< Back

Multimedia Application

Web Development with React I

Availability

Available

Quest master

StackUp

Est. time

1 hour(s)

Your status

In Progress

Reward

$5

No. of rewards

600

Start date

13 Jun 2023, 12:00 pm

Deadline

24 Jun 2023, 12:00 pm

11 days left

Review by

04 Jul 2023, 12:00 pm

Created

04 Jun 2023, 9:35 pm

Submit quest

Introduction

With the rise of technology, almost any task and workflow can be made simple with a Web Application. For example, communication is made simple with the improvements in technology. It is important to know how to display various types of content on the web application.

Recall in Quest 1 of this campaign we learnt how to display images. In this quest, we will take it a step further by creating a local file manager. In this file manager, we want to display the various types of media on our web application. This is similar to platforms like Google Drive, Microsoft OneDrive and Apple’s iCloud.

A file manager is a software application that allows users to manage files and folders stored on their computer or other devices such as external hard drives, USB flash drives, and networked storage.

Most operating systems come with a built-in file manager, such as Windows Explorer on Windows, Finder on macOS, and Nautilus on Linux. There are also third-party file managers available that offer additional features and functionality beyond what the built-in options provide.

To get an idea of what you'll be building, you can watch this video here:

Without further ado, let us start creating our file manager.

Learning Outcomes

By the end of this quest, you will be able to:

- Display PDF documents, display video and images, as well as play audio.
- Create a chart with React to see the most types of assets within your file server
- Rename and delete files from the local file directory

Deliverable

This quest has **two** deliverables. See the last step for details.

1. Link to Firebase Project
2. Screenshot of completed project

Questing solo?

Discuss about this quest with your fellow Stackies on Discord! They're a friendly bunch and always down to lend a hand.

Take me there!

Not on Discord?

Quest Walkthrough

Step 1: Project Setup

First, let's create a new directory for our React project. Open the directory with a code editor of your choice. The directory you have opened with your code editor can be any directory within your computer file system. But we highly recommend you create a folder to store your React projects as it would be easier to reference in the future! Open your built-in terminal and run the following commands line by line.

```
npx create-react-app multimedia-app cd multimedia-app npm
install react-chartjs-2 npm run start
```

With the first command, we created a React app named **multimedia-app**. The second changed your directory to the folder created by React. The third command downloaded and installed the react-chartjs-2 package which we will need in this quest. This package allows us to draw beautiful graphs and charts on our React Web Application.

The last command opens a local development server such that you will be able to see the changes you have made to your React Application live!

Now, let's create some new folders to organize the files that we will be working with. Firstly, in the ***src*** folder, create a folder called ***components***. In the ***components*** folder, we will create files that can be used and imported to other files within our React Application.

Step 2: Custom Header

Now, let us make a simple header component. We have done this before in other quests. This should be familiar to you by now. Within our ***components*** folder, create a file called **Header.js**. For this header, we will draw 2 text elements within our <header> element.

Copy the code below into the **Header.js** file

```
export const Header = () => {   return (     <header
style={styles.header}>       <h1
style={styles.headerText}>stackup-username-initial’s Drive</h1>
<p style={styles.headerSubText}>A File Manager created by
stackup-username</p>     </header>   ); }   const styles = {
header: {     backgroundColor: '#333',     color: '#fff',
textAlign: 'center',     padding: '10px'   },   headerText: {
fontSize: '30px'   },   headerSubText: {     fontSize: '15px'
} }
```

The code above is how we will create the structure of our header. We will also use inline styling to style our text elements. styles is the object that contains the styling for our header. In line 1 you may notice that we are exporting the functional component, **Header**. The export keyword allows us to use this component in other JavaScript files we have created in the React project.

Important: You need to change “stackup-username” with your StackUp profile name in line 5. Also, add your initials to line 4 by replacing "stackup-username-initial". Initial's are the first letter representing your name. For example, if your name is stackie, your initial would be S.

Step 3: Generating Data

Since this is a local application, we will need to generate our own data. File managers like Google Drive will store the files in a storage drive and have the file details, like their path, name, and type stored on a database.

Within the **src** folder, create a file called **data.js**. Within this file, we will be storing our data for our file manager. In this quest, we will be learning how to display 4 types of media assets. They are videos, images, documents and audio.

Now, you will need to add your own files. In the ***public*** folder, create a new folder called ***file-server***. You will need to place at least one file for each of the following types here:

- Image File, e.g. .jpg and .png
- Video File, e.g. .mp4 and .mov
- Audio File, e.g. .mp3 and .aac
- Document File, e.g. .pdf

Next, open **data.js** file and copy the code in 'working code' found at the end of this step and paste it.

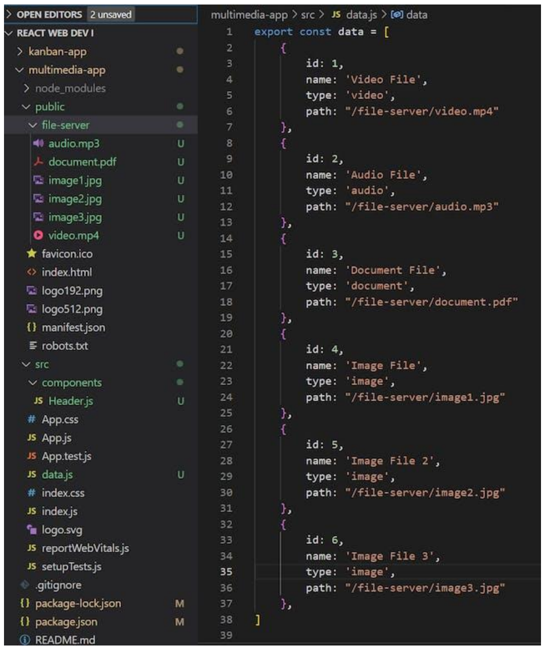
In the 'working code', data is an array object with objects within the array. Each object is a file entity that will contain the file metadata. For the property path of each object, the path will need to be in relative relation to the **public** folder within the React Application. The React Application will act as the storage drive where the actual file actually resides and the **data.js** file will be where the file data is saved at.

Important: As part of your deliverable, you need to source for 2 more files and update **data.js** accordingly. For example, you can have 2 more image files and add the 2 new images to the ***file-server*** folder. Once you have added the images to the folder, update the **data.js** file as well to reflect the addition of the 2 image file. In summary, you should have a total of 6 files in ***file-server***. You can take a look at the image in 'expected output' to have an idea of how to name your files.

Working code

```
export const data = [

  {

    id: 1,

    name: 'Video File',

    type: 'video',
    path: "/file-server/video.mp4"

  },

  {

    id: 2,

    name: 'Audio File',

    type: 'audio',

    path: "/file-server/audio.mp3"

  },

  {

    id: 3,
    name: 'Document File',

    type: 'document',

    path: "/file-server/document.pdf"

  },

  {

    id: 4,

    name: 'Image File',

    type: 'image',

    path: "/file-server/image.png"

  },
]
```

Expected output



```
export const data = [
    {
        id: 1,
        name: 'Video File',
        type: 'video',
        path: "/file-server/video.mp4"
    },
    {
        id: 2,
        name: 'Audio File',
        type: 'audio',
        path: "/file-server/audio.mp3"
    },
    {
        id: 3,
        name: 'Document File',
        type: 'document',
        path: "/file-server/document.pdf"
    },
    {
        id: 4,
        name: 'Image File',
        type: 'image',
        path: "/file-server/image1.jpg"
    },
    {
        id: 5,
        name: 'Image File 2',
        type: 'image',
        path: "/file-server/image2.jpg"
    },
    {
        id: 6,
        name: 'Image File 3',
        type: 'image',
        path: "/file-server/image3.jpg"
    },
]
```

Step 4: React Project Set Up

In this step, we will edit the **index.html** file within the ***public*** folder. This file is the base file that our React application will be built on when we run npm run build at the end of our development.

Open the **index.html** file. In line 27, change the element to “My File Manager”.

Next, we will open the **index.css** file within our ***src*** folder. Copy the code in 'working code' and add it to **index.css**. This is a CSS code snippet that applies some common styles to various HTML elements on a web page.

The first block of code applies the border-box value to the box-sizing property for all elements, including pseudo-elements like ::before and ::after. This means that any padding or border applied to an element will be included in its specified width and height, rather than adding extra space around it.

The second block of code sets the display property to block for various types of media elements, such as images, videos, and iframes. This ensures that these elements take up their full available width and height on the page and can be easily styled with CSS.

The third block of code sets the font property to inherit for form elements such as inputs, buttons, and text areas. This means that these elements will inherit the font settings of their parent element, rather than using their own default font.

The final block of code applies a few styles to all elements on the page. It sets the margin and padding properties to 0, which removes any default margins or padding that might be applied by the browser. It also applies the border-box value to the box-sizing property, just like the first block of code, and sets the font property to inherit. Overall, this block of code helps ensure a consistent default styling across all elements on the page.

Usually this is a good practice to add these properties to any web application as it will remove unexpected behaviour of HTML and CSS.

Working code

Step 5: App.js Structure

Now, we will work on the **App.js** file. This file will be the main entry file for our React Application. Copy the code in 'working code' found at the end of this step, and replace the code within **App.js** with the copied code.

In lines 1 and 2, we will import React, useState, and useEffect from the React library, as well as the **data.js** file. In line 3, we will also import the <Header/> component we created in step 2 to be used in **App.js** file. These dependencies are needed to create the App component that will be exported as the default.

In lines 6 to 9, we will define four different state variables using the useState hook, each with an initial value. myFiles is an array of files, selectedFile is the currently selected file, filePath is the URL selected for the file manager, and showChartModal is a boolean that determines whether to display a chart modal.

In line 26, we will use a .map() function where we will iterate through the **data.js** file. From there we will display the name property within our data file object. We will also set the parent <div> element to be clickable. We will add a onClick property to the <div> element and whenever the user clicks on the <div> element, it will trigger the code block. For this case, we want to set the state for selectedFile to store the file data that is clicked. In this click event, we also want to check if the current selectedFile is equal to the current item that is clicked. If so, we will set selectedFile state to null as we want to create a toggle.

Lastly, from line 43 to line 62, we want to display the actual file on our web application. In lines 45, 48, 51 and line 54, we conditionally check if the file type is of a certain value. This way of writing a conditional statement is also known as a ternary operator. You may notice that in lines 46, 49, 52 and line 55, we will show a custom component such as AudioPlayer which will be written in the next step. For example, if the selectedFile file type is an image, we will display the <ImageViewer> component.

Working code

Step 6: Custom Components (ImageViewer, DocumentViewer, AudioPlayer, VideoPlayer)

In this step, we will create custom components that will display our files onto our web application.

- To display images, we use the <img> element tags.
- To display PDF documents, we can use the <iframe> tag.
- To play audio files, we can use the <audio> and <source> tags
- To play video files, we use the <video> and <source> tags.

Video Component

Within the **components** folder, create a file named **VideoPlayer.js** and add the code below to the file.

```
export const VideoPlayer = ({ path }) => {   return (     <div
style={{ width: "100%" }}>       <video controls>
<source src={path} type="video/mp4" />       </video>     </div>
) }
```

Audio Component

Within the **components** folder, create a file named **AudioPlayer.js** and add the code below to the file.

```
export const AudioPlayer = ({ path }) => {   return (     <div
style={{width:"100%"}}>       <audio controls
style={{width:"100%"}}>         <source src={path}
type="audio/mpeg" />       </audio>     </div>   ) }
```

Document Component

Within the **components** folder, create a file called **DocumentViewer.js** and add the code below to the file.

```
export const DocumentViewer = ({ path }) => {   return (
<div style={{width:"100%"}}>       <iframe style={{width:"100%",
height: "300px"}} src={path}></iframe>     </div>   ) }
```

Image Component

Within the **components** folder, create a file called **ImageViewer.js** and add the code below to the file.

```
export const ImageViewer = ({ path }) => {   return (     <div
style={{width:"100%"}}>       <img src={path} />     </div>   )
}
```

For each component, we will take in a **path** as a prop. This is such that we can point the file location to our web application to display the file accordingly.

Lastly, open up **App.js** file and we will import the four custom components we have created earlier to our **App.js** file. Add the code below to line 4 of our **App.js** file.

```
import { AudioPlayer } from './components/AudioPlayer'; import {
DocumentViewer } from './components/DocumentViewer'; import {
VideoPlayer } from './components/VideoPlayer'; import {
ImageViewer } from './components/ImageViewer';
```

You can check the image in 'expected output' to see how your App.js file should look like now.

Expected output

Step 7: File Manager Tools

Now, we will add the features and buttons for our file manager. The four functions we want to create are

1. Rename files
2. Delete files
3. View the file type breakdown
4. Download the file

Copy the 'working code' found at the end of this step and insert it in line 28 of our **App.js** file. You can check the 'expected output' to see if you have pasted the code in the correct place.

This code will make use of an <div> element with three <button> as its children. The parent <div> will separate each button based on the gap and spacing.

Important: There are only 3 buttons here, we are missing the fourth operation and which is to Delete a file. As part of your deliverable, create another button that will delete the file from the file manager when the user clicks on the button. When the file is deleted, the file should be hidden from view. Do refer to the video in the introduction at the beginning of this quest to see how this function will work.

Working code

Expected output

Step 8: Drawing Charts (Pie Chart and Bar Graph)

In this step, we will be creating 2 charts – a Pie Chart and a Bar Graph. We will use the "react-chartjs-2" dependency to draw our graph.

Copy the code snippet below and paste it to line 8 in **App.js**, after the import statements.

```
import { Pie, Bar } from 'react-chartjs-2'; import {  Chart as
ChartJS,  CategoryScale,  LinearScale,  BarElement,  Title,
ArcElement,  Tooltip,  Legend } from 'chart.js';
ChartJS.register(  CategoryScale,  LinearScale,  BarElement,
Title,  Tooltip,  Legend,  ArcElement );
```

The code above imports two components from the 'react-chartjs-2' library, which are 'Pie' and 'Bar', used for rendering pie and bar charts respectively. In addition, it also imports several elements and scales from the 'chart.js' library, such as 'CategoryScale', 'LinearScale', 'BarElement', 'Title', 'ArcElement', 'Tooltip', and 'Legend'.

In the next few lines, it registers the imported elements with ChartJS by calling the ChartJS.register() method, passing all the elements as arguments. This is necessary to make these elements available to be used in the chart components that will be rendered by 'Pie' and 'Bar'.

Next, we will need to create the options for our bar chart. The options are like configurations to change how our bar chart looks and feel. Copy the code below to line 37 of **App.js**, just before the return statement.

```
var barChartOptions = {   responsive: true,   plugins: {
legend: {     position: 'top',    },    title: {     display:
true,     text: 'Files Breakdown',    },   },  };
```

Next, we are going to create the modal view for our files breakdown. In step 5, we created a state to keep track of the visibility of our modal. Now, we will implement the modal in this step.

Copy the code below to line 52 of **App.js** file, just after '<>'.

```
{showChartModal && (     <div style={styles.modal}>      <div
style={styles.modalContent}>       <div
```

```
style={styles.modalHeader}>        <p style={{ fontWeight:
"bold" }}>Files Breakdown</p>        <button
style={styles.closeButton} onClick={() =>
setShowChartModal(false)}>close</button>       </div>       <div
style={styles.modalBody}>        <Pie         data={{
labels: ['Video', 'Audio', 'Document', 'Image'],
datasets: [           {            label: 'Files Breakdown',
data: [myFiles.filter(file => file.type === 'video').length,
myFiles.filter(file => file.type === 'audio').length,
myFiles.filter(file => file.type === 'document').length,
myFiles.filter(file => file.type === 'image').length],
backgroundColor: [

            'rgba(255, 99, 132, 0.2)',

            'rgba(54, 162, 235, 0.2)',

            'rgba(255, 206, 86, 0.2)',

            'rgba(75, 192, 192, 0.2)',

           ],             borderColor: [

            'rgba(255, 99, 132, 1)',

            'rgba(54, 162, 235, 1)',
            'rgba(255, 206, 86, 1)',

            'rgba(75, 192, 192, 1)',

           ],            borderWidth: 1,          },
],          }}         />        <Bar         data={{
labels: ['Video', 'Audio', 'Document', 'Image'],
datasets: [           {            label: 'Files Breakdown',
data: [myFiles.filter(file => file.type === 'video').length,
myFiles.filter(file => file.type === 'audio').length,
myFiles.filter(file => file.type === 'document').length,
myFiles.filter(file => file.type === 'image').length],
backgroundColor: [

            'rgba(255, 99, 132, 0.2)',

            'rgba(54, 162, 235, 0.2)',

            'rgba(255, 206, 86, 0.2)',

            'rgba(75, 192, 192, 0.2)',

           ],            borderColor: [

            'rgba(255, 99, 132, 1)',

            'rgba(54, 162, 235, 1)',
            'rgba(255, 206, 86, 1)',

            'rgba(75, 192, 192, 1)',

           ],            borderWidth: 1,           },
],          }}         options={barChartOptions}        />
</div>      </div>     </div>    )}
```

This block of code is a conditional rendering that displays a modal containing two charts, pie and bar charts when the state variable showChartModal is true. In lines 53, the conditional statement checks if showChartModal is true and if so, it renders the modal component.

The modal is defined in lines 53-113, with its own styles defined using CSS in JS syntax. It consists of a modal header, a modal body containing two charts, and a close button.

The pie chart is defined in lines 61-84, using the Pie component from 'react-chartjs-2' library. The chart displays the breakdown of file types in an array of files called myFiles. It takes in two props, data and options. The data prop contains an object with two keys, labels and datasets. The labels key holds an array of strings representing each file type. The datasets key contains an array with one object, representing the data to be displayed in the chart. It holds an array of integers with the number of files for each file type, and arrays of colors for the background and border colors of each slice in the chart.

Similarly, the bar chart is defined in lines 85-109, using the Bar component from the same library. It also takes in two props, data and options. The data prop is structured similarly to the pie chart and holds the same data to be displayed in a bar chart format. The options prop holds an object with custom options for the chart, such as the axis labels, grid lines, and tooltips.

Lastly, add the code in 'working code' at the end of this step to line 246 of **App.js**. This code will be the styling of our modal. In Quest 1, you may recall that we also created a modal component to display the images and their metadata. We will use similar styling for this quest's modal container as well.

The styles for the modal and its components are defined using CSS in JS syntax in lines 247-297. The modal is styled to display in the center of the screen and to have a transparent black background. The modal content, header, and body are styled to have a white background and a border radius, with the header and body having different font sizes and margins. The close button is styled to have a red background and white text.

To check if you have copied the code correctly, you can check the 'expected output' at the end of this step. Do note that the full code of App.js is not shown. Instead, the parts where you have pasted code in this step are in red boxes.

Working code

Expected output

Step 9: Building Our Application

Congratulations on making it to this step. Once you are satisfied with your File Manager Application, it is time to build it for the web. What React will do is they will compile the code base across multiple JavaScript files into one HTML file.

To build the application, run the following command in your ***multimedia-app*** directory.

```
npm run build
```

This code will simply create a build of the application and place the build into a new folder called ***build***. Everything in this folder is the final product that can be now used in production.

In the next step, we will be looking to deploy our app on the Internet using a web host provider. There are two sets of instructions available. Step 9 will provide instructions on uploading to Google Firebase, while Step 10 will provide instructions on uploading to Netlify.

A reminder that for web development quest submissions, you are **free to host your static site on a web host provider of your choice**. However, you should deploy your site using command line interface tools, as deployment via terminal is a part of the screenshot requirement (see last step).

Step 10: Project Deployment to Google Firebase (Option 1)

We will be using Firebase hosting services to host our web application. In the event you run into any errors with Firebase, you may visit Step 4 of this tutorial page for some frequently encountered errors and their solutions.

Before we can upload our web application to Firebase, we will need to tell Firebase that this project is a Firebase project. If you have yet to install Firebase CLI tools, you are strongly advised to go through the Installation to Firebase quest.

Now, within your code editor, open the terminal and enter the following command.

```
firebase login
```

This will redirect us to our web browser. We will need to log in to our Google Account on the web browser as Firebase is a Google service. Behind the scenes, Google is trying to authenticate your account to grant access to your terminal. Upon successful login, you may return to your code editor. You should be able to see a success message on the terminal.

Next, we will need to initialise this project to our Firebase account. Enter this command to do so.

```
firebase init
```

We need to select the “**Hosting: Configure files for Firebase Hosting and …**” We will need to use our arrow keys on our keyboard to interact with the CLI. Once you have moved down your arrow onto the row you want, press “SPACEBAR” on your keyboard to select and “ENTER/RETURN” to continue.

Next, move your arrow keys to the row “**Create a new project**” as we are going to create a new project. Press “ENTER/RETURN” to confirm your selection.

You are now required to provide a unique project id. This project id used for your project website domain. Give your project id the name* in this naming convention: “**Stackupusername-multimedia-app**”. **Replace stackupusername with your Stackup Profile Name.** In the event that the project id you wish to give is unavailable, do add numbers to your StackUp name to make the id unique. For example, instead of <stackup username>-multimedia-app, you can give an id of stackie823-multimedia-app.

***On naming your project id**: It must be 6 to 30 lowercase letters, digits or hyphens. It must start with a letter. If your StackUp profile name does not meet this requirement, you should ensure that the stackupname used in your project id is like your StackUp profile name. Otherwise, you will receive an error.

Next, you will be asked for the project directory to look for. This directory is where Firebase will upload all the files to the cloud. When we built our application, React automatically created a **build** folder for us. In your terminal, type in **build** and right after, hit the “ENTER/RETURN” to confirm the folder.

Important: Now Firebase is asking if you want them to rewrite all files into a single-page app. Hit “**N**” on your keyboard. If we press “Y”, it will overwrite all the code that we have in the “index.html” file.

You may choose to upload your code here to GitHub as well. For this quest, we will not need to do so, as such, select “**N**”.

When Firebase asks to overwrite the index.html file that resides within the “build” folder, hit “**N**” as we want to retain the current **index.html** file that we have written.

Once you see the statement 'Firebase initialization complete!', we have completed setting up our project on Firebase. We may now move on to deploying our code.

In your terminal, input the following command.

```
firebase deploy
```

We have now deployed the code to the internet. Firebase will also return the default link to access the project and you may use that “Hosting URL” to share with others.

Step 11: Project Deployment to Netlify (Option 2)

Firstly, we will need to install the Netlify CLI tool using the following command in your terminal:

```
npm install netlify-cli -g
```

You will also need to create an account on Netlify. Thereafter, ensure that you are logged in to Netlify on your browser.

Next, we will obtain an access token so that you can deploy your project from your terminal. We will use the following command in your terminal:

```
netlify login
```

A new browser tab or window will be opened asking you to authorize Netlify CLI to access Netlify on your behalf. Click on the Authorize button. In the event that the browser tab or window is not opened, you can also copy the URL in your terminal and input it in your browser. Once authorization has been granted, your terminal should now have the message "You are now logged into your Netlify account!"

Next, in your terminal, enter

```
netlify deploy
```

You will come across the following prompts and these are the responses you should make:

- What would you like to do?: **Create & configure a new site**
- Team: **<choose the default option>**
- Site name: **<yourusername>-multimedia-app** (ensure you replace <yourusername> with your StackUp username)

On asked to provide a publish directory, enter **build**. Once the files have been uploaded, you will be provided with a "Website Draft URL". Go to that website and check that your website works as intended.

If everything looks good, use the following command:

```
netlify deploy --prod
```

Once again, you will be asked for the publish directory. Use **build**.

When you see the message "Deploy is live!" and your website URL is provided, you know that your site has been successfully deployed. Visit your website again and ensure that everything works well.

Step 12: Let’s Ace Your Submissions! Preparing Your Submission!

You have reached the end! Now to successfully complete this quest! There are **2 deliverables** that are required for this quest – a screenshot and a web link.

But before that, let's do a check to ensure you have made all the necessary changes to the website.

1. In Step 2, you should have added your StackUp profile name and initials to the Header Component
2. In Step 4, you need to have 1 file of each file type (video, image, document (pdf) and audio) within the **public/file-server** folder. You will also need to add 2 more files of any file type to **data.js** file
3. In Step 7, you completed the code for the “Delete” button. The button code should work. When clicked, the deleted file should not be shown on the Web Application.

Screenshot

Take a screenshot of your code editor. Your screenshot should show:

- your full screen, including your taskbar (for Windows and Linux) / dock (for MacOS)
- The file-server folder expanded
- The portion of your data.js code which shows the additional two files you sourced
- Your terminal (either in-built or a separate window) with the output showing that multimedia-app has been deployed, with the hosting URL shown

When labelling your screenshot, make sure to follow the format provided: C71_Q3_yourstackupname.png.

Web Link

In this quest, you created a website and hosted it on either Firebase or Netlify (or a web hosting service provider of your choice). A sample of how your website should look is shown in 'Your website' of 'expected output'.

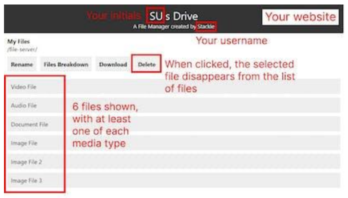
In the description box when you click 'Submit Quest', submit your website's link in the following format:

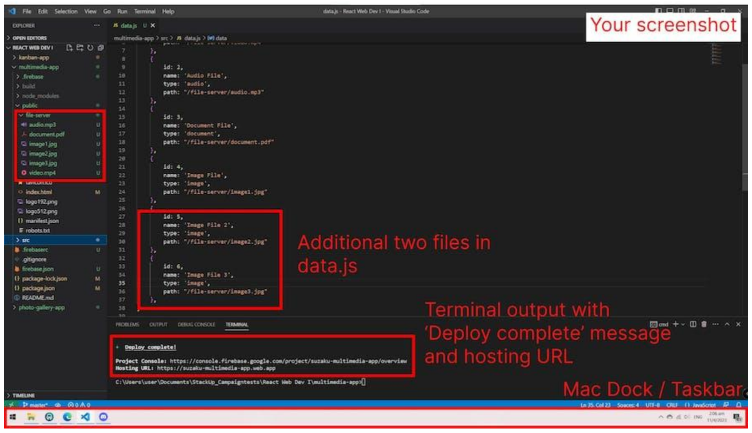
```
[u][url=https://yourwebsitehere.com]Multimedia App[/url][/u]
```

Note: For Firebase users, in the event that your website hosted on “.web.app” is not working well, you can also submit the URL with the .firebaseapp.com domain, e.g. https://stackupname-multimedia-app.firebaseapp.com

Your submission will automatically be rejected if you do not follow the BBCode format listed above. If you are uncertain if your BBCode format is correct, you can test it out on a BBCode viewer. The output should show the words 'Multimedia App' underlined, and hyperlinked to your site.

Expected output




data.js - React Web Dev I - Visual Studio Code
Your screenshot
multimedia-app > src > data.js > data
file-server
audio.mp3
document.pdf
image1.jpg
image2.jpg
image3.jpg
video.mp4
Additional two files in data.js
Terminal output with 'Deploy complete' message and hosting URL
Mac Dock / Taskbar

```
    },
    {
        id: 2,
        name: 'Audio File',
        type: 'audio',
        path: "/file-server/audio.mp3"
    },
    {
        id: 3,
        name: 'Document File',
        type: 'document',
        path: "/file-server/document.pdf"
    },
    {
        id: 4,
        name: 'Image File',
        type: 'image',
        path: "/file-server/image1.jpg"
    },
    {
        id: 5,
        name: 'Image File 2',
        type: 'image',
        path: "/file-server/image2.jpg"
    },
    {
        id: 6,
        name: 'Image File 3',
        type: 'image',
        path: "/file-server/image3.jpg"
    },
```

```
+ Deploy complete!

Project Console: https://console.firebase.google.com/project/suzaku-multimedia-app/overview
Hosting URL: https://suzaku-multimedia-app.web.app

C:\Users\user\Documents\StackUp_Campaigntests\React Web Dev I\multimedia-app>
```

Back to Campaign

Web Development with React I

binong

Current: US$138.00

Timezone: Asia/Manila

Home

Profile

Earn

Campaigns

Quests

Bounties

My Progress

Learn

Calendar

Help

T&Cs