ReactJS

A JavaScript library for building user interfaces

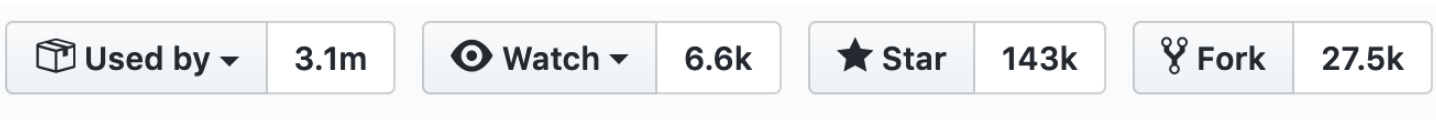
Why ReactJS was developed ?

- Existing ones are heavy-weight Frameworks
- Need for a light-weight library
- Complexity of 2-way data binding
- Update to Real DOM is performance intensive
- Bad UX from using “cascading updates” of DOM tree

What is ReactJS?

- Developed by Facebook.
- ReactJS is an open-source JavaScript library which is used for building user interfaces specifically for single page applications.
- React is a **view layer library**, **not** a **framework** like Backbone, Angular, etc.
- You can’t use React to build a fully-functional web app.
- Documentation at reactjs.org

Who uses React ?



How does React tackle challenges ?

- Uses 1-way data binding (**not 2-way** like Angular)
- Virtual DOM (Efficient for frequent updates)
- Easy to understand what a component will render
- JSX - Easy to mix HTML and JS
- React dev tools and excellent community
- Server-side rendering (useful for SEO)

Core Tenets of React

- Introducing JSX
- Components
- States and Props
- Lifecycle

Introduction to JSX

- JSX (JavaScript XML) is a syntax extension to JavaScript.

```
const element = <h1 className="greeting">Hello, world!</h1>;
```

- You can embed any JavaScript expression in JSX by wrapping it in curly braces.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const element = (
 <h1>
    Hello, {formatName(user)}!
 </h1>
);
```

Introduction to JSX

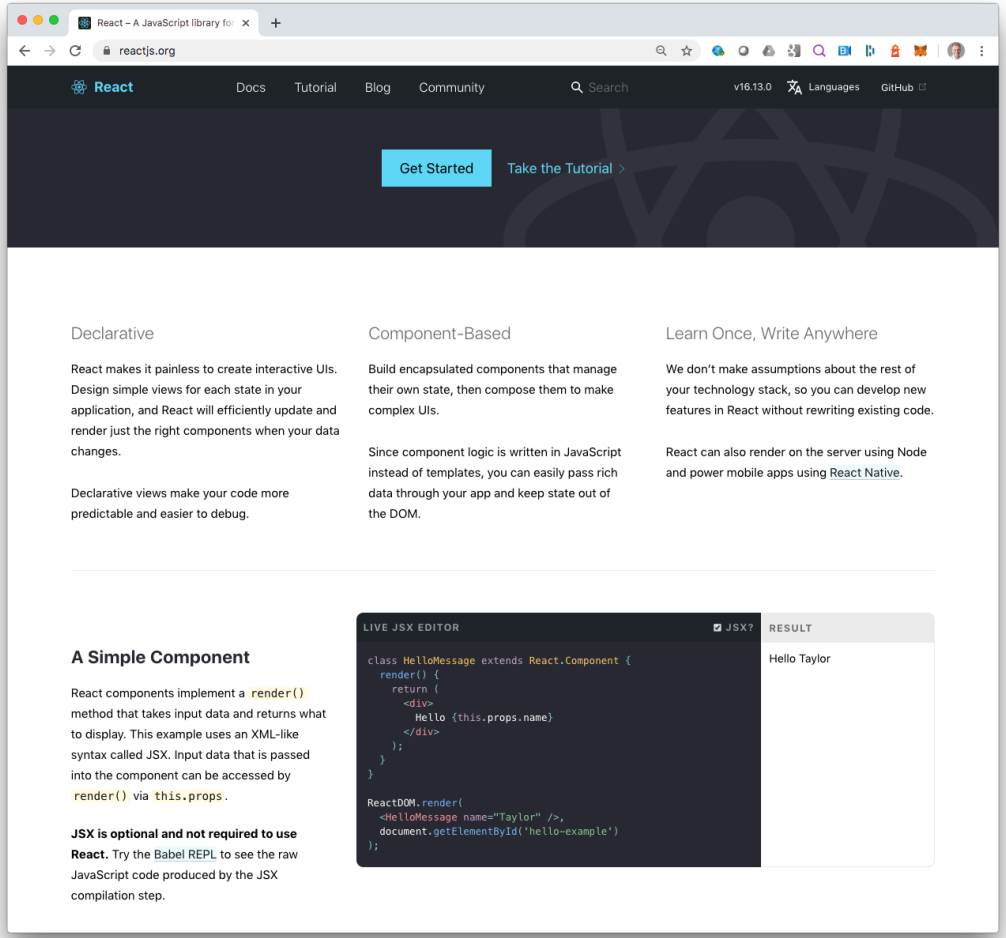
- JSX represents objects. Like XML, JSX tags have tag names, attributes and children.
- Fundamentally, JSX just provides syntactic sugar for the React.createElement(component, props, ...children) function.

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

compiles into:

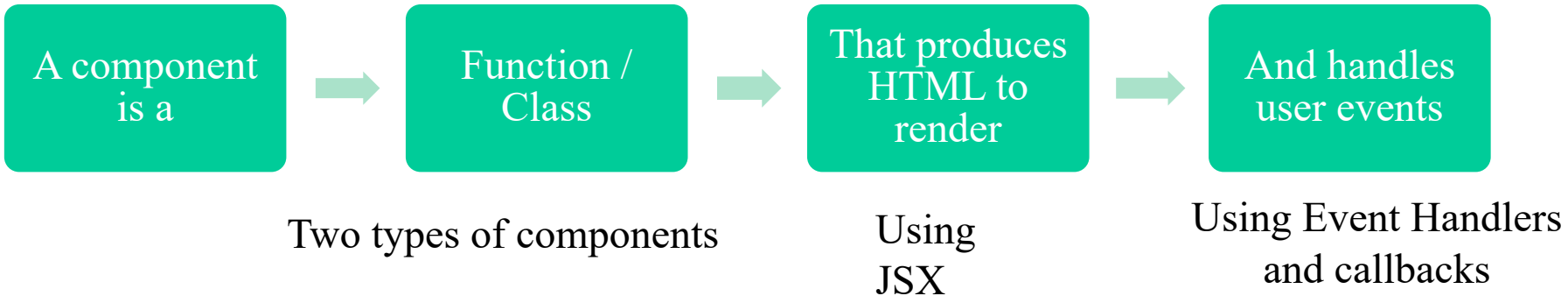
```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

Learning Component way at reactjs.org!



What are Components ?

- Components are building blocks of React Application
- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.



Functional / Stateless Components

- These are simple components that do not maintain their own state.
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Props

- The method to pass a data from parent component to child component.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

Class / Stateful Components

- **Class components** are ECMAScript 6 (ES6) **classes** that can maintain a state, independent existence and a lifecycle of its own.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

State

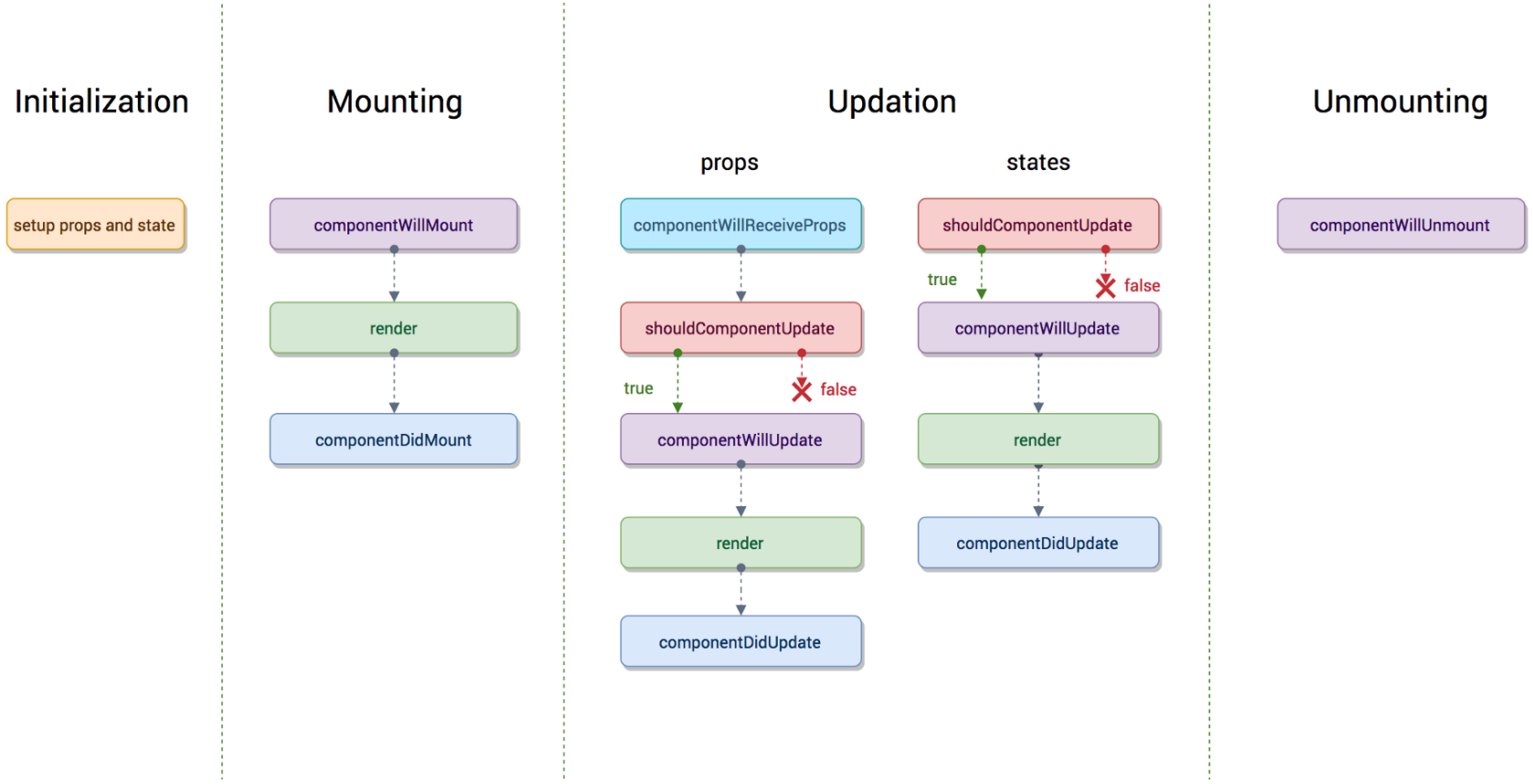
- State is a JavaScript Object containing the component data.
- It represents **internal** state of the component
- Accessed via **this.state**
- State must be initialized when component is created
- State can only be updated using setState() method
- setState() can only add or change the properties and never remove a property.
- When a component's state data changes, the rendered markup will be updated by re-invoking render() method of the class.

State vs Props

- **State** is referred to the **local state of the component** which cannot be accessed and modified outside of the component and can only be used & modified inside the component.
- **Props,** on the other hand, make components reusable by giving **components** the ability to **receive data** from the parent component in the form of props.

STATE	PROPS
Internal data	External data
Can be changed inside component	Cannot be changed
Cannot be changed by parent component	Can be changed by parent component

Component Lifecycle



Samples at reactjs.org

- See:
  - A Simple Component
  - A Stateful Component
  - An Application
  - A Component Using External Plugins

React-Bootstrap

Bootstrap Front-end framework Rebuilt for React

https://react-bootstrap.github.io/

React-Bootstrap

- Replaces the Bootstrap JavaScript.
- Each component has been built from scratch as a true React Component.
- No unneeded dependencies like jQuery because methods and events using jQuery are done imperatively by directly manipulating the DOM. In contrast, React uses updates to the state to update the virtual DOM.

Installation

- Recommended installation via the npm package.

```
npm install react-bootstrap bootstrap
```

- Doesn’t ship with any included CSS. However, a Bootstrap stylesheet **is required** to use these components.

```
{/* The following line can be included in your src/index.js or App.js file*/}

import 'bootstrap/dist/css/bootstrap.min.css';
```

Installation

- CDN can also be used to import the Bootstrap CSS file.

```
<link
  rel="stylesheet"
  href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
  integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
  crossorigin="anonymous"
/>
```

Importing

- Import individual components like react-bootstrap/Button rather than the entire library to reduce the amount of code you end up sending to the client.

```
import Button from 'react-bootstrap/Button';

// or less ideally
import { Button } from 'react-bootstrap';
```

A Vanilla Bootstrap Component

```
import React from 'react';

function Example()  {
  return (
    <div class="alert alert-danger alert-dismissible fade show" role="alert">
      <strong>Oh snap! You got an error!</strong>
      <p>
        Change this and that and try again.
      </p>
      <button type="button" class="close" data-dismiss="alert" aria-label="Close">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
  )
}
```

A React-Bootstrap Component

```
import React, { Component } from 'react';
import Alert from 'react-bootstrap/Alert';

function Example() {
  return (
    <Alert dismissible variant="danger">
      <Alert.Heading>Oh snap! You got an error!</Alert.Heading>
      <p>
        Change this and that and try again.
      </p>
    </Alert>
  )
}
```

React-Bootstrap with state

- State can be passed within React-Bootstrap components as a prop.
- This makes it easier to manage the state as updates are made using React's state instead of directly manipulating the state of the DOM.

React-Bootstrap Examples

- The following Code Sandbox examples show basic usage of React Bootstrap Components such as Jumbotron, Toast, Container, Button
- Without Bootstrap cdn for css- https://codesandbox.io/s/github/react-bootstrap/code-sandbox-examples/tree/master/basic
- With Bootstrap cdn-https://codesandbox.io/s/github/react-bootstrap/code-sandbox-examples/tree/master/basic-cdn

Related URLs

- React Bootstrap: https://react-bootstrap.github.io/
- React Bootstrap Components: https://react-bootstrap.github.io/components/alerts
- Code Sandbox Examples: https://github.com/react-bootstrap/code-sandbox-examples

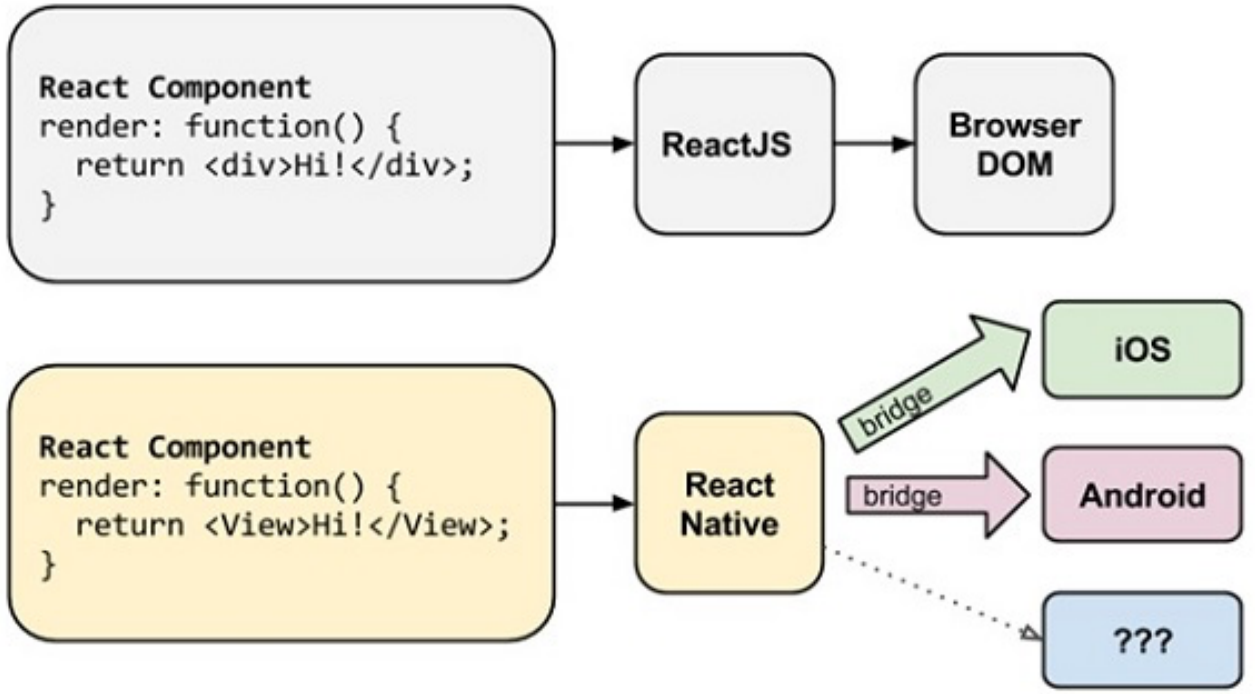
React Native

A framework for building native apps with React.

React Native

- Build native mobile apps using JavaScript and React

https://facebook.github.io/react-native/



React Native - Overview

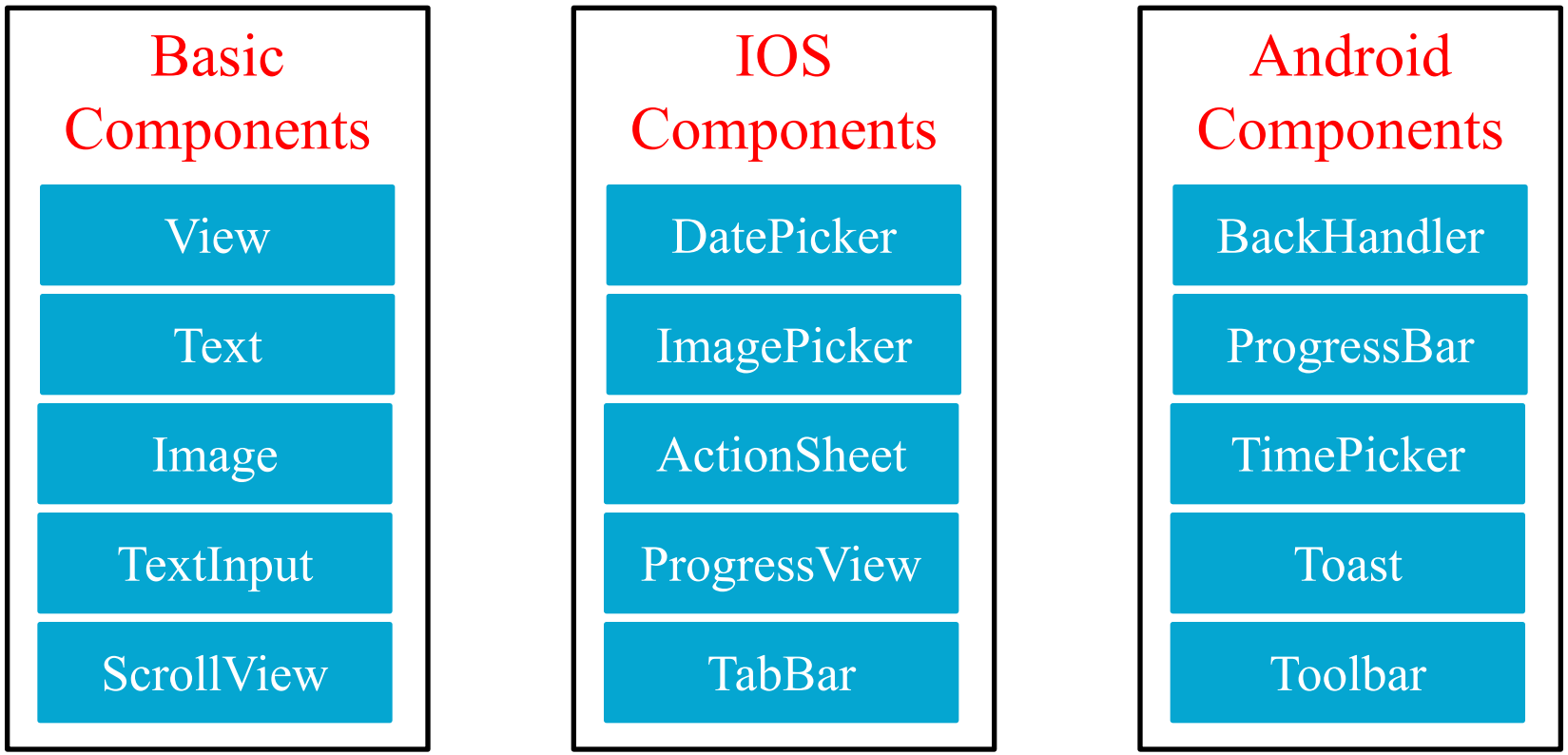
- An embedded instance of JavaScriptCore.
- React components with bindings to Native UI components.
- Manipulating calls into Objective C & java for behavior.
- And polyfills for some web APIs.

React vs React Native

- **React** (a.k.a. ReactJS or React.js) is is a Java**Script library** you use for building dynamic, high performing, responsive UI for your web interfaces.
- **React Native** is an entire **platform** allowing you to build native, cross-platform mobile apps.
- React.js is the heart of React Native, and it embodies all React's principles and syntax, so the learning curve is easy.

React Native - Components

- React Native provides a number of built-in components.



React Native - Stylesheets

- React Native implements a strict subset of CSS.
- Here's an example of how stylesheets are created in React Native:

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 30
  }
});
```

- Then, that style is applied using inline syntax:

```
<View style={styles.container}>
    ...
</View>
```

React Native - Installation

- The easiest way is to use GitHub's Create React Native App:

  https://github.com/react-community/create-react-native-app

- Make sure you have Node v6 or later. No Xcode or Android Studio installation required. Use 'npm':

```
npx  create-react-native-app
```

- Then run the following commands to create and run a new React Native project:

```
create-react-native-app MyApp
cd MyApp
yarn web

yarn ios (for iOS emulator)
yarn android (for android emulator)
```

React Native – Modify and Run your app

- The “npm start” starts a development server for you and print a QR code in your terminal.
- Install the **Expo Client app** on your iOS or Android phone and connect to the same wireless network as your computer. Using the Expo Client app, scan the QR code from your terminal to open your project. See:

  https://expo.io/

- Open App.js in your text editor of choice and edit some lines. The application should reload automatically once you save your changes.

React Native – Hello World

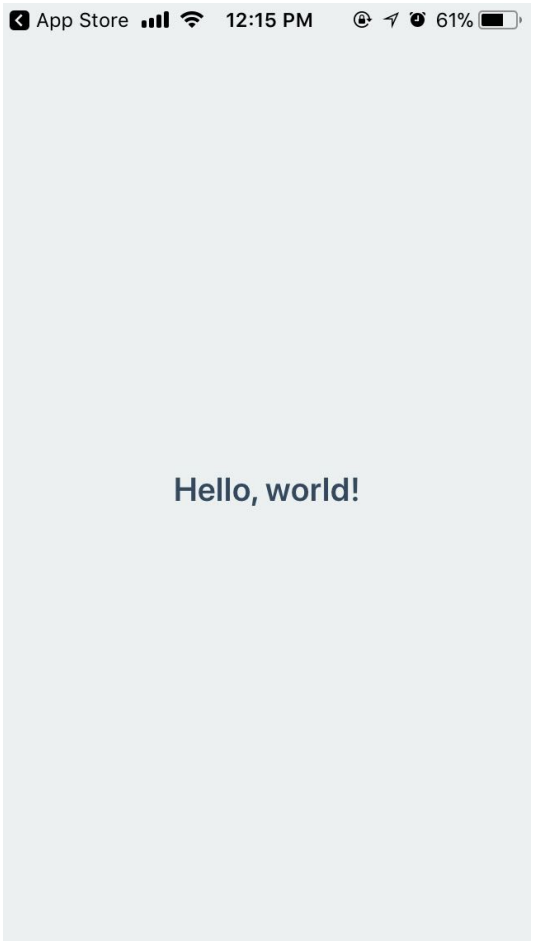
```
1  import React, { Component } from 'react';
2  import { Text, View, StyleSheet } from 'react-native';
3  import { Constants } from 'expo';
4
5  export default class App extends Component {
6    render() {
7      return (
8        <View style={styles.container}>
9          <Text style={styles.paragraph}>
10           Hello, world!
11         </Text>
12       </View>
13     );
14   }
15 }
16
```

See: https://snack.expo.io/BJ-uC-nrb

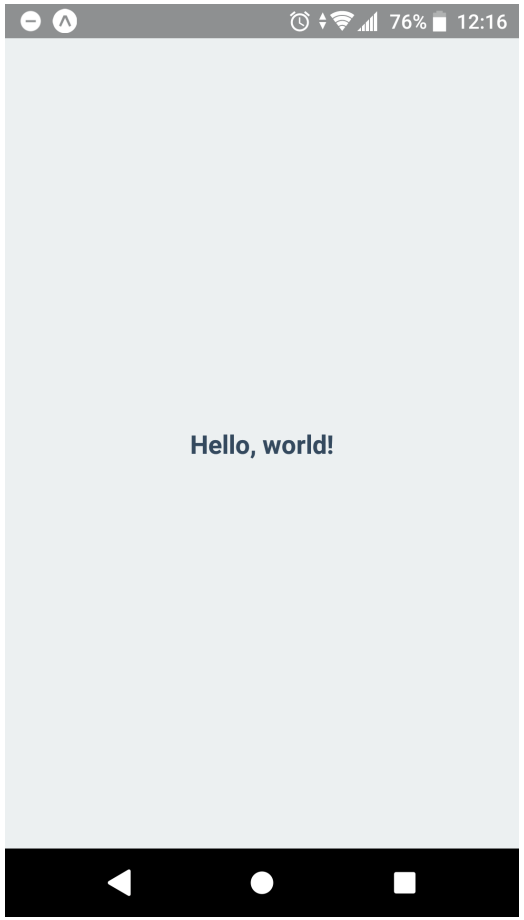
React Native – Hello World (cont'd)

```
17 const styles = StyleSheet.create({
18   container: {
19     flex: 1,
20     alignItems: 'center',
21     justifyContent: 'center',
22     paddingTop: Constants.statusBarHeight,
23     backgroundColor: '#ecf0f1',
24   },
25   paragraph: {
26     margin: 24,
27     fontSize: 18,
28     fontWeight: 'bold',
29     textAlign: 'center',
30     color: '#34495e',
31   },
32 });
33
```

React Native – Hello World (cont'd)



iOS

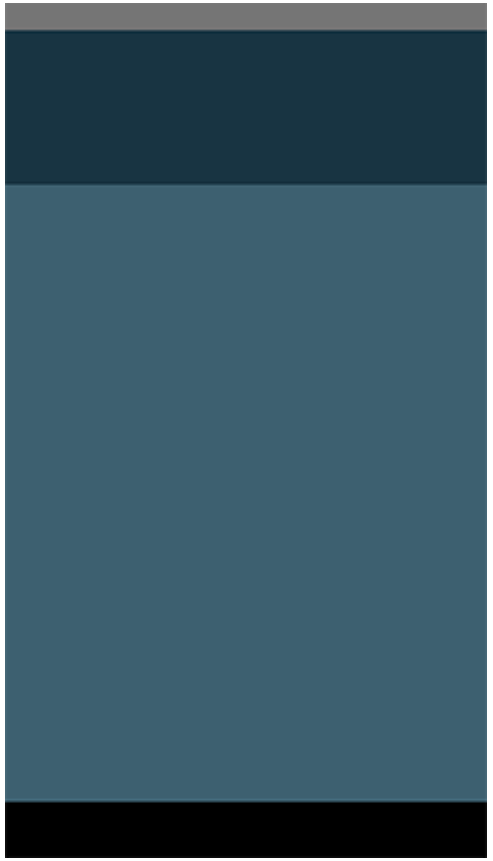


Android

React Native – A calculator example

- https://kylewbanks.com/blog/react-native-tutorial-part-2-designing-a-calculator
- Laying out the Calculator

```
render() {
  return (
    <View style={{flex: 1}}>
            <View style={{flex: 2, backgroundColor:
'#193441'}}></View>
            <View style={{flex: 8, backgroundColor:
'#3E606F'}}></View>
    </View>
  )
}
```



- Put styles into a style.js

```
import Style from './Style';
...

<View style={Style.rootContainer}>
    <View style={Style.displayContainer}></View>
    <View style={Style.inputContainer}></View>
</View>
```

React Native – A calculator example (cont'd)

- Adding the Input Buttons: creating an **InputButton** class that will be used for displaying each button on the calculator

```
// InputButton.js

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

import Style from './Style';

export default class InputButton extends Component {

    render() {
    return(
            <View style={Style.inputButton}>
                <Text style={Style.inputButtonText}>{this.props.value}</Text>
            </View>
    )
  }

}
```

React Native – A calculator example (cont'd)

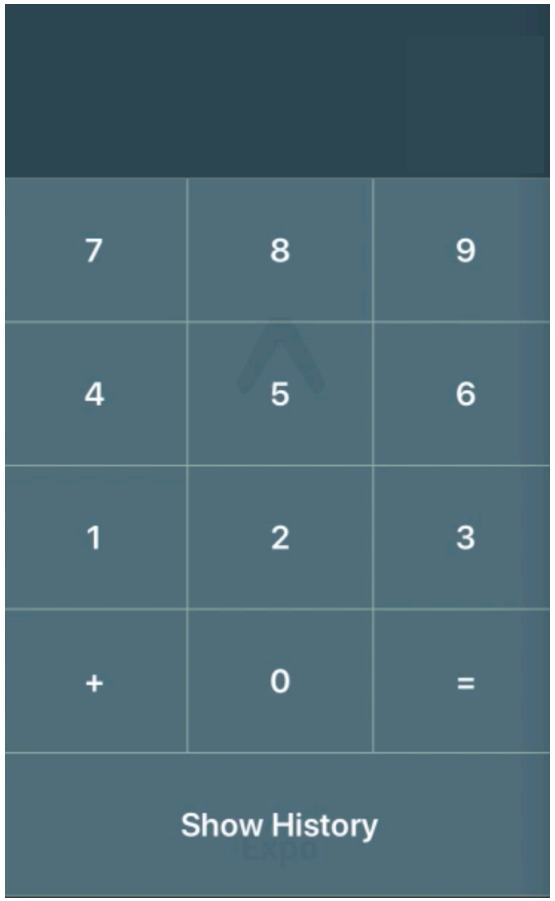
- Adding the Input Buttons in the calculator class

```
// App.js
...
import InputButton from './InputButton';
// Define the input buttons that will be displayed in the calculator.
const inputButtons = [
  [7, 8, 9],
  [4, 5, 6],
  [1, 2, 3],
  ['+' , 0, '=',] ,
  ["Show History"]
];

class ReactCalculator extends Component {
    render() {
    return (
            <View style={Style.rootContainer}>
                <View style={Style.displayContainer}></View>
                <View style={Style.inputContainer}>
          {this._renderInputButtons()}
        </View>
      </View>
    )
  }
}
```

React Native – A calculator example (cont'd)

- Adding the Input Buttons in the calculator class

```
  /**
     * For each row in `inputButtons`, create a row View and add
create an InputButton for each input in the row.
   */
    _renderInputButtons() {
        let views = [];

    for(var r = 0; r < inputButtons.length; r ++) {
            let row = inputButtons[r];

            let inputRow = [];
      for(var i = 0; i < row.length; i ++) {
                let input = row[i];

                inputRow.push(
          <InputButton value={input} key={r + "-" + i} />
        );
      }

     views.push(<View style={Style.inputRow} key={"row-" +
r}>{inputRow}</View>)
    }

    return views;
  }
```



React Native – A calculator example (cont'd)

- Handling Touch Events: Firstly update the **InputButton** to use a **Touchable** view instead of the **View** it currently uses

```
// InputButton.js
...

    render() {
    return(
            <TouchableHighlight style={Style.inputButton}
                                underlayColor="#193441"
                                onPress={this.props.onPress}>
                <Text style={Style.inputButtonText}>{this.props.value}</Text>
            </TouchableHighlight>
    )
  }

…
```

You should also notice that we pass on the **onPress** prop to the **TouchableHighlight** view, so we'll need to provide that from our presenting **Component**:

React Native – A calculator example (cont'd)

- Handling Touch Events: Update the calculator class which uses the input buttons

```
// App.js  Calculator class
...

    _renderInputButtons() {
    …
        inputRow.push(
           <InputButton
                    value={input}
            onPress={this._onInputButtonPressed.bind(this, input)}
                    key={r + "-" + i}/>
    );
  }

    _onInputButtonPressed(input) {
        //handle press button events here
  }

…
```

React Native – A calculator example (cont'd)

- Using State: State allows us to update the UI of our applications based on dynamic data.
- The first thing we'll use **State** for is to update the display based on the numbers entered by the user

```
// App.js
…
class ReactCalculator extends Component {

    constructor(props) {
          super(props);

        this.state = {
            inputValue: 0
        }
  }
```

React Native – A calculator example (cont'd)

```
// App.js
…
    render() {
    return(
            <View style={Style.rootContainer}>
                <View style={Style.displayContainer}>
                    <Text style={Style.displayText}>{this.state.inputValue}</Text>
            </View>
            <View style={Style.inputContainer}>
                {this._renderInputButtons()}
            </View>
      </View>
    )
  }
```

Running the app, you should see that zero is displayed in **Text** view. That's because we set the value to **this.state.inputValue**, which we initialized to zero during the constructor.

React Native – A calculator example (cont'd)

- More things to do regarding the state:
  - Modify the inputValue in the button pressed events handler
  - Add more states to save intermediate results and used as status indicators
  - Saving calculation histories

React Native – A calculator example (cont'd)

- Now we have a home screen to display the calculator, and we need to add another screen to show calculation history
- Using FlatList to display a list of results (like a UITableView in IOS)

```
class HistoryScreen extends React.Component {

  render() {
    return (
      <View style={{ flex: 1}}>
        <FlatList
          data={historyData}
          renderItem={({item}) => <Text style={Style.historyItem}>{item.key}</Text>}
    />
      </View>
  );
 }
}
```

React Native – A calculator example (cont'd)

- React **Navigation**: Navigation between the Home Screen (Calculator) and the History Screen
- Install React Navigation in your project

```
npm install --save react-navigation
```

- Then you can quickly create an app with a home screen and a history screen:

```
import {
  StackNavigator,
} from 'react-navigation';

const App = StackNavigator({
  Home: { screen: HomeScreen },
  History: { screen: HistoryScreen },
});
```

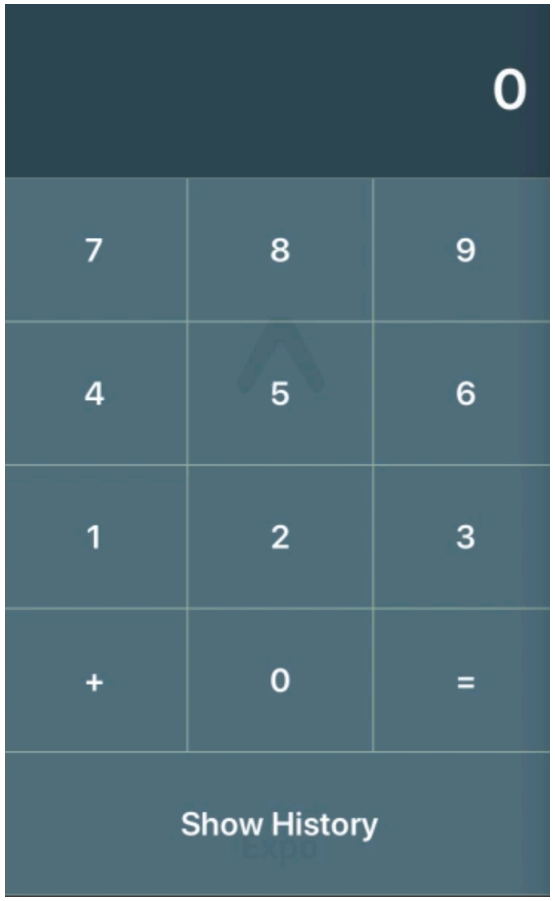
React Native – A calculator example (cont'd)

- The Final Stretch: pass history data from calculator screen to history screen

```
//Calculator Screen
...

_handleShowHistoryButtonPressed() {
 this.props.navigation.navigate('History', {history: equations});
}

...

//History Screen
...

Render() {
    const { params } = this.props.navigation.state;
    const historyData = params ? params.history : [];
}

...
```

React Native – A calculator example (cont'd)

- Demo: Run the React Calculator in the IOS emulator



References

- React JS – Get Started https://reactjs.org/docs/hello-world.html
- Tutorial: Intro To React https://reactjs.org/tutorial/tutorial.html
- Free & Paid Courses for React JS
  https://reactjs.org/community/courses.html
- More Project Examples of React JS
  https://reactjs.org/community/examples.html
- React Native – Get Started https://facebook.github.io/react-native/docs/getting-started.html

Next-Generation JavaScript features

React Apps are typically built with latest version of JavaScript, ECMAScript 6 (ES6).
Using Next Generation features allows us to write clean and more robust code.

- let and const
- Arrow Functions
- Exports and Imports
- Spread and Rest Operators
- Destructuring

1. let & const replacing ‘var’

let

- "Let is the new var"
- **let** allows you to declare variables that are limited to a scope of a block statement, or expression on which it is used, unlike the var keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

```
function varTest() {
  var x = 1;
  {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}
```

```
function letTest() {
  let x = 1;
  {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
 console.log(x);  // 1
}
```

let (cont'd)

- The other difference between var and let is that the latter is initialized to value only when parser evaluates it. E.g.

```
function do_something() {
     console.log(bar); // undefined
     console.log(foo); // ReferenceError
     var bar = 1;
     let foo = 2;
}
```

More Info : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

const

- The **const declaration** creates a read-only reference to a value. It does **not** mean the value it holds is immutable, just that the variable identifier cannot be reassigned. For instance, in the case where the content is an object, this means the object's contents (e.g., its properties) can be altered.
- More Info:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

const (cont'd)

```
// NOTE: Constants can be declared with uppercase or lowercase, but a common
// convention is to use all-uppercase letters.
// define MY_FAV as a constant and give it the value 7
const MY_FAV = 7;

// this will throw an error - Uncaught TypeError: Assignment to constant variable.
MY_FAV = 20;
// MY_FAV is 7
console.log('my favorite number is: ' + MY_FAV);

// trying to redeclare a constant throws an error -  Uncaught SyntaxError: Identifier 'MY_FAV' has already been declared
const MY_FAV = 20;

// the name MY_FAV is reserved for constant above, so this will fail too
var MY_FAV = 20;

// this throws an error too
let MY_FAV = 20;
```

const (cont'd)

```
// it's important to note the nature of block scoping
if (MY_FAV === 7) {
   // this is fine and creates a block scoped MY_FAV variable
   // (works equally well with let to declare a block scoped non const variable)
    let MY_FAV = 20;

   // MY_FAV is now 20
   console.log('my favorite number is ' + MY_FAV);

   // this gets hoisted into the global context and throws an error
   var MY_FAV = 20;
}

// MY_FAV is still 7
console.log('my favorite number is ' + MY_FAV);

// throws an error - Uncaught SyntaxError: Missing initializer in const declaration
const FOO;
```

const (cont'd)

```
// const also works on objects
const MY_OBJECT = {'key': 'value'};

// Attempting to overwrite the object throws an error - Uncaught TypeError: Assignment to constant variable.
MY_OBJECT = {'OTHER_KEY': 'value'};

// However, object keys are not protected,
// so the following statement is executed without problem
MY_OBJECT.key = 'otherValue';  // Use Object.freeze() to make object immutable

// The same applies to arrays
const MY_ARRAY = [];
// It's possible to push items into the array
MY_ARRAY.push('A');  // ["A"]
// However, assigning a new array to the variable throws an error - Uncaught TypeError: Assignment to constant variable.
MY_ARRAY = ['B'];
```

2. ES6 Arrow Functions

2. ES6 Arrow Functions

- An **arrow function expression** is a syntactically **compact** alternative to a regular function expression. Modeled after PHP arrow functions.
- Besides a shorter syntax, they offer advantages when it comes to keeping the scope of the 'this' keyword.

```
function regularFunc(arg1,arg2){
        console.log(arg)
}

function regularFunc (num) {
        return num*2;
}
```

```
const arrowFunc = (arg1,arg2) => {
        console.log(arg);
}
const arrowFunc = arg => {  console.log(arg);
}
const arrowFunc = () => {..}
const arrowFunc = num => num*2
```

3. Exports and Imports (modules)

- In React projects (and actually in all modern JavaScript projects), you split your code across **multiple JavaScript files**, so-called **modules**. You do this to keep each file/ module focused and manageable.
- To access functionality in another file, you need export (to make it available) and import (to get access) statements.
- You have two different types of exports: **default** (unnamed) and **named exports**.
- A file can only **contain one default** and an **unlimited** amount **of named exports.**

person.js

```
const person = {
      name: 'Max';
}
export default person;
```

util.js

```
export const clean = () =>{
      ..
}

export const baseData = 10;
```

app.js

```
import person from './person.js';
import prs from './person.js';
```

```
import {baseData} from './util.js';
import {clean as cln} from './util.js';
import * as bundled from 'util.js';
```

Default export

Named export

4. Spread and Rest operators

• • •

Spread operator (…)

- Used to split up array elements OR object properties .
- The spread operator is extremely useful for **cloning arrays and objects**. Since both are reference types (and not primitives), copying them safely (i.e. preventing future mutation of the copied original) can be tricky. With the spread operator you have an easy way of creating a (shallow!) clone of the object or array.

```
const oldArray = [1, 2, 3];
const newArray = [...oldArray, 4, 5]; // This now is [1, 2, 3, 4, 5];
const oldObject = {
        name: 'Himanshu'
};
 const newObject = {
        ...oldObject,
       age: 28
};
 newObject would then be
 {
        name: 'Himanshu',
        age: 28
 }
```

Rest operator (…)

- Used to **merge** a list of function arguments into an array.

```
function sortArgs = ( ...args) => {
      return args.sort();
}
```

5. Destructuring

- Destructuring allows you to easily **access** the values of **arrays** or **objects** and assign them to variables.

```
const array = [1, 2, 3];
const [a, b] = array;
console.log(a); // prints 1
console.log(b); // prints 2
```

```
const myObj = {
name: 'Himanshu', age: 26
}
const {name} = myObj;
console.log(name);  // prints 'Himanshu'
console.log(age);  // prints undefined
```

Destructuring (cont'd)

Destructuring is very useful when working with function arguments. Consider this example:

```
const printName = (personObj) => {
        console.log(personObj.name);
}
printName({name: 'Himanshu', age: 28}); // prints 'Himanshu'
```

Here, we only want to print the name in but we pass a complete person object to the function. Of course this is no issue but it forces us to call personObj.name inside of our function.

We can condense this code with destructuring:

```
const printName = ({name}) => {
        console.log(name);
}
printName({name: 'Himanshu', age: 28}); // prints 'Himanshu'
```

We get the same result as above but we save some code. By destructuring, we simply pull out the name property and store it in a variable/ argument named name which we then can use in the function body.

Some Important JS Array Functions

A lot of React concepts rely on working with arrays (in immutable ways). Some particularly important ones are the following :

- map() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- find() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find
- findIndex() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex
- filter() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- reduce() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=b
- concat() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b
- slice() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
- splice() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice