Jesse Goodspeed
Nikki Jack
Michael Saunders
Naveed Shah

TuringChat: Project Report

Introduction

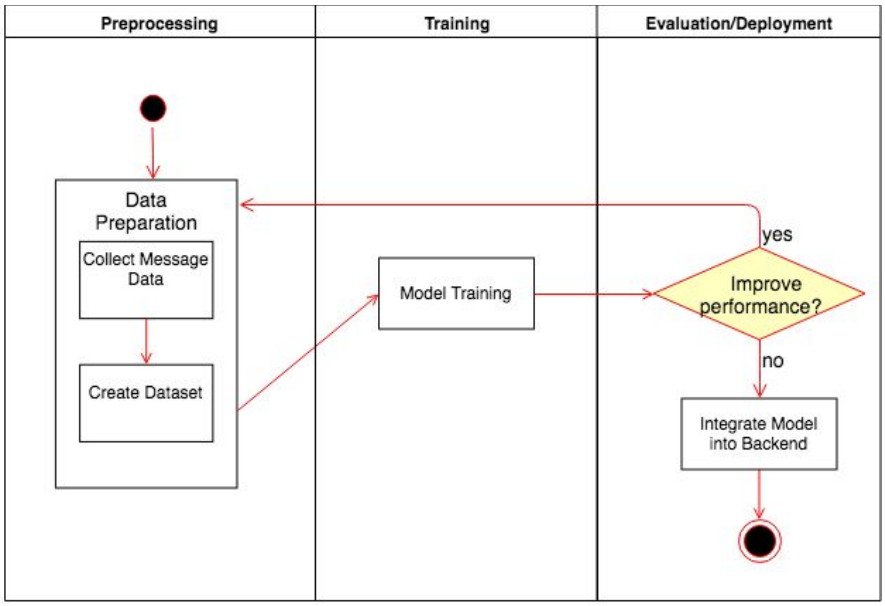
We implemented an online messaging application that features multiple chatbots and can run on Android and iOS mobile platforms. This project is implemented in Python, Swift, and Java.

Generative Chatbot

To create a conversational chatbot, we had to determine which type would be best for the application. Natural language chatbots fall into two general categories: generative and selective. Whereas selective models output canned responses, generative models are much more adaptable and can generate nuanced responses dependent on sufficient training. Machine learning techniques are used to produce both categories of chatbots. For our project, we implemented a generative chatbot to allow for the most versatile responses. We wanted to have a generalized conversational agent that was capable of responding uniquely to different messages.

We followed a similar implementation as that used for machine translation where input sentences are outputted as foreign language translations. This is commonly done with recurrent neural networks that are able to account for semantic relationships over a series of data. Instead of mapping phrases to their counterparts in foreign languages, our implementation maps phrases to likely responses in the same language, based on an individual's messaging history.

Messaging data was collected for two team members from Facebook Messenger and Google Hangouts. Once the messaging data was collected, the data was formatted into a single dataset to use for training the machine learning model. For the training phase, a script was run in a cloud GPU server to run through training iterations quickly. We later moved model training onto a local GPU-machine for new chatbots to lower costs. After training is completed (training epochs were 500K iterations), models are evaluated to gauge performance and considered for integration into the backend of the messaging application. Below is an activity diagram of the different steps involved in developing the chatbot model.



Backend

Third party libraries used:

- Channels → For websockets
- Tensorflow → For model training and use
    - Numpy → utilized by Tensorflow

Structure of the backend:

Within the backend folder there is a folder for each app in the project, an app is an encapsulated component of a project. There are two main components the user folder and the chat folder. The user app handles user registration login and authentication. The chat app handles the http requests for getting and creating rooms for users to select. The chat app also contains the consumer which is used to handle websocket connections which also handle the room model and group socketing over the channel layer for group communication.

API routes(HTTP):

POST	/user/register	Create new user
POST	/user/login	Login user using auth return unique string for front end to verify with
GET	/chat/create	Return a list of available bots to include in a room
POST	/chat/create	Create new room with a selection of bots given at GET
GET	/chat/rooms	Return list of rooms to join

Websocket(WS):

/socket	Initiates connection to websocket

Websocket consumer functions:

Connect:	Open a websocket on client request
Recive_json:	Whenever a json message is sent to the consumer, dispatches some routine
Disconnect:	Handle cleanup on websocket disconnect

Consumer routines:

Join_room: Connects consumer to room over channel layer
Send_room: Sends message to connected room over channel layer
Leave_room: Removes consumer from room in channel layer

Here is the format for ***json*** sent over the websockets:

```
{       “command” :
        “room”:
        “Message”: (for send command)
        “uid”: (for join command)
        “username”:
}
```

Two Examples:

```
{ "command": "join", "room": "1",  "username": "michael", "uid": 1 }
{ "command": "send", "room": "1",  "username": "michael", "uid": 1, “message”: “Hello world!” }
```

Android

The Android application utilizes the “org.java-websocket” library to implement it’s features. The Android application contains 9 Java files and 8 Layout files. When the application is opened by the user, it navigates to the activity_login.xml layout which is linked to the LoginActivity. The application prompts the user for their email and password. Upon entering their information and clicking the Login button, the LoginActivity sends a JSON request to the backend server located online through a POST request. The server sends back a response and the LoginActivity checks the response code. If it was a successful login (code 200), the activity changes to the Recycler view. If the login was not a success (code 500), it will display an error to the user using Snackbar. The Recycler view initiates a Chatroom content class, an ItemFragment, and an ItemRecyclerViewAdapter. These work in sync to attain the list of chatrooms from the server and populate the recycler view which the user sees as a “list of chatrooms”. Once a user clicks on a chatroom, the onBindViewHolder function in the ItemRecyclerViewAdapter saves the room number using shared preferences and starts the ChatRoomActivity. The ChatRoomActivity connects to the websocket and makes a request to join the respective room that the user clicked on. Then the user can send messages to this Chatroom which get delivered to the server in JSON format. The server will respond back with either a response from a bot or another live user in the chatroom. The user can conversate and communicate with the bots and/or the other live users in the chatroom. The bots will always respond to each message the user sends, creating an interactive, lively, and entertaining session. The user can have a lot of fun playing with the bots trying to attain funny responses or the user can try to communicate properly and dispher what the bots are saying. All in all, the app functions as an entertainment app.

IOS

Third party libraries used:

- Starcream → for Websockets
- PKHUD → for load progress indicator animation

iOS app description:

Structure: each ViewController representing a visible screen has a helper swift file with a class extension. Inside the extension there is additional functionality such as making an HTTP request, websocket connection callbacks and Core Data (internal database) manipulation.

Rooms and Messages are saved in Core Data. These models have one to many relationship (One Room has many Messages). So even when there is no Internet connection, user can review the contents of chatrooms in which he previously was hanging out.

User gets informed about **ERRORS** with a pop-up view 1) if there is no internet connection, 2) if they have entered wrong login credentials, 3) if the username they're trying to register is already in use, 4) if any of input fields are empty, 5) if password too short during registration

Inside the chat room, the **message bubbles** have dynamic height, and the view automatically scrolls down to the latest message in the room. Also only messages from other people include a username. (All this was hard to align and implement.)

Challenges and Resolutions

Training the computational model via laptop was slow and taking more than eight hours. This prompted us to look into other training schemes such as using GPUs with a cloud service until we found a cheaper means with a GPU-machine.

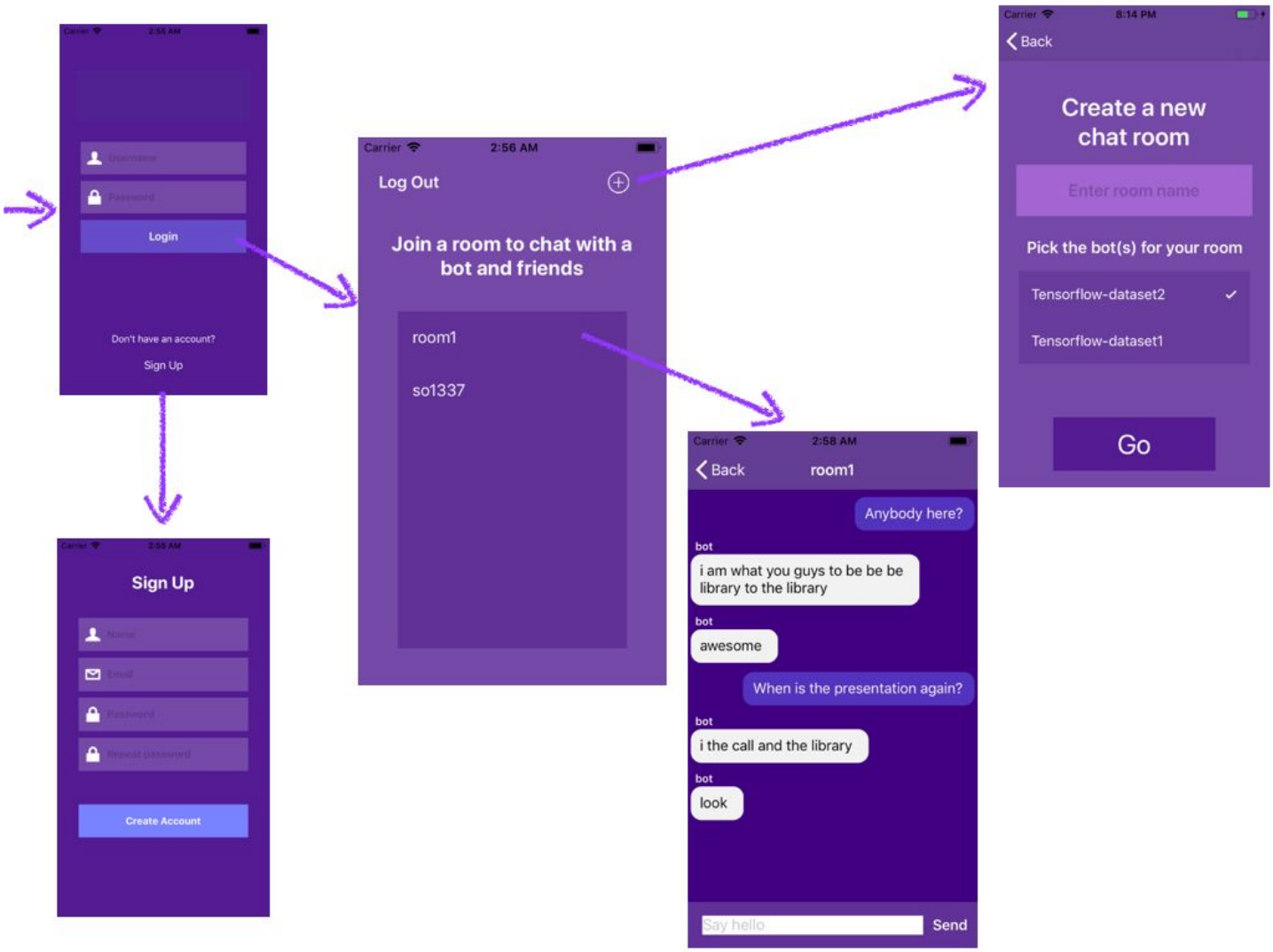
Password hashing: Mongoengine (MongoDB) User model in Django had no way to use the Python "bcrypt lib" to add a password hashing functionality. Since saving user passwords securely is very important, we switched to Django built-in (SQLite) User model that hashes passwords automatically with PBKDF2 hashing algorithm.

Authentication: Django "Channels" (socket lib) has built-in sessions authentication functionality, but we could not use it because our product is a mobile app, and not a web app. Therefore, there is nowhere to put cookies. Django JWT authentication worked for regular REST routes, but did not work with sockets. Our sockets needed to tell users apart, so to make it work, we had to sacrifice security. The user id is currently the authorization token that is sent to mobile apps during successful login. No authentication is placed for the socket connection handshake. Once the mobile apps are connected to socket, they send a message with a "join" command that also includes the authorization token (user id), so that the socket knows which user is going to which room.

Achievements

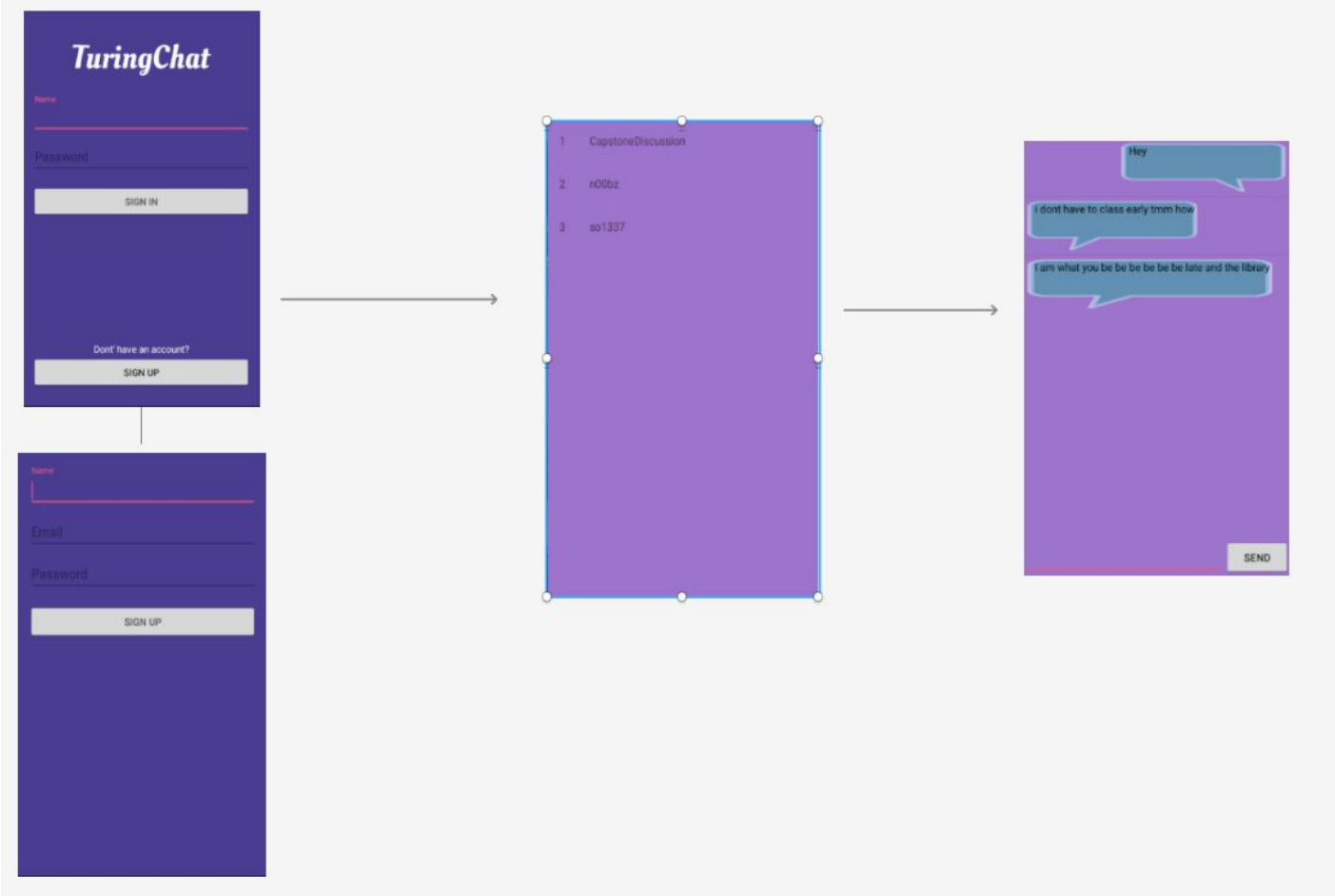
- Created message dataset and trained a responsive conversational chatbot
- Deployed Django REST server and redis server on Digital Ocean
- Deployed functioning chatbot on backend server
- Two bots running in tandem on same server using tensorflow
- Designed and implemented user interfacing front end applications that operate in both mobile iOS and Android environments.
- Successfully integrated and tested backend and mobile applications (they work together)

IOS Wireframes



iOS UML Diagram (Autogenerated)

Android



File Structure for Android:

Java:

- ChatArrayAdapter
- ChatMessage
- ChatRoomActivity
- ChatRoomContent
- ItemFragment
- LoginActivity
- MyItemRecyclerViewAdapter
- RecyclerActivity
- SignUpActivity

Layouts:

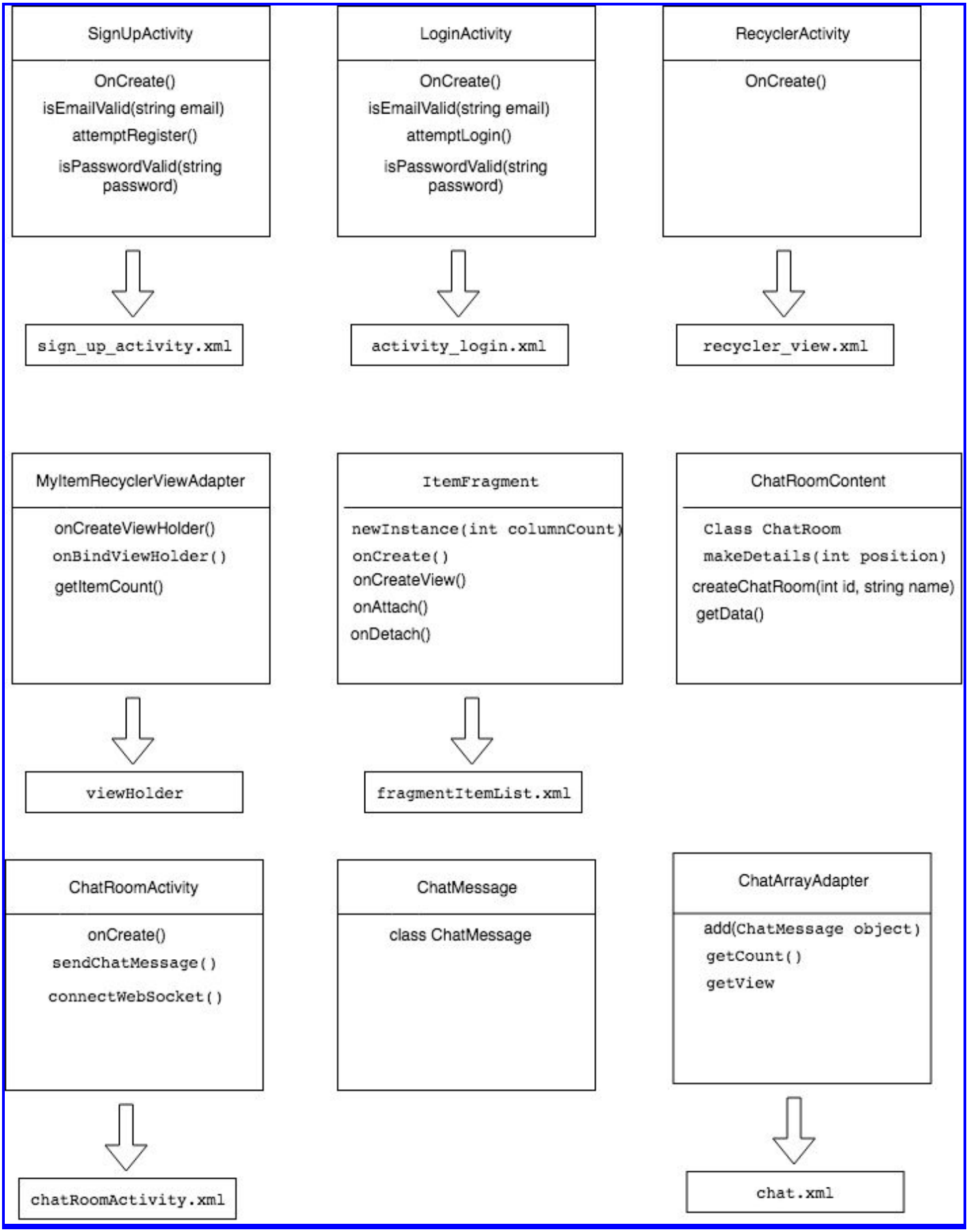
- Activity_login.xml (The user login page)

Chat.xml (The space for each chat message)
Chat_room_activity.xml (The background activity that handles the chat room)
Fragment_item.xml
Fragment_item_list.xml
My_message.xml
Recycler_item.xml
Sign_up_activity.xml

Drawables (other then Android default):

Bubble_a.png (The incoming message bubble displayed to the user)
Bubble_b.png (The outgoing message bubble displayed to the user)

Android Class Diagram


SignUpActivity
OnCreate()
isEmailValid(string email)
attemptRegister()
isPasswordValid(string password)
sign_up_activity.xml
LoginActivity
OnCreate()
isEmailValid(string email)
attemptLogin()
isPasswordValid(string password)
activity_login.xml
RecyclerActivity
OnCreate()
recycler_view.xml
MyItemRecyclerViewAdapter
onCreateViewHolder()
onBindViewHolder()
getItemCount()
viewHolder
ItemFragment
newInstance(int columnCount)
onCreate()
onCreateView()
onAttach()
onDetach()
fragmentItemList.xml
ChatRoomContent
Class ChatRoom
makeDetails(int position)
createChatRoom(int id, string name)
getData()
ChatRoomActivity
onCreate()
sendChatMessage()
connectWebSocket()
chatRoomActivity.xml
ChatMessage
class ChatMessage
ChatArrayAdapter
add(ChatMessage object)
getCount()
getView
chat.xml

Backend composite/class Diagram