ReactからVueへの転向: 思考の変化とアプローチの違い

森山 凪

2024/11/01

MNTSQ

自己紹介

森山 凪
モリヤマ ナギ

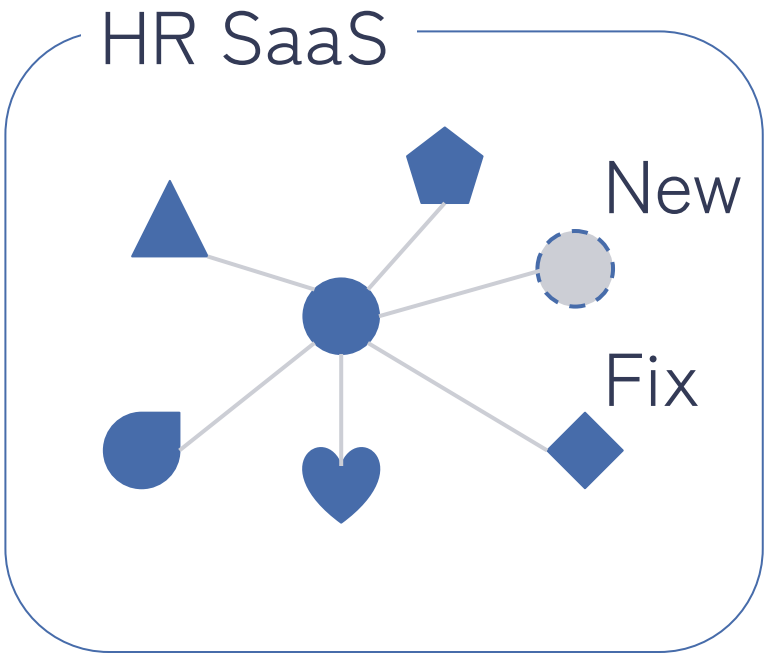
MNTSQ株式会社
フロントエンド 4年目

- 3年
- 1年
- 3ヶ月（趣味）

過去やってきたこと

自己紹介

前職はHR領域のSaaSをマイクロサービスアーキテクチャで開発



新サービスの開発方針

GraphQLからTypeScriptの型を自動生成

Storybook x ChromaticでのUIの差分検知

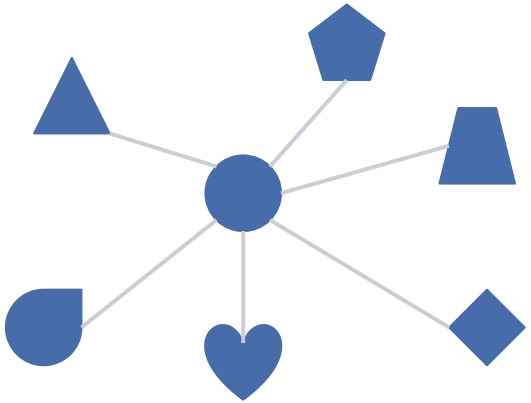
Gherkin記法でユースケースをGit管理

Cypress x CucumberでE2E

API、UI、ユースケース
変われば悲鳴をあげるFE

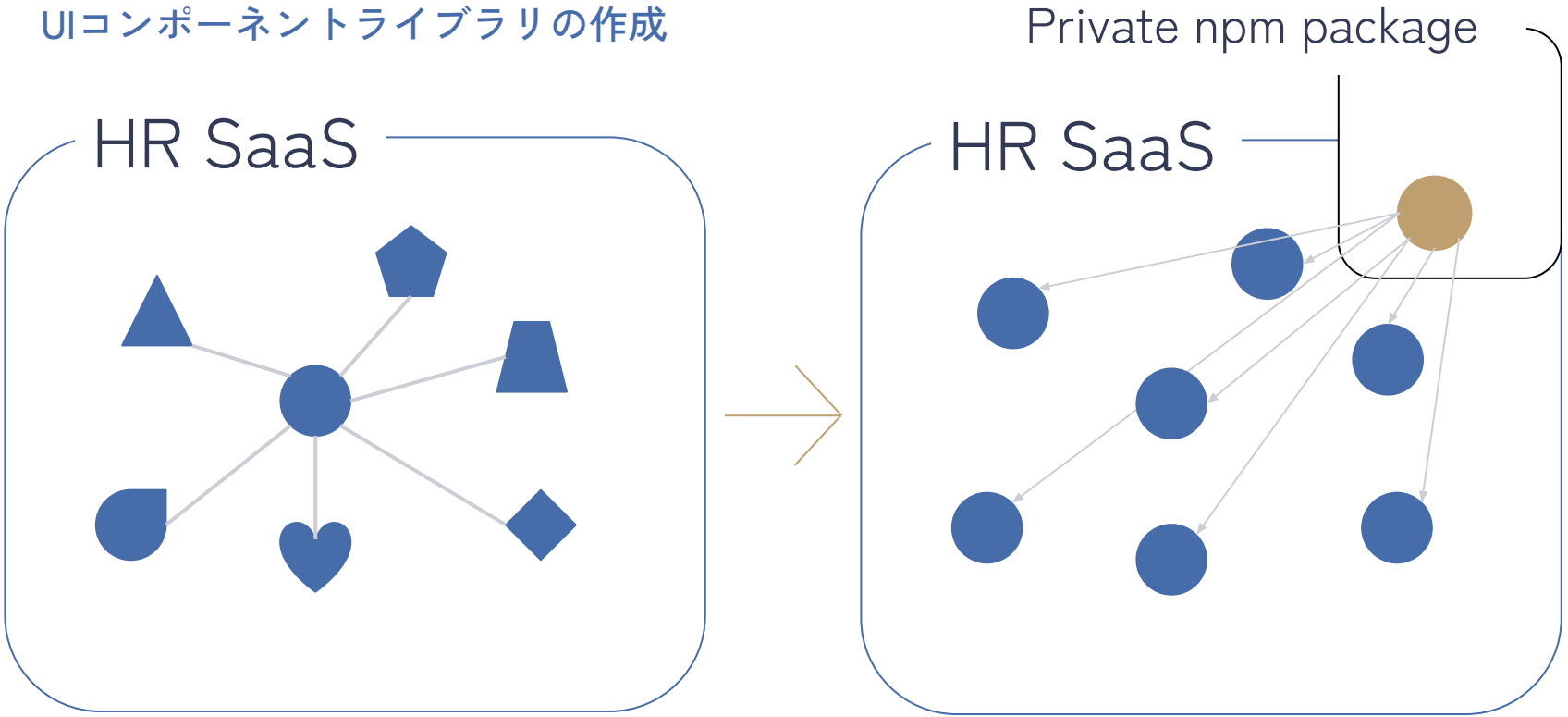
マイクロサービスそれぞれを見てみると、、、

HR SaaS



あれ、、
各サービスで微妙にデザインに違うな

UIコンポーネントライブラリの作成



ReactからVueへの転向: 思考の変化とアプローチの違い

目次

目次

ライフサイクル表現の変化

ライフサイクルの変化

React

useEffect

Vue

→ lifecycle hook

ライフサイクルの変化

React

useEffect 

Vue

→ lifecycle hook

ライフサイクルの変化

React

useEffect() =

Vue

1. update
2. mount
3. unMount

useEffect()

update
mount
unMount

どうやってuseEffectで
3つのlifecycleを表現しているのか？

useEffectのlifecycle表現

```
const Component = () => {

    useEffect( ()=>{ }, [ ] )

}
```

1. onMount
2. onUpdate
3. onUnmount

useEffectのlifecycle表現

```
const Component = () => {



    useEffect(()=>{
        // 1回目はonMount



    },[])
}
```

1. **onMount**
    - Component()実行でuseEffect()も呼ばれる。
    - useEffectの初回実行がmount
2. onUpdate
3. onUnmount

useEffectのlifecycle表現

```
const Component = () => {

    const [state1,] = useState()
    const [state2,] = useState()

    useEffect(()=>{
        // 1回目はonMount
        // 2回目以降はonUpdate


    },[state1, state2]) // 依存配列
}
```

1. onMount
    - Component()実行でuseEffect()も呼ばれる。
    - useEffectの初回実行がmount。
2. **onUpdate**
    - useEffectの第2引数が変化する度にuseEffectが再発火する。
    - state1, state2どちらが変わっても発火。
3. onUnmount

useEffectのlifecycle表現

```
const Component = () => {

    const [state1,] = useState()
    const [state2,] = useState()

    useEffect(()=>{
        // 1回目はonMount
        // 2回目以降はonUpdate

        return () => {} // onUnmount
    },[state1, state2])
}
```

1. onMount
    - Component()実行でuseEffect()も呼ばれる。
    - useEffectの初回実行がmount。
2. onUpdate
    - useEffectの第2引数が変化する度にuseEffectが再発火する。
    - state1, state2どちらが変わっても発火。
3. **onUnmount**
    - useEffectの第1引数の返り値がunmount。

なるほど！！

useEffectはムズい

useEffectムズい

lifecycleを理解している人でないと
（useEffectは）難しい

useEffectのなんのため？

useEffectの恩恵は何？

useEffectの恩恵は何？ → 機能的な凝集性が高い

useEffectの恩恵は何？ → 機能的な凝集性が高い

```
const Component = () => {
  // API コール
  useEffect(()=> {
    const response = apiCall(payload)
    setResponse(response)
  }, [payload])

  // タイマー処理
  useEffect(()=>{
    const timerId = setInterval(() => {
      // tick process
    }, 1000)
    return () => clearInterval(timerId)
  },[])
}
```

useEffectの恩恵は何？ → 機能的な凝集性が高い

```
const Component = () => {
  // API コール
  useEffect(()=> {
    const response = apiCall(payload)
    setResponse(response)
  }, [payload])

  // タイマー処理
  useEffect(()=>{
    const timerId = setInterval(() => {
      // tick process
    }, 1000)
    return () => clearInterval(timerId)
  },[])
}
```

useEffectの恩恵は何？ → 機能的な凝集性が高い

```
const Component = () => {
  // API コール
  useEffect(()=> {
    const response = apiCall(payload)
    setResponse(response)
  }, [payload])

  // タイマー処理
  useEffect(()=>{
    const timerId = setInterval(() => {
      // tick process
    }, 1000)
    return () => clearInterval(timerId)
  },[])
}
```

- **APIコール**
  - mount時にAPIコール
  - payloadのupdate時にAPIコール
- **タイマー**
  - mount時にタイマーをスタート
  - unMount時にタイマーをクリア

useEffectの恩恵は何？ → 機能的な凝集性が高い

```
const Component = () => {
  // API コール
  useEffect(()=> {
    const response = apiCall(payload)
    setResponse(response)
  }, [payload])

  // タイマー処理
  useEffect(()=>{
    const timerId = setInterval(() => {
      // tick process
    }, 1000)
    return () => clearInterval(timerId)
  },[])
}
```

- **APIコール**
  - mount時にAPIコール
  - payloadのupdate時にAPIコール
- **タイマー**
  - mount時にタイマーをスタート
  - unMount時にタイマーをクリア

機能的な凝集性が高い→**再利用がしやすい**

lifecycle hookは？

lifecycle hookの恩恵は？

lifecycle hookの恩恵は？ → 時間的な凝集性が高い

lifecycle hookの恩恵は？ → 時間的な凝集性が高い

```
export default {
  watch: {
    payload: {  this.fetchData();  } // APIコール
  },
  mounted() {
    this.fetchData(); // APIコール
    this.timerId = setInterval(this.tick, 1000);     // タイマー処理
  },
  beforeUnmount() {
    clearInterval(this.timerId);     // タイマー処理
  }
};
```

lifecycle hookの恩恵は？ → 時間的な凝集性が高い

```
export default {
  watch: {
    payload: {  this.fetchData();  } // APIコール
  },
  mounted() {
    this.fetchData(); // APIコール
    this.timerId = setInterval(this.tick, 1000);     // タイマー処理
  },
  beforeUnmount() {
    clearInterval(this.timerId);     // タイマー処理
  }
};
```

lifecycle hookの恩恵は？ → 時間的な凝集性が高い

```
export default {
  watch: {
    payload: {  this.fetchData();  } // APIコール
  },
  mounted() {
    this.fetchData(); // APIコール
    this.timerId = setInterval(this.tick, 1000);     // タイマー処理
  },
  beforeUnmount() {
    clearInterval(this.timerId);     // タイマー処理
  }
};
```

- **update**
  - APIコール
- **mount**
  - APIコール
  - タイマースタート
- **unMount**
  - タイマーをクリア

lifecycle hookの恩恵は？ → 時間的な凝集性が高い

```
export default {
  watch: {
    payload: {  this.fetchData();  } // APIコール
  },
  mounted() {
    this.fetchData(); // APIコール
    this.timerId = setInterval(this.tick, 1000);     // タイマー処理
  },
  beforeUnmount() {
    clearInterval(this.timerId);     // タイマー処理
  }
};
```

- **update**
  - APIコール
- **mount**
  - APIコール
  - タイマースタート
- **unMount**
  - タイマーをクリア

時間軸で機能は分断される。
物理的なまとまりが無いので機能の関連性を見つけるのは難しい。

useEffect と lifecycle比較

ライフサイクルの変化

useEffect

- **APIコール**
  - mount時にAPIコール
  - payloadのupdate時にAPIコール
- **タイマー**
  - mount時にタイマーをスタート
  - unMount時にタイマーをクリア

機能的な凝集性が高い。
機能で**再利用**がしやすい。
発火タイミングを読み解く必要がある。

lifecycle hook

- **update**
  - APIコール
- **mount**
  - APIコール
  - タイマースタート
- **unMount**
  - タイマーをクリア

時間的な凝集性が高い。
いつ何が起きるか分かりやすい。
機能が時間で切られて散らばる。

実装のアプローチ

ComposableのLifecycle hookが◎

```
function useInterval() {
    let timerId

    onMounted(() => {
    timerId = setInterval(() => {
                // tick process
            }, 1000)
    });

    onUnmounted(() => {
        clearInterval(timerId)
    });
}
```

Composableで機能的な凝集性が高い
lifecycle hook郡を定義できる。

機能的な凝集性の高さを維持しつつ、
内部で発火タイミングも明確化。

ComposableのLifecycle hookが◎

```
function useInterval() {
    let timerId

    onMounted(() => {
    timerId = setInterval(() => {
                // tick process
            }, 1000)
    });

    onUnmounted(() => {
        clearInterval(timerId)
    });
}
```

Composableで機能的な凝集性が高い
lifecycle hook郡を定義できる。

機能的な凝集性の高さを維持しつつ、
内部で発火タイミングも明確化。

ComposableのLifecycle hookが◎

```
function useInterval() {
  let timerId

  onMounted(() => {
  timerId = setInterval(() => {
            // tick process
        }, 1000)
  });

  onUnmounted(() => {
    clearInterval(timerId)
  });
}
```

Composableで機能的な凝集性が高い
lifecycle hook郡を定義できる。

機能的な凝集性の高さを維持しつつ、
内部で発火タイミングも明確化。

いいとこどり！

目次

1. ライフサイクル表現の変化
2. **構文の変化**
3. まとめ

構文の変化

React（JSX）

```
<Button
  className="red"
  onClick={submit}
  data-cy="submitButton"
>
  Save
</Button>
```

Vue（テンプレート）

```
<Button
  class="red"
  @click="submit"
  data-cy="submitButton"
>
  Save
</Button>
```

React（JSX）

```
<Button
  className="red"
  onC
  data
>
  Save
</Button>
```

Vue（テンプレート）

```
<Button
  class="red"
  Save
</Button>
```

一番大きな変化

構文の変化

React（JSX）

```
<Button
  className="red"
  onC
  data-cy="submitButton"
>
  Save
</Button>
```

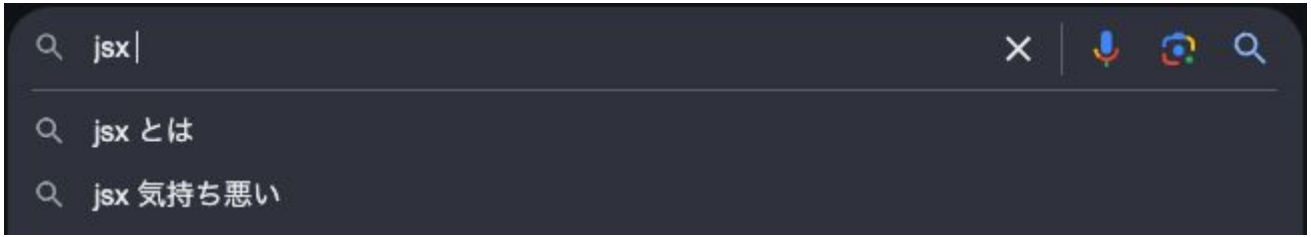
Vue（テンプレート）

```
<Button
  class="red"
  data-cy="submitButton"
>
  Save
</Button>
```

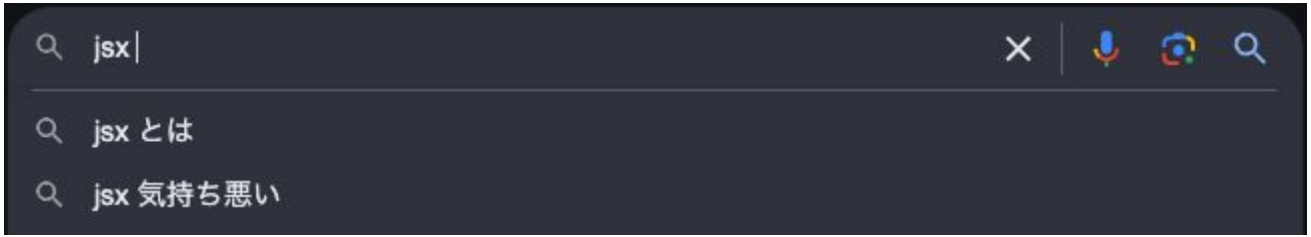
Vueには詳しいと思うので、、、

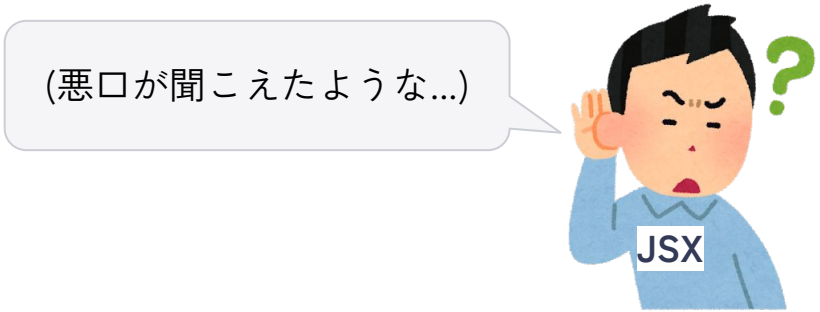
ReactのJSXって？？

ReactのJSXって？？

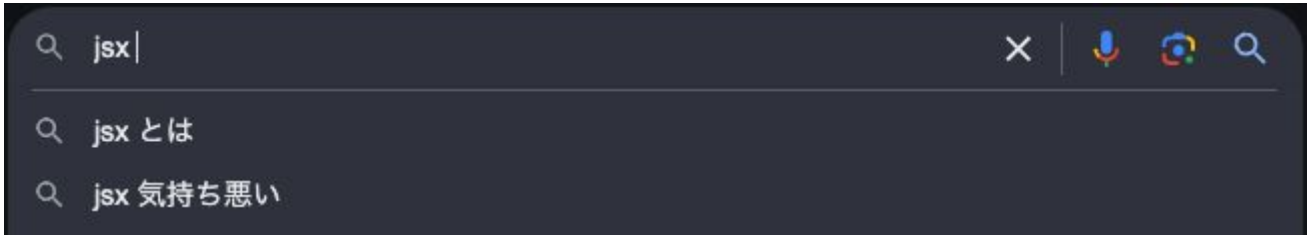


ReactのJSXって？？





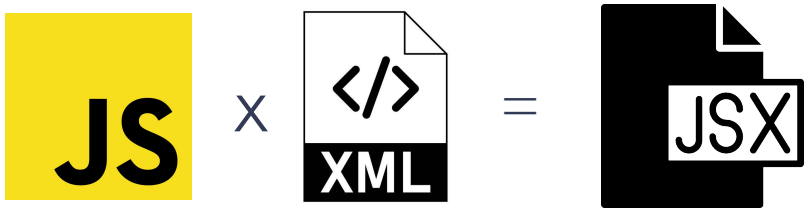
ReactのJSXって？？



気持ち悪い ≒ よく知らないから？

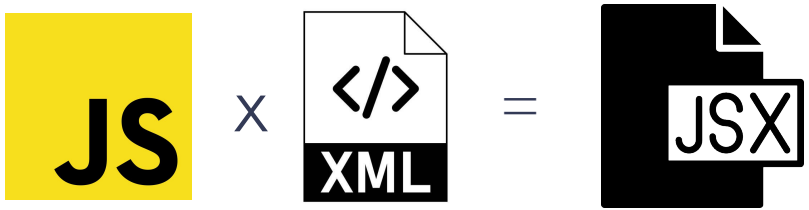
JSXとは何者なのか？

JSXはJavaScriptの構文拡張



- テンプレート拡張じゃない
- JavaScriptとXMLの組み合わせ
- ECMAScript2015に対する構文拡張

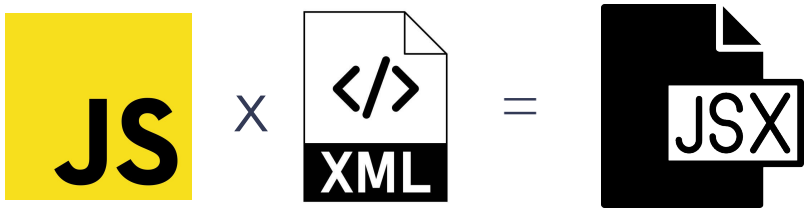
JSXはJavaScriptの構文拡張



- テンプレート拡張じゃない
- JavaScriptとXMLの組み合わせ
- ECMAScript2015に対する構文拡張

なにがJavaScriptに拡張されたの？

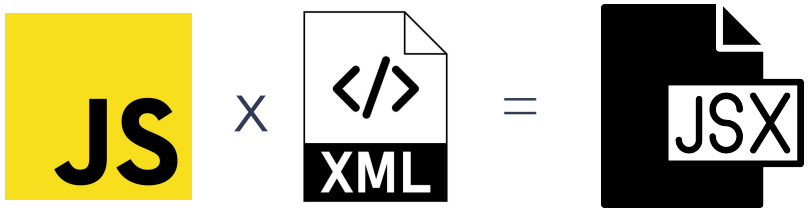
JSXはJavaScriptの構文拡張



- テンプレート拡張じゃない
- JavaScriptとXMLの組み合わせ
- ECMAScript2015に対する構文拡張

なにがJavaScriptに拡張されたの？
JSX.Element型

JSXはJavaScriptの構文拡張



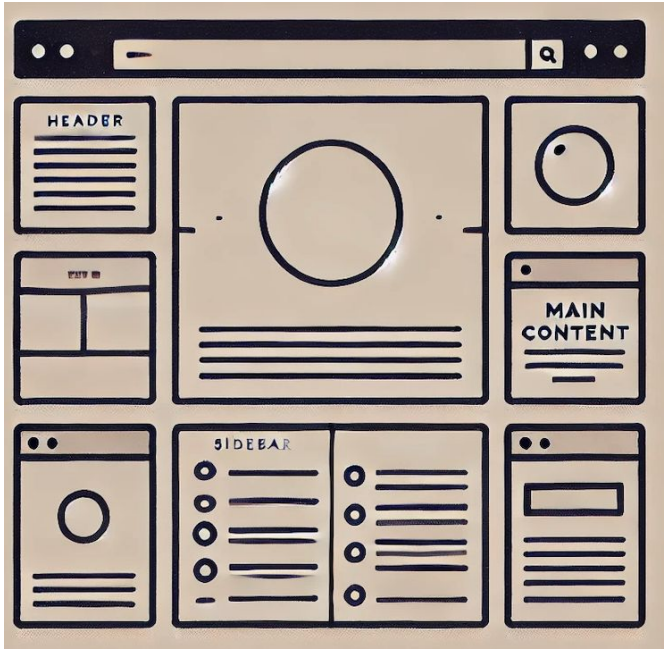

```
const str: string = "hello world"


const Element: JSX.Element = <span>hoge</span>
```

- テンプレート拡張じゃない
- JavaScriptとXMLの組み合わせ
- ECMAScript2015に対する構文拡張
- JSX.Element型が使える
- TypeScriptが言語サポート

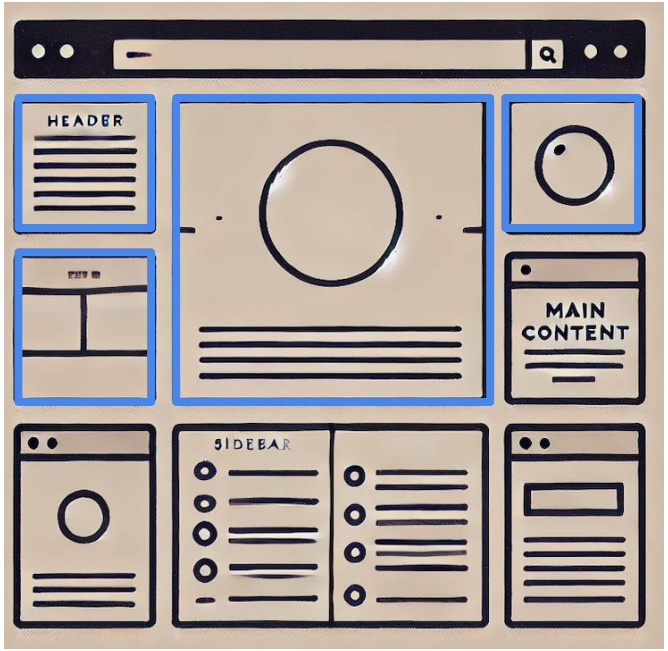
なぜReactはJSXなのか？

なぜReactはJSXなのか？　→コンポーネントという関心の分離にフォーカスするため

なぜReactはJSXなのか？　→コンポーネントという関心の分離にフォーカスするため

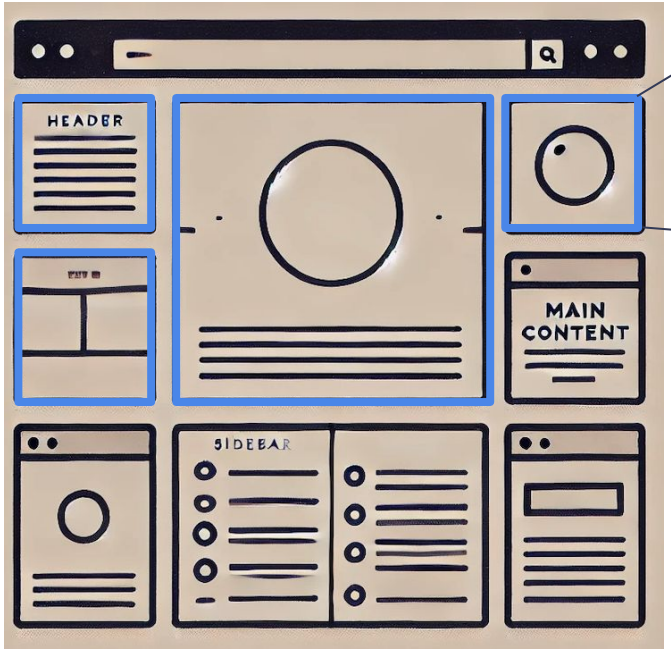


なぜReactはJSXなのか？　→コンポーネントという関心の分離にフォーカスするため



分離すべきは**コンポーネント**。

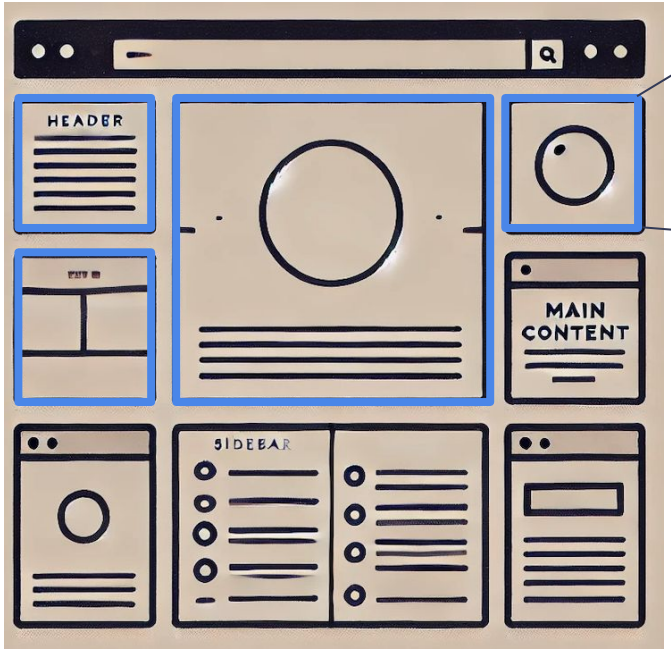
なぜReactはJSXなのか？　→コンポーネントという関心の分離にフォーカスするため




```
<template>
  <div></div>
</template>

<script>
</script>
```

分離すべきは**コンポーネント**。

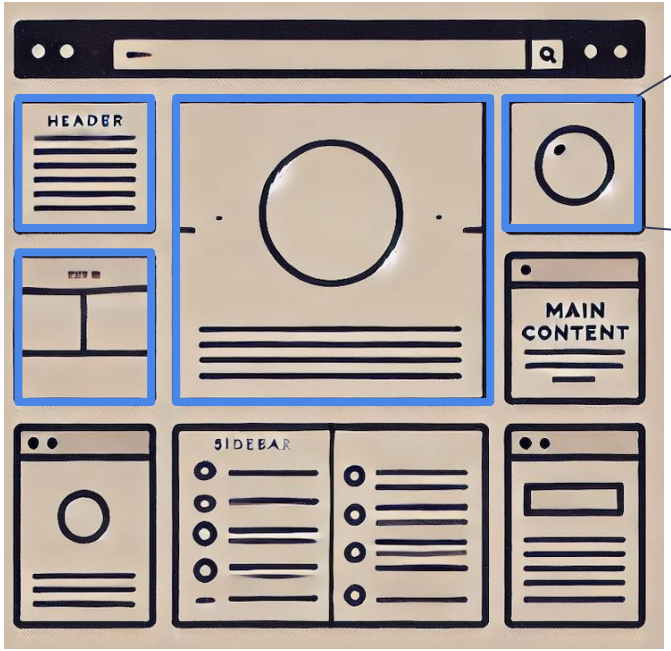
なぜReactはJSXなのか？　→コンポーネントという関心の分離にフォーカスするため




```
<template>
  <div></div>
</template>

<script>
</script>
```

分離すべきは**コンポーネント**。
UI・ロジックではない。
技術の分離は**関心の分離には寄与しない**。

なぜReactはJSXなのか？　→コンポーネントという関心の分離にフォーカスするため

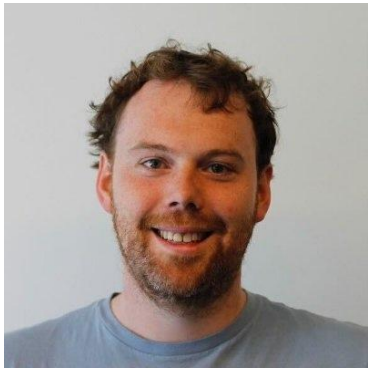



```
<template>
  <div></div>
</template>

<script>
</script>
```

分離すべきは**コンポーネント**。
UI・ロジックではない。
技術の分離は**関心の分離には寄与しない**。

であれば一つの技術（JavaScript）の表現力を活かし
コンポーネントの分離にフォーカスする。
JavaScriptドリブン

なぜReactはJSXなのか？



Reactチーム初期メンバー: Pete Hunt
React rethinking best practice

A framework cannot know how to separate your concerns for you.

フレームワークは開発者がどのように関心の分離をするのか分からない。

Building components, not templates

テンプレートではなくコンポーネントをつくる。

ではVueは？

Vueもやりたいことは同じ

公式ドキュメントより

なぜSFCなのか（*why-sfc*）
関心の分離がファイルタイプの分離と同じではない

コンポーネント内では、そのテンプレート、ロジック、およびスタイルが本質的に結合されており、それらを連結することで、実際にコンポーネントがよりまとまり、保守しやすくなります。

Vueもやりたいことは同じ

公式ドキュメントより

なぜSFCなのか（*why-sfc*）
関心の分離がファイルタイプの分離と同じではない

コンポーネント内では、そのテンプレート、ロジック、およびスタイルが本質的に結合されており、それらを連結することで、実際にコンポーネントがよりまとまり、保守しやすくなります。

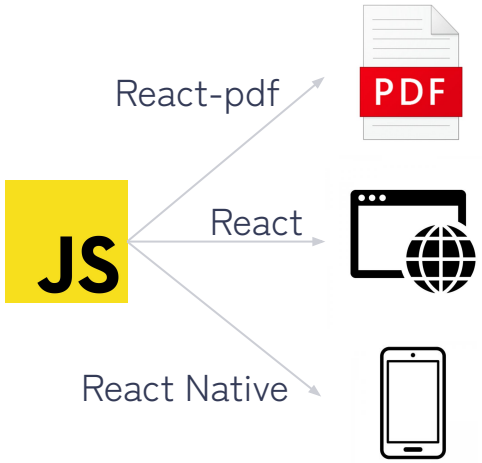
SFCという概念が独立性の高いコンポーネントを作るギブス

構文の違い≒思想の違い

Webアプリケーションの捉え方

React

アプリケーションのレンダリング先がブラウザ



Vue

動的なHTMLテンプレート

```
<template>
  <body>
    <header> {{ header }} </header>
    <main> {{ content }} </main>
    <footer> {{ footer }} </footer>
  </body>
<template>
```

思考の切り替え

React
UI（JSX）を返す関数を書く

Vue
ディレクティブ等で動的なHTMLを作る

```
// JSX
{condition && <span>Visible</span>}
<ul>
  {items.map(item => <li key={item.id}>{item.name}</li>)}
</ul>
```

```
<!-- Vue Template -->
<span v-if="condition">Visible</span>
<ul>
  <li v-for="item in items" :key="item.id">{{ item.name }}</li>
</ul>
```

目次

1. 構文の変化
2. ライフサイクル表現の変化
3. まとめ

まとめ

まとめ

- useEffectはムズい。
- lifecycle hookは**機能的な凝集**を意識
- ReactはJavaScriptドリブン
- Vueは動的HTML
  - （Vapor modeに期待）

ご清聴ありがとうございました。

MNTSQ