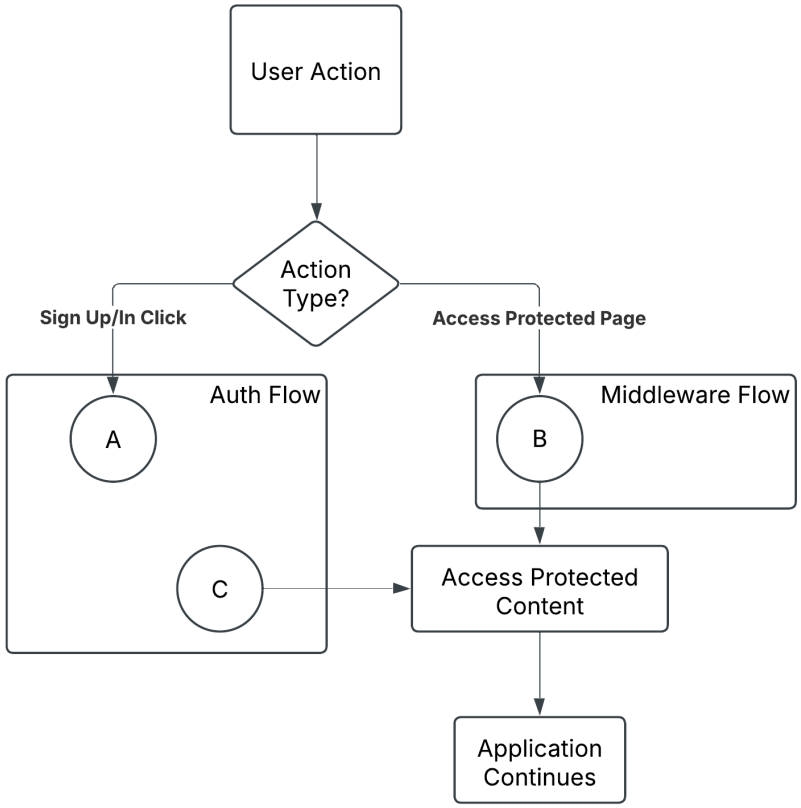
Information Management System

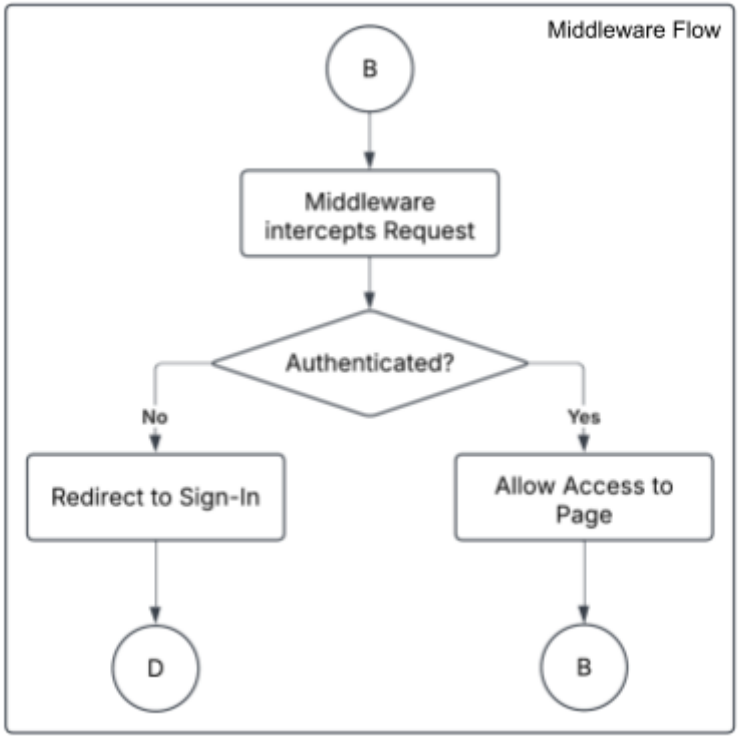
Criterion B - Design

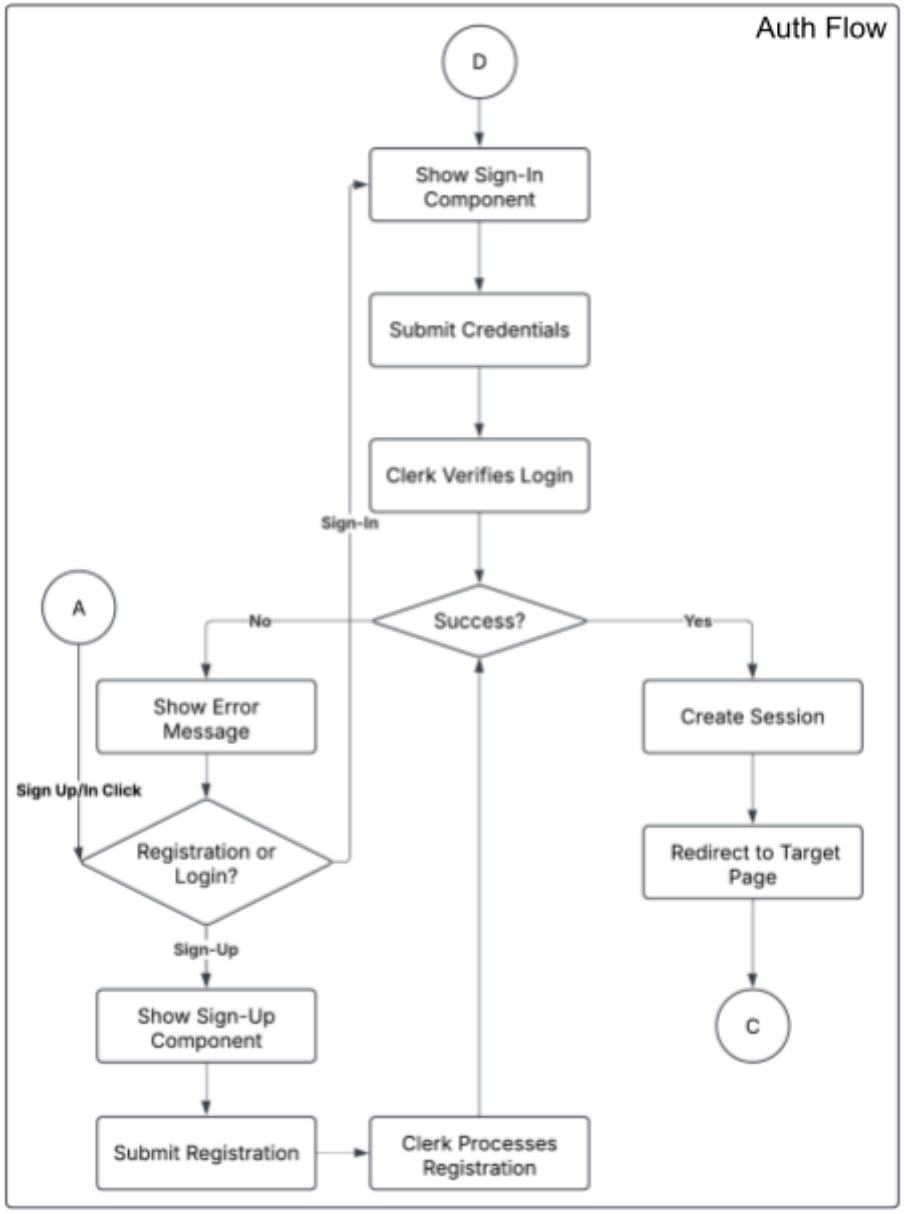
Table of contents:

System flowcharts

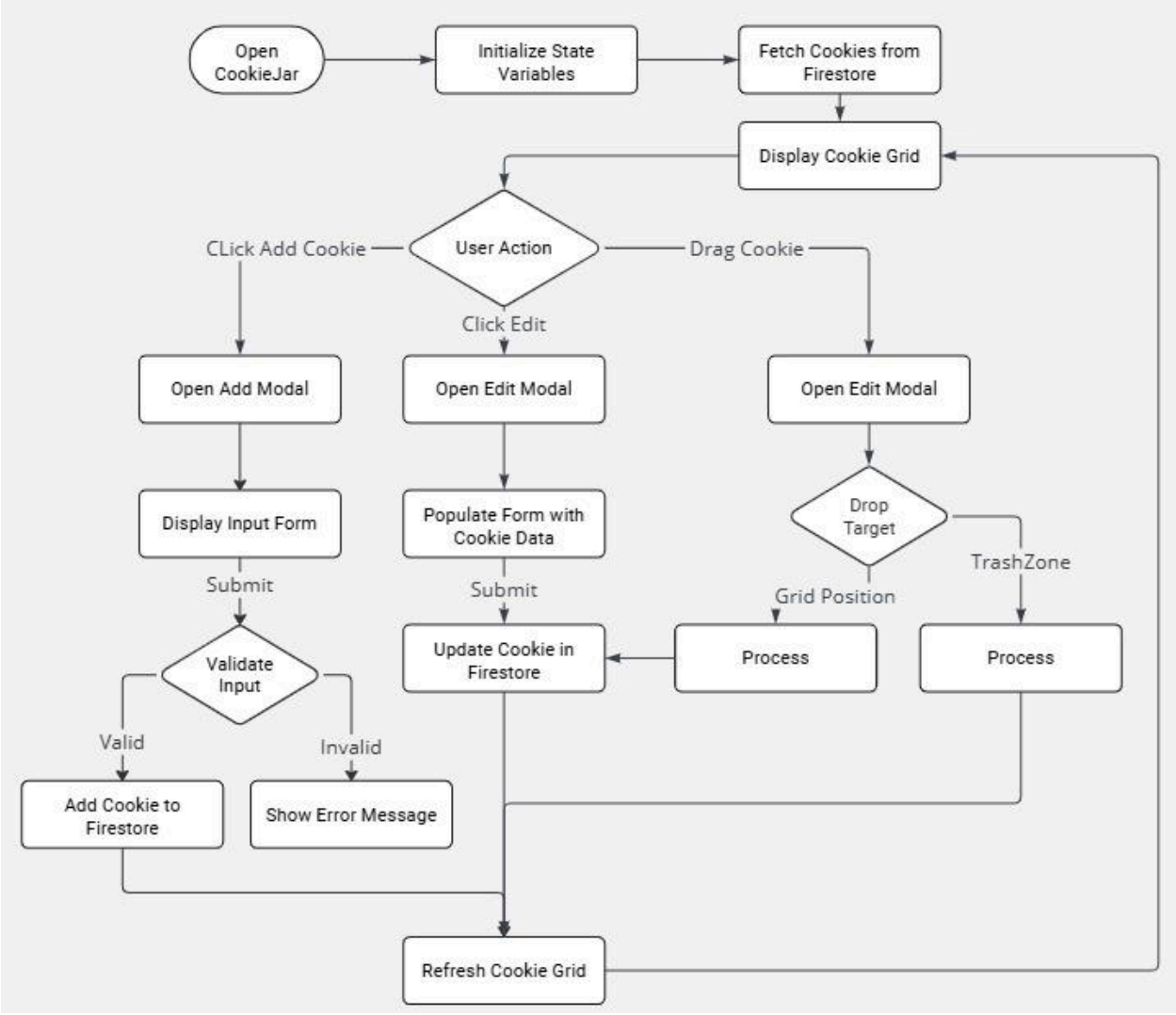
Authentication



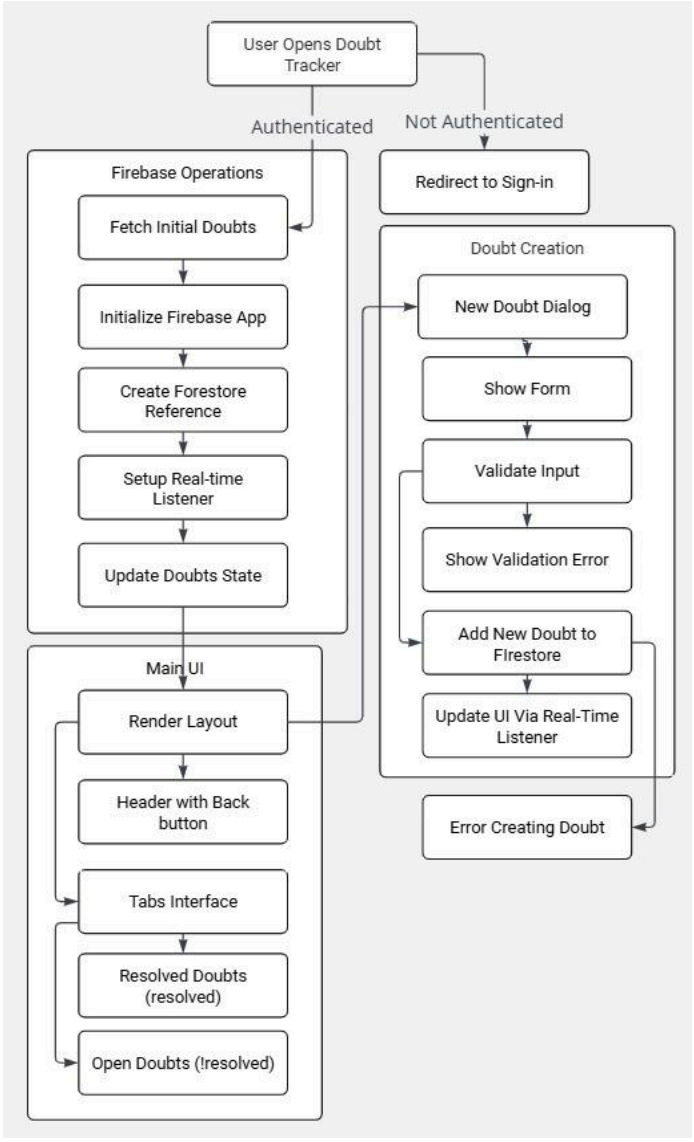

Middleware Flow
B
Middleware intercepts Request
Authenticated?
No
Yes
Redirect to Sign-In
Allow Access to Page
D
B

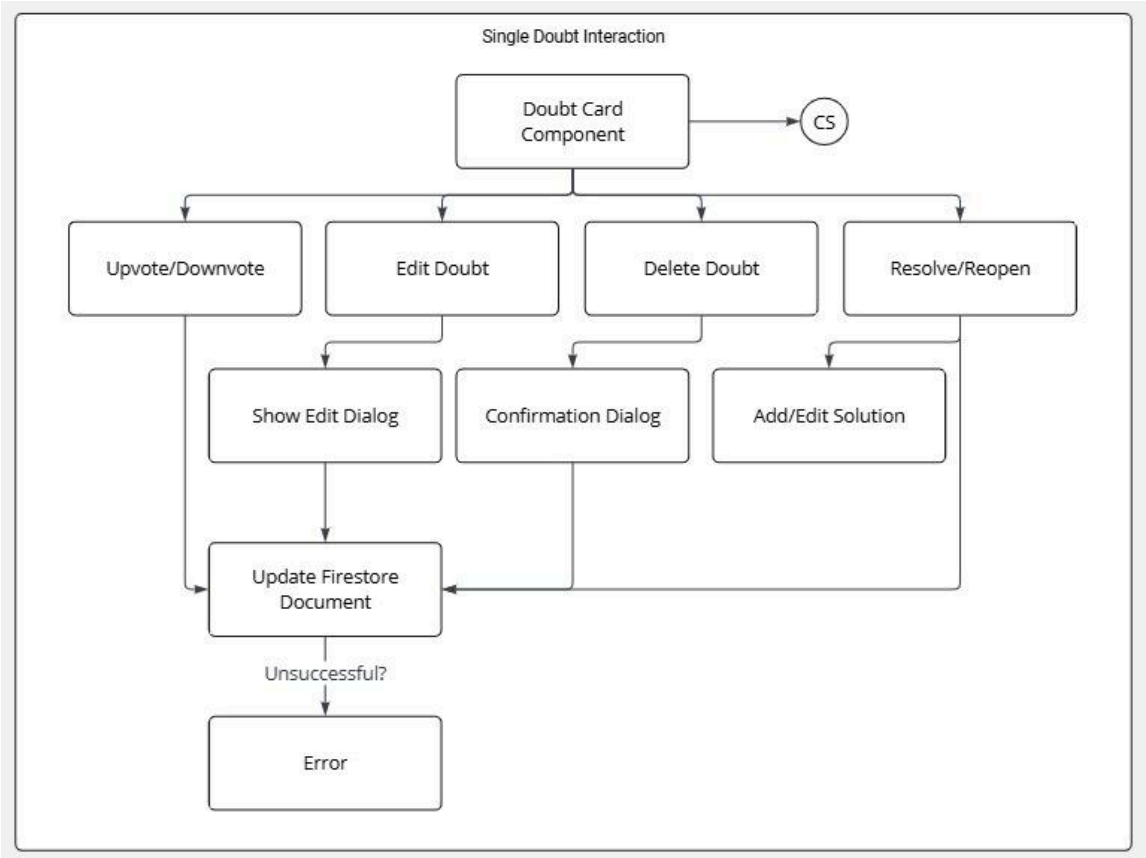

Auth Flow
D
Show Sign-In Component
Submit Credentials
Clerk Verifies Login
Sign-In
A
No
Success?
Yes
Show Error Message
Create Session
Sign Up/In Click
Registration or Login?
Redirect to Target Page
Sign-Up
C
Show Sign-Up Component
Submit Registration
Clerk Processes Registration

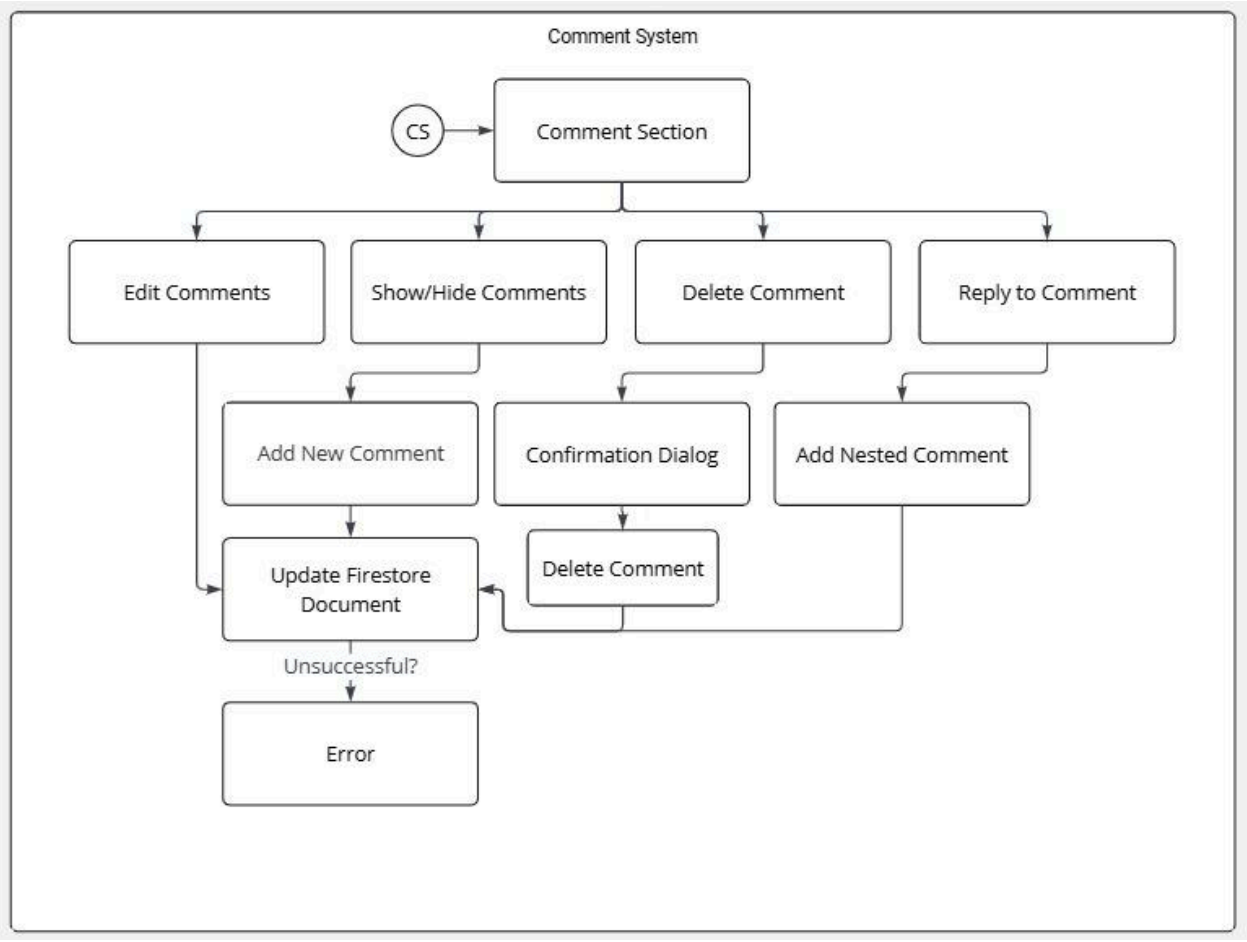
CookieJar


Open CookieJar
Initialize State Variables
Fetch Cookies from Firestore
Display Cookie Grid
User Action
CLick Add Cookie
Click Edit
Drag Cookie
Open Add Modal
Open Edit Modal
Open Edit Modal
Display Input Form
Populate Form with Cookie Data
Drop Target
TrashZone
Submit
Submit
Grid Position
Validate Input
Update Cookie in Firestore
Process
Process
Valid
Invalid
Add Cookie to Firestore
Show Error Message
Refresh Cookie Grid

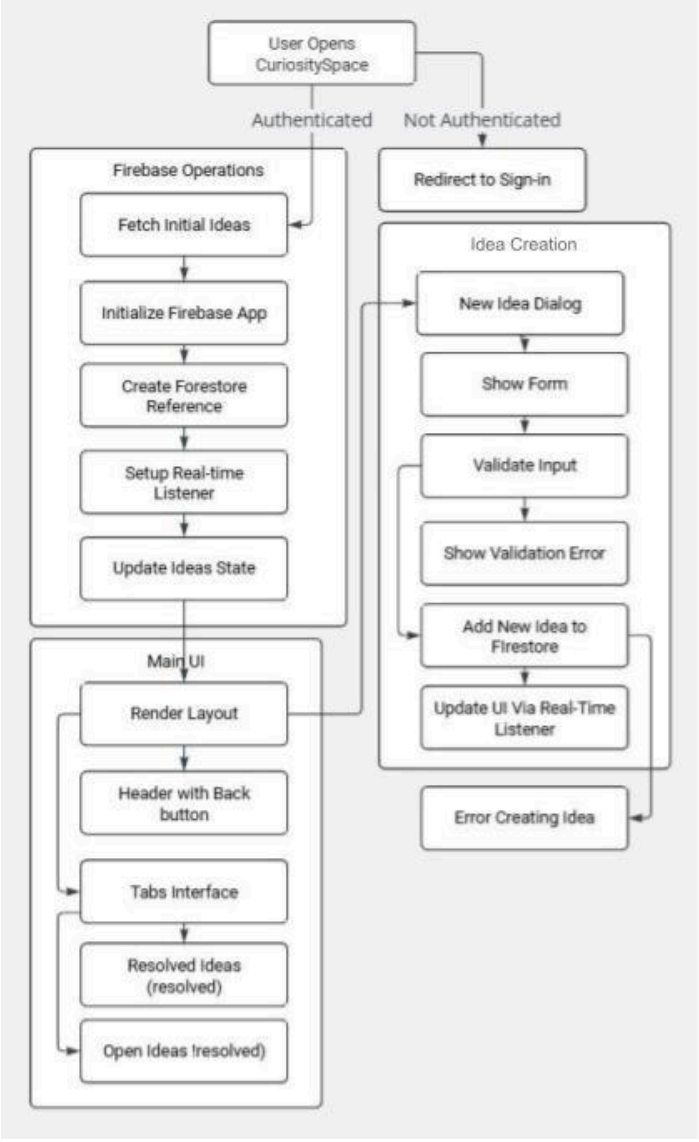
DoubtTracker

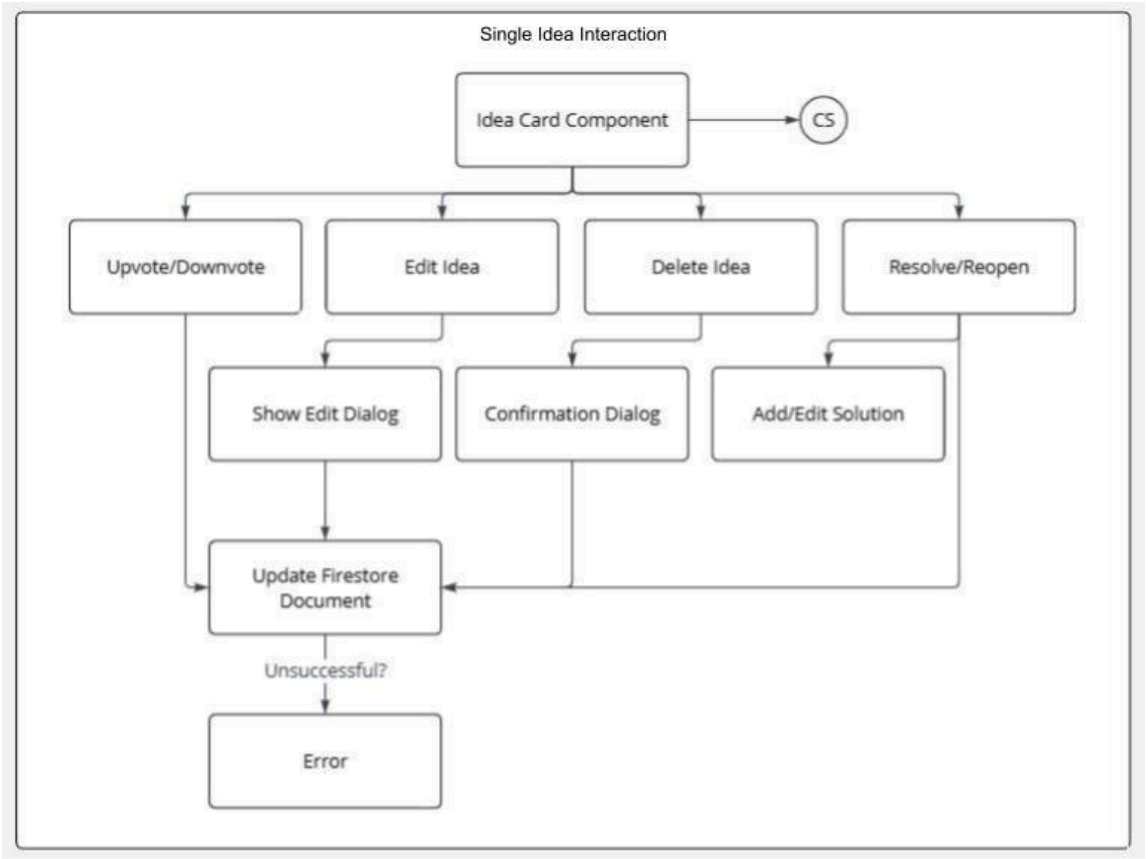


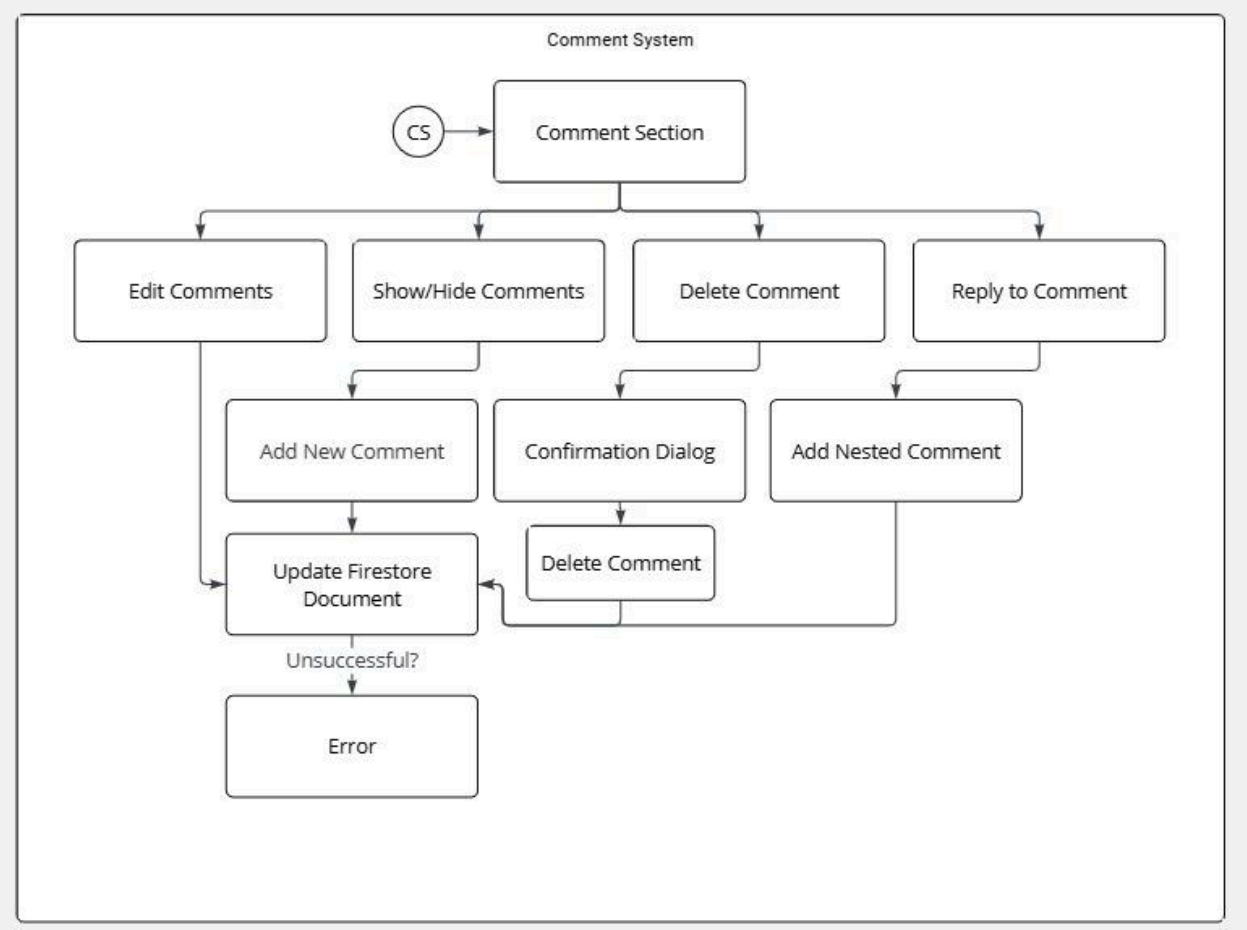



Comment System
CS
Comment Section
Edit Comments
Show/Hide Comments
Delete Comment
Reply to Comment
Add New Comment
Confirmation Dialog
Add Nested Comment
Update Firestore Document
Delete Comment
Unsuccessful?
Error

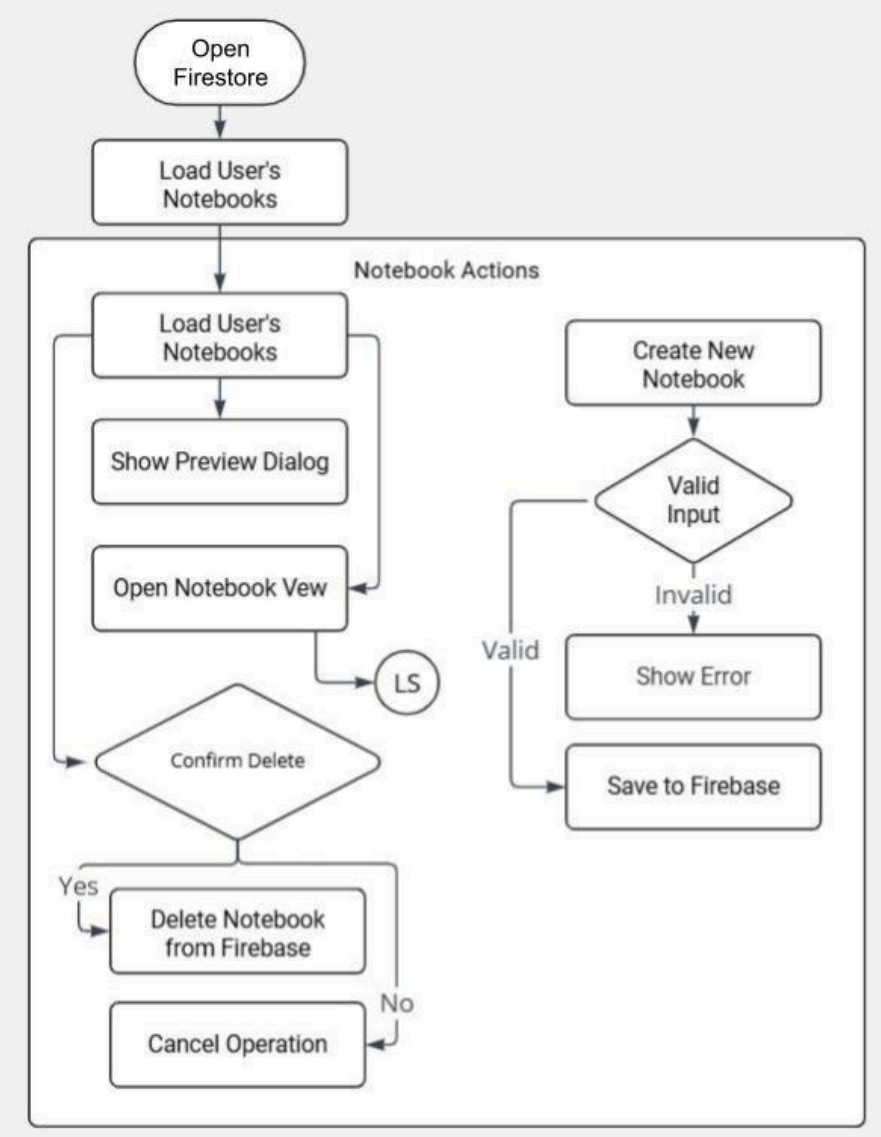
CuriositySpace

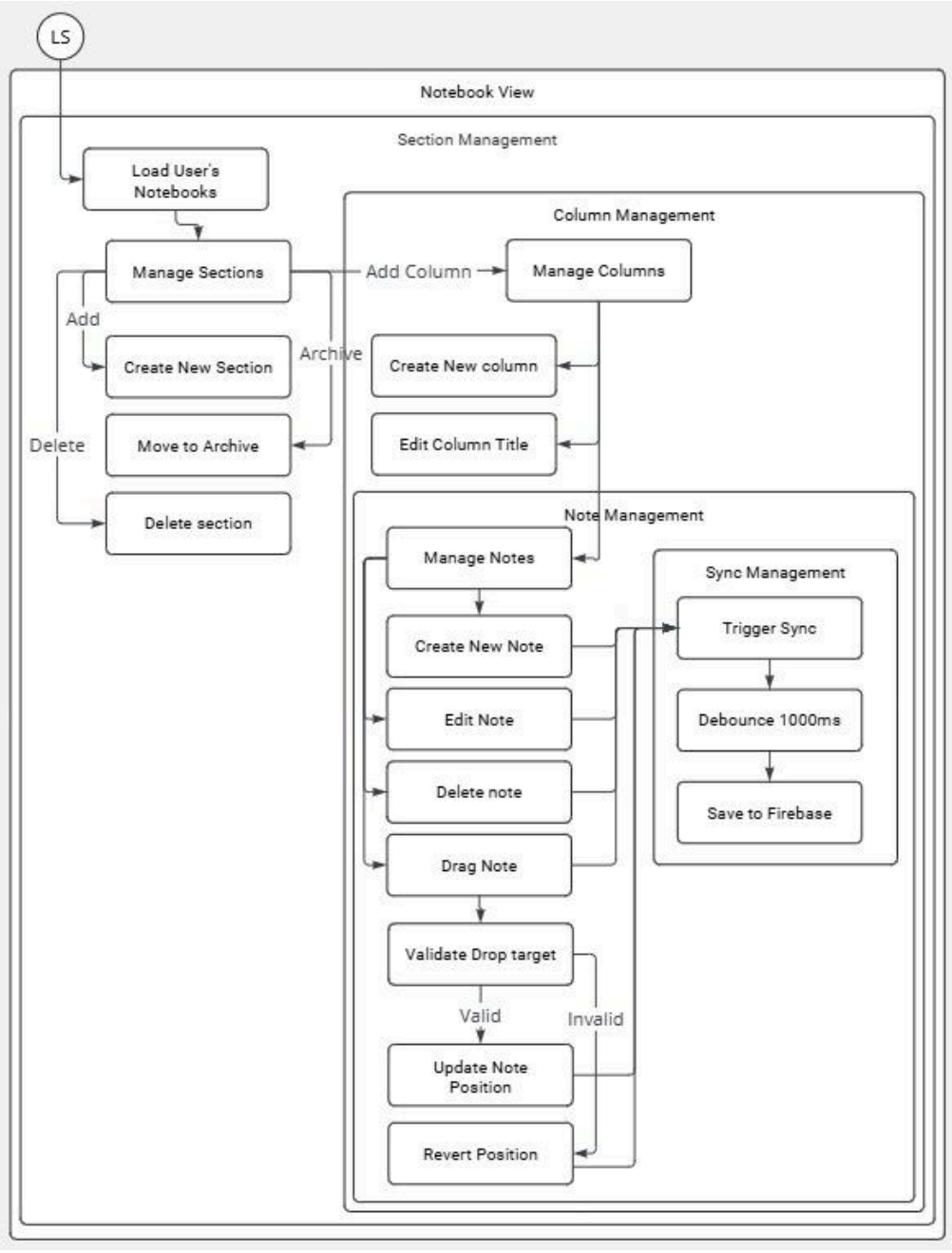




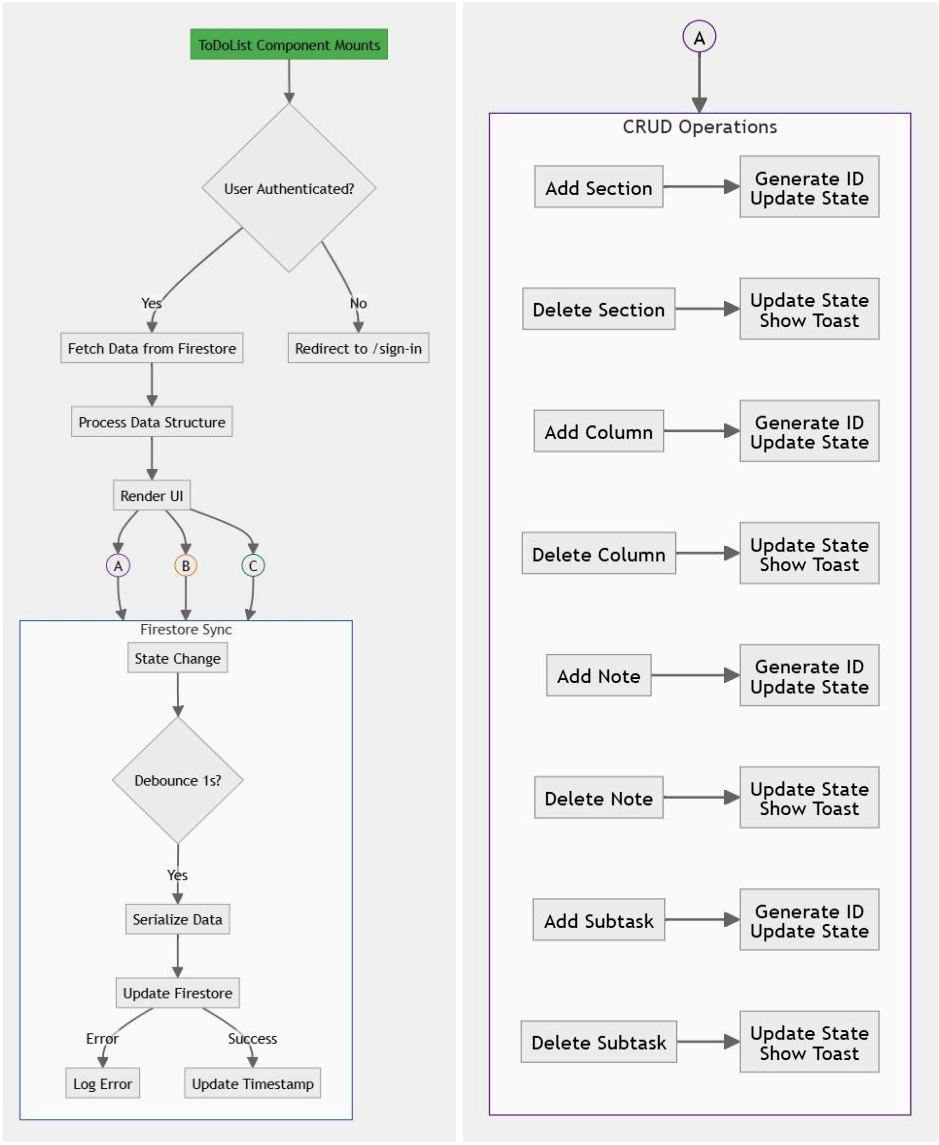

Comment System
CS
Comment Section
Edit Comments
Show/Hide Comments
Delete Comment
Reply to Comment
Add New Comment
Confirmation Dialog
Add Nested Comment
Update Firestore Document
Delete Comment
Unsuccessful?
Error

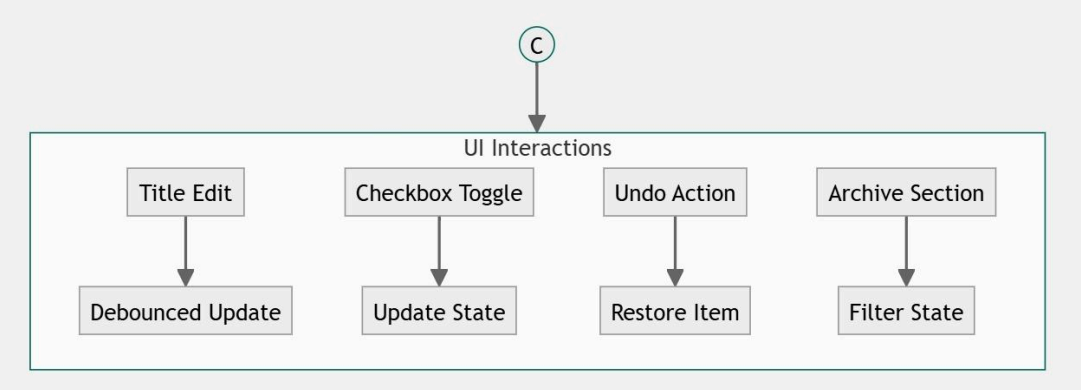
Notebooks

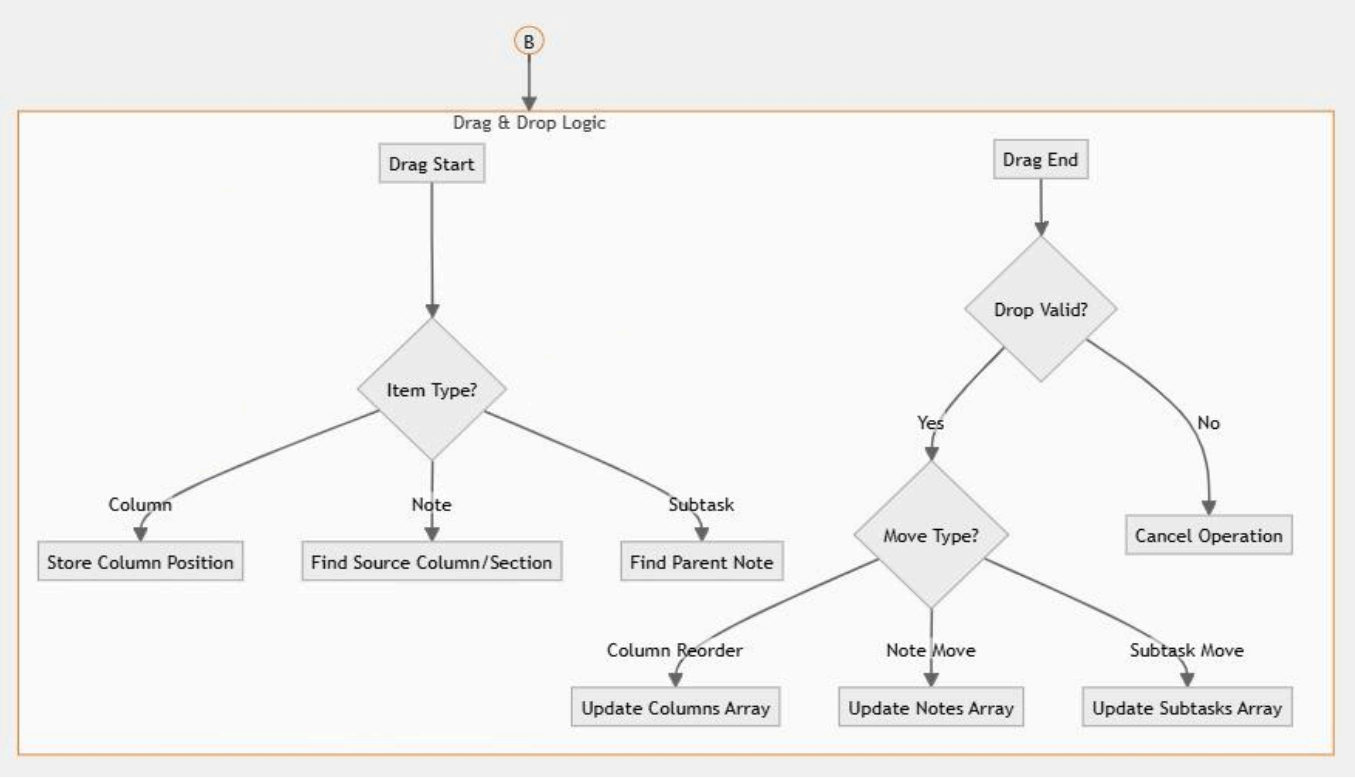

Open Firestore
Load User's Notebooks
Notebook Actions
Load User's Notebooks
Show Preview Dialog
Open Notebook Vew
LS
Confirm Delete
Yes
Delete Notebook from Firebase
No
Cancel Operation
Create New Notebook
Valid Input
Invalid
Valid
Show Error
Save to Firebase


LS
Notebook View
Section Management
Load User's Notebooks
Manage Sections
Add
Create New Section
Archive
Delete
Move to Archive
Delete section
Column Management
Add Column
Manage Columns
Create New column
Edit Column Title
Note Management
Manage Notes
Create New Note
Edit Note
Delete note
Drag Note
Validate Drop target
Valid
Invalid
Update Note Position
Revert Position
Sync Management
Trigger Sync
Debounce 1000ms
Save to Firebase

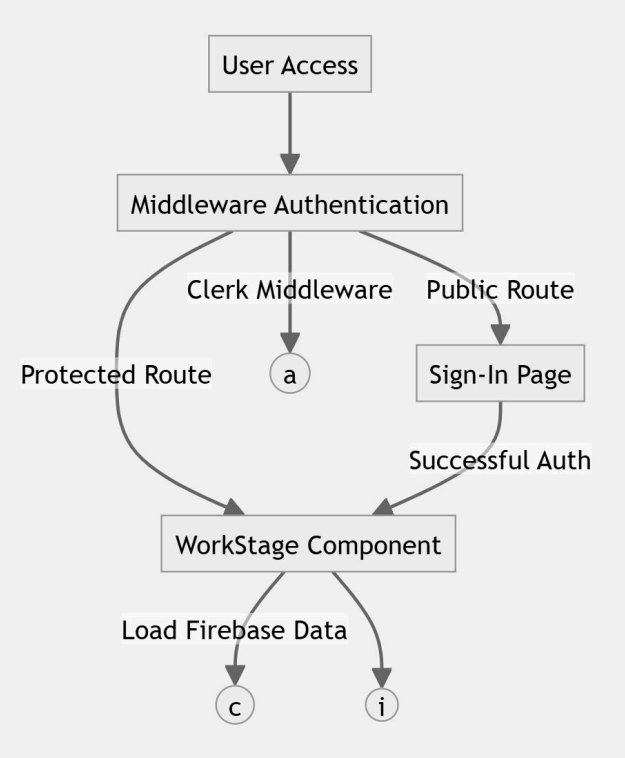
ToDoList

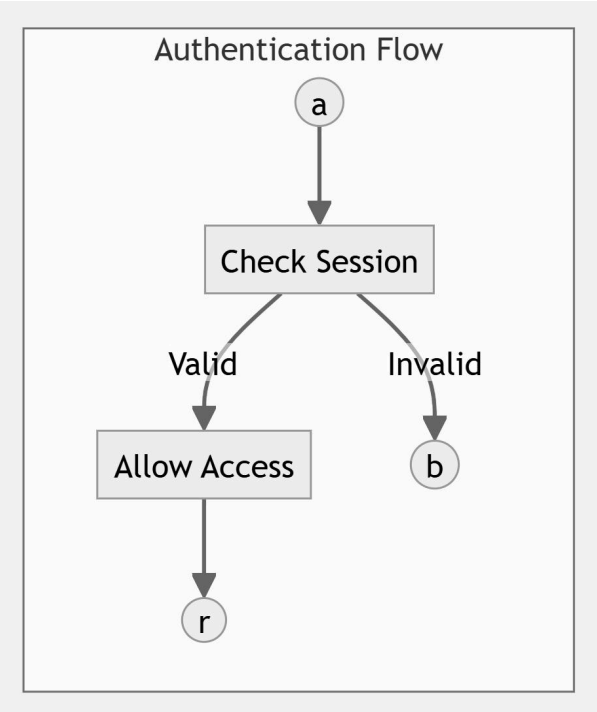

ToDoList Component Mounts
User Authenticated?
Yes
No
Fetch Data from Firestore
Redirect to /sign-in
Process Data Structure
Render UI
A
B
C
Firestore Sync
State Change
Debounce 1s?
Yes
Serialize Data
Update Firestore
Error
Success
Log Error
Update Timestamp
A
CRUD Operations
Add Section
Generate ID Update State
Delete Section
Update State Show Toast
Add Column
Generate ID Update State
Delete Column
Update State Show Toast
Add Note
Generate ID Update State
Delete Note
Update State Show Toast
Add Subtask
Generate ID Update State
Delete Subtask
Update State Show Toast

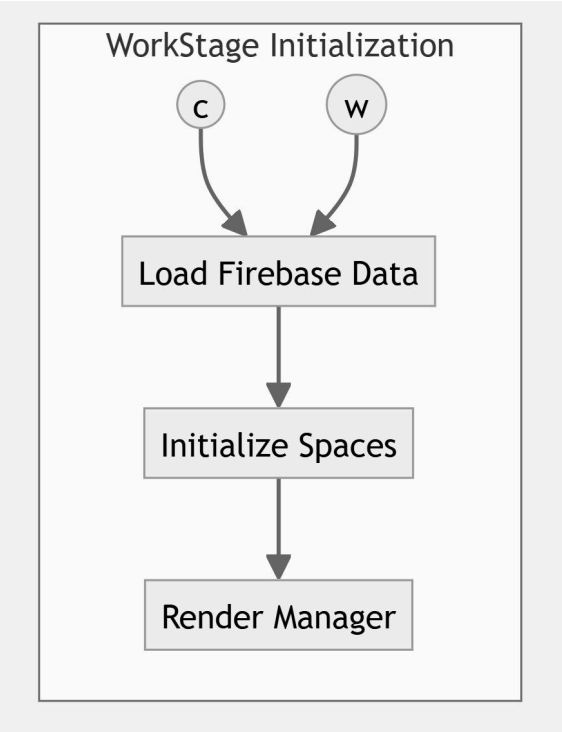

C
UI Interactions
Title Edit
Checkbox Toggle
Undo Action
Archive Section
Debounced Update
Update State
Restore Item
Filter State

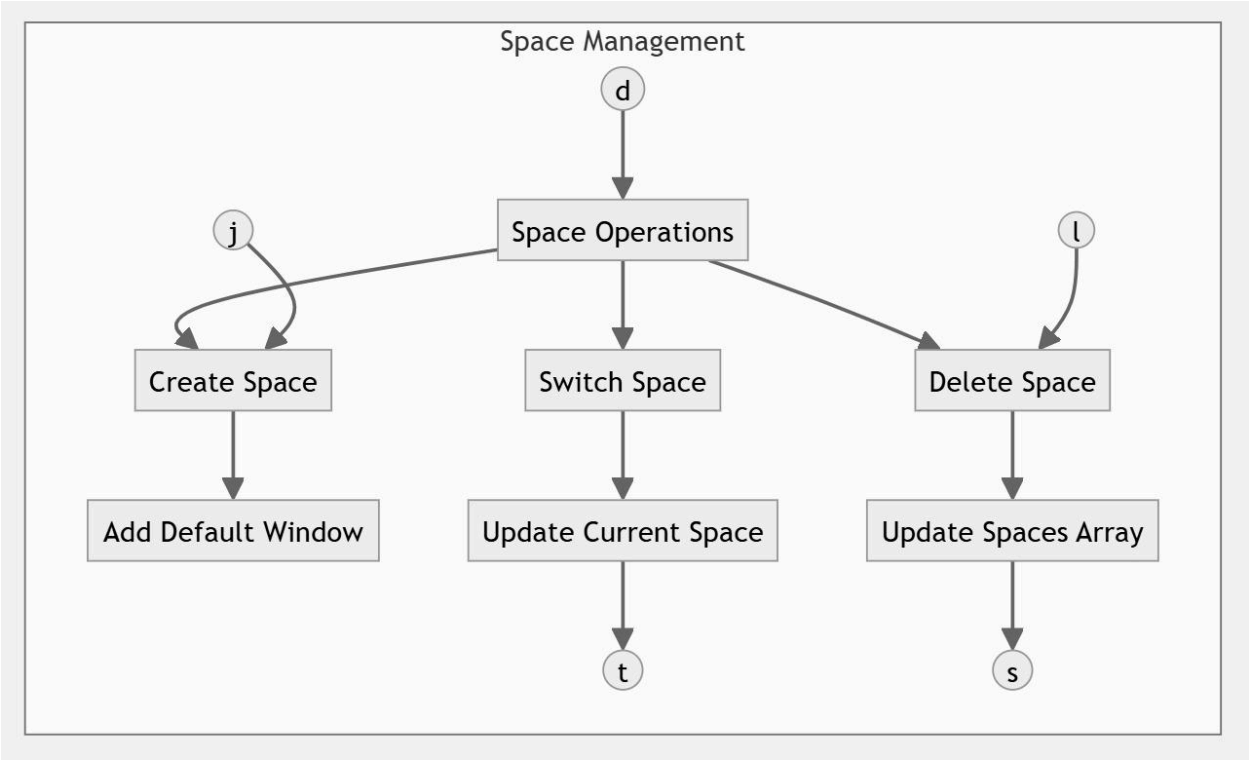

B
Drag & Drop Logic
Drag Start
Drag End
Item Type?
Drop Valid?
Column
Note
Subtask
Yes
No
Store Column Position
Find Source Column/Section
Find Parent Note
Move Type?
Cancel Operation
Column Reorder
Note Move
Subtask Move
Update Columns Array
Update Notes Array
Update Subtasks Array

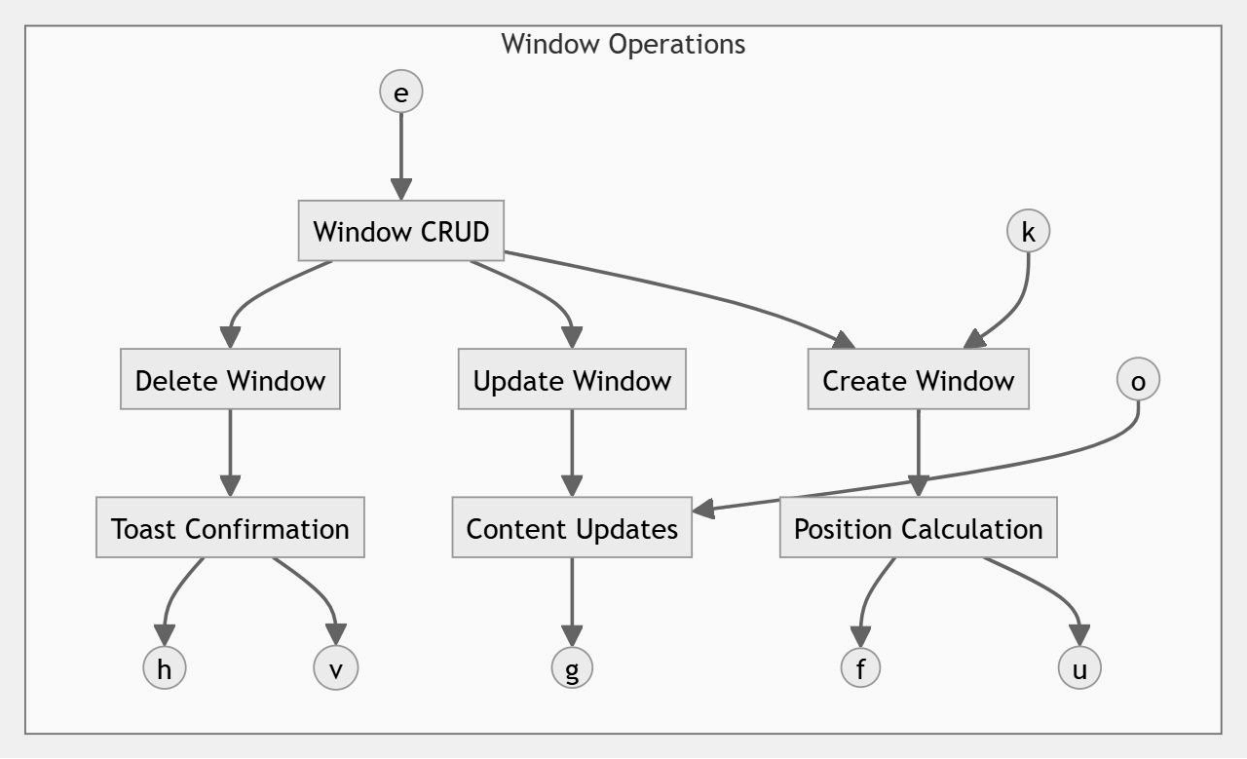
StageManager

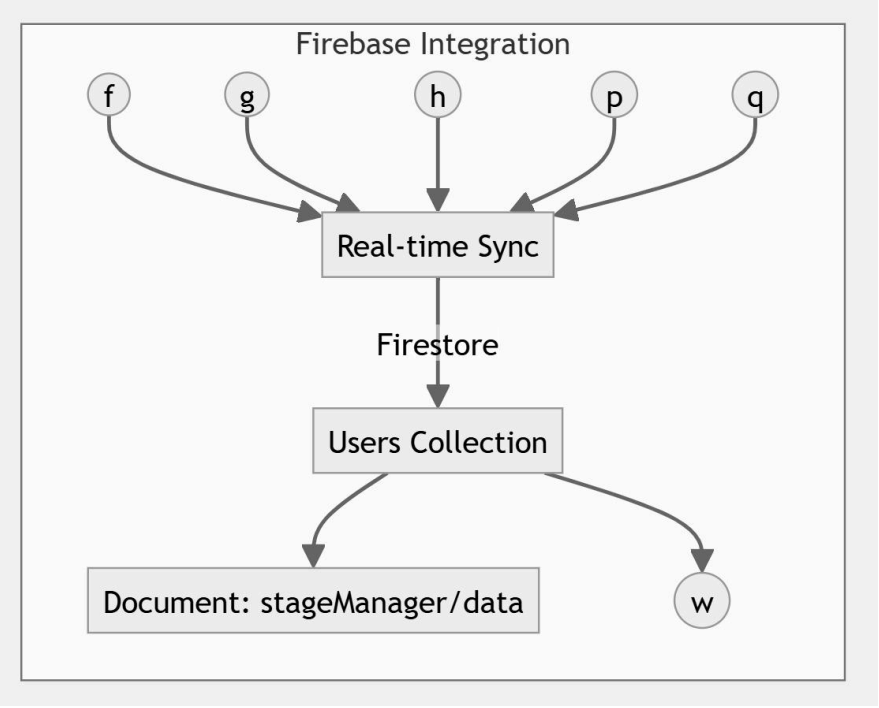


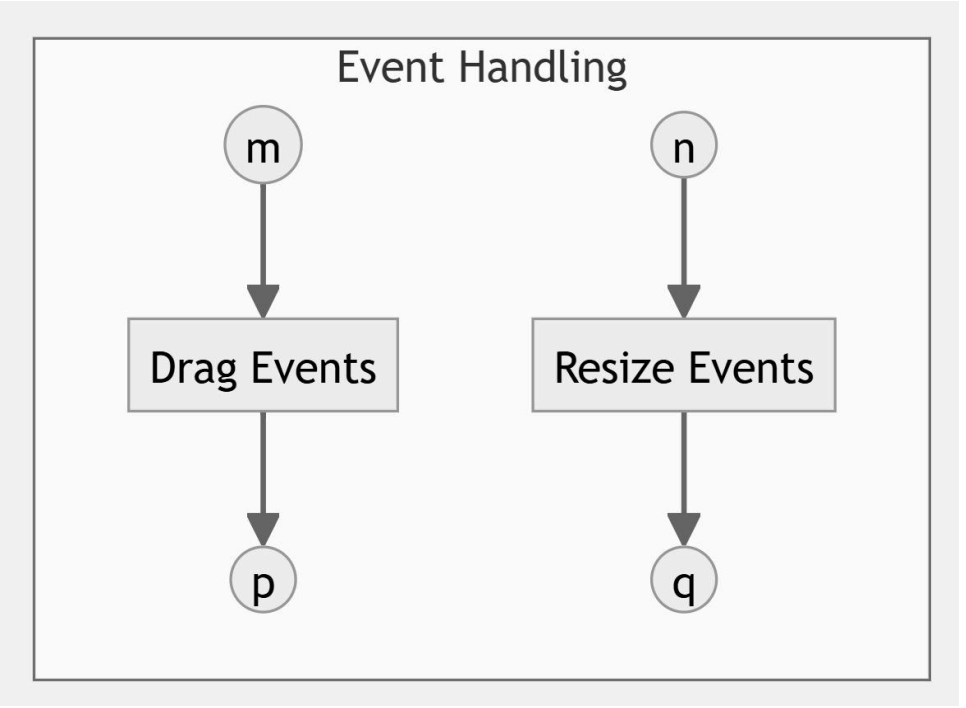


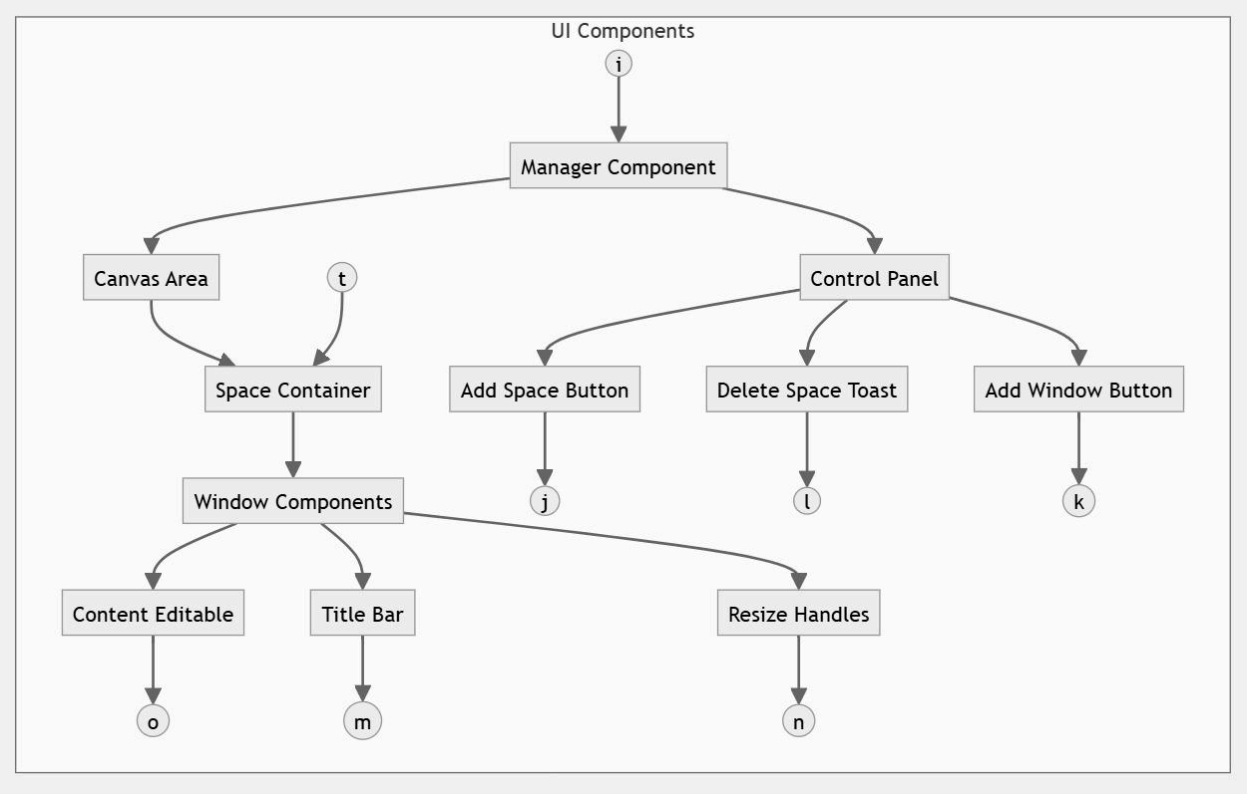



Space Management
d
Space Operations
j
l
Create Space
Switch Space
Delete Space
Add Default Window
Update Current Space
Update Spaces Array
t
s


Window Operations
e
Window CRUD
k
Delete Window
Update Window
Create Window
o
Toast Confirmation
Content Updates
Position Calculation
h
v
g
f
u

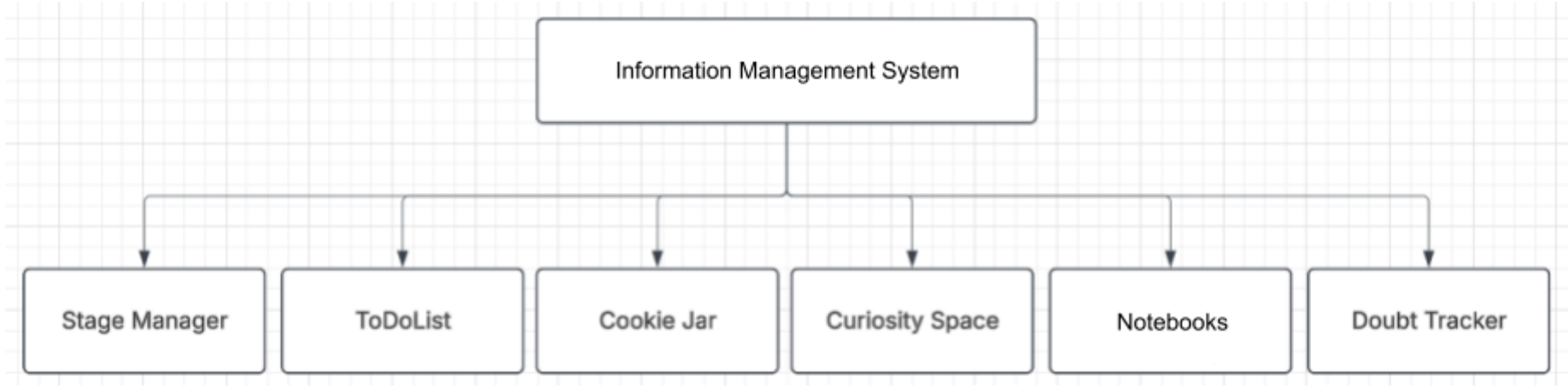

Firebase Integration
f
g
h
p
q
Real-time Sync
Firestore
Users Collection
Document: stageManager/data
w


Event Handling
m
n
Drag Events
Resize Events
p
q

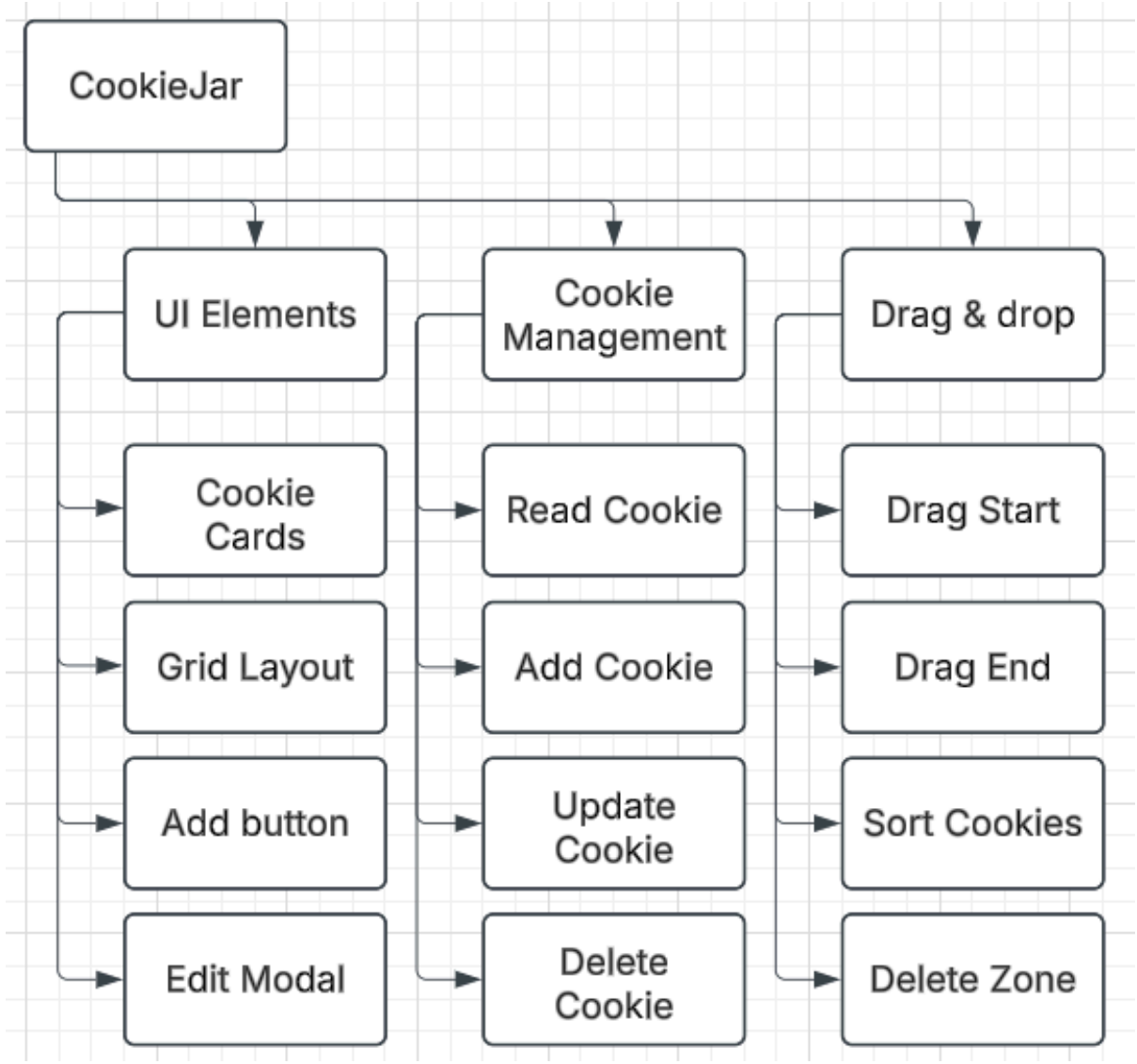

UI Components
i
Manager Component
Canvas Area
t
Control Panel
Space Container
Add Space Button
Delete Space Toast
Add Window Button
Window Components
j
l
k
Content Editable
Title Bar
Resize Handles
o
m
n

Modular Abstraction Diagrams

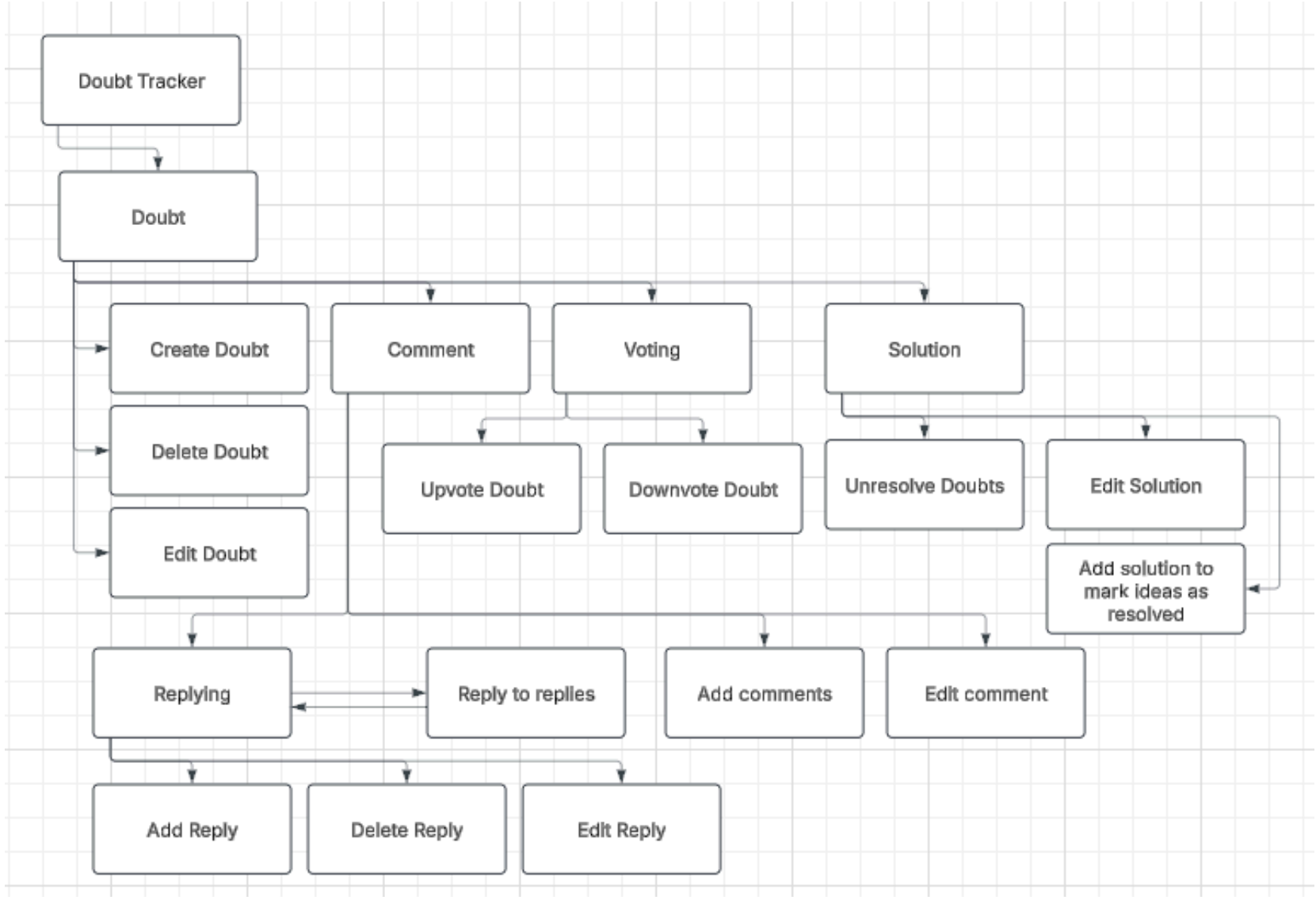
Overview



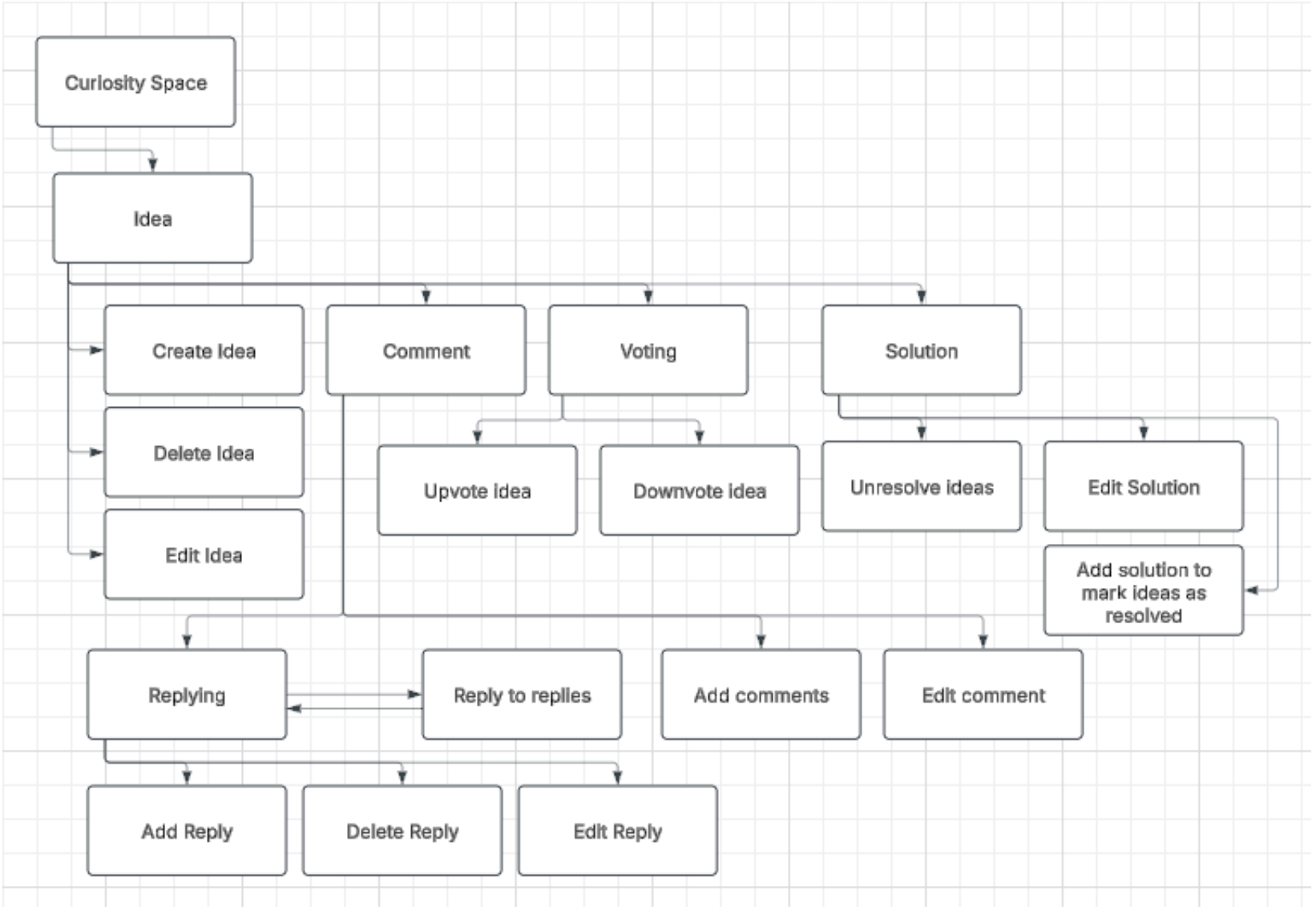
CookieJar


CookieJar
UI Elements
Cookie Management
Drag & drop
Cookie Cards
Read Cookie
Drag Start
Grid Layout
Add Cookie
Drag End
Add button
Update Cookie
Sort Cookies
Edit Modal
Delete Cookie
Delete Zone

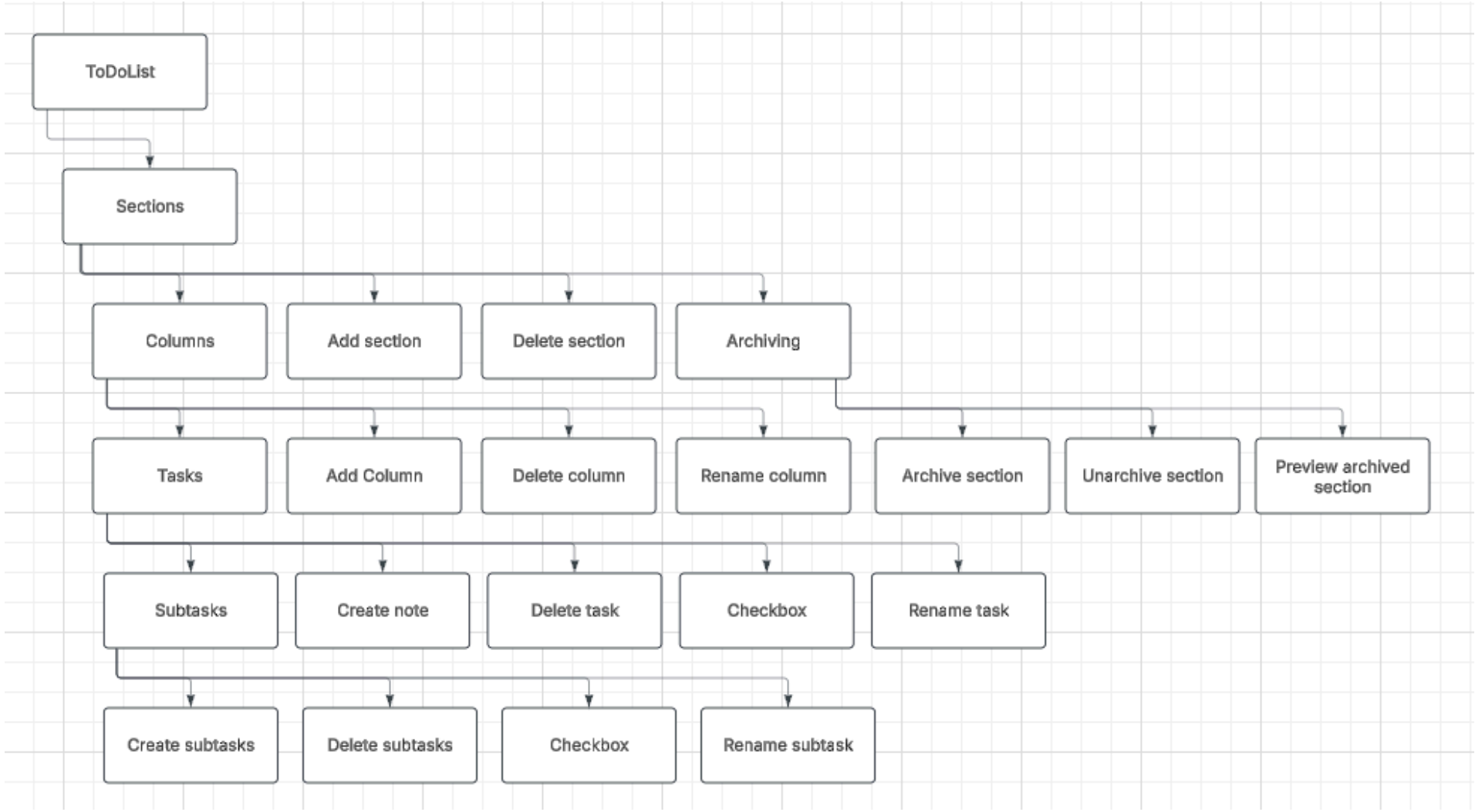
DoubtTracker



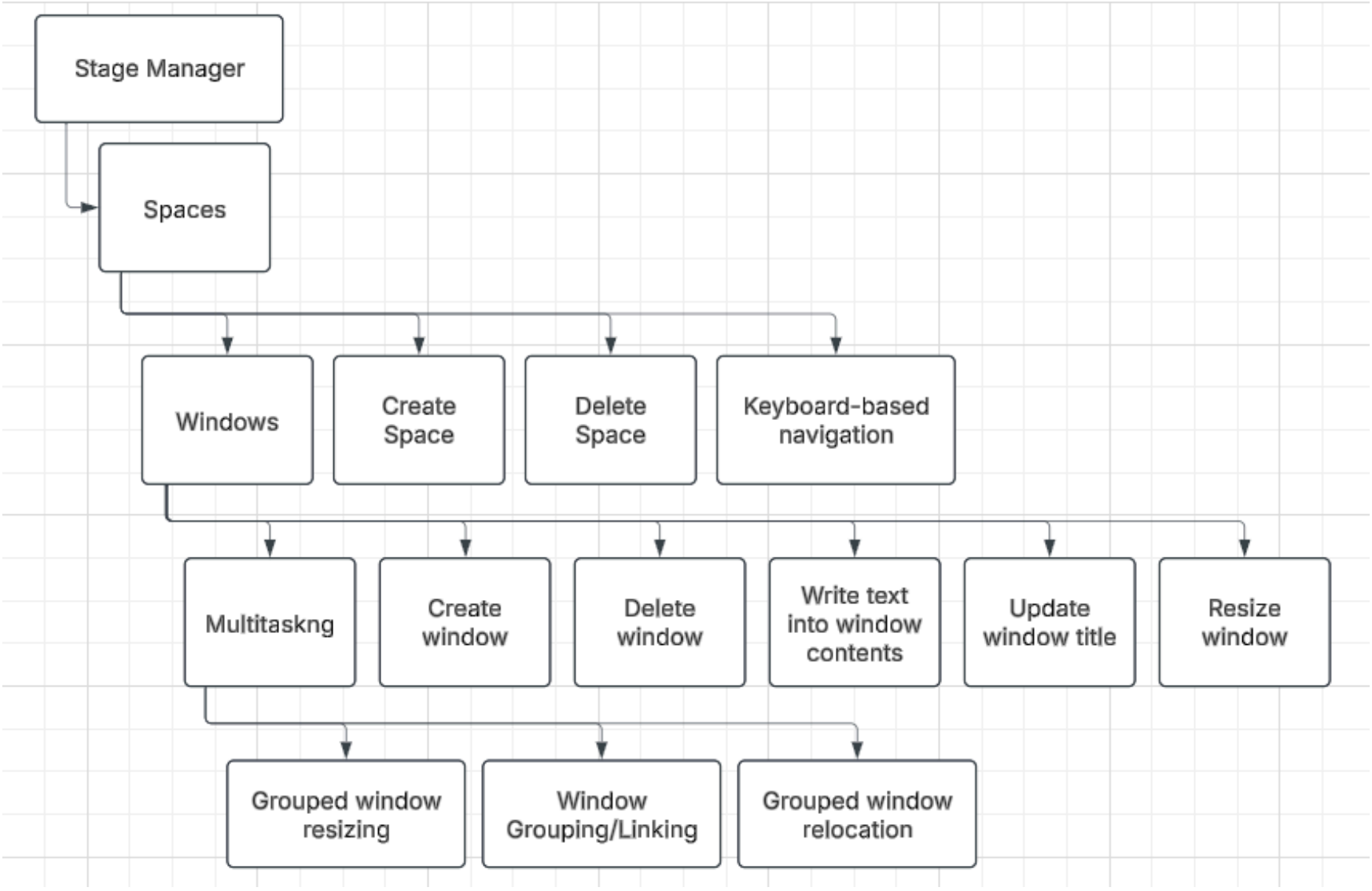
CuriositySpace



ToDoList


ToDoList
Sections
Columns
Add section
Delete section
Archiving
Tasks
Add Column
Delete column
Rename column
Archive section
Unarchive section
Preview archived section
Subtasks
Create note
Delete task
Checkbox
Rename task
Create subtasks
Delete subtasks
Checkbox
Rename subtask

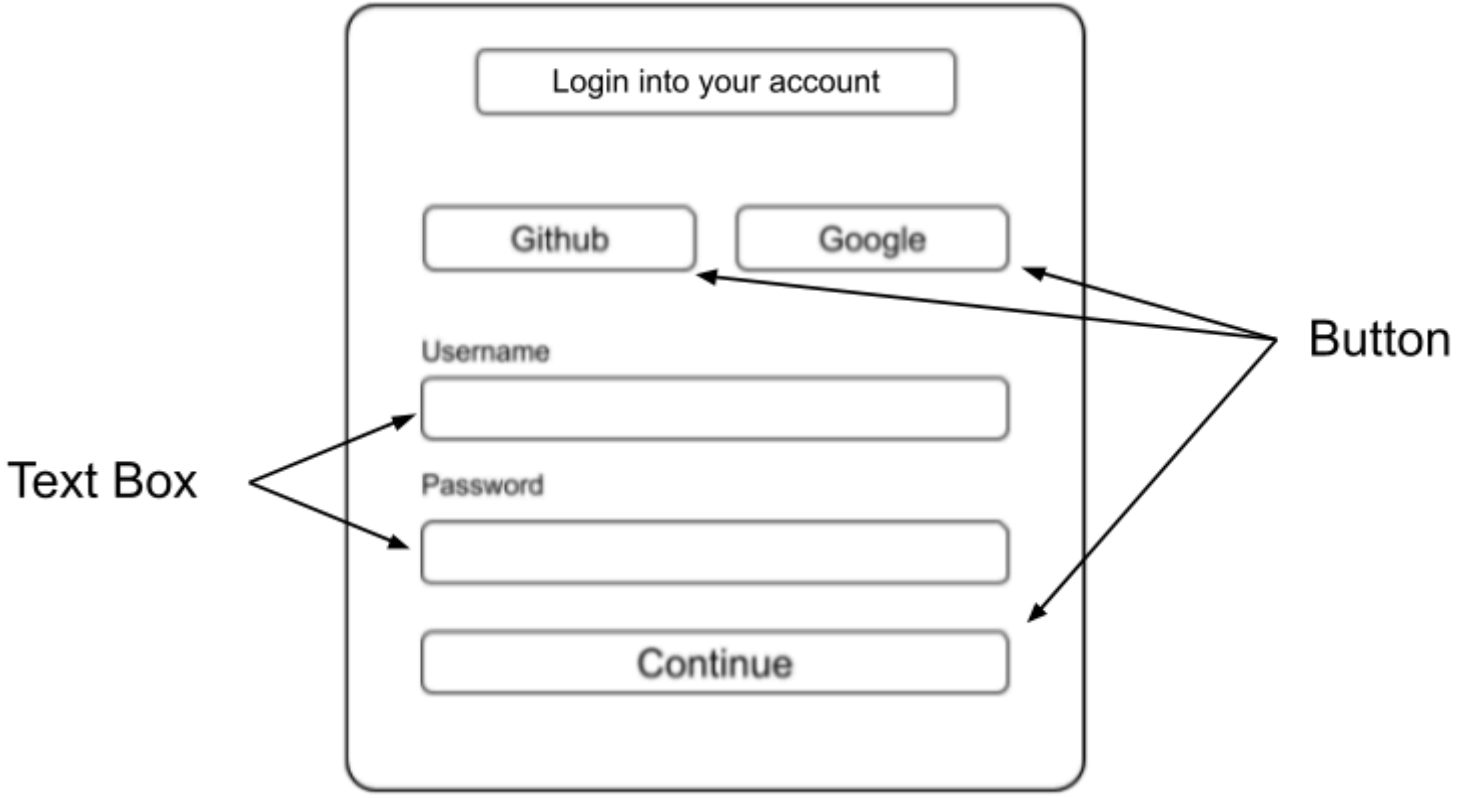
StageManager



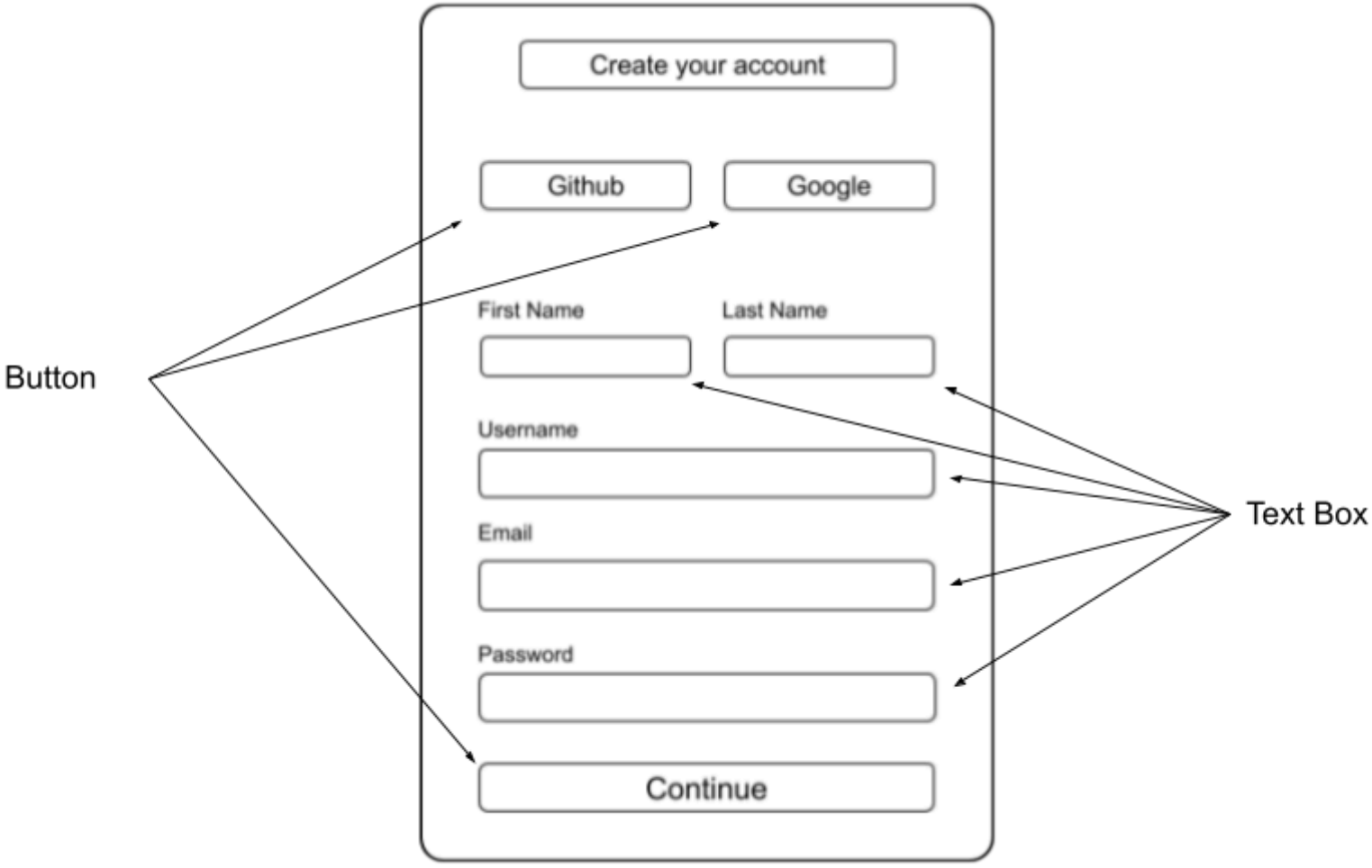
Screen designs

Login and registration:

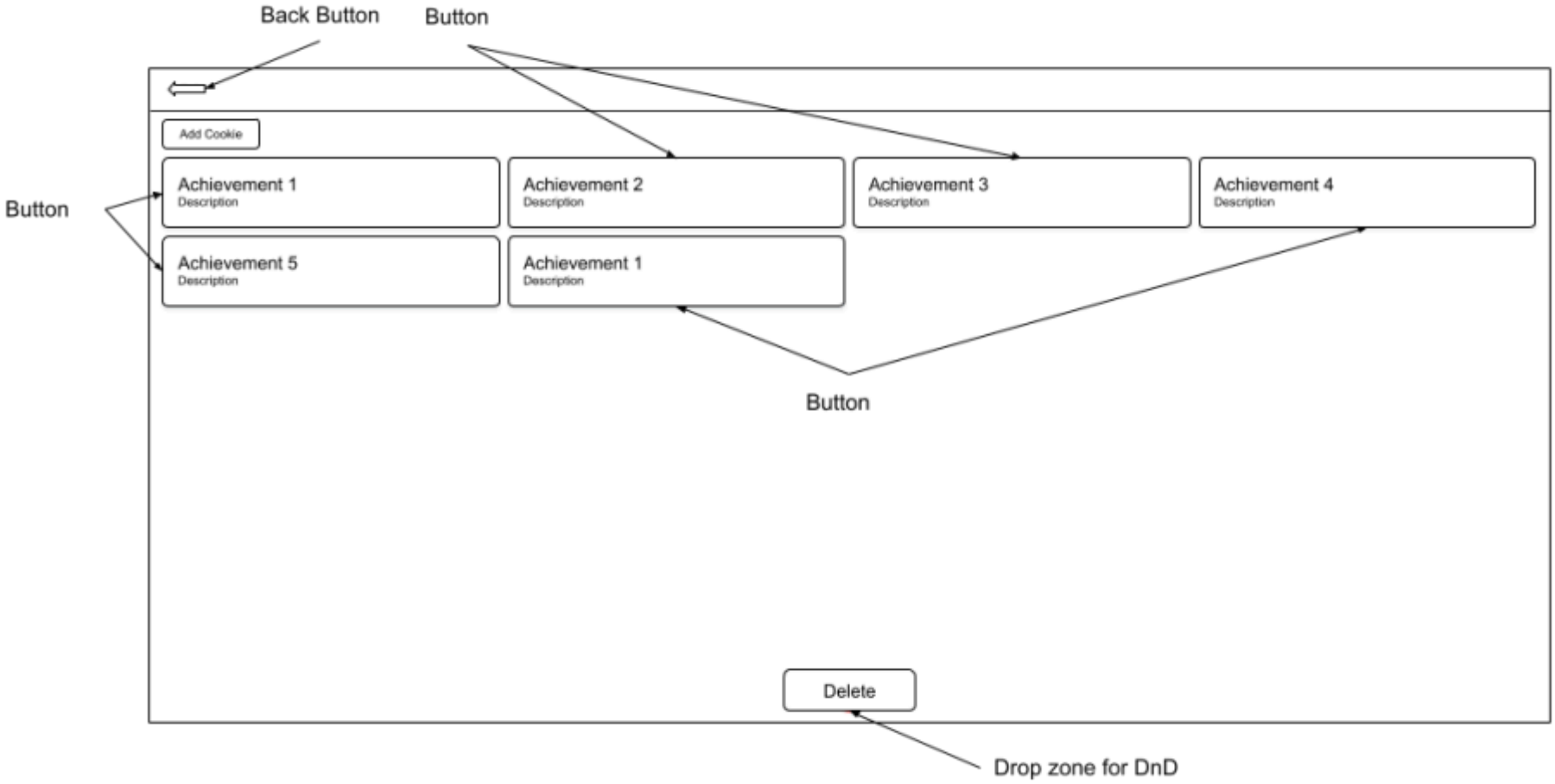
Login



Registration

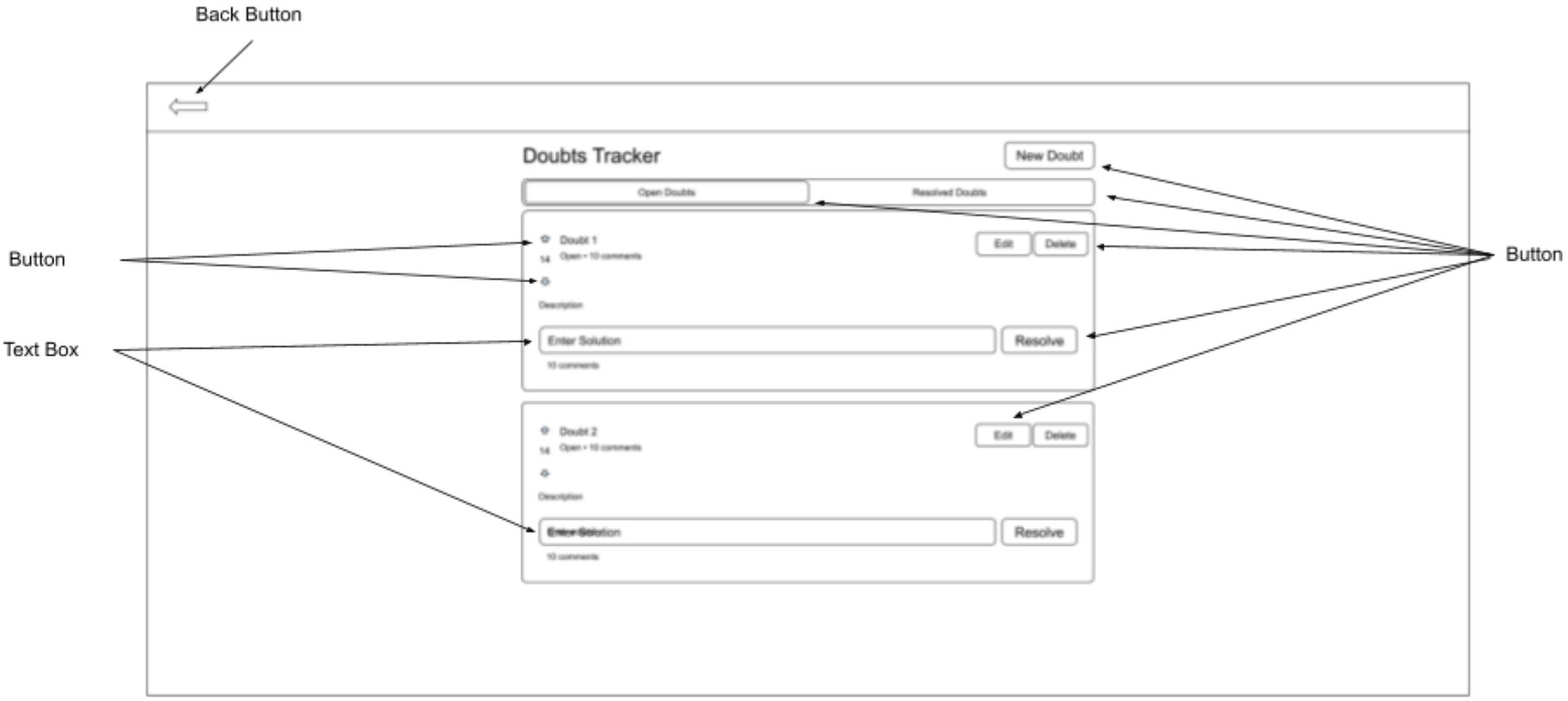


Cookie Jar

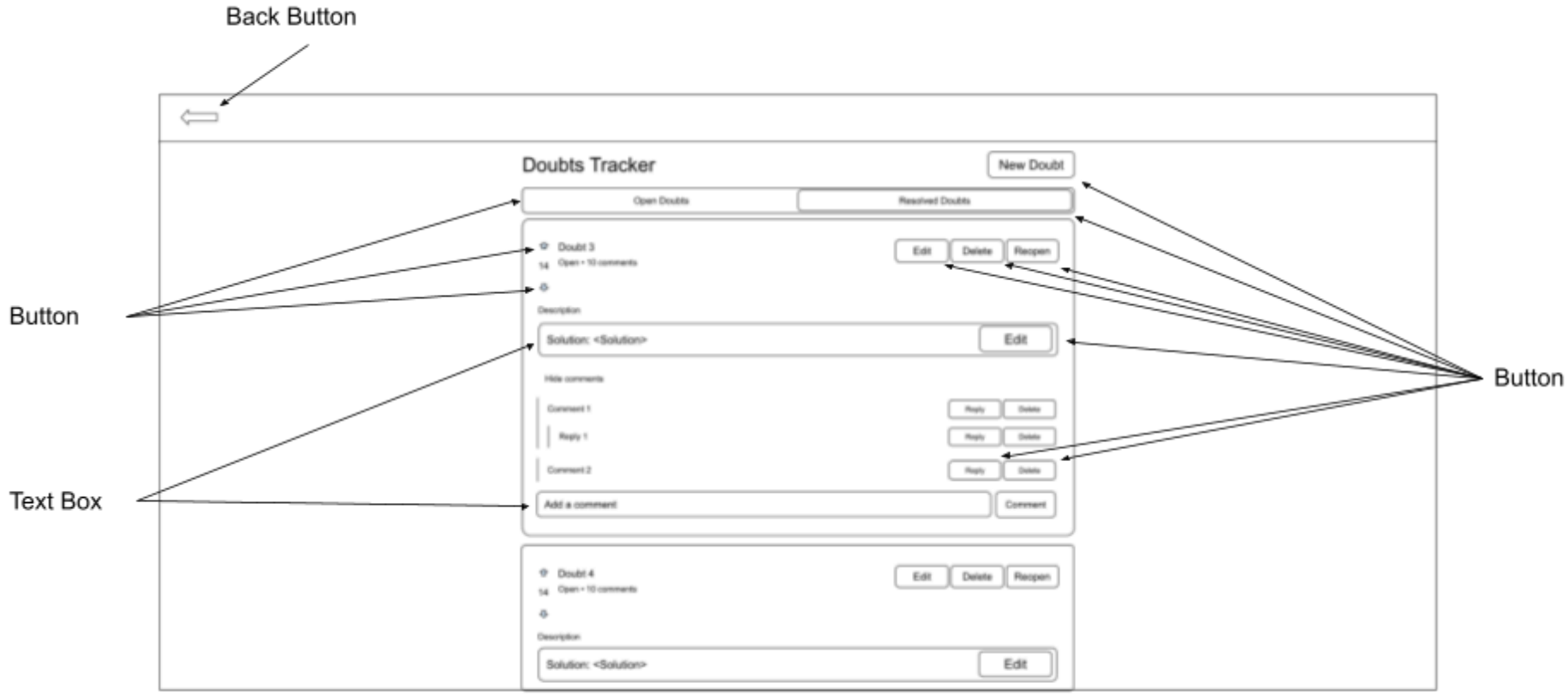

Back Button
Button
Add Cookie
Achievement 1
Description
Achievement 2
Description
Achievement 3
Description
Achievement 4
Description
Button
Achievement 5
Description
Achievement 1
Description
Button
Delete
Drop zone for DnD

DoubtsTracker

DoubtsTracker Open Doubts

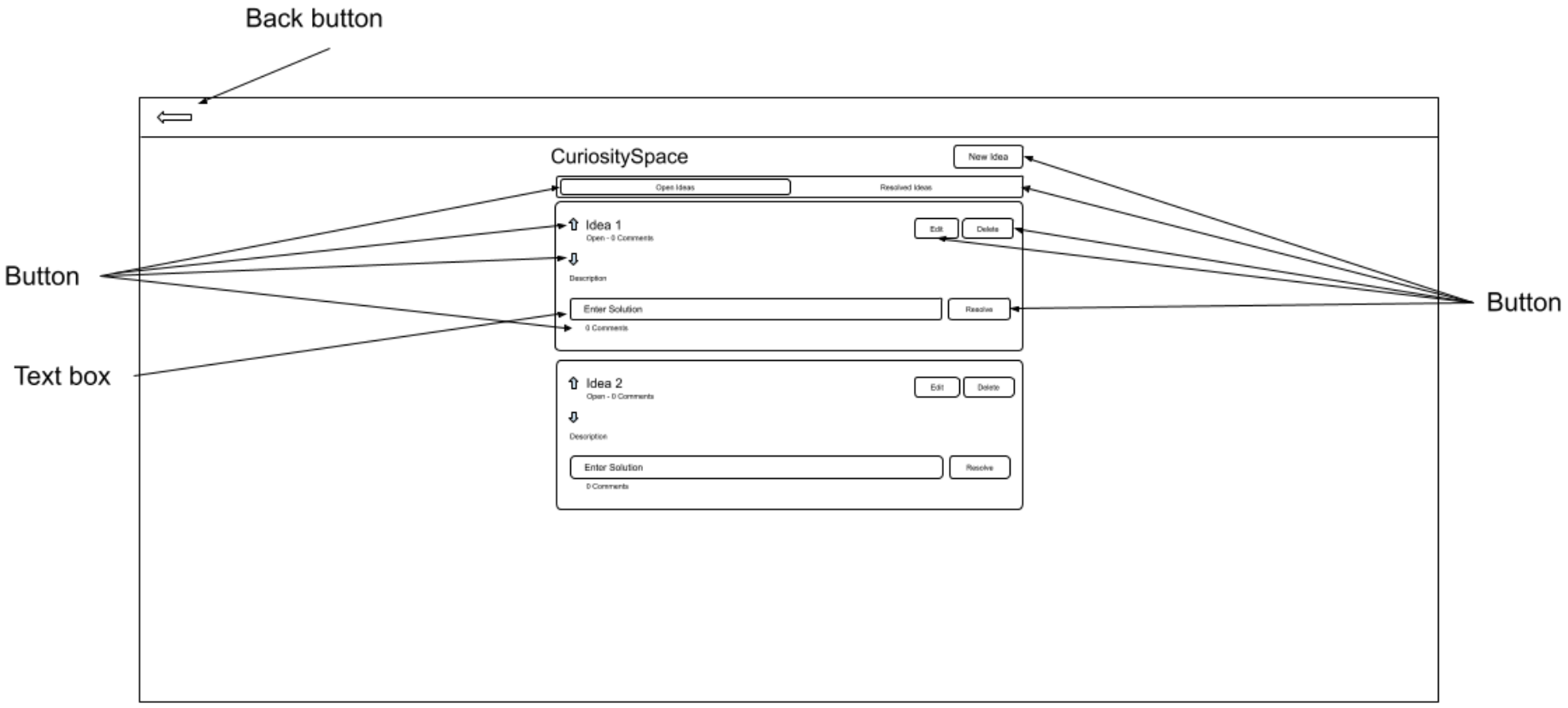


DoubtsTracker Resolved Doubts

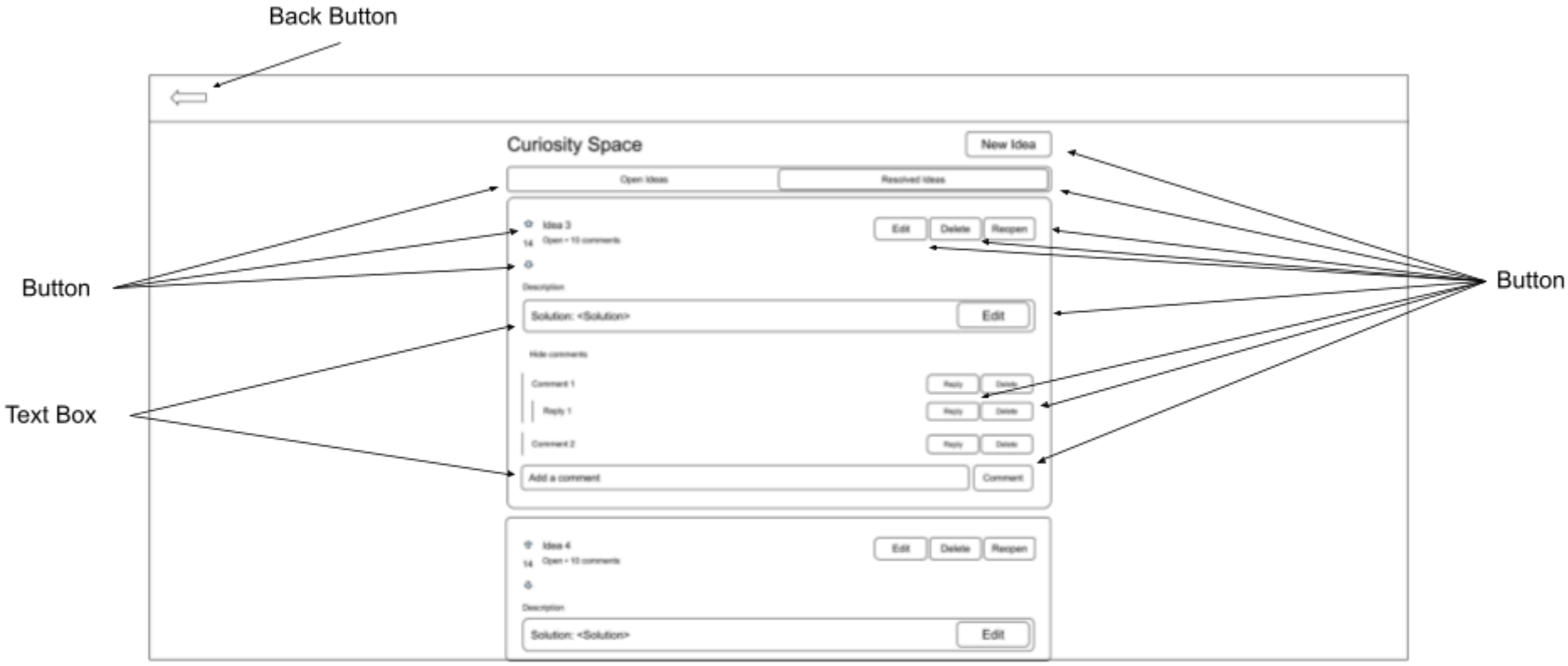


CuriositySpace

CuriositySpace Open Ideas



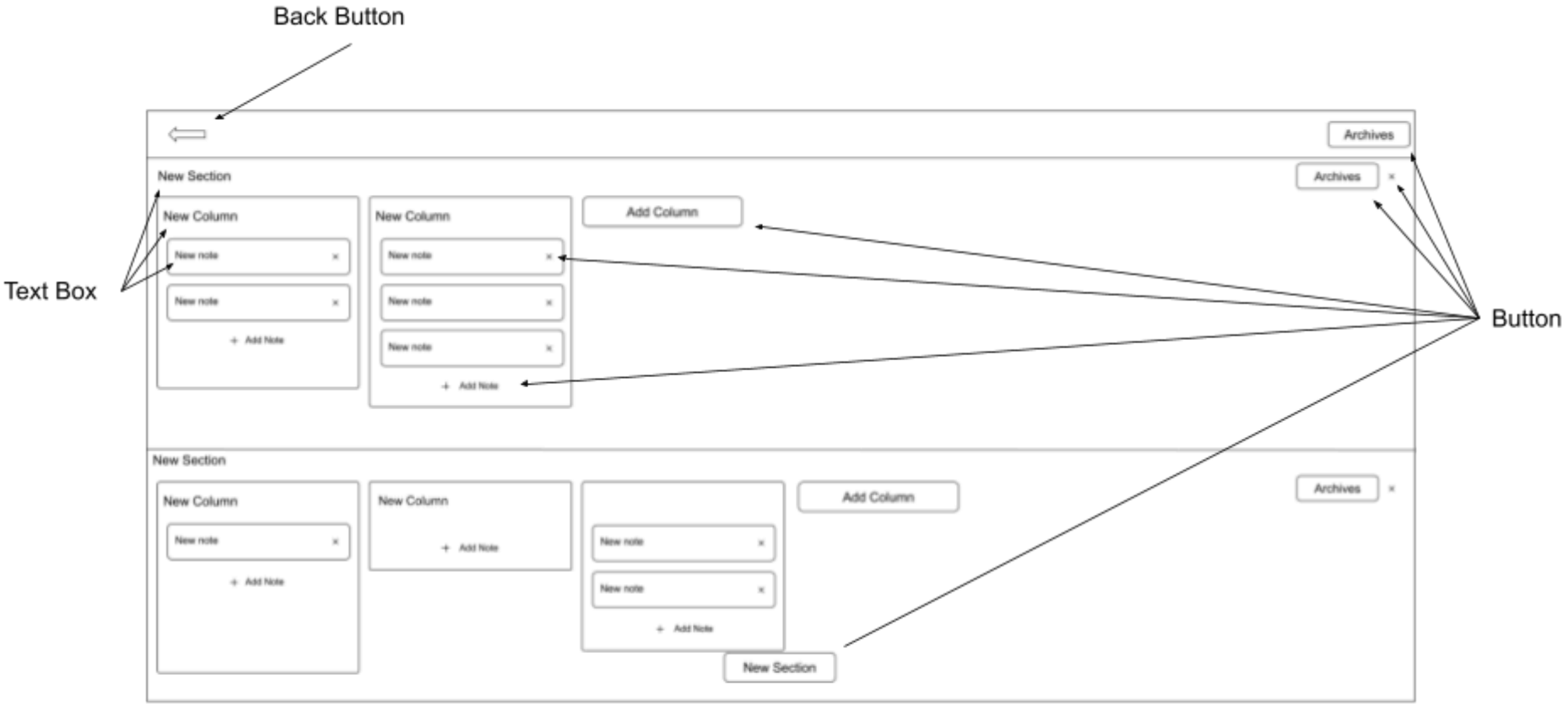
CuriositySpace Resolved Ideas



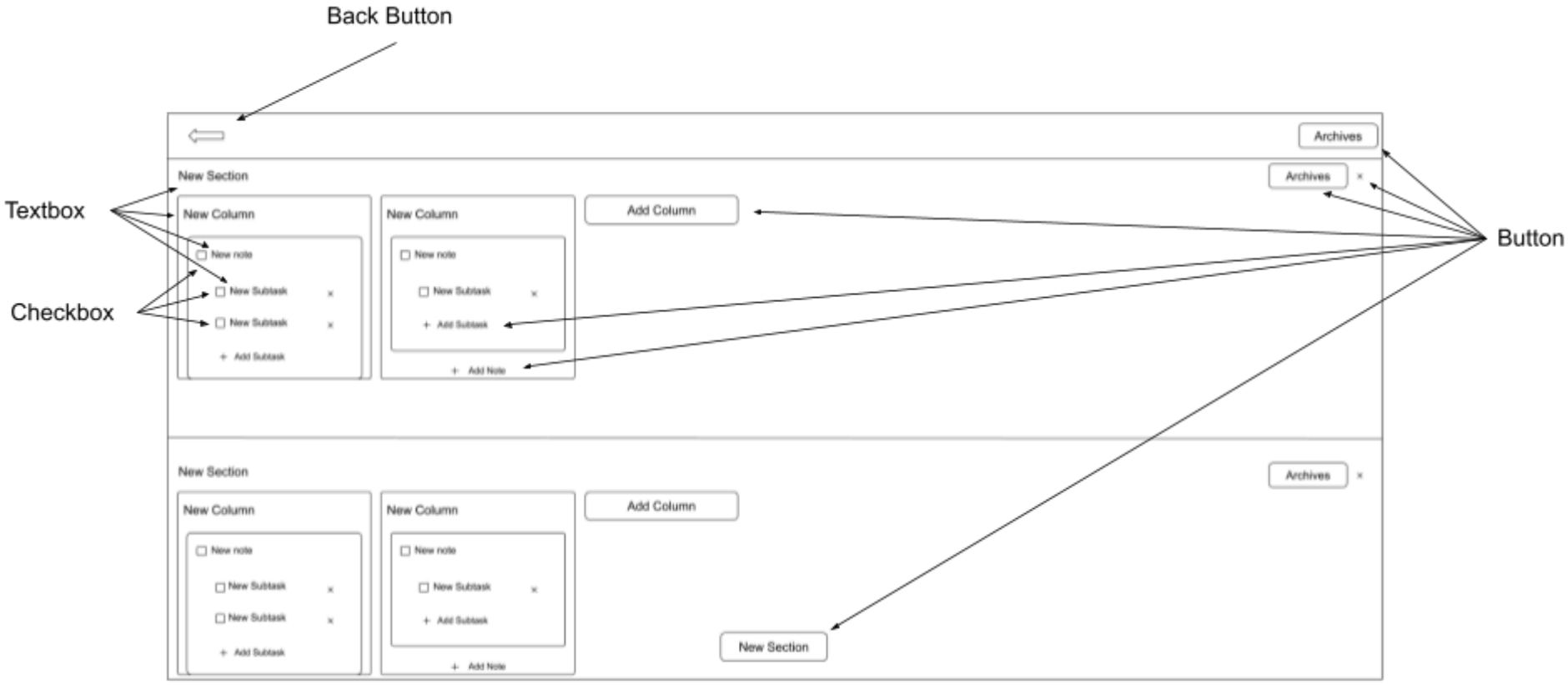
Notebooks


Back button
My Notebooks
New Notebook
Button
Notebook 1
Edit
Notebook 2
Edit
Notebook 3
Edit
Notebook 4
Edit
Notebook 5
Edit
Notebook 6
Edit
Notebook 7
Edit

Notes

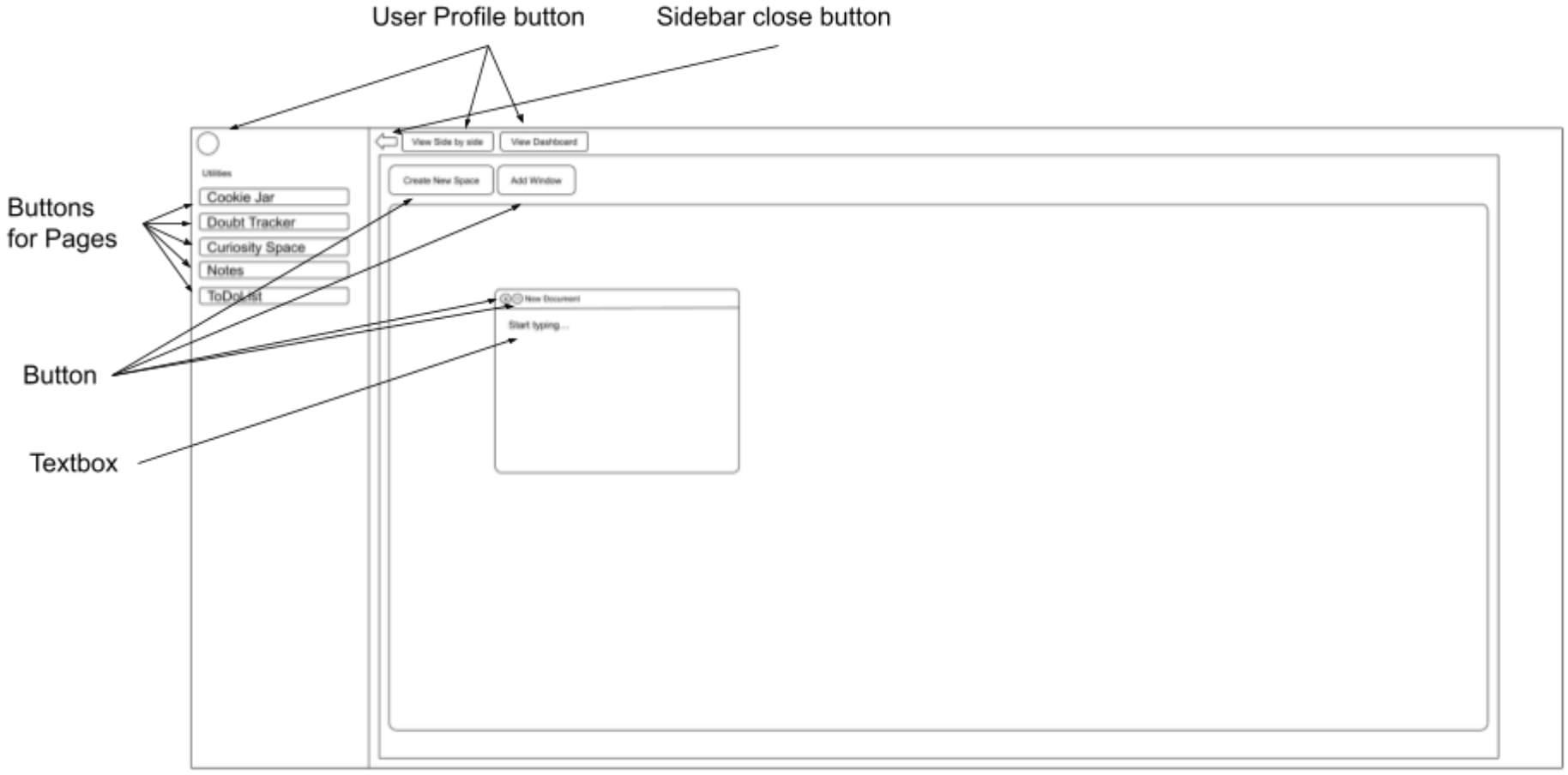

Back Button
Archives
New Section
Archives
New Column
New note
New note
Add Note
New Column
New note
New note
New note
Add Note
Add Column
Text Box
Button
New Section
New Column
New note
Add Note
New Column
Add Note
New note
New note
Add Note
Add Column
Archives
New Section

ToDoList

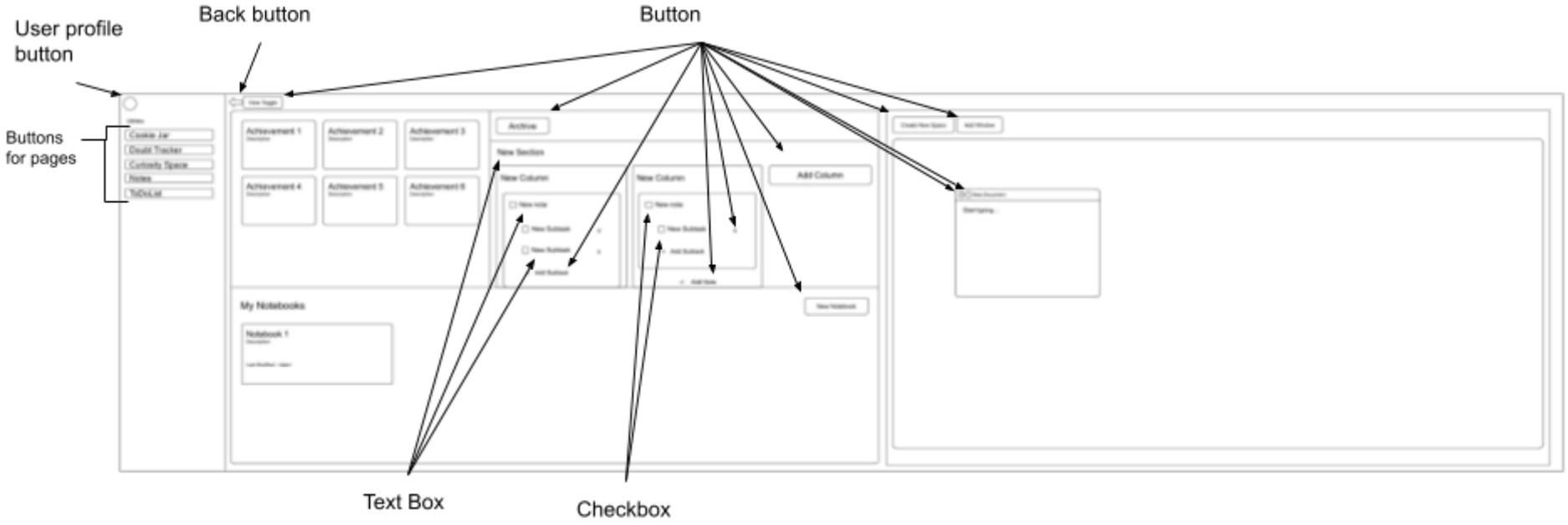

Back Button
Textbox
Checkbox
Button
Archives
New Section
Archives
New Column
New Column
Add Column
New note
New Subtask
New Subtask
Add Subtask
New note
New Subtask
Add Subtask
Add Note
New Section
Archives
New Column
New Column
Add Column
New note
New Subtask
New Subtask
Add Subtask
New note
New Subtask
Add Subtask
Add Note
New Section

Main Page

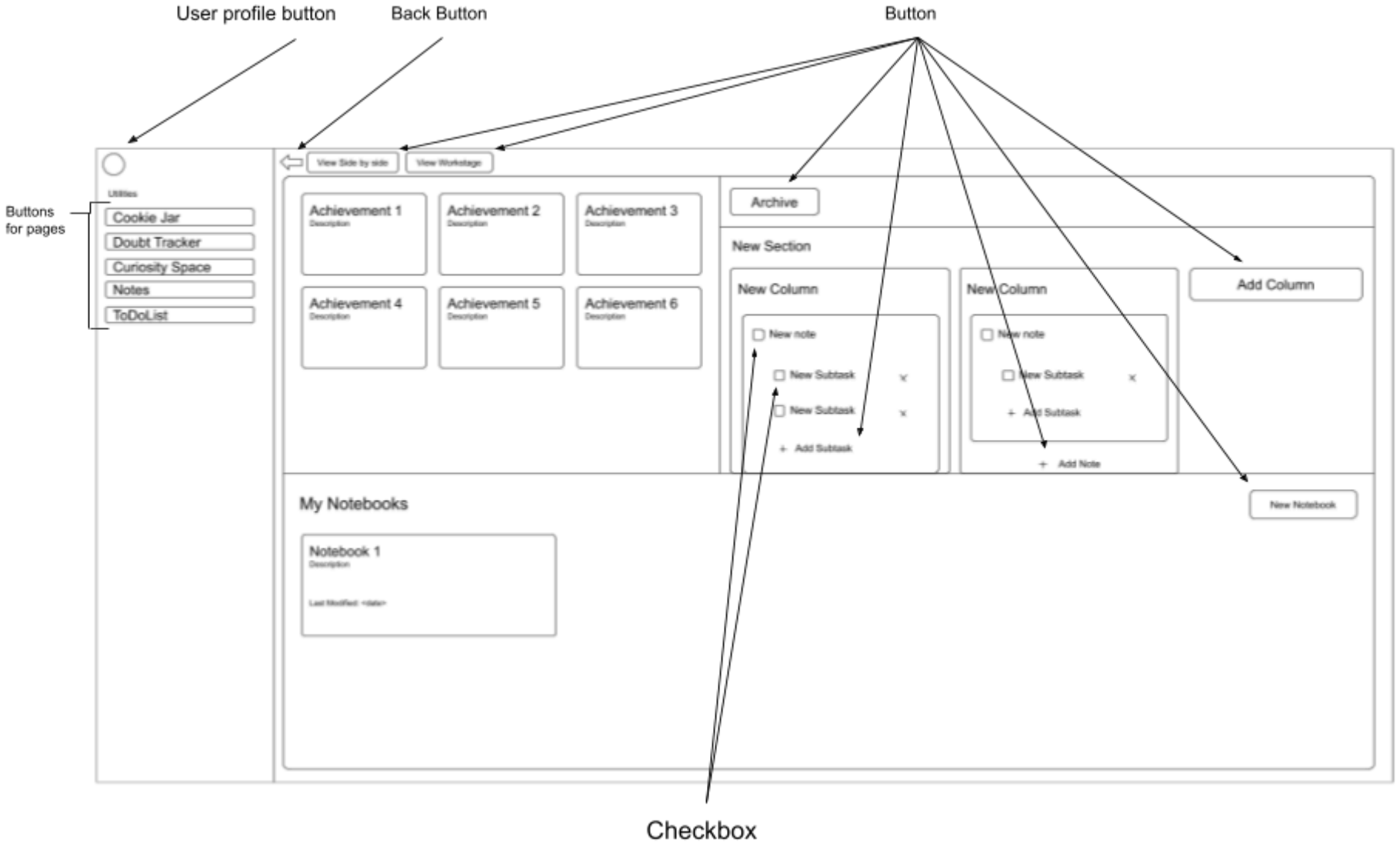
Workstage view main page



Side-by-Side view main page

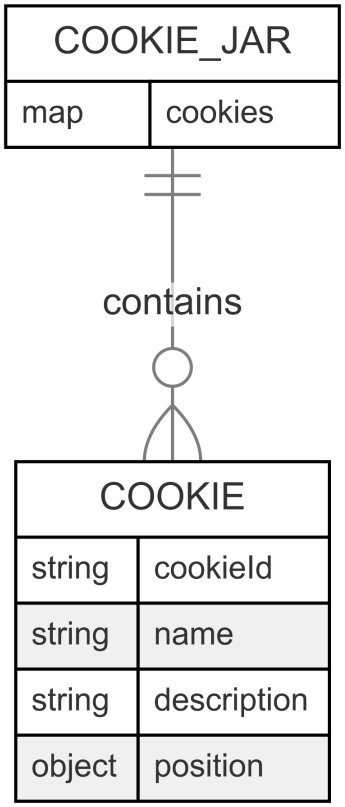


Dashboard view main page

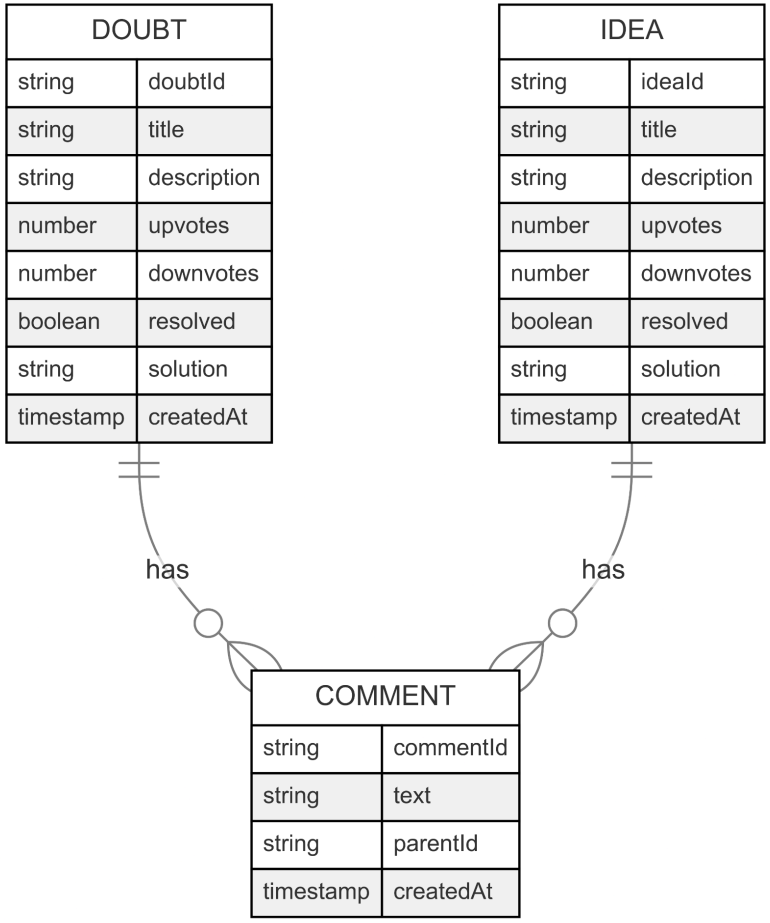


Database (Entity Relationship Diagrams)

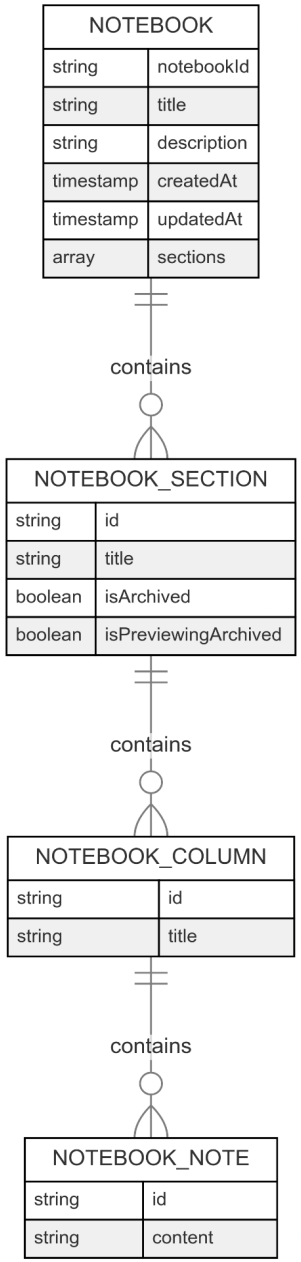
CookieJar



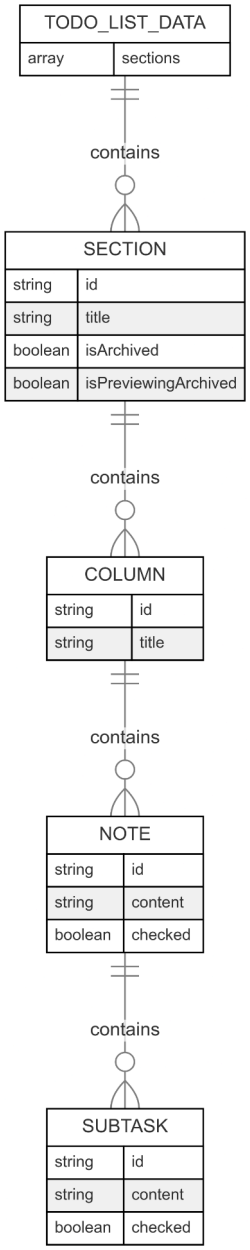
DoubtTracker and CuriositySpace



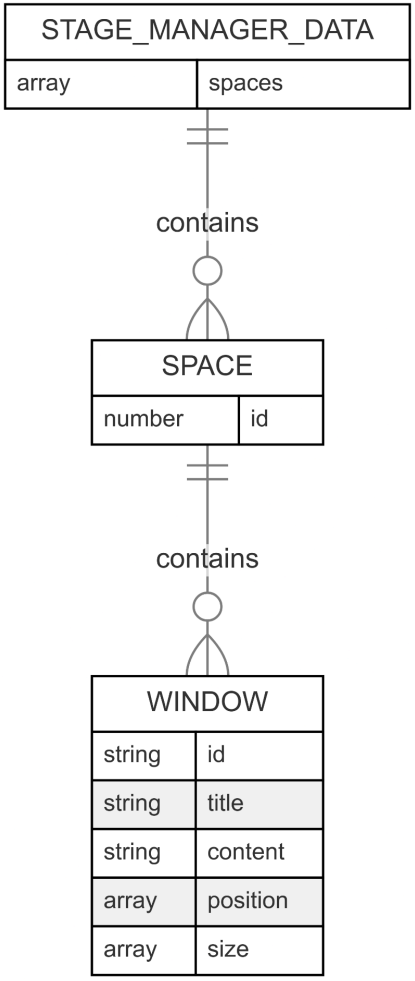
Notebooks


NOTEBOOK
string notebookId
string title
string description
timestamp createdAt
timestamp updatedAt
array sections
contains
NOTEBOOK_SECTION
string id
string title
boolean isArchived
boolean isPreviewingArchived
contains
NOTEBOOK_COLUMN
string id
string title
contains
NOTEBOOK_NOTE
string id
string content

ToDoList


TODO_LIST_DATA
array
sections
contains
SECTION
string
id
string
title
boolean
isArchived
boolean
isPreviewingArchived
contains
COLUMN
string
id
string
title
contains
NOTE
string
id
string
content
boolean
checked
contains
SUBTASK
string
id
string
content
boolean
checked

StageManager



Pseudocode

Authentication Logic

Handling Protected Route Access (Middleware Flow)

```
// Middleware Logic applied to incoming requests
FUNCTION HandleRequest(request):
    // Define routes that do not require authentication
    DEFINE publicRoutes = ["/sign-in", "/sign-up", "/public-info"]

    // Get the requested path from the request
    requestedPath = GET request.path

    // Check if the requested path is in the list of public routes
    isPublic = CHECK_IF requestedPath IS IN publicRoutes

    // If the route is NOT public, authentication is required
    IF NOT isPublic THEN
        // Check if the user has a valid, active session
        isAuthenticated = CHECK_USER_SESSION()

        // If the user is not authenticated, redirect them to the sign-in page
        IF NOT isAuthenticated THEN
            REDIRECT user TO "/sign-in"
            STOP // Prevent further processing of the request
        END IF
    END IF

    // If the route is public OR the user is authenticated for a protected route,
    // allow the request to proceed to the target page/handler.
    PROCEED WITH request
```

```
END FUNCTION
```

Processing User Login/Registration (Auth Service Interaction)

```
// Login Attempt
FUNCTION HandleLogin(username, password):
    // Send credentials to Authentication Service for verification
    response = AUTH_SERVICE.verifyCredentials(username, password)

    IF response.isSuccessful THEN
        // Create a user session (e.g., set a cookie or token)
        CREATE_USER_SESSION(response.userId)
        // Redirect user to the main application dashboard
        REDIRECT user TO "/dashboard"
    ELSE
        // Display an error message to the user (e.g., "Invalid credentials")
        SHOW_ERROR("Invalid username or password.")
    END IF
END FUNCTION

// Registration Attempt
FUNCTION HandleRegistration(email, password, otherDetails):
    // Check if the email is already registered with the Authentication Service
    emailExists = AUTH_SERVICE.checkEmailExists(email)

    IF emailExists THEN
        // Display an error message (e.g., "Email already in use")
        SHOW_ERROR("Email address is already registered.")
    ELSE
        // Attempt to create a new user account via the Authentication Service
        response = AUTH_SERVICE.createUser(email, password, otherDetails)
```

```
        IF response.isSuccessful THEN
            // Automatically log the user in by creating a session
            CREATE_USER_SESSION(response.userId)
            // Redirect user to the main application dashboard or a welcome page
            REDIRECT user TO "/dashboard"
        ELSE
            // Display a generic registration error message
            SHOW_ERROR("Registration failed. Please try again.")
        END IF
    END IF
END FUNCTION
```

Cookie Jar

Loading and Displaying Cookies

```
// When the Cookie Jar module is loaded for a logged-in user
FUNCTION LoadCookies(userId):
    // Define the expected structure for a single cookie
    DEFINE CookieStructure = { id, name, description, position {x, y} }

    // Access the user's specific data storage area
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)

    // Attempt to retrieve the collection of cookies from storage
    storedCookiesData = userStorage.GET_COLLECTION("cookieJar/cookies")

    // Initialize an empty list to hold the processed cookies
    cookieList = CREATE_EMPTY_LIST()

    // Process the retrieved data
```

```
    IF storedCookiesData EXISTS THEN
        FOR EACH cookieId, cookieData IN storedCookiesData:
            // Validate cookieData against CookieStructure (optional but good practice)
            // Create a cookie object including its ID
            cookieObject = CREATE_OBJECT(CookieStructure)
            cookieObject.id = cookieId
            cookieObject.name = cookieData.name
            cookieObject.description = cookieData.description
            cookieObject.position = cookieData.position OR {x: 0, y: 0} // Default position if missing

            ADD cookieObject TO cookieList
        END FOR

        // Sort cookies based on their position (y then x) for consistent display
        SORT cookieList BY position.y THEN position.x
    END IF

    // Display the sorted cookieList in the user interface grid
    DISPLAY cookieList ON UI_GRID()
END FUNCTION
```

Managing Cookies (Add, Update, Delete)

```
// Adding a New Cookie
FUNCTION AddCookie(userId, name, description):
    // Validate input: Ensure name and description are not empty
    IF name IS EMPTY OR description IS EMPTY THEN
        SHOW_ERROR("Name and description cannot be empty.")
        RETURN // Stop processing
    END IF

    // Generate a unique ID for the new cookie
```

```
    newCookieId = GENERATE_UNIQUE_ID()

    // Determine initial position (e.g., next available grid spot or default 0,0)
    // NOTE: Actual position calculation might be complex, simplified here.
    initialPosition = {x: 0, y: 0} // Placeholder

    // Create the new cookie data object
    newCookieData = { name: name, description: description, position: initialPosition }

    // Access user's storage and add/update the cookie data
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    userStorage.SET_ITEM_IN_COLLECTION("cookieJar/cookies", newCookieId, newCookieData)

    // Refresh the UI to show the new cookie
    CALL LoadCookies(userId)
END FUNCTION

// Updating an Existing Cookie
FUNCTION UpdateCookie(userId, cookieId, updatedName, updatedDescription):
    // Validate input: Ensure name and description are not empty
    IF updatedName IS EMPTY OR updatedDescription IS EMPTY THEN
        SHOW_ERROR("Name and description cannot be empty.")
        RETURN
    END IF

    // Access user's storage
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)

    // Get the existing cookie data to preserve its position
    existingCookie = userStorage.GET_ITEM_FROM_COLLECTION("cookieJar/cookies", cookieId)
    IF NOT existingCookie EXISTS THEN
         SHOW_ERROR("Cookie not found.")
         RETURN
```

```
    END IF

    // Create updated cookie data, keeping the original position
    updatedCookieData = { name: updatedName, description: updatedDescription, position: existingCookie.position }

    // Update the cookie data in storage
    userStorage.UPDATE_ITEM_IN_COLLECTION("cookieJar/cookies", cookieId, updatedCookieData)

    // Refresh the UI
    CALL LoadCookies(userId)
END FUNCTION

// Deleting a Cookie
FUNCTION DeleteCookie(userId, cookieId):
    // Access user's storage
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)

    // Remove the cookie data associated with the cookieId
    userStorage.DELETE_ITEM_FROM_COLLECTION("cookieJar/cookies", cookieId)

    // Refresh the UI
    CALL LoadCookies(userId)
END FUNCTION
```

Handling Cookie Reordering (Drag & Drop Logic)

```
// When a drag-and-drop operation finishes
FUNCTION HandleCookieDrop(userId, draggedCookieId, targetPositionIndex):
    // 1. Get the current list of cookies (ideally from the UI state before fetching)
    currentCookieList = GET_CURRENT_DISPLAYED_COOKIES()

    // 2. Find the original index of the dragged cookie
```

```
originalIndex = FIND_INDEX_OF_COOKIE(currentCookieList, draggedCookieId)

// 3. Determine the new index based on the drop target
newIndex = targetPositionIndex // This might represent dropping onto another cookie or a grid slot

// 4. If indices are valid and different, reorder the list
IF originalIndex IS VALID AND newIndex IS VALID AND originalIndex ≠ newIndex THEN
    // Create a new list by moving the dragged cookie
    reorderedList = MOVE_ITEM_IN_LIST(currentCookieList, originalIndex, newIndex)

    // 5. Recalculate positions for all cookies in the reordered list
    // (Assuming a grid layout, e.g., 4 columns)
    DEFINE gridColumns = 4
    updatedCookiesWithPositions = CREATE_EMPTY_LIST()
    FOR index = 0 TO LENGTH(reorderedList) - 1:
        cookie = reorderedList[index]
        cookie.position.x = index MOD gridColumns
        cookie.position.y = FLOOR(index / gridColumns)
        ADD cookie TO updatedCookiesWithPositions
    END FOR

    // 6. Persist the updated positions to storage
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    // Prepare data for batch update (more efficient)
    batchUpdates = {}
    FOR EACH cookie IN updatedCookiesWithPositions:
         // Get only the data needed for storage (excluding ID if ID is the key)
         cookieDataToStore = { name: cookie.name, description: cookie.description, position: cookie.position }
         batchUpdates[cookie.id] = cookieDataToStore
    END FOR
    // Update the entire 'cookies' collection/document part in one go
    userStorage.UPDATE_COLLECTION("cookieJar/cookies", batchUpdates)
```

```
        // 7. Update the UI with the reordered list
        DISPLAY updatedCookiesWithPositions ON UI_GRID()
    END IF
END FUNCTION
```

Doubt Tracker & Curiosity Space Modules (Combined Logic)

Real-time Loading and Displaying Posts/Comments

```
// When the Post module (Doubt Tracker or Curiosity Space) is loaded for a user
FUNCTION LoadPostsRealTime(userId, postType): // postType could be 'doubts' or 'ideas'
    // Define the structure for a Post and Comment
    DEFINE PostStructure = { id, title, description, upvotes, downvotes, resolved, solution, createdAt }
    DEFINE CommentStructure = { id, text, parentId, createdAt }

    // Access the user's specific storage area for the given post type
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postsCollection = userStorage.GET_COLLECTION(postType) // e.g., 'doubts' or 'ideas'

    // Define sorting (e.g., newest first)
    query = QUERY(postsCollection).SORT_BY("createdAt", "DESCENDING")

    // Establish a real-time listener on the query
    LISTEN_FOR_UPDATES(query, (updatedSnapshot) ⇒ {
        postList = CREATE_EMPTY_LIST()

        FOR EACH postDoc IN updatedSnapshot.documents:
            postData = postDoc.data
            // Validate postData (optional)
            postObject = CREATE_OBJECT(PostStructure)
```

```
            postObject.id = postDoc.id
            // Assign data from postData to postObject...
            postObject.title = postData.title
            postObject.description = postData.description
            // ... (assign other fields)

            ADD postObject TO postList
        END FOR

        // Update the UI with the latest list of posts
        DISPLAY postList ON UI()

        // Note: Loading comments typically happens separately when a post is expanded
        // or could be a sub-listener, simplified here.
    })

    // Store the listener reference to unsubscribe later (e.g., on component unmount)
    STORE_LISTENER_REFERENCE()
END FUNCTION

// Logic for loading and structuring comments (can be called when a post is expanded)
FUNCTION LoadCommentsRealTime(userId, postType, postId):
     commentsCollection = GET_STORAGE_AREA_FOR_USER(userId).GET_SUBCOLLECTION(postType, postId, "comments")
     query = QUERY(commentsCollection).SORT_BY("createdAt", "ASCENDING")

     LISTEN_FOR_UPDATES(query, (commentSnapshot) => {
         flatCommentList = CREATE_EMPTY_LIST()
         FOR EACH commentDoc IN commentSnapshot.documents:
             // Create comment object from commentDoc.data...
             ADD commentObject TO flatCommentList
         END FOR

         // Process flatCommentList into a nested structure based on parentId
```

```
        nestedComments = BUILD_NESTED_COMMENTS(flatCommentList)

        // Update the UI for the specific post's comment section
        DISPLAY nestedComments FOR postId ON UI()
    })
    STORE_COMMENT_LISTENER_REFERENCE(postId)
END FUNCTION
```

Managing Posts (Create, Edit, Delete, Resolve/Reopen)

```
// Creating a New Post
FUNCTION CreatePost(userId, postType, title, description):
    IF title IS EMPTY OR description IS EMPTY THEN
        SHOW_ERROR("Title and description are required.")
        RETURN
    END IF

    newPostData = {
        title: title,
        description: description,
        upvotes: 0,
        downvotes: 0,
        resolved: FALSE,
        solution: "",
        createdAt: GET_CURRENT_SERVER_TIMESTAMP()
    }

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postsCollection = userStorage.GET_COLLECTION(postType)
    postsCollection.ADD(newPostData)
    // UI updates handled by the real-time listener
END FUNCTION
```

```
// Editing Post Details
FUNCTION EditPost(userId, postType, postId, newTitle, newDescription):
    IF newTitle IS EMPTY OR newDescription IS EMPTY THEN
        SHOW_ERROR("Title and description are required.")
        RETURN
    END IF

    updates = { title: newTitle, description: newDescription }
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId)
    postRef.UPDATE(updates)
    // UI updates handled by the real-time listener
END FUNCTION

// Deleting a Post
FUNCTION DeletePost(userId, postType, postId):
    // Optional: Add confirmation dialog
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId)
    // Need to also delete subcollections like comments if they exist
    DELETE_SUBCOLLECTION(postRef, "comments") // Simplified representation
    postRef.DELETE()
    // UI updates handled by the real-time listener
END FUNCTION

// Resolving a Post
FUNCTION ResolvePost(userId, postType, postId, solutionText):
    IF solutionText IS EMPTY THEN
        SHOW_ERROR("Solution cannot be empty.")
        RETURN
    END IF
```

```
    updates = { resolved: TRUE, solution: solutionText }
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId)
    postRef.UPDATE(updates)
    // UI updates handled by the real-time listener
END FUNCTION

// Reopening a Post
FUNCTION ReopenPost(userId, postType, postId):
    updates = { resolved: FALSE, solution: "" } // Clear solution on reopen
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId)
    postRef.UPDATE(updates)
    // UI updates handled by the real-time listener
END FUNCTION
```

Handling Votes and Comments

```
// Handling Votes
FUNCTION UpdateVote(userId, postType, postId, voteType): // voteType is 'upvote' or 'downvote'
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    postRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId)

    IF voteType == 'upvote' THEN
        // Use atomic increment operation
        postRef.UPDATE({ upvotes: INCREMENT(1) })
    ELSE IF voteType == 'downvote' THEN
        postRef.UPDATE({ downvotes: INCREMENT(1) })
    END IF
    // UI updates handled by the real-time listener
END FUNCTION
```

```
// Adding a Comment/Reply
FUNCTION AddComment(userId, postType, postId, commentText, parentCommentId = NULL):
    IF commentText IS EMPTY THEN
        SHOW_ERROR("Comment cannot be empty.")
        RETURN
    END IF

    newCommentData = {
        text: commentText,
        parentId: parentCommentId, // Store parent ID for replies
        createdAt: GET_CURRENT_SERVER_TIMESTAMP()
    }

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    commentsCollection = userStorage.GET_SUBCOLLECTION(postType, postId, "comments")
    commentsCollection.ADD(newCommentData)
    // UI updates handled by the comment real-time listener
END FUNCTION

// Editing a Comment
FUNCTION EditComment(userId, postType, postId, commentId, newText):
    IF newText IS EMPTY THEN
        SHOW_ERROR("Comment cannot be empty.")
        RETURN
    END IF

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    commentRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId, "comments", commentId)
    commentRef.UPDATE({ text: newText })
    // UI updates handled by the comment real-time listener
END FUNCTION

// Deleting a Comment
```

```
FUNCTION DeleteComment(userId, postType, postId, commentId):
    // Need to handle deleting replies recursively if deleting a parent comment
    // Simplified here:
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    commentRef = userStorage.GET_DOCUMENT_REFERENCE(postType, postId, "comments", commentId)
    // Add logic here to find and delete replies first if necessary
    commentRef.DELETE()
    // UI updates handled by the comment real-time listener
END FUNCTION
```

Notebooks

Notebook Lifecycle (Create, Load List, Delete)

```
// Loading the List of Notebooks (e.g., for the Notebook Manager view)
FUNCTION LoadNotebookList(userId):
    DEFINE NotebookMetaStructure = { id, title, description, updatedAt }

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    notebooksCollection = userStorage.GET_COLLECTION("notebooks")

    // Retrieve all documents from the notebooks collection
    notebookDocs = notebooksCollection.GET_ALL_DOCUMENTS()

    notebookList = CREATE_EMPTY_LIST()
    FOR EACH doc IN notebookDocs:
        data = doc.data
        // Basic validation
        IF data.title EXISTS THEN
            notebookMeta = CREATE_OBJECT(NotebookMetaStructure)
            notebookMeta.id = doc.id
```

```
            notebookMeta.title = data.title
            notebookMeta.description = data.description OR ""
            notebookMeta.updatedAt = data.updatedAt // Assuming timestamp exists
            ADD notebookMeta TO notebookList
        END IF
    END FOR

    // Sort notebooks, e.g., by last updated time
    SORT notebookList BY updatedAt DESCENDING

    // Display the list of notebooks in the UI
    DISPLAY notebookList ON UI()
END FUNCTION

// Creating a New Notebook
FUNCTION CreateNotebook(userId, title, description = ""):
    IF title IS EMPTY THEN
        SHOW_ERROR("Notebook title cannot be empty.")
        RETURN NULL // Indicate failure
    END IF

    newNotebookData = {
        title: title,
        description: description,
        userId: userId,
        sections: [], // Initialize with empty sections array
        createdAt: GET_CURRENT_SERVER_TIMESTAMP(),
        updatedAt: GET_CURRENT_SERVER_TIMESTAMP()
    }

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    notebooksCollection = userStorage.GET_COLLECTION("notebooks")
```

```
    // Add the new notebook document, letting the storage generate the ID
    newNotebookRef = notebooksCollection.ADD(newNotebookData)

    // Return the ID of the newly created notebook
    RETURN newNotebookRef.id
END FUNCTION

// Deleting a Notebook
FUNCTION DeleteNotebook(userId, notebookId):
    // Optional: Add confirmation dialog

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    notebookRef = userStorage.GET_DOCUMENT_REFERENCE("notebooks", notebookId)

    // Delete the notebook document
    notebookRef.DELETE()

    // Refresh the notebook list in the UI
    CALL LoadNotebookList(userId)
END FUNCTION
```

Notebook Content Management (Loading/Saving Sections, Columns, Notes)

```
// Loading Content of a Specific Notebook (e.g., when opening it)
FUNCTION LoadNotebookContent(userId, notebookId):
    DEFINE SectionStructure = { id, title, columns }
    DEFINE ColumnStructure = { id, title, notes }
    DEFINE NoteStructure = { id, content }

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    notebookRef = userStorage.GET_DOCUMENT_REFERENCE("notebooks", notebookId)
```

```
    notebookDoc = notebookRef.GET_DOCUMENT()

    IF notebookDoc EXISTS THEN
        notebookData = notebookDoc.data
        // Store the loaded sections in the application's state
        // This assumes 'sections' is an array directly within the notebook document
        // Perform necessary validation/structuring
        loadedSections = VALIDATE_AND_STRUCTURE(notebookData.sections, SectionStructure, ColumnStructure, NoteStructure)
        SET_APPLICATION_STATE("currentNotebookSections", loadedSections)

        // Display the loaded content in the notebook UI
        DISPLAY loadedSections ON NOTEBOOK_UI()
    ELSE
        SHOW_ERROR("Notebook not found.")
        // Potentially redirect user back
    END IF
END FUNCTION

// Saving Notebook Content (Called by Debounced Sync)
FUNCTION SaveNotebookContent(userId, notebookId, currentSectionsState):
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    notebookRef = userStorage.GET_DOCUMENT_REFERENCE("notebooks", notebookId)

    // Prepare data for update, only including sections and the updated timestamp
    updateData = {
        sections: currentSectionsState, // The current state from the UI/application
        updatedAt: GET_CURRENT_SERVER_TIMESTAMP()
    }

    // Update the notebook document using MERGE to avoid overwriting other fields
    notebookRef.UPDATE(updateData, { merge: TRUE })
END FUNCTION
```

```
// Adding a New Section to the Current Notebook
FUNCTION AddSection(currentSectionsState):
    newSection = {
        id: GENERATE_UNIQUE_ID(),
        title: "New Section",
        columns: [ { id: GENERATE_UNIQUE_ID(), title: "New Column", notes: [] } ] // Start with one column
    }
    newState = ADD newSection TO currentSectionsState
    SET_APPLICATION_STATE("currentNotebookSections", newState)
    // Saving happens via debounced sync
END FUNCTION

// Adding a New Column to a Section
FUNCTION AddColumn(currentSectionsState, targetSectionId):
    newState = MAP currentSectionsState WHERE section.id == targetSectionId:
        ADD { id: GENERATE_UNIQUE_ID(), title: "New Column", notes: [] } TO section.columns
    END MAP
    SET_APPLICATION_STATE("currentNotebookSections", newState)
    // Saving happens via debounced sync
END FUNCTION

// Adding a New Note to a Column
FUNCTION AddNote(currentSectionsState, targetSectionId, targetColumnId):
     newState = MAP currentSectionsState WHERE section.id == targetSectionId:
         MAP section.columns WHERE column.id == targetColumnId:
             ADD { id: GENERATE_UNIQUE_ID(), content: "New Note" } TO column.notes
         END MAP
     END MAP
     SET_APPLICATION_STATE("currentNotebookSections", newState)
     // Saving happens via debounced sync
END FUNCTION

// Updating Note Content (e.g., after inline editing)
```

```
FUNCTION UpdateNoteContent(currentSectionsState, sectionId, columnId, noteId, newContent):
    newState = MAP currentSectionsState WHERE section.id == sectionId:
        MAP section.columns WHERE column.id == columnId:
            MAP column.notes WHERE note.id == noteId:
                note.content = newContent
            END MAP
        END MAP
    END MAP
    SET_APPLICATION_STATE("currentNotebookSections", newState)
    // Saving happens via debounced sync
END FUNCTION

// Deleting Sections/Columns/Notes (Similar pattern, update state, sync saves)
FUNCTION DeleteItem(currentSectionsState, type, sectionId, columnId = NULL, noteId = NULL):
    // Logic to find and remove the item (section, column, or note) based on type and IDs
    // ... find the item and remove it from the nested structure ...
    newState = REMOVE_ITEM(currentSectionsState, type, sectionId, columnId, noteId)
    SET_APPLICATION_STATE("currentNotebookSections", newState)
    // Saving happens via debounced sync
    // Optional: Show confirmation/undo toast
END FUNCTION
```

Handling Note Reordering/Movement (Drag & Drop)

```
// When a drag-and-drop operation finishes for a Note
FUNCTION HandleNoteDrop(currentSectionsState, draggedNoteId, sourceLocation, targetLocation):
    // sourceLocation = { sectionId, columnId }
    // targetLocation = { sectionId, columnId, targetNoteId (optional, for position) }

    // 1. Find and remove the note from the source location in a temporary state
    tempState = REMOVE_NOTE(currentSectionsState, draggedNoteId, sourceLocation)
    IF noteNotFound THEN RETURN // Safety check
```

```
    // 2. Find the target index in the target column
    targetIndex = DETERMINE_TARGET_INDEX(tempState, targetLocation)

    // 3. Insert the dragged note into the target location at the target index
    newState = INSERT_NOTE_AT_INDEX(tempState, draggedNote, targetLocation, targetIndex)

    // 4. Update the application state
    SET_APPLICATION_STATE("currentNotebookSections", newState)
    // Saving happens via debounced sync
END FUNCTION
```

Debounced Data Synchronization

```
// Global or Module-level variables
DEFINE debounceTimer = NULL
DEFINE debounceDelay = 1000 // milliseconds (e.g., 1 second)

// Function called whenever the notebook content state changes
FUNCTION OnContentChange(userId, notebookId, currentSectionsState):
    // Clear any existing timer to reset the debounce period
    CLEAR_TIMER(debounceTimer)

    // Set a new timer
    debounceTimer = SET_TIMER(() ⇒ {
        CALL SaveNotebookContent(userId, notebookId, currentSectionsState)
    }, debounceDelay)
END FUNCTION

// NOTE: Every function that modifies 'currentSectionsState' (AddSection, AddColumn, AddNote, UpdateNoteContent, DeleteItem, HandleNoteDrop,
etc.)
// should call OnContentChange AFTER updating the state.
```

ToDo List

Loading and Saving ToDo List Structure

```
// When the ToDo List module is loaded for a user
FUNCTION LoadToDoList(userId):
    // Define expected data structures
    DEFINE SubtaskStructure = { id, content, checked }
    DEFINE TaskStructure = { id, content, checked, subtasks: LIST OF SubtaskStructure }
    DEFINE ColumnStructure = { id, title, notes: LIST OF TaskStructure }
    DEFINE SectionStructure = { id, title, columns: LIST OF ColumnStructure, isArchived: BOOLEAN }

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    // Specific document path for ToDo list data
    todoListRef = userStorage.GET_DOCUMENT_REFERENCE("todoList", "data")

    todoListDoc = todoListRef.GET_DOCUMENT()

    IF todoListDoc EXISTS THEN
        todoData = todoListDoc.data
        // Extract sections, handling potential missing data and ensuring structure
        loadedSections = VALIDATE_AND_STRUCTURE(todoData.sections, SectionStructure, ...) // Similar validation as notebooks

        // Separate archived and active sections based on isArchived flag
        activeSections = FILTER loadedSections WHERE section.isArchived IS FALSE
        archivedSections = FILTER loadedSections WHERE section.isArchived IS TRUE

        // Store in application state
        SET_APPLICATION_STATE("currentToDoSections", activeSections)
        SET_APPLICATION_STATE("archivedToDoSections", archivedSections)
```

```
        // Display active sections in the main UI
        DISPLAY activeSections ON TODO_UI()
    ELSE
        // If no data exists, initialize with empty state and save it
        initialState = { sections: [] }
        todoListRef.SET(initialState)
        SET_APPLICATION_STATE("currentToDoSections", [])
        SET_APPLICATION_STATE("archivedToDoSections", [])
        DISPLAY [] ON TODO_UI()
    END IF
END FUNCTION

// Saving ToDo List Data (Called by Debounced Sync)
FUNCTION SaveToDoList(userId, currentActiveSections, currentArchivedSections):
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    todoListRef = userStorage.GET_DOCUMENT_REFERENCE("todoList", "data")

    // Combine active and archived sections back into one list for saving
    allSections = COMBINE(currentActiveSections, currentArchivedSections)

    updateData = {
        sections: allSections,
        lastUpdated: GET_CURRENT_SERVER_TIMESTAMP()
    }

    // Overwrite the document with the combined, current state
    todoListRef.SET(updateData)
END FUNCTION

// Debounced Synchronization (Same logic as Notebooks Module 5.4)
// Uses SaveToDoList as the function to call after the delay.
// FUNCTION OnToDoContentChange(userId, activeSections, archivedSections): ... calls SaveToDoList ...
```

Managing List Items (Sections, Columns, Tasks, Subtasks - Add, Edit, Delete)

```
// Add Section, Add Column, Add Task (Note), Add Subtask
// These follow the same pattern as the Notebooks module (5.2):
// 1. Create new item object with unique ID and default values.
// 2. Update the application state (currentToDoSections).
// 3. Trigger the debounced save via OnToDoContentChange.
FUNCTION AddToDoItem(state, type, parentId1 = NULL, parentId2 = NULL): // parentIds specify section/column/task
    // ... create newItem based on type ...
    newState = ADD newItem TO HIERARCHY(state, parentId1, parentId2)
    SET_APPLICATION_STATE("currentToDoSections", newState)
    CALL OnToDoContentChange( ... )
END FUNCTION

// Editing Titles/Content (Sections, Columns, Tasks, Subtasks)
// Similar to Notebooks (5.2 UpdateNoteContent):
// 1. Find the item in the application state by IDs.
// 2. Update its title or content property.
// 3. Trigger the debounced save.
FUNCTION UpdateToDoItem(state, type, ids, newContentOrTitle):
    newState = MAP state TO FIND_AND_UPDATE_ITEM(ids, newContentOrTitle)
    SET_APPLICATION_STATE("currentToDoSections", newState)
    CALL OnToDoContentChange( ... )
END FUNCTION

// Deleting Items (Sections, Columns, Tasks, Subtasks)
FUNCTION DeleteToDoItemWithUndo(state, type, ids): // ids = {sectionId, columnId, taskId, subtaskId}
    // 1. Find the item and its parent hierarchy to facilitate undo
    itemToDelete = FIND_ITEM(state, ids)
    parentContext = FIND_PARENT_CONTEXT(state, ids)
    originalIndex = FIND_ITEM_INDEX(parentContext, itemToDelete)
```

```
    IF itemNotFound THEN RETURN

    // 2. Update the state by removing the item
    newState = REMOVE_ITEM(state, ids)
    SET_APPLICATION_STATE("currentToDoSections", newState)
    CALL OnToDoContentChange( ... ) // Trigger save immediately or debounced

    // 3. Show a Toast notification with an Undo action
    SHOW_TOAST({
        message: type + " deleted.",
        action: "Undo",
        onUndo: () => {
            // Re-insert the item at its original position
            restoredState = INSERT_ITEM_AT_INDEX(newState, itemToDelete, parentContext, originalIndex)
            SET_APPLICATION_STATE("currentToDoSections", restoredState)
            CALL OnToDoContentChange( ... ) // Trigger save for undo
        }
    })
END FUNCTION
```

Task State Management (Checkbox Toggle, Archiving)

```
// Toggling Checkbox for Task or Subtask
FUNCTION ToggleCompletion(state, ids): // ids = {sectionId, columnId, taskId, subtaskId (optional)}
    newState = MAP state TO FIND_ITEM(ids):
        item.checked = NOT item.checked
    END MAP
    SET_APPLICATION_STATE("currentToDoSections", newState)
    CALL OnToDoContentChange( ... )
END FUNCTION

// Archiving a Section
```

```
FUNCTION ArchiveSection(activeSectionsState, archivedSectionsState, sectionId):
    sectionToArchive = FIND_ITEM(activeSectionsState, {sectionId: sectionId})
    IF sectionNotFound THEN RETURN

    // Remove from active, add to archived with flag set
    newActiveState = REMOVE_ITEM(activeSectionsState, {sectionId: sectionId})
    sectionToArchive.isArchived = TRUE
    newArchivedState = ADD sectionToArchive TO archivedSectionsState

    SET_APPLICATION_STATE("currentToDoSections", newActiveState)
    SET_APPLICATION_STATE("archivedToDoSections", newArchivedState)
    CALL OnToDoContentChange( ... )
END FUNCTION

// Unarchiving a Section
FUNCTION UnarchiveSection(activeSectionsState, archivedSectionsState, sectionId):
    sectionToUnarchive = FIND_ITEM(archivedSectionsState, {sectionId: sectionId})
    IF sectionNotFound THEN RETURN

    // Remove from archived, add to active with flag unset
    newArchivedState = REMOVE_ITEM(archivedSectionsState, {sectionId: sectionId})
    sectionToUnarchive.isArchived = FALSE
    newActiveState = ADD sectionToUnarchive TO activeSectionsState

    SET_APPLICATION_STATE("currentToDoSections", newActiveState)
    SET_APPLICATION_STATE("archivedToDoSections", newArchivedState)
    CALL OnToDoContentChange( ... )
END FUNCTION
```

Handling Item Reordering/Movement (Drag & Drop Logic)

```
// Handles Column, Task, and Subtask drag/drop end events
```

```
FUNCTION HandleToDoDrop(currentState, draggedItemType, draggedItemId, sourceLocation, targetLocation):
    // sourceLocation = {sectionId, columnId, parentTaskId (for subtask)}
    // targetLocation = {sectionId, columnId, parentTaskId (for subtask), targetItemId (optional)}

    // --- Logic similar to Notebooks (5.3) ---

    // 1. Identify the dragged item data based on ID and type
    draggedItem = FIND_ITEM(currentState, sourceLocation, draggedItemId)
    IF itemNotFound THEN RETURN

    // 2. Create temporary state removing item from source
    tempState = REMOVE_ITEM(currentState, sourceLocation, draggedItemId)

    // 3. Determine the target index based on targetLocation (where it was dropped)
    targetIndex = DETERMINE_TARGET_INDEX(tempState, targetLocation)

    // 4. Insert the item into the target location at the calculated index
    //    (Handles moves between columns/tasks or reordering within the same parent)
    newState = INSERT_ITEM_AT_INDEX(tempState, draggedItem, targetLocation, targetIndex)

    // 5. Update application state
    SET_APPLICATION_STATE("currentToDoSections", newState)
    CALL OnToDoContentChange( ... ) // Trigger debounced save
END FUNCTION
```

Stage Manager (WorkStage)

Loading and Persisting Workspace Layout

```
// When the Stage Manager module is loaded for a user
FUNCTION LoadStageLayout(userId):
```

```
    // Define expected structures
    DEFINE WindowStructure = { id, title, content, position: [x, y], size: [width, height] }
    DEFINE SpaceStructure = { id: number, windows: LIST OF WindowStructure } // Use number ID for easy indexing if needed

    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    // Specific document for Stage Manager data
    stageDataRef = userStorage.GET_DOCUMENT_REFERENCE("stageManager", "data")

    stageDoc = stageDataRef.GET_DOCUMENT()

    IF stageDoc EXISTS THEN
        stageData = stageDoc.data
        // Validate and structure the loaded spaces and windows
        loadedSpaces = VALIDATE_AND_STRUCTURE(stageData.spaces, SpaceStructure, WindowStructure)

        // Ensure at least one space exists, create default if none
        IF LENGTH(loadedSpaces) == 0 THEN
            defaultSpace = { id: 0, windows: [ { id: GENERATE_UNIQUE_ID(), title: "New Document", content: "", position: [100, 100], size: [400,
300] } ] }
            loadedSpaces = [defaultSpace]
            // Persist the default state immediately
            CALL SaveStageLayout(userId, loadedSpaces)
        END IF

        // Set the initial state
        SET_APPLICATION_STATE("stageSpaces", loadedSpaces)
        SET_APPLICATION_STATE("currentStageSpaceId", loadedSpaces[0].id) // Default to the first space
        DISPLAY loadedSpaces[0] ON STAGE_UI() // Display the first space
    ELSE
        // Create and save the default initial state if no data exists
        defaultSpace = { id: 0, windows: [ { id: GENERATE_UNIQUE_ID(), title: "New Document", content: "", position: [100, 100], size: [400, 300]
} ] }
        initialState = { spaces: [defaultSpace] }
```

```
        stageDataRef.SET(initialState)
        SET_APPLICATION_STATE("stageSpaces", [defaultSpace])
        SET_APPLICATION_STATE("currentStageSpaceId", 0)
        DISPLAY defaultSpace ON STAGE_UI()
    END IF
END FUNCTION

// Saving the entire Stage Layout (Called whenever 'stageSpaces' state changes)
// NOTE: Unlike ToDo/Notebooks, Stage Manager might benefit from saving the *entire* layout
// on change rather than debouncing, as layout changes are discrete events. Or debounce lightly.
FUNCTION SaveStageLayout(userId, currentSpacesState):
    userStorage = GET_STORAGE_AREA_FOR_USER(userId)
    stageDataRef = userStorage.GET_DOCUMENT_REFERENCE("stageManager", "data")

    updateData = {
        spaces: currentSpacesState
        // lastUpdated: GET_CURRENT_SERVER_TIMESTAMP() // Optional timestamp
    }

    // Overwrite the document with the current layout state
    stageDataRef.SET(updateData)
END FUNCTION
```

Space Management (Create, Switch, Delete)

```
// Creating a New Space
FUNCTION CreateSpace(userId, currentSpacesState):
    // Determine the next available ID (e.g., max current ID + 1)
    nextId = MAX(currentSpacesState.map(s => s.id)) + 1

    // Create a default window for the new space
    defaultWindow = {
```

```
        id: GENERATE_UNIQUE_ID(),
        title: "New Document",
        content: "",
        position: [100, 100], // Default position
        size: [400, 300]     // Default size
    }

    newSpace = { id: nextId, windows: [defaultWindow] }

    // Update state and trigger save
    newState = ADD newSpace TO currentSpacesState
    SET_APPLICATION_STATE("stageSpaces", newState)
    // Switch to the newly created space
    SET_APPLICATION_STATE("currentStageSpaceId", nextId)
    CALL SaveStageLayout(userId, newState)

    // Update UI to show the new space
    DISPLAY newSpace ON STAGE_UI()
END FUNCTION

// Switching Between Spaces
FUNCTION SwitchSpace(targetSpaceId):
    // Validate targetSpaceId exists in currentSpacesState (optional)
    SET_APPLICATION_STATE("currentStageSpaceId", targetSpaceId)
    // Update UI to display the windows of the targetSpaceId
    targetSpace = FIND_SPACE_BY_ID(GET_APPLICATION_STATE("stageSpaces"), targetSpaceId)
    DISPLAY targetSpace ON STAGE_UI()
END FUNCTION

// Deleting a Space
FUNCTION DeleteSpace(userId, currentSpacesState, currentSpaceId, spaceIdToDelete):
    // Prevent deleting the last remaining space
    IF LENGTH(currentSpacesState) ⩽ 1 THEN
```

```
        SHOW_ERROR("Cannot delete the last space.")
        RETURN
    END IF

    // Show confirmation dialog
    CONFIRM("Are you sure you want to delete Space " + (spaceIdToDelete + 1) + "?", () ⇒ {
        // Filter out the space to delete
        newState = FILTER currentSpacesState WHERE space.id ≠ spaceIdToDelete

        // Determine the next active space ID
        nextActiveSpaceId = currentSpaceId
        IF currentSpaceId == spaceIdToDelete THEN
            // If deleting the current space, switch to the first remaining one
            nextActiveSpaceId = newState[0].id
        END IF

        // Update state and save
        SET_APPLICATION_STATE("stageSpaces", newState)
        SET_APPLICATION_STATE("currentStageSpaceId", nextActiveSpaceId)
        CALL SaveStageLayout(userId, newState)

        // Update UI
        CALL SwitchSpace(nextActiveSpaceId) // Display the new current space
        SHOW_TOAST("Space deleted.")
    })
END FUNCTION
```

Window Management (Create, Move, Resize, Update Content, Delete)

```
// Creating a New Window in the Current Space
FUNCTION AddWindowToCurrentSpace(userId, currentSpacesState, currentSpaceId):
    targetSpace = FIND_SPACE_BY_ID(currentSpacesState, currentSpaceId)
```

```
    IF targetSpace IS NULL THEN RETURN

    // Calculate staggered position for the new window
    staggerOffset = LENGTH(targetSpace.windows) * 20
    newPosition = [100 + staggerOffset, 100 + staggerOffset]

    newWindow = {
        id: GENERATE_UNIQUE_ID(),
        title: "New Document",
        content: "",
        position: newPosition,
        size: [400, 300] // Default size
    }

    // Update the state for the specific space
    newState = MAP currentSpacesState WHERE space.id == currentSpaceId:
        ADD newWindow TO space.windows
    END MAP

    SET_APPLICATION_STATE("stageSpaces", newState)
    CALL SaveStageLayout(userId, newState)
    // UI should update automatically if displaying the current space's windows
END FUNCTION

// Updating Window Position (e.g., after dragging)
FUNCTION UpdateWindowPosition(userId, currentSpacesState, spaceId, windowId, newPosition):
    newState = MAP currentSpacesState WHERE space.id == spaceId:
        MAP space.windows WHERE window.id == windowId:
            window.position = newPosition
        END MAP
    END MAP
    SET_APPLICATION_STATE("stageSpaces", newState)
    CALL SaveStageLayout(userId, newState)
```

```
END FUNCTION

// Updating Window Size (e.g., after resizing)
FUNCTION UpdateWindowSize(userId, currentSpacesState, spaceId, windowId, newSize):
    newState = MAP currentSpacesState WHERE space.id == spaceId:
        MAP space.windows WHERE window.id == windowId:
            window.size = newSize
        END MAP
    END MAP
    SET_APPLICATION_STATE("stageSpaces", newState)
    CALL SaveStageLayout(userId, newState)
END FUNCTION

// Updating Window Title or Content
FUNCTION UpdateWindowTitleContent(userId, currentSpacesState, spaceId, windowId, propertyToUpdate, newValue):
    newState = MAP currentSpacesState WHERE space.id == spaceId:
        MAP space.windows WHERE window.id == windowId:
            window[propertyToUpdate] = newValue // Update either 'title' or 'content'
        END MAP
    END MAP
    SET_APPLICATION_STATE("stageSpaces", newState)
    CALL SaveStageLayout(userId, newState)
END FUNCTION

// Deleting a Window
FUNCTION DeleteWindow(userId, currentSpacesState, spaceId, windowId, windowTitle):
     // Show confirmation toast
    SHOW_TOAST({
        message: "Delete window '" + windowTitle + "'?",
        action: "Delete",
        onAction: () => {
            newState = MAP currentSpacesState WHERE space.id == spaceId:
                space.windows = FILTER space.windows WHERE window.id ≠ windowId
```

```
            END MAP

            SET_APPLICATION_STATE("stageSpaces", newState)
            CALL SaveStageLayout(userId, newState)
            SHOW_TOAST("Window deleted.")
            // UI updates automatically
        }
    })
END FUNCTION
```

Validation

Field Name	Data Validation Type	Explanation
Cookie Jar Module		
Cookie Name	• Presence Check	Must not be empty; User must provide a name for the cookie.
	• Type Check	Must be text (string).
Cookie Description	• Presence Check	Must not be empty; User must provide a description for the cookie.
	• Type Check	Must be text (string).
Doubt Tracker Module		
Doubt Title	• Presence Check	Must not be empty; User must provide a title for the doubt.
	• Type Check	Must be text (string).
Doubt Description	• Presence Check	Must not be empty; User must provide a description for the doubt.
	• Type Check	Must be text (string).
Doubt Solution	• Presence Check	Must not be empty when marking doubt as 'Resolved'.
	• Type Check	Must be text (string).
Doubt Comment	• Presence Check	Must not be empty when submitting a comment.

Field Name	Data Validation Type	Explanation
	• Type Check	Must be text (string).
Curiosity Space Module		
Idea Title	• Presence Check	Must not be empty; User must provide a title for the idea.
	• Type Check	Must be text (string).
Idea Description	• Presence Check	Must not be empty; User must provide a description for the idea.
	• Type Check	Must be text (string).
Idea Solution	• Presence Check	Must not be empty when marking idea as 'Resolved'.
	• Type Check	Must be text (string).
Idea Comment	• Presence Check	Must not be empty when submitting a comment.
	• Type Check	Must be text (string).
To-Do List Module		
Section Title	• Presence Check	Must not be empty; User must provide a title for the section.
	• Type Check	Must be text (string).
Column Title	• Presence Check	Must not be empty; User must provide a title for the column.
	• Type Check	Must be text (string).

Field Name	Data Validation Type	Explanation
Note Content	• Presence Check	Must not be empty; User must provide content for the note.
	• Type Check	Must be text (string).
Subtask Content	• Presence Check	Must not be empty; User must provide content for the subtask.
	• Type Check	Must be text (string).
Continuous Info Space Module		
Notebook Title	• Presence Check	Must not be empty; User must provide a title for the notebook.
	• Type Check	Must be text (string).
Section Title	• Presence Check	Must not be empty; User must provide a title for the section.
	• Type Check	Must be text (string).
Column Title	• Presence Check	Must not be empty; User must provide a title for the column.
	• Type Check	Must be text (string).
Note Content	• Presence Check	Must not be empty; User must provide content for the note.
	• Type Check	Must be text (string).
Notebook Description	• Type Check	Must be text (string).
Note Description	• Type Check	Must be text (string).

Field Name	Data Validation Type	Explanation
Stage Manager Module		
Space Name	• Presence Check	Must not be empty; User must provide a name for the Space.
	• Type Check	Must be text (string).
Window Title	• Presence Check	Must not be empty; User must provide a title for the Window.
	• Type Check	Must be text (string).
All Text Fields	• Presence Check (where indicated)	Ensures that required text fields are not left blank.
	• Type Check	Ensures that the input is of the expected data type (text/string).

Test Plan

Tabular test plan

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
Cookie Jar	Create Cookie	Name: "Achieved Goal", Description: "Completed IA Crit A"	Normal	New cookie card "Achieved Goal" with description "Completed IA Crit A" is created and displayed.	1, 2, 9
Cookie Jar	Create Cookie (Missing Name)	Description: "Just a description"	Abnormal	Error message (if implemented, otherwise handled gracefully), Cookie creation is prevented.	1, 9
Cookie Jar	Edit Cookie	Select "Achieved Goal", Change Description to "Revised IA Doc"	Normal	Cookie card "Achieved Goal" description updates to "Revised IA Doc".	3, 9
Cookie Jar	Delete Cookie	Select "Achieved Goal", Click "Delete"	Normal	Cookie card "Achieved Goal" is removed from the Cookie Jar.	4, 9
Cookie Jar	Reorder Cookies (Drag & Drop)	Drag "Achieved Goal" cookie to a new position	Normal	"Achieved Goal" cookie is reordered to the new position, and the order persists after refresh.	5, 9

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
Cookie Jar	Persistence Across Sessions	Create cookies, close app, reopen app	Normal	Cookies created in previous session are still present and displayed in the Cookie Jar.	2, 9
Doubt Tracker	Create Doubt	Title: "Maths Doubt", Description: "Integration issue"	Normal	New doubt card "Maths Doubt" with description "Integration issue" is created and displayed.	1, 2, 10
Doubt Tracker	Create Doubt (Missing Title)	Description: "Just a description"	Abnormal	Error message (if implemented, otherwise handle gracefully), Doubt creation is prevented.	1, 10
Doubt Tracker	Resolve Doubt	Select "Maths Doubt", Click "Resolve", Solution: "Use substitution"	Normal	"Maths Doubt" card is marked as resolved, Solution "Use substitution" is displayed.	8, 10
Doubt Tracker	Reopen Doubt	Select "Maths Doubt" (Resolved), Click "Reopen"	Normal	"Maths Doubt" card is marked as open, Solution is hidden.	8, 10
Doubt Tracker	Upvote Doubt	Select "Maths Doubt", Click "Upvote"	Normal	Upvote count for "Maths Doubt" increments by 1.	6, 10

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
Doubt Tracker	Downvote Doubt	Select "Maths Doubt", Click "Downvote"	Normal	Downvote count for "Maths Doubt" increments by 1.	6, 10
Doubt Tracker	Add Comment	Select "Maths Doubt", Add comment: "Need more details"	Normal	Comment "Need more details" is added to "Maths Doubt" and displayed.	7, 10
Doubt Tracker	Edit Comment	Select comment "Need more details", Edit to "Clarify question"	Normal	Comment text updates to "Clarify question".	7, 10
Doubt Tracker	Delete Comment	Select comment "Clarify question", Delete comment	Normal	Comment "Clarify question" is removed from the Doubt card.	7, 10
Curiosity Space	Create Idea	Title: "New App Idea", Description: "AI powered note-taking"	Normal	New idea card "New App Idea" with description "AI powered note-taking" is created and displayed.	1, 2, 11
Curiosity Space	Resolve Idea	Select "New App Idea", Click "Resolve", Solution: "Research APIs"	Normal	"New App Idea" card is marked as resolved, Solution "Research APIs" is displayed.	8, 11

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
To-Do List	Create Section	Click "Add Section"	Normal	New section "New Section" is created and displayed.	1, 2, 12
To-Do List	Create Column	Select "New Section", Click "Add Column"	Normal	New column "New Column" is created within "New Section".	1, 2, 12
To-Do List	Create Note/Task	Select "New Column", Click "Add Note"	Normal	New note "New note" is created within "New Column".	1, 2, 12
To-Do List	Check/Uncheck Task	Check checkbox next to "New note"	Normal	Note text is visually marked as completed (e.g., line-through).	12
To-Do List	Delete Section	Select "New Section", Click "Archive"	Normal	"New Section" is archived (removed from main view, accessible in archive).	4, 12
To-Do List	Reorder Sections (Drag & Drop)	Drag "New Section" to a new position	Normal	"New Section" is reordered to the new position.	5, 12
To-Do List	Add Subtask	Select "New note", Click "Add Subtask"	Normal	New subtask "New Subtask" is created under "New note".	1, 2, 12
Continuous Info Space	Create Notebook	Click "New Notebook", Title: "Maths Notes"	Normal	New notebook card "Maths Notes" is created and displayed.	1, 2, 13

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
Continuous Info Space	Delete Notebook	Select "Maths Notes", Click "Delete"	Normal	"Maths Notes" notebook card is removed.	4, 13
Continuous Info Space	Create Section in Notebook	Open "Maths Notes", Click "Add Section"	Normal	New section "New Section" is created within "Maths Notes" notebook.	1, 2, 13
Continuous Info Space	Create Column in Section	Open "Maths Notes", Select "New Section", Click "Add Column"	Normal	New column "New Column" is created within "New Section" in "Maths Notes".	1, 2, 13
Continuous Info Space	Create Note in Column	Open "Maths Notes", Select "New Column", Click "Add Note"	Normal	New note "New note" is created within "New Column" in "Maths Notes".	1, 2, 13
Stage Manager	Create Space	Click "Create New Space"	Normal	New space "Space 2" (if Space 1 exists) is created.	1, 2, 14
Stage Manager	Delete Space	Select "Space 2", Click "Delete Space"	Normal	"Space 2" is deleted.	4, 14
Stage Manager	Create Window	Select "Space 1", Click "Add Window"	Normal	New window "New Document" is created in "Space 1".	1, 2, 14

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
Stage Manager	Delete Window	Select "New Document", Click "Close" (X)	Normal	"New Document" window is deleted from "Space 1".	4, 14
Stage Manager	Move Window (Drag Title Bar)	Drag "New Document" window	Normal	"New Document" window is moved to the dragged position.	14
Stage Manager	Resize Window (Drag Resizer)	Drag the corner resizer of "New Document" window	Normal	"New Document" window is resized as dragged.	14
Stage Manager	Persist Layout	Create Spaces & Windows, arrange them, close & reopen app	Normal	Spaces and Windows layout (arrangement, positions, sizes) are persisted across sessions.	2, 14
General Application	Intuitive UI Navigation	Navigate through all modules	Normal	User can easily navigate between modules and understand the UI elements without extensive learning.	15
General Application	Cross-browser Compatibility (Chrome)	Access application via Chrome browser	Normal	Application functions correctly and is usable in Chrome without errors or UI issues.	16

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
General Application	Cross-browser Compatibility (Firefox)	Access application via Firefox browser	Normal	Application functions correctly and is usable in Firefox without errors or UI issues.	16
General Application	Cross-browser Compatibility (Safari)	Access application via Safari browser	Normal	Application functions correctly and is usable in Safari without errors or UI issues.	16
General Application	Cross-browser Compatibility (Edge)	Access application via Edge browser	Normal	Application functions correctly and is usable in Edge without errors or UI issues.	16
General Application	Cross-device Compatibility (Desktop)	Access application on a desktop computer	Normal	Application functions correctly and is usable on a desktop computer, UI is responsive.	16
General Application	Cross-device Compatibility (Laptop)	Access application on a laptop	Normal	Application functions correctly and is usable on a laptop, UI is responsive.	16
General Application	Cross-device Compatibility (Tablet)	Access application on a tablet	Normal	Application functions correctly and is usable on a tablet, UI is responsive (if tablet support is implemented).	16

Module/Feature	Test Case	Test Data/Input	Type	Expected Result	Success Criteria (Reference)
General Application	User Authentication (Sign-in)	Attempt to access application without signing in	Normal	User is redirected to the sign-in page and cannot access application features without authentication.	17
General Application	Real-time Data Sync	Open app on two devices with same account, make changes on one	Normal	Changes made on one device are reflected in real-time (or near real-time) on the other device.	18
General Application	Fast Data Retrieval	Open each module and perform data-loading actions	Normal	Modules and data load quickly with minimal loading time, providing a responsive user experience.	19
All Modules/Features	Input Validation (e.g., long text)	Enter very long text strings in all input fields	Extreme	Application handles long inputs gracefully without crashing or causing UI issues.	Robustness (Implied)
All Modules/Features	No Input	Attempt to perform actions without entering required information	Abnormal	Application prevents actions or provides informative error messages when required input is missing.	Input Validation (Implied)

Testing General and Database Functionality of Information Management System:

User Authentication:

1. **Register New User Account:** Register using a new email and password. Verify successful account creation and redirection to the application dashboard. (Checks Firebase Authentication for new user creation and proper user session setup).
2. **Attempt Duplicate Registration:** Try registering again using the *same* email as in test 1. Verify the application *prevents* duplicate registration and displays an appropriate error message. (Checks Firebase Authentication to ensure duplicate user creation is blocked).
3. **Login with Valid Credentials:** Login using the email and password created in test 1. Verify successful login and redirection to the application dashboard. (Checks Firebase Authentication to match email and password, and establishes user session).
4. **Login with Invalid Credentials:** Attempt login using an *incorrect* password for the registered email. Verify login is *prevented* and an appropriate error message is displayed. (Checks Firebase Authentication to ensure login fails with incorrect credentials).
5. **Multiple Account Isolation:** Log in with two different user accounts. Verify that data (cookies, doubts, ideas, to-do lists, notebooks, spaces/windows) is isolated between the two accounts and not accessible across accounts. (Checks Firebase Firestore rules to ensure data segregation based on user ID).

Cookie Jar Module:

6. **Create New Cookie:** In the Cookie Jar, create a new cookie with a name and description. Verify the cookie card is created and displayed in the Cookie Jar. (Checks Firebase Firestore to ensure new cookie data is stored under the user's cookieJar collection and data is persisted).
7. **Edit Existing Cookie:** Edit the name and description of an existing cookie. Verify the cookie card is updated with the new information in the Cookie Jar. (Checks Firebase Firestore to ensure cookie data is updated in the database and changes are reflected in the UI).
8. **Delete Cookie:** Delete an existing cookie from the Cookie Jar. Verify the cookie card is removed from the Cookie Jar. (Checks Firebase Firestore to ensure cookie data is deleted from the database and UI is updated).

9. **Reorder Cookies (Drag and Drop):** Drag and drop cookie cards to reorder them in the Cookie Jar. Verify the new order is saved and persists after refreshing or reopening the application. (Checks Firebase Firestore to ensure cookie positions are updated in the database and the new order is maintained).

Doubt Tracker & Curiosity Space (Idea Tracker) Modules:

10. **Create New Doubt/Idea:** In the Doubt Tracker/Curiosity Space, create a new doubt/idea with a title and description. Verify the doubt/idea card is created and displayed in the respective module list. (Checks Firebase Firestore to ensure new doubt/idea data is stored under the user's posts/nugget collection respectively and data is persisted).
11. **Edit Existing Doubt/Idea:** Edit the title and description of an existing doubt/idea. Verify the doubt/idea card is updated with the new information. (Checks Firebase Firestore to ensure doubt/idea data is updated in the database and changes are reflected in the UI).
12. **Delete Doubt/Idea:** Delete an existing doubt/idea. Verify the doubt/idea card is removed from the list. (Checks Firebase Firestore to ensure doubt/idea data is deleted from the database and UI is updated).
13. **Resolve Doubt/Idea:** Resolve an open doubt/idea by adding a solution. Verify the doubt/idea is marked as resolved and the solution is displayed. (Checks Firebase Firestore to ensure the 'resolved' status and 'solution' are updated in the database and UI reflects the resolved state).
14. **Reopen Resolved Doubt/Idea:** Reopen a resolved doubt/idea. Verify the doubt/idea is marked as open again and the solution is hidden. (Checks Firebase Firestore to ensure the 'resolved' status and 'solution' are reverted in the database and UI reflects the reopened state).
15. **Upvote/Downvote Doubt/Idea:** Upvote or downvote a doubt/idea. Verify the vote count is updated on the doubt/idea card. (Checks Firebase Firestore to ensure 'upvotes' or 'downvotes' count is incremented in the database and the vote count is updated in the UI).
16. **Add Comment to Doubt/Idea:** Add a comment to a doubt/idea. Verify the comment is displayed in the comment section. (Checks Firebase Firestore to ensure the comment data is stored under the doubt/idea's comments subcollection and the comment is displayed in the UI).
17. **Edit Comment:** Edit an existing comment. Verify the comment text is updated. (Checks Firebase Firestore to ensure the comment data is updated in the database and the edited comment is displayed in the UI).

18. **Delete Comment:** Delete an existing comment. Verify the comment is removed from the comment section. (Checks Firebase Firestore to ensure the comment data is deleted from the database and the comment is removed from the UI).

To-Do List Module:

19. **Create New Section:** In the To-Do List, create a new section. Verify a new section is added to the To-Do List. (Checks Firebase Firestore to ensure the new section data is stored and persisted).
20. **Edit Section Title:** Edit the title of a section. Verify the section title is updated in the To-Do List. (Checks Firebase Firestore to ensure the section title is updated in the database and UI reflects the change).
21. **Delete Section:** Delete an existing section. Verify the section and all its columns and notes are removed from the To-Do List. (Checks Firebase Firestore to ensure the section data and associated data are deleted from the database and UI is updated).
22. **Create New Column:** In a section, create a new column. Verify a new column is added to the section. (Checks Firebase Firestore to ensure the new column data is stored under the section and persisted).
23. **Edit Column Title:** Edit the title of a column. Verify the column title is updated in the To-Do List. (Checks Firebase Firestore to ensure the column title is updated in the database and UI reflects the change).
24. **Delete Column:** Delete an existing column. Verify the column and all its notes are removed from the section. (Checks Firebase Firestore to ensure the column data and associated notes are deleted from the database and UI is updated).
25. **Add New Note/Task:** In a column, add a new note/task. Verify a new note/task is added to the column. (Checks Firebase Firestore to ensure the new note/task data is stored under the column and persisted).
26. **Edit Note/Task Content:** Edit the content of a note/task. Verify the note/task content is updated in the To-Do List. (Checks Firebase Firestore to ensure the note/task content is updated in the database and UI reflects the change).
27. **Delete Note/Task:** Delete an existing note/task. Verify the note/task is removed from the column. (Checks Firebase Firestore to ensure the note/task data is deleted from the database and UI is updated).
28. **Check/Uncheck Note (Task Completion):** Check and uncheck a note/task to mark it as complete/incomplete. Verify the check status is saved and visually indicated (e.g., strikethrough). (Checks Firebase Firestore to ensure the 'checked' status of the note/task is updated and the UI reflects the completion status).

29. **Reorder Columns (Drag and Drop):** Drag and drop columns within a section to reorder them. Verify the new column order is saved and persists. (Checks Firebase Firestore to ensure the column order within the section is updated and the new order is maintained).
30. **Reorder Notes (Drag and Drop):** Drag and drop notes within a column to reorder them. Verify the new note order is saved and persists. (Checks Firebase Firestore to ensure the note order within the column is updated and the new order is maintained).
31. **Move Note between Columns (Drag and Drop):** Drag and drop a note from one column to another (within the same or different section). Verify the note moves to the new column and is removed from the original column. (Checks Firebase Firestore to ensure the note is moved to the new column's data in the database and removed from the original column).
32. **Add Subtask to Note:** Add a subtask to a note. Verify the subtask is added and displayed under the note. (Checks Firebase Firestore to ensure the subtask data is stored under the parent note and persisted).
33. **Delete Subtask:** Delete a subtask. Verify the subtask is removed from the note. (Checks Firebase Firestore to ensure the subtask data is deleted from the database and UI is updated).
34. **Check/Uncheck Subtask:** Check and uncheck a subtask to mark it as complete/incomplete. Verify the subtask's check status is saved and visually indicated. (Checks Firebase Firestore to ensure the 'checked' status of the subtask is updated and the UI reflects the completion status).
35. **Reorder Subtasks (Drag and Drop):** Drag and drop subtasks within a note to reorder them. Verify the new subtask order is saved and persists. (Checks Firebase Firestore to ensure the subtask order within the note is updated and the new order is maintained).
36. **Move Subtask between Notes (Drag and Drop):** Drag and drop a subtask from one note to another (within the same column). Verify the subtask moves to the new note and is removed from the original note. (Checks Firebase Firestore to ensure the subtask is moved to the new note's data in the database and removed from the original note).

Continuous Information Space Module:

37. **Create New Notebook:** In the Continuous Information Space, create a new notebook. Verify a new notebook card is created in the Notebook Manager. (Checks Firebase Firestore to ensure new notebook data is stored under the user's notebooks collection and persisted).

38. **Edit Notebook Title/Description:** Edit the title and description of an existing notebook. Verify the notebook card is updated with the new information. (Checks Firebase Firestore to ensure notebook data is updated in the database and changes are reflected in the UI).
39. **Delete Notebook:** Delete an existing notebook. Verify the notebook card is removed from the Notebook Manager. (Checks Firebase Firestore to ensure notebook data is deleted from the database and UI is updated).
40. **Create New Section in Notebook:** Open a notebook and create a new section. Verify a new section is added to the notebook. (Checks Firebase Firestore to ensure the new section data is stored under the notebook and persisted).
41. **Edit Section Title (Notebook):** Edit the title of a section within a notebook. Verify the section title is updated in the notebook. (Checks Firebase Firestore to ensure the section title is updated in the database and UI reflects the change).
42. **Delete Section (Notebook):** Delete a section from a notebook. Verify the section and all its columns and notes are removed from the notebook. (Checks Firebase Firestore to ensure the section data and associated data are deleted from the database and UI is updated).
43. **Create New Column (Notebook Section):** In a notebook section, create a new column. Verify a new column is added to the section. (Checks Firebase Firestore to ensure the new column data is stored under the section and persisted).
44. **Edit Column Title (Notebook Section):** Edit the title of a column within a notebook section. Verify the column title is updated in the notebook section. (Checks Firebase Firestore to ensure the column title is updated in the database and UI reflects the change).
45. **Delete Column (Notebook Section):** Delete a column from a notebook section. Verify the column and all its notes are removed from the section. (Checks Firebase Firestore to ensure the column data and associated notes are deleted from the database and UI is updated).
46. **Add New Note (Notebook Column):** In a notebook column, add a new note. Verify a new note is added to the column. (Checks Firebase Firestore to ensure the new note data is stored under the column and persisted).
47. **Edit Note Content (Notebook):** Edit the content of a note within a notebook column. Verify the note content is updated in the notebook. (Checks Firebase Firestore to ensure the note content is updated in the database and UI reflects the change).
48. **Delete Note (Notebook):** Delete a note from a notebook column. Verify the note is removed from the column. (Checks Firebase Firestore to ensure the note data is deleted from the database and UI is updated).

49. **Reorder Sections (Drag and Drop - Notebook):** Drag and drop sections within a notebook to reorder them. Verify the new section order is saved and persists. (Checks Firebase Firestore to ensure the section order within the notebook is updated and the new order is maintained).
50. **Reorder Columns (Drag and Drop - Notebook Section):** Drag and drop columns within a notebook section to reorder them. Verify the new column order is saved and persists. (Checks Firebase Firestore to ensure the column order within the notebook section is updated and the new order is maintained).
51. **Reorder Notes (Drag and Drop - Notebook Column):** Drag and drop notes within a notebook column to reorder them. Verify the new note order is saved and persists. (Checks Firebase Firestore to ensure the note order within the notebook column is updated and the new order is maintained).

Stage Manager Module:

52. **Create New Space:** In the Stage Manager, create a new space. Verify a new space is added to the space switcher. (Checks Firebase Firestore to ensure new space data is stored and persisted for the user's stage manager data).
53. **Delete Space:** Delete an existing space. Verify the space is removed from the space switcher. (Checks Firebase Firestore to ensure space data is deleted from the database and UI is updated).
54. **Switch Between Spaces:** Switch between different created spaces. Verify the application switches to the selected space and displays its windows. (Checks UI state to ensure correct space and its windows are loaded and displayed).
55. **Create New Window in Space:** In a space, create a new window. Verify a new window is added to the current space. (Checks Firebase Firestore to ensure new window data is stored under the current space and persisted).
56. **Delete Window:** Delete an existing window from a space. Verify the window is removed from the space. (Checks Firebase Firestore to ensure window data is deleted from the database and UI is updated).
57. **Move Window (Drag):** Drag a window within a space to a new position. Verify the window moves to the new position and the new position is saved. (Checks Firebase Firestore to ensure window position is updated in the database and the new position is maintained).
58. **Resize Window (Drag Edges):** Resize a window by dragging its edges. Verify the window resizes and the new size is saved. (Checks Firebase Firestore to ensure window size is updated in the database and the new size is maintained).

59. **Persistence of Spaces and Windows Layout:** Create spaces and windows, arrange them, and then close and reopen the application (or refresh the page after logging out and back in). Verify that the spaces and windows are restored in the same layout and configuration as before. (Checks Firebase Firestore to ensure the entire stage manager data including spaces and windows layouts is loaded and persisted across sessions).

General Application Functionality:

60. **Cross-Browser Compatibility:** Test all core functionalities (creating, editing, deleting items in each module, drag-and-drop, login/logout, space switching) on major web browsers (Chrome, Firefox, Safari, Edge). Verify that the application functions correctly and without UI issues on each browser.
61. **Cross-Device Compatibility:** Test all core functionalities on different devices (desktop, laptop, tablet - if applicable). Verify that the application is usable and responsive on different screen sizes and input methods.
62. **User Authentication Security:** Attempt to access another user's data by manually changing user IDs in the browser's local storage or making direct Firestore requests (if you know how to do this for testing purposes). Verify that Firebase security rules prevent unauthorized data access. (Verifies Firebase Firestore rules are correctly configured and enforced).
63. **Real-time Data Synchronization:** Open the application on two different devices or browser windows logged in with the same user account. Make changes in one location (e.g., add a cookie, check a to-do item). Verify that the changes are reflected in real-time on the other device/window. (Observes real-time updates in the UI across multiple instances of the application).
64. **Fast Data Retrieval:** Navigate to each module (Cookie Jar, Doubt Tracker, etc.) and observe the loading time for data to appear. Verify that data is retrieved and displayed quickly, ensuring a responsive user experience. (Subjective observation of loading times and application responsiveness).
65. **Intuitive User Interface:** Ask Isht (or another representative user) to perform common tasks within the application (e.g., create a to-do list, add a doubt, organize notebooks). Observe their ease of use and gather feedback on the intuitiveness of the navigation and UI elements. (Subjective usability assessment based on user observation and feedback).