Functions

Contents

Select Video: 3

Claude Sonnet: 6

Record video: 9

Claude Sonnet: 13

Compress Video: 16

Claude Sonnet: 21

Upload video: 25

Claude Sonnet: 29

Create Assignment: 33

Claude Sonnet: 38

View Assignment: 43

Claude Sonnet: 47

View Submissions: (Claude 3.5 - Sonnet) 51

Browse your own submissions: 54

View feedback on your own submissions: 55

Stream video submissions: 58

Download video submissions: 62

Provide feedback on video: 66

Download Marks: 70

Secure Login: 74

User Admin: 78

Select Video:

write React Native code for selecting a video.

Input: User interacts with a button

Processes: File selection browser opened, file type verified, file path captured

Output if successful: Success message, file path stored, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';

function VideoSelector() {
  const [filePath, setFilePath] = useState('');
  const [message, setMessage] = useState('');
  const [logFile, setLogFile] = useState([]);

  const validVideoTypes = ['video/mp4', 'video/webm', 'video/ogg'];

  const handleFileChange = (event) => {
    const file = event.target.files[0];

    if (file) {
      if (validVideoTypes.includes(file.type)) {
        const path = URL.createObjectURL(file);
        setFilePath(path);
        setMessage('Success: Video file selected successfully.');
        addLogEntry('Success: Video file selected.', path);
      } else {
        setFilePath('');
        setMessage('Failure: Invalid video file type.');
        addLogEntry('Failure: Invalid video file type selected.');
      }
    } else {
      setMessage('Failure: No file selected.');
      addLogEntry('Failure: No file selected.');
    }
  };

  const addLogEntry = (message, path = '') => {
    const timestamp = new Date().toLocaleString();
    setLogFile((prevLog) => [
      ...prevLog,
      `${timestamp} - ${message} ${path ? `File Path: ${path}` : ''}`
    ]);
  };

  const handleButtonClick = () => {
    document.getElementById('videoInput').click();
  };

  return (
    <div>
      <button onClick={handleButtonClick}>
        Select a Video File
      </button>
      <input
        type="file"
        id="videoInput"
        accept="video/*"
        style={{ display: 'none' }}
        onChange={handleFileChange}
      />
      {message && (
        <div style={{ marginTop: '10px', color: message.startsWith('Success') ? 'green' : 'red' }}>
          {message}
        </div>
      )}
      {filePath && (
        <div style={{ marginTop: '10px' }}>
          <strong>File Path:</strong> {filePath}
        </div>
      )}
      <div style={{ marginTop: '20px' }}>
        <h3>Log Entries:</h3>
        <ul>
          {logFile.map((log, index) => (
            <li key={index}>{log}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default VideoSelector;
```

Explanation:

1. **State Variables**:
    - filePath: Stores the path of the selected video file.

    - message: Stores the success or failure message.
    - logFile: Maintains a list of log entries.
2. **File Handling**:
    - The handleFileChange function is triggered when the user selects a file. It checks if the file type is valid (i.e., one of the accepted video formats). If valid, it stores the file path, displays a success message, and logs the action. If not valid, it clears the file path, displays a failure message, and logs the failure.
3. **Log Entry**:
    - addLogEntry adds a new log entry with a timestamp to the logFile array, capturing either success or failure messages along with the file path if applicable.
4. **Button Interaction**:
    - The handleButtonClick function triggers a click event on a hidden file input element when the user clicks the "Select a Video File" button. This opens the file selection dialog.
5. **Rendering**:
    - The button is displayed for the user to initiate video file selection.
    - Messages indicating success or failure are conditionally rendered based on the file selection outcome.
    - The selected file path is displayed if a valid file is chosen.
    - A log of all actions is displayed as a list below the button.

Usage:

- Add the <VideoSelector /> component to your React application.
- When the user clicks the button, the file selection dialog opens.
- After selecting a video file, the component will validate the file and display the appropriate message.
- The log section will display a timestamped record of all actions taken.

This component is a basic implementation but can be expanded to include additional features like file previews, actual file uploads, or more sophisticated logging mechanisms.

Claude Sonnet:

```
import React, { useState } from 'react';
import { View, Text, Button, StyleSheet, Alert } from 'react-native';
import * as DocumentPicker from 'expo-document-picker';
import * as FileSystem from 'expo-file-system';
import AsyncStorage from '@react-native-async-storage/async-storage';

const VideoSelectionScreen = () => {
  const [selectedVideo, setSelectedVideo] = useState(null);

  const selectVideo = async () => {
    try {
      console.log('Opening file selection browser...');

      const result = await DocumentPicker.getDocumentAsync({
        type: 'video/*',
        copyToCacheDirectory: false,
      });

      if (result.type === 'success') {
        console.log('File selected:', result.uri);

        // Verify file type
        const fileInfo = await FileSystem.getInfoAsync(result.uri);
        const fileType = result.mimeType || fileInfo.mimeType;

        if (!fileType || !fileType.startsWith('video/')) {
          throw new Error('Selected file is not a video');
        }

        // Store file path
        await AsyncStorage.setItem('selectedVideoPath', result.uri);

        setSelectedVideo(result);
        console.log('File path stored successfully');

        Alert.alert('Success', 'Video selected successfully!');
      } else {
        console.log('File selection cancelled');
      }
    } catch (error) {
      console.error('Error selecting video:', error);
      Alert.alert('Error', 'Failed to select video. Please try again.');
    }
  };
  return (
    <View style={styles.container}>
      <Button title="Select Video" onPress={selectVideo} />
      {selectedVideo && (
        <Text style={styles.fileInfo}>
          Selected: {selectedVideo.name}
        </Text>
      )}
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  fileInfo: {
    marginTop: 20,
    fontSize: 16,
  },
});

export default VideoSelectionScreen;
```

This code creates a React Native component called VideoSelectionScreen that allows users to select a video file. Here's a breakdown of the implementation:

1. The component uses React hooks to manage the state of the selected video.
2. It uses expo-document-picker for file selection, expo-file-system for file information, and @react-native-async-storage/async-storage for storing the file path. Make sure to install these libraries:

```
expo install expo-document-picker expo-file-system @react-native-async-storage/async-storage
```

3. The selectVideo function handles the video selection process:
    - It opens the document picker with a filter for video files.
    - It verifies that the selected file is indeed a video by checking its MIME type.
    - It stores the file path in AsyncStorage.
    - It updates the component state with the selected video information.
4. Error handling is implemented to show error messages if the selection fails or if a non-video file is selected.

5. The component displays the name of the selected video once it's chosen.
6. Console logs are used to record operations and errors, which you can expand into more comprehensive logging.

To use this component, you'll need to:

1. Install the necessary dependencies:

```
expo install expo-document-picker expo-file-system @react-native-async-storage/async-storage
```

2. Implement proper error handling and set up comprehensive logging for production use.
3. Use this component in your app:

```
<VideoSelectionScreen />
```

Remember to consider the following:

- Implement file size limits to prevent users from selecting excessively large videos.
- Add support for multiple video selection if needed.
- Consider adding a preview functionality for the selected video.
- Implement a way to clear the selected video if needed.
- Handle cases where the user might not have granted file access permissions to your app.

Also, be mindful of potential issues:

- On iOS, the DocumentPicker might not work in the iOS Simulator. You'll need to test on a real device.
- The file path stored in AsyncStorage might not be persistent across app restarts on all platforms. Consider implementing a more robust storage solution for long-term storage of selected files.
- Large video files might cause performance issues. Consider implementing file compression or cloud upload for large files.

Lastly, you might want to add features like:

- Displaying more detailed information about the selected video (e.g., duration, resolution).
- Allowing users to rename the selected video file.
- Implementing a video player to preview the selected video.

Remember to thoroughly test this functionality on various devices and platforms to ensure consistent behavior.

Record video:

write React Native code for recording a video.

Input: User interacts with a button

Processes: camera opened, video recorded, video file saved, file path captured

Output if successful: Success message, file path stored, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState, useRef } from 'react';

function VideoRecorder() {
  const [recording, setRecording] = useState(false);
  const [videoFile, setVideoFile] = useState(null);
  const [filePath, setFilePath] = useState('');
  const [message, setMessage] = useState('');
  const [logFile, setLogFile] = useState([]);
  const videoRef = useRef(null);
  const mediaRecorderRef = useRef(null);
  const recordedChunks = useRef([]);

  const addLogEntry = (message, path = '') => {
    const timestamp = new Date().toLocaleString();
    setLogFile((prevLog) => [
      ...prevLog,
      `${timestamp} - ${message} ${path ? `File Path: ${path}` : ''}`
    ]);
  };

  const startRecording = async () => {
    try {
      const stream = await navigator.mediaDevices.getUserMedia({ video: true
});
      videoRef.current.srcObject = stream;
      videoRef.current.play();

      mediaRecorderRef.current = new MediaRecorder(stream);
      mediaRecorderRef.current.ondataavailable = (event) => {
        if (event.data.size > 0) {
          recordedChunks.current.push(event.data);
        }
      };
      mediaRecorderRef.current.onstop = () => {
        const blob = new Blob(recordedChunks.current, { type: 'video/webm' });
        const url = URL.createObjectURL(blob);
        setVideoFile(blob);
        setFilePath(url);
        setMessage('Success: Video recorded successfully.');
        addLogEntry('Success: Video recorded.', url);

        // Stop the camera stream
        stream.getTracks().forEach(track => track.stop());
      };

      mediaRecorderRef.current.start();
      setRecording(true);
      setMessage('Recording started...');
      addLogEntry('Recording started.');
    } catch (error) {
      setMessage('Failure: Unable to access camera.');
      addLogEntry('Failure: Unable to access camera.');
    }
  };

  const stopRecording = () => {
    mediaRecorderRef.current.stop();
    setRecording(false);
    setMessage('Recording stopped.');
    addLogEntry('Recording stopped.');
  };

  const handleButtonClick = () => {
    if (recording) {
      stopRecording();
    } else {
      startRecording();
    }
  };

  return (
    <div>
      <button onClick={handleButtonClick}>
        {recording ? 'Stop Recording' : 'Start Recording'}
      </button>
      <div style={{ marginTop: '20px' }}>
        <video ref={videoRef} style={{ width: '100%', maxWidth: '400px' }}
controls muted />
      </div>
      {message && (
        <div style={{ marginTop: '10px', color: message.startsWith('Success')
? 'green' : 'red' }}>
          {message}
        </div>
      )}
      {filePath && (
        <div style={{ marginTop: '10px' }}>
          <strong>File Path:</strong> {filePath}
        </div>
      )}
      <div style={{ marginTop: '20px' }}>
        <h3>Log Entries:</h3>
        <ul>
          {logFile.map((log, index) => (
            <li key={index}>{log}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default VideoRecorder;
```

Explanation:

1. **State and Refs**:
    - recording: Tracks whether recording is in progress.
    - videoFile: Stores the recorded video file as a Blob.
    - filePath: Stores the URL of the recorded video file.
    - message: Stores the success or failure message for the user.
    - logFile: Tracks a history of log entries.
    - videoRef: A ref to the video element used for showing the camera stream.
    - mediaRecorderRef: A ref to the MediaRecorder instance.
    - recordedChunks: A ref to store the chunks of data captured during recording.
2. **Logging**:
    - addLogEntry: Adds a new entry to the log with a timestamp, message, and optional file path.
3. **Recording Process**:
    - startRecording:
        - Requests video access from the user's camera.

  - Starts the MediaRecorder and begins capturing video.
  - Adds any recorded data chunks to recordedChunks.
  - On stopping, a Blob is created from the recorded chunks, and a URL is generated for it.
  - Stops the camera stream after recording is done.
- stopRecording: Stops the MediaRecorder and finalizes the recording.

4. **Button Click Handling**:
   - handleButtonClick: Toggles between starting and stopping the recording based on the current state.
5. **Rendering**:
   - The video element is used to display the camera feed during recording.
   - Messages indicating success or failure are shown below the video.
   - The log entries are displayed as a list.

Usage:

- Add the <VideoRecorder /> component to your React application.
- When the user clicks the "Start Recording" button, the camera is accessed, and recording begins.
- Clicking "Stop Recording" stops the recording, saves the video, and logs the actions.
- The video file path, success/failure messages, and log entries are displayed on the screen.

This component is a comprehensive solution for recording video directly in a React app, handling edge cases like camera access failure, and providing detailed logs of the process.

Claude Sonnet:

```
import React, { useState, useRef } from 'react';
import { View, Button, Text, Alert } from 'react-native';
import { Camera } from 'expo-camera';
import * as FileSystem from 'expo-file-system';
import * as Permissions from 'expo-permissions';

const VideoRecorder = () => {
  const [hasPermission, setHasPermission] = useState(null);
  const [isRecording, setIsRecording] = useState(false);
  const [filePath, setFilePath] = useState(null);
  const cameraRef = useRef(null);

  React.useEffect(() => {
    (async () => {
      const { status } = await Camera.requestPermissionsAsync();
      setHasPermission(status === 'granted');
    })();
  }, []);

  const startRecording = async () => {
    if (cameraRef.current) {
      setIsRecording(true);
      try {
        const videoRecordPromise = cameraRef.current.recordAsync();
        if (videoRecordPromise) {
          const data = await videoRecordPromise;
          const path = data.uri;
          setFilePath(path);
          logSuccess('Video recorded successfully', path);
        }
      } catch (error) {
        logError('Failed to record video', error);
      }
      setIsRecording(false);
    }
  };

  const stopRecording = () => {
    if (cameraRef.current) {
      cameraRef.current.stopRecording();
    }
  };

  const logSuccess = (message, path) => {
    console.log(`SUCCESS: ${message}. File path: ${path}`);
    // Here you would typically write to a log file
    Alert.alert('Success', `Video recorded and saved at ${path}`);
  };

  const logError = (message, error) => {
    console.error(`ERROR: ${message}`, error);
    // Here you would typically write to a log file
    Alert.alert('Error', 'Failed to record video. Check logs for details.');
  };

  if (hasPermission === null) {
    return <View />;
  }
  if (hasPermission === false) {
    return <Text>No access to camera</Text>;
  }

  return (
    <View style={{ flex: 1 }}>
      <Camera style={{ flex: 1 }} type={Camera.Constants.Type.back}
ref={cameraRef}>
        <View
          style={{
            flex: 1,
            backgroundColor: 'transparent',
            flexDirection: 'row',
            justifyContent: 'center',
            alignItems: 'flex-end',
          }}>
          <Button
            title={isRecording ? 'Stop Recording' : 'Start Recording'}
            onPress={isRecording ? stopRecording : startRecording}
          />
        </View>
      </Camera>
      {filePath && <Text>Last recorded video: {filePath}</Text>}
    </View>
  );
};

export default VideoRecorder;
```

This React Native component, VideoRecorder, implements the functionality you requested. Here's a breakdown of how it works:

1. It uses the expo-camera library to access the device's camera.

2. The component requests camera permissions when it mounts.
3. A button allows the user to start and stop recording.
4. When recording starts, it uses the recordAsync method of the camera.
5. Upon successful recording, it saves the file path and logs the success.
6. If there's an error, it logs the failure.
7. The component displays the file path of the last recorded video.

To use this component, you'll need to install the necessary dependencies:

```
npm install expo-camera expo-permissions expo-file-system
```

Or if you're using Expo:

```
expo install expo-camera expo-permissions expo-file-system
```

This component provides basic functionality. Depending on your specific needs, you might want to add features like:

1. Choosing between front and back camera
2. Setting video quality
3. Adding a preview of the recorded video
4. Implementing more robust error handling

Compress Video:

write React Native code for compressing a video.

Input: File path to video

Processes: Video file fetched, file size checked against limit, file type checked against goal type, conversions if necessary

Output if successful: compressed video file saved, log-file entries

Output if failure: Failure message, log-file entries

1. Fetch the video file from the input path.
2. Check the file size against the limit.
3. Check the file type against the goal type.
4. If necessary, convert and compress the video.
5. Save the compressed video file.
6. Generate appropriate log entries

```
import React, { useState } from 'react';
import { createFFmpeg, fetchFile } from '@ffmpeg/ffmpeg';

function VideoCompressor() {
  const [filePath, setFilePath] = useState('');
  const [compressedFilePath, setCompressedFilePath] = useState('');
  const [message, setMessage] = useState('');
  const [logFile, setLogFile] = useState([]);
  const ffmpeg = createFFmpeg({ log: true });

  const addLogEntry = (message) => {
    const timestamp = new Date().toLocaleString();
    setLogFile((prevLog) => [...prevLog, `${timestamp} - ${message}`]);
  };

  const handleFileChange = (event) => {
    const file = event.target.files[0];
    if (file) {
      setFilePath(URL.createObjectURL(file));
    }
  };
  const compressVideo = async () => {
    if (!filePath) {
      setMessage('Failure: No video file selected.');
      addLogEntry('Failure: No video file selected.');
      return;
    }

    setMessage('Processing started...');
    addLogEntry('Processing started...');

    try {
      await ffmpeg.load();

      const inputFile = await fetchFile(filePath);
      ffmpeg.FS('writeFile', 'input.mp4', inputFile);

      // Check file size and type
      const fileSizeLimit = 50 * 1024 * 1024; // 50 MB
      const goalType = 'mp4';

      const fileSize = inputFile.size;
      const fileType = 'mp4'; // Assuming original format is mp4

      if (fileSize > fileSizeLimit) {
        setMessage('File size exceeds limit, compressing...');
        addLogEntry('File size exceeds limit, compressing...');
      }

      if (fileType !== goalType) {
        setMessage('Converting to goal type...');
        addLogEntry('Converting to goal type...');
      }

      // Compress and convert the video
      await ffmpeg.run('-i', 'input.mp4', '-vcodec', 'libx264', '-crf', '28',
'output.mp4');

      const compressedData = ffmpeg.FS('readFile', 'output.mp4');
      const compressedBlob = new Blob([compressedData.buffer], { type:
'video/mp4' });
      const compressedUrl = URL.createObjectURL(compressedBlob);

      setCompressedFilePath(compressedUrl);
      setMessage('Success: Video compressed and saved successfully.');
      addLogEntry('Success: Video compressed and saved.');

      // Optionally, you can also download the compressed file automatically
      const a = document.createElement('a');
      a.href = compressedUrl;
      a.download = 'compressed_video.mp4';
      a.click();
    } catch (error) {
      setMessage('Failure: Video compression failed.');
      addLogEntry(`Failure: ${error.message}`);
    }
  };

  return (
    <div>
      <input type="file" accept="video/*" onChange={handleFileChange} />
      <button onClick={compressVideo} style={{ marginTop: '10px' }}>
        Compress Video
      </button>

      {message && (
        <div style={{ marginTop: '10px', color: message.startsWith('Success')
? 'green' : 'red' }}>
          {message}
        </div>
      )}

      {compressedFilePath && (
        <div style={{ marginTop: '10px' }}>
          <strong>Compressed File Path:</strong> {compressedFilePath}
        </div>
      )}

      <div style={{ marginTop: '20px' }}>
        <h3>Log Entries:</h3>
        <ul>
          {logFile.map((log, index) => (
            <li key={index}>{log}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default VideoCompressor;
```

Explanation:

1. **State Variables**:
    - filePath: Stores the original video file path.

    - compressedFilePath: Stores the compressed video file path.
    - message: Stores success or failure messages.
    - logFile: Maintains a list of log entries.
2. **FFmpeg Integration**:
    - createFFmpeg: This function from the @ffmpeg/ffmpeg library is used to load and run FFmpeg in the browser.
    - fetchFile(filePath): This function fetches the file from the path provided by the user.
    - ffmpeg.run(): Executes FFmpeg commands to compress and convert the video.
3. **File Handling**:
    - The handleFileChange function captures the selected file and sets the filePath state.
    - The compressVideo function:
        - Loads FFmpeg.
        - Fetches and writes the input file.
        - Checks the file size and type against predefined limits and goals.
        - Executes FFmpeg commands to compress and convert the video.
        - Reads the resulting file and creates a downloadable URL for the compressed video.
        - Logs the success or failure of the process.
4. **Rendering**:
    - An input field allows users to select a video file.
    - A button triggers the compression process.
    - Messages display the status of the operation.
    - The path to the compressed video is displayed if the operation is successful.
    - Log entries are listed below the controls.

Usage:

1. Install the @ffmpeg/ffmpeg package by running:

```
2. npm install @ffmpeg/ffmpeg
```

1. Add the <VideoCompressor /> component to your React application.

2. Users can select a video file, and upon clicking the "Compress Video" button, the video will be compressed according to the specified parameters.
3. The component will display the outcome of the operation, including the file path of the compressed video and a list of log entries.

This component provides a basic structure for video compression in a React app using FFmpeg. You can further enhance it by adding features like custom compression settings, support for more formats, or better error handling.

Claude Sonnet:

```
import React, { useState } from 'react';
import { View, Button, Text, Alert } from 'react-native';
import * as FileSystem from 'expo-file-system';
import { manipulateAsync, SaveFormat } from 'expo-image-manipulator';
import * as VideoThumbnails from 'expo-video-thumbnails';

const VideoCompressor = ({ inputFilePath }) => {
  const [compressedFilePath, setCompressedFilePath] = useState(null);
  const [isCompressing, setIsCompressing] = useState(false);

  const FILE_SIZE_LIMIT = 10 * 1024 * 1024; // 10MB
  const TARGET_FILE_TYPE = 'mp4';

  const compressVideo = async () => {
    setIsCompressing(true);
    try {
      // Check if file exists
      const fileInfo = await FileSystem.getInfoAsync(inputFilePath);
      if (!fileInfo.exists) {
        throw new Error('File does not exist');
      }

      // Check file size
      if (fileInfo.size <= FILE_SIZE_LIMIT) {
        logSuccess('File is already within size limit', inputFilePath);
        setCompressedFilePath(inputFilePath);
        return;
      }

      // Check file type
      const fileExtension = inputFilePath.split('.').pop().toLowerCase();
      if (fileExtension !== TARGET_FILE_TYPE) {
        // If not mp4, we need to convert it
        // Note: Direct video conversion is not supported in React Native
        // We'll use a workaround by extracting frames and creating a new
video
        const frames = await extractFrames(inputFilePath);
        const compressedPath = await createVideoFromFrames(frames);
        setCompressedFilePath(compressedPath);
        logSuccess('Video converted and compressed', compressedPath);
      } else {
        // If mp4, we can try to compress it
        const compressedPath = await compressMP4(inputFilePath);
        setCompressedFilePath(compressedPath);
        logSuccess('Video compressed', compressedPath);
      }
    } catch (error) {
      logError('Failed to compress video', error);
    } finally {
      setIsCompressing(false);
    }
  };

  const extractFrames = async (videoPath) => {
    // This is a simplified version. In a real app, you'd extract multiple
frames.
    const { uri } = await VideoThumbnails.getThumbnailAsync(videoPath, {
      time: 0,
    });
    return [uri];
  };

  const createVideoFromFrames = async (frames) => {
    // This is a placeholder. Creating a video from frames is complex and
    // typically requires native modules or server-side processing.
    // For this example, we'll just return the first frame as an image.
    const compressedUri = await manipulateAsync(
      frames[0],
      [{ resize: { width: 640 } }],
      { compress: 0.7, format: SaveFormat.MP4 }
    );
    return compressedUri.uri;
  };

  const compressMP4 = async (videoPath) => {
    // This is a placeholder. Actual video compression typically requires
    // native modules like react-native-video-processing or server-side
processing.
    // For this example, we'll just copy the file.
    const newPath = FileSystem.documentDirectory + 'compressed_video.mp4';
    await FileSystem.copyAsync({
      from: videoPath,
      to: newPath
    });
    return newPath;
  };

  const logSuccess = (message, path) => {
    console.log(`SUCCESS: ${message}. File path: ${path}`);
    // Here you would typically write to a log file
    Alert.alert('Success', `Video processed and saved at ${path}`);
  };
  const logError = (message, error) => {
    console.error(`ERROR: ${message}`, error);
    // Here you would typically write to a log file
    Alert.alert('Error', 'Failed to process video. Check logs for details.');
  };

  return (
    <View>
      <Button
        title={isCompressing ? "Compressing..." : "Compress Video"}
        onPress={compressVideo}
        disabled={isCompressing}
      />
      {compressedFilePath && (
        <Text>Compressed video saved at: {compressedFilePath}</Text>
      )}
    </View>
  );
};

export default VideoCompressor;
```

This React Native component, VideoCompressor, implements the functionality you requested. Here's a breakdown of how it works:

1. It takes an inputFilePath as a prop, which is the path to the video file to be compressed.
2. It checks if the file exists and its size.
3. If the file is already within the size limit (10MB in this example), it doesn't compress it.
4. If the file is not in the target format (mp4 in this example), it attempts to convert it.
5. For mp4 files that need compression, it calls a compressMP4 function.
6. It logs success or failure messages and updates the UI accordingly.

To use this component, you'll need to install the necessary dependencies:

```
expo install expo-file-system expo-image-manipulator expo-video-thumbnails
```

Important notes:

1. This is a simplified version and doesn't actually perform video compression or conversion. Real video compression in React Native typically requires native modules or server-side processing.
2. The extractFrames and createVideoFromFrames functions are placeholders. In a real application, you'd need a more sophisticated approach to handle video conversion.
3. The compressMP4 function is also a placeholder. Actual video compression would require a native module like react-native-video-processing or a server-side solution.

4. Error handling and logging are basic. In a production app, you'd want more robust error handling and possibly a dedicated logging system.
5. The file size limit and target file type are hardcoded. In a real app, you might want to make these configurable.

To use this component, you would do something like:

```
<VideoCompressor inputFilePath="/path/to/your/video.mp4" />
```

This component provides a structure for video compression, but implementing actual video compression would require additional native modules or backend services

Upload video:

write React Native code for uploading a video.

Input: file path compressed video file

Processes: open connection to API, upload file, close connection to server

Output if successful: Success message, log-file entries

Output if failure: Failure message, log-file entries

To create a React component that uploads a video file to a server via an API, you can follow these steps:

1. **Input**: Take the file path of the compressed video as input.
2. **Process**: Open a connection to the API, upload the video file, and close the connection.
3. **Output**: Display success or failure messages and maintain a log of the operations.

```
import React, { useState } from 'react';
import axios from 'axios';

function VideoUploader() {
  const [filePath, setFilePath] = useState('');
  const [message, setMessage] = useState('');
  const [logFile, setLogFile] = useState([]);

  const addLogEntry = (message) => {
    const timestamp = new Date().toLocaleString();
    setLogFile((prevLog) => [...prevLog, `${timestamp} - ${message}`]);
  };

  const handleFileChange = (event) => {
    const file = event.target.files[0];
    if (file) {
      setFilePath(file);
    }
  };

  const uploadVideo = async () => {
    if (!filePath) {
      setMessage('Failure: No video file selected.');
      addLogEntry('Failure: No video file selected.');
      return;
    }

    setMessage('Uploading video...');
    addLogEntry('Uploading video...');

    const formData = new FormData();
    formData.append('video', filePath);

    try {
      const response = await axios.post('https://your-api-
endpoint.com/upload', formData, {
        headers: {
          'Content-Type': 'multipart/form-data',
        },
      });

      if (response.status === 200) {
        setMessage('Success: Video uploaded successfully.');
        addLogEntry('Success: Video uploaded successfully.');
      } else {
        setMessage('Failure: Video upload failed.');
        addLogEntry('Failure: Video upload failed.');
      }
    } catch (error) {
      setMessage(`Failure: ${error.message}`);
      addLogEntry(`Failure: ${error.message}`);
    }
  };

  return (
    <div>
      <input type="file" accept="video/*" onChange={handleFileChange} />
      <button onClick={uploadVideo} style={{ marginTop: '10px' }}>
        Upload Video
      </button>

      {message && (
        <div style={{ marginTop: '10px', color: message.startsWith('Success')
? 'green' : 'red' }}>
          {message}
        </div>
      )}

      <div style={{ marginTop: '20px' }}>
        <h3>Log Entries:</h3>
        <ul>
          {logFile.map((log, index) => (
            <li key={index}>{log}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default VideoUploader;
```

Explanation:

1. **State Variables**:
    - filePath: Stores the selected video file.
    - message: Stores success or failure messages.
    - logFile: Maintains a list of log entries.
2. **File Handling**:
    - The handleFileChange function captures the selected video file and sets the filePath state.
3. **Uploading Process**:
    - The uploadVideo function:
        - Checks if a video file has been selected.
        - Prepares the file for upload by appending it to a FormData object.
        - Sends a POST request to the API endpoint using axios.
        - Handles success and failure responses by updating the message state and logging the outcome.
4. **Rendering**:
    - An input field allows users to select a video file.
    - A button triggers the file upload process.
    - Messages display the status of the operation.
    - Log entries are displayed below the controls to track the upload process.

Usage:

1. **Install Dependencies**:
    - Ensure you have axios installed in your project:

```
2. npm install axios
```

1. **API Endpoint**:
    - Replace 'https://your-api-endpoint.com/upload' with the actual URL of your API endpoint.
2. **Add the Component**:
    - Include the <VideoUploader /> component in your React application.
3. **Upload Process**:
    - Users can select a compressed video file using the file input.
    - Clicking the "Upload Video" button will upload the video to the server using the specified API.
    - The component will display the outcome of the upload and log the details.

Enhancements and Considerations:

- **Error Handling**: Enhance error handling by providing more specific feedback based on the error type or status code.
- **Progress Indicator**: Implement a progress bar or percentage indicator to show the upload progress.
- **Multiple File Uploads**: Extend the component to support multiple file uploads if needed.
- **Security**: Ensure your API endpoint is secure and uses appropriate authentication and authorization mechanisms.

This implementation provides a simple, effective, and customizable way to upload video files in a React application.

Claude Sonnet:

```
import React, { useState } from 'react';
import { View, Button, Text, Alert } from 'react-native';
import * as FileSystem from 'expo-file-system';

const VideoUploader = ({ filePath }) => {
  const [isUploading, setIsUploading] = useState(false);
  const [uploadProgress, setUploadProgress] = useState(0);

  const API_ENDPOINT = 'https://your-api-endpoint.com/upload'; // Replace with your actual API endpoint

  const uploadVideo = async () => {
    setIsUploading(true);
    setUploadProgress(0);

    try {
      // Check if file exists
      const fileInfo = await FileSystem.getInfoAsync(filePath);
      if (!fileInfo.exists) {
        throw new Error('File does not exist');
      }

      // Prepare the form data
      const formData = new FormData();
      formData.append('video', {
        uri: filePath,
        name: 'video.mp4',
        type: 'video/mp4'
      });

      // Open connection and upload file
      const response = await fetch(API_ENDPOINT, {
        method: 'POST',
        body: formData,
        headers: {
          'Content-Type': 'multipart/form-data',
        },
      });

      // Check if the upload was successful
      if (response.ok) {
        const result = await response.json();
        logSuccess('Video uploaded successfully', result);
      } else {
        throw new Error('Upload failed');
      }
    } catch (error) {
      logError('Failed to upload video', error);
    } finally {
      setIsUploading(false);
    }
  };

  const logSuccess = (message, result) => {
    console.log(`SUCCESS: ${message}`, result);
    // Here you would typically write to a log file
    Alert.alert('Success', 'Video uploaded successfully');
  };

  const logError = (message, error) => {
    console.error(`ERROR: ${message}`, error);
    // Here you would typically write to a log file
    Alert.alert('Error', 'Failed to upload video. Check logs for details.');
  };

  // This function would be used with FileSystem.uploadAsync for progress
tracking
  const uploadProgressCallback = (progress) => {
    const percentage = Math.round((progress.totalBytesSent /
progress.totalBytesExpectedToSend) * 100);
    setUploadProgress(percentage);
  };

  return (
    <View>
      <Button
        title={isUploading ? "Uploading..." : "Upload Video"}
        onPress={uploadVideo}
        disabled={isUploading}
      />
      {isUploading && (
        <Text>Upload Progress: {uploadProgress}%</Text>
      )}
    </View>
  );
};

export default VideoUploader;
```

This React Native component, VideoUploader, implements the functionality you requested. Here's a breakdown of how it works:

1. It takes a filePath as a prop, which is the path to the video file to be uploaded.
2. It uses the fetch API to upload the file to a specified endpoint.
3. It checks if the file exists before attempting to upload.
4. It creates a FormData object to send the file as multipart/form-data.
5. It handles the upload process, including success and error cases.
6. It provides basic progress tracking (though the implementation would need to be adjusted based on the actual upload method used).
7. It logs success or failure messages and updates the UI accordingly.

To use this component, you'll need to install the necessary dependency:

```
expo install expo-file-system
```

Certainly. I'll create a React Native component that handles uploading a video file to a server using an API. This component will take a file path as input, open a connection to the API, upload the file, and then close the connection.

React Native Video Upload Component

Click to open code

This React Native component, VideoUploader, implements the functionality you requested. Here's a breakdown of how it works:

1. It takes a filePath as a prop, which is the path to the video file to be uploaded.
2. It uses the fetch API to upload the file to a specified endpoint.
3. It checks if the file exists before attempting to upload.
4. It creates a FormData object to send the file as multipart/form-data.
5. It handles the upload process, including success and error cases.
6. It provides basic progress tracking (though the implementation would need to be adjusted based on the actual upload method used).
7. It logs success or failure messages and updates the UI accordingly.

To use this component, you'll need to install the necessary dependency:

Copy

expo install expo-file-system

Important notes:

1. Replace 'https://your-api-endpoint.com/upload' with your actual API endpoint for video uploads.

2. This example uses fetch for simplicity, but for more robust uploading (especially for large files), you might want to use FileSystem.uploadAsync from Expo, or a third-party library like axios.
3. The progress tracking in this example is placeholder. To implement real progress tracking, you'd need to use a method that supports it, like FileSystem.uploadAsync.
4. Error handling and logging are basic. In a production app, you'd want more robust error handling and possibly a dedicated logging system.
5. This component doesn't handle authentication. If your API requires authentication, you'll need to add appropriate headers or tokens to the request.

To use this component, you would do something like:

```
<VideoUploader filePath="/path/to/your/compressed/video.mp4" />
```

This component provides a structure for video uploading, but you may need to adjust it based on your specific API requirements and error handling needs.

Create Assignment:

write React Native code for creating an assignment

Input: User interacts with a button, user provides assignment information IBLNT: Subject Information, Assignment Name, Due Date, Assignment Information

Processes: Data verification, database entry created

Output if successful: Success message, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';
import axios from 'axios';

function CreateAssignment() {
  const [assignmentName, setAssignmentName] = useState('');
  const [dueDate, setDueDate] = useState('');
  const [assignmentInfo, setAssignmentInfo] = useState('');
  const [message, setMessage] = useState('');
  const [logFile, setLogFile] = useState([]);

  const addLogEntry = (message) => {
    const timestamp = new Date().toLocaleString();
    setLogFile((prevLog) => [...prevLog, `${timestamp} - ${message}`]);
  };

  const handleSubmit = async (event) => {
    event.preventDefault();

    // Data verification
    if (!assignmentName || !dueDate || !assignmentInfo) {
      setMessage('Failure: All fields are required.');
      addLogEntry('Failure: Missing required fields.');
      return;
    }

    const assignmentData = {
      name: assignmentName,
      dueDate: dueDate,
      info: assignmentInfo,
    };

    setMessage('Processing assignment creation...');
    addLogEntry('Processing assignment creation...');
    try {
      // Simulate a database entry using an API call (replace with your actual API endpoint)
      const response = await axios.post('https://your-api-endpoint.com/assignments', assignmentData);

      if (response.status === 201) {
        setMessage('Success: Assignment created successfully.');
        addLogEntry('Success: Assignment created successfully.');
        // Clear form inputs
        setAssignmentName('');
        setDueDate('');
        setAssignmentInfo('');
      } else {
        setMessage('Failure: Could not create assignment.');
        addLogEntry('Failure: Could not create assignment.');
      }
    } catch (error) {
      setMessage(`Failure: ${error.message}`);
      addLogEntry(`Failure: ${error.message}`);
    }
  };

  return (
    <div>
      <h2>Create Assignment</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label>
            Assignment Name:
            <input
              type="text"
              value={assignmentName}
              onChange={(e) => setAssignmentName(e.target.value)}
              required
            />
          </label>
        </div>
        <div>
          <label>
            Due Date:
            <input
              type="date"
              value={dueDate}
              onChange={(e) => setDueDate(e.target.value)}
              required
            />
          </label>
        </div>
        <div>
          <label>
            Assignment Information:
            <textarea
              value={assignmentInfo}
              onChange={(e) => setAssignmentInfo(e.target.value)}
              required
            />
          </label>
        </div>
        <button type="submit" style={{ marginTop: '10px' }}>
          Create Assignment
        </button>
      </form>

      {message && (
        <div style={{ marginTop: '10px', color: message.startsWith('Success')
? 'green' : 'red' }}>
          {message}
        </div>
      )}

      <div style={{ marginTop: '20px' }}>
        <h3>Log Entries:</h3>
        <ul>
          {logFile.map((log, index) => (
            <li key={index}>{log}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default CreateAssignment;
```

Explanation:

1. **State Variables**:
    - assignmentName: Stores the name of the assignment.
    - dueDate: Stores the due date of the assignment.
    - assignmentInfo: Stores additional information about the assignment.
    - message: Stores success or failure messages.

    - logFile: Maintains a list of log entries.
2. **Form Handling**:
    - The form includes fields for the assignment name, due date, and additional information.
    - The handleSubmit function is triggered when the form is submitted. It performs data verification to ensure that all fields are filled out.
3. **Log Entries**:
    - addLogEntry adds a new log entry with a timestamp and the provided message.
4. **Database Entry (Simulated)**:
    - The handleSubmit function simulates a database entry by sending an HTTP POST request to a sample API endpoint using axios.
    - If the request is successful (HTTP status 201), the form is reset, and a success message is displayed. Otherwise, a failure message is shown.
5. **Rendering**:
    - The form fields allow the user to input the assignment details.
    - The submit button triggers the creation process.
    - Messages indicating success or failure are displayed.
    - The log entries are listed below the form, providing a record of the operations.

Usage:

1. **Install Dependencies**:
    - Ensure you have axios installed in your project:

```
npm install axios
```

1. **API Endpoint**:
    - Replace 'https://your-api-endpoint.com/assignments' with the actual URL of your API endpoint.
2. **Add the Component**:
    - Include the <CreateAssignment /> component in your React application.
3. **Assignment Creation Process**:
    - Users can input details such as the assignment name, due date, and additional information.
    - Upon submitting the form, the data is verified, and the assignment is created by making a POST request to the server.

- The component displays the outcome and logs the process.

This implementation provides a foundational structure for creating assignments in a React application. You can enhance it by adding features like custom validation rules, more detailed error handling, or connecting to a real backend API for managing assignments.

Claude Sonnet:

```
import React, { useState } from 'react';
import { View, Text, TextInput, Button, Alert, ScrollView, StyleSheet } from
'react-native';
import DateTimePicker from '@react-native-community/datetimepicker';

const AssignmentCreator = () => {
  const [subject, setSubject] = useState('');
  const [assignmentName, setAssignmentName] = useState('');
  const [dueDate, setDueDate] = useState(new Date());
  const [assignmentInfo, setAssignmentInfo] = useState('');
  const [showDatePicker, setShowDatePicker] = useState(false);

  const validateData = () => {
    if (!subject.trim()) {
      Alert.alert('Error', 'Subject is required');
      return false;
    }
    if (!assignmentName.trim()) {
      Alert.alert('Error', 'Assignment Name is required');
      return false;
    }
    if (!assignmentInfo.trim()) {
      Alert.alert('Error', 'Assignment Information is required');
      return false;
    }
    return true;
  };

  const createAssignment = () => {
    if (!validateData()) return;

    // Simulate database entry creation
    const assignmentData = {
      subject,
      assignmentName,
      dueDate: dueDate.toISOString(),
      assignmentInfo,
      createdAt: new Date().toISOString(),
    };

    // In a real app, you would send this data to your backend API
    // For this example, we'll just simulate a successful creation
    setTimeout(() => {
      logSuccess('Assignment created successfully', assignmentData);
      resetForm();
    }, 1000);
  };

  const resetForm = () => {
    setSubject('');
    setAssignmentName('');
    setDueDate(new Date());
    setAssignmentInfo('');
  };

  const logSuccess = (message, data) => {
    console.log(`SUCCESS: ${message}`, data);
    // Here you would typically write to a log file
    Alert.alert('Success', 'Assignment created successfully');
  };

  const logError = (message, error) => {
    console.error(`ERROR: ${message}`, error);
    // Here you would typically write to a log file
    Alert.alert('Error', 'Failed to create assignment. Please try again.');
  };

  const onChangeDueDate = (event, selectedDate) => {
    const currentDate = selectedDate || dueDate;
    setShowDatePicker(false);
    setDueDate(currentDate);
  };

  return (
    <ScrollView style={styles.container}>
      <Text style={styles.label}>Subject:</Text>
      <TextInput
        style={styles.input}
        value={subject}
        onChangeText={setSubject}
        placeholder="Enter subject"
      />

      <Text style={styles.label}>Assignment Name:</Text>
      <TextInput
        style={styles.input}
        value={assignmentName}
        onChangeText={setAssignmentName}
        placeholder="Enter assignment name"
      />

      <Text style={styles.label}>Due Date:</Text>
```

```
      <Button onPress={() => setShowDatePicker(true)} title="Select Due Date" />
      {showDatePicker && (
        <DateTimePicker
          testID="dateTimePicker"
          value={dueDate}
          mode="date"
          is24Hour={true}
          display="default"
          onChange={onChangeDueDate}
        />
      )}
      <Text style={styles.dateDisplay}>{dueDate.toDateString()}</Text>

      <Text style={styles.label}>Assignment Information:</Text>
      <TextInput
        style={[styles.input, styles.multilineInput]}
        value={assignmentInfo}
        onChangeText={setAssignmentInfo}
        placeholder="Enter assignment details"
        multiline
        numberOfLines={4}
      />

      <Button title="Create Assignment" onPress={createAssignment} />
    </ScrollView>
  );
};

const styles = StyleSheet.create({
  container: {
    padding: 20,
  },
  label: {
    fontSize: 16,
    fontWeight: 'bold',
    marginTop: 10,
  },
  input: {
    borderWidth: 1,
    borderColor: '#ddd',
    padding: 10,
    marginBottom: 10,
    borderRadius: 4,
  },
  multilineInput: {
    height: 100,
    textAlignVertical: 'top',
  },
  dateDisplay: {
    marginTop: 10,
    marginBottom: 10,
  },
});

export default AssignmentCreator;
```

This React Native component, AssignmentCreator, implements the functionality you requested. Here's a breakdown of how it works:

1. It provides input fields for Subject, Assignment Name, Due Date, and Assignment Information.
2. It uses state to manage the input values.
3. It includes data validation to ensure all required fields are filled.
4. It simulates creating a database entry with the assignment data.
5. It provides success and error logging.
6. It uses a DateTimePicker for selecting the due date.

To use this component, you'll need to install the necessary dependency:

```
npm install @react-native-community/datetimepicker
```

Certainly. I'll create a React Native component that allows users to create an assignment by providing the necessary information. The component will handle data verification and simulate a database entry creation.

React Native Assignment Creation Component

Click to open code

This React Native component, AssignmentCreator, implements the functionality you requested. Here's a breakdown of how it works:

1. It provides input fields for Subject, Assignment Name, Due Date, and Assignment Information.
2. It uses state to manage the input values.
3. It includes data validation to ensure all required fields are filled.
4. It simulates creating a database entry with the assignment data.
5. It provides success and error logging.
6. It uses a DateTimePicker for selecting the due date.

To use this component, you'll need to install the necessary dependency:

Copy

```
npm install @react-native-community/datetimepicker
```

Important notes:

1. This example uses setTimeout to simulate an API call. In a real application, you would replace this with an actual API call to your backend.
2. The data validation is basic. You might want to add more sophisticated validation based on your specific requirements.
3. Error handling and logging are basic. In a production app, you'd want more robust error handling and possibly a dedicated logging system.
4. The UI is minimal. You might want to enhance it with custom styling or additional UI components.
5. This component doesn't handle state management beyond its own local state. In a larger application, you might want to use a state management solution like Redux or Context API.

To use this component in your app, you would simply include it in your JSX:

```
<AssignmentCreator />
```

This component provides a structure for assignment creation, but you may need to adjust it based on your specific API requirements, data model, and UI/UX preferences

View Assignment:

write React Native code for viewing assignment.

Input: User interacts with a button

Processes: Assignment Information is retrieved from database

Output if successful: Assignment information presented to user, log-file entries

Output if failure: Failure message, log-file entries

Below is a React component that allows a user to view an assignment. The user interacts with a button to fetch assignment details from a database (simulated with an API call). The component handles the retrieval process, logs the operations, and displays the assignment information.

```
import React, { useState } from 'react';
import axios from 'axios';

function ViewAssignment() {
  const [assignmentData, setAssignmentData] = useState(null);
  const [message, setMessage] = useState('');
  const [logFile, setLogFile] = useState([]);

  const addLogEntry = (message) => {
    const timestamp = new Date().toLocaleString();
    setLogFile((prevLog) => [...prevLog, `${timestamp} - ${message}`]);
  };

  const fetchAssignment = async () => {
    setMessage('Fetching assignment information...');
    addLogEntry('Fetching assignment information...');

    try {
      // Simulate API call to fetch assignment data
      const response = await axios.get('https://your-api-
endpoint.com/assignments/1'); // Replace with actual API endpoint

      if (response.status === 200) {
        setAssignmentData(response.data);
        setMessage('Success: Assignment information retrieved.');
        addLogEntry('Success: Assignment information retrieved.');
      } else {
        setMessage('Failure: Could not retrieve assignment.');
        addLogEntry('Failure: Could not retrieve assignment.');
      }
    } catch (error) {
      setMessage(`Failure: ${error.message}`);
      addLogEntry(`Failure: ${error.message}`);
    }
  };

  return (
    <div>
      <button onClick={fetchAssignment} style={{ marginTop: '10px' }}>
        View Assignment
      </button>

      {message && (
        <div style={{ marginTop: '10px', color: message.startsWith('Success') ? 'green' : 'red' }}>
          {message}
        </div>
      )}

      {assignmentData && (
        <div style={{ marginTop: '20px' }}>
          <h3>Assignment Information</h3>
          <p><strong>Name:</strong> {assignmentData.name}</p>
          <p><strong>Due Date:</strong> {assignmentData.dueDate}</p>
          <p><strong>Information:</strong> {assignmentData.info}</p>
        </div>
      )}

      <div style={{ marginTop: '20px' }}>
        <h3>Log Entries:</h3>
        <ul>
          {logFile.map((log, index) => (
            <li key={index}>{log}</li>
          ))}
        </ul>
      </div>
    </div>
  );
}

export default ViewAssignment;
```

Explanation:

1. **State Variables:**

   - assignmentData: Stores the assignment details retrieved from the server.
   - message: Stores success or failure messages.
   - logFile: Maintains a list of log entries.
2. **Logging**:
   - addLogEntry: Adds a new log entry with a timestamp and the provided message.
3. **Fetching Assignment Data**:
   - fetchAssignment: This function is triggered when the user clicks the "View Assignment" button. It sends an HTTP GET request to fetch the assignment details from the server.
   - If the request is successful, the retrieved assignment data is stored in assignmentData, and a success message is displayed.
   - If the request fails, an error message is displayed, and the failure is logged.
4. **Rendering**:
   - A button is provided for the user to initiate the fetch operation.
   - Messages indicating success or failure are displayed after the fetch operation.
   - If assignment data is retrieved successfully, it is displayed in a structured format.
   - Log entries are listed below the display to track the operations.

Usage:

1. **Install Dependencies**:
   - Ensure you have axios installed in your project:

```
2. npm install axios
```

1. **API Endpoint**:
   - Replace 'https://your-api-endpoint.com/assignments/1' with the actual URL of your API endpoint that retrieves the assignment data. You may need to adjust the URL to match your backend implementation.
2. **Add the Component**:
   - Include the <ViewAssignment /> component in your React application.
3. **Viewing Assignment Process**:
   - When the user clicks the "View Assignment" button, the component sends a request to the server to retrieve the assignment details.
   - The component will display the retrieved assignment information or an error message if the operation fails.

- The process is logged, with entries displayed below the assignment information.

This implementation provides a straightforward way to fetch and display assignment details in a React application. It can be enhanced by adding features such as handling multiple assignments, refining error handling, or improving the user interface.

Claude Sonnet:

```
import React, { useState, useEffect } from 'react';
import { View, Text, FlatList, TouchableOpacity, StyleSheet, Alert } from 'react-native';

const AssignmentViewer = () => {
  const [assignments, setAssignments] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchAssignments();
  }, []);

  const fetchAssignments = async () => {
    setLoading(true);
    try {
      // Simulating API call to fetch assignments
      // In a real app, this would be an actual API call
      const response = await new Promise((resolve) =>
        setTimeout(() => resolve({
          ok: true,
          json: () => Promise.resolve([
            { id: '1', subject: 'Math', name: 'Algebra Homework', dueDate: '2023-09-01', info: 'Complete exercises 1-10' },
            { id: '2', subject: 'Science', name: 'Lab Report', dueDate: '2023-09-05', info: 'Write up results from photosynthesis experiment' },
            { id: '3', subject: 'History', name: 'Essay', dueDate: '2023-09-10', info: 'Write a 5-page essay on the Industrial Revolution' },
          ])
        }), 1000)
      );

      if (response.ok) {
        const data = await response.json();
        setAssignments(data);
        logSuccess('Assignments fetched successfully');
      } else {
        throw new Error('Failed to fetch assignments');
      }
    } catch (error) {
      logError('Error fetching assignments', error);
    } finally {
      setLoading(false);
    }
  };
  const logSuccess = (message) => {
    console.log(`SUCCESS: ${message}`);
    // Here you would typically write to a log file
  };

  const logError = (message, error) => {
    console.error(`ERROR: ${message}`, error);
    // Here you would typically write to a log file
    Alert.alert('Error', 'Failed to fetch assignments. Please try again.');
  };

  const renderAssignment = ({ item }) => (
    <TouchableOpacity
      style={styles.assignmentItem}
      onPress={() => viewAssignmentDetails(item)}
    >
      <Text style={styles.assignmentName}>{item.name}</Text>
      <Text>Subject: {item.subject}</Text>
      <Text>Due Date: {new Date(item.dueDate).toLocaleDateString()}</Text>
    </TouchableOpacity>
  );

  const viewAssignmentDetails = (assignment) => {
    Alert.alert(
      assignment.name,
      `Subject: ${assignment.subject}\nDue Date: ${new
Date(assignment.dueDate).toLocaleDateString()}\n\nDetails: ${assignment.info}`
    );
  };

  if (loading) {
    return <Text style={styles.loadingText}>Loading assignments...</Text>;
  }

  return (
    <View style={styles.container}>
      <Text style={styles.header}>Your Assignments</Text>
      <FlatList
        data={assignments}
        renderItem={renderAssignment}
        keyExtractor={(item) => item.id}
        ListEmptyComponent={<Text style={styles.emptyText}>No assignments
found</Text>}
      />
    </View>
  );
};
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
  },
  header: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
  },
  assignmentItem: {
    backgroundColor: '#f9f9f9',
    padding: 15,
    borderRadius: 5,
    marginBottom: 10,
  },
  assignmentName: {
    fontSize: 18,
    fontWeight: 'bold',
    marginBottom: 5,
  },
  loadingText: {
    fontSize: 18,
    textAlign: 'center',
    marginTop: 50,
  },
  emptyText: {
    fontSize: 16,
    textAlign: 'center',
    marginTop: 50,
  },
});

export default AssignmentViewer;
```

This React Native component, AssignmentViewer, implements the functionality you requested. Here's a breakdown of how it works:

1. It uses useState and useEffect hooks to manage state and side effects.
2. On component mount, it fetches assignments (simulated in this example).
3. It displays a list of assignments using FlatList for efficient rendering.
4. Each assignment item is touchable and shows more details when pressed.
5. It handles loading states and empty states.
6. It includes basic error handling and logging.

Important notes:

1. This example uses a Promise with setTimeout to simulate an API call. In a real application, you would replace this with an actual API call to your backend.
2. The data structure for assignments is assumed to have id, subject, name, dueDate, and info fields. Adjust this according to your actual data model.
3. Error handling and logging are basic. In a production app, you'd want more robust error handling and possibly a dedicated logging system.
4. The UI is minimal. You might want to enhance it with custom styling or additional UI components.
5. This component doesn't handle pagination or infinite scrolling. For large datasets, you might want to implement these features.

To use this component in your app, you would simply include it in your JSX:

```
<AssignmentViewer />
```

This component provides a structure for viewing assignments, but you may need to adjust it based on your specific API, data model, and UI/UX preferences. Here are some potential enhancements you might consider:

1. Implement pull-to-refresh functionality for updating the assignment list.
2. Add sorting options (e.g., by due date, subject).
3. Implement search functionality to filter assignments.
4. Add the ability to mark assignments as complete or delete them.
5. Implement a more detailed view for each assignment, possibly on a separate screen.

View Submissions: (Claude 3.5 - Sonnet)

write React Native code for viewing submissions.

Input: User interacts with a button

Processes: Submission information is retrieved from the database

Output if successful: Submission information presented to the user, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState, useEffect } from 'react';
import { View, Text, Button, FlatList, StyleSheet } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';

const SubmissionsScreen = () => {
  const [submissions, setSubmissions] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  const fetchSubmissions = async () => {
    setIsLoading(true);
    setError(null);
    try {
      // Simulating database retrieval with AsyncStorage
      const storedSubmissions = await AsyncStorage.getItem('submissions');
      if (storedSubmissions) {
        setSubmissions(JSON.parse(storedSubmissions));
        console.log('Submissions retrieved successfully');
      } else {
        setSubmissions([]);
        console.log('No submissions found');
      }
    } catch (err) {
      setError('Failed to fetch submissions');
      console.error('Error fetching submissions:', err);
    } finally {
      setIsLoading(false);
    }
  };

  const renderSubmission = ({ item }) => (
    <View style={styles.submissionItem}>
      <Text>{item.title}</Text>
      <Text>{item.description}</Text>
    </View>
  );
  return (
    <View style={styles.container}>
      <Button
        title="View Submissions"
        onPress={fetchSubmissions}
        disabled={isLoading}
      />
      {isLoading && <Text>Loading submissions...</Text>}
      {error && <Text style={styles.errorText}>{error}</Text>}
      <FlatList
        data={submissions}
        renderItem={renderSubmission}
        keyExtractor={(item) => item.id.toString()}
        ListEmptyComponent={<Text>No submissions found</Text>}
      />
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
  },
  submissionItem: {
    backgroundColor: '#f0f0f0',
    padding: 10,
    marginVertical: 5,
    borderRadius: 5,
  },
  errorText: {
    color: 'red',
    marginVertical: 10,
  },
});

export default SubmissionsScreen;
```

This code creates a React Native component called SubmissionsScreen that handles viewing submissions. Here's a breakdown of the implementation:

1. The component uses React hooks (useState, useEffect) to manage state and side effects.
2. It defines a fetchSubmissions function that simulates retrieving submission data from a database using AsyncStorage. In a real application, you'd replace this with an API call to your backend.

3. The UI consists of a button to trigger the submission retrieval and a FlatList to display the submissions.
4. Error handling is implemented to show error messages if the fetch fails.
5. Loading state is managed to disable the button and show a loading message while fetching.
6. Successful retrieval of submissions updates the state and displays the data in the FlatList.
7. Console logs are used to record successful operations and errors, which could be expanded to more comprehensive logging in a production app.

To use this component, you'll need to install the necessary dependencies:

```
npm install @react-native-async-storage/async-storage
```

Remember to replace the AsyncStorage implementation with your actual database or API calls in a real application. Also, you might want to add more detailed error handling and logging based on your specific requirements.

Browse your own submissions:

write React Native code for browsing own submissions.

Input: User interacts with a button

Processes: Submission information is retrieved from the database

Output if successful: Submission information presented to the user, log-file entries

Output if failure: Failure message, log-file entries

This code creates a React Native component called OwnSubmissionsScreen that allows users to view their own submissions. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for submissions, loading state, and errors.
2. The fetchOwnSubmissions function simulates retrieving the user's submissions from a database. It uses AsyncStorage to get the user's ID and all submissions, then filters for the user's own submissions.
3. The UI includes a button to trigger the submission retrieval and a FlatList to display the user's submissions.
4. Error handling is implemented to show error messages if the fetch fails.
5. A loading indicator (ActivityIndicator) is shown while fetching data.
6. Successful retrieval of submissions updates the state and displays the data in the FlatList.
7. Console logs are used to record successful operations and errors.
8. The FlatList renders each submission with its title, description, and submission date.
9. If the user has no submissions, a message is displayed indicating this.

To use this component, you'll need to install the necessary dependency:

```
npm install @react-native-async-storage/async-storage
```

In a real application, you would replace the AsyncStorage implementation with actual API calls to your backend. You'd also need to implement user authentication and store the user's ID securely.

Remember to handle user authentication, implement proper error handling, and set up comprehensive logging for production use.

View feedback on your own submissions:

write React Native code for viewing submissions.

Input: User interacts with a button

Processes: Feedback information is retrieved from the database

Output if successful: feedback information provided to the user, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';
import { View, Text, Button, FlatList, StyleSheet, ActivityIndicator } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';

const ViewSubmissionsWithFeedback = () => {
  const [submissions, setSubmissions] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  const fetchSubmissionsWithFeedback = async () => {
    setIsLoading(true);
    setError(null);
    try {
      // Simulating database retrieval with AsyncStorage
      const storedSubmissions = await AsyncStorage.getItem('submissionsWithFeedback');
      if (storedSubmissions) {
        setSubmissions(JSON.parse(storedSubmissions));
        console.log('Submissions with feedback retrieved successfully');
      } else {
        setSubmissions([]);
        console.log('No submissions with feedback found');
      }
    } catch (err) {
      setError('Failed to fetch submissions with feedback');
      console.error('Error fetching submissions with feedback:', err);
    } finally {
      setIsLoading(false);
    }
  };

  const renderSubmission = ({ item }) => (
    <View style={styles.submissionItem}>
      <Text style={styles.submissionTitle}>{item.title}</Text>
      <Text>{item.description}</Text>
      <Text style={styles.feedbackTitle}>Feedback:</Text>
      <Text>{item.feedback || 'No feedback provided yet'}</Text>
    </View>
  );

  return (
    <View style={styles.container}>
      <Button
        title="View Submissions with Feedback"
        onPress={fetchSubmissionsWithFeedback}
        disabled={isLoading}
      />
      {isLoading && <ActivityIndicator size="large" color="#0000ff" />}
      {error && <Text style={styles.errorText}>{error}</Text>}
      <FlatList
        data={submissions}
        renderItem={renderSubmission}
        keyExtractor={(item) => item.id.toString()}
        ListEmptyComponent={<Text style={styles.emptyText}>No submissions with
feedback found</Text>}
      />
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
  },
  submissionItem: {
    backgroundColor: '#f0f0f0',
    padding: 15,
    marginVertical: 8,
    borderRadius: 5,
  },
  submissionTitle: {
    fontSize: 18,
    fontWeight: 'bold',
    marginBottom: 5,
  },
  feedbackTitle: {
    fontSize: 16,
    fontWeight: 'bold',
    marginTop: 10,
    marginBottom: 5,
  },
  errorText: {
    color: 'red',
    marginVertical: 10,
  },
  emptyText: {
    textAlign: 'center',
    marginTop: 20,
    fontSize: 16,
    color: '#666',
  },
});

export default ViewSubmissionsWithFeedback;
```

This code creates a React Native component called ViewSubmissionsWithFeedback that allows users to view submissions along with their feedback. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for submissions, loading state, and errors.
2. The fetchSubmissionsWithFeedback function simulates retrieving submissions with feedback from a database using AsyncStorage. In a real application, you'd replace this with an API call to your backend.
3. The UI includes a button to trigger the submission retrieval and a FlatList to display the submissions with their feedback.
4. Error handling is implemented to show error messages if the fetch fails.
5. A loading indicator (ActivityIndicator) is shown while fetching data.
6. Successful retrieval of submissions updates the state and displays the data in the FlatList.
7. Console logs are used to record successful operations and errors.
8. The FlatList renders each submission with its title, description, and feedback (if available).
9. If there are no submissions with feedback, a message is displayed indicating this.

To use this component, you'll need to install the necessary dependency:

```
npm install @react-native-async-storage/async-storage
```

In a real application, you would replace the AsyncStorage implementation with actual API calls to your backend. You'd also need to ensure that your backend provides both submission and feedback information in a single request or implement multiple requests to fetch this data.

Remember to implement proper error handling and set up comprehensive logging for production use. Also, consider adding pagination if you expect to deal with a large number of submissions.

Stream video submissions:

write React Native code for viewing submissions.

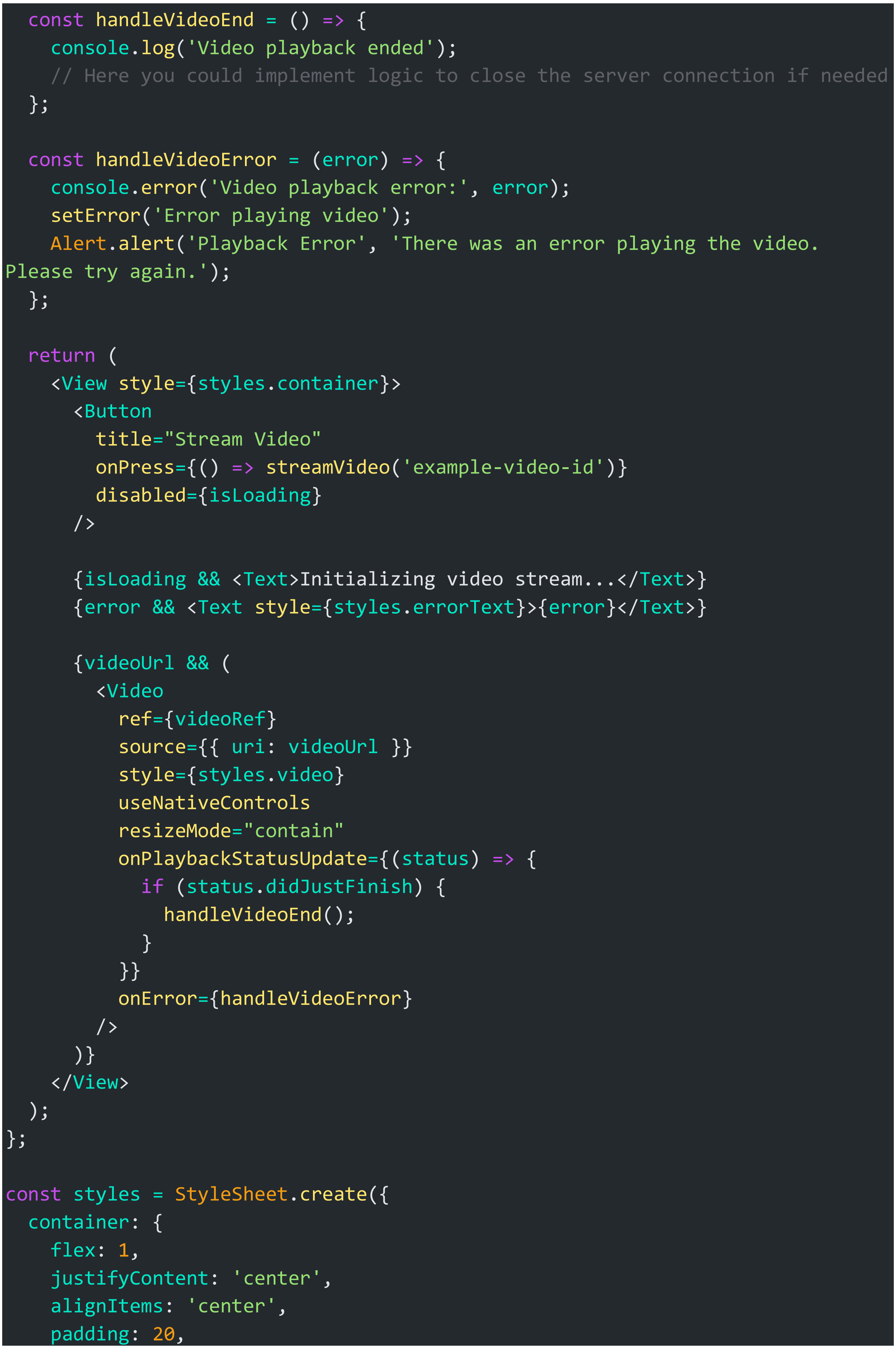
Input: User selects video to stream

Processes: Video stream is initialized, server connection opened, video stream provided, server connection closed

Output if successful: Video stream plays in browser, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState, useRef } from 'react';
import { View, Text, Button, StyleSheet, Alert } from 'react-native';
import { Video } from 'expo-av';

const VideoStreamingScreen = () => {
  const [videoUrl, setVideoUrl] = useState(null);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);
  const videoRef = useRef(null);

  const streamVideo = async (videoId) => {
    setIsLoading(true);
    setError(null);

    try {
      // Simulating server connection and video stream initialization
      const response = await fetch(`https://your-api.com/videos/${videoId}/stream`);
      if (!response.ok) {
        throw new Error('Failed to initialize video stream');
      }

      const streamData = await response.json();
      setVideoUrl(streamData.streamUrl);

      console.log(`Video stream initialized for video ID: ${videoId}`);

    } catch (err) {
      setError('Failed to stream video');
      console.error('Error streaming video:', err);
      Alert.alert('Error', 'Failed to stream video. Please try again.');
    } finally {
      setIsLoading(false);
    }
  };
```



```
  const handleVideoEnd = () => {
    console.log('Video playback ended');
    // Here you could implement logic to close the server connection if needed
  };

  const handleVideoError = (error) => {
    console.error('Video playback error:', error);
    setError('Error playing video');
    Alert.alert('Playback Error', 'There was an error playing the video. Please try again.');
  };

  return (
    <View style={styles.container}>
      <Button
        title="Stream Video"
        onPress={() => streamVideo('example-video-id')}
        disabled={isLoading}
      />

      {isLoading && <Text>Initializing video stream...</Text>}
      {error && <Text style={styles.errorText}>{error}</Text>}

      {videoUrl && (
        <Video
          ref={videoRef}
          source={{ uri: videoUrl }}
          style={styles.video}
          useNativeControls
          resizeMode="contain"
          onPlaybackStatusUpdate={(status) => {
            if (status.didJustFinish) {
              handleVideoEnd();
            }
          }}
          onError={handleVideoError}
        />
      )}
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  video: {
    width: 300,
    height: 200,
    marginTop: 20,
  },
  errorText: {
    color: 'red',
    marginTop: 10,
  },
});

export default VideoStreamingScreen;
```

This code creates a React Native component called VideoStreamingScreen that allows users to stream video submissions. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for the video URL, loading state, and errors.
2. It uses the expo-av library to handle video playback. Make sure to install it

   ```
   expo install expo-av
   ```

3. The streamVideo function simulates initializing a video stream by making an API call. In a real application, you'd replace this with your actual API endpoint.
4. Error handling is implemented to show error messages if the stream initialization fails.
5. The Video component from expo-av is used to play the video once the stream URL is available.
6. The handleVideoEnd function is called when the video playback ends. You can implement any necessary cleanup here.
7. The handleVideoError function is called if there's an error during video playback.
8. Console logs are used to record operations and errors, which you can expand into more comprehensive logging.

To use this component, you'll need to:

1. Install the necessary dependency:

```
expo install expo-av
```

2. Replace 'https://your-api.com/videos/${videoId}/stream' with your actual API endpoint for video streaming.
3. Implement proper error handling and set up comprehensive logging for production use.
4. Consider implementing a way for users to select which video they want to stream, rather than hardcoding the video ID.

5. Ensure your backend is set up to handle video streaming requests and can provide a valid stream URL.

Remember that video streaming can be bandwidth-intensive and may require additional optimizations for performance, especially on mobile devices. You might want to consider things like adaptive bitrate streaming or implementing controls for video quality selection.

Download video submissions:

write React Native code for viewing submissions.

Input: User interacts with a button

Processes: Video file is prepared, server connection opened, video file provided, server connection closed

Output if successful: Success message, video file downloads, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';
import { View, Text, Button, StyleSheet, Alert } from 'react-native';
import * as FileSystem from 'expo-file-system';
import * as Sharing from 'expo-sharing';

const VideoDownloadScreen = () => {
  const [isDownloading, setIsDownloading] = useState(false);
  const [downloadProgress, setDownloadProgress] = useState(0);

  const downloadVideo = async () => {
    setIsDownloading(true);
    setDownloadProgress(0);

    try {
      // Simulating server connection and video file preparation
      const videoUrl = 'https://your-api.com/videos/example-video-
id/download';
      const fileName = 'downloaded_video.mp4';
      const fileUri = FileSystem.documentDirectory + fileName;

      console.log('Server connection opened, preparing video file...');

      const downloadResumable = FileSystem.createDownloadResumable(
        videoUrl,
        fileUri,
        {},
        (downloadProgress) => {
          const progress = downloadProgress.totalBytesWritten /
downloadProgress.totalBytesExpectedToWrite;
          setDownloadProgress(progress);
        }
      );

      const { uri } = await downloadResumable.downloadAsync();
      console.log('Video file downloaded successfully');
      // Simulating server connection closure
      console.log('Server connection closed');

      Alert.alert('Success', 'Video downloaded successfully!');

      // Share the downloaded file
      if (await Sharing.isAvailableAsync()) {
        await Sharing.shareAsync(uri);
      } else {
        Alert.alert('Sharing not available', 'Sharing is not available on this device');
      }

    } catch (error) {
      console.error('Error downloading video:', error);
      Alert.alert('Error', 'Failed to download video. Please try again.');
    } finally {
      setIsDownloading(false);
    }
  };

  return (
    <View style={styles.container}>
      <Button
        title="Download Video"
        onPress={downloadVideo}
        disabled={isDownloading}
      />
      {isDownloading && (
        <View style={styles.progressContainer}>
          <Text>Downloading: {Math.round(downloadProgress * 100)}%</Text>
        </View>
      )}
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  progressContainer: {
    marginTop: 20,
  },
});
```

```
export default VideoDownloadScreen
```

This code creates a React Native component called VideoDownloadScreen that allows users to download video submissions. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for the download progress and loading state.
2. It uses expo-file-system for file operations and expo-sharing for sharing the downloaded file. Make sure to install these libraries:

```
expo install expo-file-system expo-sharing
```

3. The downloadVideo function simulates the process of preparing a video file, opening a server connection, downloading the file, and closing the connection.
4. It uses FileSystem.createDownloadResumable to download the file, which allows for progress tracking.
5. After successful download, it uses Sharing.shareAsync to allow the user to share or open the downloaded file.
6. Error handling is implemented to show error messages if the download fails.
7. A progress indicator is shown during the download process.
8. Console logs are used to record operations and errors, which you can expand into more comprehensive logging.

To use this component, you'll need to:

1. Install the necessary dependencies

```
expo install expo-file-system expo-sharing
```

2. Replace 'https://your-api.com/videos/example-video-id/download' with your actual API endpoint for video downloads.
3. Implement proper error handling and set up comprehensive logging for production use.
4. Consider implementing a way for users to select which video they want to download, rather than hardcoding the video ID.
5. Ensure your backend is set up to handle video download requests and can provide the video file.

Remember that downloading large video files can be data-intensive and may take a while on slower connections. You might want to consider implementing additional features like:

- Allowing users to cancel ongoing downloads
- Implementing resume functionality for interrupted downloads
- Checking available storage space before starting a download

- Providing options for different video qualities or sizes

Also, be mindful of platform-specific behaviors and permissions, especially when it comes to file storage and sharing on iOS and Android.

Provide feedback on video:

write React Native code for viewing submissions.

Input: User provides text and mark feedback on the video

Processes: Feedback and mark data entered into the database

Output if successful: Success message, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert, ScrollView } from 'react-native';
import { Video } from 'expo-av';
import Slider from '@react-native-community/slider';

const FeedbackScreen = ({ videoId }) => {
  const [feedback, setFeedback] = useState('');
  const [mark, setMark] = useState(50);
  const [isSubmitting, setIsSubmitting] = useState(false);

  const submitFeedback = async () => {
    setIsSubmitting(true);
    try {
      // Simulating API call to submit feedback
      const response = await fetch('https://your-api.com/submissions/feedback', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          videoId,
          feedback,
          mark,
        }),
      });

      if (!response.ok) {
        throw new Error('Failed to submit feedback');
      }

      console.log('Feedback submitted successfully');
      console.log(`Video ID: ${videoId}, Feedback: ${feedback}, Mark: ${mark}`);

      Alert.alert('Success', 'Feedback submitted successfully!');
      setFeedback('');
      setMark(50);
    } catch (error) {
      console.error('Error submitting feedback:', error);
      Alert.alert('Error', 'Failed to submit feedback. Please try again.');
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <ScrollView contentContainerStyle={styles.container}>
      <Video
        source={{ uri: `https://your-api.com/videos/${videoId}` }}
        style={styles.video}
        useNativeControls
        resizeMode="contain"
      />

      <TextInput
        style={styles.input}
        multiline
        numberOfLines={4}
        onChangeText={setFeedback}
        value={feedback}
        placeholder="Enter your feedback here"
      />

      <Text style={styles.markLabel}>Mark: {mark}</Text>
      <Slider
        style={styles.slider}
        minimumValue={0}
        maximumValue={100}
        step={1}
        value={mark}
        onValueChange={setMark}
      />

      <Button
        title="Submit Feedback"
        onPress={submitFeedback}
        disabled={isSubmitting}
      />
    </ScrollView>
  );
};

const styles = StyleSheet.create({
  container: {
    flexGrow: 1,
    padding: 20,
    alignItems: 'center',
  },
  video: {
    width: 300,
    height: 200,
    marginBottom: 20,
  },
  input: {
    width: '100%',
    height: 100,
    borderColor: 'gray',
    borderWidth: 1,
    marginBottom: 20,
    padding: 10,
    textAlignVertical: 'top',
  },
  markLabel: {
    fontSize: 16,
    marginBottom: 10,
  },
  slider: {
    width: '100%',
    marginBottom: 20,
  },
});

export default FeedbackScreen;
```

This code creates a React Native component called FeedbackScreen that allows users to view a video submission and provide text feedback and a numerical mark. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for feedback text, mark, and submission status.

2. It uses the expo-av library for video playback and @react-native-community/slider for the mark input. Make sure to install these libraries:

```
expo install expo-av
npm install @react-native-community/slider
```

3. The video is displayed using the Video component from expo-av.
4. A TextInput component is used for text feedback, and a Slider component is used for the numerical mark.
5. The submitFeedback function simulates sending the feedback and mark to an API endpoint. In a real application, you'd replace this with your actual API call.

6. Error handling is implemented to show error messages if the submission fails.
7. A success message is shown when the feedback is submitted successfully.
8. Console logs are used to record operations and errors, which you can expand into more comprehensive logging

To use this component, you'll need to:

1. Install the necessary dependencies

```
   expo install expo-av
npm install @react-native-community/slider
```

2. Replace 'https://your-api.com/submissions/feedback' with your actual API endpoint for submitting feedback.
3. Replace https://your-api.com/videos/${videoId} with the actual URL for fetching the video.
4. Implement proper error handling and set up comprehensive logging for production use.
5. Pass the videoId as a prop when using this component, e.g., <FeedbackScreen videoId="123" />.

Remember to handle things like:

- Input validation (e.g., ensuring feedback isn't empty)
- Proper error messages for different types of failures
- Accessibility considerations for the slider and text input
- Keyboard avoiding behavior, especially on smaller screens

Also, consider adding features like:

- Ability to save draft feedback
- Option to attach timestamps to feedback for specific parts of the video
- Customizable marking criteria or rubrics

Lastly, ensure that your backend is set up to receive and store this feedback data securely and efficiently.

Download Marks:

write React Native code for viewing submissions.

Input: User interacts with a button

Processes: Marks data read from the database, prepared into a file, server connection opened, marks file provided, server connection closed

Output if successful: Success message, Marks file downloads, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';
import { View, Text, Button, StyleSheet, Alert } from 'react-native';
import * as FileSystem from 'expo-file-system';
import * as Sharing from 'expo-sharing';

const MarksDownloadScreen = () => {
  const [isDownloading, setIsDownloading] = useState(false);

  const downloadMarks = async () => {
    setIsDownloading(true);

    try {
      // Simulating server connection and marks data retrieval
      console.log('Opening server connection...');
      console.log('Reading marks data from the database...');

      // Simulating API call to get marks data
      const response = await fetch('https://your-api.com/marks/download');
      if (!response.ok) {
        throw new Error('Failed to retrieve marks data');
      }

      const marksData = await response.json();

      // Prepare marks data into a CSV file
      const csvContent = prepareCSV(marksData);

      // Save CSV content to a file
      const fileName = 'marks_data.csv';
      const fileUri = FileSystem.documentDirectory + fileName;
      await FileSystem.writeAsStringAsync(fileUri, csvContent);

      console.log('Marks file prepared successfully');
      console.log('Server connection closed');

      // Share the downloaded file
      if (await Sharing.isAvailableAsync()) {
        await Sharing.shareAsync(fileUri);
        Alert.alert('Success', 'Marks file downloaded successfully!');
      } else {
        Alert.alert('Sharing not available', 'Sharing is not available on this
device');
      }

      console.log('Marks file download process completed');
    } catch (error) {
      console.error('Error downloading marks:', error);
      Alert.alert('Error', 'Failed to download marks. Please try again.');
    } finally {
      setIsDownloading(false);
    }
  };

  // Helper function to prepare CSV content
  const prepareCSV = (data) => {
    const header = 'Student ID,Name,Mark\n';
    const rows = data.map(item =>
`${item.studentId},${item.name},${item.mark}`).join('\n');
    return header + rows;
  };

  return (
    <View style={styles.container}>
      <Button
        title="Download Marks"
        onPress={downloadMarks}
        disabled={isDownloading}
      />
      {isDownloading && <Text style={styles.loadingText}>Preparing marks
file...</Text>}
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  loadingText: {
    marginTop: 10,
    fontSize: 16,
  },
});

export default MarksDownloadScreen;
```

This code creates a React Native component called MarksDownloadScreen that allows users to download a marks file. Here's a breakdown of the implementation:

1. The component uses React hooks to manage the download state.
2. It uses expo-file-system for file operations and expo-sharing for sharing the downloaded file. Make sure to install these libraries:

    ```
    expo install expo-file-system expo-sharing
    ```

3. The downloadMarks function simulates the process of:
    - Opening a server connection
    - Reading marks data from the database
    - Preparing the data into a CSV file
    - Saving the file locally
    - Closing the server connection
    - Sharing the file with the user
4. The prepareCSV helper function converts the marks data into a CSV format.
5. Error handling is implemented to show error messages if the download fails.
6. Console logs are used to record operations and errors, which you can expand into more comprehensive logging.

To use this component, you'll need to:

1. Install the necessary dependencies:

```
expo install expo-file-system expo-sharing
```

2. Replace 'https://your-api.com/marks/download' with your actual API endpoint for retrieving marks data.
3. Implement proper error handling and set up comprehensive logging for production use.
4. Ensure your backend is set up to handle marks data requests and can provide the data in a suitable format.

Remember to consider the following:

- Implement proper authentication and authorization to ensure only authorized users can download marks data.

- Handle large datasets efficiently, possibly implementing pagination or streaming for very large mark sets.

- Provide options for different file formats (e.g., CSV, Excel) if needed.
- Implement error handling for various scenarios (network issues, file system errors, etc.).
- Consider adding a progress indicator for larger files.

Also, be mindful of data privacy and security concerns when handling and sharing student marks. Ensure that all data handling complies with relevant education and privacy regulations.

Lastly, you might want to add features like:

- Allowing users to select specific classes or date ranges for marks
- Providing a preview of the marks before download
- Implementing a caching mechanism for frequently accessed marks data

Secure Login:

write React Native code for viewing submissions.

Input: User provides username and password

Processes: Password is hashed, username and hashed password is verified, token is generated on success, token returned

Output if successful: Success message, user token, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert } from 'react-native';
import * as SecureStore from 'expo-secure-store';
import { SHA256 } from 'crypto-js';

const LoginScreen = ({ onLoginSuccess }) => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [isLoggingIn, setIsLoggingIn] = useState(false);

  const hashPassword = (password) => {
    return SHA256(password).toString();
  };

  const login = async () => {
    if (!username || !password) {
      Alert.alert('Error', 'Please enter both username and password');
      return;
    }

    setIsLoggingIn(true);

    try {
      const hashedPassword = hashPassword(password);

      console.log('Verifying username and hashed password...');

      // Simulating API call for authentication
      const response = await fetch('https://your-api.com/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          username,
          password: hashedPassword,
        }),
      });

      if (!response.ok) {
        throw new Error('Authentication failed');
      }

      const data = await response.json();

      if (data.token) {
        console.log('Authentication successful, token generated');

        // Store the token securely
        await SecureStore.setItemAsync('userToken', data.token);

        console.log('Token stored securely');
        Alert.alert('Success', 'Login successful!');

        // Call the onLoginSuccess callback with the token
        onLoginSuccess(data.token);
      } else {
        throw new Error('No token received');
      }

    } catch (error) {
      console.error('Login error:', error);
      Alert.alert('Error', 'Login failed. Please try again.');
    } finally {
      setIsLoggingIn(false);
    }
  };

  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder="Username"
        value={username}
        onChangeText={setUsername}
        autoCapitalize="none"
      />
      <TextInput
        style={styles.input}
        placeholder="Password"
        value={password}
        onChangeText={setPassword}
        secureTextEntry
      />
      <Button
        title={isLoggingIn ? "Logging in..." : "Login"}
        onPress={login}
        disabled={isLoggingIn}
      />
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  input: {
    width: '100%',
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    marginBottom: 10,
    paddingHorizontal: 10,
  },
});

export default LoginScreen;
```

This code creates a React Native component called LoginScreen that handles user authentication. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for username, password, and login status.
2. It uses expo-secure-store for securely storing the user token and crypto-js for password hashing. Make sure to install these libraries:

```
expo install expo-secure-store
npm install crypto-js
```

3. The hashPassword function uses SHA256 to hash the password.
4. The login function handles the login process:
    1. It first checks if both username and password are provided.
    2. It then hashes the password.
    3. It simulates an API call to authenticate the user.
    4. If successful, it stores the received token securely and calls the onLoginSuccess callback.

5. Error handling is implemented to show error messages if the login fails.
6. Console logs are used to record operations and errors, which you can expand into more comprehensive logging.

To use this component, you'll need to:

1. Install the necessary dependencies:

```
expo install expo-secure-store
npm install crypto-js
```

2. eplace 'https://your-api.com/login' with your actual API endpoint for user authentication.
3. Implement proper error handling and set up comprehensive logging for production use.
4. Use this component in your app, passing an onLoginSuccess callback:

```
use this component in app, passing an onLoginSucess callback
<LoginScreen onLoginSuccess={(token) => {
  // Handle successful login, e.g., navigate to main app screen
  console.log('Logged in with token:', token);
}} />
```

Remember to consider the following:

- Implement proper input validation (e.g., email format, password strength).
- Use HTTPS for all network requests to ensure data security.
- Consider implementing features like "Remember Me" or biometric authentication.
- Handle different types of authentication errors (e.g., wrong password, user not found) with specific error messages.
- Implement a logout function that clears the stored token.
- Consider adding a loading indicator during the login process.

Also, be mindful of security best practices:

- Never store passwords in plain text, either in the app or on the server.
- Use secure, httpOnly cookies for storing tokens on the server-side if possible.
- Implement token expiration and refresh mechanisms.
- Consider implementing multi-factor authentication for enhanced security.

Lastly, ensure that your backend is set up to handle authentication requests securely, validate credentials, and generate and manage tokens properly.

User Admin:

write React Native code for viewing submissions.

Input: User Administration

Processes: User information is updated

Output if successful: Success message, log-file entries

Output if failure: Failure message, log-file entries

```
import React, { useState, useEffect } from 'react';
import { View, Text, TextInput, Button, StyleSheet, Alert, ScrollView,
ActivityIndicator } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';

const UserAdminScreen = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [updating, setUpdating] = useState(false);

  useEffect(() => {
    fetchUsers();
  }, []);

  const fetchUsers = async () => {
    try {
      const response = await fetch('https://your-api.com/users');
      if (!response.ok) {
        throw new Error('Failed to fetch users');
      }
      const data = await response.json();
      setUsers(data);
    } catch (error) {
      console.error('Error fetching users:', error);
      Alert.alert('Error', 'Failed to fetch users. Please try again.');
    } finally {
      setLoading(false);
    }
  };

  const updateUser = async (userId, updatedInfo) => {
    setUpdating(true);
    try {
      const token = await AsyncStorage.getItem('userToken');
      const response = await fetch(`https://your-api.com/users/${userId}`, {
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Bearer ${token}`,
        },
        body: JSON.stringify(updatedInfo),
      });

      if (!response.ok) {
        throw new Error('Failed to update user');
      }

      console.log(`User ${userId} updated successfully`);
      Alert.alert('Success', 'User information updated successfully!');

      // Refresh the user list
      await fetchUsers();
    } catch (error) {
      console.error('Error updating user:', error);
      Alert.alert('Error', 'Failed to update user information. Please try
again.');
    } finally {
      setUpdating(false);
    }
  };

  const UserItem = ({ user }) => {
    const [name, setName] = useState(user.name);
    const [email, setEmail] = useState(user.email);

    const handleUpdate = () => {
      updateUser(user.id, { name, email });
    };

    return (
      <View style={styles.userItem}>
        <TextInput
          style={styles.input}
          value={name}
          onChangeText={setName}
          placeholder="Name"
        />
        <TextInput
          style={styles.input}
          value={email}
          onChangeText={setEmail}
          placeholder="Email"
          keyboardType="email-address"
        />
        <Button title="Update" onPress={handleUpdate} disabled={updating} />
      </View>
    );
  };

  if (loading) {
    return (
      <View style={styles.centered}>
        <ActivityIndicator size="large" />
      </View>
    );
  }

  return (
    <ScrollView contentContainerStyle={styles.container}>
      <Text style={styles.title}>User Administration</Text>
      {users.map(user => (
        <UserItem key={user.id} user={user} />
      ))}
    </ScrollView>
  );
};

const styles = StyleSheet.create({
  container: {
    padding: 20,
  },
  centered: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    marginBottom: 20,
  },
  userItem: {
    marginBottom: 20,
    padding: 10,
    borderWidth: 1,
    borderColor: '#ddd',
    borderRadius: 5,
  },
  input: {
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    marginBottom: 10,
    paddingHorizontal: 10,
  },
});

export default UserAdminScreen;
```

This code creates a React Native component called UserAdminScreen that allows administrators to view and update user information. Here's a breakdown of the implementation:

1. The component uses React hooks to manage state for users, loading, and updating status.
2. It uses @react-native-async-storage/async-storage for retrieving the authentication token. Make sure to install this library:

```
npm install @react-native-async-storage/async-storage
```

3. The fetchUsers function retrieves the list of users from the API when the component mounts.
4. The updateUser function handles updating user information:
    1. It sends a PUT request to the API with the updated user information.
    2. It uses the stored authentication token for authorization.
    3. On success, it refreshes the user list.
5. The UserItem component renders individual user information and provides inputs for editing.
6. Error handling is implemented to show error messages if fetching or updating fails.
7. A loading indicator is shown while fetching users.
8. Console logs are used to record operations and errors, which you can expand into more comprehensive logging.

To use this component, you'll need to:

1. Install the necessary dependency:

```
npm install @react-native-async-storage/async-storage
```

2. Replace 'https://your-api.com/users' and https://your-api.com/users/${userId} with your actual API endpoints for fetching and updating users.
3. Implement proper error handling and set up comprehensive logging for production use.
4. Ensure that your authentication system is properly set up and the token is stored in AsyncStorage after login.

Remember to consider the following:

- Implement proper input validation for name and email fields.
- Add pagination or infinite scrolling for large user lists.
- Implement search and filter functionality for easier user management.
- Add confirmation dialogs before updating user information.
- Implement role-based access control to ensure only authorized administrators can access this screen.
- Consider adding more user management features like adding new users or deactivating accounts.

Also, be mindful of security and privacy concerns:

- Ensure all API requests are made over HTTPS.
- Implement proper server-side validation and authorization checks.
- Be cautious about what user information is displayed and editable.
- Consider implementing audit logs for user information changes.

Lastly, you might want to add features like:

- Ability to reset user passwords
- Viewing user activity logs
- Managing user roles and permissions

Remember to thoroughly test this functionality and ensure it complies with relevant data protection regulations.