Cross-platform mobile app development

INTRODUCTION TO REACT NATIVE

Table of contents

1.1 Javascript

- Javascript: scripting language, widely used in Web application development.
- Javascript Framework is a set of libraries built on top of the Javascript programming language.
- Javascript executes on client side (on browser) and also on server side (e.g. NodeJS)
  - Client-side: handling HTML objects in the browser, controlling input data, handling events, creating effects, etc.
  - Server-side (backend): handle the business logic of the application

```
< script type =
"text/javascript" >
 //JavaScript goes here </
script >
```

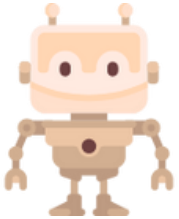
1.1 Javascript (2)

- Javascript engines
  - JavaScript code runs in a JavaScript “engine”.
- Babel
  - Babel is a configurable compiler for permission to use JavaScript language features new (and extensions, like JSX), compile "down" to older JavaScript versions supported on a wide variety of tools
  - When initializing a React Native app, the config file **babel.config.js** figure generated in the project

1.1 Javascript (3)

Write JS code the traditional way

```
< script type = "text/javascript"
>
        //JavaScript goes here
</ scripts >
```



Babel

Preprocessor

```
[ 1 , 2 , 3 ].map(n => n
+ 1 );
```

```
[ 1 , 2 , 3 ].map( function
(n) {
return n + 1 ;
});
```

"Modern" javascript

React Native uses Babel as preprocessor

ES6 + JSX

ECMAScript 6 + Extension

1.2 ES6 (ECMAScript6)

- Syntax highlights of ES6:
  - **Let** and **const .** keywords
  - Loop **for of**
  - Template literals
  - Default value for parameter
  - Arrow Function
  - Building classes
  - Modules
  - Rest Parameters (or Rest Operator)
  - Spread operator
  - Destructive assignment

1.2 ES6 (ECMAScript6) (2)

- Keyword**let Và const**
  - let to declare variables

```
for ( let i = 0 ; i < 5 ; i ++) { }
console . log ( i ); // undefined

console . log ( i ); // 0,1,2,3,4
```

  - const helps to mean constants

```
// Declare & Initialize Constant PI
const PI = 3.14 ;
console . log ( PI ); // 3.14

// reassigning value to Constant will

error
PI = 10 ; // error
```

1.2 ES6 (ECMAScript6) (3)

- Loop **for of**
  - Iterate over arrays or iterate over objects easily

```
// Loop through the array
let letters = [ "a" , "b" , "c" ];
 for ( let letter of letters ) {
console . log ( letter );
}
```

- Template literals
  - Create strings on multiple lines and perform string interpolation allowing variables or expressions to be embedded in a string
    - Template literals are created using the character pair ` `

```
let str = `String
above multiple lines!` ;
let a = 6 ; let b = 9 ;
let result = `The sum of ${ a } and ${ b } is: ${ a + b }
.` ; console . log ( result ); // Sum of 6 and 9 is: 15.
```

1.2 ES6 (ECMAScript6) (4)

- Arrow Function: The arrow syntax (=>) is one way function abbreviation.
  - **(param1, param2, …, paramN) => { statements }**
  - **(param1, param2, …, paramN) => expression**

    // equivalent to: **=> { return expression; }**
  - // Parentheses are optional when there is only one parameter name:
  - **(singleParam) => { statements }**

    singleParam => { statements }

    written with a pair of parentheses.

    () => { statements }

    // The parameter list for a function with no parameters must be

1.2 ES6 (ECMAScript6) (5)

Arrow Function

```
// Function Expression
var sum = function ( a , b ) {
return a + b ;
}
console . log ( sum ( 2 , 3 )); //
5

// Arrow function
var sum = ( a , b ) => a + b ;
console . log ( sum ( 2 , 3 )); //
5
```

- Default value for parameter

```
function sayHello ( name = "A" ) {
var name = name ;
return `Hello ${ name } !` ;
}
console . log ( sayHello ()); // Hello A!
console . log ( sayHello ( 'B' )); // Hello B!
```

1.2 ES6 (ECMAScript6) (6)

- Building classes

```
// Create a class
class Rectangle {
// constructor (constructor)
constructor ( length , width ) {
this . length = length ;
this . width = width ;
}

// Method of class
getArea () {
        return this . length * this . width ;
}
}
```

- Modules
  - Each module is represented by a separate **.js file**
  - Export or import command in a module to export or import variables, functions, classes or any other entity to / from other modules or files

1.2 ES6 (ECMAScript6) (7)

- Rest Parameters (or Rest Operator)
  - Pass an arbitrary number of parameters to the function as an array Add in front of the operator parameter **...** (three dots)

```
function sortNumbers (... numbers ) {
return numbers . sort ();
}
console . log ( sortNumbers ( 3 , 5 , 7 ));
console . log ( sortNumbers ( 3 , 5 , 7 , 1 , 0 ));
```

- Spread operator
  - The Spread operator (i.e. splits) an array and passes the value into the specified function

```
let colors = [ "Green" , "Red" ];
```

- Destructive assignment
  - Expressions make it easy to extract values from arrays or properties from objects, into separate variables by providing a concise syntax.

```
let [ a , b ] = colors ;
```

1.3 JSX

- **JSX = Javascript + XML** : An extension for JavaScript which is used to build UI interfaces.
- The syntax of JSX is similar to XML, including the pair: opening tag and closing tag

```
<JSXElementName    JSX_Attributes>

</JSXElementName>
```

- JSX converts from XML-like syntax to native JavaScript
  - XML elements are converted to function calls
  - XML attributes converted to objects
- Any JavaScript expression can be embedded in JSX by enclosing it in curly braces: {.....}

1.3 JSX (2)

For example:

Code JavaScript

```
React.createElement("div", {className: "red"}, "Children Text");
React.createElement(MyCounter, {count: 3 + 5});

React.createElement(DashboardUnit, {"data-index": "2"},
  React.createElement("h1", null, "Scores"),
  React.createElement(Scoreboard, {className: "results", scores: gameScores})
);
```

Corresponding JSX code

```
<div className="red">Children Text</div>;
<MyCounter count={3 + 5} />;

var gameScores = {
  player1: 2,
  player2: 5
};
<DashboardUnit data-index="2">
  <h1>Scores</h1>
  <Scoreboard className="results" scores={gameScores} />
</DashboardUnit>;
```

1.3 JSX (3)

Embedding Expressions in JSX

```
const name = 'A‘;
const element = <h1>Welcome to
 {name}</h1>;
ReactDOM.render( }
element,
document.getElementById( 'root' )
);
```

```
function sayHi(name) {
if (name) {
return <p>Hello, {name}
 !</p>}
else {
return <p>Hello there!</p>;}
}
```

Specifying attributes with JSX

```
const element = <img src={user.avatarUrl}></img>;
```

JSX Object: use **React.createElement()** to compile one JSX object to regular JSX

```
const element = React.createElement(
"p" ,
{ className: "welcome" },
"Welcome!"
);
const element = <p className= "welcome" >Welcome!</p>
```

Table of contents

2.1 Introduction to React

- **React** is "a JS library for building application interfaces"
  - Instead of using HTML templates, react builds components.
- React follows a component-driven development model
  - Idea: divide the user interface into a set of components Reusable
  - Combine and nest components to create complete user interfaces correction
- Benefits:
  - The source code of the components is easily reused
  - Better code readability - components are stored in classes their own Easy testing
  -

2.1 Introduction to React (2)

- Install React Development Environment: Settings

  node.js and other additional packages - via the Node package manager (Node Package Manager - npm).

  The basic concepts:

  - components props
  - state
  - Life cycle VirtualDOM
  - props
  - VirtualDOM
  -

2.1 Introduction to React

(3)

- **Components** are UI components that are broken down, Independent and reusable.
  - Components can be functions (stateless) or classes (stateful) in JS.
  - Component will have properties props (properties) and state (if defined by class).
  - Each component is executed and returned inside the render()
  - function.

    To distinguish between React components and HMTL tags, all React components must be written in **CamelCase style** (contiguous phrases that begin each word with a capital letter, no spaces or interspersed punctuation) and must **start with the letter**
    flower .

2.1 Introduction to React (4)

- **Props** are attributes passed into a component and is read-only.
  - Example 1: `<button className="active">Sign up</button>`
    - The attribute passed in that can be accessed is className and child via **props** .
    - props.className syntax will give the value " **active** " and props.child the value " **Sign up** ".
  - Example 2:

```
class Welcome extends React.Component {
render() {
                    return <h1> Hello {this .props.name } </h1> ;
}
}
const element = <Welcome name = "Paint"
/> ;
```

2.1 Introduction to React (5)

- **State** is the state that belongs to the component itself,

  managed by itself and cannot be accessed from the outside.

  - State is used to store information (data) composition, which may change over time.
  - State can only be used when using stateful components.

  Manipulation functions

- ➢ Visit: **this.state.variable**
- ➢ Change: **this.setState({variable: value});**

2.1 Introduction to React (6)

Life cycle is a life cycle of a React

- component from the time it was first rendered and every time re-render (mounting) and when removing components (unmounting).
- componentDidMount and componentWillUnmount respectively .

- These two methods can be overridden when using stateful components (classes).

2.1 Introduction to React (7)

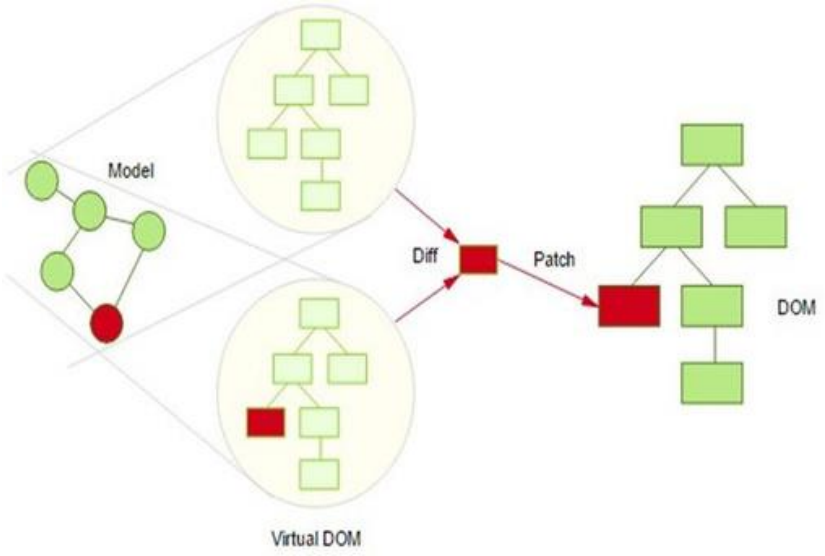
- **Virtual DOM**
  - DOM (Document Object Model) is an application programming interface that allows Javascript or another type of script to read and manipulate the contents of a document.
    - Single Page Applications (SPA): DOM is download the browser 1 time

      JS updates the DOM continuously as the user manipulates the application. Every time the DOM changes, the browser must render

    - **This causes delay**
    - React introduces VirtualDOM to speed up the application, reducing the time when the browser re-renders the page

2.1 Introduction to React (8)

- **Virtual DOM**
  - Mechanism of action:
    - When the new component is available
      added to the UI, a Virtual DOM
    - will be created
      When a component's state changes, React will update the Virtual DOM while keeping the previous version of the Virtual DOM for comparison, which helps find the changed Virtual
    - DOM object.
      When it finds a change, React just updates the object on the real DOM.

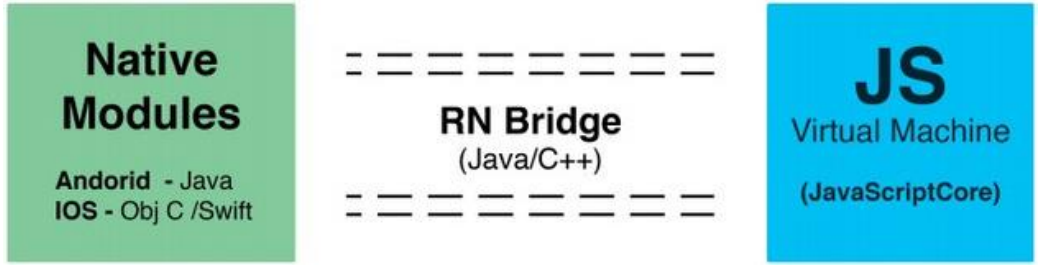


2.2 Introduction to ReactNative

- React Native is a JavaScript framework for writing applications with Translatable to native code on multiple platforms: iOS, Android First
- announced by Facebook in 2015
  - Open source project - https://github.com/facebook/react-
  - **Programming model similar to React** : Component-based, JSX, native stateful or stateless component, props, state,...
- React Native creates hybrid apps in which the user's JavaScript code is packaged and interpreted by a JS engine for native code.
  - Instead of using HTML elements, React Native provides some React components are available to developers, which helps to create native UI elements on the target platform.

2.2 Introduction to ReactNative (2)

- ReactNative Architecture



  - **Native Code/Modules** : written in Swift and Java for the native platform
  - **Javascript VM** : RN using JavaScriptCore (Safari)
    - In the case of Android, RN bundles JavaScriptCore with the app
      In the case of Chrome's debug mode, RN uses the V8 engine
    - and communicates with native code via WebSocket

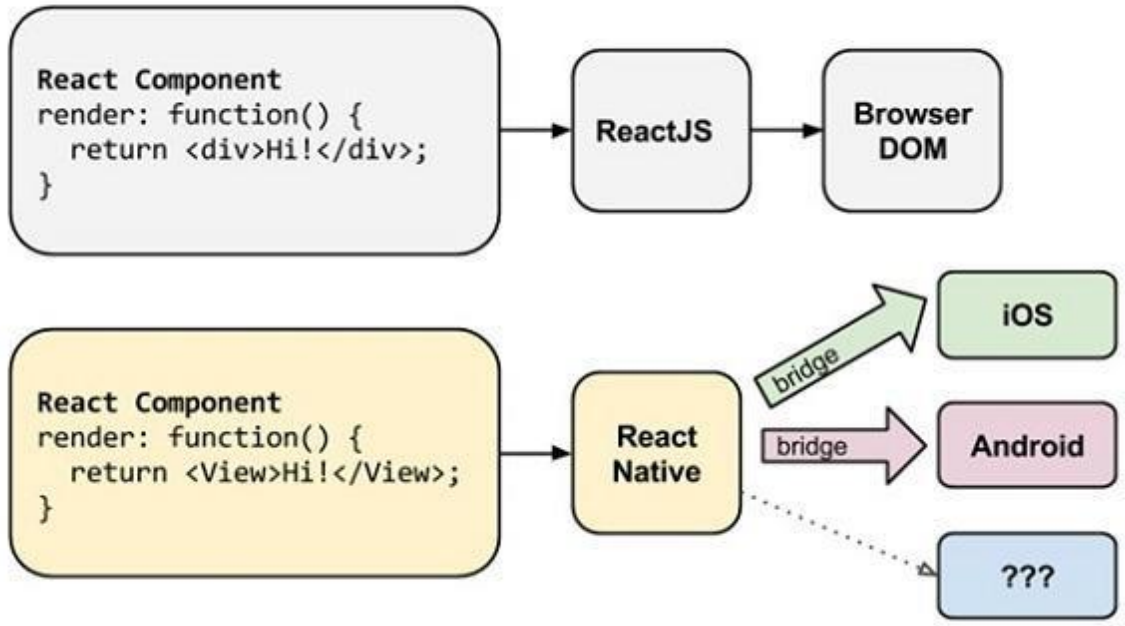
  - **React Native Bridge** : a responsible C++/Java bridge communication between native thread and Javascript
    - A custom protocol is used to transmit the message

2.2 Introduction to ReactNative (3)

- Basic principles

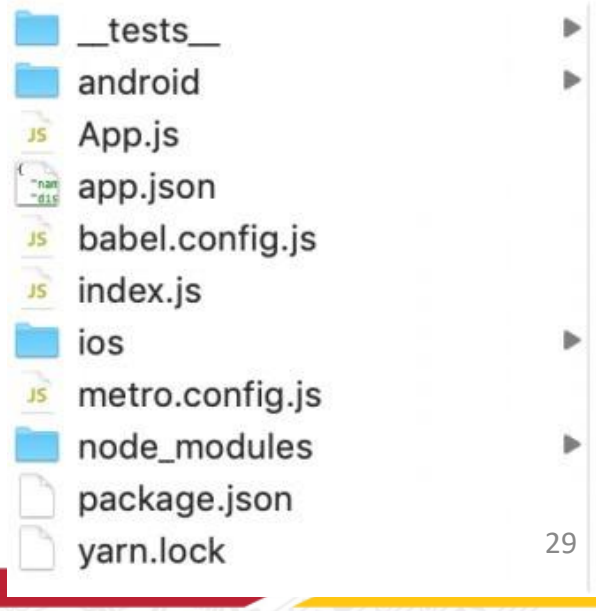
React components through the ReactJS library are translated **into** browser DOMs

React **Natives** components through the React Native framework are translated into platform-appropriate native code



2.3 First ReactNative Application

- ReactNative development tools
  - node.js and the Node package manager (Node Package Manager - npm)
    - Install the RN package: `npm install -g react-native-cli`
  - IDE: Visual Studio Code (or Atom,....), Xcode, Android Studio,...
- ReactNative project folder structure



2.3 First ReactNative Application (2)

```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

export default class HelloWorldApp extends Component {
        render() { render() render components
return (
<View>

                <Text>Hello world!</Text>

</View>
);
}
}
```

Must import everything that needs to be used; import is ES6 syntax

Component: usually things see on screen

keyword “class” and “extensions”

JSX: XML embedded in JavaScript

Here is the content in the file: **App.js**

{ Component } from 'react';
// here is the destructuring syntax from ES6 // equivalent to:
import React from “react”;
let Component = React.Component;

3.1 ReactNative Components

- ReactNative Component has the same characteristics as component of React. The difference is that they bind to native UI elements.
  - Component <View> is a common container of interface elements other users
  - s
  - <Text> is a piece of text
  - <Image> is an image
  - Component <Button> is a button
  - s
    Component
    s
    Component
    s
    ...

Web (ReactJS)	React Native	Android	iOS
<input>	<TextInput>	EditText	UITextField
<p>	<Text>	TextView	UITextView
	<View>	Android.view	UIView

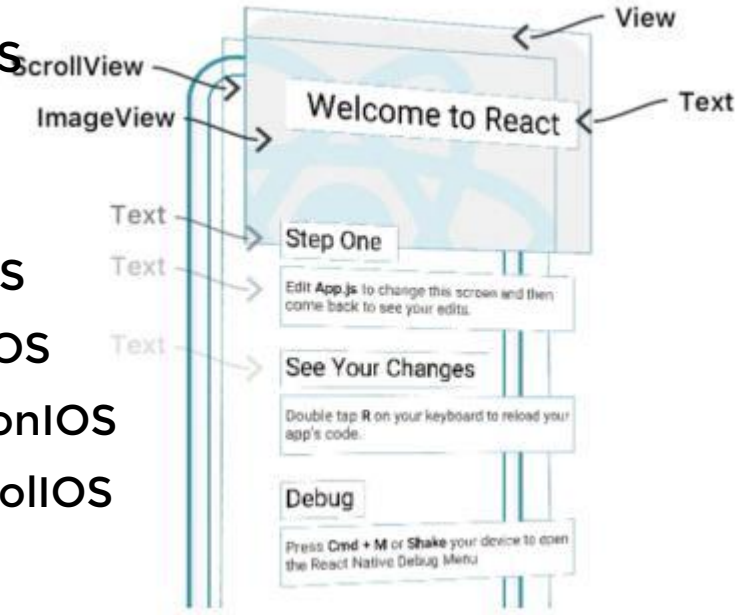
3.1 ReactNative Components (2)

- Basic React Native components:
  - https://facebook.github.io/react-native/docs/components-and-apis.html Basic Components
  - User Interface
  - List Views Most common UI
    - View
    - Text
    - Image
    - TextInput
    - ScrollView
    - StyleSheet
    - Button
    - Switch
    - Picker
    - Slider
    - Onscreen list views
      - FlatList
      - SectionList
  - ...
- Components shared by other developers:
  - http://www.awesome-react-native.com/#components

3.1 ReactNative Components (3)

- Android or iOS platform-specific components:
  - iOS-specific
  - Android-specific

- iOS components
  - ActionSheetIOS
  - AlertIOS
  - DatePickerIOS
  - ImagePickerIOS
  - ProgressViewIOS
  - PushNotificationIOS
  - SegmentControlIOS
  - TabBarIOS



- Android components
  - BackHandler
  - DatePickerAndroid
  - DrawerLayoutAndroid
  - PermissionsAndroid
  - ProgressBarAndroid
  - TimePickerAndroid
  - ToastAndroid
  - ToolbarAndroid
  - ViewPagerAndroid

3.1 ReactNative Components (4)

- The general structure of a component:

•To build a city
the first part needs to be
imported Components
from a set of libraries
react
• Syntax :
import React,{Component}
from 'react'
-Then create a next class
redundant Component

class MyComponent extends Component {

}

```
import React from 'react';

class MyComponent extends React.Component {
   constructor() {
      super();
    // This constructor defines
    // the state of the component
   }

   render() {
      return(
         // This defines the reactive UI
         // associated with the component
      )
   }

   my_method() {
      //This is a custom method
   }
}
```

3.2 ReactNative Components Lifecycle

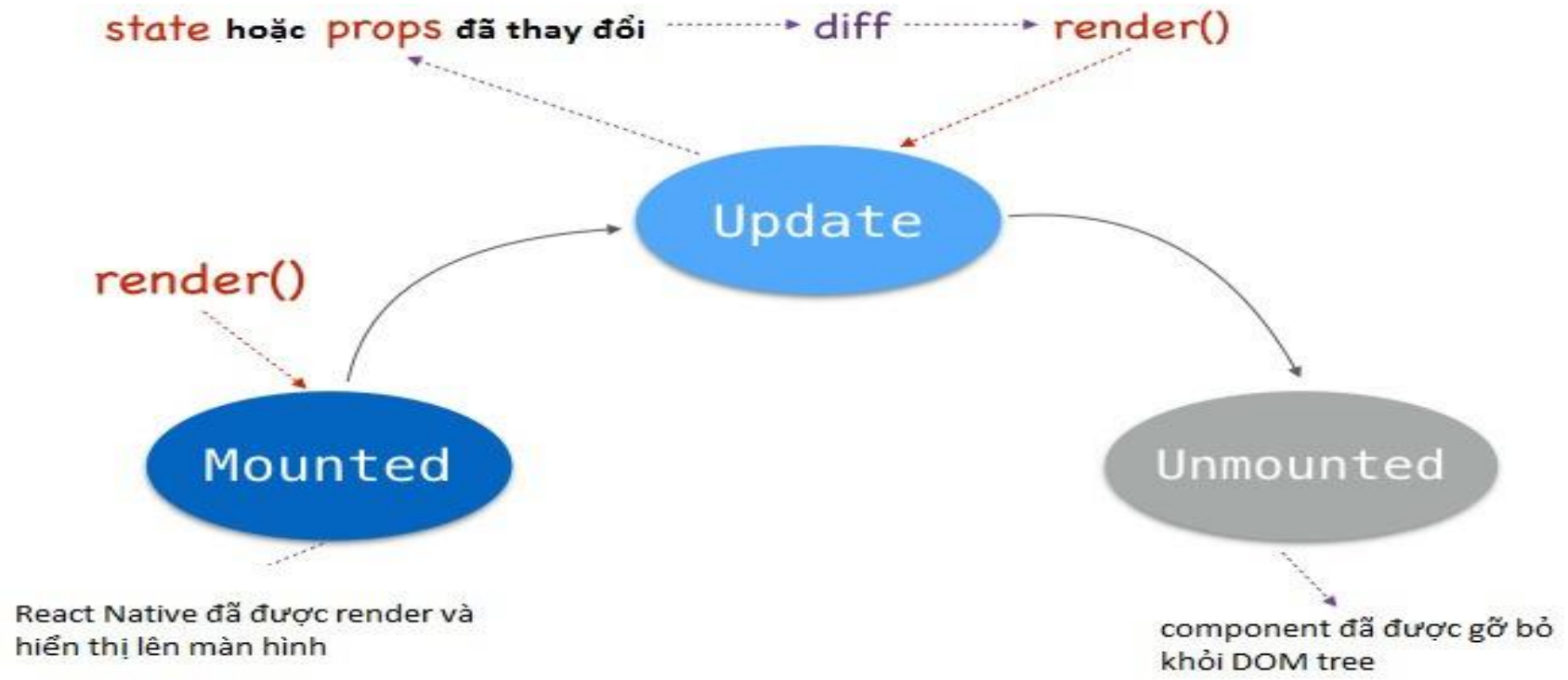
- Each ReactNative component will have its own lifecycle It. This lifecycle is executed corresponding to the process by which the user interacts with the application
- A Component life cycle consists of 3 stages main:

Mounted / Update / UnMounted

- **Mounted** : Contains methods that are called when the Component is instantiated and Inserted into the DOM.
- **Update** : Contains methods that are called when a Component is re-rendered.
- **UnMounted** : Contains methods that are called when a Component is removed from the DOM

3.2 ReactNative Components Lifecycle (2)

- Describe RN Component lifecycle stages



3.2 ReactNative Components Lifecycle (3)

- Mounted stage methods:

first. constructor(object Props) : the constructor of a Component normally this method always takes a props or other parameters.

2. componentWillMount() : this method happens once later constructor and in this method we will not reder the assignment represent Component

3. render() : works after componentWillMount , this method is used to render React Elements to the user interface. Everything in this method will be displayed on the screen.

4 componentDidMount() : this method is called only once After rendering occurs, the Elements are now displayed on screen and can be accessed by this.refs. Pine Usually we use this method to write code

- related to delay or sync API

3.2 ReactNative Components Lifecycle (4)

❖ UnMounted stage methods:

first. **componentWillReceiveProps(onject nextProps)** : The parent of this component will pass in a new props and this method will initialize the interface again. We can update the internal state via this.setState() method before the render method is called.

2. **shouldComponentUpdate(object nextProps, object nextState)** : This method has a return value of boolean type.

- A Component can be rendered or not rendered, this method ensures that the component will be re-rendered if true is returned. We often use this method to check if props or state has changed, if there is a change, we return true to continue rendering() and return false to not run.
  render() method.

3 **render()** : this method is only called again when shouldComponentUpdate returns true.

. **componentDidUpdate(object prevProps,object prevState)** : Method This method executes only when render() occurs. Now the interface is ready

4 update

3.2 ReactNative Components Lifecycle (5)

- Update phase methods:
  - componentWillUnmount() : This method only occurs when the application is completely closed.

3.3 Status management

- State in React Native apps can be definition is a set of values that a component uses and manages.
- Component state is a JavaScript object declared when a component is created.
- The RN component is the same as the React component:
  - props = properties: parameters can be passed during component initialization, then props cannot be changed
  - state : initial value of state initialized in constructor, then state can be changed via setState() function

3.3 Status Management (2)

```
import React, { Component } from 'react';
import { AppRegistry , Text, View } from 'react-native';

class Greeting extends Component {
•
render() {
return (
<Text>Hello { this.props.name }!</Text>
•
);
}
}

export default class LotsOfGreetings extends Component {
render() {
return (
<View style={{ alignItems : 'center'}}> •
<Greeting name=' Rexxar ' /> •
<Greeting name='Jaina' />
<Greeting name=' Valeera ' /> •
</View>
);
}
}
```

Greeting component when created will be assigned values from the side in addition to **this.props.name**
Then this value cannot be changed

Greeting components Value passed in the "name" instance
The result is 3 greetings displayed on the screen

```
import React , {C omponent } from "react" ;
import { Button , Text , View , SafeAreaView } from "react-native" ;
class Counter extends Component {
state = { counter: this . props . value };
render ( props ) {
return (
< View >
< Text > {this . state . counter } </ Text >
< Button

this . state . counter + 1 });} }
title = { "Click here" }
/>
</ View >
);
}
}
class MyApp extends Component {
render () {
return (
<SafeAreaView> _ _
< Counter value = { 10 } />
</ SafeAreaView >
);
}
}
export default MyApp ;
```

```
onPress = { () => { this . setState ({ counter:
```

3.3 Status Management (4)

- Hooks: is a new addition in React 16.8, ReactNative supports Hooks since release 0.59
  - Functions that allow "import" of React state and lifecycle features from components written in the form functions. Hooks don't work inside classes.
- React provides several built-in Hooks
  - useEffect adds the ability to execute side effects from a functional component
    - useContext allows to register React context
    - useReducer allows to manage the local state of complex components using reducer
- Programmers can create their own Hooks

3.3 Status Management (5)

- Hooks are JavaScript functions, but they apply set two additional rules
- Invoke Hook only at top level
  - Don't call Hooks inside loops, thing events or nested functions
- Invoke Hooks only from written components React's functional form
  - Don't call Hooks from regular JavaScript functions

```
import React , { useState } from "react" ;
import { Button , Text , View , SafeAreaView } from "react-native" ;
const Counter = props => {
const [ counter , setCounter ] = useState ( props . value );
return (
< View >
< Text > { counter } </ Text >
< Button
                                        onPress = { () => { setCounter ( counter + 1 );} }
title = { "Click here" }
/>
</ View >
);
}
const MyApp = () => {
return (
<SafeAreaView> _ _
< Counter value = { 10 } />
</ SafeAreaView >
);
}
export default MyApp ;
```

3.3 Status Management (6)

- Compare props and state

State	Props
Internal data	external data
Mutable	Immutable
Created in the component	Inherited from a parent
Can only be updated in the component	Can be updated by parent component
Can be passed down as props	Can be passed down as props
Can set default values inside Component	Can set default values inside Component

4.1 Common UI Elements

- <View> - Similar to HTML , no default look defined but can be styled. Also commonly used to layout child components.
- <Text> - Display text to the screen. They can be nested to each other - styles that apply to ancestors will also apply to offspring:

```
< Text style = { { fontWeight: 'bold' } } >
This is some bold text.
                < Text style = { { fontStyle: 'italic'
} } >
This is some bold and italic
text.
</ Text >
</ Text >
```

4.1 Common UI Elements (2)

- <Image> - displays an image on the screen.
  - Can render local and network images.
  - The width and height can be set via the style attribute. Thing This must be done when loading network images.
  - If the width and height provided do not match the dimension actual width and height, scaling or stretching will occur

    out depending on the resizeMode property of the image.
- Local image: load an image named 'Trex.png' located in same directory as the current JavaScript file:

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

- Remote Image: load image at specified URL:

```
< Image style = { { width: 150 , height: 200 } }
source = { { uri: 'http://via.placeholder.com/150x200' } } />
```

4.1 Common UI Elements (3)

- <Image> - displays an image on the screen.
  - In React Native it is possible to specify separate images for each Platform by specifying the extension image.ios.png and image.android.png
  - Also can provide images for each device size different.
  - Syntax: @2x or @3x

```
.
├── button.js
└── img
    ├── check@2x.png
    └── check@3x.png
```

4.1 Common UI Elements (4)

- <TextInput> - allows the user to enter text.
  - The onChangeText event handler property allows the developer to respond to any changes the user makes.

```
< View style = { { padding: 10 , flex: 1 , justifyContent: 'center' } } >
< TextInput
style = { { height: 40 } }
placeholder = "Type here to translate!"
onChangeText = { ( text ) => this . setState ({ text }) }
/>
< Text style = { { padding: 10 , fontSize: 42 } } >
 {this . state . text . split ( ' ' ). map (( word ) => word && '✍' ). join
                                                              ( ' ' ) }

</ View >
>
```

4.1 Common UI Elements (5)

```
import React, { Component } from 'react';
import { AppRegistry, Text Input } from 'react-native';
export default class UselessTextInput extends Component {
constructor(props) {
super(props);
this.state = { text: 'Useless Placeholder' };
}
render() {
return (
< Text Input
                    style={{height: 40, borderColor: 'gray', borderWidth: 1}}
onChangeText ={(text) => this.setState({text})}
value={this.state.text}
/>
); This is the name of the state variable to receive
} text from Text Input box
}
```

Button Components

```
<Buttons
onPress={() => {
        Alert.alert('You tapped the button!');
}}
title="Press Me"
/>
```

- Render (render)
  - Blue Label on iOS
  - Blue rounded rectangle with white text on Android
- "onPress"
  - An event handler can be called function
  - Could be an anonymous function
  - It is possible to specify a property “color” to change the color of the button
- Build your own button using any "Touchable" component
  - The "Touchable" components provide the ability to record touch gestures, and display a response when a gesture is recognized.
  - No default style provided, must define own style

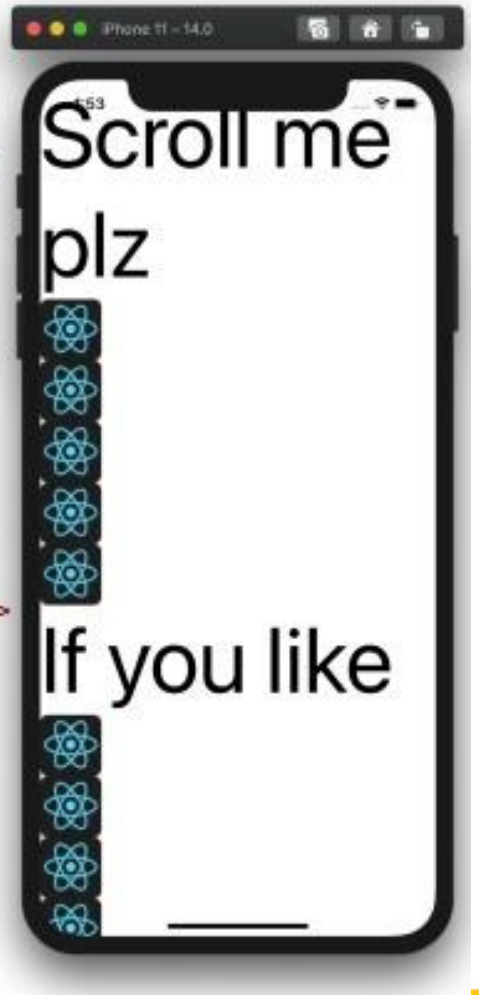
4.2 ScrollView

- ScrollView is a generic scrolling container that can store multiple components and views
  - Items don't have to be identical
  - Can scroll both vertically and horizontally
  - All ScrollView elements are visible
- ScrollViews can be configured to allow pagination through views using a swipe gesture (**prop pagingEnabled**)
  - On Android it is possible to do horizontal swiping between views with ViewPager
  - On iOS, ScrollView with a single item can be used to allow user to zoom content

4.2 ScrollView (2)

```
const logo = {
  uri: 'https://reactnative.dev/img/tiny_logo.png',
  width: 64,
  height: 64
};

export default App = () => (
  <ScrollView>
    <Text style={{ fontSize: 96 }}>Scroll me plz</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 96 }}>If you like</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Text style={{ fontSize: 96 }}>Scrolling down</Text>
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    <Image source={logo} />
    …
  </ScrollView>
);
```

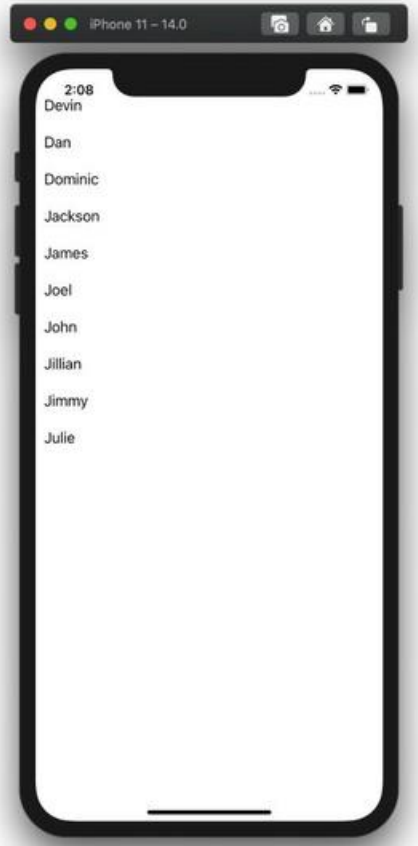
Example



4.2 ListView

- React Native provides components for displaying lists data, using **FlatList** or **SectionList**
- **FlatList** component renders a scrolling list
  - Works well for long lists of data where the number of items can change over time
  - Unlike ScrollView, FlatList renders only the currently active elements display on screen, not all elements at once
- The FlatList component requires two props:
  - **Data** : the source of information for the list
  - **renderItem** : a function that takes an item from the source and returns the component
  - display format
- **SectionList** displays a set of data divided into logical sections

```
import React from 'react' ;
import { FlatList , StyleSheet , Text , View } from 'react-native' ;

const FlatListBasics = () => { Example (II)
return (
< View style = { styles . container } >
< FlatList
data = { [
{ key: 'Devin' },
{ key: 'Dan' },
{ key: 'Dominic' },
{ key: 'Jackson' },
{ key: 'James' },
{ key: 'Joel' },
{ key: 'John' },
{ key: 'Jillian' },
{ key: 'Jimmy' },
{ key: 'Julie' },
] }
renderItem = { ({ item }) =>
< Text style = { styles . item } > { item . key } </ Text > }
/>
</ View >
);
}
export default FlatListBasics ;
```



5.1 Networking in ReactNative

- RN provides fetch function for related processes to networking
  - Similar to XMLHttpRequest or networking APIs other
- Network operations are asynchronous
  - **fetch()** method returns a **Promise** that makes it easy to write code that works asynchronously
  - **Promise** object that represents the final completion (or failure) of an asynchronous operation and its resulting value.

5.2 The fetch() function

- The fetch specification differs from jQuery.ajax() in two main ways:
  - Promise returned from fetch() will not reject with HTTP error status even if response is HTTP 404 or 500
    - Instead, it will resolve normally (with ok what's stopping the set to false) and it will reject only on network failure or if there are any request from completing
  - By default, fetch will not send or receive any cookies from the server
    - If the site relies on maintaining user sessions that may result in unauthenticated requests (to send cookies, the credential init option must be set)

5.2 The fetch() function (2)

- Syntax
  - Fetch takes as input a URL and retrieves data from the URL Optional second
  - parameter allows customizing the HTTP request: specifying additional headers or actual methods, for example
    make a GET/POST request

```
fetch ('https://mywebsite.com/endpoint/', {
method: 'POST',
headers: {
Accept: 'application/json',
                    'Content-Type': 'application/json',
},

body: JSON.stringify({
firstParam: 'yourValue',
secondParam: 'yourOtherValue', }),
});
```

Method of sending request: GET or POST

Using JSON to transfer information across the web
JSON is a simple javascript protocol to convert data structure to string

5.2 fetch() function (3)

- Response handling: response object
  - A fetch call will return a promise
  - When the promise completes, it returns a response object.
  - A response object with properties and methods
    - Important method: Body.json()
    - Get the Response stream and read it until it's done. The
    - result is the body text as JSON

```
const json = await response.json()

const text = await response.text()
```

5.3 Promise

- A promise is an object returned for attachment callbacks, instead of passing callbacks into a function

```
function getMoviesFromApiAsync() {
return fetch ('https://facebook.github.io/react-native/movies.json')
.then((response) => response.json())
.then((responseJson) => {
return responseJson.movies;
})
.catch((error) => {
console.error(error);
});
}
```

Promise : The Promise object represents final result (complete or fail) of an asynchronous operation and the value its results.

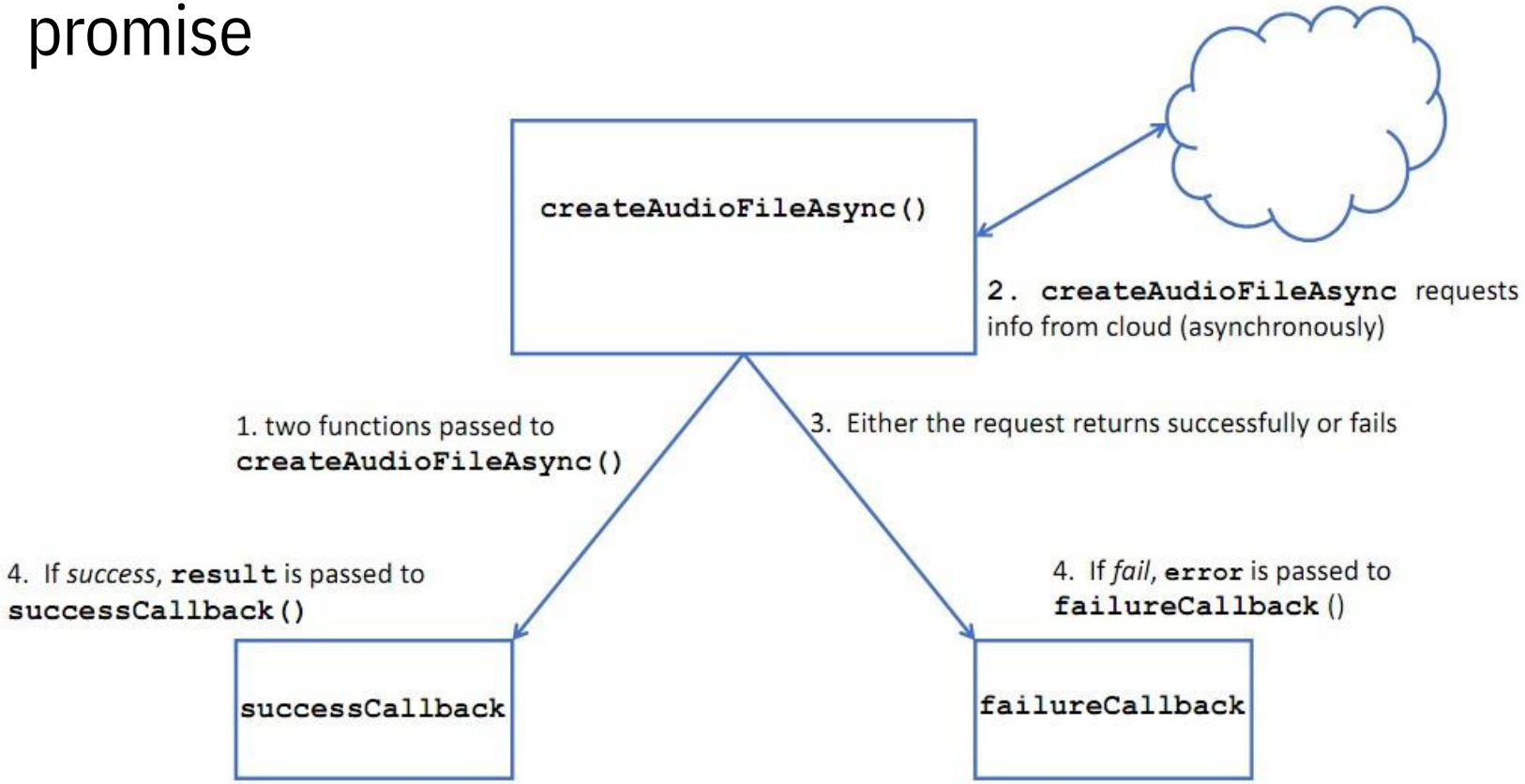
5.3 Promises (2)

- For example, we need to construct the function **createAudioFileAsync()**
  - This function will perform an asynchronous operation that creates
  - an audio file. The function takes as input 3 parameters: a configuration record and two callback functions (one is called if the audio file is created successfully and the other is called if it
  - happens). error)
    Normally createAudioFileAsync() function will be built like this:

```
function successCallback(result) {
console.log("Audio file ready at URL: " + result);
}

function failureCallback(error) {
                    console.log("Error generating audio file: " + error);
}

createAudioFileAsync(audioSettings, successCallback, failureCallback);
```

5.3 Promises (3)

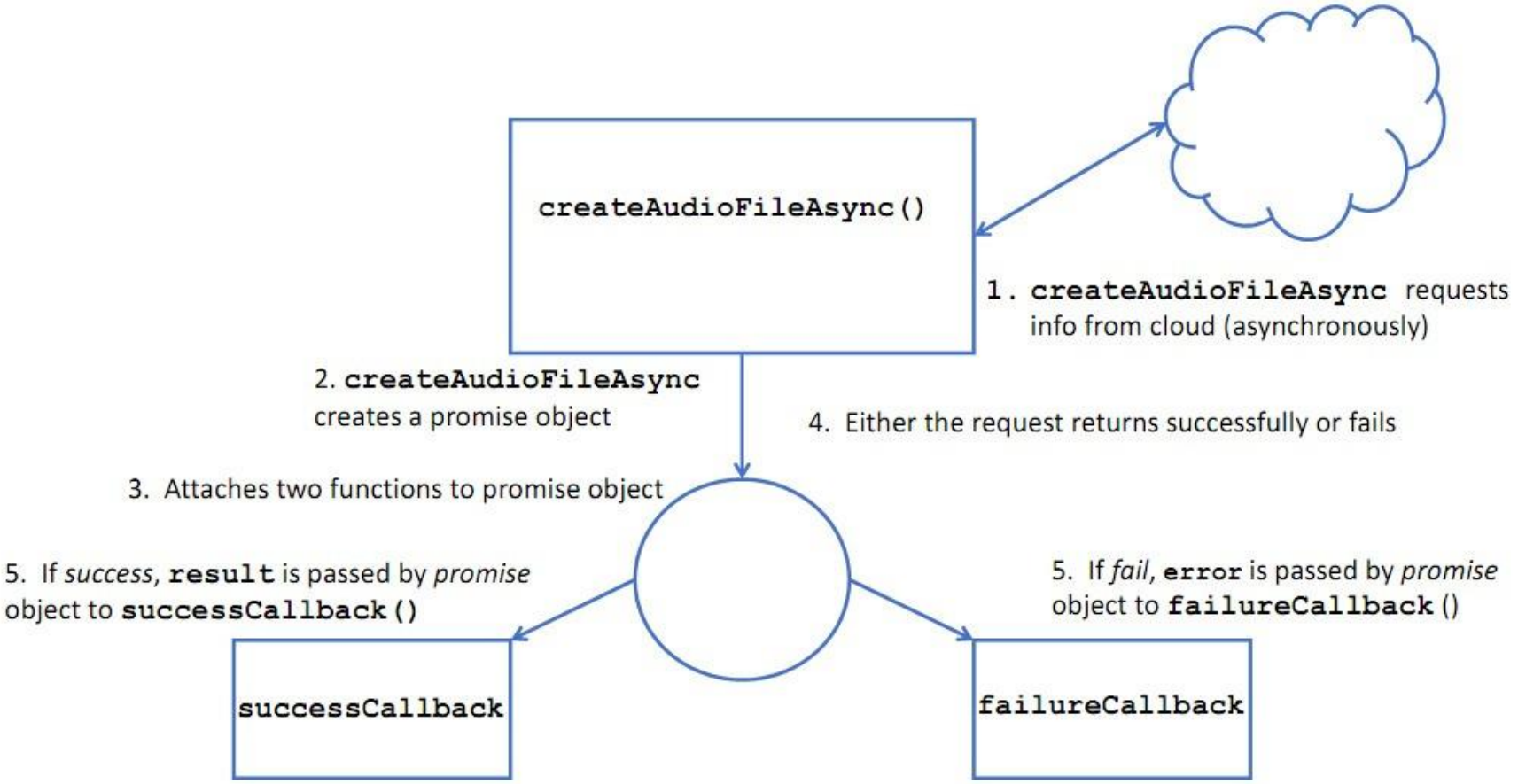
- Regular installation, when there is no promise



5.3 Promises

(4)

- Install using promises



5.3 Promises (5)

- We can attach our callbacks via the promise returns _

```
const promise = createAudioFileAsync(audioSettings);
promises. then (successCallback, failureCallback);
```

- The abbreviation can be used as follows:

```
createAudioFileAsync(audioSettings). then (successCallback, failureCallback);
```

- These are called asynchronous function calls. Unlike Like old-fashioned callbacks, a promise is included with some guarantees:

  Callbacks will never be called before completion current executor of the JavaScript event loop
  - Callbacks are added with **then() syntax** even if the asynchronous operation succeeds or fails
  - Multiple callbacks can be added by calling **then()** multiple times okay called chaining

5.3 Promises (6)

- Promise chain
  - The common need is to perform two or more asynchronous operations contiguously
    - each subsequent activity starts when the previous one succeeds
    - the result from the previous step is the promise for the next step
  - Do this by creating a chain of promises (promise chain).

Old-fashioned syntax

```
doSomething(function(result) {
doSomethingElse(result, function(newResult) {
doThirdThing(newResult, function(finalResult) {
        console.log('Got the final result: ' + finalResult);
}, failureCallback);
}, failureCallback);
}, failureCallback);
```

New syntax

```
doSomething(). then (function(result)
{    return doSomethingElse(result);
})
. then (function(newResult) {
   return doThirdThing(newResult);
})
. then (function(finalResult) {
console.log('Got the final result: ' + finalResult);
})
. catch (failureCallback);
```

5.3 Promises (7)

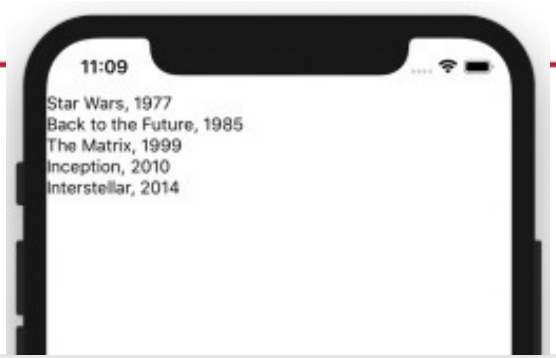
- Promise chain
  - The arguments of **then()** are optional, and **catch(failureCallback)** stands for **then(null, failureCallback)** .
  - The way of writing is represented by arrow functions (=>):

```
doSomething()
. then (result => doSomethingElse(result))
. then (newResult => doThirdThing(newResult)) . then
(finalResult => {
        console.log(`Got the final result: ${finalResult}`);
})
. catch (failureCallback);
```

For example

```
import React from 'react';
import { FlatList, ActivityIndicator,
        Text, View, SafeAreaView } from 'react-native';

export default class FetchExample extends React.Component {

  constructor(props) {
    super(props);
    this.state = { isLoading: true }
  }

  componentDidMount() {
    return fetch('https://facebook.github.io/react-native/movies.json')
      .then((response) => response.json())
      .then((responseJson) => {

        this.setState({
          isLoading: false,
          dataSource: responseJson.movies,
        }, function () {

        });

      })
      .catch((error) => {
        console.error(error);
      });
  }
```

Example (2)




```
render() {
    if (this.state.isLoading) {
      return (
        <View style={{ flex: 1, padding: 20 }}>
          <ActivityIndicator />
        </View>
      )
    }

    return (
      < SafeAreaView style={{ flex: 1, paddingTop: 20 }}>
        <FlatList
          data={this.state.dataSource}
          renderItem={({ item }) => <Text>{item.title}, {item.releaseYear}</Text>}
          keyExtractor={({ id }, index) => id}
        />
      </SafeAreaView>
    );
  }
}
```