Analysis Report

Client-Server Architecture Analysis of Spreadsheet Application

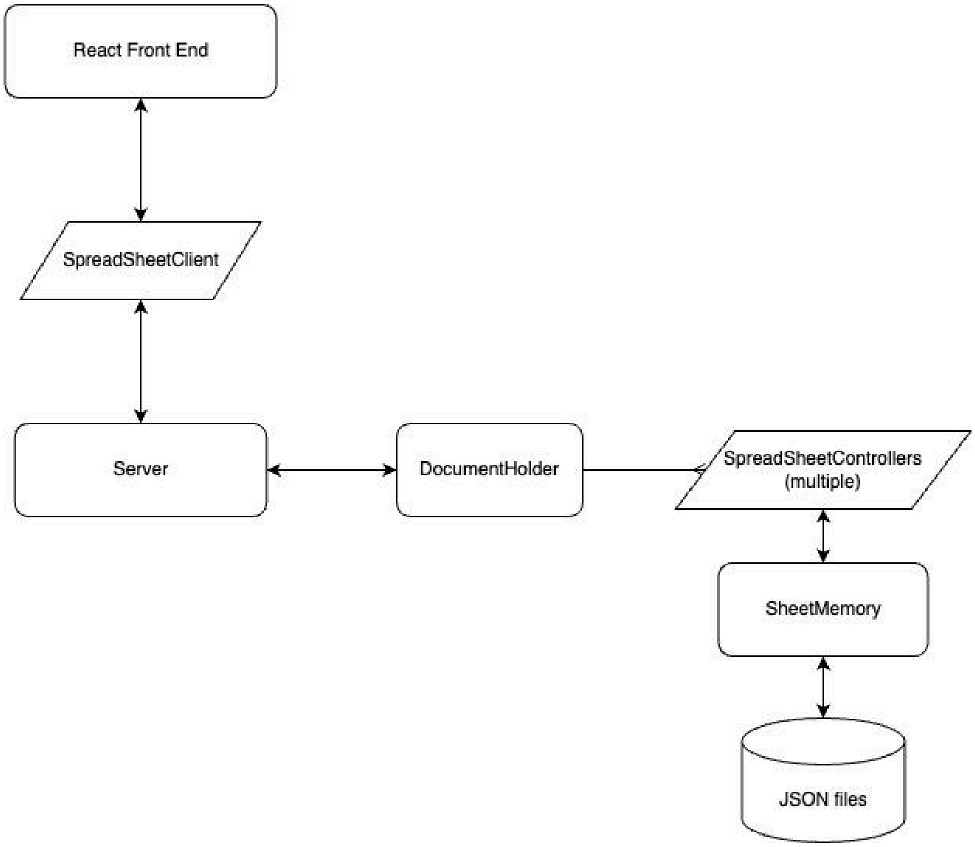
Overview

This analysis focuses on understanding the client-server architecture of a spreadsheet application using a NodeJS backend and a React TypeScript frontend. The report covers the design patterns implemented in the project, the interaction between the client and server through RESTful APIs, user management, and the multi-screen user interface (UI) on the frontend.

Part 1: Understanding the Project Structure

Repository Structure

The project follows a **Model-View-Controller (MVC)** architecture, consisting of:



1. **View (Frontend - React TypeScript)**:
    - The **View** handles UI rendering and user input, and communicates with the backend via the `SpreadSheetClient`.
    - Key components include `Button.tsx`, `SheetComponent.tsx`, `LoginPageComponent.tsx`, and `SpreadSheet.tsx`. These components render the interface and allow users to interact with the spreadsheet.
    - The `SpreadSheetClient` acts as a bridge between the **View** and the **Controller**, sending API requests to update and fetch spreadsheet data.
2. **Controller (SpreadSheetClient and Backend Controller)**:
    - The **Controller** is divided into two layers:
        - The frontend `SpreadSheetClient`, which forwards user actions (e.g., cell edits, formula changes) to the backend.
        - The backend `SpreadSheetController.ts`, which processes these actions, applies business logic, and updates the **Model** (spreadsheet data).
    - It ensures that user permissions and actions are correctly enforced and that updates are propagated between the frontend and the **Model**.
3. **Model (Backend - Node.js)**:
    - The **Model** consists of the underlying spreadsheet data and logic. Key files include:
        - **`SheetMemory.ts`**: Manages the in-memory data for the spreadsheet.
        - **`Cell.ts`**: Represents individual spreadsheet cells, storing formulas, values, and dependencies.
        - **`DocumentHolder.ts`**: Manages multiple spreadsheet instances and provides persistence by saving data to and loading data from JSON files.
    - The **Model** ensures data consistency and handles formula evaluation, cell value calculations, and data persistence.

Part 2: Exploring Design Patterns

1. Frontend Design Patterns

- **Component Composition and Reusability**:
  - React components are modular and reusable, promoting a clean and maintainable codebase.
  - **Example**: `SheetComponent` is used to render a grid of cells, with each cell represented as a `button` element.
- **State Management with Hooks**:
  - React hooks such as `useState` and `useEffect` are used extensively to manage state and handle side effects.
  - **Example**: In `SpreadSheet.tsx`, various state variables like `formulaString` and `statusString` are managed using `useState`.
- **Controlled Components**:
  - The frontend uses controlled components to manage form inputs and user interactions.
  - **Example**: `LoginPageComponent` uses controlled inputs for managing user login.

2. Backend Design Patterns

- **Singleton Pattern**:
  - Backend components like `SpreadSheetController` manage application-wide data, ensuring a single source of truth.
  - **Example**: The `SpreadSheetController` class manages all spreadsheet-related operations, such as adding tokens, cells, and managing user access.
- **Model-View-Controller (MVC)**:
  - The backend loosely follows the MVC pattern, with `SpreadSheetController` acting as the controller and `SheetMemory` as the model.

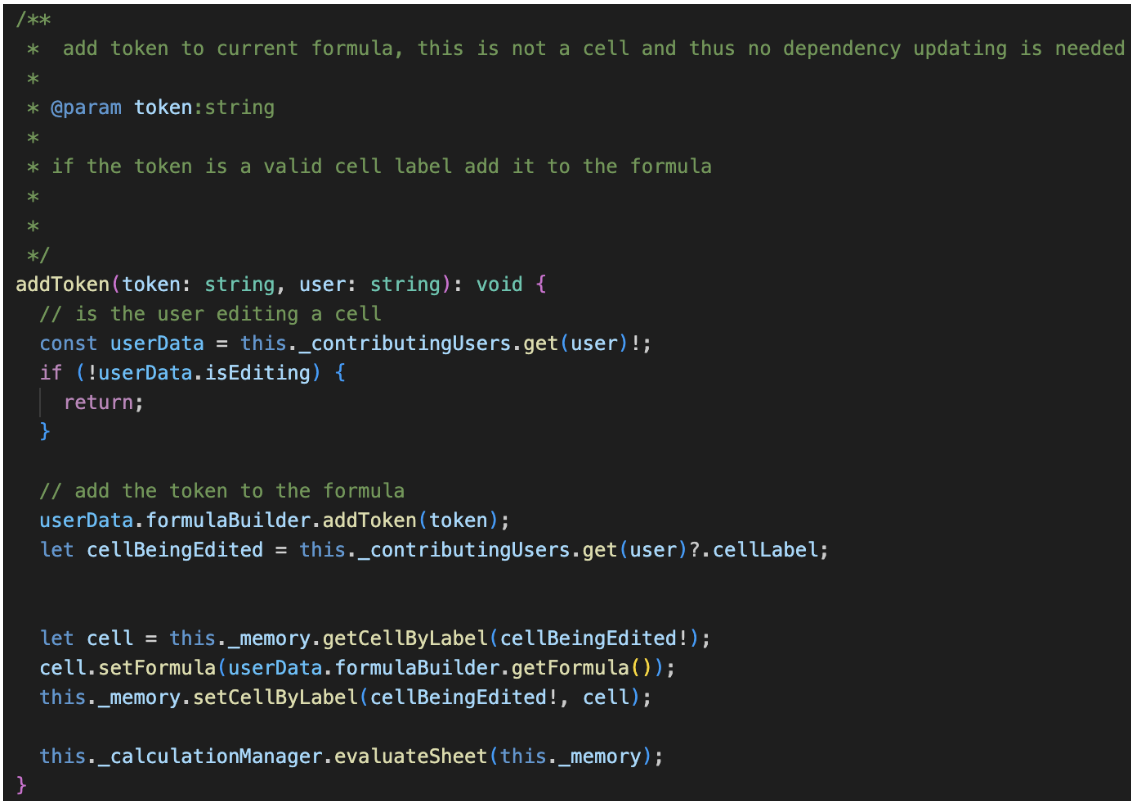
- **Example**: `SpreadSheetController` handles user requests and updates the `SheetMemory` model accordingly.

- **Observer Pattern (Polling Mechanism)**:
    - The frontend uses a polling mechanism to periodically fetch updates from the backend, mimicking an observer pattern.
    - **Example**: `useEffect` in `SpreadSheet.tsx` sends requests to the backend every 50 milliseconds to check for updates.

Part 3: Analyzing the Backend (NodeJS)

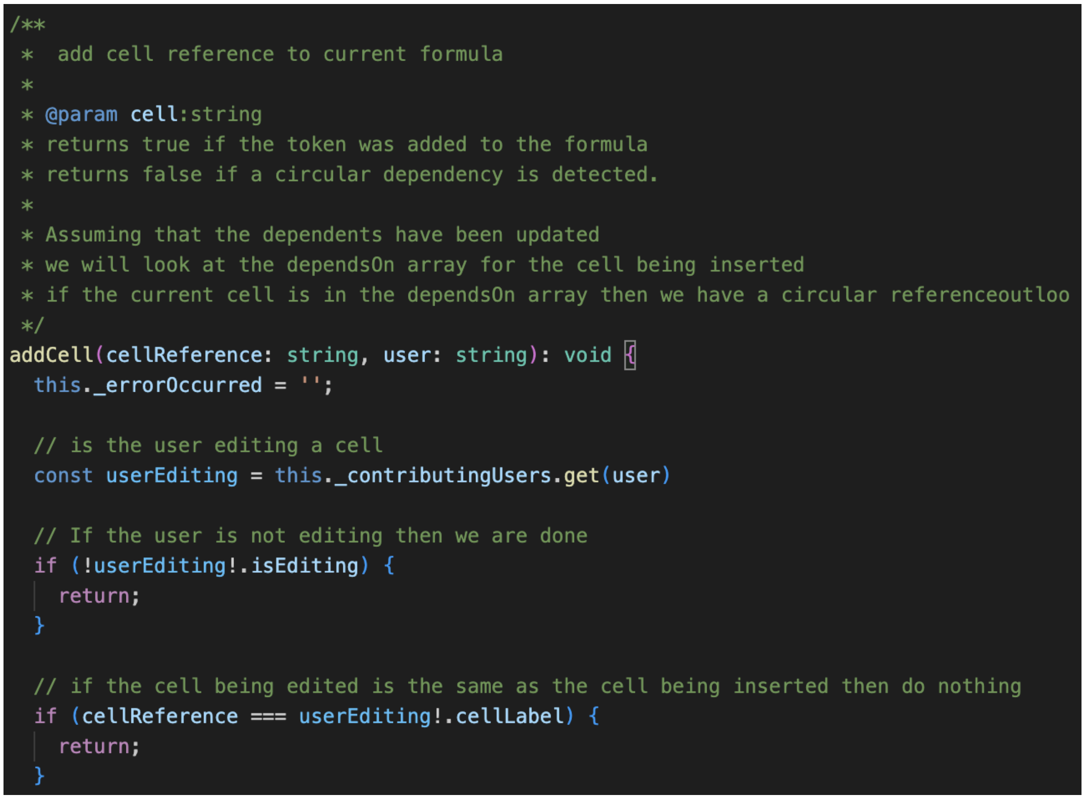
1. RESTful API

- **CRUD Operations**:
    - The backend supports CRUD operations through various endpoints. Examples include adding tokens to a cell, viewing a document, and updating cell data.
    - **Example**: The `addToken` method in `SpreadSheetController.ts` allows users to add tokens to the current formula.

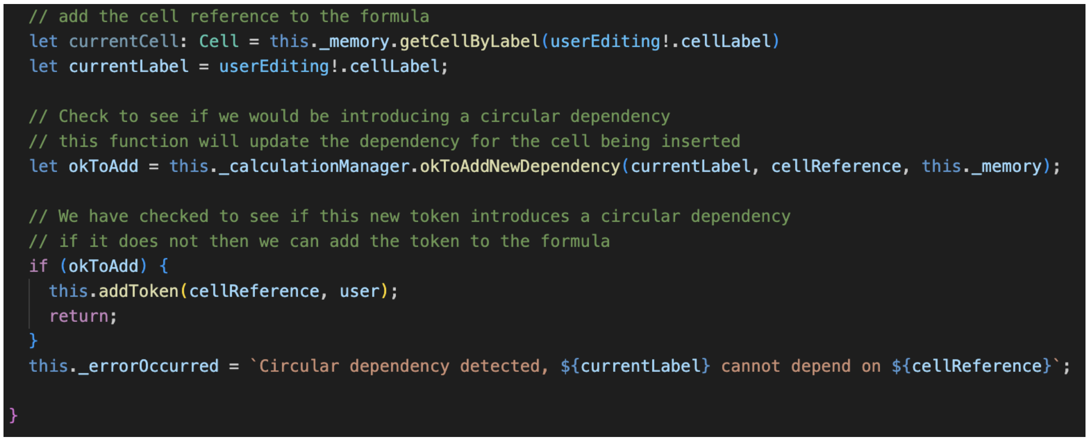


```
/**
 *  add token to current formula, this is not a cell and thus no dependency updating is needed
 *
 * @param token:string
 *
 * if the token is a valid cell label add it to the formula
 *
 *
 */
addToken(token: string, user: string): void {
  // is the user editing a cell
  const userData = this._contributingUsers.get(user)!;
  if (!userData.isEditing) {
    return;
  }

  // add the token to the formula
  userData.formulaBuilder.addToken(token);
  let cellBeingEdited = this._contributingUsers.get(user)?.cellLabel;


  let cell = this._memory.getCellByLabel(cellBeingEdited!);
  cell.setFormula(userData.formulaBuilder.getFormula());
  this._memory.setCellByLabel(cellBeingEdited!, cell);

  this._calculationManager.evaluateSheet(this._memory);
}
```

- **Data Validation and Error Handling**:
    - The backend validates user inputs and manages errors such as circular dependencies between cells.
    - **Example**: The `addCell` method checks for circular dependencies before adding a cell reference to the formula.



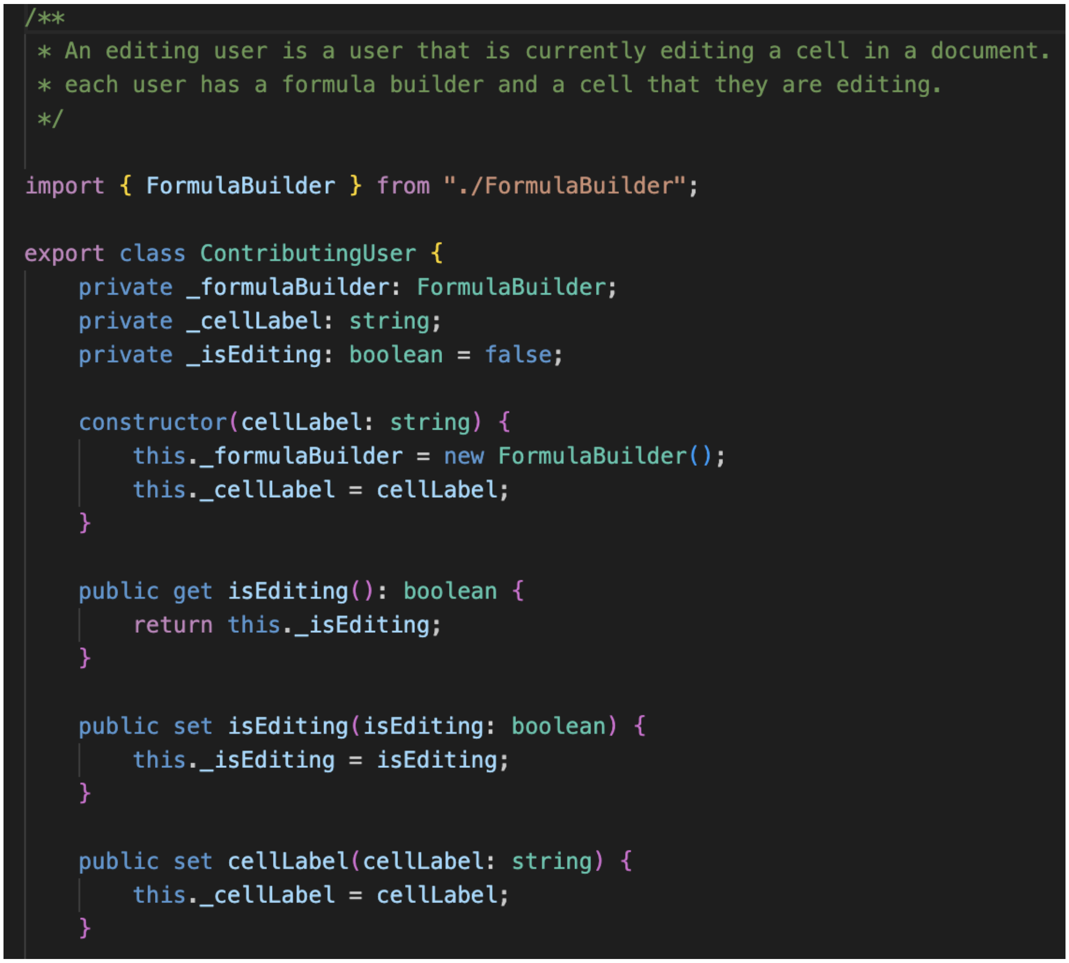
```
/**
 *  add cell reference to current formula
 *
 * @param cell:string
 * returns true if the token was added to the formula
 * returns false if a circular dependency is detected.
 *
 * Assuming that the dependents have been updated
 * we will look at the dependsOn array for the cell being inserted
 * if the current cell is in the dependsOn array then we have a circular referenceoutloo
 */
addCell(cellReference: string, user: string): void {
  this._errorOccurred = '';

  // is the user editing a cell
  const userEditing = this._contributingUsers.get(user)

  // If the user is not editing then we are done
  if (!userEditing!.isEditing) {
    return;
  }

  // if the cell being edited is the same as the cell being inserted then do nothing
  if (cellReference === userEditing!.cellLabel) {
    return;
  }
```



```
  // add the cell reference to the formula
  let currentCell: Cell = this._memory.getCellByLabel(userEditing!.cellLabel)
  let currentLabel = userEditing!.cellLabel;

  // Check to see if we would be introducing a circular dependency
  // this function will update the dependency for the cell being inserted
  let okToAdd = this._calculationManager.okToAddNewDependency(currentLabel, cellReference, this._memory);

  // We have checked to see if this new token introduces a circular dependency
  // if it does not then we can add the token to the formula
  if (okToAdd) {
    this.addToken(cellReference, user);
    return;
  }
  this._errorOccurred = `Circular dependency detected, ${currentLabel} cannot depend on ${cellReference}`;

}
```

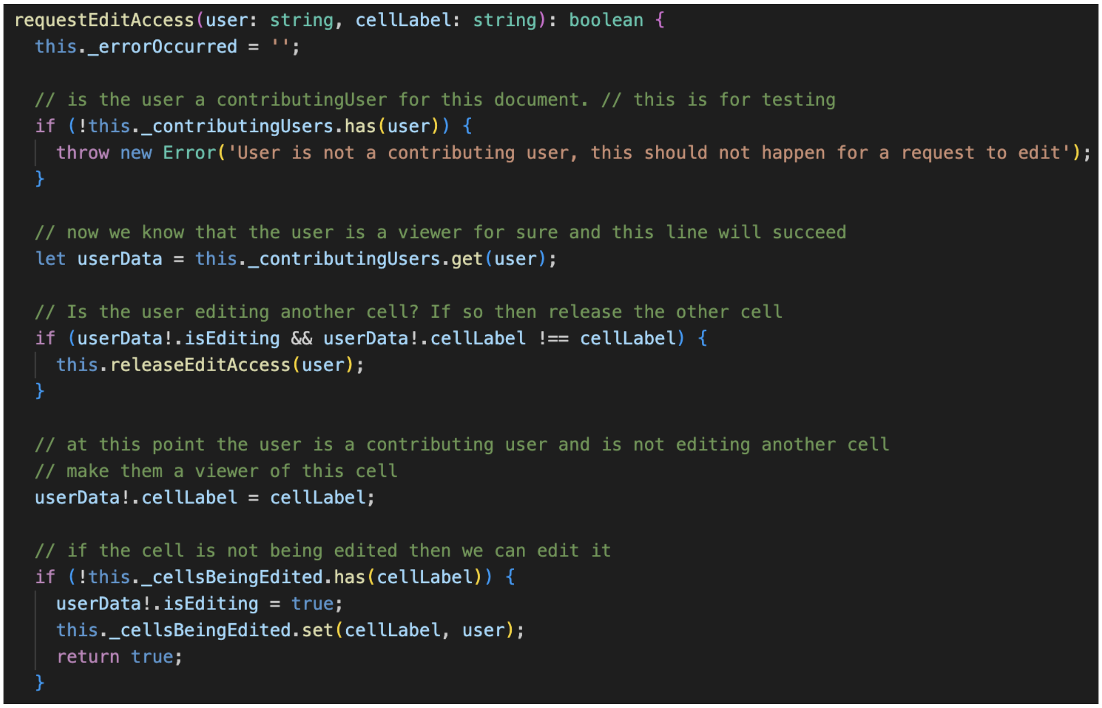
2. User Management

- **User Authentication and Access Control**:
    - User sessions are managed using browser storage, but there are no advanced authentication mechanisms like JWT or OAuth.
    - **Example**: The `ContributingUser` class in the backend manages which user is editing which cell, ensuring exclusive editing rights.

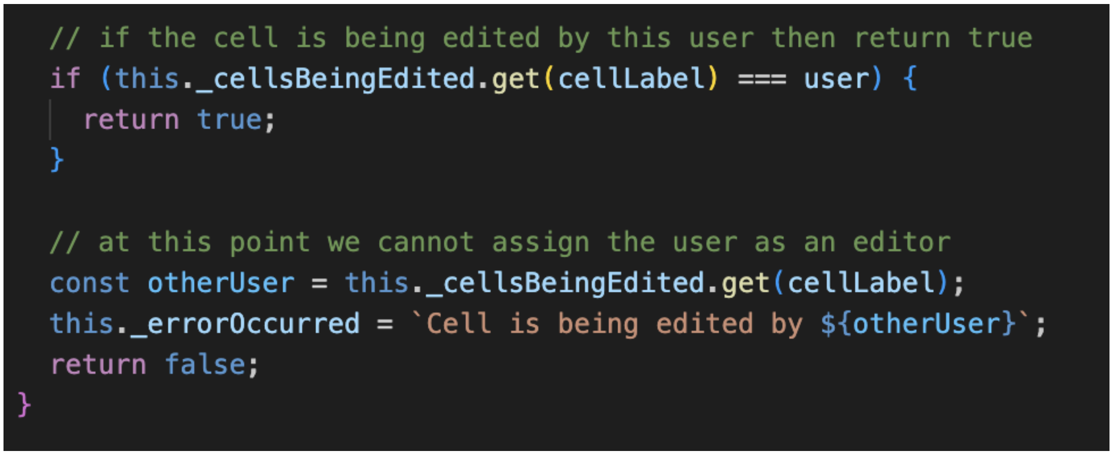


```
/**
 * An editing user is a user that is currently editing a cell in a document.
 * each user has a formula builder and a cell that they are editing.
 */

import { FormulaBuilder } from "./FormulaBuilder";

export class ContributingUser {
    private _formulaBuilder: FormulaBuilder;
    private _cellLabel: string;
    private _isEditing: boolean = false;

    constructor(cellLabel: string) {
        this._formulaBuilder = new FormulaBuilder();
        this._cellLabel = cellLabel;
    }

    public get isEditing(): boolean {
        return this._isEditing;
    }

    public set isEditing(isEditing: boolean) {
        this._isEditing = isEditing;
    }

    public set cellLabel(cellLabel: string) {
        this._cellLabel = cellLabel;
    }
```

- **User Roles and Permissions**:
    - The backend ensures that only one user can edit a specific cell at a time. It manages permissions based on user roles such as editor and viewer. Data security is managed mainly via validation checks. No encryption or password management is indicated.
    - **Example**: The `requestEditAccess` method checks if a cell is being edited by another user and prevents concurrent edits.



```
requestEditAccess(user: string, cellLabel: string): boolean {
  this._errorOccurred = '';

  // is the user a contributingUser for this document. // this is for testing
  if (!this._contributingUsers.has(user)) {
    throw new Error('User is not a contributing user, this should not happen for a request to edit');
  }

  // now we know that the user is a viewer for sure and this line will succeed
  let userData = this._contributingUsers.get(user);

  // Is the user editing another cell? If so then release the other cell
  if (userData!.isEditing && userData!.cellLabel !== cellLabel) {
    this.releaseEditAccess(user);
  }

  // at this point the user is a contributing user and is not editing another cell
  // make them a viewer of this cell
  userData!.cellLabel = cellLabel;

  // if the cell is not being edited then we can edit it
  if (!this._cellsBeingEdited.has(cellLabel)) {
    userData!.isEditing = true;
    this._cellsBeingEdited.set(cellLabel, user);
    return true;
  }
```

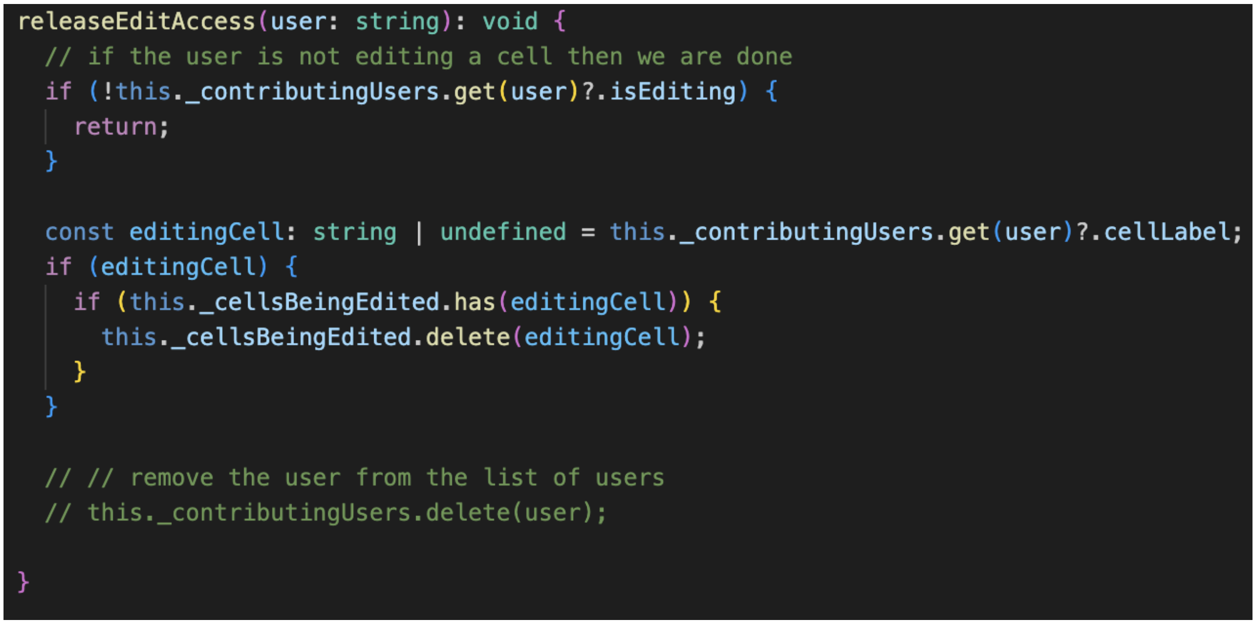


```
  // if the cell is being edited by this user then return true
  if (this._cellsBeingEdited.get(cellLabel) === user) {
    return true;
  }

  // at this point we cannot assign the user as an editor
  const otherUser = this._cellsBeingEdited.get(cellLabel);
  this._errorOccurred = `Cell is being edited by ${otherUser}`;
  return false;
}
```

3. Middleware and Error Handling

- The backend appears to use manual error handling within the `SpreadSheetController`, where methods like `addCell` and `addToken` return errors directly to the client.

- **Example**: The `releaseEditAccess` method removes a user's editing rights and updates the system accordingly.

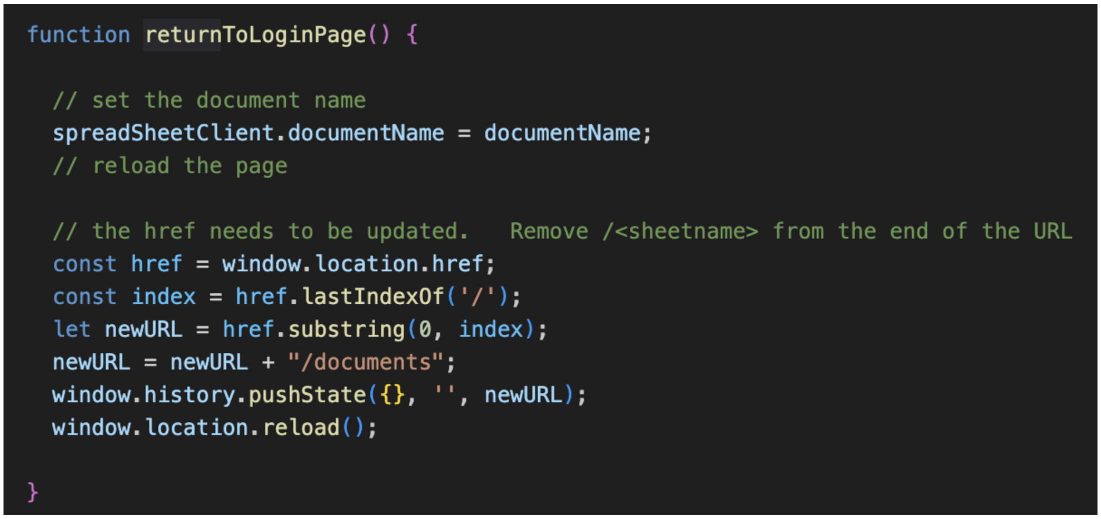


```
releaseEditAccess(user: string): void {
  // if the user is not editing a cell then we are done
  if (!this._contributingUsers.get(user)?.isEditing) {
    return;
  }

  const editingCell: string | undefined = this._contributingUsers.get(user)?.cellLabel;
  if (editingCell) {
    if (this._cellsBeingEdited.has(editingCell)) {
      this._cellsBeingEdited.delete(editingCell);
    }
  }

  // // remove the user from the list of users
  // this._contributingUsers.delete(user);

}
```

Part 4: Analyzing the Frontend (React TypeScript)

1. Multi-Screen Navigation

- **Routing Setup**:
  - The frontend does not use React Router but handles navigation manually through the `window.location.href` manipulation.
  - **Example**: In `SpreadSheet.tsx`, the `returnToLoginPage` function manually redirects the user to the login page.



```
function returnToLoginPage() {

  // set the document name
  spreadSheetClient.documentName = documentName;
  // reload the page

  // the href needs to be updated.   Remove /<sheetname> from the end of the URL
  const href = window.location.href;
  const index = href.lastIndexOf('/');
  let newURL = href.substring(0, index);
  newURL = newURL + "/documents";
  window.history.pushState({}, '', newURL);
  window.location.reload();

}
```

- **Protected Routes**:
  - There is no advanced route protection, but user session management is used to conditionally render components based on the user's login state.
  - **Example**: The `LoginPageComponent` displays different views based on the presence of a logged-in user.

2. State Management

- **Local State Management**:
  - React's `useState` is used extensively to manage local component state.
  - **Example**: The `SpreadSheet.tsx` component manages various states like `currentCell` and `currentlyEditing`.
- **Session Storage**:
  - User state is persisted across sessions using `sessionStorage`.
  - **Example**: `window.sessionStorage.getItem('userName')` is used to initialize the `userName` state in `SpreadSheet.tsx`.

3. API Interaction

- **Asynchronous Operations**:
  - The frontend interacts with the backend using `fetch` to send and receive data.
  - **Example**: The `SpreadSheetClient` class handles API interactions such as adding tokens and cells to the spreadsheet.
- **Real-Time Updates**:
  - The frontend periodically polls the backend for updates, simulating real-time interaction.
  - **Example**: The `useEffect` hook in `SpreadSheet.tsx` fetches updates every 50 milliseconds.

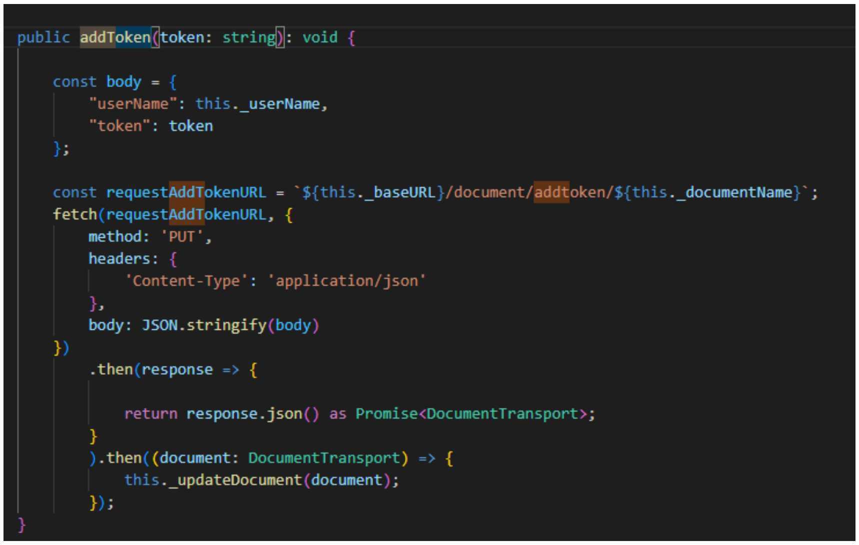
4. User Interface

- **Dynamic UI Rendering**:
  - The UI updates dynamically based on state changes, such as selecting cells or editing formulas.

    - **Example**: `getCellClass` in `SheetComponent.tsx` returns different class names based on the state, changing the appearance of cells.
- **Conditional Rendering**:
    - Components like `SheetComponent` and `KeyPad` render based on the state and user actions.
    - **Example**: `SheetComponent` renders cells differently based on the `currentlyEditing` state.

Part 5: Frontend and Backend Interaction

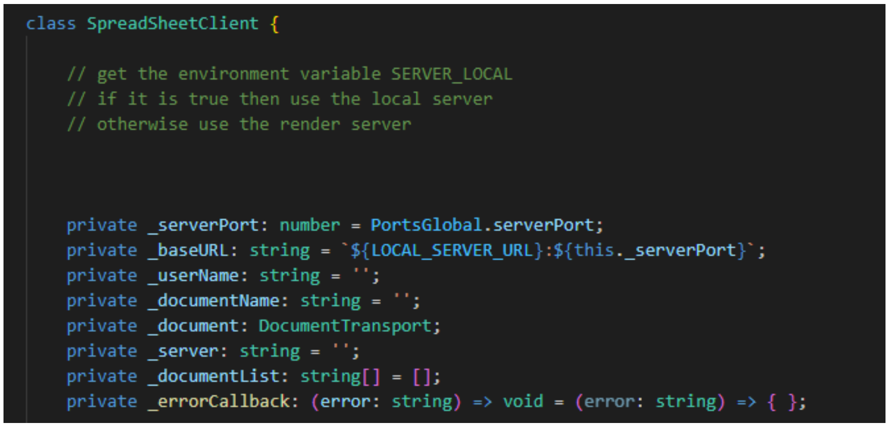
1. API Request-Response Flow

- **Request-Response Flow**:
    - The frontend sends requests to backend endpoints like `/document/addtoken` and `/document/view` and updates the state based on responses.
    - **Example**: The `addToken` method in `SpreadSheetClient.ts` sends a `PUT` request to the server, and the response updates the document state.
    - 

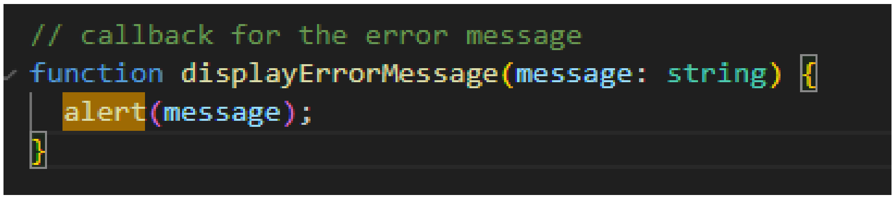
```
public addToken(token: string): void {

    const body = {
        "userName": this._userName,
        "token": token
    };

    const requestAddTokenURL = `${this._baseURL}/document/addtoken/${this._documentName}`;
    fetch(requestAddTokenURL, {
        method: 'PUT',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(body)
    })
        .then(response => {

            return response.json() as Promise<DocumentTransport>;
        }
        ).then((document: DocumentTransport) => {
            this._updateDocument(document);
        });
}
```

- **Error Handling**:
    - The frontend handles errors returned by the server, displaying alerts or updating the UI state.

- **Example**: If the backend returns an error (e.g., a circular dependency), it is displayed in the UI through an alert.



```
class SpreadSheetClient {

    // get the environment variable SERVER_LOCAL
    // if it is true then use the local server
    // otherwise use the render server



    private _serverPort: number = PortsGlobal.serverPort;
    private _baseURL: string = `${LOCAL_SERVER_URL}:${this._serverPort}`;
    private _userName: string = '';
    private _documentName: string = '';
    private _document: DocumentTransport;
    private _server: string = '';
    private _documentList: string[] = [];
    private _errorCallback: (error: string) => void = (error: string) => { };
```

- 

```
// callback for the error message
function displayErrorMessage(message: string) {
  alert(message);
}
```

2. Real-Time Interaction

- **Polling Mechanism**:
  - The frontend uses a polling mechanism to mimic real-time updates, sending periodic requests to the backend.
  - **Example**: The `useEffect` hook in `SpreadSheet.tsx` continuously polls the backend to update the UI with the latest spreadsheet data.

Key Challenges and Design Choices

1. **Performance**:
   - The polling mechanism, while effective for small updates, may become inefficient for larger data sets or more users. A WebSocket-based approach would be more efficient.
2. **Security**:
   - The current user management system lacks robust authentication and role-based access control. Implementing JWT or OAuth could enhance security.

3. **User Experience**:
   - Manual navigation using `window.location.href` could be improved with a routing library like React Router for smoother transitions and better UX.

Conclusion

This analysis highlights the strengths and areas for improvement in the client-server architecture of the spreadsheet application. The project employs well-known design patterns and manages client-server interactions effectively. However, there are opportunities to enhance performance, security, and user experience through the adoption of advanced techniques such as real-time communication with WebSockets, secure authentication, and a more sophisticated routing system.