

# Stanford University, CS 193A

## Homework 7: TreeChat

### (Pairs allowed)

*Assignment spec and idea by Marty Stepp. Thanks to CAs Sarah Egler, Shreya Shankar, Ashley Taylor, and Gracie Young for help creating and revising this assignment.*

This assignment serves as a capstone or "final project" for the course, spanning many topics seen throughout the quarter. The assignment also tests your knowledge of new topics such as services, notifications, user logins, and localization. Make sure to check out the **Development Strategy** at the end of the document for tips on getting started. You are allowed to use any libraries you like on this assignment.

This is a **pair assignment**. You may complete it alone or with a partner. Please read our [Pairs web page](#) for some of our policies and suggestions if you do plan to work in a pair.

### Starter Files:


-  [treechat-starter-files.zip](#) (ZIP of all starter files)

Since this is your final assignment, we want you to build most of it yourself and do not provide much to get you started. The ZIP above contains only a few image and sound resource files that you may want to use.

### File and Project Names:

- Your **Android Studio project's** name must be EXACTLY the following, case-sensitive, including the underscores:
  - cs193a\_hw7\_**SUNETID** (example: cs193a\_hw7\_ksmith12)Or if you are working in a pair, please list both partners' SUNetIDs separated by an underscore:
  - cs193a\_hw7\_**SUNETID1\_SUNETID2** (example: cs193a\_hw7\_ksmith12\_tjones5)
- You must have activity/view classes with the exact names given below under "Program Description," case sensitive, in corresponding file names (e.g. class FooActivity in file **FooActivity.kt**). You may add other activities, views, classes, and files to your app if you

like, but you must have the exact activity file names below at a minimum, spelled and capitalized exactly as written here.

- When you are finished, ZIP your project ([instructions](#)) into a file with the following exact file name, including spelling and capitalization:
  - cs193a\_hw7\_**SUNETID**.zip (example: cs193a\_hw7\_ksmith12.zip)
- If you use **libraries** from the Maven repository (ones that are added to a project by declaring them in your **build.gradle** file), these will be auto-downloaded when we compile your project to grade it. If you use any other local libraries that you downloaded manually, make sure they are located in **app/libs/** and are included in your ZIP that you turn in.
- Turn in link:  Turn in HW7 here.

## Program Description:

For this assignment you will write a chat program named **TreeChat**. This is a multi-user chatting system where users can join chat channels and send messages to each other. This program is somewhat similar to a very simplified version of the popular Slack app/website. It also bears resemblance to an older system called Internet Relay Chat ("IRC") that was popular in the 1990s and early 2000s.

Your app will store data about all chat rooms ("channels"), users, and messages that are sent. The exact format and storage of this data is up to you. But we highly recommend that you use **Firestore** because it has been taught in class and has good support for updating clients about newly arriving data, such as chat messages, which will be helpful for this project. See the **Data section** of the spec for more information.

An important aspect of the assignment is that once opened, your app should continue to receive messages even after the app is closed. You will achieve this by writing a **service** that will run in the background and will display **notifications** when new messages arrive.

Since this is your final assignment, we want you to have the freedom to design its UI and activities as you see fit. But your app should contain at least the following functionality. For each category of functionality, we will also describe that functionality's name and appearance in our own reference solution. If you want to try to match our names or appearance, that is fine, but you are not required to do so and are encouraged to be creative.

- **Welcome screen:** Provide some kind of initial screen or activity displaying the app's name, a logo of some kind, and some buttons or options to proceed to the rest of the app. The exact appearance is up to you.

In our reference solution, we call this `WelcomeActivity`; a screenshot is below.

- **Sign in:** You must provide a mechanism for users to log in to TreeChat. You can do this any reasonable way you like, so long as it uses "real" user accounts that are actually stored on a server. In particular, the ways shown in lecture are acceptable, such as:
  - *Firestore login:* We discussed how **Firestore** allows you to create and sign in accounts using its FirebaseAuth system. If you use this option, you need to write a UI that prompts the user to enter their account name or email and their password to log in. In this option you must also provide a "Create Account" feature where a user can sign up for an account to use the app.
  - *Google/Facebook/Twitter/etc. third-party login:* We also discussed how you can use a library such as Google Sign-In to allow a third party tech company to completely manage accounts for your app. If you use this option, you do not need to explicitly provide any "Create Account" UI because the user would create their account by going to the third-party company's web site. These libraries typically provide their own UIs for popping up a sign-in page that then redirects back to your app and sends you the signed-in user information.
  - *Writing your own account system from scratch:* We do not recommend this option. But if you really want to, you can write your own system that stores user names and (hopefully encrypted) passwords in your own data storage. When the user tries to log in, you would need to verify their user name or password yourself. In this option you must also provide a "Create Account" feature where a user can sign up for an account to use the app. Please note that using a local SQL database is not an appropriate choice for this feature because the user accounts should be stored in some global server area for your app, not on each user's individual phone, because the latter is insecure. Again, you probably do not want to choose this option because it is the most work and is hard to build securely, but if you want to try to do so, it is up to you.
  - *Multiple sign-in methods:* You can optionally allow the user to create accounts and sign-in in multiple ways. For example, you could provide both a FirebaseAuth sign-in and a Google Sign-In option. You are not required to do this, but it could make your app more accessible to a broad range of users.

Whichever user sign-in system you use, your account creation should be reasonably **robust**. For example, if the user tries to create an account that already exists, or if the user types in an incorrect password, or if the user tries to sign in using an account that does not exist, your app should provide a descriptive error message rather than crashing or displaying nothing.

If you find that you are unable to get any of the login systems/libraries working, for partial credit you can write a "**fake sign-in**" system. Such a fake system would simply store a fixed set of user account names and passwords in some local storage in the app such as a file. You would provide a UI for the user to type their account name and password. Your code would check against the local data to make sure the username is found and the password is correct when the user tries to log in. This will not receive full credit, but it would allow you to complete the rest of the app and test it and receive partial points.

In our reference solution, we used multiple sign-in methods including `FirebaseAuth` and `Google Sign-In`. We call this functionality `SignInActivity` and `CreateAccountActivity`; a screenshot of the sign-in is below.

- **View list of channels (*OPTIONAL*):** Once the user has signed in, you should direct them to an activity displaying all currently available chat channels. A chat **channel** is an online room that users can join and send messages to each other. This is similar to the concept of a channel in an app such as Slack or a chat service such as IRC. A user can join a channel and send a message to it, and every user in the channel will see the message. It is fine for your app to restrict a user to a single channel at a time; if they exit channel A and enter another channel B, they see the messages for B only, not A.

Your list of channels should be some kind of scrollable list view of all channels active in the system. A `ListView` would be fine for implementing this. The user should be able to click on a channel to join it.

You should also provide a mechanism for a user to create a new channel. For example, you could place an `EditText` at the bottom of the page allowing the user to create and join a new room. The lifespan of channels is up to you; in some systems such as IRC, a channel disappears once no users are in it. In other systems, such as Slack, a channel persists forever unless it is explicitly deleted by an administrator. Your app does not need to support any administrator features and does not need to support deleting chat channels, but you are welcome to provide these as optional extra features.

NOTE: This activity/feature is *optional* but recommended. If you want to make the assignment simpler, we will allow you to implement your app with all users and messages in a single, global chat "channel" rather than having separate channels. This would simplify the implementation but would make the app less featureful. If you do not implement the channel list feature, then on sign-in you should send the user into the one-and-only channel to begin chatting with the other users.

In our reference solution, we call this `ChannelListActivity`; a screenshot is below.

- **View a specific channel:** Once the user has chosen to join a particular chat channel, jump to a new activity displaying that channel. You should somehow display a list of the users who are presently in the channel, as well as a list of messages sent to that channel. You should also provide a way for the user to exit the channel and go back to the list of channels.

You must provide a UI for the user to type a message and send it to the channel. Every message is a broadcast to the entire channel and will be received by all other users currently in the channel. (By default this app does not support a "DM" direct message system from one user to a single other user.) The simplest UI for sending a message would be to place an `EditText` and Send button somewhere on the screen.

Note that you will need to refresh the channel as new messages arrive. If you use Firebase to store your data, this will be easier, because Firebase has nice functionality to "watch" a given child of data within its overall data set. If you write code to listen for value events on the appropriate child in your data that corresponds to the user's current channel, your code will be automatically notified when new child data nodes (messages) arrive, so that you can update the screen to display them. Also note that the list of users in the channel should behave similarly; if a new user joins the channel, or an existing user exits the channel, the screen should promptly update to reflect this change.

When a user joins a channel, there will be messages that were sent to the channel prior to the user arriving. How you handle these messages is up to you. If you want to fetch and display all messages ever sent to that channel in its lifetime, that is fine. Or if you want to display only messages newer than the date/time at which the user arrived, that is also fine. (Slack does the former; IRC does the latter.)

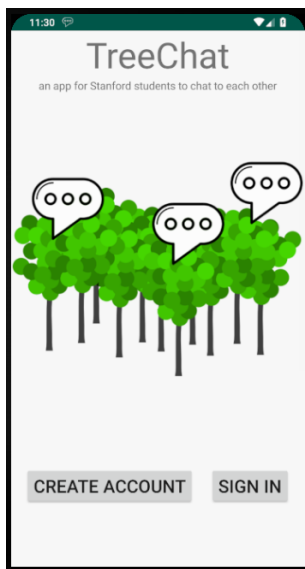
In our reference solution, we call this `ChannelActivity`; a screenshot is below.

- **Notification of new messages:** Once the app has loaded, even if the user exits the app, you should still display **notifications** to let them know when new messages have been received. You will do this by writing a **service** that watches for new messages and builds notifications as they arrive. When the user clicks on the notification, you should jump back into the app and go to the appropriate channel to see the messages. If you are using Firebase as recommended, you can listen for value events on the user's last known channel.

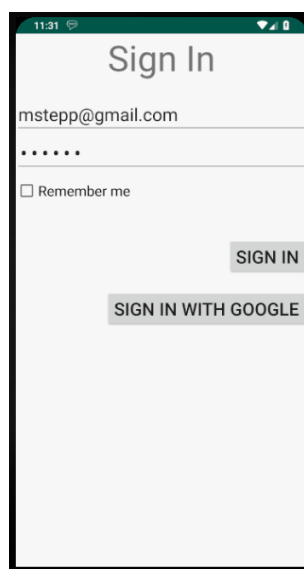
We will be somewhat lenient about the exact details of the notifications. For example, in most chat programs the notification contains the text of the message, but you do not need to do this. Also, many chat programs consolidate repeated or multiple notifications into a single one, but you do not need to do this. The exact appearance and behavior of the notification is up to you, so long as they appear when messages arrive and are clickable to view the messages.

In our reference solution, we call our service `TreeChatService`; a screenshot of a notification from it is below.

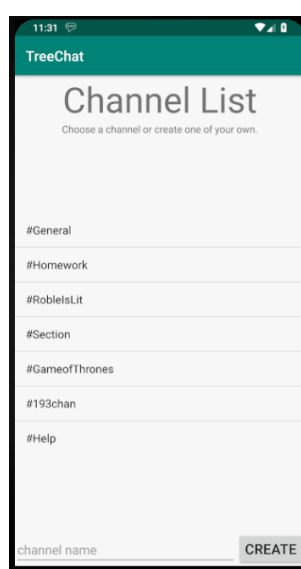
The following **screenshots** show the appearance of the various UI in our reference solution. Again, you are not required to exactly match our names or appearance, but you may try to do so if you like.



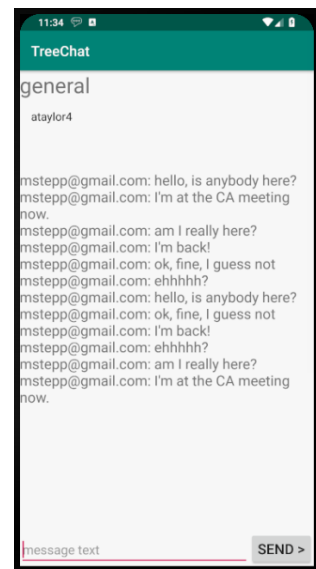
WelcomeActivity



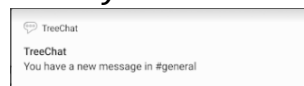
SignInActivity



ChannellistActivity



ChannelActivity



notification of new  
message

*Screenshots of app's various activities*

## Data:

As stated previously, we strongly recommend that you use **Firestore** to store the data for your TreeChat app. The exact format of this data is up to you. But in our own reference solution, we use a representation similar to the one described below.

An individual **message** is a JSON object containing the author of the message, the text of the message, and the time it was sent.

```

1 {
2   "from": "treebot@example.com",
3   "text": "Welcome to TreeChat! Thank you for using this app.",
4   "timestamp": "Thu Mar 07 00:50:47 PST 2019"
5 }
```

A **channel** stores a name, a description, and a list of messages that have been sent to it.

```

1 {
2   "name": "general",
3   "description": "Company-wide announcements and work-based matters",
4   "members": [
5     ...
6   ],
7   "messages": [
8     ...
9   ]
10 }
```

```

9     ]
10  }
```

A **user** stores a username, a real name, an email address, and perhaps other details about the user.


```

1 {
2     "username": "test",
3     "name": "Testy Testerson",
4     "email": "test@test.org",
5     "human": true
6 }
```

The **overall data set** is some combination of the above.

```

1 {
2     "treechat": {
3         "channels": {
4             "general": {...},
5             "help": {...}
6         },
7         "users": {
8             "test": {...},
9             "treebot": {...}
10        }
11    }
12 }
```

Our example file  [initial\\_data.json](#) is a hypothetical example of some channels and users that might be useful to you, though you do not need to use it or match its structure.

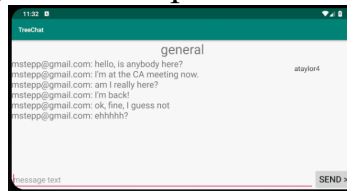
## Extra (Optional) Features:

It's already a lot of work to build the basic app and add the previously described features. But if you want to extend this assignment further, here are some ideas, or come up with your own:

- **Localization:** Localize the text and some images in your app so that it localizes to at least one other locale. You don't need to localize the actual messages sent by the users to the channels, but the overall welcome and sign-in text and images can be localized. If you choose to do this, make sure to tell us in your comments and/or app UI which locale you added so that we'll know how to test it.
- **Multiple sign-in methods:** As mentioned in the previous section about sign-in, you can support multiple ways of signing in the users. For example, you could support both Google Sign-In and Facebook Sign-In.
- **"Remember me" checkbox:** Add a checkbox or other UI that remembers the most recent logged-in user so that person can log in again without needing to type their username and

password. You could do this by storing information in the app's Preferences. (Really this information should be stored in an encrypted form, but since this app is just for fun, you do not need to do so.

- **User profile pictures:** Make it so that users can set a profile image for their account. You could even make it so that the user can take a profile picture using their device's camera.
- **Direct user-to-user messages (DMs):** Right now the app supports only channel-wide broadcast messages. Extend it to allow one user to directly send a message to another. Slide up in those DMs!
- **Better layout for landscape mode:** Include separate layout files for landscape orientation (in **layout-land/**) that better take advantage of the wide screen. The following screenshot is an example of the channel activity in landscape mode.



- **Ability to edit or delete a past message:** Currently once a message is sent to a channel, it is there forever. Make a UI so that the person who sent a message can delete and/or edit that message.
- **"Like" button or reactions:** Make it so that user can "like" a message sent by another user, or can "react" to a message with various smiley faces or emoji. Show a message's count of likes next to that message.
- **Formatting on messages:** Allow formatting on messages, such as with Markdown, HTML, or another format. You could allow colors, fonts, and other formatting.
- **Emoji support:** Allow the user to type emoji in their messages.
- **Sound effects:** Make your app play sounds when a user sends or receives a message.
- **Administrator or moderator features:** Allow a user to be specified as an "admin" or "mod". Give mods extra functionality on the channel / channel-list UI that allows them to rename or delete channels, edit or delete all messages, and so on. The mechanism by which a person becomes a mod is up to you. Perhaps the admin can designate a person as a mod, or perhaps the person who creates a channel becomes the mod of that channel, etc.
- **Channels or messages go away:** Make it so that when everyone exits a channel, the channel goes away. (You might want to have a few channels that don't go away, such as "General" or "Help.") Or make it so that a user's messages disappear after some amount of time, such as 24 hours or 5 minutes, so that the messages are similar to ones on SnapChat.
- **Other:** This assignment has lots of opportunities for extension features. Please feel free to think of your own and try them out. Be creative and have fun!



## Style Requirements:

Here are some important style requirements below that we ask you to follow. If you do not follow all of these requirements, you will not receive full credit for the assignment.

- **Comments:** Write a **comment header** in your main activity **.kt** file containing your name and email address along with the name of your app and a very brief description of your program, along with any special instructions that the user might need to know in order to use it properly (if there are any). Also write a brief comment header at the top of every **method** in your **.kt** code that explains the method's purpose. All of these comments can be brief (1-2 lines or sentences); just explain what the method does and/or when/why it is called. You do not need to use any particular comment format, nor do you need to document exactly what each parameter or return value does. The following is a reasonable example of a comment header at the top of a **.kt** activity file:

```
1 /*
2  * Kelly Smith <ksmith12@stanford.edu>
3  * CS 193A, Winter 2049 (instructor: Mrs. Krabappel)
4  * Homework Assignment 6
5  * NumberGame 2.05 - This app shows two numbers on the screen and asks
6  * the user to pick the larger number. Perfect for Berkeley students!
7  * Note: Runs best on big Android tablets because of 1000dp font choice.
8  */
```

- **Redundancy:** If you perform a significant operation that is exactly or almost exactly the same in multiple places in your code, avoid this redundancy, such as by moving that operation into a helping method. See the lecture code from our lecture #1 for examples of this.
- **Reduce unnecessary use of global variables and private fields:** While you are allowed to have private fields ("instance variables") in your program, you should try to minimize fields unless the value of that field is used in several places in the code and is important throughout the lifetime of the app. Our lecture code from Week 1 shows examples of reducing fields by accessing values from widgets instead (using `getText` and similar). (We consider it okay to declare widgets as fields if you use the ButterKnife library and its `@BindView` annotation.)
- **Reduce mutability:** Use `val` rather than `var` as much as possible when declaring Kotlin variables. Use immutable collections instead of mutable ones where appropriate.
- **Naming:** Give variables, methods, classes, etc. descriptive names. For example, other than a for loop counter variable such as `int i`, do not give a variable a one-letter name. Similarly, if you give an `id` property value to a widget such as a `TextView`, apply a similar style standard as if it were a variable name and choose a descriptive name.

---

*Survey:* After you turn in the assignment, we would love for you to fill out our optional [anonymous CS 193A homework survey](https://web.stanford.edu/class/cs193a/homework/7-treechat/hw7-spec.html) to tell us how much you liked / disliked the assignment,

how challenging you found it, how long it took you, etc. This information helps us improve future assignments.

*Honor Code Reminder:* Please remember to follow the **Honor Code** when working on this assignment. Submit your own work and do not look at others' solutions. Also please do not give out your solution and do not place a solution to this assignment on a public web site or forum. If you need help, please seek out our available resources to help you.

*Copyright © Stanford University and Marty Stepp. Licensed under Creative Commons Attribution 2.5 License. All rights reserved.*