Contents

Overview

It's group project time. This is one of the most important moments in the course. You will put together all that you've learned and build something super cool that will look great on as a portfolio piece. You'll learn the trials and tribulations of collaboration with other students. Many students feel they learn as much during these weeks as the rest of the course combined. Now is your moment to put everything to use in a realistic scenario.

1 Requirements

1.1 General requirements

All group projects with Kickstart Coding must adhere to the following criteria:

1. **Must be a full-stack web application, with both frontend and backend components**
2. Must have a presentation.
3. Must be deployed live to Heroku.[1]
4. Must have a polished, attractive user-interface, be clear in function to the general public, and not be filled test data or placeholders.
5. Must have multiple pages[2] displaying different data.
6. Must support user-supplied data which it saves to a database.
7. Must use GitHub[3] to collaborate.

[1] Deployment to other server environments, such as Digital Ocean or AWS, is also permissible, as long as it is publicly available.

[2] Or routes, in the case of a single-page app

[3] Using GitLab or BitBucket is okay, just not recommended

1.2 Code Requirements

- In addition to the general requirements, the Kickstart Frontend project must use React and React Router to build a single-page app.[4]
- There must be a back-end that saves data to the database, and exposes an API that allows the front-end to read and write data to and from the database.
- You may use Postgres or MongoDB for your database system[5]
- Must use 1 or more `npm` packages or libraries that was not used in class[6]
- The backend may be written in Python[7], or Node.js[8]
- Use of advanced CSS techniques such as CSS grid, flexbox, animations, transforms, SVG, etc[9]

1.3 Soft-requirements

- A clean, attractive design
- The idea must be realistic.[10]
- Clean coding practices, including unit testing of a few core parts.
- Modular React, with components heavily broken-down.
- Have a `README.md` written in Markdown that has instructions on how to run your project, and includes screenshots or GIFs of it in action, and a link to the live site.

1.4 Nice-to-haves

- Use or ingest a publicly available data-set or API.

  For APIs, take a look at this list: `github.com/abhishekbanthia/Public-APIs`

  This Quora question has a lot of links to data-sets. Some of these might be easy to use (e.g., CSV or JSON based ones) while others may be more difficult:

  `quora.com/Where-can-I-find-large-datasets-open-to-the-public`
- Using Redux is purely optional, and possibly *not* recommended.
- User sign-up and log-in system.[11]

[4] If you decide to use Django, and want to use authentication for your app, it's okay to put authentication on a separate page from your React-powered SPA, for a "hybrid" approach, as long as almost all of your front-end is done in the React-powered SPA.

[5] MySQL or another noSQL database also okay.

[6] Consider trying out different React widget libraries.

[7] Using Django, Flask, Bottle or some similar framework.

[8] Using Express.js, such as in a MERN stack.

[9] While technically not forbidden, avoid relying too much on canned templates for this project. Can you build an attractive template from scratch?

[10] It's okay to build it as a proof of concept, or even for it to be humorous, but the goal must be realistic and factual, something people would actually use. To contrast, don't make a "Moon Landing UI" or "Shiba Chat" as your project unless you actually have realistic expectations eventually writing software for the moon or for Shibas to be able to talk. While these might be fun as toy exercises, they are not appropriate for projects.

[11] Single sign-on with other services such as Google or Facebook is also okay. However, if you are not using a pre-made one, this can easily be a rabbit hole, so be careful with spending too much time on a user system.

2 Breaking down roles

With this project, you can have the cleanest separation of roles. Naturally, individuals can do work in multiple roles, but with full-stack single-page apps, the roles were never clearer:

- *Design / PM* - Draw mock-ups, create HTML and CSS click-through, spec-out core application logic, and generate image assets
- *Front-end component engineer* - Build out components based on, work mostly on lower-level components in isolation to make them behave correctly and look attractive
- *Front-end API engineer* - Work with the component engineer and the back-end engineer to build out the *App* component, handling the state transitions and logic
- *Back-end engineer* - Write in Python or Node.js to create an API (e.g. REST API) for the front-end engineer to use to access and store data

3 Milestones

All written milestones should be submitted as Markdown. Feel free to re-use parts of this Markdown in documenting your project's `README.md`. Also, feel free to "get ahead": If you finish one milestone and want to just jump right into application development, don't let this structure hold you back.

3.1 Milestone 1 - Group formation, project specification

For the first milestone, the following must be submitted by the deadline specified.

1. Group information: The names of the students in your group. You should have 2-4 students in your group.
2. Software goal and concept: Explain in terms of one or more "user stories" of the problem you're software is attempting to solve.
3. Submit the project specifications. This including which APIs or data it might be using, and broadly what work you expect might be involved. This should also include some cursory wireframing: You don't have to wireframe every single feature or screen, but at least get the core ones down, just to ensure everyone in your group is broadly on the same page visually.
4. Challenges and unknowns: Factor in problems you may encounter, given the limited time available and what your team has so far learned of web development. For potential "show-stoppers", include contingency plans.
5. Group roles: Roughly outline the roles of each group member. Example responsibilities include *front-end engineer / design*, *back-end engineer*, and *API & data engineer*. It's okay if you don't follow 100% one role or another, or mix it up.

3.2 Milestone 2 - Templates, wireframes, proof-of-concepts

1. HTML & CSS "Click-through" template of core concepts
2. Full wireframes of **all** screens
3. Proof of concepts for core data, API, core view logic

Click-through template: The next milestone involves fleshing out your idea by building out wireframes and static HTML & CSS templates. Focus on styling, so include only dummy data at this stage. Feel free to leverage code you find online, but note that by the end, you should try to heavily customized any starter template code to the point that it is not immediately obvious where it came from.

Full wireframes: Try to thoroughly plan out every page and aspect of your application at this point, possibly cutting down features until you have 1 or 2 core features you build the project around.[12]

API & data proof-of-concepts: Those working on API and back-end code should start work on basic proof-of-concepts. Build a script that fetches the *exact data you need* from the API, either as a CLI script or a little server based on an activity or sample project.[13]

3.2.1 What to submit

1. Screenshots of your wireframes and static pages
2. GitHub organization for your group
3. Ideally: Proof of concept of core data, API, or backend concepts

3.3 Milestone 3 - MVP

Put it all together! Your goal is to build out the MVP. This will involve intense work on both the front-end and the back-end to achieve the required goal.

3.3.1 What to submit

1. The repo for your project
2. A link to your live deployed site

3.4 Milestone 4 - Final presentations

Now that your MVP is up, your next goal is smoothing over bugs, polishing, and adding any extra essential features before demo day.

3.4.1 Presentation requirements

- You should use Google Docs or a similar slideshow technology
- You must tell your user story (the problem you are trying to solve)
- Each group member should speak about something
- You should dive into the technical challenges you overcame
- Part of your presentation must involve a live demo of the site

[12] See "How to Wireframe" section for tips here.
[13] If you are not using an API, backend engineers should then work on proof-of-concepts of the site itself.

3.4.2 What to submit

Submit this to the class channel!

- A link to *a publicly accessible* Google Docs or similar [14]
- A link to your final live deployed site

3.5 Milestone 5 (optional) - Portfolio

Class is over, but your project lives on. Now you have the chance to flesh out your project and polish it into a shining portfolio piece.

1. Flesh out README.md with full description
2. Buy domain, and/or spend more time on logo and branding
3. Add to resume
4. Add any additional stand-out features and generally polish the project
5. Add a logo and description to your organization.[15]

[14] The presentation computer might not work with PowerPoint or Keynote.
[15] If you have time, fleshing it out to looking like a viable, real project – such as a start-up proof-of-concept – can be great for your tech resume.

4 Project idea tips

Don't know where to start? Have ideas but not sure if they are good? Here are some tips.

4.1 How to brainstorm for topics

Ask yourself these questions:

1. What are you interested in outside of programming?
   - What are you passionate about?
   - Your niche expertise + tech = magic
2. What non-tech field or industry really needs some "tech-love"?
   - A lot of fields have software that lags behind the cutting edge.
3. What are issues you face as a coding student?
   - A well-executed code learning tool can get really popular
   - Maybe something that Kickstart Coding students could use in next cohort

4.1.1 How to get started

1. Browse APIs, public data sources
   - Mix and match - combine them at random to get ideas
2. If your idea is too big, think of a "beachhead"
   - You may want to "conquer" all of certain space, but you have to start somewhere
   - How do you limit your idea's scope? Think: Geographically, topically, market, demographic, experience level, enterprise vs consumer

4.2 How to wireframe

- Wireframe until you can't any more
- Deconstruct each other's wireframes
  - "What happens if I click here?"
  - "Where does this data come from?"
- A wireframe is complete when everything that can be clicked leads to another wireframe, and when every bit of data has an explanation (either from a known API, or a user-entered form, which also has a wireframe)

4.3 Thinking like a user

1. Assume the user cannot keep up habits
   - If the app requires constant attention or use, they probably won't keep up with it, unless you make it really easy[16]
2. Assume the user is drunk
   - Everything has to be extra-obvious, but also eloquent
3. Why should a user use your app instead of *XYZ*?
   - Think of at least 1 thing your app does differently, then go all in on that.

4.4 Idea Pitfalls

1. **User participation:** Be cautious of project ideas that are primarily public facing communication platforms requiring a lot of user participation to be useful at all.[17]
2. **Start-up business models:** Be cautious of project ideas that would require a real business structure or marketing system to even make sense. [18]
3. **Games:** Be cautious of project ideas that are purely video-games. Games are a lot harder to make since graphics requires programming concepts we haven't talked much about, and also multiplayer games have a lot more complications. Finally, games must be *fun*, which is not an easy requirement.
4. **Content-based:** Be cautious of project ideas that rely on content you don't have. [19]

4.5 Collaborating with GitHub

4.5.1 Starting

1. One student must start by creating an "Organization" and adding the other students
2. Under that organization, create the repo.
3. Ensure you create a `.gitignore` when creating the repo.[20]
4. Consider starting your project with an in-class activity

[16] Example: A fridge inventory app which requires tediously entering in every item in your fridge. Many of these exist but aren't used much because people don't have time to keep them updated.
[17] Examples of this: *Craigslist* or *Facebook* - think how boring these sites would be without anyone using it! It would barely even be clear what the site was meant to do.
[18] Examples of this: *Amazon*, *Uber* – while you could make a proof-of-concept of the technical side, much of the "genius" of these companies have more to do with their business model.
[19] Examples of this: A tutorial site for learning Django might be awesome, it requires somebody to write the tutorials.
[20] Look online for more files to add to it, such as `.DS_Store` if you have macOS users in your group.

4.5.2 Avoiding merge conflicts

One of the hardest parts about group work is avoiding merge conflicts. These will be the bane of your existence.

1. Avoid modifying the same file. If you must, communicate to make sure you have the latest version from everyone, and there are no un-committed changes on someones computer that might be broken.
2. Use branches and GitHub Pull Requests to combine your code.
3. Remember to always pull before pushing!
4. If all fails and you get your repo in some sad state, copy whatever you want to keep out of it, delete the directory, and clone again!

5 Remember

> Do one thing, and do it well!