Cubbit

Unibo 2022/2023

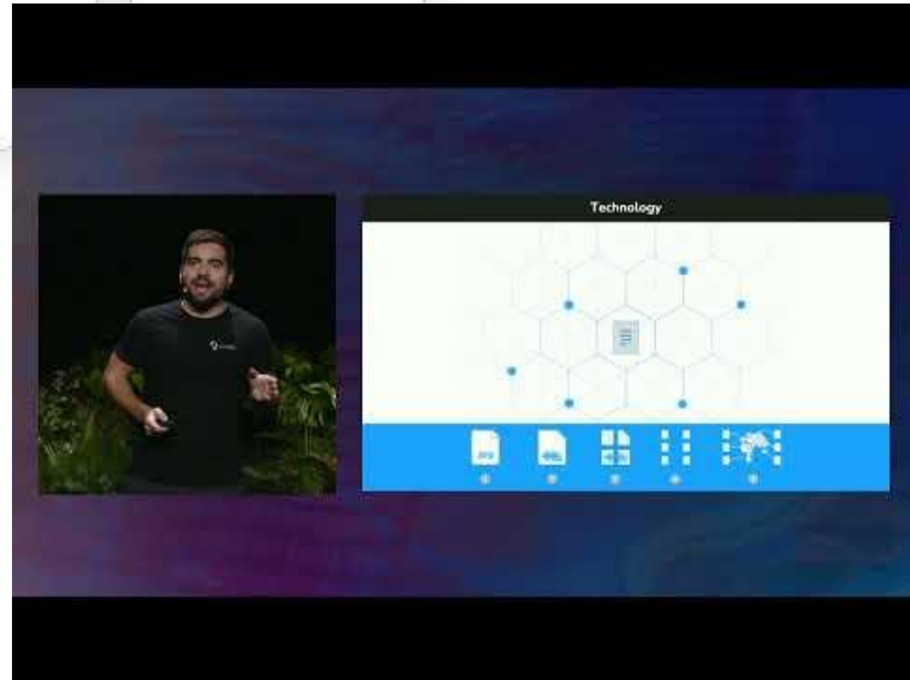
React Native

Aka Another way to save time

giulio.cinelli@cubbit.io

Want to know more about Cubbit?

www.cubbit.io



index.tsx

- What is RN?
- Why RN?
- What do I need to build a RN app?
- How does it work?

Cubbit

What is React Native?

It’s a Framework to build the frontend of Native Applications

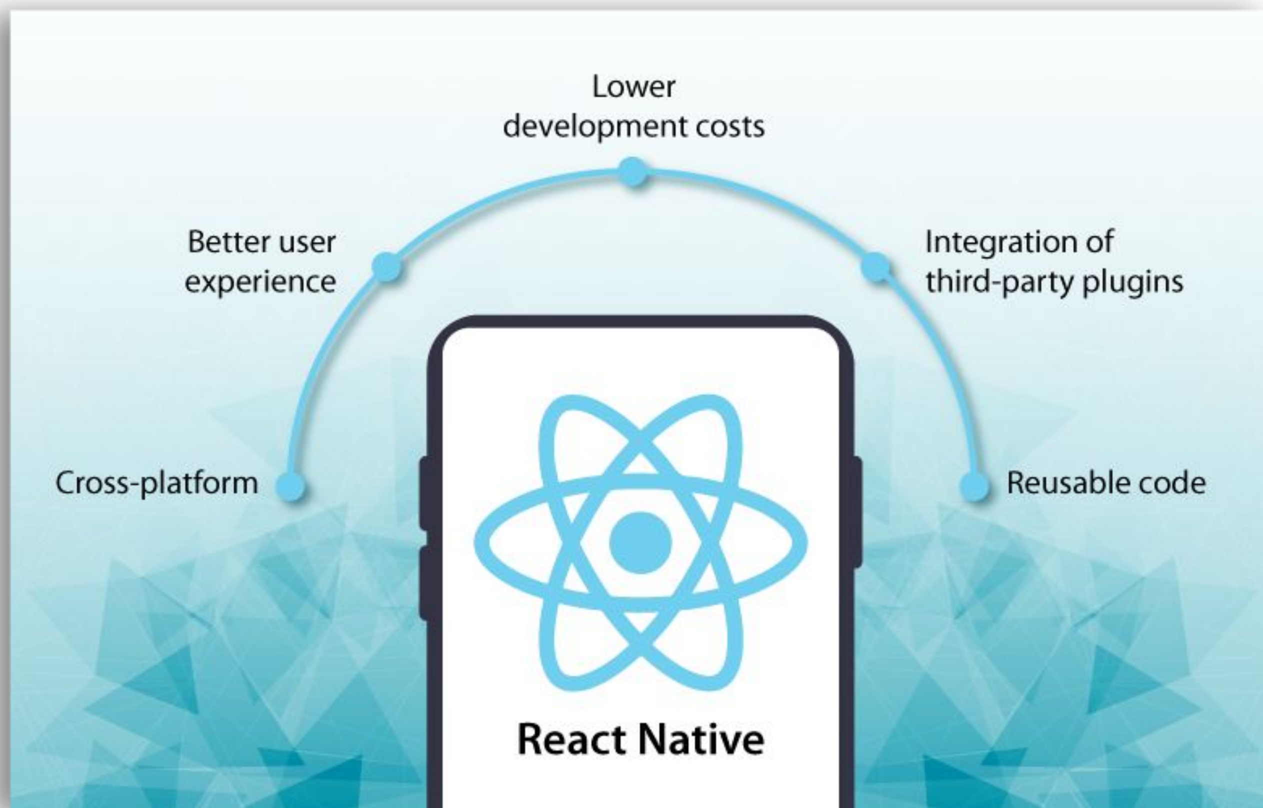
It’s a smart way to write UI elements and logics only once for multiple clients

It’s a safe place for web (React) developers

It’s made to render native user interfaces

It can be used into an existent Native application

Why React Native?



The two main players

Apple - iOS

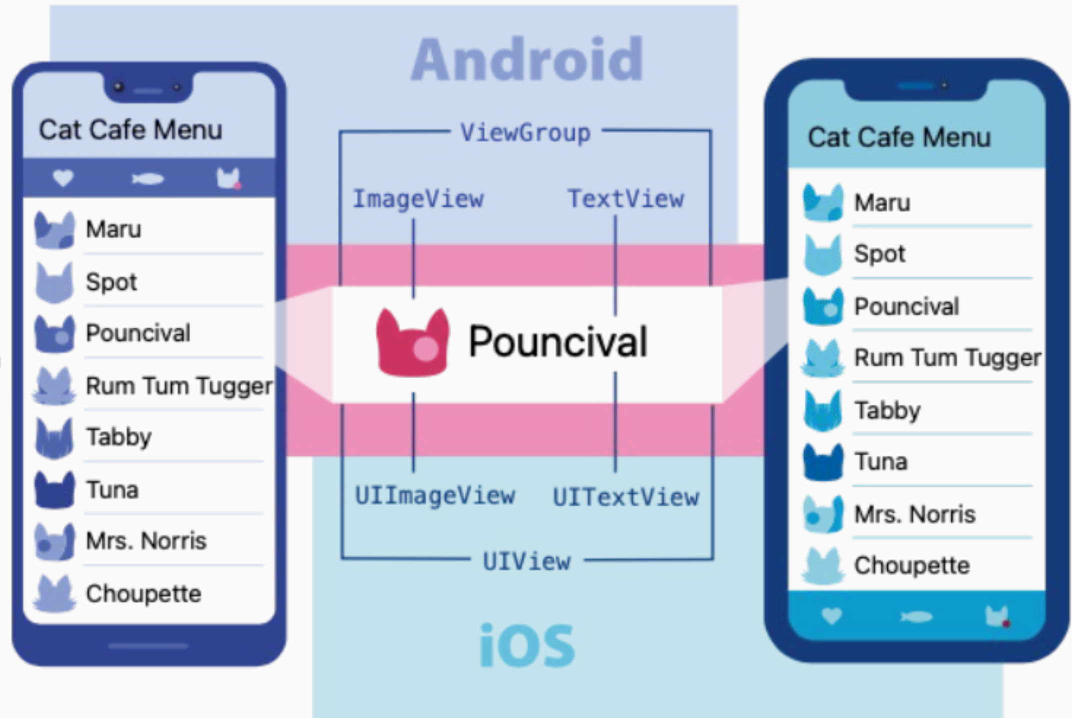
Google - iOS

The two main players

Apple - iOS	Google - Android
Few devices per year	Lot of brands, tons of devices
Very strict publication rules on the store	Quite permissive Store (but at least 2 to handle)
Cupertino Design (Apple's Human Interface Guidelines)	Material Design
Swift / Objective-c	Kotlin / Java
XCode	Android Studio

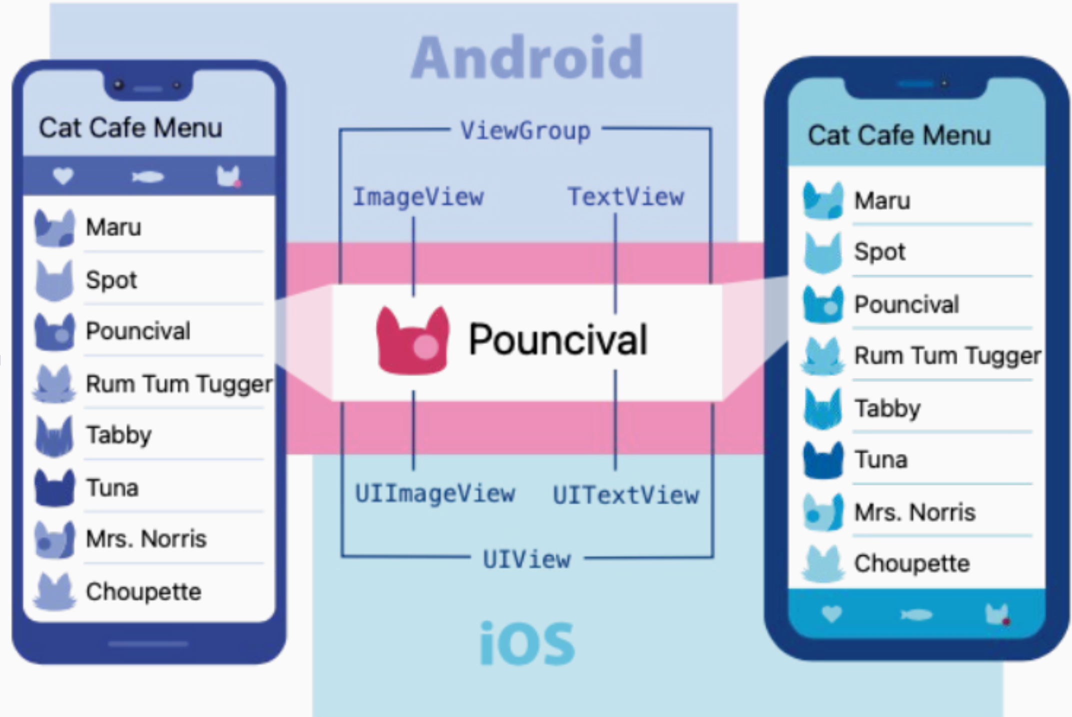
Views

In Android and iOS development, a view is the basic building block of UI: a small rectangular element on the screen which can be used to display text, images, or respond to user input



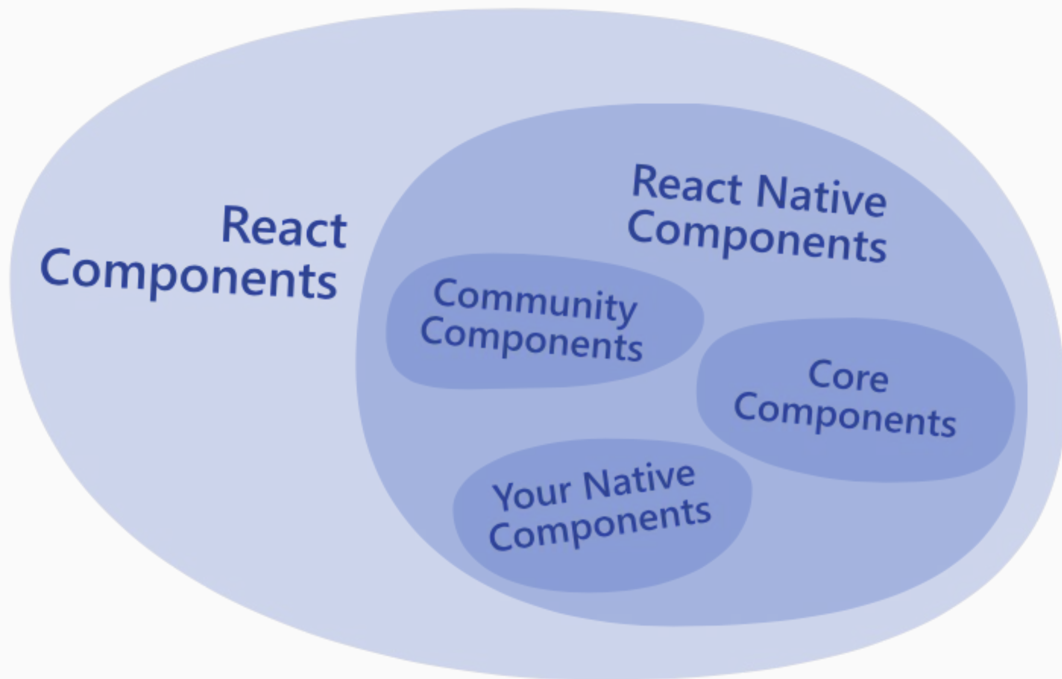
Native Components

In Android development, you write views in Kotlin or Java; in iOS development, you use Swift or Objective-C.
With React Native, you can invoke these views with **JavaScript** using **React components**. At runtime, React Native creates the corresponding Android and iOS views for those components.
Because React Native components are backed by the same views as Android and iOS, **React Native apps look, feel, and perform like any other apps**.



API Structure

Because React Native uses the same API structure as React components, you'll need to understand React components APIs to get started



react.tsx

Fundamentals

Cubbit

- What is React JS?
- Why is it so important?
- What does ReactNative take from React?

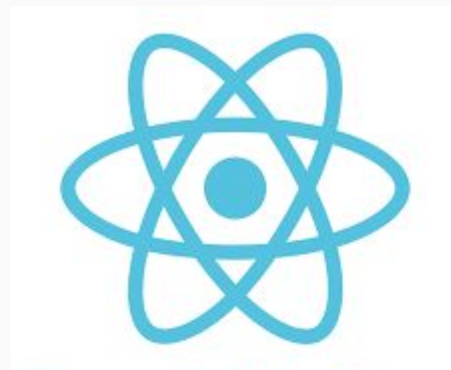
What is ReactJS?

It’s a Javascript Framework for building user web interfaces

Build encapsulated components that manage their own state, then compose them to make complex UIs.

It is not a simple template engine, it is based on jsx to render single components. It handles the DOM to render the final HTML

It builds SPA with a deep focus on rendering single components, not whole pages



JSX

JSX stands for JavaScript XML.

JSX is an XML/HTML like extension to JavaScript.

Just like HTML, JSX tags can have tag names, attributes, and children. If an attribute is wrapped in curly braces, the value is a JavaScript expression.

React elements are immutable. They cannot be changed.

The only way to change a React element is to render a new element every time:

```
const myelement = <h1>I'm a title!</h1>;

------

const myelement = (
    <div>
      <p>I am a paragraph.</p>
      <p>I am a paragraph too.</p>
    </div>
  );

------

const x = 5;

const conditional_element = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

Components

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Components come in two types, Class components and Function components, in this tutorial we will concentrate on Function components.

```
function Room() {
    return <h2>Hi, I am a Room!</h2>;
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

Components Props

Components can be passed as props, which stands for properties.

Props are like function arguments, and you send them into the component as attributes.

```
function Room(props) {
    return <h2>I am a {props.color} Room!</h2>;
}

ReactDOM.render(<Room color="red"/>, document.getElementById('root'));
```

Components in Components

We can refer to components inside other components

```
function Room(props) {
    return <h2>I'm Room {props.name}!</h2>;
}

function University() {
    return (
        <>
            <h1>What rooms are there in this university? </h1>
            <Room name="Ecolani 1" />
            <Room name="Ecolani 2" />
            <Room name="Ecolani 3" />
            <Room name="Ranzani" />
        </>
    );
}

ReactDOM.render(<University />, document.getElementById('root'));
```

Handling Events

React events are named using camelCase, rather than lowercase.

With JSX you pass a function as the event handler, rather than a string.

Your event handlers will be passed instances of SyntheticEvent, a cross-browser wrapper around the browser's native event.

It has the same interface as the browser's native event, including stopPropagation()and preventDefault(), except that events work identically across all browsers.

```
function Form()
{
    function handleSubmit(e)
    {
      e.preventDefault();
      console.log('You clicked submit.');
    }

    return (
        <form onSubmit={handleSubmit}>
            <button type="submit">Submit</button>
        </form>
    );
}
```

Hooks

Hooks allow function components to have access to states and other React features

- useState
- useContext
- useEffect

You can define your own custom hooks

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

DOM Renderer

```
const root = ReactDOM.createRoot(
  document.getElementById('root')
);

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

Hello, world!

It is 12:26:46 PM.

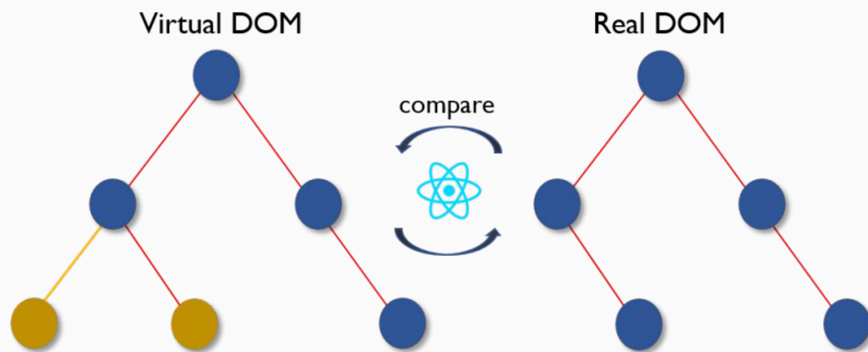
Console Sources Network Timeline

```
▼<div id="root">
  ▼<div data-reactroot>
     <h1>Hello, world!</h1>
    ▼<h2>
       <!-- react-text: 4 -->
       "It is "
       <!-- /react-text -->
       <!-- react-text: 5 -->
       "12:26:46 PM"
       <!-- /react-text -->
       <!-- react-text: 6 -->
       "."
       <!-- /react-text -->
     </h2>
   </div>
 </div>
```

Virtual DOM

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called reconciliation.

This approach enables the declarative API of React: You tell React what state you want the UI to be in, and it makes sure the DOM matches that state. This abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.



components.tsx

ReactNative

Cubbit

- What are the differences with ReactJS?
- Can I reuse ReactJS code for my app?

Core Components

REACT NATIVE UI COMPONENT	ANDROID VIEW	IOS VIEW	WEB ANALOG	DESCRIPTION
<View>	<ViewGroup>	<UIView>	A non-scrollling <div>	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
<Text>	<TextView>	<UITextView>	<p>	Displays, styles, and nests strings of text and even handles touch events
<Image>	<ImageView>	<UIImageView>	<img>	Displays different types of images
<ScrollView>	<ScrollView>	<UIScrollView>	<div>	A generic scrolling container that can contain multiple components and views
<TextInput>	<EditText>	<UITextField>	<input type="text">	Allows the user to enter text

Components

React

```
import React, {useState} from 'react';

export function HomeScreen()
{
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>You clicked {count} times</p>
            <button
                onClick={() => setCount(count + 1)}>
                Click me!
            </button>
        </div>
    );
}
```

React Native

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

export function HomeScreen()
{
    const [count, setCount] = useState(0);

    return (
        <View>
            <Text>You clicked {count} times</Text>
            <Button
                onPress={() => setCount(count + 1)}
                title="Click me!"
            />
        </View>
    );
};
```

Style

With React Native, you style your application using JavaScript.
All of the core components accept a prop named style. The style names and values usually match how CSS works on the web, except names are written using camel casing, e.g. **backgroundColor** rather than **background-color**.

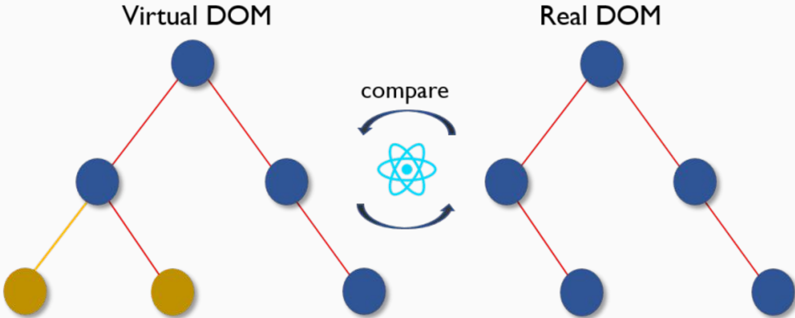
The style prop can be a plain old JavaScript object. That's what we usually use for example code. You can also pass an array of styles - the last style in the array has precedence, so you can use this to inherit styles.

As a component grows in complexity, it is often cleaner to use **StyleSheet.create** to define several styles in one place

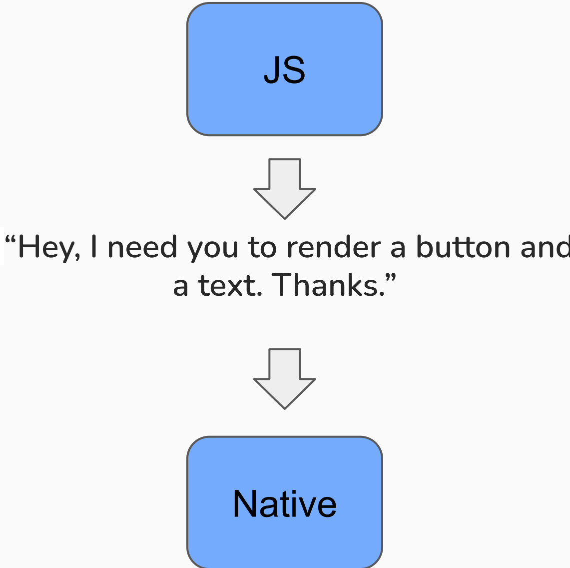
```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

const LotsOfStyles = () => {
    return (
      <View style={styles.container}>
        <Text style={styles.red}>just red</Text>
        <Text style={styles.bigBlue}>just bigBlue</Text>
        <Text style={[styles.bigBlue, styles.red]}>
            bigBlue, then red
         </Text>
        <Text style={[styles.red, styles.bigBlue]}>
            red, then bigBlue
         </Text>
      </View>
    );
};

const styles = StyleSheet.create({
  container: {
    marginTop: 50,
  },
  bigBlue: {
    color: 'blue',
    fontWeight: 'bold',
    fontSize: 30,
  },
  red: {
    color: 'red',
  },
});
```

Renderer

React



React Native



JavaScriptCore

When using React Native, you're going to be running your JavaScript code in two environments:

- In most cases, React Native will use JavaScriptCore, the JavaScript engine that powers Safari. Note that on iOS, JavaScriptCore does not use JIT due to the absence of writable executable memory in iOS apps.

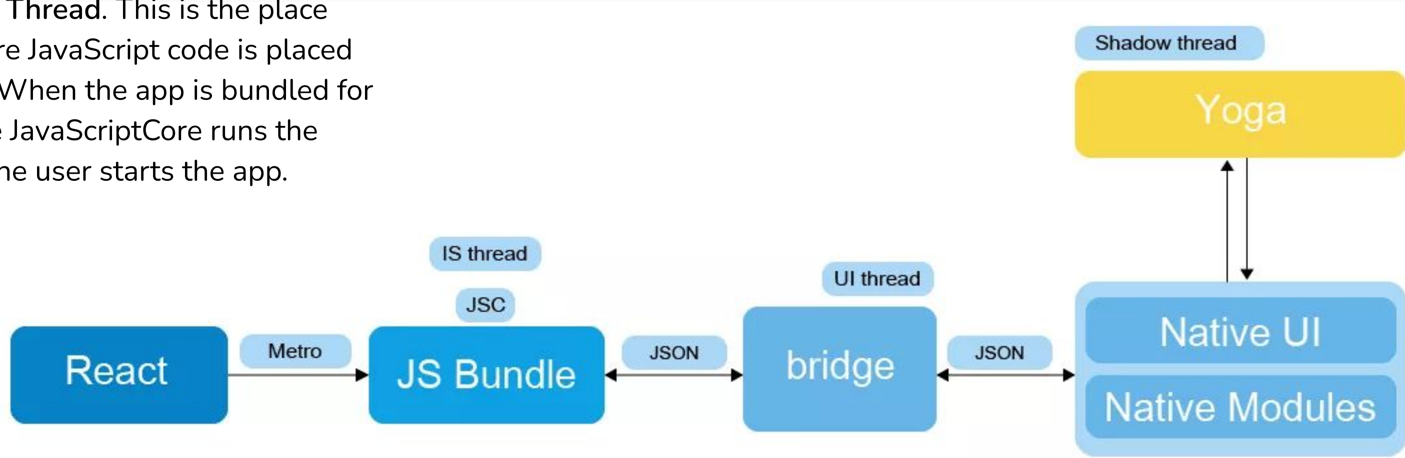
- When using Chrome debugging, all JavaScript code runs within Chrome itself, communicating with native code via WebSockets. Chrome uses V8 as its JavaScript engine.

The JavaScriptCore framework provides the ability to evaluate JavaScript programs from within Swift, Objective-C, and C-based apps.

Also Android can run JavascriptCore

Under the hood - Old architecture

The JavaScript Thread. This is the place where the entire JavaScript code is placed and compiled. When the app is bundled for production, the JavaScriptCore runs the bundle when the user starts the app.



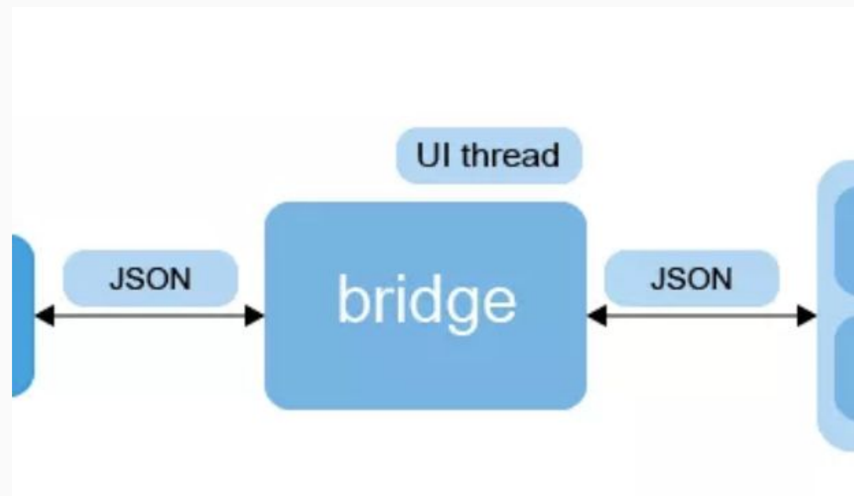
The Native Thread. This is the place where the native code is executed. This component handles the user's interface and ensures seamless communication with the JS thread whenever the app needs to update the UI, run native functions, etc.

The Shadow Thread. It is the place where the layout of your application is calculated. This cross-platform framework handles this task with the help of Facebook's own layout engine called Yoga. It transforms flexbox layouts, calculates them and sends them to the app's interface.

Old architecture limitations

The Bridge had some intrinsic limitations:

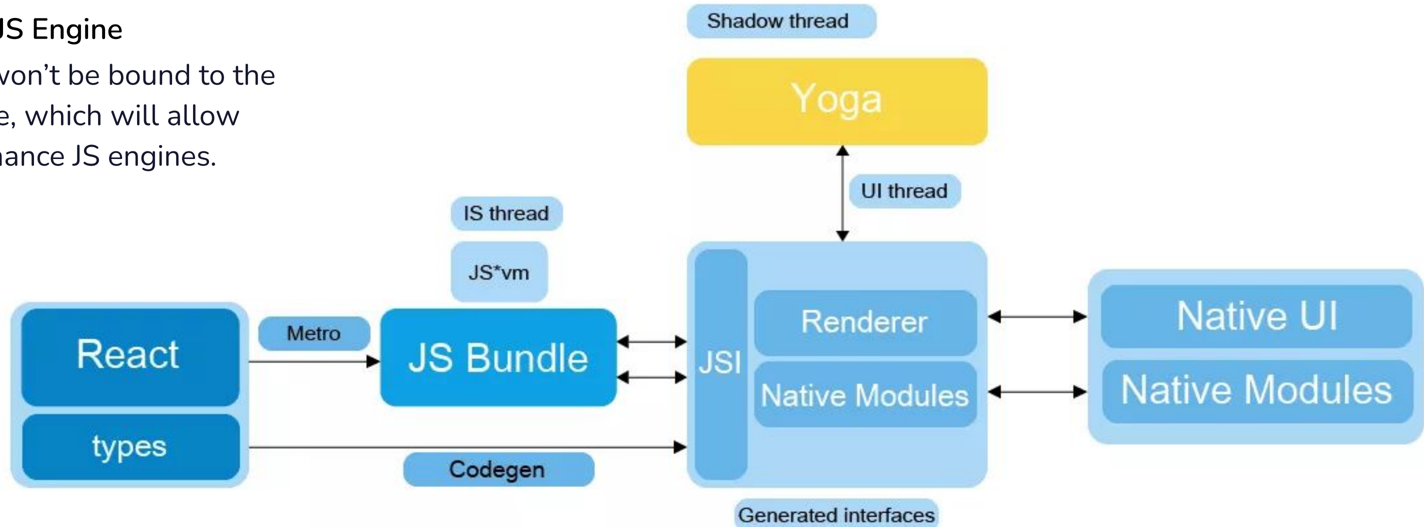
- **It was asynchronous:** one layer submitted the data to the bridge and asynchronously "waited" for the other layer to process them, even when this was not really necessary.
- **It was single-threaded:** JS used to work on a single thread; therefore, the computation that happened in that world had to be performed on that single thread.
- **It imposed extra overheads:** every time one layer had to use the other one, it had to serialize some data. The other layer had to deserialize them. The chosen format was JSON for its simplicity and human-readability, but despite being lightweight, it was a cost to pay.



Under the hood - Next generation

Possibility to Use Any JS Engine

The JavaScript thread won't be bound to the JavaScriptCore anymore, which will allow using any high-performance JS engines.



Depreciation of the Bridge

With the introduction of React Native re-architecture, the Bridge will be gradually eliminated and substituted with a new component called the JavaScript Interface (JSI). This element will also serve as an enabler for a new Fabric and TurboModules.

JSI - one more advantage brought by JSI is the entire synchronization of JS thread and native modules. With the help of the JavaScriptInterface, JS will be able to hold reference to Hot Objects and invoke methods on them. It will also come with a concept of shared ownership, allowing the native side to communicate directly with the JS thread.

Fast Refresh

Fast Refresh is a React Native feature that allows you to get near-instant feedback for changes in your React components. With Fast Refresh enabled, most edits should be visible within a second or two.

- If you edit a module that only exports React component(s), Fast Refresh will update the code only for that module, and re-render your component. You can edit anything in that file, including styles, rendering logic, event handlers, or effects.

- If you edit a module with exports that *aren't* React components, Fast Refresh will re-run both that module, and the other modules importing it. So if both Button.jsx and Modal.jsx import Theme.js, editing Theme.js will update both components.

- Finally, if you edit a file that's imported by modules outside of the React tree, Fast Refresh will fall back to doing a full reload. You might have a file which renders a React component but also exports a value that is imported by a non-React component. For example, maybe your component also exports a constant, and a non-React utility module imports it. In that case, consider migrating the constant to a separate file and importing it into both files. This will re-enable Fast Refresh to work. Other cases can usually be solved in a similar way.

bootstrap.tsx

Start a new project

- Expo CLI
- React Native CLI

Expo CLI

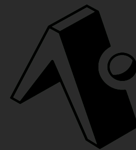
Expo is a set of tools built around React Native

Since the code is written in JavaScript, Expo bundles it up and serves it from S3.

Every time you publish your app, Expo updates those assets and then pushes them to your app so you've always got an up-to-date version.

Expo app is a container ready to receive the updated JavaScript code as a bundle.

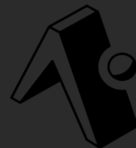
```
$ npm install --global expo-cli

$ expo init my-project

$ cd my-project

$ expo start
```



Expo CLI - PRO

- Fast development bootstrap
- 0 knowledge on mobile development
- Manage dependencies installation
- Remote Live Reload
- Remote CI/CD with native code (pay feature)
- Remote upload to Stores (pay feature)

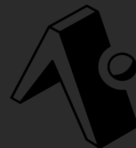
```
$ npm install --global expo-cli

$ expo init my-project

$ cd my-project

$ expo start
```



Expo CLI - CONS

- Build is done by Expo, in their cloud
- Native modules not supported
- Expo apps don't support background code execution
- Javascript source is hosted in their cloud
- The app is going to be at least around 30MB on iOS and at least around 20MB on Android because Expo is including all libraries in the App for now

```
$ npm install --global expo-cli

$ expo init my-project

$ cd my-project

$ expo start
```



EXPO

Live demo

React Native CLI

React Native is shipped with a powerful CLI

Metro is the daemon ready to compile and move javascript code into the Chrome Debugger in order to enable live reload.

Subsequently, the compilation of the native code is launched which incorporates the react native framework

For this configuration it is necessary to set xcode and android studio as if it were developed natively

```
$ brew install node

$ brew install watchman

$ npx react-native init AwesomeProject

$ npx react-native start

--------

$ npx react-native run-ios
```

React Native CLI - PRO

- Full power over the development of the app
- Ability to develop native modules
- Android and iOS native project accessible
- No Expo dependencies
- Final Bundle size limited to your app (+ RN framework)

```
$ brew install node

$ brew install watchman

$ npx react-native init AwesomeProject

$ npx react-native start

--------

$ npx react-native run-ios
```

React Native CLI - CONS

- Initial configuration requires awareness of native development
- Dependencies occasionally require manual configuration
- No remote live reload

```
$ brew install node

$ brew install watchman

$ npx react-native init AwesomeProject

$ npx react-native start

--------

$ npx react-native run-ios
```

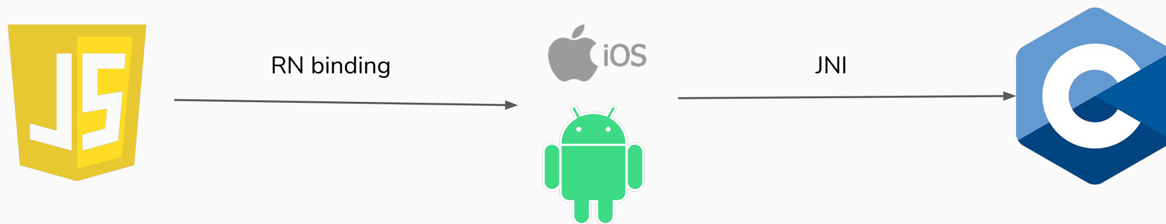
Native Modules

Sometimes a React Native app needs to access a native platform API that is not available by default in JavaScript, for example the native APIs to access Apple or Google Pay.

Maybe you want to reuse some existing Objective-C, Swift, Java or C++ libraries without having to reimplement it in JavaScript, or write some high performance, multi-threaded code for things like image processing.

The NativeModule system exposes instances of Java/Objective-C/C++ (native) classes to JavaScript (JS) as JS objects, thereby allowing you to execute arbitrary native code from within JS.

Bindings are a core concept in native modules development. When we want to call Java or Swift native code from javascript, we need to define common interfaces. Data types also need to be shared.



Arguments types

JAVA	KOTLIN	JAVASCRIPT
Boolean	Boolean	?boolean
boolean		boolean
Double	Double	?number
double		number
String	String	string
Callback	Callback	Function
ReadableMap	ReadableMap	Object
ReadableArray	ReadableArray	Array

Arguments types

OBJECTIVE-C	JAVASCRIPT
NSString	string, ?string
BOOL	boolean
NSNumber	?boolean
double	number
NSNumber	?number
NSArray	Array, ?Array
NSDictionary	Object, ?Object
RCTResponseSenderBlock	Function (success)
RCTResponseSenderBlock, RCTResponseErrorBlock	Function (failure)
RCTPromiseResolveBlock, RCTPromiseRejectBlock	Promise

RN CLI

Live demo

Questions?