instructions and priority levels.
  - **Perform Tests**: Conduct tests (e.g., blood analysis, imaging) following standard protocols, with quality control checks.
  - **Record Conditions**: Document test conditions (e.g., equipment used, sample details) for traceability.
- **Result Management**:
  - **Upload Results**: Enter test results into the system, including structured data and attachments (e.g., images, PDFs).
  - **Validate Results**: Review results for accuracy and flag abnormalities for doctor attention.
  - **Share Results**: Make results available to doctors and patients, with secure access controls.

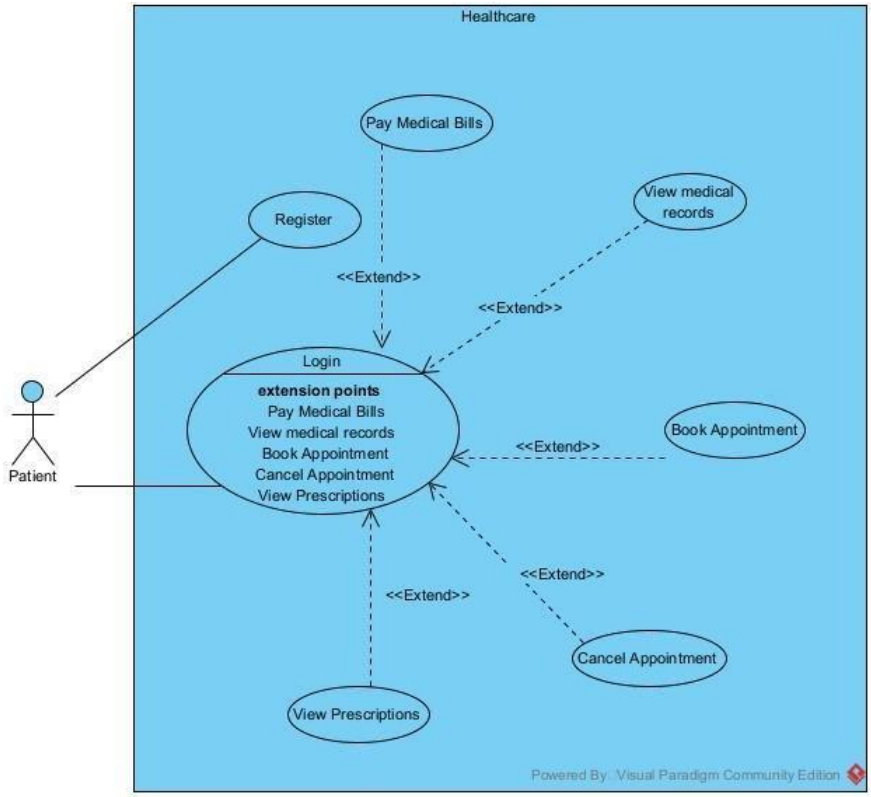
- **Equipment and Compliance**:
    - **Maintain Equipment**: Log equipment usage and maintenance schedules to ensure reliability.

    - **Ensure Compliance**: Follow safety and regulatory standards (e.g., CLIA) for lab operations.
    - **Audit Trails**: Record all test-related activities for compliance and quality assurance.
- **Communication**:
    - **Notify Doctors**: Alert doctors when results are uploaded, with priority notifications for urgent findings.
    - **Inform Patients**: Notify patients of result availability, with instructions for accessing reports.
    - **Receive Alerts**: Get notifications for new test orders or clarifications from doctors.

1.3. Use Cases

The system supports the following use cases, organized by functional area:

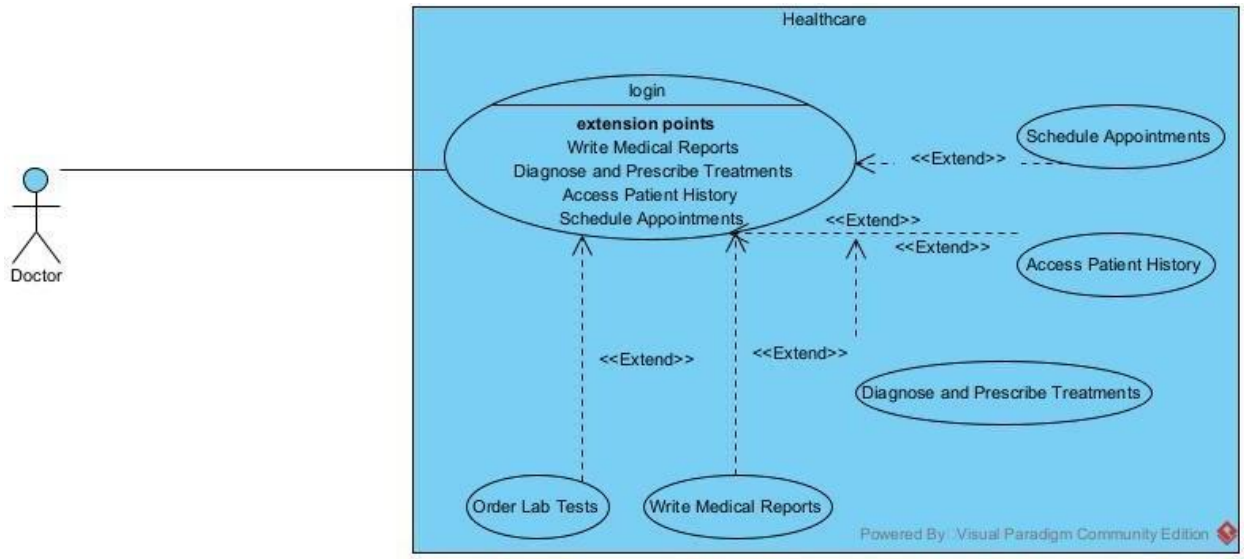
1.3.1 Patient Management



- **Register/Login**:
    - Patients create accounts with personal details (name, contact, insurance info).
    - Secure login using credentials or multi-factor authentication.
    - Password recovery and account verification processes.
- **View Medical Records**:

- Access a comprehensive view of medical history, including diagnoses, prescriptions, test results, and doctor notes.
- Download or print records for personal use or sharing with other providers.

- **Book/Cancel Appointments**:
  - Browse available doctor schedules and book appointments.
  - Receive confirmation and reminders via email/SMS.
  - Modify or cancel appointments with automated notifications to doctors.
- **View Prescriptions**:
  - Access active and past prescriptions with details (medication, dosage, duration).
  - Request prescription refills through the system.
  - Receive notifications for prescription approvals or renewals.
- **Pay Medical Bills**:
  - View itemized bills for services, tests, or medications.
  - Make secure online payments via credit card, bank transfer, or insurance-linked accounts.
  - Track payment history and download receipts.

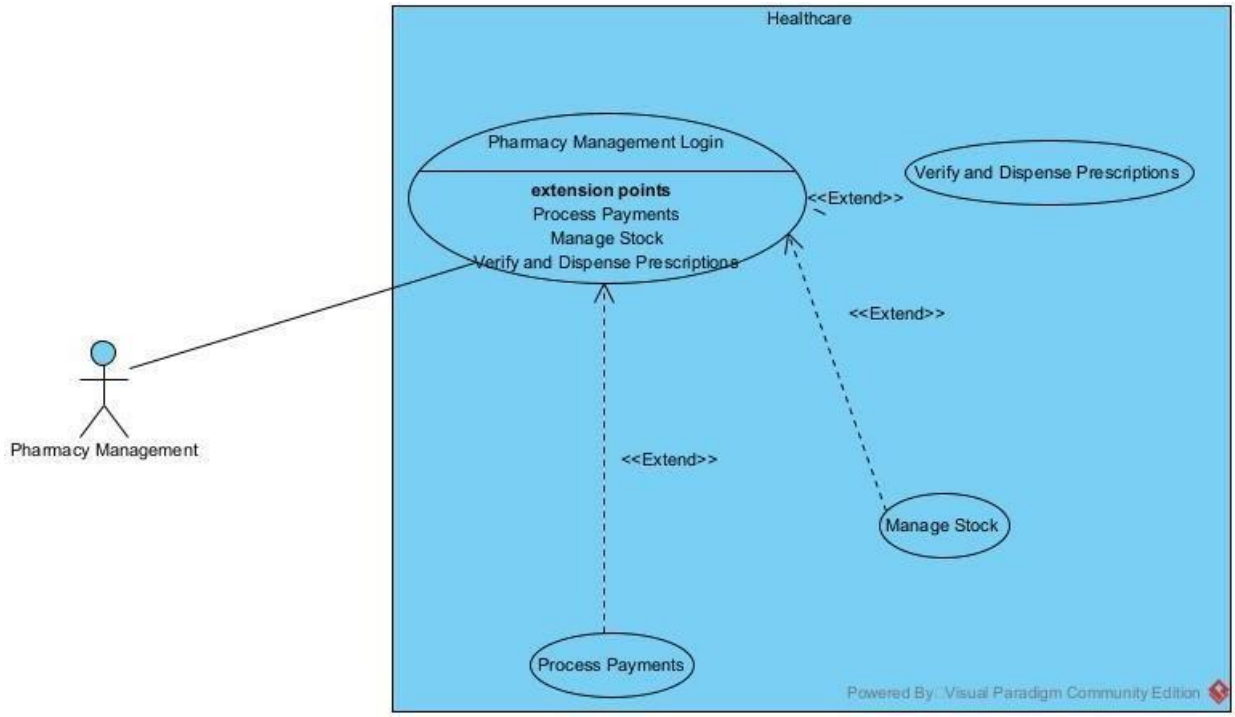


1.3.2 Doctor Functions

- **Schedule Appointments**:
  - Set availability and manage appointment slots.
  - Receive notifications for new, modified, or canceled appointments.
  - View daily/weekly schedules with patient details.
- **Access Patient History**:
  - Retrieve patient records, including past visits, diagnoses, medications, and test results.
  - Filter records by date, condition, or treatment type.
- **Diagnose and Prescribe Treatments**:
  - Record diagnoses based on consultations and test results.

- Prescribe medications or therapies, specifying dosage and instructions.
- Send prescriptions directly to the pharmacy module.

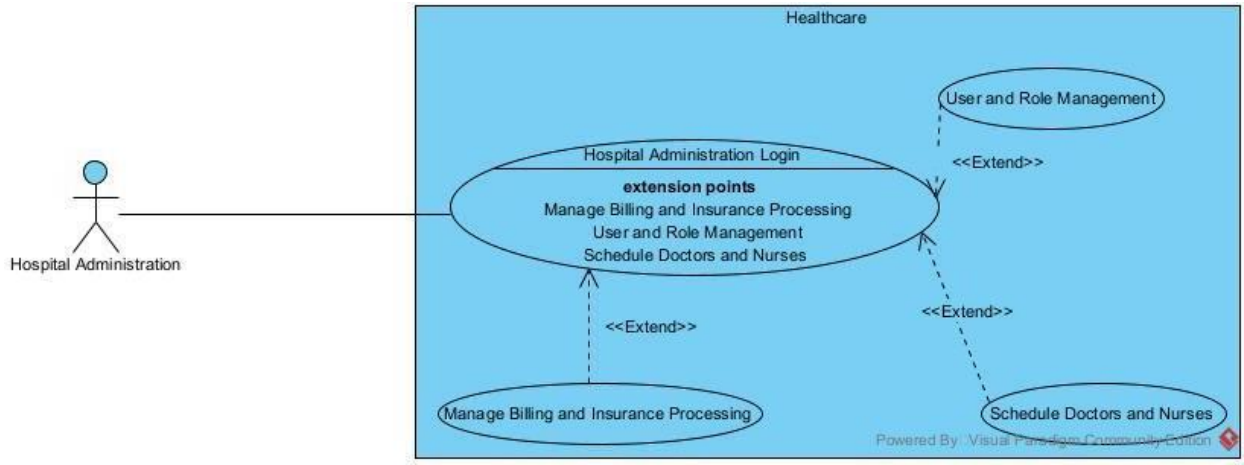
- **Order Lab Tests**:
  - Request diagnostic tests (e.g., blood work, X-rays) with specific instructions.
  - Track test progress and receive results notifications.
- **Write Medical Reports**:
  - Document consultation summaries, diagnoses, and treatment plans.
  - Share reports with patients or other healthcare providers securely.



1.3.3 Pharmacy Management

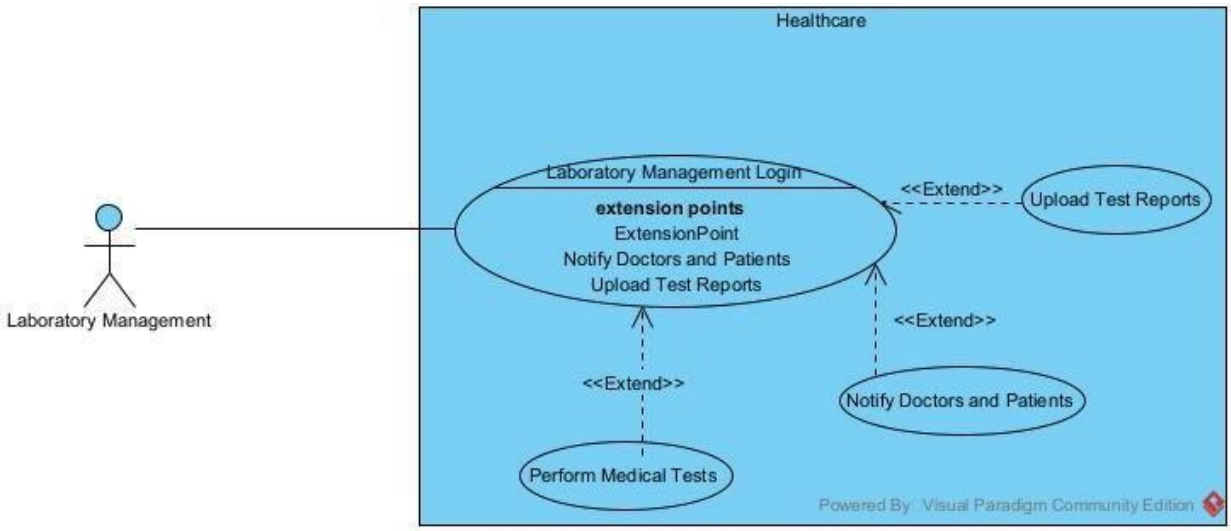
- **Verify and Dispense Prescriptions**:
  - Validate prescriptions against patient records and doctor authorization.
  - Dispense medications and update prescription statuses.
  - Provide patients with medication instructions and side-effect warnings.
- **Manage Stock**:
  - Track inventory levels for medications and supplies.
  - Set alerts for low stock and initiate reordering processes.
  - Record stock transactions (receipts, dispensing, expirations).
- **Process Payments**:
  - Generate bills for dispensed medications.
  - Accept payments via cash, card, or insurance claims.
  - Issue receipts and update payment records.

1.3.4 Hospital Administration



- **User and Role Management**:
  - Create, update, or deactivate user accounts for all actors.
  - Assign roles (e.g., doctor, nurse) with specific permissions.
  - Audit user activities for security and compliance.
- **Schedule Doctors and Nurses**:
  - Create and manage shift schedules for healthcare staff.
  - Ensure coverage for all required services.
  - Notify staff of schedule changes via email/SMS.
- **Manage Billing and Insurance Processing**:
  - Generate bills for patient services, tests, and medications.
  - Submit claims to insurance providers and track statuses.
  - Reconcile payments and handle disputes or refunds.

1.3.5 Laboratory Management


Healthcare
Laboratory Management Login
extension points
ExtensionPoint
Notify Doctors and Patients
Upload Test Reports
<<Extend>>
Upload Test Reports
<<Extend>>
Notify Doctors and Patients
<<Extend>>
Perform Medical Tests
Laboratory Management
Powered By Visual Paradigm Community Edition

- **Perform Medical Tests**:
  - Execute tests as per doctor orders, following standard protocols.
  - Record test conditions and ensure quality control.
- **Upload Test Reports**:
  - Enter test results into the system with detailed reports.
  - Attach supporting files (e.g., images, PDFs) as needed.
  - Validate results for accuracy before submission.
- **Notify Doctors and Patients**:
  - Alert doctors when test results are available.
  - Notify patients of result availability and provide access instructions.

1.4. System Functions

The system incorporates the following core functionalities to support the use cases:

1.4.1 User Authentication & Authorization

- Secure login with username/password, biometrics, or two-factor authentication.
- Role-based access control (RBAC) to restrict functionalities based on user roles.
- Session management with automatic logout for inactivity.
- Compliance with data protection regulations (e.g., HIPAA, GDPR).

1.4.2 Electronic Health Records (EHR) Management

- Centralized storage of patient records, including medical history, test results, and prescriptions.
- Secure access with encryption and audit trails for all record modifications.
- Interoperability with external systems (e.g., other hospitals, labs) using standards like HL7 or FHIR.
- Searchable and filterable records for quick retrieval.

1.4.3 Appointment Scheduling

- Real-time availability tracking for doctors and resources (e.g., consultation rooms).
- Automated conflict detection to prevent double-booking.
- Integration with notification systems for reminders and updates.
- Support for recurring appointments and group scheduling.

1.4.4 Billing & Insurance Processing

- Automated bill generation based on services, tests, and medications.
- Integration with insurance provider systems for eligibility checks and claim submissions.
- Support for multiple payment gateways and refund processing.
- Financial reporting for administrators, including revenue and outstanding payments.

1.4.5 Prescription & Pharmacy Management

- Electronic prescription creation, validation, and transmission to pharmacies.
- Drug interaction checks and allergy alerts during prescription creation.
- Inventory tracking with automated stock alerts and reorder suggestions.
- Prescription refill management with doctor approval workflows.

1.4.6 Medical Test & Report Management

- Test order creation with detailed instructions and priority levels.
- Result entry with support for structured data and file attachments.
- Automated result validation and flagging of abnormal findings.
- Secure sharing of results with doctors and patients.

1.4.7 Notification & Communication

- Email and SMS notifications for appointments, test results, prescriptions, and billing updates.
- Secure in-system messaging for communication between patients, doctors, and staff.
- Customizable notification templates and delivery schedules.
- Audit trails for all communications to ensure traceability.

2. Analyze Requirements

2.1. Decompose the system in microservices with Django

The healthcare service system is decomposed into independent microservices, each handling a specific domain from the requirements (e.g., User Authentication, EHR Management, Appointment Scheduling). Each microservice is a standalone Django project using Django REST Framework (DRF) for API development. The microservices communicate via RESTful APIs, use separate PostgreSQL databases, and are containerized with Docker for scalability. An API Gateway routes requests, and asynchronous tasks (e.g., notifications) use RabbitMQ for event- driven communication. Security is ensured with JWT authentication, HTTPS, and HIPAA/GDPR compliance.

Microservices:

1. **User Service**: Manages authentication, user accounts, and role-based access control (RBAC) for all actors.
2. **Patient Service**: Handles patient functions like appointment booking, medical record access, and billing.
3. **EHR Service**: Manages electronic health records, including diagnoses, vitals, and test results.
4. **Appointment Service**: Manages doctor schedules and patient appointments.

5. **Billing Service**: Processes bills, payments, and insurance claims.

6. **Prescription Service**: Manages electronic prescriptions and refill workflows.
7. **Pharmacy Service**: Handles prescription dispensing and inventory management.
8. **Laboratory Service**: Manages test orders and result uploads.
9. **Notification Service**: Sends email/SMS notifications and manages templates.
10. **Communication Service**: Handles secure in-system messaging.

Architecture:

- **Communication**: REST APIs for synchronous calls; RabbitMQ for asynchronous events.
- **Database**: Each service has its own PostgreSQL database.
- **Deployment**: Docker containers orchestrated with Kubernetes.
- **Security**: JWT authentication, encrypted data, role-based access control.

2.2 Classes with Attributes of Service Models (Models)

Each microservice has Django models representing its domain entities. Below are the key model classes with attributes for each service, ensuring alignment with the requirements (e.g., User Authentication, EHR, Billing).

```
user_service/users/models.py
from django.db import models
from django.contrib.auth.models import AbstractUser
from uuid import uuid4

class User(AbstractUser):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    role = models.CharField(max_length=50, choices=[
        ('patient', 'Patient'), ('doctor', 'Doctor'), ('nurse', 'Nurse'),
        ('admin', 'Administrator'), ('pharmacist', 'Pharmacist'), ('lab_tech',
'Laboratory Technician')
    ])
    phone = models.CharField(max_length=15, blank=True)
    insurance_id = models.CharField(max_length=50, blank=True)
    is_active = models.BooleanField(default=True)

class Permission(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    name = models.CharField(max_length=100, unique=True)
    description = models.TextField(blank=True)

class RolePermission(models.Model):
    role = models.CharField(max_length=50)
    permission = models.ForeignKey(Permission, on_delete=models.CASCADE)
```

```
patient_service/patients/models.py
class PatientProfile(models.Model):
```

```
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    date_of_birth = models.DateField()
    address = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

class PatientAppointment(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey(PatientProfile, on_delete=models.CASCADE)
    appointment_id = models.UUIDField()  References Appointment Service
    status = models.CharField(max_length=20, choices=[('booked', 'Booked'),
('cancelled', 'Cancelled')])

class PatientBill(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey(PatientProfile, on_delete=models.CASCADE)
    bill_id = models.UUIDField()  References Billing Service
    status = models.CharField(max_length=20, choices=[('pending', 'Pending'),
('paid', 'Paid')])

ehr_service/ehr/models.py
class MedicalRecord(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey('patients.PatientProfile',
on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class Diagnosis(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    record = models.ForeignKey(MedicalRecord, on_delete=models.CASCADE)
    icd_code = models.CharField(max_length=10)
    description = models.TextField()
    diagnosed_by = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    diagnosed_at = models.DateTimeField(auto_now_add=True)

class Vital(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    record = models.ForeignKey(MedicalRecord, on_delete=models.CASCADE)
    blood_pressure = models.CharField(max_length=10)
    heart_rate = models.IntegerField()
    temperature = models.FloatField()
    recorded_at = models.DateTimeField(auto_now_add=True)

appointment_service/appointments/models.py
```

```
class DoctorSchedule(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    doctor = models.ForeignKey(User, on_delete=models.CASCADE)
    start_time = models.DateTimeField()
    end_time = models.DateTimeField()
    is_available = models.BooleanField(default=True)

class Appointment(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey('patients.PatientProfile',
on_delete=models.CASCADE)
    doctor = models.ForeignKey(User, on_delete=models.CASCADE)
    schedule = models.ForeignKey(DoctorSchedule, on_delete=models.CASCADE)
    status = models.CharField(max_length=20, choices=[('booked', 'Booked'),
('cancelled', 'Cancelled')])
    created_at = models.DateTimeField(auto_now_add=True)

billing_service/billing/models.py
class Bill(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey('patients.PatientProfile',
on_delete=models.CASCADE)
    amount = models.DecimalField(max_digits=10, decimal_places=2)
    description = models.TextField()
    status = models.CharField(max_length=20, choices=[('pending', 'Pending'),
('paid', 'Paid')])
    created_at = models.DateTimeField(auto_now_add=True)

class InsuranceClaim(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    bill = models.ForeignKey(Bill, on_delete=models.CASCADE)
    insurance_id = models.CharField(max_length=50)
    status = models.CharField(max_length=20, choices=[('pending', 'Pending'),
('approved', 'Approved'), ('denied', 'Denied')])
    submitted_at = models.DateTimeField(auto_now_add=True)

prescription_service/prescriptions/models.py
class Prescription(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey('patients.PatientProfile',
on_delete=models.CASCADE)
    doctor = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    medication = models.CharField(max_length=100)
    dosage = models.CharField(max_length=50)
```

```
    status = models.CharField(max_length=20, choices=[('active', 'Active'),
('dispensed', 'Dispensed'), ('expired', 'Expired')])
    created_at = models.DateTimeField(auto_now_add=True)

class RefillRequest(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    prescription = models.ForeignKey(Prescription, on_delete=models.CASCADE)
    status = models.CharField(max_length=20, choices=[('pending', 'Pending'),
('approved', 'Approved'), ('denied', 'Denied')])
    requested_at = models.DateTimeField(auto_now_add=True)
```

pharmacy_service/pharmacy/models.py

```
class Inventory(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    medication = models.CharField(max_length=100)
    quantity = models.IntegerField()
    expiry_date = models.DateField()
    last_updated = models.DateTimeField(auto_now=True)

class DispenseRecord(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    prescription = models.ForeignKey('prescriptions.Prescription',
on_delete=models.CASCADE)
    quantity = models.IntegerField()
    dispensed_at = models.DateTimeField(auto_now_add=True)
```

laboratory_service/laboratory/models.py

```
class TestOrder(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    patient = models.ForeignKey('patients.PatientProfile',
on_delete=models.CASCADE)
    doctor = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    test_type = models.CharField(max_length=100)
    priority = models.CharField(max_length=20, choices=[('normal', 'Normal'),
('urgent', 'Urgent')])
    ordered_at = models.DateTimeField(auto_now_add=True)

class TestResult(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    test_order = models.ForeignKey(TestOrder, on_delete=models.CASCADE)
    result_data = models.TextField()
    attachment = models.FileField(upload_to='test_results/', blank=True)
    uploaded_at = models.DateTimeField(auto_now_add=True)
```

notification_service/notifications/models.py

```
class Notification(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.TextField()
    type = models.CharField(max_length=20, choices=[('email', 'Email'), ('sms',
'SMS'), ('in_app', 'In-App')])
    status = models.CharField(max_length=20, choices=[('sent', 'Sent'),
('pending', 'Pending'), ('failed', 'Failed')])
    sent_at = models.DateTimeField(auto_now_add=True)

class Template(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    name = models.CharField(max_length=100)
    content = models.TextField()
    type = models.CharField(max_length=20, choices=[('email', 'Email'), ('sms',
'SMS')])

communication_service/communications/models.py
class Message(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    sender = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='sent_messages')
    recipient = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='received_messages')
    content = models.TextField()
    is_read = models.BooleanField(default=False)
    sent_at = models.DateTimeField(auto_now_add=True)

class Thread(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
    participants = models.ManyToManyField(User)
    last_message_at = models.DateTimeField(auto_now=True)
```

2.3 Determine Functions in Services (Views)

```python
user_service/users/views.py
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from rest_framework.permissions import IsAuthenticated, IsAdminUser
from .models import User, Permission
from .serializers import UserSerializer, PermissionSerializer
from django.contrib.auth.hashers import make_password
```

```
import jwt
from datetime import datetime, timedelta

class RegisterView(APIView):
    def post(self, request):
        serializer = UserSerializer(data=request.data)
        if serializer.is_valid():
            serializer.validated_data['password'] =
make_password(serializer.validated_data['password'])
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class LoginView(APIView):
    def post(self, request):
        username = request.data.get('username')
        password = request.data.get('password')
        user = User.objects.filter(username=username).first()
        if user and user.check_password(password):
            token = jwt.encode({
                'user_id': str(user.id), 'role': user.role, 'exp':
datetime.utcnow() + timedelta(hours=24)
            }, 'secret_key', algorithm='HS256')
            return Response({'token': token}, status=status.HTTP_200_OK)
        return Response({'error': 'Invalid credentials'},
status=status.HTTP_401_UNAUTHORIZED)

class UserProfileView(APIView):
    permission_classes = [IsAuthenticated]
    def get(self, request):
        serializer = UserSerializer(request.user)
        return Response(serializer.data)
    def put(self, request):
        serializer = UserSerializer(request.user, data=request.data,
partial=True)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

patient_service/patients/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import PatientProfile, PatientAppointment, PatientBill
```

```
from .serializers import PatientProfileSerializer, PatientAppointmentSerializer,
PatientBillSerializer

class AppointmentBookingView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = PatientAppointmentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save(patient=request.user.patientprofile)
            Call Appointment Service API to confirm booking
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class MedicalRecordsView(APIView):
    permission_classes = [IsAuthenticated]
    def get(self, request):
        Call EHR Service API to retrieve records
        return Response({'records': []}, status=status.HTTP_200_OK)

ehr_service/ehr/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import MedicalRecord, Diagnosis, Vital
from .serializers import MedicalRecordSerializer, DiagnosisSerializer,
VitalSerializer

class DiagnosisView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = DiagnosisSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save(diagnosed_by=request.user)
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class VitalUpdateView(APIView):
    permission_classes = [IsAuthenticated]
    def put(self, request, vital_id):
        vital = Vital.objects.get(id=vital_id)
        serializer = VitalSerializer(vital, data=request.data, partial=True)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

```
appointment_service/appointments/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import Appointment, DoctorSchedule
from .serializers import AppointmentSerializer, DoctorScheduleSerializer

class AppointmentCreateView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = AppointmentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            Publish event to Notification Service
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

billing_service/billing/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import Bill, InsuranceClaim
from .serializers import BillSerializer, InsuranceClaimSerializer

class BillCreateView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = BillSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

prescription_service/prescriptions/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import Prescription
from .serializers import PrescriptionSerializer

class PrescriptionCreateView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = PrescriptionSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save(doctor=request.user)
            Notify Pharmacy Service
            return Response(serializer.data, status=status.HTTP_201_CREATED)
```

```
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

pharmacy_service/pharmacy/views.py

```
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import Inventory, DispenseRecord
from .serializers import InventorySerializer, DispenseRecordSerializer

class DispenseView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = DispenseRecordSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            Update Prescription Service status
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

laboratory_service/laboratory/views.py

```
from restolean
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import TestOrder, TestResult
from .serializers import TestOrderSerializer, TestResultSerializer

class TestResultUploadView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = TestResultSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            Notify EHR and Notification Services
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

notification_service/notifications/views.py

```
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import Notification
from .serializers import NotificationSerializer

class SendNotificationView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = NotificationSerializer(data=request.data)
```

```
        if serializer.is_valid():
            serializer.save()
            Trigger external SMS/Email service
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

communication_service/communications/views.py
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from .models import Message
from .serializers import MessageSerializer

class SendMessageView(APIView):
    permission_classes = [IsAuthenticated]
    def post(self, request):
        serializer = MessageSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save(sender=request.user)
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

2.5 Determine REST API Connecting Services

The microservices communicate via RESTful APIs using JSON over HTTPS, with JWT authentication. Below are key APIs for each service, focusing on inter-service interactions to support the requirements (e.g., appointment booking, prescription dispensing). APIs are designed to be stateless, idempotent where applicable, and follow REST conventions.

User Service APIs

POST /api/users/register

- Description: Register a new user (patient, doctor, etc.).

- Request: { "username": "string", "password": "string", "role": "string", "phone": "string", "insurance_id": "string" }

- Response: { "id": "uuid", "username": "string", "role": "string" }

POST /api/users/login

- Description: Authenticate user and issue JWT.

- Request: { "username": "string", "password": "string" }

- Response: { "token": "string" }

GET /api/users/profile

- Description: Retrieve authenticated user's profile.

- Response: { "id": "uuid", "username": "string", "role": "string", "phone": "string" }

Patient Service APIs

POST /api/patients/appointments

- Description: Book an appointment by calling Appointment Service.

- Request: { "doctor_id": "uuid", "schedule_id": "uuid", "time": "datetime" }

- Response: { "id": "uuid", "status": "booked" }

GET /api/patients/records

- Description: Retrieve medical records by calling EHR Service.

- Response: [{ "id": "uuid", "diagnosis": "string", "date": "datetime" }]

POST /api/patients/payments

- Description: Process payment by calling Billing Service.

- Request: { "bill_id": "uuid", "amount": "decimal", "method": "string" }

- Response: { "id": "uuid", "status": "paid" }

EHR Service APIs

POST /api/ehr/diagnoses

- Description: Record a diagnosis.

- Request: { "patient_id": "uuid", "icd_code": "string", "description": "string" }

- Response: { "id": "uuid", "icd_code": "string" }

PUT /api/ehr/vitals/{vital_id}

- Description: Update patient vitals.

- Request: { "blood_pressure": "string", "heart_rate": "integer", "temperature": "float" }

- Response: { "id": "uuid", "blood_pressure": "string" }

POST /api/ehr/results

- Description: Upload test results from Laboratory Service.

- Request: { "test_order_id": "uuid", "result_data": "string", "attachment": "file" }

- Response: { "id": "uuid", "result_data": "string" }

Appointment Service APIs

POST /api/appointments

- Description: Create an appointment.

- Request: { "patient_id": "uuid", "doctor_id": "uuid", "schedule_id": "uuid" }

- Response: { "id": "uuid", "status": "booked" }

GET /api/appointments/doctor/{doctor_id}

- Description: Retrieve doctor's schedule.

- Response: [{ "id": "uuid", "start_time": "datetime", "is_available": "boolean" }]

Billing Service APIs

POST /api/billing/bills

- Description: Generate a bill.

- Request: { "patient_id": "uuid", "amount": "decimal", "description": "string" }

- Response: { "id": "uuid", "amount": "decimal" }

POST /api/billing/claims

- Description: Submit an insurance claim.

- Request: { "bill_id": "uuid", "insurance_id": "string" }

- Response: { "id": "uuid", "status": "pending" }

Prescription Service APIs

POST /api/prescriptions

- Description: Create a prescription.

- Request: { "patient_id": "uuid", "medication": "string", "dosage": "string" }

- Response: { "id": "uuid", "medication": "string" }

PUT /api/prescriptions/{id}/dispense

- Description: Update prescription status to dispensed (called by Pharmacy Service).

- Request: { "status": "dispensed" }

- Response: { "id": "uuid", "status": "dispensed" }

Pharmacy Service APIs

POST /api/pharmacy/dispense

- Description: Dispense a prescription.

- Request: { "prescription_id": "uuid", "quantity": "integer" }

- Response: { "id": "uuid", "quantity": "integer" }

GET /api/pharmacy/stock

- Description: Retrieve inventory levels.

- Response: [{ "id": "uuid", "medication": "string", "quantity": "integer" }]

Laboratory Service APIs

POST /api/laboratory/tests

- Description: Create a test order.

- Request: { "patient_id": "uuid", "doctor_id": "uuid", "test_type": "string", "priority": "string" }

- Response: { "id": "uuid", "test_type": "string" }

POST /api/laboratory/results

- Description: Upload test results.

- Request: { "test_order_id": "uuid", "result_data": "string", "attachment": "file" }

- Response: { "id": "uuid", "result_data": "string" }

Notification Service APIs

POST /api/notifications/send

- Description: Send a notification (email/SMS).

- Request: { "user_id": "uuid", "message": "string", "type": "email|sms" }

- Response: { "id": "uuid", "status": "sent" }

GET /api/notifications/{user_id}

- Description: Retrieve user notification history.

- Response: [{ "id": "uuid", "message": "string", "sent_at": "datetime" }]

Communication Service APIs

POST /api/messages

- Description: Send a secure message.

- Request: { "recipient_id": "uuid", "content": "string" }

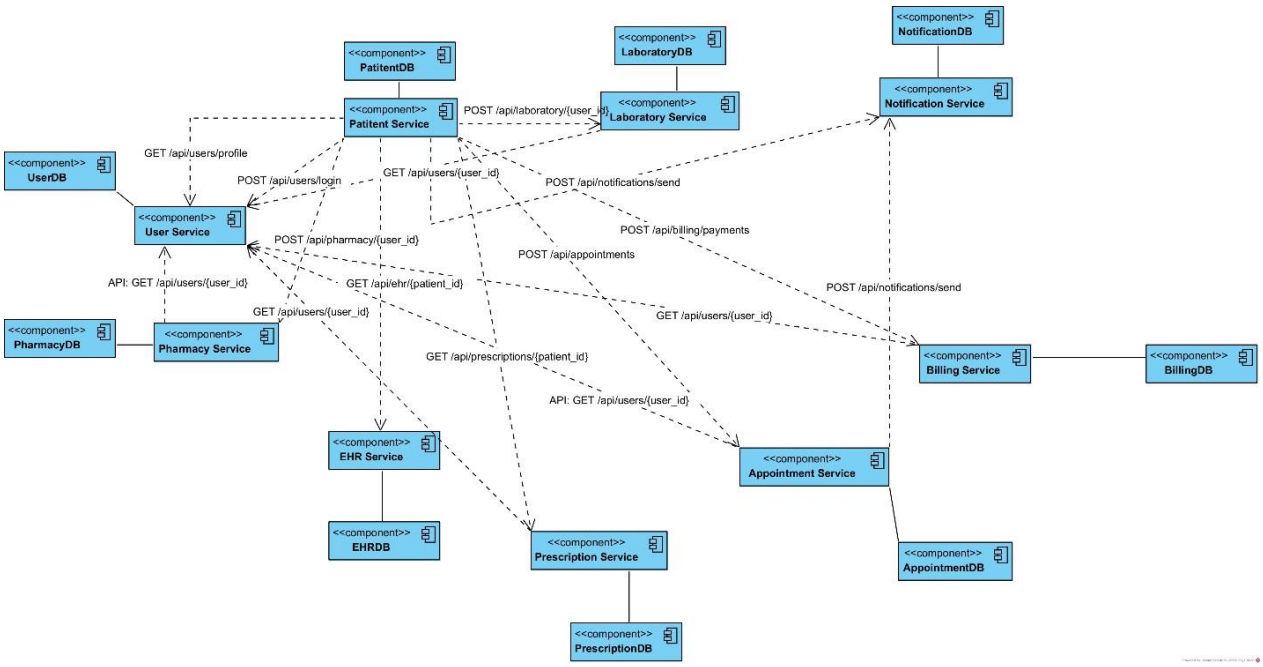
- Response: { "id": "uuid", "content": "string" }

GET /api/messages/{user_id}

- Description: Retrieve message history.

- Response: [{ "id": "uuid", "sender_id": "uuid", "content": "string" }]

Component diagram



CHAPTER 2: DESIGN HEALTHCARE SYSTEM WITH MICROSERVICES AND DJANGO

2.1 Design Services/Components

Microservices:

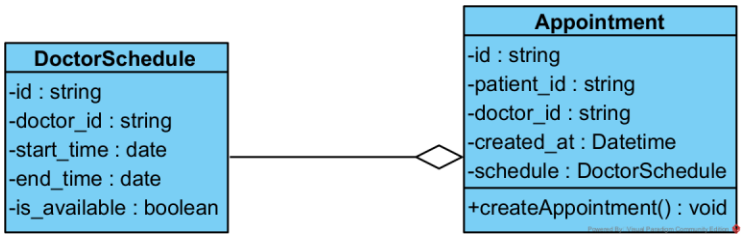
1. **User Service**: Manages user authentication, authorization, and role-based access control (RBAC) for all actors (1.4.1).
2. **Patient Service**: Handles patient-specific functions like appointment booking, medical record access, and billing (1.2.1, 1.3.1).
3. **EHR Service**: Manages electronic health records, including diagnoses, vitals, and test results (1.4.2).
4. **Appointment Service**: Manages doctor schedules and patient appointments (1.4.3).
5. **Billing Service**: Processes bills, payments, and insurance claims (1.4.4).
6. **Prescription Service**: Manages electronic prescriptions and refill workflows (1.4.5).
7. **Pharmacy Service**: Handles prescription dispensing and inventory management (1.3.3).
8. **Laboratory Service**: Manages test orders and result uploads (1.4.6).
9. **Notification Service**: Sends email/SMS notifications and manages templates (1.4.7).
10. **Communication Service**: Handles secure in-system messaging (1.4.7).

Architecture Overview:

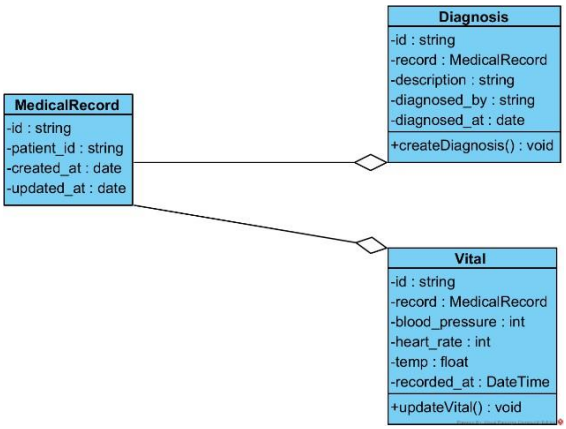
- **Communication**: REST APIs for synchronous calls; RabbitMQ for asynchronous events.
- **Database**: Each service uses a separate PostgreSQL database (Database-per-Service pattern).
- **Security**: JWT authentication, HTTPS, encrypted data storage, and audit logs for HIPAA/GDPR compliance.
- **Deployment**: Docker containers orchestrated with Kubernetes, monitored with Prometheus and Grafana.
- **Scalability**: Stateless services allow horizontal scaling; caching (Redis) for frequently accessed data.

2.2 Design classes and methods in component

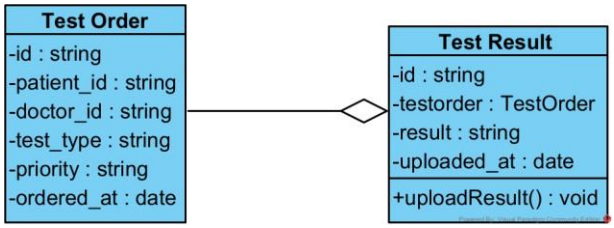
Appointment Service



EHR Service

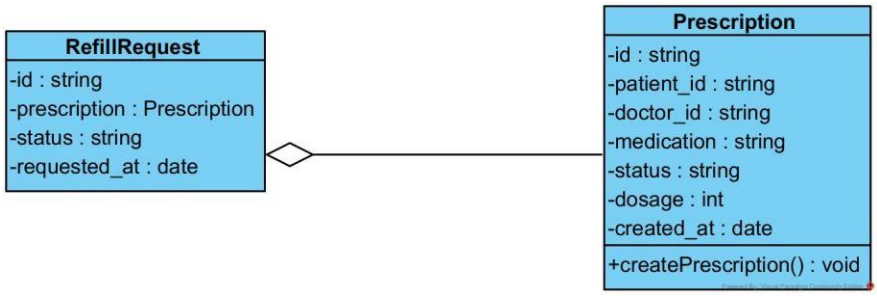


Laboratory Service



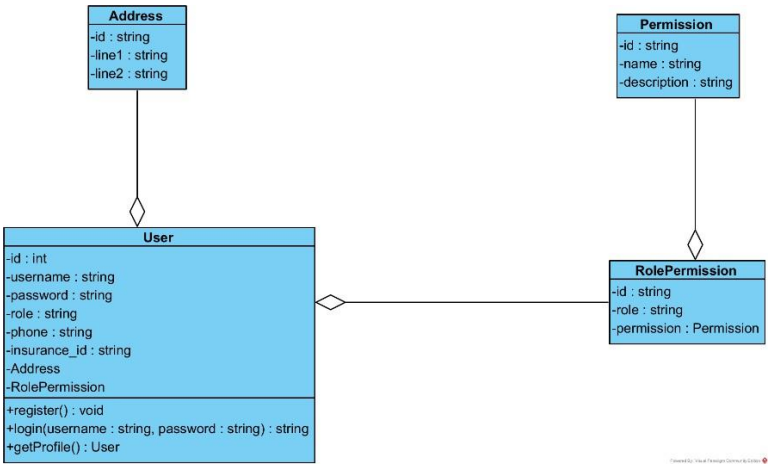
Inventory Service

Prescription Service

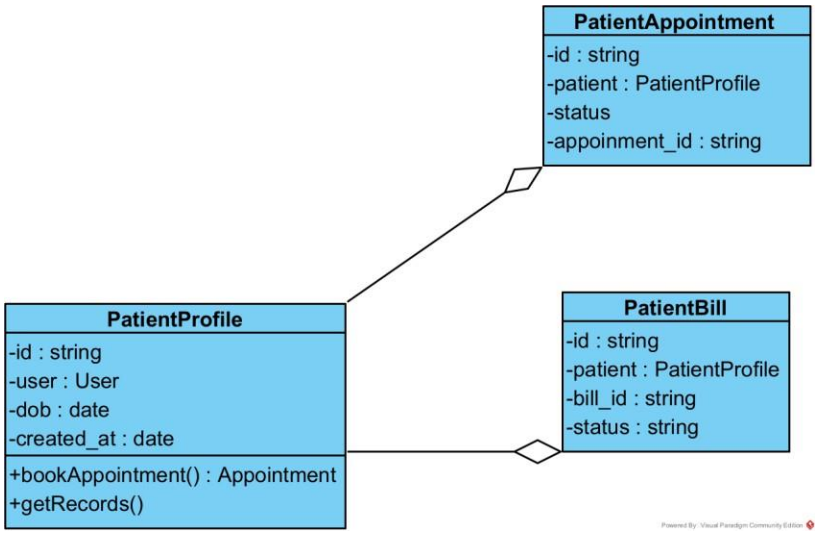


Bill Service

User Service



Patient Service



2.3 Design API

User Service APIs

```
POST /api/users/register
- Description: Register a new user (patient, doctor, nurse, etc.).
- Requirement: User registration (1.2.4, 1.3.1, 1.3.4).
- Request:
  {
    "username": "string",
    "password": "string",
    "role": "patient|doctor|nurse|admin|pharmacist|lab_tech",
    "phone": "string",
    "insurance_id": "string"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "username": "string",
    "role": "string"
  }
```

- Error (400 Bad Request):

```
{
  "errors": {"field": "error message"}
}
```

POST /api/users/login

- Description: Authenticate user and issue JWT token.
- Requirement: User authentication (1.4.1).
- Request:

```
{
  "username": "string",
  "password": "string"
}
```

- Response (200 OK):

```
{
  "token": "string"
}
```

- Error (401 Unauthorized):

```
{
  "error": "Invalid credentials"
}
```

GET /api/users/{user_id}

- Description: Retrieve user details (e.g., role, phone, insurance_id).
- Requirement: User profile access (1.2.1, 1.2.2, 1.4.1).
- Request: None (JWT token in Authorization header)
- Response (200 OK):

```
{
  "id": "uuid",
  "username": "string",
  "role": "string",
  "phone": "string",
```

```
    "insurance_id": "string"
  }
- Error (404 Not Found):
  {
    "error": "User not found"
  }

PUT /api/users/{user_id}
- Description: Update user profile (e.g., phone, insurance_id).
- Requirement: User profile management (1.2.1, 1.2.4).
- Request:
  {
    "phone": "string",
    "insurance_id": "string"
  }
- Response (200 OK):
  {
    "id": "uuid",
    "phone": "string",
    "insurance_id": "string"
  }
- Error (400 Bad Request):
  {
    "errors": {"field": "error message"}
  }
```

Patient Service APIs

```
POST /api/patients/appointments
- Description: Book an appointment by calling Appointment Service.
- Requirement: Appointment booking (1.2.1, 1.3.1).
- Request:
```

```
  {
    "doctor_id": "uuid",
    "start_time": "datetime"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "status": "booked",
    "doctor_id": "uuid",
    "start_time": "datetime"
  }
- Error (400 Bad Request):
  {
    "errors": {"field": "error message"}
  }

GET /api/patients/records
- Description: Retrieve patient medical records from EHR Service.
- Requirement: Medical record access (1.2.1, 1.3.1).
- Request: None (JWT token for patient)
- Response (200 OK):
  [
    {
      "id": "uuid",
      "diagnosis": "string",
      "date": "datetime"
    }
  ]
- Error (403 Forbidden):
  {
    "error": "Unauthorized access"
  }
```

```
POST /api/patients/payments
- Description: Process payment by calling Billing Service.
- Requirement: Bill payment (1.2.1, 1.3.1).
- Request:
  {
    "bill_id": "uuid",
    "amount": "decimal",
    "method": "credit_card|insurance"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "status": "paid"
  }
- Error (400 Bad Request):
  {
    "errors": {"bill_id": "Invalid bill"}
  }

GET /api/patients/prescriptions
- Description: Retrieve patient prescriptions from Prescription Service.
- Requirement: Prescription viewing (1.2.1, 1.3.1).
- Request: None (JWT token for patient)
- Response (200 OK):
  [
    {
      "id": "uuid",
      "medication": "string",
      "status": "active|dispensed|expired"
    }
  ]
```

- Error (403 Forbidden):

```
{
  "error": "Unauthorized access"
}
```

EHR Service APIs

POST /api/ehr/diagnoses

- Description: Record a diagnosis for a patient.
- Requirement: Diagnosis recording (1.2.2, 1.3.2).
- Request:

```
{
  "patient_id": "uuid",
  "icd_code": "string",
  "description": "string"
}
```

- Response (201 Created):

```
{
  "id": "uuid",
  "icd_code": "string",
  "description": "string"
}
```

- Error (400 Bad Request):

```
{
  "errors": {"icd_code": "Invalid code"}
}
```

POST /api/ehr/vitals

- Description: Record patient vitals.
- Requirement: Vital signs management (1.2.3, 1.3.2).
- Request:

```
{
```

```
    "patient_id": "uuid",
    "blood_pressure": "string",
    "heart_rate": "integer",
    "temperature": "float"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "blood_pressure": "string",
    "heart_rate": "integer"
  }
- Error (400 Bad Request):
  {
    "errors": {"heart_rate": "Invalid value"}
  }

POST /api/ehr/results
- Description: Upload test results from Laboratory Service.
- Requirement: Test result integration (1.2.7, 1.3.5).
- Request:
  {
    "test_order_id": "uuid",
    "result_data": "string",
    "attachment": "file"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "result_data": "string"
  }
- Error (400 Bad Request):
  {
```

```
    "errors": {"test_order_id": "Invalid order"}
  }

GET /api/ehr/{patient_id}
- Description: Retrieve patient medical records.
- Requirement: Medical record access (1.2.1, 1.2.2, 1.3.1, 1.3.2).
- Request: None (JWT token for patient/doctor/nurse)
- Response (200 OK):
  [
    {
      "id": "uuid",
      "diagnoses": [{"icd_code": "string"}],
      "vitals": [{"blood_pressure": "string"}],
      "results": [{"result_data": "string"}]
    }
  ]
- Error (403 Forbidden):
  {
    "error": "Unauthorized access"
  }
```

Appointment Service APIs

```
POST /api/appointments
- Description: Create an appointment.
- Requirement: Appointment scheduling (1.2.1, 1.3.1).
- Request:
  {
    "patient_id": "uuid",
    "doctor_id": "uuid",
    "start_time": "datetime"
  }
```

- Response (201 Created):

```
  {
    "id": "uuid",
    "status": "booked",
    "start_time": "datetime"
  }
```

- Error (400 Bad Request):

```
  {
    "errors": {"start_time": "Schedule unavailable"}
  }
```

```
GET /api/appointments/doctor/{doctor_id}
```

- Description: Retrieve doctor's available schedules.
- Requirement: Schedule viewing (1.2.1, 1.2.2, 1.3.1).
- Request: None (JWT token for patient/doctor)
- Response (200 OK):

```
  [
    {
      "id": "uuid",
      "start_time": "datetime",
      "is_available": "boolean"
    }
  ]
```

- Error (404 Not Found):

```
  {
    "error": "Doctor not found"
  }
```

```
PUT /api/appointments/{id}
```

- Description: Update or cancel an appointment.
- Requirement: Appointment management (1.2.1, 1.3.1).
- Request:

```
  {
    "status": "booked|cancelled",
    "start_time": "datetime"
  }
- Response (200 OK):
  {
    "id": "uuid",
    "status": "string"
  }
- Error (400 Bad Request):
  {
    "errors": {"status": "Invalid status"}
  }
```

Billing Service APIs

```
POST /api/billing/bills
- Description: Generate a bill for a patient.
- Requirement: Bill generation (1.2.1, 1.2.4, 1.3.1).
- Request:
  {
    "patient_id": "uuid",
    "amount": "decimal",
    "description": "string"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "amount": "decimal",
    "status": "pending"
  }
- Error (400 Bad Request):
```

```
  {
    "errors": {"amount": "Invalid amount"}
  }

POST /api/billing/payments
- Description: Process a payment for a bill.
- Requirement: Payment processing (1.2.1, 1.3.1).
- Request:
  {
    "bill_id": "uuid",
    "amount": "decimal",
    "method": "credit_card|insurance"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "status": "paid"
  }
- Error (400 Bad Request):
  {
    "errors": {"bill_id": "Invalid bill"}
  }

POST /api/billing/claims
- Description: Submit an insurance claim.
- Requirement: Insurance claim processing (1.2.6, 1.4.4).
- Request:
  {
    "bill_id": "uuid",
    "insurance_id": "string"
  }
- Response (201 Created):
```

```
  {
    "id": "uuid",
    "status": "pending"
  }
- Error (400 Bad Request):
  {
    "errors": {"insurance_id": "Invalid ID"}
  }
```

Prescription Service APIs

```
POST /api/prescriptions
- Description: Create a prescription for a patient.
- Requirement: Prescription creation (1.2.2, 1.3.2).
- Request:
  {
    "patient_id": "uuid",
    "medication": "string",
    "dosage": "string"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "medication": "string",
    "status": "active"
  }
- Error (400 Bad Request):
  {
    "errors": {"medication": "Invalid medication"}
  }

GET /api/prescriptions/{patient_id}
```

- Description: Retrieve patient prescriptions.
- Requirement: Prescription viewing (1.2.1, 1.3.1).
- Request: None (JWT token for patient/doctor/pharmacist)
- Response (200 OK):

```
[
  {
    "id": "uuid",
    "medication": "string",
    "status": "active|dispensed|expired"
  }
]
```

- Error (403 Forbidden):

```
{
  "error": "Unauthorized access"
}
```

PUT /api/prescriptions/{id}/dispense

- Description: Update prescription status to dispensed.
- Requirement: Prescription dispensing (1.2.5, 1.3.3).
- Request:

```
{
  "status": "dispensed"
}
```

- Response (200 OK):

```
{
  "id": "uuid",
  "status": "dispensed"
}
```

- Error (400 Bad Request):

```
{
  "errors": {"status": "Invalid status"}
}
```

```
POST /api/prescriptions/refills
- Description: Request a prescription refill.
- Requirement: Prescription refill (1.2.1, 1.3.1).
- Request:
  {
    "prescription_id": "uuid"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "status": "pending"
  }
- Error (400 Bad Request):
  {
    "errors": {"prescription_id": "Invalid prescription"}
  }
```

Pharmacy Service APIs

```
POST /api/pharmacy/dispense
- Description: Dispense a prescription and update inventory.
- Requirement: Prescription dispensing (1.2.5, 1.3.3).
- Request:
  {
    "prescription_id": "uuid",
    "quantity": "integer"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "quantity": "integer"
```

```
  }
- Error (400 Bad Request):
  {
    "errors": {"quantity": "Insufficient stock"}
  }

GET /api/pharmacy/stock
- Description: Retrieve medication inventory levels.
- Requirement: Inventory management (1.2.5, 1.3.3).
- Request: None (JWT token for pharmacist)
- Response (200 OK):
  [
    {
      "id": "uuid",
      "medication": "string",
      "quantity": "integer",
      "expiry_date": "date"
    }
  ]
- Error (403 Forbidden):
  {
    "error": "Unauthorized access"
  }

PUT /api/pharmacy/stock/{id}
- Description: Update medication inventory.
- Requirement: Inventory management (1.2.5, 1.3.3).
- Request:
  {
    "quantity": "integer",
    "expiry_date": "date"
  }
```

- Response (200 OK):

```
{
  "id": "uuid",
  "quantity": "integer"
}
```

- Error (400 Bad Request):

```
{
  "errors": {"quantity": "Invalid quantity"}
}
```

Laboratory Service APIs

POST /api/laboratory/tests

- Description: Create a test order for a patient.
- Requirement: Test order creation (1.2.2, 1.3.2).
- Request:

```
{
  "patient_id": "uuid",
  "doctor_id": "uuid",
  "test_type": "string",
  "priority": "normal|urgent"
}
```

- Response (201 Created):

```
{
  "id": "uuid",
  "test_type": "string",
  "priority": "string"
}
```

- Error (400 Bad Request):

```
{
  "errors": {"test_type": "Invalid type"}
}
```

```
POST /api/laboratory/results
- Description: Upload test results.
- Requirement: Test result upload (1.2.7, 1.3.5).
- Request:
  {
    "test_order_id": "uuid",
    "result_data": "string",
    "attachment": "file"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "result_data": "string"
  }
- Error (400 Bad Request):
  {
    "errors": {"test_order_id": "Invalid order"}
  }

GET /api/laboratory/results/{patient_id}
- Description: Retrieve patient test results.
- Requirement: Test result access (1.2.1, 1.2.2, 1.3.1, 1.3.2).
- Request: None (JWT token for patient/doctor)
- Response (200 OK):
  [
    {
      "id": "uuid",
      "test_type": "string",
      "result_data": "string",
      "uploaded_at": "datetime"
    }
```

```
  ]
- Error (403 Forbidden):
  {
    "error": "Unauthorized access"
  }
```

Notification Service APIs

```
POST /api/notifications/send
- Description: Send an email/SMS/in-app notification.
- Requirement: Notification delivery (1.4.7).
- Request:
  {
    "user_id": "uuid",
    "message": "string",
    "type": "email|sms|in_app"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "status": "sent"
  }
- Error (400 Bad Request):
  {
    "errors": {"type": "Invalid type"}
  }

GET /api/notifications/{user_id}
- Description: Retrieve user notification history.
- Requirement: Notification history (1.2.1, 1.2.2, 1.4.7).
- Request: None (JWT token for user)
- Response (200 OK):
```

```
  [
    {
      "id": "uuid",
      "message": "string",
      "type": "string",
      "sent_at": "datetime"
    }
  ]
- Error (403 Forbidden):
  {
    "error": "Unauthorized access"
  }
```

Communication Service APIs

```
POST /api/messages
- Description: Send a secure in-system message.
- Requirement: Secure messaging (1.4.7).
- Request:
  {
    "recipient_id": "uuid",
    "content": "string"
  }
- Response (201 Created):
  {
    "id": "uuid",
    "content": "string",
    "sent_at": "datetime"
  }
- Error (400 Bad Request):
  {
    "errors": {"recipient_id": "Invalid recipient"}
```

```
  }

GET /api/messages/{user_id}
- Description: Retrieve user message history.
- Requirement: Messaging history (1.2.1, 1.2.2, 1.4.7).
- Request: None (JWT token for user)
- Response (200 OK):
  [
    {
      "id": "uuid",
      "sender_id": "uuid",
      "content": "string",
      "sent_at": "datetime"
    }
  ]
- Error (403 Forbidden):
  {
    "error": "Unauthorized access"
  }
```

2.4 Conclusion

The microservices architecture, implemented with Django and DRF, meets the healthcare system's requirements with modularity, scalability, and security. Each service has well-defined models, methods, and APIs, connected via RESTful endpoints. The class diagrams clarify entity relationships, and the API specifications and sequence diagram illustrate inter-service communication. Security measures (JWT, encryption, RBAC) ensure HIPAA/GDPR compliance, while Docker/Kubernetes and RabbitMQ support deployment and asynchronous tasks. The design is extensible and mitigates challenges like latency with caching and event-driven patterns.