Lecture 06

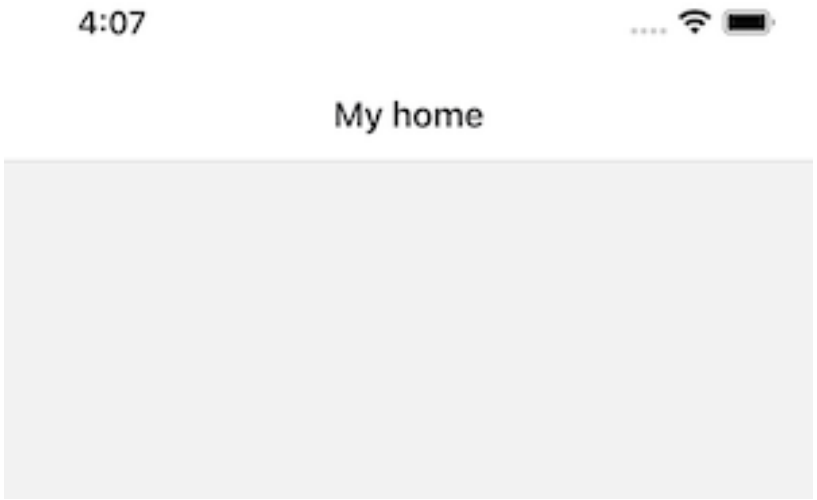
React Native Navigation (2)

Time Tracking App

Configuring the header bar

- Use the `title` property in `options` to set a fixed title for each screen.

```
<Stack.Screen name="Home"
component={HomeScreen}
    options={{ title: 'My home' }} />
```



Using Params in the Title

- To set a dynamic title based on parameters, options should be a function that receives route and returns a title.
- The argument that is passed in to the options function is an object with the following properties:
  - navigation - The navigation object for the screen.
  - route - The route object for the screen

```
<Stack.Screen
    name="Profile"
    component={ProfileScreen}
    options={({ route }) => ({ title: route.params.name })}
/>
```

Updating the Title Dynamically with `setOptions`

- It's often necessary to update the `options` configuration for the active screen from the mounted screen component itself. We can do this using `navigation.setOptions`

```
<Button
    onPress={() => navigation.setOptions({ title: 'Updated!' })}>
    Update the title
</Button>
```

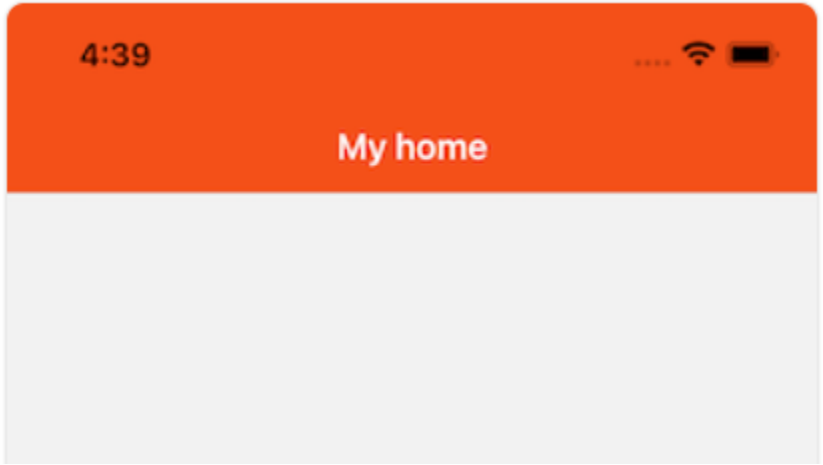
Updating the Title Dynamically with `setOptions`

- It's often necessary to update the `options` configuration for the active screen from the mounted screen component itself. We can do this using `navigation.setOptions`

```
<Button
    onPress={() => navigation.setOptions({ title: 'Updated!' })}
>
    Update the title
</Button>
```

Customizing Header Styles

- Three key properties for styling:
  - `headerStyle:` Styles the header background.
  - `headerTintColor:` Sets the color for the back button and title.
  - `headerTitleStyle:` Customizes font properties for the title.



Customizing Header Styles

```
<Stack.Screen name="Home" component={HomeScreen} options={{
    title: 'My home',
    headerStyle: { backgroundColor: '#f4511e' },
    headerTintColor: '#fff',
    headerTitleStyle: { fontWeight: 'bold' },
}} />
```

Sharing common `options` across screens

- Instead of repeating styles for each screen, set `screenOptions` in `Stack.Navigator`.

```
<Stack.Navigator screenOptions={{
    headerStyle: { backgroundColor: '#f4511e' },
    headerTintColor: '#fff',
    headerTitleStyle: { fontWeight: 'bold' },
}}>
    <Stack.Screen name="Home" component={HomeScreen}
        options={{ title: 'My home' }} />
    <Stack.Screen name="Details" component={DetailsScreen} />
</Stack.Navigator>
```

Replacing the Title with a Custom Component

- Use `headerTitle` to replace the text title with a custom component, such as an image or button.

```
function LogoTitle() {
    return (
        <Image style={{ width: 50, height: 50 }}
            source={require('@expo/snack-static/react-native-logo.png')} />
    );
}

<Stack.Screen name="Home" component={HomeScreen}
    options={{ headerTitle: (props) => <LogoTitle {...props} /> }}
/>
```

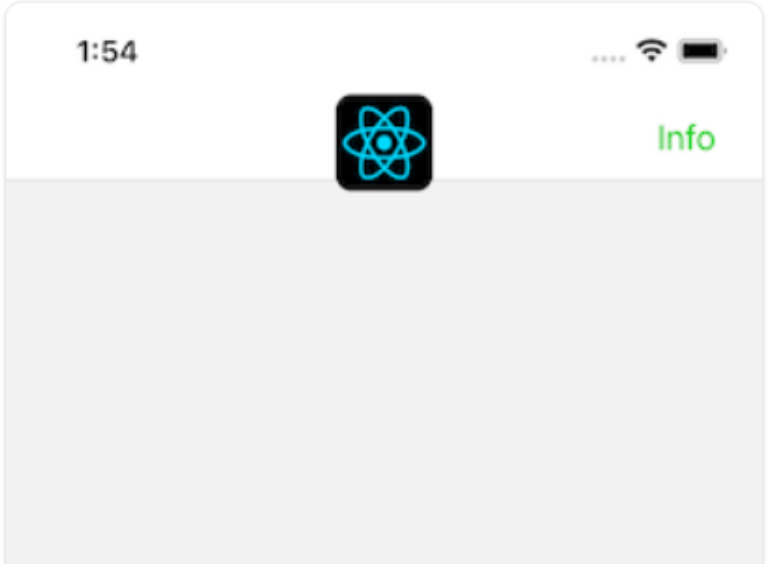
Difference Between `title` and `headerTitle`

- `title` is used for multiple navigation types like tab bars and drawers.
- `headerTitle` is specific to stack navigators and replaces the default `Text` component.

Header buttons

- You can place buttons in the header using `headerLeft` (left side) or `headerRight` (right side).

```
<Stack.Screen
    name="Home"
    component={HomeScreen}
    options={{
        headerRight: () => (
            <Button onPress={() =>
alert('This is a button!')}>Info</Button>
        ),
    }}
/>
```



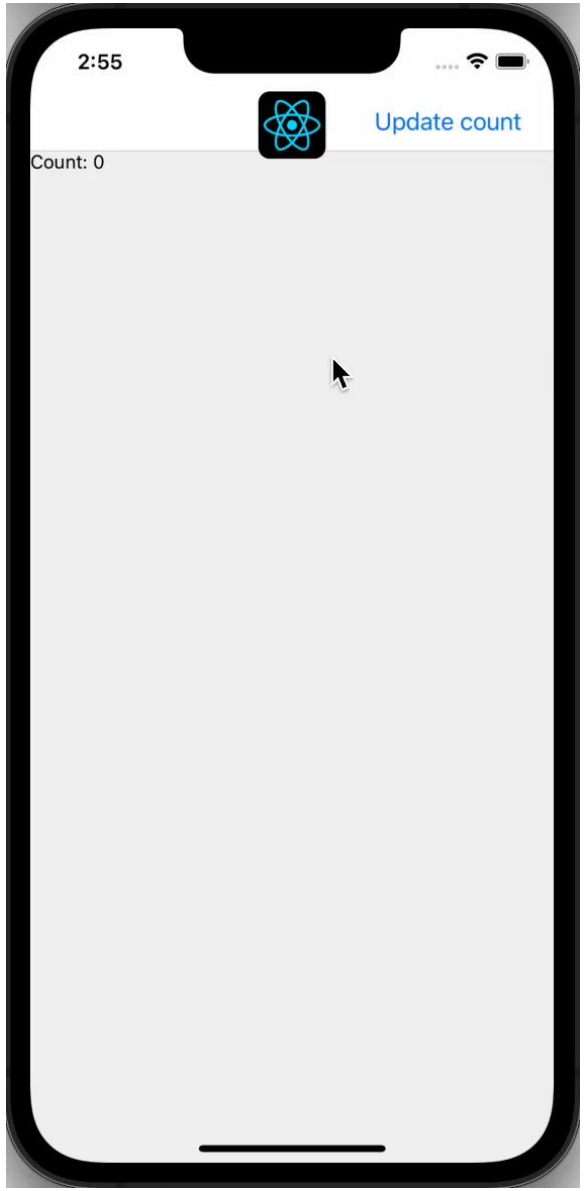
Header interaction with its screen component

- When we define our button this way, the `this` variable in `options` is not the `HomeScreen` instance, so you can't call `setState` or any instance methods on it.
  - It's common to want the buttons in your header to interact with the screen that the header belongs to.
- To make the button interact with the screen's state, use `navigation.setOptions`.

Header interaction with its screen component

- For this use case, we need to use `navigation.setOptions` to update our options.
  - By using `navigation.setOptions` inside the screen component, we get access to screen's props, state, context etc.

- Here we update the `headerRight` with a button with `onPress` handler that has access to the component's state and can update it.



Header interaction with its screen component

```
function HomeScreen() {
    const navigation = useNavigation();
    const [count, setCount] = React.useState(0);

    React.useEffect(() => {
        navigation.setOptions({
            headerRight: () => (
                <Button onPress={() => setCount((c) => c + 1)}>
                    Update count
                </Button>
            ),
        });
    }, [navigation]);

    return <Text>Count: {count}</Text>;
}
```

Customizing the Back Button

- React Navigation provides platform-specific defaults for back buttons. On iOS, the back button shows the title of the previous screen when space allows.
- Customization Options
  - `headerBackTitle`: Changes the back button text.
  - `headerBackTitleStyle`: Styles the back button text.
  - `headerBackImageSource`: Sets a custom image for the back button.

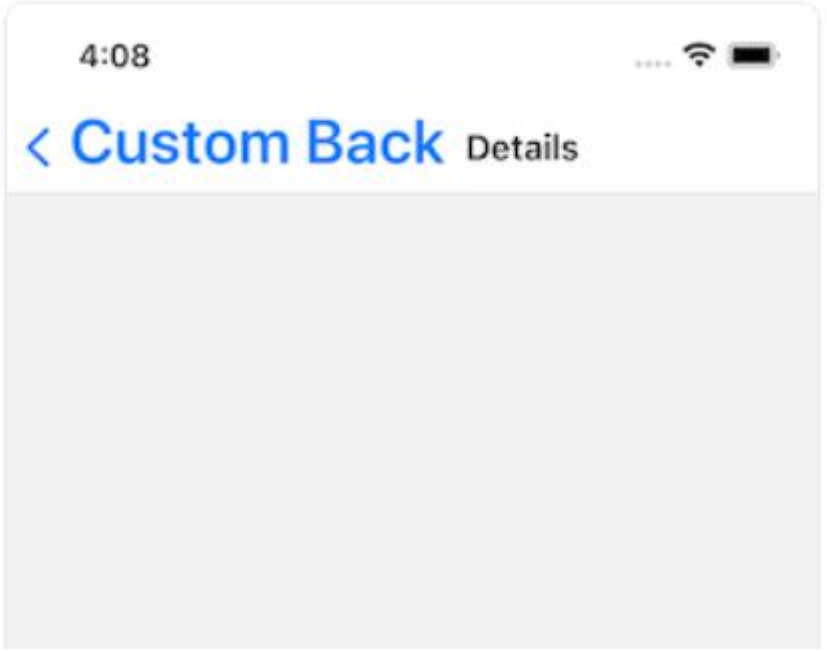
Customizing the Back Button

```
<Stack.Navigator>
    <Stack.Screen name="Home" component={HomeScreen} />
    <Stack.Screen
        name="Details"
        component={DetailsScreen}
        options={{
            headerBackTitle: 'Custom Back',
            headerBackTitleStyle: { fontSize: 30 },
        }} />
</Stack.Navigator>
```

Overriding the Back Button

- If you need a completely custom back button, use `headerLeft`

```
<Stack.Screen
  name="Details"
  component={DetailsScreen}
  options={{
    headerLeft: () => (
      <Button
        onPress={() => alert('Custom Back Pressed')}
      >
        Back
      </Button>
    ),
  }}
/>
```



What is Nesting Navigators?

- It means placing a navigator inside a screen of another navigator.
  - Example: A **Tab** Navigator inside a **Stack** Navigator.

Key Behaviors of Nested Navigators

- **Independent Navigation History**: Each navigator maintains its own back navigation.
- **Separate Screen Options**: A nested navigator's options (e.g., title) don't affect the parent.
- **Independent Params**: Params of a nested screen are not accessible from parent/child screens.
- **Navigation Actions Bubble Up**: If a child navigator cannot handle an action, the parent will.
- **Navigator-Specific Methods**: Methods like openDrawer are only available inside that navigator.
- **No Inherited Parent Events**: Screens inside a nested navigator don't receive parent events.
- **Parent UI Renders on Top**: A drawer placed inside a stack appears under the stack's header.

Example: Nesting Navigators

```
const HomeScreen = () => (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
        <Text>Home Screen</Text>
    </View>
);

const SettingsScreen = () => (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
        <Text>Settings Screen</Text>
    </View>
);

const DetailsScreen = () => (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
        <Text>Details Screen</Text>
    </View>
);
```

Example: Nesting Navigators

```
const Tab = createBottomTabNavigator();

function MyTabs() {
    return (
        <Tab.Navigator>
            <Tab.Screen name="Home" component={HomeScreen} />
            <Tab.Screen name="Settings" component={SettingsScreen} />
        </Tab.Navigator>
    );
}
```

Example: Nesting Navigators

```
const { Navigator, Screen } = createNativeStackNavigator();

export default function App() {
    return (
        <NavigationContainer>
            <Navigator>
                <Screen name="Tabs" component={MyTabs} />
                <Screen name="Details" component={DetailsScreen} />
            </Navigator>
        </NavigationContainer>
    );
}
```

Navigating in Nested Navigators

- Navigate to a screen inside a nested navigator:

```
navigation.navigate('Tabs', {
    screen: 'Settings'
});
```

- Passing params when navigating:

```
navigation.navigate('Tabs', {
    screen: 'Settings',
    params: { user: 'jane' }
});
```

Avoiding Multiple Headers

- Prevent duplicate headers by hiding the parent header:

```
<Stack.Screen
    name="Home" component={HomeTabs}
    options={{ headerShown: false }}
/>
```

- Hide all headers:

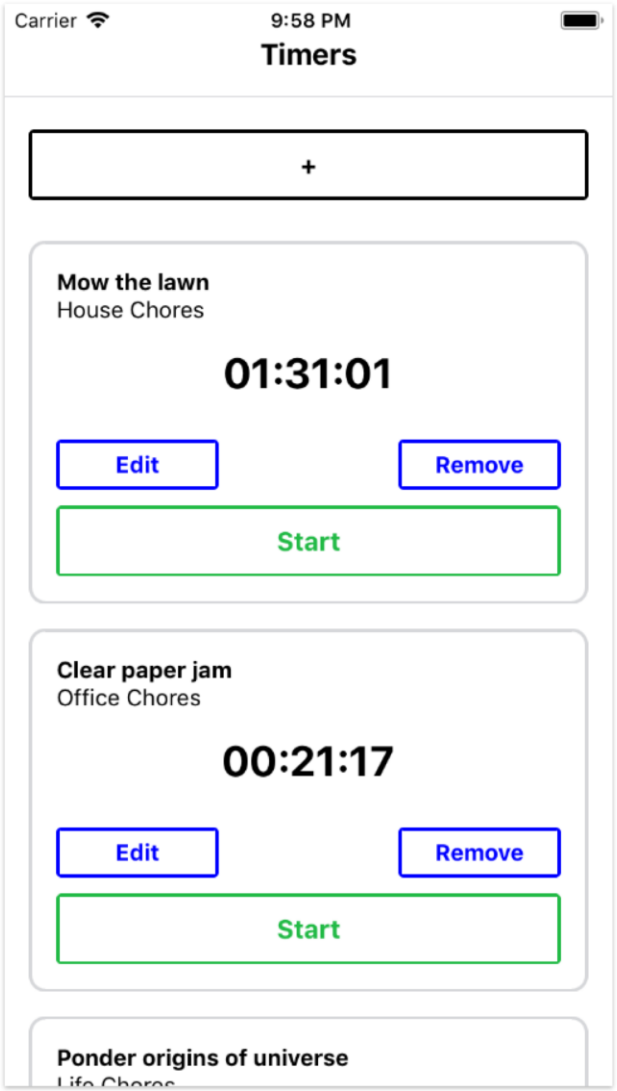
```
<Stack.Navigator screenOptions={{ headerShown: false }}>
```

Time Tracking App

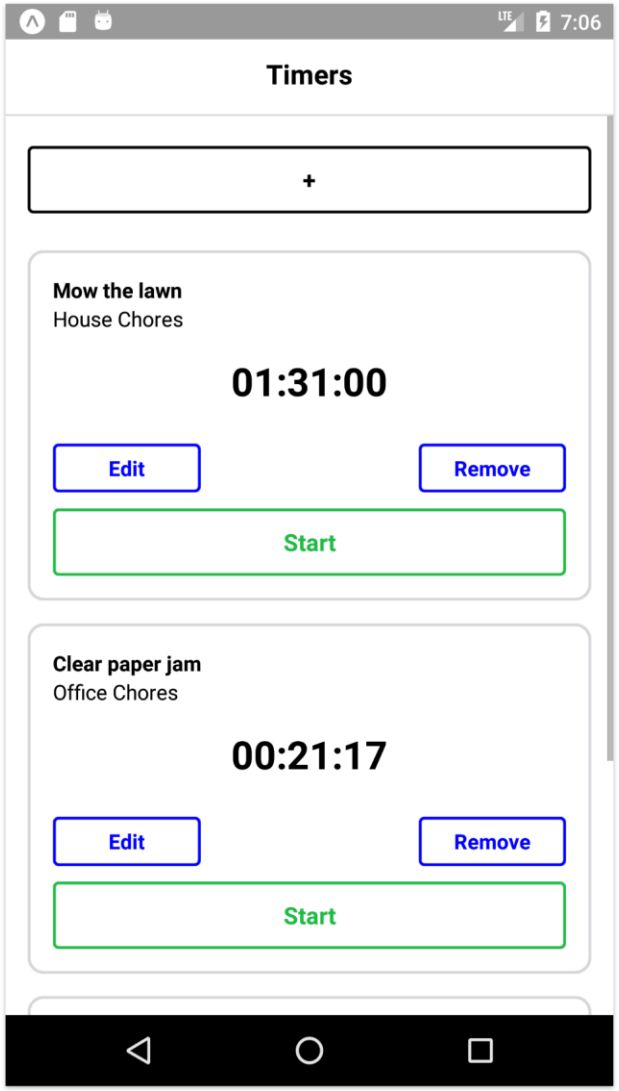
- Creating the Time Tracking App
  - App analysis & design
- 7-step development process
- Learning some components
  - `TouchableOpacity`

The finished app will look like this:

iOS

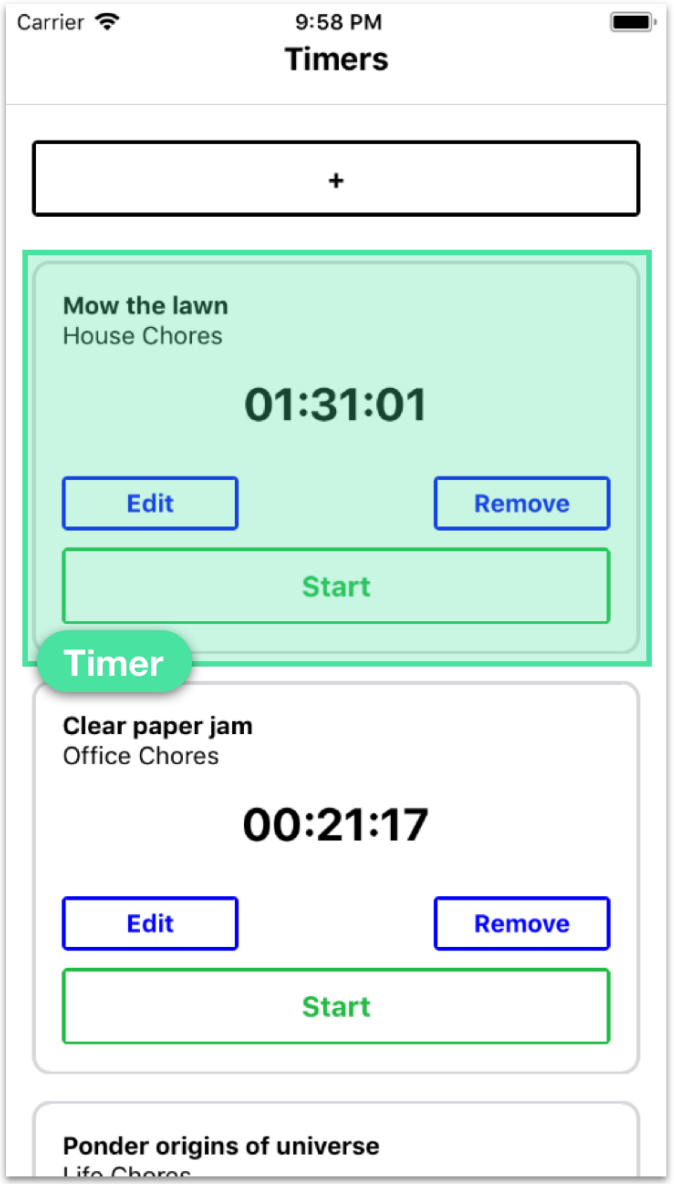


Android



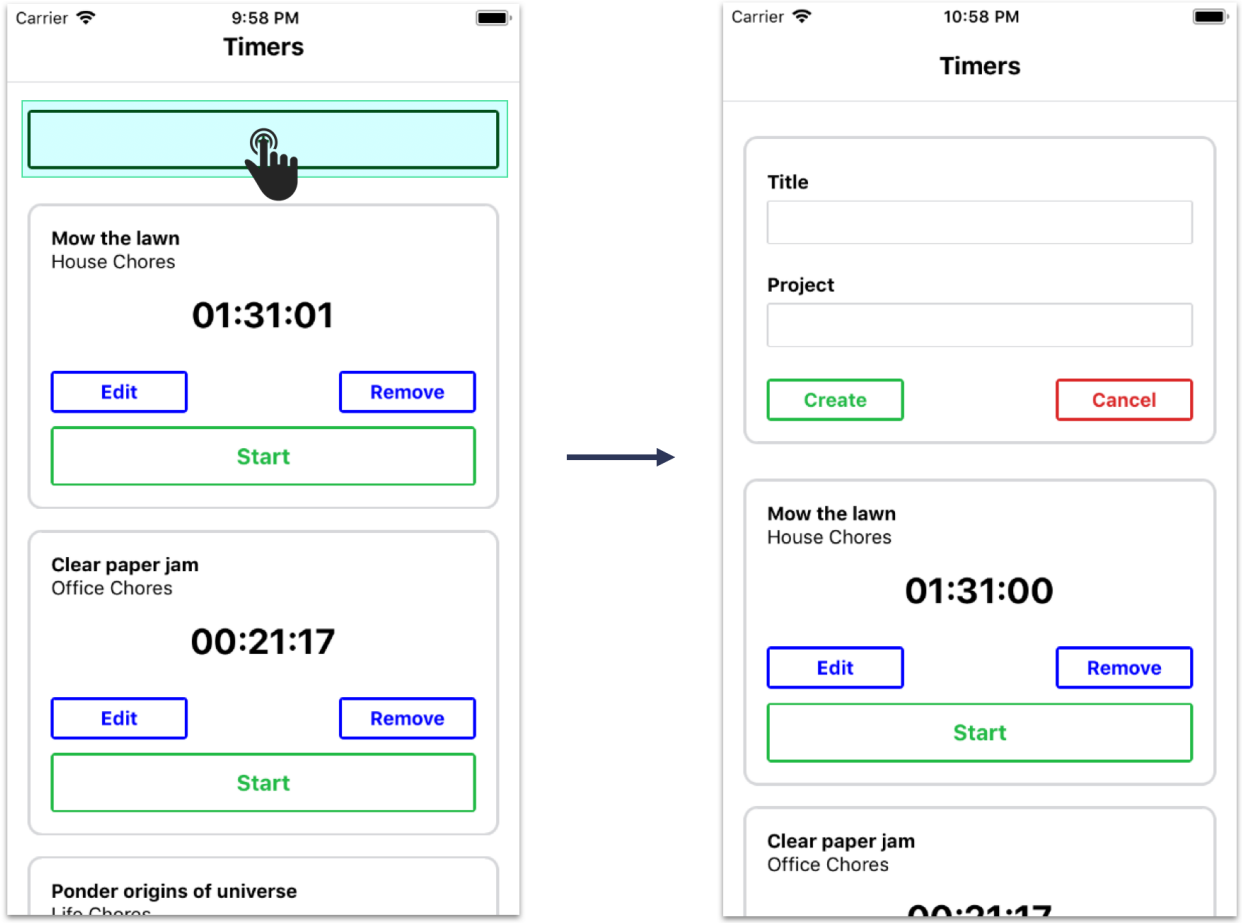
Breaking the app into components

- We'd want a `Timer` component for each timer



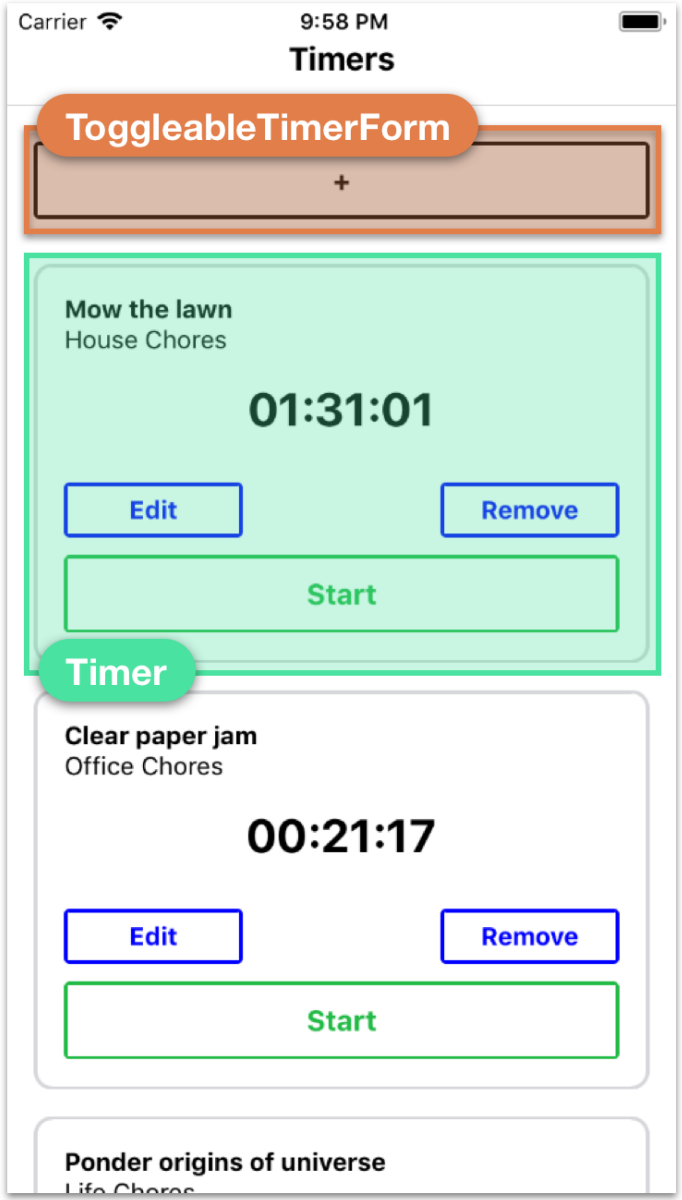
A form to add timers

- When the “+” button is pressed, the component changes into a form.
- When the form is closed, the component changes back into a “+” button.



A form to add timers

- **Solution:** A custom component called `ToggleableTimerForm`
  - Decides to render a button or a form
  - Based on a boolean state

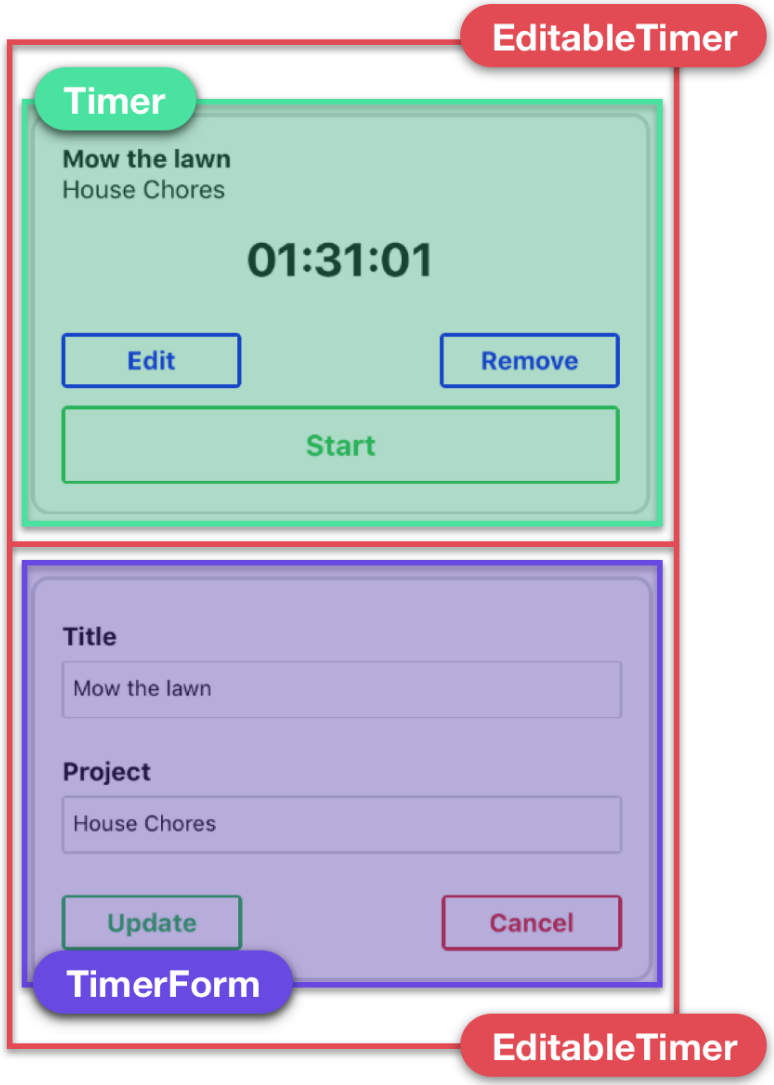


The Timer component’s functionalities

- Timer turns into a form when the user clicks “Edit”
- Timer deletes itself when “Remove” is pressed
- Time has buttons for starting and stopping

The Timer component's functionalities

- `EditableTimer`
  - Decides to render either a `Timer` or a `TimerForm` component
  - Based on a boolean state
- Also, all buttons in the app have similar appearance.
  - Use a component called `TimerButton`



7-step process for building React Native application

1. Break the app into components
2. Build a static version of the app
   (Use props instead of state for values)
3. Determine what should be stateful
4. Determine in which component each piece of state should live
5. Hardcode initial states
6. Add inverse data flow
   (Create functions where states live and pass it down to the component which handles user events)
7. Add network communication (if applicable)

The app’s component tree

- `App`: Root container
  - `ToggleableTimerForm`: Displays a form to add new timer
    - `TimerButton`: Displays the “+” button
    - `TimerForm`: Displays a new timer’s create form
  - `EditableTimer`: Displays either a timer or an edit form
    - `Timer`: Displays a given timer
    - `TimerForm`: Displays a given timer’s edit form

Build a static version of the app

Components, props & minimal event handling

Explaination of Components & Props

```
<ToggleableTimerForm isOpen={false} />
```

```
<EditableTimer
    id="1"
    title="Mow the lawn"
    project="House Chores"
    elapsed="3126"
    isRunning
/>
```

```
<EditableTimer
    id="2"
    title="Bake squash"
    project="Kitchen Chores"
    elapsed="4251"
    editFormOpen
/>
```

For boolean attributes:

- present -> true
- absent -> false

`EditableTimer` component

```
export default function EditableTimer(props) {
    if (props.editFormOpen) {
        return <TimerForm
            id={props.id}
            title={props.title}
            project={props.project}
        />;
    } else {
        return (
            <Timer
                id={props.id}
                title={props.title}
                project={props.project}
                elapsed={props.elapsed}
                isRunning={props.isRunning}
            />
        );
    }
}
```

`TimerForm` component

```
export default function TimerForm(props) {
    const submitText = props.id ? 'Update' : 'Create';
    // code omitted
}
```

Text inputs:

```
<View style={styles.attributeContainer}>
    <Text style={styles.textInputTitle}>Project</Text>
    <View style={styles.textInputContainer}>
        <TextInput
            style={styles.textInput}
            underlineColorAndroid="transparent"
            defaultValue={project}
        />
    </View>
</View>
```

`TimerButton` component

```
export default function TimerButton(
    { color, title, small, onPress }) {
    return (
        <TouchableOpacity
            style={[
                styles.button,
                { borderColor: color }
            ]}
            onPress={onPress}>
            <Text style={[
                styles.buttonText,
                small ? styles.small : styles.large,
                { color }
            ]}>{title}</Text>
        </TouchableOpacity>
    );
}
```

(*) On pressed down, the opacity of the component is decreased, dimming it.

`ToggleableTimerForm` component

```
export default function ToggleableTimerForm({ isOpen }) {
    return (
        <View style={[
            styles.container,
            !isOpen && styles.buttonPadding
        ]}>
            {
                isOpen ?
                    <TimerForm />
                    :
                    <TimerButton title="+" color="black" />
            }
        </View>
    );
}
```

[Utility function] Human-readable time

```
export const millisecondsToHuman = (ms) => {
    const seconds = Math.floor((ms / 1000) % 60);
    const minutes = Math.floor((ms / 1000 / 60) % 60);
    const hours = Math.floor(ms / 1000 / 60 / 60);

    const humanized = [
        pad(hours.toString(), 2),
        pad(minutes.toString(), 2),
        pad(seconds.toString(), 2),
    ].join(':');

    return humanized;
};
```

[Utility function] Zero padding

```
const pad = (numberString, size) => {
    let padded = numberString;
    while (padded.length < size) {
        padded = `0${padded}`;
    }
    return padded;
};
```

(*) `` `0${padded}` `` is a template literal

[Utility function] Generating timer data

```
export const newTimer = (attrs = {}) => {
    const timer = {
        title: attrs.title || 'Timer',
        project: attrs.project || 'Project',
        id: uuidv4(),
        elapsed: 0,
        isRunning: false,
    };

    return timer;
};
```

uuidv4 is a secure way to generate unique ids

(very low chance of duplication)

```
> npm install uuidv4
```

Reference: https://scaleyourapp.com/uuid-guid-oversimplified-are-they-really-unique/

How to make the timer running?

- Based on `isRunning` boolean value
- May execute code with `useEffect` hook

```
setTimeout(someFunction, TIME_INTERVAL)
```

```
elapsed = isRunning ? elapsed + TIME_INTERVAL : elapsed;
```

- Every `useEffect` hook may return a clean-up function that executes when:
  - component is unmount
    (removed from the virtual DOM)
  - is going to be re-rendered
    (before each render)

Implementing the app

Follow 7-step process as close as possible

Teacher gives live instructions to complete the app...

Determine what should be stateful

~= State criteria =~

- **Is it passed in from a parent via props?**
  If so, it probably isn't state.
- **Does it change over time?**
  If not, it probably isn't state.
  A key criterion of stateful data: it changes!
- **Can you compute it based on any other state or props in your component?**
  If so, it's not state.

(*) For simplicity, we want to strive to represent state with as few data points as possible.

In which component should a state be placed?

~= **Facebook's Thinking in React guide**=~

- For each piece of state:
  - Identify all components that are rendered based on it
  - Find a common ancestor component
    (a component above all those components in the hierarchy)
  - Either the common ancestor or another component higher up in the hierarchy should own the state.
  - If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common ancestor component.

Hard-code initial states

```
const [timers, setTimers] = useState([
    {
        title: 'Mow the lawn',
        project: 'House Chores',
        id: uuidv4(),
        elapsed: 0,
        isRunning: true,
    },
    {
        title: 'Bake donuts',
        project: 'Kitchen Chores',
        id: uuidv4(),
        elapsed: 0,
        isRunning: false,
    },
]);
```