Advanced Actions Lab

In this lab we won't be adding any new capability, just refactoring and improving our code using some cool new best practices we learned in the last lecture.

Before we begin, let's prove a point.

1. Edit Register.js. Find "SET_EMAIL" in the changePerson method.
2. Change "SET_EMAIL" to "SET_EMAL" or any other typo you like.
3. Run and test by editing the email address. What happens?

   _____
4. Is this a problem? Yes! Is is a likely problem? Again, yes, because typos inside magic strings are a major problem in any language or framework. Let's see what we can do to fix this.

Making action type constants

5. Create a new file called action-types.js in the store folder.
6. Add SET_EMAIL action type:

```
export const SET_EMAIL = "SET_EMAIL";
```

7. Edit reducers.js. At the top add ...

```
import { SET_EMAIL } from './action-types';
```

8. Below that, change the reducer case from "SET_EMAIL" to SET_EMAIL (without the quotes).
9. Now let's do it in the UI. Edit Register.js. Add this to the top:

```
import SET_EMAIL from './store/action-types';
```

10. And in the changePerson method, remove the quotes around SET_EMAIL.
11. Run and test.

Do you see what is happening? First, in a good IDE, you'll be notified immediately that SET_EMAL is undefined, but even if your IDE doesn't catch it, the app won't even compile! You must type it correctly for it to compile.

Now, we could implement this practice for all of the action types, but we'd have one import for every single action type in every JavaScript file where they're being used. let's explore a different way.

Making an action type enumeration

12. Edit action-types.js.
13. Put all of the action types into a single exported enumeration. Do something like this:

```
export const actionTypes = {
  SET_EMAIL: "SET_EMAIL",
  SET_CELL: "SET_CELL",
  SET_NAME_FIRST: "SET_NAME_FIRST",
  ... And so on for whatever action types you want
};
```

14. Now go change the imports in reducers.js and Register.js to import actionTypes instead of SET_EMAIL.
15. While you're there, you'll notice this breaks because SET_EMAIL isn't defined, but actionTypes.SET_EMAIL definitely is. Change both of those mentions of SET_EMAIL to actionTypes.SET_EMAIL.

Notice one thing; when you typed "actionTypes." your IDE prompted you with all of the valid action types. No more having to memorize them! No more having to import all of them. Cool, right?

16. Go ahead and do this in reducers.js for all of the actionTypes in the case statements.

Action creators

17. Create a file in the store folder called actions.js
18. Write an action creator for the SET_EMAIL action. Maybe call it *setEmail*? It is a function that should receive in the new email address and return a single object with a type of "SET_EMAIL" (hint: use your action type enumeration or action type constant from the lab steps above) and a payload of the new email address.
19. export this action creator function.
20. Edit Register.js. import your action creator function from actionCreators.js

```
import { setEmail } from './store/actions';
```

21. Refactor the dispatch to call this setEmail function instead of using the hardcoded action object you had before. Kind of like this:

```
store.dispatch(setEmail(e.target.value));
```

Now how easy is that? No need for imports of actionTypes. No need to remember what properties are needed for each action. Less opportunity to mess it up because your IDE will prompt you with the input parameters for setEmail so you remember what it needs.

22. Implement action creators for setCell and whatever other actions you may have created so far.

Our action creators are fairly cool, but we shouldn't settle for *fairly* cool. Let's go further to make it *super* cool with an action creator enumeration.

Using an action creator enumeration

23. Edit actions.js. Remove all of the export keywords so that nothing is exported yet.
24. At the bottom, add something like this:

```
export const actions = {
  setEmail, setCell,
};
```

25. Now go back to Register.js and change the import to a single import of *actions*.
26. In the changePerson method, change the dispatch for setting email to use actions.setEmail(e.target.value) instead of just setEmail(e.target.value)
27. Run and test, making sure it still works.

The payoff!

Now let's see why we'd do this. Please implement the entire change for setting the last name...

28. In actionTypes.js, add a new type for "SET_NAME_LAST".
29. in actions, add a new creator for setNameLast(lastName). Don't forget to add it to the actionCreators enumeration.
30. In store/reducers.js, start to add the reducer case for actionTypes.SET_NAME_LAST. Note that when you type in "actionTypes.", a good IDE will see that there is an entry for SET_NAME_LAST so you don't have to remember how you typed it. No opportunity for typos!
31. Go ahead and finish implementing that reducer case. (hint: copy/paste SET_NAME_FIRST and modify it).
32. Back in Register.js's changePerson method, find where we're setting the last name. Again, start typing "store.dispatch(actions." and in a good IDE, intellisense will prompt you with the "setNameLast" method and will remind you that you need to provide the new last name you're setting it to. Again, no opportunity for typos and no having to remember what you or someone else called the type!

Once you've got your new lastname action updating properly you can be finished. Enjoy a rest!