React Native Development Tasks

Sai Kumar Reddy Kaluvakolu

ID: 001258100
Repo Link: https://github.com/sai-gif/Todo

Task 1

Q1.1 : Attach screenshots of your app running on an emulator and on a physical Android or iOS device.

A:

- All the images have been uploaded at the end of the document

Q1.2: Describe any differences you observed between running the app on an emulator versus a physical device.

A:

- **Emulator:**
  - Setup takes a lot of time and requires significant computational resources.
  - Allows testing multiple devices.
- **Physical Device:**
  - Real-world performance testing.
  - Limited to one device at a time.

Q2.1: Explain the steps you followed to set up an emulator in Android Studio or Xcode.

A:

1. Installed Java and set the system path.
2. Installed android studio and setup sdk
3. Used Android Studio's AVD Manager to create an emulator.
4. Created a new React Native project:
   - Attempted `npm install -g react-native-cli` (deprecated and did not work).
   - Used updated steps from the React Native documentation:

     ```
     npx @react-native-community/cli@latest init todoapp
     ```

5. Configured Android environment variables such as `ANDROID_HOME` along with other env involved.
6. Resolved Gradle issues with the following commands:

   ```
   ./gradlew clean
   ./gradlew assembleDebug --info
   npx react-native start --reset-cache
   ```

7. Ran the application with npm run android

Q2.2: Discuss any challenges you faced during the setup and how you overcame them.

A:

- **Challenges:**
  - `react-native-cli` was deprecated, and the professor's commands did not work.
  - Missing Android environment variables caused `npm run android` to fail.
  - The emulator did not launch automatically and threw errors.
- **Resolutions:**
  - Used updated CLI commands from React Native documentation.
  - Set environment variables (`ANDROID_HOME`, `PATH`) using tutorials.
  - Manually launched the emulator and cleared Gradle caches.

Q3.1: Describe how you connected your physical device to run the app using Expo.

A:

- Installed the Expo app from the Play Store.
- ran react native application using expo Framework
- Scanned the QR code provided in the terminal after running the app.
- Followed the Expo tutorial for a seamless setup without issues:
- https://docs.expo.dev/tutorial/create-your-first-app/ click.

Q3.2: Include any troubleshooting steps if you encountered issues.

A:

- Fortunately, everything was smooth when setting up the app on Expo.

Q4.1 and 4.2: Compare and contrast using an emulator versus a physical device for React Native development.

A:

- Set up takes a lot of time on an emulator and requires significant computational resources. Using an emulator, we can test multiple devices, whereas physical devices are limited to one.
- **Emulator:**
  - **Advantages:** Cost-effective, supports multiple devices, built-in debugging tools.
  - **Disadvantages:** Slower performance, limited hardware testing, less accurate representation.
- **Physical Device:**
  - **Advantages:** Real-world performance testing, accurate hardware, and sensor usage.
  - **Disadvantages:** Requires physical devices, less convenient for testing multiple OS versions.

Q 5: Identify a common error you encountered when starting your React Native app. Explain the cause of the error and the steps you took to resolve it.

A:

- **Error Encountered:**
  - `npm install -g react-native-cli` is deprecated, and the commands provided by the professor did not work.
  - Running `npm run android` failed due to missing Android environment variables.
  - The emulator failed to launch and caused build errors.
- **Cause of the Errors:**
  - **Deprecated CLI Installation:** The `react-native-cli` is no longer supported, and the latest community CLI must be used.
  - **Missing Environment Variables:** The Android development environment was not properly configured (e.g., `ANDROID_HOME`, `PATH` variables missing).
  - **Gradle Build Issues:** Stale caches and leftover build artifacts caused build failures.
  - **Emulator Launch Issues:** The emulator did not launch automatically and required manual intervention.
- **Steps Taken to Resolve the Errors:**
  1. Used the updated CLI commands from React Native documentation:

     ```
     npx @react-native-community/cli@latest init todoapp
     ```

  2. Configured environment variables:

     ```
     export ANDROID_HOME=/path/to/Android/Sdk
     export PATH=$PATH:$ANDROID_HOME/emulator
     export PATH=$PATH:$ANDROID_HOME/platform-tools
     ```

  3. Cleaned and rebuilt the project using Gradle:

     ```
     ./gradlew clean
     ./gradlew assembleDebug --info
     ```

  4. Manually started the emulator using Android Studio or terminal commands:

     ```
     emulator -avd YourEmulatorName
     ```

  5. Reset the Metro bundler cache to ensure a clean start:

     ```
     npx react-native start --reset-cache
     ```

Task 2

Q2.1: How did you implement marking tasks as complete?

A:

- Modified task objects to include a `completed` property:

```
{ id: Date.now().toString(), text: task, completed: false }
```

- Created a custom checkbox component.
- Styled completed tasks with strikethrough and grey text using:

```
textDecorationLine: 'line-through', color: 'grey'
```

Q2.2: How did you implement data persistence using AsyncStorage?

A:

- Used `AsyncStorage` to save tasks whenever the state changed. This was achieved using the `useEffect` hook to monitor changes in the `tasks` state:

```
useEffect(() => {
    saveTasks(tasks);
}, [tasks]);
```

- The `saveTasks` function serialized the tasks to JSON and stored them using `AsyncStorage.setItem`:

```
try {
    const jsonValue = JSON.stringify(tasks);
    await AsyncStorage.setItem('tasks', jsonValue);
} catch (e) {
    console.error("Failed to save tasks.");
}
```

- Loaded tasks when the page was first rendered by retrieving them with `AsyncStorage.getItem`:

```
const loadTasks = async () => {
    try {
        const jsonValue = await AsyncStorage.getItem('tasks');
        setTasks(jsonValue != null ? JSON.parse(jsonValue) : []);
    } catch (e) {
        console.error("Failed to load tasks.");
    }
};
```

- Addressed an edge case where no tasks were stored in `AsyncStorage`. This was handled by checking for `null` and returning an empty array:

```
const loadTasks = async () => {
    try {
        const jsonValue = await AsyncStorage.getItem('tasks');
        setTasks(jsonValue != null ? JSON.parse(jsonValue) : []);
    } catch (e) {
        console.error("Failed to load tasks.");
    }
};
```

Q2.3: How did you implement task editing?

A:

- Allowed users to edit a task's content by tapping on it. Only one task can be edited at a time, ensuring clarity and focus during the editing process.

- Managed the UI for editing tasks by conditionally rendering a `TextInput` in place of the task text when a task is in editing mode. The specific task is identified using its `id`, and the `editingTaskId` and `editingText` states are updated accordingly.
- Rendered the editing interface using the following component structure:

```
<View style={styles.editContainer}>
    <TextInput
        style={styles.input}
        value={editingText}
        onChangeText={setEditingText}
        onBlur={cancelEditTask}  % Cancels editing if clicked outside
    />
    <TouchableOpacity onPress={() => confirmEditTask(item.id)}>
        <Text style={styles.okButton}>OK</Text>  % Confirms changes
    </TouchableOpacity>
</View>
```

- Implemented an update function to modify the task in the state array. The function updates the specific task and clears the editing mode:

```
const confirmEditTask = (id) => {
    setTasks(tasks.map(task =>
        task.id === id ? { ...task, text: editingText } : task
    ));
    setEditingTaskId(null);
    setEditingText('');
};
```

- Handled two primary actions:
  - **Select:** When a task is tapped, it enters editing mode, and the `TextInput` is displayed with the task's current text pre-filled.
  - **Unselect:** Clicking outside the `TextInput` triggers the `onBlur` event, which discards any changes and exits editing mode.
- Key interactions:
  - **OK Button:** Explicitly saves the updated task content and exits editing mode.
  - **Click Outside (Blur):** Cancels the editing and resets the `editingTaskId` and `editingText` states.
- This approach ensures that task edits are controlled, changes are only saved when explicitly confirmed by the user, and accidental modifications are prevented.

Q2.4: Describe the animations you implemented.

A:

- Used the Animated API to create fade-out effects when deleting tasks:

```
const fadeOut = new Animated.Value(1);
Animated.timing(fadeOut, {
    toValue: 0,
    duration: 300,
    useNativeDriver: true,
}).start(() => {
    // Remove task from the list
});
```

- **Key Benefits:**
  - Smooth transitions.
  - Enhanced user experience.
  - Reduced cognitive load.

Disclosure

During the setup and development of the React Native app, I utilized AI tools to assist in troubleshooting and resolving errors. These tools were instrumental in identifying updated commands, configuring environment variables, and addressing issues such as:

- Resolving deprecated commands for setting up the React Native environment.
- Configuring Android emulator environment variables.
- Debugging Gradle build errors and emulator launch issues.
- Providing guidance on styling components, such as creating a custom checkbox.

The AI support helped streamline the debugging process and ensured a smoother development experience.

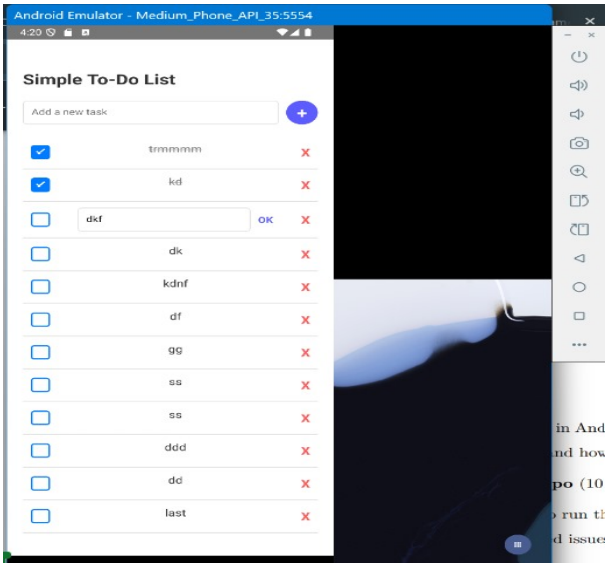

Figure 1: App running on an Android emulator.

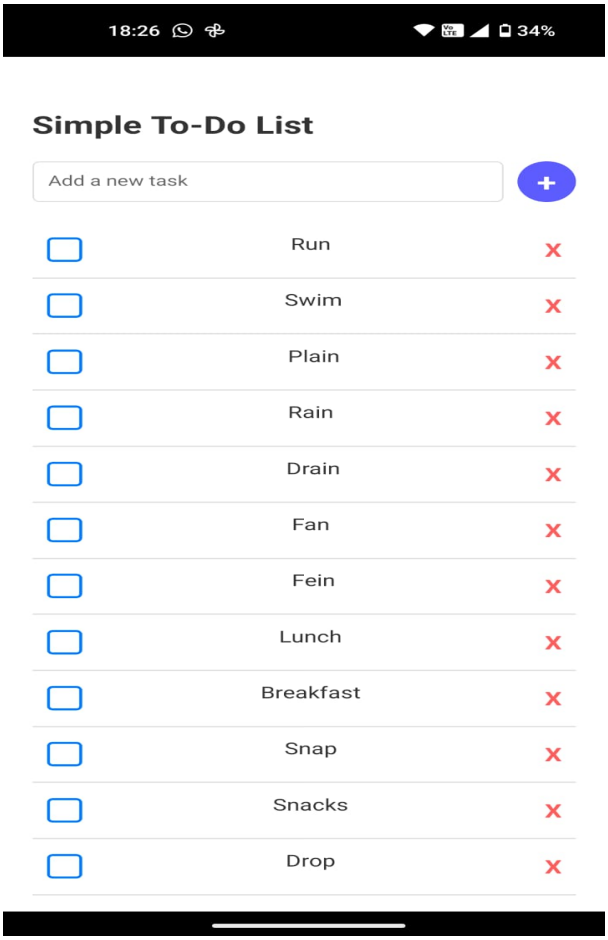

Figure 2: App running on an Android physical device.

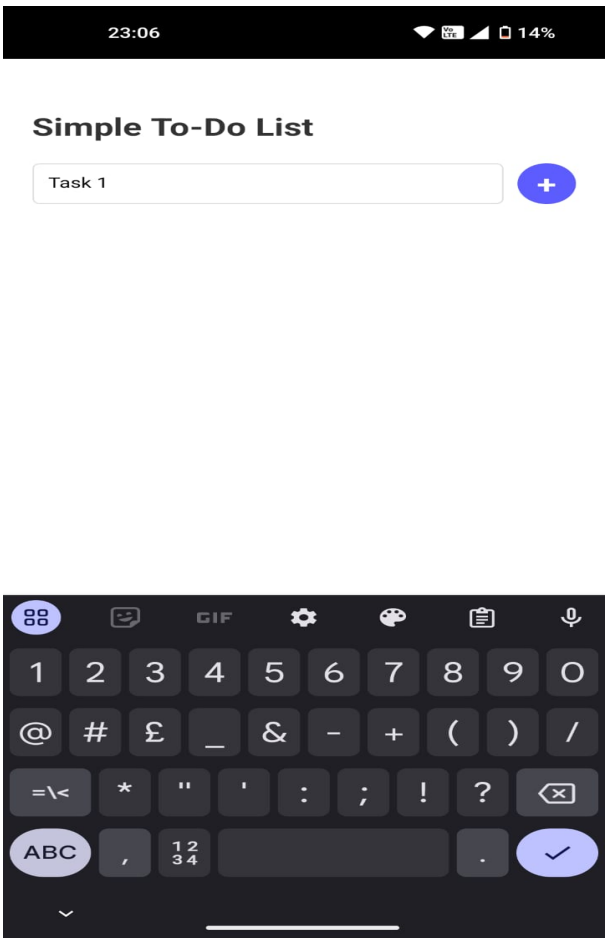

Figure 3: Add tasks in the app

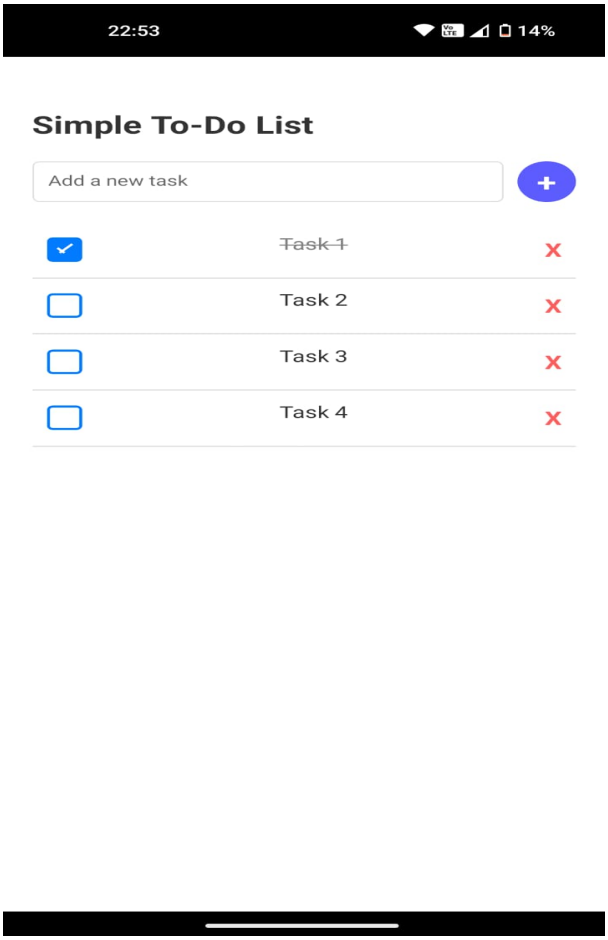

Figure 4: completing tasks in the app

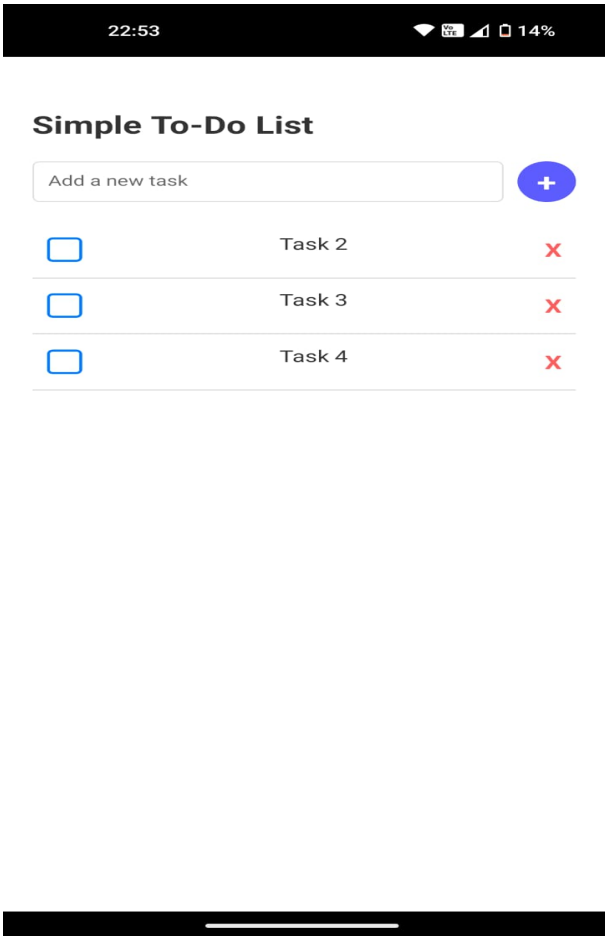

Figure 5: remove tasks in the app