Information Management System

Criterion C - Development

Table of contents:

-

Techniques Used

Success Criteria	**Modules/Features/ Components**	**Techniques**
General Functionality (Applies across Modules):		
1. Content Creation	All Modules (Cookie Jar, Doubt Tracker, Curiosity Space, To-Do List, Continuous Information Space, Stage Manager)	React Components, JSX, State Management (useState), Event Handlers (onClick, onSubmit), Asynchronous Operations (Promises, Async/Await), Database Interaction (Firestore $addDoc$, $setDoc$), Data Structures (Objects, Arrays), UUID Generation ($crypto.randomUUID()$), Server Timestamps.
2. Content Persistence	All Modules	Databases (Firestore), Asynchronous Operations (Promises, Async/Await), Server Components ("use server"), Data Serialization/Deserialization (implicit in Firestore interactions), Data Structures (Objects, Arrays, Interfaces), Client Components ("use client"), useEffect.
3. Content Editing	All Modules	React Components, JSX, State Management (useState), Event Handlers (onChange, onBlur, onClick), Asynchronous Operations (Promises, Async/Await), Database Interaction (Firestore

Success Criteria	**Modules/Features/ Components**	**Techniques**
		$updateDoc$, $setDoc$), Two-way Data Binding (implied through state and event handlers), Conditional Rendering, Data Structures.
4. Content Deletion	All Modules	React Components, Event Handlers (onClick), Asynchronous Operations (Promises, Async/Await), Database Interaction (Firestore $deleteDoc$), Confirmation Dialogs (using Radix UI Dialog/Toast, or browser's $confirm$), State Management (useState), Array Manipulation ($filter$).
5. Content Reordering (Drag and Drop)	Cookie Jar, To-Do List, Continuous Information Space, Stage Manager	dnd-kit library (DndContext, useDraggable, useDroppable, useSensors, PointerSensor, KeyboardSensor, closestCorners, closestCenter), React Components, Event Handlers (onDragStart, onDragEnd), State Management (useState), Asynchronous Operations (Promises, Async/Await, in update handlers), Coordinate Transformations, Array Manipulation ($arrayMove$), Data Structures, Callbacks
6. Voting Functionality (Doubts & Ideas)	Doubt Tracker, Curiosity Space	React Components, Event Handlers (onClick), State Management (useState),

Success Criteria	**Modules/Features/ Components**	**Techniques**
		Asynchronous Operations (Promises, Async/Await), Database Interaction (Firestore $updateDoc$, $increment$), Data Structures.
7. Commenting Functionality (Doubts & Ideas)	Doubt Tracker, Curiosity Space	React Components, JSX, State Management (useState, useRef, useEffect), Event Handlers (onClick, onChange, onSubmit), Asynchronous Operations (Promises, Async/Await), Database Interaction (Firestore $addDoc$, $updateDoc$, $deleteDoc$, $collection$, $query$, $orderBy$, $onSnapshot$), Recursion (for nested comments), Data Structures (Nested Arrays/Objects, Interfaces), Dynamic Rendering.
8. Status Management (Doubts & Ideas)	Doubt Tracker, Curiosity Space	React Components, Event Handlers (onClick), State Management (useState), Asynchronous Operations (Promises, Async/Await), Database Interaction (Firestore $updateDoc$), Conditional Rendering, Data Structures.
Module-Specific Criteria:		
9. Cookie Jar - Cookie Management	Cookie Jar	All techniques listed for Content Creation, Persistence, Editing, Deletion, and Reordering, specifically applied

Success Criteria	Modules/Features/ Components	Techniques
		to the $Cookie$ data structure. Custom Hooks (useToast).
10. Doubt Tracker - Doubt Resolution	Doubt Tracker	All techniques listed for Content Creation, Persistence, Editing, Deletion, Voting, Commenting, and Status Management, specifically applied to the $Doubt$ data structure. Custom Hooks (useToast, useUser) .
11. Curiosity Space (Idea Tracker) - Idea Resolution	Curiosity Space	All techniques listed for Content Creation, Persistence, Editing, Deletion, Voting, Commenting, and Status Management, specifically applied to the $Idea$ data structure. Custom Hooks (useToast, useUser) .
12. To-Do List - Task Management	To-Do List	All techniques listed for Content Creation, Persistence, Editing, Deletion, and Reordering, specifically applied to the nested $Section$, $Column$, $Note$, and $Subtask$ data structures. Complex Drag and Drop (nested draggables and droppables), Custom Hooks (useToast, useUser), Recursion
13. Continuous Information Space - Notebook Management	Continuous Information Space	All techniques listed for Content Creation, Persistence, Editing, Deletion, and Reordering, specifically applied to the nested $Notebook$,

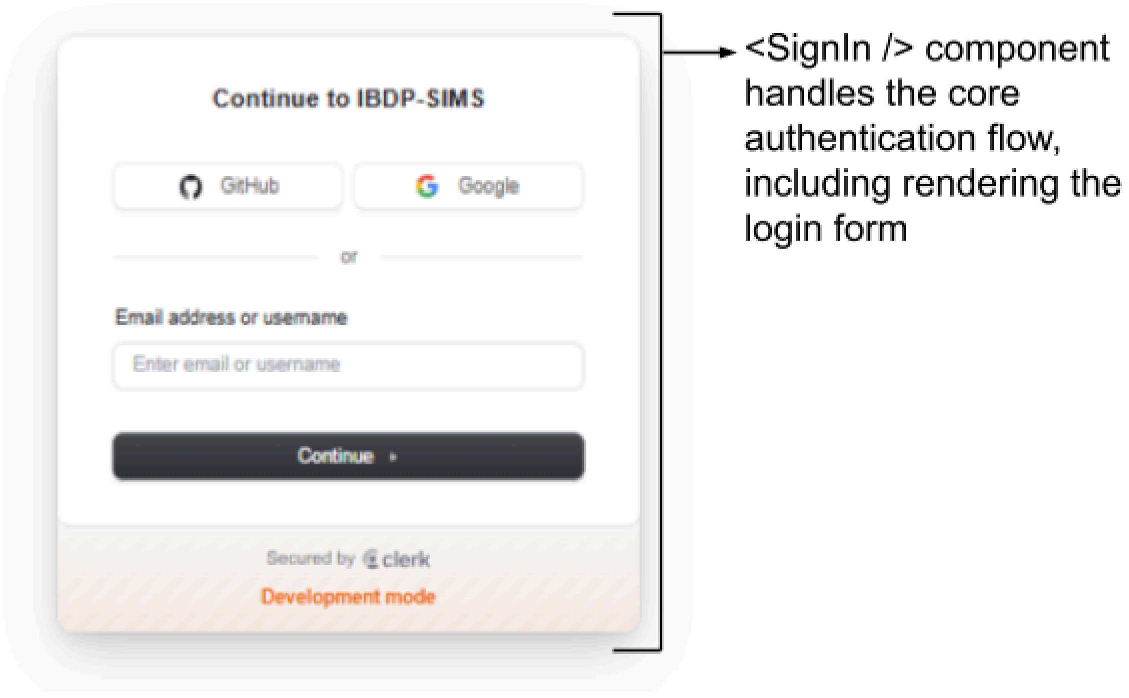
Success Criteria	Modules/Features/ Components	Techniques
		$Section$, $Column$, and $Note$ data structures. Custom Hooks (use404Redirect) .
14. Stage Manager - Space & Window Management	Stage Manager	All techniques listed for Content Creation, Persistence, Editing, Deletion, and Reordering, specifically applied to the $Space$ and $Window$ data structures. Custom Hooks (useSpaceId, usePosition, useSize). Advanced Drag and Drop (with snapping). Custom Event Dispatcher., OOP (Classes, Objects, Encapsulation), Context API
General Application Success (Usability, Performance, Robustness):		
15. Intuitive User Interface	All Modules	React Components, JSX, UI Libraries (Radix UI, Headless UI, Lucide React), CSS-in-JS (Tailwind CSS), Component Composition, Conditional Rendering, Event Handlers, State Management, Clean Code Principles, CSS Modules
16. Cross-Browser & Device Compatibility	All Modules	Web Technologies (HTML, CSS, JavaScript), Responsive Design (Tailwind CSS utility classes), Browser API Usage (standard APIs), Polyfills (potentially, though not explicitly shown in this code,

Success Criteria	**Modules/Features/ Components**	**Techniques**
		may be used by underlying libraries).
17. User Authentication & Security	User Authentication (Clerk integration)	Clerk Authentication, Middleware (Clerk Middleware), Protected Routes (Next.js routing), API Keys (Firebase, Clerk - handled securely via environment variables).
18. Real-time Data Synchronization	All Modules	Firestore Real-time Updates (onSnapshot), Asynchronous Operations, State Management.
19. Fast Data Retrieval	All Modules	Firestore Queries, Asynchronous Operations, Optimized Data Structures, Efficient Rendering (React's virtual DOM), Debouncing, Server Components ("use server")
20. Component-Based Architecture for Maintainability and Scalability	Application Structure	React Components, Component Composition, Modular Programming (import/export), Code Organization (file structure), Single Responsibility Principle (attempted, though can always be improved), Typescript, Zod

Individual Pages

Login/SignUp page

The Login/Sign-up page serves as the entry point to the application, allowing new users to create accounts and existing users to securely access their personalized information management system. Users who do not have an account can sign up, while users with existing credentials can log in to access the main application features. This page is crucial for ensuring user authentication and data security before accessing any protected modules.


Continue to IBDP-SIMS
GitHub
Google
or
Email address or username
Enter email or username
Continue
Secured by clerk
Development mode
<SignIn /> component handles the core authentication flow, including rendering the login form

Techniques Used:

- **React Components**: The Login/Sign-up page is built using React components, primarily leveraging the `<SignIn />` component provided by the Clerk library. This component encapsulates the complex logic and UI elements required for user authentication, promoting modularity and reusability.
    - *Reasoning:* React components allow for a structured and organized approach to building user interfaces. Using the `<SignIn />` component abstracts away the intricacies of building an authentication form from scratch.
- **JSX (JavaScript XML)**: JSX is used to define the structure and composition of the React components within the Login/Sign-up page. It allows for writing HTML-like syntax within JavaScript files, making the component structure declarative and easier to understand.

    - *Reasoning:* JSX simplifies the process of creating and manipulating the DOM in React, enhancing developer productivity and code readability.
- **Clerk Authentication**: The application utilizes Clerk for comprehensive user authentication management. The `<SignIn />` component handles the core authentication flow, including rendering the login form, processing user credentials, managing sessions, and ensuring secure authentication.
    - *Reasoning:* Integrating Clerk provides robust and secure authentication capabilities out-of-the-box, handling complex security considerations and reducing development effort for implementing secure user authentication from scratch.
- **Next.js Routing**: Next.js's file-system based routing is employed to define the Login/Sign-up page as a specific route accessible at `/sign-in`. This route is automatically configured by placing the `page.tsx` file within the `app/sign-in/[[...sign-in]]` directory, streamlining navigation and page structure management.
    - *Reasoning:* Next.js routing simplifies the process of creating navigation paths within the application. The file-system based routing is intuitive and reduces the configuration needed for page routing.
- **UI Libraries (Clerk UI components)**: The visual elements and styling of the Login/Sign-up page are largely provided by Clerk's pre-built UI components, specifically the `<SignIn />` component. This component is designed to be user-friendly and visually consistent with modern web applications.
    - *Reasoning:* Utilizing UI libraries like Clerk's components accelerates UI development and ensures a consistent and professional user interface without requiring extensive custom styling and component creation.
- **Component Composition**: The `page.tsx` file effectively uses component composition by embedding the `<SignIn />` component within the overall page structure. This demonstrates a core React principle of building complex UIs by combining smaller, self-contained components.
    - *Reasoning:* Component composition is a fundamental technique in React for creating maintainable and scalable applications. It allows for breaking down the UI into manageable parts and reusing components across the application.
- **Server-Side Rendering (SSR) / Client-Side Rendering (CSR) (Next.js default)**: While not explicitly defined in the code snippet, Next.js manages the rendering strategy. For authentication pages, Next.js may lean towards Client-Side Rendering to handle interactive form elements efficiently. This default behavior of Next.js contributes to a faster initial page load and a more interactive user experience.

    - *Reasoning:* Next.js's automatic handling of rendering strategies optimizes performance and user experience. Choosing CSR or SSR appropriately for different page types can significantly impact application responsiveness.
- **Form Handling (implicitly by Clerk)**: The `<SignIn />` component implicitly handles the form submission process for the login form. It manages user input, form validation, and submission logic, abstracting away the need for manual form handling in the `page.tsx` file.
    - *Reasoning:* Abstracting form handling logic into the Clerk component simplifies the `page.tsx` file and reduces the amount of code needed to manage user input and form submissions, making the code cleaner and less error-prone.
- **Secure Password Handling (by Clerk)**: Clerk is responsible for the secure handling of user passwords during the sign-up and login processes. This includes password hashing, secure storage, and protection against common security vulnerabilities.
    - *Reasoning:* Delegating password security to Clerk ensures that best practices are followed and reduces the risk of security breaches related to password management, as Clerk is a specialized authentication service.
- **Error Handling (by Clerk)**: The `<SignIn />` component incorporates built-in error handling mechanisms to manage login failures and provide informative feedback to the user. This includes displaying error messages for incorrect credentials or other authentication issues.
    - *Reasoning:* Integrated error handling within the `<SignIn />` component improves the user experience by providing immediate and helpful feedback in case of login problems, guiding users to resolve issues and successfully log in.

Imports used:

```
import { SignIn } from "@clerk/nextjs";
```

Functions used:

There are no specific functions defined within the `app\sign-in\[[...sign-in]]\page.tsx` file itself. The page primarily renders the `<SignIn />` component from Clerk, which encapsulates all the necessary functionality for the login/sign-up process. The logic for authentication is handled internally by the Clerk library.

Main Page - Workstage View, Dashboard View, and Side-by-Side View

The Main Page is the central hub of the application, providing users with access to all core modules and functionalities. It offers three distinct views designed to cater to different user needs and workflows:

- **Workstage View**: This view, powered by the Stage Manager component, provides a flexible and visual workspace. Users can create and manipulate Spaces and Windows to organize their tasks, notes, and information in a spatial manner. This view is ideal for active work sessions, allowing users to arrange different tools and resources as needed.
- **Dashboard View**: In contrast, the Dashboard View offers a consolidated overview of the user's information management system. It presents key modules like the Cookie Jar, To-Do List, and Continuous Information Space in a structured layout, allowing users to quickly review achievements, tasks, and notes. This view is suited for planning, review, and gaining a holistic understanding of their personal information landscape.
- **Side-by-Side View**: This view combines the functionalities of both the Workstage and Dashboard views, displaying them concurrently in a split-screen layout. This allows users to simultaneously engage with the active workspace of the Stage Manager and the overview provided by the Dashboard, facilitating a workflow that requires both detailed work and a broad perspective. Users can toggle between "Toggle View" (either Workstage OR Dashboard) and "Side-by-Side View" using a button in the header.

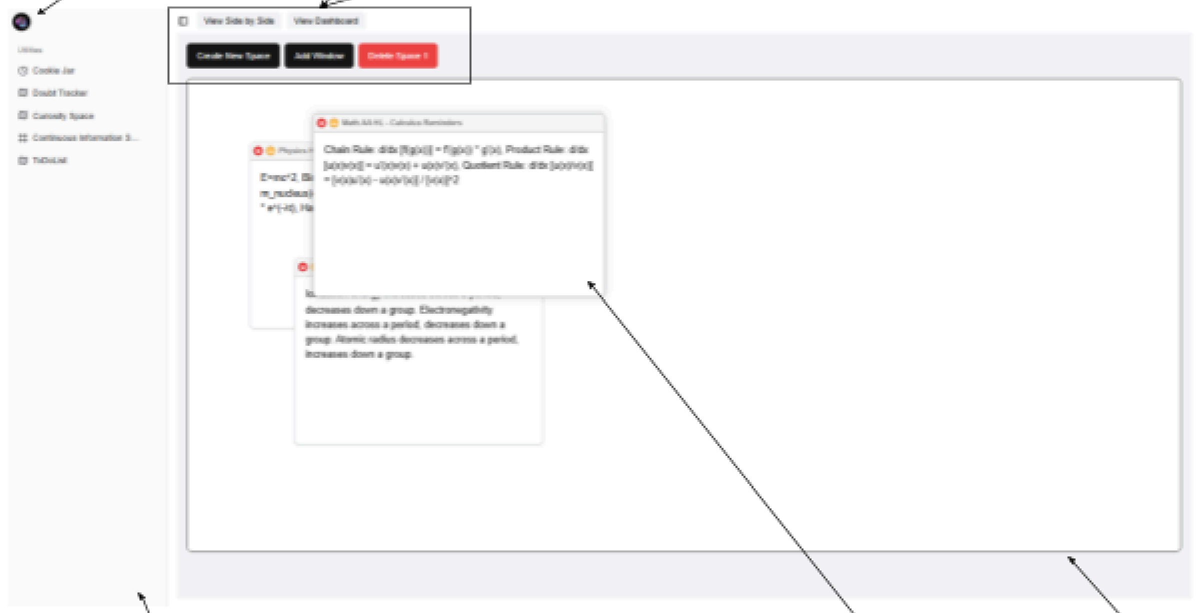
Screenshots:

Workstage View Screenshot

<UserButton /> displays the Useer icon, which bring up a settings modal for the user's account, this component is directly managed by Clerk Authentication.

Spaces are created using the <Space> component, allowing users to define distinct workspaces within the Stage Manager.

All buttons use a <Button> from Shadcn UI.



Chain Rule: d/dx [f(g(x))] = f'(g(x)) * g'(x), Product Rule: d/dx [u(x)v(x)] = u'(x)v(x) + u(x)v'(x), Quotient Rule: d/dx [u(x)/v(x)] = [v(x)u'(x) - u(x)v'(x)] / [v(x)]^2

E=mc^2, ... m_nucleus ... * e^(-λt), Ha...

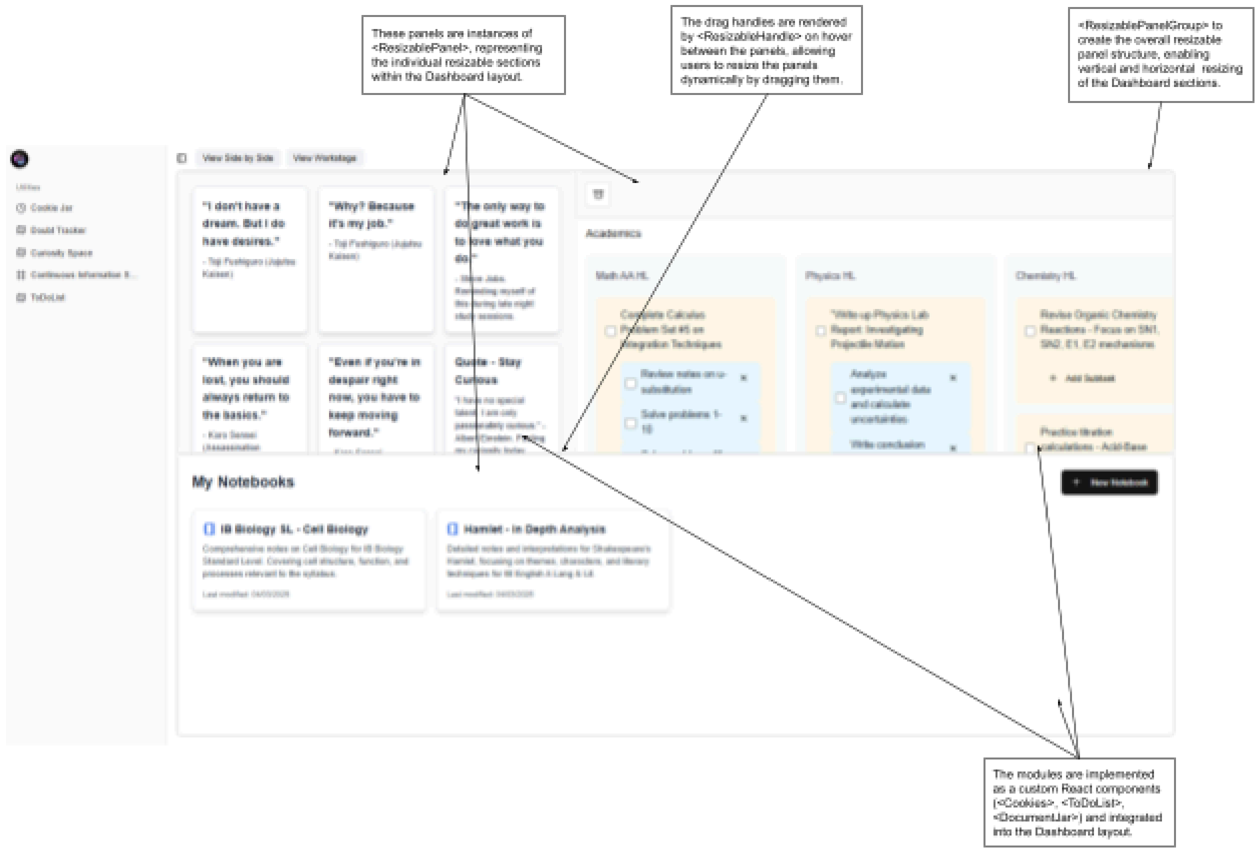
... decreases down a group. Electronegativity increases across a period, decreases down a group. Atomic radius decreases across a period, increases down a group.

The Workstage View is seamlessly integrated into the application's sidebar layout using <SidebarInset> to provide visual spacing and <SidebarTrigger> to control sidebar toggling, both from Radix UI.

These windows are instances of the <BasicWindow> component, the fundamental building blocks of the Workstage, enabling users to display and interact with content.

This area is rendered using the <Manager> component, which is a modified version of the "React Kitten" library, providing the core framework for drag-and-drop window management and spatial layout.

Dashboard View Screenshot:



These panels are instances of <ResizablePanel>, representing the individual resizable sections within the Dashboard layout.

The drag handles are rendered by <ResizableHandle> on hover between the panels, allowing users to resize the panels dynamically by dragging them.

<ResizablePanelGroup> to create the overall resizable panel structure, enabling vertical and horizontal resizing of the Dashboard sections.

The modules are implemented as a custom React components (<Cookies>, <ToDoList>, <DocumentJar>) and integrated into the Dashboard layout.

"I don't have a dream. But I do have desires."
- Toji Fushiguro (Jujutsu Kaisen)

"Why? Because it's my job."
- Toji Fushiguro (Jujutsu Kaisen)

"The only way to do great work is to love what you do."
- Steve Jobs. Reminding myself of this during late night study sessions.

"When you are lost, you should always return to the basics."
- Koro Sensei (Assassination

"Even if you're in despair right now, you have to keep moving forward."
- Koro Sensei

Quote - Stay Curious
"I have no special talent. I am only passionately curious." - Albert Einstein. Putting my curiosity today

Academics

Math AA HL
Complete Calculus Problem Set #5 on Integration Techniques
Review notes on u-substitution
Solve problems 1-10

Physics HL
"Write-up Physics Lab Report: Investigating Projectile Motion
Analyze experimental data and calculate uncertainties
Write conclusion

Chemistry HL
Revise Organic Chemistry Reactions - Focus on SN1, SN2, E1, E2 mechanisms
Add Subtask
Practice titration calculations - Acid-Base

My Notebooks

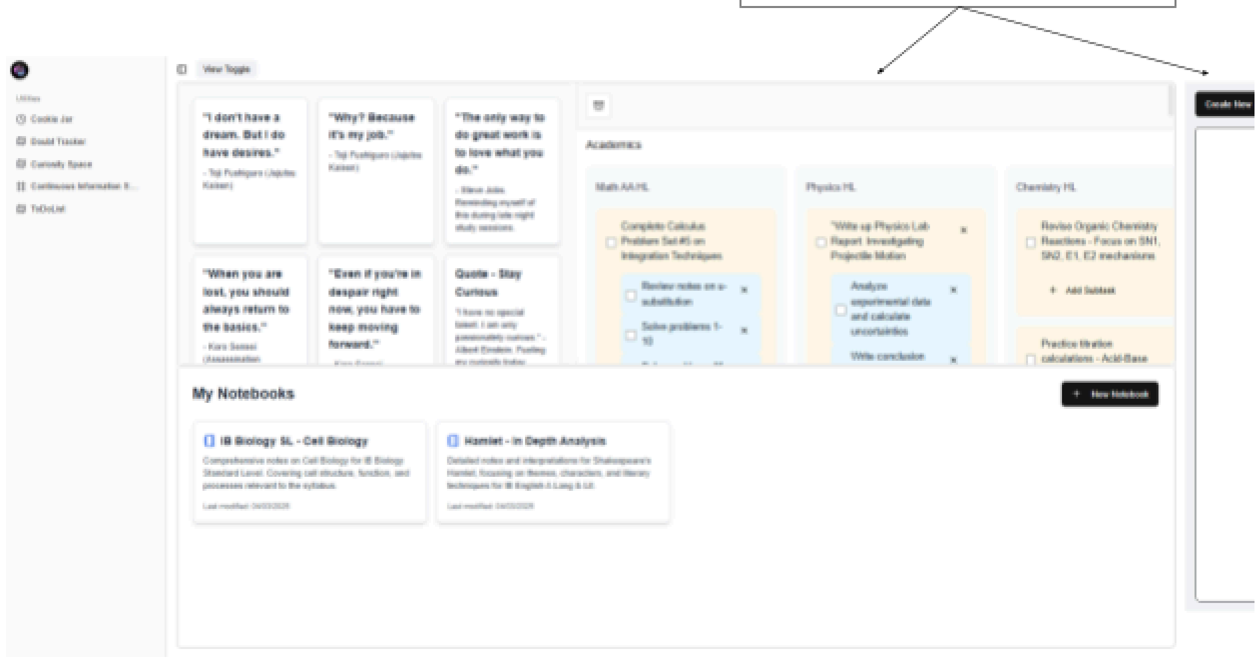
IB Biology SL - Cell Biology
Comprehensive notes on Cell Biology for IB Biology Standard Level. Covering cell structure, function, and processes relevant to the syllabus.
Last modified: 04/03/2025

Hamlet - In Depth Analysis
Detailed notes and interpretations for Shakespeare's Hamlet, focusing on themes, characters, and literary techniques for IB English A Lang & Lit.
Last modified: 04/03/2025

Side-by-Side View Screenshot:

Side-by-side unscrolled view:

Flexbox layout is used to arrange the Dashboard and Workstage components side-by-side, and conditional rendering (based on viewMode state) controls whether this layout is displayed.



"I don't have a dream. But I do have desires."
- Toji Fushiguro (Jujutsu Kaisen)

"Why? Because it's my job."
- Toji Fushiguro (Jujutsu Kaisen)

"The only way to do great work is to love what you do."
- Steve Jobs. Reminding myself of this during late night study sessions.

"When you are lost, you should always return to the basics."
- Koro Sensei (Assassination

"Even if you're in despair right now, you have to keep moving forward."
- Koro Sensei

Quote - Stay Curious
"I have no special talent. I am only passionately curious." - Albert Einstein. Pushing my curiosity today.

Complete Calculus Problem Set #5 on Integration Techniques
Review notes on u-substitution
Solve problems 1-10

"Write up Physics Lab Report: Investigating Projectile Motion
Analyze experimental data and calculate uncertainties
Write conclusion

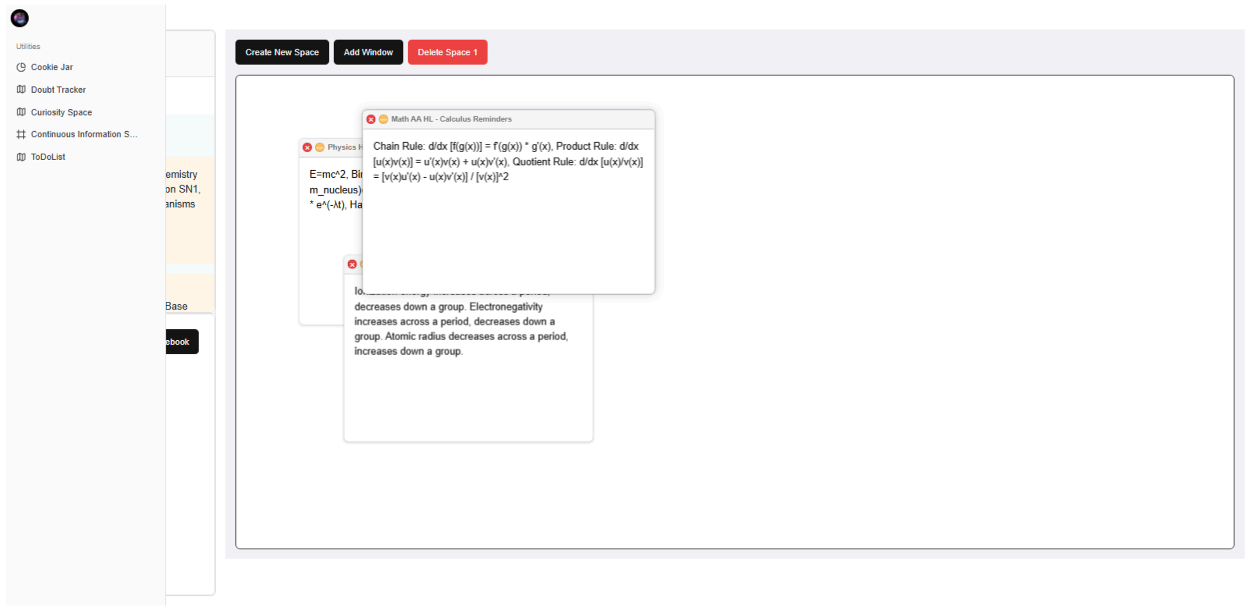
Revise Organic Chemistry Reactions - Focus on SN1, SN2, E1, E2 mechanisms

Practice titration calculations - Acid-Base

IB Biology SL - Cell Biology
Comprehensive notes on Cell Biology for IB Biology Standard Level. Covering cell structure, function, and processes relevant to the syllabus.
Last modified: 04/03/2025

Hamlet - In Depth Analysis
Detailed notes and interpretations for Shakespeare's Hamlet, focusing on themes, characters, and literary techniques for IB English A Lang & Lit.
Last modified: 04/03/2025

Side-by-side scrolled view:



Chain Rule: d/dx [f(g(x))] = f'(g(x)) * g'(x), Product Rule: d/dx [u(x)v(x)] = u'(x)v(x) + u(x)v'(x), Quotient Rule: d/dx [u(x)/v(x)] = [v(x)u'(x) - u(x)v'(x)] / [v(x)]^2

E=mc^2, Bi
m_nucleus)
* e^(-λt), Ha

Io
decreases down a group. Electronegativity increases across a period, decreases down a group. Atomic radius decreases across a period, increases down a group.

Techniques used:

- **React Components**: The Main Page and its different views are constructed using React components. The `Page` component in `page.tsx` acts as the main container, orchestrating the rendering of `WorkStage`, `Dashboard`, and handling view switching logic. Individual modules within the Dashboard (CookieJar, ToDoList etc.) and the Stage Manager are also implemented as reusable React components.
    - *Reasoning:* React's component-based architecture enables modularity and reusability, making it easier to manage the complexity of the Main Page with its multiple views and modules.
- **JSX (JavaScript XML)**: JSX is extensively used within `app\page.tsx` and the related components to define the UI structure declaratively. It facilitates the composition of UI elements and components in a clear and maintainable way.
    - *Reasoning:* JSX simplifies UI development in React by allowing developers to write HTML-like structures directly within JavaScript, improving readability and development speed.
- **State Management (useState)**: The `useState` hook is crucial for managing the view state of the Main Page. It controls which view is currently active (`viewMode` state: 'toggle' or 'side-by-side') and, in "toggle" mode, whether the Dashboard or Workstage is displayed (`showDashboard` state). These states drive conditional rendering and UI updates.
    - *Reasoning:* State management with `useState` is essential for creating interactive and dynamic UIs in React. It allows components to re-render and update the UI in response to user interactions and data changes.
- **Conditional Rendering**: Conditional rendering is heavily employed to switch between the different views. Based on the `viewMode` and `showDashboard` state variables, different components (`WorkStage`, `Dashboard`, or both in Side-by-Side) are rendered. This allows the Main Page to dynamically adapt its display based on user preferences and actions.
    - *Reasoning:* Conditional rendering is a powerful React feature for creating flexible UIs that can display different content or components based on application state, user input, or other conditions.
- **Component Composition**: The `Page` component demonstrates effective component composition by integrating `SidebarProvider`, `AppSidebar`, `SidebarInset`, `WorkStage`, and `Dashboard` components. This composition allows for building a complex layout and functionality by combining smaller, specialized components.

    - *Reasoning:* Component composition promotes code reusability, modularity, and maintainability. It allows for breaking down complex UI requirements into smaller, manageable pieces that are easier to develop and test.
- **UI Libraries (Radix UI, Tailwind CSS)**: The Main Page leverages UI libraries like Radix UI for components like `SidebarProvider`, `SidebarInset`, and `Sheet`, and Tailwind CSS for styling. These libraries contribute to a consistent, accessible, and visually appealing user interface.
    - *Reasoning:* UI libraries provide pre-built, well-tested, and accessible UI components, significantly speeding up development and ensuring a professional look and feel for the application. Tailwind CSS facilitates rapid and consistent styling across components.
- **CSS-in-JS (Tailwind CSS)**: Tailwind CSS, a CSS-in-JS framework, is used for styling the Main Page and its components. Utility classes from Tailwind CSS are applied directly within the JSX to style elements and manage layout, enhancing styling efficiency and consistency.
    - *Reasoning:* Tailwind CSS allows for rapid styling and UI adjustments directly within the component code, improving development speed and ensuring a consistent design language throughout the application.
- **React Resizable Panels**: The Dashboard View utilizes the `react-resizable-panels` library to implement resizable panels. This library enables users to adjust the size of different sections within the Dashboard, providing a customizable layout and user experience.
    - *Reasoning:* Using a library like `react-resizable-panels` provides a ready-made solution for implementing complex resizable layouts, saving significant development time and effort compared to building such functionality from scratch.
- **Scroll Area (Radix UI ScrollArea)**: Within the Dashboard View, the Radix UI `ScrollArea` component is used to manage scrollable content within the resizable panels. This ensures that content within each module can be scrolled independently and efficiently, regardless of the panel's size.
    - *Reasoning:* `ScrollArea` components from UI libraries provide a standardized and accessible way to handle scrollable content, ensuring smooth scrolling behavior and consistent UI across different browsers and devices.
- **Modification of "React Kitten" Library (Stage Manager/Workstage View)**: The Workstage View is built upon a modified version of the "React Kitten" library (renamed to `components\stage-manager)`. This library, originally developed by "rohanrhu" on GitHub, provides the foundational structure and logic for

creating draggable and resizable window interfaces. The modifications involve adapting the library to the specific requirements of the application, including integrating it with the overall application architecture and enhancing its features.

  - *Reasoning:* Leveraging and modifying an existing library like "React Kitten" significantly reduces the development time and complexity of building a sophisticated feature like the Stage Manager from scratch. It allows for building upon proven code and focusing on customization and integration rather than core functionality development.

Imports used:

Import statements used in `app\page.tsx`

```
import { useState } from "react";
import { AppSidebar } from "@/components/sidebar-07/app-sidebar";
import {
  SidebarInset,
  SidebarProvider,
  SidebarTrigger,
} from "@/components/ui/sidebar";
import Dashboard from "@/app/components/Dashboard";
import WorkStage from "@/app/components/WorkStage";
```

Functions used:

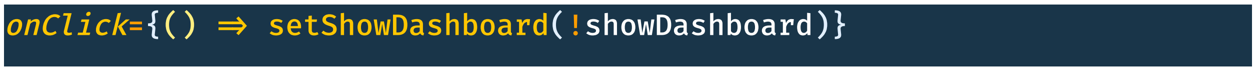
- **`handleSideBySideToggle`**: This function is defined within the `Page` component and is responsible for toggling the `viewMode` state between "toggle" and "side-by-side". It's triggered by the "View Side by Side" / "View Toggle" button click.

    - *Functionality:* This function updates the `viewMode` state, which in turn triggers a re-render of the Main Page component, causing it to switch between "toggle" view (Workstage or Dashboard) and "side-by-side" view (Workstage and Dashboard simultaneously).
    - *Screenshot:* handleSideBySideToggle `function from` app\page.tsx

```
const handleSideBySideToggle = () => {
  setViewMode(viewMode === "toggle" ? "side-by-side" : "toggle");
  console.log(
    "Side by Side Toggle Clicked - viewMode:",
    viewMode === "toggle" ? "side-by-side" : "toggle"
  ); // Log after setting state
};
```

- **`onClick` handler for "View Dashboard" / "View Workstage" button**: This inline function is defined directly within the JSX of the `Page` component and is attached to the "View Dashboard" / "View Workstage" button. It is only visible when in "toggle" `viewMode`.

- *Functionality:* This handler toggles the **`showDashboard`** state, which, when in "toggle" **`viewMode`**, determines whether the Dashboard or the Workstage component is rendered.
- *Screenshot:* onClick `handler for the "View Dashboard" / "View Workstage" button in` app\page.tsx



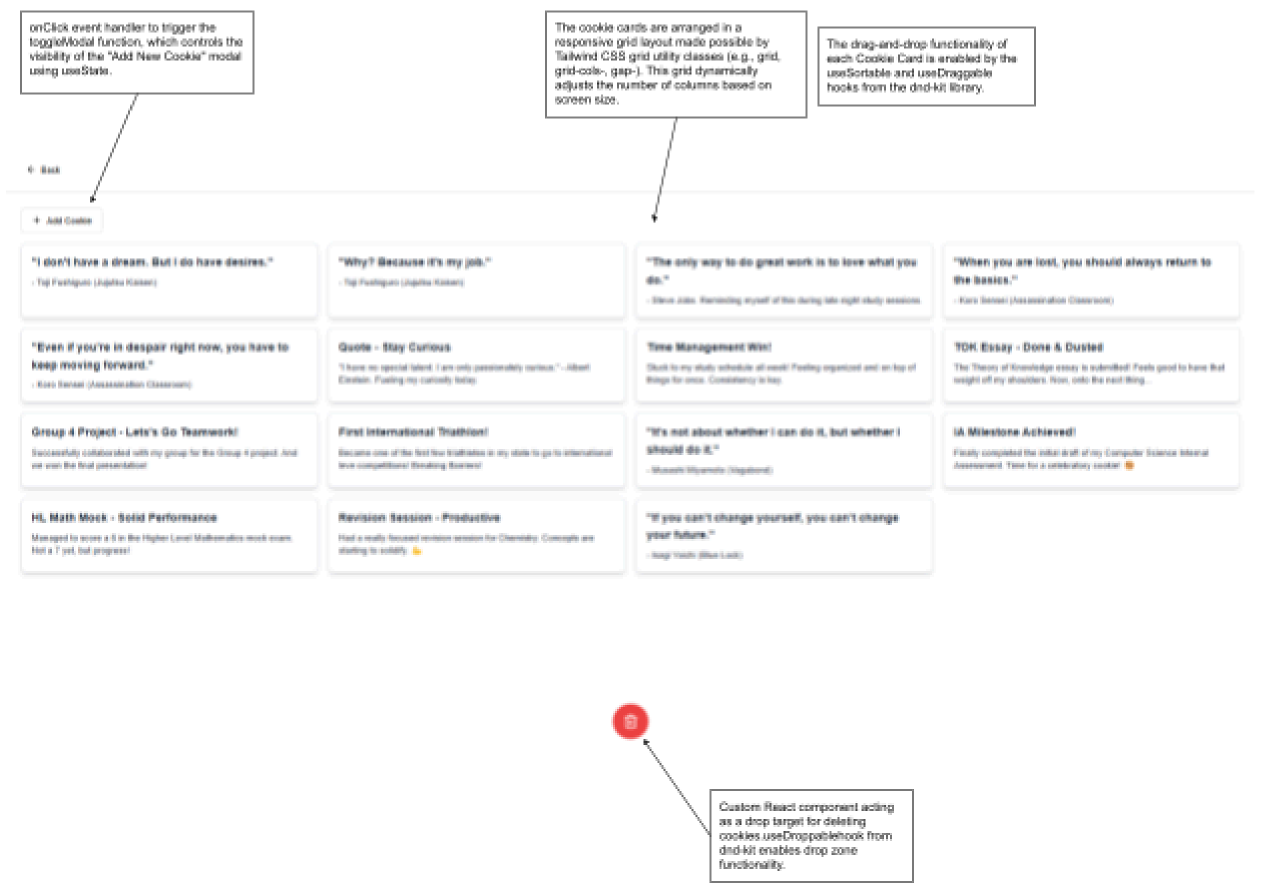
```
onClick={() ⇒ setShowDashboard(!showDashboard)}
```

-

Cookie Jar

The Cookie Jar module is designed to help users manage and visualize their personal achievements or "cookies" – inspired by the motivational concept popularized by David Goggins. This module allows users to create, view, edit, delete, and reorder cookies in a visually engaging grid layout. Users can add new cookies with a name and description, modify existing cookies, remove cookies, and rearrange them using drag-and-drop functionality to personalize their digital cookie jar. The module is designed to be intuitive and motivating, providing a visual representation of the user's accomplishments.

Screenshot:



onClick event handler to trigger the toggleModal function, which controls the visibility of the "Add New Cookie" modal using useState.

The cookie cards are arranged in a responsive grid layout made possible by Tailwind CSS grid utility classes (e.g., grid, grid-cols-, gap-). This grid dynamically adjusts the number of columns based on screen size.

The drag-and-drop functionality of each Cookie Card is enabled by the useSortable and useDraggable hooks from the dnd-kit library.

Custom React component acting as a drop target for deleting cookies.useDroppablehook from dnd-kit enables drop zone functionality.

← Back

+ Add Cookie

"I don't have a dream. But I do have desires."
- Toji Fushiguro (Jujutsu Kaisen)

"Why? Because it's my job."
- Toji Fushiguro (Jujutsu Kaisen)

"The only way to do great work is to love what you do."
- Steve Jobs. Reminding myself of this during late night study sessions.

"When you are lost, you should always return to the basics."
- Koro Sensei (Assassination Classroom)

"Even if you're in despair right now, you have to keep moving forward."
- Koro Sensei (Assassination Classroom)

Quote - Stay Curious
"I have no special talent. I am only passionately curious." - Albert Einstein. Fueling my curiosity today.

Time Management Win!
Stuck to my study schedule all week! Feeling organized and on top of things for once. Consistency is key.

TOK Essay - Done & Dusted
The Theory of Knowledge essay is submitted! Feels good to have that weight off my shoulders. Now, onto the next thing...

Group 4 Project - Let's Go Teamwork!
Successfully collaborated with my group for the Group 4 project. And we won the final presentation!

First International Triathlon!
Became one of the first few triathletes in my state to go to international level competitions! Breaking Barriers!

"It's not about whether I can do it, but whether I should do it."
- Musashi Miyamoto (Vagabond)

IA Milestone Achieved!
Finally completed the initial draft of my Computer Science Internal Assessment. Time for a celebratory cookie! 🍪

HL Math Mock - Solid Performance
Managed to score a 5 in the Higher Level Mathematics mock exam. Not a 7 yet, but progress!

Revision Session - Productive
Had a really focused revision session for Chemistry. Concepts are starting to solidify. 💪

"If you can't change yourself, you can't change your future."
- Isagi Yoichi (Blue Lock)

Techniques used:

- **React Components**: The Cookie Jar is implemented using multiple React components, including `CookieJar` (the main page component), `SortableCookieCard` (for individual draggable cookie cards), and `TrashZone` (for the deletion drop target). This component-based structure promotes modularity, reusability, and maintainability of the codebase.

    - *Reasoning:* React components enable the encapsulation of UI elements and logic, making it easier to manage the complexity of the Cookie Jar module and its features.
- **JSX (JavaScript XML)**: JSX is used extensively to define the structure and rendering logic of the Cookie Jar components. It allows for a declarative and intuitive way to describe the UI, embedding HTML-like syntax directly within JavaScript code.
    - *Reasoning:* JSX enhances code readability and developer productivity by providing a familiar and efficient way to construct and update the UI elements of the Cookie Jar.
- **State Management (useState, useEffect)**: `useState` is used for managing various aspects of the Cookie Jar's UI and data, including: `isModalOpen` (controlling the visibility of the Add/Edit Cookie modal), `cookies` (storing the array of cookie objects), `newCookie` (managing input values in the modal), `isEditing` (tracking modal edit mode), `editingCookieId` (storing the ID of the cookie being edited), and `activeId` (tracking the currently dragged cookie for visual feedback). `useEffect` is used to fetch cookies from the database when the component mounts and to synchronize data updates.
    - *Reasoning:* State management is crucial for React applications to handle dynamic data and user interactions. `useState` and `useEffect` hooks enable efficient updates and side effects management within the Cookie Jar component.
- **Drag and Drop Functionality (dnd-kit library)**: The `dnd-kit` library is integrated to enable drag-and-drop reordering of cookie cards within the Cookie Jar grid and deletion via dragging to the `TrashZone`. Hooks like `DndContext`, `useDraggable`, `useDroppable`, `useSensors`, `closestCenter`, `rectSortingStrategy`, and `useSortable` are utilized to implement this complex interaction.
    - *Reasoning:* Drag and drop provides an intuitive and engaging user experience for reordering and deleting cookies. `dnd-kit` is a powerful library that abstracts away the complexity of implementing drag-and-drop interactions, offering robust and customizable solutions.
- **Asynchronous Operations (Promises, Async/Await)**: Asynchronous operations using Promises and `async/await` are employed for interacting with the Firestore database. Functions like `readCookies`, `addCookie`, `updateCookie`, and `deleteCookie` use asynchronous operations to handle data fetching and updates, ensuring non-blocking UI and efficient data management.

    - *Reasoning:* Asynchronous operations are essential for handling database interactions in web applications, preventing UI freezes and ensuring a smooth user experience while waiting for data to be retrieved or updated.
- **Database Interaction (Firestore)**: The Cookie Jar module interacts with Google Firestore to persist cookie data. Functions are implemented to perform CRUD operations on the "cookieJar" collection in Firestore, allowing for data persistence, retrieval, and manipulation.
    - *Reasoning:* Firestore provides a scalable and real-time NoSQL database solution, ideal for storing and synchronizing user data in web applications. Using Firestore ensures data persistence across sessions and devices.
- **Modals (Radix UI Dialog)**: The Radix UI Dialog component is used to implement the modal for adding and editing cookies. This dialog provides a focused UI for data input and management, improving user experience by separating the cookie creation/editing process from the main Cookie Jar grid.
    - *Reasoning:* Modals are effective UI patterns for isolating user tasks and preventing distractions from the main application view. Radix UI Dialog provides accessible and customizable modal components.
- **Grid Layout (CSS Grid)**: CSS Grid is utilized to create the responsive grid layout for displaying cookie cards. The `` `grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 xl:grid-cols-${gridCols} gap-4` `` TailwindCSS classes dynamically adjust the number of columns based on screen size, ensuring optimal display across different devices.
    - *Reasoning:* CSS Grid provides a powerful and flexible system for creating complex and responsive layouts, essential for displaying items in a structured and visually appealing manner, especially for varying screen sizes.
- **UI Libraries (Radix UI, Tailwind CSS, Lucide React)**: The Cookie Jar module utilizes several UI libraries: Radix UI for modal and UI primitives, Tailwind CSS for styling, and Lucide React for icons (Plus, Trash2, Edit). These libraries contribute to a consistent design language, accessibility, and accelerated UI development.
    - *Reasoning:* Utilizing established UI libraries significantly reduces development time and ensures a polished and accessible user interface. These libraries offer pre-built components and styling solutions, promoting consistency and best practices in UI development.
- **Custom Hooks (useToast)**: The `useToast` custom hook is used to display toast notifications for user feedback, such as confirmation messages after adding or deleting a cookie. This enhances the user experience by providing non-intrusive feedback for actions within the Cookie Jar.

    - *Reasoning:* Custom hooks like `useToast` encapsulate reusable logic and UI patterns, promoting code reusability and cleaner component structure. Toast notifications provide user-friendly feedback without disrupting the user workflow.

Imports used:

Import statements used in `components\CookieJar.tsx`

```
import React, { useState, useEffect } from "react";
import { Plus, Trash2, Edit } from "lucide-react";
import { DndContext, closestCenter, KeyboardSensor, PointerSensor, useSensor, useSensors,
DragEndEvent, DragStartEvent, useDraggable, useDroppable } from "@dnd-kit/core";
import { arrayMove, sortableKeyboardCoordinates, rectSortingStrategy, SortableContext,
useSortable } from "@dnd-kit/sortable";
import { CSS } from "@dnd-kit/utilities";
import { Button } from "@/components/ui/button";
import { readCookies, updateCookiePositions, deleteCookie, updateCookie, Cookie, addCookie } from
"@/lib/cookies-actions";
```

Functions used:

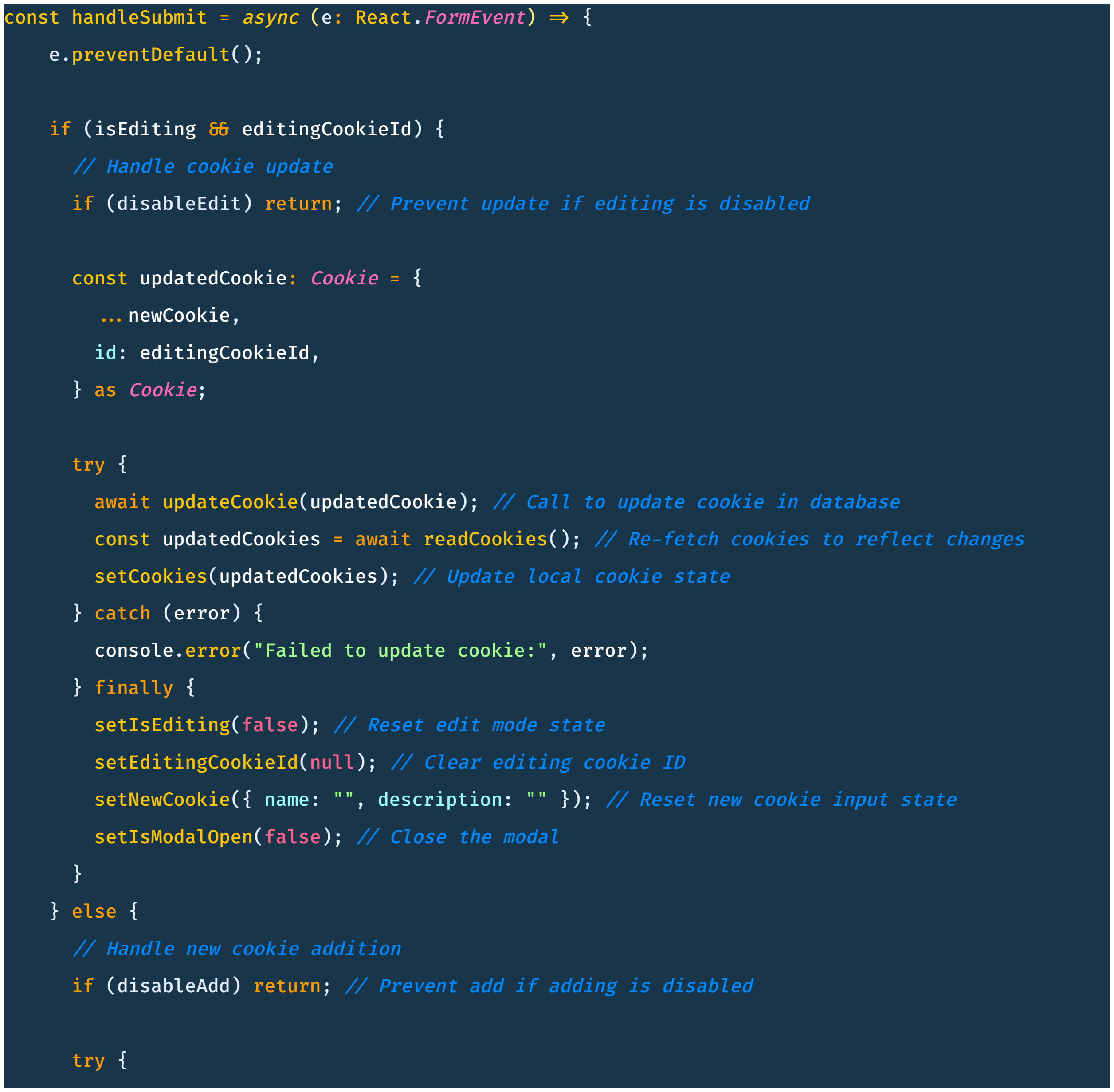
- **`fetchCookies` (useEffect callback)**: This asynchronous function, defined within the `useEffect` hook, is responsible for fetching cookie data from Firestore when the `CookieJar` component mounts. It calls the `readCookies` action and updates the `cookies` state with the fetched data.

    - *Functionality:* Ensures that the Cookie Jar displays the latest cookie data from the database upon initial load and when data changes.
    - *Screenshot:* `fetchCookies` function within the `useEffect` hook in `components\CookieJar.tsx`

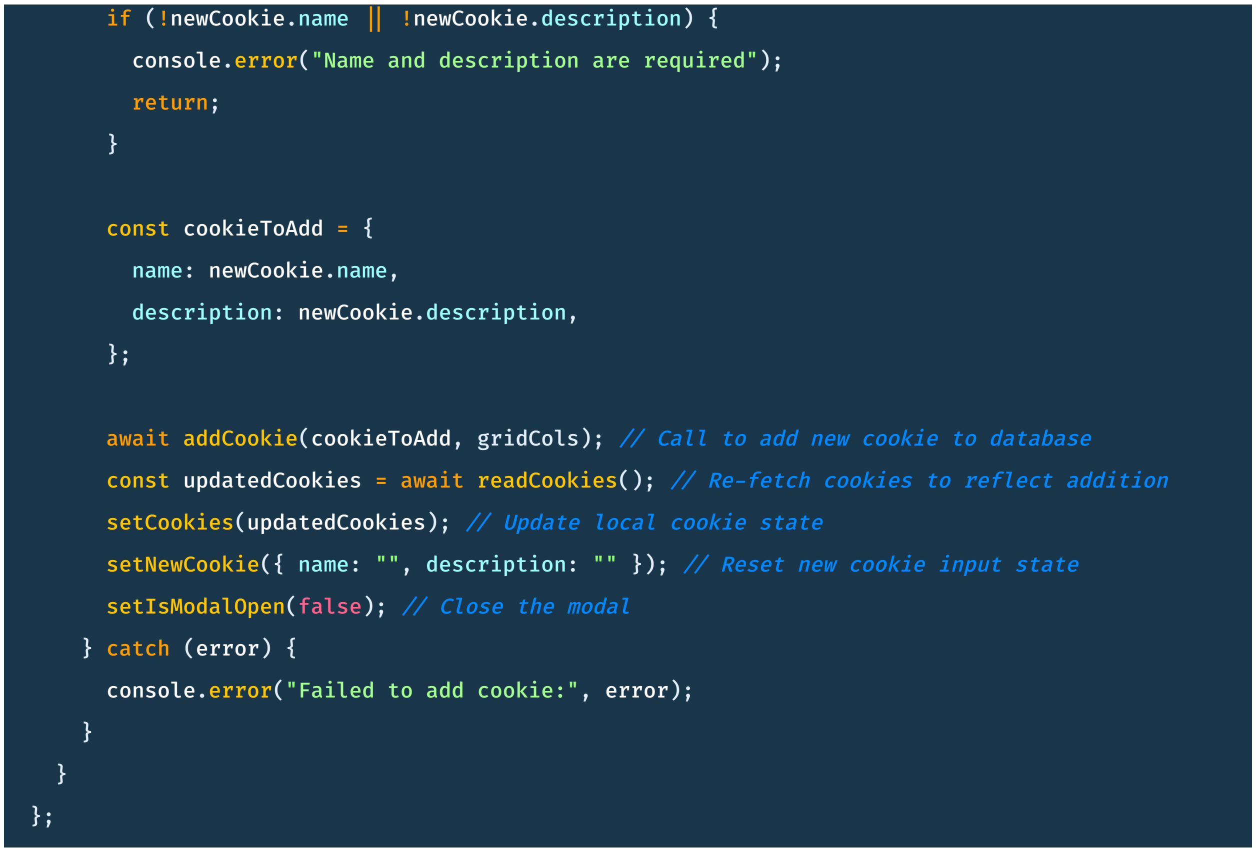
```
useEffect(() => {
    async function fetchCookies() {
      const fetchedCookies = await readCookies(); // Call to fetch cookies from database
      setCookies(fetchedCookies); // Update cookies state with fetched data
    }
    fetchCookies();
  }, []);
```

- **`handleSubmit`**: This asynchronous function handles the form submission for both adding and updating cookies in the modal. It determines whether to call `addCookie` or `updateCookie` based on the `isEditing` state, interacts with the Firestore database to persist changes, re-fetches cookies to update the UI, and closes the modal.

    - *Functionality:* Manages the data submission process for creating new cookies and saving edits to existing ones, orchestrating database updates and UI refreshes.
    - *Screenshot:* `handleSubmit` function in `components\CookieJar.tsx`

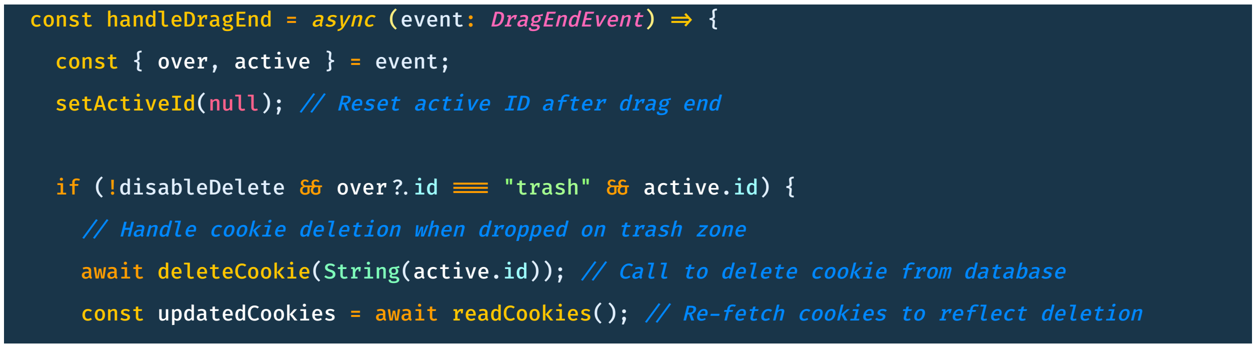


```
const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    if (isEditing && editingCookieId) {
      // Handle cookie update
      if (disableEdit) return; // Prevent update if editing is disabled

      const updatedCookie: Cookie = {
        ...newCookie,
        id: editingCookieId,
      } as Cookie;

      try {
        await updateCookie(updatedCookie); // Call to update cookie in database
        const updatedCookies = await readCookies(); // Re-fetch cookies to reflect changes
        setCookies(updatedCookies); // Update local cookie state
      } catch (error) {
        console.error("Failed to update cookie:", error);
      } finally {
        setIsEditing(false); // Reset edit mode state
        setEditingCookieId(null); // Clear editing cookie ID
        setNewCookie({ name: "", description: "" }); // Reset new cookie input state
        setIsModalOpen(false); // Close the modal
      }
    } else {
      // Handle new cookie addition
      if (disableAdd) return; // Prevent add if adding is disabled

      try {
```



```
      if (!newCookie.name || !newCookie.description) {
        console.error("Name and description are required");
        return;
      }

      const cookieToAdd = {
        name: newCookie.name,
        description: newCookie.description,
      };

      await addCookie(cookieToAdd, gridCols); // Call to add new cookie to database
      const updatedCookies = await readCookies(); // Re-fetch cookies to reflect addition
      setCookies(updatedCookies); // Update local cookie state
      setNewCookie({ name: "", description: "" }); // Reset new cookie input state
      setIsModalOpen(false); // Close the modal
    } catch (error) {
      console.error("Failed to add cookie:", error);
    }
  }
};
```

- **`handleDragEnd`**: This asynchronous function is triggered after a drag-and-drop operation ends within the `DndContext` provider. It determines if a cookie was dropped on the "trash" droppable zone for deletion, or if cookies were reordered within the grid. It updates the cookie positions in the database using `updateCookiePositions` and refreshes the UI by re-fetching cookies.
    - *Functionality:* Handles the logic for persisting cookie reordering and deletion actions triggered by drag and drop, updating the database and UI accordingly.
    - *Screenshot:* `handleDragEnd` function in `components\CookieJar.tsx`



```
const handleDragEnd = async (event: DragEndEvent) => {
  const { over, active } = event;
  setActiveId(null); // Reset active ID after drag end

  if (!disableDelete && over?.id === "trash" && active.id) {
    // Handle cookie deletion when dropped on trash zone
    await deleteCookie(String(active.id)); // Call to delete cookie from database
    const updatedCookies = await readCookies(); // Re-fetch cookies to reflect deletion
```

```
      setCookies(updatedCookies); // Update local cookie state
      return;
    }

    setCookies((items) => {
        const oldIndex = items.findIndex((item) => item.id === active.id); // Find the original index of the dragged cookie
        const newIndex = items.findIndex((item) => item.id === over?.id); // Find the new index based on drop target

      if (oldIndex === -1 || newIndex === -1) {
        return items; // Return original items if indices are invalid
      }

      const reorderedItems = arrayMove(items, oldIndex, newIndex); // Reorder items array

      // Update positions based on new order in the grid
      const updatedItemsWithPositions = reorderedItems.map((item, index) => ({
        ...item,
        position: {
          x: index % gridCols, // Calculate x position based on column
          y: Math.floor(index / gridCols), // Calculate y position based on row
        },
      }));

        updateCookiePositions(updatedItemsWithPositions); // Call to update cookie positions in database
      return updatedItemsWithPositions; // Return updated items array
    });
  };
```

-

Doubt Tracker & Curiosity Space (Idea Tracker)

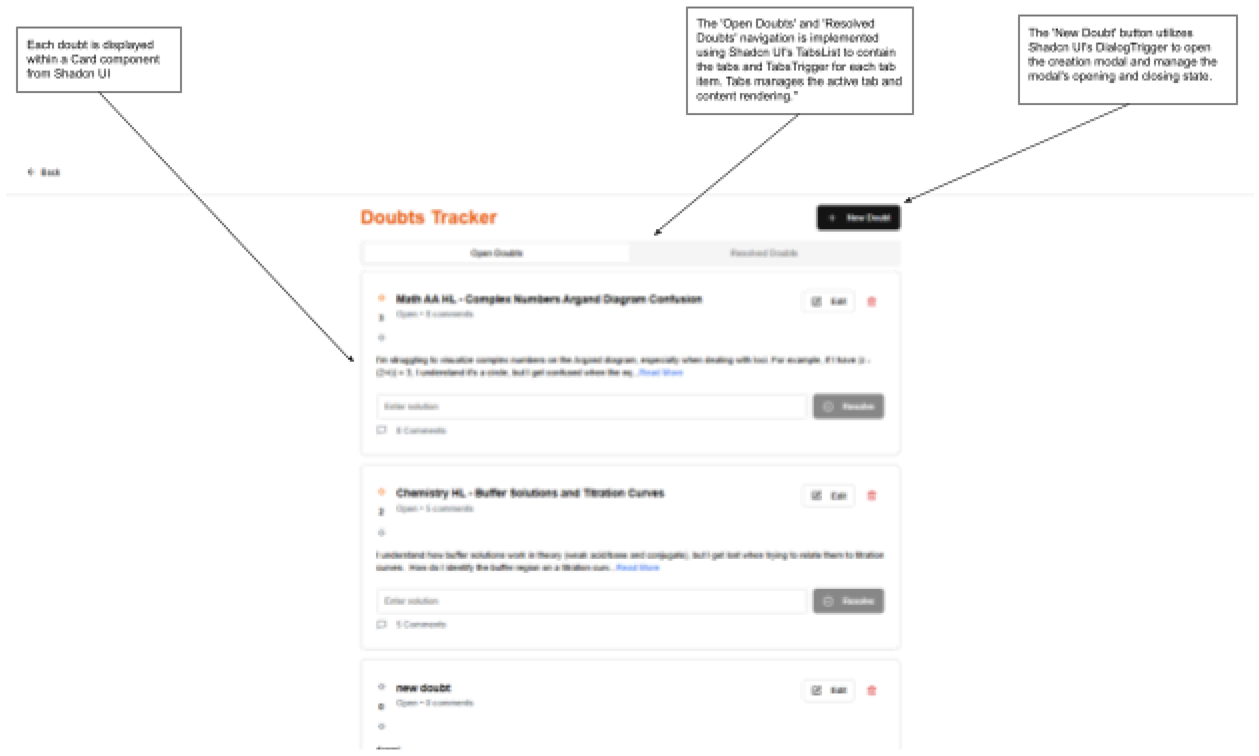
The Doubt Tracker and Curiosity Space (Idea Tracker) modules serve as dedicated spaces for users to log, manage, and resolve their doubts and ideas, respectively. While distinct in purpose – one for addressing uncertainties and the other for capturing and developing innovative thoughts – they share a common architectural structure and core functionalities. Both modules empower users to:

- **Capture Content**: Create new entries for doubts or ideas, specifying a title and detailed description via a creation modal.
- **Manage Status**: Track the progress of doubts and ideas, marking them as 'Open' or 'Resolved'. Resolved items can include a solution for future reference.
- **Engage in Discussion**: Foster collaboration and clarification through a commenting system, allowing users to add, edit, and delete text-based comments, including nested replies for in-depth discussions.
- **Indicate Importance**: Utilize a voting system (upvotes and downvotes) to signal the perceived importance or value of doubts and ideas within the community or for personal prioritization.
- **Organize & Review**: View doubts and ideas in categorized lists (Open and Resolved tabs), facilitating focused review and management of each type of entry.
- **Edit and Delete**: Modify or remove doubts and ideas as needed, ensuring the system remains current and relevant to the user's evolving needs.

These modules are designed to promote active engagement with learning and creative processes, providing structured environments to address challenges and nurture innovative thinking.

Screenshots:

- **Doubt Tracker Page Screenshot: Since the two components mirror each other, only one will be used/shown for explanation purposes.**



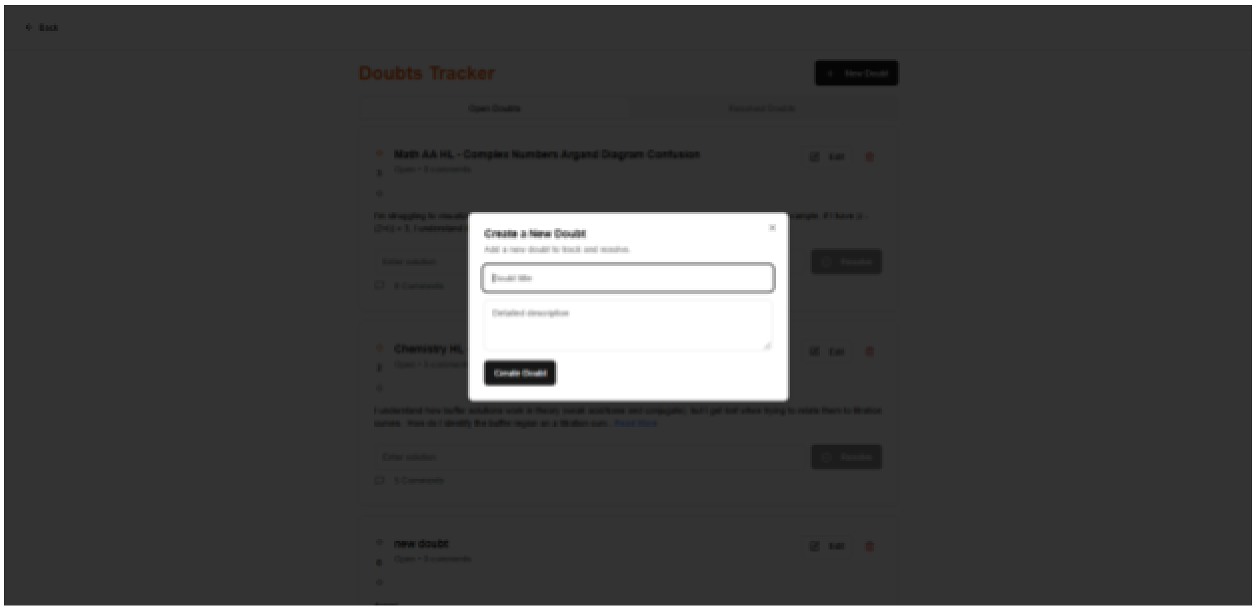
Each doubt is displayed within a Card component from Shadcn UI

The 'Open Doubts' and 'Resolved Doubts' navigation is implemented using Shadcn UI's TabsList to contain the tabs and TabsTrigger for each tab item. Tabs manages the active tab and content rendering."

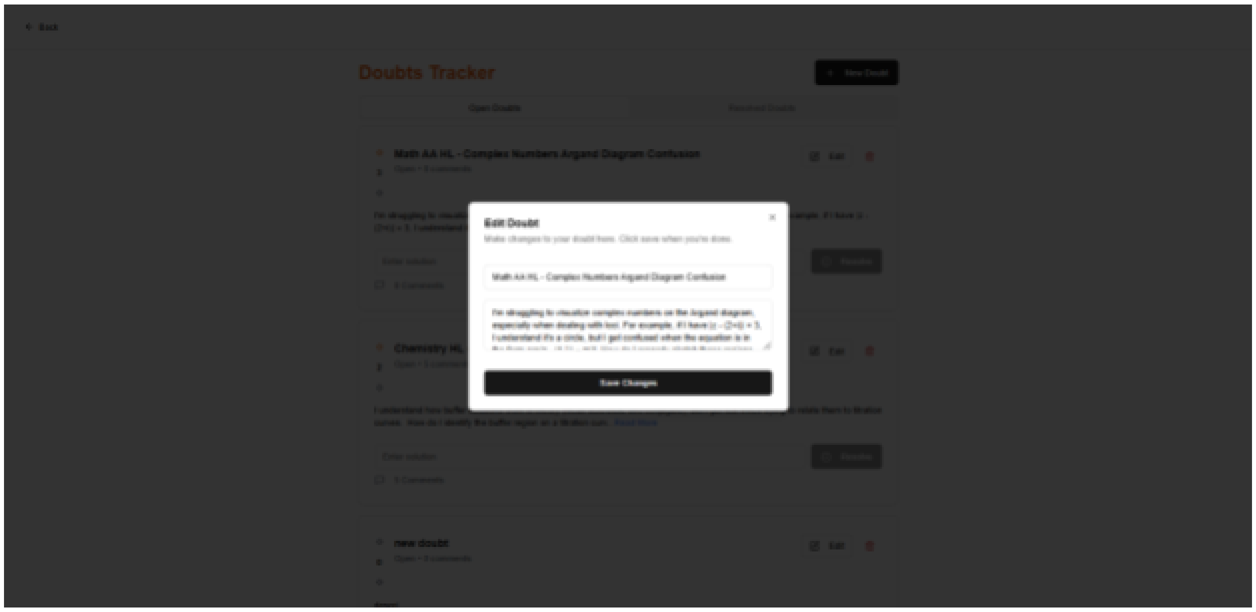
The 'New Doubt' button utilizes Shadcn UI's DialogTrigger to open the creation modal and manage the modal's opening and closing state.

Back
Doubts Tracker
New Doubt
Open Doubts
Resolved Doubts
Math AA HL - Complex Numbers Argand Diagram Confusion
Edit
Enter solution
Resolve
Chemistry HL - Buffer Solutions and Titration Curves
new doubt

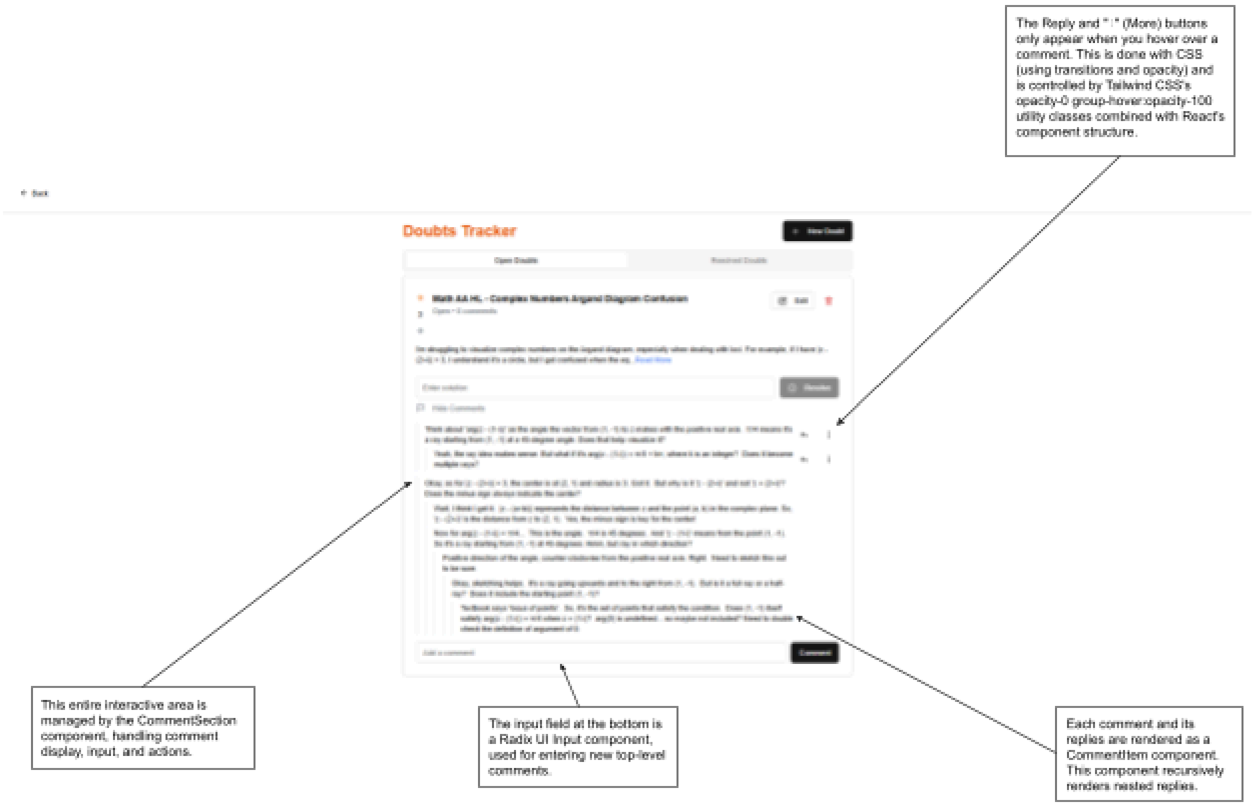
- **Creation Modal Screenshot:**



- **Edit Modal Screenshot:**


Back
Doubts Tracker
New Doubt
Open Doubts
Resolved Doubts
Math AA HL - Complex Numbers Argand Diagram Confusion
Edit
Edit Doubt
Make changes to your doubt here. Click save when you're done.
Math AA HL - Complex Numbers Argand Diagram Confusion
Save Changes
Chemistry HL
Enter solution
Resolve
new doubt

- **Comment Section Screenshot:**



The Reply and "⋮" (More) buttons only appear when you hover over a comment. This is done with CSS (using transitions and opacity) and is controlled by Tailwind CSS's opacity-0 group-hover:opacity-100 utility classes combined with React's component structure.

This entire interactive area is managed by the CommentSection component, handling comment display, input, and actions.

The input field at the bottom is a Radix UI Input component, used for entering new top-level comments.

Each comment and its replies are rendered as a CommentItem component. This component recursively renders nested replies.

Techniques used:

- **React Components**: Both Doubt Tracker and Curiosity Space are built using React components, sharing a similar component structure for list rendering (`DoubtList/IdeaList`), individual item display (`DoubtCard/IdeaCard`), comment sections (`CommentSection`), and creation forms (`NewDoubtForm/NewIdeaForm`). This highlights the reusability and modularity achieved through React's component architecture.
    - *Reasoning:* Component reusability reduces code duplication and development time, while modularity improves maintainability and scalability of the application.
- **JSX (JavaScript XML)**: JSX is used to define the UI structure for both modules, providing a declarative and readable syntax for creating components and their layouts. The consistency in JSX usage across both modules further emphasizes their architectural similarity.
    - *Reasoning:* JSX makes the UI code more intuitive and easier to work with, enhancing developer productivity and code clarity.
- **State Management (useState, useEffect)**: `useState` hooks are used extensively in both modules to manage UI states such as lists of doubts/ideas, modal visibility, input values in forms, and comment section expansion. `useEffect` hooks are utilized to fetch data from Firestore and set up real-time data synchronization, demonstrating a consistent approach to data handling.
    - *Reasoning:* State management is crucial for dynamic UIs, allowing components to react to user interactions and data changes. Consistent state management patterns across modules simplify development and maintenance.
- **Asynchronous Operations (Promises, Async/Await)**: Asynchronous operations are fundamental for interacting with Firestore in both modules. Functions for CRUD operations (Create, Read, Update, Delete) on doubts, ideas, and comments are implemented using Promises and `async/await`, ensuring efficient and non-blocking data handling.
    - *Reasoning:* Asynchronous operations are essential for web applications interacting with databases, ensuring responsiveness and preventing UI freezes during data operations.
- **Database Interaction (Firestore)**: Both modules rely on Google Firestore for data persistence, utilizing the same Firebase setup and Firestore API. Firestore is used to store and retrieve doubts, ideas, and their associated comments, showcasing a unified backend strategy for both modules.

    - *Reasoning:* Firestore provides a robust and scalable NoSQL database, allowing for real-time data synchronization and efficient data management, critical for the interactive features of both modules.
- **Real-time Data Synchronization (Firestore `onSnapshot`)**: `onSnapshot` listeners are implemented in both `DoubtTracker.tsx` and `IdeaTracker.tsx` within `useEffect` hooks to provide real-time updates from Firestore. This ensures that changes made by any user are immediately reflected across all active sessions, enhancing collaboration and data consistency.
    - *Reasoning:* Real-time data synchronization with `onSnapshot` provides a dynamic and collaborative user experience, ensuring that users always see the most up-to-date information without manual refreshes.
- **Recursion (for Nested Comments)**: The `CommentSection` component in both modules implements recursion to render nested comments and replies. This allows for creating threaded conversations and managing hierarchical comment structures efficiently.
    - *Reasoning:* Recursion is an elegant and efficient algorithmic technique for handling hierarchical data structures like nested comments, simplifying the code needed to render and manage complex comment threads.
- **Conditional Rendering**: Conditional rendering is extensively used within both modules to manage UI variations based on data and user interactions. Examples include: displaying different content based on doubt/idea status ('Open' vs 'Resolved' tabs), showing edit or view modes for doubts/ideas and comments, and conditionally rendering UI elements based on component state.
    - *Reasoning:* Conditional rendering allows for creating dynamic and context-aware UIs that adapt to different states and user actions, improving the user experience and interface clarity.
- **UI Libraries (Radix UI, Tailwind CSS, Lucide React)**: Both modules consistently utilize Radix UI for UI primitives like `Tabs`, `Dialog`, and other components, Tailwind CSS for styling, and Lucide React for icons (Plus, Edit, Trash2, etc.). This uniform use of UI libraries contributes to a cohesive visual style and consistent user experience across both modules.
    - *Reasoning:* Consistent use of UI libraries ensures a unified design language and reduces development effort by leveraging pre-built, accessible, and well-styled components.
- **Custom Hooks (useUser, useToast)**: Custom hooks like `useUser` (for Clerk authentication context) and `useToast` (for displaying toast notifications) are consistently used across both modules. While not explicitly shown in the provided snippets *for these specific components*, they are used throughout the project and are relevant to the overall architecture and techniques employed.

This demonstrates the application of custom hooks for encapsulating and reusing cross-cutting concerns.

- *Reasoning:* Custom hooks promote code reusability and separation of concerns, making the codebase cleaner, more maintainable, and easier to test by encapsulating specific functionalities and logic.

Imports used:

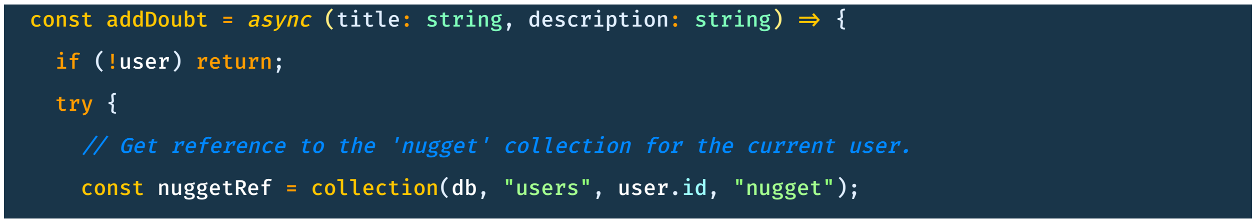
Import statements from `components\DoubtTracker.tsx`

```
import React, { useState, useEffect } from "react";
import DoubtList from "@/components/doubts-tracker/DoubtList";
import NewDoubtForm from "@/components/doubts-tracker/NewDoubtForm";
import { Button } from "@/components/ui/button";
import { Tabs, TabsContent, TabsList, TabsTrigger } from "@/components/ui/tabs";
import { Dialog, DialogContent, DialogDescription, DialogHeader, DialogTitle, DialogTrigger }
from "@/components/ui/dialog";
import { Plus } from "lucide-react";
import { useUser } from "@clerk/nextjs";
import { db } from "@/lib/firebase";
import { collection, addDoc, updateDoc, deleteDoc, query, orderBy, onSnapshot, increment,
serverTimestamp, doc } from "firebase/firestore";
```

Functions used:

- **`addDoubt` / `addIdea`**: These asynchronous functions (one in `DoubtTracker.tsx`, one in `IdeaTracker.tsx`, but structurally similar) handle the creation of new doubts or ideas in Firestore. They interact with the Firestore database to add a new document to the respective "nugget" or "ideas" collection, setting initial values and using `serverTimestamp()` for creation time.

  - *Functionality:* Persists new doubt or idea entries to the database when users submit the creation form.
  - *Screenshot:* `addDoubt` or `addIdea` function from `components\DoubtTracker.tsx`



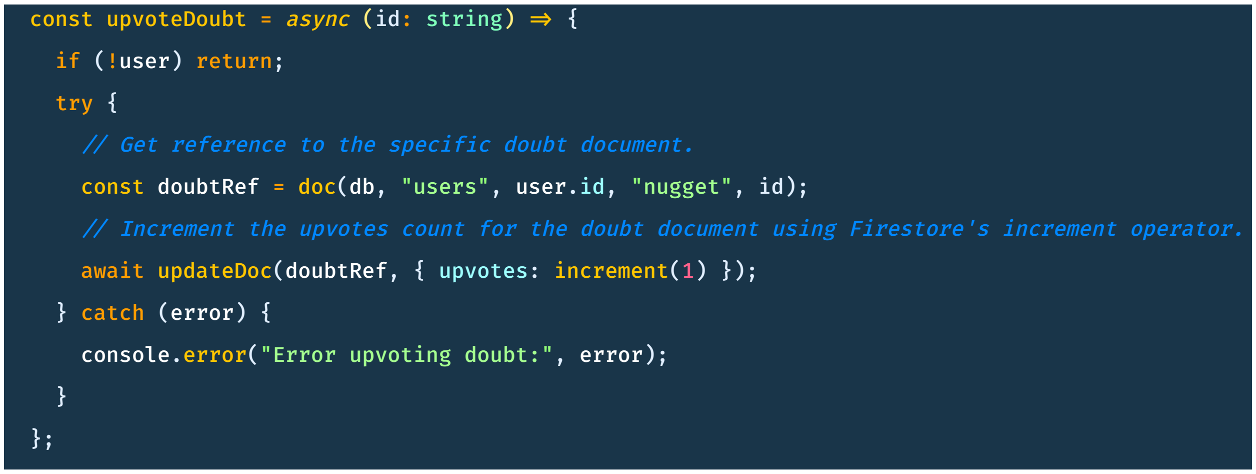
```
const addDoubt = async (title: string, description: string) ⇒ {
  if (!user) return;
  try {
    // Get reference to the 'nugget' collection for the current user.
    const nuggetRef = collection(db, "users", user.id, "nugget");
    // Add a new document to the 'nugget' collection with the provided title and description.
    await addDoc(nuggetRef, {
      title,
      description,
      upvotes: 0,
      downvotes: 0,
      resolved: false,
      solution: "",
      createdAt: serverTimestamp(), // Use Firestore server timestamp for creation time.
    });
  } catch (error) {
    console.error("Error creating doubt:", error);
  }
};
```

- **`resolveDoubt`** / **`resolveIdea`**: These asynchronous functions (again, one in each component file and structurally similar) manage the status update of doubts and ideas to 'Resolved'. They interact with Firestore to update the corresponding document, setting the `resolved` field to `true` and storing the provided `solution`.
    - *Functionality:* Handles the user action of marking a doubt or idea as resolved, updating its status and solution in the database.
    - *Screenshot:* `resolveDoubt` function from `components\DoubtTracker.tsx`

```
const resolveDoubt = async (id: string, solution: string) => {
  if (!user) return;
  try {
    // Get reference to the specific doubt document within the user's 'nugget' collection.
    const doubtRef = doc(db, "users", user.id, "nugget", id);
    // Update the doubt document to mark it as resolved and add the provided solution.
    await updateDoc(doubtRef, { resolved: true, solution });
  } catch (error) {
    console.error("Error resolving doubt:", error);
  }
};
```

- **`upvoteDoubt` / `upvoteIdea`**: These asynchronous functions implement the voting functionality. When called, they interact with Firestore to increment the `upvotes` count for the specific doubt or idea document, using the Firestore `increment()` operator for atomic updates.

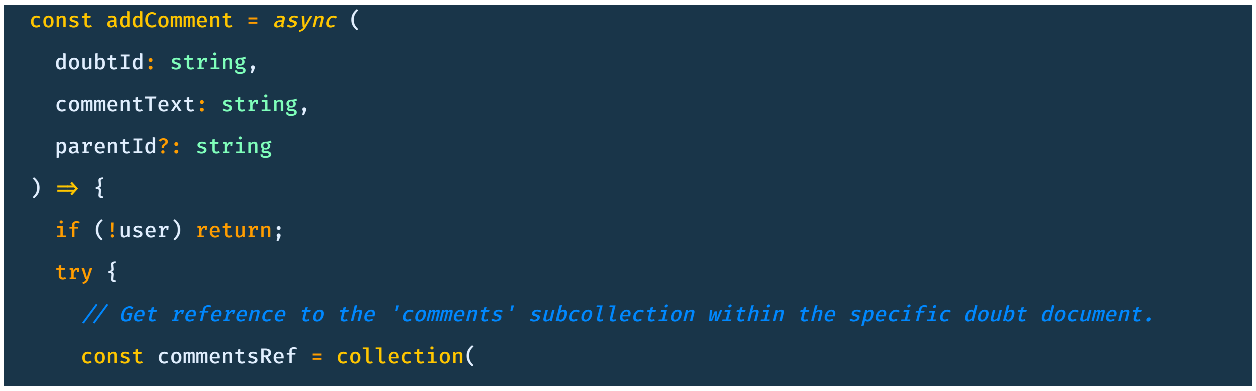
    - *Functionality:* Manages user upvotes, updating the vote count in the database in real-time.
    - *Screenshot:* `upvoteDoubt` function from `components\DoubtTracker.tsx`



```
const upvoteDoubt = async (id: string) => {
  if (!user) return;
  try {
    // Get reference to the specific doubt document.
    const doubtRef = doc(db, "users", user.id, "nugget", id);
    // Increment the upvotes count for the doubt document using Firestore's increment operator.
    await updateDoc(doubtRef, { upvotes: increment(1) });
  } catch (error) {
    console.error("Error upvoting doubt:", error);
  }
};
```

- **`addComment`**: This asynchronous function, shared conceptually between both modules, handles adding new comments to doubts or ideas. It interacts with Firestore to add a new document to the "comments" subcollection of the relevant doubt or idea, including the comment text and parent comment ID (if it's a reply).

    - *Functionality:* Persists new comments to the database, handling both top-level comments and replies within comment threads.
    - *Screenshot:* `addComment` function from `components\DoubtTracker.tsx`



```
const addComment = async (
  doubtId: string,
  commentText: string,
  parentId?: string
) => {
  if (!user) return;
  try {
    // Get reference to the 'comments' subcollection within the specific doubt document.
    const commentsRef = collection(
        db,
        "users",
        user.id,
        "nugget",
        doubtId,
        "comments"
      );
      // Add a new comment document to the 'comments' subcollection.
      await addDoc(commentsRef, {
        text: commentText,
            parentId: parentId || null, // Store parentId if it's a reply, otherwise null for
top-level comments.
           createdAt: serverTimestamp(), // Use Firestore server timestamp for comment creation
time.
      });
    } catch (error) {
      console.error("Error adding comment:", error);
    }
  };
```

- **`useEffect` (for data fetching and real-time updates)**: The `useEffect` hook in both `DoubtTracker.tsx` and `IdeaTracker.tsx` (again, structurally very similar) is responsible for fetching initial doubt/idea data from Firestore and setting up a real-time listener using `onSnapshot`. This hook demonstrates a consistent pattern for data management and synchronization across both modules.
    - *Functionality:* Ensures that the doubt and idea lists are populated with data from the database on component mount and are kept up-to-date with real-time changes.
    - *Screenshot:* `useEffect` hook from either `components\DoubtTracker.tsx`

```
  useEffect(() ⇒ {
    if (!user) return;
    const nuggetRef = collection(db, "users", user.id, "nugget");
    const q = query(nuggetRef, orderBy("createdAt", "desc"));
    const unsubscribe = onSnapshot(q, (snapshot) ⇒ {
      const loadedDoubts: Doubt[] = [];
      snapshot.forEach((docSnap) ⇒ {
        loadedDoubts.push({
          id: docSnap.id,
          title: docSnap.data().title,
          description: docSnap.data().description,
          upvotes: docSnap.data().upvotes,
          downvotes: docSnap.data().downvotes,
          resolved: docSnap.data().resolved,
          solution: docSnap.data().solution,
          comments: []
        });
      });
      setDoubts(loadedDoubts);
    });
    return () => unsubscribe();
  }, [user]);
```

-

Notebooks / Continuous Information Space

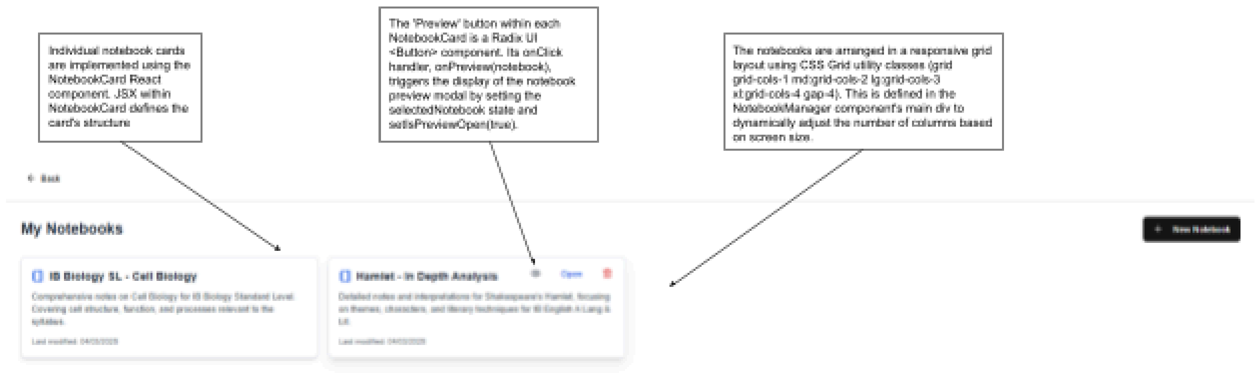
The Continuous Information Space module, often referred to as "Notebooks," provides a hierarchical and flexible system for users to organize and manage information. It is structured into two main views:

- **Notebooks Main Page (Notebook Management)**: This page serves as the central hub for managing all notebooks. Users can view a list of their notebooks, create new notebooks, preview existing ones, and open notebooks to access their content. This view facilitates organization and quick access to different information spaces.
- **Notebook View (Specific Notebook Page)**: Upon opening a notebook, users are directed to this view, which displays the contents of a specific notebook in a structured layout. Within a notebook, information is organized hierarchically into sections, columns, and notes. This view allows users to:
    - Create and manage notebooks, sections, columns, and notes.
    - Reorder notes within columns using drag-and-drop.
    - Archive sections for long-term storage and decluttering, with options to preview and unarchive them.

This module is designed to mirror a digital notebook, providing a structured and adaptable space for note-taking, research, and information organization within the application.

Screenshots:

- **Notebooks Main Page (Notebook Management) Screenshot:**


Individual notebook cards are implemented using the NotebookCard React component. JSX within NotebookCard defines the card's structure

The 'Preview' button within each NotebookCard is a Radix UI <Button> component. Its onClick handler, onPreview(notebook), triggers the display of the notebook preview modal by setting the selectedNotebook state and setIsPreviewOpen(true).

The notebooks are arranged in a responsive grid layout using CSS Grid utility classes (grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4 gap-4). This is defined in the NotebookManager component's main div to dynamically adjust the number of columns based on screen size.

Back

My Notebooks

New Notebook

IB Biology SL - Cell Biology

Comprehensive notes on Cell Biology for IB Biology Standard Level. Covering cell structure, function, and processes relevant to the syllabus.

Last modified: 04/03/2025

Hamlet - In Depth Analysis

Open

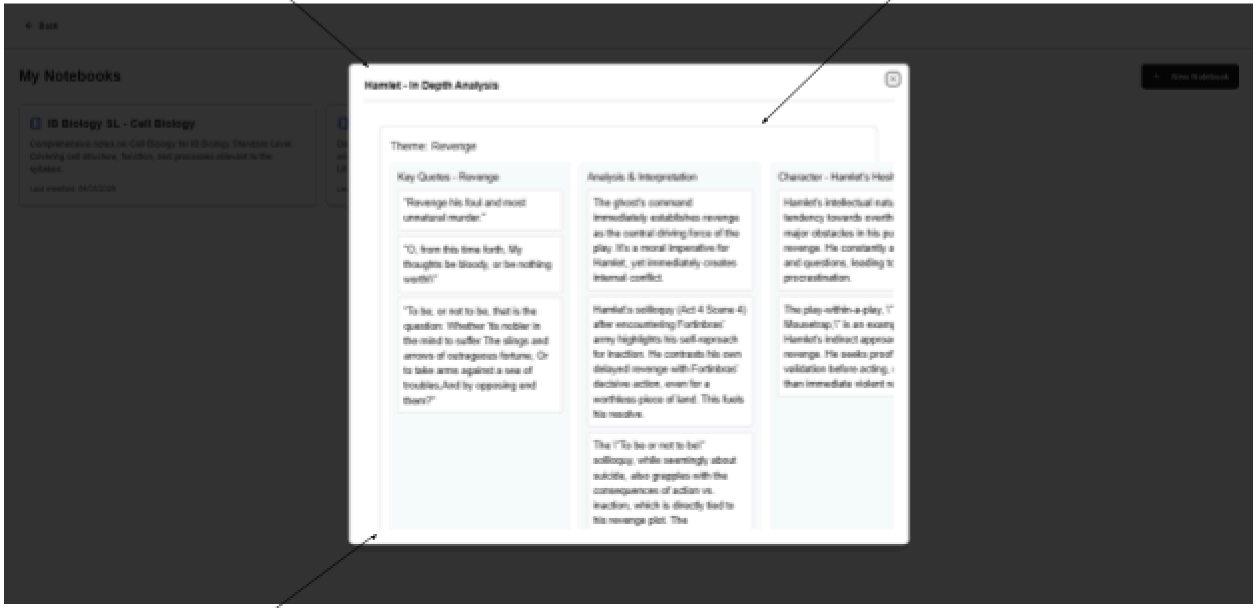
Detailed notes and interpretations for Shakespeare's Hamlet, focusing on themes, characters, and literary techniques for IB English A Lang & Lit.

Last modified: 04/03/2025

- **Notebooks Main Page notebook preview Screenshot:**

The Notebook Preview is implemented using Radix UI <Dialog> and <DialogContent> components. The visibility of this modal is controlled by the isPreviewOpen state in the NotebookManager component.

Sections, Columns, and Notes within the preview are rendered using nested .map() functions within the <DialogContent>. The structure mirrors the hierarchical data structure of notebooks, sections, columns, and notes. Data is accessed from the selectedNotebook.sections state.



IB Biology SL - Cell Biology

Comprehensive notes on Cell Biology for IB Biology Standard Level. Covering cell structure, function, and processes relevant to the syllabus.

Last modified: [illegible]

Hamlet - In Depth Analysis

Theme: Revenge

Key Quotes - Revenge

"Revenge his foul and most unnatural murder."

"O, from this time forth, My thoughts be bloody, or be nothing worth!"

"To be, or not to be, that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a sea of troubles,And by opposing end them?"

Analysis & Interpretation

The ghost's command immediately establishes revenge as the central driving force of the play. It's a moral imperative for Hamlet, yet immediately creates internal conflict.

Hamlet's soliloquy (Act 4 Scene 4) after encountering Fortinbras' army highlights his self-reproach for inaction. He contrasts his own delayed revenge with Fortinbras' decisive action, even for a worthless piece of land. This fuels his resolve.

The \"To be or not to be\" soliloquy, while seemingly about suicide, also grapples with the consequences of action vs. inaction, which is directly tied to his revenge plot. The

Character - Hamlet's Hesi

Hamlet's intellectual natu tendency towards overth major obstacles in his pu revenge. He constantly a and questions, leading to procrastination.

The play-within-a-play, \" Mousetrap,\" is an examp Hamlet's indirect approa revenge. He seeks proof validation before acting, than immediate violent a

The content area within the <DialogContent> is made scrollable using CSS overflow-auto class. This ensures that long notebook previews can be viewed without overflowing the modal.

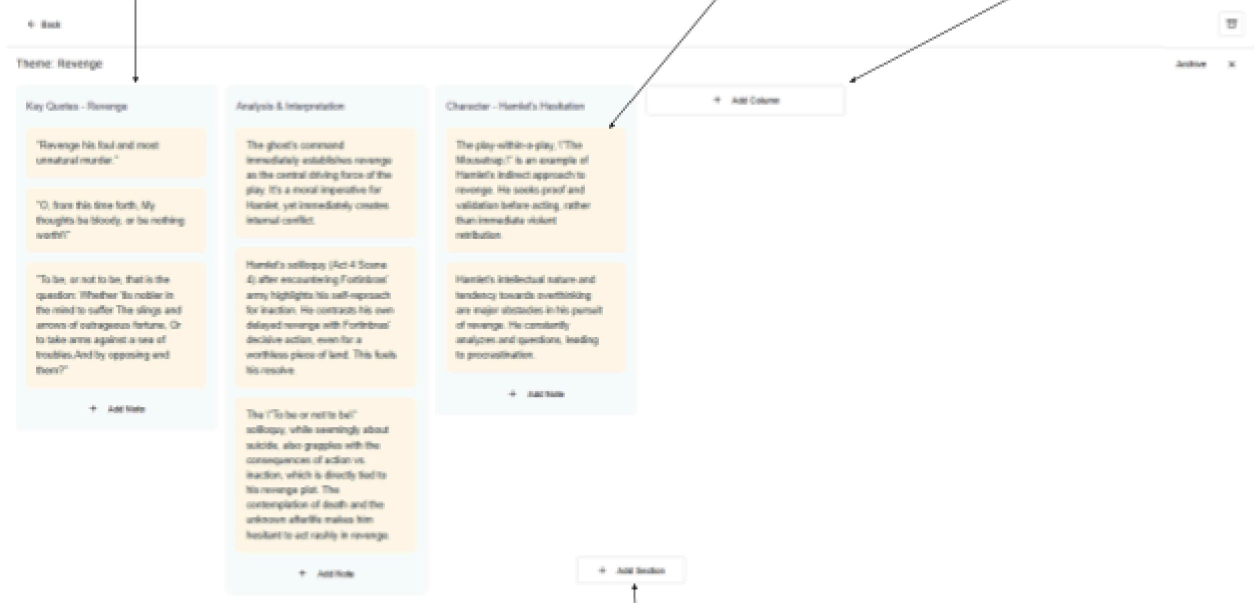
- **Notebook View (Specific Notebook Page) Screenshot:**

Sections and Columns are laid out horizontally using Flexbox (flex space-x-4 min-h-[200px] pb-4) within the TodoListInterface component, enabling horizontal scrolling for notebook content.

Each column (representing a section's column) is rendered using the reused and adapted DraggableColumn component from the ToDoList module. This component, along with DroppableColumn, enables drag-and-drop reordering of columns.

Individual notes within columns are rendered using the reused and adapted DraggableNote component. These components, rendered within DroppableColumn, enable drag-and-drop reordering of notes within columns and moving notes between columns.

The 'Add Column' button within each section, also a <Button> component, uses the addColumn(section.id) handler. This adds a new column to the specific section's columns array in the component's state, dynamically rendering a new column within that section.



Theme: Revenge

Key Quotes - Revenge

"Revenge his foul and most unnatural murder."

"O, from this time forth, My thoughts be bloody, or be nothing worth!"

"To be, or not to be, that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a sea of troubles,And by opposing end them?"

Analysis & Interpretation

The ghost's command immediately establishes revenge as the central driving force of the play. It's a moral imperative for Hamlet, yet immediately creates internal conflict.

Hamlet's soliloquy (Act 4 Scene 4) after encountering Fortinbras' army highlights his self-reproach for inaction. He contrasts his own delayed revenge with Fortinbras' decisive action, even for a worthless piece of land. This fuels his resolve.

The ("To be or not to be)" soliloquy, while seemingly about suicide, also grapples with the consequences of action vs. inaction, which is directly tied to his revenge plot. The contemplation of death and the unknown afterlife makes him hesitant to act rashly in revenge.

Character - Hamlet's Hesitation

The play-within-a-play, ("The Mousetrap") is an example of Hamlet's indirect approach to revenge. He seeks proof and validation before acting, rather than immediate violent retribution.

Hamlet's intellectual nature and tendency towards overthinking are major obstacles in his pursuit of revenge. He constantly analyzes and questions, leading to procrastination.

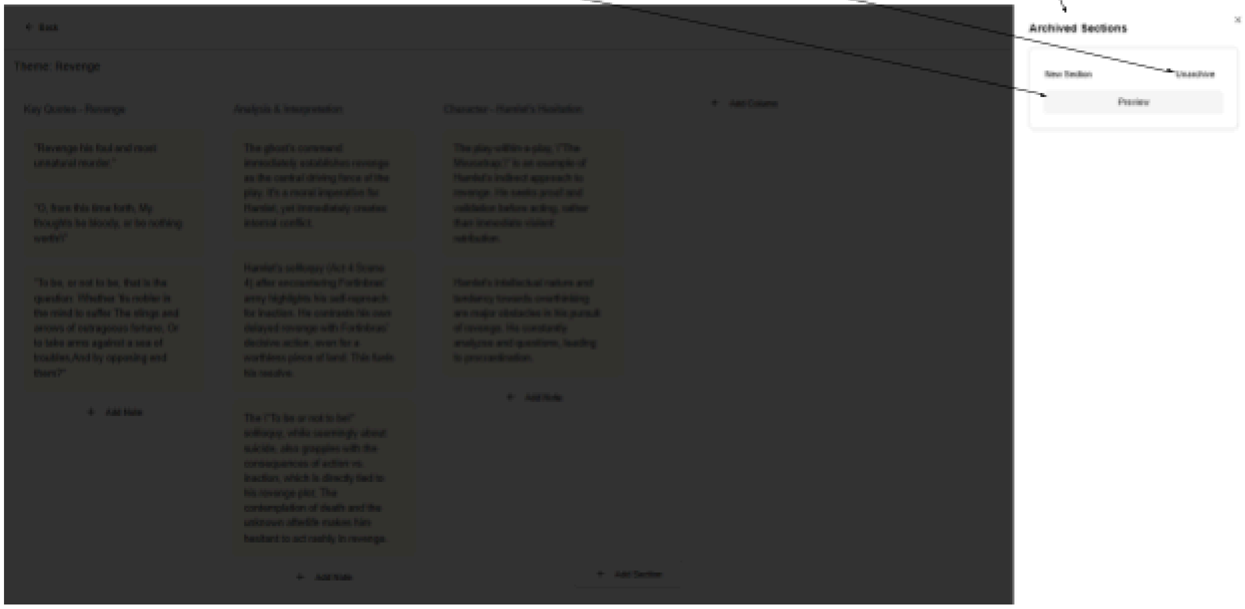
The 'Add Section' button, implemented using <Button>, is positioned at the end of the section list. Its onClick handler, addSection(), adds a new section to the sections state, dynamically rendering a new section column.

- **Archived Panel Screenshot:**

The 'Preview' button within each archived section's <CardContent> is a Radix UI <Button> component. Its onClick handler, previewArchivedSection(section), opens the Archived Section Preview modal, displaying a preview of the archived section's content.

The 'Unarchive' button within each archived section's <CardHeader> is a Radix UI <Button> component. Its onClick handler, unarchiveSection(section.id), removes the section from the archivedSections state and adds it back to the main sections state, effectively unarchiving the section.

The 'Archived Sections' panel is implemented using Shadcn UI <Sheet> and <SheetContent> components. The <SheetTrigger> component (Archive icon button) controls the panel's visibility, toggling the Sheet open and closed.



"Revenge his foul and most unnatural murder."

"O, from this time forth, My thoughts be bloody, or be nothing worth!"

"To be, or not to be, that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a sea of troubles,And by opposing end them?"

The ghost's command immediately establishes revenge as the central driving force of the play. It's a moral imperative for Hamlet, yet immediately creates internal conflict.

Hamlet's soliloquy (Act 4 Scene 4) after encountering Fortinbras' army highlights his self-reproach for inaction. He contrasts his own delayed revenge with Fortinbras' decisive action, even for a worthless piece of land. This fuels his resolve.

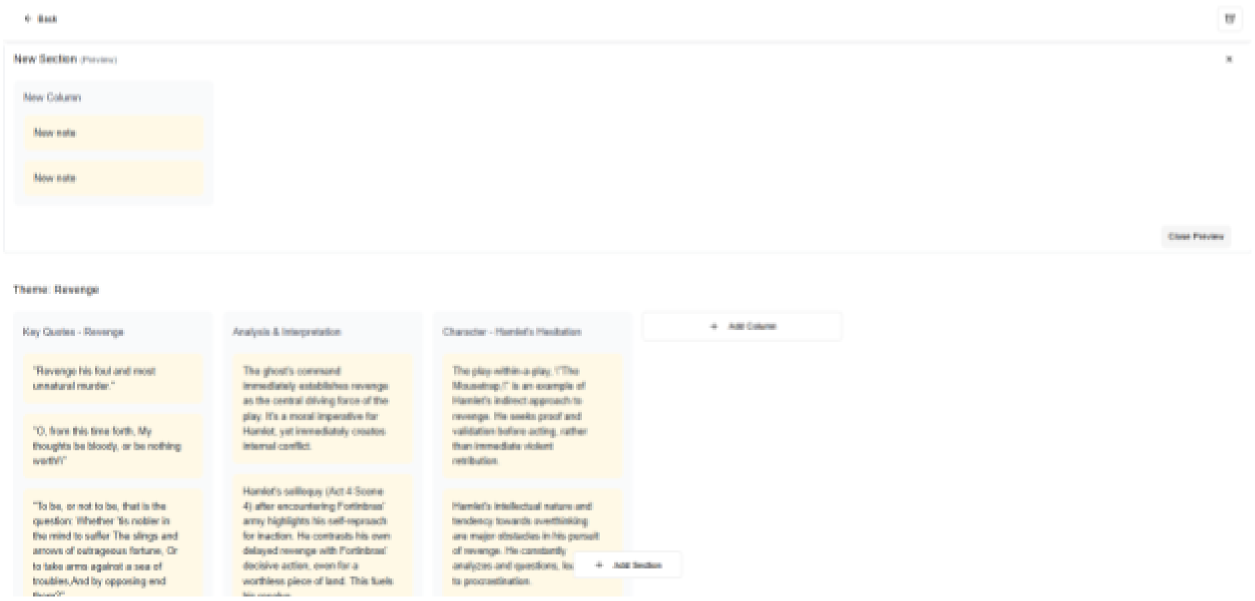
The ("To be or not to be") soliloquy, while seemingly about suicide, also grapples with the consequences of action vs. inaction, which is directly tied to his revenge plot. The contemplation of death and the unknown afterlife makes him hesitant to act rashly in revenge.

The play-within-a-play, ("The Mousetrap)" is an example of Hamlet's indirect approach to revenge. He seeks proof and validation before acting, rather than immediate violent retribution.

Hamlet's intellectual nature and tendency towards overthinking are major obstacles in his pursuit of revenge. He constantly analyzes and questions, leading to procrastination.

- **Archived Section Preview Screenshot:**

The entire Archived Section Preview is displayed using Radix UI's <Dialog> and <DialogContent> components. The modal's visibility is controlled by the previewSection state; when previewSection is set to a valid section object (through actions like clicking 'Preview' in the Archived Sections panel), React conditionally renders this <Dialog> component, making the preview UI visible on the page. If previewSection is null, the <Dialog> is not rendered, and the preview is hidden.



"Revenge his foul and most unnatural murder."

"O, from this time forth, My thoughts be bloody, or be nothing worth!"

"To be, or not to be, that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a sea of troubles,And by opposing end them?"

The ghost's command immediately establishes revenge as the central driving force of the play. It's a moral imperative for Hamlet, yet immediately creates internal conflict.

Hamlet's soliloquy (Act 4 Scene 4) after encountering Fortinbras' army highlights his self-reproach for inaction. He contrasts his own delayed revenge with Fortinbras' decisive action, even for a worthless piece of land. This fuels his resolve.

The play-within-a-play, ("The Mousetrap)" is an example of Hamlet's indirect approach to revenge. He seeks proof and validation before acting, rather than immediate violent retribution.

Hamlet's intellectual nature and tendency towards overthinking are major obstacles in his pursuit of revenge. He constantly analyzes and questions, leading to procrastination.

Techniques used:

Notebooks Main Page (Notebook Management):

- **React Components**: The Notebooks Main Page is constructed using React components, including `NotebookManager` (the main component for this page) and `NotebookCard` (for displaying individual notebooks in the list). This component-based structure facilitates modularity and organization of the UI.
    - *Reasoning:* React components are used to break down the UI into manageable and reusable parts, improving code maintainability and readability.
- **JSX (JavaScript XML)**: JSX is used to define the UI structure of the Notebooks Main Page, enabling declarative and efficient rendering of the notebook list and related UI elements.
    - *Reasoning:* JSX simplifies UI development by allowing HTML-like syntax within JavaScript, making it easier to visualize and manipulate the component's structure.
- **State Management (useState, useEffect)**: `useState` is used to manage component-specific states such as `notebooks` (the list of notebooks), `isModalOpen` (visibility of the "New Notebook" modal), `isPreviewOpen` (visibility of the notebook preview dialog), `selectedNotebook` (the notebook being previewed), and `newNotebook` (input values for new notebook creation). `useEffect` is used to load notebooks from Firestore upon component mount.
    - *Reasoning:* State management is essential for handling dynamic data and user interactions. `useState` and `useEffect` facilitate efficient data fetching, UI updates, and component lifecycle management.
- **Asynchronous Operations (Promises, Async/Await)**: Asynchronous operations using Promises and `async/await` are employed for interacting with Firestore to fetch and manipulate notebook data. Functions like `readNotebooks`, `addNotebook`, and `deleteNotebook` utilize asynchronous patterns for non-blocking database interactions.
    - *Reasoning:* Asynchronous operations are crucial for handling database calls in web applications, ensuring that the UI remains responsive while waiting for data to be retrieved or updated.
- **Database Interaction (Firestore)**: The Notebooks Main Page interacts with Firestore to manage notebook data. Functions are implemented to read, create, and delete notebook documents in the "notebooks" collection, enabling data persistence and retrieval.

    - *Reasoning:* Firestore provides a scalable and real-time NoSQL database, suitable for storing and managing user-generated content like notebooks and notes.
- **Modals (Radix UI Dialog)**: Radix UI Dialog components are used to implement modals for creating new notebooks and previewing existing ones. Modals provide focused UI elements for specific tasks, enhancing user experience by streamlining interactions.
    - *Reasoning:* Modals are effective UI patterns for creating focused interactions and preventing distractions from the main page content. Radix UI Dialog offers accessible and customizable modal components.
- **Grid Layout (CSS Grid)**: CSS Grid is used to arrange notebook cards in a responsive grid layout on the Notebooks Main Page. This ensures a visually organized and adaptable display of notebooks across different screen sizes.
    - *Reasoning:* CSS Grid provides a powerful layout system for creating structured and responsive grids, ideal for displaying lists of items in a visually appealing and organized manner.
- **Client-Side Routing (Next.js `useRouter`, `useParams`)**: Next.js routing, specifically the `useRouter` hook, is used to handle navigation to the Notebook View page when a user opens a notebook. `useParams` is used in the Notebook View page to extract the `notebookId` from the URL, enabling dynamic routing based on notebook IDs.
    - *Reasoning:* Next.js routing simplifies navigation within the application, allowing for seamless transitions between different pages and views, and dynamic routing enables parameter-based page access.
- **Custom Hooks (use404Redirect)**: The `use404Redirect` custom hook is used in the `ContinuousInfoSpacePage` component (parent route for Notebook View) to handle cases where a user might navigate to an invalid notebook ID. It redirects the user to the main Notebooks management page, improving user experience by gracefully handling potentially broken or invalid routes.
    - *Reasoning:* Custom hooks like `use404Redirect` encapsulate reusable navigation logic, enhancing application robustness and user-friendliness by automatically handling navigation errors and redirecting users to valid application areas.
- **Array Mapping (`.map()`)**: Array mapping is used extensively to render lists of notebooks and, within the preview modal, to render sections, columns, and notes. This technique efficiently generates dynamic lists of UI elements based on data arrays.

    - *Reasoning:* Array mapping is a fundamental JavaScript and React technique for efficiently rendering lists of components or UI elements based on array data, promoting code conciseness and readability.

Notebook View (Specific Notebook Page):

- **React Components**: The Notebook View is also built using React components, including `TodoListInterface` (the main component for the Notebook View page), `DraggableColumn`, `DroppableColumn`, and `DraggableNote` (these components, despite their names, are adapted and reused from the ToDoList module for managing sections, columns, and notes within Notebooks, showcasing component reusability across modules).
    - *Reasoning:* React's component model allows for reusing and adapting components across different parts of the application, reducing development effort and promoting consistency.
- **JSX (JavaScript XML)**: JSX is used to structure the Notebook View page, defining the layout of sections, columns, notes, and UI controls. It ensures a declarative and maintainable representation of the Notebook View's UI.
    - *Reasoning:* JSX simplifies the creation and manipulation of the DOM in React, making the component structure more intuitive and easier to manage.
- **State Management (useState, useEffect)**: `useState` is used extensively to manage the state of the Notebook View page, including `sections` (list of sections), `archivedSections` (list of archived sections), `previewSection` (section being previewed), `activeNote` (currently dragged note), and various states for editing and UI interactions. `useEffect` is used to fetch notebook data from Firestore and synchronize updates.
    - *Reasoning:* State management is crucial for handling the dynamic and interactive nature of the Notebook View, allowing for real-time updates and user-driven changes to the notebook structure and content.
- **Drag and Drop Functionality (dnd-kit library - reused and adapted)**: Drag and drop functionality from `dnd-kit` is reused and adapted from the ToDoList module to enable reordering of columns and notes within the Notebook View. Components like `DndContext`, `useDraggable`, `useDroppable`, and related hooks are employed to implement drag-and-drop for notebook organization.
    - *Reasoning:* Reusing and adapting drag-and-drop components and logic from the ToDoList module demonstrates efficient code reuse and reduces development effort for implementing similar functionalities in different parts of the application.

- **Asynchronous Operations (Promises, Async/Await)**: Asynchronous operations are used for fetching notebook data (`getNotebook` action) and updating notebook sections (`updateNotebook` action) in Firestore. These operations ensure non-blocking UI and efficient data handling for the Notebook View.
  - *Reasoning:* Asynchronous operations are essential for handling database interactions, maintaining UI responsiveness and providing a smooth user experience while data is being loaded or saved.
- **Database Interaction (Firestore)**: The Notebook View interacts with Firestore to fetch and update notebook data. Functions are used to retrieve a specific notebook by ID (`getNotebook`) and to update the sections array of a notebook (`updateNotebook`), enabling data persistence and real-time updates.
  - *Reasoning:* Firestore provides a backend for storing and managing notebook data, ensuring data persistence and enabling real-time synchronization across different sessions and devices.
- **Sheet Component (Radix UI Sheet)**: Radix UI Sheet component is used to implement the "Archived Sections" panel, providing an off-canvas panel to display and manage archived sections. Sheets are used to present supplementary UI elements without disrupting the main page layout.
  - *Reasoning:* Radix UI Sheet offers an accessible and user-friendly way to implement side panels or drawers for displaying additional content or controls, improving UI organization and user experience.
- **Card Component (Radix UI Card)**: Radix UI Card components are used to display archived sections within the "Archived Sections" panel and potentially for notebook previews (though not explicitly shown in screenshots, Card components are commonly used for content containers). Cards provide a visually distinct and structured container for displaying information.
  - *Reasoning:* Radix UI Card components offer a standardized and visually appealing way to present information blocks, enhancing UI clarity and visual hierarchy.
- **Custom Hooks (useToast, use404Redirect, use404Redirect - reused)**: The `useToast` custom hook is reused to display toast notifications for user feedback, such as confirmation messages after deleting a section. The `use404Redirect` hook, reused from the main Notebooks page, ensures graceful navigation in case of invalid notebook IDs, demonstrating consistent use of custom hooks for cross-cutting concerns.
  - *Reasoning:* Reusing custom hooks promotes code consistency and reduces redundancy, ensuring a unified approach to common functionalities like toast notifications and route error handling throughout the application.

- **Hierarchical Data Structure (Sections, Columns, Notes)**: The Notebook View is structured around a hierarchical data model consisting of Notebooks, Sections, Columns, and Notes. This nested structure allows for a flexible and organized approach to information management, mirroring a physical notebook with sections and subsections.
    - *Reasoning:* Hierarchical data structures are well-suited for organizing information in a logical and scalable manner. This structure allows users to break down complex information into manageable units and navigate through them efficiently.

Imports used:

- **Notebooks Main Page (Notebook Management):** Import statements from `components\ContinuousInfoSpace\ContinuousInfoSpaceDocMan.tsx`

```
import React, { useState, useEffect } from "react";
import { Plus, Trash2, Notebook, Eye } from "lucide-react";
import { DndContext, closestCenter, KeyboardSensor, PointerSensor, useSensor, useSensors } from
"@dnd-kit/core";
import { SortableContext, rectSortingStrategy } from "@dnd-kit/sortable";
import { Button } from "@/components/ui/button";
import { Dialog, DialogContent, DialogHeader, DialogTitle } from "@/components/ui/dialog";
import { useRouter } from "next/navigation";
import { readNotebooks, deleteNotebook, addNotebook, getNotebook, notebook } from
"@/lib/continuous-info-space-doc-man-actions";
```

- **Notebook View (Specific Notebook Page):** Import statements from `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace[notebookId]\page.tsx`

```
import React, { useEffect, useState } from "react";
import { useRouter, useParams } from "next/navigation";
import { BackButton } from "@/components/ui/custom-ui/back-button";
import { getNotebook, updateNotebook } from "@/lib/continuous-info-space-doc-man-actions";
import { Archive, Plus, X } from "lucide-react";
import { Button } from "@/components/ui/button";
import { useUser } from "@clerk/nextjs";
import { Sheet, SheetContent, SheetHeader, SheetTitle, SheetTrigger } from
"@/components/ui/sheet";
import { Card, CardContent, CardHeader, CardTitle } from "@/components/ui/card";
import { DndContext, DragOverlay, useSensors, useSensor, PointerSensor, closestCorners,
DragStartEvent, DragEndEvent, useDroppable, useDraggable } from "@dnd-kit/core";
```

```
import { useToast } from "@/hooks/use-toast";
import { ToastAction } from "@/components/ui/toast";
```

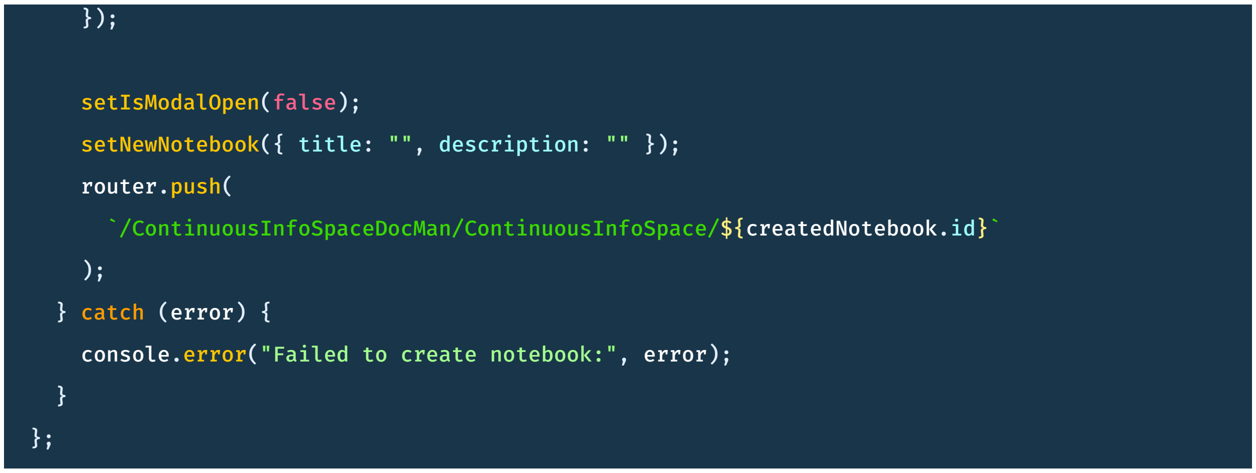
Functions used:

- **Notebooks Main Page (Notebook Management):**
  - **`loadNotebooks` (useEffect callback):** Fetches all notebooks for the user when the component mounts, using the `readNotebooks` action.
    - *Functionality:* Populates the list of notebooks displayed on the main page.
    - *Screenshot:* `loadNotebooks` function within the `useEffect` hook in `components\ContinuousInfoSpace\ContinuousInfoSpaceDocMan.tsx`

```
  const loadNotebooks = async () => {
    const fetchedNotebooks = await readNotebooks();
    setNotebooks(fetchedNotebooks);
  };
```

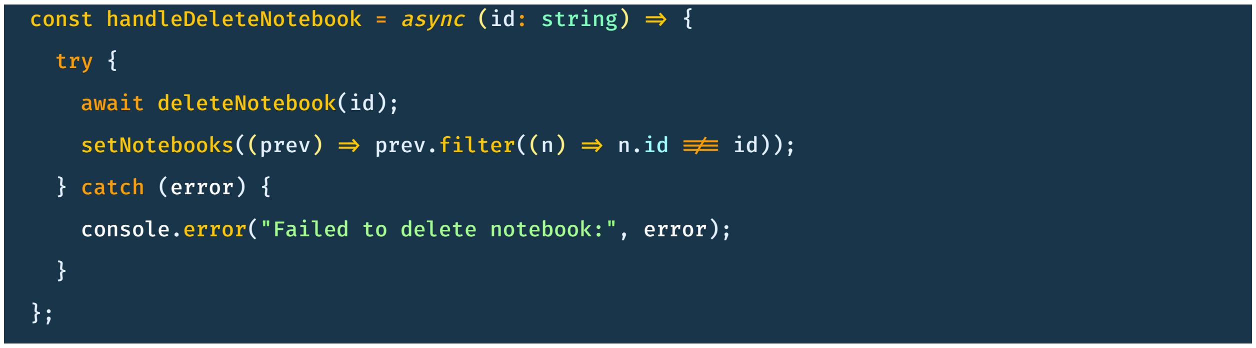
  - **`handleCreateNotebook`**: Handles the submission of the "New Notebook" modal form. It calls the `addNotebook` action to create a new notebook in Firestore, closes the modal, and redirects the user to the newly created notebook's view page.
    - *Functionality:* Manages the notebook creation process, persisting new notebooks to the database and navigating the user to the new notebook.
    - *Screenshot:* `handleCreateNotebook` function in `components\ContinuousInfoSpace\ContinuousInfoSpaceDocMan.tsx`

```
  const handleCreateNotebook = async (e: React.FormEvent) => {
    e.preventDefault();
    if (!newNotebook.title) return;

    try {
      const createdNotebook = await addNotebook({
        title: newNotebook.title,
        description: newNotebook.description,
```



```
    });

    setIsModalOpen(false);
    setNewNotebook({ title: "", description: "" });
    router.push(
      `/ContinuousInfoSpaceDocMan/ContinuousInfoSpace/${createdNotebook.id}`
    );
  } catch (error) {
    console.error("Failed to create notebook:", error);
  }
};
```

- **`handleDeleteNotebook`**: Handles deleting a notebook. Calls the `deleteNotebook` action to remove a notebook from Firestore and updates the local state by filtering out the deleted notebook.
  - *Functionality:* Manages the notebook deletion process, removing notebooks from the database and updating the UI.
  - *Screenshot:* `handleDeleteNotebook` function in `components\ContinuousInfoSpace\ContinuousInfoSpaceDocMan.tsx`

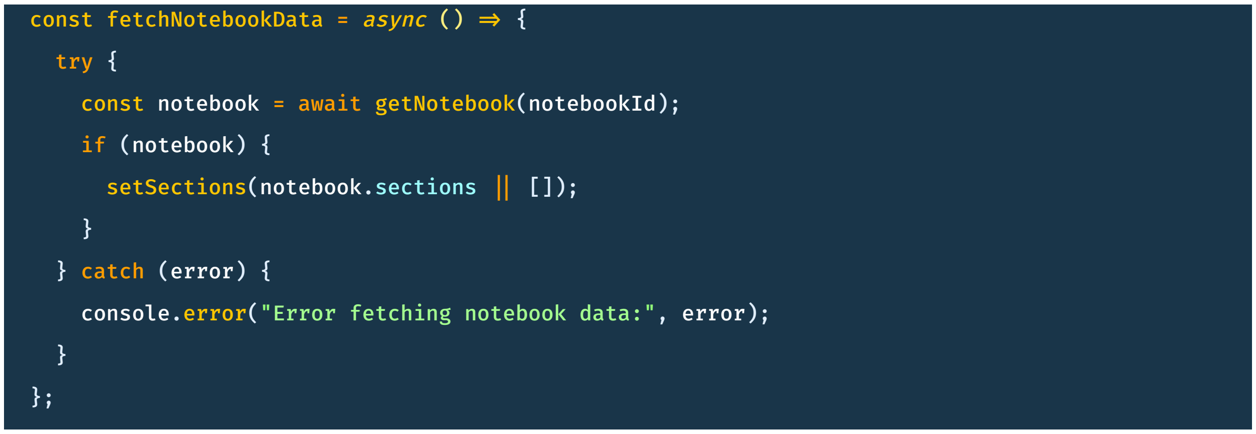


```
const handleDeleteNotebook = async (id: string) => {
  try {
    await deleteNotebook(id);
    setNotebooks((prev) => prev.filter((n) => n.id !== id));
  } catch (error) {
    console.error("Failed to delete notebook:", error);
  }
};
```

- **Notebook View (Specific Notebook Page):**

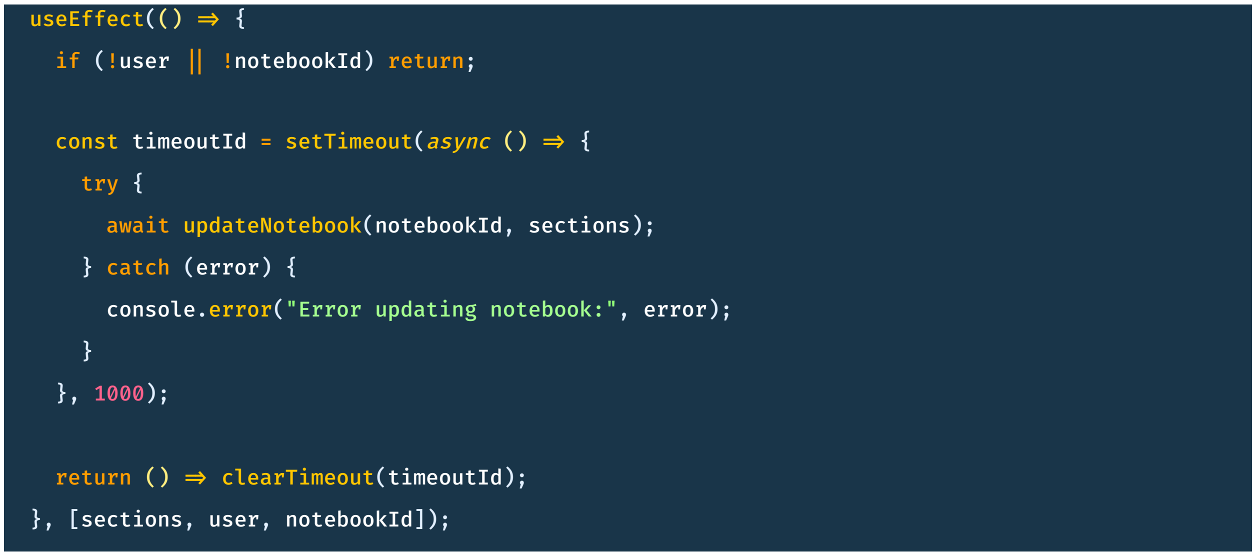
  - **`fetchNotebookData` (useEffect callback)**: Fetches data for a specific notebook from Firestore based on the `notebookId` parameter from the URL when the component mounts. It uses the `getNotebook` action to retrieve notebook data and updates the `sections` state.
    - *Functionality:* Loads the content of a specific notebook for display in the Notebook View.

        - *Screenshot:* `fetchNotebookData` function within the `useEffect` hook in `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace[notebookId]\page.tsx`



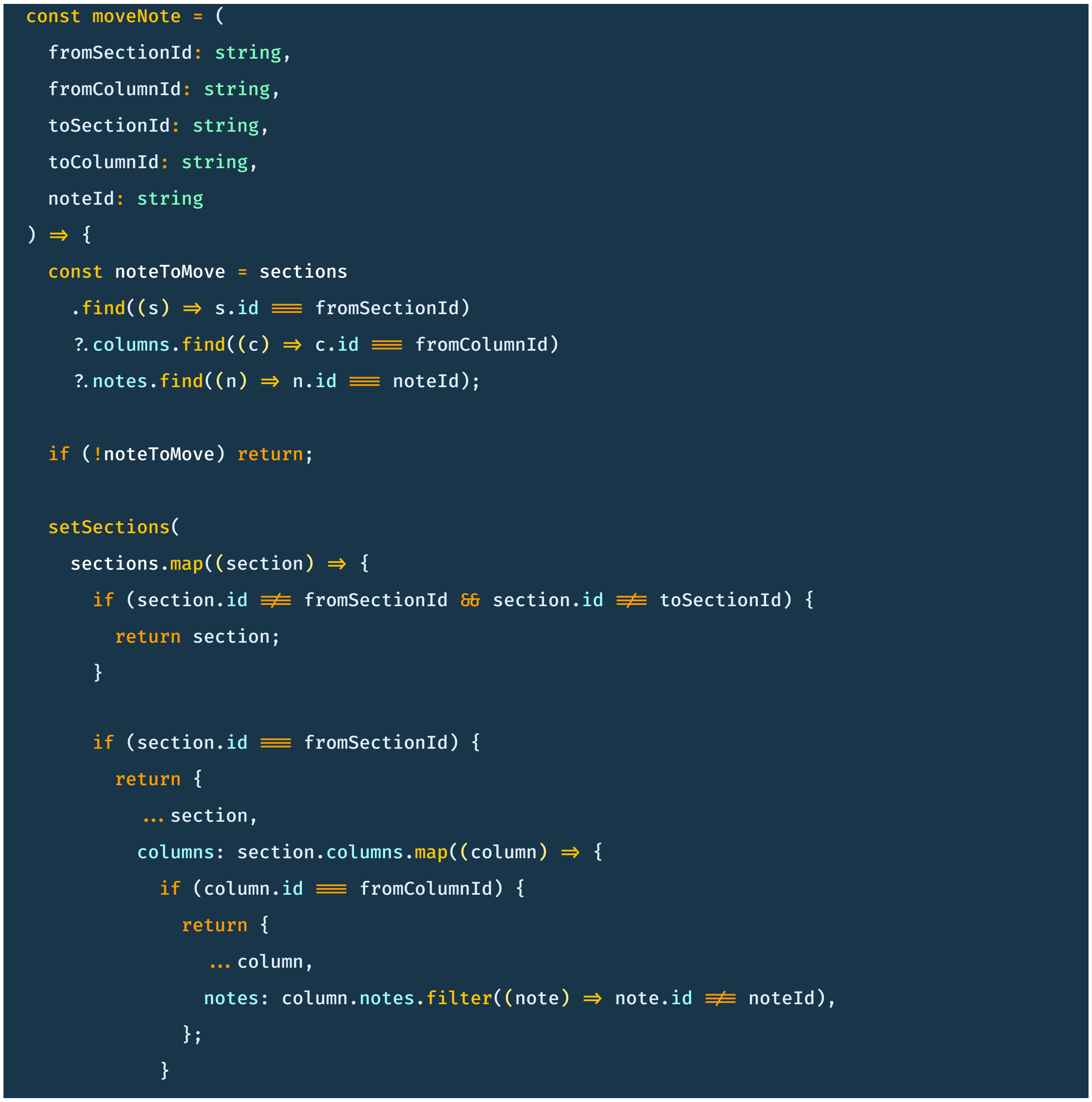
```
const fetchNotebookData = async () => {
  try {
    const notebook = await getNotebook(notebookId);
    if (notebook) {
      setSections(notebook.sections || []);
    }
  } catch (error) {
    console.error("Error fetching notebook data:", error);
  }
};
```

    - **Debounced `useEffect` for `updateNotebook`**: This `useEffect` hook is responsible for saving changes to the notebook structure (sections, columns, notes) to Firestore. It uses `setTimeout` to debounce the database updates, improving performance by reducing the frequency of write operations.
        - *Functionality:* Persists changes made to the notebook structure to the database, ensuring data persistence and synchronization.
        - *Screenshot:* Debounced `useEffect` hook for `updateNotebook` in `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace[notebookId]\page.tsx`



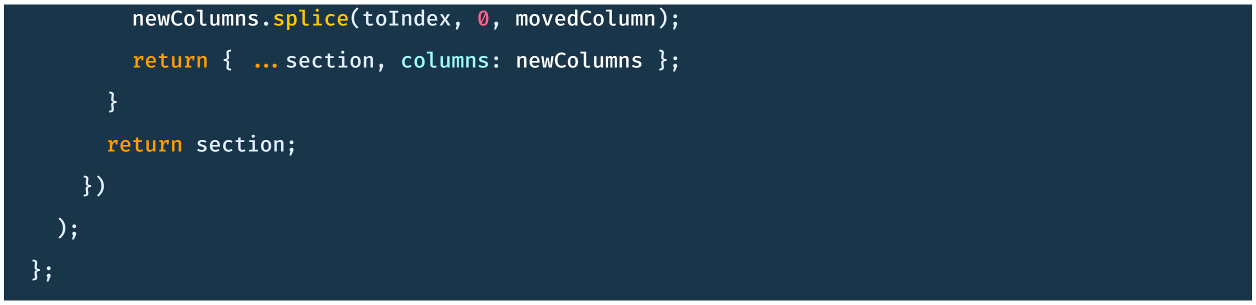
```
useEffect(() => {
  if (!user || !notebookId) return;

  const timeoutId = setTimeout(async () => {
    try {
      await updateNotebook(notebookId, sections);
    } catch (error) {
      console.error("Error updating notebook:", error);
    }
  }, 1000);

  return () => clearTimeout(timeoutId);
}, [sections, user, notebookId]);
```

- **`moveNote`**, **`moveColumn`**: These functions handle the logic for reordering notes within columns and columns within sections using drag and drop. They update the **`sections`** state to reflect the new order after a drag-and-drop operation.
  - *Functionality:* Manages the reordering of elements within the Notebook View UI based on user drag-and-drop interactions, updating the component's state to reflect the new arrangement.
  - *Screenshot:* `moveNote` and `moveColumn` functions from `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace[notebookId]\page.tsx`

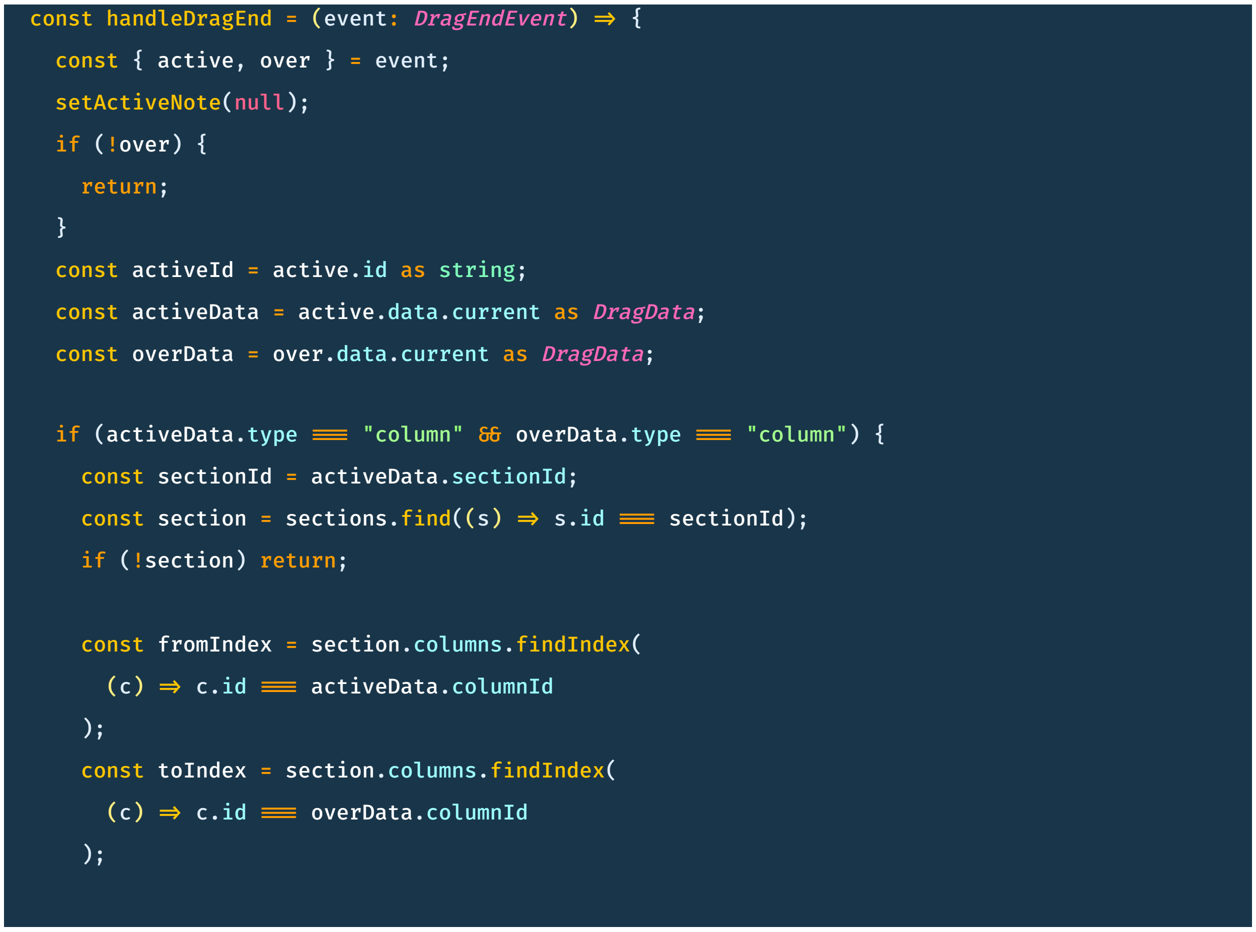


```
const moveNote = (
  fromSectionId: string,
  fromColumnId: string,
  toSectionId: string,
  toColumnId: string,
  noteId: string
) => {
  const noteToMove = sections
    .find((s) => s.id === fromSectionId)
    ?.columns.find((c) => c.id === fromColumnId)
    ?.notes.find((n) => n.id === noteId);

  if (!noteToMove) return;

  setSections(
    sections.map((section) => {
      if (section.id !== fromSectionId && section.id !== toSectionId) {
        return section;
      }

      if (section.id === fromSectionId) {
        return {
          ...section,
          columns: section.columns.map((column) => {
            if (column.id === fromColumnId) {
              return {
                ...column,
                notes: column.notes.filter((note) => note.id !== noteId),
              };
            }
            if (column.id === toColumnId && fromSectionId === toSectionId) {
              return {
                ...column,
                notes: [...column.notes, noteToMove],
              };
            }
            return column;
          }),
        };
      }

      if (section.id === toSectionId) {
        return {
          ...section,
          columns: section.columns.map((column) => {
            if (column.id === toColumnId) {
              return {
                ...column,
                notes: [...column.notes, noteToMove],
              };
            }
            return column;
          }),
        };
      }

      return section;
    })
  );
};
```

```
const moveColumn = (
  sectionId: string,
  fromIndex: number,
  toIndex: number
) => {
  setSections(
    sections.map((section) => {
      if (section.id === sectionId) {
        const newColumns = [...section.columns];
        const [movedColumn] = newColumns.splice(fromIndex, 1);
```


```
        newColumns.splice(toIndex, 0, movedColumn);
        return {  ...section, columns: newColumns };
      }
      return section;
    })
  );
};
```

- **`handleDragEnd`**: This function is triggered after a drag-and-drop operation within the Notebook View. It determines if a note or column was dragged and calls the appropriate `moveNote` or `moveColumn` function to update the component's state.
  - *Functionality:* Orchestrates the drag-and-drop handling logic for reordering notes and columns, calling the specific state update functions based on the drag event.
  - *Screenshot:* `handleDragEnd` function in `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace[notebookId]\page.tsx`


```
const handleDragEnd = (event: DragEndEvent) => {
  const { active, over } = event;
  setActiveNote(null);
  if (!over) {
    return;
  }
  const activeId = active.id as string;
  const activeData = active.data.current as DragData;
  const overData = over.data.current as DragData;

  if (activeData.type === "column" && overData.type === "column") {
    const sectionId = activeData.sectionId;
    const section = sections.find((s) => s.id === sectionId);
    if (!section) return;

    const fromIndex = section.columns.findIndex(
      (c) => c.id === activeData.columnId
    );
    const toIndex = section.columns.findIndex(
      (c) => c.id === overData.columnId
    );
    if (fromIndex !== toIndex) {
      moveColumn(sectionId, fromIndex, toIndex);
    }
    return;
  }

  if (activeData.type === "note" || !activeData.type) {
    if (!overData?.type || !overData.columnId) {
      return;
    }
    let fromSectionId, fromColumnId;
    sections.some((section) => {
      return section.columns.some((column) => {
        const found = column.notes.some((note) => note.id === activeId);
        if (found) {
          fromSectionId = section.id;
          fromColumnId = column.id;
          return true;
        }
        return false;
      });
    });

    const toSectionId = overData.sectionId;
    const toColumnId = overData.columnId;

    if (fromSectionId && fromColumnId && toSectionId && toColumnId) {
      moveNote(
        fromSectionId,
        fromColumnId,
        toSectionId,
        toColumnId,
        activeId
      );
    }
  }
};
```

-

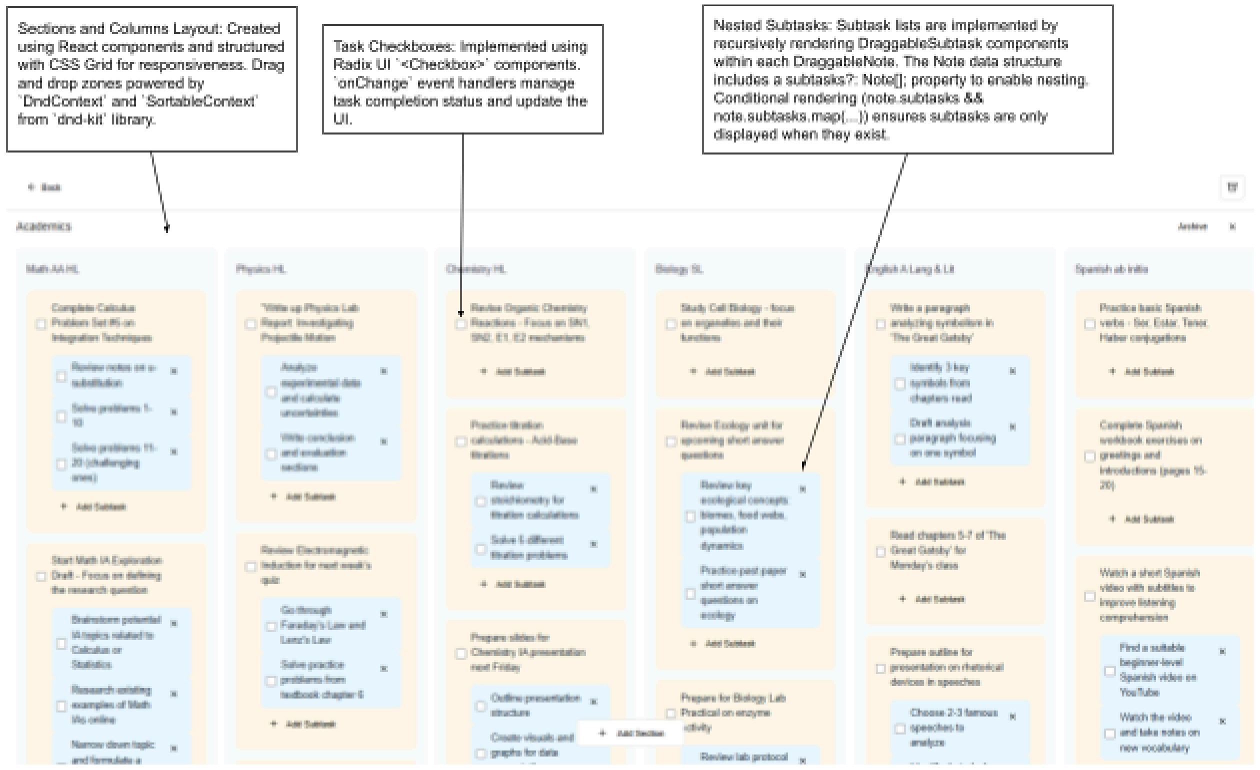
To-Do List

The To-Do List module is designed to provide users with a flexible and hierarchical task management system. It allows users to organize tasks into sections and columns, creating a visual and structured overview of their to-dos. Within this module, users can:

- **Organize Tasks Hierarchically**: Structure tasks into sections and columns, allowing for categorization and prioritization of to-dos.
- **Manage Tasks with Notes and Subtasks**: Create individual tasks (notes) within columns, and further break down tasks into smaller, manageable subtasks, facilitating detailed task decomposition.
- **Track Task Completion**: Utilize checkboxes to mark tasks and subtasks as complete, providing a visual indication of progress.
- **Reorder Tasks**: Reorder notes within columns and columns within sections using drag-and-drop functionality, enabling dynamic prioritization and workflow adjustments.
- **Archive Sections**: Archive entire sections to declutter the active to-do list while retaining sections for future reference in an "Archive" panel.

This module aims to enhance user productivity by providing a visually organized and interactive space for managing tasks, breaking down large projects, and tracking progress towards completion.

Screenshot:


Sections and Columns Layout: Created using React components and structured with CSS Grid for responsiveness. Drag and drop zones powered by `DndContext` and `SortableContext` from `dnd-kit` library.

Task Checkboxes: Implemented using Radix UI `<Checkbox>` components. `onChange` event handlers manage task completion status and update the UI.

Nested Subtasks: Subtask lists are implemented by recursively rendering DraggableSubtask components within each DraggableNote. The Note data structure includes a subtasks?: Note[]; property to enable nesting. Conditional rendering (note.subtasks && note.subtasks.map(....)) ensures subtasks are only displayed when they exist.

Math AA HL
- Complete Calculus Problem Set #5 on Integration Techniques
  - Review notes on u-substitution
  - Solve problems 1-10
  - Solve problems 11-20 (challenging ones)
- Start Math IA Exploration Draft - Focus on defining the research question
  - Brainstorm potential IA topics related to Calculus or Statistics
  - Research existing examples of Math IAs online
  - Narrow down topic and formulate a

Physics HL
- Write up Physics Lab Report: Investigating Projectile Motion
  - Analyze experimental data and calculate uncertainties
  - Write conclusion and evaluation sections
- Review Electromagnetic Induction for next week's quiz
  - Go through Faraday's Law and Lenz's Law
  - Solve practice problems from textbook chapter 6

Chemistry HL
- Review Organic Chemistry Reactions - Focus on SN1, SN2, E1, E2 mechanisms
- Practice titration calculations - Acid-Base titrations
  - Review stoichiometry for titration calculations
  - Solve 5 different titration problems
- Prepare slides for Chemistry IA presentation next Friday
  - Outline presentation structure
  - Create visuals and graphs for data

Biology SL
- Study Cell Biology - focus on organelles and their functions
- Review Ecology unit for upcoming short answer questions
  - Review key ecological concepts: biomes, food webs, population dynamics
  - Practice past paper short answer questions on ecology
- Prepare for Biology Lab Practical on enzyme activity
  - Review lab protocol

English A Lang & Lit
- Write a paragraph analyzing symbolism in 'The Great Gatsby'
  - Identify 3 key symbols from chapters read
  - Draft analysis paragraph focusing on one symbol
- Read chapters 5-7 of 'The Great Gatsby' for Monday's class
- Prepare outline for presentation on rhetorical devices in speeches
  - Choose 2-3 famous speeches to analyze

Spanish ab initio
- Practice basic Spanish verbs - Ser, Estar, Tener, Haber conjugations
- Complete Spanish workbook exercises on greetings and introductions (pages 15-20)
- Watch a short Spanish video with subtitles to improve listening comprehension
  - Find a suitable beginner-level Spanish video on YouTube
  - Watch the video and take notes on new vocabulary

Techniques used:

- **React Components**: The To-Do List is built using a component-based architecture in React. Key components include `ToDoList` (the main component), `DraggableColumn`, `DroppableColumn`, and `DraggableNote`. This modular approach promotes code organization and reusability.
  - *Reasoning:* React components enable the encapsulation of UI elements and task management logic, simplifying development and maintenance of the To-Do List module.
- **JSX (JavaScript XML)**: JSX is used to define the structure and rendering of the To-Do List UI, providing a clear and declarative way to describe the component layout and elements.
  - *Reasoning:* JSX enhances code readability and developer efficiency by allowing HTML-like syntax within JavaScript, making UI structure easier to define and understand.
- **State Management (useState, useEffect)**: `useState` hooks are essential for managing the dynamic state of the To-Do List, including `sections` (the list of sections, columns, and notes), `activeDragItem` (tracking dragged items during drag and drop), and other UI states. `useEffect` is utilized to

synchronize To-Do List data with Firestore, ensuring data persistence and real-time updates.
    - *Reasoning:* State management is crucial for handling dynamic task data and user interactions within the To-Do List. `useState` and `useEffect` enable efficient state updates and lifecycle management.
- **Drag and Drop Functionality (dnd-kit library)**: The `dnd-kit` library is employed to implement drag-and-drop functionality, enabling users to reorder columns and notes within the To-Do List. Hooks like `DndContext`, `useDraggable`, `useDroppable`, and `closestCorners` are used to manage drag and drop interactions and maintain task order.
    - *Reasoning:* Drag and drop provides an intuitive and efficient way for users to reorganize tasks and columns, enhancing the usability and flexibility of the To-Do List for task prioritization and management.
- **Asynchronous Operations (Promises, Async/Await)**: Asynchronous operations using Promises and `async/await` are crucial for interacting with the Firestore database. Functions for fetching To-Do List data and saving changes are implemented asynchronously to prevent blocking the UI thread and ensure a smooth user experience.
    - *Reasoning:* Asynchronous operations are necessary for handling database interactions, ensuring that the application remains responsive while performing data operations in the background.
- **Database Interaction (Firestore)**: The To-Do List module utilizes Google Firestore to persist and retrieve task data. Firestore is used to store the hierarchical structure of sections, columns, and notes, allowing for persistent task management across user sessions.
    - *Reasoning:* Firestore provides a scalable and real-time NoSQL database, suitable for storing and synchronizing To-Do List data, ensuring data integrity and accessibility across devices.
- **Sheet Component (Radix UI Sheet)**: The Radix UI Sheet component is used to implement the "Archived Sections" panel. This panel, accessed via the "Archive" button, displays archived sections in an off-canvas manner, allowing users to review and manage archived content without cluttering the main To-Do List view.
    - *Reasoning:* Radix UI Sheet provides an accessible and user-friendly way to implement side panels and drawers, improving UI organization and providing access to secondary functionalities without disrupting the main workflow.
- **Card Component (Radix UI Card)**: Radix UI Card components are used to visually structure and contain archived sections within the "Archived Sections" panel. Cards provide a clear visual separation and organization for archived content, enhancing readability and UI clarity.

    - *Reasoning:* Radix UI Card components offer a standardized and visually appealing way to present information blocks, improving UI aesthetics and content organization within the archived sections panel.
- **Component Reusability (DraggableColumn, DroppableColumn, DraggableNote)**: The To-Do List reuses and adapts core drag-and-drop components (`DraggableColumn`, `DroppableColumn`, `DraggableNote`) that are structurally similar to components potentially used in other modules (like Notebooks). This reuse of components demonstrates efficient code utilization and a consistent architectural pattern across the application.
    - *Reasoning:* Component reusability is a core principle of React development, reducing code duplication, improving maintainability, and promoting consistency across different parts of the application.
- **Nested Data Structures (Sections, Columns, Notes, Subtasks)**: The To-Do List data is organized in a nested hierarchical structure, including Sections containing Columns, Columns containing Notes, and Notes optionally containing Subtasks. This nested data structure allows for detailed task breakdown and hierarchical organization within the To-Do List.
    - *Reasoning:* Nested data structures are ideal for representing hierarchical information, enabling users to organize tasks into logical categories and sub-categories, reflecting real-world task management workflows.

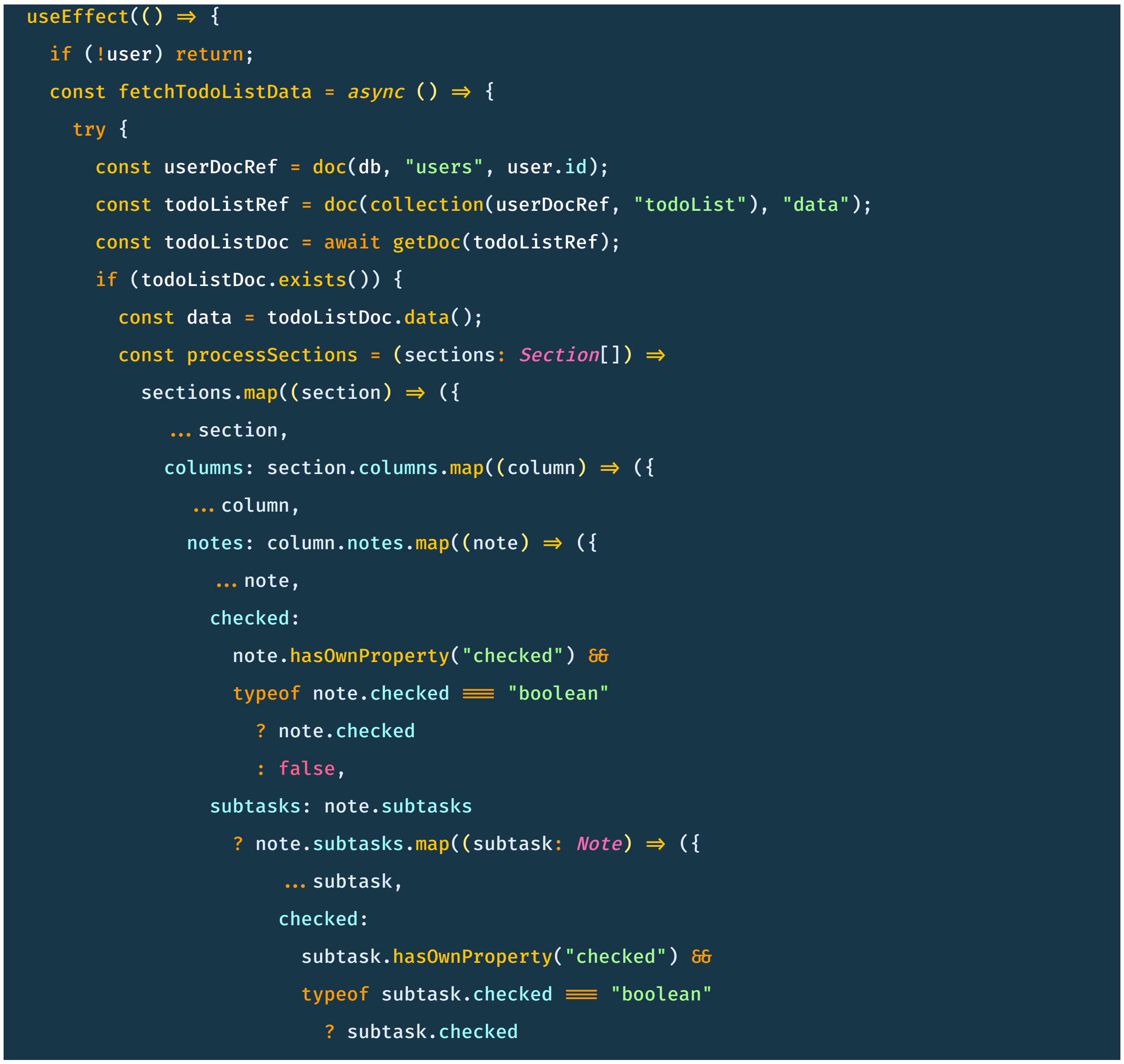
Imports used:

`Import statements from` components\ToDoList.tsx

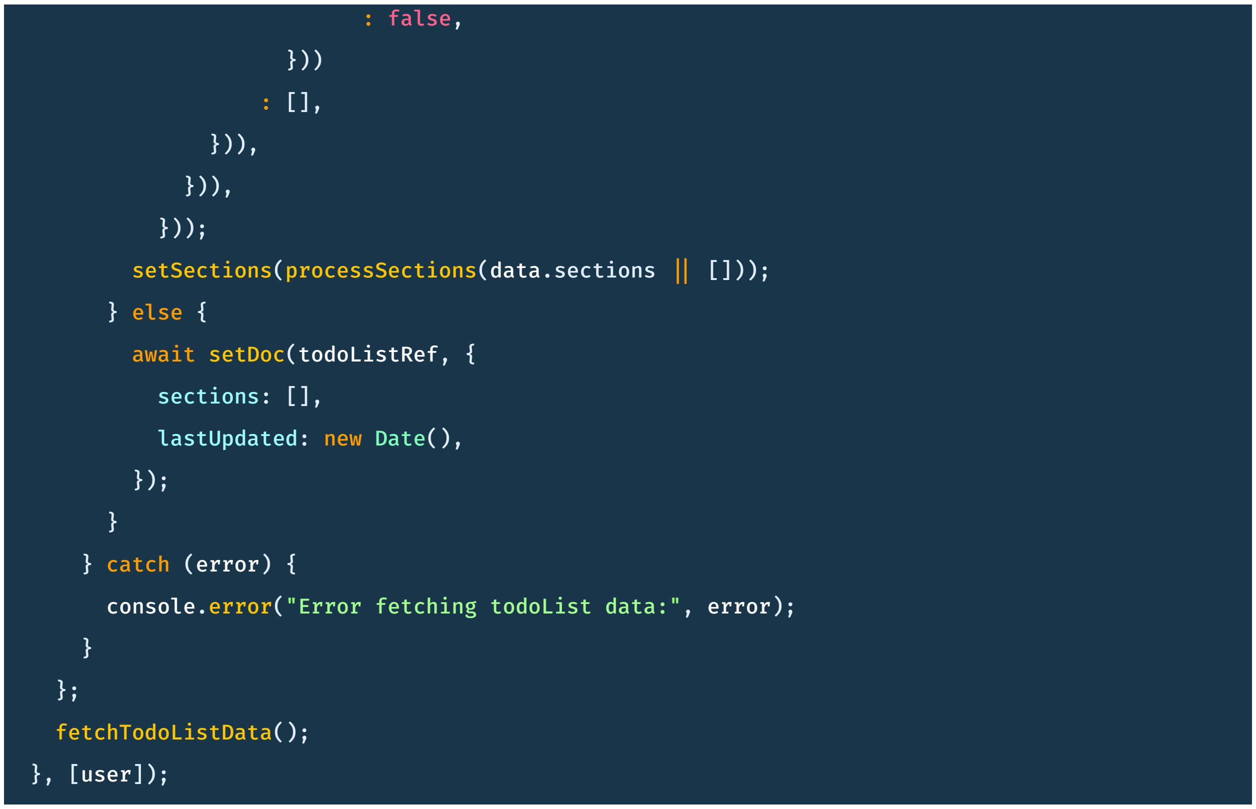
```
import React, { useEffect, useState } from "react";
import { useRouter } from "next/navigation";
import { Archive, Plus, X, ArrowLeft } from "lucide-react";
import { Button } from "@/components/ui/button";
import { useUser } from "@clerk/nextjs";
import { doc, getDoc, setDoc, collection } from "firebase/firestore";
import { db } from "@/lib/firebase";
import { Sheet, SheetContent, SheetHeader, SheetTitle, SheetTrigger } from
"@/components/ui/sheet";
import { DndContext, DragOverlay, useSensors, useSensor, PointerSensor, closestCorners,
DragStartEvent, DragEndEvent, useDroppable, useDraggable } from "@dnd-kit/core";
import { useToast } from "@/hooks/use-toast";
import { ToastAction } from "@/components/ui/toast";
import { BackButton } from "@/components/ui/custom-ui/back-button";
```

Functions used:

- **`fetchTodoListData` (useEffect callback)**: This asynchronous function, within a `useEffect` hook, fetches To-Do List data from Firestore when the `ToDoList` component mounts. It retrieves the structured data (sections, columns, notes) and updates the `sections` state, populating the To-Do List UI with user data.
    - *Functionality:* Loads the user's To-Do List data from the database and initializes the component's state with this data.
    - *Screenshot:* `fetchTodoListData` function within the `useEffect` hook in `components\ToDoList.tsx`

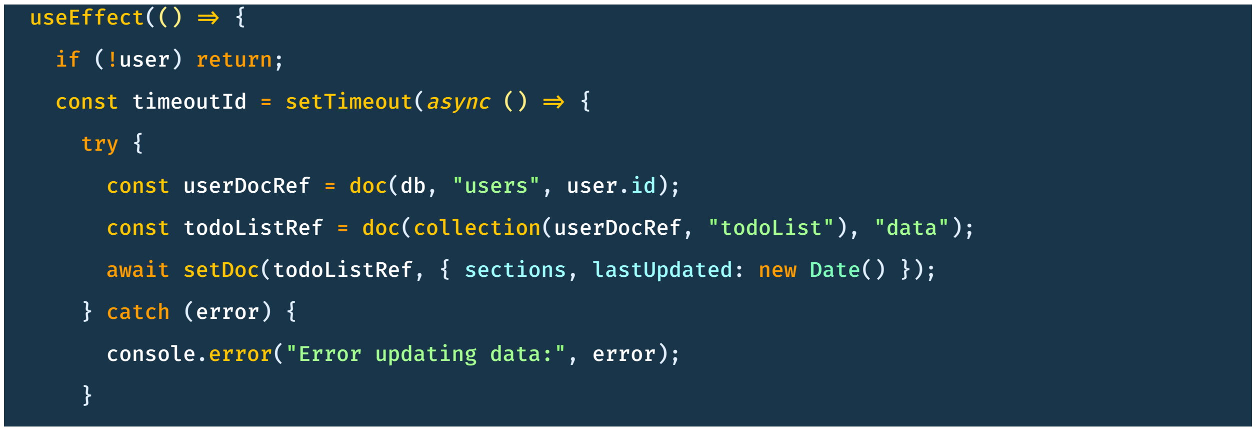


```
useEffect(() => {
  if (!user) return;
  const fetchTodoListData = async () => {
    try {
      const userDocRef = doc(db, "users", user.id);
      const todoListRef = doc(collection(userDocRef, "todoList"), "data");
      const todoListDoc = await getDoc(todoListRef);
      if (todoListDoc.exists()) {
        const data = todoListDoc.data();
        const processSections = (sections: Section[]) =>
          sections.map((section) => ({
            ...section,
            columns: section.columns.map((column) => ({
              ...column,
              notes: column.notes.map((note) => ({
                ...note,
                checked:
                  note.hasOwnProperty("checked") &&
                  typeof note.checked === "boolean"
                    ? note.checked
                    : false,
                subtasks: note.subtasks
                  ? note.subtasks.map((subtask: Note) => ({
                      ...subtask,
                      checked:
                        subtask.hasOwnProperty("checked") &&
                        typeof subtask.checked === "boolean"
                          ? subtask.checked
```



```
                          : false,
                    }))
                  : [],
              })),
            })),
          }));
        setSections(processSections(data.sections || []));
      } else {
        await setDoc(todoListRef, {
          sections: [],
          lastUpdated: new Date(),
        });
      }
    } catch (error) {
      console.error("Error fetching todoList data:", error);
    }
  };
  fetchTodoListData();
}, [user]);
```

- **Debounced `useEffect` for data saving**: This `useEffect` hook implements a debounced save mechanism to persist changes to the To-Do List data in Firestore. Using `setTimeout`, it delays database updates, reducing write frequency and optimizing performance, especially during rapid UI interactions.
  - *Functionality:* Persistently saves changes made to the To-Do List structure and task data to the database in an efficient manner, minimizing database operations.
  - *Screenshot:* Debounced `useEffect` hook for data saving in `components\ToDoList.tsx`



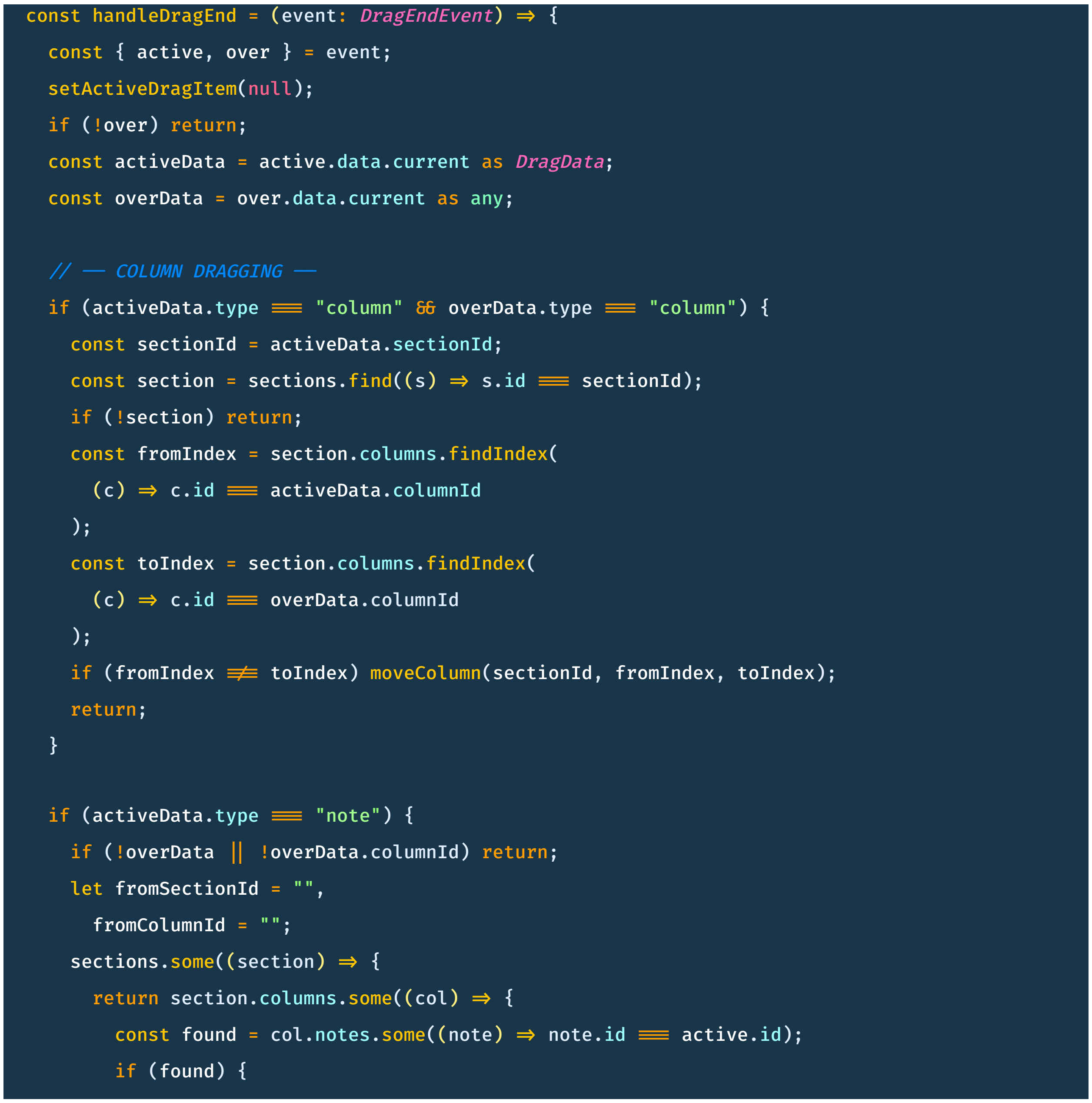
```
useEffect(() => {
  if (!user) return;
  const timeoutId = setTimeout(async () => {
    try {
      const userDocRef = doc(db, "users", user.id);
      const todoListRef = doc(collection(userDocRef, "todoList"), "data");
      await setDoc(todoListRef, { sections, lastUpdated: new Date() });
    } catch (error) {
      console.error("Error updating data:", error);
    }
  }, 1000);
  return () => clearTimeout(timeoutId);
}, [sections, user]);
```

- **`moveNote`**, **`moveColumn`**: These functions handle the core logic for reordering notes within columns and columns within sections using drag and drop. They update the `sections` state to reflect the new order, ensuring the UI accurately reflects the user's drag-and-drop actions.
  - *Functionality:* Implements the state updates necessary to reflect the reordering of tasks and columns based on user drag-and-drop interactions within the To-Do List.
  - *Screenshot:* `moveNote` and `moveColumn` functions from `components\ToDoList.tsx`

```
const moveNote = (
  fromSectionId: string,
  fromColumnId: string,
  toSectionId: string,
  toColumnId: string,
  noteId: string
) => {
  // Handles moving notes between columns and sections
  // Maintains note data integrity during moves
  const noteToMove = sections
    .find((s) => s.id === fromSectionId)
    ?.columns.find((c) => c.id === fromColumnId)
    ?.notes.find((n) => n.id === noteId);
  if (!noteToMove) return;
  setSections(
    sections.map((section) => {
      if (section.id === fromSectionId) {
        return {
          ...section,
          columns: section.columns.map((column) => {
            if (column.id === fromColumnId) {
              return {
                ...column,
                notes: column.notes.filter((note) => note.id !== noteId),
              };
            }
            if (column.id === toColumnId && fromSectionId === toSectionId) {
              return { ...column, notes: [...column.notes, noteToMove] };
            }
            return column;
          }),
        };
      }
      if (section.id === toSectionId) {
        return {
          ...section,
          columns: section.columns.map((column) => {
            if (column.id === toColumnId) {
              return { ...column, notes: [...column.notes, noteToMove] };
            }
            return column;
          }),
        };
      }
      return section;
    })
  );
};
```

```
const moveColumn = (
  sectionId: string,
  fromIndex: number,
  toIndex: number
) => {
  setSections(
    sections.map((section) => {
      if (section.id === sectionId) {
        const newColumns = [...section.columns];
        const [movedColumn] = newColumns.splice(fromIndex, 1);
        newColumns.splice(toIndex, 0, movedColumn);
        return { ...section, columns: newColumns };
      }
      return section;
    })
  );
};
```

- **`handleDragEnd`**: This function, triggered at the end of a drag-and-drop operation within the `DndContext` provider, determines if a note or column was moved. It calls the appropriate `moveNote` or `moveColumn` function to update the To-Do List state based on the drag event's outcome.
    - *Functionality:* Orchestrates the drag-and-drop handling, determining the type of drag operation and calling the relevant state update functions to persist the reordering of tasks or columns.
    - *Screenshot:* `handleDragEnd` function in `components\ToDoList.tsx`



```
const handleDragEnd = (event: DragEndEvent) => {
  const { active, over } = event;
  setActiveDragItem(null);
  if (!over) return;
  const activeData = active.data.current as DragData;
  const overData = over.data.current as any;

  // --- COLUMN DRAGGING ---
  if (activeData.type === "column" && overData.type === "column") {
    const sectionId = activeData.sectionId;
    const section = sections.find((s) => s.id === sectionId);
    if (!section) return;
    const fromIndex = section.columns.findIndex(
      (c) => c.id === activeData.columnId
    );
    const toIndex = section.columns.findIndex(
      (c) => c.id === overData.columnId
    );
    if (fromIndex !== toIndex) moveColumn(sectionId, fromIndex, toIndex);
    return;
  }

  if (activeData.type === "note") {
    if (!overData || !overData.columnId) return;
    let fromSectionId = "",
      fromColumnId = "";
    sections.some((section) => {
      return section.columns.some((col) => {
        const found = col.notes.some((note) => note.id === active.id);
        if (found) {
          fromSectionId = section.id;
          fromColumnId = col.id;
          return true;
        }
        return false;
      });
    });
    const toSectionId = overData.sectionId;
    const toColumnId = overData.columnId;
    if (fromSectionId && fromColumnId && toSectionId && toColumnId) {
      moveNote(
        fromSectionId,
        fromColumnId,
        toSectionId,
        toColumnId,
        active.id as string
      );
    }
  }

  if (activeData.type === "subtask") {
    if (!overData || !activeData.columnId) return;
    if (overData.type === "subtask-container") {
      const toParentNoteId = overData.parentNoteId;
      const toColumnId = overData.columnId;
      const { sectionId, columnId, parentNoteId, subtaskId } = activeData;
      if (
        !sectionId ||
        !columnId ||
        !parentNoteId ||
        !subtaskId ||
        !toColumnId
      )
        return;

      if (parentNoteId === toParentNoteId && columnId === toColumnId) {
        let fromIndex = -1,
          toIndex = -1;
        sections.forEach((section) => {
          if (section.id === sectionId) {
            section.columns.forEach((col) => {
```

```
        if (col.id === columnId) {
          col.notes.forEach((note) => {
            if (note.id === parentNoteId && note.subtasks) {
              fromIndex = note.subtasks.findIndex(
                (s) => s.id === subtaskId
              );
              if (over.id.toString().startsWith("subtask-")) {
                const hoveredSubtaskId = over.id
                  .toString()
                  .replace("subtask-", "");
                toIndex = note.subtasks.findIndex(
                  (s) => s.id === hoveredSubtaskId
                );
              } else {
                toIndex = note.subtasks.length;
              }
            }
          });
        }
      });
    }
  });
  if (fromIndex !== -1 && toIndex !== -1 && fromIndex !== toIndex) {
    moveSubtaskWithinParent(
      sectionId,
      columnId,
      parentNoteId,
      fromIndex,
      toIndex
    );
  }
} else {
  moveSubtaskToAnotherParent(
    sectionId,
    columnId,
    parentNoteId,
    subtaskId,
    toParentNoteId,
    toColumnId
  );
}
```

```
    }
  }
};
```

- **`deleteSection`**, **`deleteColumn`**, **`deleteNote`**, **`deleteSubtask`**: These functions implement the deletion logic for sections, columns, notes, and subtasks respectively. They update the `sections` state to remove the deleted items, and utilize the `useToast` hook to display user feedback messages (toast notifications) confirming the deletion and offering an "Undo" action.
    - *Functionality:* Manages the deletion of various elements within the To-Do List hierarchy, updating the component's state and providing user feedback via toast notifications.
    - *Screenshot:* `deleteSection`, `deleteColumn`, `deleteNote`, and `deleteSubtask` functions from `components\ToDoList.tsx`

```
const deleteSection = (sectionId: string) => {
  const section = sections.find((s) => s.id === sectionId);
  setSections(sections.filter((section) => section.id !== sectionId));
  toast({
    title: "Section deleted",
    description: `${section?.title || "Section"} has been deleted.`,
    action: (
      <ToastAction
        altText="Undo"
        onClick={() => {
          if (section) setSections((prev) => [...prev, section]);
        }}
      >
        Undo
      </ToastAction>
    ),
  });
};
```

```
const deleteColumn = (sectionId: string, columnId: string) => {
  const section = sections.find((s) => s.id === sectionId);
  const column = section?.columns.find((c) => c.id === columnId);
  setSections(
    sections.map((section) =>
      section.id === sectionId
```

```
        ? {
            ...section,
            columns: section.columns.filter((col) => col.id !== columnId),
          }
        : section
    )
  );
  toast({
    title: "Column deleted",
    description: `${column?.title || "Column"} has been deleted.`,
    action: (
      <ToastAction
        altText="Undo"
        onClick={() => {
          if (column) {
            setSections((prev) =>
              prev.map((s) =>
                s.id === sectionId
                  ? { ...s, columns: [...s.columns, column] }
                  : s
              )
            );
          }
        }}
      >
        Undo
      </ToastAction>
    ),
  });
};
```

```
const deleteNote = (sectionId: string, columnId: string, noteId: string) => {
  const section = sections.find((s) => s.id === sectionId);
  const note = section?.columns
    .find((c) => c.id === columnId)
    ?.notes.find((n) => n.id === noteId);
  setSections(
    sections.map((section) =>
      section.id === sectionId
        ? {
            ...section,
          columns: section.columns.map((col) =>
            col.id === columnId
              ? {
                  ...col,
                  notes: col.notes.filter((note) => note.id !== noteId),
                }
              : col
          ),
        }
      : section
  )
);
toast({
  title: "Note deleted",
  description: `${note?.content || "Note"} has been deleted.`,
  action: (
    <ToastAction
      altText="Undo"
      onClick={() => {
        if (note) {
          setSections((prev) =>
            prev.map((s) =>
              s.id === sectionId
                ? {
                    ...s,
                    columns: s.columns.map((c) =>
                      c.id === columnId
                        ? { ...c, notes: [...c.notes, note] }
                        : c
                    ),
                  }
                : s
            )
          );
        }
      }}
    >
      Undo
    </ToastAction>
  ),
});
```

```
};
```

```
const deleteSubtask = (
  sectionId: string,
  columnId: string,
  parentNoteId: string,
  subtaskId: string
) => {
  let deletedSubtask: Note | undefined;
  setSections((prevSections) =>
    prevSections.map((section) => {
      if (section.id !== sectionId) return section;
      return {
        ...section,
        columns: section.columns.map((column) => {
          if (column.id !== columnId) return column;
          return {
            ...column,
            notes: column.notes.map((note) => {
              if (note.id !== parentNoteId) return note;
              const newSubtasks = note.subtasks?.filter((subtask) => {
                if (subtask.id === subtaskId) {
                  deletedSubtask = subtask;
                  return false;
                }
                return true;
              });
              return {
                ...note,
                subtasks: newSubtasks,
              };
            }),
          };
        }),
      };
    })
  );

  toast({
    title: "Subtask deleted",
    description: `${deletedSubtask?.content || "Subtask"} has been deleted.`,
    action: (
      <ToastAction
        altText="Undo"
        onClick={() => {
          if (deletedSubtask) {
            setSections((prev) =>
              prev.map((s) =>
                s.id === sectionId
                  ? {
                      ...s,
                      columns: s.columns.map((c) =>
                        c.id === columnId
                          ? {
                              ...c,
                              notes: c.notes.map((n) =>
                                n.id === parentNoteId
                                  ? {
                                      ...n,
                                      subtasks: [
                                        ...(n.subtasks || []),
                                        deletedSubtask!,
                                      ],
                                    }
                                  : n
                              ),
                            }
                          : c
                      ),
                    }
                  : s
              )
            );
          }
        }}
      >
        Undo
      </ToastAction>
    ),
  });
};
```

-

Firestore Methods

This section will showcase the Firestore database interactions implemented within the application. Each subsection will focus on a specific function, providing a code snippet and explanation of the Firestore techniques employed.

1. `addCookie` (from `components\CookieJar.tsx`)

This function is responsible for creating new cookie documents within the user's `cookieJar` collection in Firestore. It demonstrates the use of the `setDoc` method to add a new document. The function also generates a unique ID for each cookie client-side using `crypto.randomUUID()` before storing it, and sets the initial `position` for new cookies.

```
export async function addCookie(
  newCookie: Omit<Cookie, "id">,
   gridCols: number = 4, // gridCols parameter is defined but not currently used in positioning
logic
): Promise<void> {
  try {
    const cookieJarDocRef = await getCookieJarDocRef();
    const docSnap = await getDoc(cookieJarDocRef);
       const  id  =  crypto.randomUUID();  //  Generate  a  unique  ID  for  the  new  cookie  using
crypto.randomUUID().

     // Determine the existing cookie data. If the document doesn't exist yet, start with an empty
cookies object.
    const existingData = docSnap.exists()
      ? (docSnap.data() as CookieJarDocument)
      : { cookies: {} };

    await setDoc(cookieJarDocRef, {
      cookies: {
        ...existingData.cookies, // Merge with existing cookies
        [id]: {
          name: newCookie.name,
          description: newCookie.description,
          position: { x: 0, y: 0 } // Set initial position for new cookies to (0,0)
        }
      }
    });
  } catch (error) {
    console.error("Error adding cookie to Firestore:", error);
  }
}
```

Firestore Techniques Highlighted:

- `setDoc`: Demonstrates the use of `setDoc` to create a new document in a Firestore collection.
- **Client-Side ID Generation**: Shows the generation of a unique document ID client-side using `crypto.randomUUID()` before database insertion.
- **Initial Data Structure**: Illustrates the structure of the data being written to Firestore for a new cookie document, including fields like `name`, `description`, and `position`.

---

2. `readCookies` (from `components\CookieJar.tsx`)

The `readCookies` function retrieves all cookie documents associated with the currently authenticated user from Firestore. It uses the `getDocs` method to fetch all documents from the `cookieJar` collection. The retrieved data is then processed and transformed into an array of `Cookie` objects for use within the application. The function also includes error handling and returns an empty array if no cookies are found or in case of errors.

```
export async function readCookies(): Promise<Cookie[]> {
  try {
    const cookieJarDocRef = await getCookieJarDocRef();
    const docSnap = await getDoc(cookieJarDocRef);

    // If the cookieJar document does not exist, return an empty array.
    if (!docSnap.exists()) {
      return [];
    }

    const data = docSnap.data() as CookieJarDocument;
    // Convert the record of cookies into an array of Cookie objects, including the ID from the
document key.
    return Object.entries(data.cookies).map(([id, cookie]) => ({
      id,
      ...cookie
    })).sort((a, b) => {
      // Sort cookies primarily by y-position (row) and then by x-position (column) to maintain
grid order.
      if (a.position?.y === b.position?.y) {
        return (a.position?.x || 0) - (b.position?.x || 0); // Default to 0 if position is not
defined.
      }
      return (a.position?.y || 0) - (b.position?.y || 0); // Default to 0 if position is not
defined.
    });
  } catch (error) {
    console.error("Error reading cookies from Firestore:", error);
    return []; // Return an empty array in case of an error to prevent app crashes.
  }
}
```
Firestore Techniques Highlighted:

- `getDocs`: Demonstrates the use of `getDocs` to retrieve multiple documents from a Firestore collection.
- **Data Deserialization**: Showcases the process of iterating through a `QuerySnapshot` and extracting data from each Firestore `DocumentSnapshot` to transform it into application-specific objects.
- **Empty State Handling**: Illustrates how to handle cases where no documents are found in a collection, returning an empty array to prevent errors.

---

3. `updateCookiePositions` (from `components\CookieJar.tsx`)

The `updateCookiePositions` function efficiently updates the `position` field of multiple cookie documents in Firestore. It uses the `setDoc` method in conjunction with the `merge: true` option. This approach ensures that only the `position` field is updated in each cookie document, while other existing data remains untouched. This method is optimized for scenarios like drag-and-drop reordering where multiple documents may need updates simultaneously.

```
export async function updateCookiePositions(
  updatedCookies: Cookie[]
): Promise<void> {
  try {
    const cookieJarDocRef = await getCookieJarDocRef();
    const docSnap = await getDoc(cookieJarDocRef);

    // If the cookieJar document does not exist, return as there's nothing to update.
    if (!docSnap.exists()) return;

    const data = docSnap.data() as CookieJarDocument;
     const updatedData = { ...data.cookies }; // Create a shallow copy of the existing cookies
data.

    updatedCookies.forEach(cookie => {
      // Only update if the cookie exists in the current data.
      if (updatedData[cookie.id]) {
        updatedData[cookie.id] = {
          ...updatedData[cookie.id],
               position: cookie.position // Update only the position, preserving other cookie
properties.
        };
      }
    });

    await setDoc(cookieJarDocRef, {
        cookies: updatedData // Update Firestore with the modified cookies data containing new
positions.
    });
  } catch (error) {
    console.error("Error updating cookie positions in Firestore:", error);
  }
}
```

Firestore Techniques Highlighted:

- `setDoc` **with** `merge: true`: Demonstrates the efficient update of specific fields in existing Firestore documents without overwriting the entire document.
- **Bulk Updates**: Shows how to perform updates on multiple Firestore documents within a single function, optimizing database operations for scenarios like list reordering.
- **Selective Field Update**: Illustrates updating only the `position` field while preserving other cookie properties, showcasing granular data manipulation in Firestore.

---

4. `deleteCookie` (from `components\CookieJar.tsx`)

The `deleteCookie` function removes a cookie document from Firestore. Instead of directly deleting the document, it updates the `cookieJar` document by removing the entry corresponding to the cookie's `id` from the `cookies` record. This approach assumes that cookies are stored as a record within a single `cookieJar` document per user.

```
export async function deleteCookie(id: string): Promise<void> {
  try {
    const cookieJarDocRef = await getCookieJarDocRef();
    const docSnap = await getDoc(cookieJarDocRef);

    // If the cookieJar document does not exist, there's nothing to delete, so just return.
    if (!docSnap.exists()) return;

    const data = docSnap.data() as CookieJarDocument;
    // Use destructuring to remove the cookie with the specified ID from the cookies record.
    const { [id]: deletedCookie, ...remainingCookies } = data.cookies;

    await setDoc(cookieJarDocRef, {
      cookies: remainingCookies // Update Firestore with the remaining cookies.
    });
  } catch (error) {
    console.error("Error deleting cookie from Firestore:", error);
  }
}
```

Firestore Techniques Highlighted:

- **Document Update for Deletion**: Demonstrates an alternative to direct document deletion by modifying a record field within a document to remove an entry.
- **Record Manipulation**: Shows how to manipulate records (objects) within Firestore documents, specifically removing a key-value pair to achieve deletion.
- **Conditional Logic**: Includes a check for document existence (`docSnap.exists()`) before attempting deletion, handling cases where the document might not exist.

---

5. `addComment` (from `components\DoubtTracker.tsx` or `components\IdeaTracker.tsx`)

The `addComment` function adds a new comment to a doubt or idea in Firestore. It uses `addDoc` to create a new comment document within the `comments` subcollection, which is nested under a specific doubt or idea document. This demonstrates how to structure nested data in Firestore for hierarchical relationships, such as comments belonging to a doubt. The function also uses `serverTimestamp()` to record the comment's creation time on the Firestore server.

```
const addComment = async (
  doubtId: string,
  commentText: string,
  parentId?: string
) ⇒ {
  if (!user) return;
  try {
    // Get reference to the 'comments' subcollection within the specific doubt document.
    const commentsRef = collection(
      db,
      "users",
      user.id,
      "nugget",
      doubtId,
      "comments"
    );
    // Add a new comment document to the 'comments' subcollection.
    await addDoc(commentsRef, {
      text: commentText,
        parentId: parentId || null, // Store parentId if it's a reply, otherwise null for top-level comments.
        createdAt: serverTimestamp(), // Use Firestore server timestamp for comment creation time.
    });
  } catch (error) {
    console.error("Error adding comment:", error);
  }
};
```

Firestore Techniques Highlighted:

- **`addDoc` to Subcollections**: Demonstrates adding data to a Firestore subcollection, showcasing how to model one-to-many relationships.
- **Nested Data Structure**: Illustrates the creation of nested data within Firestore, using subcollections to organize comments under their parent doubts or ideas.
- **`serverTimestamp`**: Shows the use of `serverTimestamp()` to ensure accurate and consistent timestamping of data entries directly from the Firestore server.

---

6. `resolveDoubt/resolveIdea` (from `components\DoubtTracker.tsx` or `components\IdeaTracker.tsx`)

The `resolveDoubt` (or `resolveIdea`) function updates the status of a doubt (or idea) to 'Resolved' in Firestore. It uses the `updateDoc` method to modify specific fields of an existing doubt/idea document, namely the `resolved` field (set to `true`) and the `solution` field (to store the provided solution text). This showcases how to selectively update specific attributes of a Firestore document.

```
  const resolveDoubt = async (id: string, solution: string) => {
    if (!user) return;
    try {
      // Get reference to the specific doubt document within the user's 'nugget' collection.
      const doubtRef = doc(db, "users", user.id, "nugget", id);
      // Update the doubt document to mark it as resolved and add the provided solution.
      await updateDoc(doubtRef, { resolved: true, solution });
    } catch (error) {
      console.error("Error resolving doubt:", error);
  }
};
```

Firestore Techniques Highlighted:

- `updateDoc`: Demonstrates the use of `updateDoc` to selectively modify specific fields within an existing Firestore document.
- **Selective Field Updates**: Illustrates updating only the `resolved` status and `solution` fields of a document, without affecting other data.
- **Boolean and String Data Types**: Shows how to update fields with different data types (boolean for `resolved`, string for `solution`) in Firestore.

---

7. `useEffect` with `onSnapshot` (from `components\DoubtTracker.tsx` or `components\IdeaTracker.tsx`)

This `useEffect` hook, implemented in both `DoubtTracker.tsx` and `IdeaTracker.tsx`, sets up a real-time listener for changes in a Firestore collection (either "nugget" for doubts or "ideas" for ideas). It uses `onSnapshot` to subscribe to real-time updates, ensuring that the application UI automatically reflects any changes made to the data in Firestore without requiring manual refreshes. The hook processes the snapshot data to update the component's state, displaying the latest data.

```
useEffect(() ⇒ {
  if (!user) return;
  const nuggetRef = collection(db, "users", user.id, "nugget");
  const q = query(nuggetRef, orderBy("createdAt", "desc"));
  const unsubscribe = onSnapshot(q, (snapshot) ⇒ {
    const loadedDoubts: Doubt[] = [];
    snapshot.forEach((docSnap) ⇒ {
      loadedDoubts.push({
        id: docSnap.id,
        title: docSnap.data().title,
        description: docSnap.data().description,
        upvotes: docSnap.data().upvotes,
        downvotes: docSnap.data().downvotes,
        resolved: docSnap.data().resolved,
        solution: docSnap.data().solution,
        comments: [],
      });
    });
    setDoubts(loadedDoubts);
  });
  return () ⇒ unsubscribe();
}, [user]);
```

Firestore Techniques Highlighted:

- `onSnapshot`: Demonstrates the use of `onSnapshot` to establish a real-time listener for changes in a Firestore collection.
- **Real-time Data Synchronization**: Showcases how to implement real-time updates in a React application using Firestore, ensuring data consistency across clients.
- **Querying and Ordering Data**: Includes a Firestore query (`query`, `orderBy`) within the `onSnapshot` setup, demonstrating how to fetch and order data in real-time.

---

8. Debounced `useEffect` for `updateNotebook` (from `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace\[notebookId]\page.tsx`)

This `useEffect` hook, found in the Notebook View component, demonstrates a performance optimization technique for Firestore operations. It uses `setTimeout` to debounce calls to `updateNotebook`, which persists changes to notebook data in Firestore. This debouncing mechanism batches rapid UI updates, such as those from drag-and-drop interactions, into less frequent database write operations, improving application performance and reducing Firestore costs.

```
useEffect(() ⇒ {
  if (!user || !notebookId) return;

  // Add debounce to prevent too many writes
```

```
    const timeoutId = setTimeout(async () => {
      try {
        await updateNotebook(notebookId, sections);
      } catch (error) {
        console.error("Error updating notebook:", error);
      }
    }, 1000); // Wait 1 second after last change

    return () => clearTimeout(timeoutId);
  }, [sections, user, notebookId]);
```

Firestore Techniques Highlighted:

- **Performance Optimization**: Illustrates a practical approach to optimizing Firestore write operations using debouncing.
- `setTimeout` **and** `clearTimeout`: Shows the implementation of debouncing using JavaScript's `setTimeout` and `clearTimeout` functions within a `useEffect` hook.
- **Controlled Database Writes**: Demonstrates how to control the frequency of database write operations in response to rapid user interface interactions, enhancing efficiency.

---

9. `handleDragEnd` (from `app\ContinuousInfoSpaceDocMan\ContinuousInfoSpace\[notebookId]\page.tsx`)

The `handleDragEnd` function, used in the Notebook View, manages the persistence of drag-and-drop reordering actions for notes and columns in Firestore. While the drag-and-drop logic itself is client-side, this function is crucial for taking the results of a drag operation and translating them into updates to the Firestore database. It calls `moveNote` or `moveColumn` functions, which in turn trigger the debounced `updateNotebook` `useEffect` to save the changes to Firestore.

```
  const handleDragEnd = (event: DragEndEvent) => {
    const { active, over } = event;
```

```
setActiveNote(null);

if (!over) {
  return;
}

const activeId = active.id as string;
const activeData = active.data.current as DragData;
const overData = over.data.current as DragData;

if (activeData.type === "column" && overData.type === "column") {
  const sectionId = activeData.sectionId;
  const section = sections.find((s) => s.id === sectionId);
  if (!section) return;

  const fromIndex = section.columns.findIndex(
    (c) => c.id === activeData.columnId
  );
  const toIndex = section.columns.findIndex(
    (c) => c.id === overData.columnId
  );

  if (fromIndex !== toIndex) {
    moveColumn(sectionId, fromIndex, toIndex);
  }
  return;
}

if (activeData.type === "note" || !activeData.type) {
  if (!overData?.type || !overData.columnId) {
    return;
  }

  let fromSectionId, fromColumnId;
  sections.some((section) => {
    return section.columns.some((column) => {
      const found = column.notes.some((note) => note.id === activeId);
      if (found) {
        fromSectionId = section.id;
        fromColumnId = column.id;
        return true;
        }
        return false;
      });
    });

    const toSectionId = overData.sectionId;
    const toColumnId = overData.columnId;

    if (fromSectionId && fromColumnId && toSectionId && toColumnId) {
      moveNote(
        fromSectionId,
        fromColumnId,
        toSectionId,
        toColumnId,
        activeId
      );
    }
  }
};
```

Firestore Techniques Highlighted:

- **Indirect Firestore Update**: Demonstrates how UI interactions (drag and drop) trigger data persistence in Firestore, showcasing the flow from user action to database update.
- **State Management as an Interface to Firestore**: Illustrates how UI state updates (via `moveNote`, `moveColumn`) are used as an intermediary step before data is written to Firestore via the debounced `useEffect`, controlling data flow and update frequency.
- **Orchestration of UI and Database Operations**: Shows how a UI event handler (`handleDragEnd`) orchestrates both UI state changes and subsequent database update operations.

-

Other Common Code

Usage of `useToast` Custom Hook

This section showcases the practical application of the `useToast` custom hook within your application. The `useToast` hook, defined in `hooks\use-toast.ts`, provides a reusable mechanism for displaying non-intrusive toast notifications to provide user feedback. This example highlights how the hook is invoked within a component to trigger a toast message, enhancing user experience by providing immediate confirmation or affirmation of actions.

```
import { useToast } from "@/hooks/use-toast";
...
const ToDoList = ({
...
}: ToDoListProps) => {
...
  const { toast } = useToast();
...
  const deleteSection = (sectionId: string) => {
    const section = sections.find((s) => s.id === sectionId);
    setSections(sections.filter((section) => section.id !== sectionId));
    toast({
      title: "Section deleted",
      description: `${section?.title || "Section"} has been deleted.`,
      action: (
        <ToastAction
          altText="Undo"
          onClick={() => {
            if (section) setSections((prev) => [...prev, section]);
          }}
        >
          Undo
        </ToastAction>
      ),
    });
  };
```

Technique Highlight:

- **Custom Hooks for User Feedback**: Demonstrates the use of custom hooks to encapsulate and reuse UI logic for displaying user feedback, promoting code reusability and improved user experience through non-intrusive notifications.

`cn` Utility Function Usage

This section demonstrates the use of the `cn` (classNames) utility function, defined in `lib\utils.ts`, for efficient and conditional class name management within React components. This function leverages `clsx` and `tailwind-merge` to handle conditional class names and resolve Tailwind CSS class conflicts. The example will show how `cn` is used within a component's `className` prop to dynamically apply styles.

button.tsx

```
const Button = React.forwardRef<HTMLButtonElement, ButtonProps>(
  ({ className, variant, size, asChild = false, ...props }, ref) => {
    const Comp = asChild ? Slot : "button"
    return (
      <Comp
        className={cn(buttonVariants({ variant, size, className }))}
        ref={ref}
        {...props}
      />
    )
  }
)
Button.displayName = "Button"
```

Technique Highlight:

- **Utility Functions for Code Organization**: Illustrates the use of utility functions to promote code organization and reusability, particularly for managing complex class name logic in React components, and for efficient Tailwind CSS styling.

`BackButton` Component

This section showcases the `BackButton` custom UI component, defined in `components\ui\custom-ui\back-button.tsx`. This component encapsulates the navigation logic for moving back to the appropriate parent route within the application. By abstracting this navigation pattern into a reusable component, the codebase is simplified, and navigation logic is centralized.

```tsx
export function BackButton() {
  const router = useRouter();
  const pathname = usePathname();

  const getParentPath = (path: string) => {
    const segments = path.split("/").filter(Boolean);
    const parentSegments = segments.slice(0, -1);
    return parentSegments.length ? `/${parentSegments.join("/")}` : "/";
  };

  const checkPageExists = async (path: string): Promise<boolean> => {
    try {
      const response = await fetch(path);
      return response.status !== 404;
    } catch (error) {
      return false;
    }
  };

  const findValidParentPath = async (currentPath: string): Promise<string> => {
    if (currentPath === "/") return "/";

    const exists = await checkPageExists(currentPath);
    if (exists) return currentPath;

    const parentPath = getParentPath(currentPath);
    return findValidParentPath(parentPath);
  };

  const handleBack = async () => {
    const initialParentPath = getParentPath(pathname);
    const validPath = await findValidParentPath(initialParentPath);
    router.push(validPath);
  };

  return (
    <Button variant="ghost" onClick={handleBack}>
      <ArrowLeft className="h-4 w-4" />
      Back
    </Button>
  );
}
```

Technique Highlight:

- **Reusable UI Components for Navigation**: Demonstrates the creation and use of custom UI components to encapsulate common navigation patterns, enhancing code reusability and maintainability across different pages.

CSS `@layer utilities` Usage

This section highlights the use of CSS `@layer utilities` within your `globals.css` file, specifically showcasing the `.text-balance` utility class. This demonstrates how Tailwind CSS's layer feature is used to extend and organize utility classes, improving CSS maintainability and organization by separating utility classes into logical layers.

```
@layer utilities {
  .text-balance {
    text-wrap: balance;
  }
}
```

Technique Highlight:

- **CSS Layering for Organization**: Demonstrates the use of CSS layering in Tailwind CSS to improve the organization and maintainability of stylesheets, separating utility classes into logical groups for better code management.

Type Definitions/Interfaces Example

This section provides an example of a TypeScript interface or type definition used in your codebase. The example will showcase the `Cookie` interface from `lib\cookies-actions.ts` (or another relevant interface like `Note`, `Column`, or `Section` interfaces), highlighting how TypeScript interfaces are used to define data structures and enforce type safety within the application.

```
interface Note {
  id: string;
  content: string;
}

interface Column {
  id: string;
  title: string;
  notes: Note[];
  isEditing: boolean;
}

interface Section {
  id: string;
  title: string;
  columns: Column[];
  isEditing: boolean;
  isArchived?: boolean;
  isPreviewingArchived?: boolean;
}

interface DragData {
  type: "note" | "column";
  sectionId: string;
  columnId?: string;
  noteId?: string;
}
```

Technique Highlight:

- **Typescript Interfaces for Type Safety**: Demonstrates the use of TypeScript interfaces to define data structures and enforce type safety, enhancing code maintainability, reducing errors, and improving overall code quality through static typing.

-

Database Implementation

.env.local

Stores environment variables specific to the development environment, mainly Firebase and Clerk API keys and secrets. This file must not be revealed to the client side, but it has been pasted here for reference.

```
# Firebase Variables
NEXT_PUBLIC_FIREBASE_API_KEY=AIzaSyBQHNUrVdXgV3fY0RDh52Etg3dqywnPWIc
NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN=ibdp-sims.firebaseapp.com
NEXT_PUBLIC_FIREBASE_PROJECT_ID=ibdp-sims
NEXT_PUBLIC_FIREBASE_STORAGE_BUCKET=ibdp-sims.firebasestorage.app
NEXT_PUBLIC_FIREBASE_MESSAGING_SENDER_ID=86203166000
NEXT_PUBLIC_FIREBASE_APP_ID=1:86203166000:web:22a76045c725eb788facf5
# Clerk Authentication Variables
NEXT_PUBLIC_CLERK_PUBLISHABLE_KEY=pk_test_bWFpbi1idWxsZnJvZy0yNS5jbGVyay5hY2NvdW50cy5kZXYk
CLERK_SECRET_KEY=sk_test_1KVagQnu4CNvvI5wyddaDAVtnF5XAmW2ZYxG5pE1SX
```

firebase.ts

Initialized the Firebase application and provided access to the Firestore database instance for use throughout the application.

```
import { initializeApp, getApps, getApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";

const firebaseConfig = {
  apiKey: process.env.NEXT_PUBLIC_FIREBASE_API_KEY,
  authDomain: process.env.NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN,
  projectId: process.env.NEXT_PUBLIC_FIREBASE_PROJECT_ID,
  storageBucket: process.env.NEXT_PUBLIC_FIREBASE_STORAGE_BUCKET,
  messagingSenderId: process.env.NEXT_PUBLIC_FIREBASE_MESSAGING_SENDER_ID,
  appId: process.env.NEXT_PUBLIC_FIREBASE_APP_ID,
};

if (Object.values(firebaseConfig).some(value ⇒ !value)) {
  throw new Error("Missing Firebase environment variables");
}

const app = getApps().length > 0 ? getApp() : initializeApp(firebaseConfig);
const db = getFirestore(app);

export { app, db };
```

firestore.indexes.json

This configuration file defines composite indexes for Firestore queries

```
{
  "indexes": [
    {
      "collectionGroup": "cookieJar",
      "queryScope": "COLLECTION",
      "fields": [
        {
          "fieldPath": "position.y",
          "order": "ASCENDING"
        },
        {
          "fieldPath": "position.x",
          "order": "ASCENDING"
        }
      ]
    }
  ],
  "fieldOverrides": []
}
```

For every file that required database interactions, the following import statement was added to enable firestore interactions. This is a general representation and not all of the below imported methods are used in a single file.

```
import { db } from "@/lib/firebase";
import { doc, getDoc, getDocs, setDoc, addDoc, updateDoc, deleteDoc, collection, query, orderBy,
onSnapshot, increment, serverTimestamp } from "firebase/firestore";
```

-

Libraries

The following is the package.json file of the application, containing all the dependencies for the application.

```json
{
  "name": "ibdp-sims",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  },
  "dependencies": {
    "@clerk/nextjs": "^6.10.6",
    "@dnd-kit/core": "^6.2.0",
    "@dnd-kit/sortable": "^9.0.0",
    "@dnd-kit/utilities": "^3.2.2",
    "@headlessui/react": "^2.2.0",
    "@radix-ui/react-checkbox": "^1.1.4",
    "@radix-ui/react-dialog": "^1.1.4",
    "@radix-ui/react-dropdown-menu": "^2.1.4",
    "@radix-ui/react-label": "^2.1.1",
    "@radix-ui/react-scroll-area": "^1.2.2",
    "@radix-ui/react-separator": "^1.1.1",
    "@radix-ui/react-slot": "^1.1.1",
    "@radix-ui/react-switch": "^1.1.2",
    "@radix-ui/react-tabs": "^1.1.1",
    "@radix-ui/react-toast": "^1.2.4",
    "@radix-ui/react-tooltip": "^1.1.6",
    "class-variance-authority": "^0.7.1",
    "classnames": "^2.5.1",
    "clsx": "^2.1.1",
    "dynamic": "^4.2.2",
    "firebase": "^11.2.0",
    "lucide-react": "^0.462.0",
    "next": "^14.2.17",
    "next-themes": "^0.4.4",
    "path": "^0.12.7",
    "react": "^18",
    "react-dom": "^18",
    "react-resizable-panels": "^2.1.7",
    "tailwind-merge": "^2.5.5",
    "tailwindcss-animate": "^1.0.7",
    "zod": "^3.24.1"
  },
  "devDependencies": {
    "@types/node": "^20",
    "@types/react": "^18",
    "@types/react-dom": "^18",
    "eslint": "^8",
    "eslint-config-next": "14.2.17",
    "postcss": "^8",
    "tailwindcss": "^3.4.1",
    "typescript": "^5"
  }
}
```

-

Bibliography

Bell, Joe. Class Variance Authority. n.d., cva.style/docs. Accessed 15 Nov. 2024.

Castillo, Dany (dcastil). tailwind-merge. GitHub, GitHub, Inc., n.d., github.com/dcastil/tailwind-merge. Accessed 15 Nov. 2024.

Clerk. Clerk Documentation. Clerk, n.d., clerk.com/docs. Accessed 7 Nov. 2024.

Demers, Claudéric (clauderic). Dnd Kit Documentation. n.d., docs.dndkit.com/. Accessed 5 Dec. 2024.

Edwards, Luke (lukeed). clsx. GitHub, GitHub, Inc., n.d., github.com/lukeed/clsx. Accessed 15 Nov. 2024.

Google. "Cloud Firestore Documentation." Firebase, Google, n.d., firebase.google.com/docs/firestore. Accessed 8 Nov. 2024.

Lucide Contributors. Lucide. n.d., lucide.dev/. Accessed 14 Nov. 2024.

McDonnell, Colin (colinhacks), and contributors. Zod. n.d., zod.dev/. Accessed 20 Nov. 2024.

Meta. React. Meta, n.d., react.dev/. Accessed 5 Nov. 2024.

Microsoft. TypeScript Handbook. TypeScript, Microsoft, n.d., www.typescriptlang.org/docs/handbook/. Accessed 6 Nov. 2024.

Mozilla Developer Network Contributors. MDN Web Docs. Mozilla, n.d., developer.mozilla.org/. Accessed 15 Dec. 2024.

Refaiea, Yahia (yahiarefaiea). React Resizable Panels. n.d., react-resizable-panels.yahia.io/. Accessed 10 Dec. 2024.

shadcn. shadcn/ui. n.d., ui.shadcn.com/docs. Accessed 12 Nov. 2024.

Tailwind Labs. Tailwind CSS Documentation. Tailwind Labs, n.d., tailwindcss.com/docs. Accessed 10 Nov. 2024.

Vercel. Next.js Documentation. Vercel, n.d., nextjs.org/docs. Accessed 5 Nov. 2024.

WorkOS. Radix Primitives Documentation. Radix UI, WorkOS, n.d., www.radix-ui.com/primitives/docs. Accessed 13 Nov. 2024.

Claude 3.5 Sonnet. Anthropic, 20 June 2024.

ChatGPT o1. OpenAI, 5 December 2024.

ChatGPT 4o. OpenAI, 13 May 2024.

DeepSeek R1. DeepSeek, 20 January 2025.

DeepSeek V3. DeepSeek, December 2024.

Gemini 1.5 Pro. Google, 23 May 2024.

Gemini 1.5 Flash. Google, 14 May 2024.

Gemini 2.0 Flash. Google, 5 February 2025.

Gemini 2.0 Flash Thinking Experimental 01-21. Google, 21 January 2025.