Lab 3: Building a Mobile App

Rakesh Reddy Dodda

November 17, 2024

Task 1: Screenshots of Your App

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

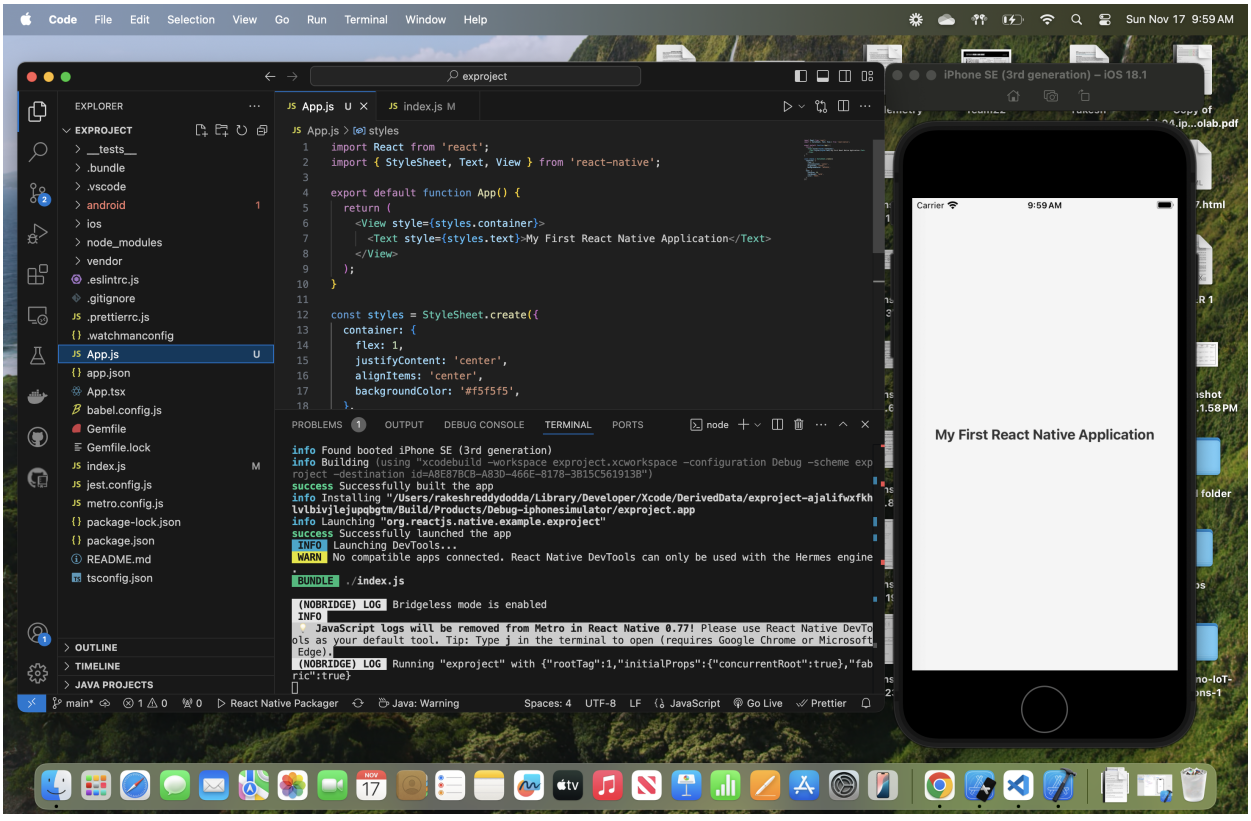

Figure 1: Simulator Image

Figure 2: Physical Device Running App

1. When running our app in the simulator, it is a bit slower than on a physical device.

2. The simulator does not have full access to the hardware properties.

3. The simulator uses the host network, so testing on real-world conditions will be problematic. A physical device, however, connects through Wi-Fi, allowing us to experience real-world problems and solve them.

4. The user experience in the simulator does not feel realistic as we are using a touchpad. On a physical device, we can experience real-world usage.

Setting Up an Emulator

1. Install the Xcode application from App Store.
2. After installing, open Xcode and complete the setup.
3. Install command line tools using `xcode-select --install`.
4. Set up the React Native environment:
   - Install Node.js and npm.
   - Install React Native CLI globally using `npm install -g react-native-cli`.
5. Create a React Native project using:

   ```
   npx react-native init ProjectName
   ```

6. Navigate to the project directory using the `cd` command.
7. Run the project using:

   ```
   npx react-native run-ios
   ```

Challenges

While setting up the simulator, I encountered several errors as this was my first time using it. The primary issue was starting the simulator. Initially, I ran the project without creating a simulator. Later, I found a solution online to open a simulator using Xcode. Additional errors occurred due to outdated Metro Bundler caches. Resolving these required removing `node_modules` and reinstalling them.

Running the App on a Physical Device Using Expo

Steps to Connect to Expo:

1. Install Expo Go on your physical device.
2. Set up the developer environment using terminal commands:

```
npm install -g expo-cli
npx expo init YourProjectName
cd YourProjectName
```

3. Edit the `App.js` code.
4. Run the app using:

```
npx expo start
```

5. Scan the generated QR code with your phone camera to open the app.

Troubleshooting Steps

1. Ensure both devices are connected to the same network to open the app.
2. Temporarily disable firewall settings if the app does not open.
3. Resolve missing module errors by installing required packages (e.g., `async-storage`).

Comparison of Emulator vs. Physical Device

Aspect	**Simulator**	**Physical Device**
Setup	Requires Xcode installation	Requires Expo Go app installation
Performance	Slower due to virtualization	Faster due to hardware usage
Device Access	Limited access	Full access
Debugging	Built-in tools available	Requires separate setup
Screen Resolution	Multiple resolutions available	Fixed to device screen

Table 1: Comparison of Simulator and Physical Device

Advantages and Disadvantages

Simulator:

Advantages:

- It is cost-efficient; no need for a physical device.
- We can use different devices for testing.
- Built-in tools are available for debugging issues.
- It works under any environmental condition.

Disadvantages:

- It works slower than a physical device.
- It does not have full hardware access; it has only limited access.
- It requires a development environment.

Physical Device:

Advantages:

- Here, we can experience the real user experience.
- The performance accuracy is higher compared to the simulator.
- It has full access to hardware.

Disadvantages:

- Testing on a physical device requires a separate setup.
- You need to have your own physical device.
- You can experience battery drain on your device.

Troubleshooting a Common Error in React Native

The common error encountered is the `ModuleNotFound` error, which occurs when the Expo module is not installed correctly. To fix this, ensure that the module is listed in the `package.json` file. If you do not find the module, install it by running the following command in the terminal:

```
npm install expo
```

After installing the package, clear the cache using:

```
npm cache clean --force
```

Now, you can run the app using:

```
npx expo start
```

Task 2: (A) Mark Tasks as Complete

Code for Toggle Function

Adding a toggle function to mark the task as complete
code for this is

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

With this we can edit the task whenever we want.

Styling Completed Tasks

```
taskCompletedText: {
    textDecorationLine: 'line-through',
    color: 'gray',
}
```

```
<TouchableOpacity onPress={() => toggleTask(item.id)}>
    <Text
        style={[
            styles.taskText,
            item.completed && styles.taskCompletedText,
        ]}
    >
        {item.text}
    </Text>
</TouchableOpacity>
```

I have initialized the touchableOpacity OnPress function
this works when we touch the task we get a line on
the text as it shows completed.

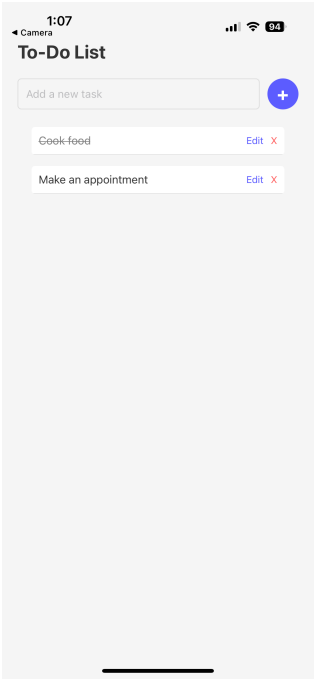

Figure 3: Mark task as complete

(B) Persisting Data Using AsyncStorage

To store the tasks after closing the app we are using AsyncStorage function which uses local storage to store the tasks.

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

I Used AsyncStorage.getItem to retrieve tasks on app load

so now we can close and open the app the date will not be lost.

```
useEffect(() => {
    const loadTasks = async () => {
        try {
            const storedTasks = await AsyncStorage.getItem('tasks');
            if (storedTasks) setTasks(JSON.parse(storedTasks));
        } catch (e) {
            console.error('Failed to load tasks:', e);
        }
    };
    loadTasks();
}, []);
```

(C) Edit task

This funcion is used to edit the existing task.
If you have entered the task incorrect we can edit that using edit button.

```
const editTask = (taskId) => {
    const task = tasks.find((task) => task.id === taskId);
    setEditingTaskId(taskId);
    setEditingText(task.text);
};
```

Now the task is editable next the task should be modified in
the state array foemat. We need to use the rendering approach method
to display the tasks.

We used editTask() and updateTask() functions for this.

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

```
{editingTaskId === item.id ? (
```

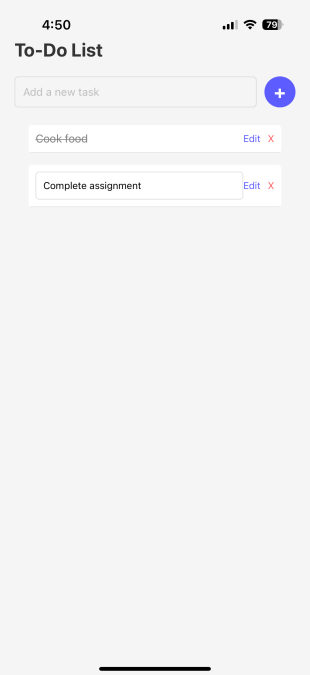

Figure 4: Edit task

```
    <TextInput
        style={styles.input}
        value={editingText}
        onChangeText={setEditingText}
        onSubmitEditing={updateTask}
    />
) : (
    <TouchableOpacity onPress={() => toggleTask(item.id)}>
        <Text style={[
            styles.taskText,
            item.completed && styles.taskCompletedText
        ]}>
            {item.text}
        </Text>
    </TouchableOpacity>
)}
```

(D) Add Animation

When we click the task we get a animation effect. Here i have used timimg function to set the time.

```
const animatedValue = new Animated.Value(0);

const animateAddTask = () => {
    Animated.timing(animatedValue, {
        toValue: 1,
        duration: 500,
        useNativeDriver: true,
    }).start(() => animatedValue.setValue(0));
};

const scale = animatedValue.interpolate({
    inputRange: [0, 1],
    outputRange: [0.9, 1],
});
```

How Animation enhance user experience

The animations added in the SimpleToDoApp greatly improve the user experience by adding smooth and visually pleasing transitions. Interaction feels more responsive.

- Upon adding a task, an immediate scaling animation is applied to visually provide feedback, making the task appear to smoothly scale from its smaller size to full size. This provides a sense of satisfaction and clarity.
- A fade-out and shrink animation is used to make the task deletion process appear smooth, visually fading out and shrinking the task.
- When a task is marked as complete, it features a stroke animation to visually signal the change from an incomplete task to a complete task.
- Editing tasks triggers a safe fade-in animation to the input field, helping users transition into editing mode seamlessly.

Overall, these animations provide clear information and make the app look polished, engaging, and user-friendly. The app creates consistent visual

cues that make it easy for users to monitor their tasks and actions, making the experience enjoyable and satisfying.

GitHub Repository

You can access it using the following link:

SimpleToDoApp GitHub Repository

SimpleToDoAppExpo GitHub Repository

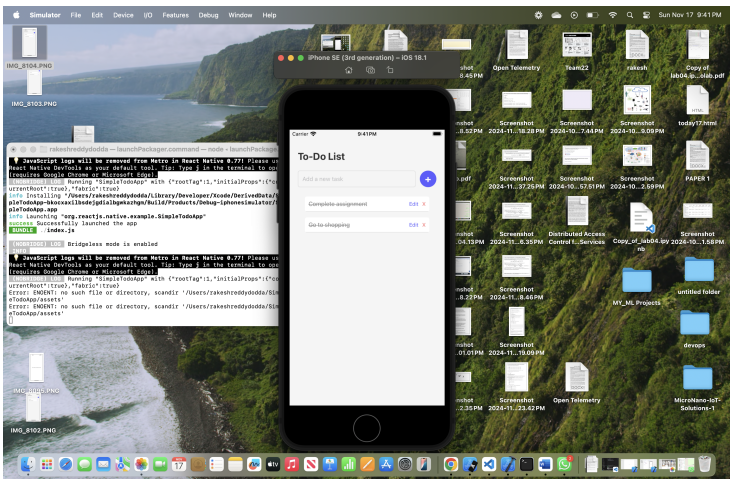

Figure 5: App running on simulator

Figure 6: App running on simulator

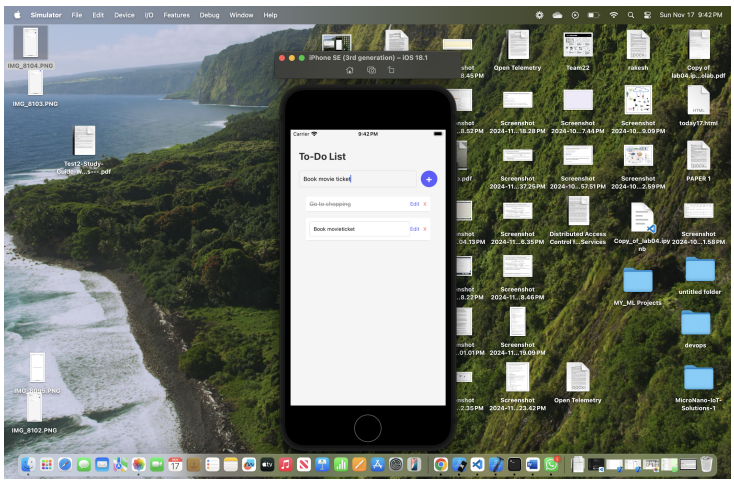

Figure 7: App running on simulator

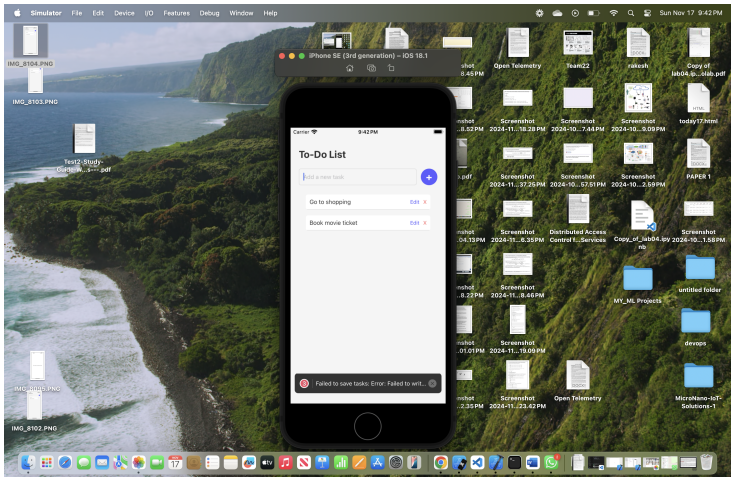

Figure 8: App running on simulator