Subject Name: **Front End Engineering**

Subject Code: CS186

Cluster: iGamma

Department: **DCSE**

Group: **G19**



Project Name: Event Gallery

**Submitted By:** Vanshika 2110991959 G19

# Submitted To:

Ms. Pritpal Kaur

# React -Firebase Event Gallery

## Introduction

Welcome to the documentation of our Event Gallery, a React-based web application. In this documentation, we will explore the key components that make up our Event Gallery and their respective functionalities. The Event Gallery is designed to showcase and manage a collection of images, making it easy for users to view and interact with these images in a visually appealing way.

## Technologies Used:

* Firebase
* CSS
* JavaScript
* React

## Key Components

### EventGallery.jsx

### The "EventGallery" component is the core component of our application.

### Renders a user-friendly interface for managing and viewing images in the gallery.

### Allows users to upload images.

### Displays images in a grid layout.

### Provides a modal for a larger view of images.

### Handles the state of the selected image and modal.

### Config.jsx

* Initializes the Firebase app with the provided configuration.
* Provides references to Firebase Storage and Firestore services.
* Makes the initialized app and services available for other parts of the application.

### Title.jsx

* Displays the logo, name, and subtitle of the application.
* Provides a visually appealing header for your application.
* Offers a brief description to introduce the application's purpose.

### UploadForm.jsx

### Provides an image upload form for users.

### Validates selected files against allowed file types.

### Displays an error message for invalid file types.

### Shows the name of the selected file.

### Renders the ProgressBar component to indicate upload progress when a valid file is selected.

### ProgressBar.jsx

### Visualizes the progress of image uploads with a progress bar.

### Uses animations from the Framer Motion library for smooth transitions.

### Monitors changes in the upload process and manages the file state upon completion.

### ImageGrid.jsx

### Retrieves image data from Firestore using the useFirestore hook.

### Utilizes the "framer-motion" library to add animations for a smoother and more interactive user experience.

### Maps through the fetched image documents to render them in a grid.

### Each image is wrapped in a motion-div to enable animations on hover.

### The setSelectedImg function is used to allow users to click on images, select them, and trigger a larger view in a modal.

### Modal.jsx

### Displays an enlarged view of selected images in a modal format.

### Allows users to close the modal by clicking outside the image.

### Utilizes animations from the Framer Motion library for smooth transitions.

### UseFirestore.jsx

### Fetches documents from a specified Firestore collection.

### Orders documents based on a specified field and direction.

### Provides real-time updates to the consuming component when the Firestore data changes.

### Automatically unsubscribes from real-time updates when the component unmounts.

### Usestorage.jsx

### Manages the upload process of images to Firebase Storage.

### Tracks and provides feedback on the upload progress.

### Handles and reports any errors that occur during the upload.

### Retrieves and provides the URL of the uploaded image.

### Optionally saves the URL to Firebase Firestore with a timestamp.

### CSS Styling

### The CSS styles used in the Event Gallery application are responsible for its visual appearance and user interface. The styles are defined for various components, such as the form, progress bar, image grid, and modal. These styles create a cohesive and visually appealing experience for users.

### Index.js

* Initializes and renders your Event Gallery application.
* Loads the main component, EventGallery, which serves as the starting point of your application.
* Sets up the required CSS styling by importing the 'eventgal.css' file.

## Usage

## The application starts with the initialization of Firebase app and services.

## It then proceeds to render the Event Gallery page.

## The Event Gallery page fetches images from Firestore.

## The Image Grid component is responsible for displaying the images in a grid.

## When an image is clicked, it triggers the display of an enlarged image modal.

## The user can close the modal to return to the Image Grid.

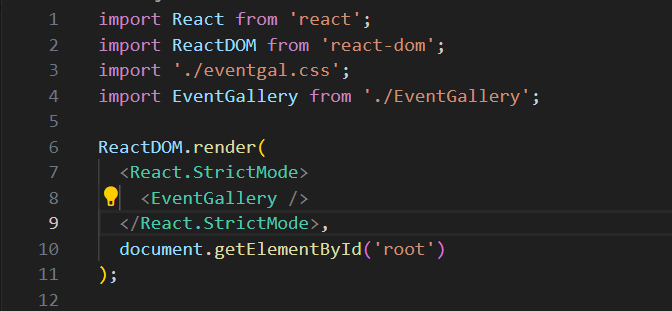
## The flowchart also includes the start and end points of the application.

## Flow Chart:

**Components Documentation**

## Index.js

The "index.js" file is the entry point for your React application. It initializes the application by rendering the root component, "App," into the root HTML element on the web page.



* The ReactDOM.render function is used to render your application's main component, which is EventGallery, into the root HTML element of your webpage.
* It wraps the EventGallery component in a React.StrictMode to highlight potential problems in your application during development.

## EventGallery.jsx

The EventGallery component is the main entry point for our Event Gallery application. It is responsible for rendering the key elements of our gallery, including the title, image upload form, image grid, and the modal for enlarged images.



**Functions:**

* The EventGallery component is the core of our Event Gallery application.
* It uses the useState hook to manage the state of the selected image, initially set to null.

Within this component, the following elements are rendered:

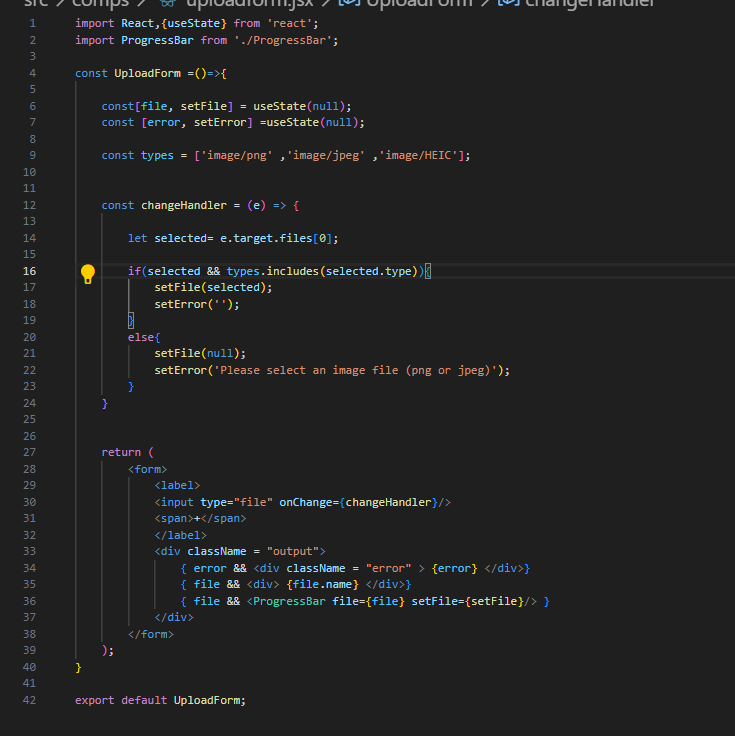
* **<Title/>:** Displays the title, logo, and a brief description of the gallery.
* **<UploadForm/>:** Allows users to select and upload images.
* **<ImageGrid/>:** Displays the uploaded images in a grid layout. It takes the setSelectedImg function as a prop to handle image selection.
* **<Modal/>:** Displays an enlarged view of selected images in a modal when a user clicks on an image in the grid. It also takes the setSelectedImg function as a prop to manage the modal's state.



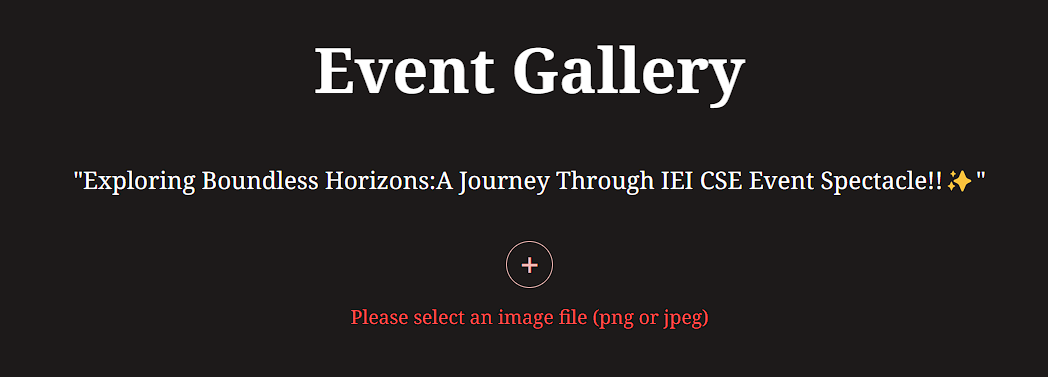
## 2. UploadForm.jsx

### Description:

### The UploadForm component is a React component responsible for creating an image upload form within your application. It allows users to select and upload image files, provides feedback on the selected file, and uses the ProgressBar component to show the upload progress. It also handles file type validation.

****

* The UploadForm component is a functional React component that provides an image upload form.
* It uses the useState hook to manage the state of the selected file (file) and any potential errors (error).
* The component defines an array of allowed file types (types) for validation, which includes 'image/png,' 'image/jpeg,' and 'image/HEIC.'
* The changeHandler function is triggered when a file is selected. It checks if the selected file is of an allowed type and updates the file state accordingly. If an invalid file type is selected, it sets an error message.
* The form includes an input field for selecting a file and displays a '+' symbol to indicate file selection. It also provides a visual indication of errors and the selected file's name.
* If a valid file is selected, the component renders the ProgressBar component to show the upload progress. The file and setFile props are passed to the ProgressBar component for managing the upload process

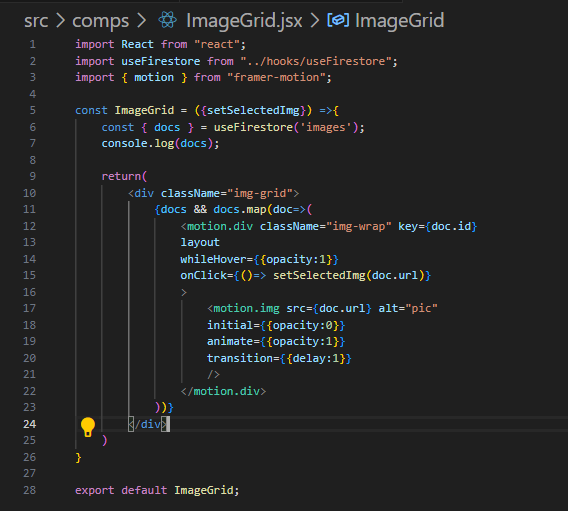


## ImageGrid.jsx

### Description:

### The ImageGrid component is designed to display images in a grid layout within your Event Gallery application.

### It uses the useFirestore hook to fetch image data from the 'images' collection in your Firestore database.



* The **ImageGrid** component is a functional React component that takes a prop called **setSelectedImg** to handle image selection for the modal.
* It uses the **useFirestore** hook to retrieve image data. The hook takes a collection name, in this case, 'images,' to fetch image information from the Firestore database.
* Inside the component, a grid layout is created to display images. It maps through the retrieved **docs** (documents) and renders each image.
* For each image, it creates a **motion.div** with animations enabled by the Framer Motion library. The **whileHover** animation provides a visual effect when hovering over the image, and the image's opacity is animated with a delay.



* When an image is clicked, the **setSelectedImg** function is called with the image's URL as a parameter to open the image in the modal.

## Title.jsx

### Description:

### The Title component is a React component responsible for rendering the title and header of your application. It includes an image, main title, subtitle, and a brief description. This component contributes to the visual identity of your application.

### 

### The Title component is a functional React component that renders the title and header section of your application.

### It includes an image with the class name "logo," which displays the application's logo. The src attribute is set to the logo.png image.

### The component displays a main title, "IE(I) CSE Student chapter, Punjab," using the <h1> HTML element. This represents the name or branding of your application.

### A subtitle, "Event Gallery," is displayed using the <h2> HTML element, providing context for the application's purpose.

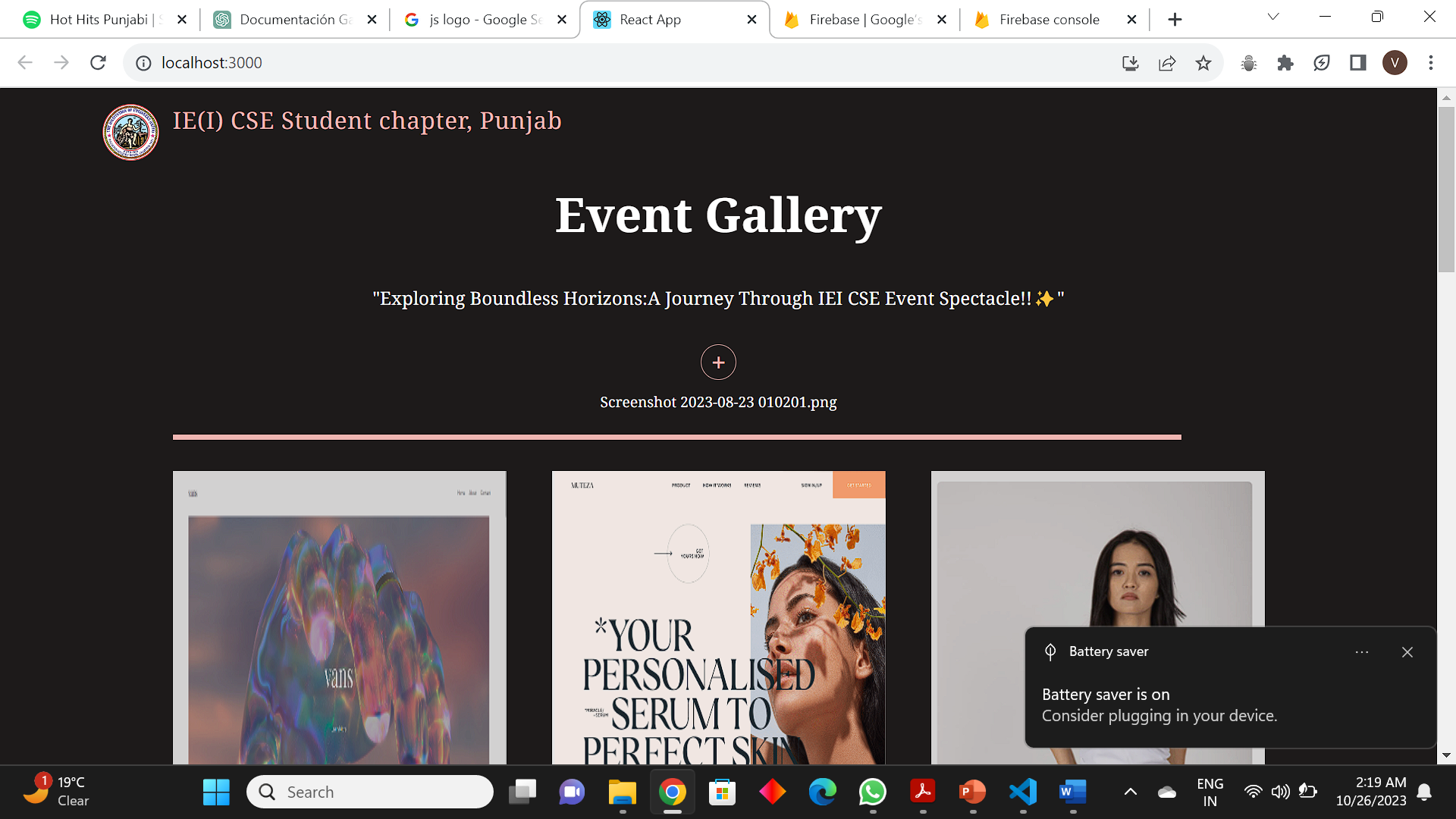
### A brief description enclosed in <p> tags adds additional information to the header section, enhancing the user's understanding of the application's purpose.

### 

## ProgressBar.jsx

**Description:**

The ProgressBar component is a React component responsible for displaying a progress bar during the image upload process. It visualizes the progress of image uploads using animations from the Framer Motion library. The component receives file information and upload progress data through props and manages the file upload process using a custom useStorage hook.





* The **ProgressBar** component is a functional React component that visualizes the progress of image uploads during the upload process.
* It receives two props: **file** and **setFile**. The **file** prop contains information about the file being uploaded, and **setFile** is a function used to manage the file state.
* Within the component, it uses the **useStorage** hook to manage the upload process. The **useStorage** hook is responsible for uploading the image and tracking its progress. The **progress** and **url** values are retrieved from the hook.
* The component logs the **progress** and **url** to the console, providing transparency into the upload progress and the URL of the uploaded image.
* An **useEffect** hook is used to watch for changes in the **url** and, when the upload is complete, set the file state to **null**.
* The progress bar is rendered as a **motion.div** element with animations from the Framer Motion library. The **width** property is animated from 0% to the current **progress** value, visualizing the upload progress.

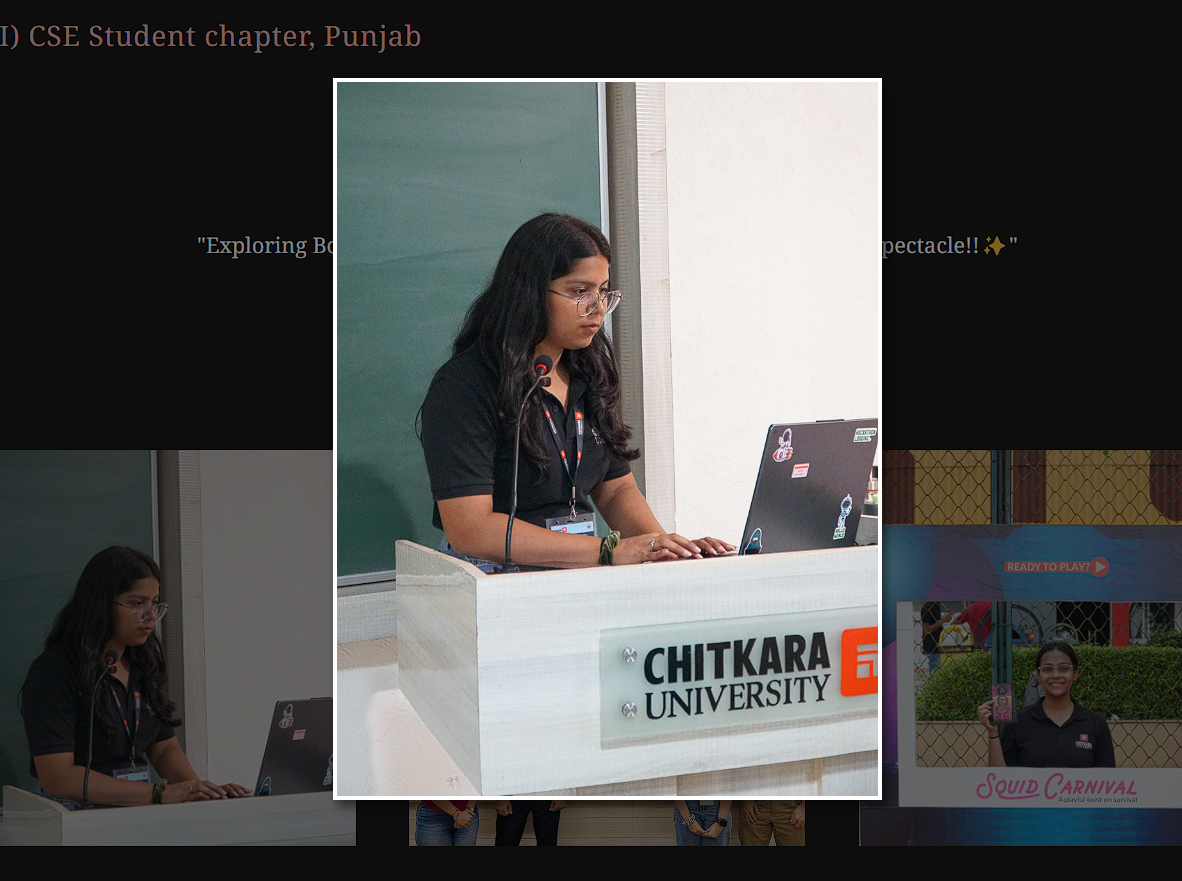
## Modal.jsx

**Description:**

The **Modal** component is a React component responsible for displaying an enlarged view of selected images in a modal. It provides a visual overlay effect and allows users to close the modal by clicking outside the image. The component utilizes animations from the Framer Motion library to create smooth transitions and interactions.



* The **Modal** component is a functional React component that displays an enlarged view of selected images in a modal format.
* It receives two props: **selectedImg** and **setSelectedImg**. The **selectedImg** prop represents the URL of the image to be displayed in the modal, and **setSelectedImg** is a function to manage the state of the selected image.

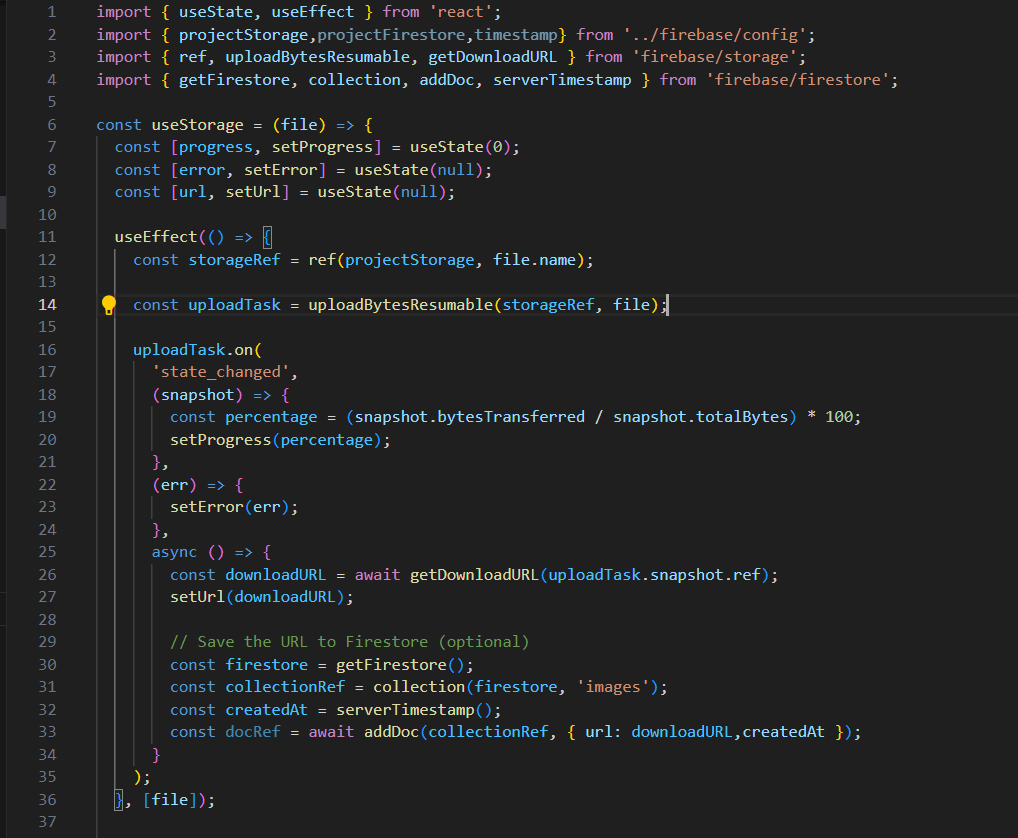


* The component listens for clicks on the "backdrop" element, which is used to close the modal. When a click occurs outside the image, the **handleclick** function is triggered. It checks if the clicked element has the class "backdrop" and, if so, calls **setSelectedImg(null)** to close the modal.
* The modal is rendered as a **motion.div** element with the class "backdrop." It includes animations for opacity transitions, making the modal appear smoothly. The enlarged image is rendered inside the modal as a **motion.img** element, allowing for a smooth vertical animation when it becomes visible.

## useStorage.jsx

**Description:**

The **useStorage** hook is a custom React hook designed to manage the process of uploading images to Firebase Storage and optionally saving their download URLs to Firebase Firestore. This hook provides feedback on the upload progress, any errors that occur during the upload, and the URL of the uploaded image.

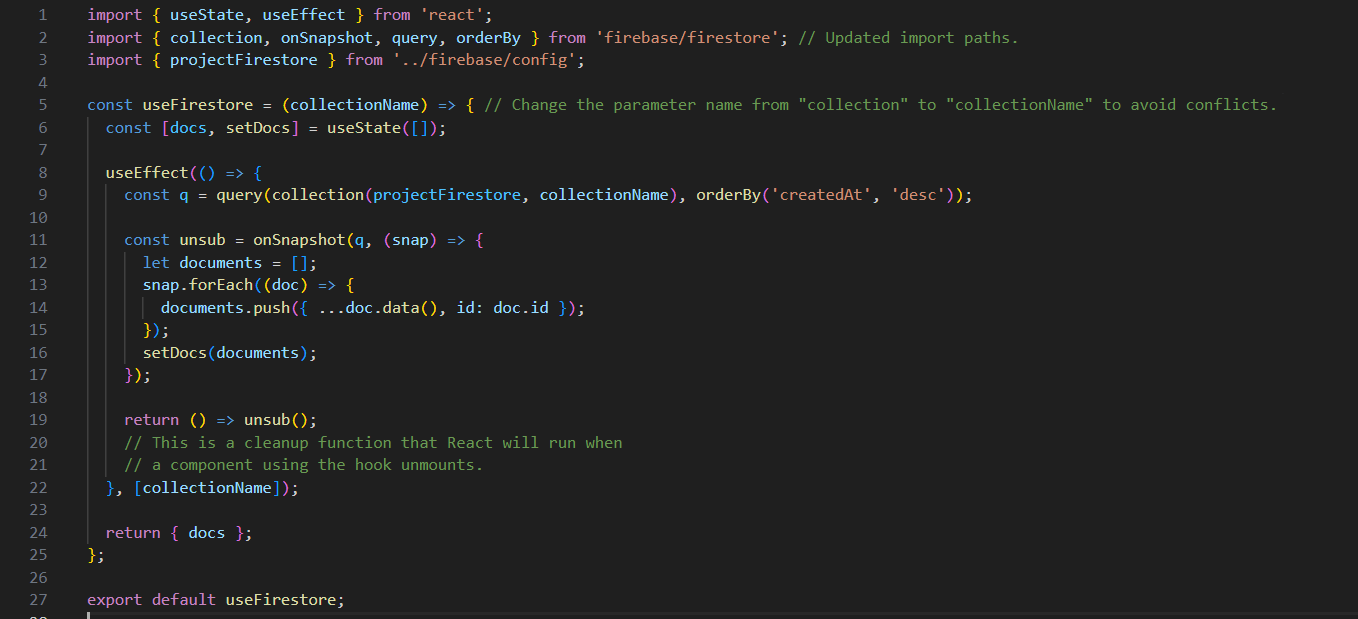


* The **useStorage** hook is a custom hook that takes the **file** as an argument, which represents the image to be uploaded.
* It defines three pieces of state:
  + **progress**: Keeps track of the upload progress as a percentage.
  + **error**: Stores any errors that might occur during the upload.
  + **url**: Stores the URL of the uploaded image.
* Inside the **useEffect**, the hook initiates the upload process by creating a reference to the image in Firebase Storage using **ref(projectStorage, file.name)**.
* It then uses **uploadBytesResumable** to create an upload task with the storage reference and the selected file.
* The **uploadTask** object listens to the 'state\_changed' event, tracking the progress of the upload and updating the **progress** state accordingly.
* If an error occurs during the upload, the **error** state is set to the error message.
* Once the upload is complete, it retrieves the download URL of the uploaded image using **getDownloadURL** and sets the **url** state with this URL.
* Optionally, it can save the download URL and a timestamp to Firebase Firestore. This involves creating a Firestore collection, generating a timestamp using **serverTimestamp**, and using **addDoc** to add a document with the URL and timestamp to the collection.

## useFirestore.jsx

**Description:**

The **useFirestore** hook is a custom React hook designed to fetch and listen to changes in a Firebase Firestore collection. It retrieves documents from a specified Firestore collection, orders them by a specified field, and provides a real-time updated list of documents to the consuming component.



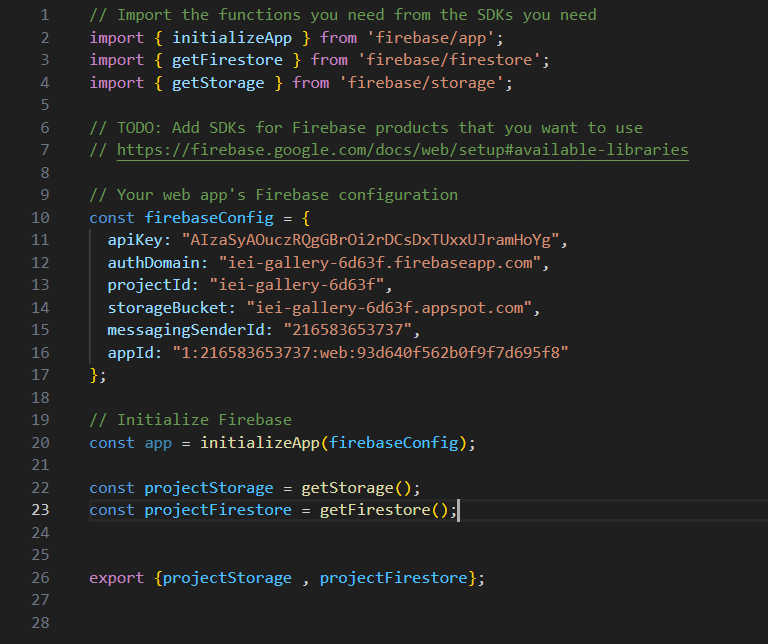
* The **useFirestore** hook is a custom hook that takes **collectionName** as its argument. This name represents the Firestore collection from which you want to retrieve documents.
* It defines a piece of state, **docs**, which will store the documents from the Firestore collection.
* Inside the **useEffect** hook, a query is created by combining the Firestore collection reference from **projectFirestore**, the specified **collectionName**, and an **orderBy** clause to sort documents by the 'createdAt' field in descending order.
* The **onSnapshot** function is used to listen to changes in the specified query. When the query results change, it triggers a callback function. Inside the callback, documents are extracted from the snapshot, and their data and unique IDs are added to the **documents** array.
* The **setDocs** function is then used to update the **docs** state with the new array of documents, providing real-time updates to the consuming component.
* The **return** statement exposes the **docs** state, which contains the list of documents from the Firestore collection.

Top of Form

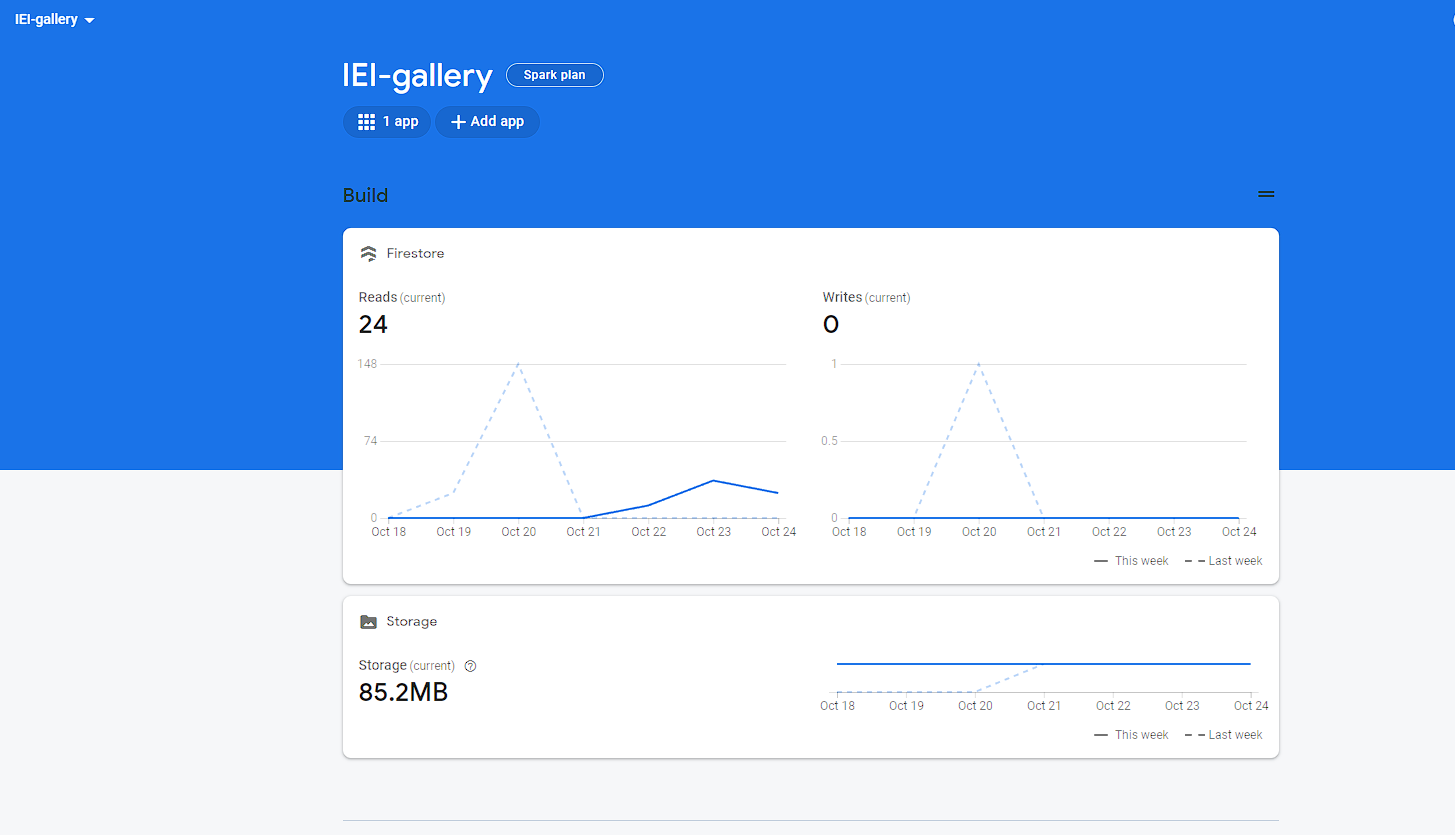
## Config.jsx

**Description:**

It is responsible for configuring and initializing Firebase for use in a web application. It sets up Firebase services such as Firestore and Storage and exports the references to these services for use in other parts of the application.



* The code snippet starts by importing the necessary functions from Firebase SDKs: **initializeApp** for initializing the Firebase app, **getFirestore** for Firestore, and **getStorage** for Firebase Storage.
* It defines the Firebase configuration object **firebaseConfig**, which contains the necessary configuration values. These values include your Firebase project's API key, authentication domain, project ID, storage bucket, messaging sender ID, and app ID.
* The **initializeApp** function is called with the **firebaseConfig** object to initialize the Firebase app. This should typically be done when the application starts to set up the Firebase environment.



* After initializing the app, it sets up references to Firebase services:
* **projectStorage** is a reference to Firebase Storage and can be used to interact with and store files in Firebase Storage.
* **projectFirestore** is a reference to Firebase Firestore, which allows you to interact with a NoSQL database in Firebase.
* Finally, the **projectStorage** and **projectFirestore** references are exported for use in other parts of the application.