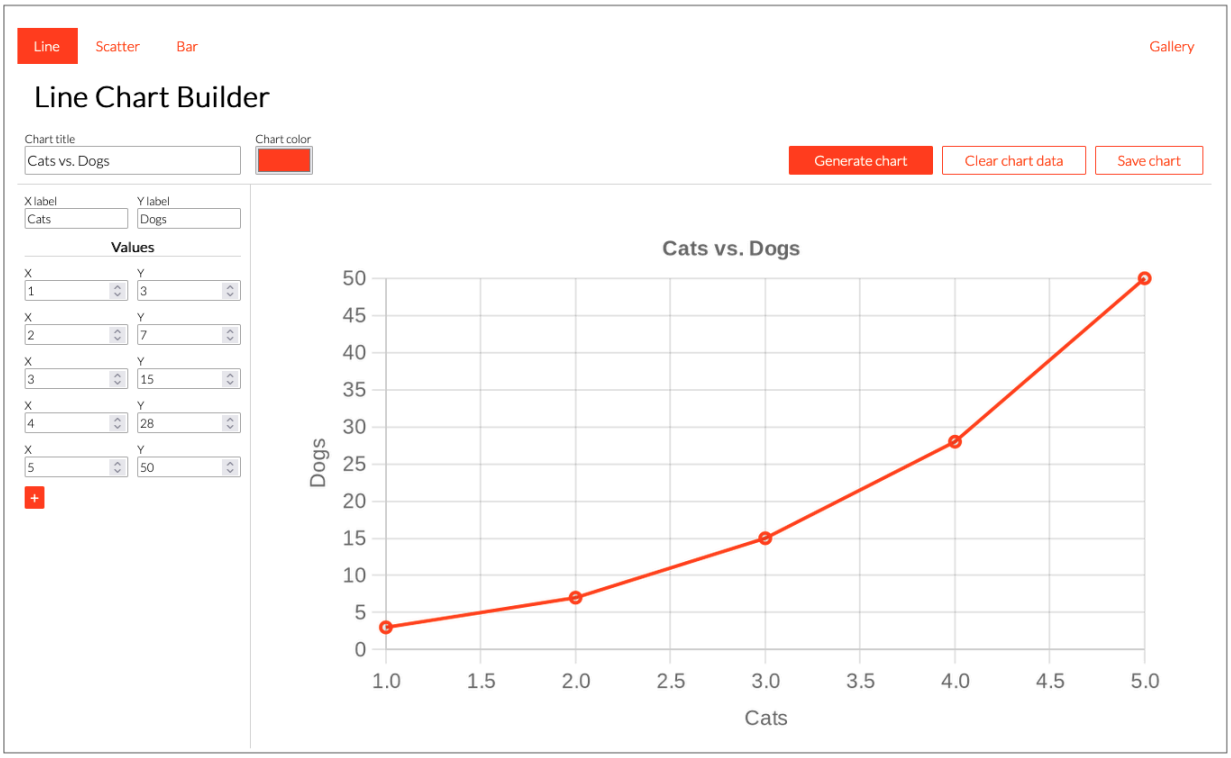
CS 362 Final Project

Code due at the time of your final project grading demo
(grading demos will occur during finals week; more details TBA)

In this course, a final project will take the place of formal exams to test your understanding of the material. For this project, you will work in teams of 3-4 to implement a complete test suite for an existing application and to develop a CI/CD pipeline for that application. The application you will be working with and your goals for the project are outlined below.

Chart building application

The application you'll be working with for the final project is a web application that allows users to build simple charts:



The app specifically allows the user to build three different kinds of charts, line charts, scatter plots, and bar charts. The user enters chart data in a sidebar on the left-hand side of the app, sets chart options such as title and color, and then generates the chart by clicking on the "generate chart" button. The application also includes an option to

save charts to a “gallery”. The charts themselves are generated using a web service called the QuickChart API. Links to videos depicting specific flows within the application are included below.

The code for this application will be included in the GitHub repo that will be created for you for the project. More details about the app’s implementation are included in the README of that repo.

Final project GitHub repos

You can create a GitHub repository for your team containing the application code for the project using the following GitHub Classroom link:

https://classroom.github.com/a/dINa9FAJ

All members of your team should follow that link to join the repo after it’s created. There are a few things to note about the final project repos:

- GitHub Codespaces is configured for the repo, so you will be able to start a Codespace to work on the project if you’d like. This will work the same way as it did for the assignments in this course.
- You will have full administrative permissions on the repository. This will give you control over some of the repo’s settings that will be useful for this project. See below for more details. You’ll also be able to make the repo public if you want to.

Final project goals

You will have several goals for this project:

- Implement unit tests for relevant parts of the application.
- Implement integration tests for relevant parts of the application’s UI.
- Implement end-to-end tests for important flows within the application.
- Implement a CI/CD pipeline for the application.
- Perform code review on all work for the application.

Each of these goals is described in more detail below.

Unit tests

In the project directory `src/lib/`, there are a few files of library code you should write unit tests for. Below is some information about each of those files and the unit tests you should write for it:

- **`sortPoints.js`** – This file contains a single function `sortPoints()` for sorting an array of data points input by the user. Read the code documentation for this function to better understand how it works. You should write one or more unit tests to verify that this function correctly sorts a valid array of points. Do not worry about testing error cases for this function, since it is not set up to perform error handling.

- **`chartStorage.js`** – This file contains five functions that store and load chart data for the app. The functions `saveChart()`, `loadAllSavedCharts()`, and `loadSavedChart()` all operate on the set of charts saved in the app's "gallery" page. The functions `updateCurrentChartData()` and `loadCurrentChartData()` are used to cache the data associated with a chart under construction, so the user can switch between chart types (line, scatter, and bar) and have the data they entered maintained across pages. Read the code documentation for these functions to better understand how they work.

  You should write unit tests to verify that these functions work correctly. Note that these functions all rely on `localStorage` for storing chart data. **This means you will need to test these functions in a DOM-based environment.** You should be able to do this by using the JSDOM Jest testing environment for the tests you write for these functions (you won't need to actually render anything to test these functions).

- **`generateChartImg.js`** – This file contains a single function `generateChartImg()` that generates a chart image from chart data. Read the code documentation for this function to better understand the parameters it takes and how the function works. You should write one or more unit tests to verify that this function works correctly.

  Note that `generateChartImg()` uses the QuickChart API to generate the chart image. Ordinarily, you would use a fake (e.g., via MSW) in your tests when working with APIs and other external services to avoid flakiness. However, this is not a requirement for this project. You do **not** have to use MSW in this project.

Any tests that connect with the API will simply be flaky. You will not be penalized for this.

Unit test requirements

Your unit tests should satisfy the following requirements:

- They should run in Jest.
- They should pass.
- They should only test public functions (i.e. any functions exported through `module.exports`) in `src/lib/`.
- They should adequately cover valid inputs to the functions being tested.
- They **DO NOT** need to cover error cases.
- An application build process is described in the README included with the code for the project. **You do not need to build the app to perform unit tests.** You can perform unit tests directly on files in `src/lib/`.

UI integration tests

Many of the UI's features are too complex to test with integration tests and will need to be tested with end-to-end tests. However, there are a couple UI features that have appropriate scope for verification through integration tests. Write integration tests for the features described below:

- **Adding values in the chart builder (video)** – Write one or more integration tests to verify that the "add values" button (i.e. the "+" button) in the sidebar of the chart builder behaves correctly. Each time the user clicks this button, it should add a new pair of input fields for the user to enter new X and Y values. Clicking the button should not impact any data the user has already entered.

- **Alerts displayed for missing chart data (video #1, video #2)** – If the user tries to generate a chart without supplying axis labels or without supplying any data, an alert will be displayed. Write integration tests to verify that this happens correctly. The alerts are displayed using the `alert()` method. You can use a spy on this method to help verify that the alerts are correctly displayed.

- **Clearing chart data (video)** – Write one or more integration tests to verify that the "clear chart data" button correctly clears all chart data entered by the user. Clicking this button should clear the chart title, color, X & Y labels, and any X or Y data points the user entered, and it should reset the page to display just one pair

of input fields for entering X and Y values.

- **Data correctly sent to chart generation function (video)** – Write one or more integration tests to verify that the chart data entered by the user is correctly sent to the chart generation function `generateChartImg()`. **You don't need to verify that the chart image is correctly generated.** You also don't need to mock the API for this integration test. Instead, since you just want to verify that data from the user is correctly sent to the chart generation function, you can simply spy on that function while also stubbing it to return any valid image URL (e.g. `http://placekitten.com/480/480`). You can rely on the unit tests you wrote above for `generateChartImg()` and the end-to-end tests you'll write below to verify that the chart image is correctly generated.

Integration test requirements

Your integration tests should satisfy the following requirements:

- They should run in Jest.
- They should pass.
- They should verify the behaviors described above. Note that these behaviors are duplicated across the three chart builder pages (`bar.html`, `line.html`, and `scatter.html`). **You only need to test these behaviors on one of those pages.**
- They should use the DOM Testing Library tools we explored in class to interact with the app the way a user would.
    - If it is simply not possible for you to use a visual characteristic to get a reference to a particular element on the page, you can assign a test ID to that element.
- You will need to render the HTML and JS files for one of the chart building pages (e.g. `src/line/line.html` and `src/line/line.js`) into a testable DOM to perform these integration tests. You can do this in a manner similar to the way we did it in lecture.
- An application build process is described in the README included with the code for the project. **You do not need to build the app to perform integration tests.** You can perform integration tests by directly rendering files from the `src/` directory for one of the chart building pages (e.g. `src/line/line.html` and `src/line/line.js`).

End-to-end tests

There are important larger flows in the app that can only be tested via end-to-end tests. Write full end-to-end tests to verify the features described below:

- **Chart is correctly generated (video)** – Write an end-to-end test to verify that a chart image is correctly generated when the user supplies the needed data for the chart and clicks the “generate chart” button. To verify that this flow behaves correctly, you can simply assert that an image appears in the document after the “generate chart” button is clicked.

- **Chart data is maintained across pages (video)** – When the user enters data on one of the chart building pages, that data should be maintained if they navigate to a different chart building page. For example, the user should be able to enter data in the line chart builder, and that data should be maintained if they then navigate to the scatter plot builder. Write an end-to-end test to verify that this works correctly.

- **Saving a chart to the “gallery” (video)** – After a user generates a chart, they can click the “save chart” button to save that chart to the app’s “gallery”. Write an end-to-end test to verify that this works correctly. You can use the title of the chart to craft an assertion to verify that the chart is correctly saved in the gallery.

- **Re-opening a saved chart (video)** – After a chart is saved in the gallery, a user can click on that chart in the gallery to re-open the chart builder with that chart’s data displayed there along with the generated chart image. Write an end-to-end test to verify that this works correctly.

End-to-end test requirements

Your end-to-end tests should satisfy the following requirements:

- They should run with Cypress.
- They should pass.
- They should verify the behaviors described above.
- They should use the Cypress Testing Library to find elements in the app the way a user would.
    - If it is simply not possible for you to use a visual characteristic to get a reference to a particular element on the page, you can assign a test ID to that element.

- You will need to build and serve the application to be able to run tests against it with Cypress. The README included with the code for the project describes how to do this (TL;DR: you can just run `npm start` to build and serve the app).

CI/CD pipeline

You should use GitHub Actions to build a continuous integration and continuous deployment pipeline for your application. This CI/CD pipeline should do the following things:

- Run all tests when a pull request into the `main` branch is made and when new commits are pushed to the branch associated with this pull request.
    - You should set your project repo up to prevent merging a pull request if tests fail on it.
- Run all tests when commits are pushed to the `main` branch.
- Build the app when commits are pushed to the `main` branch.
- Deploy the built app (i.e. the files produced in the `dist/` directory by the command `npm run build`) either to GitHub Pages or to one of your team members' ENGR web space.

Code review

You must perform code review on all code in the main branch of your project repository. In other words, no code can be committed directly to the `main` branch of the repository. Instead, all new code must be developed in a separate branch, and it can only be merged into the `main` branch via pull request.

To enforce this requirement, you should set up a branch protection rule on the `main` branch of your project repository on GitHub. This branch protection rule should include the following protections:

- Require a pull request before merging
- Require status checks to pass before merging
    - This means your tests will have to pass on the pull request before you can merge it.
- Require conversation resolution before merging

Tips for working with the project code

Here are a few tips for working with the application code for the project:

- Make sure to read the README included in the project repository to understand how to run or build the application.
- CSS styles are incorporated into the application via `require()` statements in the JavaScript code, which is made possible by the build tool that's being used to build and run the application. So that these `require()` statements don't cause problems with tests you run on the un-built application through Jest, a configuration is added to `package.json` that will essentially ignore CSS `require()` statements during testing with Jest. Do not modify this configuration or the file it relies on in the directory `src/__mocks__/`. You can read more about this configuration in the Jest docs.

Grading demonstrations

To get a grade for your project, your team must do a brief (10-15 minute) demonstration to me (Hess) of your project's functionality. To get a grade for your project, your team must do a demo. These demos will be scheduled during finals week. I'll send more details on scheduling demos for the final project when we get closer to that time.

Submission

All code for your final project must be merged into the main branch of the repo created for your team using the GitHub Classroom link above before your grading demo for the project.

Grading criteria

Grades for the final project will be assigned to your entire team. The final project is worth 100 points total, broken down as follows:

- 20 points – Submission contains unit tests satisfying all requirements listed above
- 20 points – Submission contains integration tests satisfying all requirements listed above
- 20 points – Submission contains end-to-end tests satisfying all requirements listed above
- 20 points – Submission contains a CI/CD pipeline specification satisfying all requirements listed above
- 20 points – All code added to the `main` branch of your project repository was code reviewed, as described above

Remember also that if your team does not do a demo for your project, you will receive a zero for it.

Individual grades

Your individual grade for the project will be based on your team's grade and also on evidence of your meaningful participation in your team's work on the project, including from these sources:

- The commit log of your GitHub repository.
- Your presence at and participation in your team's project demo.
- A team evaluation completed by each member of your project team.

In particular, if your GitHub commit log shows that you did not make meaningful contributions to your team's implementation of your app, if you do not participate in your team's demonstration of your app (without explicit prior approval by me), or if your project teammates submit team evaluations in which they agree that you did not do an appropriate share of the work on your final project, you will receive a lower grade on the project than your teammates. I may use other sources as evidence of your participation, as well.