EUROPEAN UNIVERSITY OF LEFKE

FACULTY OF ENGINEERING

Graduation Project 2

Bug Tracker

Brendan Chukwudi Chukwuemeka

194192

Effective coordination is crucial for any team or company working on a project, especially when transparent and responsive communication with consumers is required to address reported bugs. This project aims to provide various tools and workflows to enable teams to achieve these goals efficiently.

Supervisor

Prof. Cem Burak Kalyoncu

23rd May, 2024.

Table of Contents

1. Introduction

1.1. Problem definition

It is a well-known fact that one of the major problems faced by teams working on a project is the coordination of people and efficient allocation of resources. It is also known that finished projects, no matter how much time and resources have been sunk in their development usually have what consumers or users of the product consider to be bugs or complaints.

These complaints could arise from a failure to fulfill the task for which it was intended, design choices of the development team, temporary malfunctions or glitches in the system, confusion from customers as to the use of the products etc.

Customers would like these bugs to be resolved in as quick, effective and personalized a manner as possible in order to carry on with their activities, development teams would also like these bugs to be resolved quickly and efficiently as negative reviews on a team's products reduce the chances of further business as in the case of a for profit company or reduce the products effectiveness as in the case of a nonprofit.

It is also in the best interest of the development teams to personalize this bug resolution process as a lot of research has shown that customers are more likely to do business with a company if it offered personalized experiences [1] [2].

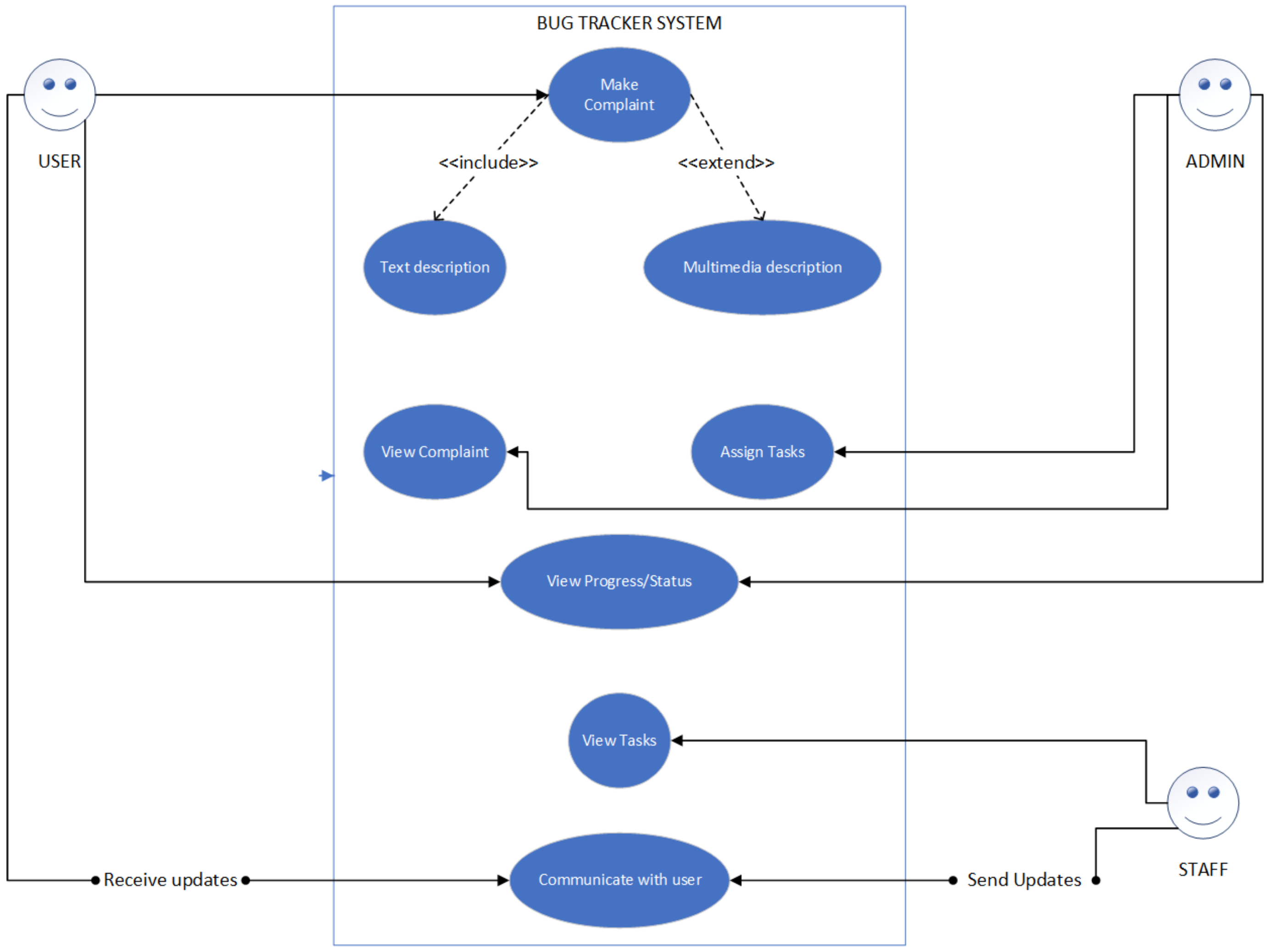
1.2 Goals

- A major goal of this project aims to facilitate the bug resolution process for the team(s) by providing features such as the ability to view problem resolution progress, issue ranking, issue assignment etc. in an easy-to-use interface that simplifies and expedites the process.

- In addition, this project aims to enable a more effective and transparent bug reporting and monitoring process for users by providing a platform which allows them to directly report bugs and complaints and view the progress of its resolution.

- This project also aims to facilitate team management by enabling workflows for efficient team creation, staff addition and removal, project state management as well as providing miscellaneous tools like time tracking, calendar events and discussions among staff.

USE CASE DIAGRAM



2. Literature Survey

Over time, numerous methods have been created and adopted by development teams for bug tracking purposes, these methods have significantly improved the way bugs are tracked and resolved however, there are issues with these methods or lack of features that this project seeks to address. In addressing these issues, three popular bug tracking systems will be analyzed and compared to this project namely Google's issue tracker, Zoho bug tracker and Live agent.

- **Issue tracker by Google:** Introduced in March of 2017, Google's Issue is a tool used for reporting [2] and tracking bugs and feature requests during product development at Google [3]. It is not publicly available for use as a consumer product and there are several restrictions placed on its use [3]. It offers features that allow users to create issues, view tasks assigned to them and communicate with other staff. This however is a case of specialized software as this tool is intended solely for google authorized users and partners [3], requires a certain level of technical ability and there is no feature to let the general public and users of Google products report bugs or view bug resolution progress. This project on the other hand will be a more general product which in addition to enabling quality bug tracking will add features that will enable customer bug reporting and communication.

- **Zoho bug tracker:** Created by Zoho Corporation Private Limited, Zoho bug tracker is a general-purpose bug tracking software that enables teams to create issues, assign issues to staff, view the progress of bug resolution and also provides various miscellaneous functions at the user interface and features level that assist in team management such as time tracking. A major shortcoming of this software and one which this project seeks to address is the opaqueness of the bug resolution process to the consumers. Users have no easy way of monitoring the progress of their bug reports or viewing the solution plan either as the only way they can be sure they are being attended to is if they are contacted by the staff by email, potentially long wait line chats, social media or phone.

- **Live Agent:** Established in 2006, live agent is the most popular help desk and is often rated as the number one issue tracking software on the market due to its many features in addition to its bug tracker. Like Zoho bug tracker above, it provides features for creating and assigning issues, it also allows for progress monitoring and also like Zoho bug tracker it falters in its inability to let customers consistently view progress of their bug reports or solution plan to deal with the bug. It only provides a standard communication stream where customers can only have one off conversations with staff regarding their issues.

3. Background Information

3.1 Required & Used software

- **Flutter:**
  Flutter is an open source, cross platform, software development kit created by Google. I will be using flutter due to its ability to create beautiful and capable cross platform applications. It has very comprehensive documentation. Its use is also very widespread leading to lots of accessible resources and a great community.

- **Android studio:**
  Android studio is the IDE (Integrated development environment) I will use due to it having official support by flutter, being the official IDE for the android operating system and its ease of use.

- **MySQL:**
  MySQL is my database of choice due to its ease of use, ability to handle complex queries and simple integration with flutter applications.

3.2 Other Software

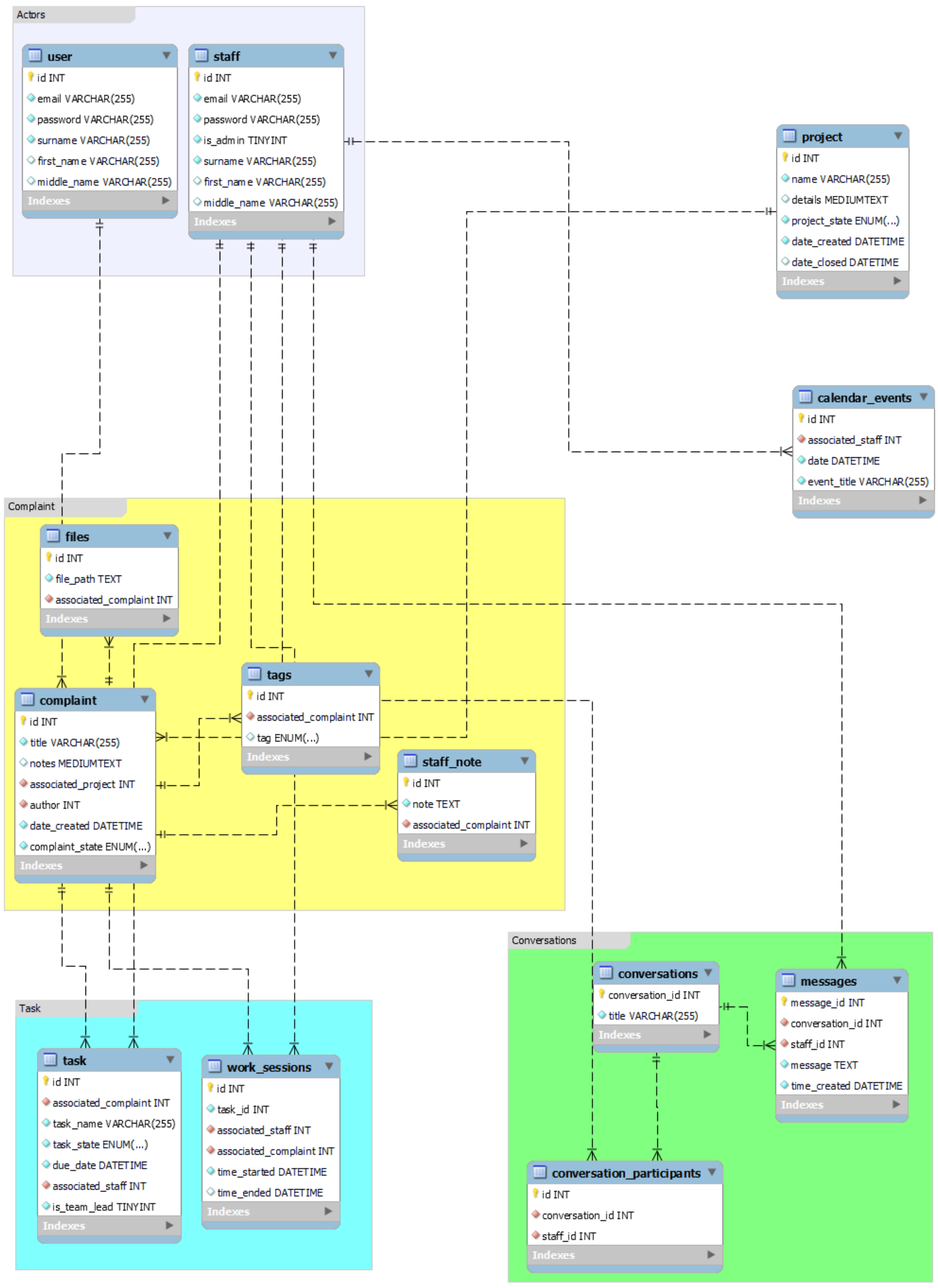
- **Git:**
  I will use the git version control system because it’s a very capable and popular version control system.

4. Design Documents

4.1 Database Entity Relationship Diagram

This module shows the structure of the database utilized for the project; the tables are arranged in groups; each group contains tables that interact together and satisfy specific functionalities within the project. These functionalities include handling actors, complaints, task, conversations, project or calendar events related.

As can be seen from the diagram, all connected tables are related by one-to-many relationships indicated by the horizontal equals sign on one end (representing the "one" side) and "crows-feet" (representing the "many" side) on the other. For example; the complaint table in the complaint group is connected to the files table which is also in the complaint group; one complaint can have many files.



5. Methodology

In tackling this project, my first step was to get done with the user interface as I correctly assumed looking back that it would take the most time as such, I will be explaining that first. While explaining the UI, I will first tackle the Admin UI then the User UI and lastly the staff UI since the “normal” staff UI is a derivative of that of the admin.

My main inspiration while designing the UI was the Zoho Bug Tracker application developed by Zoho Corporation Private Limited and the little similarities between the UIs are as a result of that; for example, they follow an overall black and orange theme and similarly shaped containers for the admin’s homepage.

ADMIN UI

The admin section consists of seven pages, the home page, discuss page, calendar page, project page, bugs page, tasks page and the staff page. Each page can be navigated to by navigation bars on the left of each page. Every page contains a toolbar at the top with buttons enabling the admin to add new staff or projects as well as change their password.

Home Page

When the admin logs in, they see the home page first. This page contains a greeting with their name and company name; Then two “fast access containers” showing the numbers of open and closed bugs.

Next comes 4 containers showing a preview of their tasks, their tasks due today, their overdue tasks and all bugs.

The idea behind this arrangement is that the homepage shows a preview of what to expect; a way to scroll through any new tasks for maybe any which catches their eye or what the state of the task is. This concept is also manifested in the codebase, the UI component showing a single task in each task section in the home page is a version of a taskPreviewCard made for the homepage as opposed to a taskOverviewCard used in the tasks page, the idea being that the preview card is less important than an overview card and therefore shows less detail. The bugPreviewCard is also different from the bugOverviewCard for the same reason.

The taskPreviewCard consists of a column of two rows; the first row consists of the title of the task on one end and the due date on the other, the second row consists of the associated bug on one end and the state of the task on the other.

The bugPreviewCard also consists of a column, first the bug title and date created on opposite ends of a row and then the related project beneath the bug title.

Clicking on either the open or closed bugs fast access container leads to the bugs page, clicking on any task or bug leads to the task details page or bug details page respectively.

The page has also been designed to rearrange itself depending on the size of the screen so no information is cut out and the structure remains.

Discuss Page

Next is the discuss page, this page enables the admin start discussions with other staff members, on this page the admin can also see every conversation they are a part of.

At the top of the page, there is a new discussion button and a search bar, on clicking the new discussion button, a side page taking up about three quarters of the main page loads up showing a form to enter the topic of the discussion and beneath that all staff are shown with checkboxes enabling the admin to select as many participants as necessary.

Failure to add the topic generates an error stating the topic is necessary and failure to select participants generates an error letting them know 2 participants are necessary themselves included. Then they can click the start button.

Beneath the new discussion button is a table showing the list of discussions the current staff which is the admin currently is a participant of, the table shows the topic and the participants; participants are displayed as a scrollable horizontal row of circles containing initials of the staff, the idea being that if there are a lot of them, they will be able to fit and the tooltip that shows when they're hovered upon gives their full name. If there are no discussions or if a searched discussion doesn't exist, a stylized placeholder text "NONE" is shown instead.

Clicking a discussion leads to the messages page where the messages for that discussion are displayed. Each message is displayed in a message bubble with the sender's name on the top and the contents of the message in a sky-blue or white bubble depending on if the message is from the admin or not respectively.

Beneath the messages is a text field for entering messages and a send button, clicking enter after adding the message will also send the message.

Calendar Page

This page enables the admin to view a calendar, they are also able to add activities to that calendar, activities that will be indicated by dots on that day in the calendar.

At the top of the page is the add activity button, clicking this button leads a side sheet to come in on the right containing a choose date button and a text field to add the activity. On clicking the choose date button, a popup allowing the admin to select a date comes up.

Failure to select a date or add an activity leads to a warning showing that they are necessary.

Beneath the add activity button is the calendar itself, the calendar can be modified to only show two weeks, one week or the default one month. Days with an event have small circular indicators on them with their number denoting the number of events on that day. Clicking on them shows the events beneath the calendar, the events are removable by clicking the X button shown when the event is hovered upon.

Projects Page

This page enables the admin to view available projects, they are also able to sort the project by the project state.

At the top of the page is a drop-down menu, this menu enables the projects to be sorted by state, the states are open, postponed, cancelled, closed and a non-state option: all projects.

The projects are shown in a table with the columns being project ID which shows the project id, project which shows the project name, status which shows the project status, bugs which shows with the aid of a linear progress indicator what percentage of bugs in the project are completed, created which shows the date the project was created and completed which shows the date the project was completed.

Clicking on a project leads to the project details page for that project. At the top of the project details page there is a row with the project id on one end and the date created on the other end. Underneath that another row shows the project name on one end and if it's a closed project the date closed on the other end.

Next are the project details, this is a section that describes the project.

Next is the status of the project, this shows 4 cards with the statuses either greyed out or colored based on the status of the project. Just beneath that is a button for updating the status, clicking the update status button reveals 4 check buttons in a radio format; once one is selected and the done button is pressed, the page refreshes itself with the updated status.

Finally, beneath that is a bugs section showing the related reported bugs, this is shown as a list of bugPreviewCards. Clicking on a bug leads to the bugs details page.

Bugs Page

This page enables the admin to view available bugs, they are also able to sort the bugs by bug state.

At the top of the page is a drop-down menu, this menu enables the bugs to be sorted by state, the states are pending, acknowledged, in progress, completed and a non-state option: all bugs.

The bugs are shown in a table with the columns being bug ID which shows the bug id, bug which shows the bug name, project which shows the associated project name, author which shows the email of the reporter of the bug, created which shows when the bug was created, progress which shows with the aid of a linear progress indicator what percentage of tasks related to the bug are completed, status which shows the and tags which optionally shows a scrollable horizontal list of tags added by the admin.

Clicking on a bug leads to the bug details page for that project. At the top of the bug details page is the update button, clicking the update button leads to the bug detail update page which is also a right side-sheet. The bug detail update page is for adding, updating or removing tags, teams or tasks.

At the top of the bug detail update page are the tags, these are shown as six clickable cards, if tags have already been given to the bugs the corresponding tags are lit up else, they are greyed out. Beneath that is the team section, if a team has been allocated tasks for the bug they are shown, the team lead first then the team members else, if a team hasn't been allocated then as the add team member button is clicked a "Task assignment form" consisting of a dropdown menu of staff and text field for the task is added. Then clicking the done button updates the tasks and tags list.

Back at the bug detail page, beneath the update button is a row with the bug ID on one end and the date created on the other end.

Beneath that is the author's email then beneath that the associated project, beneath that the bug's title. Below that the complaint noted from the author then optionally the files submitted by the author if any.

Beneath those is the status and beneath that the tags both are the same show color else gray out format used throughout the project.

Beneath that shows the tasks for the assigned team if any team lead first, the tasks are shown as taskPreviewCard.

Finally, beneath that are the staff notes which shows notes/solution plans intended for the author of the bug to see updates to the bug resolution process.

Tasks Page

The tasks page enables the admin to see all of their available tasks, they are also able to sort tasks by state.

At the top of the page is a row with the drop-down menu to be able to sort by state on one side and the search bar on the other side.

Beneath that is the list of tasks, each task is displayed as a taskOverviewCard. A taskOverviewCard consists of a column of data, first is a row with the related complaint id on one end and the due date on the other, next is the related complaint, next is a row with the task on one end and the status on the other end and lastly the related project name.

Clicking the taskOverviewCard leads to the task details page, at the top of the page is a row with the related project on one end and the update button on the other if the admin is not viewing the task detail page by clicking on a task on the bug detail page, this arrangement is so only an assignee can interact meaningfully with their task, clicking the update button loads a side-sheet from the right; at the top of this task detail update page, a checkbox to mark the task as completed is visible if the task is not already.

Beneath that is the transfer task dropdown where task can be transferred by selecting a different staff.

Beneath that, the checkbox to mark the entire complaint as visible if the staff is the team lead and then beneath that is a text field for sending notes to the reporter, also if the staff is the team lead.

Finally, beneath that is the done button.

Back to the task detail page, beneath the update section is another row with the task ID on one end and the associated complaint's ID on the other.

Beneath that has a row with the associated complaint on one end and the due date on the other.

Beneath that are the complaint notes submitted by the reporter, beneath that are the files and finally beneath that in bold letters is the actual task they have been given.

Beneath that, if the task has not been completed or transferred; there's a row consisting of the start session/end session button on end, the time spent on the current task if there's currently a session in progress in the center and the open sessions log button on the other end.

Clicking the open session log button leads to the sessions log page, at the top of the page is a switch view button to enable switching between the default table view and a more graphical view of the sessions log.

The default table view is a table consisting of index, start date, end date and time taken columns; the total completed work sessions is beneath that. The graphical view consists of a timeline of the work sessions on the left and the total completed work sessions on the right.

Finally, beneath the time tracking feature interface is the team for the project showing every staff member assigned to the project.

The page has been designed for security and data integrity in mind as features are only visible when necessary and the admin cannot tamper unnecessarily with tasks.

Staff Page

This page is for employee management, the admin can view staff and manage, modify or delete their data.

At the top of the staff page is the search bar for searching staff based on their name, id or email.

Beneath that is the list of staff, each staff is denoted with a staffOverviewCard. A staffOverviewCard consists of a circle avatar with the staff's initials then a column with the staff id on top, full name then their email at the bottom.

Clicking on a staffOverviewCard leads to the staff details page, at the top of the page is a row with their id on one end and a delete button on the other, clicking the delete button leads to a confirmation popup warning that the action is irreversible and providing options to either proceed or cancel the action.

Beneath that is their email and an update button, clicking the update button reveals a text field intended for their updated email as well as done and cancel buttons below.

Beneath that is their name and also an update button, clicking the button reveals three text fields for their updated surname, first name and middle name as well as done and cancel buttons.

Finally, beneath that is a section showing all of their tasks shown as taskPreviewCards.

Next up is the user UI.

USER UI

The user UI consists of one main page and 2 auxiliary pages. At the very top of the page before the main content is a toolbar with their initials in a circle avatar at the top, clicking that provides a route to change their password.

When the user logs in they are directed to the home page. At the top of the page is a greeting "Welcome [user's name]" beneath that is a row; on one end is a dropdown menu to be able to sort their complaints by their state; states being pending, acknowledged, in progress, completed and a non-state all complaints. In the middle is a search bar to be able to find specific complaints and on the last end is the new complaint button.

Clicking the new complaint button leads to the new complaint page, at the top of the page is a text field intended for the project ID of interest, then beneath is a text field for the bug title, beneath that is the text field for optional notes regarding the complaints; beneath that is a link for optionally adding files and finally below that is the submit button

Beneath the new complaint button is the list of all complaints submitted by the user, each complaint is represented by a bugOverviewCard. Clicking on a bugOverviewCard leads to the complaint detail page.

At the top of the page is a row with the complaint id on one end and the date created on the other.

Beneath that is the associated project. Beneath that is the actual complaint in bold letters.

Beneath those are the complaint notes if any. Beneath that are the files if any.

Beneath that is the status of the complaint represented as 4 “table” that are either lit up or greyed out based on the complaint’s status.

Finally, beneath that are the staff notes which are notes sent by the team lead of their current task with information for the reporter.

STAFF UI

The staff UI consists of pages already described in the admin section; the task page, discuss page and calendar page with all the same functionalities described.

MISCELLANEOUS

New Staff

For the admin to add a new staff, they have to click the + button in their toolbar, then click on new staff on the pop up that appears.

This leads to a side sheet sliding in from the right and taking up about 75 percent of the page. At the top of the page is a check box to optionally mark the new staff as an admin.

Beneath that is the text field for their surname, beneath that a text field for their first name (optional), beneath that is another text field for their middle name which is also optional and beneath that is a text field for their email.

Failure to input the non-optional fields leads to an error beneath the corresponding text field.

After successfully filling the form, clicking done shows a popup indicating that the staff has been created success fully and their default password is 000000 i.e., six zeros; a copy id button is also provided.

New Project

In order for the admin to start/add a new project. they have to click the + button in their toolbar, then click on new project on the pop up that appears; this leads to a side sheet sliding in from the right and taking up about 75 percent of the page.

At the top of the page is a text field for the project name and beneath that is a text field for optional project details.

Failure to input the non-optional fields leads to an error beneath the corresponding text field.

After successfully filling the form, clicking done shows a popup indicating that the project has been created successfully, it also shows the project id and a copy id button.

Update Password

This action is possible for all actors by clicking on their initials at the extreme right of the toolbar and clicking update password. Doing this causes a side sheet to slide in from the right to show a page that takes up about 25 percent of the screen.

At the top of this update password page is a text field for the previous password, and beneath that a text field for the new password. Failure to enter any leads to an error text beneath the text field.

Clicking the update button beneath the text fields shows a success popup if information is entered correctly.

I will now be explaining the details behind the implementation. I will be explaining them following how they would appear to the actor. I will not be explaining the UI any further and only explain functionalities present in each file.

The database functions are all present in a file called db, the db files contains a class called DB the class contains a _conn property representing a connection, it contains a connect method that uses user input settings to connect, an isConnected method that checks if the _conn property is non null and a close method that closes the connection. The rest are for MySQL database queries and will be explained as they are used.

Main.dart

The entry file is the main file, at the beginning it checks to see if the current database project is already connected and if so, it closes it before reconnecting again asynchronously, if there is a problem with connecting to the database a debug error message is provided to the console. This disconnection and reconnection process is to prevent multiple connections and use just one. Once the database is connected; it first provides “change notifiers” to the rest of the project then runs the application. Change notifiers are a concept in flutter that enables objects to watch certain properties and then “react” based on changes; each change notifier provided will be explained as the first use is reached.

Sign_in.dart

Next is the sign-in file, the UI is built as described above. This file is used as the point of attachment for a window watcher package that enables the database to be closed asynchronously when the window is closed.

The sign in authentication process is as follows, Once the inputs have been validated I.e., once values have been entered for the email and password fields, the database function getStaffDataUsingEmail function gets called that uses the input email to retrieve the staff information from the database. If data for such staff is available then the password is checked for correctness using the authenticatePasswordHash function; this function takes in the entered password and the hashed password gotten from the staff data and returns true if they are the same when the entered password is hashed using the same sha512 hashing function; if this fails then an incorrect password error is provided, if it is successful then a global actor Id and global actor full name variable is set and the actor is redirected to either the admin main page or staff main page based on the isadmin property gotten from the database. However, if no such staff exists the process is repeated but for a user using the getUserDataUsingEmail function and redirecting to the user main page on successful authentication. If there is also no such user then an invalid email error is provided.

This file also contains the sign-up button that leads to the sign-up page for new users/reporters.

Sign_up.dart

Once the user inputs have been validated (i.e., values have been entered for the required fields such as email, password, and surname), the sign-up process proceeds as follows:

1. Database Connection: The system attempts to connect to the database.
2. Email Validation: The getUserDataUsingEmail function is called with the input email to check if a user with that email already exists in the database.
3. User Creation: If no existing user is found (i.e., actorData is null), the system proceeds to create a new user with the provided details (surname, first name, middle name, email, and password) using the addNewUser function.
4. Setting Global Variables: If the new user is successfully created (i.e., newUserID is not null), the following global variables are set:
    - globalActorID is set to the new user's ID.
    - globalActorName is set to the full name constructed from the provided surname, first name, and middle name.
    - actorIsAdmin is set to false.
    - actorIsStaff is set to false.
    - actorIsUser is set to true.
5. Navigation and Confirmation: The user is then navigated to the ComplaintPage, and a confirmation message is shown using ScaffoldMessenger, indicating that the new user was created successfully.
6. Error Handling: If the user creation fails (i.e., newUserID is null), an error message is shown, prompting the user to try again later. If a user with the provided email already exists, a message is shown to inform the user that the email is already in use.

Error Handling

If the email already exists in the database, the user is notified with an error message indicating that the user already exists. If the user creation fails due to any other reason, an error message is shown to try again later. These error messages are displayed using ScaffoldMessenger, which shows a SnackBar with the appropriate message.

Next, I will be explaining the admin section, the entry point for the admin section is the admin_main_page.dart file

Admin_main_page.dart

Admin Main Page

The AdminMainPage serves as the central hub for administrative operations, offering a streamlined interface with a static navigation rail and adaptable page content.

Functionality

Within the _AdminMainPageState class, the widget's state is managed to facilitate dynamic updates. This ensures seamless transitions between different sections of the application.

Custom Navigation Rail

The CustomNavigationRail widget orchestrates the presentation of the navigation rail, allowing users to navigate through various sections of the application effortlessly.

Key Features

- AppBar Control: It enables the toggling of an app bar's visibility, providing users with an unobtrusive navigation experience.
- Navigation: The rail presents a series of navigation options, allowing users to select their desired destination with ease.

Select Page Function

The selectPageAdmin function dynamically determines the appropriate page widget based on the current selection index. This ensures that users are directed to the relevant content according to their chosen destination.

Overview Provider

The Overview class functions as a provider, managing the application's overview state. This includes facilitating smooth transitions between specific pages and notifying listeners of any state changes to prompt widget updates as required. This provider is part of the ones in the main..dart

Notification Mechanism

By extending the ChangeNotifier class, the Overview class ensures that any changes in the application's state are communicated to relevant listeners, guaranteeing a responsive and intuitive user experience.

The first admin page to be explained is the home page

Admin Home Page

The home page contains the fast access container and large container classes described in the UI section.

An enum defines the type of fast access containers there are: openBugs and closedBugs, when building a fast access container, the home page passes an enum to the class since its constructor requires one. A fast access container uses a future builder which defers the building until an asynchronous function is done calling. The function being called in this instance is getBugNumbers function which calls a load Complaints source function which loads a complaints source list with complaints, the getNugNumbers function then returns a list with the length of openbugs and the length of closed bugs. It does this by checking the list of complaints for which either has state completed or not.

When the fast access container is clicked, the entire page is moved to the bugs page. It does this by calling the switch to bug function in the overview class, this function then changes the selected index variable to that of the bug (5 in this case) and then notifies all listeners, the main page in this case.

The same enum approach is taken with the large containers, the 4 enums are myTasks, tasksDueToday, overdueTasks, allBugs. Each is passed while drawing the 4 large containers; if they are task oriented like the first 3 enums then the function called by the future builder is the loadTaskSourceByStaff else it is the loadComplaintsSource function. For task related large containers; they are then sorted by state and the appropriate tasks are used to draw the large containers else if it is bug related the complaints are then used to draw the container.

Discuss page

A crucial part of this page is the discuss class which has an integer id property, a string title property and a participants property which is a list of Staff objects.

DiscussPage and SearchBar Interaction

The DiscussPage widget displays a list of discussions and includes a search bar to filter them. As users type into the SearchBar, the onChanged callback updates the searchBarString variable, and the page redraws to show only matching discussions.

Drawing the Table

The discussions table is built using FutureBuilder and custom row widgets. FutureBuilder handles asynchronous data loading with the loadDiscussionsSource function, showing a loading indicator while waiting. Once data is ready, it filters discussions based on the searchBarString.

Building Table Rows

The buildTableRow function creates each table row. It displays the discussion title, which navigates to MessagesPage when clicked, and shows participants in a horizontally scrolling list with each participant represented by a CircleAvatar and a tooltip with their full name. The headers are created by buildTableHeaders.

Loading Discussions Source

The loadDiscussionsSource function fetches discussions from the database. It retrieves discussions involving the specified staffID, fetches their titles and participant information, and constructs Discuss objects, which are then stored in discussionsSource.

The actor types into the SearchBar, updating the search string and redrawing the UI. FutureBuilder manages data loading, showing a progress indicator until data is ready. The loaded discussions are filtered and displayed in a table, with clickable rows for detailed views.

Process of starting a new discussion

When the new discussion button is clicked, the new discussion page is opened as described in the UI section.

OnPress Algorithm and Participant Selection

When a user interacts with the NewDiscussion widget, they can enter a topic and select participants for the discussion. The topic is entered into a TextFormField, and participants are selected using a ListView.builder that displays each staff member with a checkbox. As the user checks or unchecks these boxes, the selectedStaff list is updated accordingly. The setState function is called to refresh the UI whenever changes are made to the selection.

When the "Start" button is pressed, the onPress callback is executed. This callback first validates the form to ensure that the topic is not empty. If the topic passes validation, the callback then checks if at least two participants, including the current user (globalActorID), are selected. If this condition is not met, an error message is displayed by setting participantChoiceError to true and calling setState to refresh the UI and show the error.

If the form validation and participant check pass, the startDiscussion method from the DiscussionUpdates provider is called to create a new discussion. After successfully starting the discussion, the form is reset, and a confirmation message is displayed using a SnackBar.

startDiscussion Function

The startDiscussion method in the DiscussionUpdates class is responsible for initiating a new discussion. It takes the topic and list of participants as parameters and calls the addDiscussion method from the database utility to insert this new discussion into the database. If the insertion is successful, notifyListeners() is called to inform any listeners about the new discussion the discussion page in this case which cause it to redraw. If there is a failure, an error message is logged.

Database Integration with addDiscussion Function

The addDiscussion function manages the actual database operations. First, it inserts the discussion title into the conversations table and retrieves the conversationID of the newly created discussion. Then, it inserts each participant into the conversation_participants table using this conversationID. If all participant insertions are successful, the function returns true. If any insertion fails, it returns false.

Messages Page

Clicking on a discussion leads to the messages page for that discussion.

Initialization and Scrolling

When the MessagesPage is initialized, the initState method ensures the message list is scrolled to the bottom so that the latest messages are immediately visible to the user. This is done using a WidgetsBinding callback that jumps to the maximum scroll extent once the widget tree is built.

Building the UI

The page layout consists of two main components: the message list and the message input area.

1. Message List:
    - The message list is built using a FutureBuilder, which asynchronously loads messages via the retrieveMessages function. While the messages are being fetched, a CustomCircularProgressIndicator is displayed.
    - Once the messages are loaded, they are displayed in a reverse order (latest at the bottom) using a ListView.builder. This list view is controlled by a ScrollController to manage scrolling behavior.
2. Message Input:
    - Below the message list, there is a text field for typing new messages and a send button. The text field is linked to a TextEditingController to handle the input.
    - When the send button is pressed or the enter key is used, the sendMessage function is triggered. This function checks if the input is non-empty, then calls the addMessage method from MessageUpdates to send the message and finally clears the text field.

Sending Messages

The sendMessage function is crucial for handling message submission. It first checks if the text field is empty, clearing it if it is. If not, it invokes addMessage from MessageUpdates with the sender's ID, the discussion ID, and the message text. After the message is sent, the text field is cleared to be ready for the next input.

MessageUpdates Class

The MessageUpdates class manages the process of sending messages. Its addMessage method interacts with the database:

- It sends the message details to the database using the db.addMessage method.
- If the message is successfully added to the database, it calls notifyListeners to update the UI.

Database Operations

The db.addMessage function handles the actual insertion of messages into the database:

- It inserts a new message into the messages table with the relevant details: discussion ID, sender ID, message content, and the current timestamp.
- The function returns true if the insertion is successful, or false if it fails.

Retrieving Messages

The retrieveMessages function fetches messages for the specified discussion:

- It queries the database for messages related to the discussion ID.
- For each message retrieved, it creates a MessageBubble object, which includes fetching sender details to display the sender's full name.
- These MessageBubble objects are then returned for display in the message list.

MessageBubble Class

The MessageBubble class represents individual messages:

- Each MessageBubble displays the sender's name and the message text.
- It adjusts the alignment of the message based on whether the message was sent by the current user (isMe property).

Putting It All Together

The workflow of the MessagesPage can be summarized as follows:

1. Initialization: The page ensures the message list is scrolled to the bottom on load.
2. Displaying Messages: The FutureBuilder loads messages using retrieveMessages, displaying them in a list view.
3. Sending Messages: When a message is sent, sendMessage calls MessageUpdates.addMessage, which interacts with the database to add the message.
4. Database Interaction: The db.addMessage function inserts the message into the database, and retrieveMessages fetches and processes these messages into MessageBubble objects for display.

Calendar Page

Calendar Page State Management

Initialization:

Upon initialization, the Calendar Page establishes default settings and initializes the current date as the focused day. This ensures that users are presented with a familiar starting point when accessing the calendar.

Disposal:

Proper disposal of resources is essential for optimizing memory usage and preventing memory leaks. When the Calendar Page is no longer in use, resources are disposed of efficiently to maintain application performance.

Event Handling and Display

Day Selection:

The Calendar Page dynamically responds to user interactions when selecting specific days. Upon selecting a day, the interface updates to display the associated events, providing users with a comprehensive view of their schedule for that day.

Range Selection:

Long-pressing on the calendar enables users to select a range of dates, offering enhanced flexibility for managing events over multiple days. The interface adjusts accordingly to provide a clear indication of the selected date range and its associated events.

Future Builder Usage for Asynchronous Operations

Loading Events:

As calendar events are retrieved asynchronously from the database, the FutureBuilder widget displays a loading indicator, indicating to users that data is being retrieved.

Building Calendar:

Once events are loaded, the TableCalendar widget is constructed, presenting users with an interactive calendar interface populated with their events. This seamless integration ensures a smooth user experience when navigating through dates and viewing associated events.

Calendar Utilities for Date Manipulation

Event Class Definition:

- The Event class represents individual calendar events, encapsulating essential attributes such as unique identifiers (id) and titles (title).

Date Functions:

Utility functions for generating hash codes and constructing date ranges are essential for date manipulation throughout the application. These functions streamline date-related operations, ensuring consistency and accuracy in event handling.

Constants:

Defining constants for today's date, the first day of the calendar, and the last day of the calendar establishes reference points for date calculations and ensures uniformity in date handling across the application.

Database Interactions for Event Management

Load Calendar Events Function:

The loadCalendarEvents function retrieves events associated with a specific staff member from the database. This function populates the calendar with relevant data, providing users with a comprehensive overview of their scheduled activities.

Get Calendar Events Function:

The getCalendarEvents function fetches calendar events for a staff member, facilitating event retrieval and display within the application. By leveraging database queries, this function ensures timely and accurate retrieval of event data.

Add and Delete Calendar Activity Functions:

The addCalendarActivity and deleteCalendarActivity functions enable users to seamlessly add or remove events from their calendars. These functions interact directly with the database, ensuring data integrity and consistency in event management operations.

Adding a new activity

Clicking the add activity button leads to the new activity page.

Detailed Explanation:

1. Date Selection:
    - Users can select a date for the activity by tapping on the "Choose date" button. This action triggers a date picker dialog where users can select a date from the calendar.
    - If a date is not selected before attempting to add the activity, an error message prompts the user to choose a date.
2. Activity Input:
    - Once a date is chosen, users can input details about the activity in the text field provided.
    - The input field expands dynamically to accommodate longer descriptions if necessary.

- Validation ensures that the activity field is not left empty before submission.

3. Submission:

   - Upon completion of the form, users can submit the activity by tapping the "Add activity" button.
   - The form undergoes validation to ensure that both the date and activity fields are filled.
   - If validation passes, the activity details are saved, and an attempt is made to add the activity to the database.

4. Database Interaction:

   - The addCalendarActivity function is called to add the activity to the database, passing the staff ID, selected date, and activity title as parameters.
   - Upon successful addition, the parent widget is redrawn to reflect the new activity.
   - If the addition fails, a notification informs the user to try again later.

5. Feedback to User:

   - After submitting the activity, the user receives feedback via a Snackbar notification.
   - If the activity is added successfully, a confirmation message is displayed.
   - In case of failure, an error message indicates that the addition process was unsuccessful and suggests trying again later.

Projects Page

Filtering Projects:

The filterProjectsSource function adjusts the displayed projects based on the selected filter value. When "All Projects" is selected, it returns all projects; otherwise, it filters projects by the selected state.

Loading Projects:

Within the loadProjectsSource function, projects are fetched from the database and transformed into a list of Project objects. It employs the getAllProjects function from the database utility to retrieve project data. For each retrieved project, a Project object is created and added to the list of processed projects. If no projects are found, the projects source list is set to an empty list.

Future Builder for Projects:

Using the FutureBuilder widget, projects data is asynchronously loaded and the UI is updated accordingly. During the loading process, a circular progress indicator signals that data is being retrieved. Once loaded, the projects are filtered based on the selected dropdown value and displayed in a tabular format. If projects match the filter criteria, a table with project details, including ID, name, status, etc., is constructed. If no projects meet the filter criteria, an empty screen placeholder indicates the absence of projects.

Change Notifiers for Project States:

The ComplaintStateUpdates, TaskStateUpdates, and ProjectStateUpdates classes facilitate the update of complaint, task, and project states, respectively. Each class offers methods to modify the state of the corresponding entity in the database. Upon a successful state update, listeners are notified to update the UI. If an error occurs during the update process, a debug message is printed to highlight the failure.

Building Table Rows:

Through the buildTableRow function, a table row for a project is constructed, comprising its ID, name, status, completion percentage, creation date, and completion date (if available). Project details are formatted into ListTile widgets for each cell in the row. The project ID is made tappable for users to navigate to the project detail page. Completion percentage is displayed using a custom linear percent indicator, and date values are formatted into readable string formats.

Building Table Headers:

The buildTableHeaders function generates table headers for various project details, such as ID, name, status, etc. Each header name is formatted as a ListTile widget with a specified text style, forming the header row of the table.

Calculating Project Completion Percentage:

In the getPercentageOfProjectCompleted function, the completion percentage of a project is calculated based on the number of completed complaints associated with it. Complaints related to the project are fetched from the database using the retrieveComplaintsByProject function. Following this, the ratio of completed complaints to total complaints is computed, and the percentage value is returned.

<u>Project details page</u>

Project State and Date:

- The getCurrentProjectState function plays the main role in displaying the current state of the project.
- By querying the database based on the project's ID, this function asynchronously fetches the project's current state.
- The retrieved state is displayed on the page using a chip component, which visually indicates whether the project is in an open, postponed, cancelled, or closed state.
- The color of the chip dynamically adjusts based on the project's state, offering immediate visual feedback.

Updating Project State:

- Users can interact with the radio buttons presented on the page to modify the project's state.
- Upon selecting a new state and confirming the action, the updateProjectState function is called to execute the necessary database update operation.
- This function accepts parameters such as the project ID and the newly selected state, initiating an update query to reflect the changes in the database.
- Upon successful completion of the update process, listeners are notified to trigger a UI refresh, ensuring that the updated project state is accurately reflected in real-time.
- In the event of any anomalies or errors encountered during the update operation, a debug message is generated to facilitate troubleshooting and debugging.

Bugs Related to the Project:

- The loadComplaintsSourceByProject function is responsible for gathering all bug reports associated with the chosen project.
- Through a database query tailored to the project ID, this function retrieves the relevant bug reports asynchronously.
- Once the bug reports are successfully loaded, they are presented on the page in the form of bug preview cards, offering users a concise overview of each reported issue.

Data Loading and State Retrieval:

- To prevent UI blocking and maintain responsiveness, data loading tasks, such as fetching the project's current state and retrieving associated bug reports, are performed asynchronously.
- Leveraging asynchronous programming techniques, these functions execute database queries efficiently to fetch the required data based on the project ID.
- Upon receiving the requested data, the associated UI components are updated dynamically to reflect the most up-to-date information retrieved from the database.
- By employing asynchronous methodologies, users can seamlessly interact with the application while data retrieval operations are underway, ensuring a smooth and uninterrupted user experience.

<u>Bugs Page</u>

The Complaint class is essential for managing bugs on the Bug Page

Properties:

- ticketNumber: Unique ID for each bug.
- complaint: Description or title of the bug.
- complaintNotes: Additional bug details.
- complaintState: Current bug status (e.g., open, closed).
- associatedProject: Project linked to the bug.
- dateCreated: Bug creation date and time.
- author: Bug report creator.
- tags: Tags categorizing the bug.

Methods:

- Complaint.fromResultRow constructor: Creates a Complaint object from database results.

Data Retrieval and Filtering:

- Initially, the loadComplaintsSource function retrieves comprehensive bug data from the database, including vital details such as bug ID, description, associated project, author, creation date, status, and tags.
- Following data retrieval, the filterComplaintsSource function allows users to refine bug exploration based on specific criteria, such as bug status (e.g., open, closed). This function tailors bug results to match user preferences.

User Interaction and Navigation:

- User engagement on the Bugs Page is facilitated by the interactive CustomDropDown component. This tool enables users to dynamically filter and sort bugs based on various parameters, including bug status, project association, and author. Upon selection, the page promptly reflects the chosen filter, enhancing user experience and navigation.

Table Creation and Bug Presentation:

- At the core of bug presentation lies the buildTableRow function. This function meticulously constructs each row within the bug table, compiling essential bug details such as bug ID, bug description, associated project name, author email, creation date, progress status, current bug status, and associated tags. It ensures a comprehensive overview of each bug's attributes.
- To complement this, the buildTableHeaders function establishes clear column headers such as "BUG ID," "BUG DESCRIPTION," "PROJECT NAME," "AUTHOR EMAIL," "CREATION DATE," "PROGRESS," "STATUS," and "TAGS," enhancing user comprehension and navigation while perusing bug data.

Visualization of Bug Progress:

- For improved understanding of bug resolution status, the getPercentageOfComplaintCompleted function calculates the percentage of completed tasks for individual bugs. This numerical representation is visually rendered using a custom linear progress indicator, providing users with a quick overview of each bug's progress status.

State Management and Database Updates:

- Behind the scenes, the ComplaintStateUpdates class handles bug state modifications. Whether altering a bug's status or updating associated tags, this class ensures database accuracy and consistency. It facilitates seamless bug tracking and management, contributing to an efficient bug tracking system.

Bug detail Page

The initState method in the BugDetailPage class is responsible for ensuring that when the bug detail page is initialized, any bug complaint in the "Pending" state is automatically transitioned to the "Acknowledged" state. This is achieved by scheduling a callback after the first frame is rendered, using WidgetsBinding.instance.addPostFrameCallback. Within this callback, the acknowledgeComplaint method is called.

The acknowledgeComplaint method updates the state of a bug complaint to "Acknowledged" if it's currently in the "Pending" state. It checks the complaint state and utilizes a utility method to update the state. If the update is successful, a notification is displayed to inform the user.

The main build method constructs the user interface of the bug detail page, displaying detailed information about the bug complaint such as its ID, creation date, author, project, description, notes, status, tags, assigned team, and staff notes. It provides options for updating bug details through a side sheet accessed via a button. Asynchronous data loading is implemented using future builders to display dynamic content like complaint notes, tags, and task previews.

The getCurrentComplaintState function retrieves the current state of a bug complaint from the database. It takes the complaint ID as input, queries the database for the corresponding state, and returns it. If the retrieval fails, it defaults to "Pending" and logs an error message.

The retrieveTags function retrieves tags associated with a bug complaint from the database. It takes the complaint ID as input, queries the database for associated tags, and returns them as a list. If no tags are found, it returns null.

The loadTasksSourceByComplaint function loads tasks associated with a bug complaint from the database. It takes the complaint ID as input, queries the database for associated tasks, processes them into Task objects, and stores them in a list for display. If no tasks are found, it initializes an empty list.

Bug detail update page

Constructor:

- The class constructor accepts parameters such as constraints, complaint, currentTags, and currentTasks to initialize the page's state.

State Initialization (initState):

- The initState method initializes state variables like controllers for task inputs, dropdown values for team members, and lists of selected tags.
- It categorizes current tasks into team lead and team member tasks, populates corresponding UI elements, and manages the visibility of the team section based on task presence.

Widget Build:

- The build method constructs the page's user interface using Flutter widgets in a column layout.
- It features sections for selecting tags, toggling the visibility of the team section, displaying team lead and team member tasks, and buttons for adding or removing team members and submitting updates.
- Widgets like FilterChip, CheckboxListTile, and custom TaskAssignmentForm facilitate user interactions.

Task and Tag Management:

- Users select tags from a list of options using FilterChip widgets.
- The team section can be toggled on/off using a CheckboxListTile, allowing users to focus on specific update aspects.
- Task assignments for team lead and members are managed via TaskAssignmentForm widgets, including dropdowns for staff selection and text fields for task input.
- Buttons allow adding new team members or removing existing ones, with corresponding state updates.

Updating Bug Details (onPress):

- The onPress method handles bug detail updates when the "Done" button is pressed.
- It validates task inputs and displays an error dialog if any task is missing.
- Task updates are categorized as new or updated based on changes compared to original tasks.
- Functions like updateComplaintTags and wipeAndUpdateTasks are called to update tags and task assignments, respectively.
- Bug state is updated to "In Progress" if there are new or updated tasks, with success messages displayed.

Database Interaction:

- The updateComplaintTags function communicates with the database to add or remove tags associated with a bug complaint.
- wipeAndUpdateTasks manages database operations for adding/updating tasks, considering preserved original task IDs.
- The addTasks function inserts new tasks, deletes existing ones, and updates task IDs in related sessions within the database.

Tasks Page

State Management

1. State Variables:
    - dropDownValue: This variable holds the current selection from a dropdown menu. It is initialized to the first value in a predefined list of task choices. The dropdown allows users to filter tasks based on their state (e.g., "All tasks", "New (2 days)").
    - searchBarString: This variable stores the current search query entered by the user. It is used to filter tasks further by matching the search string against task descriptions.
2. State Watching:
    - context.watch<TasksUpdate>(): This statement uses the Provider package to listen for updates to task data. When task data changes (e.g., a task is added, removed, or updated), the widget tree is rebuilt to reflect these changes.
    - context.watch<TaskStateUpdates>(): Similar to the above, this statement listens for updates to task states (e.g., changes in task progress or status). When state changes are detected, the widget tree is rebuilt to ensure the UI accurately represents the current state of tasks.

Task Loading and Filtering

1. FutureBuilder for Asynchronous Task Loading:
    - The FutureBuilder widget manages the asynchronous operation of loading tasks. It calls the loadTasksSourceByStaff function to load tasks specific to the current staff member. If the future is still executing, a loading indicator (e.g., a circular progress indicator) is shown. Once the future completes, the loaded tasks are processed through the filterTasksSource function to prepare them for display.
2. Loading Tasks from the Database:
    - loadComplaintsSource: This function loads all complaints from the database. Complaints are necessary for creating Task objects as each task is associated with a complaint.
    - getTasksByStaff: This function retrieves tasks associated with a specific staff member from the database. It takes the staff ID as a parameter and returns a list of tasks assigned to that staff member.
    - getStaffDataUsingID: This function retrieves detailed staff data for the specified staff ID. This data is used to associate tasks with the correct staff member.

- Task Processing: The results from the database are processed into Task objects. Each task is associated with a specific complaint and staff data. The processed tasks are then assigned to the global tasksSource list, making them available for filtering and display.

Task Filtering

1. Initial Filter by Dropdown Value:
    - The filtering process starts by checking the value of dropDownValue. If it is set to 'All tasks', the entire task list (tasksSource) is selected. For other values, tasks are filtered based on their state (e.g., 'New (2 days)'). This allows users to narrow down the list of tasks according to specific criteria. For example:
        - If dropDownValue is 'New (2 days)', only tasks with a state title of 'New' are selected.
        - If dropDownValue matches other task states, tasks are filtered accordingly.
2. Search Filter by Search Query:
    - The list of tasks filtered by the dropdown value is further filtered based on the search query entered by the user. The search is case-insensitive to prevent mismatches. The search query (searchBarString) is compared against task descriptions. Tasks whose descriptions contain the search query (converted to uppercase for comparison) are included in the final filtered list. If the search query is empty or consists only of whitespace, the dropdown filter results are returned without additional filtering.

Dropdown and Search Bar Interactions

1. CustomDropDown:
    - This custom widget allows users to select a task filter criterion from a dropdown menu. When a new value is selected, the dropDownValue is updated, and the state is refreshed to reflect this change in the task list. The dropdown provides options such as "All tasks", "New (2 days)", and other task states.
2. SearchBar:
    - This custom widget allows users to enter a search query to filter tasks by their descriptions. Each time the input changes, the searchBarString is updated, and the state is refreshed to apply the search filter to the task list. The search bar ensures that tasks matching the user's query are displayed, enhancing the user’s ability to find specific tasks quickly.

Detailed Function Descriptions

1. filterTasksSource:

- This function takes two parameters: filter (the dropdown value) and searchBarString (the search query). It returns a list of tasks that match the specified filter criteria.
- The function first filters tasks based on the dropdown value. If the value is 'All tasks', all tasks are included. Otherwise, it filters tasks by their state.
- Next, it filters the list further based on the search query. If the query is non-empty, tasks whose descriptions contain the query are included in the final list.

2. loadTasksSourceByStaff:
    - This asynchronous function takes the staff ID as a parameter and loads tasks associated with the specified staff member.
    - It first calls loadComplaintsSource to load all complaints, which are necessary for creating Task objects.
    - It then calls getTasksByStaff to retrieve tasks assigned to the staff member.
    - It also calls getStaffDataUsingID to retrieve detailed staff data.
    - The function processes the retrieved tasks into Task objects, associating each task with a specific complaint and staff member. The processed tasks are assigned to the global tasksSource list.

Function Interactions and Flow

1. Initialization:
    - When the TasksPage widget is initialized, it sets the initial values for dropDownValue and searchBarString.
    - The FutureBuilder widget is used to initiate the task loading process by calling loadTasksSourceByStaff.
2. Task Loading:
    - loadTasksSourceByStaff retrieves tasks and related data from the database. It processes this data into Task objects and updates the global tasksSource list.
    - Once the tasks are loaded, the FutureBuilder completes, and the filtered tasks are displayed.
3. Filtering and Display:
    - The filterTasksSource function is called to filter tasks based on the dropdown value and search query.
    - The filtered tasks are displayed in the UI. If no tasks match the filters, an empty placeholder is shown.

4. User Interactions:
    - When the user selects a different dropdown value, dropDownValue is updated, and the state is refreshed to apply the new filter.
    - When the user enters a search query, searchBarString is updated, and the state is refreshed to apply the search filter.
    - The task list is dynamically updated based on the current filters, ensuring that the displayed tasks match the user's criteria.

<u>Task detail page</u>

Initialization and State Management

Upon initialization, the TaskDetailPage retrieves the active work session for the task. This is executed in the initState method, utilizing a post-frame callback to ensure the context is fully built before accessing it. If an active work session is identified, a timer is established to periodically update the UI.

The setActiveWorkSession method is responsible for checking if there is an active work session for the task by querying the database. If an active session is found, the UI is updated every minute to reflect the ongoing session time. Concurrently, the setTaskStateAsInProgress method ensures that the task state is updated to "in progress" under specific conditions. This automatic update occurs if the task is not already in a final state (such as completed or transferred) and if the task is being viewed by the assigned staff member, rather than from the bug detail page.

Task State Transition Logic

The logic for setting the task state as "in progress" is critical for maintaining accurate task statuses. The getCurrentTaskState function retrieves the current state of the task from the database using the task ID. This state is compared against various predefined task states, such as inProgress, dueToday, completed, overdue, and transferred.

The setTaskStateAsInProgress method includes several conditional checks. Firstly, it verifies if the current task state is not already inProgress, dueToday, completed, overdue, or transferred. Secondly, it ensures that the task is not being viewed from the bug detail page (i.e., widget.viewingFromBug is false). Lastly, it confirms that the current user (globalActorID) is the one assigned to the task. If all conditions are satisfied, the method updates the task state to inProgress. This involves checking if the state is mounted (a property ensuring the widget is still in the widget tree), using the TaskStateUpdates provider to notify listeners about the state change, and displaying a confirmation message via a SnackBar.

Time Tracking Mechanism

Time tracking is a core feature of the TaskDetailPage, allowing users to start and end work sessions and log the time spent on tasks. When a user initiates a work session, the startWorkSession function is called, creating a new session entry in the database. The session's start time is recorded, and a Timer object updates the UI every minute to show the elapsed time. If the session starts successfully, the task state is automatically set to inProgress.

When a user ends a work session, the endWorkSession function updates the session's end time in the database. The Timer is stopped, and the session state is updated to reflect that the session has ended. A SnackBar displays the success or failure of the operation. The UI shows the elapsed time for the active session, updating every minute. The getTimeDifference function calculates the difference between the current time and the session start time.

Backend Interaction

Several backend functions support the functionality of the TaskDetailPage. The retrieveActiveWorkSession function queries the database to check for an active work session for a given task and returns a WorkSession object if an active session is found. The startWorkSession function creates a new work session in the database and records the session start time, while the endWorkSession function updates the end time of an active work session in the database and returns a boolean indicating the success of the operation. Additionally, the loadTasksSourceByComplaint function retrieves tasks associated with a specific complaint from the database and processes the retrieved tasks into task objects.

Work Session and Task Classes

The WorkSession class represents a work session with properties for ID, start date, and end date, used to track the start and end times of work sessions. The Task class represents a task with properties including ID, task description, task state, associated complaint, due date, assigned staff, and team lead status. It provides a factory constructor to initialize the task state based on conditions such as due date and completion status.

Sessions Log

Initialization and State Management

The SessionsLog widget is a stateful widget that receives a taskID as a parameter. This taskID is used to retrieve all work sessions related to the specific task. The state management in _SessionsLogState is primarily focused on handling the view mode (raw or stylized) and fetching the session data.

Retrieving Work Sessions

The retrieveAllWorkSessions function is responsible for fetching all work sessions associated with the given taskID. This function interacts with the database to retrieve the session data. It processes the raw database results into a list of WorkSession objects, each containing a start date, an optional end date, and an ID.

Building the User Interface

The build method in _SessionsLogState constructs the UI based on the fetched session data. The FutureBuilder is used to handle asynchronous data fetching:

1. Loading State: While waiting for the data to be fetched, a custom circular progress indicator is displayed.
2. Data Available State: Once the data is fetched, it is passed to the Padding widget, which further determines the view mode to be displayed.

Switching Views

A HeaderButton is provided to switch between the raw and stylized views. This button toggles the logView state variable and calls setState to rebuild the UI with the new view mode.

Raw View

In the raw view, the session data is displayed in a tabular format. The buildHeader function generates the table headers, while the LogRow widget is used to display each session's details in rows. The LogRow displays the index, start date, end date, and time taken for each session. The total time spent across all completed sessions is calculated and displayed by the TotalCompletedWorkSessionTime widget.

Stylized View

In the stylized view, the session data is displayed in a timeline format using the timelines package. The TimeLine widget constructs a visual representation of the work sessions:

1. Timeline: The outer timeline displays each session as a tile, showing the end date (if available) and the total time spent.
2. Inner Timeline: Each session's start date and total time spent are displayed within the inner timeline.

The InnerTimeLine widget is used within the TimeLine to show more granular details for each session.

Calculating Total Time

The TotalCompletedWorkSessionTime widget calculates the total time spent on completed sessions. It iterates through the list of WorkSession objects, summing the durations of sessions that have an end date. The total time is then formatted into a readable string, showing the time in months, days, hours, minutes, and seconds.

Enumerations

The LogView enum defines the possible views for the session log: raw and stylized. This enum is used to control the view mode of the SessionsLog:

- LogView.raw: Represents the raw tabular view of the session data.
- LogView.stylized: Represents the stylized timeline view of the session data.

LogRow Class

The LogRow class is a stateless widget that represents a single row in the raw view table. It displays the index, start date, end date (or "On going" if the session is not completed), and the time taken for each session. It uses a specific text style to ensure consistency and readability.

Function Interactions

1. Session Retrieval: retrieveAllWorkSessions fetches session data from the database and processes it into WorkSession objects.
2. View Toggle: The HeaderButton toggles the logView state variable between raw and stylized, calling setState to refresh the UI.
3. Raw View Display: In the raw view, the buildHeader function and LogRow widget are used to display session data in a table format, with TotalCompletedWorkSessionTime calculating and displaying the total time spent.
4. Stylized View Display: In the stylized view, the TimeLine widget constructs a visual timeline, using InnerTimeLine for detailed session display.
5. Total Time Calculation: The TotalCompletedWorkSessionTime widget calculates the total duration of all completed work sessions and formats it for display.

<u>Task detail update page</u>

As described in the UI section, this page facilitates updating the details of a specific task.

initState

The initState function is a crucial lifecycle method in Flutter, called once when the state object is first created. This function initializes state variables and performs any necessary setup for the state object.

- Initialization of dropDownValue: This variable is set to the current staff member, identified by globalActorID. This ensures the dropdown menu displays the current user as the default selection when the page loads.
- Processing tasksSource: The function processes a list called tasksSource, which contains all tasks associated with a specific complaint. It extracts unique team members assigned to these tasks, populating the dropdown for task transfer.

Clicking the Done Button

When the "Done" button is clicked, several actions are performed, depending on the state of the various controls and inputs on the page. These actions include updating the task's state, transferring the task to another staff member, marking the complaint as completed, and adding notes to the complaint. Here's a detailed explanation of each action:

Marking Task as Completed

- Condition: This action is triggered if the taskCompleted boolean flag is set to true.
- Function Call: The function context.read<TaskStateUpdates>().updateTaskState is called.
- Parameters: It requires taskID and newState.
    - taskID: This is the unique identifier of the task being updated.
    - newState: This is set to TaskState.completed, indicating that the task has been completed.
- Outcome: The task's state is updated in the database or the state management system, marking it as completed.

Transferring the Task

- Condition: This action occurs if the dropDownValue (selected staff member) is different from the current assignedStaff of the task.
- Creating a New Task:
    - Function Call: The function context.read<TasksUpdate>().addNewTransferredTask is called.
    - Parameters: A new Task object is created with the following properties:
        - id: Set to 0 initially as a placeholder.
        - task: The task description from the current task.
        - taskState: Set to TaskState.received, indicating the new task is in the received state.
        - associatedComplaint: The complaint associated with the current task.
        - dueDate: The due date from the current task.
        - assignedStaff: The staff member selected in the dropdown.
        - isTeamLead: Set to false.
    - Outcome: A new task is created and assigned to the selected staff member.
- Updating Current Task State:
    - Function Call: The function context.read<TaskStateUpdates>().updateTaskState is called again.
    - Parameters: It requires taskID and newState.
        - taskID: This is the unique identifier of the current task.

    - newState: This is set to TaskState.transferred, indicating that the task has been transferred.
  - Outcome: The current task's state is updated to transferred.

Marking Complaint as Completed

- Condition: This action is performed if the user is a team lead (widget.task.isTeamLead is true) and the complaintCompleted boolean flag is set to true.
- Function Call: The function context.read<ComplaintStateUpdates>().updateComplaintState is called.
- Parameters: It requires complaintID and newState.
  - complaintID: This is the unique identifier of the complaint associated with the task.
  - newState: This is set to ComplaintState.completed, indicating that the complaint has been resolved.
- Outcome: The complaint's state is updated in the database or the state management system, marking it as completed.

Adding Staff Notes

- Condition: This action is performed if the user is a team lead (widget.task.isTeamLead is true) and the userNoteController.text is not empty.
- Function Call: The function context.read<StaffNotesUpdates>().addStaffNoteToComplaint is called.
- Parameters: It requires complaintID and note.
  - complaintID: This is the unique identifier of the complaint associated with the task.
  - note: The text entered by the user in the userNoteController field.
- Outcome: The note is added to the complaint, providing additional information or context related to the task or complaint.

Staff Page

The key components handle updating staff details, deleting staff records, loading staff data, and filtering staff based on search string.

Staff Page Functionality

The StaffPage class represents the staff management page. This class contains the _StaffPageState class, which manages the state of the StaffPage. The state class maintains a searchBarString to store the user's search input. It builds the page layout, incorporating a search bar and a list of staff members.

When constructing the layout, the build method employs a future builder to wait for staff data to load. It updates the user interface accordingly. This future builder relies on the loadStaffSource function, which fetches and processes staff data from the database. The build method also includes logic to determine the screen's width to adjust the search bar's layout dynamically.

The filterStaffSource function filters the list of staff based on the user's search input. It accepts searchBarString as a parameter. If the search string is empty, the function returns the complete list of staff. Otherwise, it filters the staff list by matching the search string against the staff ID, full name, or email. The function uses a helper method, getFullName, to construct the full name of a staff member from their name components.

Staff Updates Management

The StaffUpdates class manages and notifies listeners about updates to staff data. It contains several methods:

1. updateStaffEmail: This method updates a staff member's email in the database. It accepts the staff ID and the new email as parameters. The method executes a database query to update the email and checks the result to determine if the update was successful. If successful, it notifies listeners to refresh the displayed data.
2. updateStaffName: Similar to updateStaffEmail, this method updates a staff member's name components in the database. It accepts the staff ID, surname, first name, and middle name as parameters. It performs a database query to update the name components and checks the result for success. Upon success, it notifies listeners.
3. deleteStaff: This method deletes a staff member from the database. It accepts the staff ID as a parameter and executes a series of database queries to remove records associated with the staff member. The queries are executed in a specific order to prevent foreign key constraint violations. If all queries succeed, the method notifies listeners of the deletion.

Database Operations

The database operations are encapsulated within specific functions that interact with the database to perform updates and deletions. These functions are used within the StaffUpdates methods.

- updateStaffEmail: Executes an SQL query to update a staff member's email in the database. It checks if the update was successful by examining the result and returns a boolean indicating success.
- updateStaffName: Executes an SQL query to update a staff member's name components. It checks the result for success and returns a boolean indicating the outcome.
- wipeStaffDataFromDatabase: Deletes a staff member and all associated records from the database. The function executes multiple SQL queries to delete records in a specific order, ensuring no foreign key constraint violations. It returns a boolean indicating whether all deletions were successful.

Data Loading and Filtering

The loadStaffSource function loads the list of staff members from the database and processes them into Staff objects. It retrieves all staff records from the database. If records are found, it processes each record into a Staff object and stores it in the staffSource list. If no records are found, it initializes staffSource as an empty list. This function is called within the FutureBuilder in _StaffPageState to load and display the staff data when the page is accessed.

Interaction and Logical Flow

The user interacts with the staff management page, triggering various operations:

1. User Access and Search: When the user accesses the staff management page, the StaffPage and _StaffPageState classes are initialized. The user can enter a search string to filter the staff list, updating searchBarString and triggering a redraw of the page to display filtered results.
2. Data Loading: The FutureBuilder within the build method of _StaffPageState calls loadStaffSource to fetch and process the staff data from the database. The processed data is stored in staffSource and used to display staff details.
3. Data Filtering: The filterStaffSource function filters the staff list based on the search string and returns the filtered list for display.
4. Data Updates: When the user updates a staff member's email or name through the interface, the StaffUpdates methods (updateStaffEmail and updateStaffName) are called. These methods perform database operations to update the staff details and notify listeners to refresh the displayed data. If a staff member is deleted, deleteStaff is called to remove the staff member and associated records from the database, and listeners are notified of the change.

Staff details page

Stateful Widget and State Management

The **StaffDetailPage** class is a stateful widget that displays detailed information about a staff member and allows for updates to their email and name. The **_StaffDetailPageState** class manages the state of this widget, including tracking whether the user intends to update the email or name.

State Variables

- **updatedEmail**, **updatedSurname**, **updatedFirstName**, **updatedMiddleName**: These hold the updated values for the staff's email and name.
- **updateEmailIntent**, **updateNameIntent**: Boolean flags to indicate whether the user intends to update the email or name.
- **formKeys**: A list of form keys used to validate the email and name update forms.

Build Method

The **build** method constructs the UI elements and watches for changes in task and staff updates using context. The method listens to three primary change notifiers: **TasksUpdate**, **TaskStateUpdates**, and **StaffUpdates**. These notifiers trigger UI updates when relevant data changes.

Watching for Updates

- **context.watch<TasksUpdate>()**: Monitors updates to tasks.
- **context.watch<TaskStateUpdates>()**: Monitors changes in task states.
- **context.watch<StaffUpdates>()**: Monitors updates to staff details.

Updating Staff Email and Name

When the user opts to update the email or name, the state variables **updateEmailIntent** and **updateNameIntent** are set to true, triggering the display of input forms. The forms use validation logic to ensure the inputs are correct before submitting.

Email Update Process

1. The user clicks the "Update" button next to the email.
2. The **updateEmailIntent** flag is set to true, triggering the display of the email update form.
3. The user enters the new email and submits the form.
4. The form's validator checks the input.
5. If valid, the **updateStaffEmail** method of **StaffUpdates** is called to update the email in the database.
6. Upon successful update, the state is refreshed, and a success message is displayed.

Name Update Process

1. The user clicks the "Update" button next to the name.
2. The **updateNameIntent** flag is set to true, triggering the display of the name update form.
3. The user enters the new name details and submits the form.
4. The form's validators check the inputs.
5. If valid, the **updateStaffName** method of **StaffUpdates** is called to update the name in the database.
6. Upon successful update, the state is refreshed, and a success message is displayed.

Deletion of Staff

The staff can be deleted using the "Delete" button, which triggers a confirmation alert. If confirmed, the **deleteStaff** method of **StaffUpdates** is called to remove the staff from the database. Upon successful deletion, the user is navigated back from the detail page, and a success message is displayed.

Change Notifiers

Change notifiers in this context serve as state management tools that notify listeners when a change occurs. They extend the **ChangeNotifier** class and provide methods for updating data, which in turn call **notifyListeners** to trigger a UI update.

TaskStateUpdates

- The **updateTaskState** method updates the state of a task in the database.
- If the update is successful, **notifyListeners** is called to update the UI.

TasksUpdate

- The **wipeAndUpdateTasks** method updates a list of tasks in the database.
- The **addNewTransferredTask** method adds a new task to the database.
- Both methods call **notifyListeners** upon successful updates.

StaffUpdates

- The **updateStaffEmail** method updates a staff's email in the database.
- The **updateStaffName** method updates a staff's name in the database.
- The **deleteStaff** method deletes a staff from the database.
- All methods call **notifyListeners** upon successful updates.

Asynchronous Operations and FutureBuilder

Asynchronous operations are handled using **Future** and **FutureBuilder**. The **FutureBuilder** widget builds its content based on the state of the asynchronous operation.

loadTasksSourceByStaff

- This function loads tasks associated with a specific staff member.
- It retrieves tasks from the database and processes them into **Task** objects.
- The **FutureBuilder** in the **build** method uses this function to display the current tasks of the staff.

Database Interactions

Database operations are performed using asynchronous functions that interact with the database to update staff details or retrieve tasks.

Email Update

The **updateStaffEmail** function updates the staff's email in the database and returns a boolean indicating success.

Name Update

The **updateStaffName** function updates the staff's name in the database and returns a boolean indicating success.

Staff Deletion

The **wipeStaffDataFromDatabase** function deletes all records associated with a staff member, ensuring the deletion is performed in an order that prevents foreign key constraint violations. It returns a boolean indicating success.

USER FILES

Complaint Page

App Bar and Menu

The app bar serves as the navigation header for the complaints page, featuring a title and additional options accessible via a menu. The menu includes:

- **Update Password Option**: Upon clicking this option, a side sheet appears, allowing the user to update their password securely. This functionality is achieved through the **SideSheet.right** method.

Filtering and Searching Complaints

The **filterComplaintsSource** function is pivotal for refining the displayed list of complaints. It takes two parameters:

1. **filter**: This parameter represents the selected filter option in the dropdown menu. Based on its value, the list of complaints is filtered.
2. **searchBarString**: It captures the user's search input. If provided, it further refines the filtered list of complaints.

The function systematically filters the complaints using a series of conditional statements:

- **Dropdown Filter**: If the selected filter is "All complaints," the entire **complaintsSource** list is returned. Otherwise, only complaints with a matching state title are included in the filtered list.
- **Search Bar Filter**: If the search bar is empty, the filtered list is returned as is. Otherwise, complaints whose descriptions contain the search query are included in the filtered list.

Loading Complaints

The **loadComplaintsSourceByUser** function is tasked with asynchronously retrieving complaints associated with the logged-in user. Its functionality involves several steps:

1. **Data Retrieval**: The function interacts with a data source (**db**) to fetch complaint data. Specifically, it queries the database to obtain complaints linked to the user identified by **userID**.
2. **Processing Complaints**: Once the complaints are retrieved, they undergo processing to convert them into **Complaint** objects. This process involves additional database queries to fetch associated tags, projects, and author information for each complaint.
3. **Storing Complaints**: The processed complaints are stored in the **complaintsSource** list for subsequent display on the UI.
4. **Handling Empty Results**: If no complaints are found for the user, the **complaintsSource** list is cleared.

Complaint State Updates

The **ComplaintStateUpdates** class extends **ChangeNotifier** and facilitates real-time updates of complaint states and tags. It exposes two asynchronous methods:

1. **updateComplaintState**: This method is responsible for updating the state of a specific complaint identified by **complaintID**. It triggers an interaction with the data source (**db**) to perform the state update. Upon success, it notifies listeners to refresh the UI.
2. **updateComplaintTags**: It updates the tags associated with a complaint identified by **complaintID**. Similar to **updateComplaintState**, it interacts with the data source to add new tags to the complaint. Upon successful completion, it notifies listeners to refresh the UI.

Both methods handle potential failure scenarios gracefully, printing debug messages in case of errors.

Interaction with Data Source

The loadComplaintsSourceByUser function queries the backend (db) to fetch complaints associated with the current user. It retrieves complaint records using the user's ID, processes each complaint by fetching additional details like tags and projects, and populates the complaintsSource list. If no complaints are found, the list is cleared.

The ComplaintStateUpdates class contains methods (updateComplaintState and updateComplaintTags) for updating complaint states and tags in the database (db). These methods asynchronously communicate with the backend to execute state and tag updates. Upon success, they notify listeners to refresh the UI. If errors occur, debug messages are printed for troubleshooting.

STAFF SECTION

This section consists of parts explained in the Admin section, tasks page, calendar page and discuss page.

MISCELLANEOUS

Tools file

This file contains mainly utility code/tools used in various sections of the codebase

- normalize0to1: This function takes a value and ensures it falls within the range of 0 to 100. It then normalizes this value to a range between 0 and 1 and returns it.
- getPercentage: Given a number and a total, this function calculates the percentage of the number in relation to the total and returns it as a double.

- determineContainerDimensionFromConstraint: This function calculates the dimension of a container based on a constraint value and a value to subtract from it. If the result is greater than 0, it returns the difference; otherwise, it returns 0.
- convertToDateString: Given a DateTime object, this function converts it into a formatted string representing the date and time in the 'yyyy-MM-dd HH' format.
- convertToDateStringSessionsLog: Similar to convertToDateString, but it formats the date and time in the 'yyyy-MM-dd HH:mm' format.
- getTimeDifference: Calculates the time difference between two DateTime objects, newer and older. It returns a formatted string representing the difference in months, days, hours, minutes, and optionally, seconds if viewing from a sessions log.
- hashPassword: Takes a password string, converts it to bytes using UTF-8 encoding, computes its SHA-512 hash, and returns the hashed result as a string.
- authenticatePasswordHash: Given a plain text password and a hashed password, this function checks if the hashed version of the plain text password matches the provided hashed password. It returns true if they match, false otherwise.
- getFullNameFromNames: Constructs a full name string from provided surname, first name, and middle name. It concatenates these values with spaces in between, ensuring that if first name or middle name is null, they are replaced with empty strings.
- getInitialsFromName: Given a full name, this function extracts the initials from each name part (separated by spaces) and returns them as a string of uppercase letters.

6. Conclusion

6.1 Benefits

a. Benefits to users:

- Reducing bottleneck situations for development teams by letting them quickly and efficiently receive, assign, monitor and resolve tasks/bugs.
- Customers with complaints will be able to effectively communicate these complaints to development teams.
- Creating a more transparent bug resolution process for users

b. Benefits to me:

- Acquiring a good understanding of intuitive user interface design
- Expanding and reinforcing my knowledge on Database management systems and querying a database
- Gain a good grasp of the flutter software development kit and the dart programming language.

6.2 Ethics

In future works I plan on incorporating these

- **Data privacy:** Users of the software should be able to use the system with the assurance that their personal data is not being used for malicious purposes such as being sold to third parties without their express permission. In order to do this, the project has to follow established privacy policies such as the GDPR (General Data Protection Regulation) in EU member states and KVKK (Kişisel Verileri Koruma Kanunu) in Turkey.

- **Monitoring for extremists/ sanctioned companies:** In order to make the platform as safe as possible, measures should be in place to prevent extremist groups from joining the platform and remove them if they find their way in. Companies/groups/ individuals sanctioned by the international community should also not be allowed on the platform such as ones that support the 2022 Russian invasion of Ukraine.

- **Prevention of Brand Impersonation:** Brand impersonation means entities falsely identifying themselves as a well-known brand usually for malicious purposes. There should be measures such as brand authentication put in place to such activity to a minimum.

Why did I choose this project?

I chose this project because it struck a good balance between being demanding enough to require a lot of effort and research and being a doable project; while providing a genuinely useful tool that has the potential to facilitate the product development process for teams.

6.3 Future Works

I intend to carry on working on this project after graduation due to its potential, Improvements can be made to the overall security of the platform by implementing modern cryptographic techniques such as 2-factor authentication, I also intend to integrate a specialized artificial intelligence model that learns patterns of users and serves as an assistant, Also an AI chatbot feature that will serve as the first phase of communication with the consumer incase their reports are trivial and can be solved easily.

I also intend to add spam protection as well as update the UI to be more “classic”.

7. References

[1]. Epsilon. (2018, Jan. 9). *New Epsilon research indicates 80% of consumers are more likely to make a purchase when brands offer personalized experiences* [online]. Available: https://www.epsilon.com/us/about-us/pressroom/new-epsilon-research-indicates-80-of-consumers-are-more-likely-to-make-a-purchase-when-brands-offer-personalized-experiences.

[2]. N. Arora, D. Ensslen, L. Fiedler, W. Liu, K. Robinson, E. Stein, & G. Schüler. (2021, Nov. 12). *The value of getting personalization right or wrong is multiplying* [online]. Available: https://www.mckinsey.com/capabilities/growth-marketing-and-sales/our-insights/the-value-of-getting-personalization-right-or-wrong-is-multiplying.