1. Mocking as a Technique

Your Census datasource from the Server sprint was a great place to use mocks in testing. Written naively, unit testing that source class would likely send API requests *every time* the class was queried. This would be expensive if the API wasn't free (and makes your tests rely on an internet connection!) We suggested using mocks to avoid this in the handout.

Ideally, mocks are fully transparent to the code that uses them. E.g., if your cache or adapter code needed to be aware of whether it was receiving mocked or real API responses, there is a high risk that testing with mocks would not exercise some vital path in your code.

Answer two questions for this item:

- **Reflect on mocking in 2.1 and 2.2 (pick *one* of your repositories to write about): to what extent did you mock as described above on sprint 2.1 or 2.2? What would you or another developer need to do, if anything, to improve your mocking? If you are unsure, here is a hint: dependency injection is a great way to resolve this problem, since your code can accept any object that implements the right interface. Identify the interface that makes the most sense – e.g., on CSV, your parser could work identically with *any* `Reader` object. (If you did all this, say how the process works.)**
    - For sprints 2.1 and 2.2 we used an interface that was implemented by both the mocked data class and the real API datasource class. When using caching, the cache class is indeed fully transparent to whether it is working with real or mocked data, since the type of the object provided to it was the interface. The server class is the only one that "knows" the actual datasource type. When using the application, the server gets initialized with a real API datasource object, but in testing we are initializing it with a mocked datasource object. In the Server class implementation we pass this object down to the cache class, which uses it to handle the query request.

- **Reflect on mocking in 3.1: what will need to change in your Mock application when you begin to integrate your backend API server? We will cover how to send requests, parse JSON data, etc. in TypeScript; don't worry about that for this question. Focus instead of the structure of your application. Where will code need to change? What tests will need to be updated, or completely rewritten? Finally, where might you *still* want to use mocking, even after integration is completed in the next sprint?**
    - The SelectHistory TypeScript file is the specific place where most of the code updates will be done for API server integration. That is where we are currently calling a getTable method which retrieves local data stored in a separate tsx file. Ideally, we think that we should be able to retrieve both mocked data and real

API data as JSON files. This would enable two important things: being able to easily switch between using real data and mocking because both ways would parse JSON files, and making us capable of handling larger amounts of data that cannot easily be hard-coded in a TypeScript file. We are not exactly sure how this will be merged with an interface, dependency-injection backend strategy, but we're curious to find out!
Mocked data will have to be completely changed and stored in json files, instead of being hard-coded, so testing will undergo some significant changes. We will still use mocking in testing to effectively manage resources and not overwhelm the API server with lots of requests.

3. Code Review (Your Own Work)

Reviewing your own code can sometimes be useful. Submit (as part of this reflection) comments on things you dislike about your Mock code, and what you might have done differently, given time. *Be honest; no code is perfect, especially at this stage.* However, also be compassionate; it is OK to say "I'm not sure what to do about this" (provided that's honest) but it isn't OK to say "this looks good to me". Both partners should participate in this exercise.

- Extend accessibility such that the application can be used on different mobile devices with different resolutions
- TypeScript action listeners - being able to switch between dropdown menu options through arrow keys and press Enter to select an option. Our current implementation does not work with screen readers.
- We currently have a login option but there is no option to create a new account. If we had more time we would also use it for security purposes and preventing users from seeing private data
- Improve user interface for enhanced user experience, through better fonts, more readable table data, background graphics
- Make table data stored in text/CSV files instead of being hard-coded in a TypeScript file, which would allow us to work with larger-scale data.

Meta Reflection

1. Was the *programming* experience new?

In what ways did working with *front-end* programming in *TypeScript* differ from the work you did on the CSV sprint (or projects with Java, Pyret, etc. in other classes)? Did anything surprise you?

It is a bit unfamiliar to work with 3 different programming languages in one file. Also, we have found that TypeScript and React code might sometimes have a harder-to-comprehend structure that has you jump between a lot of files in order to properly understand how everything is tied together. A very cool thing is the TypeScript

narrowing, through which the compiler knows the type of objects based on different branches of type-checking.

2. Was the *testing experience new?*

In what ways did writing *tests* for this sprint differ from testing you've done before? What was the same? Did anything surprise you?

Testing with Playwright was very straightforward and easy to write, especially since the code lines are readable and similar to natural language. We still however kept a rigorous standard on testing by handling hopefully all edge cases and predicting various user behaviors.

3. Bugs?

Did you encounter any bugs during your work on this sprint? If yes, what were they and how did you fix them? If no, how do you think that you managed to avoid them? Is it possible that any bugs remain, lurking in your code undiscovered?

We encountered bugs mostly in the beginning, when we were making changes to the gearup code, and most of them were related to incorrect type handling due to the fact that we were not modifying the interface types and fields that we were working with. There's always a chance that a bug is lurking in our code, perhaps specific to table retrieval. However, our tests are rigorous so we don't think this is likely.