Lab Assignment 3

Deepak Sarun Yuvachandran

November 17th 2024

Task 1: Set Up the Dev Environment

(1) Screenshots of Your App

App running in emulator:

App running in physical device:

Differences Observed

- **Performance:** Apps on the emulator may experience lag, especially with animations or intensive operations, while physical devices typically run smoother.
- **Hardware-Specific Features:** Testing GPS, camera, or accelerometer functionality is more reliable on a physical device. Emulators may simulate these but lack accuracy.
- **Screen Resolution:** Physical devices give a true representation of the display, whereas emulator scaling might vary.
- **Input Mechanism:** On emulators, interactions are simulated via a mouse or keyboard, whereas physical devices rely on real touch gestures.

(2) Setting Up an Emulator

Detailed Steps to Set Up an Emulator in Android Studio:

1. **Install Required Tools:** Download and install Android Studio from `https://developer.android.com/studio`. Ensure you install the Android SDK during the initial setup.
2. **Open the Device Manager:** Navigate to `Tools > Device Manager` in Android Studio. Alternatively, use the shortcut from the toolbar.
3. **Create a New Virtual Device:**
   3.1. Click "Create Device".
   3.2. Select the desired hardware profile.
   3.3. Review specifications such as screen size, resolution, and RAM.
4. **Select a System Image:** Choose the target API level (API 35 for Android 15). Download the system image if not already available.
5. **Configure Emulator Settings:** Adjust options such as startup orientation, RAM allocation, and virtual SD card size. Enable or disable camera and microphone simulation.
6. **Launch the Emulator:** Start the emulator by clicking the play button in the Device Manager.

Tips for Emulator Optimization

- Use Quick Boot for faster subsequent launches.
- Enable GPU Acceleration: Go to `Preferences > Emulator` and enable hardware-accelerated graphics.
- Increase RAM Allocation for faster app load times.

Challenges and How to Overcome Them

- **Emulator Freezes or Crashes:**
  - Check virtualization (Intel VT-x/AMD-V) is enabled in BIOS.
  - Update GPU drivers for compatibility.
- **HAXM Installation Error:** Install HAXM manually from the Android Studio SDK Manager. For Windows users, enable Hyper-V in `Control Panel > Programs > Turn Windows Features On/Off`.
- **Slow Startup:** Reduce emulator resolution to a lower DPI.

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

Detailed Steps to Connect a Physical Device:

1. Install Node.js and ensure it is the latest LTS version.
2. Install Expo CLI: `npm install -g expo-cli`
3. Initialize Your React Native Project:

```
expo init my-app
cd my-app
```

4. Run the Expo Development Server:

```
expo start
```

5. Download the Expo Go app and scan the QR code displayed in the terminal.

Troubleshooting Steps for Physical Device Setup

- **App Fails to Load:**
  - Run `expo doctor` for diagnosing dependency issues.
  - Ensure the physical device and development machine are on the same Wi-Fi network.
- **Network Problems:** Switch to LAN or USB debugging using the Expo development tools.
- **Cache Problems:** Clear Metro bundler cache:

```
expo start --clear
```

- Use the Expo app to test push notifications, media, and performance.

(4) Comparison of Emulator vs. Physical Device

Criteria	**Emulator**	**Physical Device**
Setup Requirements	Requires Android Studio or Xcode installation and configuration.	Requires enabling developer mode, USB debugging (Android), or provisioning profiles (iOS).
Performance	May lag, especially on low-spec machines.	Generally smoother and closer to real-world performance.
Device Diversity	Can simulate multiple devices, screen sizes, and OS versions easily.	Limited to devices you physically own.
Realism	Simulated gestures and hardware interactions (e.g., GPS, camera).	Real touch responsiveness and accurate hardware testing.
Debugging Tools	Preloaded debugging tools (e.g., layout inspector, memory profiler).	Limited debugging; requires external tools (e.g., React Native Debugger).
Resource Usage	High CPU and RAM usage; requires virtualization enabled on the host machine.	Minimal resource impact on the development machine.
Hardware-Specific Testing	Limited; simulated hardware behavior may not match real devices.	Accurate testing of hardware features like accelerometer, camera, and sensors.
Ease of Use	Easy to test across various configurations without owning multiple devices.	Requires manual switching between physical devices.
Network Testing	Simulates various network conditions but may not replicate real-world usage.	Real-world network conditions provide accurate behavior testing.
Cost	Free; no need to buy devices.	Requires investment in physical devices.
Battery Testing	Allows simulation of different battery states (low, charging, etc.).	Real battery performance testing under actual usage conditions.
User Experience Testing	Limited; lacks actual user interaction feedback.	Real-world testing with accurate feedback on gestures and navigation.
Best Use Cases	Initial testing, layout debugging, layout and screen sizes.	Final testing, performance evaluation, and accurate testing.

Table 1: Comparison of Emulator vs. Physical Device

(5) Troubleshooting a Common Error

Error: "Unable to resolve module ."

Cause: Missing package, improper linking, or compatibility issues.

Steps to Resolve:

- Check if the module is installed:

  ```
  npm list module-name
  ```

- If not, install it:

  ```
  npm install module-name
  ```

- Clear cache and rebuild the project:

  ```
  npm start --reset-cache
  ```

- Verify compatibility of React Native version with the module.

Task 2: Building a Simple To-Do List App

(a) Mark Tasks as Complete

Toggle Function for Marking Tasks as Completed: The `toggleTaskCompletion` function is responsible for toggling the `completed` status of tasks. When a user clicks the "Complete" button, it toggles the task's completion state.

```
// Toggle task completion (mark as complete or undo completion)
Complexity is 3 Everything is cool!
const toggleTaskCompletion = (taskId) => {
  const updatedTasks = tasks.map((item) =>
    item.id === taskId ? { ...item, completed: !item.completed } : item // Toggle completion status
  );
  setTasks(updatedTasks); // Update tasks state
  saveTasks(updatedTasks); // Save updated tasks to AsyncStorage
};
```

Styling Completed Tasks Differently: When a task is marked as completed:

- The text gets a strikethrough.
- The color changes to gray using conditional styling.

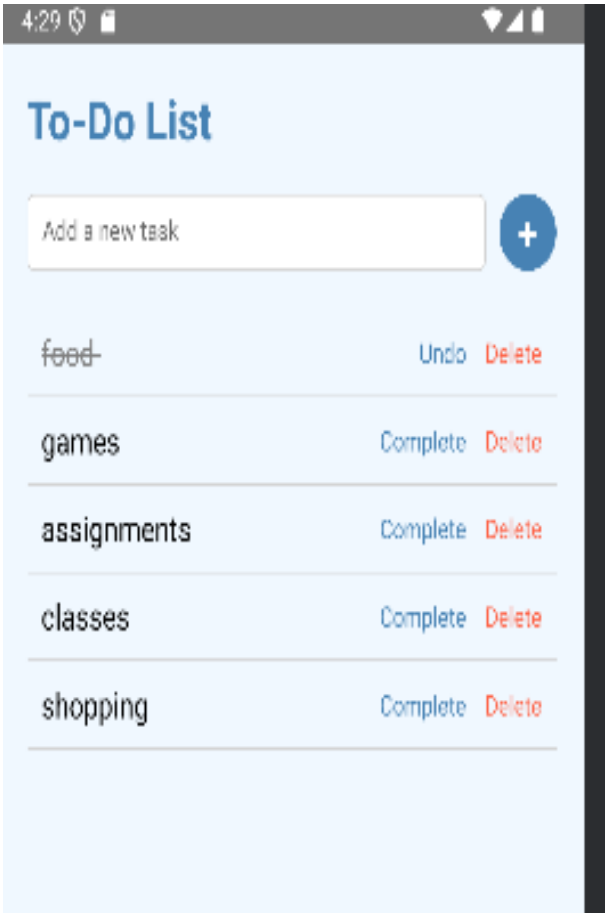
```
<Text
  style={[
    styles.taskText,
    item.completed && styles.completedTaskText, // Strike-through if completed
    item.completed && {color: '#808080' }, // Change color to gray for completed tasks
  ]}
>
  {item.text} {/* Display task text */}
</Text>
```

Explanation: The `completed` property in each task object is toggled between `true` and `false`. This change is reflected in the state by using the `setTasks` function, which updates the tasks array. The `saveTasks` function ensures the updated list is saved to `AsyncStorage`, so the completion status persists across app restarts.

Example of a completed task:

- `food` is marked as a completed task.

Screenshot of Marking Tasks as Complete:



(b) Persist Data Using AsyncStorage

Implementing Data Persistence: `AsyncStorage` is used to persist the tasks so that they remain saved even after the app is closed and reopened.

Functions:

- `loadTasks`: Retrieves the tasks from `AsyncStorage`.
- `saveTasks`: Stores the tasks in `AsyncStorage`.

Explanation: When the app is launched, the tasks are loaded from `AsyncStorage` by calling `loadTasks` inside the `useEffect` hook. This ensures the task list is available on startup. When a task is added, deleted, or edited, the updated task list is saved back to `AsyncStorage` using `saveTasks`.

Screenshot of Data Persistence:

```
const loadTasks = async () => {
  try {
    const storedTasks = await AsyncStorage.getItem('tasks'); // Retrieve tasks from AsyncStorage
    if (storedTasks) setTasks(JSON.parse(storedTasks)); // If tasks exist, update state
  } catch (error) {
    console.error(error); // Log any error that occurs while fetching tasks
  }
};

// Save tasks to AsyncStorage
Complexity is 3 Everything is cool!
const saveTasks = async (tasks) => {
  try {
    await AsyncStorage.setItem('tasks', JSON.stringify(tasks)); // Save updated tasks to AsyncStorage
  } catch (error) {
    console.error(error); // Log any error that occurs while saving tasks
  }
};
```

(c) Edit Tasks

Allow Users to Tap on a Task to Edit: The `openEditModal` function opens a modal with the current task's content when a user taps on a task. The input field in the modal is pre-filled with the task's text, allowing the user to edit it.

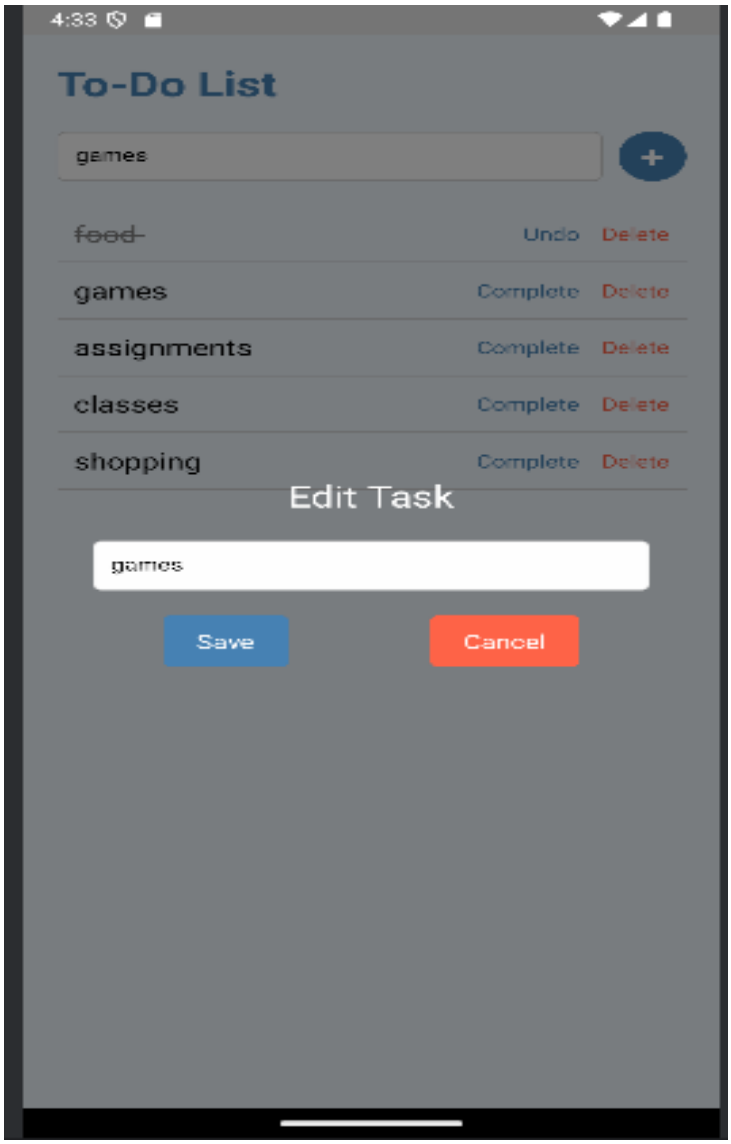
```
const openEditModal = (task) => {
  setSelectedTask(task); // Set the task being edited
  setTask(task.text); // Pre-fill the input with the task's text
  setModalVisible(true); // Show the modal
};
```

Update Function to Modify Task in the State: When the user saves the edited task, the `editTask` function updates the task in the `tasks` state array and saves it to `AsyncStorage`.

```
const editTask = () => {
  if (selectedTask) {
    const updatedTasks = tasks.map((item) =>
      item.id === selectedTask.id ? { ...item, text: task } : item // Update the text of the selected task
    );
    setTasks(updatedTasks); // Update tasks state
    saveTasks(updatedTasks); // Save the updated tasks to AsyncStorage
    setModalVisible(false); // Close the modal
    setTask(''); // Clear the input field
    setSelectedTask(null); // Clear the selected task
  }
};
```

Explanation: The state is updated by mapping through the `tasks` array and modifying the text of the selected task. This ensures the task is updated in the state, and the new list is saved in `AsyncStorage` for persistence.

Screenshot of Editing Tasks:



(d) Add Animations

Using the Animated API: The `Animated` API is used to add an animation effect when a task is added to the list. This animation enlarges the "Add" button briefly to provide feedback when a task is added.

```
const spider = new Animated.Value(0); // Create animation value
```

```
const addTask = () => {
  if (task.trim()) {
    const newTask = { id: Date.now().toString(), text: task, completed: false }; // Create new task object
    const updatedTasks = [...tasks, newTask]; // Add new task to the tasks list
    setTasks(updatedTasks); // Update state with the new tasks list
    saveTasks(updatedTasks); // Save updated tasks to AsyncStorage
    setTask(''); // Clear input field

    // Trigger animation to indicate a new task was added
    Animated.sequence([
      Animated.timing(spider, { toValue: 1, duration: 300, useNativeDriver: true }), // Scale up animation
      Animated.timing(spider, { toValue: 0, duration: 300, useNativeDriver: true }), // Scale down animation
    ]).start(); // Start animation sequence
  } else {
    Alert.alert('Input Error', 'Task cannot be empty.'); // Show alert if input is empty
  }
};
```

Animation Style: The animated style for the "Add" button is applied through `Animated.Text` and uses the `scale` transform to provide a visual feedback when a task is added.

```
// Animated style for task add button
const animatedStyle = {
  transform: [
    {
      scale: spider.interpolate({
        inputRange: [0, 1],
        outputRange: [1, 1.2], // Slightly grow when animation is triggered
      }),
    },
  ],
};
```

Explanation: The animation is triggered when a new task is added. It scales the "Add" button up briefly (to 1.2 times its size) and then shrinks it back to its original size. This gives users feedback that their action has been registered. The `Animated.Value` is used to control the animation's state, and `Animated.timing` is used to change the value over time.

Screenshot of Animation Effect:

```
<Animated.Text style={[styles.addButtonText, animatedStyle]}>+</Animated.Text> {/* Animated plus sign */}
```

GitHub Project Link