CSE 331: Software Design & Implementation

Homework 8 (due Friday, May 26th 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 in vscode or use the command: `rm -r client/node_modules && rm -r server/node_modules`).
- Run the command: `zip -r submission.zip client/ server/` to generate a zipped file containing both of those folders.
- Go to Gradescope and select the `submission.zip` file that was created by running the last command.
- Make sure you get all the autograder points! If you have an issue and need to try something else you'll likely need to run `npm install --no-audit` in both of the directories again to get your `node_modules` back.

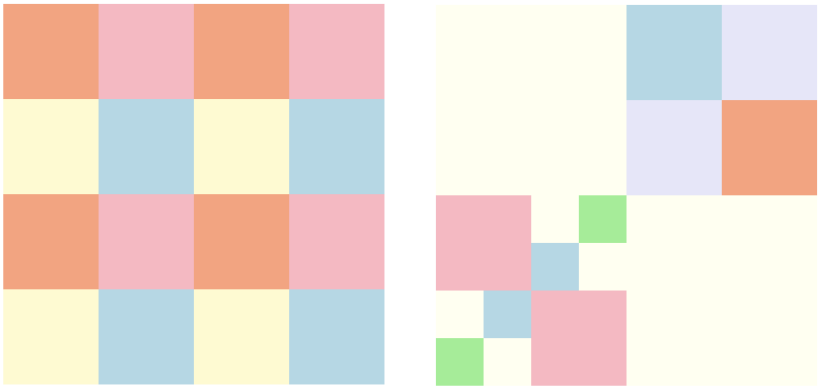
Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-23sp-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 other 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.

1. One Foot in the Save (20 points)

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

1. Save the (string) contents of a file with a given name.
2. Load the last-saved contents of a file with a given name.
3. List the names of all files currently saved.

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

Some additional notes:

- Don't worry too much about the idea of a "file" if that's confusing. You can think of it as some data stored with a label name. Specifically the contents will be a string. So this is essentially a way to store and lookup string data.
- To implement these operations, you will need a data structure for storing the contents of the files. Think about the options discussed in lecture about appropriate data structures for different designs such as `Array`, `List`, `Map`, and decide which makes sense.
- 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 "reset" function that you can call at the end of each test to remove any files you saved during the test. (Not required, but encouraged.)
- It is sufficiecnt for the Load and List operations to be GET operations since they are a simple request that just returns data, but the Save operation requires passing in the string file contents, so we will need to use a POST request rather than GET so we can accept this contents through the body of the request.

Squares

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

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

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:

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

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

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

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

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

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 uses the "`any`" type in TypeScript.

2. Couldn't Square Less (20 points)

Implement the following two operations on Squares in `squares.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 a problem from the section 8 worksheet helpful here.

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`.

3. Rage Split (20 points)

In `editor.tsx` there is a mostly empty React component `Editor`. Instead of rendering a `SquareElem` in `App`, as it currently does, change `App` to render an `Editor` instead, and the `Editor` will render the `SquareElem`. This will allow us to add some design editing functionalities in `Editor` and other functionalities in the `App`

When a square element is selected, the `Editor` 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 `Editor` 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.[1] A good choice, in this case, would be to store the root of the current square and the path to the selected solid square, if any.
- 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` related functions in `squares.ts`.
- 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.)

[1] The same was true when we wrote server-side code in the first problem: state first, code second.

Before getting started on the next two questions, review the week 8 section code for how to make server requests and update the state upon their completion. In the section example, requests are made in `client/App.tsx` and the `server` directory contains the routes being made.

Remember the work we already did in previous questions, there's no need to re-implement some functionality in our client that our server already provides for us.

Make sure that you include proper error handling related to making requests.

4. Pick-or-Treat (20 points)

Change the `App` component to have a starting screen that asks the user to type in a name for their square design before opening the page to to edit the squares.

Then, add a "Save" button, in the editor, 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 I recommend having the `App` pass an `onSave` callback to the editor component. When this is invoked, it should send the server a file (the current state of the square design with the name being the one the user specified).

The server expects the file contents to be a string. You can call `toJson` (from `square.ts`) to turn a `Square` into JSON and then call `JSON.stringify` to turn that into a string.

To make a POST fetch request, a second argument to the fetch function is required to specify the type of request and include any data to send through the body of the request. This argument should be in the form:

`{method: 'POST', body: body, headers: {'Content-Type': 'application/json'}}`

where "body" can be replaced with the variable holding the data you want to send.

While you're at it, it would be useful to add a "Back" button as well, in the editor component, that goes back to the starting screen. That should also be a callback.

Run the app a few times and save designs under different file names to test this functionality.

5. Service With a File (20 points)

Change the `App` component to show the user the names of all the existing files. Clicking on any of them should open that file in its last-saved state 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.

An additional note:

- 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.)

6. 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.