Bluetooth BLE Integration in React Native Expo (New Architecture, iOS & Android).

Table of contents

By Chinweike Chinonso Michael

7 min. read · View original

Press enter or click to view image in full size

Table of contents

Welcome to my space

If you've ever thought *"How do I get my React Native app talking to Bluetooth devices without losing my sanity?"* — then this article is for you.

The goal here is simple: to walk you through a step-by-step guide on how to integrate Bluetooth Low Energy (BLE) into your **React Native** project using the **new architecture** (yes, both iOS and Android included 🎉).

And hey — I've been in your shoes. Sharing this to make it easier for devs who just want a clear and working Bluetooth setup, without the doc-hunting.

Full source code: https://github.com/cmcWebCode40/React-Native-Expo-Bluetooth-Integration

Bluetooth Classic vs Bluetooth Low Energy (BLE)

Before we dive into the juicy stuff, let's quickly clear up the difference between Bluetooth Classic and BLE:

In this guide, we'll focus **solely on BLE** — scanning, connecting, and exchanging data (READ/WRITE).

We'll be using the react-native-ble-manager package. It supports both the old and new architectures, but a few tweaks are required to make it work smoothly in an **Expo + React Native project**.

Project setup using Expo

```
npx create-expo-app@latest
```

Install the `react-native-ble-manager` library

You can as well use this package in your react native expo project.

Install the react native react-native-ble-manager library

```
npx expo install react-native-ble-manager
```

In your `app.json` file extend the plugin config to enable the package work.

```
"plugins": [
      "expo-router",
      [
        "react-native-ble-manager",
        {
          "isBleRequired": true,
          "neverForLocation": true,
          "companionDeviceEnabled": false,
          "bluetoothAlwaysPermission": "Allow BLE
DEMO APP to connect to bluetooth devices"
        }
      ]
    ]
```

To test anything Bluetooth-related, you'll **need a real device**. No simulator magic here — BLE just doesn't work in emulators.

Now, if you're using `npx expo run:android` or `npx expo run:ios`, make sure your device is physically connected (ADB for Android).

Alternatively, you can go the recommended route by creating a **custom development build** using EAS. This method gives you a permanent dev client on your device you can reuse anytime.

First, update your `eas.json` to include the `developmentClient: true` flag:

```
{
  "cli": {
    "version": ">= 3.9.0"
  },
  "build": {
    "development": {
      "developmentClient": true,
      "distribution": "internal",
      "ios": {
        "resourceClass": "m-medium"
      },
      "channel": "development"
    }
  }
}
```

Then, run the build:

```
eas build --profile development --platform android
```

Once the build is installed on your device, you can simply start the dev server:

```
npx expo start
```

then scan the QR code to launch the app inside your custom client.

📚 Want to learn more? Check out the official Expo docs on creating dev builds.

Basic UI Development

To keep our BLE demo clean and modular, I created a few reusable components that handle both connection states and device lists.

Full source code here:

👉 GitHub Repo

We'll structure the UI using **three components** inside a folder named `components/bluetooth/`:

1. `ConnectedState.tsx` – When you're connected to a BLE device
2. `DisconnectedState.tsx` – When you're not connected (yet)
3. `PeripheralList.tsx` – To show scanned devices and connect to them

Let's take a peek into each one.

For the `components/bluetooth/ConnectedState.tsx` Component.

This component appears once a peripheral is successfully connected. It displays service info and lets you perform **READ**, **WRITE**, and **DISCONNECT** actions.

```
import { View, Text, TouchableOpacity, StyleSheet
} from "react-native";
import React from "react";
import { PeripheralServices } from
"@/types/bluetooth";
```

```
interface ConnectedStateProps {
  bleService: PeripheralServices;
  onRead: () => void;
  onWrite: () => void;
  onDisconnect: (id: string) =>
void;
}
const ConnectedState:
React.FunctionComponent<ConnectedSt
ateProps> =
({bleService,onDisconnect,onRead,on
Write}) => {
  return (
    <>
      <View style={styles.card}>
        <Text style={styles.info}>
          Peripheral ID:
{bleService.peripheralId} dBm
        </Text>
        <Text style=
{styles.info}>Service ID:
{bleService.serviceId}</Text>
      </View>
      <View style=
{styles.actionButtons}>
        <TouchableOpacity
          onPress={onRead}
          style={styles.button}
        >
          <Text style=
{styles.buttonText}>READ</Text>
        </TouchableOpacity>
        <TouchableOpacity
          onPress={onWrite}
          style={styles.button}
        >
```

```
          <Text style=
{styles.buttonText}>WRITE</Text>
        </TouchableOpacity>
        <TouchableOpacity
          onPress={() =>
onDisconnect(bleService.peripheralI
d)}
          style=
{styles.disconnectButton}
        >
          <Text style=
{styles.buttonText}>DISCONNECT</Tex
t>
        </TouchableOpacity>
      </View>
    </>
  );
};
export default ConnectedState;
const styles = StyleSheet.create({
    actionButtons: {
      flexDirection: "row",
      marginTop: 16,
    },
    button: {
      backgroundColor: "#007AFF",
      padding: 12,
      borderRadius: 8,
      marginHorizontal: 8,
      flexGrow: 1,
    },
    disconnectButton: {
      backgroundColor: "red",
      padding: 12,
      borderRadius: 8,
      marginHorizontal: 8,
    },
    buttonText: {
      color: "#fff",
      fontSize: 16,
      textAlign: "center",
      fontWeight: "500",
    },
    info: {
      fontSize: 14,
      color: "#333",
    },
    card: {
      backgroundColor: "#fff",
      padding: 16,
      marginVertical: 8,
      borderRadius: 10,
      shadowColor: "#000",
      shadowOffset: { width: 0,
height: 2 },
      shadowOpacity: 0.1,
      shadowRadius: 4,
      elevation: 3,
    },
  });DisconnectedState.tsx
```

For the `components/bluetooth/PeripheralList.tsx` Component.

This is what users see when there's no connection. They can scan for peripherals and tap to connect.

```
import { StyleSheet, Text, TouchableOpacity } from
"react-native";
import React from "react";
import PeripheralList from "./PeripheralList";
import { StrippedPeripheral } from
"@/types/bluetooth";

interface DisconnectedStateProps {
  peripherals:
StrippedPeripheral[];
  isScanning: boolean;
  onScanPress: () => void;
  onConnect: (peripheral:
StrippedPeripheral) =>
Promise<void>;
}

const DisconnectedState:
React.FunctionComponent<Disconnecte
dStateProps> = ({
  isScanning,
  onScanPress,
  peripherals,
  onConnect,
}) => {
  return (
    <>
      <TouchableOpacity style=
{styles.scanButton} onPress=
{onScanPress}>
        <Text style=
{styles.scanButtonText}>
          {isScanning ?
"Scanning..." : "Start Scan"}
        </Text>
      </TouchableOpacity>
      {peripherals.length > 0 ? (
        <PeripheralList onConnect=
{onConnect} peripherals=
{peripherals} />
      ) : (
        <Text style=
{styles.emptyText}>No peripherals
found</Text>
      )}
    </>
  );
};
export default DisconnectedState;
const styles = StyleSheet.create({
  scanButton: {
    backgroundColor: "#007AFF",
    padding: 12,
    borderRadius: 8,
    marginBottom: 16,
  },
  scanButtonText: {
    color: "#fff",
    fontSize: 16,
    textAlign: "center",
    fontWeight: "500",
  },
  emptyText: {
    fontSize: 16,
    color: "#666",
    marginTop: 20,
  },
});
```

For the `components/bluetooth/PeripheralList.tsx` Component.

This component handles the rendering of available BLE peripherals in a simple list.

```
import { StrippedPeripheral } from
"@/types/bluetooth";
import React from "react";
import {
  View,
  FlatList,
  StyleSheet,
  Text,
  TouchableOpacity,
} from "react-native";
```

```
interface PeripheralListProps {
  peripherals:
StrippedPeripheral[];
  onConnect: (peripheral:
StrippedPeripheral) =>
Promise<void>;
}
const PeripheralList:
```

```
React.FC<PeripheralListProps> = ({
  peripherals,
  onConnect,
}) => {
  return (
    <View style={styles.container}>
      <FlatList
        data={peripherals}
        keyExtractor={(item) =>
item.id}
        renderItem={({ item }) => (
          <TouchableOpacity
onPress={() => onConnect(item)}
style={styles.card}>
            <Text style=
{styles.title}>{item.name ??
"Unknown Device"}</Text>
            <Text style=
{styles.subtitle}>
              Local Name:
{item.localName ?? "N/A"}
            </Text>
            <Text style=
{styles.info}>RSSI: {item.rssi}
dBm</Text>
            <Text style=
{styles.info}>ID: {item.id}</Text>
          </TouchableOpacity>
        )}
      />
    </View>
  );
```

```
};
const styles = StyleSheet.create({
  container: {
    flex: 1,
    // padding: 16,
  },
  card: {
    backgroundColor: "#fff",
    padding: 16,
    marginVertical: 8,
    borderRadius: 10,
    shadowColor: "#000",
    shadowOffset: { width: 0,
height: 2 },
    shadowOpacity: 0.1,
    shadowRadius: 4,
    elevation: 3,
  },
  title: {
    fontSize: 18,
    fontWeight: "bold",
    marginBottom: 4,
  },
  subtitle: {
    fontSize: 14,
    color: "#555",
    marginBottom: 4,
  },
  info: {
    fontSize: 14,
    color: "#333",
  },
});
export default PeripheralList;
```

Now lets dive in to the implementation in our Home screen in the `app/(tabs)/index.tsx` folder

1. State Initialisation & Event Handlers

```
const [isScanning, setIsScanning] =
useState(false);
  const [peripherals, setPeripherals] = useState(
    new Map<Peripheral["id"], Peripheral>()
  );
  const [isConnected, setIsConnected] =
useState(false);
  const [bleService, setBleService] =
useState<PeripheralServices | undefined>(
    undefined
  );
```

```
useEffect(() => {
  BleManager.start({ showAlert:
false })
    .then(() =>
console.debug("BleManager
started."))
    .catch(console.error);
  const listeners = [

BleManager.onDiscoverPeripheral(han
dleDiscoverPeripheral),

BleManager.onStopScan(handleStopSca
n),

BleManager.onConnectPeripheral(hand
leConnectPeripheral),

BleManager.onDidUpdateValueForChara
cteristic(handleUpdateValueForChara
cteristic),

BleManager.onDisconnectPeripheral(h
andleDisconnectedPeripheral),
  ];
  handleAndroidPermissions();
  return () => listeners.forEach(l
=> l.remove());
}, []);
```

Initialises the BLE manager and declares your four key state variables (`isScanning`, `peripherals`, `isConnected`, `bleService`), wires up handlers for discovery, scan stop, connect, characteristic updates, and disconnect events, requests Android BLE permissions, and cleans up all listeners on unmount.

`bleService` (State) & `PeripheralServices` (Types)

We import this type from `types/bluetooth.ts`:

```
export type PeripheralServices = {
  peripheralId: string;
  serviceId:   string;
  transfer:    string;
  receive:     string;
}
```

`bleService` just holds those four UUIDs (device ID, service UUID, write-char UUID, notify-char UUID) so you can safely pass them into your read/write/disconnect calls.

2. Scanning for Devices

startScan()

```
const startScan = async () => {
  const state = await BleManager.checkState();

  if (state === "off") {
          if (Platform.OS ===
"ios") {
            Alert.alert(
              "Enable Bluetooth",
              "Please enable
Bluetooth in Settings to
continue.",
              [
                { text: "Cancel",
style: "cancel" },
                { text: "Open
Settings", onPress: () =>
Linking.openURL("App-
Prefs:Bluetooth") },
              ]
            );
          } else {
            await
BleManager.enableBluetooth();
          }
if (!isScanning) {
    setPeripherals(new Map());
    setIsScanning(true);
    BleManager.scan(
      SERVICE_UUIDS,
      SECONDS_TO_SCAN_FOR,
      ALLOW_DUPLICATES,
      {
        matchMode:
BleScanMatchMode.Sticky,
        scanMode:
BleScanMode.LowLatency,
        callbackType:
BleScanCallbackType.AllMatches,
      }
    );
  }
};
}
```

1. Check & enable Bluetooth.
2. On **iOS**, apps **cannot** enable Bluetooth programmatically due to platform restrictions, so we redirect users to the Settings app. While on **Android**, you can toggle it in-app via `BleManager.enableBluetooth()`.
3. Reset your list and kick off a 5-second scan.
4. `ALLOW_DUPLICATES` ensures RSSI updates show up.

handleDiscoverPeripheral

```
const handleDiscoverPeripheral = (peripheral:
Peripheral) => {
    if (!peripheral.name) {
      peripheral.name = "NO NAME";
    }
    setPeripherals((map) => {
      return new Map(map.set(peripheral.id,
peripheral));
    });
  };
```

3. Connecting to a Peripheral

```
const connectPeripheral = async (
    peripheral: Omit<Peripheral, "advertising">
  ) => {
    try {
      if (peripheral) {
        setPeripherals((map) => {
          let p = map.get(peripheral.id);
          if (p) {
            p.connecting = true;
            return new Map(map.set(p.id, p));
          }
          return map;
        });
       await BleManager.connect(peripheral.id);
        console.debug(`[connectPeripheral]
[${peripheral.id}] connected.`);
        setPeripherals((map) => {
          let p = map.get(peripheral.id);
          if (p) {
            p.connecting = false;
            p.connected = true;
            return new Map(map.set(p.id, p));
          }
          return map;
        });
        // give bonding time
        await sleep(900);
        /* Test read current RSSI value, retrieve
services first */
        const peripheralData = await
BleManager.retrieveServices(peripheral.id);
        if (peripheralData.characteristics) {
          const peripheralParameters = {
            peripheralId: peripheral.id,
            serviceId: DEVICE_SERVICE_UUID,
            transfer:
TRANSFER_CHARACTERISTIC_UUID,
            receive: RECEIVE_CHARACTERISTIC_UUID,
          };
          setBleService(peripheralParameters);
          setIsConnected(true);
        }
        setPeripherals((map) => {
          let p = map.get(peripheral.id);
          if (p) {
            return new Map(map.set(p.id, p));
          }
          return map;
        });
      }
    } catch (error) {
      console.error(
        `[connectPeripheral][${peripheral.id}]
connectPeripheral error`,
        error
      );
    }
  };
```

Key Points.

1. Connect by ID.
2. Wait ~`900` ms for bonding.
3. Pull services/characteristics and stash your UUIDs in `bleService`.

Service & Characteristic UUIDs

You can hardcode these if your IoT device advertises them (as I did), or fetch them dynamically:

```
const data = await
BleManager.retrieveServices(peripheral.id);
console.log(data.characteristics);
```

4. Writing & Reading Data

write()

```
const write = async () => {
  const MTU = 255;
   if (bleService) {
      const data = Array.from(new
TextEncoder().encode("Hello World"));
      await BleManager.write(
        bleService.peripheralId,
        bleService.serviceId,
        bleService.transfer,
        data,
        MTU
      );
   }
};
```

What's going on here?

1. We use `TextEncoder` to convert `"Hello World"` into bytes—BLE deals with raw data, not plain text.
2. Then we call `BleManager.write()` to send it to the device.
3. The `MTU` (Maximum Transmission Unit) in Bytes tells the BLE stack how many bytes can be sent in one go—255 bytes works fine for short messages like this.

> *TextEncoder is available in most modern JS environments, but if you get an error saying it's undefined, just install the polyfill:*

```
yarn add text-encoding
```

read()

```
const read = async () => {
   if (bleService) {
      const response = await BleManager.read(
        bleService.serviceId,
        bleService.peripheralId,
        bleService.receive
      );
      return response;
   }
};
```

5. Disconnecting

```
const disconnectPeripheral = async (peripheralId:
string) => {
      await BleManager.disconnect(peripheralId);
      setBleService(undefined);
      setPeripherals(new Map());
      setIsConnected(false);
  };
```

Below is a nutshell of `BluetoothDemoScreen` code . Please Note the states, effects and logic hidden for clarity. Clone the repo for the full code and give it a spin.

```
const BluetoothDemoScreen: React.FC = () => {
/* states, useeffect, logic here */
  return (
    <View style={styles.container}>
      <Text style={styles.header}>Bluetooth
Demo</Text>
      {!isConnected ? (
        <DisconnectedState
          peripherals=
{Array.from(peripherals.values())}
          isScanning={isScanning}
          onScanPress={startScan}
          onConnect={connectPeripheral}
        />
      ) : (
        bleService && (
          <ConnectedState
            bleService={bleService}
            onRead={read}
            onWrite={write}
            onDisconnect={disconnectPeripheral}
          />
        )
      )}
    </View>
  );
};
/* UI style Sheet here*/
export default BluetoothDemoScreen;
```

This article covers the core stuff to get you up and running. But yeah — there's a lot more you can do with `react-native-ble-manager` that I didn't touch here. Docs are linked below if you're curious.

Conclusion