Design Document for Supper Solver

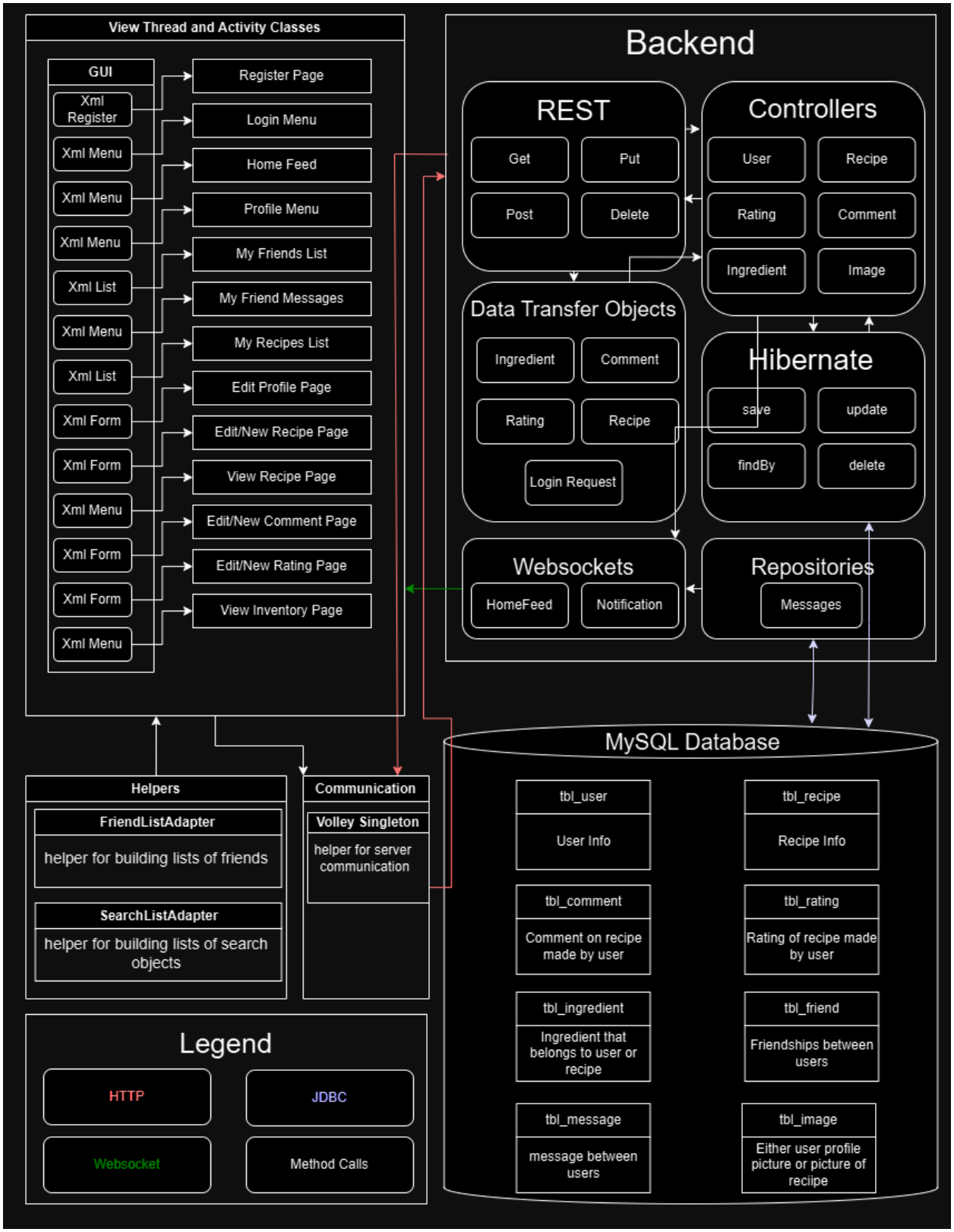
Group 2_Jabir_1

Member1 Carson Irving: 25% contribution

Member2 Aden Koziol: 25% contribution

Member3 Brandon Liao: 25% contribution

Member4 Ryan Holden: 25% contribution


View Thread and Activity Classes
GUI
Xml Register
Xml Menu
Xml Menu
Xml Menu
Xml List
Xml Menu
Xml List
Xml Form
Xml Form
Xml Menu
Xml Form
Xml Form
Xml Menu
Register Page
Login Menu
Home Feed
Profile Menu
My Friends List
My Friend Messages
My Recipes List
Edit Profile Page
Edit/New Recipe Page
View Recipe Page
Edit/New Comment Page
Edit/New Rating Page
View Inventory Page
Backend
REST
Get
Put
Post
Delete
Controllers
User
Recipe
Rating
Comment
Ingredient
Image
Data Transfer Objects
Ingredient
Comment
Rating
Recipe
Login Request
Hibernate
save
update
findBy
delete
Websockets
HomeFeed
Notification
Repositories
Messages
Helpers
FriendListAdapter
helper for building lists of friends
SearchListAdapter
helper for building lists of search objects
Communication
Volley Singleton
helper for server communication
MySQL Database
tbl_user
User Info
tbl_recipe
Recipe Info
tbl_comment
Comment on recipe made by user
tbl_rating
Rating of recipe made by user
tbl_ingredient
Ingredient that belongs to user or recipe
tbl_friend
Friendships between users
tbl_message
message between users
tbl_image
Either user profile picture or picture of reciipe
Legend
HTTP
JDBC
Websocket
Method Calls

Frontend

Communication with Back End

The frontend uses Android's Volley library combined with RequestQueue to efficiently communicate with the backend server via HTTP requests. Different activities/screens require and use different HTTP requests based on their respective function, unique requests are handled in the backend using unique URL mappings to send specific requests. Allowing for specific data interactions.

- Post:
    - **Purpose**: POST requests are typically used to create new resources on the server, like adding a new user or recipe.
    - **Implementation**: Create a JsonObjectRequest or StringRequest with Request.Method.POST to send JSON data to the backend.
    - **Example Use**: If a user submits a new recipe, the app sends a POST request containing the recipe details as JSON. The backend then processes and stores the new recipe in the database.
- Get:
    - **GET requests are used to retrieve data from the server, such as fetching user lists, recipes, or other stored resources.**
    - You would use JsonObjectRequest, StringRequest, or JsonArrayRequest with Request.Method.PUT to send an updated JSON object to the backend.
    - **Example Use:** If a user wants to search for a recipe or user, GET requests retrieve either a list of users or recipes from the backend, which are then displayed in the app's ListView.
- Put:
    - **Purpose:** PUT requests are typically used to update existing resources on the server, such as modifying user profile information or updating a recipe.
    - **Implementation**: You would use JsonObjectRequest with Request.Method.PUT to send an updated JSON object to the backend.
    - **Example Use**: When a user updates their profile information, a PUT request sends the modified data to the backend, which then updates the user's information in the database.

- Delete:
    - **Purpose:** DELETE requests are used to remove resources from the server, such as deleting a user account or a recipe.
    - **Implementation:** You would create a StringRequest with Request.Method.DELETE to specify the resource (usually by ID) to be deleted from the server.
    - **Example Use:** If a user wants to delete one of their recipes, a DELETE request is sent with the recipe's ID to the backend, which then removes the recipe from the database.

Screens

The screens within our application use adapters, volley, websockets, and other android utilities to perform their respective functions.

- CommentScreen, EditProfileScreen, FriendScreen, HomeFeedScreen, InventoryScreen, LoginScreen, MessageScreen, PostRecipeScreen, ProfileScreen, RatingScreen, RecipeScreen, SearchScreen, SendMessageScreen, SignupScreen.

Adapters

Adapters are used for features within our application that require manipulating lists to create dynamic screens.

MyCustomAdapter:

- The adapter takes an ArrayList<String> and a Context object, storing them as instance variables. This setup allows MyCustomAdapter to access the list data and interact with other components in the application's context.
- Uses the LayoutInflater to inflate a custom layout (custom_layout.xml) for each list item. Displays each string in the list within a TextView
- Includes a button in each item's layout with an OnClickListener.
- When clicked, retrieves extra data from the current activity's intent. Initiates an intent to open a different screen, passing along relevant data such as USERID, username, and the currently selected list item.

Volley Singleton

VolleySingleton is used for features within our application that require data to be accessed from the backend server via HTTP requests.

- The class is structured as a singleton, meaning only one instance of VolleySingleton is created and accessed throughout the application.
- The RequestQueue manages and executes HTTP requests in a queue. This instance of RequestQueue is created using Volley.newRequestQueue() and is initialized only if it doesn't already exist, reducing resource usage by reusing the same queue across the app.
- The addToRequestQueue() method allows adding requests to this centralized queue, making it easy to manage all network requests through the singleton instance.
- The ImageLoader object within VolleySingleton helps manage and cache image requests.
- The class uses LruCache to cache images, optimizing network efficiency and user experience by preventing unnecessary image reloads.

Websockets

WebSocketManager and the interface **WebSocketListener** are used within our application to provide functionality to features that require full-duplex communication between the frontend and backend.

WebSocketListener Interface:

The WebSocketListener interface defines the methods needed to handle WebSocket events. Classes that implement this interface can respond to:

1. **Connection Open**: onWebSocketOpen(ServerHandshake handshakedata)—called when the WebSocket connection is successfully established.
2. **Message Received**: onWebSocketMessage(String message)—called when a new message arrives from the server.
3. **Connection Closed**: onWebSocketClose(int code, String reason, boolean remote)—called when the WebSocket connection closes, providing the code and reason.
4. **Error**: onWebSocketError(Exception ex)—called when an error occurs in the WebSocket connection.

By implementing this interface, a class can handle WebSocket communication events directly.

WebSocketManager:

- getInstance() ensures only one instance of WebSocketManager exists.
- setWebSocketListener(WebSocketListener listener) attaches a listener that will handle WebSocket events.
- removeWebSocketListener() detaches the listener when it's no longer needed, preventing event handling when the WebSocket is inactive.

- connectWebSocket(String serverUrl) initiates a connection to the WebSocket server specified by serverUrl.
- sendMessage(String message) sends a message to the connected WebSocket server if the connection is open.
- disconnectWebSocket() closes the connection with the server, stopping further WebSocket communication.

Backend (Table Relationships and Columns described under MySQL diagram on page 4)

Communication with Front End

The backend uses mappings to update the MySQL database based one these http requests:

•Post: Used to send information when creating an object in the database (Also used for logging in as we do not want to pass credentials through the url)

•Get: Requesting information from the database, any identifiers or information passed from frontend will be in the url of the requests

•Put: Requesting an update of an object in the database, usually passing an identifier and the object you would like to update the current one with.

•Delete: Requesting to delete an object from the database, passed with an identifier

Controllers

The controllers specify the mappings of the http requests that are used by the front and back end to communicate with one another.

These include:

•User Controller

•Recipe Controller

•Rating Controller

•Ingredient Controller

•Image Controller

•Comment Controller

Repositories

The repositories specify how the java application will interface with the MySQL database. Most of the simple repository commands will be already implemented by hibernate, though more complex actions will need to be manually added.
We currently have a repository for all objects included with the controllers above and a repository to store messages which are created within our chat websocket, which is why there is no controller needed for messages.

Models

The model classes are defining the objects themselves, and what parameters they must contain. Inside the model classes are also where the relationships between objects are defined to be used to construct the MySQL database.
We currently have these models/objects:

- MUser
- MRecipe
- MRating
- MComment
- MMessage
- MIngredient
- MImage
- MFriend

Data Transfer Objects (DTO)

The Data Transfer Objects are smaller objects that are used as a subset of requests, often to make it easier for the front end to create an object without needing to fill all of the parameters that you might otherwise need to fill. We have also used a DTO to allow the front end to pass a username and password in JSON for login inside of a post.
These are the DTO's that we currently use:

- CommentDTO
- IngredientDTO
- LoginRequestDTO
- RatingDTO

•RecipeDTO

WebSockets

Web sockets are what we use when our front end and back end need to use full-duplex communication. We currently use this for our chat, notifications, and for our home feed algorithm.

Our current websocket classes that we have are:

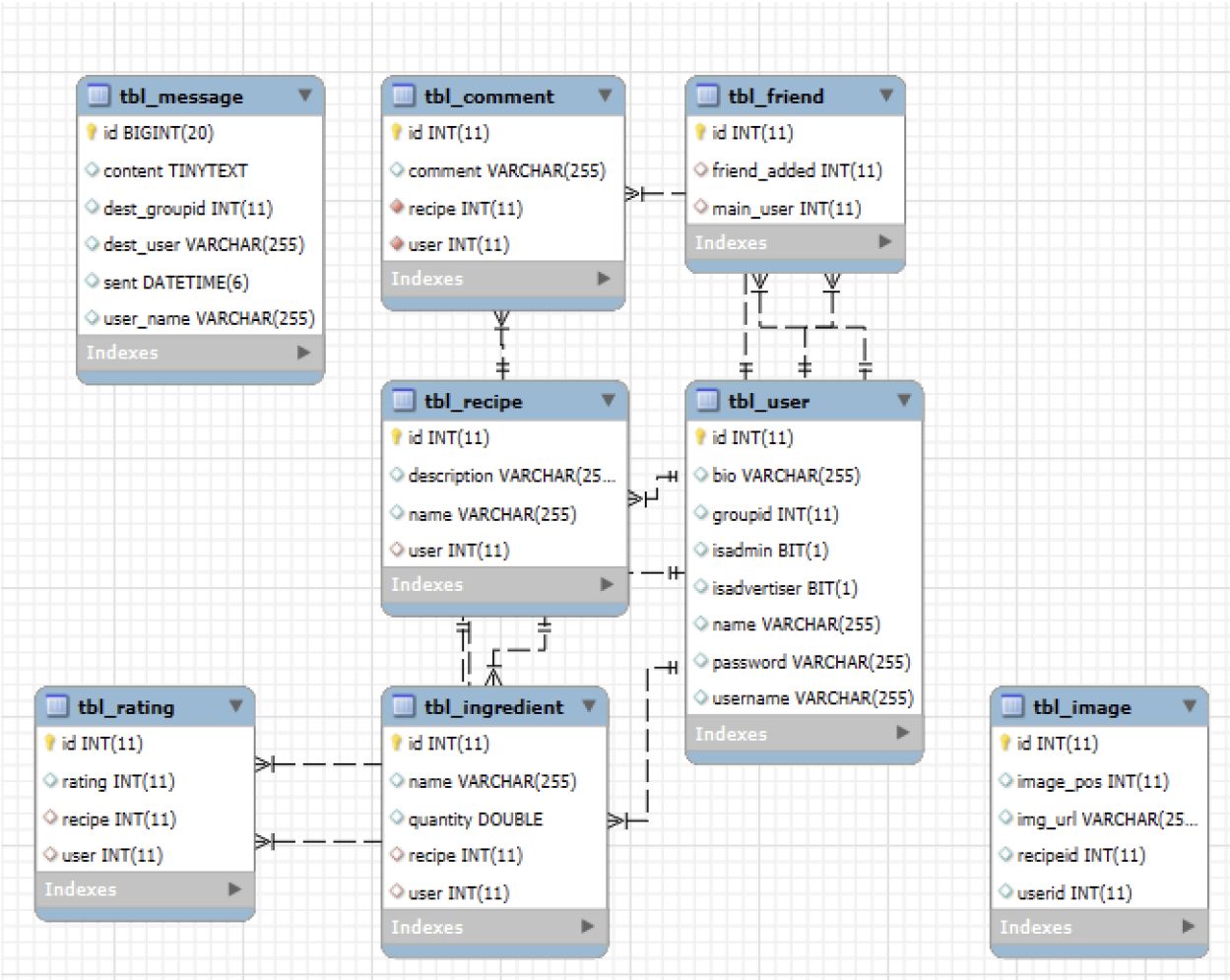
•HomeFeedWebsocket

•NotificationWebSocket (Contains chat features as well)

•WebSocketConfig (Needed class to get spring boot to recognize our websockets)

Other

Finally we have one other class that does not fit into these other categories, that being our image host class. It is used to specify where to store our image files on the server, and where to host them so they can be grabbed from a link.



tbl_message is the table responsible for storing messages sent over the chat websocket.

1. Id (Unique identifier that is auto generated on save)
2. Content (The text itself stored inside the message)
3. dest_groupID (If group message what groupID it was sent to, otherwise 0)
4. dest_user (If DM username of the user it was sent to, otherwise null)
5. Sent (Date Time message was sent, used to sort by timestamp)\
6. User_name (Username of sender)

Tbl_comment is the table responsible for storing comments on recipes.

1. ID (Unique identifier that is auto generated on save)
2. Comment (Text content of the comment)
3. Recipe (Recipe object the comment was left on)
4. User (User object of the user who commented)

Tbl_friend is the table that stores friend relationships between users

1. ID (Auto Generated Unique Identifier)
2. main_user (The user that added another user)
3. friend_added (The user that was added by the main_user)

Tbl_user is the table that stores the parameters for our user object.

1. ID (Auto Generated Unique Identifier)
2. Bio (Users bio that is shown on their profile)
3. GroupID (Integer that represents the group that a user belongs to)
4. IsAdmin (Boolean that represents if a user is an admin or not)
5. IsAdvertiser (Boolean that represents if a user is an advertiser)
6. Name (Holds users actual name)
7. Password (Holds users passwords for login)
8. Username (Holds users username for login)

Tbl_recipe contains the parameters for our recipe object.

1. ID (Auto Generated Unique Identifier)
2. Description (Stores description of recipe)
3. Name (Stores name of recipe)
4. User (User object that represents the user that posted the recipe)

Tbl_ingredient contains parameters for ingredients and what object they belong to

1. ID (Auto Generated Unique Identifier)
2. Name (Name of ingredient)
3. Quantity (Amount of ingredient a user has)
4. Recipe (A recipe object if the ingredient is contained within a recipe)
5. User (A user object if the ingredient is owned by a user)

Tbl_rating contains rating value and who assigned it to what recipe

1. ID (Auto Generated Unique Identifier)
2. Rating (Integer rating 1-5 of a recipe by user)
3. Recipe (Recipe object that the rating was left on)
4. User (User object that left the rating on the recipe)

Tbl_image contains image parameters and the link to fetch them with from the server

1. ID (Unique identifier that is auto generated on save)
2. Image_pos (An integer that declares what the position of the image is within a recipe or a users profile)
3. Img_url (Stores the url of the image, where the image is hosted)
4. recipeID (The ID of the recipe that the image is associated with [0 if associated with a user])

5. userID (The ID of the user that the image is associated with [0 if associated with a recipe])

Relationships:

Recipe
Recipe one to many Comment

Recipe one to many Ingredients

Recipe one to many Rating

User
User one to many Comment

User one to many Recipe

User one to many Rating

User one to many Ingredients

User Many to Many Friends