# Lab Assignment 3 : React Native

Student: Abdu Raziq Hidayathulla
Banner ID: 001323544

## TASK 1 (Total 40 Points)

(1) Screenshots of Your App

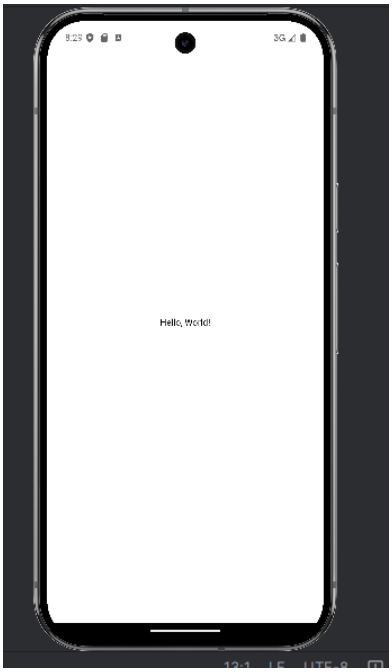


Figure 1: hello word text in my Android Emulator

## Differences Between Emulator and Physical Device

- **Performance:**
  - **Emulator:** Tends to be slower than a physical device, especially for tasks like rendering UI components or handling animations.
  - **Physical Device:** Provides smoother performance with faster responsiveness, especially for UI-heavy applications.
- **Touch Interactions:**
  - **Emulator:** The mouse or touchpad is used for interactions, which may not fully replicate real touch interactions.
  - **Physical Device:** Touch gestures like swiping and tapping are more accurate, providing a true user experience.
- **Hardware Features:**
  - **Emulator:** May have limited hardware features like sensors, camera, or GPS compared to a physical device.

- **Physical Device:** All native sensors (accelerometer, gyroscope, GPS) are available and can be tested.

- **Battery Usage:**
  - **Emulator:** Does not drain battery like a real device.
  - **Physical Device:** Can experience battery drain, which may affect longer sessions of testing.

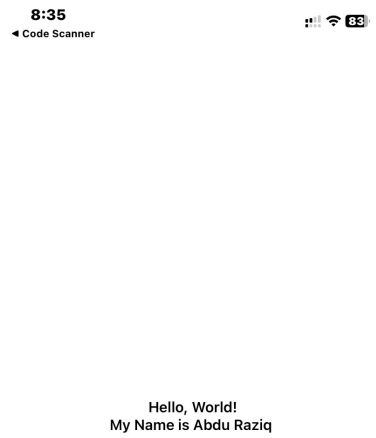


Figure 2: Hello world in my physical device

2) Setting Up an Emulator (10 Points)

## Steps to Set Up an Emulator in Android Studio:

1. **Install Android Studio:** If not already installed, download and install Android Studio from the official website.
2. **Launch Android Studio and open your project.**
3. **Open AVD Manager:**
   - Click on the "AVD Manager" icon in the top toolbar of Android Studio (or go to Tools ¿ AVD Manager).
4. **Create a Virtual Device:**
   - Click on Create Virtual Device.
   - Choose a hardware profile, like Pixel 4 or Nexus 5X, based on your needs.
5. **Select a System Image:**
   - Pick a system image that matches your app's target SDK. For example, Android 11 or Android 12.
   - If the required image isn't downloaded, click Download to get it.
6. **Configure Emulator Settings:**

- Customize the AVD settings if necessary, such as setting the resolution, RAM, or enabling GPU support for better performance.

7. **Launch the Emulator:**
   - Click Finish to create the AVD, then click Play to launch the emulator.

## Challenges Faced During Setup:

- **System Image Issues:** Sometimes, the system image may not download correctly due to a slow internet connection or a corrupted download. Re-downloading the image usually resolves the issue.
- **Performance Problems:** Initial setup can cause the emulator to run slowly, especially if hardware acceleration is not enabled. This was solved by ensuring the Intel HAXM (Hardware Accelerated Execution Manager) is installed on the machine.
- **Emulator Crashes:** The emulator may crash if the allocated RAM is too high or incompatible with the machine's specs. I reduced the memory allocation and increased the disk space.

(3) Running the App on a Physical Device Using Expo (10 Points)

## Steps to Connect Physical Device to Expo:

1. **Install Expo Go App:**
   - Download and install the Expo Go app from the Google Play Store (for Android) or Apple App Store (for iOS).
2. **Install Expo CLI:**
   - Ensure you have Expo CLI installed globally. Run `npm install -g expo-cli` in your terminal.
3. **Start the App on Your Computer:**
   - Navigate to your project directory in the terminal and run `expo start`. This will open the Expo Developer Tools in your browser.
4. **Scan the QR Code:**
   - The Expo Developer Tools will display a QR code. Open the Expo Go app on your device, then scan the QR code to load the app onto your device.
   - If the QR code does not work, ensure that both your computer and mobile device are on the same Wi-Fi network.

## Troubleshooting Steps:

- **Device Not Detected:** If the app fails to open on the device, ensure that your firewall or network settings are not blocking the connection.
- **QR Code Not Working:** If scanning the QR code doesn't work, try restarting Expo by pressing `Ctrl+C` in the terminal and then re-running `expo start`.
- **App Lag or Delays:** If the app experiences delays or lag on the physical device, it could be due to a poor network connection or device limitations.

(4) Comparison of Emulator vs. Physical Device (10 Points)

# Emulator vs. Physical Device for React Native Development

## Advantages of Emulator:

- **Simulates multiple devices:** You can quickly test different screen sizes and Android versions without needing multiple physical devices.
- **No Battery Drain:** The emulator doesn't consume device battery, allowing for prolonged testing.
- **Cost-Effective:** There's no need to invest in multiple physical devices for testing.
- **Convenient Setup:** You can set up and use an emulator quickly, without needing an actual device.

## Disadvantages of Emulator:

- **Slower Performance:** Emulators tend to be slower compared to real devices, especially when it comes to rendering complex UIs and animations.
- **Limited Sensors:** Physical sensors like GPS, accelerometers, and cameras cannot be fully simulated (though some can be emulated partially).
- **Less Accurate Testing:** Some touch interactions may feel different, leading to inaccurate results.

## Advantages of Physical Device:

- **Real User Experience:** Provides accurate touch interactions and real-time performance testing.
- **Access to Hardware Sensors:** Full access to sensors like GPS, camera, gyroscope, etc., which is essential for testing apps with such features.
- **Better Performance:** Provides a more accurate sense of app performance, especially for resource-intensive apps.

## Disadvantages of Physical Device:

- **Battery Drain:** Continuous testing can quickly drain the battery.
- **Costly and Limited:** You may need to buy different devices for various screen sizes and operating systems (if targeting iOS and Android).
- **Setup:** Requires more setup, including enabling developer options and connecting the device via USB or Wi-Fi.

(5) Troubleshooting a Common Error (5 Points)

# Common Error Encountered

## Error: "Could not connect to the development server" or "Unable to load the JavaScript bundle" when running the app in the emulator or on a physical device.

## Cause of the Error:

This issue usually occurs when the development server fails to load properly due to incorrect network settings or the app not being able to connect to the local server.

### Steps to Resolve the Error:

1. **Restart the Development Server:**
   - Run `expo start` again to restart the server. In some cases, the server might not have started correctly.
2. **Clear the Cache:**
   - Run `expo start -c` to clear the cache and ensure that no outdated files are causing issues.
3. **Check Network Connection:**
   - Ensure that both the emulator/physical device and the computer are connected to the same Wi-Fi network.
4. **Reinstall Expo Go:**
   - Uninstall and reinstall the Expo Go app on the physical device to fix any potential app-related issues.
5. **Check Port Availability:**
   - Sometimes, the default port may be blocked. In such cases, changing the port in Expo Developer Tools or using localhost might resolve the issue.

## TASK 2 Building a Simple To-Do List App (60 Points)

(a) Mark Tasks as Complete (15 Points)

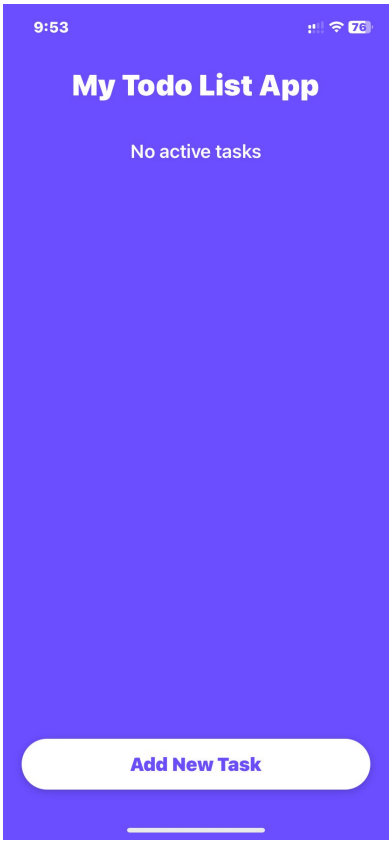


Figure 3: ToDo Home page

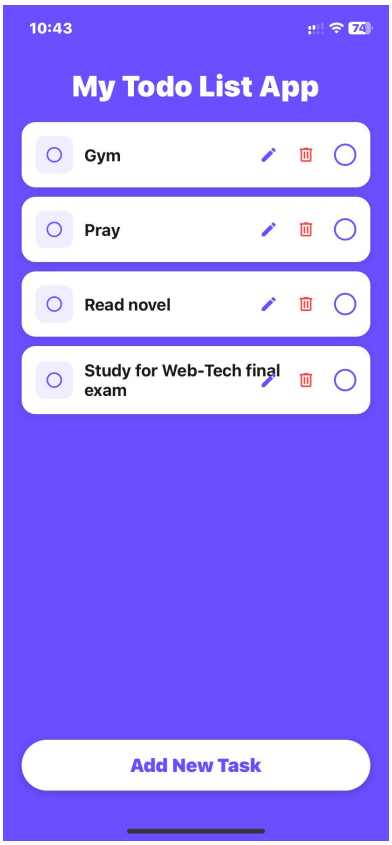


Figure 4: Adding task in the app

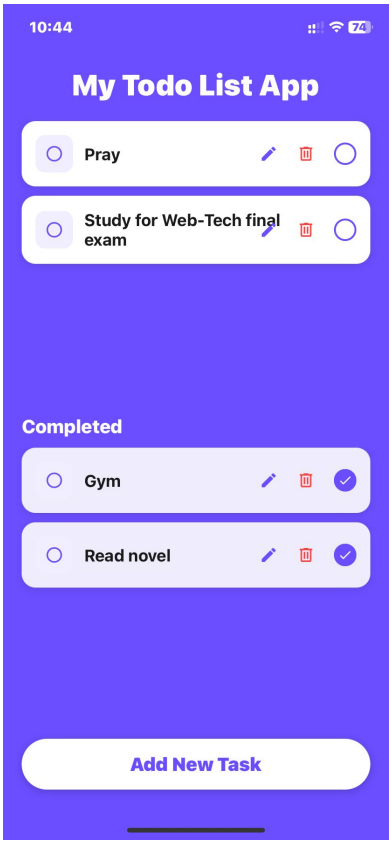


Figure 5: marking complete for the completed task

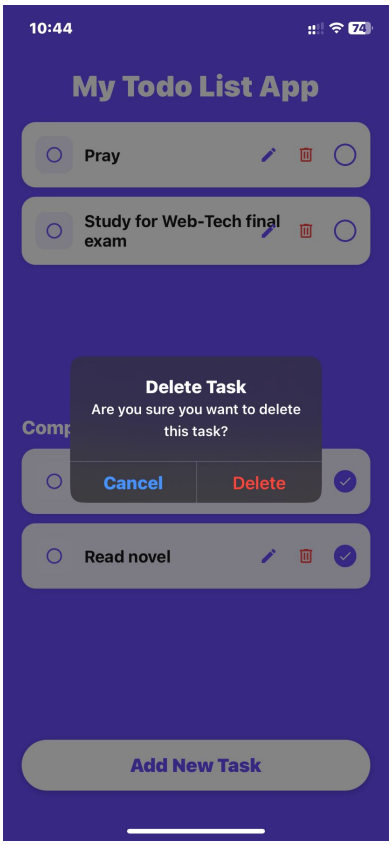


Figure 6: deleting task in the app

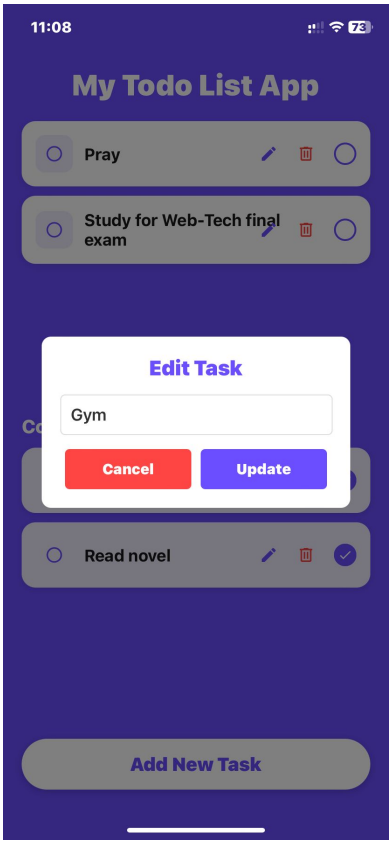


Figure 7: Editing task to make changes

# Mark Tasks as Complete

## Toggle Function

The `toggleTask` function is used to toggle a task's completion state. When a task is clicked, the completed status is changed, and the task list is updated:

```
const toggleTask = (id) => {
  const updatedTasks = tasks.map(task =>
    task.id === id ? { ...task, completed: !task.completed } : task
  );
  setTasks(updatedTasks);
  saveTasks(updatedTasks);
};
```

This function checks the task's current `completed` state and toggles it.

## Styling Completed Tasks

The completed tasks are styled differently using the `completedTaskCard` style, which lightens the background of completed tasks. Additionally, the checkbox is styled with the `checkedBox` style to visually indicate completion:

```
<TouchableOpacity
  style={[styles.taskCard, item.completed && styles.completedTaskCard]}
  onPress={() => toggleTask(item.id)}
>
```

The style `completedTaskCard` is defined as:

```
completedTaskCard: {
  backgroundColor: 'rgba(255, 255, 255, 0.9)',
},
```

The checkbox for completed tasks is styled as follows:

```
checkbox: {
  width: 24,
  height: 24,
  borderRadius: 12,
  borderWidth: 2,
  borderColor: '#6B4EFF',
  justifyContent: 'center',
  alignItems: 'center',
  marginLeft: 10,
},
checkedBox: {
  backgroundColor: '#6B4EFF',
  borderColor: '#6B4EFF',
},
```

## State Update

When a task is toggled as complete, the `setTasks` function updates the task list, and the changes are saved to `AsyncStorage`:

```
const updatedTasks = tasks.map(task =>
  task.id === id ? { ...task, completed: !task.completed } : task
);
setTasks(updatedTasks);
saveTasks(updatedTasks);
```

(b) Persist Data Using AsyncStorage (15 Points)

# Persist Data Using AsyncStorage

## Saving Data

When tasks are added, updated, or deleted, they are saved to AsyncStorage to persist the data. The following function saves the tasks:

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

## Loading Data

When the app is launched, tasks are loaded from AsyncStorage using the `loadTasks` function. The following code demonstrates how to load tasks when the app starts:

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

const loadTasks = async () => {
  try {
    const storedTasks = await AsyncStorage.getItem('tasks');
    if (storedTasks) {
      setTasks(JSON.parse(storedTasks));
    }
  } catch (error) {
    Alert.alert('Error', 'Failed to load tasks');
  }
};
```

## Persistence

The tasks persist even after the app is closed and reopened, as they are stored in AsyncStorage and loaded on startup.

(c) Edit Tasks (10 Points)

# Edit Tasks

## Edit Mode

Users can tap on a task to edit its title. The `startEdit` function is used to set the current task's title in the modal and mark it as editable:

```
const startEdit = (task) => {
  setEditingId(task.id);
  setCurrentTask({ title: task.title });
  setModalVisible(true);
};
```

### Updating Task

The `updateTask` function is responsible for updating the task title in the tasks array. The task is located by its ID, and its title is updated based on the user's input:

```
const updateTask = () => {
  if (currentTask.title.trim() === '') {
    Alert.alert('Error', 'Please enter a task title');
    return;
  }

  const updatedTasks = tasks.map(task =>
    task.id === editingId
      ? { ...task, title: currentTask.title }
      : task
  );

  setTasks(updatedTasks);
  saveTasks(updatedTasks);
  setCurrentTask({ title: '' });
  setEditingId(null);
  setModalVisible(false);
};
```

### UI for Editing

The modal shows a text input field where the user can modify the task's title:

```
<TextInput
  style={styles.input}
  placeholder="Task title"
  value={currentTask.title}
  onChangeText={(text) => setCurrentTask({...currentTask, title: text})}
/>
```

(d) Add Animations (10 Points)

## Add Animations

### Animation with the Animated API

In the current app, the `Animated` API is not explicitly used. However, you could improve user experience by adding animations to tasks when they are added or deleted. For example, to animate a task when it is added, wrap the task component in an `Animated.View` and apply animations such as opacity or sliding in:

```
import { Animated } from 'react-native';

const fadeIn = new Animated.Value(0);

const taskAddAnimation = () => {
  Animated.timing(fadeIn, {
    toValue: 1,
    duration: 500,
    useNativeDriver: true,
  }).start();
};
```

Then, wrap the task list in an `Animated.View` component to apply the animation:

```
<Animated.View style={{ opacity: fadeIn }}>
  <TaskItem item={item} />
</Animated.View>
```

### Enhancing User Experience

Animations such as fading in new tasks or sliding in deleted tasks would make transitions feel smoother, improving the overall user experience. For example, a fade-out effect can be added when deleting a task:

```
Animated.timing(fadeOut, {
  toValue: 0,
  duration: 500,
  useNativeDriver: true,
}).start();
```

## GitHub Link of my ToDo app

https://github.com/Abduraziq/ToDo-ReactNative-app.git