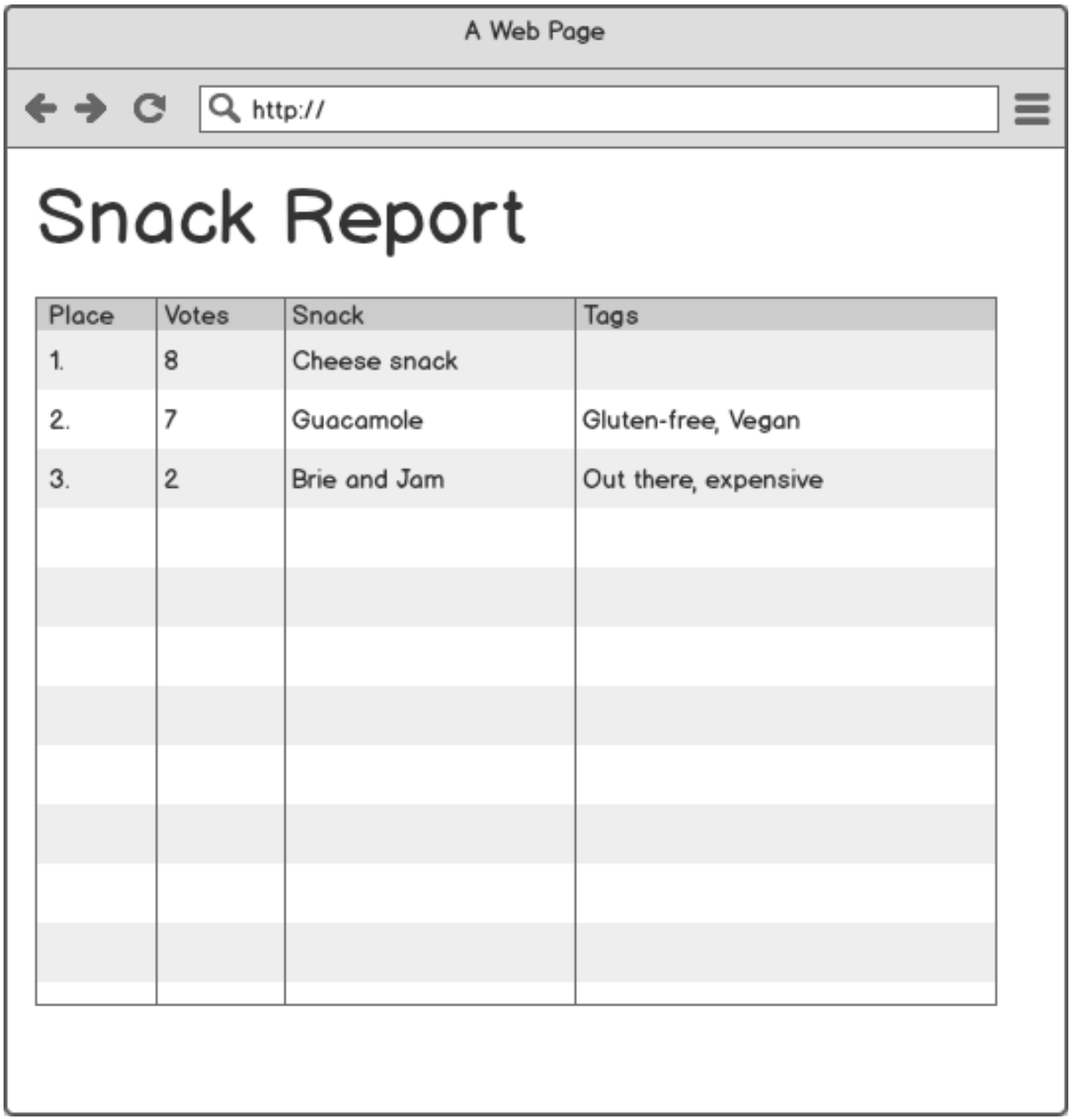
Exercise 1

Snack Report



1.1 Report table UI

Let's start by creating the interface for our new Snack Report.

To get started:

1. Create a new folder/file: `modules/client/pages/snack-report`.
2. In that folder, add two new files: `snack-report-ui.tsx`, `snack-report-ui.stories.tsx`

Create a placeholder UI component in `snack-report-ui.tsx`:

```
import * as React from "react";

export const SnackReportUI: React.SFC = props => {
  return <div className="snack-report">Hello!</div>;
};
```

Set up initial stories in the stories file:

1. Use `storiesOf` and `.add` to create an example - see home page stories for an example.
2. import the new `snack-report-ui.stories.tsx` file in `client/stories.ts`

Run `yarn dev:storybook` and use that to build your UI.

Run `yarn test:unit --watch` to run the interactive test runner.
While that's running, hit `p` and type `stories`. You should see your example show up as a test case!

If your test case is failing, go verify it visually in Storybook. If you're happy with it, you can hit `u` in this interactive test mode to update the saved snapshot of your stories and bless this as the new expected state.

1.2 Parameterize Report UI

Let's update our component to be a function of data.

React components fundamentally just produce new HTML as a function of the attributes passed into them (props). `React.SFC` is just a function type that takes props and produces HTML.

Our UI component doesn't currently take any props except the React defaults. Let's update our component to drive the UI from data.

Add an interface to your ui file, e.g.

```
export interface SnackReportUIProps {
  // more to come
}
```

Update the type of your component to be `React.SFC<SnackReportUIProps>`.

Define an interface to describe the inputs to your component. I suggest creating an additional interface called `SnackReportRow`, and having your `SnackReportUIProps` take an array of them called `rows`.

Once you declare your props type, you should see that you now have type errors in your snack report stories. Each entry in `SnackReportUIProps` needs to be an attribute of you `SnackReportUI` JSX tag. You can interpolate non-string values with e.g. `<SnackReportUI foo={1 + 1} />`. Inline objects need `{{` and `}}`, as the outer curly braces simply escape input.

Update your story example props to have the same data you had hard-coded initially. Typescript should guide you in providing a valid set of props as input.

Once your story type checks, update your code to generate rows from your props. You may need to read about how to create Lists in React – they must have a `key` property.

Check your unit tests. They may or may not be passing. If they are not, the diff should be minor. Hit `u` to update your snapshot.

1.3 Empty and Loading

Add a storybook story and implement an empty state for the report.

Next, our report will load it's data after the table is shown to the user. We therefore need to

support a loading state.

Update your props to represent this as a potentially `null` data property. Your new `rows` property will have a type that looks something like `SnackReportRow[] | null`.

Use an `if` to guard against the null case. Now, hover over your `rows` property before the statement, in the `then` clause, and in the `else` clause. (You may need to add an unnecessary reference to `props.rows` in order to do this.) How does the type change?

Don't forget to add a storybook story for your loading state.

2.1 Snack Report Page Container

Add an `index.tsx` file to your `snack-report` folder. export a new `SFC` that renders your snack report in the loading state, and export it with the name `SnackReport`.

Using the home page as an example, add a URL for your snack report to the router and add a link in the header, just like the other pages.

2.2 Page Container Test

Add a `__tests__` folder to `snack-report`, and add a `snack-report-page.test.tsx` file to that.

Using "Begins in a loading state" test from `home-page.test.tsx`, create a unit test for your snack report.

To keep things simple, assert that the loading message appears anywhere on the page.

2.3.1 Get ready to wire to GraphQL

Let's get some data showing up!

We've got an existing GraphQL query for our dashboard. Ultimately, this will not serve our needs, but it can serve as a good starting point to connect our new snack report component to the server.

1 Open up http://localhost:3000/graphiql and run the query from `DashboardSnacks.graphql`. GraphiQL is a super useful tool for interactively testing graphql queries.

2 Create a new query called `SnackReport` in the `graphql-queries` folder. Aside from changing the name, leave the rest of the query alone.

Once you save the file, open `graphql-types.ts`. You should now see a `SnackReportQuery` type in this file.

This file is generated whenever you add a new query or mutation to the `client` module – it contains argument and result types for every query and mutation you write.

If `SnackReportQuery` does **not** show up here, you have an error in your query. Look at the output of your webpack server. Look for lines that start with a blue `[graphql-types]`. Errors in the graphql type generation process show up here.

3 In `snack-report/index.tsx`, define a function `dataToRows(data: SnackReportQuery): SnackReportRow[]`.

Add a `data-to-rows.test.ts` file to the `__tests__` folder. Test-drive the implementation of this function. Note that `allSnacks` will return null if the query fails for some reason – be sure to handle that case – just treat it like an empty result for now.

Use a sensible approach to populating `place` here. Leave `tags` blank for now

Remember that you can use `p` to run just this test.

TIP: In each test you write, define a `const data: SnackReportQuery` and a `const expected: SnackReportRow[]` before calling your function and asserting equality. Explicitly defining these variables with the input and output types will allow TypeScript to help you write the test.

2.3.2 – Wire your page up to the GraphQL query

We're going to use an Apollo higher-order component ("HOC") to connect our dumb UI component to the results of our query.

`react-apollo` comes with a function called `graphql` that takes a graphql query and some options, and returns a function that will wire up a component to a query for you.

We're now ready to replace our always-loading snack report with one which can show actual data.

Replace your current `SnackReportPage` with the following skeleton (update as necessary):

```
const wireToApollo = graphql<
  SnackReportQuery,
  {},
  SnackReportUIProps
>(require("client/graphql-queries/SnackReport.graphql"), {
  props(result): SnackReportUIProps {
    // Use `dataToProps` to implement me
  }
});

export const SnackReportPage = wireToApollo(SnackReportUI)
```

`graphl` takes 3 type arguments:

1. The type of the result of the query – `SnackReportQuery`
2. The type of the props of coming into the connected component – we have none, so we use `{}`.
3. The type of the props of the component it will wrap – `SnackReportUIProps`.

1 We now need to implement the `props` function to produce our `SnackReportUIProps` from the `result`. `result.data.allSnacks` will be present with the results once the the

result is loaded. Before that, it will be called with `result.data.loading === true`

Use your `dataToProps` function to implement `props`.

TADA – your report should now have data after a brief loading state.

Your snack report page test should still be passing.

2 Add a test for your snack page using the "Shows the snacks in a list" home page test as an example

---

3.1 A new graphl query for our report

We were able to get something up and running by copying our initial DashboardSnacks query, but this isn't going to work in the long run.

For one thing, the dashboard doesn't show most-popular-snack first.

Let's start dipping our toes into the server side by adding a new endpoint designed for our page.

1 Add a `topSnacks` to `schema.graphql` side-by-side with allSnacks. We'll start by creating a new endpoint just like allSnacks, but with a different order.

2 Change `SnackReport.graphql` to use `topSnacks` instead of `allSnacks`. Go look at your webpack output. You should now see some new type errors to show up.

3 In VS Code, hit cmd-shift-P and run the "Reload Window" command. VS Code doesn't necessarily pick up changes to the auto-generated types when we change graphql files – this is often a good idea after changing .graphql files if typescript in VSCode seems to be confused.

4 Fix the type errors referenced in the webpack output. Continue iterating until Webpack gives you a clean report.

5 Using `snack-query.test.ts`, create a `top-snacks-query.test.ts` that will query `topSnacks` instead of `allSnacks`.

5 Open `query.ts` in `graphql-api/resolvers`. This is where the queries on our `Query` type are implemented. Duplicate `allSnacks` and call it `topSnacks`. Test drive this by running `test:unit` focused on your top-snacks test.

6 Fix any other failing tests.

3.2 Top Snacks for Real

We're going to update the resolver for `topSnacks` to sort results by vote count descending.

1 Update `top-snacks-query.test.ts` to create multiple snacks with differing counts, and to expect them to come back in order of voteCount descending. Watch it fail.

2 Go look at the definition of `countForSnack` in `vote-record.ts`. What is a `DataLoader`? What is this thing for and what does it do?

3 Update your `topSnacks` implementation to sort the snacks by vote count descending using the `countForSnacks` DataLoader and `sortBy`.

TIP: DataLoader has a `loadMany` method.

3.3 Tags Graphql query (frontend)

Let's get tags set up. For our first step, let's add a mock endpoint to provide a fixed set of tags for all snacks.

1 Add `tags: [String!]!` to `Snack` in `schema.graphql`. The `!`s mean we promise to not have null tags in the array and we'll always return an array.

2 Create a new query called `snack-tags-query.test.ts` test that creates a snack and queries for its tags.

3 Add a `tags` resolver to `resolvers/snack.ts` – much like the `voteCount` resolver which is already there. Make your test pass by returning hard-coded data.

4 Update your page test and snack query page to show the tags.

Tip Don't forget to check your webpack output when you update graphql files. It will help point out where you still have work to do.

At this point you should have some static tags showing on the snack report.

3.4 Tags backend

Run `yarn db:migrate:make create-tags-and-taggings` to create a migration.

1 Create two tables:

- `tags` which has an autoincrementing `id` and a `name`.
- `taggings` which has an id, a `snackId` and a `tagId`. Both should be foreign key constraints to their respective tables

See the other migration for examples.

Run `yarn db:migrate:latest` to migrate **just your development database**.

Run `NODE_ENV=test yarn db:migrate:latest` to migrate **just your test database**.

2 Create a `tag-record.ts` and `tagging-record.ts` with `Id`, `Unsaved`, and `Saved` types as well as new empty repositories.

3 Create `recordInfo` definitions in `record-infos.ts`.

4 Create a `forSnack` `DataLoader` in `TaggingRepository` using `loaderOf(...).allBelongingTo(...)`. Don't forget to add a repository test.

5 Create a `forTagging` `DataLoader` in `TagRepository` using `loaderOf(...).owning(...)`. Don't forget to add a repository test.

6 Update your tag test to set up some tags/taggings.

7 Add the two new repositories to your GraphQL context in `context.ts`.

8 Update your `tags` resolver to use the two new data loaders and watch your tests pass.

9 Add some tags and taggings to your database and see them in the snack report.