CSE186 Winter 2025

Assignment 8

In this assignment you create Diligent, a Slack style messaging system, as a Single Page Full Stack Web App using the NERP Stack: Node.js, Express, React, and PostgreSQL, plus Material-UI. The specification of Diligent is large and open ended, you are not expected to complete everything. You are, however, expected to gain an understanding of how sophisticated a well-tested Full Stack Web App can be developed in limited time.

This assignment is worth 25% of your final grade.

Late submissions will not be graded.

Installation

See instructions in Assignment 1 and ensure you are running the LTS version of Node.js. You will also need to have downloaded and installed Docker Desktop: https://www.docker.com/products/docker-desktop.

Setup

Download the starter code archive from Canvas and expand into an empty folder. I recommend, if you have not already done so, creating a folder for the class and individual folders beneath that for each assignment.

Do not modify any configuration files in the starter code or any source file that contains a comment instructing you not to do so.

Note only the contents of `e2e/test`, `backend/src`, `backend/api`, `backend/sql` and `frontend/src` are included in your submission.

To setup the development environment, ***navigate to the folder where you extracted the starter code*** and run the following command:

```
$ npm install
```

This will take some time to execute as `npm` downloads and installs all the packages required to build the assignment.

To start the database Docker container, in the `backend` folder run:

```
$ docker compose up -d
```

he first time this runs it will take a while to download and install the backend Docker PostgreSQL image and start the database service for your server to run against.

To start the frontend and backend dev servers, run the following command:

```
$ npm start
```

The frontend and backend servers can be run individually from their respective folders by running `npm start` in separate console / terminal windows.

To run the linter against your API or UI code, run the following command in the `backend` or `frontend` folder:

```
$ npm run lint
```

To fix some linter errors, run the following command in the `backend` or `frontend` folder:

```
$ npm run lint -- --fix
```


To run API tests run the following command in the `backend` folder:

```
$ npm test
```

To run UI tests run the following command in the `frontend` folder:

```
$ npm test
```

To run end-to-end tests run the following command in the `e2e` folder:

```
$ npm test
```

To stop the database, run the following command in the `backend` folder

```
$ docker compose down
```

Resetting Docker

If you run into problems with your dev database, or simply want to re-set it to its initial state, issue the following commands to shut docker down:

```
$ docker stop $(docker ps -aq)
$ docker rm $(docker ps -aq)
```

Then restart the development database:

```
$ docker compose up -d
```

Web App Specification

This system must be a Single Page Web Application using the following technologies.

Browser:	React & Material-UI
Server:	Node.js, Express & OpenAPI
Storage Tier:	PostgreSQL

You can only use packages provided by the starter code. If you run `npm install` to in any directory containing a `package.json` to use a 3rd party package, parts of your submission will fail to build in the grading system and you will receive reduced marks for the assignment. In extreme cases you may receive no marks at all.

The server must present an OpenAPI constrained authenticated API to the SPA running in the browser and the server must store information other than its secrets in a PostgreSQL database.

Do NOT have the user interface simply download the contents of the database at startup and run all subsequent operations from an in-browser-memory cache. Submissions taking this approach will be *severely* marked down.

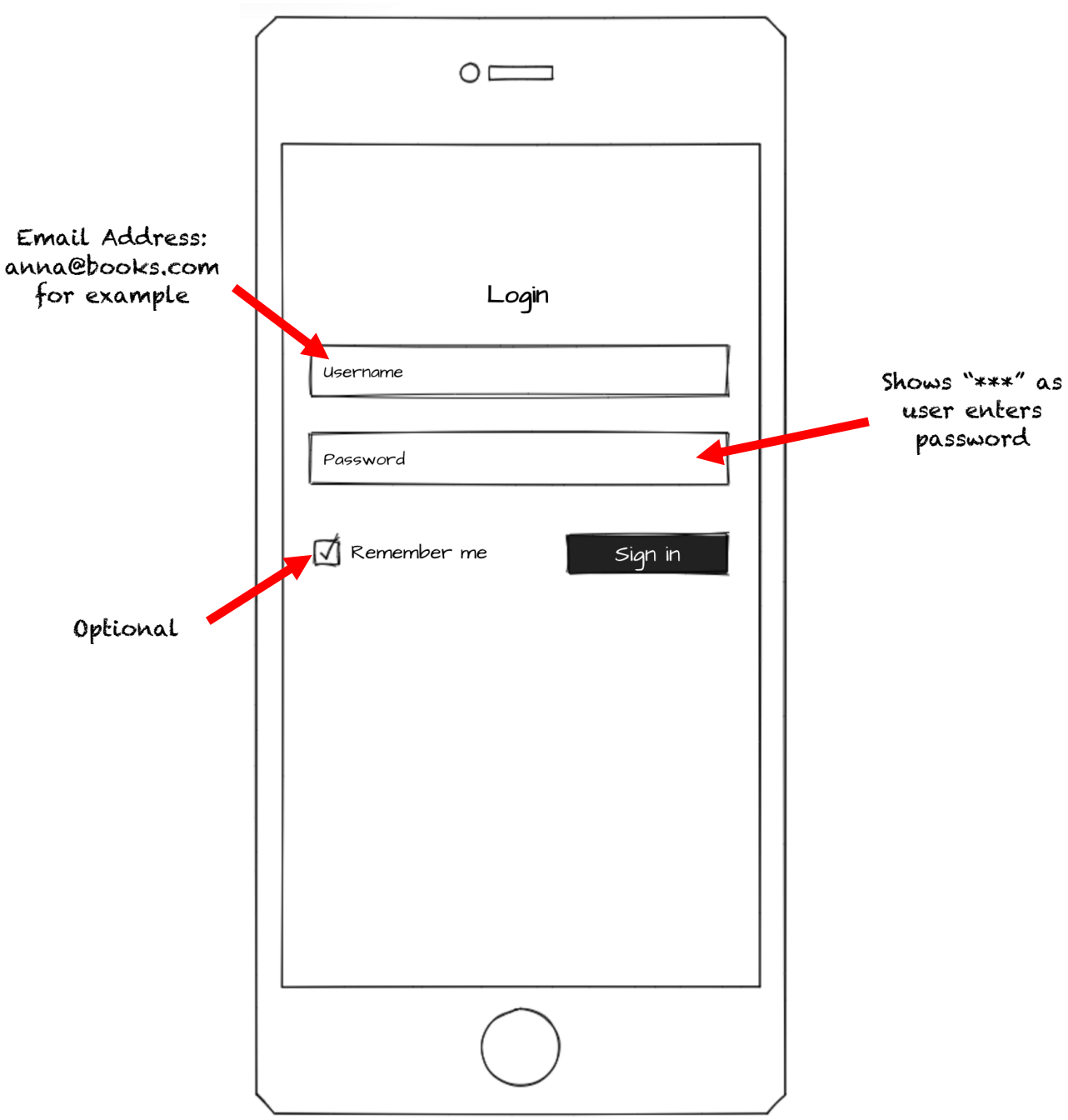
A “mobile first” approach should be taken; some principal operations being:

- Login / Authentication
- Workspace Selection
- Channel Selection
- Message Viewer
- Message Composer
- Settings

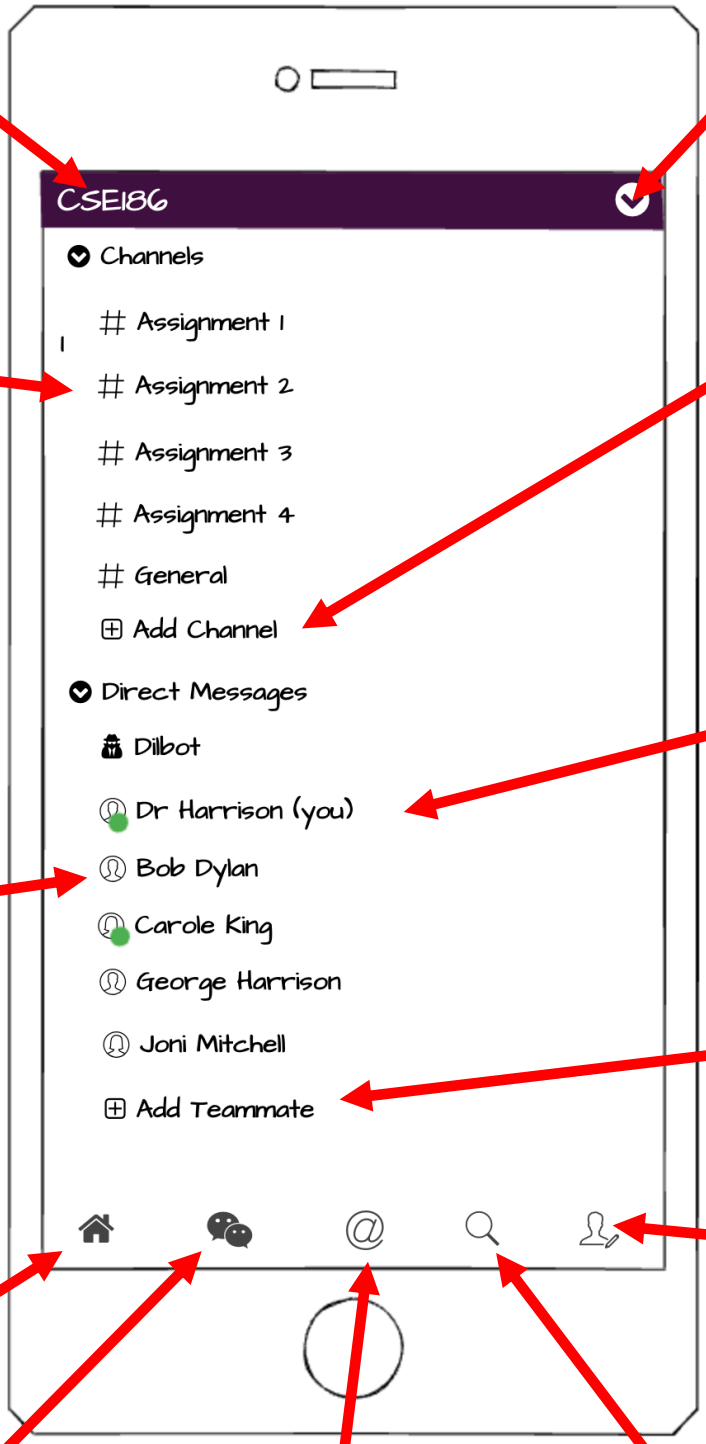
Each is examined in detail on the following pages.

Login

The concept of a user is important when designing Web Applications. Here, the logged in user should only be able to see workspaces they are a member of and their own direct messages. Use The Authenticated Books Example as a guide.



Mobile: Channel & Direct Message Selections (The 'Home" Page)



Currently Selected Workspace

Shows Drop-down-list of owned workspaces and those logged in user is a member of

Selecting a channel shows the message viewer for that channel

Shows dialog/screen to create new channel in current workspace if the logged in user owns it

A 'green dot' on a user shows that user is logged in — unless they set their status to "away"

Selecting user shows the message viewer for direct message from that user to the logged in user

Shows dialog/screen to add members to current workspace if the logged in user owns it

Shows the user profile screen

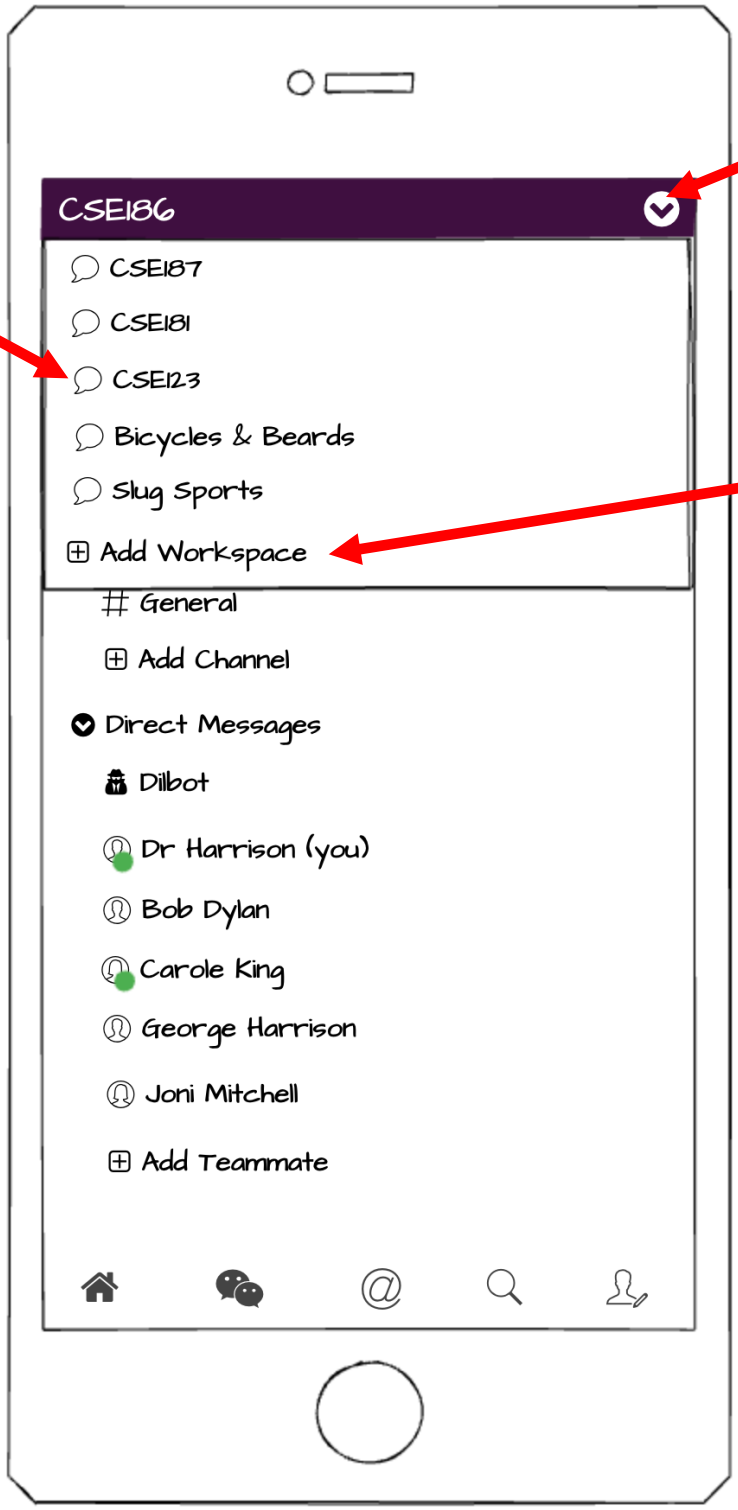
Brings user back to this screen

Shows all direct messages for this user

Shows all messages where the current user was mentioned with @user-name

Pops up an entry field to enter a search term to filter the message list

Mobile: Workspace Selection / Creation

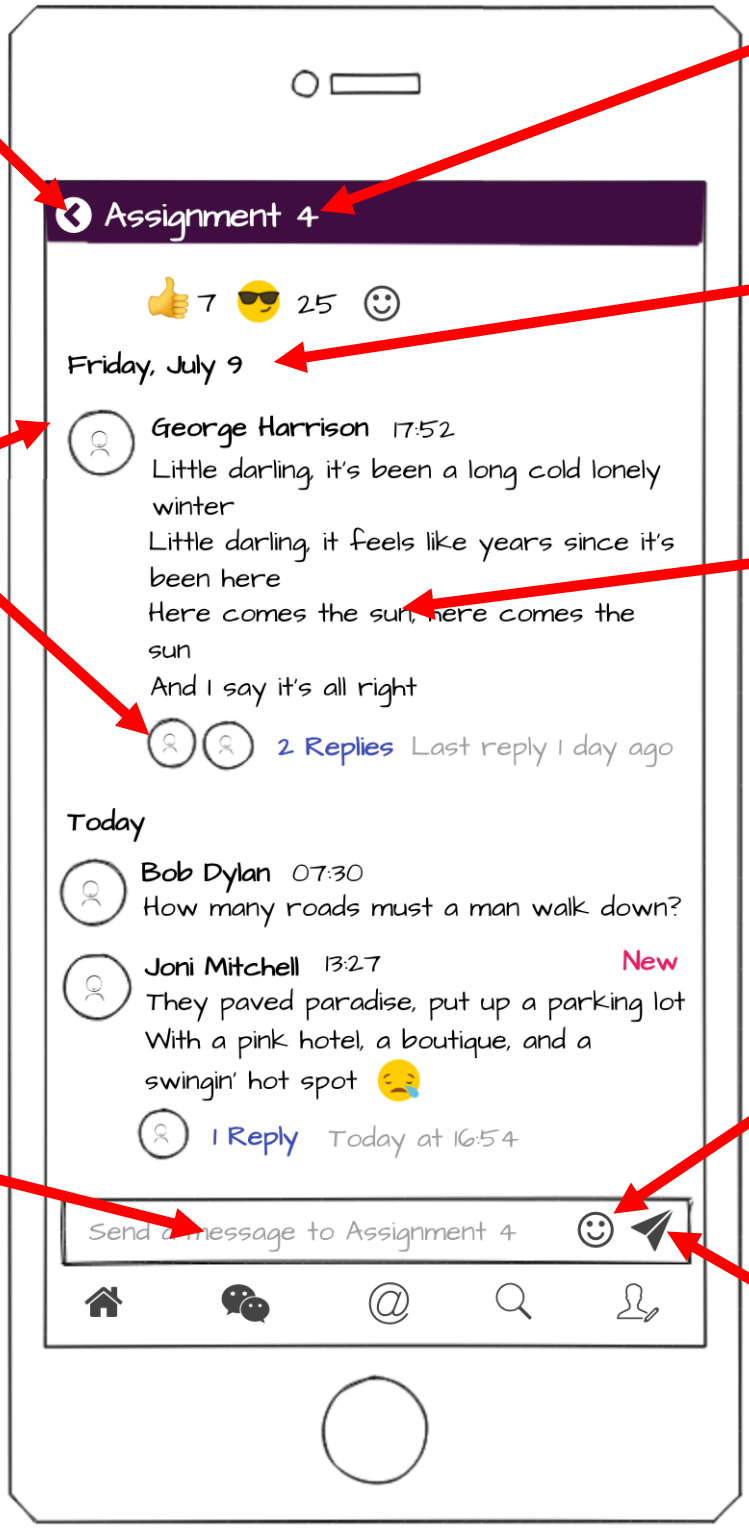


Selecting a different workspace closes drop-down list and the main view refreshes to show channels for the newly selected workspace

Shows Drop-down-list of owned workspaces and those logged in user is a member of

Shows dialog to create new workspace and, perhaps, add members to it

Mobile: Message Viewer



Returns to Home Page

Selection Indicator: Shows which channel or Direct messages are being shown

Messages Grouped by date, most recent at the bottom

Avatars of Users

Clicking anywhere on a message takes you to the thread viewer for that message

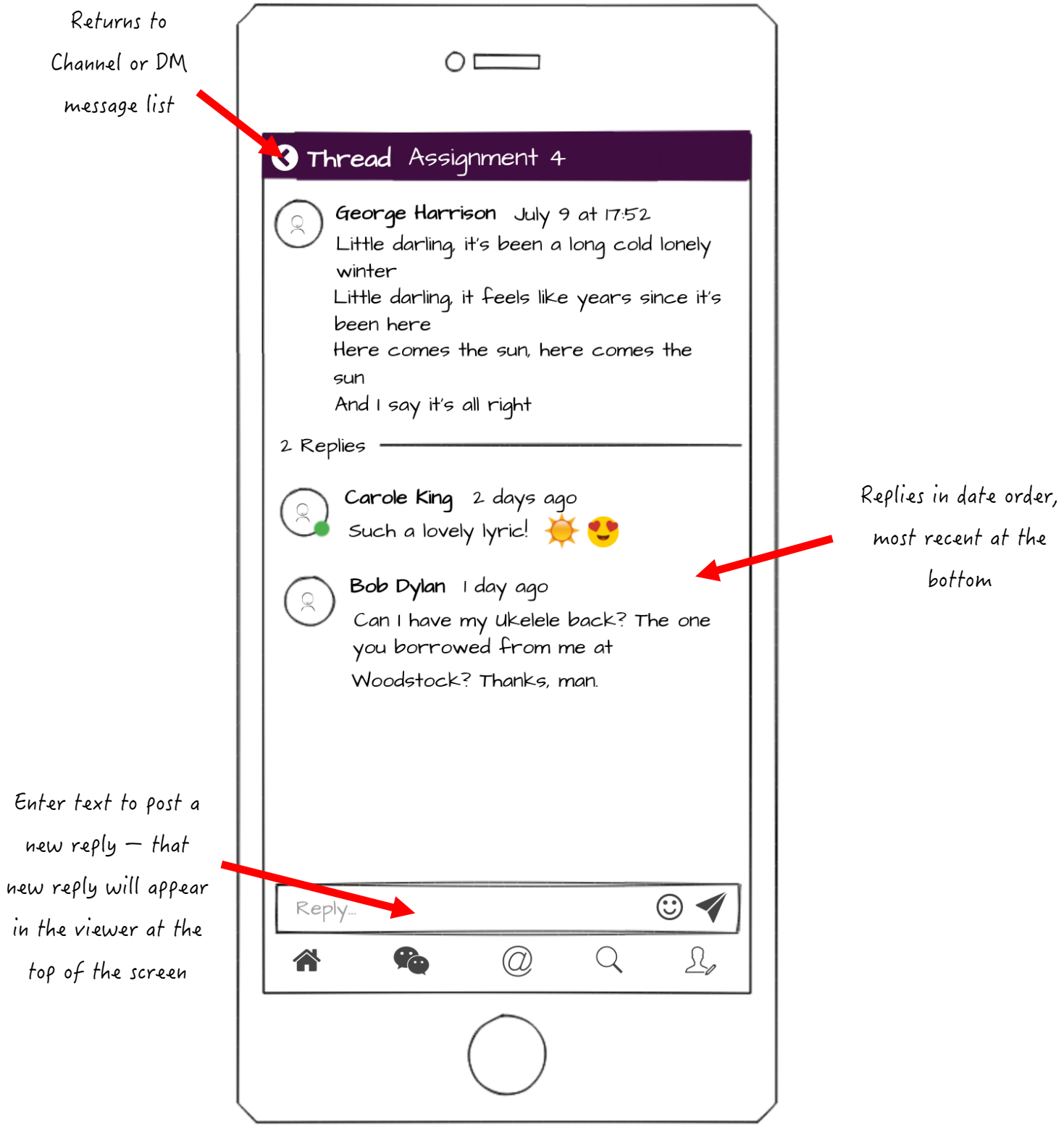
Enter text to post a new message in the channel or to the DM correspondent

Insert an Emoji

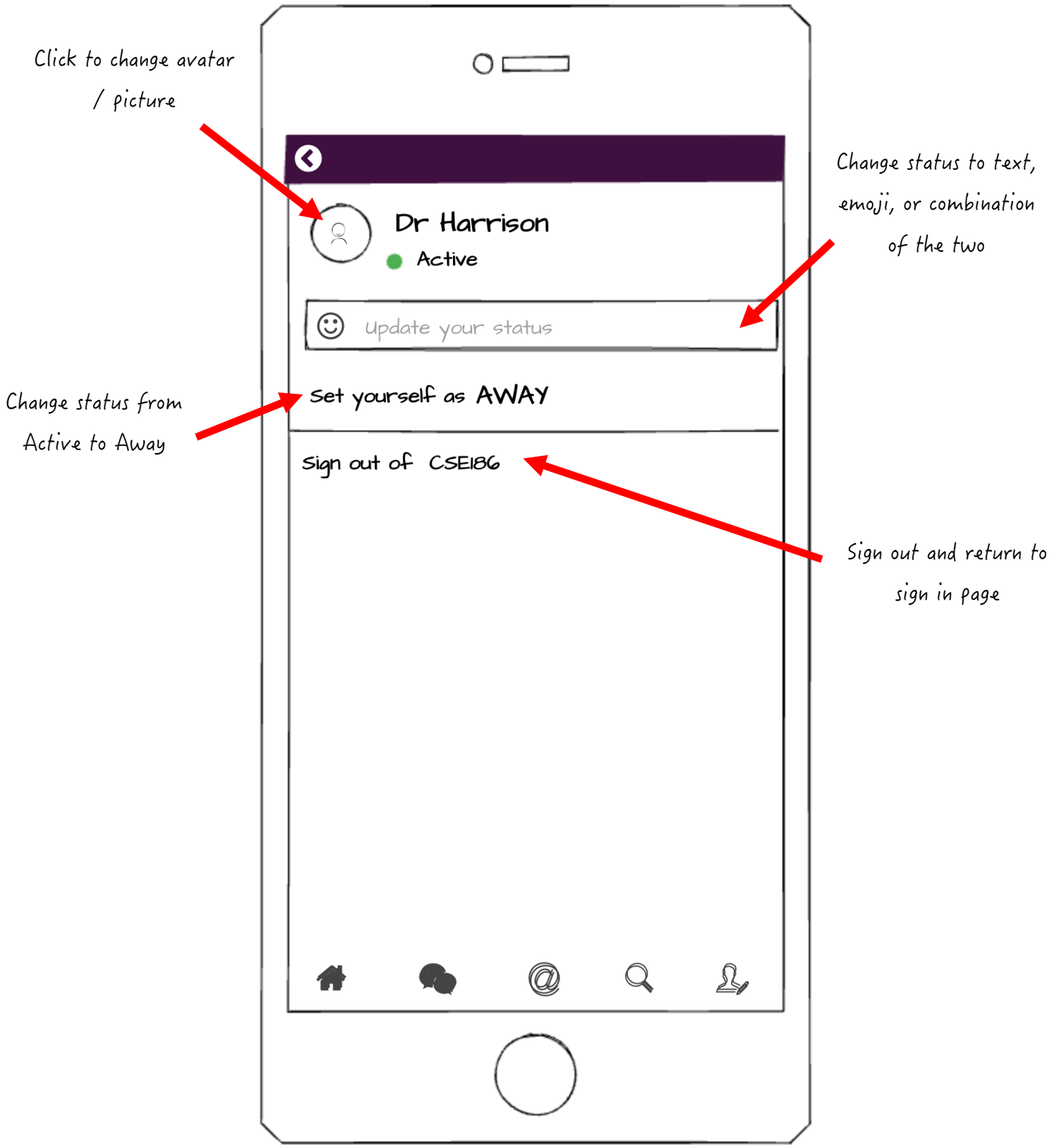
Click to send

Bottom buttons work the same way everywhere!

Mobile: Thread Viewer



Mobile: Settings

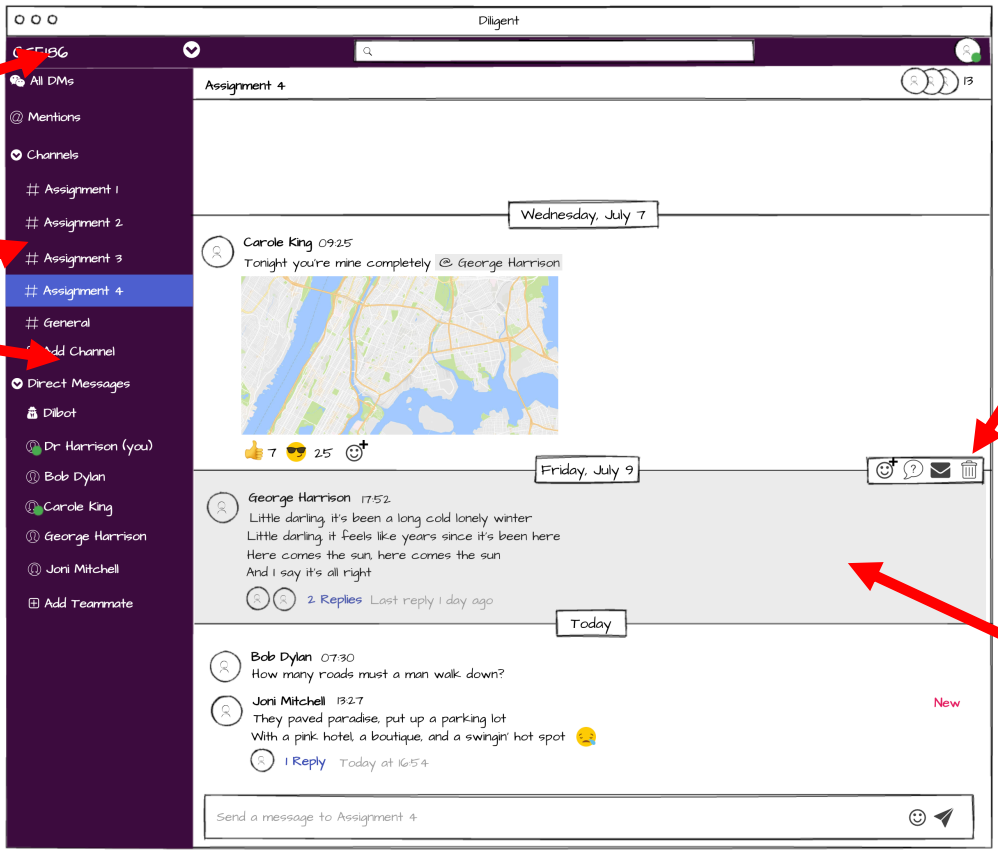


Desktop: Wide View

Select Workspace

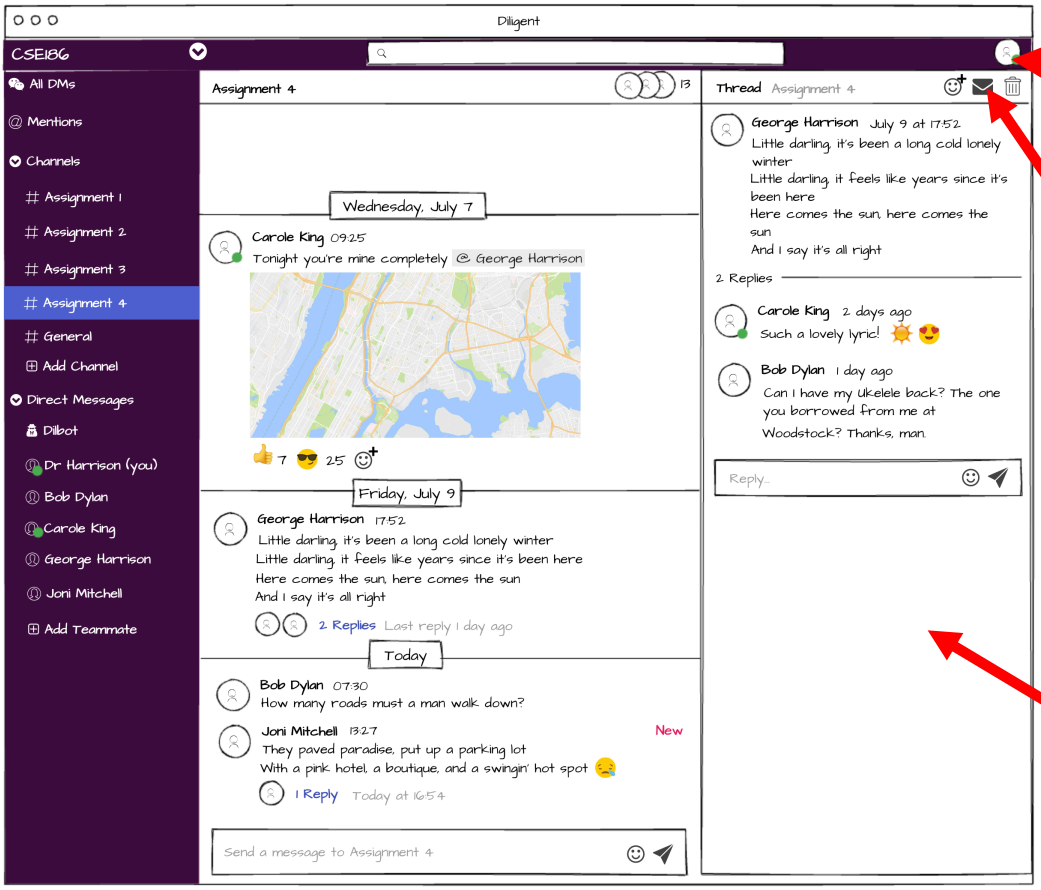
Select:

* All DMS
* Mentions
* User for DMs



Show "response / reply / mark unread / delete" meu when mouse hovers over message

Highlight message when mouse hovers over it



Show settings as a drop down

"response / mark unread / delete" menu

Show reply pane when message clicked

Desktop: Narrow View & Navigation Popup

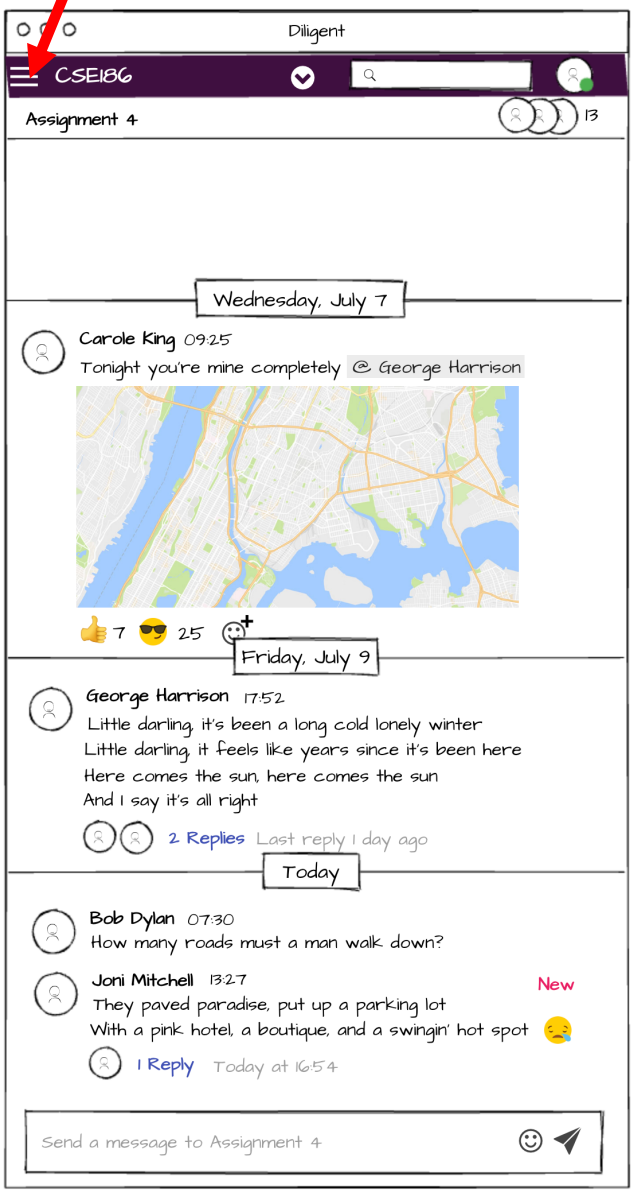
When very narrow, starts to work like mobile version

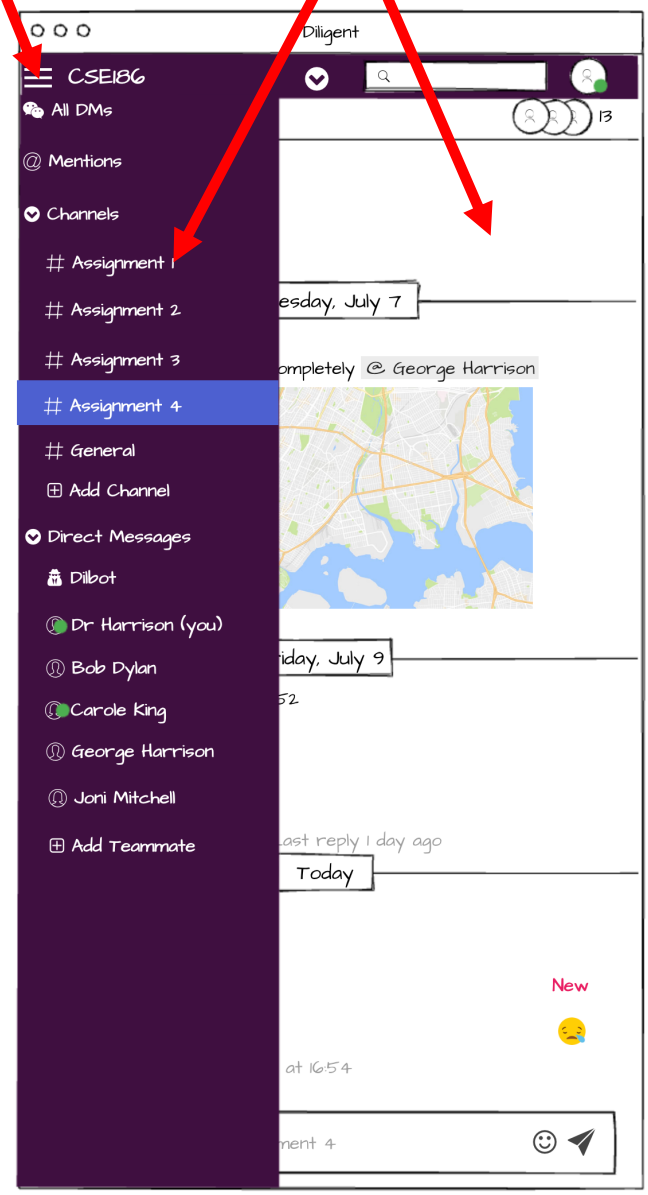
When just a little narrow...

Selection pane is hidden and menu appears

Clicking menu shows selection pane

Clicking on message list or selecting a channel/user hides selection pane





Development Guidelines

Whilst you can construct the user interface any way you like, this finished front end must be a Single Page, Material UI, React Application; React Routes may prove useful. Strong recommendation is to take a “Mobile First” approach.

Background

Listings in this system can have any properties you deem necessary to complete the assignment.

Note that your API must be constrained by an OpenAPI version 3.0.3 schema. You should endeavor to make this schema as restrictive as possible.

For example, your schema should specify the acceptable format of any message property. If a message property in any other form is presented, your system should reject it.

Database Schema

Write your database schema in `backend/sql/schema.sql`. No starter tables are provided, you should design the schema as you see fit. Add startup data to `backend/sql/data.sql`

What steps should I take to tackle this?

You should base your implementation on your Assignment 7 solution plus the Authenticated Book and tested Login Examples.

There is a huge amount of information available on OpenAPI 3.0.3 at https://swagger.io/specification/ and http://spec.openapis.org/oas/v3.0.3. Also consult the Petstore example at https://petstore3.swagger.io/ and make good use of the on-line schema validator at https://editor.swagger.io/.

A good resource for querying JSON stored in PostgreSQL is: https://www.postgresqltutorial.com/postgresql-json/

Many Material-UI Examples and sandboxes for experimentation are available: https://mui.com/

A plausible development schedule

There are any number of ways of approaching this task, but the first few vertical slices through the full stack that need consideration, in the order they need considering, might be:

- Authentication
  - Browser
    - Login “screen”
    - There’s a good one you can borrow from the MUI “getting started” templates.
  - Server
    - Single, unauthenticated end point to present credentials (email/password) and return a security token (JWT)
  - Storage
    - Database table with a primary key and a JSONB column for user data

- Workspaces
  - Browser
    - A drop down-list for workspaces
    - Simple “home page” with a toolbar containing one of the workspaces drop-down lists
  - Server
    - Authenticated GET endpoint for workspaces
    - Only return workspaces the authenticated user is a member of
  - Storage
    - Database table with a primary key and a JSONB column for workspace “objects”
    - Join table to link users to workspaces

- Channels
  - Browser
    - List of channels
    - Navigation bar containing the list of channels
    - Add navigation bar to the home page
  - Server
    - Authenticated GET endpoint for channels in a workspace
    - Reject requests from users not a member of the workspace
  - Storage
    - Database table with a primary key and a JSONB column for channel “objects”
    - Foreign Key to link channels to workspaces

And so on, the next vertical stripe being for messages and will include GET, then POST and DELETE endpoints, each of which should be developed vertically.

Having said all that, you can develop in any sequence you like but experience has shown developing vertically is far more successful than developing horizontally.

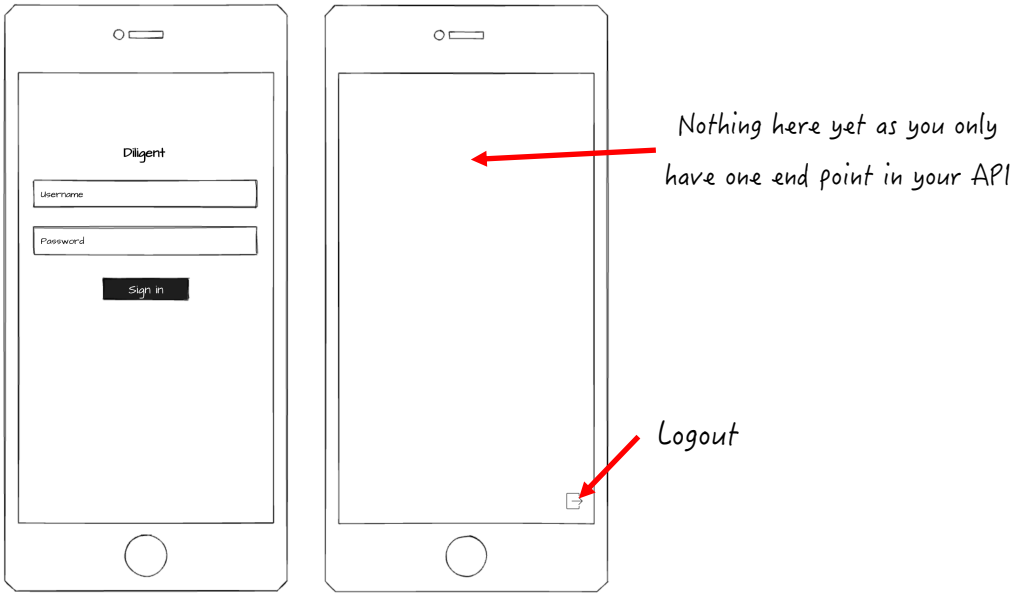
Most importantly, keep on top of your code coverage. A submission that meets all the Fundamental and Basic Requirements and all the Quality Requirements (see below) and has comprehensive end-to-end tests will score 60% so write tests as you go. A very simple implementation with perfect code coverage is likely to score better than a fancy one with poor code coverage.

How much code will you need to write?

This is a difficult question to answer, but not including your OpenAPI Schema, something in the order of 1,000 to 2,000 lines of JavaScript & JSX might be a reasonable estimate for a well-tested basic implementation. For more sophisticated implementations, your code base could be quite large, but if it gets out of control you did something wrong so ask for guidance in office hours.

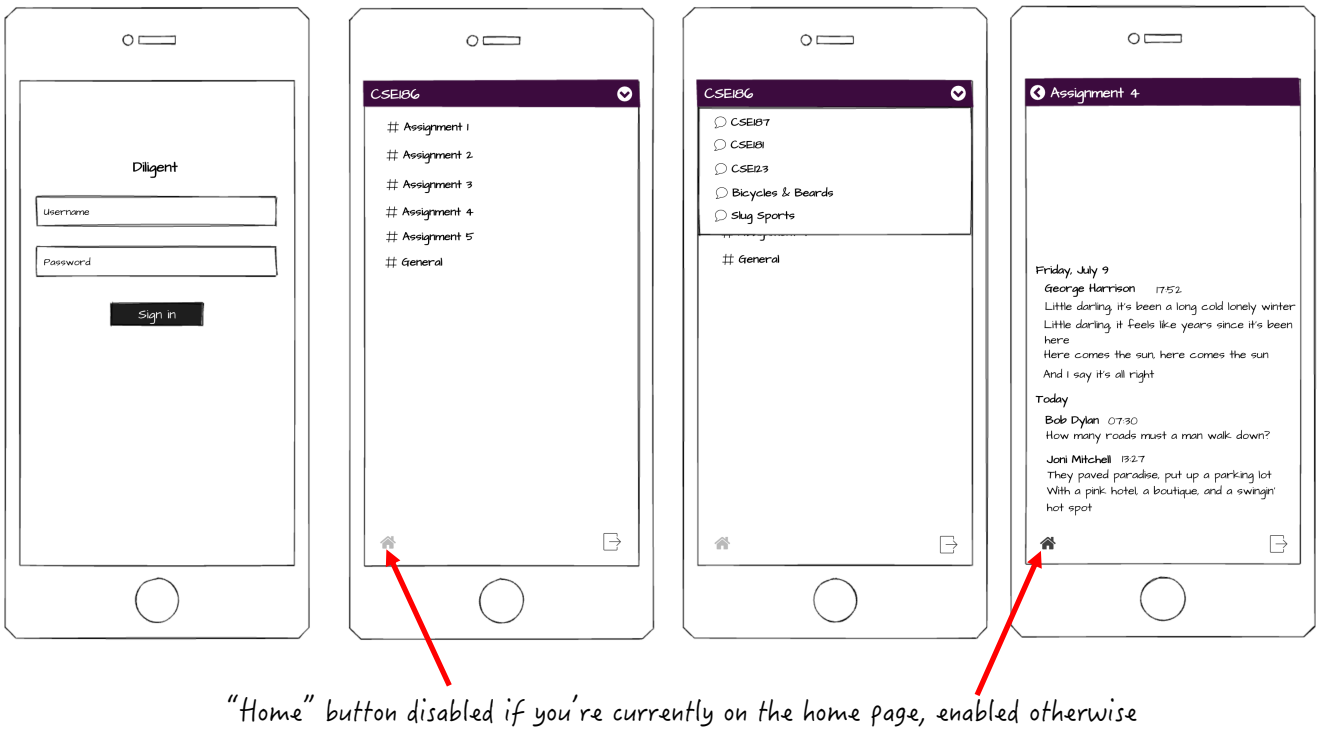
First Milestone

Assuming you've taken the advice above, your first milestone is to have the authentication vertical slice completed, with tests and perfect code coverage on the UI and the Server plus end-to-end tests. Your mobile UI might look something like this:



Basic Requirement

To satisfy the Basic Requirement (see above) you need to have implemented three additional vertical slices (workspaces, channels, messages), maintained excellent code coverage, and added more end-to-end tests. Your mobile UI might look something like this:



Grading scheme

The following aspects will be assessed:

1. (10%) **Fundamental Requirements** (all must be met for other requirements to be assessed)

    - Data is stored in multiple PostgreSQL tables with only id, foreign key, and JSONB columns
    - Server exposes an authenticated OpenAPI restricted RESTful interface
    - UI is written in React and Material UI

2. (20%) **Basic Requirements**

    - Users can log in if and only if they supply valid credentials
    - Once logged in users can see a list of the workspaces they own and those they are a member of
    - Selecting a workspace shows a list of the channels in it
    - Selecting a channel shows a list of the messages in it

2. (40%) **Advanced Requirement**

    - Server remembers where each user was when they were last logged in (5%)
        - i.e. which workspace/channel/message was selected
        - Do ***not*** store this information in the browser local storage
    - UI is as specified in the wireframes above - see "handwritten" notes for details (35%)
        - The more features you implement, the more marks will be awarded
        - But remember, do not implement features at the expense of code quality

3. (10%) **Stretch Requirement**

    - End-to-End tests exist for *all* implemented functionality.

4. (20%) **Quality Requirements**

    - No linter warnings/errors/suppressions 5%
    - Mean of Statement, Branch, Function, and Line Code Coverage across frontend and backend
        - 100% 15%
        - >= 99% 10%
        - >= 98% 5%
        - < 98% 0%

5. (-100%) **Did you give credit where credit is due?**

    a. Your submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%). You will also be subject to the university academic misconduct procedure as stated in the class academic integrity policy.

    b. Your submission is determined to be a copy of a past or present student's submission (-100%). You will also be subject to the university academic misconduct procedure as stated in the class academic integrity policy.

    c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

        - < 35% copied code No deduction
        - 35% to 50% copied code (-50%)
        - > 50% copied code (-100%)

        Note that code distributed in class does not count against the copied total but must be cited.

Additional Requirement

You are also required to make a short YouTube video of your Web App demonstrating the features you have successfully implemented.

- Create it under your UCSC CruzID Account
- Make it “unlisted” - only those with the link can see it
- No more than three minutes
- Demonstrate all the features you successfully implemented
- Include a bulleted list of the features implemented in the video description
- Screen capture from your laptop is fine
- Keep it simple - no marks for fancy effects and/or editing

Mandatory User Accounts

You can have as many users as you like in your submission, but you must have these two and they must have at least three workspaces with at least four channels in each:

molly@books.com / mollymember
anna@books.com / annaadmin

You can use their password hashes from The Authenticated Book Example if you like.

Do NOT store plain text passwords in the database!

What to submit

Run the following command to create the submission archive:

```
$ npm run zip
```

You must also paste a link to your YouTube video as a comment on the canvas submission.

***** UPLOAD** `CSE186.Assignment8.Submission.zip` **TO THE CANVAS ASSIGNMENT AND SUBMIT *****

