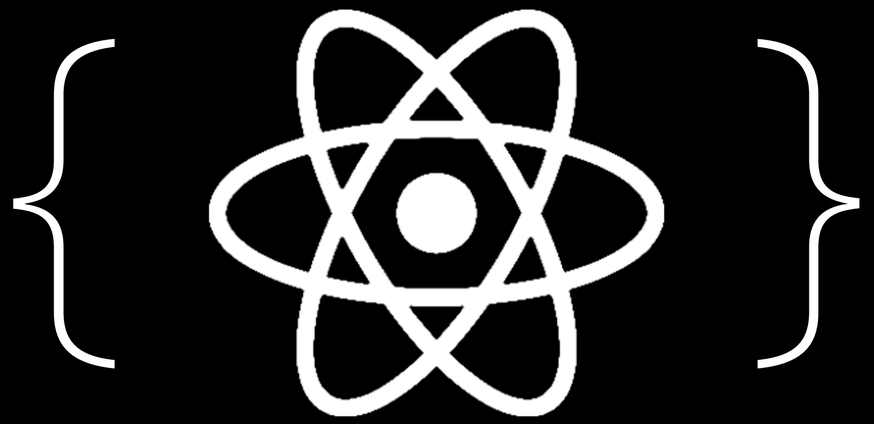
>_Q



React Native

#2

김나람 / 스튜디오 그로테스큐

// EXPORT

// Export

- https://mdn.io/export
- ES6 표준
- 필요한 작은 단위로 스크립트를 쪼개어 작성할 때 사용
- export 한 내용은 다른 파일에서 import 해서 사용할 수 있음
- 변수, 상수, 함수, 클래스 등 모든 자바스크립트 객체는 export 할 수 있음

// Import

- https://mdn.io/import
- ES6 표준
- export 처리가 된 모듈들을 읽어올 때 사용
- npm 모듈의 경우 모듈 이름만 명시
  예) import moment from 'moment'
- 프로젝트 내에 있는 모듈의 경우 상대 경로로 표시
  예) import utils from './utils'
- 확장명이 js인 모듈일 경우 확장명은 생략함

// Named Export

```
// 먼저 선언한 함수 내보내기
export { myFunction };

// 상수 내보내기
export const foo = Math.sqrt(2);

// 가져오기
import { myFunction } from './module-name'
import { foo } from './module-name'
```

// Named Export

- 한 파일 내에서 여러개의 값을 내보낼 수 있음
- 가져올 때에도 내보내기에 사용한 이름을 그대로 사용
  as 문법을 이용해서 강제 변경도 가능

// Default Export

```
export default function() {}
// 혹은
export default class {}

// 가져올 때엔
import functionName from './module-name'
import MyClass from './file-name'
```

// Default Export

- 하나의 모듈 안에서 하나만 내보내기 가능
- 가져올 때에 이름은 마음대로 지을 수 있음
  일반적으로는 모듈 이름과 동일하게 짓는 것이 관례
- as 로 새로운 이름을 지정할 수 있음

// Stylesheet Export

```
// CommonStyle.js

export default StyleSheet.create( {
  safetyMarginTop: { ... }
  container: { ... }
} );
```

- 오브젝트 형태로 기술

// Stylesheet Export

```
// CommonStyle.js

export default StyleSheet.create( {
  safetyMarginTop: {
    marginTop: Platform.OS === 'ios' ? 0 || 15,
  }
  container: {
    flex: 1,
    ...
  }
} );
```

// Stylesheet Import

```
import CommonStyle from './styles/CommonStyle'

<View style={ [ CommonStyle.container, CommonStyle.safetyMarginTop ] }>

</View>
```

- 배열을 이용해 복수개의 스타일을 지정할 수 있다

// Class Export

```
// Heading1.js

export default class Heading1 extends Component {
  render() { ... }
}
```

// Class Import

```
import H1 from './components/Heading1'

<View style={ ... }>
  <H1>Headline</H1>
</View>
```

- 배열을 이용해 복수개의 스타일을 지정할 수 있다

// Fucntion Export

```
// trimRight.js

export default text => {
  return ...
}
```

// Function Import

```
import trimRight from './utils/trimRight'

<View style={ ... }>
  <TextInput value={ trimRight( this.state.value ) }/>
</View>
```

// Contants Export

```
// ENV.js

export default {
  API_KEY: "qwer-1234-asdf-2345",
  API_SECRET: "rewq-4321-fdsa-5432",
}
```

// Constants Import

```
import ENV from './config/ENV'

api.init( {
  apiKey: ENV.API_KEY,
  secret: ENV.API_SECRET,
} );
```

// BOILERPLATE 활용하기

// 보일러플레이트

- 사전적 의미로는 "표준 문안"
- 위키피디아의 Boilerplate Code 에 대한 설명으로는
  - 최소한의 변경으로 재사용할 수 있는 것
  - 적은 수정만으로 여러 곳에 활용 가능한 코드, 문구
  - 각종 문서에서 반복적으로 인용되는 문서의 한 부분
- 자주 반복되는 초기 세팅을 모아놓은 템플릿도 포함됨

// Codelab Boilerplate for React Native CLI

- https://github.com/GrotesQ/Codelab-Boilerplate-For-RN-CLI
- native-base, react-navigation
- CocoaPod 기반 ios 세팅
- 2개의 페이지를 포함하고 있는 Stack Navigator 등

// Codelab Boilerplate for Expo

- https://github.com/GrotesQ/Codelab-Boilerplate-For-Expo
- native-base, react-navigation
- Font async load에 대한 처리 포함
- 2개의 페이지를 포함하고 있는 Stack Navigator 등

// 함수형 컴포넌트

// 함수형 컴포넌트

- 클래스 형태가 아닌 단일 함수로 구성되는 컴포넌트
- 커스텀 컴포넌트를 만드는 가장 단순한 형태
- props 는 사용 가능
- state와 lifecycle 함수들은 사용 불가

// 함수형 컴포넌트

- 단순한 형태인 만큼 "데이터의 출력"이라는 명확한 역할
- 데이터를 제어하는 컴포넌트는 클래스 컴포넌트로 작성하고, 데이터를 출력하는 컴포넌트는 함수형 컴포넌트로 작성해 역할을 명확히 분리할 수 있다
- 리액트 개발팀에서도 권장하는 형태라 함수형 컴포넌트에서 이용할 수 있는 hook 등이 추가되는 중

// 함수형 컴포넌트 선언

```
const H1 = props => {
    return (
        <Text style={ ... }>{props.value}</Text>
    );
}
```

// 함수형 컴포넌트 사용

```
<View>
  <H1 value="함수형 컴포넌트 사용 예"/>
</View>
```

```
const H1 = props => {
    return (
        <Text style={ ... }>{props.value}</Text>
    );
}
```

```
<View>
  <H1 value="함수형 컴포넌트 사용 예"/>
</View>
```

// RENDER에서 배열 처리

// NodeList (HTML)

- HTML 노드가 리스트 형태를 이루는 것
- document.querySelectorAll 등으로 탐색해 얻을 수 있음
- forEach 등을 통해서 처리함

// NodeList

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

- 여기에서 의 목록이 NodeList에 해당

// React의 경우

```
<View>
  {
    [
      <Text>Item 1</Text>,
      <Text>Item 2</Text>,
      <Text>Item 3</Text>,
      <Text>Item 4</Text>,
      <Text>Item 5</Text>,
    ]
  }
</View>
```

- Node로 이루어진 배열을 화면에 출력할 수 있음

// map 활용

```
<View>
  { list.map( item => <Text>{ item }</Text> ) }
</View>
```

- https://mdn.io/Array/map
- map() 의 결과는 배열이므로 Node로 이루어진 배열을 쉽게 만들 수 있음

// (unique) key

- 배열을 이용해 리스트를 표현할 경우 각각의 리스트 아이템에 고유성을 부여하기 위해 key 할당 필요
  - 배열 내의 아이템에 id 값 등을 이용할 수 있다면 가장 권장
  - map, forEach 함수에서 제공하는 index를 이용할 수 있으나 index는 고유성을 보장하는 값이 아니므로 권장하지 않음
  - new Date().getTime() 과 index의 조합 정도가 차선책
  - 몇몇 컴포넌트의 경우 String Type의 key만 허용함

// key 예시 (1)

```
<View>
  { list.map( item => (
    <Text key={ item.id }>{ item }</Text>
  ) }
</View>
```

- 데이터 내부에 고유키로 사용할 수 있는 id 등을 직접 이용

// key 예시 (2)

```
<View>
  { list.map( ( item, index ) => (
    <Text key={ new Date().getTime().toString() + index >{ item }</Text>
  ) ) }
</View>
```

- new Date().getTime() 을 toString() 한 후 map의 index를 더해서 고유키를 생성함

// AJAX

// AJAX

- Asynchronous JavaScript and XML
  비동기로 XML을 다루는 자바스크립트
- 서버측의 응답이 끝난 후 프론트엔드단에서 서버측으로 다시 데이터 요청을 처리하기 위한 기술
- Gmail에 적극적으로 도입되며 큰 반향
- 초기에는 XML이 쓰였고 지금은 주로 JSON이 사용되지만 용어 자체는 AJAX로 고정됨

// Axios

- https://github.com/axios/axios
- 자바스크립트에서 가장 왕성하게 사용되는 AJAX 라이브러리
- node.js 서버와 프론트엔드 자바스크립트 엔진에서 모두 동작
- Promise 패턴으로 사용할 수 있음

// 사용 예시

```
import axios from 'axios';

axios.get('/user?ID=12345')
  .then(function (response) {
    // 통신 성공시
    console.log(response);
  })
  .catch(function (error) {
    // 오류 발생시
    console.log(error);
  })
  .then(function () {
    // 최종적으로 항상 실행 (옵션)
  });
```

// async / await

- https://mdn.io/async / https://mdn.io/await
- ECMA Script 2017 표준
- Promise의 메소드 체인 형태를 개선하기 위한 제안
- 반드시 async function 안에서만 await를 사용할 수 있다는 제약 사항 있음

// 사용 예시

```
// Promise 패턴
axios.get('/user?ID=12345')
  .then( response => {
    console.log( response );
  } );

// await 패턴
async getUser() {
  const response = await axios.get('/user?ID=12345');
  console.log( response );
}
```

// catch 처리

```
async getUser() {
  try {
    const response = await axios.get('/user?ID=12345');
    const profile = await axios.get('/profile?ID=12345');
    console.log( response, profile );
  }
  catch( error ) {
    console.log( error );
  }
}
```

// TEXT INPUT

// Text Input (and Form Elements)

- 리액트는 state -> view 방향의 단방향 데이터 연결 형태이므로 사용자 입력을 받는 폼 요소의 경우 문제 발생
- 폼 요소의 value에 state의 값을 연결하면 사용자가 입력을 무시하고 state의 값이 다시 value에 할당되며 사용자의 관점에서는 read only 요소인 것 처럼 화면 변화가 없음
- update / change 이벤트를 통해 setState 를 해줘야 함
- https://facebook.github.io/react-native/docs/textinput.html

// 사용 예시

```
state = {
  text: '',
}

onChangeText = text => {
  this.setState( { text } );
}

// render
<TextInput onChangeText={ this.onChangeText }
           value={ this.state.text }/>
```