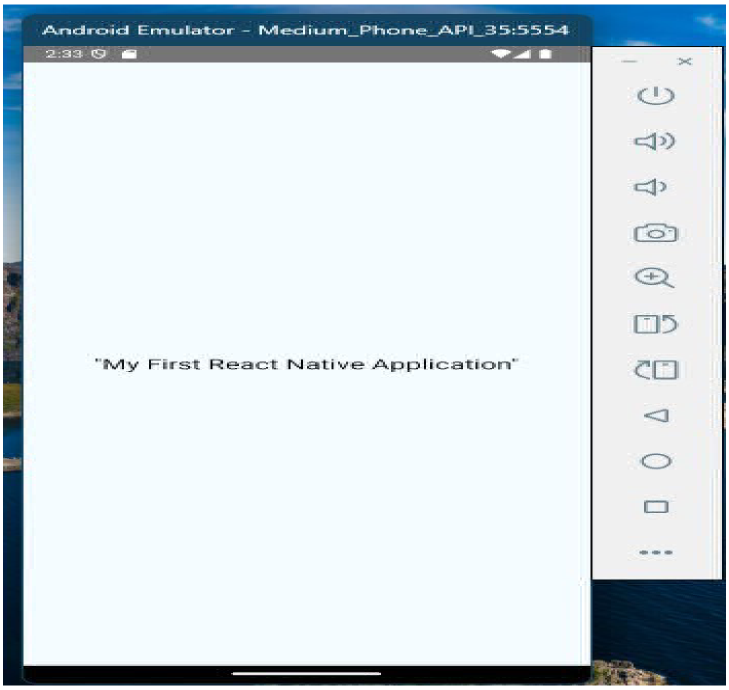
Lab 3: Building a Mobile App with React Native

Charitha Vennapusala

November 17th 2024

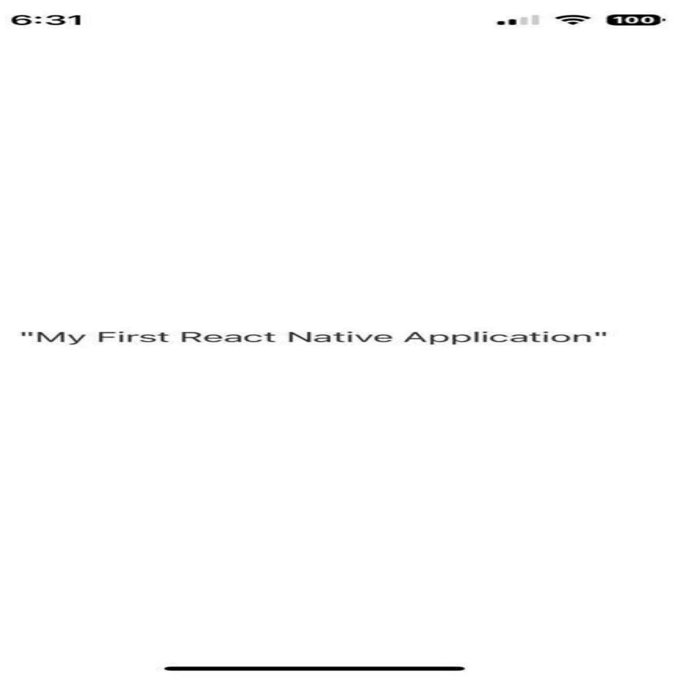
Task 1: Set Up the Dev Environment

(1) Screenshots of Your App



"My First React Native Application"

Figure 1: App running in emulator



"My First React Native Application"

Figure 2: App running on a physical device

Differences Observed

On Emulator:

- Running the app on the Android emulator was quite smooth.
- The emulator simulates the device environment and provides a close approximation of how the app would look on a real device.
- However, it tends to be slower, especially during startup, and performance might not be as smooth compared to a physical device.
- The app displayed correctly but took longer to start due to the emulator's initialization time.
- UI elements looked consistent with how they were coded but might not reflect actual device performance.
- Testing was limited to simulated gestures and touch interactions, which may not accurately represent real-world user behavior.
- Certain hardware-specific features like GPS and accelerometer are simulated but may not provide reliable results.
- The emulator allowed for easy switching between different screen sizes and resolutions, which is useful for testing layouts on various devices.

On Physical Device:

- Running the app on the physical device was significantly faster.
- The app was more responsive, and it felt more like an actual native app.

- Expo Go provides a faster, more real-world representation of how the app behaves.
- The performance was better, and there were no delays or lag, unlike the emulator.
- The real device allowed me to test actual touch interactions and gestures, which felt more natural compared to the emulator.
- Real-world testing on a physical device made it possible to evaluate battery usage and power efficiency, which cannot be simulated in an emulator.
- It was easier to verify features like camera integration, real GPS data, and push notifications on the physical device.
- Debugging real-world issues like connectivity (Wi-Fi/mobile data) was more accurate on a physical device compared to the emulator's simulated environment.
- The display on the physical device provided a true representation of screen resolution, brightness, and color accuracy, which are often scaled on the emulator.

(2) Setting Up an Emulator

Steps to Set Up the Emulator:

1. **Install Android Studio:** Download and install Android Studio from the official website (`https://developer.android.com/studio`).
2. **Install Android SDK Tools:**
   - Open Android Studio and navigate to `Settings > Appearance & Behavior > System Settings > Android SDK`.
   - Select the **SDK Platforms** tab and install the latest Android version (e.g., Android 15, API Level 35).
   - Switch to the **SDK Tools** tab and install the following tools:
     - Android SDK Build-Tools
     - Android Emulator
     - Android Emulator hypervisor driver
     - Android SDK Platform-Tools
     - Intel x86 Emulator Accelerator (HAXM)
   - Click **Apply** to download and install these components.
3. **Set Up an Android Virtual Device (AVD):**
   - Open the AVD Manager, accessible via `Welcome Screen > Configure > AVD Manager`, or from the main interface under `Tools > AVD Manager`.
   - Create a new virtual device:
     - Device: Select **Medium Phone**.
     - System Image: Choose a system image for API Level 35.
   - Configure the AVD:

3.
    - Adjust settings to allocate sufficient resources.
    - Enable hardware acceleration for better performance.

4. **Start the Emulator:** Launch the emulator using the **Play** button in the AVD Manager. Wait for the emulator to boot up completely.

Challenges Faced and Solutions:

- **Slow Performance:**
    - The emulator was initially slow, leading to long app load times.
    - Enabling hardware acceleration through the Intel HAXM installer resolved the issue.
- **System Image Issues:**
    - Encountered problems downloading the correct system images.
    - Resolved by ensuring the latest version of Android Studio and proper SDK updates were installed.
- **Internet Access Issues:**
    - At one point, the emulator couldn't connect to the internet.
    - Restarting the emulator and checking the AVD network settings fixed the problem.

(3) Running the App on a Physical Device Using Expo

Steps to Connect the Physical Device:

1. **Set Up Expo:**
    - Installed the Expo CLI globally using:

      ```
      npm install -g expo-cli
      ```

    - Created a new Expo project:

      ```
      npx expo init ExpoProject
      cd ExpoProject
      npx expo start
      ```

    - This launched the Expo developer tools in the browser.
2. **Install Expo Go on the Physical Device:**
    - Downloaded and installed the Expo Go app from the Google Play Store on my Android device.
3. **Connect Both Devices:**

- Ensured that my development machine and physical Android device were connected to the same Wi-Fi network.

4. **Scan the QR Code:**
    - Opened the Expo Go app and used it to scan the QR code displayed in the Expo developer tools on the browser.
    - The app loaded automatically on the physical device.

5. **Modify and Test the App:**
    - Made changes to the `App.js` file to display "My First React Native Application".
    - Observed that changes were reflected instantly in the Expo Go app without requiring a rebuild.

Troubleshooting Steps for Physical Device Setup:

- **App Not Loading:**
    - Restarted the Expo server with:

      ```
      npx expo start --clear
      ```

    - Ensured the Expo Go app was up to date and reinstalled it if necessary.
- **Network Issues:**
    - Verified that both devices were on the same Wi-Fi network.
    - Restarted the router to resolve any connectivity issues.
    - Used LAN or USB debugging when the Wi-Fi network was unreliable.
- **Cache Problems:**
    - Cleared the Metro bundler cache:

      ```
      npx expo start --clear
      ```

    - Deleted the `node_modules` folder and reinstalled dependencies using:

      ```
      npm install
      ```

- **Performance Optimization:**
    - Disabled unnecessary processes and apps on the physical device to ensure smooth app performance.

(4) Comparison of Emulator vs. Physical Device

Comparison between Emulator and Physical Device

Aspect	**Emulator**	**Physical Device**
Performance	Slower and less responsive.	Faster and more accurate to real-world performance.
Touch Interaction	Simulated touch gestures may not reflect actual user behavior.	Real touch interactions provide more reliable testing of user experience.
Setup	Requires installation of Android Studio and setup of AVD.	Requires only the Expo Go app and a shared Wi-Fi connection.
Accessibility	Useful for testing various screen sizes and API levels without needing multiple devices.	Limited to the physical device you have access to.
Battery/Hardware Testing	No real hardware interactions, such as battery or camera testing.	Allows testing real-world hardware functionality like sensors, camera, and GPS.
Convenience	Available on the development machine, eliminating the need for additional hardware.	Requires a physical device to be available at all times.

Table 1: Comparison between Emulator and Physical Device

Emulator: Advantages:

- Provides a close simulation of the device environment, useful for testing different device sizes and Android versions.
- Easy to reset and configure with various hardware profiles and system images.
- Can be faster for certain development tasks as the app is launched in a controlled environment.

Disadvantages:

- Performance is slower, especially on lower-end computers, and can be resource-intensive.
- May not always reflect real-world app performance due to the lack of actual hardware interactions.

Physical Device: Advantages:

- Offers real-world performance and behavior, including real-time interactions with hardware components like GPS, camera, and sensors.
- Provides more accurate testing for responsiveness and app fluidity.

Disadvantages:

- Requires a physical device that is available and connected via USB or over Wi-Fi.
- May not be practical to test every potential device or screen size.

(5)Troubleshooting a Common Error

Error Encountered

When I first ran the Expo project on my Android emulator, the app didn't load, and I got the following error message in the terminal:

> **Error:** Unable to connect to development server.

Cause

This issue was caused by the emulator not being properly connected to the development server due to network issues or a misconfigured Expo environment.

Steps to Resolve

1. **Check Network Connection:** I ensured that both my development machine and the Android emulator were connected to the same network.
2. **Restart Expo Server:** Running `npx expo start --clear` helped clear the cache and resolve any server-side issues.
3. **Reset Metro Bundler Cache:** Restarting the Metro bundler with `npx react-native start --reset-cache` cleared stale files and reestablished the development environment.
4. **Verify Emulator Settings:**
    - I checked the Android Emulator settings to confirm it had internet access.
    - Restarting the emulator helped establish a stable connection.

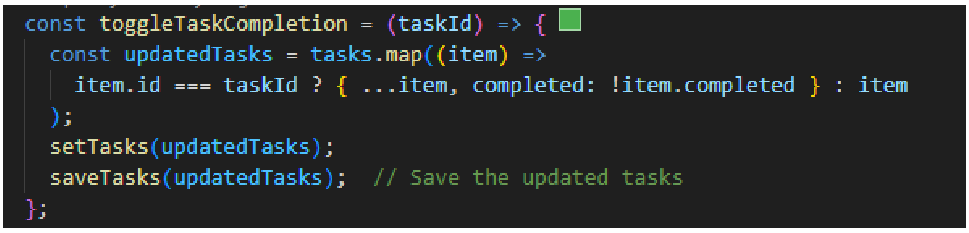
Task 2: Building a Simple To-Do List App

(a) Mark Tasks as Complete

Implementation: A toggle function (`toggleTaskCompletion`) was added to mark tasks as completed or not.

- When the user taps on the checkmark, the task's completed status is toggled. This triggers a state update and modifies the task's appearance, reflecting the completion status.
- Tasks marked as complete will have a strikethrough on the text (`textDecorationLine: 'line-through'`) and their color changes to gray (`color: #808080`) to indicate completion.

Screenshots for toggling task completion:



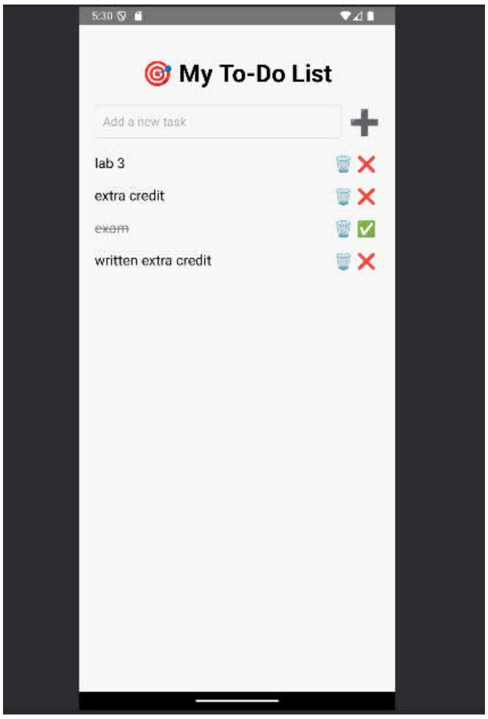
```
const toggleTaskCompletion = (taskId) => {
  const updatedTasks = tasks.map((item) =>
    item.id === taskId ? { ...item, completed: !item.completed } : item
  );
  setTasks(updatedTasks);
  saveTasks(updatedTasks);  // Save the updated tasks
};
```

Figure 3: Screenshot showing the initial list of tasks.

Explanation:

- The toggleTaskCompletion function maps over the tasks, checking each task's id. When a match is found, it toggles the completed property.
- The UI updates automatically because the state is modified and passed into the component's render method, triggering a re-render.

Visual Outcome: Completed tasks will be displayed with a strikethrough, and their color changes to gray (#808080) to indicate they are marked as completed.



🎯 My To-Do List

lab 3 🗑️ ❌

extra credit 🗑️ ❌

~~exam~~ 🗑️ ✅

written extra credit 🗑️ ❌

Figure 4: Screenshot showing a task marked as completed with strikethrough and gray text.

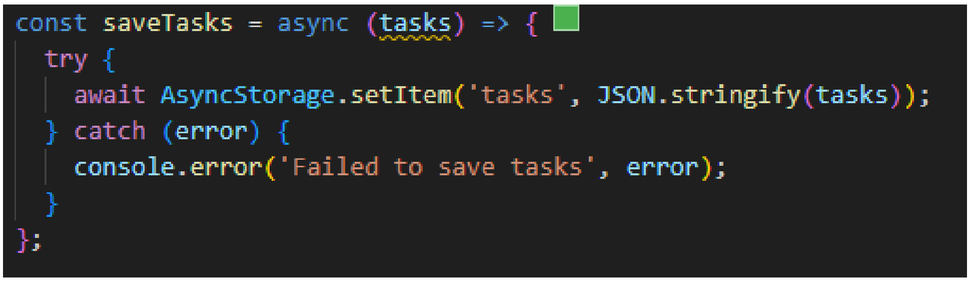
(b) Persist Data Using AsyncStorage

Implementation:

- `AsyncStorage` was used to persist the list of tasks, ensuring that they are saved across app restarts.
- Whenever the tasks array is updated (adding, deleting, or toggling tasks), the updated tasks are saved in `AsyncStorage`.

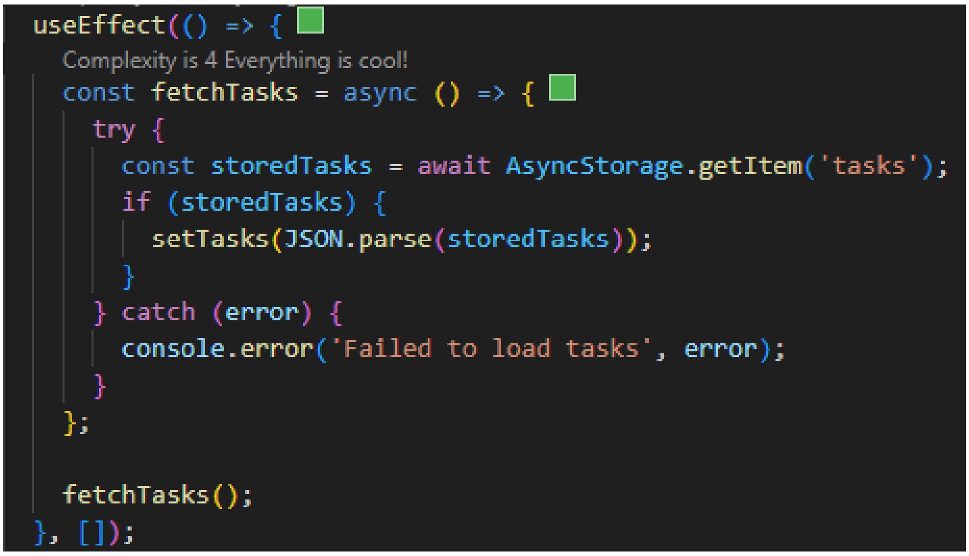
- On app launch, tasks are retrieved from `AsyncStorage` and loaded into the state.

Screenshots for saving and loading tasks:



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

Figure 5: Screenshot showing the tasks being saved.



```
useEffect(() => {
  Complexity is 4 Everything is cool!
  const fetchTasks = async () => {
    try {
      const storedTasks = await AsyncStorage.getItem('tasks');
      if (storedTasks) {
        setTasks(JSON.parse(storedTasks));
      }
    } catch (error) {
      console.error('Failed to load tasks', error);
    }
  };

  fetchTasks();
}, []);
```

Figure 6: Screenshot showing tasks loaded from AsyncStorage when app start.

Explanation:

- Saving Tasks: Each time the task list changes (add, delete, or update), the tasks are stored in AsyncStorage.
- Fetching Tasks: Upon app launch or reload, tasks are fetched from AsyncStorage and set in the state, preserving the user's data.
- On app launch, tasks are retrieved from `AsyncStorage` and loaded into the state.

Visual Outcome: The tasks will persist across app restarts, maintaining the user's list even if the app is closed and reopened.

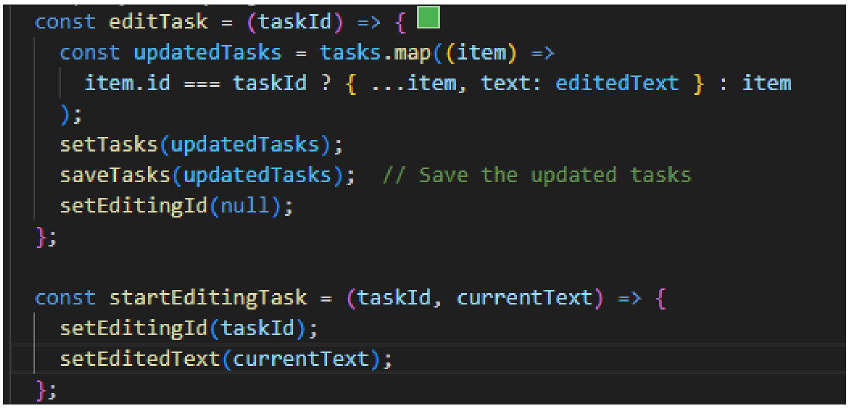
(c) Edit Tasks

Implementation:

- Allow users to tap on a task, which will enable an input field for editing the task's text.

- After editing, users can submit the changes by pressing Enter.
- The state is updated with the modified text, and the updated list of tasks is saved back to `AsyncStorage`.

Screenshots for editing tasks:



```
const editTask = (taskId) => {
  const updatedTasks = tasks.map((item) =>
    item.id === taskId ? { ...item, text: editedText } : item
  );
  setTasks(updatedTasks);
  saveTasks(updatedTasks);  // Save the updated tasks
  setEditingId(null);
};

const startEditingTask = (taskId, currentText) => {
  setEditingId(taskId);
  setEditedText(currentText);
};
```

Figure 7: Screenshot showing a task in editing mode.

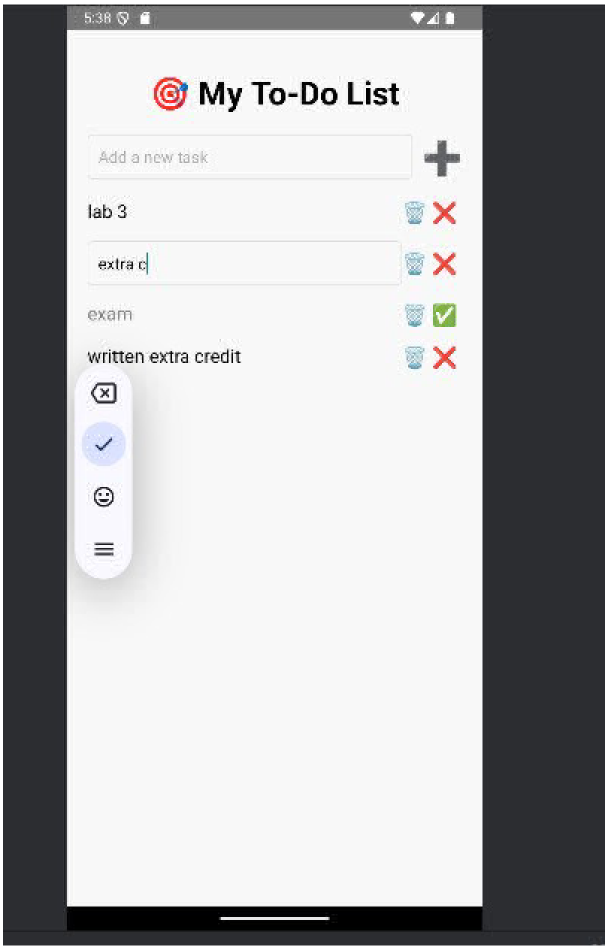


Figure 8: Screenshot showing the edited task with updated text.

Explanation:

- When the user taps on a task, the startEditingTask function is called, which sets the editingId and shows an input field with the current task text.
- After editing the text, the editTask function is triggered to update the task in the state array, and the updated tasks list is saved.

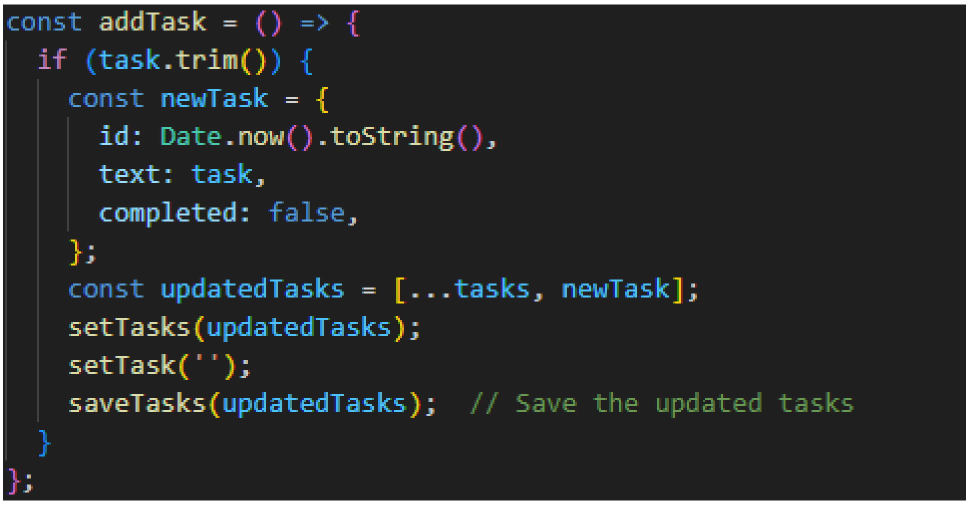
Visual Outcome: When a task is being edited, the text becomes editable. After editing, the task's text is updated, and the UI reflects the change.

(d) Add Animations

Implementation:

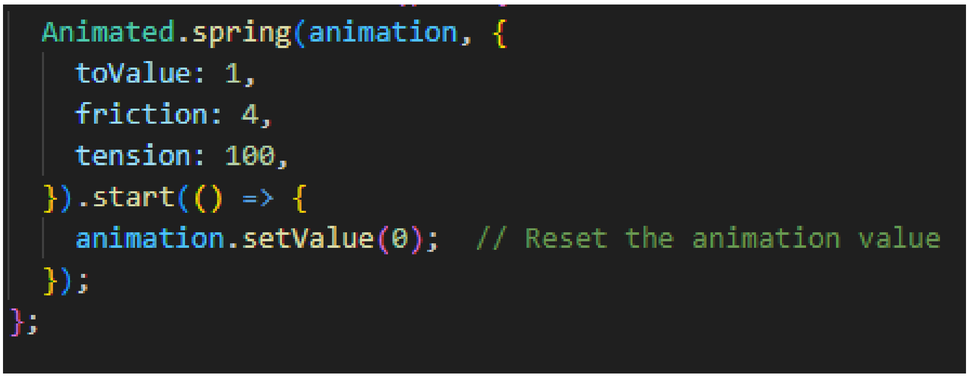
- The `Animated` API was used to add an animation effect to the "+" button when a new task is added.
- The button slightly enlarges when tapped, providing feedback to the user.
- The animation effect is triggered using `Animated.spring` with a friction value to create a bouncing effect when the user taps the add button.

Screenshots for "Add Task" button animation:



```
const addTask = () => {
  if (task.trim()) {
    const newTask = {
      id: Date.now().toString(),
      text: task,
      completed: false,
    };
    const updatedTasks = [...tasks, newTask];
    setTasks(updatedTasks);
    setTask('');
    saveTasks(updatedTasks);  // Save the updated tasks
  }
};
```

Figure 9: "Add Task" animation button.



```
  Animated.spring(animation, {
    toValue: 1,
    friction: 4,
    tension: 100,
  }).start(() => {
    animation.setValue(0);  // Reset the animation value
  });
};
```

Figure 10: Trigger the animation.

Explanation:

- The animation value is animated when the user taps the ”+” button to add a new task. The spring animation provides a bouncing effect, and once completed, the animation value resets to 0.

Visual Outcome: The “+” button for adding tasks features an animation effect, providing a more engaging user experience.

GitHub Repository

The source code for this project can be found on GitHub: **GitHub Repository Link**.