

React Native

#1

김나람 / 스튜디오 그로테스큐

발표자 소개

@ 스튜디오 그로테스큐

김나람 a.k.a. Unknown

- 액션스크립트 기반 웹, 키오스크, 임베디드 등
- 개발 자바스크립트 기반 웹, 데스크탑/모바일 앱 개발
- 최근엔 리액트/리액트 네이티브를 기반으로 한 작업을 주로 진행중

{ 주요 연혁 }

- 빅히트 엔터테인먼트 글로벌 오디션 접수 시스템 구축
- 스마트 러닝 월라 애플리케이션 개발
- 삼성 센스 시리즈 9 매뉴얼 애플리케이션 제작
- 원룸 이사 서비스 짐카 유저 애플리케이션 및 배차 솔루션 제작

// 개발 방식에 따른
모바일 애플리케이션의 분류

// 네이티브 애플리케이션

운영체제를 구성하는 언어와 같은 언어로 만들어지는 애플리케이션



[iOS]
Objective-C or Swift



[안드로이드]
Java or Kotlin

장점: 높은 성능이 보장됨
단점: 개발에 투입되는 리소스가 많아짐

// 하이브리드 애플리케이션

- 웹뷰를 이용해 보여지는 영역에 HTML을 보여줌
- HTML+CSS+JS 를 이용한 화면 구현
- 네이티브로 만들어진 앱 컨테이너로 네이티브 기능 연동
- 장점 : 웹 기술에 익숙하다면 빠른 개발 가능
- 단점 : 뷰 퍼포먼스 최적화가 힘듦

// 하이브리드 앱의 정의

Hybrid apps embed a mobile web site inside a native app,
possibly using a hybrid framework like Apache
Cordova and Ionic or Appcelerator Titanium.

하이브리드 앱은 모바일 웹 사이트를 감싼 네이티브 앱으로, Apache
Cordova 나 Ionic, Appcelerator Titanium 등을 사용할 수 있다.

// 크로스 플랫폼 네이티브 애플리케이션

- 네이티브가 아닌 언어로 작성하지만 빌드 단계에서 네이티브로 변환됨
- 리액트 네이티브(JS), 우니티(C#), 자마린(C#), 플러터(Dart) 등
- 장점 : 자신에게 익숙한 언어로 빠른 개발 가능
- 단점 : 네이티브 연동 기능이 많을 경우에는 효율성 떨어짐

// Quora에 올라온 질문

Is React Native actually native or hybrid?

리액트 네이티브는 정확히 네이티브인가요 하이브리드인가요?

// 답변

Actually React Native is neither native nor hybrid.
A React Native app is an actual mobile app.

정확히는 네이티브도 하이브리드도 아닙니다.
다만 리액트 네이티브 앱은 실제 모바일 앱입니다.

// 공식 사이트에 따르면

With React Native, you don't build a "mobile web app",
an "HTML5 app", or a "hybrid app".

You build a real mobile app that's indistinguishable from
an app built using Objective-C or Java.

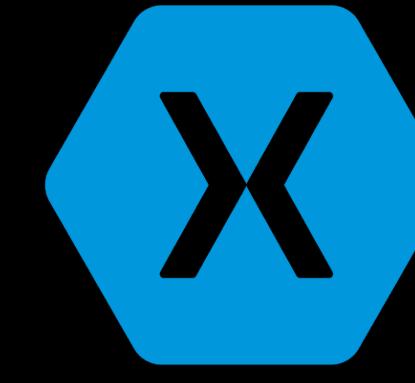
리액트 네이티브를 사용해 "모바일 웹 앱", "HTML5 앱" 혹은
"하이브리드 앱"을 만드는 것이 아닙니다.

Objective-C 또는 Java를 사용하여 만든 앱과 구별 할 수 없는
실제 모바일 앱을 만들 수 있습니다.

// 주요 크로스 플랫폼
네이티브 애플리케이션들의
제작 환경



화면 구성: Unity Editor
로직: C#

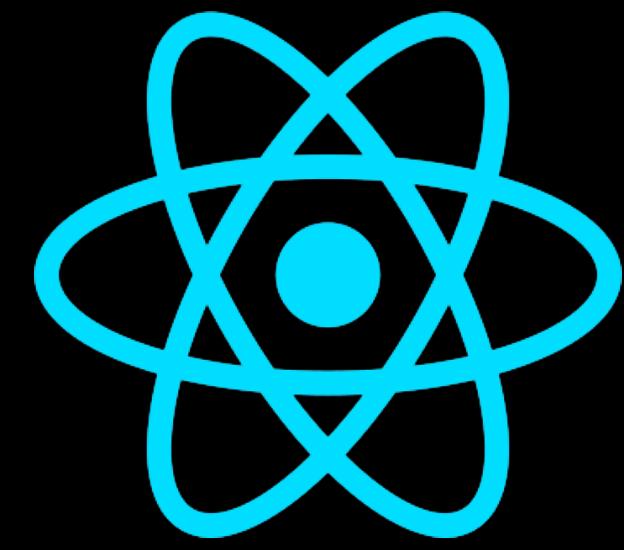


Xamarin

화면 구성: Xamarin.Forms
로직: C#



화면 구성 & 로직: Dart



React Native

화면 구성: JSX (중 XML)
로직: JSX (중 JS)

// JSX

// JSX

- Javascript + XML
- ECMA Script 6 표준의 자바스크립트 사용
- JSX 내에 스타일시트를 포함함
웹의 CSS와 많은 개념을 공유하지만 독자적인 포맷
CSS 파일로 작성 불가

// JSX 예시

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <Text style={styles.instructions}>To get started, edit App.js</Text>
  <Text style={styles.instructions}>{instructions}</Text>
  <Button title={'Hello There!' } onPress={() => {} }/>
</View>
```

// JSX to Android

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <Text style={styles.instructions}>To get started, edit App.js</Text>
  <Text style={styles.instructions}>{instructions}</Text>
  <Button title={'Hello There!'} onPress={() => {}}/>
</View>
```

The diagram illustrates the mapping of JSX code to a mobile application interface. On the left, the JSX code is shown, and on the right, the resulting UI elements are displayed on a smartphone screen. Colored arrows point from specific parts of the code to their corresponding components or text on the phone.

- A red arrow points from the first `<Text>` element to the text "Welcome to React Native!" on the phone screen.
- A green arrow points from the second `<Text>` element to the text "To get started, edit App.js" on the phone screen.
- A pink arrow points from the third `<Text>` element to the placeholder text "Double tap R on your keyboard to reload, Shake or press menu button for dev menu" on the phone screen.
- A cyan arrow points from the `<Button>` element to the blue button labeled "HELLO THERE!" on the phone screen.

// JSX to iOS

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <Text style={styles.instructions}>To get started, edit App.js</Text>
  <Text style={styles.instructions}>{instructions}</Text>
  <Button title={'Hello There!'} onPress={() => {}}/>
</View>
```

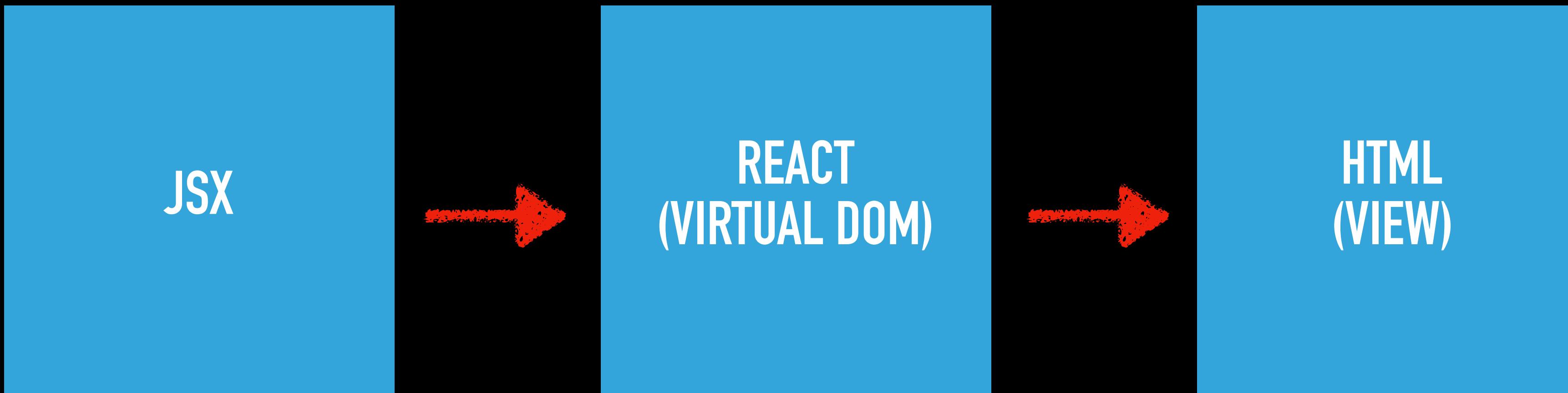
Welcome to React Native!

To get started, edit App.js

Press Cmd+R to reload,
Cmd+D or shake for dev menu

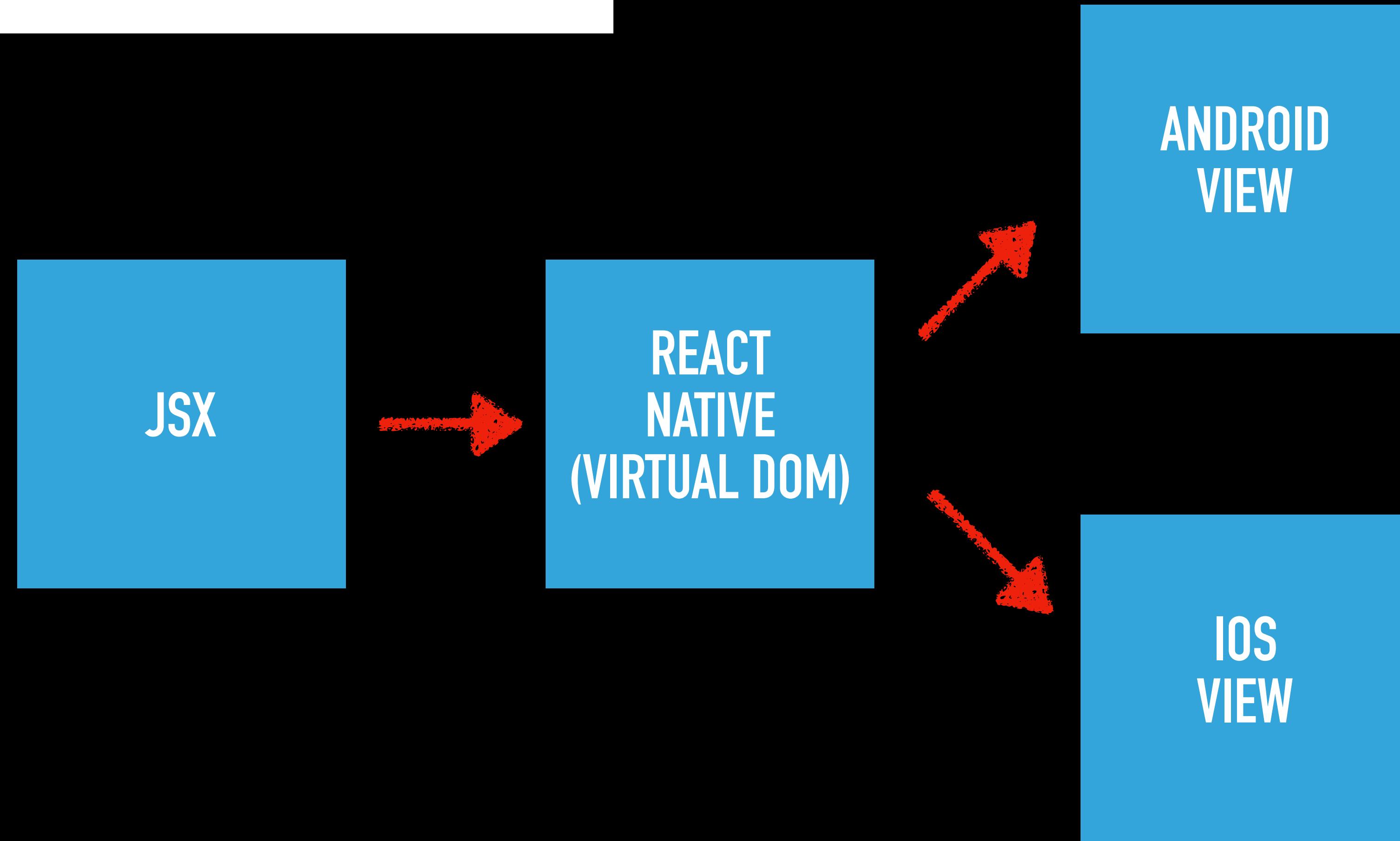
Hello There!

// 리액트의 경우



JSX를 HTML로 렌더링 함

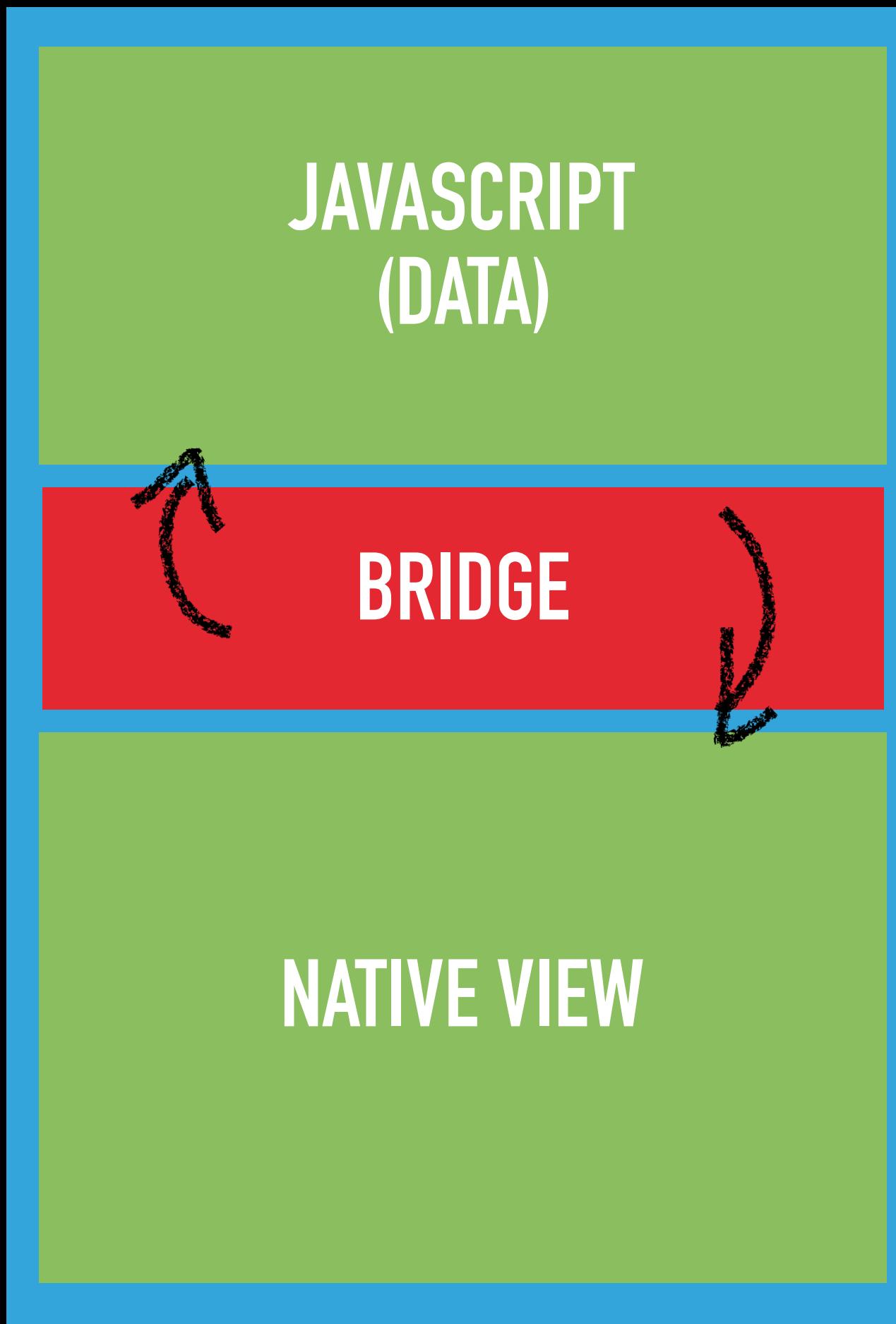
// 리액트 네이티브의 경우



JSX를 네이티브 뷰 형태로 렌더링 함

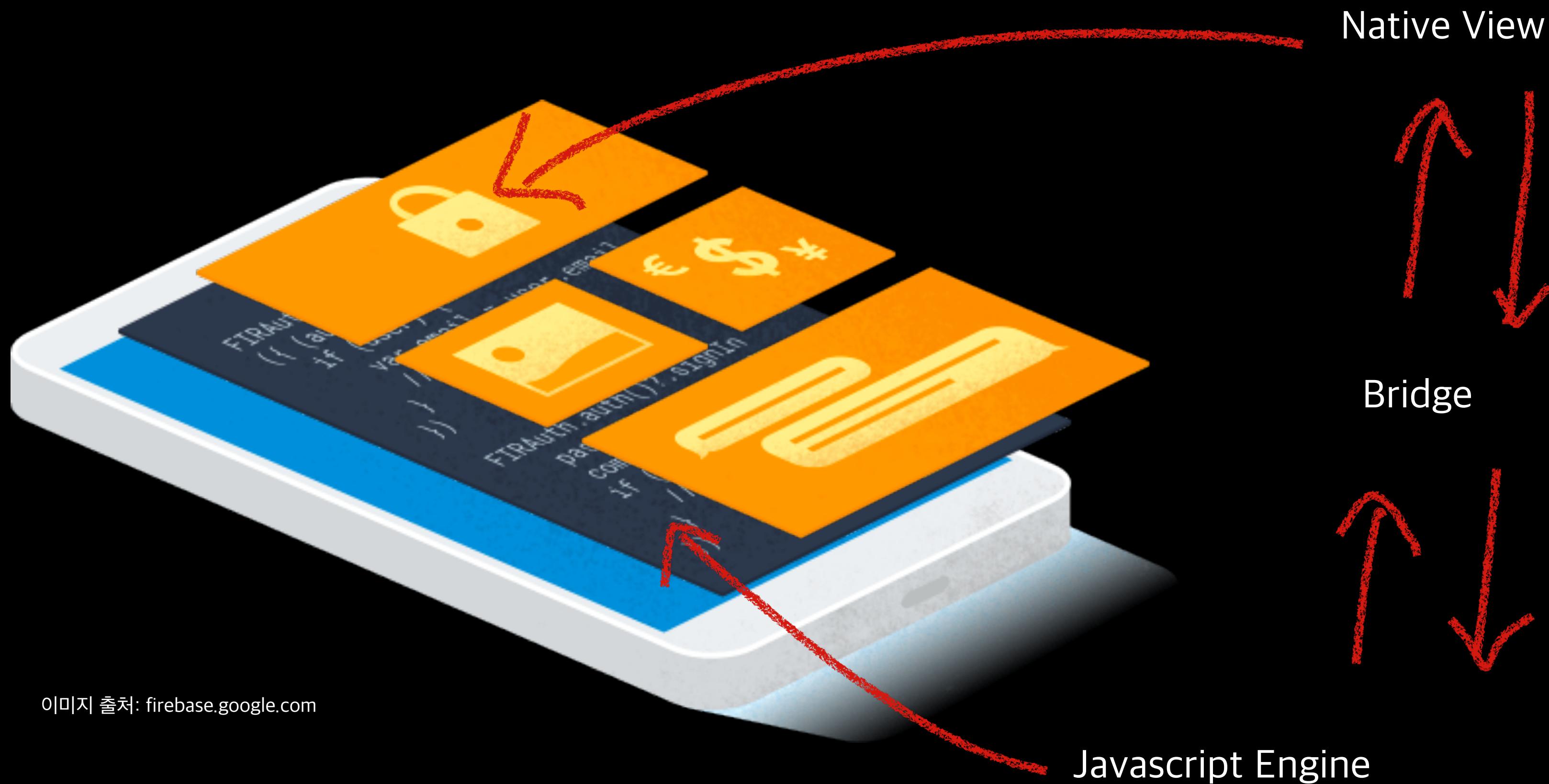
// JSX를 '어떠한 뷰'의 형태로
렌더링 하는 기술이
리액트의 핵심 기술

// 애플리케이션의 구조



데이터 처리는 자바스크립트가 수행하고
뷰와 자바스크립트 엔진을
연결하는 브릿지가 존재

// 애플리케이션의 구조



// 개발 환경 만들기

// Xcode (Mac 전용)

- 맥용 통합 개발 환경 도구
- 대부분의 맥용 개발 도구는 Xcode 의존성이 있으므로 가장 먼저 세팅해주는 것이 필요함
- 맥 앱스토어에서 다운로드

// Homebrew (Mac 전용)

- 맥용 패키지 매니저
(최근엔 리눅스도 지원)
- 패키지 매니저를 통해서 프로그램을 쉽게 설치하고, 버전 별로 관리하고, 삭제할 수 있어 널리 사용
- 몇몇 개발툴은 홈브류를 이용한 설치를 기본으로 함
- brew.sh 의 명령어를 터미널에 붙여넣어 설치

// Chocolatey (Windows 전용)

- 윈도우즈용 패키지 매니저
- 패키지 매니저를 통해서 프로그램을 쉽게 설치하고, 버전 별로 관리하고, 삭제할 수 있어 널리 사용
- 몇몇 개발들은 홈브류를 이용한 설치를 기본으로 함
- chocolatey.org 의 명령어를 파워쉘 또는 cmd에 붙여넣어 설치

// JDK 8

- Java Development Kit
- 최신 버전은 10이지만 리액트 네이티브 관련 도구들은 8을 권장
- 맥 : 오라클 홈페이지에서 다운로드
- 윈도우즈 : choco 를 통해 설치

```
→ ~ choco install jdk8
```

// Python 2

- 직접 사용할 일은 없음
- 개발 도구들이 의존성을 가지고 있으므로 설치 필요
- 맥: 시스템 기본으로 포함 되어 있음
- 윈도우즈 : choco 를 통해 설치

```
→ ~ choco install python2
```

// node.js

- 개발 도구들이 의존성을 가지고 있으므로 설치 필요
- 맥: brew 를 통해 설치

```
→ ~ brew install node
```

- 윈도우즈 : choco 를 통해 설치

```
→ ~ choco install nodejs.install
```

// npm 관련 주의할 점

- node.js 설치시 npm 은 같이 설치됨
- node.js 설치시 최고 관리자 권한을 요구하기 때문에 npm도 최고 관리자 권한으로 설치됨
- CLI 류의 글로벌 설치가 필요한 경우 최고 관리자 권한을 요구하기 때문에 불편함

// 해결책 : Mac의 경우

- brew로 node.js를 설치하면 권한 문제가 발생하지 않음
- 이미 인스톨러로 설치된 node.js가 있다면 제거 후 brew로 설치

// 해결책 : Windows의 경우

- npm config를 설정한다

```
→ ~ npm config set prefix C:\Users\<username>\AppData\Roaming\npm
```

참고: <https://kjwsx23.tistory.com/140>

- cygwin 을 설치하고 리눅스 명령어에 익숙해지는 것도 좋다
- 아이폰 개발을 위해 맥을 구입하는 것도 좋은 방..법이다

// Yarn (npm 으로 대체 가능)

- 페이스북에서 만든 node.js 패키지 매니저
- 페이스북에서 만든 라이브러리들과 상성이 좋음
- 맥: brew로 설치

```
➔ ~ brew install yarn
```

- 윈도우즈: choco로 설치

```
➔ ~ choco install yarn
```

// Android Studio

- 안드로이드 통합 개발 환경 도구
- SDK 매니저, AVD (Android Virtual Device Manager) 매니저 등을 사용하기 위해 설치
- 안드로이드 개발자 센터에서 다운로드
<https://developer.android.com/studio>

// Watchman (Mac 전용)

- 페이스북에서 만든 개발 지원 도구
- 프로젝트의 파일 변경 내역을 주시함
- brew로 설치

```
→ ~ brew install watchman
```

// React Native CLI

- 프로젝트 초기 생성, 플러그인 연결, 에뮬레이터 실행 등에 사용
- yarn으로 설치

```
→ ~ yarn add global react-native-cli
```

// Expo CLI (옵션)

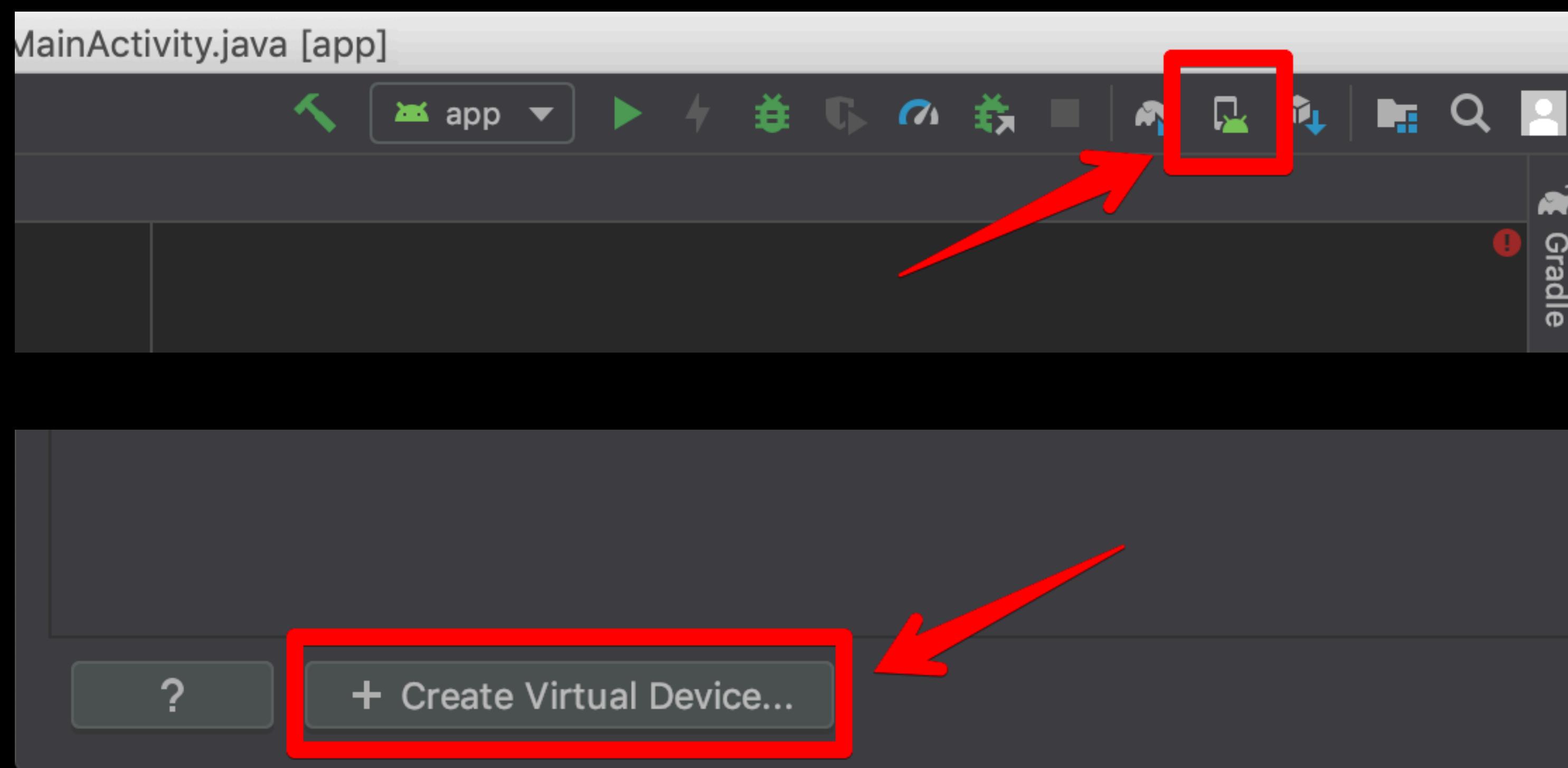
- Expo 프레임워크를 사용할 경우 사용

```
→ ~ yarn add global expo-cli
```

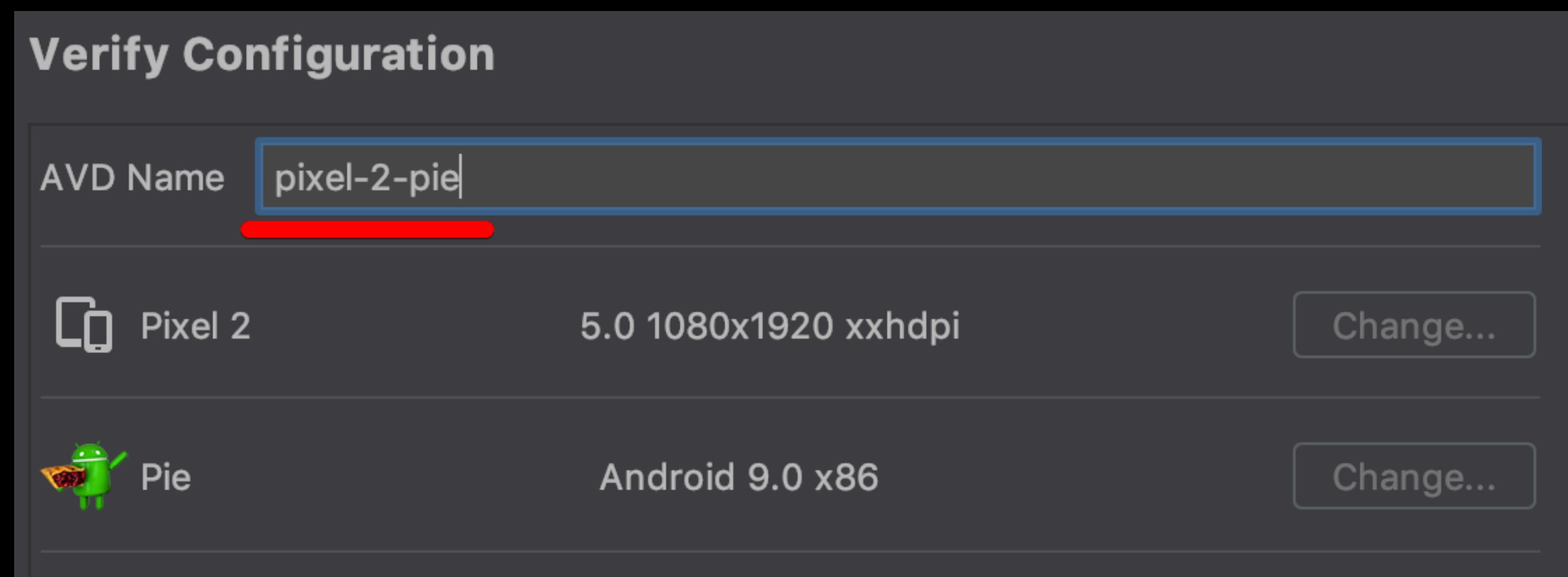
// AVD 생성

- Android Virtual Device
- 초기 1회 생성시 Android Studio의 AVD Manager 필요
- 이후 실행시 쉘에서 호출 가능
- 호출하기 쉬운 이름으로 생성하는 것이 좋음
예) Pixel 2 XL (대소문자 혼합, 공백 문자 입력하기 어려움)
pixel-2-xl (좋음)

// AVD 최초 생성



// AVD 이름 설정



// AVD 실행

- 개발한 코드를 확인하기 위해 안드로이드 에뮬레이터 실행 필요
- ANDROID_HOME 환경변수 설정 필요
참고: <https://bit.ly/2v2Wllr>
- 쉘에서 실행

```
→ ~ emulator -avd <name>
```

```
→ ~ emulator -avd pixel-2-pie
```

// AVD가 생성/실행 되지 않을 때

- AVD를 생성하고 실행하기 위해서는 많은 CPU 자원과 RAM 자원을 필요로 함
- Windows의 경우 가상화 기술이 지원되어야 가능
참고: <https://bit.ly/2Gi2JBk>
- 에뮬레이터를 구동시키기 힘들다면 실제 기기를 USB로 연결하는 방법을 사용할 수 있음

// 프로젝트 생성

// 프로젝트 생성

- 쉘에서 react-native-cli 를 이용해 생성

```
→ ~ react-native init HelloRN
```

- 이후 작업을 위해 생성된 디렉토리 이동

```
→ ~ cd HelloRN
```

// 메트로 번들러 실행

- JSX를 읽어들여서 네이티브와 연결하는 역할
- 에뮬레이터 실행 전 항상 번들러를 먼저 실행하는 것이 좋음

→ ~ **yarn start**

- 번들러 실행 후 새로운 쉘에서 이후 명령 진행

// 안드로이드 테스트

- 에뮬레이터 실행

```
→ ~ emulator -avd <name>
```

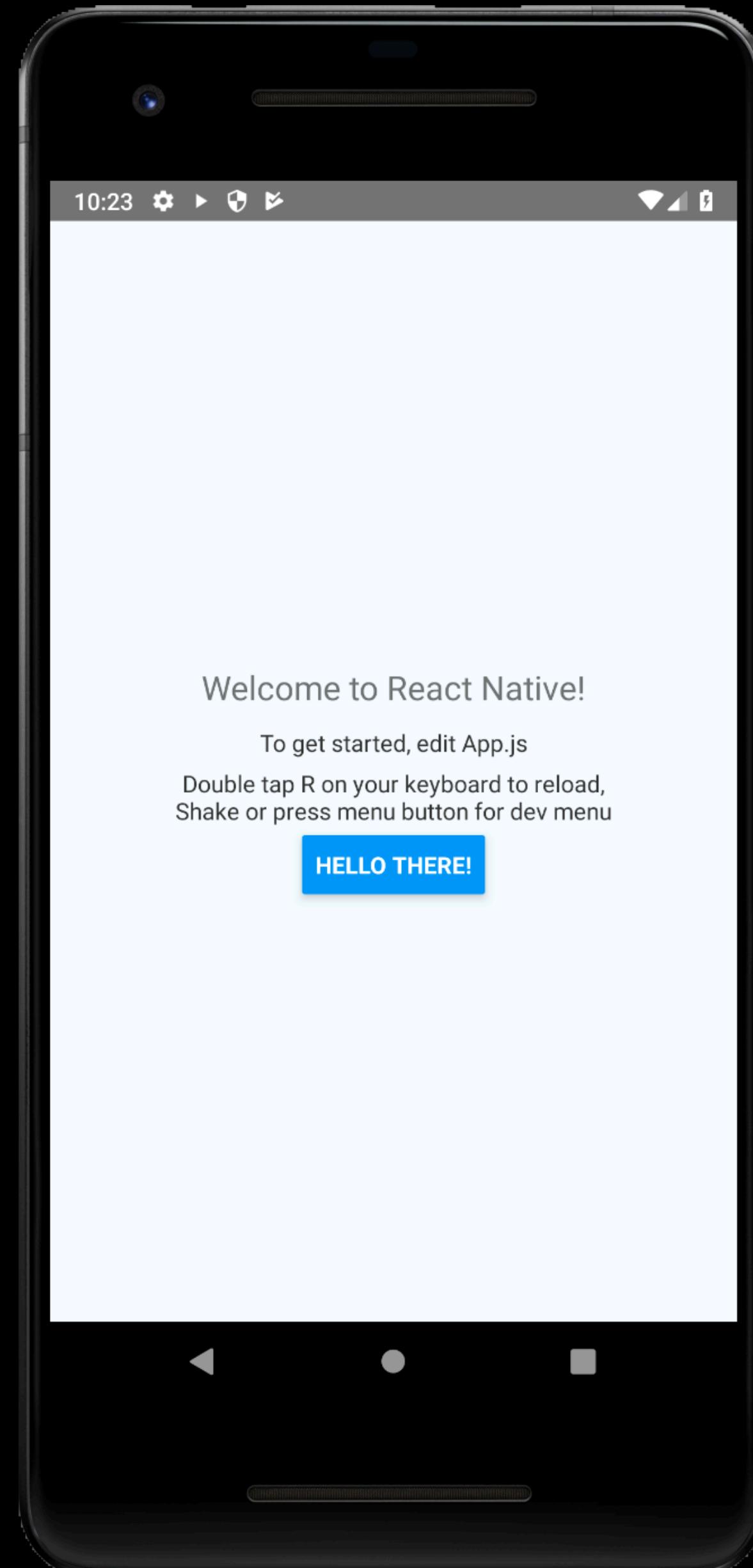
- 에뮬레이터 실행 후 새로운 웹에서 react-native-cli 명령으로 에뮬레이터에 앱을 설치/실행

```
→ ~ react-native run-android
```

// 실행 결과

```
helloRN — unk@Unkui-MBP-2 — -zsh — 80x24
node node /usr/local      emulator      ~/tests/helloRN
10:21:44 D/app-debug.apk: Uploading app-debug.apk onto device 'emulator-5554'
10:21:44 D/Device: Uploading file onto device 'emulator-5554'
10:21:44 D/ddms: Reading file permission of /Users/unk/tests/helloRN/android/app/
build/outputs/apk/debug/app-debug.apk as: rw-r--r--
10:21:45 V/ddms: execute: running pm install -r -t "/data/local/tmp/app-debug.ap
k"
10:21:51 V/ddms: execute 'pm install -r -t "/data/local/tmp/app-debug.apk"' on 'emulator-5554' : EOF hit. Read: -1
10:21:51 V/ddms: execute: returning
10:21:51 V/ddms: execute: running Emulator cal/tmp/app-debug.apk"
10:21:52 V/ddms: execute 'rm "/data/local/tmp/app-debug.apk' on 'emulator-5554' : EOF hit. Read: -1
10:21:52 V/ddms: execute: returning
Installed on 1 device.

BUILD SUCCESSFUL in 9s
27 actionable tasks: 1 executed, 26 up-to-date
info Running /Users/unk/Library/Android/sdk/platform-tools/adb -s emulator-5554
reverse tcp:8081 tcp:8081
info Starting the app on emulator-5554 (/Users/unk/Library/Android/sdk/platform-
tools/adb -s emulator-5554 shell am start -n com.hellorn/com.hellorn.MainActivity...
Starting: Intent { cmp=com.hellorn/.MainActivity }
→ helloRN
```



// 아이폰 테스트

- react-native-cli 명령으로 에뮬레이터에 앱을 설치/실행

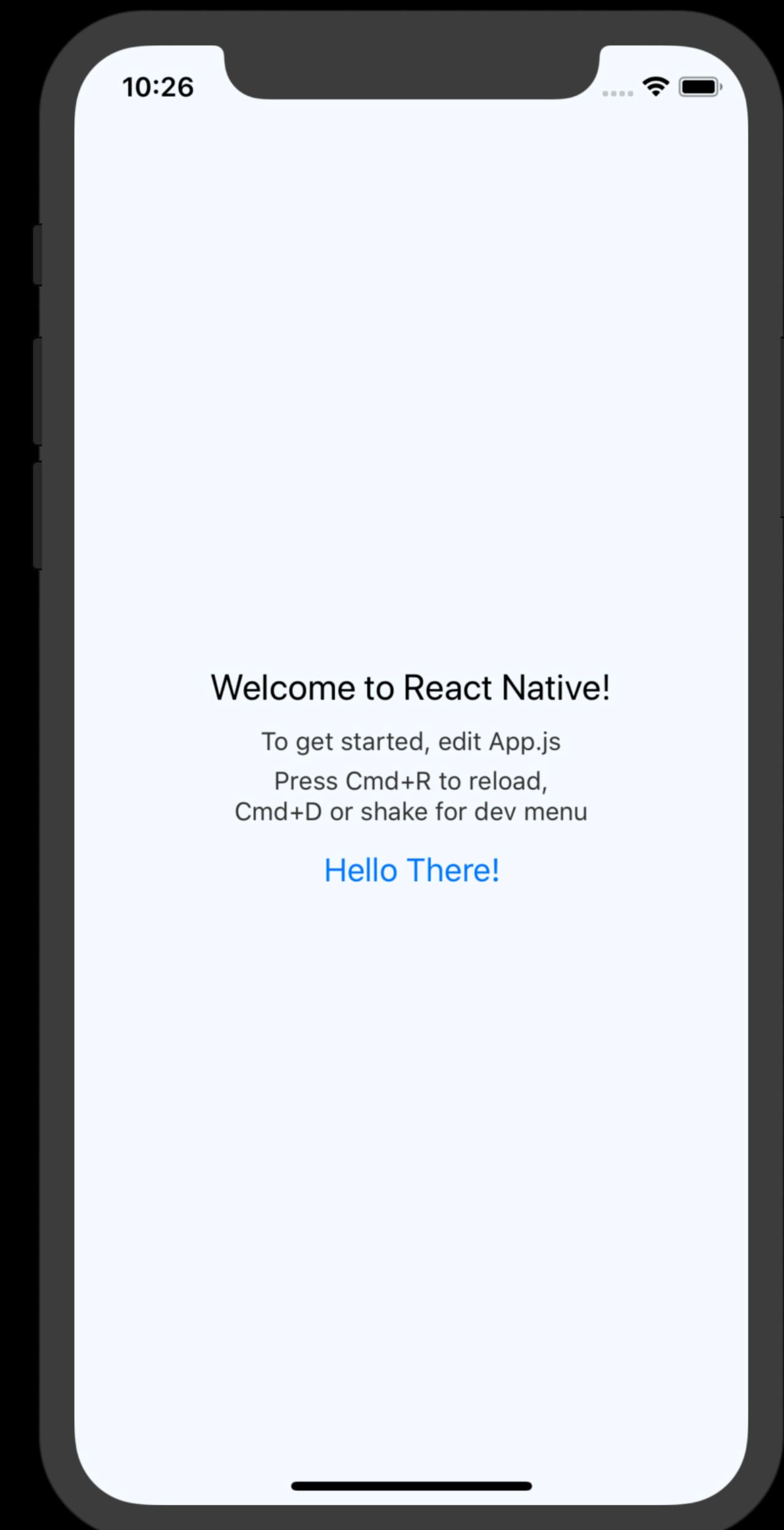
```
➔ ~ react-native run-ios
```

// 실행 결과

```
helloRN — unk@Unkui-MBP-2 — -zsh — 80x24
node < node /usr/local/Cell...
~/tests/helloRN

info + [[ -n '' ]]
info + case "$CONFIGURATION" in
info + [[ iphonesimulator == *simulator ]]
+ [[ -n '' ]]
info + echo 'Skipping bundling in Debug for the Simulator (since the packager bundles for you). Use the FORCE_BUNDLING flag to change this behavior.'
Skipping bundling in Debug for the Simulator (since the packager bundles for you). Use the FORCE_BUNDLING flag to change this behavior.
Metro Bundler
react-native-cli
info + exit 0
info
info ** BUILD SUCCEEDED **

info Installing build/helloRN/Build/Products/Debug-iphonesimulator/helloRN.app
info Launching org.reactjs.native.example.helloRN
org.reactjs.native.example.helloRN: 33082
→ helloRN
```



// 코드 분석

// App.js 요약

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

const instructions = Platform.select({...});

type Props = {};

export default class App extends Component<Props> {...}

const styles = StyleSheet.create({...});
```

// 클래스 선언

```
export default class App extends Component<Props> {  
  render() {...}  
}
```

- ES6의 클래스 문법
- 이 앱은 간단한 한 페이지로 구성되어 있음
- 하나의 클래스가 하나의 페이지를 구성하고 있음

```
// export
```

```
class App extends Component<Props> {...}  
export default App;
```

- 클래스 선언과 export 는 분리해서 작성해도 무관

// render

```
render() {  
  return (  
    <View/>  
  );  
}
```

- 클래스의 render 함수에 화면에 표시할 XML을 작성
- 단일 XML 노드, 혹은 XML 노드의 배열을 반환해야 한다
- 표시할 수 있는 컴포넌트는 공식 문서를 참조
<https://facebook.github.io/react-native/docs>

// Platform 분기

```
const instructions = Platform.select({
  ios: 'Press Cmd+R to reload,\n' + 'Cmd+D or shake for dev menu',
  android:
    'Double tap R on your keyboard to reload,\n' +
    'Shake or press menu button for dev menu',
});
```

- Platform.select() 를 이용해 OS별로 다른 내용을 할당
- ios, android 환경에 맞는 각각의 단축키 안내를 입력

// XML 구조

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <Text style={styles.instructions}>To get started, edit App.js</Text>
  <Text style={styles.instructions}>{instructions}</Text>
</View>
```

- View 컴포넌트가 전체를 감싸는 형태
- 세 종류의 Text 컴포넌트가 출력
- 세 번째 Text는 앞서 만들어진 `instructions` 변수 출력

// XML 노드의 구조

```
<Text style={styles.welcome}>Welcome to React Native!</Text>
```

- 하나의 노드는 열리는 태그와 닫히는 태그로 구성된다
- 열리는 태그와 닫히는 태그 안에 자식 요소들이 들어간다
 - children
- 자식 요소가 없는 경우 <Button/> 과 같은 형태로 줄여서 표시한다

// XML 노드의 속성 (props)

```
<Button  
  title="Hello There!"  
  onPress={() => alert( text: 'Nice 2 meet u!' )}  
/>
```

- 열리는 태그쪽에 속성을 입력할 수 있다
- 속성은 이름-값 쌍으로 이루어진다
- 값에 중괄호 {} 를 이용해 변수나 함수, 수식을 입력할 수 있다

// 스타일시트

```
const styles = StyleSheet.create({
  container: {justifyContent: 'center'...},
  welcome: {textAlign: 'center'...},
  instructions: {textAlign: 'center'...},
});
```

- StyleSheet.create 를 이용해 생성
- JSON 형태로 입력

// 스타일시트 규칙

```
container: {  
    flex: 1,  
    justifyContent: 'center',  
    alignItems: 'center',  
    backgroundColor: '#F5FCFF',  
},
```

- 웹의 CSS와 비슷한 규격
- Flex Layout을 기본으로 함
- 카멜 케이스 camelCase 형태로 속성명을 할당

// 스타일시트 할당

```
const styles = StyleSheet.create({
  container: {justifyContent: 'center'...},
  welcome: {textAlign: 'center'...},
  instructions: {textAlign: 'center'...},
});

<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <Text style={styles.instructions}>To get started, edit App.js</Text>
  <Text style={styles.instructions}>{instructions}</Text>
</View>
```

```
// import
```

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

- 맨 첫 부분에 import 부분을 작성
- React 는 모든 리액트 컴포넌트에서 필수로 로드
- 이 클래스에서 사용된 Component, Platform, StyleSheet, Text, View 등을 모두 import 해야 함



// 앱 제작 : 타노스 클릭

// 개요

- 앱을 실행하면 50%의 확률로 생존이 결정됨
- 참고 사이트: <http://www.didthanoskill.me>



// 프로젝트 생성

- ➔ ~ react-native init ThanosClick
- 기본으로 제공되는 App.js 의 Text 들을 지우고
우리가 표현할 내용들을 채움

// App.js

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

const result = Math.random();
let message;

if( result < 0.5 ) {
  message = '당신은 우주의 균형을 위해 희생되었습니다.';
}
else {
  message = '당신은 살아남았습니다.';
}

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ message }</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {justifyContent: 'center'...},
});
```

// Math.random()

- const 는 선언 후 변경되지 않는 상수값을 선언할 때 사용
 - Constant
- Math.random() 은 0에서 1사이의 난수를 생성함
예) 0.234523...

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

const result = Math.random();
let message;

const result = Math.random();

}

else {
    message = '당신은 살아남았습니다.';
}

type Props = {};
export default class App extends Component<Props> {
    render() {
        return (
            <View style={styles.container}>
                <Text>{ message }</Text>
            </View>
        );
    }
}

const styles = StyleSheet.create({
    container: {justifyContent: 'center'...},
});
```

// 조건 분기 - if

- let은 재할당이 가능한 변수를 선언할 때 사용
- if는 괄호() 안의 조건이 참일 경우에 실행할 코드를 작성
- else는 if의 조건이 참이 아닐 경우에 대한 코드를 작성
- 0에서 1 사이의 난수를 생성 후 0.5보다 작은지를 확인하므로 약 50%의 확률을 만들어낼 수 있음

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

const result = Math.random();

let message;

if( result < 0.5 ) {
  message = '당신은 우주의 균형을 위해 희생되었습니다.';
}
else {
  message = '당신은 살아남았습니다.';
}
```

```
type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ message }</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {justifyContent: 'center'...},
});
```

// 결과 출력

- Text 컴포넌트 안에
변수 `message`를 출력

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

const result = Math.random();
let message;

if( result < 0.5 ) {
  message = '당신은 우주의 균형을 위해 희생되었습니다.';
}
else {
  message = '당신은 살아남았습니다.';
}

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ message }</Text>
      </View>
    );
  }
}

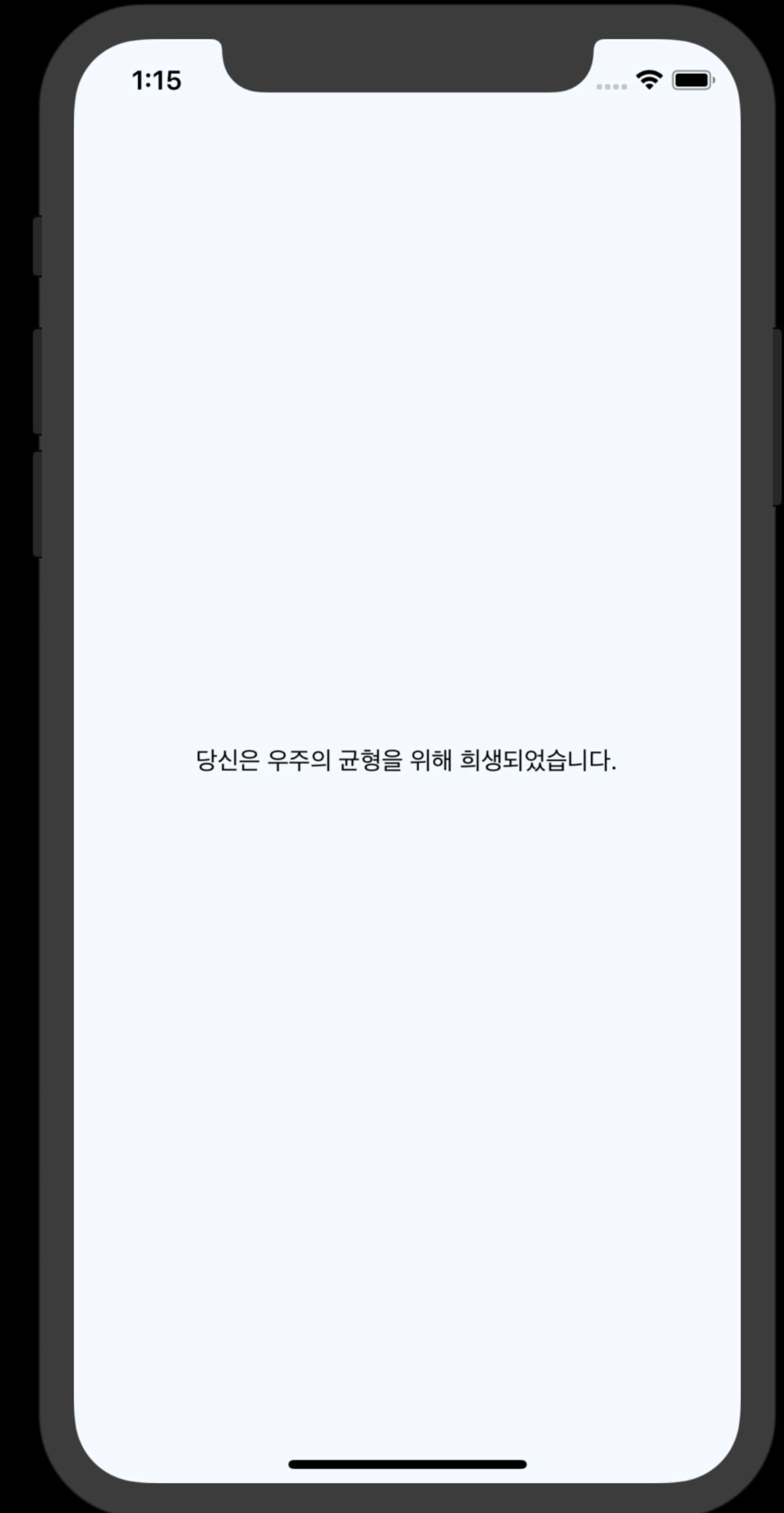
const styles = StyleSheet.create({
  container: {justifyContent: 'center'...},
});
```

// 실행

- 앱을 실행하면 50%의 확률로 생존이 결정됨
- 새로고침 할 때마다 새로운 결과

* 새로고침 단축키

- iOS : Command + R
- Android : RR



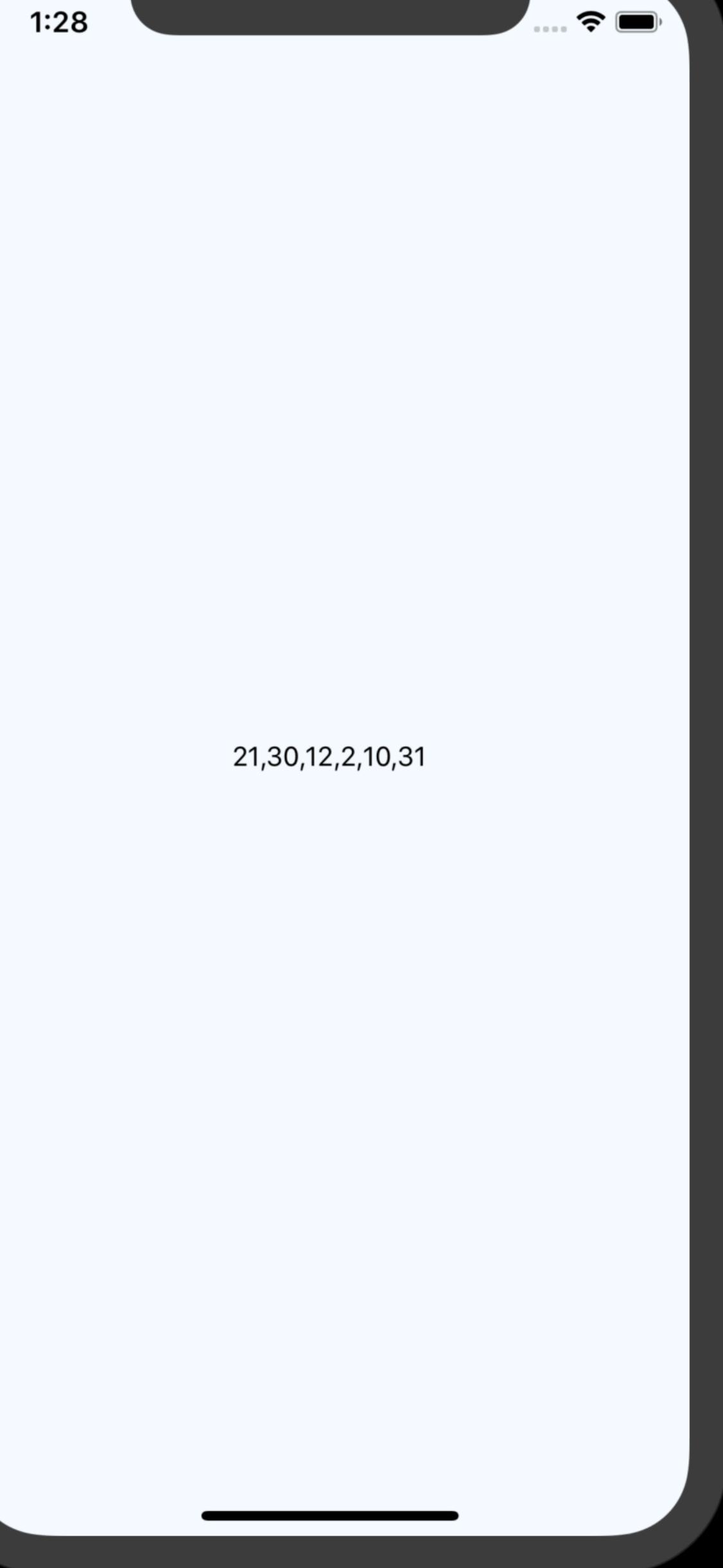
// 더 생각해 볼 문제

- 왜 확률은 정확히 50%가 아니고 약 50%일까?
힌트 : MDN에서 Math.random() 의 동작 방식에 대해서 찾아보세요
- 참고 사이트와 같이 한번 결정된 결과가 변경되지 않으려면 어떻게 해야 할까?
힌트 : 결과를 매번 새롭게 생성하지 않고 저장시킬 방법을 찾아보세요

// 앱 제작 : 로또 번호 생성기

// 개요

- 1~45의 숫자 중 6개를 임의로 골라 보여줌



21,30,12,2,10,31

// 프로젝트 생성

- ➔ ~ react-native init LottoGenerator
- 자바스크립트 라이브러리 중 underscore를 사용함. yarn 으로 추가
 - ➔ ~ cd LottoGenerator
 - ➔ ~ yarn add underscore
- 기본으로 제공되는 App.js 의 Text 들을 지우고 우리가 표현할 내용들을 채움

// App.js

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

let numbers = [];
_.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
numbers = _.shuffle( numbers );
numbers.length = 6;

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ numbers.join(',') }</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({...});
```

// import

- yarn 으로 설치한 underscore 를 import
- moment.js, decimal.js 등 DOM에 직접 관여하지 않고 데이터를 다루는 라이브러리는 무엇이든 로드 가능
- npm의 풍부한 라이브러리 자원들을 접목해 개발할 수 있음

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

```
import _ from 'underscore';
```

```
let numbers = [];
_.times( 45, iteratee: n => numbers.push( n + 1 ) );
numbers = _.shuffle( numbers );
numbers.length = 6;
```

```
type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ numbers.join(',') }</Text>
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({...});
```

// `_times`

- numbers 변수를 선언하고 빈 배열을 할당
- `_times` 는 입력한 횟수만큼 반복 실행함
참고: <https://underscorejs.org/#times>
- `push()` 명령으로 배열에 값을 추가함.
45회 반복하며 `n+1` 을 추가

배열은 [1, 2, 3, 4, ..., 42, 43, 44, 45]
와 같은 형태가 됨

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

let numbers = [];
_.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
```

```
numbers = _.shuffle( numbers );
numbers.length = 6;

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ numbers.join(',') }</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({});
```

// .shuffle

- .shuffle은 배열에 있는 값을 뒤섞어 반환함
- 뒤섞은 배열은 원본인 numbers를 다시 덮어씀

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

let numbers = [];
_.times( n: 45, iteratee: n => numbers.push( n + 1 ) );

numbers = _.shuffle( numbers );

numbers.length = 6;

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ numbers.join(',') }</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({...});
```

// length

- 배열의 .length 는 배열의 길이 값을 반환함
- .length 에 값을 설정할 경우 설정한 길이만큼만 배열을 사용하고 이후 값은 잘라냄
- shuffle로 뒤섞은 배열의 맨 앞 6개만 사용함

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

let numbers = [];
_.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
numbers = _.shuffle( numbers );
```

numbers.length = 6;

```
type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ numbers.join(',') }</Text>
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({...});
```

// 결과 출력

- `.join()` 은 배열을 하나의 문자열로 합친 결과를 반환
- 이어붙이는 문자로 `` 를 입력
[1, 2, 3, 4, 5, 6] 이라는 배열은 "1,2,3,4,5,6" 이라는 문자열로 합쳐짐

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

let numbers = [];
_.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
numbers = _.shuffle( numbers );
numbers.length = 6;

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (
      <View style={styles.container}>
        <Text>{ numbers.join(',') }</Text>
      </View>
    );
  }
}

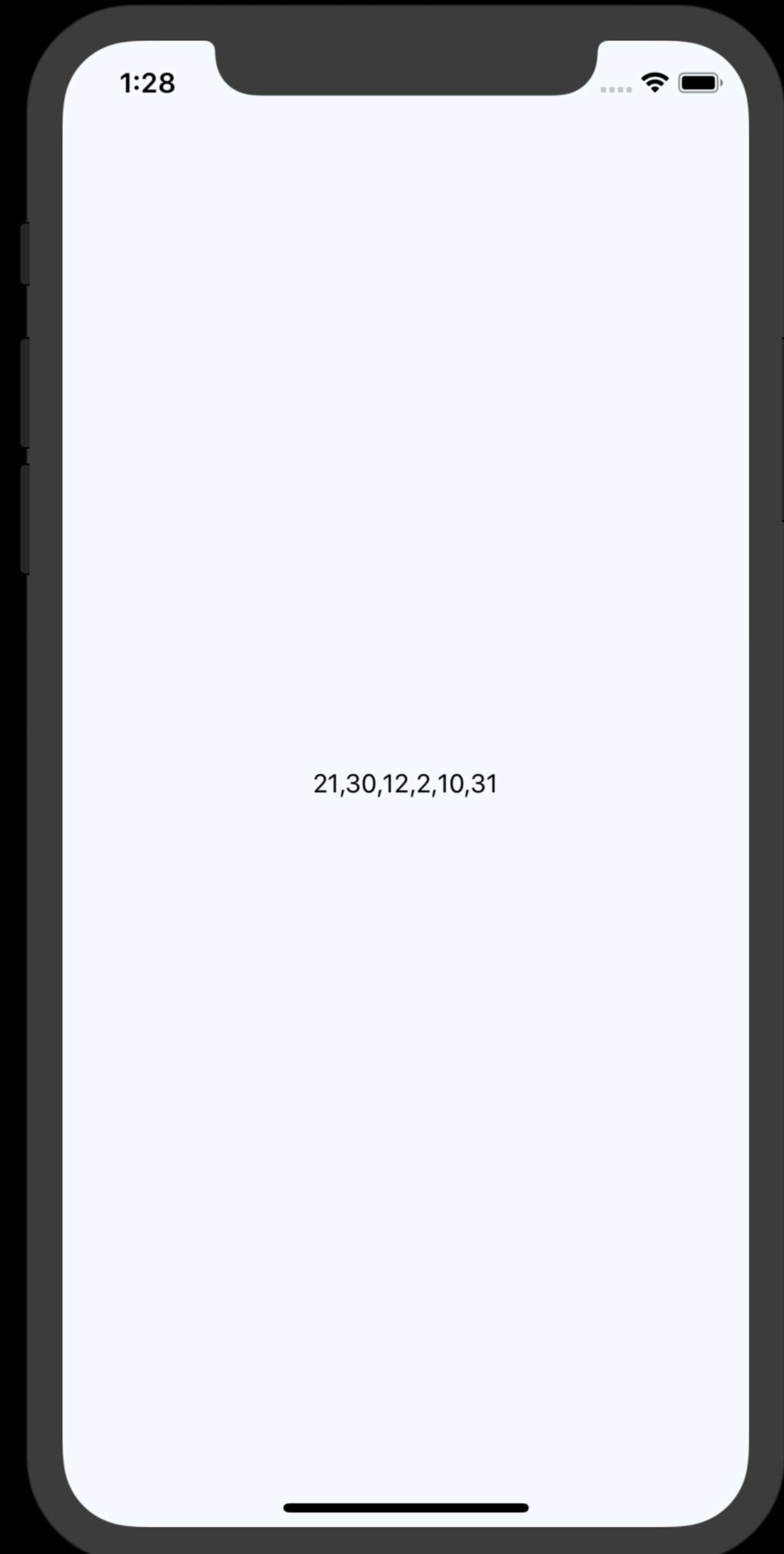
const styles = StyleSheet.create({...});
```

// 실행

- 1~45의 숫자 중 6개를 임의로 골라 보여줌
- 새로고침 할 때마다 새로운 결과

* 새로고침 단축키
- iOS : Command + R
- Android : RR

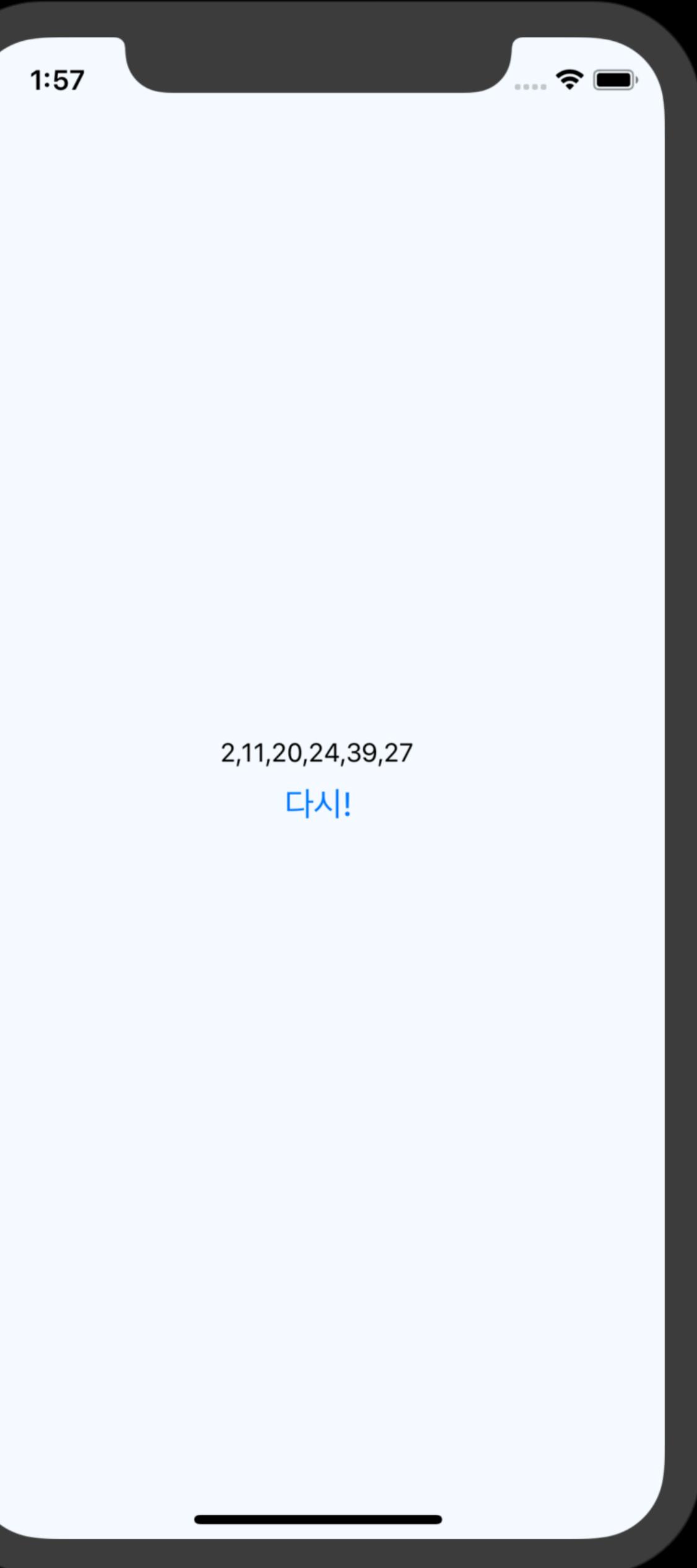
21,30,12,2,10,31



// 앱 제작 : 로또 번호 (재)생성기

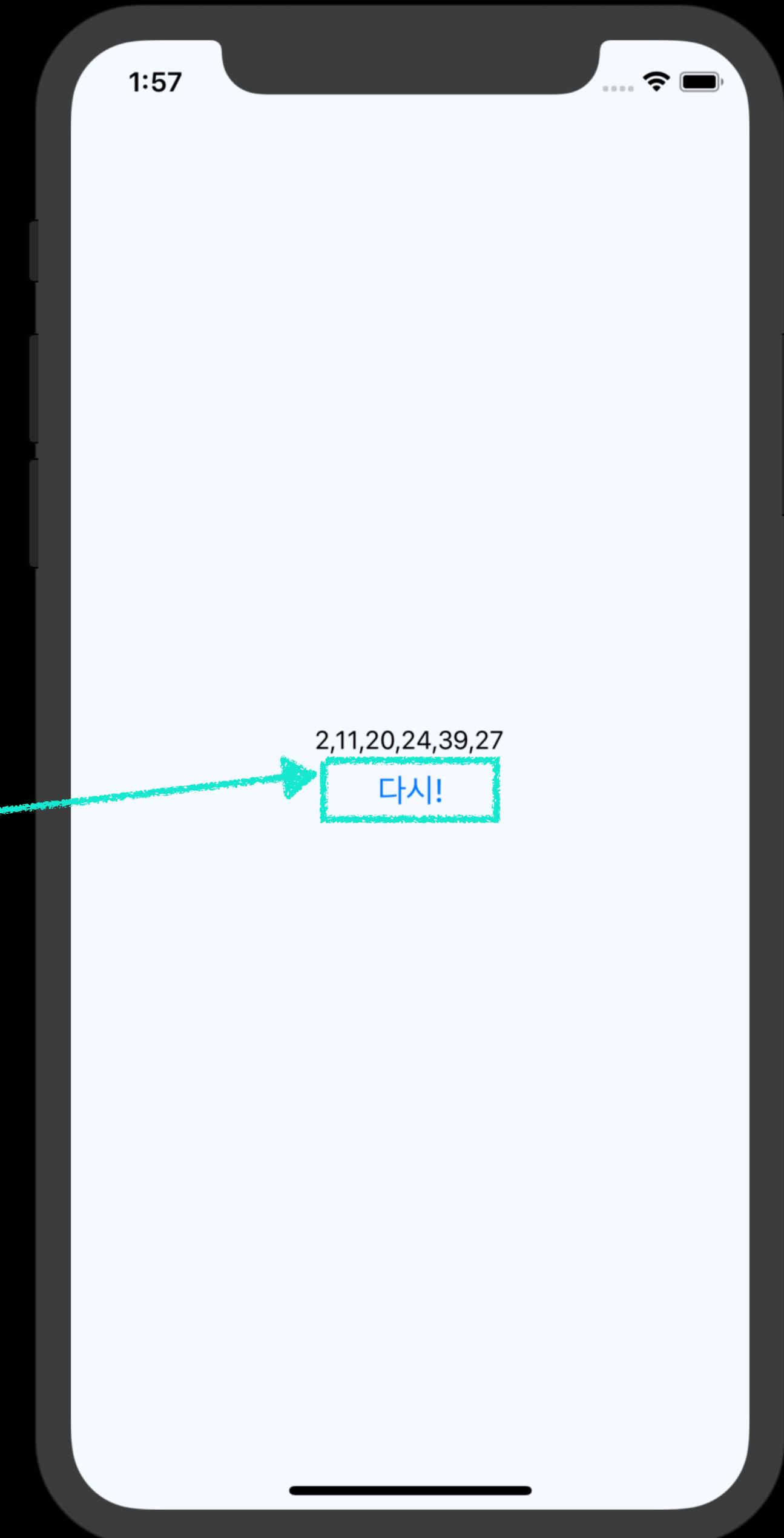
// 개요

- 1~45의 숫자 중 6개를 임의로 골라 보여줌
- "다시!" 버튼을 누르면 다시 생성함



// 기존 앱의 문제점

- 한번 생성된 결과는 앱을 재시작 해야만 갱신됨
- 번호를 다시 섞어주는 버튼을 추가해서 문제 해결



// state

- 앱이 실행된 이후에 변경될 수 있는 값들을 "상태"라고 한다
- 하나의 클래스는 하나의 state 객체를 가질 수 있다
- state 객체의 상태 변경은 리액트가 자동으로 추적하며, 자동으로 render에 반영된다.
- state 객체는 읽기 전용으로, 값을 변경할 땐 반드시 setState() 함수를 사용한다.

// state의 초기화

```
class Container extends Component {  
  state = {  
    key: 'value'  
  }  
}
```

- 클래스가 선언될 때 한번만 초기화 될 수 있다.
- 클래스의 생성자 함수 안에서도 가능하다.
- 초기화 이후에는 절대 state 를 직접 변경해선 안된다.

this.state.key = 'new value' (안됨)

// render에 state의 값을 반영

```
<Text>{ this.state.key }</Text>
```

- 중괄호 {} 를 이용해 state 내부의 값을 표시할 수 있다.

// setState()

```
this.setState( state: { key: 'new value' } );
```

- setState 함수를 이용해 state 를 업데이트 한다.
- 함수 인자로 새 값을 담은 오브젝트를 전달하면, 변경된 값을 알아서 비교하고 업데이트 한다.
- 업데이트 된 state 는 render 에 자동으로 반영된다.

// 프로젝트 생성

- react-native init LottoRegenerator
- 자바스크립트 라이브러리 중 underscore를 사용함. yarn 으로 추가

```
→ ~ cd LottoGenerator  
→ ~ yarn add underscore
```

- 기본으로 제공되는 App.js 의 Text 들을 지우고 우리가 표현할 내용들을 채움

// App.js

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

type Props = {};
export default class App extends Component<Props> {
  state = {
    numbers: [],
  };
  generate = () => {
    let numbers = [];
    _.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
    numbers = _.shuffle( numbers );
    numbers.length = 6;
    this.setState( state: {numbers} );
  };
  render() {
    return (
      <View style={styles.container}>
        <Text>{ this.state.numbers.join(',') }</Text>
        <Button title={'다시!'} onPress={this.generate}></Button>
      </View>
    );
  }
  componentDidMount() {
    this.generate();
  }
}

const styles = StyleSheet.create({...});
```

// state의 초기화

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

type Props = {};
export default class App extends Component<Props> {
```

```
state = {
  numbers: [],
};
```

```
generate = () => {
  let numbers = [];
  _.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
  numbers = _.shuffle( numbers );
  numbers.length = 6;
  this.setState( state: {numbers} );
}
render() {
  return (
    <View style={styles.container}>
      <Text>{ this.state.numbers.join(',') }</Text>
      <Button title={'다시!'} onPress={this.generate}/>
    </View>
  );
}
componentDidMount() {
  this.generate();
}

const styles = StyleSheet.create({...});
```

- numbers 배열을 state 아래에 배치하고 기본값으로 빈 배열을 할당

// render에 state 반영

- 기존과 동일하게 join() 을 사용
- number의 위치가
this.state.numbers 로 변경되었음

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

type Props = {};
export default class App extends Component<Props> {
  state = {
    numbers: [],
  };
  generate = () => {
    let numbers = [];
    _.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
    numbers = _.shuffle( numbers );
    numbers.length = 6;
    this.setState( state: {numbers} );
  };
  render() {
    return (
      <View style={styles.container}>
        <Text>{ this.state.numbers.join(',') }</Text>
        <Button title={'다시!'} onPress={this.generate}/>
      </View>
    );
  }
  componentDidMount() {
    this.generate();
  }
}

const styles = StyleSheet.create({...});
```

// generate 함수

- 기존의 `_times`, `_shuffle` 등의 로직을 그대로 활용
- 마지막에 `setState` 를 이용해 새로 생성된 배열을 state에 업데이트

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

type Props = {};
export default class App extends Component<Props> {
  state = {
    numbers: [],
  };

  generate = () => {
    let numbers = [];
    _.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
    numbers = _.shuffle( numbers );
    numbers.length = 6;
    this.setState( state: {numbers} );
  };

  render() {
    return (
      <View style={styles.container}>
        <Text>{ this.state.numbers.join(',') }</Text>
        <Button title={'다시!'} onPress={this.generate}/>
      </View>
    );
  }
  componentDidMount() {
    this.generate();
  }
}

const styles = StyleSheet.create({...});
```

// onPress

- Button 컴포넌트의 onPress 이벤트에 generate 함수를 할당
- 버튼을 누를 때마다 새로운 배열이 생성되고 뒤섞여서 state에 반영됨

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

type Props = {};
export default class App extends Component<Props> {
  state = {
    numbers: [],
  };

  generate = () => {
    let numbers = [];
    _.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
    numbers = _.shuffle( numbers );
    numbers.length = 6;
    this.setState( state: {numbers} );
  };

  render() {
    return (
      <View style={styles.container}>
        <Text>{ this.state.numbers.join(',') }</Text>
        <Button title={'다시!'} onPress={this.generate}>
          </View>
        );
      }
      componentDidMount() {
        this.generate();
      }
    }

const styles = StyleSheet.create({...});
```



The diagram illustrates the flow of data from the `generate` function to the `onPress` event of the `Button`. It consists of two blue boxes connected by an arrow. The top box contains the `generate` function code, which generates a shuffled array of numbers and updates the component's state. The bottom box contains the `onPress` prop of the `Button`, which calls the `generate` function when the button is pressed.

// componentDidMount

- 리액트 컴포넌트의 라이프 사이클 함수 중 하나인 `componentDidMount` 를 사용
-> 라이프 사이클은 다음 시간에 자세히
- 컴포넌트가 화면에 표시된 직후의 시점에서 실행됨
- 최초 state의 number가 빈 배열이므로, 화면에 표시된 직후 시점에 `generate()` 를 실행해 화면에 표시할 값을 할당해줌

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

type Props = {};
export default class App extends Component<Props> {
  state = {
    numbers: [],
  };
  generate = () => {
    let numbers = [];
    _.times( n: 45, iteratee: n => numbers.push( n + 1 ) );
    numbers = _.shuffle( numbers );
    numbers.length = 6;
    this.setState( state: {numbers} );
  };
  render() {
    return (
      <View style={styles.container}>
        <Text>{ this.state.numbers.join(',') }</Text>
        <Button title={'다시!'} onPress={this.generate}></Button>
      </View>
    );
  }
}

const styles = StyleSheet.create({...});
```

```
componentDidMount() {
  this.generate();
}
```

// 실행

- 1~45의 숫자 중 6개를 임의로 골라 보여줌
- "다시!" 버튼을 누르면 다시 생성함

