



---

# F21AO REPORT

---

Dev Part



## Group Members

**Mohamed Aman** - [ma2305@hw.ac.uk](mailto:ma2305@hw.ac.uk)

**Faisal Chaudhry** - [fmc4000@hw.ac.uk](mailto:fmc4000@hw.ac.uk)

**Mohamed Aqib Abid** - [ma4034@hw.ac.uk](mailto:ma4034@hw.ac.uk)

**Syed Arif Ali** - [sa4001@hw.ac.uk](mailto:sa4001@hw.ac.uk)

**Chen Bin** - [cb3000@hw.ac.uk](mailto:cb3000@hw.ac.uk)

## Table of Contents

1	Overview .....	3
2	Software Methodology.....	3
3	System Architecture.....	4
3.1	Microservices Overview .....	4
3.1.1	Auth Service .....	4
3.1.2	Patient Registration Service .....	4
3.1.3	Patient Treatment Service .....	4
3.2	Implementation of MVC (Model-View-Controller) .....	5
3.3	Justification for Microservices Architecture.....	6
3.3.1	Benefits of Microservices Architecture .....	6
3.3.2	Justification against a Monolithic Approach .....	6
4	Dev Phase Steps .....	7
4.1	Software Engineering Best Practices .....	7
4.2	Agile Development & Team Collaboration .....	7
4.3	Testing & Validation Strategies .....	8
4.4	API Documentation & Postman Collections .....	8
4.5	Database Optimization & Data Integrity .....	8
4.6	Security & Compliance Measures .....	8
4.7	Change Management & Version Control Enhancements .....	8
5	Change Management Strategy.....	9
5.1	Purpose .....	9
5.2	Scope of Application.....	9
5.3	Change Management Process .....	9
5.3.1	Propose a change request.....	9
5.3.2	Change the preliminary approval .....	9
5.3.3	Impact evaluation .....	10
5.3.4	Change Decision .....	11
5.3.5	Change Implementation.....	11
5.4	Change document management .....	11
5.5	Communication mechanism for change management .....	11
5.5.1	Communication meeting .....	11
5.5.2	Change notice .....	11
5.5.3	Problem feedback channel .....	12
5.6	Change risk management .....	12
5.7	Supplementary Provisions.....	12
6	Deployment Steps .....	13

6.1	Prerequisites .....	13
6.2	Cloning the Repository .....	13
6.3	Setting Up Environment Variables.....	13
6.4	Installing Dependencies .....	14
6.5	Running Each Microservice.....	14
6.6	Running MongoDB Locally (Optional) .....	14
6.7	API Testing with Postman.....	15
6.8	Dockerizing the Services (Further Implementation in Ops Phase) .....	15
6.9	Stopping the Services .....	15
6.10	Deployment to Cloud.....	15
7	Tools and Methodology Support .....	16
7.1	Version Control – Git & GitHub.....	16
7.2	Agile & Scrum Implementation Using Jira .....	16
7.3	Database – MongoDB Implementation.....	17
7.4	Microservices Interaction & Authentication via JWT .....	17
7.5	API Development & Testing – Postman.....	18
8	Functions and Endpoints .....	19
9	Test Cases.....	20
9.1	User Authentication .....	20
9.1.1	Registration Test Cases - Positive Tests .....	20
9.1.2	Registration Test Cases - Negative Tests.....	21
9.1.3	Sign-In Test Cases - Positive Tests.....	22
9.1.4	Sign-In Test Cases - Negative Tests .....	22
9.2	Patient Registration .....	22
9.2.1	Patient Registration - Positive Tests .....	22
9.2.2	Patient Registration - Negative Tests.....	23
9.3	Patient Diagnosis & Vitals .....	24
9.3.1	Diagnosis (Doctor Role) - Positive Tests .....	24
9.3.2	Diagnosis (Doctor Role) - Negative Tests.....	25
9.3.3	Vitals (Nurse Role) – Positive Tests.....	26
9.3.4	Vitals (Nurse Role) - Negative Tests.....	26
9.3.5	Treatment record – Positive Tests .....	27
9.3.6	Treatment Record - Negative Tests .....	28
9.4	Running collection in Postman: .....	29
9.5	Further Implementation .....	29
10	Resource Links.....	30

# 1 Overview

This report outlines the development of Patient Information System (PIS) designed for a tertiary care hospital. The system aims to manage patient data efficiently, including user authentication, patient registration, and treatment records. The project is structured as a microservices architecture, utilizing Node.js and MongoDB for backend development.

## 2 Software Methodology

The project follows the Agile methodology, which emphasizes iterative development, collaboration, and flexibility. Agile allows for continuous feedback and adaptation, ensuring that the system meets the evolving needs of users. The development process includes regular sprints, where features are developed, tested, and reviewed.

For this project, we used Scrum, an agile framework that structures development into fixed-length iterations. Scrum provided a clear structure for planning, development, and review, ensuring that tasks are completed efficiently. Also we used the project management tool JIRA to manage the development workflow.

Jira helped in:

- Creating and managing the product backlog, ensuring that all requirements were documented and prioritized.
- Defining sprint goals, where tasks were assigned and estimated based on priority and complexity.
- Tracking progress with Scrum boards, allowing team members to visualize tasks in different stages (To Do, In Progress, Done).
- Holding sprint reviews, where completed work was demonstrated and feedback was incorporated into the next sprint.
- Conducting retrospectives, enabling continuous process improvement after each sprint.
- Sprint Planning and Execution

Our development cycle followed regular sprints to ensure continuous progress. Each sprint lasted two weeks and included the following phases:

- Sprint Planning – The team selected tasks from the backlog, estimated effort, and defined the sprint goal.
- Development & Testing – Features were developed, tested, and reviewed iteratively.
- Daily Standups – Regular team meetings helped identify blockers and keep progress on track.
- Sprint Review – Completed features were demonstrated to gather feedback.
- Sprint Retrospective – The team reflected on challenges and improvements for the next sprint.

By following this structured approach, the project maintained a steady development pace, adapted to evolving requirements, and ensured high-quality deliverables.

## 3 System Architecture

The system architecture is designed to be scalable and maintainable. Each service communicates through REST APIs, allowing for independent deployment and updates.

The targeted deployment environment includes:

- Cloud Infrastructure: Utilizing cloud services for hosting the application.
- Containerization: Using Docker to containerize the services for easy deployment and scaling.

The system architecture is based on a microservices approach, where each service is responsible for specific functionality. This modular design enhances scalability, maintainability, and flexibility, allowing independent development and deployment of services.

### 3.1 Microservices Overview

#### 3.1.1 Auth Service

- Handles user authentication and authorization using JWT-based authentication.
- Manages user roles (e.g., Clerk, Doctor) to restrict access to specific features.
- Ensures secure login, logout and session validation.
- Provides authentication middleware for other services.

#### 3.1.2 Patient Registration Service

- Responsible for registering new patients by storing personal details such as name, contact information and medical history.
- Allows authorized users (Clerks) to retrieve patient information.
- Uses MongoDB as a database to store patient records in a structured format.
- Provides APIs to interact with the Patient Treatment Service.

#### 3.1.3 Patient Treatment Service

- Manages patient diagnoses, vitals, and treatment records.
- Doctors can add, update, and retrieve patient treatment details.
- Stores treatment history for future reference.
- Interacts with the Patient Registration Service to fetch patient details when needed.

# Visual Representation of Overall Architecture

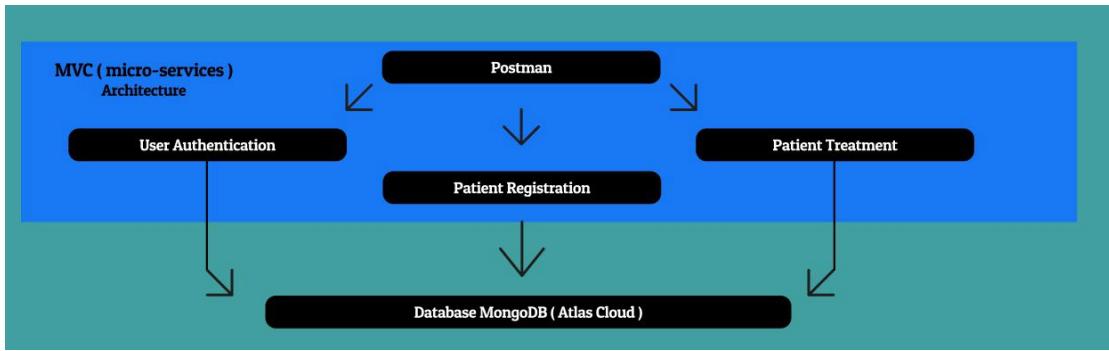


Fig. 1 - Visual Representation of Overall Architecture

## 3.2 Implementation of MVC (Model-View-Controller)

Each microservice follows the MVC (Model-View-Controller) pattern, ensuring a clean separation of concerns:

- **Model (M)** – Represents the data layer and interacts with the MongoDB database. Each service has its own models (e.g., User, Patient, Treatment).
- **Controller (C)** – Handles business logic by processing requests and interacting with the Model.
- **Routes (View equivalent in APIs)** – Defines API endpoints that clients use to access data and services.

Example: MVC in Patient Registration Service

```
services > patient-registration > src > models > JS Patient.js > ...
1 const mongoose = require('mongoose');
2 const Counter = require('.././shared/counterModel');
3
4
5 const patientSchema = new mongoose.Schema({
6   patientId: {
7     type: String,
8     unique: true
9   },
10  firstName: {
11    type: String,
12    required: true
13  },
14  lastName: {
15    type: String,
16    required: true
17  },
18  mobile: {
19    type: String
20  }
21});
```

Fig. 2 - Model (Patient.js)

```
services > patient-registration > src > controllers > JS patientController.js > registerPatient > existingPatient
1 const Patient = require('../models/patient');
2
3 // Register a new patient (Only for clerks)
4 exports.registerPatient = async (req, res) => {
5   try {
6     const { firstname, lastname, mobile, email, diseaseHistory } = req.body;
7
8     // Check if patient already exists
9     const existingPatient = await Patient.findOne({ mobile });
10
11    if (existingPatient) {
12      return res.status(400).json({ message: 'Patient with this mobile number already exists.' });
13    }
14
15    // Create and save new patient
16    const newPatient = new Patient({ firstname, lastname, mobile, email, diseaseHistory });
17    await newPatient.save();
18
19    res.status(201).json({ message: 'Patient registered successfully', patientId: newPatient.patientId });
20  } catch (error) {
21    res.status(500).json({ message: 'Error registering patient', error: error.message });
22  }
23};
```

Fig. 3 - Controller (patientController.js)

```
services > patient-registration > src > routes > JS patientRoutes.js > ...
1 const express = require('express');
2 const router = express.Router();
3 const patientController = require('../controllers/patientController');
4 const { verifyToken, checkClerkRole } = require('../middlewares/authMiddleware');
5
6 router.post('/register', verifyToken, checkClerkRole, patientController.registerPatient);
7 router.get('/all', verifyToken, checkClerkRole, patientController.getAllPatients);
8 router.get('/:pid', verifyToken, checkClerkRole, patientController.getPatientById);
9
10 module.exports = router;
```

Fig. 4 - Routes (patientRoutes.js)

### 3.3 Justification for Microservices Architecture

Here's a detailed justification for using a microservices approach over a monolithic architecture in the project.

#### 3.3.1 Benefits of Microservices Architecture

The system consists of distinct functionalities such as authentication, patient registration, and patient treatment, making microservices the ideal choice for scalability, maintainability, and fault isolation.

Factor	Microservices Advantage	Monolithic Limitation
Scalability	Each service can scale independently based on demand.	The entire application must scale together, leading to resource inefficiencies.
Maintainability	Smaller, focused services are easier to update and manage.	Any small update requires redeploying the entire application.
Fault Isolation	Failure in one service (e.g., Patient Registration) does not crash the entire system.	A bug in one module can bring down the whole system.
Technology Stack	Different services can use different technologies if needed.	The entire application is limited to a single technology stack
Continuous Deployment (CI/CD)	Services can be deployed independently, enabling faster updates.	Every change requires testing and redeploying the entire system.
Security	Role-based access control (RBAC) can be implemented per service, ensuring stricter security.	Harder to enforce service-specific security measures.

#### 3.3.2 Justification against a Monolithic Approach

A monolithic architecture would mean bundling all functionalities into a single application, causing:

- Increased deployment complexity: Every update requires full redeployment.
- Scaling inefficiencies: You cannot scale only the required modules (e.g., scaling just authentication when login traffic spikes).
- Single point of failure: A bug in one module can crash the entire system.

## 4 Dev Phase Steps

The development phase consists of the following steps:

- Requirement Analysis - Gathered requirements for authentication, patient registration, and treatment modules.
- Design - Created a design document outlining the architecture and data flow.
- Implementation - Developed the three modules using Node.js and MongoDB.
- Testing - Conducted unit and integration testing to ensure functionality.
- Documentation - Documented the code and created user guides.

Some of the key points hashed out during the Dev-Phase are mentioned below

Phase/ Milestone	Completion	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Sprint 1 (User Authentication)	100%						
Sprint 2 (Patient Registration)	100%						
Sprint 3 (Patient Treatment)	100%						
Sprint 4 (Testing and Reporting)	100%						

Fig. 6 - Gantt Chart

### 4.1 Software Engineering Best Practices

- Coding Standards & Conventions: Consistent naming conventions, structured file organization, and inline documentation to enhance maintainability.
- Error Handling & Logging: Implementing structured API error responses and using Winston/Morgan for centralized logging.
- Secure API Design: RESTful API principles with JWT authentication and role-based access control.

### 4.2 Agile Development & Team Collaboration

- Sprint Planning & Execution: Bi-weekly sprint cycles defining key deliverables and tracking progress.
- JIRA Task Management: Organized backlog, epics, and sprint tracking with screenshots of JIRA board.
- Code Reviews & Merging Strategy: Peer reviews before merging, with approval required for each pull request.

## 4.3 Testing & Validation Strategies

- Unit Testing: Mocha/Jest tests for core functionalities.
- Integration Testing: API endpoint testing to validate data flow between microservices.
- Continuous Testing in CI/CD: Automated tests triggered on GitHub Actions or Jenkins pipelines.

## 4.4 API Documentation & Postman Collections

- API Contracts: Endpoint specifications including required parameters and expected responses.
- Postman Test Suites: Automated request validation and response verification.
- Swagger/OpenAPI Documentation: API reference for external integrations.

## 4.5 Database Optimization & Data Integrity

- Schema Design Considerations: Structured MongoDB collections with validation constraints.
- Data Validation Rules: Mongoose schema validations for required fields.
- Indexing & Query Optimization: Indexed common query fields like patientId for performance improvement.

## 4.6 Security & Compliance Measures

- Role-Based Access Control (RBAC): Defined role-based permissions for clerks, doctors, and nurses.
- JWT Security Measures: Token expiration and refresh token implementation.
- Data Privacy Compliance: Anonymization of patient data for external research interfaces.

## 4.7 Change Management & Version Control Enhancements

- Git Workflow: Feature branching (feature/auth, feature/patient-registration) for modular development.
- Automated Deployments: CI/CD pipeline triggers automated builds and tests.
- Rollback & Disaster Recovery: Revert strategy for faulty deployments to minimize downtime.

# 5 Change Management Strategy

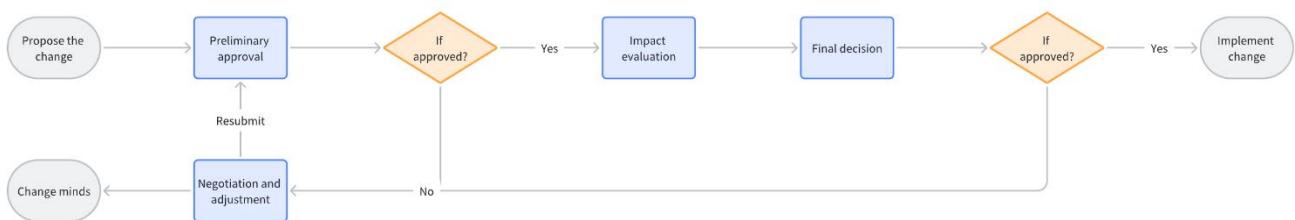
## 5.1 Purpose

To effectively manage change requests during the project development process, ensure timely evaluation, decision-making, and implementation of changes, rapidly respond to business needs, and maintain the orderly progression of the project, we have established these change management measures.

## 5.2 Scope of Application

In order to effectively manage the change request in the process of project development, ensure the timely evaluation, decision and implementation of the change, quickly respond to business needs, and maintain the orderly progress of the project, we have formulated a series of change management measures according to the Agile Change Management (ACM) framework.

## 5.3 Change Management Process



**Fig. 7 – Change Management Chart**

### 5.3.1 Propose a change request

#### 5.3.1.1 Requester

All personnel involved in the project, such as customers, project team members, stakeholders, etc., can make change requests.

#### 5.3.1.2 Submission method

The change request should be sent to the product owner by email, and the Change Request Form should be filled out (refer to Attachment 1), describing the change content, reason and expected time in detail.

### 5.3.2 Change the preliminary approval

#### 5.3.2.1 Approver

The project product owner is responsible for the Preliminary approval

### 5.3.2.2 Preliminary approval

- Review change requests for completeness and clarity to ensure that information is filled in accurately and in detail.
- Determine whether the change request is consistent with the overall objectives and scope of the project.
- Assess the business value of the change.

### 5.3.2.3 Preliminary results

- **Approved** - Go to the impact evaluation stage.
- **Disapproved** - The product owner shall communicate with the change requester, explain the reasons for the failure, and negotiate to modify the change request or abandon the change.

## 5.3.3 Impact evaluation

### 5.3.3.1 Evaluation team

The change evaluation team consists of product owner, Agile coach, development team leader, test team leader, etc.

### 5.3.3.2 Evaluating the content

- **Technical feasibility**

The development team leader assesses whether the change is technically feasible, whether it will have a significant impact on the existing system architecture, code, and the technical resources and time required to implement the change.

- **Cost impact**

Evaluate the impact of changes on project costs, including labour costs, material costs, etc.

- **Schedule impact**

Analyse the impact of the change on the project schedule, determine whether the project will be delayed, and the time of delay.

- **Quality Impact**

Quality assurance personnel evaluate the potential impact of changes on system quality, such as whether new defects will be introduced, whether it will affect system stability and security, etc.

### 5.3.3.3 Evaluation results

The evaluation team shall complete the evaluation within the specified time (generally 1-2 working days) and submit the Change Evaluation Report, which includes the evaluation conclusion (whether the change is recommended), the proposed plan for the implementation of the change, and the analysis of the impact of the change on all aspects of the project.

## 5.3.4 Change Decision

### 5.3.4.1 Decision makers

The product owner makes the final decision according to the Change Evaluation Report.

### 5.3.4.2 Decision result

- **Approve the changes**

Prioritize requirements, adjust Backlog or add the next sprint based on the priority.

- **Change rejection**

Explain the reason for rejection to the change requester, and negotiate to modify the change request or abandon the change.

- **Deferred processing**

For some change requests that have a large impact or require further study, it can be decided to postpone processing, and then re-evaluate and make decisions when conditions are ripe.

## 5.3.5 Change Implementation

### Change development and testing

The Agile coaching team shall organize relevant personnel to make a detailed change implementation plan and clarify the change tasks, responsible persons, time nodes and resource requirements.

## 5.4 Change document management

All change-related documents, including the Change Request Form, change evaluation report, change implementation plan, test report, etc., shall be recorded and saved in a timely and accurate manner for easy inquiry and retrieval.

## 5.5 Communication mechanism for change management

### 5.5.1 Communication meeting

Hold change management communication meeting as needed, chaired by product owner and attended by project team members. The meeting mainly discussed this week's change request, evaluation results, implementation progress and other issues, and coordinated to solve the contradictions and problems in the change management process.

### 5.5.2 Change notice

The product owner shall timely send a change notice to the relevant personnel of the project to inform the status and progress of the change during the various stages of change request submission, evaluation, approval, implementation, verification and release.

### 5.5.3 Problem feedback channel

Establish a problem feedback channel. When project related personnel encounter problems or have suggestions in the change management process, they can give feedback to the product owner through email, instant messaging tools, etc.

## 5.6 Change risk management

In the process of change assessment and implementation, identify possible risks and formulate corresponding risk countermeasures. For example, for changes that may cause project delays, you can add resources, adjust your schedule, and so on.

Regularly monitor and evaluate the change risk management, timely adjust the risk response strategy to ensure the smooth implementation of the change.

## 5.7 Supplementary Provisions

These Measures shall take effect as of the date of promulgation. If there are matters not mentioned, they may be supplemented and improved according to the actual situation of the project.

The project management team shall be responsible for the interpretation and implementation of this method.

## Attachment

Change Request Form		
Change Description		
Project Name:	Item Requiring Change:	Change Number:
Requested by:	Date Needed:	Date Requested:
Description of Change:		
Reason for Change:		
Priority:		
Impact of not Implementing the Change (and Reason Why)		
Change Impact		
Task/ Scope:		
Cost Evaluation:		
Additional Resources:		
Duration:		
Additional Effort:		
Impact on Deadline:		
Alternative & Recommendations:		
Comments:		
Sign Offs		
[Circle One] : 1. Approved 2. Not Approved 3. Deferred		
Comments:		
Project Sponsor/ Steering Committee Signature:	Date:	

**Fig. 8 – Change Request Form**

# 6 Deployment Steps

Deployment is a crucial phase in the software development lifecycle, ensuring that the developed application is successfully installed, configured, and made available for use. This section provides a comprehensive guide to deploying the Patient Information System (PIS) developed using Node.js and MongoDB. The deployment process includes setting up the environment, installing dependencies, running services, and testing APIs. Additionally, optional steps for containerization using Docker and cloud deployment considerations are outlined to enhance scalability and maintainability.

## 6.1 Prerequisites

Before deploying the system, ensure you have the following installed:

- Node.js (v16 or later)
- MongoDB (local or cloud instance)
- Git
- Postman (for API testing)
- Docker (for containerized deployment)

## 6.2 Cloning the Repository

Run the following command to clone the project repository:

```
git clone https://github.com/faisalmc/TCH-PIS.git  
cd TCH-PIS
```

## 6.3 Setting Up Environment Variables

Each microservice requires environment variables for configuration.

1. Create an .env file in each service directory (auth-service, patient-registration, patient-treatment).
2. Add the following variables inside .env:

```
MONGO_URI=mongodb+srv://fmc4000:PcMk4CYYjrNQ9LW8@patient-info-  
system.qj5ku.mongodb.net/patientdb?retryWrites=true&w=majority  
  
JWT_SECRET=e5b8a6d8f92c
```

## 6.4 Installing Dependencies

Navigate to the directory of each service and install the required dependencies:

```
cd auth-service  
npm install  
  
cd ../patient-registration  
npm install  
  
cd ../patient-treatment  
npm install
```

## 6.5 Running Each Microservice

Start the Authentication Service

```
cd auth-service/src  
node server.js
```

Start the Patient Registration Service

```
cd ../patient-registration/src  
node server.js
```

Start the Patient Treatment Service

```
cd ../patient-treatment/src  
node server.js
```

Each service should now be running on its respective port (3000, 3001, 3002).

## 6.6 Running MongoDB Locally (Optional)

If using a local MongoDB instance, start it with:

```
mongod --dbpath /data/db
```

Or use Docker to run MongoDB:

```
docker run -d --name mongodb -p 27017:27017 -e  
MONGO_INITDB_ROOT_USERNAME=admin -e  
MONGO_INITDB_ROOT_PASSWORD=admin mongo
```

## 6.7 API Testing with Postman

- Open Postman.
- Import the provided Postman Collection.
- Test endpoints:
  - Authentication: POST /auth/register, POST /auth/login
  - Patient Registration: POST /patients/register, GET /patients/all
  - Patient Treatment: POST /treatment/diagnosis, GET /treatment/:patientID

## 6.8 Dockerizing the Services (Further Implementation in Ops Phase)

To run services inside Docker containers, create a Dockerfile for each service:

Example Dockerfile for Auth Service

```
FROM node:16
WORKDIR /app
COPY package.json .
RUN npm install
COPY ..
CMD ["npm", "run", "dev"]
EXPOSE 3000
```

Then, build and run the container:

```
docker build -t auth-service .
docker run -p 3000:3000 auth-service
```

## 6.9 Stopping the Services

To stop the services, use:

CTRL + C # Stop a running process

Or if running with Docker:

```
docker stop <container_id>
```

## 6.10 Deployment to Cloud

For cloud deployment, methods considered are:

- AWS EC2 / Azure VM: Hosting services.
- MongoDB Atlas: Cloud database.
- Docker Compose / Kubernetes: Orchestrating multiple services.

## 7 Tools and Methodology Support

The chosen tools and methodologies support change management, continuous development, and continuous testing in the following ways:

### 7.1 Version Control – Git & GitHub

We used Git for version control with a single repository and only the master branch.

- All codes for Auth Service, Patient Registration Service, and Patient Treatment Service were maintained in a single GitHub repository.
- We used commit messages and timestamps to track updates.
- Developers followed a manual code review process before pushing changes.

#### Example Git Workflow in Our Project

- A developer modifies a service, e.g., adds patient registration logic.
- Runs local tests before committing the changes.
- Commits and pushes directly to master:
  - git add .
  - git commit -m "Created patient registration service"
  - git push origin master

### 7.2 Agile & Scrum Implementation Using Jira

The project followed the Scrum framework within Agile methodology, managed using Jira.

- Sprint Duration: We worked in two-week sprints, focusing on specific system functionalities.
- Task Management: Each feature was broken down into user stories and tasks in Jira.
- Daily Standups: We discussed progress and blockers informally.
- Sprint Reviews: At the end of each sprint, completed features were tested and deployed.

#### Jira Implementation in Our Project

- Created a Jira board for the Patient Information System.
- Defined epics for the main services (Auth, Patient Registration, Treatment).
- Created tasks like:
  - "Develop patient registration API in Node.js."
  - "Implement user authentication service"
- Tracked tasks through stages: To Do → In Progress → Testing → Done.

## 7.3 Database – MongoDB Implementation

Our project used MongoDB as a single database with separate collections for each service.

- Auth Service - Stored user credentials and role-based access.
- Patient Registration Service - Stored patient details.
- Patient Treatment Service - Stored diagnoses and prescriptions.

Example Schema in Our Project (Patient Collection in MongoDB)

- `models/Patient.js`

```
const mongoose = require('mongoose');

const Counter = require('../...../shared/counterModel');

const patientSchema = new mongoose.Schema({
    patientId: {type: String, unique: true},
    firstName: {type: String, required: true},
    lastName: {type: String, required: true},
    mobile: {type: String, required: true, unique: true},
    email: {type: String, required: true},
    diseaseHistory: {type: String},
    createdAt: {type: Date, default: Date.now}
});

module.exports = mongoose.model('Patient', patientSchema);
```

All services shared the same MongoDB database but interacted with separate collections to ensure data organization and security.

## 7.4 Microservices Interaction & Authentication via JWT

Since the system was microservices-based, services communicated via RESTful APIs and used JWT authentication for security.

How Services Interacted in Our Project

- The Auth Service issued a JWT token upon login.
- The Patient Registration and Treatment Services required a valid token to access data along with proper role.

### Example: JWT Token generation in user authentication

```
const jwt = require('jsonwebtoken');
const token = jwt.sign(
    { userId: user._id, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: '1h' }
);
res.json({ token, role: user.role });
}
catch (error) {
    res.status(500).json({ message: 'Error logging in', error: error.message });
}
```

### Example: JWT Token validation in patient registration

```
const jwt = require('jsonwebtoken');

exports.verifyToken = (req, res, next) => {
    const token = req.header('Authorization');
    if (!token) return res.status(401).json({ message: 'Access Denied' });

    try {
        const verified = jwt.verify(token.replace('Bearer ', ''), process.env.JWT_SECRET);
        req.user = verified; // Attach user data to request
        next();
    } catch (error) {
        res.status(400).json({ message: 'Invalid Token' });
    }
};
```

## 7.5 API Development & Testing – Postman

We used Postman for manual and automated API testing before deployment.

- Created a Postman collection with predefined requests for all APIs.
- Used environment variables for different setups (development, testing).
- Performed manual testing after each update to the master branch.

### Example API Test Case in Postman

For the User Authentication API (/register), we wrote test scripts to validate the response:

# 8 Functions and Endpoints

## Auth Service Endpoints

- **POST /auth/register** : Register a new user.
- **POST /auth/login** : Log in a user.

## Patient Registration Service Endpoints

- **POST /patient/register** : Register a new patient.
- **GET /patient/all** : Retrieve all patients.
- **GET /patient/:id** : Retrieve a patient by ID.

## Patient Treatment Service Endpoints

- **POST /api/treatment/diagnosis** : Add a diagnosis for a patient.
- **POST /api/treatment/vitals** : Log patient vitals.
- **GET /treatment/:patientID** : Retrieve a patient's treatment history.
- **PUT /api/treatment/medications/:patientID** : Update medications for a patient
- **DELETE /api/treatment/medications/:patientID** : delete medications for patient

## 9 Test Cases

In postman collection, named “Continuous testing”, is designed to validate the functionality and security of our healthcare management system’s API endpoints. It includes authentication, patient registration, diagnosis, vitals recording, and treatment history retrieval. The tests are structured in such a way that it simulates the real-world scenarios, both positive and negative test cases.

Key components includes:

1. Authentication – Registration and Sign-In
2. Patient Registration – Creation
3. Diagnosis and Vitals – Role-Based Access for doctors and nurses
4. Treatment Records – Retrieval

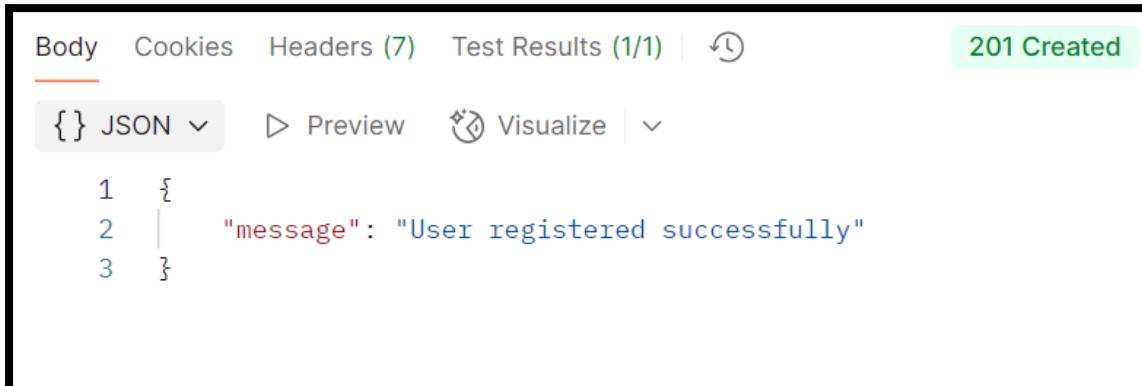
### 9.1 User Authentication

#### 9.1.1 Registration Test Cases - Positive Tests

- Successful Registration

Generates random usernames and passwords.

Checks it returns 201 status code and message in the response.



The screenshot shows the Postman interface with a successful registration response. The top navigation bar includes 'Body' (underlined), 'Cookies', 'Headers (7)', 'Test Results (1/1)', and a refresh icon. To the right, a green box displays '201 Created'. Below the navigation, there are tabs for 'JSON' (selected) and 'Preview'. The JSON response body is shown as follows:

```
1  {
2    "message": "User registered successfully"
3 }
```

## 9.1.2 Registration Test Cases - Negative Tests

- **Invalid role handling:**

Rejects invalid roles. Checks for 500 error and validation message.

The screenshot shows a Postman test result window. The status bar at the top indicates a 500 Internal Server Error, 104 ms duration, and 380 B size. The body of the response is displayed as JSON, showing an error message and validation details:

```
1 {  
2   "message": "Error registering user",  
3   "error": "User validation failed: role: 'non' is not a valid enum value for path 'role'."  
4 }
```

- **Missing Fields:**

Missing username/password triggers 500 as status code with an error message.

The screenshot shows a Postman test result window. The status bar at the top indicates a 500 Internal Server Error, 31 ms duration. The body of the response is displayed as JSON, showing an error message and validation details:

```
1 {  
2   "message": "Error registering user",  
3   "error": "Illegal arguments: undefined, number"  
4 }
```

- **Duplicate Username:**

Rejects registration if the username exists- status code as 400

The screenshot shows a Postman test result window. The status bar at the top indicates a 400 Bad Request. The body of the response is displayed as JSON, showing an error message:

```
1 {  
2   "message": "Username already exists"  
3 }
```

- **Invalid Password Format:**

Numeric passwords are rejected – error code as 500, needs to be a string

The screenshot shows a Postman test result window. The status bar at the top indicates a 500 Internal Server Error, 30 ms duration. The body of the response is displayed as JSON, showing an error message and validation details:

```
1 {  
2   "message": "Error registering user",  
3   "error": "Illegal arguments: number, number"  
4 }
```

### 9.1.3 Sign-In Test Cases - Positive Tests

- **Valid credentials**

Returns 200 okay message along with JWT token

Body Cookies Headers (7) Test Results (1/1) 200 OK 106 ms 461 B Save Response

{ } JSON Preview Visualize

```
1 {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
3     eyJ1c2VySWQiOiI2NzI1ZjAzOTk5MjYzMwMzY4NGUiLCJyb2x1IjoiZG9jdG9yIiwiaWF0IjoxNzQwMzkzOTY2LCJ  
4   "role": "doctor"
```

### 9.1.4 Sign-In Test Cases - Negative Tests

- **Invalid Credentials:**

Wrong username/password returns 401 Unauthorized

Body Cookies Headers (7) Test Results (1/1) 401 Unauthorized

{ } JSON Preview Visualize

```
1 {  
2   "message": "Invalid credentials"  
3 }
```

- **Missing Fields**

Empty username/password triggers 500 errors

Body Cookies Headers (7) Test Results (1/1) 500 Internal Server Error 30 ms 33

{ } JSON Preview Visualize

```
1 {  
2   "message": "Error logging in",  
3   "error": "Illegal arguments: undefined, string"  
4 }
```

## 9.2 Patient Registration

### 9.2.1 Patient Registration - Positive Tests

- **Successful Registration**

Generates random patient data – name, mobile number and email

Validates 201 Created and a patientId

The screenshot shows a Postman test result window. At the top right, it says "201 Created". Below that, there's a JSON response body:

```
1 {  
2   "message": "Patient registered successfully",  
3   "patientId": "1040"  
4 }
```

### 9.2.2 Patient Registration - Negative Tests

- **Duplicate Mobile Number:**

Returns 400 Bad request, if mobile exist

The screenshot shows a Postman test result window. At the top right, it says "400 Bad Request". Below that, there's a JSON response body:

```
1 {  
2   "message": "Patient with this mobile number already exists"  
3 }
```

- **Missing Fields:**

Omitting mandatory fields like firstName, lastName, etc., triggers 500 as status code with an error message

The screenshot shows a Postman test result window. At the top right, it says "500 Internal Server Error". Below that, there's a JSON response body:

```
1 {  
2   "message": "Error registering patient",  
3   "error": "Patient validation failed: lastName: Path 'lastName' is required."  
4 }
```

- **Authorization Issues:**

Missing/invalid tokens return 401 Unauthorized or 400 Invalid Token

Body Cookies Headers (7) Test Results (1/1) 401 Unauthorized 4 ms 2

{ } JSON ▾ Preview Visualize ▾

```

1  {
2   |   "message": "Access Denied"
3 }
```

Body Cookies Headers (7) Test Results (1/1) 400 Bad Request

{ } JSON ▾ Preview Visualize ▾

```

1  {
2   |   "message": "Invalid Token"
3 }
```

## 9.3 Patient Diagnosis & Vitals

### 9.3.1 Diagnosis (Doctor Role) - Positive Tests

- **Valid Diagnosis Creation**

Creates a record with status code as 201 and treatment object

Body Cookies Headers (7) Test Results (1/1) 201 Created 98 ms 1.4 KB Save Re

{ } JSON ▾ Preview Visualize ▾

```

1  {
2   |   "message": "Diagnosis recorded successfully",
3   |   "treatment": {
4   |   |   "_id": "67bb26c0e5db861ce5b8ec94",
5   |   |   "patientID": "67abada98e7056a95b8599ee",
6   |   |   "doctorID": "67bb267e763edeab0e1753f5",
7   |   |   "diagnosis": "Sample diagnosis",
8   |   |   "vitals": [
9   |   |   |   {} ] }
```

- **Updating Existing Diagnosis**

Validates medication list and updates, returns status code 201

Body Cookies Headers (7) Test Results (1/1) 201 Created 98 ms 1.4 KB Save Re

{ } JSON ▾ Preview Visualize ▾

```

1  {
2   |   "message": "Diagnosis recorded successfully",
3   |   "treatment": {
4   |   |   "_id": "67bb26c0e5db861ce5b8ec94",
5   |   |   "patientID": "67abada98e7056a95b8599ee",
6   |   |   "doctorID": "67bb267e763edeab0e1753f5",
7   |   |   "diagnosis": "Sample diagnosis",
8   |   |   "vitals": [
9   |   |   |   {} ] }
```

### 9.3.2 Diagnosis (Doctor Role) - Negative Tests

- **Unauthorized Access**

Non-doctors receive 403 Forbidden

The screenshot shows a Postman test result window. The status bar at the top right indicates "403 Forbidden". The "Test Results" section shows a single entry with the following JSON response:

```
1 {  
2   "message": "Access Denied: Doctors only"  
3 }
```

- **Missing Fields:**

Missing patientID/diagnosis triggers 400 Bad Request

The screenshot shows a Postman test result window. The status bar at the top right indicates "400 Bad Request". The "Test Results" section shows a single entry with the following JSON response:

```
1 {  
2   "message": "Error: patientID and diagnosis are required"  
3 }
```

- **Invalid Patient ID Format**

Invalid Patient-Object IDs return 500 error

The screenshot shows a Postman test result window. The status bar at the top right indicates "500 Internal Server Error". The "Test Results" section shows a single entry with the following JSON response:

```
1 {  
2   "message": "Error recording diagnosis",  
3   "error": "input must be a 24 character hex string, 12 byte Uint8Array, or an integer"  
4 }
```

### 9.3.3 Vitals (Nurse Role) – Positive Tests

- **Valid Vitals Entry**

Creates a valid entry with temperature and blood pressure

The screenshot shows a Postman test result window. At the top right, it says "201 Created". Below that, there's a JSON response body:

```
1 {  
2   "message": "Vitals recorded successfully",  
3   "treatment": {  
4     "_id": "67bb26c0e5db861ce5b8ec94",  
5     "patientID": "67abada98e7056a95b8599ee",  
6     "doctorID": "67bb267e763edeab0e1753f5",  
7     "diagnosis": "Sample diagnosis",  
8     "vitals": [  
9       {
```

### 9.3.4 Vitals (Nurse Role) - Negative Tests

- **Missing Data**

Missing temperature/bloodPressure returns 400

The screenshot shows a Postman test result window. At the top right, it says "400 Bad Request". Below that, there's a JSON response body:

```
1 {  
2   "message": "Error: patientID, temperature, and bloodPressure are required"  
3 }
```

- **Unauthorized Access:**

Non-nurses get 403 Forbidden

The screenshot shows a Postman test result window. At the top right, it says "403 Forbidden". Below that, there's a JSON response body:

```
1 {  
2   "message": "Access Denied: Nurses only"  
3 }
```

- **Invalid Patient Id**

Return 500 error code with appropriate response message

A screenshot of the Postman application interface. At the top, there's a status bar with "Body", "Cookies", "Headers (7)", "Test Results (1/1)", and a timer icon. To the right of the timer is a red box containing "500 Internal Server Error". Below the status bar, the "Headers" tab is selected, showing a single header "Content-Type: application/json". The "Body" tab is selected, displaying a JSON object with two fields: "message" and "error". The "message" field contains the string "Error logging vitals", and the "error" field contains the string "input must be a 24 character hex string, 12 byte Uint8Array, or an integer".

```
6  });
7
Body Cookies Headers (7) Test Results (1/1) 500 Internal Server Error 8 ms 374 B
[{"message": "Error logging vitals", "error": "input must be a 24 character hex string, 12 byte Uint8Array, or an integer"}]
```

### 9.3.5 Treatment record – Positive Tests

- **Retrieving Treatment History:**

Validates 200 OK and a non-empty treatment array

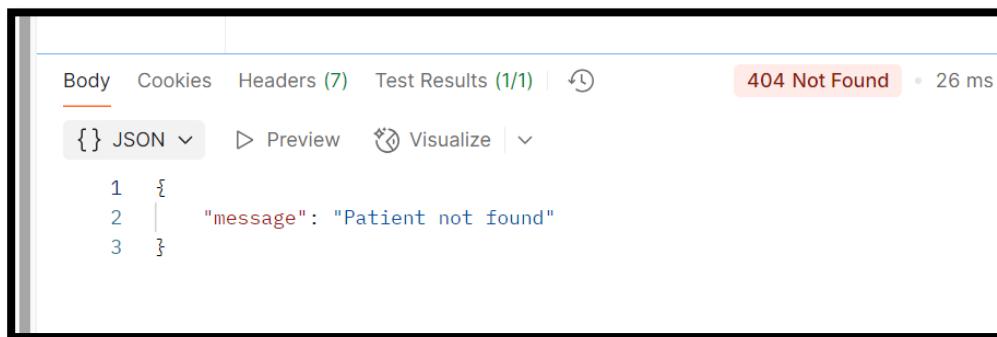
A screenshot of the Postman application interface. The URL in the address bar is "http://localhost:3001/patients/all". The "Headers" tab is selected, showing a "Content-Type: application/json" header. The "Body" tab is selected, displaying a JSON array with one element. This element is a patient object with fields like "\_id", "firstName", "lastName", "mobile", "email", "createdAt", and "patientId".

```
// Pre-request Script
pm.sendRequest({
  url: 'http://localhost:3000/auth/login',
  method: 'POST',
  header: {
    'Content-Type': 'application/json'
  },
  body: {
    mode: 'raw',
    raw: JSON.stringify({
      "username": "aa",
      "password": "aa"
    })
  }
})
[{"_id": "67abada98e7056a95b8599ee", "firstName": "aa", "lastName": "aa", "mobile": "1212", "email": "aa@aa.aaa", "createdAt": "2025-02-11T20:06:01.811Z", "patientId": "1012", "treatments": [{"name": "Treatment 1", "details": "Details for Treatment 1"}, {"name": "Treatment 2", "details": "Details for Treatment 2"}]}]
```

### 9.3.6 Treatment Record - Negative Tests

- **Invalid Patient ID:**

Invalid IDs return 404 Not Found

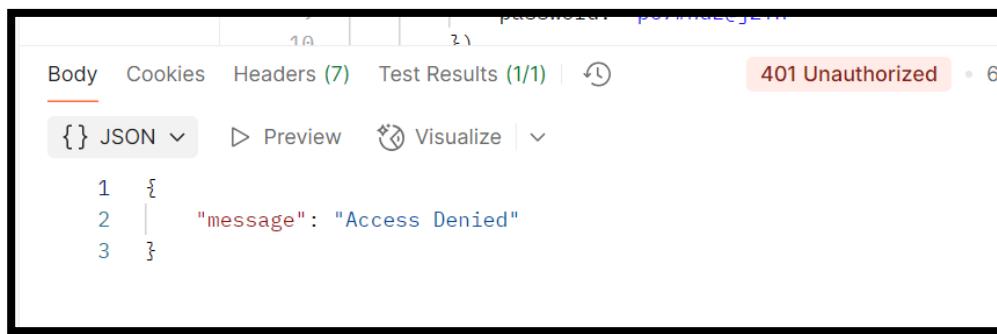


The screenshot shows a Postman test result for a failed request. The status bar at the top right indicates "404 Not Found" and a duration of "26 ms". The "Body" tab is selected, showing the following JSON response:

```
1 {  
2 |   "message": "Patient not found"  
3 }
```

- **Unauthorized Access:**

Missing tokens trigger 401 Unauthorized error



The screenshot shows a Postman test result for a failed request. The status bar at the top right indicates "401 Unauthorized" and a duration of "6 ms". The "Body" tab is selected, showing the following JSON response:

```
1 {  
2 |   "message": "Access Denied"  
3 }
```

## 9.4 Running collection in Postman:

- Open the collection link.
- Click ► Run > Run Collection.
- Configure iterations, delays, and data file if required

The screenshot shows the 'Run order' section on the left with 15 items selected, each representing a POST request to 'Auth Registration'. To the right, under 'Functional', there are sections for 'Choose how to run your collection' (with 'Run manually' selected), 'Run configuration' (Iterations set to 1, Delay set to 0 ms), and 'Data file' (Select File). At the bottom right is a large orange button labeled 'Run Continuous Testing'.

## Results:

- **Passed Tests:**

API behaves as expected, that is correct status codes, response structure and validation

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	7s 471ms	43	32 ms

All Tests Passed (43) Failed (0) Skipped (0)

## 9.5 Further Implementation

Further implementation of testing will be done using Mocha and Chai as Mocha provides a robust testing framework, while Chai enhances the readability and expressiveness of test assertions.

## 10 Resource Links

- **GitHub repository**

<https://github.com/faisalmc/TCH-PIS>

- **MongoDB**

**URI** - mongodb+srv://fmc4000:PcMk4CYYjrNQ9LW8@patient-info-system.qj5ku.mongodb.net/patientdb?retryWrites=true&w=majority

**JWT secret** - e5b8a6d8f92c

- **Service APIs – Postman**

<https://f21ao-group.postman.co/workspace/TCH-PIS~f0968d44-b64a-4c37-b89c-4ac3cccd9974/collection/41575727-3be3f492-99d6-46af-ab3d-8d4422def8fc>

- **Continuous Testing – Postman**

<https://f21ao-group.postman.co/workspace/TCH-PIS~f0968d44-b64a-4c37-b89c-4ac3cccd9974/collection/41575727-f92d10b6-d533-485f-be82-30a302bdf3d4>