Technical Documentation

Group Number: Group 5

Project Title: Poutine Time

Member Contribution:

Since Prosper Manyele and Raihan Ridom were the only members of the group each contributed in the files as they were working in a "I do this then you do this and what else is left" method.

Example: One would make the Usermodel then down the line of development the other needs to add more stuff to the Usermodel which would probably be more code then it originally had and vice versa.

Usermodel

1. **Class Definition (UserModel):**
   - Represents a user in the application.
   - Contains a single property **username**, representing the user's username.
2. **Constructor (UserModel({required this.username})):**
   - Initializes a **UserModel** instance with the provided **username**.
3. **toMap Method:**
   - Converts the **UserModel** instance into a **Map<String, dynamic>**.
   - This is useful for storing user data in a format that can be easily saved to Firestore.
4. **fromMap Static Method:**
   - Creates a **UserModel** instance from a **DocumentSnapshot** and a **userID**.
   - Retrieves user data from the **DocumentSnapshot** and **userID**.
   - Returns a new **UserModel** instance with the extracted username.

This code defines a basic user model with methods for converting to and from a format suitable for storage in Firestore. It assumes that user data is stored as a map within a Firestore document, with each user identified by a unique userID.

<u>Post Model</u>

1. **Class Definition (PostModel):**
    - Represents a post in the application.
    - Contains properties such as **id**, **userID**, **username**, **description**, **likes**, **dislikes**, **release_date**, **imageUrls**, **comments**, **threadFather**, and **channel**.
2. **Constructor (PostModel(...)):**
    - Initializes a **PostModel** instance with various properties, providing default values where applicable.
3. **toMap Method:**
    - Converts the **PostModel** instance into a **Map<String, dynamic>** for storage in Firestore.
4. **fromMap Static Method:**
    - Creates a **PostModel** instance from a **Map<String, dynamic>** obtained from Firestore, with specific handling for **Timestamp** conversion.
5. **printDetails Method:**
    - Prints details of the **PostModel** instance, useful for debugging.
6. **addComment Method:**
    - Adds a comment ID to the **comments** list, checking for duplicates.

This code defines a post model with methods for converting to and from a format suitable for storage in Firestore, as well as additional methods for managing comments and printing details.

<u>Channel Model</u>
The **ChannelModel** class represents a model for a channel, presumably used in your application to categorize or tag posts. Here's a brief overview of the class:

- **Attributes:**
    - **id**: A unique identifier for the channel.
    - **name**: The name or title of the channel.
    - **icon**: An **IconData** representing the icon associated with the channel.
- **Constructor:**
    - Initializes the **ChannelModel** with the required parameters: **id**, **name**, and **icon**.
- **Example Usage:**

- You might create instances of **ChannelModel** to represent different channels in your application, each with a unique **id**, a descriptive **name**, and an associated **icon**.

Post Controller

1. **Class Definition (PostControllerService):**
    - Manages interactions related to post entities.
2. **Constructor (PostControllerService()):**
    - Initializes the **postCollection** to reference the Firestore collection 'postCollection'.
3. **Methods for Picking Images (pickImageFromGallery(), pickImageFromCamera()):**
    - Use the **ImagePicker** to select images from the gallery or camera.
4. **uploadImages Method:**
    - Uploads a list of images to Firebase Storage and returns their download URLs.
5. **addPost Method:**
    - Adds a new post to Firestore, including the option to upload images.
6. **sortPostsTrending Method:**
    - Sorts the postList by the number of likes, making posts appear in trending order.
7. **getPosts Method:**
    - Retrieves posts from Firestore based on the selected channel or all channels.
8. **likePost and dislikePost Methods:**
    - Handles user likes and dislikes for a post, updating the Firestore document.
9. **addComment Method:**
    - Adds a comment to a post in Firestore and updates the post's comments list.
10. **fetchComment and fetchComments Methods:**

- Fetches single or multiple comment posts from Firestore based on their IDs.

11. **setPostList and getPostList Methods:**

- Sets and retrieves the **postList** variable.

This code provides a service for managing posts, including uploading images, adding posts, sorting, and interacting with Firestore for various post-related operations.

UserController

1. **Class Definition (UserController):**
    - Manages interactions related to the **User** and **UserModel** entities.
2. **Constructor (UserController()):**
    - Initializes the **UserController** instance.
3. **setUser Method:**
    - Sets the current user using the Firebase Authentication instance.
4. **getUserID Method:**
    - Retrieves the ID of the current user.
5. **getUsername Method:**
    - Retrieves the username from the associated **UserModel**.
6. **setUserModel Method:**
    - Sets the **userModel** property of the **UserController**.
7. **getUserModelData Method:**
    - Retrieves the user model data from Firestore based on the provided **userID**.
    - If an error occurs during the fetch, it creates a default user model with the user's email as the username and updates the Firestore user list.

This code provides a controller for managing user-related operations, including fetching user data from Firestore, setting user information, and interacting with the **UserModel**.

StateManager

1. **Class Definition (StateManager):**
    - Serves as a utility class for managing the initialization of controllers.
2. **Static Properties (postController and userController):**
    - Holds instances of the **PostControllerService** and **UserController** classes, respectively. These are declared as static so that they can be accessed without creating an instance of **StateManager**.
3. **Instance Property (auth):**
    - Holds an instance of **FirebaseAuth**.
4. **initialize Method:**
    - Initializes the **postController** and **userController** instances.

- This method is declared as static, so it can be called on the class itself without creating an instance of **StateManager**.

By centralizing the initialization of controllers in the **StateManager** class, this makes a convenient way for other parts of the application to access these controllers without the need to pass them as parameters repeatedly. It can be particularly useful if you want to ensure that these controllers are initialized consistently at the start of your application.

<u>Theme Provider</u>

1. **Class Definition (ThemeProvider):**
    - Manages the theme preferences for the application.
2. **Instance Variable (_isDarkTheme):**
    - Holds the current theme preference (dark or light).
3. **Constructor (ThemeProvider()):**
    - Initializes the theme preference by calling **_loadThemePreference()**.
4. **Getter (isDarkTheme):**
    - Retrieves the current theme preference.
5. **Setter (isDarkTheme):**
    - Sets the theme preference, saves it to **SharedPreferences**, and notifies listeners (widgets) about the change.
6. **_loadThemePreference Method:**
    - Loads the theme preference from **SharedPreferences** and notifies listeners about the updated preference.
7. **_saveThemePreference Method:**
    - Saves the current theme preference to **SharedPreferences**.

It uses **SharedPreferences** to persist the theme preference across application restarts.

Auth Gate

1. **Class Definition (AuthPage):**
    - Represents a screen for authentication.
2. **Constructor (AuthPage):**
    - Defines a constructor for the **AuthPage** widget.
3. **State (_AuthPage):**
    - Defines the state for the **AuthPage** widget.
4. **Instance Variable (auth):**
    - Holds an instance of **FirebaseAuth** for handling authentication.
5. **prevAuth Method:**
    - Defines a method to display the previous authentication UI using **Scaffold**, **AppBar**, and **ElevatedButton**.
    - Currently commented out, but it seems to provide a simple login and signup interface.
6. **newAuth Method:**
    - Defines a method to display the new authentication UI using **StreamBuilder**.
    - Uses **authStateChanges** to listen for changes in the authentication state.
    - If there is no authenticated user, it displays the **SignInScreen** with email and Google authentication providers.
    - If there is an authenticated user, it displays a **LoadingScreen**.
7. **build Method:**
    - Returns the UI based on the **newAuth** method, showing the authentication UI using **StreamBuilder**.

Sign In Page

1. **Class Definition (SignUpPage):**
    - Represents a screen for user registration and sign-up.
2. **State (_SignUpPageState):**
    - Defines the state for the **SignUpPage** widget.
3. **Instance Variables:**
    - **_auth**: Holds an instance of **FirebaseAuth** for handling authentication.

- **_firestore**: Holds an instance of **FirebaseFirestore** for interacting with Firestore.
- **_googleSignIn**: Holds an instance of **GoogleSignIn** for Google sign-in.
- **_emailController**, **_usernameController**, **_passwordController**: Controllers for handling user input.

4. **build Method:**
    - Builds the UI for the sign-up page, including text fields for email, username, and password.
    - Provides buttons for email/password sign-up and Google sign-in.
5. **_signUp Method:**
    - Handles the logic for email/password sign-up.
    - Retrieves user input for email, password, and username.
    - Uses **_auth.createUserWithEmailAndPassword** to create a new user.
    - Creates a **UserModel**, updates the display name, and stores user information in Firestore.
    - Navigates to the **LoadingScreen** after successful sign-up.
6. **_signInWithGoogle Method:**
    - Handles the logic for signing in with Google.
    - Retrieves the username from user input.
    - Uses **_googleSignIn.signIn** and **_auth.signInWithCredential** for Google sign-in.
    - Creates a **UserModel**, updates the display name, and stores user information in Firestore.
    - Navigates to the **LoadingScreen** after successful sign-in.

Overall, the **SignUpPage** widget provides a user interface for signing up with both email/password and Google, and it integrates with Firebase for authentication and Firestore for storing user information.

Loading Page

1. **Class Definition (LoadingScreen):**
    - Represents a loading screen that initializes and loads essential data before navigating to the main screen.
2. **State (_LoadingScreenState):**
    - Defines the state for the **LoadingScreen** widget.
3. **Instance Variable (initDataFuture):**
    - Holds a **Future** that represents the asynchronous initialization of data.
4. **initState Method:**
    - Initiates the **initDataFuture** during the widget's initialization.
5. **initializeData Method:**
    - Simulates a delay or fetches the user model and post list from controllers.
    - Sets the user in the **StateManager**.
    - Fetches post list and user model concurrently.
    - Returns a map containing the post list and user model.
6. **build Method:**
    - Builds the UI for the loading screen.
    - Displays a loading indicator while waiting for data.
    - Handles errors during data initialization.
    - Navigates to the **HomePageScreen** once data has been loaded.

Overall, the **LoadingScreen** acts as an intermediate screen that ensures essential data is loaded before transitioning to the main application screen.

Post Widget

The **PostWidget** is a complex widget that displays information about a post. Let's break down its structure and functionality:

PostWidget Class:

- **Properties:**
    - **postModel**: The post model containing information about the post.
    - **displayUsername**: Flag to control the visibility of the username.

    - **displayPostOption**: Flag to control the visibility of the post options.
    - **displayInteractions**: Flag to control the visibility of interaction buttons.
    - **displayReleaseDate**: Flag to control the visibility of the release date.
    - **isTappable**: Flag to determine whether the widget is tappable.
- **State Class (_PostWidgetState):**
    - Manages the state of the **PostWidget**.
- **Initialization (initState):**
    - Initializes the **_postModel** property with the provided **postModel**.
- **Image Section (buildImageSection):**
    - Builds a section to display images if the post has any.
    - Uses a **PageView** to allow swiping through multiple images.
    - Displays navigation arrows for image scrolling.
- **Interaction Button Press Handlers (onThumbsUpPressed, onThumbsDownPressed):**
    - Handles the press events for the thumbs-up and thumbs-down buttons.
    - Calls the corresponding methods from **PostControllerService** to update likes and dislikes.
    - Updates the state to reflect the changes in likes and dislikes.
- **Interaction Buttons Widget (interactionButtonsWidget):**
    - Builds a row of interaction buttons (thumbs-up, thumbs-down, and comment).
    - Uses the **InteractionButton** widget for each button.
- **Top Container Widget (topContainerWidget):**
    - Builds the top container that contains the username and post options.
- **Middle Container Widget (middleContainerWidget):**
    - Builds the middle container that contains the post description.
- **Bottom Container Widget (bottomContainerWidget):**
    - Builds the bottom container that contains interaction buttons and the release date.
- **Build Method (build):**
    - Constructs the entire widget using the described components.

    - Supports tapping (navigates to the detailed post view) based on the **isTappable** flag.

InteractionButton Class:

- **Properties:**
    - **iconData**: The icon data for the button.
    - **onPressed**: Callback function for button press.
    - **count**: The count associated with the button.
    - **iconColor**: Color of the icon.
    - **textColor**: Color of the count text.
- **Build Method (build):**
    - Constructs a row containing an icon button and, if applicable, a count.

Note:

- The code uses **GoogleFonts.lato** for styling text.
- The structure of the **PostWidget** allows for flexibility in displaying different elements based on flags.
- The **InteractionButton** widget is reusable and allows customization of icons, colors, and counts.

<u>Channel Side Bar</u>

The **ChannelSidebar** widget is a sidebar (drawer) that displays a list of channels and a "Trending" option. Here's an overview of its structure and functionality:

ChannelSidebar Class:

- **Properties:**
    - **channels**: List of **ChannelModel** objects representing different channels.
    - **onChannelSelected**: Callback function that is called when a channel is selected.
    - **onTrendingSelected**: Callback function that is called when the "Trending" option is selected.
- **Build Method (build):**
    - Constructs the main **Drawer** widget with a container as its child.
    - Defines the primary, text, and accent colors.

        - Creates a **ListView** containing the following elements:
            - "Channels" heading (styled with Google Fonts).
            - List of channel items using a **for** loop:
                - Each channel item is a **ListTile** with the channel name, icon, and onTap callback.
            - A "Trending" item with its own **ListTile** and onTap callback.
        - Each **ListTile** includes a **TextStyle** using Google Fonts for styling.

Note:

- The **Drawer** provides a convenient way to implement a sidebar in Flutter.
- The channel items are dynamically generated based on the provided **channels** list.
- The "Channels" heading and "Trending" option have distinct styling.
- Icons are included for each channel and the "Trending" option.
- The **onChannelSelected** and **onTrendingSelected** callbacks are triggered when the corresponding items are tapped.
- The sidebar is dismissed (**Navigator.pop(context)**) after selecting a channel or the "Trending" option.

<u>Homepage</u>

The **HomePageScreen** widget is a **StatefulWidget** representing the main screen of the application. It includes a bottom navigation bar with three tabs: "Feed," "Create Post," and "Account." The content of each tab is provided by corresponding pages (**FeedPageScreen**, **CreatePostPageScreen**, and **AccountsPage**). Here's an overview of the code:

HomePageScreen Class:

- **Properties:**
    - **_selectedIndex**: Keeps track of the currently selected tab index for the bottom navigation bar.
- **Methods:**
    - **initializeData**: An asynchronous method used in **initState** to perform any necessary data initialization. Currently, it doesn't have any specific implementation.
- **Build Method (build):**
    - Constructs a **Scaffold** with a body and a bottom navigation bar.

- The body is determined by the selected tab using **_pages[_selectedIndex]**.
- The bottom navigation bar (**BottomNavigationBar**) has three tabs:
    1. Feed
    2. Create Post
    3. Account
- The **onTap** callback updates **_selectedIndex** when a tab is tapped.

Note:

- The **_pages** list contains instances of the pages corresponding to each tab.
- The **BottomNavigationBar** provides navigation between the tabs.
- Each tab has an associated icon and label.
- The text color of the selected tab is determined by the theme brightness.
- The body of the **Scaffold** is dynamically changed based on the selected tab.

<u>Feed Page</u>

The **FeedPageScreen** is a **StatefulWidget** representing the feed page of your application. It fetches and displays posts in a ListView, and users can refresh the feed by pulling down on the screen. It also includes an **AppBar** with the application title and the currently logged-in user's username. Here's an overview of the code:

FeedPageScreen Class:

- **Properties:**
    - **_isLoading**: A boolean variable to track the loading state when fetching posts.
    - **selectedChannel**: A string variable representing the currently selected channel.
- **Methods:**
    - **_onPostsChange**: A method that is called when there's a change in the posts. It triggers a rebuild of the widget.
    - **_fetchPosts**: A method to fetch posts from the server. It sets **_isLoading** to true before fetching and false after completing the fetch.
    - **_feedAppBar**: A method that returns the **AppBar** for the feed page. It includes the application title and the currently logged-in user's username.
    - **_bodyWidget**: A method that returns the body of the widget. It displays a loading indicator while fetching, a message when there are no posts, and a **ListView** of **PostWidget** for displaying posts.

- **Lifecycle Methods:**
    - **initState**: Initializes the widget. It fetches posts, adds a listener for post changes, and sets up the selected channel.
- **Build Method (build):**
    - Constructs a **Scaffold** with an **AppBar**, **ChannelSidebar**, and the body of the widget.
    - The **ChannelSidebar** is a custom widget for displaying channel selection options and a trending option.

Note:

- The **PostWidget** is used to display individual posts.
- The **RefreshIndicator** allows users to refresh the feed by pulling down on the screen.
- The **ChannelSidebar** provides options for selecting a specific channel or viewing trending posts.

<u>**Create Post**</u>

The **_CreatePostPageScreenState** class represents the state for the **CreatePostPageScreen** widget. This page allows users to create a new post by entering a description, selecting a channel, and optionally adding images. The post can be submitted, and the user will be redirected to the home page.

Here's an overview of the key elements in the code:

- **Text Controllers:**
    - **_descriptionController**: A **TextEditingController** for the post description input field.
- **Properties:**
    - **_isLoading**: A boolean variable to track the loading state when creating a post.
    - **_selectedChannel**: The selected channel for the post, initialized with the first channel from the list.
    - **_selectedImages**: A list of selected images for the post, initially empty.
- **Methods:**
    - **_pickImageFromGallery**: A method to pick an image from the gallery and add it to the selected images list.
    - **_pickImageFromCamera**: A method to pick an image from the camera and add it to the selected images list.

    - **_createPost**: A method to create a new post with the entered data and selected images. It handles the submission of the post and displays a message afterward.
    - **_buildImageButton**: A helper method to build image selection buttons.
    - **_buildSelectedImagesSection**: A helper method to display the selected images in a GridView.
    - **_message**: A method to display a snack bar message after the post is submitted.
- **Build Method (build):**
    - Constructs the UI for the create post page, including text input fields, channel selection, image selection, and the submit button.
    - The UI includes error handling for empty descriptions and loading indicators during post creation.
    - A snack bar message is displayed when the post is successfully submitted.
- **Dispose Method (dispose):**
    - Disposes of the **_descriptionController** to avoid memory leaks.

The UI is designed using the Lato font from Google Fonts, and it follows a clean and user-friendly layout. Users can easily create posts with descriptions, select a channel, and add images to share their thoughts and experiences.

<u>Accounts Page</u>

The **AccountsPage** class represents the user account page where users can view their profile details, change the theme (dark or light), access a user guide, and sign out. Here's an overview of the key elements in the code:

- **Profile Details (profileDetails method):**
    - Displays the user's username along with a verification icon.
    - The text color is determined based on the theme (dark or light).
- **User Guide (UserGuide method):**
    - Represents a list tile for accessing a user guide. Currently, it has a placeholder action.
- **Theme Customization (ThemeCustomization method):**
    - Utilizes the **Consumer** widget from the **provider** package to observe changes in the theme state.
    - Displays a **SwitchListTile** for toggling between dark and light themes.
    - The **ThemeProvider** is used to manage the application's theme state.

- **Sign Out (SignOut method):**
    - Represents a list tile for signing out.
    - Invokes the **signOut** method when tapped, which signs the user out and navigates to the authentication page (**AuthPage**).
- **Sign Out (signOut method):**
    - Signs out the user using Firebase Authentication.
    - After signing out, it removes all routes and navigates to the sign-in page (**AuthPage**).
- **Build Method (build):**
    - Constructs the UI for the user account page.
    - Includes the app bar, profile details, user guide, theme customization, and sign-out sections.

The UI is designed with a clean and user-friendly layout, providing users with the ability to customize the theme, access a user guide, and sign out of their account. The use of the **provider** package facilitates state management for the theme, ensuring a responsive and dynamic user experience.

<u>Commenting Page</u>

The **CommentPageScreen** class represents a page for posting comments on a specific post. Here's a summary of its key features:

- **Constructor (CommentPageScreen):**
    - Takes a **PostModel** as a parameter to represent the post on which comments are being made.
- **State Class (_CommentPageScreenState):**
    - Manages the state of the **CommentPageScreen**.
- **Text Controllers:**
    - **_commentController**: Manages the text entered by the user for the comment.
- **AppBar:**
    - Displays an **AppBar** at the top with the title "Commenting."
- **Body:**
    - Consists of a **Column** containing:
        - A **PostWidget** displaying the original post that users are commenting on.

        - A **TextField** for entering comments.
        - An **ElevatedButton** to submit the comment.
- **Comment Submission (postComment method):**
    - Validates that the comment is not empty.
    - Creates a new **PostModel** instance representing the comment with relevant data.
    - Calls **StateManager.postController.addComment** to add the comment to the post.
    - Displays a snackbar with a confirmation message.
    - Navigates back to the previous screen.
- **UI Styling:**
    - Utilizes the **GoogleFonts** package for applying custom fonts.
    - Applies a maroon color theme throughout the UI.
- **Message Display (_message method):**
    - Displays a snackbar with a message confirming that the comment has been submitted.
- **Disposal of Resources (dispose method):**
    - Disposes of the **_commentController** to release resources when the widget is disposed.

This **CommentPageScreen** provides a user-friendly interface for users to view the original post and submit comments. It is designed to be intuitive and responsive, with a clean layout and appropriate use of styling.

<u>Firebase Integration</u>

1. Firebase Authentication:

- Firebase Authentication is used for user authentication.
- The **FirebaseAuth** instance is initialized and utilized for signing in, signing out, and checking the user's authentication status.
- The **AuthPage** widget is used for user authentication, including sign-in and sign-up.

2. Firebase Firestore (Database):

- Firebase Firestore is used as the backend database to store and retrieve data.

- The **PostController** class interacts with Firestore to perform CRUD operations for posts.
- **PostModel** represents a data model for posts, including fields like **userID**, **username**, **description**, **release_date**, etc.
- Posts are retrieved from Firestore and displayed in the **FeedPageScreen**.
- The **createPost** method in **_CreatePostPageScreenState** is responsible for creating a new post in Firestore.
- The **initializeData** method in **_HomePageScreen** initializes the data from Firestore.

3. Firebase Storage:

- Firebase Storage is used for storing and retrieving images associated with posts.
- Images are picked from the gallery or camera using the **ImagePicker** package and then uploaded to Firebase Storage.
- The **createPost** method in **_CreatePostPageScreenState** is responsible for adding post data, including images, to Firestore.

4. Real-time Updates:

- Real-time updates are achieved through listeners. For example, in **_FeedPageScreenState**, a listener is added to the **PostController** to trigger a rebuild when posts change.
- This real-time update helps in keeping the UI in sync with the latest data.

5. Other Firebase Services:

- The **StateManager** class serves as a centralized manager for different controllers, including **UserController** and **PostController**.
- The **UserController** manages user-related functionalities and details.
- Channel information is static in this example, but it can also be stored in Firestore and fetched dynamically.

6. Comments and Threads:

- Comments are added to posts using the **addComment** method in **PostController**.
- A **CommentPageScreen** is provided to display the post and allow users to submit comments.

7. User Sign Out:

- The **AccountsPage** includes a sign-out option using Firebase Authentication.
- The **signOut** method is called when the user taps on the 'Sign Out' button.

8. Firebase Integration:

- The Firebase services are integrated and initialized in the **main.dart** file using **Firebase.initializeApp()**.

9. Dependency Injection:

- The **StateManager** class, along with various controllers, is used to manage and access different states and functionalities across the app.

10. Security Considerations:

- Proper security measures, such as validating user inputs and securing Firestore rules, should be implemented in a production environment.

Challengers and Solutions

- **Time**
  Due to half of the group dropping out we were struggling with finding time to finish while also balance other courses so unfortunately planned features had to be cut in order to focus on making the app functionable.
- **Google Authentication**
  There was a great difficulty and making the Google Authentication work with what we had planned to do and by the of the development it was fixed but we did not understand how it was fixed.