CSS Architecture on Vue.js

Vue.jsのScoped CSSに適したCSS設計を考える

39

Takami Yamada, aka @tacamy

Design Engineer

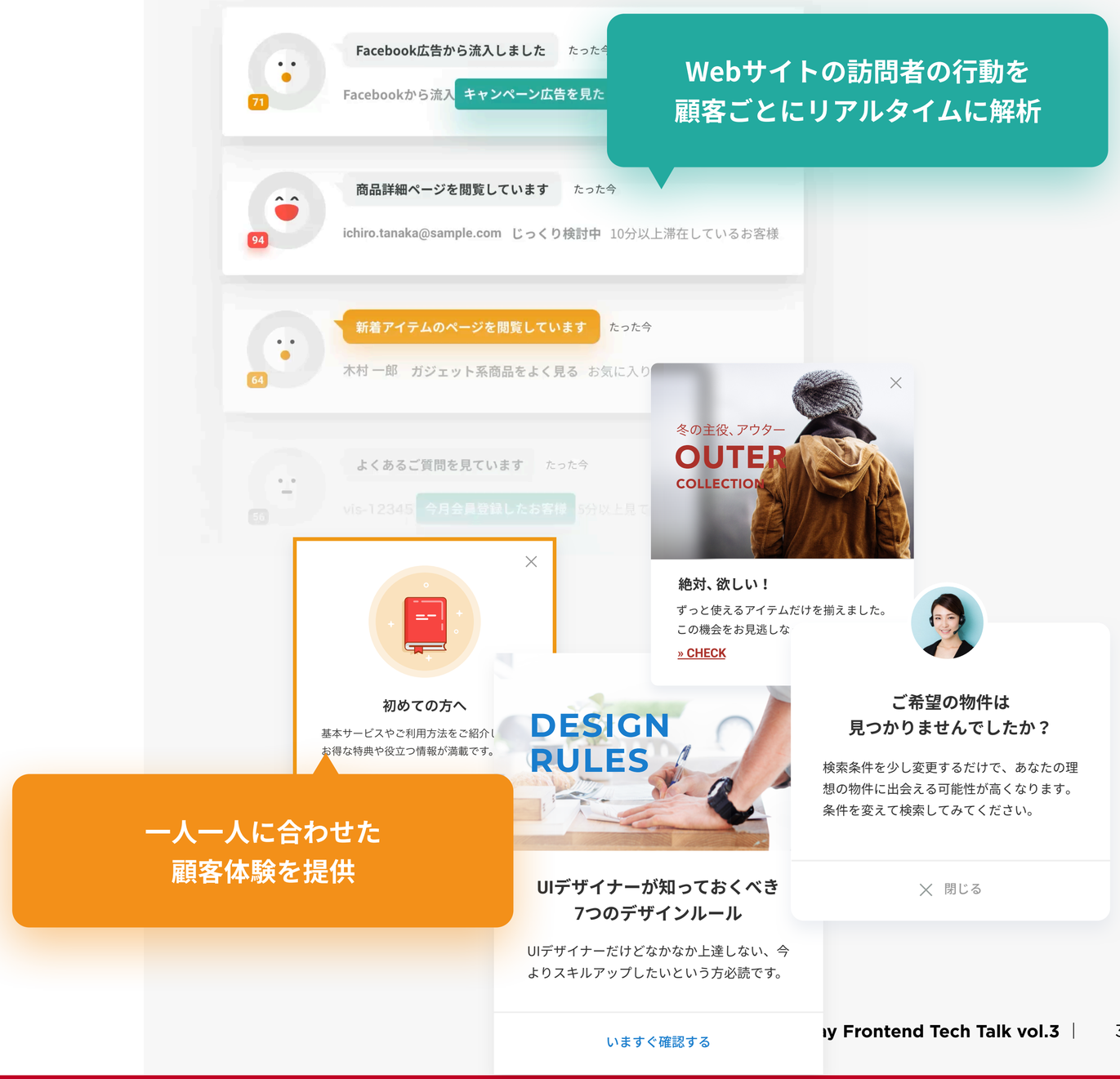
HTML / CSS : 10年

Vue.js : 2年

9月に株式会社プレイドに転職しました :tada:

KARTE

CX(顧客体験)プラットフォーム



こんな人向けの話です

- CSS設計は得意だが、Vue.jsのScoped CSSでの設計方法に迷いがある…
- Scoped CSSなら安心！と適当に書いたらスタイルがバッティングしてハマる…

コンポーネントってなんだろう？

機能や振る舞いがワンセット

ひとまとまりの機能を持つ
自己完結型の再利用可能な部品

どこで利用しても同じように動作

Vue.jsのSFCは
1つのファイルで
コンポーネントを実現

```
// コンポーネントを構成する要素
<template>
 <div class="MyComponent"></div>
</template>

// コンポーネントの状態や機能を定義
<script></script>

// コンポーネントの見た目を制御
<style scoped>
.MyComponent { /* ... */ }
</style>
```

コンポーネント単位のスコープでCSSを書ける

念願のCSSのスコープを手に入れた🎉

でも実際はただのCSS

属性セレクタによりScopedを擬似的に実現

```
// Example.vue
<template>
  <div class="Example">hi</div>
</template>

<style scoped>
.Example { color: red; }
</style>

// 生成されるHTML
<style>
.Example[data-v-f3f3eg9] {
  color: red;
}
</style>

<div class="Example" data-v-f3f3eg9>
  hi
</div>
```

コンポーネント単位で自動的に付与される

そのため、本物のスコープを持つ
Shadow DOMとは異なる

Vue.jsのScoped CSSの留意点

https://vue-loader-v14.vuejs.org/ja/features/scoped-css.html

要素セレクタへの
スタイル指定はNG 🙅‍♀️

```
// NG
<template>
  <p>hi</p>
</template>

<style scoped>
p { color: red; }
</style>

// OK 🙆‍♀️
<template>
  <p class="Example">hi</p>
</template>

<style scoped>
.Example { color: red; }
</style>
```

- 要素セレクタへのスタイル指定は、属性セレクタと組み合わせたとき何倍も遅くなる
- クラスセレクタに対するスタイル指定なら、パフォーマンスに影響なし

子コンポーネントの
ルート要素は
親スコープの影響をうける

```
// Parent.vue
<template>
  <div class="Parent Container">
    <Child />
  </div>
</template>

<style scoped>
.Container { color: red; }
</style>

// Child.vue
<template>
  <div class="Child Container">...</div>
</template>

<style scoped>
...
</style>

// 生成されるHTML
<div data-v-e1272e36 class="Parent Container">
  <div
    data-v-49e3088a data-v-e1272e36
    class="Child Container">...</div>
</div>
```

子コンポーネントの文字色もredになる

- 子のルート要素には、子自身のデータ属性＋親のデータ属性が付与される
  - 親のScoped CSS内で、子のルート要素と同じクラス名がある場合、
    子にそのスタイルが適用される

👍 親はレイアウト目的で子のルート要素をスタイリングできる
  - コンポーネント自体にはmarginを持たせず、親コンポーネントでmarginをつける等

👎 意図せぬスタイルのバッティング

- 親と子で同じクラス名がなければ問題ない

deepセレクタで
子コンポーネント以下に
影響を与えることが可能

```
// Vueファイル
<style scoped>
.Parent >>> .Child {
  /* ... */
}
</style>

// コンパイル結果
.Parent[data-v-f3f3eg9] .Child {
  /* ... */
}
```

スコープが消え去る

- deepはセレクタのネストにより優先度が高くなり、子のスタイルが負ける場合もある
- Scopedも崩壊するので極力つかわない

- Element UIなどの外部ライブラリのスタイル調整にはテーマ機能を利用
  足りないスタイルは専用のラッパーコンポーネントを用いて影響範囲を最小化

これらの特徴によるトラブルを回避する方法

命名規則に則って
要素に固有のクラス名を付与

BEMの考え方に似ている…？

コンポーネントを
BEMのBlockのように捉えてみる

.block

.ComponentName

- BEMのblockの粒度 ≒ Vueのコンポーネントの粒度 と捉える
- コンポーネントのルート要素がblockにあたる

.block__element

.ComponentName__element

- block部分がComponentNameに変わるだけで、他はBEMと同様
- .ComponentName__element__element にならないよう
  - 粒度が大きすぎる ≒ 機能も複雑になりやすい
- 詳細度を一定に保つため、CSSは1階層で記述

.block__element--modifire

.ComponentName__element._modifire

- modifireは状態によってtemplate内で付け外しすることが多いため、BEM式はさすがに冗長で見づらくなってしまう
- blockやelementと組み合わせてスタイル指定することを前提に_modifireの形式とする
  - 組み合わせることで、blockやelementより詳細度が高くなるメリットも
- modifireはJSで扱うことが多いため、ハイフンよりアンダーバーの方が楽

BEM式で命名するメリット

- 広く知られている命名ルールで共通認識があり、書き方に統一性が生まれる
- コンポーネント内のCSSに秩序が生まれる
- HTMLのクラス名を見るだけで、どこにスタイルが書かれているか分かる
- あとからコンポーネントを分割するときも、クラス名がバッティングしない

でもBEMはクラス名が長くて面倒？

$options.nameとSassの&で楽

```
<template>
  <div :class="$options.name">
    <div
      :class="`${$options.name}__element`"
    >
      ...
    </div>
  </div>
</template>

<script>
export default {
  name: 'Example'
};
</script>

<style lang="scss" scoped>
.Example {
  &__element { /* ... */ }
}
</style>
```

コンポーネント名の命名規則

https://jp.vuejs.org/v2/style-guide/index.html

Vue.jsのスタイルガイド

2単語以上で構成

- HTML要素との衝突を防止
  - HTMLでは大文字も小文字扱いになるため
    <Button>は<button>となり、HTML要素とバッティングする
  - 将来追加される未知のHTML要素のことも考慮

密結合コンポーネントの名前

- 親コンポーネントと密結合した子コンポーネントは、親コンポーネントの名前をプレフィックスとして含む

- 親コンポーネントのディレクトリの中に子コンポーネントを入れるのは非推奨
  - 同じような名前のファイルが多数できてしまい、エディタ上でのファイル切り替えが難しくなる
  - ネストが深くなると、エディタのサイドバーでコンポーネントを参照するのに不便

```
// NG
components/
|- TodoList/
   |- Item/
      |- index.vue
      |- Button.vue
   |- index.vue
```

```
// OK
components/
|- TodoList.vue
|- TodoListItem.vue
|- TodoListItemButton.vue
```

ディレクトリ構成

- mixin.scssとvariables.scssは components、layouts、pages以下 すべてのコンポーネントから読み込む
- layoutsは共通レイアウト
- pagesはルーティング
- 共通コンポーネントはcomponentsに

ここの分類を考える

```
// 例としてNuxt.jsの構造をもとに
// スタイルに関連する部分を抜粋
project/
|- assets/
   |- scss/
      |- _mixins.scss
      |- _variables.scss
   |- images/
|- components/
|- lauouts/
   |- default.vue
|- pages/
   |- index.vue
   |- directory/
      |- index.vue
```

コンポーネントの分類

コンポーネントの分類

- 複雑すぎる分類はコストがかかる
  - 所属するコンポーネントを議論するためのコスト
  - コンポーネント利用時の探すコスト
  - 正しい分類を維持していくための定期的な見直しコスト
  - 新メンバー立ち上がりまでのコスト
- 何のために分類するのか目的を明確に

コンポーネントの粒度（大きさ）

- 階層が深くなりすぎると、propsバケツリレーと$emit地獄に👹
- 細かすぎる分割はパフォーマンスにも影響あり

粒度による分類

Part / Module

- Partコンポーネント
  - 機能が成立する最小単位の部品
  - 単体でも利用可能
  - ボタン、フォームパーツ、アイコン
- Moduleコンポーネント
  - Partsを組み合わせて別の機能を持つ
  - Moduleに別Moduleも含めることもできる
  - カード、検索ボックス、ヘッダー

役割による分類

Presentational / Container

https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

- Presentationalコンポーネント
  - 再利用される前提の要素
  - 親からpropsを受け取って表示
  - 親へ$emitで伝達
  - storeに依存しない
- Containerコンポーネント
  - ページそのもの、またはページを構成する要素
  - 子へのデータの受け渡し
  - 子からイベントを受け取って処理を行う
  - storeのデータの取得や更新を行う

プロジェクトに最適な分類とは？

- 最初はなるべくシンプルにはじめて、必要に応じて分類を増やすとよさそう
- 議論に時間がかかる場合は、そもそもその分類がToo muchかも？

1st STEP

分類しないですべてのコンポーネントを同一階層に

2nd STEP

共通コンポーネントが増えてきた
探しやすさのために粒度で分類

分類（ディレクトリ）	コンポーネントの例
👉 parts	ボタン、フォームパーツ、アイコン
👉 modules	カード、検索ボックス、ヘッダー

3rd STEP

データの流れが追えなくなってきてデバッグしづらい
役割の明確化のために分類

分類（ディレクトリ）	役割	Store依存
parts	Presentational	×
modules	Presentational	×
👉 container	Container	○

理想と現実

これまでの話は理想であり
最初から理想どおりにつくるのは難しい

🎉 すべてのCSSをコンポーネントに紐づくScoped CSSにするメリット

- 構造に一貫性があり、保守しやすい（長期運用にも耐えうる）

🤔 実現するための課題

- 初期段階で、すべてのユースケースを考慮したコンポーネントをつくるのは困難
- スタートアップのプロトタイピング的なプロジェクトでは、時間をかけてつくっても無駄になる可能性
- 調整用クラスはないほうがいいけど、利用者側からするとあった方がやっぱり便利

徐々に導入する
コンポーネント的CSS設計

1st STEP

すべてのスタイルを外部CSSファイルに記述

🎉 templateのHTMLにクラスを付与するだけでスタイルを適用できる

🎉 デザイナーがCSSを書くときに、Vueコンポーネントの粒度を意識しなくてよい

🤔 記述されたスタイルがどこで利用されているかわからない

- 不要なスタイルが残り続けることで負債が蓄積し、メンテナンスコスト高
- 修正したつもりが意図しない箇所が壊れてしまう、エンジニアは触りたくない

🤔 CSSを書かないエンジニアも、状態によるクラスの付け替えルールを把握する必要がある

2nd STEP

👈 KARTEはイマココ！

外部CSSファイルとScoped CSSのハイブリッド式

- 共通コンポーネントのスタイルが書かれた外部CSSを全ページで読み込む
- それ以外のスタイルは、各コンポーネントにScoped CSSで記述

ハイブリッド式のメリット・デメリット

🎉 スピード感を維持したまま、一部でScoped CSSの恩恵を受けられる

🎉 Scoped内なら安心して、誰でもCSSを触ることができる

🤔 スタイルの適用元がわからなくなるときがある

- 共通コンポーネントに接頭辞をつける命名ルールである程度回避

🤔 Scoped内を自由に書けすぎてしまう

- UIのルールが守られていない箇所もある
- 同じような機能やスタイルがコピペであちこちに散在（でも微妙に違ったり）

🤔 コンポーネントが外部のCSSに依存しているので、CSSが古いページにコンポーネントを配置したときに表示が崩れてしまう

3rd STEP

しばらく運用して、パターンがある程度出揃った段階で
すべてのスタイルをScoped CSSでカプセル化

- 🎉 どこで使用しても同じように表示できることが保証できる
- 🎉 スタイルの影響範囲が明確で、長期的に保守しやすい
- 🤔 コンポーネントのアップデートの周知がうまくされないと
  似たようなコンポーネントがつくられてしまう
- 🤔 細かいコンポーネントが増えるとテストの手間も増える

どの方法を選択するか？

- 唯一の正しい方法は存在しない
- プロジェクトの規模・フェーズ・個々の事情による
- 外部のCSSに依存する場合も、
  リセットやベーススタイルは極力ブラウザデフォルトに合わせておくと吉

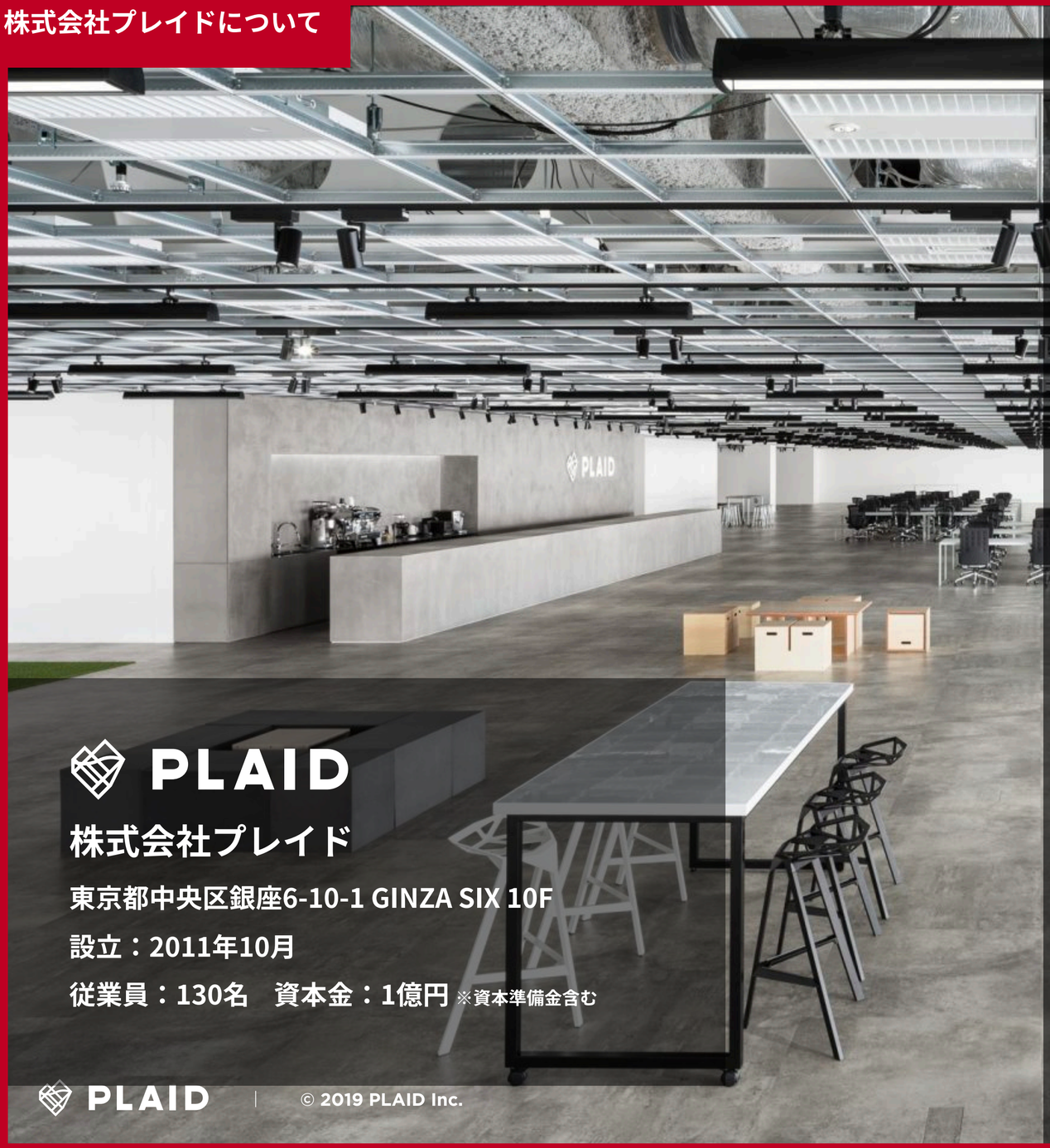
より良いCSSのゴール

≒

コンポーネント

- ✔ 予測しやすい
- ✔ 再利用しやすい
- ✔ 保守しやすい
- ✔ 拡張しやすい

CSS設計 👉 コンポーネント設計



PLAID

株式会社プレイド

東京都中央区銀座6-10-1 GINZA SIX 10F

設立：2011年10月

従業員：130名　資本金：1億円 ※資本準備金含む

株式会社プレイドでは
KARTEのコンポーネント化を
一緒に推し進めてくれる
CSS設計の得意な仲間を募集中です！

https://www.wantedly.com/projects/299653


PLAID