

# **Ticket System Integration with Discourse and Webhooks**

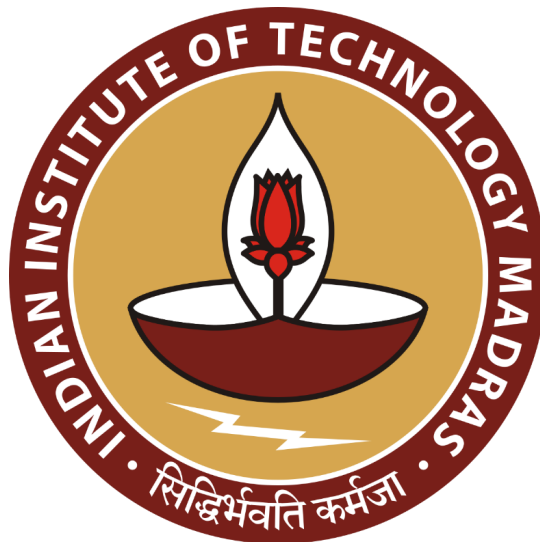
**A Project Report for the**

## **Software Engineering**

***(Final Submission)***

**Submitted By:**

- |                                  |                                                                                           |
|----------------------------------|-------------------------------------------------------------------------------------------|
| <b>1. Aditya R</b>               | <a href="mailto:21f1006862@ds.study.iitm.ac.in"><u>21f1006862@ds.study.iitm.ac.in</u></a> |
| <b>2. Ashutosh Kumar Barnwal</b> | <a href="mailto:21f1001709@ds.study.iitm.ac.in"><u>21f1001709@ds.study.iitm.ac.in</u></a> |
| <b>3. Kanishk Mishra</b>         | <a href="mailto:21f1006627@ds.study.iitm.ac.in"><u>21f1006627@ds.study.iitm.ac.in</u></a> |
| <b>4. Nikhil Guru Venkatesh</b>  | <a href="mailto:21f3000424@ds.study.iitm.ac.in"><u>21f3000424@ds.study.iitm.ac.in</u></a> |
| <b>5. Shubhankar Jaiswal</b>     | <a href="mailto:21f1006828@ds.study.iitm.ac.in"><u>21f1006828@ds.study.iitm.ac.in</u></a> |
| <b>6. Sushobhan Bhargav</b>      | <a href="mailto:22f1000948@ds.study.iitm.ac.in"><u>22f1000948@ds.study.iitm.ac.in</u></a> |
| <b>7. Utkarsh Kumar Yadav</b>    | <a href="mailto:21f1006520@ds.study.iitm.ac.in"><u>21f1006520@ds.study.iitm.ac.in</u></a> |



**IITM Online BS Degree Program,**

**Indian Institute of Technology, Madras, Chennai**

**Tamil Nadu, India, 600036**

# Contents

<b>Problem Statement.....</b>	<b>1</b>
<b>Milestone 1: Identify User Requirements.....</b>	<b>2</b>
1.1 Identifying Users of the Application.....	2
1.2 User Stories for New Features and Integrations.....	3
<b>Milestone 2 - User Interfaces.....</b>	<b>7</b>
2.1 Storyboard for the Application.....	7
2.2 Low-Fidelity Wireframes.....	22
2.3 Usability Design Guidelines and Heuristics.....	29
<b>Milestone 3 - Scheduling and Design.....</b>	<b>30</b>
3.1 Project Schedule.....	30
3.1.1 Sprints Schedule.....	30
3.1.2 Scrum Board.....	31
3.1.3 Timeline / Gantt Chart.....	32
3.2 Design of Components (Additional Features).....	32
3.3 Class Diagram.....	34
3.4 SCRUM Meeting Minutes/Details.....	36
<b>Milestone 4 - API Integrations.....</b>	<b>37</b>
4.1 Api.yaml.....	37
<b>Milestone 5 - Test cases, test suite of the project.....</b>	<b>39</b>
Introduction.....	39
Testing Framework.....	39
5.1 App Testing on Discourse API:.....	39
5.1.1 Sample Fixture.....	39
5.1.2 Test for Searching Posts from Discourse.....	40
5.1.3 Test for Retrieval of all Posts from Discourse.....	40
5.1.4 Test for Retrieval of Specific Post from Discourse.....	41
5.1.5 Test for Creating a New Post on Discourse.....	42
5.1.6 Test for Updating the Post on the Discourse.....	43

5.2 App Testing on G-Chat Webhook:	44
5.2.1 Sample Fixture	44
5.2.2 Test for sending notifications	45
5.2.3 Test for high-priority notifications	45
5.2.4 Test for urgent-priority notifications	46
5.3 Test Results:	47
5.4 Test Environment	47
Conclusion	47
<b>Milestone 6 - Final Submission</b>	<b>48</b>
6.1 Summary of work done in different milestones	48
6.2 Code Review, Issue Reporting and Tracking	50
6.3 Implementation Details of the Project	53
6.3.1 Tools and Technologies Used	53
6.3.2 Instructions to run our application	54

# Problem Statement

---

## Integration Requirements for IITM BS Ticketing System

The IITM BS team has chosen to adopt the “**Online support ticket system for the IITM BS degree program**”, for managing queries and concerns within the IITM BS degree program. As part of this integration, the team aims to seamlessly integrate the ticketing system into their existing ecosystem by implementing the following functionalities:

### 1. **Discourse Integration:**

Alongside the core functionalities of ticket creation and listing, the ticketing system is required to integrate with Discourse, a platform for community discussion and communication. This integration entails the creation of a Discourse thread for each ticket generated within the ticketing system. Various configurations and rules can be implemented to manage Discourse threads, such as setting initial thread visibility (private/public) and enabling notifications for thread activity (e.g., replies, likes). The integration will leverage the Discourse API documentation to explore available features and determine optimal integration strategies.

### 2. **Webhooks Integration:**

To address high-priority tickets promptly, the ticketing system will integrate with webhooks to enable real-time notifications for urgent and high-priority tickets. These notifications will be transmitted to GChat, facilitating swift action and escalation by higher authorities. The integration with Google Chat webhooks will enable seamless communication and collaboration, ensuring efficient handling of critical issues within the IITM BS degree program.

## Prototype Solution Implementation:

For this project, the focus will be on developing a prototype solution that demonstrates the integration capabilities of the ticketing system. The prototype will be hosted locally, providing a sandbox environment for testing and validation. Through the prototype, the integration with Discourse and webhooks will be showcased, as will the ability to handle urgent tickets effectively.

# Milestone 1: Identify User Requirements

---

## 1.1 Identifying Users of the Application

In the process of requirement gathering and analysis, it is essential to identify the various categories of users who will interact with our system.

Users can be broadly classified into three main categories:

1. **Primary Users:** Primary users are the frequent users of the system who directly engage with its functionalities. They are the primary beneficiaries of the software. For instance, in our project, primary users include:
  - a. Students of the IITM BS degree program: They will primarily interact with the support ticketing system to raise queries and concerns.
  - b. Support staff: Responsible for addressing the support tickets raised by students and managing the ticketing system.
  - c. Admins: Administrators with elevated privileges are responsible for managing the ticketing system and overseeing support operations.
2. **Secondary Users:** Secondary users are those who indirectly interact with the system, often through an intermediary. They may not directly utilize the software but rely on it for informational or coordination purposes. In our context, secondary users could encompass:
  - a. Faculty Members: Academic staff who may occasionally interact with the ticketing system to address student concerns related to academic matters.
  - b. IT Staff: Technical personnel responsible for maintaining and troubleshooting the ticketing system's infrastructure and integrations.
3. **Tertiary Users:** Tertiary users do not directly use the software but are impacted by its introduction or influence its adoption. They may include external stakeholders or entities whose actions or decisions are influenced by the software. For instance, prospective students or regulatory bodies associated with the IITM BS degree program may fall under this category.

- a. Higher authorities: They may receive notifications for high-priority and urgent tickets through the Webhooks integration, enabling them to handle escalations effectively.
- b. External Stakeholders: Individuals or entities outside the IITM BS degree program who may interact with the ticketing system for collaboration or information-sharing purposes.

## 1.2 User Stories for New Features and Integrations

The agile lifecycle commences with Behavior Driven Design (BDD), where we delve into the behavior of the application both before and during development. Continuous refinement of requirements is integral to meeting evolving expectations. In the agile approach, requirements are expressed as User Stories, which replace the traditional Software Requirements Specification (SRS) from a planning and documentation standpoint.

User Stories serve as succinct, informal descriptions in plain language, outlining what a user seeks to accomplish within the software product and the value it brings to them. Following the "Role-feature-benefit" pattern, each User Story encapsulates the smallest unit of work achievable within a single sprint:

*As a [type of user],*

*I want [an action],*

*So that [a benefit / value]*

This structured format ensures clarity and alignment between user needs, system functionalities, and the resulting benefits, facilitating efficient development iterations within the agile framework.

## **User Stories:**

### **As a Student,**

#### Scenario 1

- I want to create a support ticket for a specific concern or query,
- So that I can receive assistance from the support team promptly, I should be informed through notification or an email/GChat (webhook).

#### Scenario 2

- I want to view a list of similar tickets before creating a new one with the help of Discourse integration,
- So that I can avoid duplicating existing support requests and contribute to efficient ticket management.

#### Scenario 3

- I want to like or +1 an existing support ticket,
- So that popular concerns or queries can be prioritized by the support team.

### **As a Support Staff,**

#### Scenario 1

- I want to receive notifications when a new support ticket is created through notification or an email/GChat (webhook),
- So that I can promptly address student concerns and assist.

#### Scenario 2

- I want to mark a ticket as resolved once the concern has been addressed.
- So that students are notified through an email/GChat (webhook) and the ticket is closed appropriately.

## **As an Admin,**

### Scenario 1

- I want to categorize resolved tickets and add them to the FAQ section.
- So that future students can access updated FAQs and find answers to common queries efficiently with the integration of Discourse.

### Scenario 2

- I want to manage user roles and permissions,
- So that access to system features and data is controlled based on user roles and responsibilities.

## **As a Faculty Member,**

- I want to access the ticketing system to address academic-related concerns raised by students,
- So that I can provide support and guidance to students when needed.

## **As an IT Staff,**

- I want to ensure the reliability and security of the ticketing system's infrastructure,
- So that the system operates smoothly and student data is protected from unauthorized access or breaches.

## **Acceptance Criteria:**

- Integration with Discourse to fetch created tickets and synchronize query data between platforms.
- Implementation of real-time notifications via Google Chat for staff members whenever a new query is created in the ticket system, ensuring prompt attention to user queries.
- Timestamps are displayed on notifications to provide staff members with context regarding query creation times and enable them to prioritize responses accordingly.



- Seamless navigation between Discourse and the ticket system to facilitate easy access to previous query resolutions and prevent redundant efforts by staff members.
- User-friendly interface for staff members to manage notification settings and customize preferences for query alerts, optimizing workflow efficiency and responsiveness to user queries.

These user stories will serve as the foundation for identifying and prioritizing features and integrations to be implemented in the existing ticketing system.

They align with the SMART (Specific, Measurable, Achievable, Relevant, and Time-bound) guidelines and address the needs and expectations of various user groups involved in the support process.

## Milestone 2 - User Interfaces

---

### 2.1 Storyboard for the Application

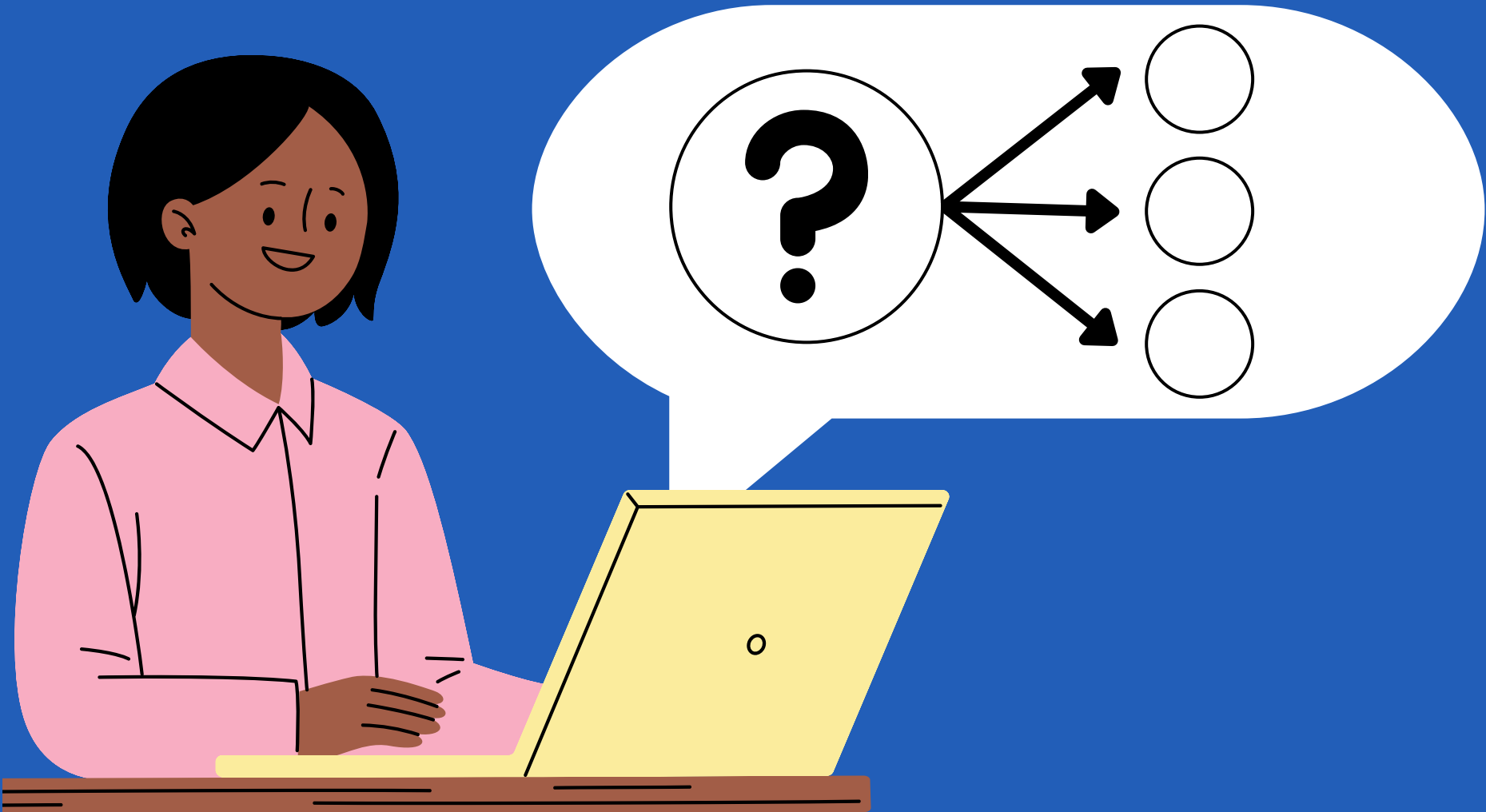
# Storyboard

A VISUAL TECHNIQUE TO DESIGN  
THE USER INTERFACE OF OUR  
APPLICATION.

# **Storyboard-1**

## SAMITA'S DILEMMA

**Samita submits a query via  
the ticket system.**



**The instructor resolves Samita's query in the ticket system, but Samita is unaware of the resolution.**



**One day later, Samita checks the ticket system and discovers her query has been resolved, making her happy.**



**Samita notices that her query was resolved one hour after creation, which she only realizes now.**



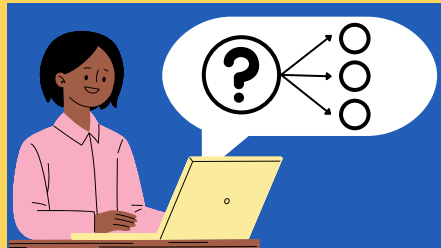


**Samita discusses this feature with her friend, who informs her that the query had been resolved previously in discourse, making her sad.**



# Storyboard-1

This storyboard showcases illustrations and descriptions demonstrating how the Online Support Ticketing System at IITM can be improved by incorporating a transparent chat feature like Discourse, which would keep users informed about the status of their queries.



Samita submits a query via the ticket system.



The instructor resolves Samita's query in the ticket system, but Samita is unaware of the resolution.



One day later, Samita checks the ticket system and discovers her query has been resolved, making her happy.



Samita notices that her query was resolved one hour after creation, which she only realizes now.

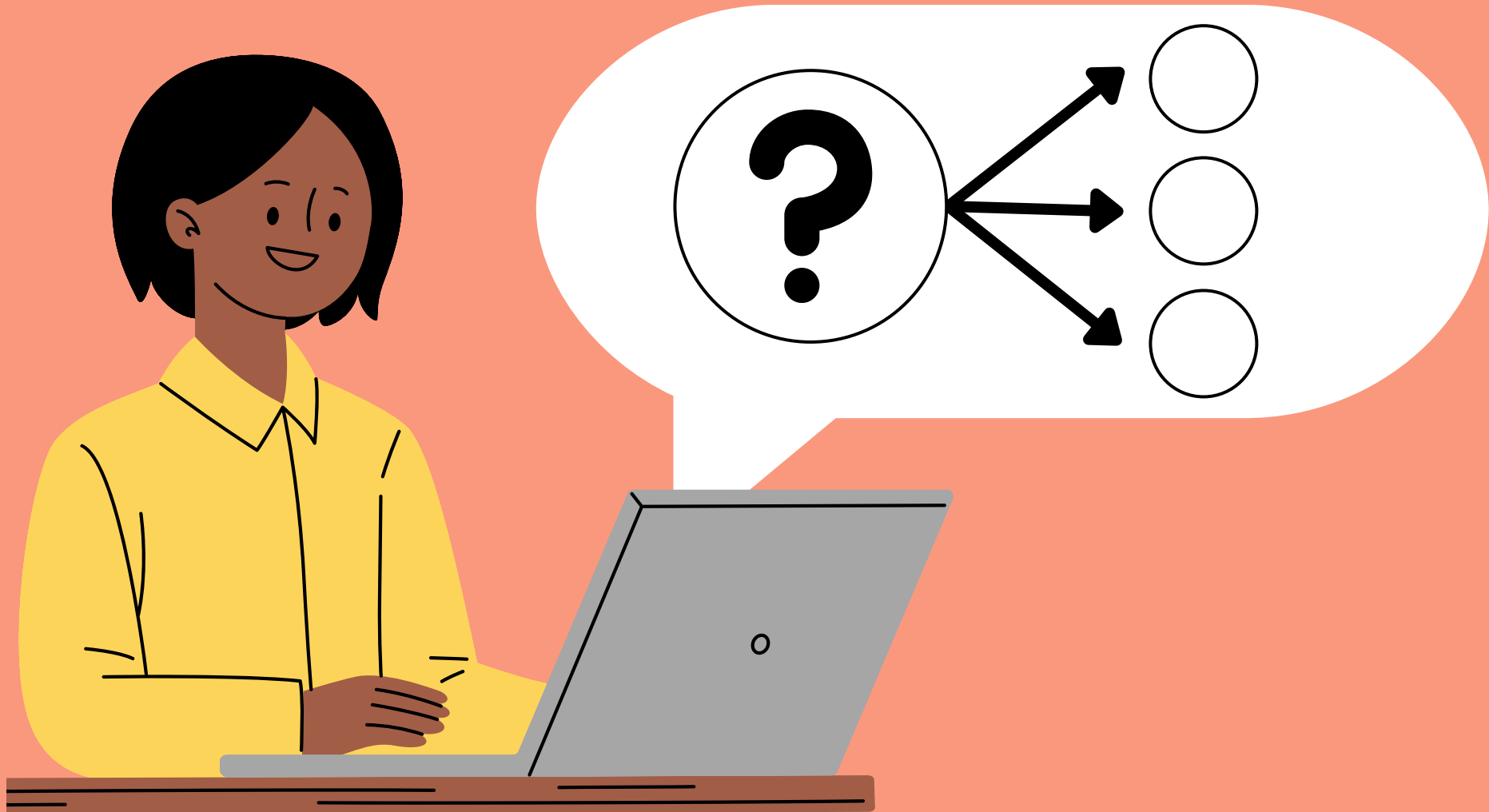


Samita discusses this feature with her friend, who informs her that the query had been resolved previously in discourse, making her sad.

# **Storyboard-2**

SUPPORT STAFF'S DILEMMA

**Samita creates a ticket with  
a query in the ticket system**



**The IITMBS staff member checks the new query four hours later, realizing the need for timely notifications to enhance efficiency.**



**Despite not receiving a notification, the staff member proceeds to read and understand the query.**



**While addressing the query, the staff member recalls having previously resolved a similar query in Discourse.**

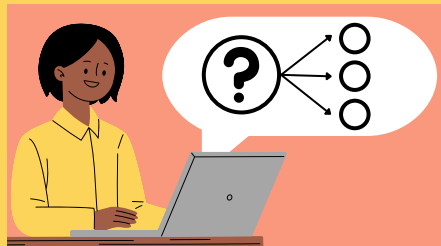


**Despite this realization, the staff member unintentionally solves the query again within the ticket system.**



# Storyboard-2

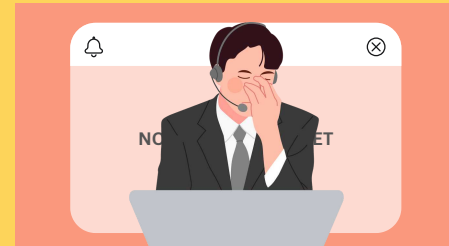
This storyboard showcases illustrations and descriptions demonstrating how the Online Support Ticketing System at IITM can be improved by incorporating a transparent chat feature like Discourse, which would keep users informed about the status of their queries.



Samita creates a ticket with a query in the ticket system



The IITMBS staff member checks the new query four hours later, realizing the need for timely notifications to enhance efficiency.



Despite not receiving a notification, the staff member proceeds to read and understand the query.



While addressing the query, the staff member recalls having previously resolved a similar query in Discourse.



Despite this realization, the staff member unintentionally solves the query again within the ticket system.



## 2.2 Low-Fidelity Wireframes

---

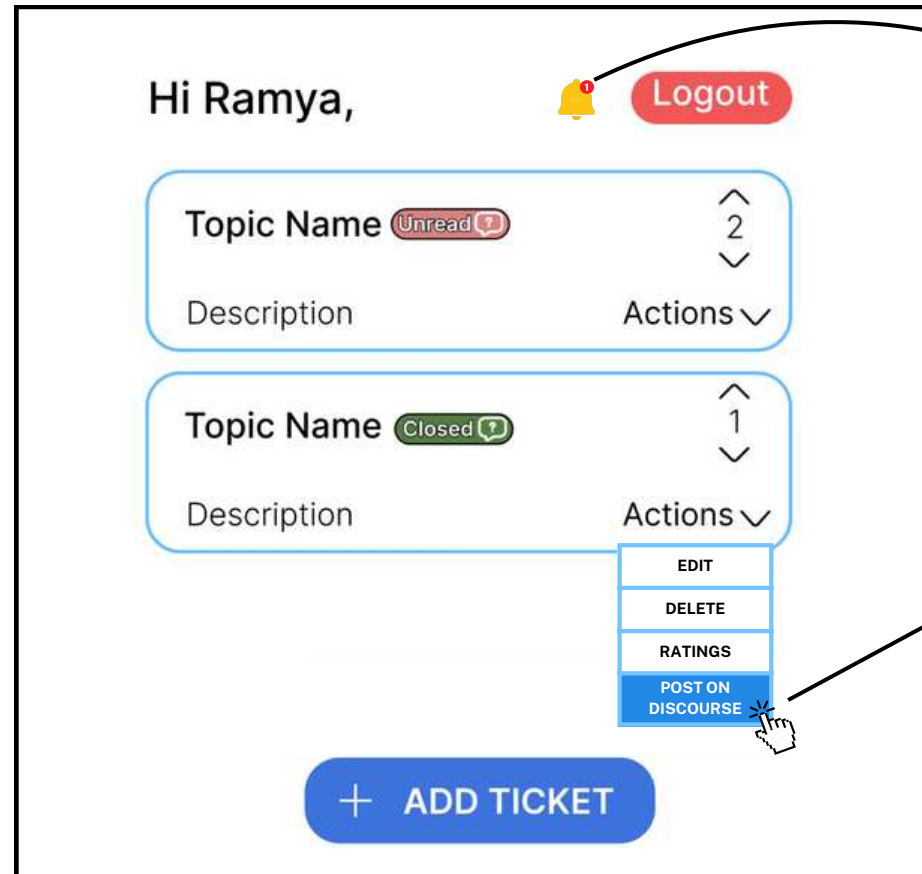
# Wireframe

A DIGITAL PROTOTYPE TO MAP OUT  
THE USER INTERFACE OF OUR  
APPLICATION.

# Wireframe: Student's view

A digital prototype to map out the user interface of our application.

Existing interface of  
the Online support  
ticket system

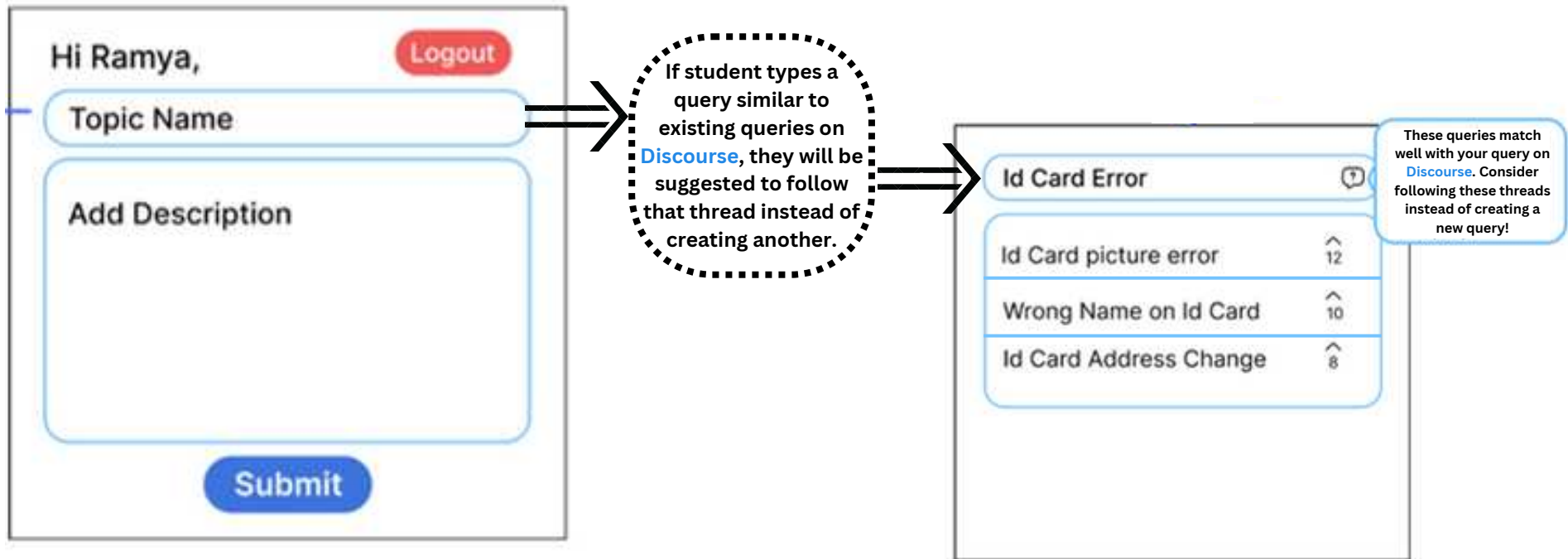


**Added** feature for  
students to receive  
timely notifications

**Added** feature for  
students to create a  
post of the issue on  
discourse

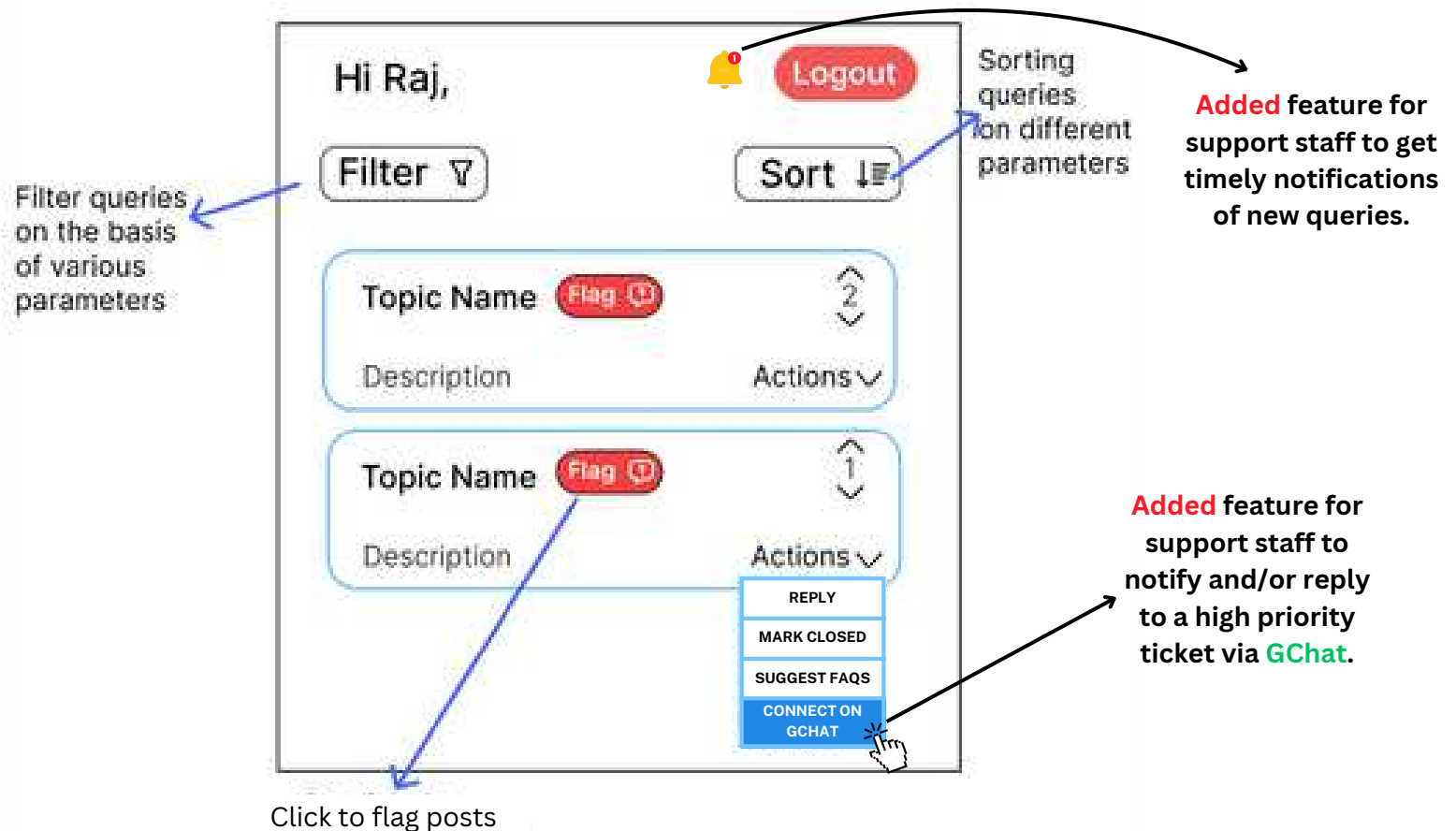
# Wireframe: Student's view

A digital prototype to map out the user interface of our application.



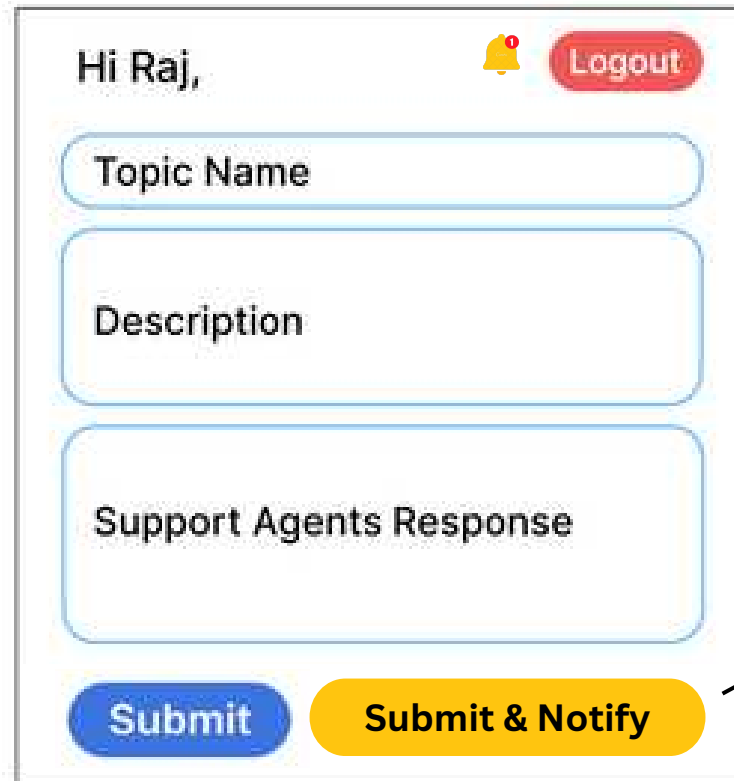
# Wireframe: Staff's view

A digital prototype to map out the user interface of our application.



# Wireframe: Staff's view

A digital prototype to map out the user interface of our application.



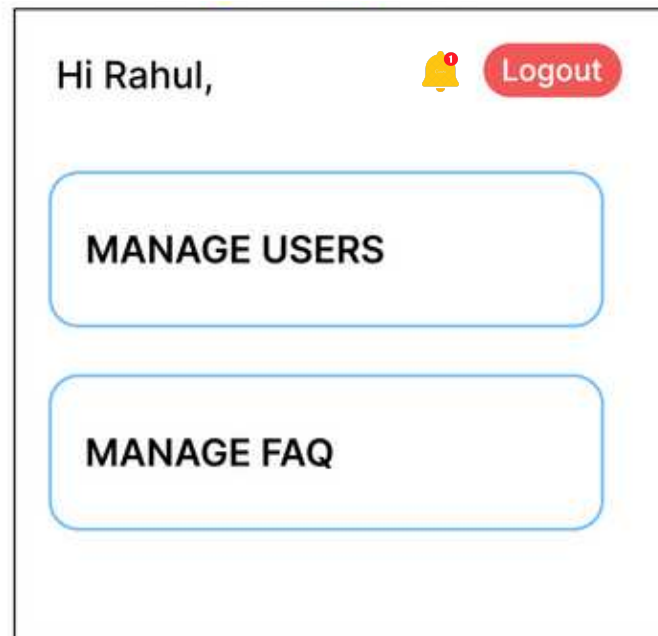
The wireframe shows a staff member's view of the application. At the top left, it says "Hi Raj,". To the right of the greeting is a yellow bell icon with a red notification badge containing the number "1". Further right is a red "Logout" button. Below the greeting are three input fields: "Topic Name", "Description", and "Support Agents Response". At the bottom are two buttons: a blue "Submit" button and a yellow "Submit & Notify" button. An arrow points from the "Submit & Notify" button to a text block on the right.

**Added** feature for support staff that uses **Google Chat** webhooks to notify high priority tickets

# Wireframe: Admin's view

A digital prototype to map out the user interface of our application.

## Admin's view- to Manage FAQs




Inclusive of all the existing features

# Wireframe: Admin's view

A digital prototype to map out the user interface of our application.

## Admin's view- to Manage FAQs

Hi Rahul,  [Logout](#)

Topic Name 2  
Description Actions ▾

Topic Name 1  
Description Actions ▾  
Edit  
Delete  
Change Category

**Button to add FAQs for admins** → [+ ADD FAQ](#)

FAQ Name ▾

Add Description

[Submit](#)

Inclusive of all the existing features

## 2.3 Usability Design Guidelines and Heuristics

The wireframes have been developed by applying the usability design guidelines and heuristics discussed above, ensuring that the interface is intuitive, efficient, and user-friendly. Some of the key principles considered during wireframe design include:

1. **Visibility of system status:** Users are provided with clear feedback about the status of their actions, such as when a ticket is successfully submitted or resolved.
2. **User control and freedom:** Users have the ability to navigate through the application easily and undo actions if necessary, promoting a sense of control over their interactions.
3. **Consistency and standards:** Interface elements and interactions are consistent across different screens, following established design patterns and standards to enhance usability.
4. **Error prevention:** Measures are implemented to prevent user errors, such as validation checks on form inputs and clear error messages to guide users in correcting mistakes.
5. **Recognition over recall:** Information and actions are presented in a way that minimizes the need for users to remember details from previous interactions, reducing cognitive load and enhancing usability.
6. **Flexibility and efficiency of use:** The interface accommodates both novice and experienced users, providing shortcuts and advanced features for efficient task completion without overwhelming beginners.

By incorporating these usability design principles into the wireframes, the aim is to create an interface that meets the needs of users, facilitates their tasks effectively, and ensures a positive overall user experience with the ticketing system application.



# Milestone 3 - Scheduling and Design

---

## 3.1 Project Schedule

### 3.1.1 Sprints Schedule

- ❖ **Sprint 1:** Identify Users and User Requirements for the Application
  - Date: 09/02/24 - 15/02/24
- ❖ **Sprint 2:** User Stories for the requirements, Milestone-1 Vetting and Final Submission
  - Date: 16/02/24 - 23/02/24
- ❖ **Sprint 3:** Storyboards and Wireframes (by following usability guidelines and heuristics), Milestone-2 Vetting and Final Submission
  - Date: 24/02/24 - 29/02/24
- ❖ **Sprint 4:** Project Schedule, Components Design and Description, Class Diagram and Minutes Of Meetings, Milestone-3 Vetting and Final Submission
  - Date: 01/03/24 - 06/03/24
- ❖ **Sprint 5:** Discourse API Integration, Database Schema and Re-Modeling, User Class, Students Class (Code, Debug, Raise Issues & Code Review)
  - Date: 07/02/24 - 11/02/24
- ❖ **Sprint 6:** For each user story, create appropriate API endpoints, adding G-Chat Webhook, code review, debug, Milestone-4 Vetting, and Final Submission
  - Date: 12/03/24 - 15/03/24
- ❖ **Sprint 7:** Design Test cases for APIs and other functionalities; Perform Unit testing using Pytest
  - Date: 16/03/24 - 21/03/24
- ❖ **Sprint 8:** Complete Unit testing, Milestone-5 Vetting and Final Submission, Frontend Modification
  - Date: 22/03/24 - 29/03/24
- ❖ **Sprint 9:** Finish Frontend, Integrate the Frontend and the Backend, Final project report, Record project presentation, Milestone-6 Vetting, and Final Submission
  - Date 30/03/24 - 16/04/24

## Project Scheduling Tools: JIRA

We primarily utilize Jira for project management, task tracking, and collaboration. Jira's well-defined interface and customizable features make it well-suited for agile project management.

### 3.1.2 Scrum Board

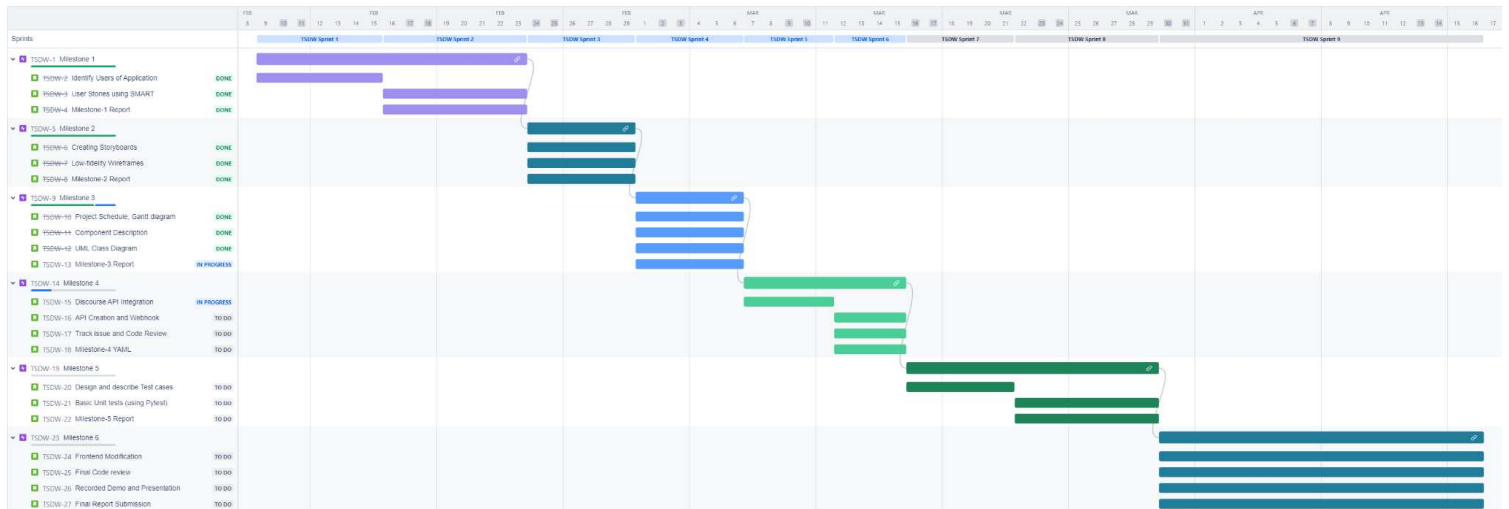
The screenshot displays the Jira Software interface for the 'SE-Project' (Software project). The top navigation bar includes 'Your work', 'Projects', 'Filters', 'Dashboards', 'Teams', 'Plans', 'Apps', and a 'Create' button. The left sidebar shows the project's navigation menu with sections for 'PLANNING' (Timeline, Backlog, Board, Goals, Issues, Add view) and 'DEVELOPMENT' (Code, Project pages, Add shortcut, Project settings). The main area is titled 'All sprints' and shows a Scrum Board with three columns: 'TO DO 3', 'IN PROGRESS 2', and 'DONE 9'. Each column contains task cards with titles, milestones, and IDs. The 'TO DO' column has three cards: 'API Creation and Webhook' (MILESTONE 4, TSDW-16), 'Track issue and Code Review' (MILESTONE 4, TSDW-17), and 'Milestone-4 YAML' (MILESTONE 4, TSDW-18). The 'IN PROGRESS' column has two cards: 'Milestone-3 Report' (MILESTONE 3, TSDW-13) and 'Discourse API Integration' (MILESTONE 4, TSDW-15). The 'DONE' column has nine cards, including 'Identify Users of Application' (MILESTONE 1, TSDW-2), 'User Stories using SMART' (MILESTONE 1, TSDW-3), 'Milestone-1 Report' (MILESTONE 1, TSDW-4), 'Creating Storyboards' (MILESTONE 2, TSDW-6), 'Low-fidelity Wireframes' (MILESTONE 2, TSDW-7), 'Milestone-2 Report' (MILESTONE 2, TSDW-8), 'Project Schedule, Gantt diagram' (MILESTONE 3, TSDW-10), 'Component Description' (MILESTONE 3, TSDW-11), and 'UML Class Diagram' (MILESTONE 3, TSDW-12). Each card includes a green status icon, a milestone label, and a user icon. A search bar and 'Add people' button are located above the columns.

Column	Task Title	Milestone	ID	Status	Assignee
TO DO 3	API Creation and Webhook	MILESTONE 4	TSDW-16	Not Started	Assignee
	Track issue and Code Review	MILESTONE 4	TSDW-17	Not Started	Assignee
	Milestone-4 YAML	MILESTONE 4	TSDW-18	Not Started	Assignee
IN PROGRESS 2	Milestone-3 Report	MILESTONE 3	TSDW-13	In Progress	Assignee
	Discourse API Integration	MILESTONE 4	TSDW-15	In Progress	Assignee
DONE 9	Identify Users of Application	MILESTONE 1	TSDW-2	Done	Assignee
	User Stories using SMART	MILESTONE 1	TSDW-3	Done	Assignee
	Milestone-1 Report	MILESTONE 1	TSDW-4	Done	Assignee
	Creating Storyboards	MILESTONE 2	TSDW-6	Done	Assignee
	Low-fidelity Wireframes	MILESTONE 2	TSDW-7	Done	Assignee
	Milestone-2 Report	MILESTONE 2	TSDW-8	Done	Assignee
	Project Schedule, Gantt diagram	MILESTONE 3	TSDW-10	Done	Assignee
	Component Description	MILESTONE 3	TSDW-11	Done	Assignee
	UML Class Diagram	MILESTONE 3	TSDW-12	Done	Assignee

[3.1] Scrum Board

### 3.1.3 Timeline / Gantt Chart

The Gantt-Chart for the project schedule is shown below. For the high-resolution image of the chart, please [click here](#).



[3.2] Timeline / Gantt diagram of Project Schedule

## 3.2 Design of Components (Additional Features)

Description of the different components of the ticketing system with integration of Discourse and Webhook:

### 1. User Component:

- Represents a generic user of the system.
- Contains attributes such as userId, roleId, password, emailId, name, and userName.
- Includes methods for logging in and editing profile information.

### 2. Ticket Component:

- Represents a support ticket created by a student.
- Contains attributes such as ticketId, title, description, creationDate, creatorId, responderId, numberOfUpvotes, isRead, isOpen, isOffensive, and isFAQ.
- Includes methods for rating answers to the ticket and managing its status.

### 3. Student Component:

- Represents a student user of the system.
- Contains attributes such as rollNumber and a list of ticket objects.

- c. Create a support ticket for their queries/concerns.
- d. Includes methods for creating, editing, deleting, and reopening tickets.
- e. Receive notification through email/gchat regarding the status of their queries (webhooks).
- f. View a list of similar tickets to avoid duplicating existing support request.
- g. Integration with Discourse for retrieving relevant tickets.
- h. Search and Filter tickets based on keywords or categories.
- i. like or +1 an existing support ticket to prioritize popular concerns.

#### **4. SupportStaff Component:**

- a. Represents a support agent user of the system.
- b. Contains attributes such as agentId and a list of ticket objects.
- c. Includes methods for checking open queries, closing tickets, flagging tickets, forwarding offensive tickets, and suggesting FAQs.
- d. Receive a notification when a new ticket is created through email/gchat (webhooks).
- e. View ticket details, and respond to the students' queries and concerns.
- f. Mark tickets as resolved once the query has been addressed.
- g. Send notifications to students after the query/concern has been resolved.
- h. Close the ticket after it has been resolved.

#### **5. Admin Component:**

- a. Represents an admin user of the system.
- b. Contains attributes such as adminId.
- c. Includes methods for adding, editing, and deleting FAQs, as well as managing users.
- d. Categorize resolved tickets and add them to the FAQ section for future reference.
- e. Define and manage user roles and permissions.
- f. Controls access to system features and data based on user roles and responsibilities.
- g. Create, Edit, and Delete user roles and assign permissions accordingly.

#### **6. IT\_Staff Component:**

- a. Represents an IT staff user of the system.

- b. Contains attributes such as staffId.
- c. Includes methods for adding and removing admins, as well as checking resolution time.
- d. Monitors and manages the reliability and security of the ticketing system's infrastructure.
- e. Implement security measures to protect student data from unauthorized access or breaches.

**7. Discourse:**

- a. Represents the component responsible for integrating with the Discourse platform.
- b. Includes methods for creating and modifying threads, as well as notifying thread replies.

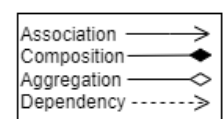
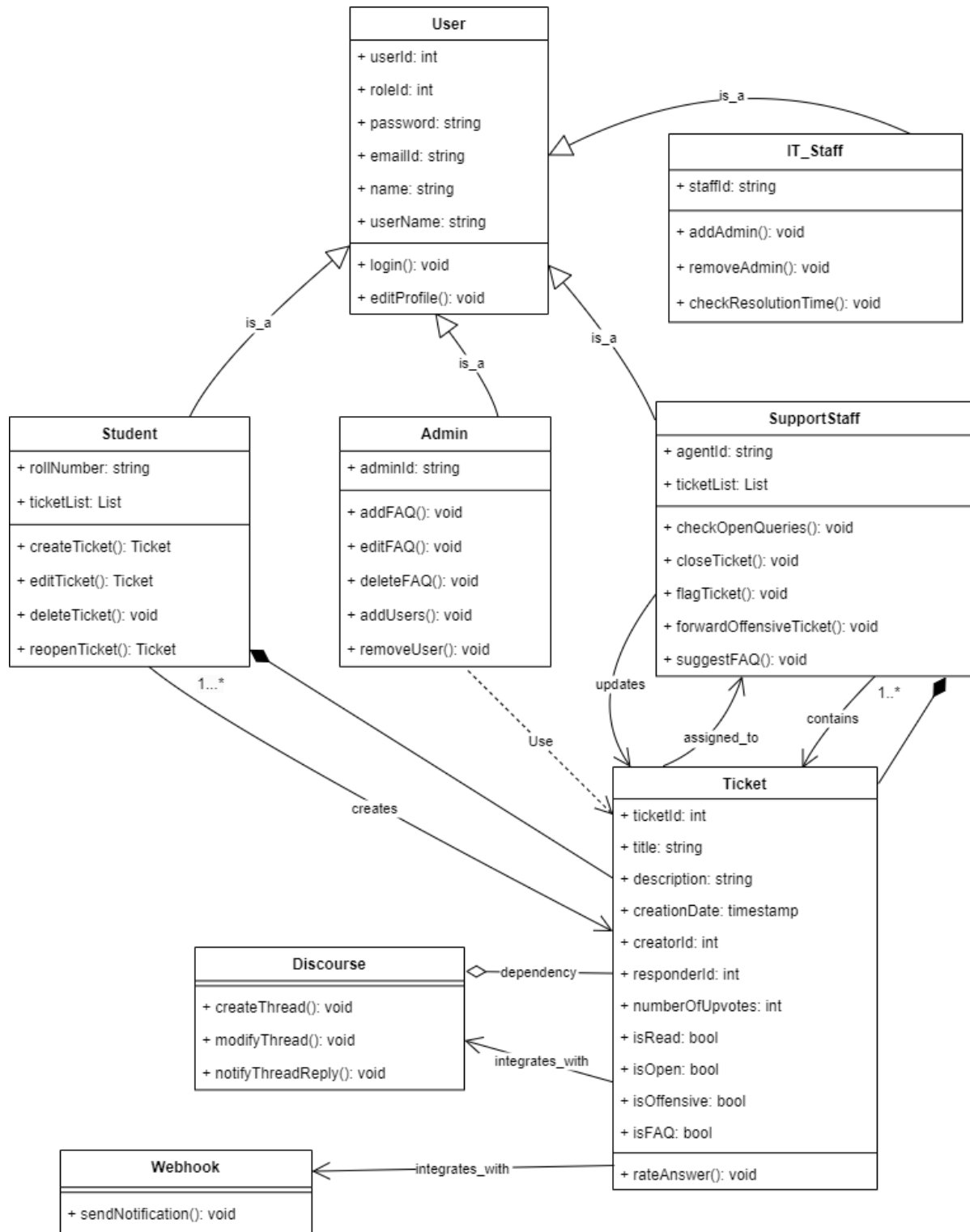
**8. Webhook:**

- a. Represents the component responsible for integrating with G-Chat using webhooks.
- b. Includes a method for sending notifications to G-Chat.

These components work together to facilitate the creation, management, and resolution of support tickets within the ticketing system. Users interact with the system through various interfaces provided by these components, allowing for efficient communication and collaboration between students, support agents, admins, and IT staff.

### **3.3 Class Diagram**

The UML class diagram provides a visual representation of the 8 components and their relationships within the ticketing system project. It illustrates the structure of the system, including the different types of users (such as students, support staff, admins, and IT staff), as well as the main entities like tickets. Additionally, it highlights the interactions between these components, such as the creation and management of tickets, integration with external platforms like Discourse and G-Chat, and the roles and responsibilities of different user types. Overall, the class diagram serves as a blueprint for the system architecture, aiding in understanding and communicating the design of the ticketing system.



[3.3] Class Diagram

### **3.4 SCRUM Meeting Minutes/Details**

**SCRUM Meetings Schedule: Every Tuesday, Thursday and Friday (20:30 - 22:30 PM)**

**Location: Google Meet**

**Attendees: Aditya R, Ashutosh Kumar Barnwal, Kanishk Mishra, Nikhil Guru Venkatesh, Shubhankar Jaiswal, Sushobhan Bhargav, and Utkarsh Kumar Yadav**

#### **Sprint 1 SCRUM meetings minutes/details:**

The team collaborated together to identify and define different Users and User requirements. Each team member presented their own ideas and insights based on their understanding of the project requirements. After identifying the users each team member was assigned a task to come up with user stories for the different users.

#### **Sprint 2 SCRUM meetings minutes/details:**

User stories were finalized, all the team members sat together in a meeting to review the milestone-1 report, and the Final submission was made.

#### **Sprint 3 SCRUM meetings minutes/details:**

The team collaborated to create storyboards and wireframes for the project. Each member gave their ideas and suggestions regarding the storyboards. A discussion took place to review the storyboards and wireframes created by the team, providing feedback and suggestions for improvements. After the final review, the Milestone-2 report was submitted.

#### **Sprint 4 SCRUM meetings minutes/details:**

Decided on Project management software (JIRA) to keep track of the project's progress. After a thorough discussion among the team members, the project schedule was decided, and a Gantt chart was created for the same. Components were designed, and a class diagram for the application was made. Following the final review, the milestone-3 report was submitted.

# Milestone 4 - API Integrations

---

## 4.1 Api.yamll

Some Screenshots of openapi.yaml file:

```
1 openapi: 3.0.3
2 info:
3   title: Ticketing System with integration of Discourse - OpenAPI 3.0
4   version: "0.0.1"
5   contact:
6     email: "team_11@ds.study.iitm.ac.in"
7   description: This API describes the endpoints available for the software created by Team 11 for the Software Engineering Project.
8
9 servers:
10   - url: http://127.0.0.1:5000
11
12 tags:
13   # New feature added
14   - name: DiscourseSearchAPI
15     description: Endpoint to Discourse Posts by its title.
16   - name: DiscoursePostAPI
17     description: Endpoint to create, read, delete, and update Discourse Posts
18
```

```
50 paths:
51   /api/discourse/search:
52     get:
53       tags:
54         - DiscourseSearchAPI
55       summary: "Search Discourse Posts"
56       description: "Search Discourse posts by title"
57       operationId: searchDiscoursePosts
58       parameters:
59         - name: q
60           in: query
61           description: "Title of the Discourse post"
62           required: true
63           schema:
64             type: string
65             example: "Search query... "
66       responses:
67         '200':
68           description: ""Discourse posts searched successfully"
69           content:
70             application/json:
71               schema:
72                 type: array
73                 items:
74                   $ref: '#/components/schemas/DiscoursePost'
75         '404':
76           description: "Discourse post not found"
77           content:
78             application/json:
79               schema:
80                 type: object
81                 properties:
82                   message:
83                     type: string
84                     example: "Discourse post with the provided title not found"
85
```



```

50  paths:
86    /api/discourse/posts:
113      post:
114        security:
115          - ApiAuth: []
116          - ApiKey: []
117        tags:
118          - DiscoursePostAPI
119        summary: "Create a Discourse Post"
120        description: "Create a new Discourse post for a ticket"
121        operationId: createDiscoursePost
122        requestBody:
123          required: true
124          content:
125            application/json:
126              schema:
127                type: object
128                properties:
129                  title:
130                    type: string
131                    description: "Title of the Discourse post"
132                    example: "Creating new Post on Discourse"
133                  content:
134                    type: string
135                    description: "Content of the Discourse post"
136                    example: "A new Post has been created on Discourse."
137                  category:
138                    type: integer
139                    description: "Category of the Discourse post"
140                    example: 4
141        responses:
142          '200':
143            description: "Discourse post created successfully"
144            content:
145              application/json:
146                schema:
147                  type: object
148                  properties:

```

# Milestone 5 - Test cases, test suite of the project

---

## Introduction

These are the testing strategies and procedures for the application. Testing is a crucial aspect of software development to ensure the application's reliability, functionality, and security.

## Testing Framework

The application's tests are written using the pytest framework, which is a widely used testing framework for Python applications. pytest offers a simple syntax for writing tests and provides powerful features for testing various aspects of the application.

### 5.1 App Testing on Discourse API:

**Purpose:** This test case verifies the functionality of the DiscourseSearchAPI class, which is responsible for searching Discourse topics and handling various operations related to Discourse posts.

#### 5.1.1 Sample Fixture

The following figure shows the sample fixture used for testing purposes. This fixture starts the app with test configuration, and the tests are carried out within the app context.

```
soft-engg-project-jan-2024-se-jan-11 > Code > backend > tests > test_discourseAPIs.py > ...
1  import pytest
2  from unittest.mock import patch
3  from flask import Flask
4  from flask_restful import Api
5  from application.api import DiscourseSearchAPI, DiscoursePostsAPI
6
7  @pytest.fixture
8  def app():
9      app = Flask(__name__)
10     api = Api(app)
11     api.add_resource(DiscourseSearchAPI, '/api/discourse/search')
12     api.add_resource(DiscoursePostsAPI, '/api/discourse/posts', '/api/discourse/posts/<int:post_id>')
13     return app
14
15  @pytest.fixture
16  def client(app):
17     return app.test_client()
18
```

### 5.1.2 Test for Searching Posts from Discourse

#### API being tested:

- <http://127.0.0.1:5000/api/discourse/search>

```
18 class DiscourseSearchAPI(Resource):
19     def get(self):
20         query = request.args.get('q', '')
21         discourse_url = 'http://localhost:4200/search.json'
22         params = {'q': query}
23
24         response = requests.get(discourse_url, params=params)
25
26         if response.status_code == 200:
27             return response.json(), 200
28         else:
29             return {'message': 'Error searching Discourse topics'}, response.status_code
30
```

#### Inputs:

- Request Method: GET

#### Expected Output:

- HTTP Status Code: 200
- JSON: {"message": "Discourse posts searched successfully"}

#### Actual Output:

- HTTP Status Code: 200
- JSON: {"message": "Discourse posts searched successfully"}

#### Result: Success

```
19 def test_discourse_search_api(client):
20     query = 'Test'
21     with patch('requests.get') as mock_get:
22         mock_get.return_value.status_code = 200
23         mock_get.return_value.json.return_value = {"message": "Discourse posts searched successfully"}
24
25         response = client.get(f'/api/discourse/search?q={query}')
26
27         assert response.status_code == 200
28         assert response.json == {"message": "Discourse posts searched successfully"}
29
```

### 5.1.3 Test for Retrieval of all Posts from Discourse

#### API being tested:

- <http://127.0.0.1:5000/api/discourse/posts>

```

39 # GET method to retrieve all posts from Discourse or a specific post
40 def get(self, post_id=None):
41     if post_id:
42         discourse_url = f'http://127.0.0.1:4200/posts/{post_id}.json'
43     else:
44         discourse_url = 'http://127.0.0.1:4200/posts.json'
45
46     response = requests.get(discourse_url)
47     return response.json(), response.status_code
48

```

#### Inputs:

- Request Method: GET

#### Expected Output:

- HTTP Status Code: 200
- JSON: {"message": "Discourse posts retrieved successfully"}

#### Actual Output:

- HTTP Status Code: 200
- JSON: {"message": "Discourse posts retrieved successfully"}

#### Result: Success

```

30 def test_discourse_posts_api_get(client):
31     with patch('requests.get') as mock_get:
32         mock_get.return_value.status_code = 200
33         mock_get.return_value.json.return_value = {"message": "Discourse posts retrieved successfully"}
34
35         response = client.get('/api/discourse/posts')
36
37         assert response.status_code == 200
38         assert response.json == {"message": "Discourse posts retrieved successfully"}
39

```

### 5.1.4 Test for Retrieval of Specific Post from Discourse

#### API being tested:

- http://127.0.0.1:5000/api/discourse/posts/<int:post\_id>

```

39 # GET method to retrieve all posts from Discourse or a specific post
40 def get(self, post_id=None):
41     if post_id:
42         discourse_url = f'http://127.0.0.1:4200/posts/{post_id}.json'
43     else:
44         discourse_url = 'http://127.0.0.1:4200/posts.json'
45
46     response = requests.get(discourse_url)
47     return response.json(), response.status_code
48

```

#### Inputs:

- Request Method: GET

### Expected Output:

- HTTP Status Code: 200
- JSON: {"message": "Discourse post retrieved successfully"}

### Actual Output:

- HTTP Status Code: 200
- JSON: {"message": "Discourse post retrieved successfully"}

**Result:** Success

```
40 def test_discourse_posts_api_get_by_id(client):
41     post_id = 23
42     with patch('requests.get') as mock_get:
43         mock_get.return_value.status_code = 200
44         mock_get.return_value.json.return_value = {"message": "Discourse posts retrieved successfully"}
45
46         response = client.get(f'/api/discourse/posts/{post_id}')
47
48         assert response.status_code == 200
49         assert response.json == {"message": "Discourse posts retrieved successfully"}
50
```

## 5.1.5 Test for Creating a New Post on Discourse

### API being tested:

- <http://127.0.0.1:5000/api/discourse/posts>

```
49 # POST method to create a new post in Discourse
50 def post(self):
51     args = self.parser.parse_args()
52     title = args['title']
53     content = args['content']
54     category = args.get('category')
55
56     if category is not None:
57         try:
58             category = int(category)
59         except ValueError:
60             return {'message': 'Category must be an integer'}, 400
61
62     discourse_url = 'http://127.0.0.1:4200/posts.json'
63     headers = {
64         'Api-Username': '22f1000948',
65         'Api-Key': '614834da24a228d0f1e69c48c07ce122b8cb2fd461837c974f0fb9d7c17de6f3'
66     }
67     payload = {
68         'title': title,
69         'raw': content,
70         'category': category
71     }
72
73     response = requests.post(discourse_url, headers=headers, json=payload)
74     return response.json(), response.status_code
75
```

### Inputs:

- Request Method: POST
- JSON: { 'title': 'Test API for creating Post on Discourse', 'content': 'Test API for creating content on Discourse', 'category': 4 }

### Expected Output:

- HTTP Status Code: 200
- JSON: { "message": "Discourse post created successfully" }

### Actual Output:

- HTTP Status Code: 200
- JSON: { "message": "Discourse post created successfully" }

### Result: Success

```
51 def test_discourse_posts_api_post(client):
52     payload = {'title': 'Test API for creating Post on Discourse', 'content': 'Test API for creating content on Discourse', 'category': 4}
53     with patch('application.api.requests.post') as mock_post:
54         mock_post.return_value.status_code = 200
55         mock_post.return_value.json.return_value = {"message": "Discourse post created successfully"}
56
57         response = client.post('/api/discourse/posts', json=payload)
58
59         assert response.status_code == 200
60         assert response.json == {"message": "Discourse post created successfully"}
61
```

## 5.1.6 Test for Updating the Post on the Discourse

### API being tested:

- [http://127.0.0.1:5000/api/discourse/posts/<int:post\\_id>](http://127.0.0.1:5000/api/discourse/posts/<int:post_id>)

```
76 # PUT method to update an existing post in Discourse
77 def put(self, post_id):
78     args = self.parser.parse_args()
79     content = args['content']
80
81     discourse_url = f'http://127.0.0.1:4200/posts/{post_id}.json'
82     headers = {
83         'Api-Username': '22f1000948',
84         'Api-Key': '614834da24a228d0f1e69c48c07ce122b8cb2fd461837c974f0fb9d7c17de6f3'
85     }
86     payload = {
87         'post': {
88             'raw': content
89         }
90     }
91
92     response = requests.put(discourse_url, headers=headers, json=payload)
93     return response.json(), response.status_code
94
```

**Inputs:**

- Request Method: PUT
- JSON: { 'content': 'Test API for updating content on Discourse' }

**Expected Output:**

- HTTP Status Code: 200
- JSON: { "message": "Discourse post updated successfully" }

**Actual Output:**

- HTTP Status Code: 200
- JSON: { "message": "Discourse post updated successfully" }

**Result: Success**

```
62 def test_discourse_posts_api_put(client):
63     post_id = 23
64     payload = {'content': 'Test API for updating content on Discourse'}
65     with patch('application.api.requests.put') as mock_put:
66         mock_put.return_value.status_code = 200
67         mock_put.return_value.json.return_value = {"message": "Discourse post updated successfully"}
68
69         response = client.put(f'/api/discourse/posts/{post_id}', json=payload)
70
71         assert response.status_code == 200
72         assert response.json == {"message": "Discourse post updated successfully"}
73
```

## 5.2 App Testing on G-Chat Webhook:

**Purpose:** This test case verifies the functionality of the `send_notification` function, which sends notifications to Google Chat with a `webhook` endpoint for processing high/urgent priority tickets.

### 5.2.1 Sample Fixture

The following figure shows the sample fixture used for testing purposes. This fixture starts the app with test configuration, and the tests are carried out within the app context.

```
soft-engg-project-jan-2024-se-jan-11 > Code > backend > tests > test_GChat_webhook.py > ...
1  import pytest
2  from unittest.mock import patch
3  from application.routes import app, send_notification
4
5  @pytest.fixture
6  def client():
7      with app.test_client() as client:
8          yield client
9
```

## 5.2.2 Test for sending notifications

### API being tested:

- `http://127.0.0.1:5000/webhook`

```
26 def send_notification(ticket_title, ticket_description, webhook_url):
27     message = {
28         "text": f"New high priority/urgent ticket:\nTitle: {ticket_title}\nDescription: {ticket_description}"
29     }
30     response = requests.post(webhook_url, json=message)
31     if response.status_code == 200:
32         print("Notification sent successfully")
33     else:
34         print("Failed to send notification")
35
```

### Inputs:

- Request Method: POST
- JSON: { "text": "New high priority/urgent ticket:\nTitle: Test Title\nDescription: Test Description" }

### Expected Output:

- HTTP Status Code: 200
- JSON: { "message": "Notification sent successfully" }

### Actual Output:

- HTTP Status Code: 200
- JSON: { "message": "Notification sent successfully" }

### Result: Success

```
10 def test_send_notification(client):
11     with patch('application.routes.requests.post') as mock_post:
12         mock_post.return_value.status_code = 200
13         webhook_url = 'https://chat.googleapis.com/v1/spaces/AAAAVFgvcso/messages?key=AIzaSyDdI0hCZtE6vySjMm-WEfRq3CPzqKqqsHI&token=bJ5KuUQVHQFNOLm'
14         send_notification("Test Title", "Test Description", webhook_url)
15
16     mock_post.assert_called_once_with(
17         webhook_url,
18         json={"text": "New high priority/urgent ticket:\nTitle: Test Title\nDescription: Test Description"}
19     )
20
```

## 5.2.3 Test for high-priority notifications

### API being tested:

- `http://127.0.0.1:5000/webhook`

```
36 @app.route('/webhook', methods=['POST'])
37 def webhook():
38     data = request.json
39     ticket_title = data.get('title')
40     ticket_description = data.get('description')
41     priority = data.get('priority')
42     if priority == 'high' or priority == 'urgent':
43         webhook_url = 'https://chat.googleapis.com/v1/spaces/AAAAVFgvcso/messages?key=AIzaSyDdI0hCZtE6vySjMm-WEfRq3CPzqKqqsHI&token=bJ5KuUQVHQFNOLm'
44         send_notification(ticket_title, ticket_description, webhook_url)
45     return '', 200
46
```



**Inputs:**

- Request Method: POST
- JSON: { 'title': 'Test Title', 'description': 'Test Description', 'priority': 'high' }

**Expected Output:**

- HTTP Status Code: 200

**Actual Output:**

- HTTP Status Code: 200

**Result: Success**

```
21 def test_webhook_high_priority(client):
22     with patch('application.routes.requests.post') as mock_post:
23         # Mock the JSON payload sent by the webhook
24         mock_request = {'title': 'Test Title', 'description': 'Test Description', 'priority': 'high'}
25
26         # Simulate the POST request to the webhook endpoint
27         response = client.post('/webhook', json=mock_request)
28
29         assert response.status_code == 200
30         mock_post.assert_called_once()
31
```

## 5.2.4 Test for urgent-priority notifications

**API being tested:**

- http://127.0.0.1:5000/webhook

```
36 @app.route('/webhook', methods=['POST'])
37 def webhook():
38     data = request.json
39     ticket_title = data.get('title')
40     ticket_description = data.get('description')
41     priority = data.get('priority')
42     if priority == 'high' or priority == 'urgent':
43         webhook_url = 'https://chat.googleapis.com/v1/spaces/AAAAVFGvcso/messages?key=AIzaSyDdI0hCZtE6vySjMm-WEfrq3CPzqKqqsHI&token=b7sKuUQVHQFNOLm'
44         send_notification(ticket_title, ticket_description, webhook_url)
45     return '', 200
46
```

**Inputs:**

- Request Method: POST
- JSON: { 'title': 'Test Title', 'description': 'Test Description', 'priority': 'urgent' }

**Expected Output:**

- HTTP Status Code: 200

**Actual Output:**

- HTTP Status Code: 200

**Result: Success**

```

32 def test_webhook_urgent_priority(client):
33     with patch('application.routes.requests.post') as mock_post:
34         # Mock the JSON payload sent by the webhook
35         mock_request = {'title': 'Test Title', 'description': 'Test Description', 'priority': 'urgent'}
36
37         # Simulate the POST request to the webhook endpoint
38         response = client.post('/webhook', json=mock_request)
39
40         assert response.status_code == 200
41         mock_post.assert_called_once()
42

```

## 5.3 Test Results:

All test cases have passed successfully.

```

===== test session starts =====
platform linux -- Python 3.10.12, pytest-7.2.2, pluggy-1.4.0
rootdir: /mnt/d/SE_Project/soft-engg-project-jan-2024-se-jan-11/Code/backend
collected 8 items

tests/test_GChat_webhook.py ... [ 37%]
tests/test_discourseAPIs.py ..... [100%]

===== 8 passed in 1.54s =====

```

## 5.4 Test Environment

- **Language:** Python
- **Testing Framework:** pytest
- **Dependencies:** Flask, patch, requests

## Conclusion

Testing is an essential aspect of the development process to ensure the application's reliability and functionality. By following the outlined test cases and procedures, we can ensure that the application meets its requirements and functions correctly in various scenarios.

# Milestone 6 - Final Submission

---

## 6.1 Summary of work done in different milestones

**Milestone 1:** The group identified the different users of the application in terms of primary, secondary, and tertiary users. Post the same user stories (by following SMART guidelines) written for the users identified earlier. 9 user stories with the mandatory features suggested by the project and some additional features as deemed necessary were compiled and submitted before the deadline.

**Milestone 2:** Two storyboards (using Comicgen and Google Slides) were depicted using comic strips:

- A storyboard was made from the perspective of a student
- A storyboard was made from the perspective of a support agent

Wireframes were designed for the application using the design heuristics which is learned in the Software Engineering course. Essential usability principles that were incorporated into our wireframe were as follows:

- Effectiveness: To ensure that users can carry out their work efficiently.
- Efficiency: Each user can effectively carry out their task in a minimal number of steps.
- Safety: Confirmation prompts to prevent any unwanted actions
- Learnability: Useful tool tips on mouse hover
- Memorability: Easy to use, consistent interface, which helps the user remember the same.

Some design heuristics that were followed during this phase were:

- Consistency: The layout of different pages is similar.
- Clean and Functional Design: Clutter-free, easy-to-recognize design that focuses on features and functionality with pleasant aesthetics.
- Ease of using the Applications: When students want to create a ticket, it will create a Post on Discourse as well. And if a ticket has a high/urgent priority level, then it will send a quick notification on Google Chat.

- **Show Status:** Students can see if support agents have viewed the ticket raised by them. After that, on answering the ticket, the ticket is displayed as closed (which can be reopened by the student if required), again showing the status of their ticket.
- **Provide Help:** Useful tool tips on mouse hover.

**Milestone 3:** A rough project schedule was drawn, keeping in mind the deadlines for each milestone and the amount of work that was required for the same. Sprints were planned for the different tasks. **Jira** was chosen as an appropriate project management tool and a Gantt Chart roadmap was plotted to keep track of the progress.

Different components were designed as UML diagrams using draw.io ([app.diagrams.net/](http://app.diagrams.net/)) to model the software system's Class Diagram.

Details of scrum meetings conducted until this stage were presented as “minutes of the meet” in the milestone 3 submission.

**Milestone 4:** API endpoints were described in the YAML file (Using OpenAPI 3.0 specification), and the coding for the backend of the application was also commenced during this milestone. The API specification was supposed to be the first version of the backend, which we believed could be subsequently changed based on any additional requirements/dependencies.

**Milestone 5:** Extensive testing (8 tests) was done using Pytest for the code written for the backend. Tests directly relating to user stories were documented in the Milestone 5 report using the format given in the instructions. All tests were passed. Frontend coding for the software was commenced in this milestone.

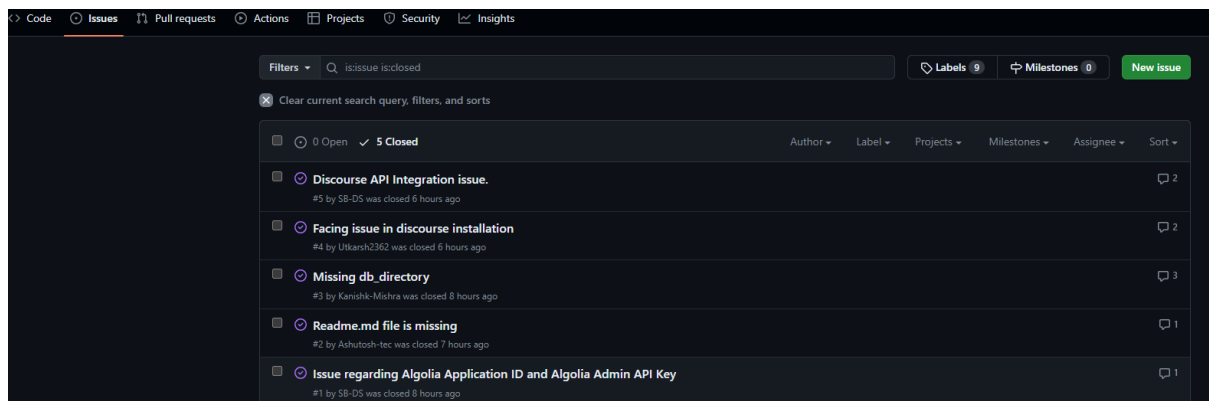
**Milestone 6:** Frontend was completed and fully integrated with the backend. Frontend-backend integration was tested manually for all the pages to ensure the software would behave as expected. The second version of the API documentation was completed based on the changes and new introductions to the API. Finally, a report was made for the completed project, and a demo video was recorded for the working application.

## 6.2 Code Review, Issue Reporting and Tracking

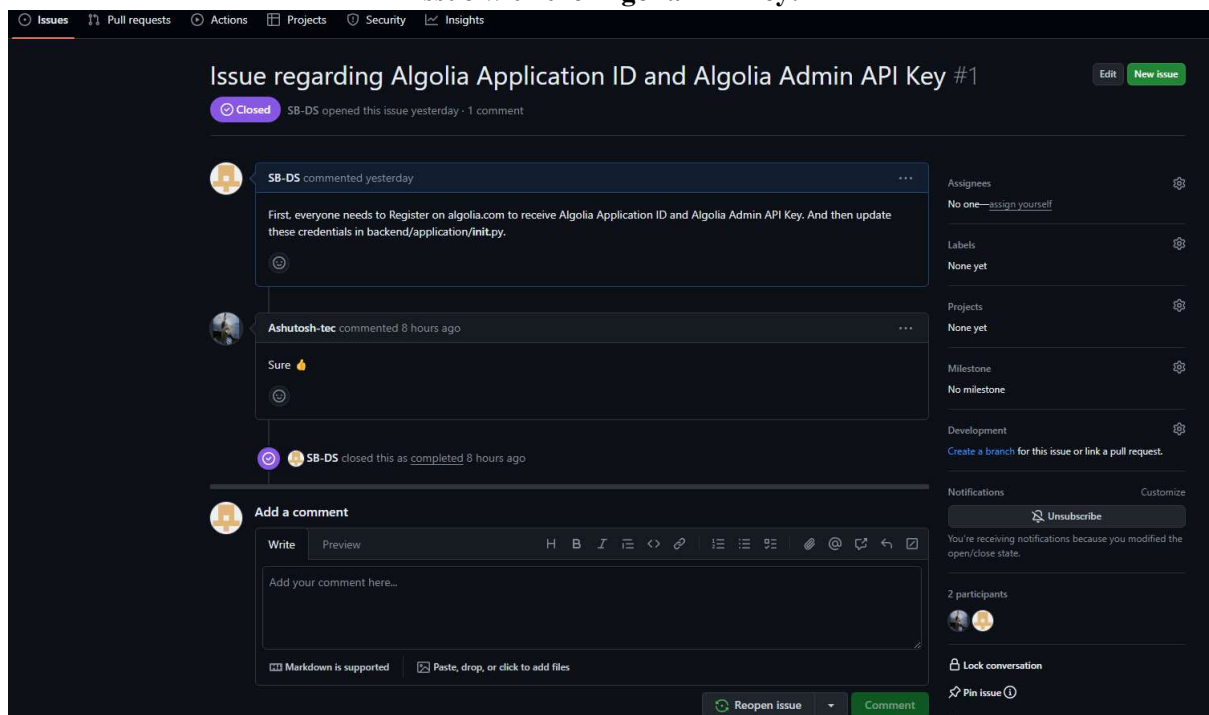
We divided up the tasks for this project. The tasks were independent of one another because of how they were chosen. There were merging problems when many components were introduced and used simultaneously, even though the tasks and coding components were independent. The relevant members reported these issues on GitHub, and the person responsible for resolving them updated and corrected the code. Various issues were created and resolved throughout the project.

ISSUES:

### Issues that were created:



### Issue with the Algolia API key:



## Issue about missing files:

Ashutosh-tec commented 8 hours ago

...

For code Readme file is not there, please check all.

SB-DS commented 8 hours ago

...

Yes, I am working on it.

SB-DS closed this as completed 8 hours ago

Add a comment

Write

Preview

H B I

Add your comment here...

Assignees

No one—[assign yourself](#)

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

[Create a branch for this issue or link a pull request.](#)

Notifications

Unsubscribe

You're receiving notifications because you modified the open/close state.

## Issue regarding missing db\_directory:

Issues

Pull requests

Actions

Projects

Security

Insights

Missing db\_directory #3

Closed

Kanishk-Mishra opened this issue 8 hours ago · 3 comments

Kanishk-Mishra commented 8 hours ago

...

Everyone, please be aware that the main file was not present during the initial run of the project. Please create it and add a "testdb.sqlite3" file before running the main.py.

adityar2309 commented 8 hours ago · edited

...

Issue resolved Test db created

adityar2309 closed this as completed 8 hours ago

SB-DS commented 8 hours ago

...

Yeah, the "testdb.sqlite3" file was created in the db\_directory folder.

SB-DS reopened this 8 hours ago

Kanishk-Mishra commented 8 hours ago

Author · ...

Assignees

No one—[assign yourself](#)

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

[Create a branch for this issue or link a pull request.](#)

Notifications

Unsubscribe

You're receiving notifications because you modified the open/close state.

3 participants

[Lock conversation](#)

[Pin issue](#)

## Issue with the Discourse Installation:

This screenshot shows a GitHub issue titled "Facing issue in discourse installation #4". The issue is marked as "Closed" and was opened by Utkarsh2362 6 hours ago. The issue description states: "Got error message after running the `d/ember-cli` command." The comment history shows Utkarsh2362 reporting the error, SB-DS providing instructions to use `d/exec yarn` before `d/rails s` and `d/ember-cli`, and Utkarsh2362 replying "Okay." The issue was closed by Utkarsh2362 as "completed" 6 hours ago. The right sidebar shows no assignees, labels, projects, milestones, or development branches. There are 2 participants.

## Issue with the Discourse API integration:

This screenshot shows a GitHub issue titled "Discourse API Integration issue. #5". The issue is marked as "Closed" and was opened by SB-DS 6 hours ago. The issue description asks: "How to add `headers` object to POST and PUT methods for Integration of Discourse API." The comment history shows SB-DS asking the question, Utkarsh2362 replying "Okay, I am working on it.", and SB-DS replying "Finally, I resolved this issue." The issue was closed by Utkarsh2362 as "completed" 6 hours ago, reopened by SB-DS 6 hours ago, and then closed again by SB-DS as "completed" 6 hours ago. The right sidebar shows no assignees, labels, projects, milestones, or development branches. There are 2 participants and options to lock conversation, pin issue, and transfer issue.

For these issues, the member responsible for solving (coding part of that issue) went through the issue details, updated the code, and pushed the changes to their respective branch, and then a pull request was created to merge code with the 'common' branch. Every member stayed up-to-date with the 'common' branch. Thus, the issues were resolved.

## 6.3 Implementation Details of the Project

### 6.3.1 Tools and Technologies Used

1. Technologies for the backend
  - a. Flask
  - b. Flask Restful (For creating API endpoints)
  - c. Flask SQLAlchemy
  - d. Pytest (Only for testing)
  - e. Swagger Editor (Only for API documentation)
  - f. Thunderclient (For checking API endpoints manually)
  - g. Mailgun - A Transactional email service provider is used to send relevant emails to the users
2. Technologies for the frontend
  - a. Vue 3 CLI
  - b. Vue Router
  - c. Vuex Store
  - d. JavaScript
  - e. Bootstrap (for aesthetics and styling)
  - f. HTML and CSS
  - g. ESLint to help with debugging the frontend
  - h. Axios and Fetch API for requests.
3. General technologies used
  - a. GitHub (for versioning, code management, tracking, reviewing, issues, etc.)
  - b. Algolia Search is used in the backend and frontend to create a smooth search experience
  - c. Jira (for project management)
  - d. ComicGen (for creating comics for storyboarding)
  - e. draw.io (app.diagrams.net/) (for creating UML diagrams)



### 6.3.2 Instructions to run our application

1. Install and run Discourse for development using Docker:

- a. Use the following link for installation and to run the discourse:

<https://meta.discourse.org/t/install-discourse-for-development-using-docker/102009>

2. To run our application:

- a. Clone the repository using:

`git clone https://github.com/Ashutosh-tec/soft-engg-project-jan-2024-se-jan-11.git`

- b. Change the directory to the “backend” directory inside the “Milestone-6-Final-Submission” directory using the command:

Linux: `cd Milestone-6-Final-Submission/Code/backend`

Windows: `cd .\Milestone-6-Final-Submission\Code\backend`

- c. Create a Python virtual environment using the command:

Linux: `python3 -m venv .venv`

Windows: `python -m venv .venv`

- d. Activate the virtual environment using the command:

Linux: `source .venv/bin/activate`

Windows: `.\venv\Scripts\activate`

- e. Install the requirements using the command:

`pip install -r requirements.txt`

- f. Now, on the terminal, type the following command to start the Flask server:

Linux: `python3 main.py`

Windows: `python main.py`

- g. In the same “backend” directory, in a new terminal inside the virtual environment, start the redis server by typing:

`redis-server`

- h. Similarly, start the celery worker in the “backend” directory inside the virtual environment by typing:

`celery -A main.celery worker -l info`

- i. Furthermore, start the celery beat in the “backend” directory inside the virtual environment by typing:

`celery -A main.celery beat --max-interval 1 -l info`

- j. For the frontend of the application, we would require Node.js and npm. We recommend installing the latest LTS version of Node.js from the following link:  
<https://nodejs.org/en>
- k. Once Node.js is installed and npm is working, **open a new terminal** and change the directory to the “Code” folder inside the “Milestone-6-Final-Submission” directory, which we saw earlier.
- l. Once inside the directory, change the directory to “frontend” by using the command:  
`cd frontend`
- m. Now, run the following command to install the necessary packages for the frontend:  
`npm install`
- n. After successfully installing the required packages, serve the frontend using this command:  
`npm run serve`
- o. The frontend server would be available at the following URL:  
<http://localhost:8080>

Please note that for the email functionality and search functionality to work, you would need API keys. To secure these API keys, they aren't part of the repository. You would need to request the keys from us.