Lab 8

The format of this lab will be a little bit different since we want to discuss how to write high quality production-ready web apps, which is difficult to do in 1.5 hours 😁. Therefore, we are going to do a **Case Study** instead.

The subject of our study today is an app I developed with my team during our *Software Quality and Reliability* course (4th year, Software Engineering) called **Cast** (https://github.com/aabounegm/cast) that acts as a showcase of many of the code quality tools you can use in your projects.
Let's start by checking out the deployed version (https://cast-iu.pages.dev/) on CloudFlare Pages. (Notice the SSR?)

Team workflow

Back to the GitHub repo.
Firstly, quality starts from the team and the harmony between them, particularly the workflow they agree upon among themselves.

In this particular instance, our workflow went like this:

1. In the beginning of the sprint, someone would create issues for every individual task, prefilled with acceptance criteria and milestoned to a particular sprint
2. After the issues are created, they are assigned to a single responsible person
3. If the assignment isn't satisfactory, issues are reassigned upon agreement of all involved parties
4. If the issue is irrelevant and assigned to no one, assign it to yourself before starting to work on it to let people know
5. Once you're done/when you want feedback, you create a PR, linking the issue with GitHub linking keywords.
6. One person should be assigned for review, preferably, randomly (see our GitHub Actions config (https://github.com/aabounegm/cast/blob/64919860d0c7ddb1fefa0d1b5dfe761ee53bfcb0/.github/workflows/reviewer-lottery.yml)).
7. Once the CI (which runs tests and other static analysis checks) passes and the reviewer accepts, the PR is merged and the branch is deleted

We also made use of GitHub's auto-merge feature, which would merge the Pull Request once all the criteria are met (at least one approval and specific checks passing).

This helped make sure that everyone in the team is working together and aware of everything that others are working on, decreasing the probability of repeating tasks or having misunderstandings about the codebase.

Code quality

As you can see, there are so many config files (https://www.youtube.com/watch?v=14WanxTD2O4) in the root of the repo, so let’s explore what some of them are for.

Formatting

For formatting, we made use of Prettier (https://prettier.io/), which comes preconfigured with the SvelteKit template. Make sure to also install the VS Code extension (https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode) and configure the editor’s options to format on save, using Prettier as the default formatter. It is a good practice to add these options to the workspace settings (under `.vscode/settings.json`) to ensure consistency across the team members as such:

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode"
}
```

Integrations exist for WebStorm (https://prettier.io/docs/en/webstorm.html), Atom (https://github.com/prettier/prettier-atom), Sublime (https://packagecontrol.io/packages/JsPrettier), and other editors (https://prettier.io/#:~:text=Editor%20Support) as well.

Static analysis

For linting, there is no tool more popular or widely used than ESLint (https://eslint.org/). It also has a VS Code extension (https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint) that highlights the problems right in the code rather than just in the CLI.

You will need to configure the plugins and parsers for ESLint to understand Svelte and TypeScript files, but you can simply use the configuration generated by SvelteKit itself. Note that one of the plugins there is called `prettier` , and it disables the rules that conflict with Prettier’s format (since ESLint has some rules that deal with formatting as well. Additionally, you can configure a setting in VS Code to fix all auto-fixable problem automatically:

```
"editor.codeActionsOnSave": {
  "source.fixAll.eslint": true
}
```

Feature-Sliced Design

For a refresher on Feature-Sliced Design, check out their documentation (https://feature-sliced.design/). As for the implementation in our project, check out the lib (https://github.com/aabounegm/cast/tree/main/src/lib) folder. You can find other implementations in the examples page (https://feature-sliced.design/examples) of the docs.
Following FSD was difficult at first, but later on it helped everyone know where exactly to jump to in the code whenever we need to fix a bug or add some new code, dramatically increasing the productivity.

Husky

To ensure that no one pushes code that violates our configuration of the tools above (Prettier and ESLint), we set up Git Hooks (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks) that will run ESLint and Prettier before every commit, and refuse to commit if it finds errors. The way to enforce the same Git hooks for all team members is through a tool called Husky (https://github.com/typicode/husky).
Simply, after following the setup instructions in their documentation, we added a `.husky/pre-commit` (https://github.com/aabounegm/cast/blob/64919860d0c7ddb1fefa0d1b5dfe761ee53bfcb0/.husky/pre-commit) file containing a script that will run before every commit on the staged files and refuse to commit if that script fails.

It’s important to have the pre-commit hook exit as fast as possible to not cause much delay in committting and hence decrease productivity, which is why `lint-staged` (https://github.com/okonet/lint-staged) is typically used to avoid running the linter/formatter on the whole project. They should run on the entire project anyway in the CI/CD.

Testing

Unit

For unit testing, we made use of Jest (https://jestjs.io/) along with the Testing Library (https://testing-library.com/) to avoid including implementation details and instead focus on testing components the way the user interacts with them.
These tests can be found in the `spec` directories almost everywhere inside `lib`. This is so that the unit tests can reside where they belong: next to the units they are testing.

Jest also exports test coverage information which tells us exactly what lines in our code are covered by tests and which ones are not, with a percentage summarizing the coverage over the whole projects. We display this number as a badge (https://shields.io/) in the README and send the data to codecov (https://about.codecov.io/) for detailed analysis.

The alternative tool that comes with SvelteKit by default nowadays is Vitest (https://vitest.dev/).

End-to-End

E2E tests run our app in a Chrome instance controlled by Cypress (https://www.cypress.io/) so that its commands would mimic actions the user would do. It tests the application as a whole by navigating through the different pages and performing different actions, and then verifying that expected outcomes did indeed take place.
E2E tests can also run in the CI since it runs chrome in headless mode by default, which means it does the rendering and everything but only in-memory without actually needing a GUI.

Sveltekit template now ships with Playwright (https://playwright.dev/) instead, which has the same purpose as Cypress.

Visual

Another relatively recently emerging form of testing relevant to GUI apps in general, or frontend web apps in particular is called **visual testing** (also known as **UI testing**). While we can use unit and E2E tests to check that components have particular CSS classes or styles, that doesn't guarantee that it will look exactly as we want. For more on that, check out this talk from GitHub Universe 2020 (https://www.youtube.com/watch?v=tIIKWGLrzkQ).

We decided to use Applitools (https://applitools.com/) for visual testing due to its ease of ease and its integrations with many testing solutions including Cypress. All we need is a driver test script (https://github.com/aabounegm/cast/blob/64919860d0c7ddb1fefa0d1b5dfe761ee53bfcb0/cypress/integration/applitools.spec.ts) as part of Cypress's integration tests and that will capture screenshots and send them to Applitool's server to be analyzed using AI. You can then go to the dashboard and do the rest there.

Lighthouse

Lighthouse (https://developer.chrome.com/docs/lighthouse/overview/) is an open-source tool by Google for measuring the performance of web pages using many metrics and insights. It reports back information about SEO-friendliness, performance, accessibility, and best practices in general. It is also available with a GUI inside of Chrome's DevTools.
These metrics should be gathered on the production version of the app in a reproducible way since it can be affected by the environment in which it runs (e.g.: extensions installed on your browser).

Lighthouse is usually run through the Chrome DevTools, but it can also be automated in the CI pipeline. Check out the GitHub Action workflow (https://github.com/aabounegm/cast/blob/64919860d0c7ddb1fefa0d1b5dfe761ee53bfcb0/.github/workflows/performance-test-pr.yml) that runs lighthouse tests to understand more.

Deployment to CloudFlare Pages

Finally, after all the quality checks pass, it’s time for deployment. During the 6th lecture, we covered the architectural pattern known as **Edge Rendering**, which server-side renders your app directly on the CDN server. One service that implements such technique is CloudFlare pages (https://pages.dev). It is so easy to use, you can set it up in 5 minutes without having to configure any custom CI/CD pipelines.

First, let’s create a demo app using the CLI: `npm create svelte@latest sverdle` . Don’t forget to run the git commands shown in the output. Next, create a GitHub repo and push your code to it.
You can replace `adapter-auto` with `adapter-cloudflare` , but this is optional since `adapter-auto` can detect the CloudFlare environment and automatically pick the correct adapter (but still recommended).

Now go back to CloudFlare Pages, log in (or sign up) to your account and go to the Pages dashboard. Create a new project by linking it to your GitHub account and selecting the repo you just created (*Note: uploading the folder directly will not work as it will host as a static website*). Select the SvelteKit preset and it will autofill `npm run build` as the build command and `.sveltekit/cloudflare` as the output directory. The only thing you need to change is to add the environment variable `NODE_VERSION=16` (to tell CloudFlare to use it instead of the default older version since SvelteKit requires at least v16).

And that’s all! your project should be ready and deployed in a matter of minutes. The CloudFlare adapter generates special scripts that CloudFlare understands to run on the CDN server before sending the webpage to the user, which is how it can perform SSR.

Homework

If you did not select the ESLint and Prettier options when setting up the SvelteKit project for the previous assignment, it is time to add them. Use appropriate configurations that allow them to read Svelte and TypeScript files, and make sure there is a script in `package.json` to run the checks (it comes with template and is typically called `check`).

Additionally, add a step in your CI pipeline that would run the linting/formatting checks and fail the CI if they don’t pass.

Then, make your comic server-side rendered (meaning that viewing page source should show the tags with your specific data). It does not necessarily need to run *only* on the server-side (*hint*: the `load` function from `+page.ts` runs on the server during the initial request, and on the client when navigating from another page).
Lastly, add the appropriate head tags to all pages/layout: title, author and description.

Deploy the app to CloudFlare Pages and submit the link along with your repo link.

Bonus: add pre-commit hooks using Husky and lint-staged (https://github.com/okonet/lint-staged).

Bonus 2: write some meaningful unit tests and run them in your CI pipeline.