React Native ScrollView and FlatList

SD550 – Web Application Development 2

Maharishi International University

Department of Computer Science

M.S. Thao Huy Vu

Maharishi International University - Fairfield, Iowa

Main Points

- TextInput
- Image
- Platform
- ScrollView
- FlatList

User Input

Controlled vs uncontrolled components

- Where is the source of truth for the value of an input?

React Native recommends always using controlled components.

`<TextInput />`

This component is like an **`<input>`** HTML element, but it's implicitly a text input type, cannot be radio or checkbox.

Pass value and **`onChangeText`** props.

You can set which keyboard pulls up on the phone with **`keyboardType`** (such as **`keyboardType='email-address'`**), toggle the **`auto-correct`** feature, set the **`textContentType`** for the phone to autofill the text (like your password on your keychain).

https://reactnative.dev/docs/textinput

Handling multiple inputs

`<form>` exists in HTML, but not in React Native.

With controlled components, we maintain the state as an object for all input values.

We can define a function that handles the data to submit.

Validating Input

Since we will use controlled input elements, we can conditionally set the state based on the validation results of the input value:

- Validate form before submitting
- Validate form after changing single input value

<KeyboardAvoidingView>

Native component to handle avoiding the virtual keyboard.

Good for simple/short forms.

Use Behavior props to set how your view will behave when the keyboard slides up.

The view moves independent of any of its child TextInputs.

```
<KeyboardAvoidingView style={styles.container}
behavior={Platform.OS === 'ios'? "padding": "height"}
keyboardVerticalOffset={100}>
....
    </KeyboardAvoidingView>
```

https://reactnative.dev/docs/keyboardavoidingview

<Image />

The **<Image />** component is pretty similar to a HTML **img** tag. However, there are a few differences, such as changing the **src** prop to **source**.

For local images, you import them with **require:**

```
<Image source={require('path/to/local/image)} />
```

For non-local images, you can pass an object with the **uri** property:

```
<Image
   source={{uri: 'https://imagesite.com/path/to/image'}}
   style={{height: 100, width: 100, resizeMode: contain}} />
```

The **<Image />** component does not have an **onPress** prop.

require() vs {uri}

- Mobile Network Speed (2G, 3G, 4G, LTE..)
- Fetch Latency
- Offline mode
- Bundle size
- Re-use the image

Image size based on Pixel Ratio

You can also use the @2x and @3x suffixes to provide images for different screen densities.

Save multiple versions of your images as following:

- **check.png**
- **check@2x.png**
- **check@3x.png**

Use it normally as:

```
<Image source={require('./check.png')} />
```

<ImageBackground>

The main distinction between **ImageBackground** and **Image**, is that it can have child elements. While the **<Image />** component is self-closing.

Platform Specific Code

React Native provides two ways to organize your code and separate it by platform:

- Using the **Platform** module.
- Using platform-specific file **extensions**.

Example - Styles

```
import {Platform, StyleSheet} from 'react-native';

const styles = StyleSheet.create({
  height: Platform.OS === 'ios' ? 200 : 100,
});
```

Example – Different Components

When your platform-specific code is more complex, you should consider splitting the code out into separate files.

```
const Component = Platform.select({
    ios: () => require('ComponentIOS'),
    android: () => require('ComponentAndroid'),
})();

<Component />;
```

Keys can be one of 'ios' | 'android' | 'native' | 'default', returns the most fitting value for the platform you are currently running on, if you're running on a phone, ios and android keys will take preference. If those are not specified, native key will be used and then the default key.

Example - Styles

```
import {Platform, StyleSheet} from 'react-native';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    ...Platform.select({
      ios: {
        backgroundColor: 'red',
      },
      android: {
        backgroundColor: 'green',
      },
   }),
  },
});
```

Platform-specific extensions

React Native will detect when a file has a **`.ios.`** or **`.android.`** extension and load the relevant platform file when required from other components.

`BigButton.ios.js`
`BigButton.android.js`

You can then require the component as follows:

```
import BigButton from './BigButton';
```

React Native will automatically pick up the right file based on the running platform.

Scrolling?

In web, browsers will automatically become scrollable for

content with heights taller than the window. In mobile, we need to do that manually.

The **`ScrollView`** component is the basic component to enable scrolling.

ScrollView

The most basic scrolling view:

- Will render ALL of its children ahead of time.
- To render an array of data, use .map()

Components in an array need a unique **key** prop.

https://reactnative.dev/docs/scrollview.html

ScrollView Example

```
export default function App() {

   const items = Array.from({length: 50}, (n,i)=>({key:i, text: `Item ${i}`}));

   return (
        <ScrollView>
             {items.map(item => <Text key={item.key}>{item.text}</Text>)}
        </ScrollView>
   );
}
```

SafeAreaView

The basic setup of **`ScrollView`** is giving boundaries to the scrollable-view, which would most likely be the screen height and width. Like an image, you can wrap **`ScrollView`** to give it these boundaries.

Normally, you can use a normal View component for this, but if you have an iPhone X, you may find that your View goes up behind the rounded corners or sensor cluster. The **`SafeAreaView`** will take care of this, giving adequate padding so the entire screen will be visible.

```
<SafeAreaView style={{ flex: 1 }}>
     <ScrollView>// content in here to fill the page</ScrollView>
</SafeAreaView>
```

FlatList

A very performant scrolling view for rendering data.

It is Virtualized or Lazy, only renders what's needed at a time.

Only the visible rows are rendered in first cycle. Rows are recycled, and rows that leave visibility may be unmounted.

Pass an array of data and a **`renderItem`** function as props.

Only updates if props are changed.

https://reactnative.dev/docs/flatlist

FlatList Example

```
<SafeAreaView style={{ flex: 1 }}>
  <FlatList
    data={arr}
    renderItem={({ item }) => <IndividualComponent data={item} />}
    keyExtractor={item => item.id}
  />
</SafeAreaView>
```

The important thing to keep in mind with **`FlatList`** is that the **`renderItem`** prop is fixed to take these parameters in its function: **`renderItem={({item, index, separators}) => {}}`**. Also, the item is always from the array you pass into the **`data`** prop - which can only take in a plain array.

FlatList Props

`keyExtractor` takes care of React's need to set a unique key on each element, just like setting a key prop on each of the components rendered by **`FlatList`**.

The **`FlatList`** wouldn’t re-render if data is mutated, this is why immutability is important.

Also, if the **`data`** needs to update because of another state or prop change, set the **`extraData`** prop to watch what would update
`extraData={this.state}`

ActivityIndicator

- Show the waiting status when the screen is loading

```
<ActivityIndicator size="large" color="#00ff00" />
```

AsyncStorage

- Similar to LocalStorage in ReactJS
- Install: npm install @react-native-async-storage/async-storage

Async storage – setItem

```
try{
    await AsyncStorage.setItem('my-key-here', 'my-text-here')
}catch(error){
}
```

AsyncStorage - getItem

```
try{
    const data = await AsyncStorage.setItem('my-key-here');
    if(data){
        //do something with data here
    }
}catch(error){
}
```

Async storage – removeItem

```
try{
    await AsyncStorage.removeItem('my-key-here')
}catch(error){
}
```