Complaint / Helpdesk Management System

Software Engineering Course Work

B.Tech – Computer Science Engineering, MNIT Jaipur

Group - 27

Kavyansh Bagdi	2023UCP1701
Nikhil Kumawat	2023UCP1713
Ankit Sharma	2023UCP1697

Github Link : https://github.com/Kavyansh-Bagdi/helpdesk-management/

Youtube Video Link : https://youtu.be/mcom59GdACQ

1. Introduction

This Software Requirements Specification (SRS) outlines the requirements for the Complaint / Helpdesk Management System. The system enables users to register, authenticate, create complaint tickets, upload images, comment on tickets, and manage ticket workflows. It includes role-based access control across three roles: **customer**, **agent**, and **admin**.

2. Objective and Scope

2.1 Objective

The system aims to:

- Provide secure and reliable complaint management.
- Allow customers to create tickets with text and images.
- Enable agents and admins to manage and update ticket statuses.
- Facilitate communication between users through comments.
- Provide image retrieval functionality for uploaded files.
- Ensure proper access control and security using JWT authentication.

2.2 Scope

The system includes:

- User registration & login (JWT-based authentication).
- Role-based access control: **customer**, **agent**, **admin**.
- Ticket creation, viewing, image upload, and status updates.
- Comment creation and viewing per ticket.
- Secure retrieval of stored images.
- Backend API services according to the OpenAPI specification.

3. Functional Requirements

3.1 Authentication

FR1.1 — User Registration

- Users shall be able to register using:
  - username
  - email
  - password
- Input validation shall follow the schema `UserRegister`.

FR1.2 — User Login

- Users shall be able to authenticate using:
  - email
  - password
- Successful login returns a **JWT token**, as defined in the API.
- Unauthenticated users cannot access protected endpoints.

3.2 User Management

FR2.1 — Role-Based Access

The system shall support three roles:

- **Customer**
  - Create tickets
  - View their own tickets
  - Add comments to their own tickets
- **Agent**
  - View all tickets
  - Update ticket status
  - Comment on any ticket
- **Admin**
  - Full access to all tickets and users
  - Same ticket privileges as agents

Unauthorized and forbidden responses are returned based on API rules (`401` and `403`).

3.3 Ticket Management

FR3.1 — Get All Tickets

- Endpoint: `GET /api/tickets`
- Accessible by:
  - **Agents**
  - **Admins**
  - **Customers** (restricted to their own tickets, enforced by backend logic)

FR3.2 — Create Ticket

- Endpoint: `POST /api/tickets`
- Customers can create a new ticket with:
  - title (required)
  - description (required)
  - optional image uploads (multipart/form-data)

Uploaded images are stored and referenced through `image_ids` .

FR3.3 — Get Ticket by ID

- Endpoint: `GET /api/tickets/{ticket_id}`
- Users can view:
  - Their own tickets (customers)
  - All tickets (agents, admins)

Forbidden access is returned when customer tries to access others' tickets.

FR3.4 — Update Ticket (Status)

- Endpoint: `PUT /api/tickets/{ticket_id}`
- Only **agents** and **admins** may update a ticket's status.
- Allowed status values:
  - `open`
  - `in_progress`
  - `closed`

3.4 Comment Management

FR4.1 — Get Comments for a Ticket

- Endpoint: `GET /api/tickets/{ticket_id}/comments`
- Any authenticated user with permission to view the ticket may view comments.

FR4.2 — Add Comment to Ticket

- Endpoint: `POST /api/tickets/{ticket_id}/comments`
- Request body: `{ "text": "comment message" }`
- All roles (customer, agent, admin) may comment on tickets they have access to.

3.5 Image Retrieval

FR5.1 — Get Image by ID

- Endpoint: `GET /api/images/{image_id}`
- Returns binary JPEG data.
- If the image does not exist, the API returns `404`.

4. Non-Functional Requirements

4.1 Performance

- **NFR1.1 Response Time**
  All API requests should respond within **2 seconds** under normal load.
- **NFR1.2 Scalability**
  The backend shall scale to support growth in:
  - User base
  - Ticket volume
  - Image uploads

4.2 Security

- **NFR2.1 Authentication**
  All protected endpoints require JWT (`BearerAuth`).
- **NFR2.2 Password Protection**
  Passwords must be securely hashed and never stored in plain text.
- **NFR2.3 Authorization**
  Access restrictions:
  - Unauthorized → `401`
  - Forbidden → `403`
- **NFR2.4 Data Protection**
  Uploaded images must not be publicly accessible without permission.

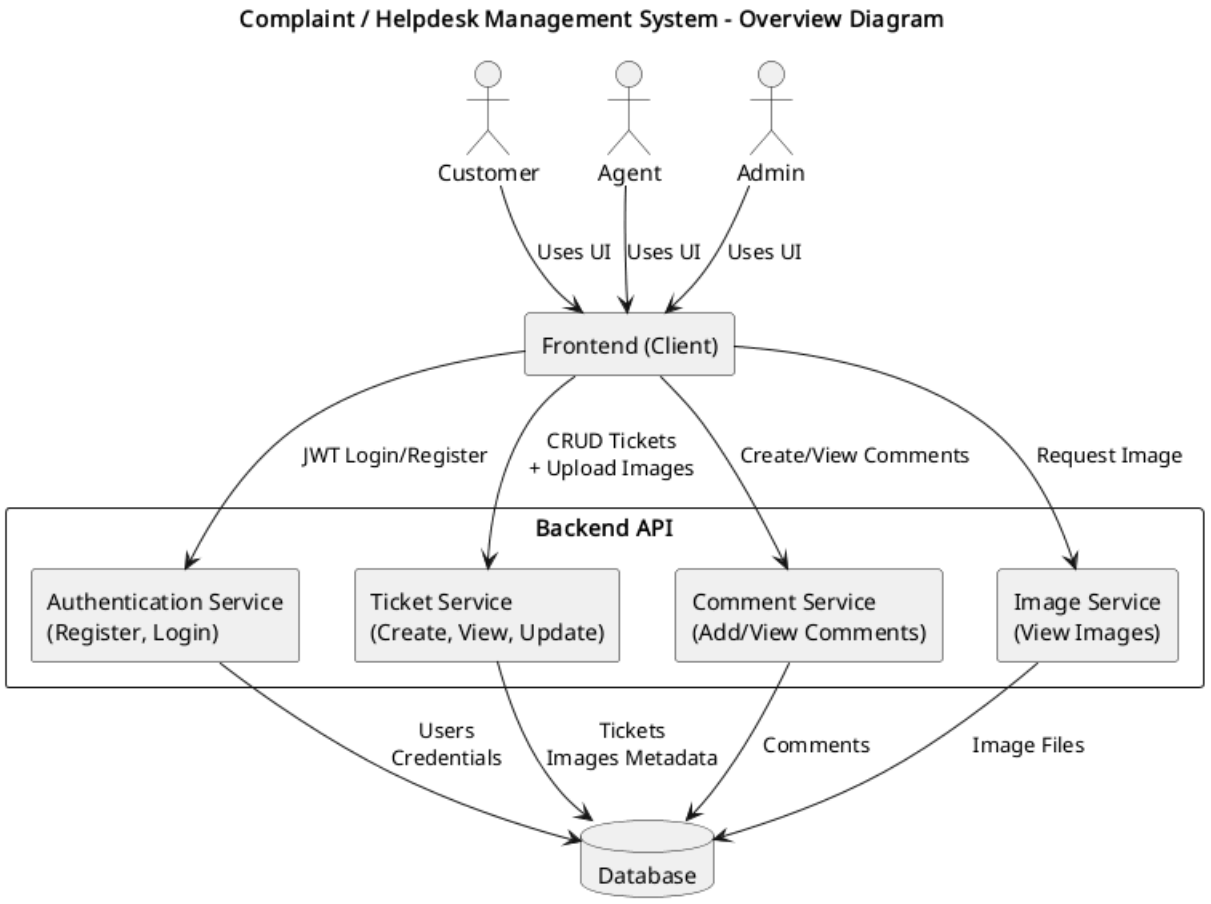
4.3 Usability

- **NFR3.1 UI Simplicity**
  The frontend (outside API scope) must present an intuitive interface for:
  - Submitting tickets
  - Uploading images
  - Viewing comments and statuses
- **NFR3.2 Accessibility**
  Interfaces should follow basic accessibility guidelines.

4.4 Maintainability

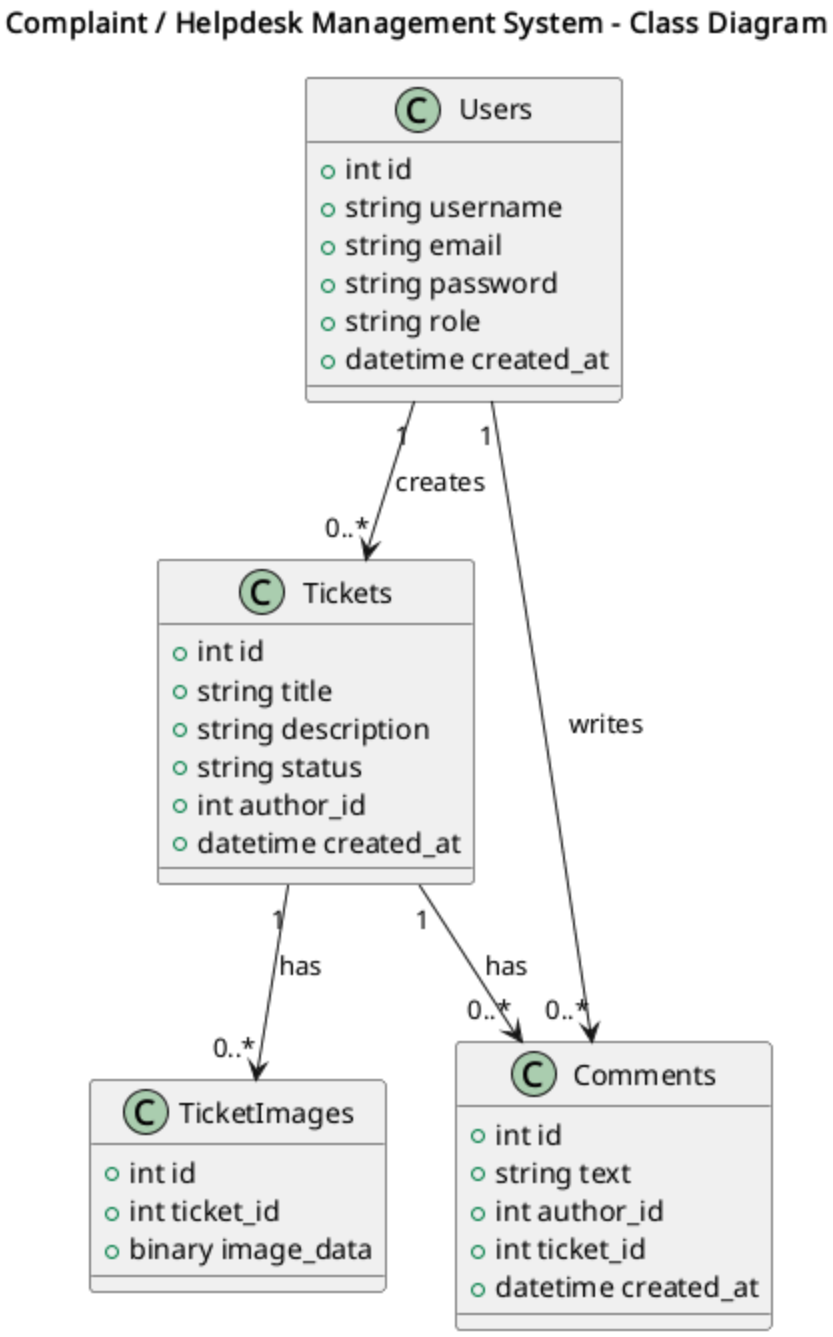
- **NFR4.1 Code Quality**
  The system codebase should follow clean architecture and be well-documented.
- **NFR4.2 Testability**
  API endpoints must be structured to support unit and integration testing.

5. System Overview Diagram

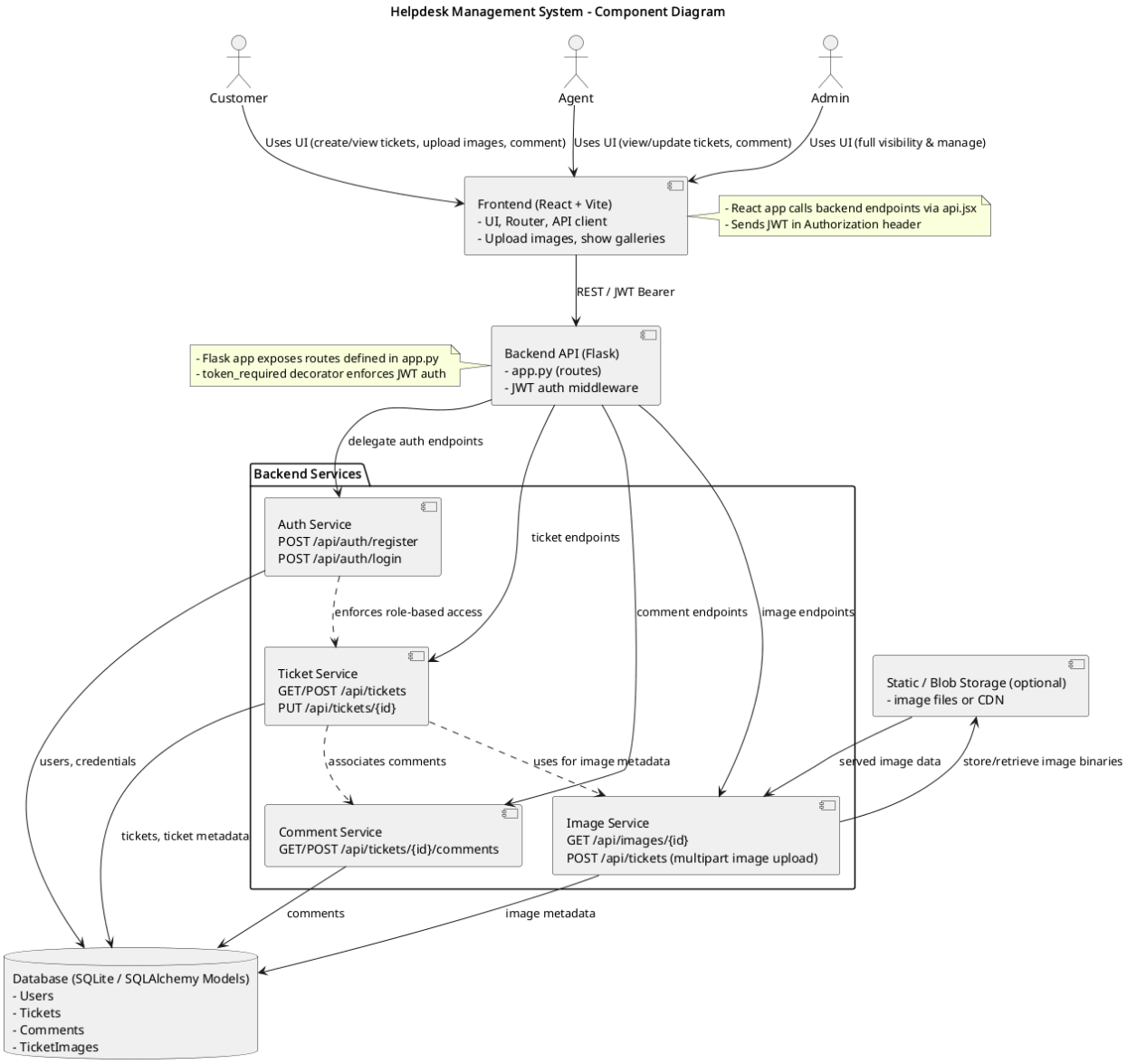


Design Documentation

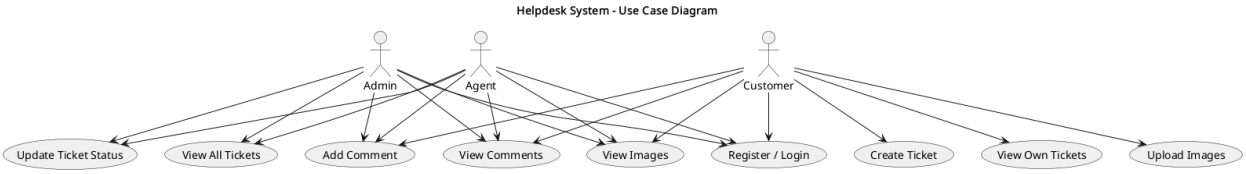
1. Class Diagram



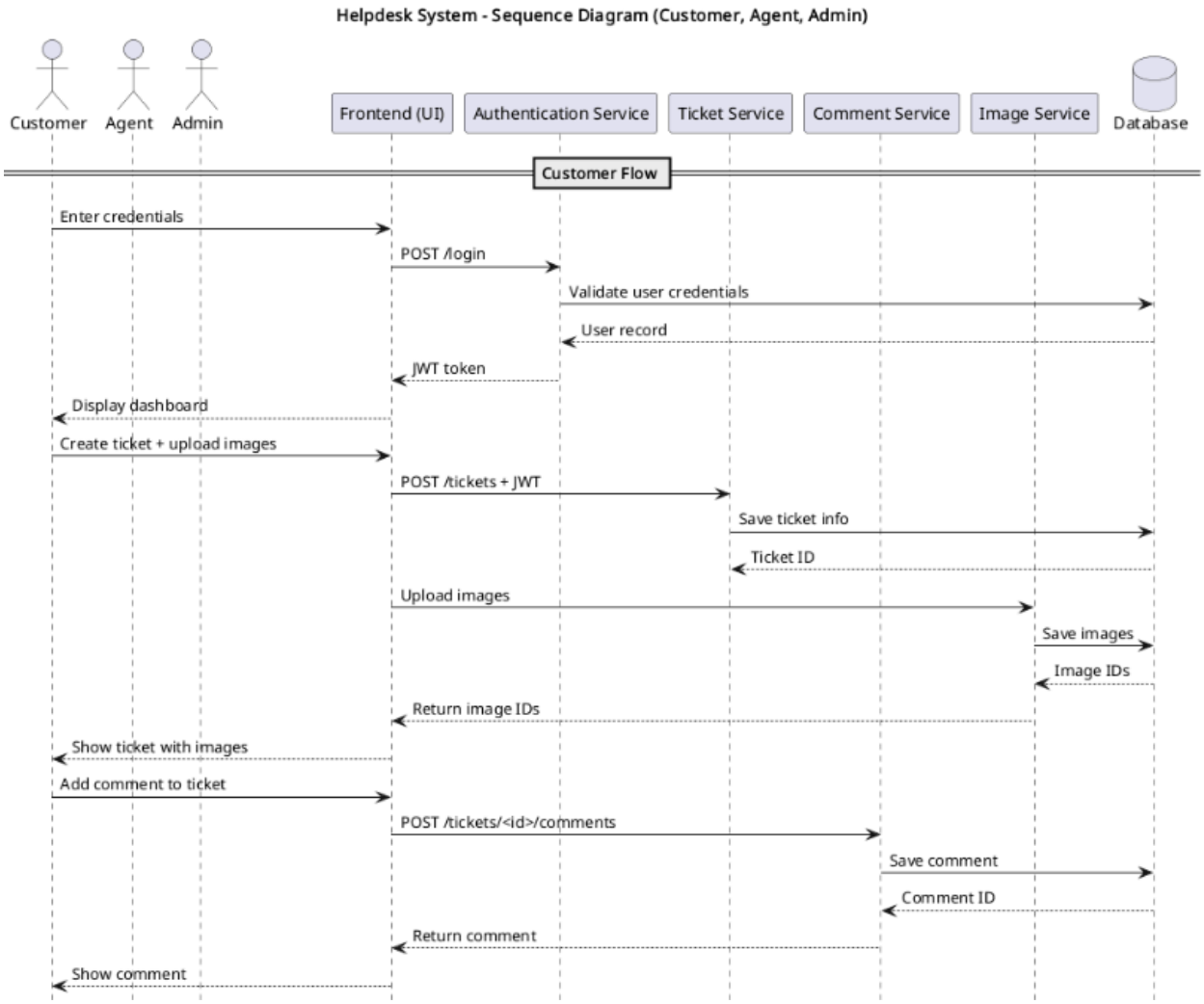
2. Component Diagram

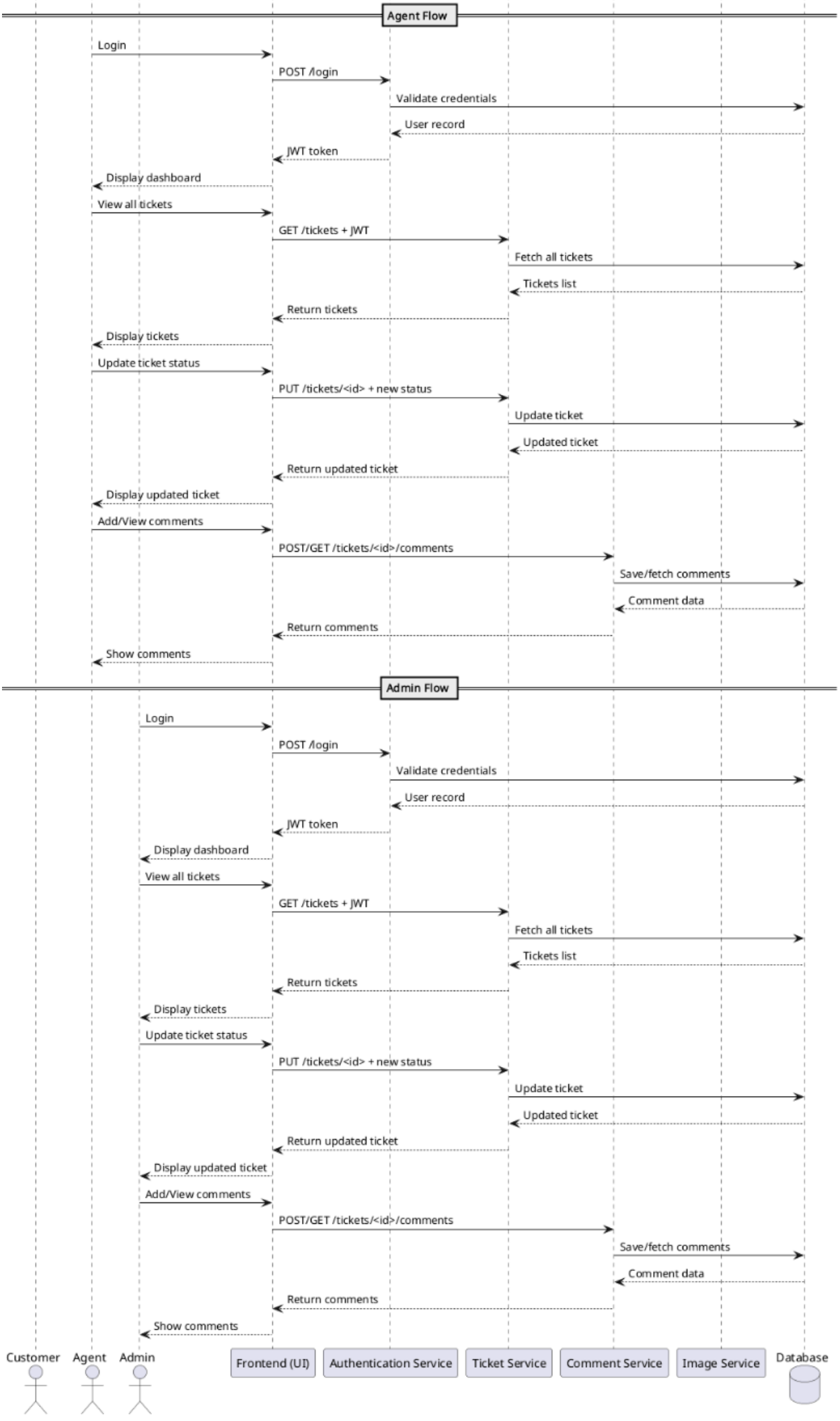


3. Use Case Diagram



4. Sequence Diagram




Agent Flow
Login
POST /login
Validate credentials
User record
JWT token
Display dashboard
View all tickets
GET /tickets + JWT
Fetch all tickets
Tickets list
Return tickets
Display tickets
Update ticket status
PUT /tickets/<id> + new status
Update ticket
Updated ticket
Return updated ticket
Display updated ticket
Add/View comments
POST/GET /tickets/<id>/comments
Save/fetch comments
Comment data
Return comments
Show comments
Admin Flow
Login
POST /login
Validate credentials
User record
JWT token
Display dashboard
View all tickets
GET /tickets + JWT
Fetch all tickets
Tickets list
Return tickets
Display tickets
Update ticket status
PUT /tickets/<id> + new status
Update ticket
Updated ticket
Return updated ticket
Display updated ticket
Add/View comments
POST/GET /tickets/<id>/comments
Save/fetch comments
Comment data
Return comments
Show comments
Customer
Agent
Admin
Frontend (UI)
Authentication Service
Ticket Service
Comment Service
Image Service
Database

1. Tech Stack

This project utilizes a modern full-stack architecture, separating the backend API from the frontend user interface.

1.1 Backend

- **Framework:** Flask (Python)
- **Database ORM:** Flask-SQLAlchemy
- **Database Migrations:** Flask-Migrate
- **Authentication:** PyJWT (JSON Web Tokens)
- **Environment Variables:** python-dotenv
- **Password Hashing:** bcrypt

1.2 Frontend

- **Framework:** React (JavaScript)
- **Build Tool:** Vite
- **Routing:** React Router
- **Styling:** Standard CSS

2. Architecture

The application follows a client-server architecture:

- **Frontend (Client):** A React single-page application (SPA) that consumes the backend API. It handles user interactions, data presentation, and routing.
- **Backend (Server):** A Flask API that provides RESTful endpoints for user authentication, ticket management, and comment management. It interacts with the database and handles business logic.

Communication between the frontend and backend occurs via HTTP requests, with data exchanged in JSON format. JWTs are used for secure authentication and authorization.

3. Key Module Descriptions

3.1 Backend Modules

- **`app.py`:** The main Flask application file, responsible for setting up the Flask app, registering blueprints, and defining API routes.
- **`models.py`:** Defines the SQLAlchemy ORM models for `Users`, `Tickets`, and `Comments` tables, representing the database schema.
- **`setup_db.py`:** Script for initializing the database and applying migrations.
- **`insert_dummy_data.py`:** Script to populate the database with sample data for testing and development.
- **`openapi.yml`:** OpenAPI specification for the backend API endpoints.

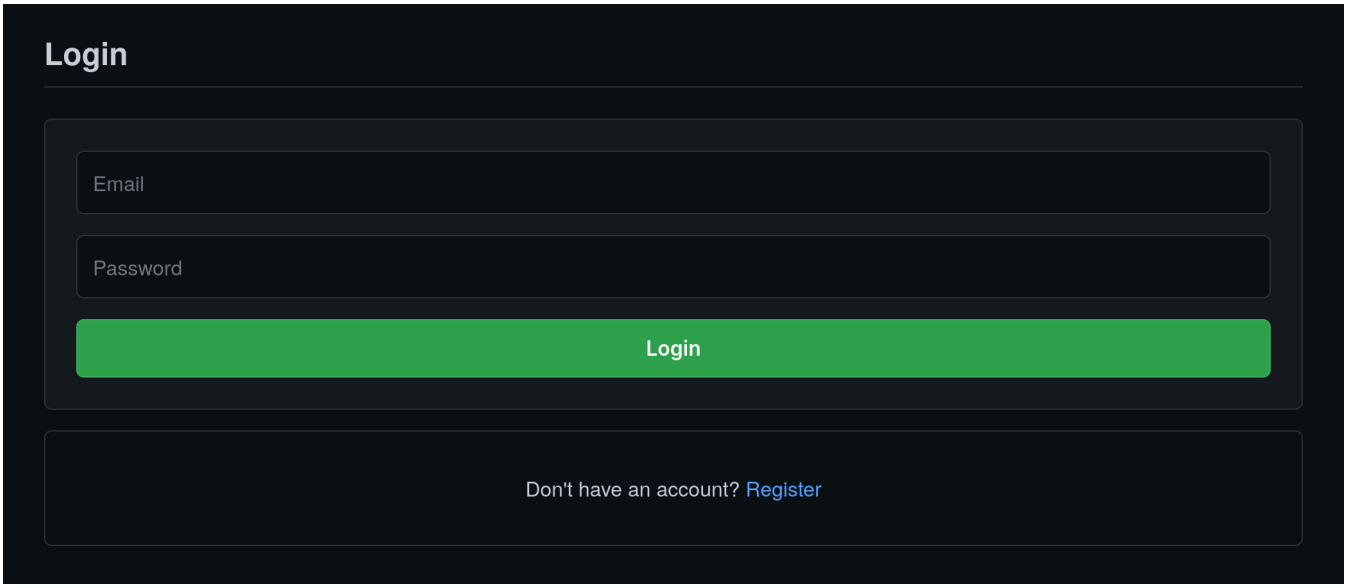
3.2 Frontend Modules

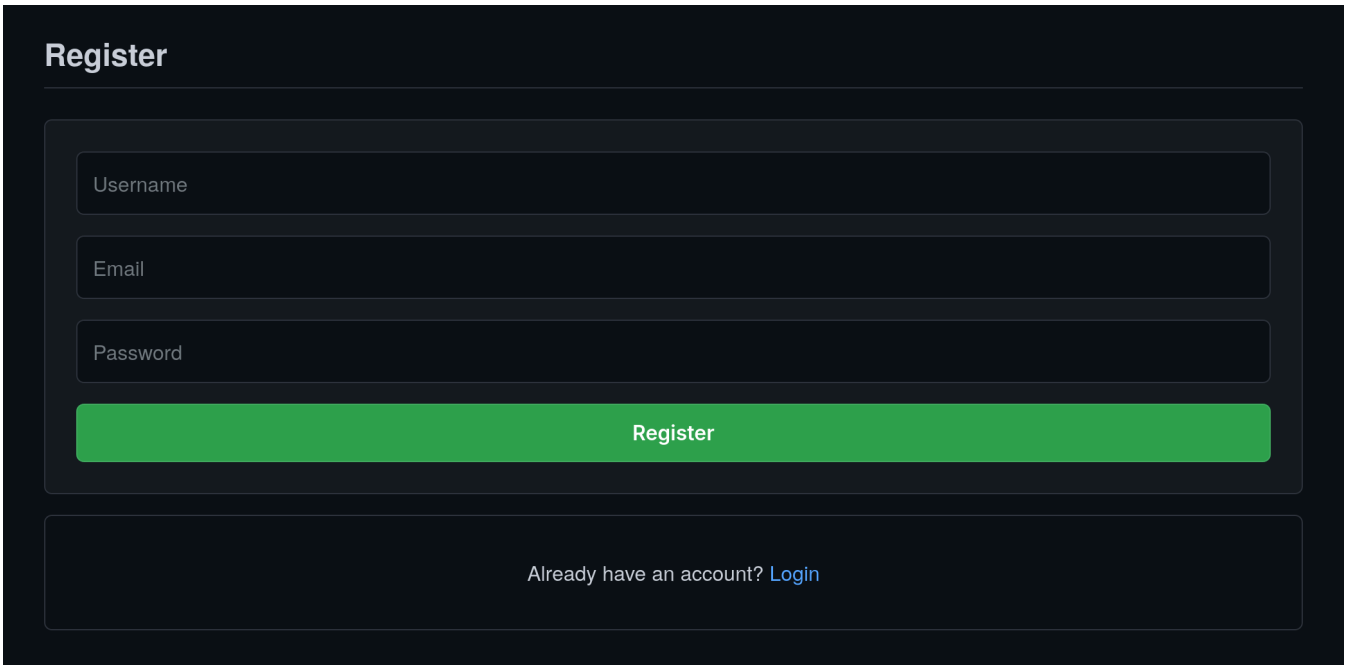
- **`main.jsx`:** The entry point of the React application, responsible for rendering the root component.
- **`router.jsx`:** Defines the application's routing using React Router, mapping URLs to specific views.
- **`api.jsx`:** Contains functions for interacting with the backend API, abstracting HTTP requests.
- **`components/Header.jsx`:** Reusable React component for the application's header/navigation.
- **`views/` (e.g., `Home.jsx`, `Login.jsx`, `Tickets.jsx`, `TicketDetail.jsx`):** React components representing different pages or views of the application.

4. Screenshots of Core Modules

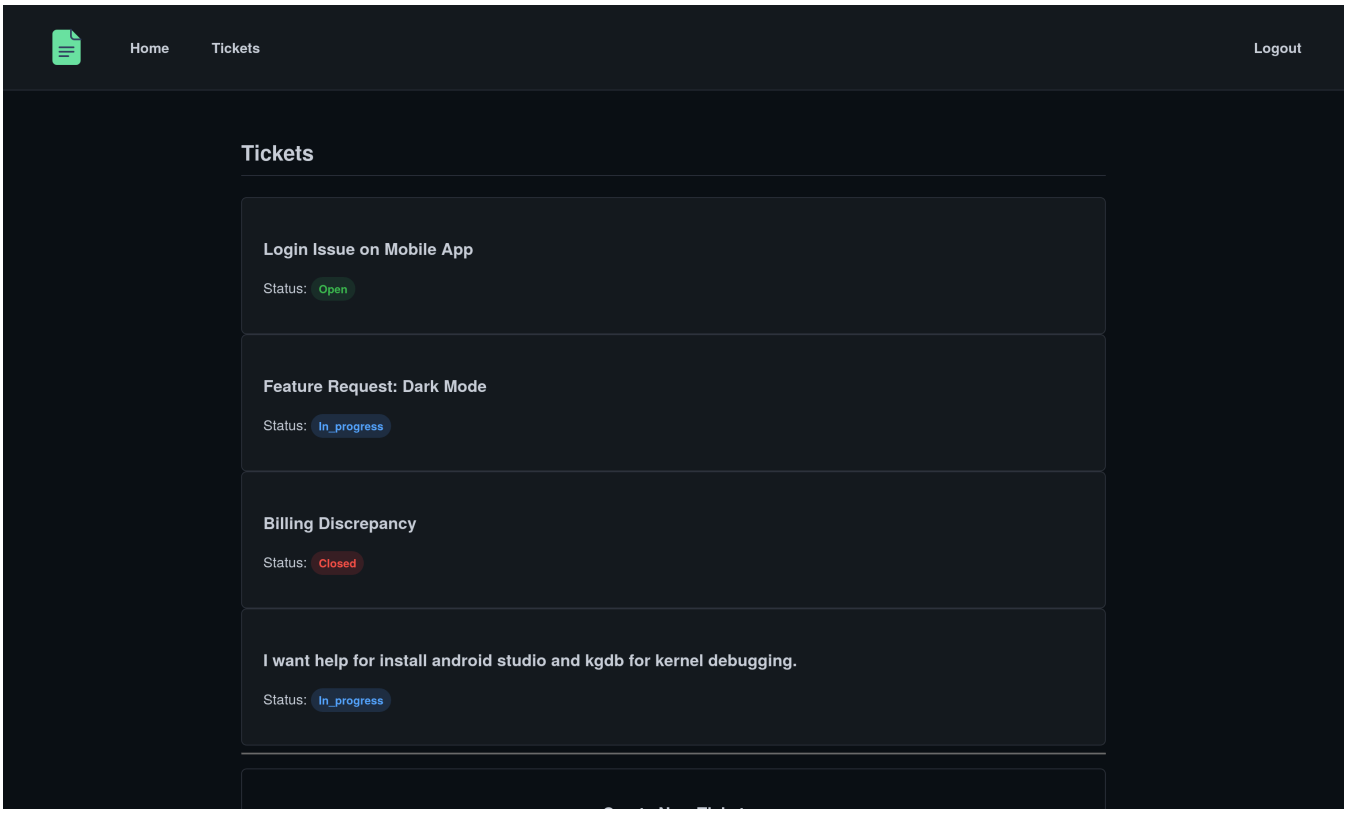
(Screenshots will be added here to visually demonstrate the core functionalities of the application, such as the login page, tickets dashboard, and ticket detail view.)

4.1 Login & Registration Page

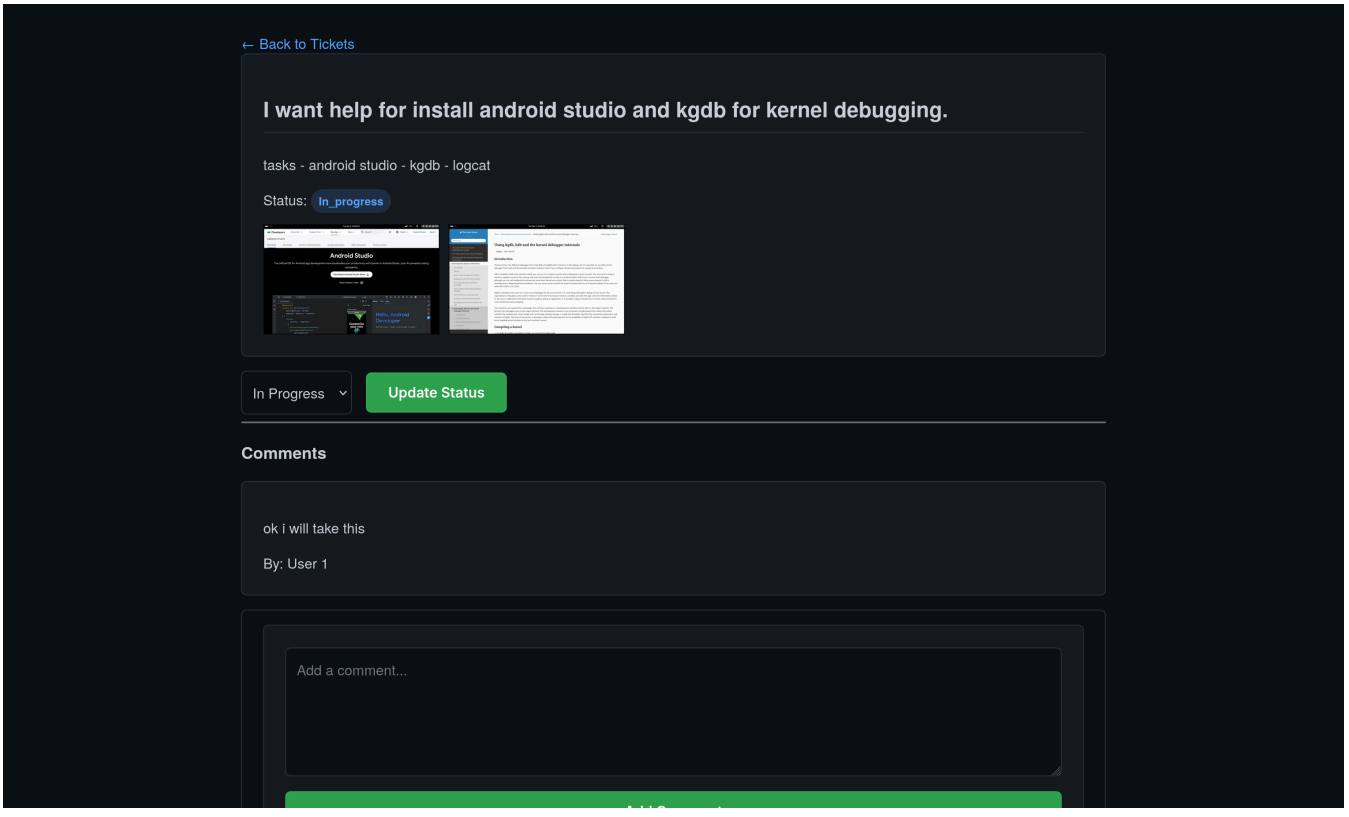




4.2 Tickets Dashboard



4.3 Ticket Detail View



5. Database Schema

Users Table

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
```

```
    role TEXT NOT NULL CHECK(role IN ('customer', 'agent', 'admin'))
);
```

Tickets Table

```
CREATE TABLE tickets (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    description TEXT NOT NULL,
    status TEXT NOT NULL CHECK(status IN ('open', 'in_progress', 'closed')),
    author_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (author_id) REFERENCES users (id)
);
```

Comments Table

```
CREATE TABLE comments (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    text TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    ticket_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (author_id) REFERENCES users (id),
    FOREIGN KEY (ticket_id) REFERENCES tickets (id)
);
```

Images Table

```
CREATE TABLE ticket_images (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    ticket_id INTEGER NOT NULL,
    image_data BLOB NOT NULL,
    FOREIGN KEY (ticket_id) REFERENCES tickets (id)
);
```

Unit Test

Authentication & Authorization

```
def test_register(client, init_database):
    response = client.post('/api/auth/register',
                           data=json.dumps(dict(
                               username='testuser',
                               email='test@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 201
    data = json.loads(response.data)
    assert data['username'] == 'testuser'
    assert data['email'] == 'test@example.com'
    assert 'id' in data

def test_login(client, init_database):
    client.post('/api/auth/register',
                data=json.dumps(dict(
                    username='testuser',
                    email='test@example.com',
                    password='password'
                )),
                content_type='application/json')

    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='test@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 200
    data = json.loads(response.data)
    assert 'token' in data

def test_login_invalid_credentials(client, init_database):
    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='wrong@example.com',
                               password='wrongpassword'
                           )),
                           content_type='application/json')
    assert response.status_code == 401
    data = json.loads(response.data)
    assert data['message'] == 'Invalid credentials'
```

Comment & Image

```
def test_register(client, init_database):
    response = client.post('/api/auth/register',
                           data=json.dumps(dict(
                               username='testuser',
                               email='test@example.com',
                               password='password'
                           )),
```

```
                            content_type='application/json')
    assert response.status_code == 201
    data = json.loads(response.data)
    assert data['username'] == 'testuser'
    assert data['email'] == 'test@example.com'
    assert 'id' in data

def test_login(client, init_database):
    client.post('/api/auth/register',
                data=json.dumps(dict(
                    username='testuser',
                    email='test@example.com',
                    password='password'
                )),
                content_type='application/json')

    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='test@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 200
    data = json.loads(response.data)
    assert 'token' in data

def test_login_invalid_credentials(client, init_database):
    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='wrong@example.com',
                               password='wrongpassword'
                           )),
                           content_type='application/json')
    assert response.status_code == 401
    data = json.loads(response.data)
    assert data['message'] == 'Invalid credentials'
```

Ticket

```
def get_auth_token(client):
    client.post('/api/auth/register',
                data=json.dumps(dict(
                    username='testuser',
                    email='test@example.com',
                    password='password'
                )),
                content_type='application/json')
    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='test@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    return json.loads(response.data)['token']

def test_create_ticket(client, init_database):
    token = get_auth_token(client)
    response = client.post('/api/tickets',
                           headers={'Authorization': f'Bearer {token}'},
                           data={
                               'title': 'Test Ticket',
```

```
                                'description': 'This is a test ticket.',
                                'images': (io.BytesIO(b"some initial text data"), 'test.jpg')
                            },
                            content_type='multipart/form-data')
    assert response.status_code == 201
    data = json.loads(response.data)
    assert data['title'] == 'Test Ticket'
    assert data['description'] == 'This is a test ticket.'
    assert 'id' in data
    assert 'image_ids' in data
    assert len(data['image_ids']) == 1

def test_get_tickets(client, init_database):
    token = get_auth_token(client)
    client.post('/api/tickets',
                headers={'Authorization': f'Bearer {token}'},
                data={
                    'title': 'Test Ticket',
                    'description': 'This is a test ticket.'
                },
                content_type='multipart/form-data')

    response = client.get('/api/tickets', headers={'Authorization': f'Bearer {token}'})
    assert response.status_code == 200
    data = json.loads(response.data)
    assert isinstance(data, list)
    assert len(data) == 1
    assert data[0]['title'] == 'Test Ticket'

def test_get_ticket(client, init_database):
    token = get_auth_token(client)
    create_response = client.post('/api/tickets',
                                  headers={'Authorization': f'Bearer {token}'},
                                  data={
                                      'title': 'Test Ticket',
                                      'description': 'This is a test ticket.'
                                  },
                                  content_type='multipart/form-data')
    ticket_id = json.loads(create_response.data)['id']

    response = client.get(f'/api/tickets/{ticket_id}', headers={'Authorization': f'Bearer
{token}'})
    assert response.status_code == 200
    data = json.loads(response.data)
    assert data['title'] == 'Test Ticket'
    assert data['id'] == ticket_id

def test_update_ticket(client, init_database):
    client.post('/api/auth/register',
                data=json.dumps(dict(
                    username='adminuser',
                    email='admin@example.com',
                    password='password'
                )),
                content_type='application/json')

    with client.application.app_context():
        admin_user = Users.query.filter_by(email='admin@example.com').first()
        admin_user.role = 'admin'
        db.session.commit()

    response = client.post('/api/auth/login',
```

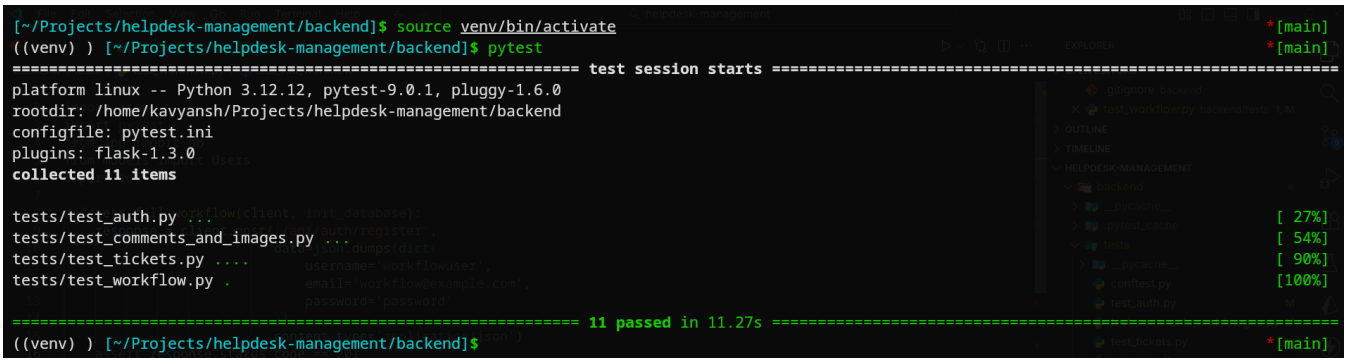
```
                            data=json.dumps(dict(
                                email='admin@example.com',
                                password='password'
                            )),
                            content_type='application/json')
    admin_token = json.loads(response.data)['token']

    user_token = get_auth_token(client)
    create_response = client.post('/api/tickets',
                                  headers={'Authorization': f'Bearer {user_token}'},
                                  data={
                                      'title': 'Test Ticket',
                                      'description': 'This is a test ticket.'
                                  },
                                  content_type='multipart/form-data')
    ticket_id = json.loads(create_response.data)['id']

    response = client.put(f'/api/tickets/{ticket_id}',
                          headers={'Authorization': f'Bearer {admin_token}'},
                          data=json.dumps(dict(status='closed')),
                          content_type='application/json')
    assert response.status_code == 200
    data = json.loads(response.data)
    assert data['status'] == 'closed'
```

System & Integration Testing

```
def test_full_workflow(client, init_database):
    response = client.post('/api/auth/register',
                           data=json.dumps(dict(
                               username='workflowuser',
                               email='workflow@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 201
    user_data = json.loads(response.data)
    user_id = user_data['id']

    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='workflow@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 200
    user_token = json.loads(response.data)['token']

    response = client.post('/api/tickets',
                           headers={'Authorization': f'Bearer {user_token}'},
                           data={
                               'title': 'Workflow Ticket',
                               'description': 'This is a ticket created during a workflow
test.',
                               'images': (io.BytesIO(b'workflow image'), 'workflow.jpg')
                           },
                           content_type='multipart/form-data')
    assert response.status_code == 201
    ticket_data = json.loads(response.data)
    ticket_id = ticket_data['id']
    assert ticket_data['title'] == 'Workflow Ticket'
```

```
    assert len(ticket_data['image_ids']) == 1
    image_id = ticket_data['image_ids'][0]

    response = client.post(f'/api/tickets/{ticket_id}/comments',
                           headers={'Authorization': f'Bearer {user_token}'},
                           data=json.dumps(dict(text='This is a workflow comment.')),
                           content_type='application/json')
    assert response.status_code == 201
    comment_data = json.loads(response.data)
    assert comment_data['text'] == 'This is a workflow comment.'

    response = client.post('/api/auth/register',
                           data=json.dumps(dict(
                               username='workflowadmin',
                               email='workflowadmin@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 201

    with client.application.app_context():
        admin_user = Users.query.filter_by(email='workflowadmin@example.com').first()
        admin_user.role = 'admin'
        db.session.commit()

    response = client.post('/api/auth/login',
                           data=json.dumps(dict(
                               email='workflowadmin@example.com',
                               password='password'
                           )),
                           content_type='application/json')
    assert response.status_code == 200
    admin_token = json.loads(response.data)['token']

    response = client.get(f'/api/tickets/{ticket_id}', headers={'Authorization': f'Bearer
{admin_token}'})
    assert response.status_code == 200
    assert json.loads(response.data)['title'] == 'Workflow Ticket'

    response = client.put(f'/api/tickets/{ticket_id}',
                          headers={'Authorization': f'Bearer {admin_token}'},
                          data=json.dumps(dict(status='in_progress')),
                          content_type='application/json')
    assert response.status_code == 200
    assert json.loads(response.data)['status'] == 'in_progress'

    response = client.get(f'/api/tickets/{ticket_id}', headers={'Authorization': f'Bearer
{user_token}'})
    assert response.status_code == 200
    assert json.loads(response.data)['status'] == 'in_progress'

    response = client.get(f'/api/tickets/{ticket_id}/comments', headers={'Authorization':
f'Bearer {admin_token}'})
    assert response.status_code == 200
    assert len(json.loads(response.data)) == 1

    response = client.get(f'/api/images/{image_id}')
    assert response.status_code == 200
    assert response.data == b'workflow image'
```

Output


[~/Projects/helpdesk-management/backend]$ source venv/bin/activate *[main]
((venv)) [~/Projects/helpdesk-management/backend]$ pytest *[main]
test session starts
platform linux -- Python 3.12.12, pytest-9.0.1, pluggy-1.6.0
rootdir: /home/kavyansh/Projects/helpdesk-management/backend
configfile: pytest.ini
plugins: flask-1.3.0
collected 11 items
tests/test_auth.py ...
tests/test_comments_and_images.py ...
tests/test_tickets.py ....
tests/test_workflow.py .
[27%]
[54%]
[90%]
[100%]
11 passed in 11.27s
((venv)) [~/Projects/helpdesk-management/backend]$ *[main]

1. Introduction

This manual provides a guide for end-users on how to set up and use the Complaint / Helpdesk Management System. It covers the steps to get the application running, how to use its core features, and discusses potential future enhancements.

2. Execution Steps

To run the application, you need to start both the backend and frontend servers.

2.1 Prerequisites

- Python 3.x
- Node.js and npm
- A cloned copy of the project repository.

2.2 Running the Backend Server

1. Open a terminal and navigate to the `backend` directory of the project.

```
cd path/to/Complaint / Helpdesk-management/backend
```

2. Activate the Python virtual environment.

```
source .venv/bin/activate
```

3. Start the Flask server.

```
flask run
```

4. The backend API will now be running at `http://127.0.0.1:5000`. Keep this terminal window open.

2.3 Running the Frontend Application

1. Open a **new** terminal window and navigate to the `frontend` directory.

```
cd path/to/Complaint / Helpdesk-management/frontend
```

2. Install the necessary Node.js packages (only required for the first time).

```
npm install
```

3. Start the frontend development server.

```
npm run dev
```

4. The application will be accessible in your web browser at `http://localhost:5173`.

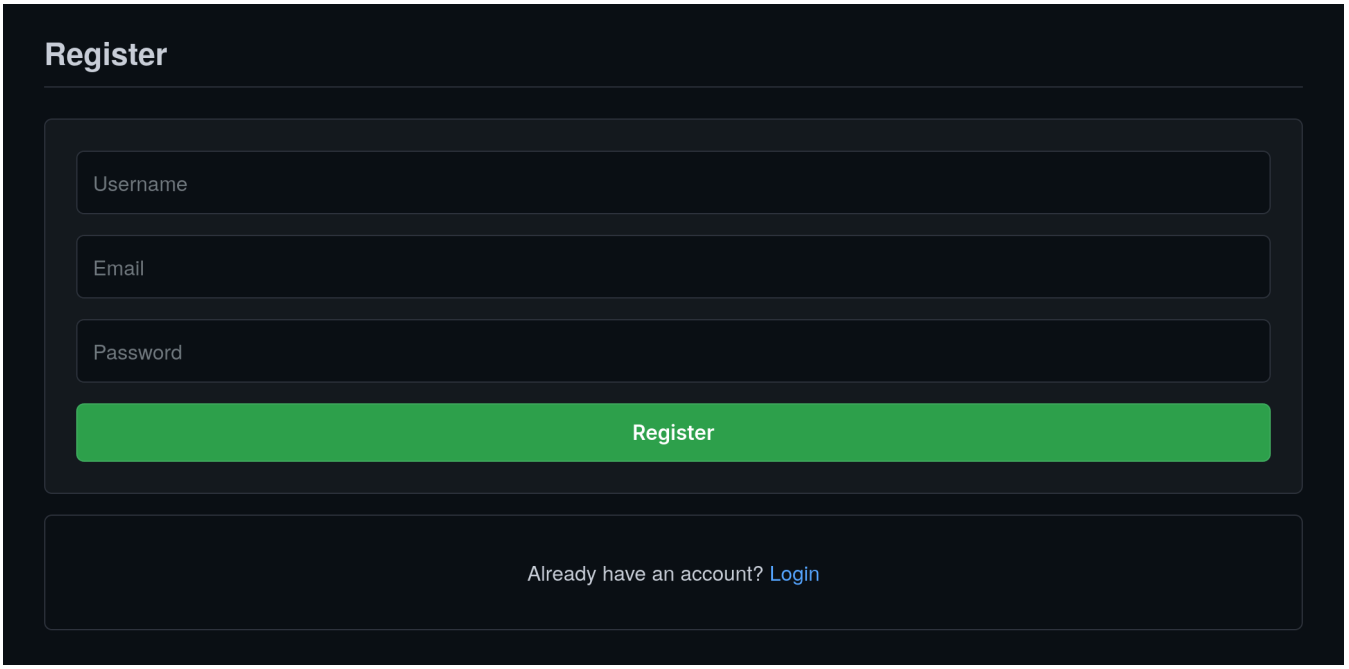
3. Using the Application: A Step-by-Step Guide

3.1 Registration

1. Navigate to the application URL (`http://localhost:5173`).

2. Click on the "Register" link in the navigation bar.
3. Fill in the registration form with your desired username, email, and password.
4. Click the "Register" button to create your account.

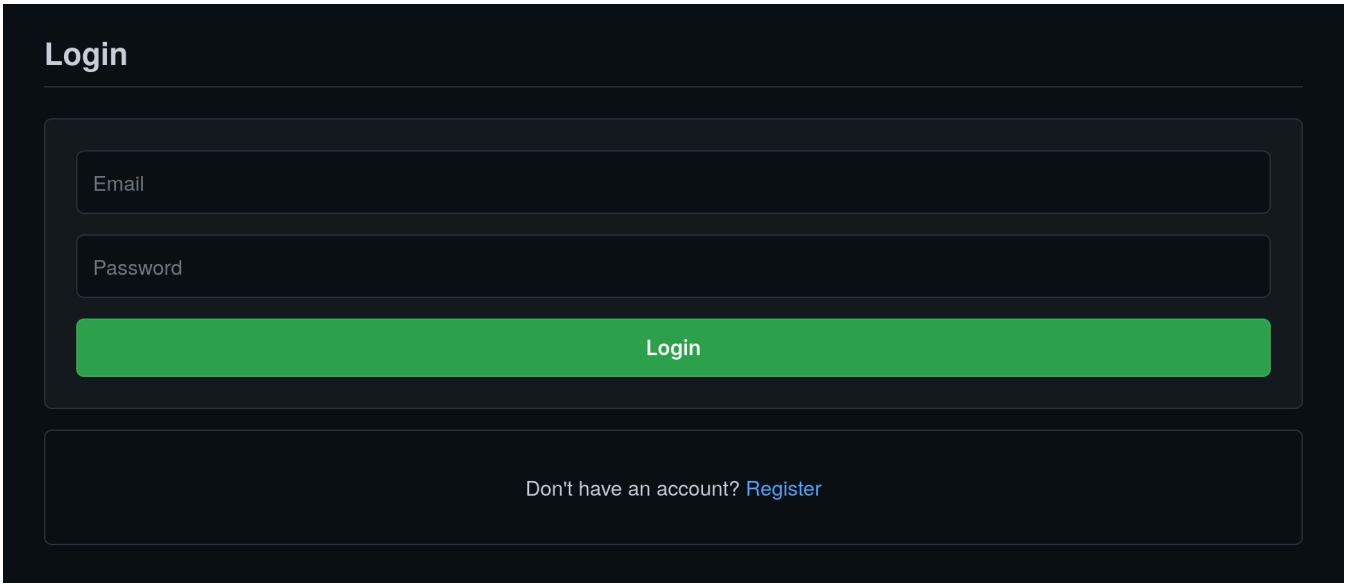
(Screenshot of the registration page)



3.2 Login

1. After registering, or if you already have an account, go to the "Login" page.
2. Enter your username and password.
3. Click the "Login" button to access your dashboard.

(Screenshot of the login page)

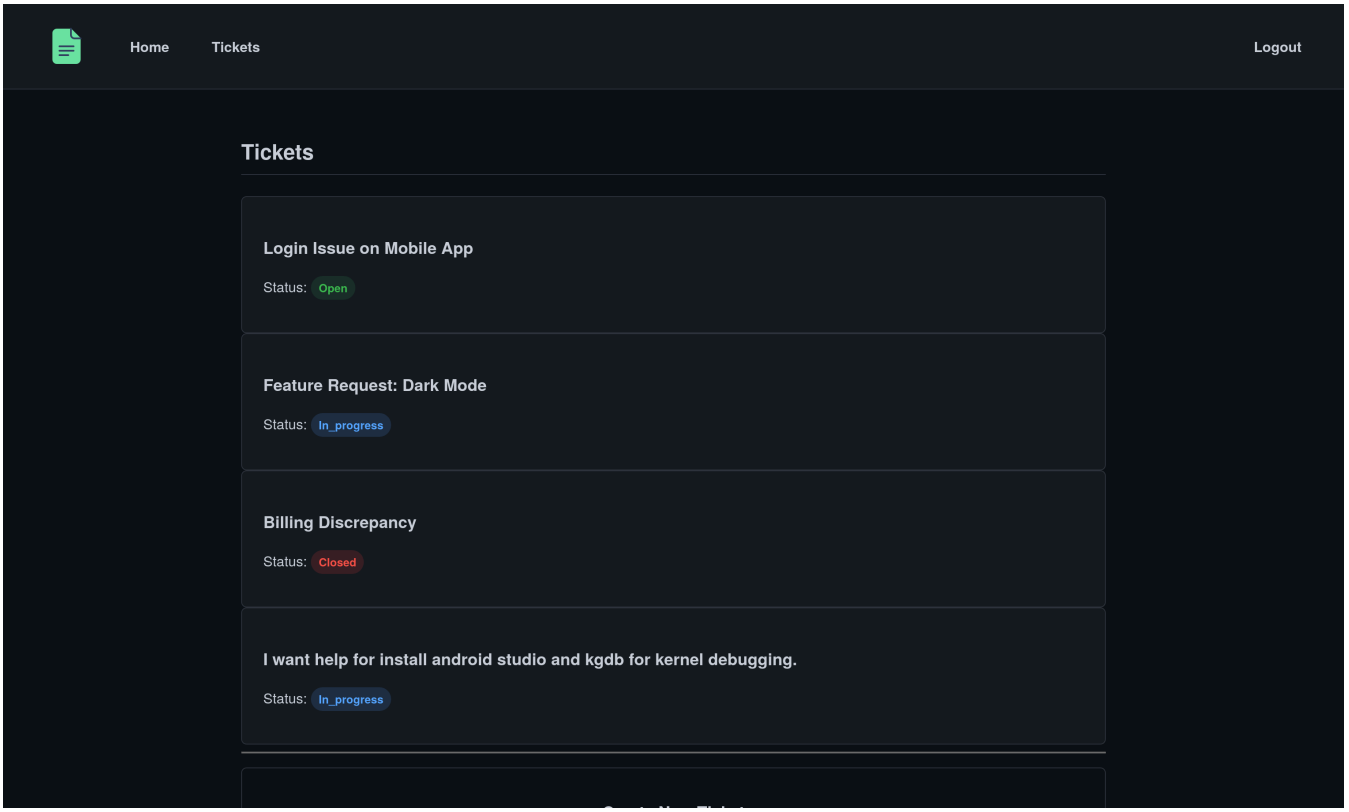


3.3 Viewing the Tickets Dashboard

1. Upon logging in, you will be redirected to the tickets dashboard.
2. This page displays a list of tickets.

- **Customers** will see a list of tickets they have created.
- **Agents and Admins** will see all tickets in the system.

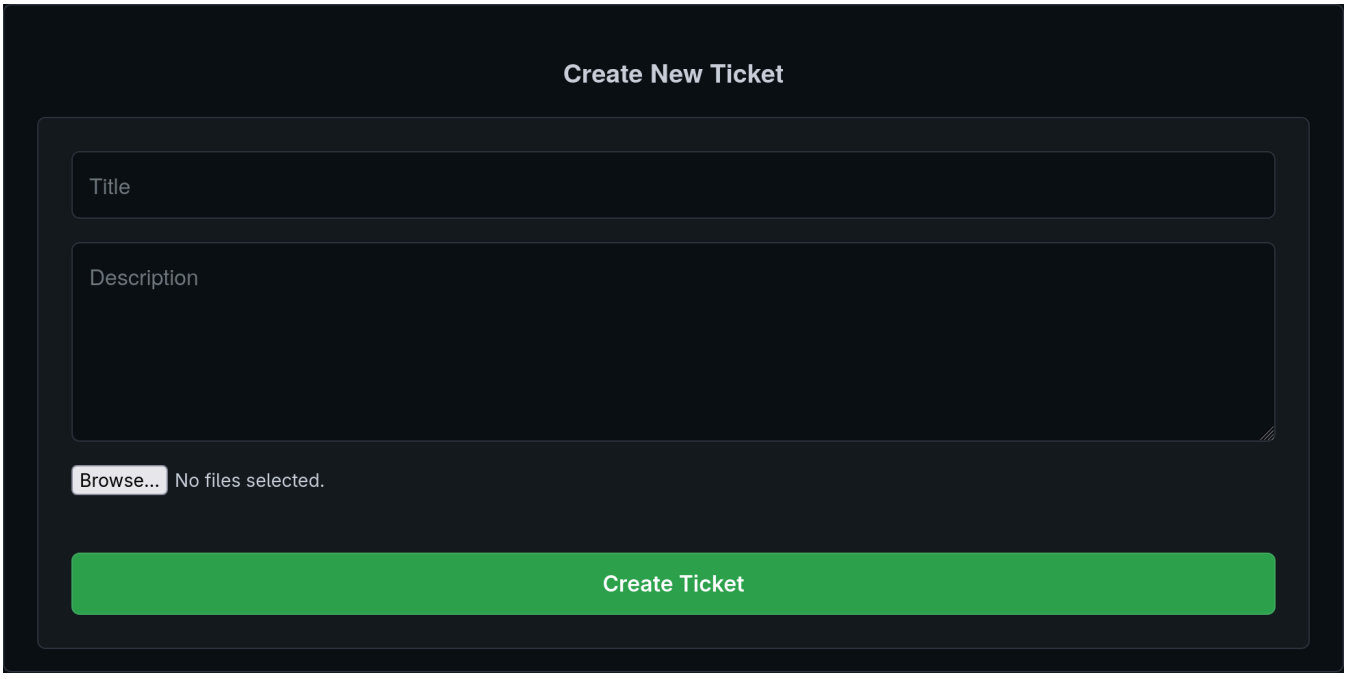
(Screenshot of the tickets dashboard)



3.4 Creating a New Ticket

1. From the dashboard, click on the "Create New Ticket" button.
2. You will be taken to a form where you can enter the ticket's title and a detailed description of the issue.
3. Click "Submit" to create the ticket. You will be redirected back to your dashboard where you can see the newly created ticket.

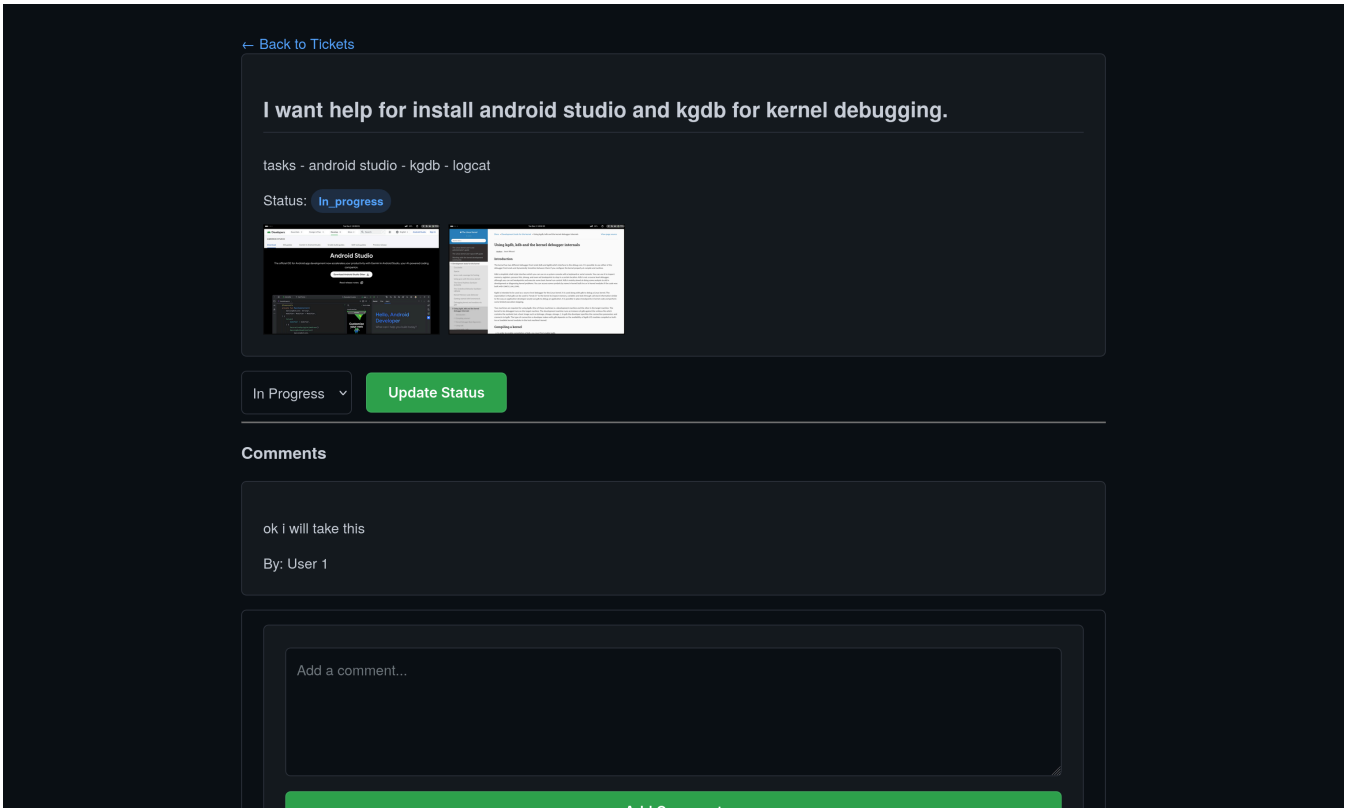
(Screenshot of the create ticket page)



3.5 Viewing a Ticket and Adding Comments

1. From the dashboard, click on the title of any ticket to view its details.
2. The ticket detail page shows the full description, status, and a history of comments
3. To add a new comment, type your message in the comment box at the bottom and click "Add Comment".

(Screenshot of the ticket detail view with comments)



4. Future Scope

The Complaint / Helpdesk Management System is a foundational application with significant potential for expansion. Future enhancements could include:

- **Enhanced User Roles and Permissions:** More granular permissions for different user roles (e.g., team leads, department managers).
- **Email Notifications:** Automatic email notifications to users when a ticket is created, updated, or commented on.
- **Ticket Assignment:** Functionality for admins or agents to assign specific tickets to individual agents.
- **Reporting and Analytics:** A dashboard for admins to view key metrics, such as ticket resolution times, agent performance, and common issue categories.
- **Full-Text Search:** A powerful search functionality to quickly find tickets based on keywords.
- **Knowledge Base Integration:** A section where users can find answers to frequently asked questions, potentially reducing the number of new tickets.