Lecture 05

React Native Navigation

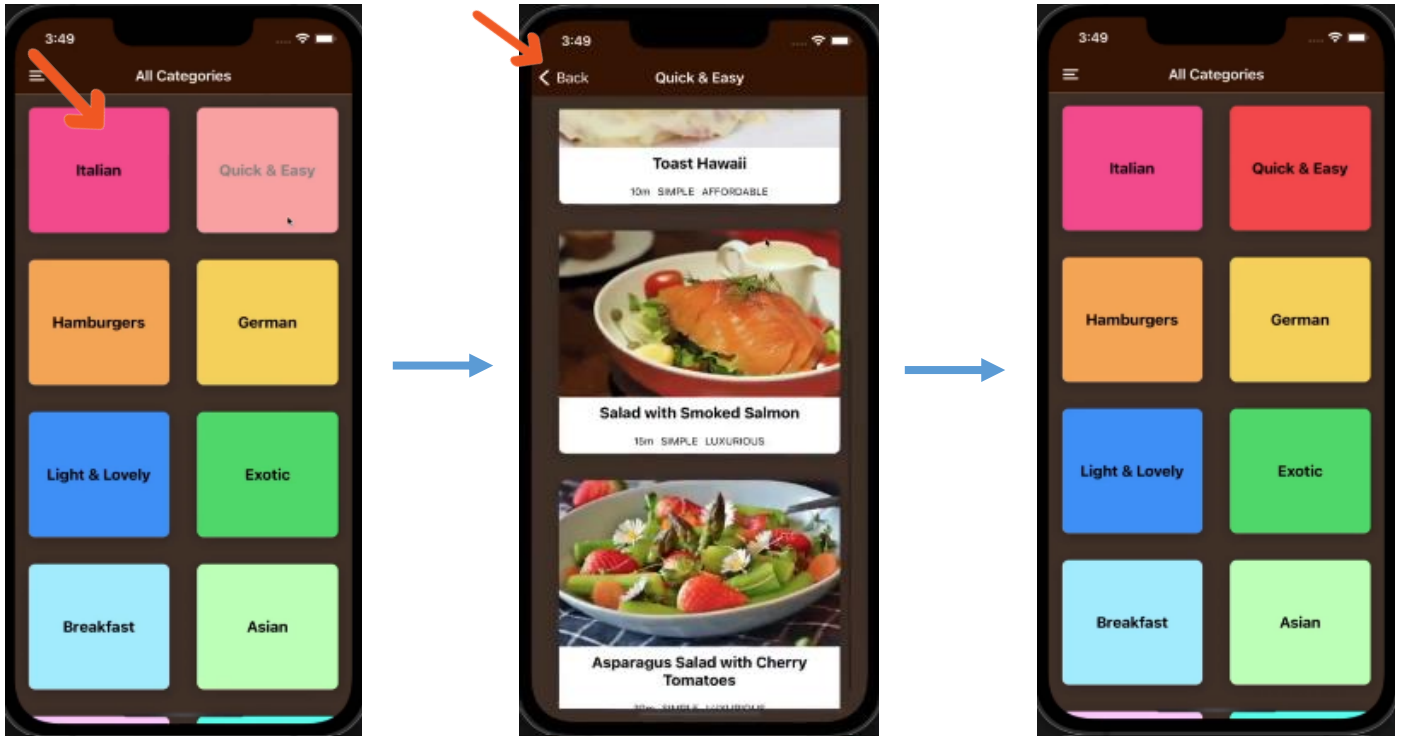
Contents

- What is Navigation?
- Creating a native stack navigator
  - Static API vs Dynamic API
- Passing data to screens
  - `useContext` hook
- Moving between screens (routes)
- Passing parameters when navigating between routes

What is Navigation?

- **Navigation:** the ability to move between different screens within an app.
- Allows users to interact with various app functionalities.

- Complex interactions are possible:
  - Going back to the previous screen
  - Animated transitions between screens
  - Switching between different screens using tap



React Navigation: A simple app

- This basic app has two screens: Home & About
  - It starts up with Home screen
- User can navigate back & forth between these screens
- Solve issues related to app’s state

Installing the native stack navigator library

- To use the native stack navigator, we need to install:

```
npm install @react-navigation/native-stack
```

Installing the elements library

- The `@react-navigation/elements` library provides a set of components that are designed to work well with React Navigation.
  - We'll use a few of these components later.

```
npm install @react-navigation/elements
```

React Navigation: **Static** vs **Dynamic** APIs

- **Static:**
  - Uses `createNativeStackNavigator()` to define the entire navigation configuration.
  - More straightforward, suitable for simpler, fixed navigation structure.
  - Suitable when screens and their options are known upfront.
- **Dynamic:**
  - Uses `Stack.Navigator` and `Stack.Screen` components to define the navigation structure.
  - Dynamically add/modify screens based on the app's state → more flexible.
  - Suitable for changing navigation structure at runtime

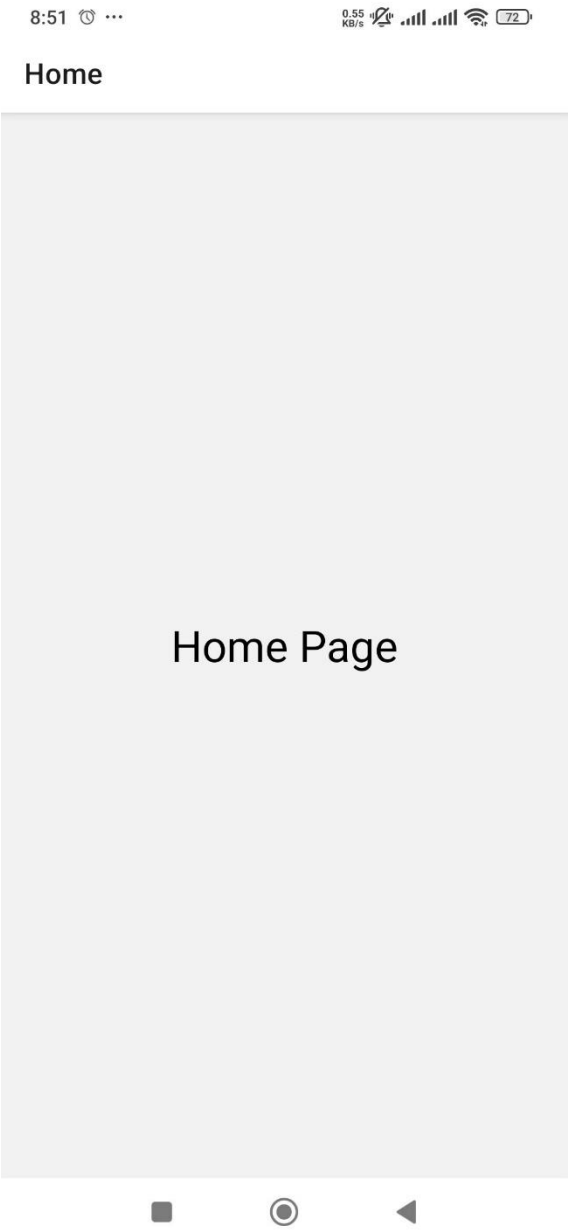
    e.g. adding/removing screens based on user actions

Creating a native stack navigator (Static API)

- The `createNativeStackNavigator` function takes a configuration object and returns a stack navigator.
  - The object contains screens and customization options.
  - The screens are React components that will be displayed by the navigator.
- The `createStaticNavigation` function takes the navigator created earlier and returns a component that can be rendered in the app.
  - It's only called once in an app.

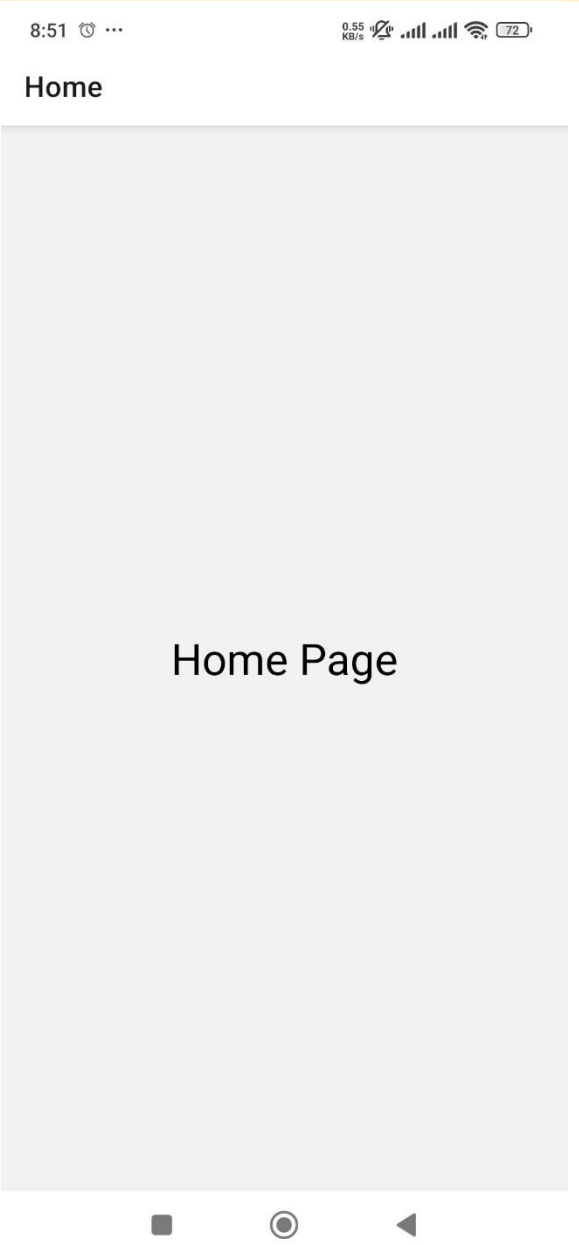
Creating a native stack navigator (Static)

```
import { createStaticNavigation } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

function HomeScreen() {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{ fontSize: 30 }}>Home Page</Text>
    </View>
  );
}

function AboutScreen() { // similar to HomeScreen }

const RootStack = createNativeStackNavigator({
  screens: {
    Home: HomeScreen,
    About: AboutScreen
  },
});

const Navigation = createStaticNavigation(RootStack);

export default function App() {
  return <Navigation />;
}
```



Creating a native stack navigator (Static)

- The styles you see for the navigation bar and the content area are the default configuration for a stack navigator.
- The casing of the route name doesn't matter -- you can use lowercase home or capitalized Home, it's up to you.



Creating a native stack navigator (Static)

- **Navigation using Static API:**
  - Uses `createStaticNavigation`, which is not a standard approach.
  - Screens are defined inside an object instead of JSX .
  - No need for `NavigationContainer`, since `createStaticNavigation` automatically wraps navigation.

```
const RootStack = createNativeStackNavigator({
  screens: {
    Home: HomeScreen,
    About: AboutScreen
  },
});

const Navigation = createStaticNavigation(RootStack);

export default function App() {
  return <Navigation />;
}
```

Creating a native stack navigator (Static)

- Limitations:
  - `createStaticNavigation` is not commonly used in React Navigation.
  - Defining screens as an object reduces flexibility.
  - This approach *may not be supported* in newer React Navigation versions.

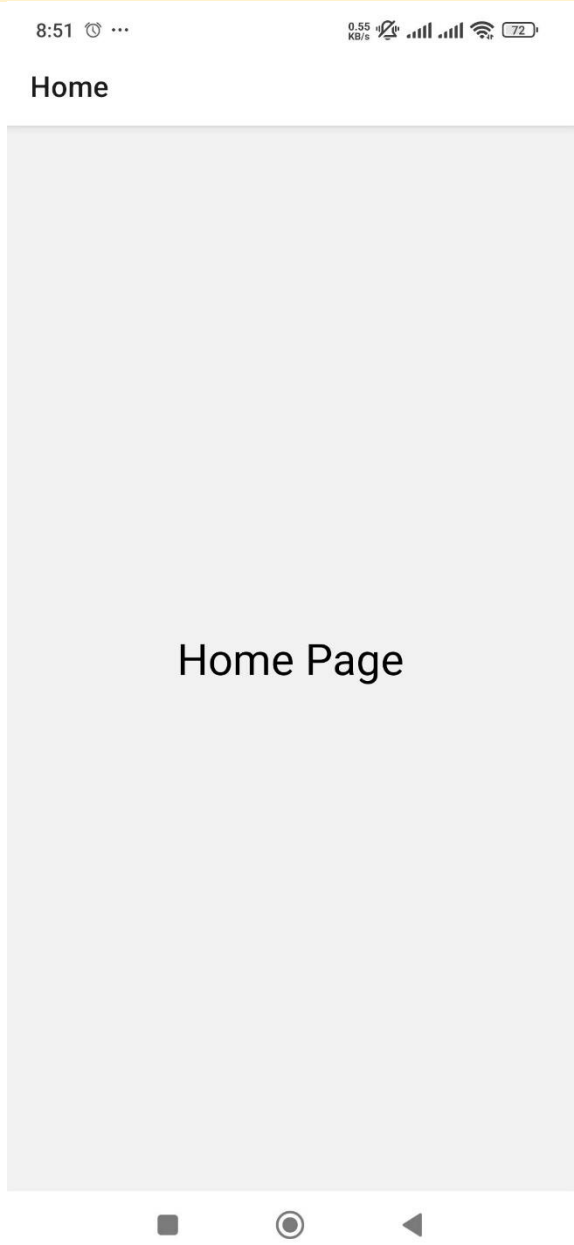
```
const RootStack = createNativeStackNavigator({
  screens: {
    Home: HomeScreen,
    About: AboutScreen
  },
});

const Navigation = createStaticNavigation(RootStack);

export default function App() {
  return <Navigation />;
}
```

Creating a native stack navigator (Dynamic API)

- The `createNativeStackNavigator` function returns an object containing 2 properties: `Screen` and `Navigator`.
  - These components are used to create & configure the navigator structure. `Navigator` should contain `Screen` children.
- The `NavigationContainer` component manages the navigation tree and navigation state.
  - It must wrap all the navigators in the app.
  - It's usually rendered as the root component of an app (the component exported from `App.js`)

Creating a native stack navigator (Dynamic)

```jsx
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

function HomeScreen() {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{ fontSize: 30 }}>Home Page</Text>
    </View>
  );
}

function AboutScreen() { // similar to HomeScreen }

const Stack = createNativeStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="About" component={AboutScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```



Creating a native stack navigator (Dynamic)

- Characteristics of Dynamic API:
  - Uses `createNativeStackNavigator` with JSX, making it more readable and maintainable.
  - Screens are defined inside `<Stack.Navigator>`, making it easy to add more screens.
  - `NavigationContainer` is used to wrap the navigator, which is required in React Navigation.

```
<NavigationContainer>
    <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="About" component={AboutScreen} />
    </Stack.Navigator>
</NavigationContainer>
```

Creating a native stack navigator (Dynamic)

- Advantages:
  - Dynamic API is the official, recommended way to set up navigation in React Native.
  - More flexible and scalable, allowing easy modifications and configurations.
  - `NavigationContainer` helps manage navigation state properly.
- Warning:
  - When using the Dynamic API, the `component` prop accepts a component, not a render function. Don't pass an inline function (e.g. `component={() => <HomeScreen />}`), or your component will unmount and remount, losing all state, when the parent component re-renders.

Configuring the initial route (Static)

- A Screen in a Navigator is also called a Route.
  - The term "route" emphasizes the idea that navigating to different screens is similar to navigating to different URLs in a web application.
- By default, the first Route is rendered. But it’s possible to select any other Route to be the initial one.
- For example:

```
const RootStack = createNativeStackNavigator({
    initialRouteName: 'About',
    screens: {
        Home: HomeScreen,
        About: AboutScreen
    },
});
```

Configuring the initial route (Dynamic)

- With Dynamic API, configuring the initial Route is done using a prop of the `Navigator` component.
- For example:

```
export default function App() {
    return (
        <NavigationContainer>
            <Stack.Navigator initialRouteName='About'>
                <Stack.Screen name="Home" component={HomeScreen} />
                <Stack.Screen name="About" component={AboutScreen} />
            </Stack.Navigator>
        </NavigationContainer>
    );
}
```

Specifying options (Static)

- Each Screen can specify some options for the Navigator, such as the title to render in the header.
- To specify the options, use an object with a `screen` property *instead of* specifying the screen component.

```
const RootStack = createNativeStackNavigator({
    initialRouteName: 'Home',
    screens: {
        Home: {
            screen: HomeScreen,
            options: {
                title: "Welcome"
            }
        },
        About: AboutScreen
    },
});
```

Specifying options (Static)

- Sometimes we will want to specify the same options for all of the screens in the navigator. For that, we can add a screenOptions property to the configuration.

```
const RootStack = createNativeStackNavigator({
    initialRouteName: 'Home',
    screenOptions: {
        headerStyle: { backgroundColor: 'tomato' },
    },
    ...
}
```

Specifying options (Dynamic)

- Any customization options can be passed in the `options` prop for each screen component.

```
<NavigationContainer>
    <Stack.Navigator initialRouteName='About'>
        <Stack.Screen name="Home" component={HomeScreen}
            options={{ title: 'Welcome' }} />
        <Stack.Screen name="About" component={AboutScreen} />
    </Stack.Navigator>
</NavigationContainer>
```

Specifying options (Dynamic)

- To specify the same options for all routes in a navigator, pass a `screenOptions` prop to the navigator:

```
<NavigationContainer>
    <Stack.Navigator initialRouteName='About'
        screenOptions={{
            headerStyle: { backgroundColor: 'tomato' }
        }} >
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="About" component={AboutScreen} />
    </Stack.Navigator>
</NavigationContainer >
```

Passing additional props

- What if we want to pass additional props to a screen?
  - As specified earlier, we cannot use an arrow function. This won't work:

```
<Stack.Screen name="Home" component={() => <HomeScreen prop1="value" />} />
```

- There are 2 approaches:
  - Use React context and wrap the navigator with a context provider to pass data to the screens (recommended).
  - Use a render callback for the screen instead of specifying a `component` prop:

```
<Stack.Screen name="Home">
    {(props) => <HomeScreen {...props} prop1="value" />}
</Stack.Screen>
```

The `useContext` hook

- A React Hook that lets you create context data (values) in a parent component and retrieve them from its descendant components.
  - The component what retrieves the data can be *many levels down* in the component tree.
- How to create and use a context?
  - Create a context provider and wrap it around the components where you want to retrieve values.
  - Call `useContext` function in a descendant component to retrieve the context value(s).

Creating & using Context Provider

- First, use the `createContext` function to create a Context.
  - The parameter specifies the context's initial value.

```
import { createContext, useContext } from 'react';

const ScreenNameContext = createContext(null);
```

- Then, wrap the context's provider around the components that will use the context value.

```
<ScreenNameContext.Provider value={scrNames}>
    <NavigationContainer>
        <Stack.Navigator>
            <Stack.Screen name="Home" component={HomeScreen} />
            <Stack.Screen name="About" component={AboutScreen} />
        </Stack.Navigator>
    </NavigationContainer >
</ScreenNameContext.Provider>
```

Retrieving Context Data from a component

- Technically, any can retrieve the context data.
- However, it's recommended that the component is one of the context provider's descendant.

```
function AboutScreen() {
    const screenNames = useContext(ScreenNameContext);
    return (
        <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
            <Text>{screenNames.about}</Text>
        </View>
    );
}
```

```
function AboutScreen() {
  const screenNames = useContext(ScreenNameContext);
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>{screenNames.about}</Text>
    </View>
  );
}

export default function App() {
  const scrNames = {
    home: 'Home Screen',
    about: 'About screen'
  };
  return (
    <ScreenNameContext.Provider value={scrNames}>
      <NavigationContainer>
        <Stack.Navigator>
          <Stack.Screen name="Home" component={HomeScreen} />
          <Stack.Screen name="About" component={AboutScreen} />
        </Stack.Navigator>
      </NavigationContainer >
    </ScreenNameContext.Provider>
  );
}
```

Moving between screens

- On a web browser, we'd be able to write something like this:

```
<a href="about.html">About Us</a>
```

- Another way to write this would be:

```
<a onclick="() => {
  window.location.href = 'about.html';
}"> About Us</a>
```

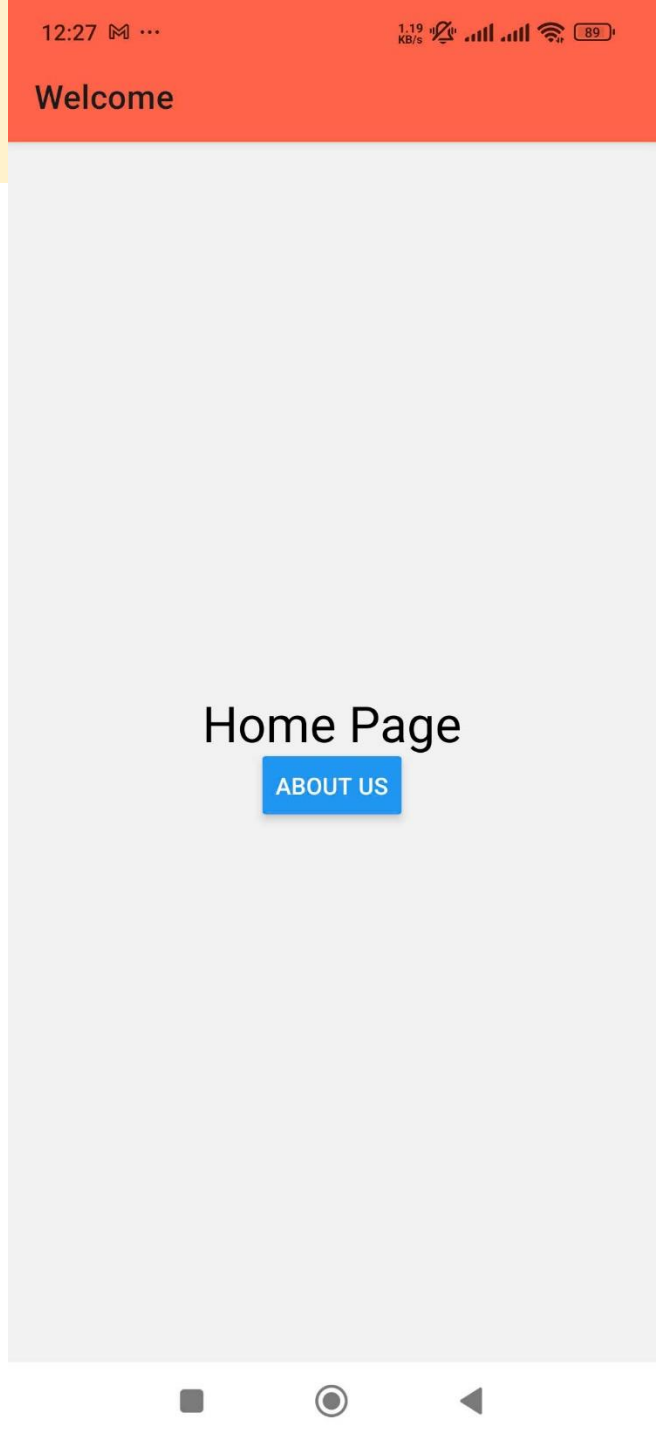
- We can do something similar to the latter, but rather than using `window.location`, we'll use the navigation object that's accessible in the screen components.

Navigating to a new screen

```
import { useNavigation } from '@react-navigation/native';

function HomeScreen() {
  const navigation = useNavigation();
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{ fontSize: 30 }}>Home Page</Text>
      <Button
        title="About Us"
        onPress={() => navigation.navigate('About')}
      />
    </View>
  );
}
```

Navigating to a new screen

- `navigation` – the `navigation` object is returned from the `useNavigation` hook.
- `navigate('About')` – we call the `navigate` function with the name of the route that we'd like to move the user to.
- Note:
  - If we call `navigation.navigate` with a route name that we haven't defined in a navigator, it'll print an error in development builds and nothing will happen in production builds (in other words, we can only navigate to routes that have been defined in the navigator).



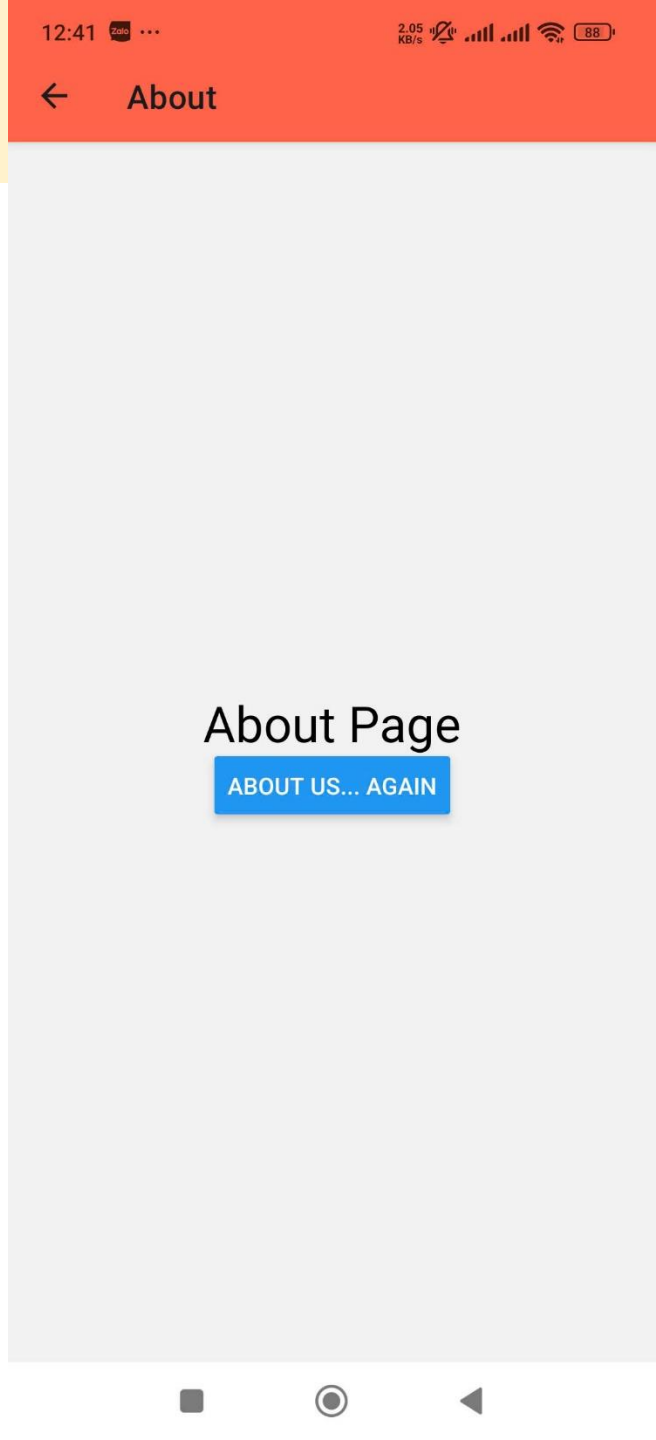
The issue of navigating to the same screen

- If you're already on the About screen and call `navigation.navigate('About')`, nothing happens.
- The `navigate` function only navigates to the screen if it's not already active.
- **What if** you *actually want* to navigate to the same screen again and again?
  - For instance, when you want to pass some unique data into a screen. e.g. you have a `ProductDetail` screen that will display a product that is passed into it as a prop.

Using `push` to add multiple instances

- If you want to open a new instance of the About screen, use `navigation.push('About')`.
- Each time `push` is called, a new About screen instance is added to the navigation stack.

```
<Button
    title="About Us... again"
    onPress={() => navigation.push('About')}
/>
```



Difference between navigate and push

- `navigate('About')` → Does nothing if already on the About screen.

- `navigate.push('About')` → Creates a new instance of the About screen.

➢ This approach is useful when **passing unique data** to each instance of a screen.

Going back

- Automatic Back Button in Header
  - The native stack navigator automatically provides a back button if there's a previous screen in the stack.
  - If there's only one screen, the back button won't appear.
- Manually Triggering Back Navigation
  - Use `navigation.goBack()` to programmatically navigate to the previous screen.
- On **Android**, React Navigation automatically calls `goBack()` when the user presses the *physical* back button.

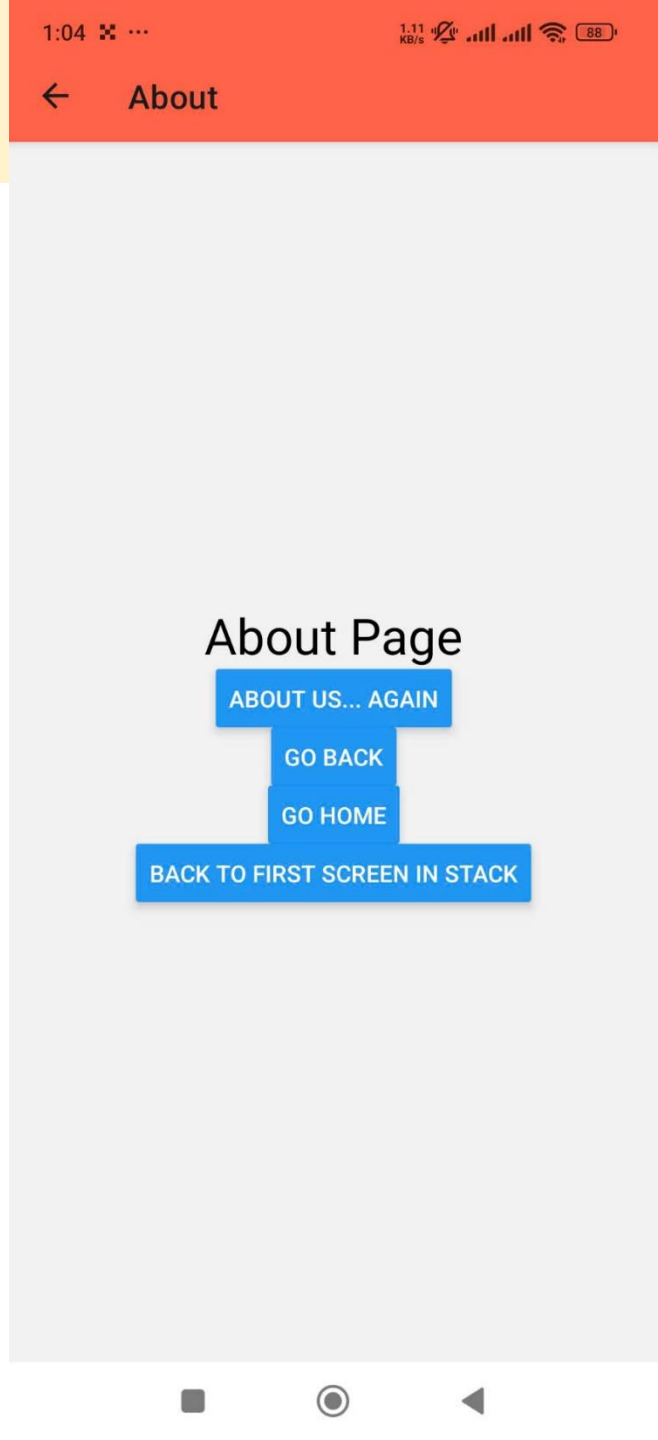
Going back

```
function AboutScreen() {
    const navigation = useNavigation();
    return (
        <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
            <Text style={{ fontSize: 30 }}>About Page</Text>
            <Button
                title="About Us... again"
                onPress={() => navigation.push('About')}
            />
            <Button
                title="Go Back"
                onPress={() => navigation.goBack()}
            />
        </View>
    );
}
```

Going back multiple screens

- Another common requirement is to be able to go back *multiple* screens -- for example, if you are several screens deep in a stack and want to dismiss all of them to go back to the first screen.

- In this case, we know that we want to go back to `Home` so we can use `popTo('Home')`. Another alternative would be `navigation.popToTop()`, which goes back to the first screen in the stack.



Going Back Multiple Screens

```
function AboutScreen() {
    const navigation = useNavigation();
    return (
        <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
            <Text style={{ fontSize: 30 }}>About Page</Text>
            <Button title="About Us... again"
                onPress={() => navigation.push('About')} />
            <Button title="Go Back" onPress={() => navigation.goBack()} />
            <Button title="Go Home" onPress={() => navigation.popTo('Home')} />
            <Button title="Back to First Screen in Stack"
                onPress={() => navigation.popToTop()} />
        </View>
    );
}
```

Going Back

- Example use cases:
  - `goBack()` → Go back one screen.
  - `popTo('Home')` → Return directly to the Home screen.
  - `popToTop()` → Reset navigation to the first screen in the stack.

Passing parameters to routes

- There are two pieces to this:
  1. Pass params to a route by putting them in an object as a second parameter to the `navigation.navigate` function:

```
navigation.navigate('RouteName', { /* params go here */ })
```

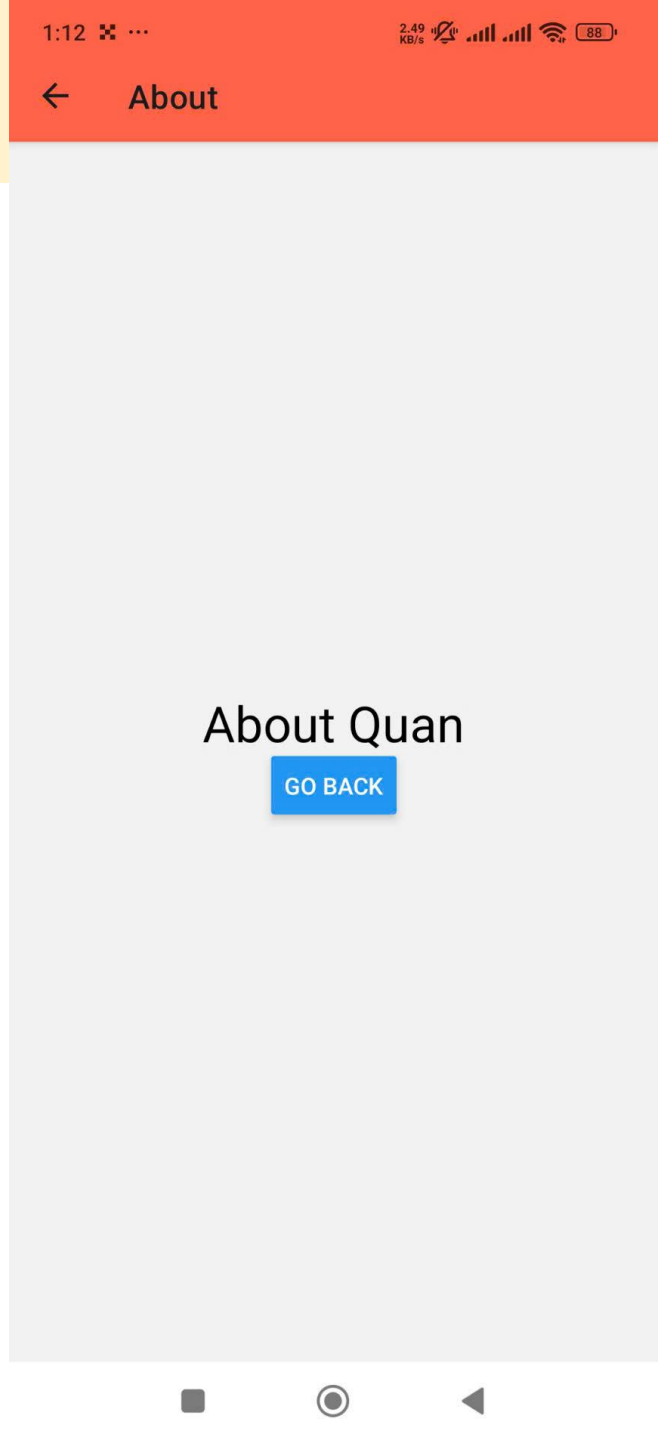
  2. Read the params in your screen component: `route.params`

Passing parameters to routes

```
function HomeScreen() {
  const navigation = useNavigation();
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{ fontSize: 30 }}>Home Page</Text>
      <Button
        title="About Us"
        onPress={() => navigation.navigate('About', { name: 'Quan' })}
      />
    </View>
  );
}
```

Receiving passed parameters

```
function AboutScreen({ route }) {
  const navigation = useNavigation();
  const { name } = route.params;
  return (
    <View style={{
      flex: 1, alignItems: 'center',
      justifyContent: 'center'
    }}>
      <Text style={{ fontSize: 30 }}>
        About {name}
      </Text>
      <Button title="Go Back"
        onPress={() => navigation.goBack()} />
    </View>
  );
}
```



Initial params

- You can also pass some initial params to a screen.
  - If you didn't specify any params when navigating to this screen, the initial params will be used.
  - They are also shallow merged with any params that you pass. Initial params can be specified using the `initialParams` prop:

```
<Stack.Screen
  name="About"
  component={AboutScreen}
  initialParams={{ name: "Us" }}
/>
```

Updating params

- Screens can also update their params, like they can update their state. The `navigation.setParams` method lets you update the params of a screen.
- Basic usage:

```
navigation.setParams({
    name: 'Vinh'
})
```

- Avoid using `setParams` to update screen options such as title. If you need to update options, use `setOptions` instead.

Passing params to a previous screen

- Params aren't only useful for passing data to a new screen, but also useful to pass data to a previous screen.
  - **Example:** you have a screen with a "Create Post" button which opens a new screen to create a post. After creating the post, you want to pass the data of the post back to the previous screen.
- To achieve this, you can use the `popTo` method to go back to the previous screen as well as pass params to it:

```
navigation.popTo('Home', { post: postText });
```

What should be in params?

- Only the minimal data required to display a screen
- Params should not be used for state management.
  - If data is needed across multiple screens, it should be stored in a global store or cache.
- **Incorrect approach**: Passing the entire user object when navigating to a Profile screen can lead to outdated data, increased code complexity, and long, problematic URLs.
- **Correct approach:** Pass only the `userId` and retrieve the user data from a global store.

What should be in params

```
// Don't do this
navigation.navigate('Profile', {
    user: {
        id: 21,
        firstName: 'Jane',
        lastName: 'Done',
        age: 25
    }
});

// Do this
navigation.navigate('Profile', { userId: 21 });
```