Project 2, handin 1

It's getting complicated!

FULL PROJECT DESCRIPTION

First off, congrats on making it past project 1!

Now that we have the foundation underneath us, let's build a more complex web app.

In project 2, you will build a real-time chat app in the browser, like Google Hangouts, Facebook Messenger, or our very own Slack. To distill these complex apps down to the basics, your chat app will only feature one public chat room where everyone can discuss things amongst one another.

However, to keep things interesting and unique for all of you, you will also build a very simple bot to accompany your app. The bot must passively listen for and respond to messages in the public room via commands (discussed below). Give the bot some flair and personality!

Lastly, because you'll be working on the same codebase for four weeks, you will also be writing tests and setting up automated testing and continuous integration infrastructure to help both yourself, and us, verify that everything is working as expected.

HANDIN DESCRIPTIONS

This project is huge, no doubt about it. But I know you can do it!

Like we discussed in the very first lecture, all complex software evolves from simplicity. Project 1 is a simple, yet solid, foundation for our web app, and it serves as a great starting point for project 2: hopefully, building and deploying a one-page web app should be a cakewalk!

To help you make the leap from project 1 to project 2, we have broken project 2 down into 2 two-week milestones, each of which you will hand in separately. Handin 1 will lay the foundation for the full project, from which handin 2 can be built. You are very strongly encouraged to expand upon your handin 1 code for handin 2. They break down as follows:

Handin 1 (what's due now)

Handin 1 will require you to set up the basic layout of your chat app, as well as build the public room that everyone can chat in. You will also start the foundation of your bot by making it listen for and respond to simple messages.

Handin 2 (what's up next)

Handin 2 will require you to expand on top of your foundation, by adding in link and image attachments. Your bot will respond to more types of messages, incorporating an external API.

HANDIN 1 INFORMATION

When

This handin must be submitted before **Tuesday, February 21, 2017 at 11:55pm**.

How

You can hand in this project by filling out this Google Form:
https://goo.gl/forms/YR8zMUxY5eZ0YqN82

PURPOSE

Build something with client-server architecture

The primary goal of this handin is to get you to move from web pages towards web apps by thinking of web development in terms of client-server architecture. Project 1 was a great start to web programming, but it was severely constrained since our clients weren't doing anything. In modern, complex applications, both the client and servers need to do work and communicate with one another. This handin has you set up client-side views using React and communication using Socket.io. Although the details of client-server architecture vary by platform, the general gist of it looks the same no matter where you go.

Get comfortable with how complexity arises

The secondary goal of this handin is to allow you to see how complexity can arise in a large application. The three ways in which we'll introduce complexity are via client-server networking, data persistence through databases, and authentication. These, in fact, are the three most common ways in which complexity is added to a software system. This handin will have you implement all three to evolve a skeleton into a more full, complex app.

WHAT YOU WILL NEED TO DO

Reuse your existing Cloud9 workspace

Move your project 1 work to a folder, and make a new folder for project 2. You shouldn't have to create a new workspace. If you were working in `~/workspace` for project 1, you can rename the `workspace/` directory to `project1/` and create another directory named `workspace`.

If you run out of disk space, you can always back your work up to Github and delete the folders locally on your workspace. The program `ncdu` will let you find ways of freeing space.

Track your work in Git & save your work via Github

You should be constantly saving your work in Git and constantly backing it up to Github. Set up your Github repository as follows:

- Owner is the course organization (https://github.com/orgs/CSUMB-SP17-CST438/dashboard)
- Name is `project2-h1-<your-email>`, where `<your-email>` should be replaced with your email (potato@csumb.edu should be `potato`, for instance)
    - Put together, if your email is potato@csumb.edu, your repositories should look like `project2-h1-potato`, NOT just `potato` by itself, or `project2-h1-potato-csumb.edu`.
- Repository is private (as opposed to public)

For help using Git or Github, please check out the the slides and links from Lecture 3.

Deploy your app using Heroku

You should be continuously integrating your code into your live site on Heroku.

Once again, you're encouraged to pick a Heroku domain that fits your theme (see below) but you may stick with the default if you can't come up with something.

For help using Heroku, please check out the slides and links from Lecture 4.

Template and render client-side views using React, JSX, and Javascript

This will be covered in Lecture 5. Follow these guidelines:

- Your public chat room must be available at the endpoint /
- Your public chatroom must have a background image

    - As always, everything should be legible! Feel free to use all of the tricks you learned from project 1
- Ensure the following are always visible:
    - The chat log of the room (messages sent and received)
    - All connected users in a list
    - Current number of connected users
    - An input field to type messages in
- Your page itself should not scroll; the chat log should scroll if it gets too long
- The chat log of the room should occupy at least 50% of the width of the screen
- All messages from other users must look consistent
    - You can easily do this by styling with the same React component
    - Username, nickname, or real name of the sender must be next to the message
    - Profile picture of the sender must be next to their name

Here are some additional resources:

- Javascript basics: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics
- repl.it - Javascript: https://repl.it/languages/javascript
- Tutorial: Intro to React: https://facebook.github.io/react/tutorial/tutorial.html
- Hello World - React: https://facebook.github.io/react/docs/hello-world.html

Build your chat bot

Your chat bot is just another fake user in the chat room. It'll be a simple way of testing your send/receive code paths, as well as a way to give your app some flair!

The simplest way to hook a bot into your chat app is to have your server scan the text of every message it gets. When it finds something to respond to, have your server send out another message!

- Your bot must be clearly identifiable in some way; here are some examples:
    - Its username, nickname, or real name can end with "`bot`"
    - Its profile picture can be clearly identifiable as a bot
    - Its messages can appear in a different styling to real users' messages
- Your bot must message the public room when people join or leave
- Your bot must respond to all messages starting with `!!` (we'll call these commands):
    - `!! about` should make your bot message the room with a brief description
    - `!! help` should make your bot message the room with all commands it recognizes

    - `!! say <something>` should make your bot message <something> to the room
    - In addition to the above, you must implement at least two other commands
    - If a command is unrecognized, your bot must message the channel saying the command is not recognized

Be creative! Your bot is how you'll add pizazz to your project. By the end of handin 2, your bot's name, picture, personality in messages, the background of your page, its design, and its layout should constitute a consistent theme. For handin 1, since you'll be wiring most things together, you won't be doing much customization, but you'll be required to do this for handin 2!!

Connect client and server using Socket.io

This will be covered in Lecture 6. Follow these guidelines:

- When someone sends a chat to the public chat room, your server should emit an event to send the message down to clients
- When someone visits your app, your bot should let all clients know about the user that just connected
    - Clients should update their active user list and count to include the new user
- When someone closes your app, your bot should let all clients know about the user that has just left
    - Clients should update their user list and count to exclude the person who left

You should be able to test multiple clients by using multiple browser tabs, incognito windows, or different browsers.

Here are some additional resources:

- Flask-SocketIO's documentation: https://flask-socketio.readthedocs.io/en/latest/
- ReactJS and SocketIO chat application: http://danialk.github.io/blog/2013/06/16/reactjs-and-socket-dot-io-chat-application/

Persist data via a database

I will not tell you how to design your relational format, but I strongly encourage you to use an ORM like SQLAlchemy to save you the hassle of writing SQL yourself. We will cover Flask with SQLAlchemy in Lecture 7. Follow these guidelines:

- Your database should be PostgreSQL, **NOT** SQLite
    - Postgres should be installed; try `psql` and see the guide below for setup
- The chat log at `/` should load with chats in recent history

    - In other words, if I send a message, and refresh the page, the message I sent should still be visible on load
- The chat log at / should not clear if your web server restarts
    - In other words, your web server should not hold onto the messages; they should be saved out to your DB and read from your DB

Here are some additional resources:

- Setting up PostgreSQL on Cloud9: https://community.c9.io/t/setting-up-postgresql/1573
- Heroku Postgres: https://devcenter.heroku.com/articles/heroku-postgresql
- Flask and PostgreSQL on Heroku: http://blog.y3xz.com/blog/2012/08/16/flask-and-postgresql-on-heroku
- Flask-SQLAlchemy documentation: http://flask-sqlalchemy.pocoo.org/2.1/

Authenticate users

All users will need to be authenticated in some way! You must implement both Google and Facebook sign-ins, and allow users to choose between the two. We will go over the basics of how to do both in Lecture 8. Follow these guidelines:

- Users should not be able to send messages until they're authenticated
- You should distinguish whether users are logged in from Facebook or Google in the UI
- To simplify things, it does not matter whether users get logged out when you close or refresh the page
- You should pull nickname, email, or real name, in addition to profile picture, from the login.
- You should not count as a connected user until you authenticate

Additional resources:

- Google Sign-In for Websites: https://developers.google.com/identity/sign-in/web/
- Facebook Login Button: https://developers.facebook.com/docs/facebook-login/web/login-button

Document your work

Your code submission must include a file named `README.md` at the top level. The file must be in Markdown (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet), and must answer the following questions:

- What is the theme you'll be using for project 2?
- How did you incorporate your theme within your project?

- What are known problems, if any, with your project?
- How would you improve it if you had more time?

HOW YOU WILL BE EVALUATED

As outlined in the course syllabus, this project will be worth 75 points, or 15% of your final grade. Those 75 points break down as follows:

- 2 points: setup
    - 1 point: Github repository is named by your email (`project2-h1-potato`)
    - 1 point: Github repository is named as described above
- 10 points: Heroku deployment
    - 10 points: web app is deployed via Heroku
- 18 points: client-side views and app layout
    - 1 point: public chat room must be available at the endpoint `/`
    - 2 points: public chat room has a background image
    - 1 point: all public chat room text and images are legible
    - 3 points: chat log of the room is always visible (messages sent and received)
    - 2 points: all connected users in a list is always visible
    - 1 point: current number of connected users is always visible
    - 2 point: input field is always visible
    - 1 point: page itself should not scroll
    - 1 point: the chat log of the room should be at least 50% of the width of the screen
    - 2 points: all messages from other users must look consistent
    - 1 point: username, nickname, or real name of the sender must be next to message
    - 1 point: profile picture of the sender must be next to their name
- 10 points: chatbot
    - 2 points: chatbot is clearly identifiable
    - 2 points: bot messages room when people connect or leave
    - 1 point: `!! about` makes the bot message the room with a description
    - 1 point: `!! help` makes the bot message the room with a list of all commands
    - 1 point: `!! say <something>` makes the bot say `<something>` to the room
    - 1 point: 1 other command
    - 1 point: another command
    - 1 point: bot acknowledges unrecognized commands
- 10 points: client-server connection
    - 2 points: all clients are able to send message to web server
    - 4 points: all messages received on server are relayed to all clients
    - 1 point: all clients show new user on connect

    - 1 point: all clients update count on connect
    - 1 point: all clients remove user on disconnect
    - 1 point: all clients update count on disconnect
- 10 points: offline persistence
    - 2 points: database is not SQLite
    - 5 points: chat log at / loads with recent or all history
    - 3 points: messages are persisted via database
- 10 points: third-party authentication
    - 3 points: users can only send messages after authentication
    - 1 point: login with Facebook button is present on page and login flow works
    - 1 point: username, nickname, email, or real name is pulled from FB
    - 1 point: profile picture is pulled from FB
    - 1 point: login with Google button is present on page and login flow works
    - 1 point: username, nickname, email, or real name is pulled from Google
    - 1 point: profile picture is pulled from Google
    - 1 point: user doesn't count as connected user until authenticated
- 5 points: documentation
    - 1 point: `README.md` is in the root of the repository on Github
    - 1 point: Theme is explained in readme file
    - 1 point: How theme is implemented is explained in readme file
    - 1 point: Acknowledgement of known problems, if they exist, in readme file
    - 1 point: Description of how to improve in readme file

COMMON PITFALLS

Javascript code isn't loading

Chances are, your JS code isn't being served in the first place. Look inside the developer console logs of your browser to debug this further.

Javascript code works, partially, but stops running for some reason

Chances are, your JS code has some sort of bug in it. Look inside the developer console logs of your browser to debug this further.

React isn't behaving as expected

You can use the React Developer Tools Chrome extension, which will add a "React" tab to the developer tools in your browser:

https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

I updated Javascript code but it isn't on the site

Your Javascript code is stale. This could be caused by two things:

- Your transpiler could be stopped. Verify that your transpiler (Webpack, Browserify, etc.) is actually running and producing a JS file output.
- Your browser could have cached your client code. Hard refresh (Ctrl/Cmd+Shift+R) to force it to refetch the JS bundle.

Uncaught TypeError: Cannot read property 'something' of null

Remember `NullPointerException` from Java? They're baaack!

In general, this error means you're trying to access a property, or field, of a null object. Here's a simple example of how this can happen: https://repl.it/FcWl/0

In React, this can happen if you call `this.setState` within the constructor -- instead of doing this, just assign it directly instead: http://stackoverflow.com/a/34962381

Each child in an array or iterator should have a unique "key" prop

Your list elements need a key element for performance reasons. See this answer here: http://stackoverflow.com/a/32256534

Clients aren't sending/receiving updates sometimes

The easiest way to debug this is using the developer tools provided in the browser. Most developer tools, Chrome included, have a "Network" tab where you can peek on communications: http://stackoverflow.com/a/5757171

You can debug on the server with good ol' `print`.

Socket.io is spewing lots of 400 bad requests in browser console logs

There are two possible causes:

- The first is that Socket.io isn't wired up on the server. Make sure that your code has `socketio.run`, not `app.run` at the bottom.
- The second is that your Python code is crashing somewhere inside a `@socket.on`. To debug this, dig into your server logs in Cloud9 and look for a Python stack trace.

Webpack breaks; I get bash: webpack: command not found

Try running `sudo npm install -g webpack` again.

OperationalError: (psycopg2.OperationalError) could not connect to server: Connection refused

Ensure your local instance of PostgreSQL is running: `sudo service postgresql start`

ADDITIONAL QUESTIONS AND CLARIFICATIONS

If you need help on the assignment, or are stuck on any part of it, don't hesitate to discuss with fellow students on Slack in `#discussion`, or to post questions on the course Facebook group. We will monitor these channels and provide additional help if needed.

If you would like clarification on any part of this document, please contact the instructors directly on Slack, the course Facebook group, or via email. Depending on the nature of your clarification we may update this document.