Question: *How to show driver's location in customer app*

Answer: To show the driver's live location in the customer app, you'll create a real-time data flow. The driver's app will continuously send its GPS coordinates to your NestJS backend, and the backend will instantly push those coordinates to the relevant customers' apps.
The best technology for this real-time communication is **WebSockets**.
Here's a breakdown of the three main parts of the system.

1. Driver's App (Expo): Sending Location Updates

The driver's app is responsible for capturing the GPS location and sending it to your server.

1. **Get Location Permissions:** First, you need to ask the driver for permission to access their location using the expo-location library.
2. **Track Position Continuously:** Use Location.watchPositionAsync to get a stream of location updates. This function will automatically provide new coordinates as the driver moves.
3. **Send Updates via WebSocket:** Connect to your NestJS backend using a WebSocket client like socket.io-client. Every time you get a new location, you'll **emit** an event to the server. To save battery, you should send updates on a set interval (e.g., every 10-15 seconds).

Conceptual Code for the Driver's App:

```
import React, { useEffect } from 'react';
import * as Location from 'expo-location';
import io from 'socket.io-client';

const socket = io('YOUR_NESTJS_SERVER_URL');

const DriverLocationTracker = () => {
  useEffect(() => {
    const startTracking = async () => {
      let { status } = await
Location.requestForegroundPermissionsAsync();
      if (status !== 'granted') {
        console.error('Permission to access location was denied');
        return;
      }

      // Watch for location changes
      Location.watchPositionAsync({
        accuracy: Location.Accuracy.High,
        timeInterval: 10000, // 10 seconds
        distanceInterval: 10, // 10 meters
      }, (location) => {
        // Send coordinates to the backend
        const { latitude, longitude } = location.coords;
        socket.emit('updateLocation', { driverId: 'your_driver_id',
latitude, longitude });
      });
    };

    startTracking();
  }, []);

  return null; // This is a background component
};
```

2. Backend (NestJS): Receiving and Broadcasting

Your NestJS server acts as the central hub, receiving locations from drivers and forwarding them to the correct customers.

1. **Set up a WebSocket Gateway:** Use the @nestjs/websockets package to create a gateway that listens for WebSocket connections.
2. **Use Rooms for Targeting:** When a customer opens the map to track a specific driver, they should join a "room" dedicated to that driver's route (e.g., route-123). This is crucial for efficiency, ensuring you only send updates to interested customers.
3. **Listen and Broadcast:**
    - The gateway will have a handler that listens for the updateLocation event from drivers.
    - When it receives an update, it will then **broadcast** a new event (e.g., driverLocationUpdated) to everyone in the corresponding room.

Conceptual Code for NestJS Gateway:

```
// in your-app.gateway.ts
import { WebSocketGateway, SubscribeMessage, WebSocketServer,
MessageBody } from '@nestjs/websockets';
import { Server, Socket } from 'socket.io';

@WebSocketGateway()
export class AppGateway {
  @WebSocketServer()
  server: Server;

  // Customer joins a room to listen for a specific route
  @SubscribeMessage('subscribeToRoute')
  handleSubscribe(client: Socket, routeId: string) {
    client.join(routeId);
  }

  // Driver sends their location
  @SubscribeMessage('updateLocation')
  handleLocationUpdate(@MessageBody() data: { routeId: string;
latitude: number; longitude: number }) {
    // Broadcast the location to all customers in that route's room
    this.server.to(data.routeId).emit('driverLocationUpdated', {
      latitude: data.latitude,
      longitude: data.longitude,
    });
  }
}
```

3. Customer's App (Expo): Displaying the Live Location

The customer's app listens for the location updates and displays the driver's icon moving on a map.

1. **Connect to WebSocket and Subscribe:** When the customer opens the tracking screen, connect to the WebSocket and **emit** the subscribeToRoute event to join the correct room.
2. **Listen for Updates:** Set up a listener for the driverLocationUpdated event from the server.
3. **Update the Map:** Use the react-native-maps library to display a map. When you receive new coordinates, update the state, which will cause the driver's Marker on the map to move to the new position. For a smoother experience, you can animate the marker's movement.

Conceptual Code for the Customer's App:

```
import React, { useEffect, useState, useRef } from 'react';
import MapView, { Marker } from 'react-native-maps';
import io from 'socket.io-client';

const socket = io('YOUR_NESTJS_SERVER_URL');
const aRouteId = 'route-123'; // The route the customer is subscribed
to

const CustomerMapView = () => {
  const [driverLocation, setDriverLocation] = useState(null);
  const mapMarkerRef = useRef(null);

  useEffect(() => {
    // 1. Subscribe to the route's room
    socket.emit('subscribeToRoute', aRouteId);

    // 2. Listen for location updates
    socket.on('driverLocationUpdated', (location) => {
      setDriverLocation(location);
      // Optional: Animate marker to new position for smoothness
      mapMarkerRef.current?.animateMarkerToCoordinate(location, 500);
    });

    return () => {
      socket.off('driverLocationUpdated'); // Clean up listener
    };
  }, []);

  return (
    <MapView style={{ flex: 1 }} initialRegion={{...}}>
```

```
      {driverLocation && (
        <Marker
          ref={mapMarkerRef}
          coordinate={driverLocation}
          title="Driver"
        >
          {/* You can use a custom car icon here */}

        </Marker>
      )}
    </MapView>
  );
};
```

By connecting these three parts, you create a robust, real-time system for your customers to track their ride.

Question: *But I want to share driver location if he click ride start button. If he click ride end button, it will end location share.*

Answer: Of course. Tying location sharing to a "Start" and "End" button is the correct and standard way to build this feature. It gives the driver control and saves battery life.
Here's how you can modify the previous setup to work exactly like that. The core idea is to make your backend the **gatekeeper**—it will only broadcast a driver's location if it knows the ride is officially active.

---

1. Driver's App: The Controller

The driver's app will now control when the ride and location sharing begins and ends.

1. **Add State and Buttons:** In your Expo app, manage the ride's status with a state variable (e.g., isRideActive). You'll have two buttons, "Start Ride" and "End Ride."
2. **"Start Ride" Logic:** When the driver clicks "Start Ride":
   - Update the state: setIsRideActive(true).
   - **Emit a startRide event** to the backend via WebSocket, telling the server this route is now live.
   - **Begin watching and sending location** using Location.watchPositionAsync, just as before.
3. **"End Ride" Logic:** When the driver clicks "End Ride":
   - Update the state: setIsRideActive(false).
   - **Emit an endRide event** to the backend, telling it to stop sharing.
   - **Stop watching for location updates.** You must call the remove() method on your

location subscription to stop the GPS and save the battery.

Conceptual Code for the Driver's App:

JavaScript

```
import React, { useState, useRef } from 'react';
import { Button, View } from 'react-native';
import * as Location from 'expo-location';
import io from 'socket.io-client';

const socket = io('YOUR_NESTJS_SERVER_URL');

const DriverRideControls = () => {
  const [isRideActive, setIsRideActive] = useState(false);
  const locationSubscription = useRef(null);
  const aRouteId = 'route-123'; // The driver's assigned route

  const startRide = async () => {
    setIsRideActive(true);
    socket.emit('startRide', { routeId: aRouteId });

    // Start tracking
    locationSubscription.current = await Location.watchPositionAsync(
      { accuracy: Location.Accuracy.High, timeInterval: 10000 },
      (location) => {
        socket.emit('updateLocation', {
          routeId: aRouteId,
          latitude: location.coords.latitude,
          longitude: location.coords.longitude,
        });
      }
    );
  };

  const endRide = () => {
    setIsRideActive(false);
    socket.emit('endRide', { routeId: aRouteId });

    // Stop tracking
```

```
   if (locationSubscription.current) {
     locationSubscription.current.remove();
   }
 };

 return (
   <View>
    {!isRideActive ? (
      <Button title="Start Ride" onPress={startRide} />
    ) : (
      <Button title="End Ride" onPress={endRide} color="red" />
    )}
   </View>
 );
};
```

---

2. Backend (NestJS): The Gatekeeper

Your backend will now track which rides are active and only share locations for those rides.

1. **Track Active Rides:** Keep a list (like a Set) of all currently active routeIds in your gateway.
2. **Add New Event Handlers:**
   - **startRide:** When this event is received, add the routeId to your activeRides set. You can also notify the customers in that room that their ride has begun.
   - **endRide:** When this event is received, remove the routeId from the activeRides set.
3. **Add a Check:** This is the most important change. In your existing updateLocation handler, **first check if the ride is active** before broadcasting the location.

Conceptual Code for NestJS Gateway:

TypeScript

```
// in your-app.gateway.ts
import { WebSocketGateway, SubscribeMessage, WebSocketServer, MessageBody } from
'@nestjs/websockets';
import { Server, Socket } from 'socket.io';
```

```
@WebSocketGateway()
export class AppGateway {
  @WebSocketServer()
  server: Server;

  // Use a Set to store the IDs of all active routes
  private activeRides = new Set<string>();

  @SubscribeMessage('startRide')
  handleStartRide(@MessageBody() data: { routeId: string }) {
    this.activeRides.add(data.routeId);
    // Notify customers that the ride has started
    this.server.to(data.routeId).emit('rideStarted');
  }

  @SubscribeMessage('endRide')
  handleEndRide(@MessageBody() data: { routeId: string }) {
    this.activeRides.delete(data.routeId);
    // Notify customers that the ride has ended
    this.server.to(data.routeId).emit('rideEnded');
  }

  @SubscribeMessage('updateLocation')
  handleLocationUpdate(@MessageBody() data: { routeId: string; latitude: number; longitude: number }) {
    // THE GATEKEEPER CHECK: Only broadcast if the ride is in the active set
    if (this.activeRides.has(data.routeId)) {
      this.server.to(data.routeId).emit('driverLocationUpdated', {
        latitude: data.latitude,
        longitude: data.longitude,
      });
    }
  }

  // ... (customer subscription logic remains the same)
}
```

---

3. Customer's App: The Viewer

The customer's app doesn't need many changes, but it can be made smarter by listening to the new rideStarted and rideEnded events to provide a better user experience.

- **Before rideStarted:** Show a message like "Waiting for driver to start the ride."
- **After rideStarted:** Display the map and the live driver marker.
- **After rideEnded:** Hide the marker and show a "You have arrived!" message.