

React,vue を使うために知っておきたい Javascript の基本

1. Javascript とは何？

元々は web ブラウザ上で複雑な動きを実装するためのもの
今は、サーバーサイド、アプリ、AR,VR など

モダン javascript

- ・ 仮想 DOM を用いるライブラリ、フレームワーク
- ・ ES2024 以降の記法を使用

2. DOM や仮想 DOM

Document Object Model

HTMLなどを解釈し木構造で表現したもの

これまでは DOM を直接操作していた

仮想 DOM

JavaScript のオブジェクトで仮想的に作られた DOM

→いきなり DOM を操作せず、js 上で仮想 DOM を操作し差分を出してから DOM に反映

3. npm や yarn 等のパッケージマネージャーの意義を知る

初期 : 今までは、1 つの js ファイルにすべての処理を記述していた

処理が複雑になるにつれてコードがカオス化

コードの再利用ができない

中期 : 細かく分けて他の js ファイルを読み込んで使っていた

コードの再利用、共通化はできるようになった

読み込み準を意識しないとエラーになる(依存関係)

何がどこから読み込まれたものかわからない

今 : npm/yarn 等のパッケージマネージャーを使用

内部では、Node.js が動いている

依存関係を勝手に解決してくれる

Import 先が明示的にわかる

世界中で公開されているパッケージをコマンド一つで利用可能

チーム内での共有も簡単に

Ex.)

Import react from “react”;

4. ECMA スクリプトとは・近代 JavaScript の転換期について

ES(ECMA Script)

JavaScript の標準規格

ES2015 で追加機能が多くあり、近代 JS の転換期といえる

ES2015 で追加された規格

・ let,const を用いた変数宣言

・ アローファンクション

・ class 構文

・ 分割代入

・ テンプレート文字列

・ スプレッド構文

・ Promiss

etc

5. モジュールバンドラー・トランスパイラ

モジュールバンドラー

複数の js(css/image) ファイルを一つにまとめるためのもの

トランスパイラ

新しい JavaScript の記法を古い基本に変換してくれる

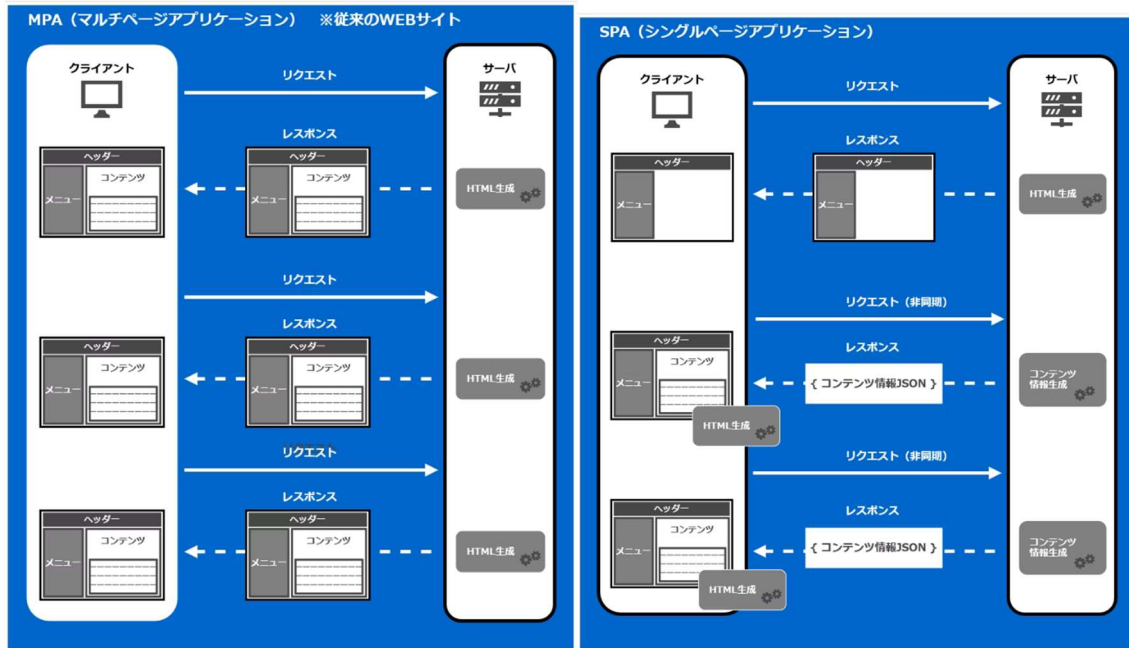
Ex) Babel, swc etc

6. SPA とは？従来のシステムとの違い

Single Page Application

モダン JavaScript が基本

HTML は一つのみで JavaScript で画面を書き換える



SPA のメリット

ページ遷移ごとのちらつきがなくなる

表示速度のアップによるユーザー体験向上(仮想 DOM)

コンポーネント分割が容易になることでの開発効率アップ

JavaScript の基本

1. const,let 等の変数宣言

var : 上書き可能 var val1="var 変数"
 val1 = "var 変数を上書き"
 再宣言可能 var val1 = "var 変数を再宣言"

let : 上書き可能
 再宣言不可

const : 上書き不可
 再宣言不可

const で定義したオブジェクトはプロパティの変更が可能

```
const val2 = {  
    name = "inagaki",  
    age:24  
}
```

val2.name= "Inagaki naoki"

const で定義したオブジェクトはプロパティの変更が可能

```
const val3 = ["dog","cat"]  
val3[0] = "bird"  
val3.push("monkey")
```

2. テンプレート文字列

従来の方法

```
const name = "inagaki"  
const message1 = "私の名前は" + name + "です。"
```

テンプレート文字列を用いた方法

```
const message2 = `私の名前は${name}です。`
```

3. アロー関数

従来の関数

```
function func1 (str){  
    return str;  
}  
  
or  
  
const func1 = function(str){  
    return str  
}
```

アロー関数

```
const func1 = (str) => {  
    return str  
}
```

```
console.log(func1("function です"))
```

変数がある時は、(str)の()を省略して str と書いてもよい

単一式の時は、{ return }を省略してかける ex.) const func1 = (str) => str

オブジェクトを返すときは、({})を使うことで一つの返却で返せる

```
const func2 = (name, age) => ({  
    a:name,  
    b:age,  
})
```

4. 分割代入

オブジェクト ver

```
const myProfile = {  
    name: "inagaki"  
    age: 24  
}  
  
const {name, age} = myProfile
```

配列 ver

```
const myProfile = ["inagaki", 24]  
const [name, age] = myProfile
```

5. デフォルト値

最初からデフォルトで値を入れる

```
const sayHello = (name = “ゲスト”) => console.log(`こんにちは${name}さん`)
```

```
const myProfile = { age: 24 }
```

6. オブジェクトの省略記法

プロパティの名前と変数の名前が同じときは片方を省略できる

省略なし	省略あり
<pre>const name = const age = const myProfile = { name: name, age: age, }</pre>	<pre>const name = const age = const myProfile = { name, age, }</pre>

7. スプレッド構文

配列の展開

```
const arr1 = [1,2]  
console.log(arr1)           [1,2]  
console.log(..arr1)         1 2  
const sumFunc = (num1, num2) => console.log(num1 + num2);  
sumFunc(arr1[0], arr1[1])    3
```

まとめる

```
const arr2 = [1, 2, 3, 4, 5]  
const [num1, num2, ...arr3] = arr2           1 2 [3,4,5]
```

配列のコピー、結合

コピー

```
const arr4 = [10, 20]  
const arr5 = [30, 40]  
const arr6 = [...arr4]           [10,20]
```

結合

```
const arr4 = [10, 20]  
const arr5 = [30, 40]  
const arr7 = [...arr4, ...arr5]   [10,20,30,40]
```

8. map や filter を使った配列の処理

従来

```
const nameArr = ["稲垣", "直輝"]
for (let index = 0; index < nameArr.length; index++){
    console.log(nameArr[index]);
}
```

map 配列.map(関数)

```
nameArr.map( (name)=>{
    return console.log(name);    "稲垣" "直輝"
})
```

map の引数の二番目は index になる

```
nameArr.map( (name, index)=> console.log(`${index + 1}` 番目は name です);
1 番目は稲垣です
2 番目は直輝です
```

filter

配列の要素を filtering して新しい配列を作る

```
const numArr = [1, 2, 3, 4, 5]
const newNumArr = numArr.filter( (num) => {
    return num % 2 ===1;    [1, 3, 5]
})
```

9. 三項演算子

ある条件？条件が true の時：条件が false の時

10. 論理演算子の本当の意味を知ろう

||, &&の本当の意味について

||：左側が truthy の時、その時点で返却する

```
x=null || "稲垣"    "稲垣"
```

&&：左が falsy の時、その時点で返却する

```
x= null || "稲垣"    null
```

React の基本

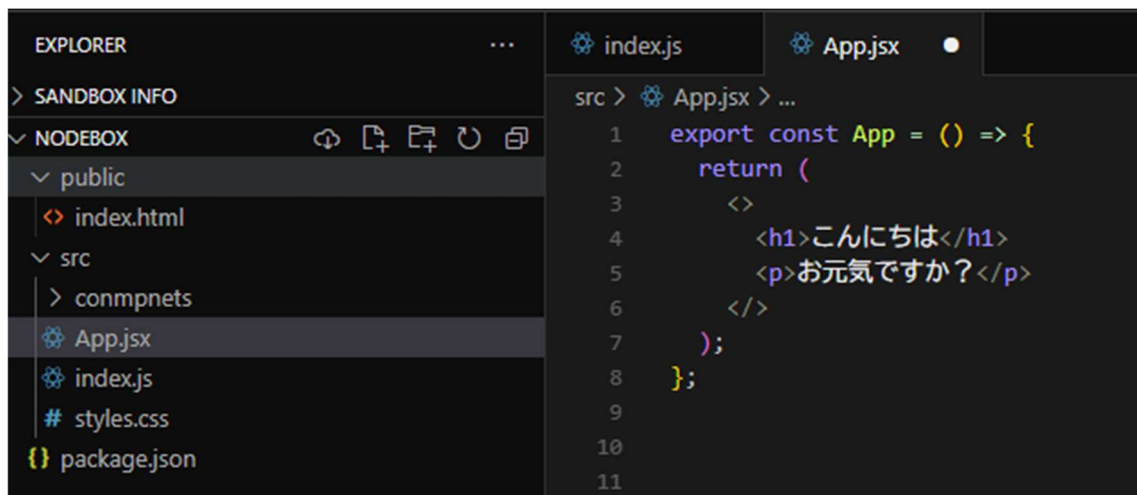
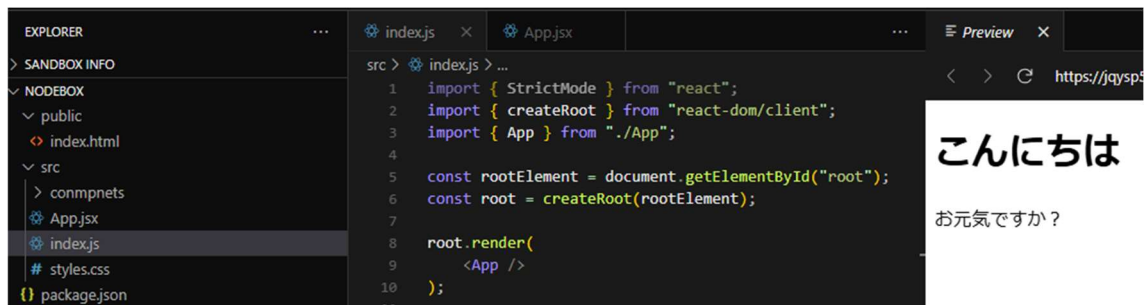
1. JSX 記法のルールを知る

2. コンポーネントの使い方を知る

再利用可能な UI の部品

アプリケーションのユーザーインターフェースを構築する基本的な単位

Ex.) 関数コンポーネント、クラスコンポーネント



3. React でのイベントやスタイルの扱いを知る

イベントの後に{}を使いその中に js の処理を書く

Ex.) `<button onClick = {“JS の記述”}> ボタン </button>`

CSS も同じようにすることで割り当てることができる

Ex.) `<h1 style = { { “css の記述” } }>`

外の{} : JavaScript

中の{} : CSS



```
src > App.jsx > ...
1  export const App = () => {
2    const onClickButton = () => alert();
3    const contentStyle = {
4      color: "blue",
5      fontSize: "18px"
6    }
7    return (
8      <>
9        <h1 style = {{ color: "red" }}>こんにちは</h1>
10       <p style = {contentStyle}>お元気ですか?</p>
11       <button onClick={onClickButton}>ボタン</button>
12     </>
13   );
14 };
```

4. Props を知る

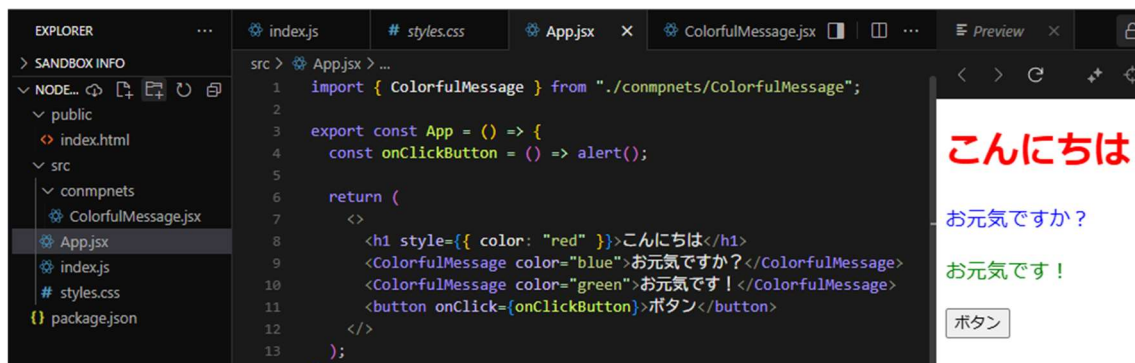
React コンポーネント間でデータをやり取りするための仕組み

Props は親から子コンポーネントに渡され、子コンポーネントで受け取って表示やロジックに使用される。



```
> SANDBOX INFO
✓ NODE...
  ✓ public
    < index.html
  ✓ src
    ✓ conmpnets
      ✨ ColorfulMessage.jsx
      ✨ App.jsx
      ✨ index.js
      # styles.css
      {} package.json

src > conmpnets > ✨ ColorfulMessage.jsx > ColorfulMessage
1  export const ColorfulMessage = (props) => {
2    const { color, children } = props;
3    const contentStyleA = {
4      color,
5      fontSize: "18px",
6    };
7
8    return <p style={contentStyleA}>{children}</p>;
9  };
10
```



```
EXPLORER
> SANDBOX INFO
✓ NODE...
  ✓ public
    < index.html
  ✓ src
    ✨ ColorfulMessage.jsx
    ✨ App.jsx
    ✨ index.js
    # styles.css
    {} package.json

src > App.jsx > ...
1  import { ColorfulMessage } from "../conmpnets/ColorfulMessage";
2
3  export const App = () => {
4    const onClickButton = () => alert();
5
6    return (
7      <>
8        <h1 style={{ color: "red" }}>こんにちは</h1>
9        <ColorfulMessage color="blue">お元気ですか？</ColorfulMessage>
10       <ColorfulMessage color="green">お元気です！</ColorfulMessage>
11       <button onClick={onClickButton}>ボタン</button>
12     </>
13   );

```

こんにちは
お元気ですか？
お元気です！
ボタン

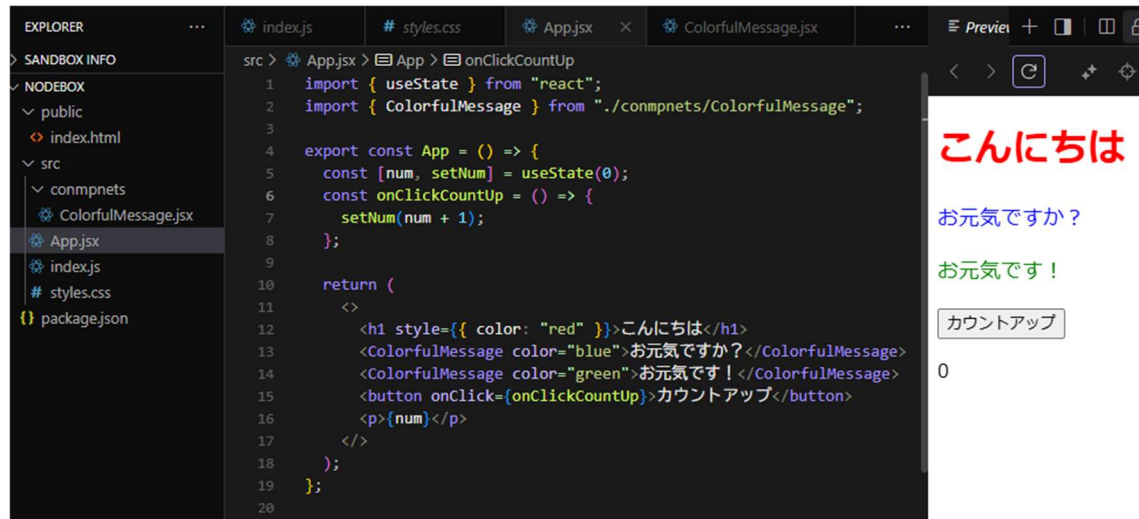
5. Stateを知る

State

react コンポーネントがその時点で保持しているデータや情報のことで、コンポーネントの状態を表す

※useState()は関数の一番上で定義する必要がある

Ex.) const [実際の値, 更新するための関数] = useState(初期値)



```
src > App.jsx > App > onClickCountUp
1 import { useState } from "react";
2 import { ColorfulMessage } from "../conmpnets/ColorfulMessage";
3
4 export const App = () => {
5   const [num, setNum] = useState(0);
6   const onClickCountUp = () => {
7     setNum(num + 1);
8   };
9
10  return (
11    <>
12      <h1 style={{ color: "red" }}>こんにちは</h1>
13      <ColorfulMessage color="blue">お元気ですか?</ColorfulMessage>
14      <ColorfulMessage color="green">お元気です!</ColorfulMessage>
15      <button onClick={onClickCountUp}>カウントアップ</button>
16      <p>{num}</p>
17    </>
18  );
19 };
20
```

※まとめて更新関数を処理するので、一回しか更新されないことがある

→今の状態に基づいて更新したいなら更新関数の引数に入れる

Ex.) setNum((prev) => prev + 1);を二つ重ねる



```
src > App.jsx > App > onClickCountUp
1 import { useState } from "react";
2 import { ColorfulMessage } from "../conmpnets/ColorfulMessage";
3
4 export const App = () => {
5   const [num, setNum] = useState(0);
6   const onClickCountUp = () => {
7     setNum((prev) => prev + 1);
8     setNum((prev) => prev + 1);
9   };
10
11  return (
12    <>
13      <h1 style={{ color: "red" }}>こんにちは</h1>
14      <ColorfulMessage color="blue">お元気ですか?</ColorfulMessage>
15      <ColorfulMessage color="green">お元気です!</ColorfulMessage>
16      <button onClick={onClickCountUp}>カウントアップ</button>
17      <p>{num}</p>
18    </>
19  );
20 };
21
```

6. 再レンダリングと副作用を知る(useEffect)

再レンダリング

React コンポーネントが再び描画され、表示が更新されること。

React では、state や props が変わるとコンポーネントが再レンダリングされる。

useEffect

react の関数コンポーネントで副作用を実行するためのフック。

Ex.) `useEffect(() => { “処理” }, [num])` num に変更があった時だけ処理を行う



```
src > App.jsx > App
1  import { useEffect, useState } from "react";
2  import { ColorfulMessage } from "../components/ColorfulMessage";
3
4  export const App = () => {
5    const [num, setNum] = useState(0);
6    const [isShowFace, setIsShowFace] = useState(false);
7    const onClickCountUp = () => {
8      setNum((prev) => prev + 1);
9    };
10   const onClickToggle = () => {
11     setIsShowFace(!isShowFace);
12   };
13
14   useEffect(() => {
15     if (num > 0) {
16       if (num % 3 === 0) {
17         isShowFace || setIsShowFace(true);
18       } else {
19         isShowFace && setIsShowFace(false);
20       }
21     }
22   }, [num]);
23
24   return (
25     <>
26       <h1 style={{ color: "red" }}>useEffect</h1>
27       <button onClick={onClickCountUp}>カウントアップ</button>
28       <p>{num}</p>
29       <button onClick={onClickToggle}>on/off</button>
30       {isShowFace && <p>^_^</p>}
31     </>
32   );
33 };
34
```

このケースでは、useEffect がない時、onClickToggle を押す度「setIsShowFace」が更新され、再レンダリングが走り if 文を毎回通っていた
それを useEffect で num の変更があるときだけ if 文を通るように制限した