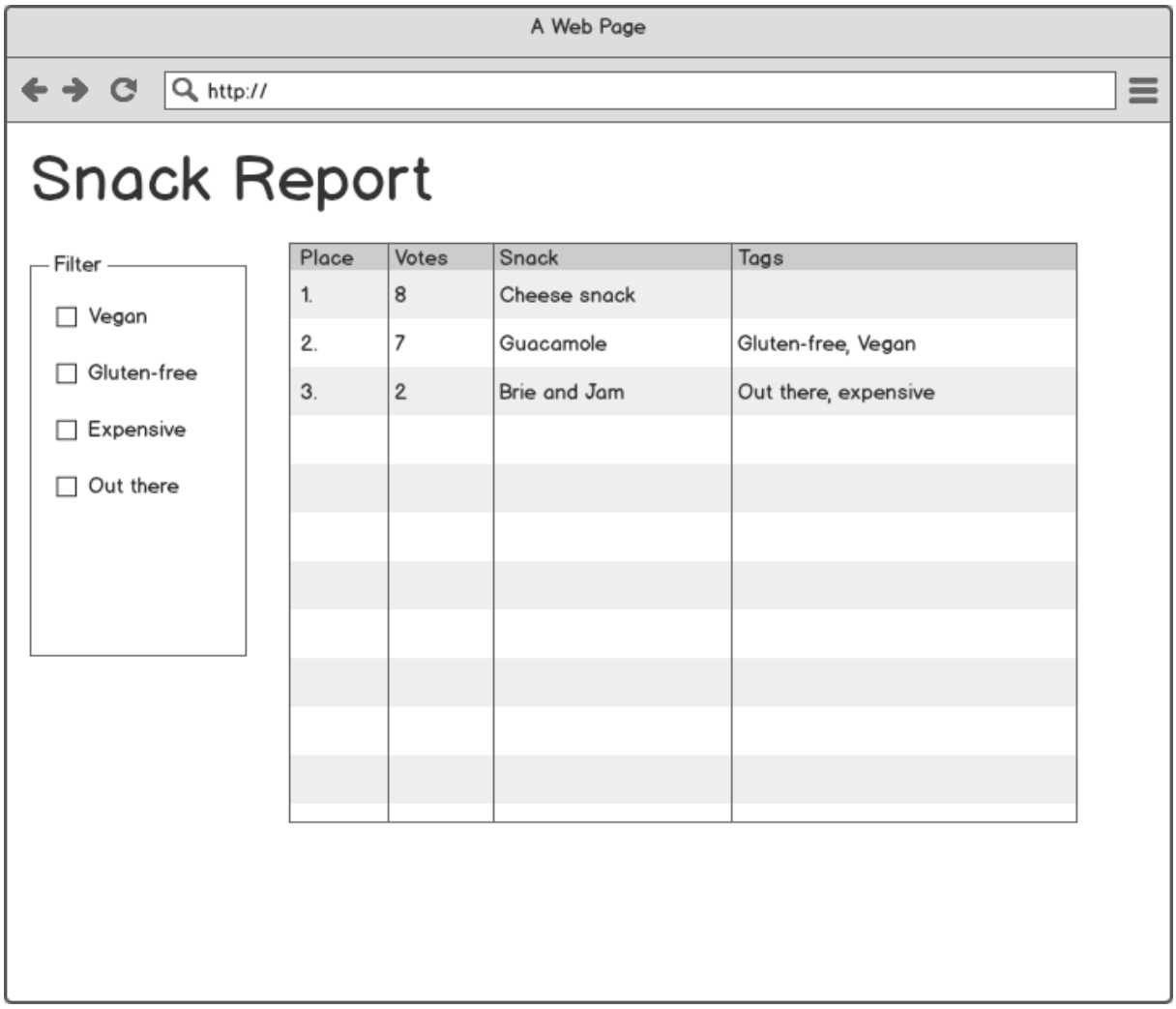
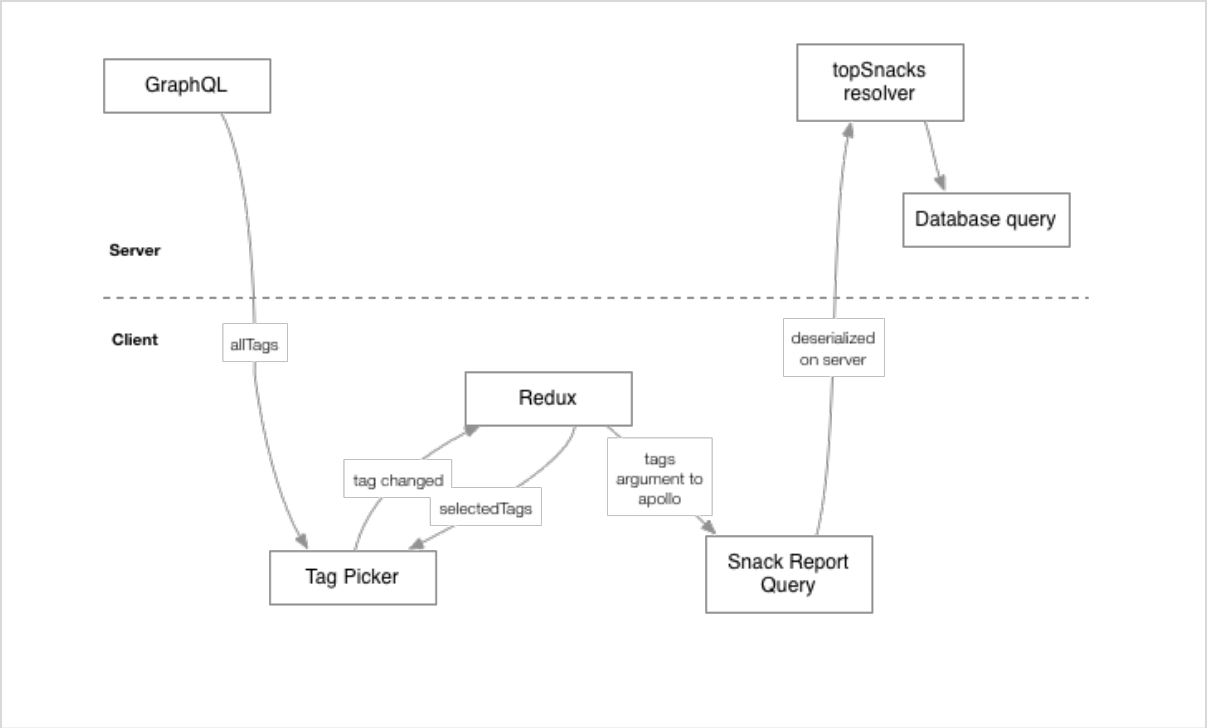
Exercise 2



Data flow:



1 TagSet abstraction

This feature is going to require supporting a concept of a set of tags. These sets of tags will need to go back and forth from client to server. We'll save them in our Redux store to track what tags are selected and they'll be shown to the user in the form of checked checkboxes.

We could define props and types at each stage in a convenient fashion, but this would likely lead to the knowledge of how a set of tags is represented being distributed through the system, and potentially require annoying conversions of our types don't line up exactly.

So let's decide on a format that has the properties we want, and build a code abstraction around it. In particular, our representation should have these properties:

- Serializable – we need to round-trip sets of tags with the server.
- Stable – equivalent sets of tags should be deeply equal
- Easy to use

The first property rules out Sets or any other class-based (non-simple) object type. The second property means simple string arrays are out as well. I suggest we use sorted arrays.

Now, Typescript/JavaScript don't have a built-in notion of sorted arrays. Ideally, we'd have

a `SortedArray<T>` type and operations that make dealing with that structure convenient, and which disallow violating the sortedness invariant. So let's build one.

We have two options for how to approach this:

1. Create a `SortedArray` module.
2. Create a `TagSet` module that happens to use sorted arrays as the underlying data structure.

I'd argue that #2 should be done in either case, because we should decouple our application from the policy decision of how we're representing sets of tags in general. Given that, and that the `sort` function makes implementation easy, let's just go with #2. (We might do both if this were going to be more complicated.)

1 Create `modules/core/tag-set.ts`.
2 In that file, add `export type Type = Flavor<ReadonlyArray<string>, 'TagSet'>`
3 Also, `export const EMPTY: Type = []`

4 Test-drive three functions:

- `has(set: Type, tag: string): boolean`
- `add(set: Type, tag: string): Type`
- `remove(set: Type, tag: string): Type`

Both `add` and `remove` should return the original if they would otherwise be a no-op. Ensure they uphold the invariants.

Note on the name Type: The name `Type` probably seems weird. This blog post has more context. The important thing: every time we use this module we want to do a `import * as TagSet from 'core/tag-set'`

Note on Flavor: `Flavor` lets us add some metadata to a type to track what it is intended to represent, so we don't inadvertently forget and start using it as a simple string array. We get help in the tooltip from the `Flavor`, but most importantly, TypeScript won't allow us to mix flavors, but it will allow an unflavored array to be passed in, which comes in handy when dealing with GraphQL results.

2.1 TagPicker component

Create a new `SFC` component in `client/components` called `TagPicker`. Its props should have:

- `tags: string[]`
- `selected: TagSet.Type`
- `onTagChange: (tag: string, value: boolean) => void`

Test drive this component in your story book. (Don't forget to import the stories file in `stories.ts`)

Add the tag picker to the snack report and add the same three new props to `SnackReportUIProps`. Provide default values on your container's `props` function. Just hard-code a list of tags for now and set up the page to have **at least one** selected tag. You should see it selected on the snack report page.

Get the styling to the point where you're happy with it. We're using Bourbon and Neat. `@import "client/styles_core"` to get access to them if you want to use the Neat grid, for example.

2.2. Tags snack page test

Add a new snack report page test to look for your selected tag.

You may find this snippet helpful to get the labels of selected checkboxes:

```
// Find checked inputs then get the ancestor label element's text
    const selectedTags = page
      .find("input[checked=true]")
      .map(n => n.closest("label").text())
      .sort();
```

2.3 Redux state mapper

Our next step will be to drive the set of selected tags from the redux store. We'll start by showing a fixed set of selected tags that the user won't be able to change. In 2.3 we'll add the ability for those to be changed by the user.

Using the home page as an example, we'll be transitioning our snack report from using one higher-order component (`graphql`), to using two – `connect` and `graphql` .

Currently, we have incoming props (of which there are none) going into `graphql` , which runs our query and produces all the props. We'll now be pulling data from three sources.

- `rows` will continue to come from graphql.
- `selectedTags` will now come from our redux store via a `mapStateToProps` function.
- `onTagChange` will eventually dispatch actions into redux to update the store's `selectedTags` . This will be provided by our `mapDispatchToProps` function.

With this exercise, we'll be moving in this direction by moving to this structure, but we'll start with just a real implementation of `mapStateToProps` . We'll stub out `mapDispatchToProps` to keep the type checker happy, but will look at the dispatching part more closely in the next exercise.

1. Add a `selectedTags` tag set to the `State` type in `state/index.ts` . Add a default value with a tag selected (so we can see something show up when we're done!) Make sure your webpack is clean.

2 Add the following preamble to the top of your `snack-report/index.ts` :

```
type StateProps = Pick<SnackReportUIProps, "selectedTags">;
type DispatchProps = Pick<SnackReportUIProps, "onTagChange">;
type ReduxConnectedProps = StateProps & DispatchProps;
type GraphQLProps = Pick<SnackReportUIProps, "rows">;

type _check = AssertAssignable<
  SnackReportUIProps,
  ReduxConnectedProps & GraphQLProps
>;
```

We will use these types to guide the implementation of our redux hooks as well as update our graphql call.

3 Define a function `mapStateToProps(state: State.Type): StateProps`. It should type check and utilize `state.selectedTags`.

4 Define a function `mapDispatchToProps(dispatch: Dispatch<any>): DispatchProps`. It should type check, and can return `() => {}` for `onTagChange`. We will implement this later.

5 Define a `wireToRedux` function that will hook our component up to the redux store.

```
const wireToRedux = connect(mapStateToProps, mapDispatchToProps);
```

6 Our `graphql` component no longer needs to provide all of the props, so change the `props` function's type signature to `props(result): GraphQLProps` and fix type errors by deleting the properties you've just added to the redux helpers.

Additionally, our `graphql` component will now how an *input*: the output of our `wireToRedux` component. Change the second type argument to `ReduxConnectedProps` accordingly. The result should be:

```
const wireToApollo = graphql<
  SnackReportQuery,
  ReduxConnectedProps,
  SnackReportUIProps
>(require("client/graphql-queries/SnackReport.graphql"), {
  props(result): GraphQLProps {
      //...
  }
}
```

7 Update the `SnackReportPage` definition to read

```
export const SnackReportPage = wireToRedux(wireToApollo(SnackReportUI));`
```

Once you've gotten all of the types shaken out, you should have a selected checkbox in your SnackReport, as defined by the `selectedTags` you chose for your `DEFAULT` state.

Your tests should continue to pass, assuming your set of selected tags did not change.

02.4 Dispatcher

Next, let's get tag selection working.

1 Update your `onTagChange` to `console.log` a message, to help us get a test up and running.

2 Update your page test to check and uncheck a new tag. You may find this helper useful:

```
const checkboxFor = (tag: string) =>
  page
    .find("label")
    .filterWhere(l => l.text().includes(tag))
    .find("input");
```

You should see your log message show up before your test fails.

3 Using the `SetPopularityAction` as an example, create a `ChangeTagAction`. You'll need to add:

1. An entry in `ActionTypeKeys`.
2. A type `ChangeTagAction` with a `type`, `tag:string`, and `value:boolean`
3. An action builder function, like `setPopularity`.
4. Add `ChangeTagAction` to the list of `ActionTypes`

4 Add a new test to `root-reducer.test.ts` to show that `state.selectedTags` is updated when these actions are dispatched to test-drive the implementation of a new case in

`reducer/index.ts`. You'll likely want to use ES spread syntax.

5 Update your `onTagChange` to call `dispatch(Actions.changeTag(tag, value))`. This will actually wire up your new action to user interaction.

2.5 Lenses for better tag set abstraction

Our `TagSet` module does a lot to decouple most of our application from the underlying structure of how we represent a set of chosen tags, but it could do better.

In implementing your reducer, its tests, and other parts of the code, you've no doubt been making use of `has`, `add` and `remove` in order to map tag-set/tag pairs to booleans (is this tag selected?) and from tag/boolean pairs to updates to the tag set – the implementation of the change tag action reducer.

This mapping is essential to our application. We're modeling membership in a set of selected tags as boolean properties – checked/unchecked – in the UI. In essence, we're establishing a mapping in our model of the software between a set of selected tags and a set of boolean values.

If your implementation looks like mine, we're effectively distributing knowledge of this mapping throughout the application. We're using `has` in places to check for membership and using a combination of `add`, and `remove` to set the logical inclusion state of a tag based on the checked property of the tag.

What if we want to change the model? We're currently modeling the set of included tags as a set of checked checkboxes, but what if we change our mind and want the set to represent the tags which are not included? We'd have to go to various places of our app. Changing `has(tag)` to `!has(tag)` and swapping `add/remove` calls.

Instead, let's represent this mapping between tag sets and booleans in a first class way. We can do that by using lenses.

A lens is basically a `get` / `set` pair that you can use to update substructure of a value Ina functional way. You can think of it as a strongly typed, memory safe, functional pointer offset. It represents how to update a real or computed value within a structure.

Our mapping is a family of lenses - one for each tag we wish to treat as a boolean value.

In particular, we can add this function to our `tag-set.ts` module:

```
export const tagValue = (tag: string): Lens<Type, boolean> =>
  Lens.of<Type, boolean>({
    get: tagSet => has(tagSet, tag),
    set: (tagSet, value) => (value ? add(tagSet, tag) : remove(tagSet, tag))
  });
```

1 Incorporate a definition for `tagValue` into your codebase and use unit tests to document and explore the use of this function.

2 Create a lens in the state module for `selectedTags` using `from/prop`, llke the one for popularity mode. This simple lens lets you deal with the property via an abstraction.

3 Update your reducer to use both lenses. Use `update` and `set`

4 Try out an equivalent implementation by combining both lenses with `comp`. Keep whichever you prefer.

5 Once you're comfortable with it, update other uses of the `TagSet` module where you're conceptually relying on the same mapping.

2.6 Empty tags by default

Now that we can select tags ourselves, change our redux store to have an empty tag set by default.

Provide an `initState` function to your `mockProvider` to instrument your test as though the user had already selected whichever tags you removed from your default state.

Your tests should continue to pass without changing assertions.

3.1 Arguments for `topSnacks`

We'll now embark on a journey to the server, to allow our `topSnacks` query to take a tag set as an argument.

1 Add a `tags: [String]` argument to to `topSnacks` in `schema.graphql`. Check for a new `TopSnacksQueryArgs` type to show up in `schema-types.ts`

2 Change the second argument to the `topSnacks` resolver to the new argument type. Log the argument in the resolver.

3 Try passing a tagSet argument to topSnacks in GraphiQL. You should see them logged in the web pack tab.

4 Using graphiql, refactor your query to take a `tags` variable and test it interactively.

5 Update the SnackReport query to take/pass args to topSnacks. Look in `graphiql-types.ts` for a new `SnackReportQueryVariables`

6 Add and fill in the blanks the following function above `props` in your report's `graphql` call:

```
options(props) {
  const variables: SnackReportQueryVariables = {
    // BLANK ;-)
  };
  return { variables };
},
```

The `options` hook lets you control how the graphql query is processed. In our case we will use it to include our new `tags` variable. By explicitly declaring `variables` with our generated variables type, we will get typescript's help in providing data in the right shape.

At this point you should see new sets of tags logged out in webpack whenever you un/check a tag.

So we now have: Store ➡ query inputs ➡ Server. As the user interacts with the tag box, redux is updated, which causes apollo to requery the server.

3.2 Tag filtering page test

So we're seeing the page round-trip to the server, but it's not actually doing anything in the UI because the server doesn't do anything with tags yet.

But it *would*. Let's prove it to ourselves by updating our tag test in our snack report page test.

3.3 Filtering by tags

Let's bring it together! All that's left is to make the provided tags filter top snacks:

1 Add a new test to the `topSnacks` query test that shows it returns only snacks tagged with the provided tags

2 Test-drive the implementation of a `findWithTagsNamed(names: string[]): Promise<Snack[]>` function on your snack repository.

The query which returns snacks tagged with *any* of the provided tags is easier to write, but it would be better if this return snacks tagged with *all* of the provided tags. Feel free to do either.

I suggest looking into the `join` helpers from knex and doing a join per tag to implement the *all* filtering.