Core APIs, Part 2

In the first part of this section, we covered a variety of React Native APIs for accessing device information. In this part, we'll focus on one fundamental feature of mobile devices: the keyboard.

This is a **code checkpoint**. If you haven't been coding along with us but would like to start now, we've included a snapshot of our current progress in the sample code for this book.

If you haven't created a project yet, you'll need to do so with:

```
$ expo init messaging --template blank@sdk-36 --yarn
```

Then, copy the contents of the directory `messaging/1` from the sample code into your new `messaging` project directory.

The keyboard

Keyboard handling in React Native can be very complex. We're going to learn how to manage the complexity, but it's a challenging problem with a lot of nuanced details.

Our UI is currently a bit flawed: on iOS, when we focus the message input field, the keyboard opens up and covers the toolbar. We have no way of switching between the image picker and the keyboard. We'll focus on fixing these issues.

We're about to embark on a deep dive into keyboard handling. We'll cover some extremely useful APIs and patterns – however, you shouldn't feel like you have to complete the entire chapter now. Feel free to stop here and return again when you're actively building a React Native app that involves the keyboard.

Why it's difficult

Keyboard handling can be challenging for many reasons:

- The keyboard is enabled, rendered, and animated natively, so we have much less control over its behavior than if it were a component (where we control the lifecycle).
- We have to handle a variety of asynchronous events when the keyboard is shown, hidden, or resized, and update our UI accordingly. These events are somewhat different on iOS and Android, and even slightly different in the simulator compared to a real device.
- The keyboard works differently on iOS and Android at a fundamental level. On iOS, the keyboard appears *on top* of the existing UI; the existing UI doesn't resize to avoid the keyboard. On Android, the keyboard *resizes* the UI above it; the existing UI will shrink to fit in the available space. We generally want interactions to feel similar on both platforms, despite this fundamental difference.
- Keyboards interact specially with certain native elements e.g. `ScrollView`. On iOS, dragging downward on a `ScrollView` can dismiss the keyboard at the same rate of the pan gesture.
- Keyboards are user-customizable on both platforms, meaning there's an almost unlimited number of shapes and sizes our UI has to handle.

In this app, we'll attempt to achieve a native-quality messaging experience. Ultimately though, there will be a few aspects that don't quite feel native. It's extremely difficult to get an app with complex keyboard interactions to feel *perfect* without dropping down to the native level. If you can't achieve the right experience in React Native, consider writing a native module for the screen that interacts heavily with the keyboard. This is part of the beauty of React Native – you can start with a JavaScript version in your initial implementation of a screen or feature, then seamlessly swap it out for a native implementation when you're certain it's worth the time and effort.

If you're lucky, you'll be able to find an existing open source native component that does exactly that!

`KeyboardAvoidingView`

In the first chapter, we demonstrated how to use the `KeyboardAvoidingView` component to move the UI of the app out from under the keyboard. This component is great for simple use cases, e.g. focusing the UI on an input field in a form.

When we need more precise control, it's often better to write something custom. That's what we'll do here, since we need to coordinate the keyboard with our custom image input method.

Our goal here is for our image picker to have the same height as the native keyboard, in essence acting as a custom keyboard created by our app. We'll want to smoothly animate the transition between these two input methods.

For a demo of the desired behavior, you can try playing around with the completed app (it's the same app as the previous section):

- On Android, you can scan this QR code from within the Expo app:

- On iOS, you can build the app and preview it on the iOS simulator or send the link of the project URL to your device like we've done in previous chapters.

On managing complexity

Since this problem is fairly complicated, we're going to break it down into 3 parts, each with its own component:

- `MeasureLayout` - This component will measure the available space for our messaging UI
- `KeyboardState` - This component will keep track of the keyboard's visibility, height, etc
- `MessagingContainer` - This component will displaying the correct IME (text, images) at the correct size

We'll connect them so that `MeasureLayout` renders `KeyboardState`, which in turn renders `MessagingContainer`.

We *could* build one massive component that handles everything, but this would get very complicated and be difficult to modify or reuse elsewhere.

`Keyboard`

We'll need to measure the available space on the screen and the keyboard height ourselves, and adjust our UI accordingly. We'll keep track of whether the keyboard is currently transitioning. And we'll animate our UI to transition between the different keyboard states.

To do this, we'll use the `Keyboard` API. The `Keyboard` API is the lower-level API that `KeyboardAvoidingView` uses under the hood.

On iOS, the keyboard uses an animation with a special easing curve that's hard to replicate in JavaScript, so we'll hook into the native animation directly using the `LayoutAnimation` API. `LayoutAnimation` is one of the two main ways to animate our UI (the other being `Animated`). We'll cover animation more in a later chapter.

Measuring the available space

Let's start by measuring the space we have to work with. We want to measure the space that our `MessageList` can use, so we'll measure from below the status bar (anything above our `MessageList`) to the bottom of the screen. We need to do this to get a numeric value for height, so we can transition between the height when the keyboard isn't visible to the height when the keyboard is visible. Since the keyboard doesn't actually take up any space in our UI, we can't rely on `flex: 1` to take care of this for us.

Measuring in React Native is always asynchronous. In other words, the first time we render our UI, we have no general-purpose way of knowing the height. If the content above our `MessageList` has a fixed height, we can calculate the initial height by taking `Dimensions.get('window').width` and subtracting the height of the content above our `MessageList` – however, this is not very flexible. Instead, let's create a container `View` with a flexible height `flex: 1` and measure it on first render. After that, we'll always have a numeric value for height.

We can measure this `View` with the `onLayout` prop. By passing a callback to `onLayout`, we can get the `layout` of the `View`. This `layout` contains values for `x`, `y`, `width`, and `height`.

messaging/components/MeasureLayout.js

```
import Constants from 'expo-constants';
import { Platform, StyleSheet, View } from 'react-native';
import PropTypes from 'prop-types';
import React from 'react';

export default class MeasureLayout extends React.Component {
  static propTypes = {
    children: PropTypes.func.isRequired,
  };

  state = {
    layout: null,
  };

  handleLayout = event => {
    const { nativeEvent: { layout } } = event;

    this.setState({
      layout: {
        ...layout,
        y:
          layout.y +
          (Platform.OS === 'android' ? Constants.statusBarHeight : 0),
      },
```

```
    });
  };

  render() {
    const { children } = this.props;
    const { layout } = this.state;

    // Measure the available space with a placeholder view set to
    // flex 1
    if (!layout) {
      return (
        <View onLayout={this.handleLayout} style={styles.container} />
      );
    }

    return children(layout);
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
});
```

Here we render a placeholder `View` with an `onLayout` prop. When called, we update `state` with the new `layout`.

Most React Native components accept an `onLayout` function prop. This is conceptually similar to a React lifecycle method: the function we pass is called every time the component updates its dimensions. We need to be careful when calling `setState` within this function, since `setState` may cause the component to re-render, in which case `onLayout` will get called again… and now we're stuck in an infinite loop!

We have to compensate for the solid color status bar we use on Android by adjusting

the `y` value, since the status bar height isn't included in the layout data. We can do this by merging the existing properties of layout, `...layout`, and an updated `y` value that includes the status bar height.

We use a new pattern here for propagating the `layout` into the children of this component: we require the `children` prop to be a function. When we use our `MeasureLayout` component, it will look something like this:

```
<MeasureLayout>
  {layout => <View ... />}
</MeasureLayout>
```

> This pattern is similar to having a `renderX` prop, where `X` indicates what will be rendered, e.g. `renderMessages`. However, using `children` makes the hierarchy of the component tree more clear. Using the `children` prop implies that these children components are the main thing the parent renders. As an analogy, this pattern is similar to choosing between `export default` and `export X`. If there's only one variable to export from a file, it's generally more clear to go with `export default`. If there's a variable with the same name as the file, or a variable that seems like the primary purpose of the file, you would also likely export it with `export default` and export other variables with `export X`. Similarly, you should consider using `children` if this prop is the "default" or "primary" thing a component renders. Ultimately this is an API style preference. Even if you choose not to use it, it's useful to be aware of the pattern since you may encounter it when using open source libraries.

We're now be able to get a precise height which we can use to resize our UI when the keyboard appears and disappears.

Keyboard events

We have the initial height for our messaging UI, but we need to update the height when the keyboard appears and disappears. The `Keyboard` object emits events to let us know when it appears and disappears. These events contain layout information, and on iOS, information about the animation that will/did occur.

KeyboardState

Let's create a new component called `KeyboardState` to encapsulate the keyboard event handling logic. For this component, we're going to use the same pattern as we did for `MeasureLayout`: we'll take a `children` function prop and call it with information about the keyboard layout.

We can start by figuring out the `propTypes` for this component. We know we're going to have a `children` function prop. We're also going to consume the `layout` from the `MeasureLayout` component, and use it in our keyboard height calculations.

messaging/components/KeyboardState.js

```
import { Keyboard, Platform } from "react-native";
import PropTypes from 'prop-types';
import React from 'react';

export default class KeyboardState extends React.Component {
  static propTypes = {
    layout: PropTypes.shape({
      x: PropTypes.number.isRequired,
      y: PropTypes.number.isRequired,
      width: PropTypes.number.isRequired,
      height: PropTypes.number.isRequired,
    }).isRequired,
    children: PropTypes.func.isRequired,
  };

  // ...
}
```

Now let's think about the `state`. We want to keep track of 6 different values, which we'll pass into the `children` of this component:

- `contentHeight`: The height available for our messaging content.
- `keyboardHeight`: The height of the keyboard. We keep track of this so we set our image picker to the same size as the keyboard.

- `keyboardVisible`: Is the keyboard fully visible or fully hidden?
- `keyboardWillShow`: Is the keyboard animating into view currently? This is only relevant on iOS.
- `keyboardWillHide`: Is the keyboard animating out of view currently? This is only relevant on iOS, and we'll only use it for fixing visual issues on the iPhone X.
- `keyboardAnimationDuration`: When we animate our UI to avoid the keyboard, we'll want to use the same animation duration as the keyboard. Let's initialize this with the value `250` (in milliseconds) as an approximation.

messaging/components/KeyboardState.js

```
// ...

const INITIAL_ANIMATION_DURATION = 250;

export default class KeyboardState extends React.Component {
  // ...

  constructor(props) {
    super(props);

    const { layout: { height } } = props;

    this.state = {
      contentHeight: height,
      keyboardHeight: 0,
      keyboardVisible: false,
      keyboardWillShow: false,
      keyboardWillHide: false,
      keyboardAnimationDuration: INITIAL_ANIMATION_DURATION,
    };
  }

  // ...
}
```

Now that we've determined which properties to keep track of, let's update them based on keyboard events.

There are 4 `Keyboard` events we should listen for:

- `keyboardWillShow` (iOS only) - The keyboard is going to appear
- `keyboardWillHide` (iOS only) - The keyboard is going to disappear
- `keyboardDidShow` - The keyboard is now fully visible
- `keyboardDidHide` - The keyboard is now fully hidden

In `componentDidMount` we can add listeners to each keyboard event;

And in `componentWillUnmount` we can remove them:

```
// ...

  componentDidMount() {
    if (Platform.OS === 'ios') {
      this.subscriptions = [
        Keyboard.addListener(
          'keyboardWillShow',
          this.keyboardWillShow,
        ),
        Keyboard.addListener(
          'keyboardWillHide',
          this.keyboardWillHide,
        ),
        Keyboard.addListener('keyboardDidShow', this.keyboardDidShow),
        Keyboard.addListener('keyboardDidHide', this.keyboardDidHide),
      ];
    } else {
      this.subscriptions = [
        Keyboard.addListener('keyboardDidHide', this.keyboardDidHide),
        Keyboard.addListener('keyboardDidShow', this.keyboardDidShow),
      ];
    }
```

```
23    }
24
25    componentWillUnmount() {
26      this.subscriptions.forEach(subscription => subscription.remove());
27    }
28
29  // ...
```

We'll add the listeners slightly differently for each platform: on Android, we don't get events for `keyboardWillHide` or `keyboardWillShow`.

Storing subscription handles in an array is a common practice in React Native. We don't know exactly how many subscriptions we'll have until runtime, since it's different on each platform, so removing all subscriptions from an array is easier than storing and removing a reference to each listener callback.

Let's use these events to update `keyboardVisible`, `keyboardWillShow`, and `keyboardWillHide` in our `state`:

messaging/components/KeyboardState.js

```
// ...

keyboardWillShow = (event) => {
  this.setState({ keyboardWillShow: true });

  // ...
};

keyboardDidShow = () => {
  this.setState({
    keyboardWillShow: false,
    keyboardVisible: true,
  });

  // ...
};
```

```
keyboardWillHide = (event) => {
  this.setState({ keyboardWillHide: true });

  // ...
};

keyboardDidHide = () => {
  this.setState({
    keyboardWillHide: false,
    keyboardVisible: false
  });
};

// ...
```

The listeners `keyboardWillShow`, `keyboardDidShow`, and `keyboardWillHide` will each be called with an `event` object, which we can use to measure the `contentHeight` and `keyboardHeight`. Let's do that now, using `this.measure(event)` as a placeholder for the function which will perform measurements.

messaging/components/KeyboardState.js

```
// ...

keyboardWillShow = (event) => {
  this.setState({ keyboardWillShow: true });
  this.measure(event);
};

keyboardDidShow = (event) => {
  this.setState({
    keyboardWillShow: false,
    keyboardVisible: true,
  });
  this.measure(event);
};
```

```
keyboardWillHide = (event) => {
  this.setState({ keyboardWillHide: true });
  this.measure(event);
};

// ...
```

For iOS it would be sufficient to calculate measurements in the `keyboardWill*` events, since the `keyboardDid*` events should receive the same `event` parameter. However, since Android only supports the `keyboardDid*` events, we also need to use `keyboardDidShow`. Calculating measurements in `keyboardDidShow` on iOS shouldn't affect the app's behavior, but we could do this conditionally by checking `Platform.OS === 'android'` if we preferred.

We can use these events to keep track of the keyboard's current state. Each `event` object will have the following properties:

- `duration` - Duration of the keyboard animation. In practice, this is typically constant across all keyboard animations. This property only exists on iOS, so we'll use a constant to approximate it on Android.
- `easing` - Easing curve used by the keyboard animation. This will be the special easing curve called `'keyboard'`, which we can use to sync our own animations with the keyboard's. This property only exists on iOS, since there isn't a specific keyboard animation on Android. We'll use `'easeInEaseOut'` as a pleasant-looking default to approximate the keyboard animation on Android.
- `startCoordinates`, `endCoordinates` - An object containing keys `height`, `width`, `screenX`, and `screenY`. These refer to the start and end coordinates of the keyboard. Normally `height`, `width`, and `screenX` will stay the same. We can use `height` to determine the height of the keyboard. The `screenY` value refers to the top of the keyboard, which we can use to determine the remaining height available to render content.

To calculate the `contentHeight`, we can take the `screenY` (top coordinate of the keyboard) and subtract `layout.y` (top coordinate of our messaging component).

```
measure = (event) => {
  const { layout } = this.props;

  const {
    endCoordinates: { height, screenY },
    duration,
  } = event;

  this.setState({
    contentHeight: screenY - layout.y,
    keyboardHeight: height,
    keyboardAnimationDuration: duration || INITIAL_ANIMATION_DURATION,
  });
};

// ...
```

> Remember, `y` coordinates lower down on the screen are larger than those higher on the screen, so this calculation will resulting in a positive value.

Note that if a hardware keyboard is connected, the `height` of the keyboard will be `0` – we'll have to handle this specially later.

Let's propagate all of these values into the `children` of this component. We'll also propagate the height of the entire component as `containerHeight`.

messaging/components/KeyboardState.js

```
// ...

render() {
  const { children, layout } = this.props;
  const {
    contentHeight,
    keyboardHeight,
    keyboardVisible,
    keyboardWillShow,
```

```
    keyboardWillHide,
    keyboardAnimationDuration,
  } = this.state;

  return children({
    containerHeight: layout.height,
    contentHeight,
    keyboardHeight,
    keyboardVisible,
    keyboardWillShow,
    keyboardWillHide,
    keyboardAnimationDuration,
  });
}

// ...
```

When we use this component, it'll look roughly like this: we'll first wrap it in our `MeasureLayout` component, and pass the `layout` in as a prop. We can then render our content using the `keyboardInfo` object.

```
<MeasureLayout>
  {layout => (
    <KeyboardState layout={layout}>
      {keyboardInfo => /* ... */}
    </KeyboardState>
  )}
</MeasureLayout>
```

Alright, we're almost there! We have `MeasureLayout` and `KeyboardState`. The last component we need is `MessagingContainer` to render the content using the sizes we've calculated.

`MessagingContainer`

Let's create a new component `MessagingContainer` to render the correct Input Method Editor (IME) at the correct size.

Once again, let's figure out the `propTypes` first. This component is going to have a lot of `props`, since it's consuming data from the previous components we wrote, in addition to more `props` which we'll pass in from `App`.

The main job of this component is to display the correct IME at any given time. Let's define constants for each potential state:

- `NONE` - Don't show any IME.
- `KEYBOARD` - The text input is focused, so the keyboard should be visible.
- `CUSTOM` - Show our custom IME. In this case, we'll show our image picker, but we could show other kinds of input here if we wanted to.

Let's create an object to hold these. We'll also export it so that other components can easily use the correct string values.

messaging/components/MessagingContainer.js

```
import {
  BackHandler,
  LayoutAnimation,
  Platform,
  UIManager,
  View,
} from 'react-native';
import PropTypes from 'prop-types';
import React from 'react';

export const INPUT_METHOD = {
  NONE: 'NONE',
  KEYBOARD: 'KEYBOARD',
  CUSTOM: 'CUSTOM',
};
```

```
// ...
```

Now for the `propTypes`. We'll begin by declaring each of the values that will be passed from `KeyboardState`. We'll define `inputMethod` and `onChangeInputMethod` to handle switching between IMEs and notifying the parent of changes. We'll also support rendering content in both the keyboard area with `renderInputMethodEditor` and the main content area with `children`. In this case, `children` should be a normal React element rather than a function like in the two components we wrote previously.

messaging/components/MessagingContainer.js

```
// ...

export default class MessagingContainer extends React.Component {
  static propTypes = {
    // From `KeyboardState`
    containerHeight: PropTypes.number.isRequired,
    contentHeight: PropTypes.number.isRequired,
    keyboardHeight: PropTypes.number.isRequired,
    keyboardVisible: PropTypes.bool.isRequired,
    keyboardWillShow: PropTypes.bool.isRequired,
    keyboardWillHide: PropTypes.bool.isRequired,
    keyboardAnimationDuration: PropTypes.number.isRequired,

    // Managing the IME type
    inputMethod: PropTypes.oneOf(Object.values(INPUT_METHOD))
      .isRequired,
    onChangeInputMethod: PropTypes.func,

    // Rendering content
    children: PropTypes.node,
    renderInputMethodEditor: PropTypes.func.isRequired,
  };

  static defaultProps = {
    children: null,
```

```
    onChangeInputMethod: () => {},
  };

  // ...

}
```

Now let's use `componentDidUpdate` to handle switching the `inputMethod`. When the keyboard transitions from hidden to visible, we want to set the `inputMethod` to `INPUT_METHOD.KEYBOARD`. When the keyboard transitions from visible to hidden, we want to set the `inputMethod` to `INPUT_METHOD.NONE`... unless we're currently displaying the image picker (the keyboard should always be hidden when we display the image picker, so we can ignore this transition).

messaging/components/MessagingContainer.js

```
// ...

componentDidUpdate(prevProps) {
  const { onChangeInputMethod } = this.props;

  if (this.props.keyboardVisible && !prevProps.keyboardVisible) {
    // Keyboard shown
    onChangeInputMethod(INPUT_METHOD.KEYBOARD);
  } else if (
    // Keyboard hidden
    !this.props.keyboardVisible &&
    prevProps.keyboardVisible &&
    this.props.inputMethod !== INPUT_METHOD.CUSTOM
  ) {
    onChangeInputMethod(INPUT_METHOD.NONE);
  }

  // ... more to come!

}
```

```
// ...
```

Since `inputMethod` will be stored in the `state` of the parent, we'll call `onChangeInputMethod` and let the parent pass this prop back down. We *could* store `inputMethod` in the `state` of `MessagingContainer`, but since the parent needs to access this value, it's best that the parent stores it.

`LayoutAnimation`

We're going to use `LayoutAnimation` to handle automatically transitioning between the various states of this component. `LayoutAnimation` is still considered experimental, and it's more common to use the other animated API, `Animated`. However, `LayoutAnimation` is the only way we can match the exact animation of the keyboard. It's used internally by the built-in `KeyboardAvoidingView` component, so it's safe for us to use despite being considered experimental.

Currently `LayoutAnimation` is disabled by default on Android, so we need to enable it by calling `UIManager.setLayoutAnimationEnabledExperimental(true)`. We can enable it anywhere in the app, but let's do it at the top of `MessagingContainer.js`, since that's the file we use it in:

messaging/components/MessagingContainer.js

```
if (
  Platform.OS === 'android' &&
  UIManager.setLayoutAnimationEnabledExperimental
) {
  UIManager.setLayoutAnimationEnabledExperimental(true);
}
```

> The `UIManager` object contains a variety of APIs for getting access to native UI elements for measuring, but we won't use it for anything else here.

`LayoutAnimation` automatically handles animating elements that should change size or appear/disappear between calls to `render`. We call `LayoutAnimation.create` to

define an animation configuration, and then `LayoutAnimation.configureNext` to enqueue the animation to run the next time `render` is called.

The `LayoutAnimation.create` API takes three parameters:

- `duration` - The duration of the animation
- `easing` - The curve of the animation. We choose from a predefined set of curves: `spring`, `linear`, `easeInEaseOut`, `easeIn`, `easeOut`, `keyboard`. The `keyboard` curve is the key to matching the keyboard's animation curve – although it only exists on iOS.
- `creationProp` - The style to animate when a new element is added: `opacity` or `scaleXY`.

In our case, we want to call this every time the component re-renders, so `componentDidUpdate` is the best place.

messaging/components/MessagingContainer.js

```
// ...

componentDidUpdate(prevProps) {

  // ... from before!

  const { keyboardAnimationDuration } = this.props;

  const animation = LayoutAnimation.create(
    keyboardAnimationDuration,
    Platform.OS === 'android'
      ? LayoutAnimation.Types.easeInEaseOut
      : LayoutAnimation.Types.keyboard,
    LayoutAnimation.Properties.opacity,
  );
  LayoutAnimation.configureNext(animation);
}

// ...
```

`LayoutAnimation` applies to the entire component hierarchy, not just the component we call it from, so this will actually animate every component in our app. It may be a better idea to selectively choose when to animate based on the exact props which have changed, but for simplicity, let's assume we always want to animate.

Handling the back button

We should add one last bit of logic in `MessagingContainer.js` to handle the hardware back button on Android. When the `CUSTOM` IME is active, we want the back button to dismiss the IME, just like it would for the device keyboard. We'll use `BackHandler` for this. When the back button is pressed, if the `CUSTOM` IME is active, we'll call `onChangeInputMethod(INPUT_METHOD.NONE)` to notify the parent.

messaging/components/MessagingContainer.js

```
// ...

componentDidMount() {
  this.subscription = BackHandler.addEventListener(
    'hardwareBackPress',
    () => {
      const { onChangeInputMethod, inputMethod } = this.props;

      if (inputMethod === INPUT_METHOD.CUSTOM) {
        onChangeInputMethod(INPUT_METHOD.NONE);
        return true;
      }

      return false;
    },
  );
}

componentWillUnmount() {
  this.subscription.remove();
}
```

```
// ...
```

Rendering the `MessagingContainer`

Now let's render this thing! We'll render an outer `View` which contains the message list and the toolbar (via `children`), and an inner `View` which renders the image picker (via `renderInputMethodEditor`).

The conditional logic is pretty complex, so let's take a look at it in-line with the code.

messaging/components/MessagingContainer.js

```
// ...

render() {
  const {
    children,
    renderInputMethodEditor,
    inputMethod,
    containerHeight,
    contentHeight,
    keyboardHeight,
    keyboardWillShow,
    keyboardWillHide,
  } = this.props;

  // For our outer `View`, we want to choose between rendering at
  // full height (`containerHeight`) or only the height above the
  // keyboard (`contentHeight`). If the keyboard is currently
  // appearing (`keyboardWillShow` is `true`) or if it's fully
  // visible (`inputMethod === INPUT_METHOD.KEYBOARD`), we should
  // use `contentHeight`.
  const useContentHeight =
    keyboardWillShow || inputMethod === INPUT_METHOD.KEYBOARD;
```

```
  const containerStyle = {
    height: useContentHeight ? contentHeight : containerHeight,
  };

  // We want to render our custom input when the user has pressed
  // the camera button (`inputMethod === INPUT_METHOD.CUSTOM`), so
  // long as the keyboard isn't currently appearing (which would
  // mean the input field has received focus, but we haven't updated
  // the `inputMethod` yet).
  const showCustomInput =
    inputMethod === INPUT_METHOD.CUSTOM && !keyboardWillShow;

  // If `keyboardHeight` is `0`, this means a hardware keyboard is
  // connected to the device. We still want to show our custom image
  // picker when a hardware keyboard is connected, so let's set
  // `keyboardHeight` to `250` in this case.
  const inputStyle = {
    height: showCustomInput ? keyboardHeight || 250 : 0,
  };

  return (
    <View style={containerStyle}>
      {children}
      <View style={inputStyle}>{renderInputMethodEditor()}</View>
    </View>
  );
}

// ...
```

In order for the toolbar to sit above the home indicator on the iPhone X, we'll need to adjust the space below the toolbar as the keyboard transitions up and down.

Supporting the iPhone X

You may skip this section if you're not testing with an iPhone X.

In the "Core Components" chapter, we used the `SafeAreaView` to support the iPhone X. This won't work here, since we want to *animate* the space below the toolbar (to avoid a jerk when the space changes).

We'll install the npm library `react-native-iphone-x-helper`[60] to help us determine if the device is an iPhone X.

In your terminal, install the library with:

```
expo install react-native-iphone-x-helper@1.2.1
```

React Native doesn't currently provide a way to determine if the device is an iPhone X. This library simply checks the device's dimensions. Hopefully in the future something better will be provided out-of-the-box for accessing the safe area insets directly.

After this finishes, import the `isIphoneX` utility function at the top of the file:

messaging/components/MessagingContainer.js

```
import { isIphoneX } from 'react-native-iphone-x-helper';
```

Now we can update the `render` method above to include extra space below the toolbar:

[60] https://www.npmjs.com/package/react-native-iphone-x-helper

messaging/components/MessagingContainer.js

```
// ...

render() {
  // ...

  // The keyboard is hidden and not transitioning up
  const keyboardIsHidden =
    inputMethod === INPUT_METHOD.NONE && !keyboardWillShow;

  // The keyboard is visible and transitioning down
  const keyboardIsHiding =
    inputMethod === INPUT_METHOD.KEYBOARD && keyboardWillHide;

  const inputStyle = {
    height: showCustomInput ? keyboardHeight || 250 : 0,

    // Show extra space if the device is an iPhone X the keyboard is
    // not visible
    marginTop:
      isIphoneX() && (keyboardIsHidden || keyboardIsHiding)
        ? 24
        : 0,
  };

  // ...
}

// ...
```

Whew, we made it. Save `MessagingContainer.js`. Now we just need to render `MessagingContainer` from `App`.

Rendering `MessagingContainer` in `App`

Head back to `App.js` and import the components we've just created:

messaging/App.js

```
// ...

import KeyboardState from './components/KeyboardState';
import MeasureLayout from './components/MeasureLayout';
import MessagingContainer, {
  INPUT_METHOD,
} from './components/MessagingContainer';

// ...
```

Let's include the `inputMethod` in the `state` of `App`, and handle changes to it.

messaging/App.js

```
// ...

export default class App extends React.Component {
  state = {
    // ...
    inputMethod: INPUT_METHOD.NONE,
  };

  // ...

  handleChangeInputMethod = (inputMethod) => {
    this.setState({ inputMethod });
  };

  handlePressToolbarCamera = () => {
    this.setState({
      isInputFocused: false,
      inputMethod: INPUT_METHOD.CUSTOM,
    });
  };
```

```
  // ...

}

// ...
```

Lastly, let's use `MeasureLayout`, `KeyboardState`, and `MessagingContainer` to render the UI components we've already written.

We'll rearrange:

- `this.renderMessageList`,
- `this.renderToolbar`, and
- `this.renderInputMethodEditor`

so that they render within `MessagingContainer`.

We can update the render method of `App` to look like this:

```
// ...

render() {
  const { inputMethod } = this.state;

  return (
    <View style={styles.container}>
      <Status />
      <MeasureLayout>
        {layout => (
          <KeyboardState layout={layout}>
            {keyboardInfo => (
              <MessagingContainer
                {...keyboardInfo}
                inputMethod={inputMethod}
                onChangeInputMethod={this.handleChangeInputMethod}
                renderInputMethodEditor={
```

```
                this.renderInputMethodEditor
              }
            >
              {this.renderMessageList()}
              {this.renderToolbar()}
            </MessagingContainer>
          )}
        </KeyboardState>
      )}
    </MeasureLayout>
    {this.renderFullscreenImage()}
  </View>
);
}

// ...
```

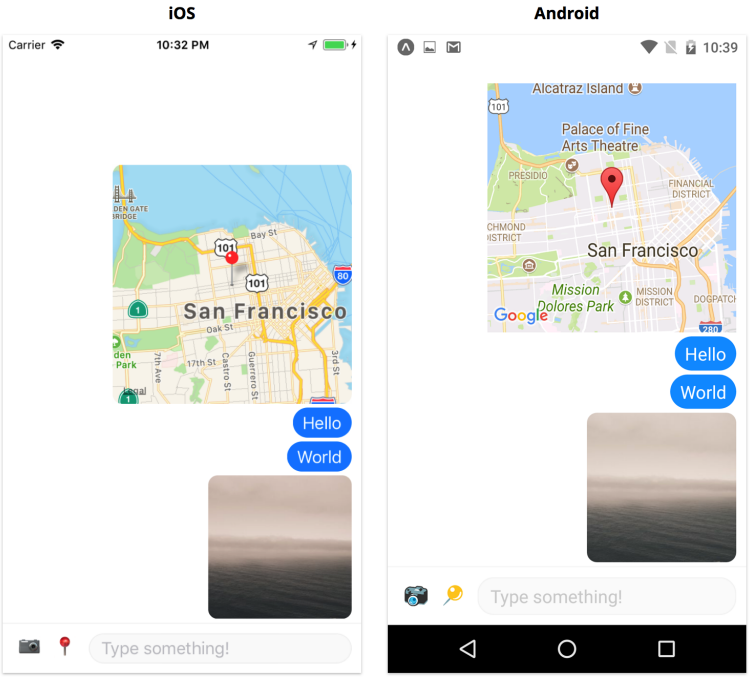
Using the `children` function prop pattern, we can pretty clearly visualize the flow of data downward into `MessagingContainer`.

Note that since `keyboardInfo` contains many properties, it's easiest to pass them all into `MessagingContainer` at once with the object spread syntax `...keyboardInfo`. If we prefer, we could also assign each property individually, e.g.

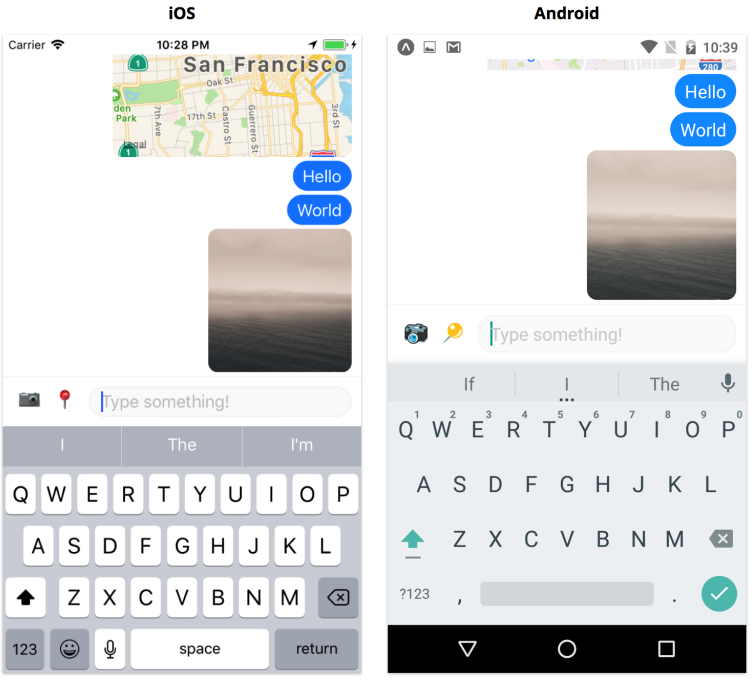
`keyboardHeight={keyboardInfo.keyboardHeight}`.

Save `App.js` and test it out! You should see the same components as before, but now they animate smoothly to avoid the keyboard as it appears and disappears.

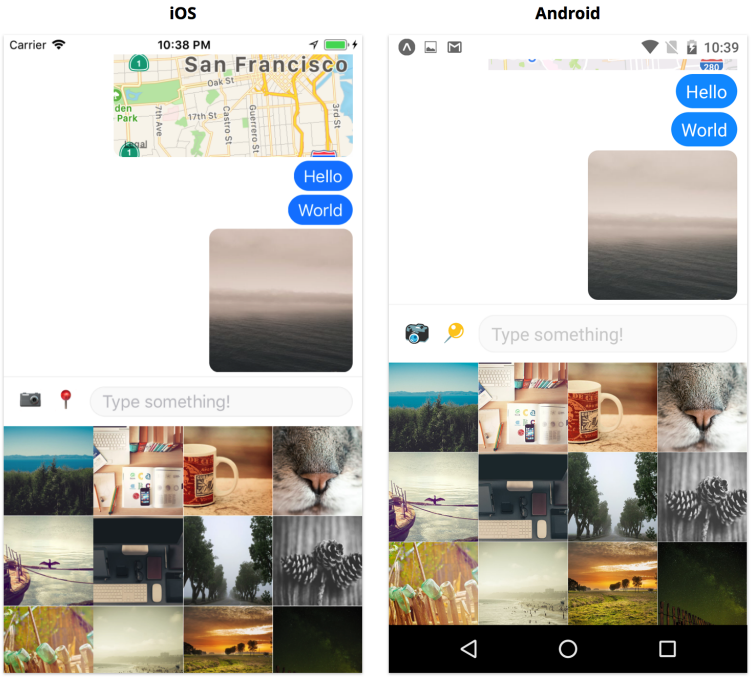
In the default state, our app should look like this:



Tapping the input field should pop open the keyboard, and smoothly transition the rest of the UI:



Tapping the camera icon should transition to the image picker:



We're Done!

We've built a messaging app UI complete with text messages, images, and maps. We notify the user of connectivity issues. We display a pixel-perfect infinite scrolling grid of photos. We smoothly animate the UI as new messages are added and removed (try it if you haven't! the `LayoutAnimation` takes care of this automatically). We handle the keyboard gracefully on both platforms.