CSE 331: Software Design & Implementation

Homework 8 (due Wednesday, May 22nd at 11:00 PM)

In this assignment, you will implement the client and server portions of an application that allows users to edit images made up of squares and to save them to and load them from a server. This assignment consists entirely of coding work. Submit your final version to "HW8 Coding". Turning in your work for this assignment is a little trickier than usual, so follow these steps carefully!:

- `cd` into the directory that contains the `/client` and `/server` directories.
- Delete the `node_modules` directories from each directory (you can do this manually or use the command: `rm -r client/node_modules && rm -r server/node_modules`).
- Generate a zipped file containing both of these folders On Mac you can run: `zip -r submission.zip client/ server/`. On Windows you can select both folders, right click, and select 'Send to' > 'Compressed (zipped) folder'.
- Go to Gradescope and select the zipped file that was created by running the last command to turn in.
- Make sure you get all the autograder points! If you decide to work on your app some more after turning it in, you'll need to run `npm install --no-audit` in both of the directories again to get your `node_modules` back.

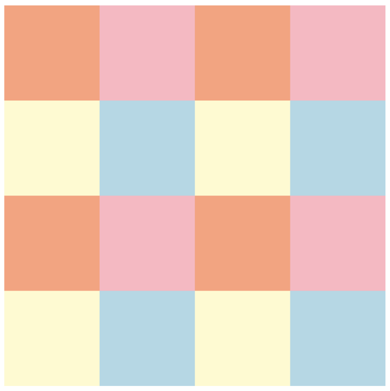
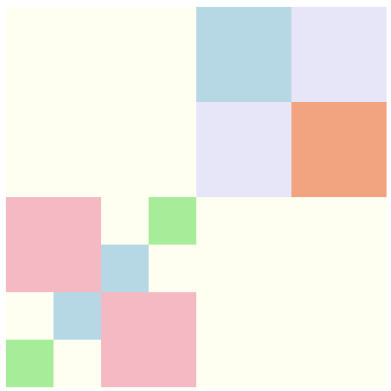
Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-24sp/cse331-24sp-materials/hw-squares.git
```

Install the node modules, with `npm install --no-audit`, in both the `client` and `server` directories.

Check out the starter code and how the app looks currently. Get a sense of what different portions of the code are doing and compare to the fully functional section and lecture examples if that's useful. Start both the `client` and `server` by running `npm run start` in both directories (you'll need to open a terminal for each of the two directories to keep everything organized). Then navigate to where the client is running `http://localhost:8080/`. Notice the square on the screen, and try clicking around.

Right now there's not much functionality, but later, the app will allow users to create and save designs of colored squares. This video **includes a walk-through of what the final product looks like**. (Your app needs to work like the app in the video but it doesn't need to look exactly the same.) Here are some example designs:

It is difficult to try to get all the pieces working at once. Instead, you should write one piece at a time, testing it individually to make sure it works. Once the main pieces work individually, you can put them together.

Note that you can only use `let` variable declarations on the `server` side, not the `/client` side (no mutation in `square.ts`).

Squares

The next problem makes use of a few inductive types. First, a Color is one of the following:

$$\textbf{type}\ \mathsf{Color} \ := \ \mathsf{red} \mid \mathsf{orange} \mid \mathsf{yellow} \mid \mathsf{green} \mid \mathsf{blue} \mid \mathsf{purple} \mid \mathsf{white}$$

Then, a Square is a tree, where each node is either a "solid" (leaf) node of a single color or a "split" (internal) node that breaks the square into four quadrants, each of which can be any square:

$$\begin{aligned} \textbf{type}\ \mathsf{Square} \ &:= \ \mathsf{solid}(\mathsf{color} : \mathsf{Color}) \\ &\mid \ \mathsf{split}(\mathsf{nw} : \mathsf{Square},\ \mathsf{ne} : \mathsf{Square},\ \mathsf{sw} : \mathsf{Square},\ \mathsf{se} : \mathsf{Square}) \end{aligned}$$

We will also need a way to refer to one of the children of a split square. We can do so as follows:

$$\textbf{type}\ \mathsf{Dir} \ := \ \mathsf{NW} \mid \mathsf{NE} \mid \mathsf{SW} \mid \mathsf{SE}$$

With that, we define a path to be a list of directions, describing how to get to the node, starting from the root:

$$\textbf{type}\ \mathsf{Path} \ := \ \mathsf{List}\langle \mathsf{Dir} \rangle$$

These types are already defined for you in `square.ts`. That file also includes functions `toJson` and `fromJson` that convert between `Square` and JSON, which we represent as "`unknown`" in TypeScript.

1. Couldn't Square Less (14 points)

Implement the following two operations on Squares in `square.ts`. These will be helper functions for parts of the UI we create in the next question.

1. Given a square and a path, retrieve the root of the subtree at that location (assuming it exists).
2. Given a square, a path, and a second square, return a new square that is the same as the first one except that the subtree whose root is at the given path is replaced by the second square.

Make sure you document and test these operations before moving to the next problem. Again, debugging other parts of the client will be easier if you can be confident that any errors you see are due to bugs in that code and not these operations.

Hint: You may find question 1 from this week's solution worksheet helpful.

Drawing Squares

The file `square_draw.tsx` includes a `SquareElem` tag that can display a square. (The code is just a recursive translation from one tree, `Square`, to another, `JSX.Element`. However, arranging these so they look right on the page is a little tricky, so we provided this for you.)

You can also tell `SquareElem` to "select" a specific solid square, by giving the path to it. That causes the square to display in a slightly different color. Solid squares also change color when the mouse hovers over them. Lastly, `SquareElem` allows you to provide a callback function in the `onClick` property, which will be called when the user clicks on any solid square, providing you a path to the one that was clicked on.

The provided code always displays a single split square, with four solid square children. When the user clicks on any of the squares, it tells them to stop that. You can get rid of this code, we'll be enhancing the app to take advantage of these properties of `SquareElem`.

2. Rage Split (16 points)

In `FileEditor.tsx` there is a mostly empty React component `FileEditor`. Instead of rendering a `SquareElem` in `App`, as it currently does, change `App` to render a `FileEditor` instead, then the `FileEditor` will render the `SquareElem`. This will allow us to place our design editing functionalities in `FileEditor` and have separate functionalities in the `App`.

When a square element is selected, the `FileEditor` will allow the user to perform the following operations to edit the square and create a design:

1. Change to a different solid color.
2. Split that square into four parts (initially, all the same solid color).
3. Merge the square with its siblings, i.e., replace its parent with a single solid square of that color.

The starter code for `FileEditor` includes some TODOs as guidance, but you should read through the following notes carefully before starting and revisit them later also:

- The first thing to think about is what state you need to keep track of in order to implement these operations. We include state in the starter that we think will be sufficient, so check that out and make sure you understand their purpose.
- You can use any HTML you want to let the user invoke these operations, but one simple choice would be BUTTON (for split & merge) and SELECT (for change color). **Note that the `SquareElem` callback `onClick` already returns the path to the selected square**.
- When any operation wants to change the square that is displayed, calculate the new `Square` with the changes, and then call `setState` with the new `Square`. That will cause React to invoke `render` again and display the new UI. Remember the `Square` operations in `square.ts` that you implemented in problem 1.
- While it is never a bad thing to write unit tests, for UI like this, you really need to see it in the browser to know that it is really working. For that reason, perform manual testing; however, you should follow the usual rules for deciding which cases to try manually. (To be clear, we will not expect you to turn in any unit tests for your UI code.)

3. Pick-or-Treat (16 points)

Update the app to have a starting screen that asks the user to type in a name for their square design before opening the page to edit the squares. This text box should be part of the `FilePicker` component.

Since we have a picking view and an editor view, the `App` component will need to conditionally render these components based on which page we should currently display.

Then, add a "Save" button, in the `FileEditor`, that causes the square design they have created to be saved under the name they typed in.

It is a cleaner design to have all the file management code in `App`, so we recommend having the `App` pass an `onSave` callback to the `FileEditor` component. When this is invoked, the `App` store the current state of the edited design and the name that the user specified in an `AssocList` (as we used in HW7) in the state.

Add a "Back" button as well, in the `FileEditor` component, that returns to the picking screen. That should also be a callback.

Run the app and save designs under different file names to test this functionality. It may be useful to use `console.log` to print out the state of your app so you can see if the saved files are properly added.

4. Service With a File (16 points)

Change the `FilePicker` component to show the user the names of all the existing design files. Clicking on any of them should open that file in its last-saved state in the `FileEditor` and allow them to continue editing it. The functionality to pick a name and create a new design from scratch should still be there as well.

To get the list of file names, we need to access the keys in an `AssocList`. For this, we have provided `get_keys` in `assoc.ts` for you to use which is a translation in TypeScript of the function defined as follows:

```
func get-keys(nil)                 := nil
     get-keys(cons((y, v), L))     := cons(y, remove(y, get-keys(L)))     for any $y, v : \mathbb{S}^*$ and $L$ : List
```

`AssocList`s are able to store duplicate keys, though for our purposes, we want more recently saved designs to overwrite previous designs with the same name. When we use `get_value` to get the design with a certain name, it'll have the correct behavior and get us the most recently saved design (closest to the front of the list) with that name, but when we display the full list of names to the user, we must actually remove any duplicates.

We recommend that you store the names as an `Array` within your `App` and pass that to your `FilePicker` component. If you do, it'll make later stages of the assignment simpler. There is a function in `list.ts` that you can use to convert the result of `get_keys` to an `Array`.

You can use any HTML you want to display the existing file names to the user, but one simple choice would be `<ul>` / `<li>` (unordered list / list items within it), with each containing an `<a>` (link) with the name of that file. The `onClick` event of a link will be called when they click on it. (You can set the `href` tag as `href=‘‘#’’` so that the link itself does nothing, but it appears clickable to indicate to the user that they can.)

At this point you have completed a fully functional client side react app! You should be able to create designs, save them, see a list of all designs, and open previous designs and edit them. However, you may notice that if you reload the page, all your designs are lost! In the next parts we will fix this.

5. The Birds and the Keys (8 points)

To keep our app state persistent across page loads and different tabs/windows, we will create a server to store our designs. Before we can create our server routes, we need a data structure to store the names and content of files. For this, we will use our mutable map interfaces that we created in HW7.

(a) Copy the contents of your mutable map **interface** from `map.ts` that you created in HW7 into the respective file in `/server/src`. (Or take it from the files you turned in on Gradescope if you removed your work from your device).

There's no need to change your interface at this point unless you received feedback for HW7 that you should make improvements.

(b) In your client side implementation of the app so far, you used the `get_keys` function for `AssocList` to get the list of all the created files. In our server, we also need a way to access all the keys in a map. Add a method to your interface, with JSDoc specification, for a function that returns an `Array<string>` containing the keys from all the (key, value) pairs in the map.

(c) Create a new mutable map **class** that implements your map interface.

Similar to Java, TypeScript has a built-in `Map` data structure which we will use as our concrete representation in this new class instead of the `AssocList` we used last week. In the constructor, create an empty `Map` to store in a field. Remember to write an Abstraction Function.

Use the methods of `Map` to implement the methods of your class. You will need to read some documentation to learn more about the methods available. Hint: it may be useful to look into `has()`, `set()`, `get()`, and `keys()`, as well as `Array.from()`.

(d) Create a factory function that has a return type of your mutable map interface and returns an instance of the class you implement in part (c). Write a JSDoc specification for the function.

(e) Copy the tests for your mutable map from `map_test.ts` that you wrote in HW7. Update them to call the factory function you created in the last part instead of the function from HW7. Otherwise, the behavior of the methods from your HW7 map should not have changed, as they both implement the same interface, so your existing tests should be sufficient; however, if you received feedback in HW7 that they were incomplete/incorrect, you should improve them now.

Write tests for the new method that returns the keys of the map.

6. One Foot in the Save (16 points)

Implement the server portion of the application by adding routes to perform the following operations:

1. `/save`, saves the "content" of a file under a given "name".
2. `/load`, loads the last-saved content of a file with a given "name".
3. `/names`, lists the names of all files currently saved.

(Note you should use the exact route and param names specified; param names given in "".)

The server should allow the file content to be any valid JSON, which we represent in TypeScript as the "`unknown`" type. The lack of type information should not be a problem as it should not be necessary to examine the content of the file. We simply store it in save, and return it in load.

For `/save`, we will use a POST request so we can accept the file content through the **body** of the request rather than the query params. It is sufficient for the `/load` and `/names` routes to be GET operations.

Properly test all of these operations before moving to the next problem. Debugging will be easier if you can be confident that any errors you see are due to how you're accessing the server and not in the server itself. Additionally, remember to incorporate all appropriate error checking.

Some additional notes:

- Don't worry too much about the idea of a "file" if that's confusing. Like in the client-only app implementation these "files" will just be square designs but formatted to be passed easily between the client and server.
- To implement these operations, use an instance of your mutable map that you updated in the previous problem, similar to how we stored transcripts using the ADT in HW7.
- The provided code just has a dummy route to remind you what the code looks like for creating and testing routes. Feel free to delete it.
- For debugging purposes, it may be useful to also have a "resetForTesting" function that you can call at the end of each test to remove any files you saved during the test. (Not required, but encouraged. See HW7 for an example.)

7. Awaken a Sleeping Client (14 points)

Now that we have a working server, we can use it in our client app.

Currently, certain UI actions in your components trigger callbacks to the `App` component which cause the `App` to save/load designs with a local `AssocList` state. In this step we will fully replace this and instead store data on the server. There should be no need to edit any components besides the `App` in this step; we'll let it be the solely responsible component for all server communication.

In `/client/server.ts` we provided functions that access each of the routes described in the last question. We provided these for you because we have not yet covered the steps to access the server from the client in lecture. Often (as you'll see in future lecture examples and HW9), we place functions to access the server in `App` or other components directly.

These server access functions accept any data that needs to be passed to the server as parameters. They also accept a callback function to be called once a response is received from the server and parsed. Read the documentation for the exported functions at the top of `server.ts` to see how to call them. These functions also have lots of helpers themselves later in the file; you don't need to look at those helpers nor any of the actual implementation (though you can if you're curious). You should **not** edit these provided functions.

In `App`, write functions (or edit existing functions) that call the `server.ts` functions to make a request to the server to save/load/list names when needed. Also, write callback functions to pass in that can handle the data that the server sends back and trigger updates to the state and child component's state accordingly.

Some additional notes:

- It's essential that, in your server, your routes and expected parameters are named exactly as described in the last question. Otherwise, in this client-server connection step, our provided `server.ts` helper functions will not work.
- Your original `AssocList` for storing designs should be totally removed from your client when this step is complete.
- While a server access is in progress (after calling a `server.ts` function and before a response is received from the server and passed to your callback), your app should be in a "loading" state while it is waiting for the needed data. See the week 8 section code for an example of how this may look.

Congratulations again! You now have a fully functional squares application with server and client components!

8. Extra Credit: Picked the Wrong Week to Stop Sniffing New (0 points)

Add any new features that seem useful! You will get points for any feature that works correctly and seems like it would be valuable to the user.