記事
Toshihiko Minamoto · 2022年12月12日 16m read

Angular: コンテナとプレゼンテーションコンポーネント（別名: smart-dumb）

こんにちは！ 今日は、Angular で最も重要なアーキテクチャパターンの 1 つについてお話しします。

パターン自体は直接 Angular に関連していませんが、Angular
はコンポーネント駆動のフレームワークであるため、このパターンは最新の Angular
アプリケーションを構築するために最も不可欠なものの 1 つです。

「コンテナ・プレゼンテーション」パターン

コンポーネントは、小さい、目的集中型、独立型、テスト可能性、そして一番重要と言える再利用可能性という特性が備わっているのが最適だと考えられています。

コンポーネントに、サーバー呼び出しを行う、ビジネスロジックが含まれている、他のコンポーネントと密に連携している、他のコンポーネントまたはサービスの内部を過度に知っている、という特徴が備わっていれば、より大きく、テスト、拡張、再利用、変更が行いにくいものになってしまします。
「コンテナ・プレゼンテーション」パターンは、これらの問題を解決するために存在します。

一般的にすべてのコンポーネントは、コンテナコンポーネント（スマート）とプレゼンテーションコンポーネント（ダム）の 2 つのグループに分けられます。

コンテナコンポーネントは、サービスからデータを取得し（ただし、サーバー API を直接呼び出さない）、何らかのビジネスロジックを含み、サービスまたは子コンポーネントにデータを配信することができます。 通常、コンテナコンポーネントは、ルーティング構成でルートが指定されたコンポーネントとして指定されるものです（もちろん、必ずしもそうとは限りません）。

プレゼンテーションコンポーネント
は、入力としてデータを取り、何らかの方法で画面に表示することのみを行います。 これらのコンポーネントはユーザー入力に反応しますが、それはローカルで独立した状態を変更することでのみ発生するものです。
アプリの残りの部分とのすべてのやり取りは、カスタムイベントを発行することで行われます。
これらのコンポーネントには非常に高い再利用可能性が備わっている必要があります。

もう少し分かりやすいように、コンテナコンポーネントとプレゼンテーションコンポーネントの例をいくつか挙げましょう。

コンテナ: About ページ、ユーザーページ、管理者パネル、注文ページなど。

プレゼンテーション: ボタン、カレンダー、テーブル、モーダルダイアログ、テーブルビューなど。

めちゃくちゃなボタンの例

コンポーネントアプローチが誤用されている例として、実際のプロジェクトにおいて私自身が体験した非常に悪い例を見てみましょう。

```
@Component({
  selector: 'app-button',
  template: `<button class="className" (click)=onClick()>{{label}}</button>`
})
export class ButtonComponent {
  @Input() action = '';
  @Input() className = '';
  @Input() label = '';

  constructor(
    private router: Router,
    private orderService: OrderService,
    private scanService: ScanService,
    private userService: UserService
  ) {}

  onClick() {
    if (this.action === 'registerUser') {
      const userFormData = this.userService.form.value;
      // some validation of user data
      // ...
      this.userService.registerUser(userFormData);
    } else if (this.action === 'scanDocument') {
      this.scanService.scanDocuments();
    } else if (this.action === 'placeOrder') {
      const orderForm = this.orderService.form.values;
      // some validation and business logic related to order form
      // ...
      this.orderService.placeOrder(orderForm);
    } else if (this.action === 'gotoUserAccount') {
      this.router.navigate('user-account');
    } // else if ...
  }
}
```

読みやすいように単純化してありますが、実際にはもっとひどいものです。 これは、ユーザーがクリックして呼び出せるすべての可能なアクションが含まれたボタンコンポーネントです。API
呼び出し、フォームの検証、サービスからの情報取得などを行います。 このようなコンポーネントは、比較的小さなアプリケーションで使用されていたとしても、あっと言う間に地獄入りしてしまうのが想像できるでしょう。私が見つけて（後でリファクタリングした）このようなボタンコンポーネントのコードは、2000
行以上にも及ぶものでした。 あり得ません！

それを書いた開発者に、すべてのロジックを 1 つのコンポーネント収めた理由を尋ねたところ、本人が行ったのは「カプセル化」だという答えが返ってきました。

理想的なコンポーネントに備わっているべき特性について思い出しましょう。

小さい - 2000 行以上のコードでできたこのボタンは小さくありません。 また、他のアクション用に別のボタンが必要という人が現れるたびに、コンポーネントが膨れ上がってしまうでしょう。

目的集中型 - このボタンはまったく無関係な多くのアクションを実行するため、目的集中型とは呼べません。

独立型 - このボタンはいくつかのサービスとフォームと密に連携しているため、いずれかを変更すると、ボタンに影響があります。

テスト可能性 - ノーコメントです。

再利用可能性 - まったく再利用できません。 使用するたびに、含まれていないアクションを追加するためにコンポーネントのコードを変更する必要があります。さらに、このボタンの不要なアクションや依存関係もすべて含まれることになります。

さらに、このボタンコンポーネントは、ネイティブの HTML
ボタンを内部的に非表示にするため、開発者はそのプロパティにアクセスできません。
これが、コンポーネントの悪い書き方としての良い例です。

このボタンコンポーネントを使用する非常に単純なコンポーネントの例を示し、コンテナ・プレゼンテーションパターンでリファクタリングしてみましょう。

```
@Component({
  selector: 'app-registration-form',
  template: `<form [formGroup]="userService.form">
  <input type="text" [formControl]="userService.form.get('username')" placeholder="Ni
ckname">
  <input type="password" [formControl]="userService.form.get('password')" placeholder
="Password">
  <input type="password" [formControl]="userService.form.get('passwordConfirm')" plac
eholder="Confirm password">
  <app-button className="button accent" label="Register" action="registerUser"></app-
button>
</form>
`
})
export class RegistrationFormComponent {
  constructor(public userService: UserService) {}
}
```

このコンポーネントの中には何もロジックがありません。フォームはサービスに格納されており、ボタンにはクリックによるすべてのロジックの呼び出しが含まれます。 現時点では、ボタンにその動作に関係のないすべてのロジックが含まれており、ボタンが処理するアクションに直接関係している親コンポーネントよりスマートです。

コンテナ・プレゼンテーションパターンでボタンコンポーネントをリファクタリングする

これらの 2 つのコンポーネントの関数を分離しましょう。
ボタンはプレゼンテーションコンポーネント、つまり小さく再利用可能である必要があります。 このボタンを含む登録フォームは、ビジネスロジックやサーバーレイヤーとのやり取りを保持するコンテナコンポーネントとすることができます。

ネイティブボタンを表示する部分は説明せずに（これについては、おそらく今後の記事で話すことにします）、主に 2 つのコンポーネントのアーキテクチャ上の関係に焦点を当てます。

リファクタリングしたボタンコンポーネント（プレゼンテーション）

```
@Component({
  selector: 'app-button',
```

```
  template: `<button class="className" (click)=onClick()>{{label}}</button>`
})
export class ButtonComponent {
  @Input() className = '';
  @Input() label = '';

  @Output() click: EventEmitter = new EventEmitter();

  onClick() {
     this.click.emit();
  }
}
```

ご覧のとおり、リファクタリングしたボタンは非常に簡潔です。 2 つの入力と 1 つの出力しか含まれません。 入力は、親コンポーネントからデータを取ってユーザーにそれを表示するために使用します（ボタンの外観はクラスと表示ボタンのラベルで変更します）。
出力は、ユーザーがボタンをクリックするたびに発行されるカスタムイベントに使用されます。

このコンポーネントは小さく、目的集中型、独立型、テスト可能性、および再利用可能性の特性を備えています。 コンポーネント自体の動作に無関係なロジックは含まれていません。 アプリケーションの残りの内部動作についてまったく認識しておらず、アプリケーションのあらゆる部分または他のアプリケーションにも安全にインポートして使用することが可能です。

リファクタリングした登録フォームコンポーネント（コンテナ）

```
@Component({
  selector: 'app-registration-form',
  template: `<form [formGroup]="userService.form">
  <input type="text" [formControl]="userService.form.get('username')" placeholder="Ni
ckname">
  <input type="password" [formControl]="userService.form.get('password')" placeholder
="Password">
  <input type="password" [formControl]="userService.form.get('passwordConfirm')" plac
eholder="Confirm password">
  <app-button className="button accent" label="Register" (click)="registerUser()"></a
pp-button>
</form>
`
})
export class RegistrationFormComponent {
  constructor(public userService: UserService) {}

  registerUser() {
    const userFormData = this.userService.form.value;
    // some validation of user data
    // ...
    this.userService.registerUser(userFormData);
  }
}
```

この登録フォームは、ボタンを使用し、ボタンクリックに反応して registerUser
メソッドを呼び出すようになりました。
このメソッドのロジックはこのフォームに密に関連しているため、それを配置する最適な場所と言えます。

これは非常に単純な例であり、コンポーネントツリーにあるレベルは 2 つだけです。
このパターンには、コンポーネントツリーのレベルが多い場合に落とし穴があります。

より高度な例

これは実際の例ではありませんが、このパターンに潜在する問題を理解する上で役立つと思います。

以下のようなコンポーネントツリーがあるとします（上から下）。

user-orders - トップレベルのコンポーネント。 サービスレイヤーに話しかけ、ユーザーとその注文に関するデータを受け取り、それを後続のツリーに渡して注文のリストをレンダリングするコンテナコンポーネントです。

user-orders-summary - 中間レベルのコンポーネント ユーザーの注文リストの上に合計注文数を表示するバーをレンダリングするプレゼンテーションコンポーネントです。

cashback - 最下レベル（リーフ/葉）コンポーネント。 ユーザーの合計キャッシュバック額を表示し、銀行口座に引き落とすためのボタンを持つプレゼンテーションコンポーネントです。

トップレベルのコンテナコンポーネント

トップレベルの user-orders コンテナコンポーネントを見てみましょう。

```
@Component({
  selector: 'user-orders',
  templateUrl: './user-orders.component.html'
})
export class UserOrdersComponent implements OnInit {
  user$: Observable<User>;
  orders$: Observable<Order[]>;

  constructor(
    private ordersService: OrdersService,
    private userService: UserService
  ) {}

  ngOnInit() {
      this.user$ = this.userService.user$;
      this.orders$ = this.ordersService.getUserOrders();
  }

onRequestCashbackWithdrawal() {
    this.ordersService.requestCashbackWithdrawal()
      .subscribe(() => /* notification to user that cashback withdrawal has been requ
ested */);
    }
}
```

```
<div class="orders-container">
    <user-orders-summary
        [orders]="orders$ | async"
        [cashbackBalanace]="(user$  | async).cashBackBalance"
        (requestCashbackWithdrawal)="onRequestCashbackWithdrawal($event)"
    >
    </user-orders-summary>
```

```
    <div class="orders-list">
      <div class="order" *ngFor="let order of (orders$ | async)"></div>
    </div>
</div>
```

ご覧のとおり、user-orders コンポーネントは、user$ と orders$ の 2 つの Obeservable を定義し、非同期パイプをテンプレートに使用してそれらを購読しています。
このコンポーネントはデータをプレゼンテーションコンポーネントの user-orders-summary に渡し、注文リストをレンダリングします。 また、user-orders-summary から発行される requestCashbackWithdrawal カスタムイベントに反応してサービスレイヤーに話しかけます。

中間レベルのプレゼンテーションコンポーネント

```
@Component({
  selector: 'user-orders-summary',
  template: `
    <div class="total-orders">Total orders: {{orders?.length}}</div>
    <cashback [balance]="cashbackBalanace" (requestCashbackWithdrawal)="onRequestCash
backWithdrawal($event)"></cashback>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserOrdersSummaryComponent {
    @Input() orders: Order[];
    @Input() cashbackBalanace: string;

    @Output() requestCashbackWithdrawal = new EventEmitter();

    onRequestCashbackWithdrawal() {
        this.requestCashbackWithdrawal.emit();
    }
}
```

このコンポーネントは、上記でリファクタリングしたボタンコンポーネントに非常に似ています。 その入力から受け取ったデータをレンダリングし、何らかのユーザーアクションでカスタムイベントを発行します。
サービスの呼び出しは行わず、ビジネスロジックも含まれません。 そのため、これは別のプレゼンテーション cashback コンポーネントを使用する純粋なプレゼンテーションコンポーネントです。

最下レベルのプレゼンテーションコンポーネント

```
@Component({
    selector: 'cashback',
    template: `
<div class="cashback">
  <span class="balance">Your cashback balance: {{balance}}</span>
  <button class="button button-
primary" (click)="onRequestCashbackWithdrawal()">Withdraw to Bank Account</button>
</div>
`,
    styleUrls: ['./cashback.component.css']
})
export class CashackComponent {
    @Input() balance: string;

    @Output() requestCashbackWithdrawal = new EventEmitter();
```

```
    onRequestCashbackWithdrawal() {
        this.requestCashbackWithdrawal.emit();
    }
}
```

これは、入力からデータを受け取って出力でイベントをスローするだけの、もう 1 つのプレゼンテーションコンポーネントです。
非常に単純で再利用可能ですが、**コンポーネントツリーに問題があります**。

user-orders-summary コンポーネントと cashback コンポーネントには、類似する入力（cashbackBalanace と balance）と同じ出力（requestCashbackWithdrawal）があることに気づいたことでしょう。
これは、コンテナコンポーネントが一番深いプレゼンテーションコンポーネントから遠すぎるためです。
この設計ではツリーのレベル数が増えるほど、問題が悪化してしまいます。
この問題について詳しく見てみましょう。

問題 1 - 中間レベルのプレゼンテーションコンポーネントにある無関係なプロパティ

user-orders-summary は cashbackBalanace の入力を受け取って、さらにツリーの下の方に渡すだけであり、それ自体がその入力を使用することはありません。
このような状況に直面した場合、コンポーネントツリーの設計に欠陥がある可能性があります。 実際の状況で使用するコンポーネントには多数の入力と出力があるため、この設計では多数の「プロキシ」入力が含まれてしまい、中間レベルのコンポーネントの再利用可能性が薄れてしまいます（子コンポーネントに密に連携させる必要があるため）。また、繰り返し可能なコードが多数含まれてしまいます。

問題 2 - 下方レベルからトップレベルのコンポーネントに湧き上がるカスタムイベント

この問題は、前の問題に非常に似ていますが、コンポーネントの出力に関連しています。
ご覧のとおり、requestCashbackWithdrawal カスタムイベントは、cashback と user-orders-summary コンポーネントで繰り返されています。 これもやはり、コンテナコンポーネントが一番深いプレゼンテーションコンポーネントから遠すぎるために起きていることです。
また、中間コンポーネント自体を再利用できないものにしてしまっています。

これらの問題に対する潜在的な解決策は少なくとも 2 つあります。

1 - ngTemplateOutlet を使用して、中間レベルのコンポーネントをよりコンテンツに依存しないコンポーネントにし、より深いコンポーネントをコンテナコンポーネントに直接公開する。
この解決策には専用の記事にするだけの価値があるため、今回はこれについて説明しないでおきます。

2 - コンポーネントツリーを再設計する。

コンポーネントツリーをリファクタリングする

作成したコードをリファクタリングして、無関係なプロパティと中間レベルのコンポーネントで湧いているイベントの問題を解決できるか見てみましょう。

リファクタリングしたトップレベルのコンポーネント

```
@Component({
  selector: 'user-orders',
  templateUrl: './user-orders.component.html'
})
export class UserOrdersComponent implements OnInit {
```

```
  orders$: Observable<Order[]>;

  constructor(
    private ordersService: OrdersService,
  ) {}

  ngOnInit() {
      this.orders$ = this.ordersService.getUserOrders();
  }
}


<div class="orders-container">
    <user-orders-summary [orders]="orders$ | async"></user-orders-summary>

    <div class="orders-list">
      <div class="order" *ngFor="let order of (orders$ | async)"></div>
    </div>
</div>
```

トップレベルのコンテナコンポーネントから user$ Observable と onRequestCashbackWithdrawal() メソッドを削除しました。 非常に単純化され、user-orders-summary コンポーネント自体をレンダリングするために必要なデータのみを渡し、その子コンポーネントの cashback には渡さないようになっています。

リファクタリングした中間レベルのコンポーネント

```
@Component({
  selector: 'user-orders-summary',
  template: `
    <div class="total-orders">Total orders: {{orders?.length}}</div>
    <cashback></cashback>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserOrdersSummaryComponent {
    @Input() orders: Order[];
}
```

これもかなり単純化されています。 入力は 1 つだけで、合計注文数をレンダリングするようになっています。

リファクタリングした最下レベルのコンポーネント

```
@Component({
    selector: 'cashback',
    template: `
<div class="cashback">
  <span class="balance">Your cashback balance: {{ (user$ | async).cashbackBalance }}<
/span>
  <button class="button button-
primary" (click)="onRequestCashbackWithdrawal()">Withdraw to Bank Account</button>
</div>
`,
    styleUrls: ['./cashback.component.css']
})
```

```
export class CashackComponent implements OnInit {
  user$: Observable<User>;

  constructor(
     private ordersService: OrdersService,
     private userService: UserService
  ) {}

  ngOnInit() {
    this.user$ = this.userService.user$;
  }

  onRequestCashbackWithdrawal() {
    this.ordersService.requestCashbackWithdrawal()
      .subscribe(() => /* notification to user that cashback withdrawal has been requ
ested */);
    }
  }
}
```

すごいです。 ご覧のとおり、**プレゼンテーションコンポーネントではなくなっています**。 トップレベルのコンポーネントに非常に似ており、コンポーネントツリーの下にあるコンテナコンポーネントになりました。
このようにリファクタリングすることで、コンポーネントツリー全体の設計と
API、およびコンポーネントツリーの上位にある 2 つのコンポーネントのロジックを単純化できました。

新しい cashback コンポーネントの再利用可能性についてはどうでしょうか？ まだ、再利用可能なままです。それには、コンポーネント自体に関連するロジックのみが含まれているため、以前として独立性を維持しています。

新しいコンポーネントツリーの設計は、管理性、合理性、および細分化がさらに改善されているようです。 湧き上がってくるイベントが無くなり、コンポーネントツリー全体で繰り返し可能な入力もありません。総合的な設計もはるかに単純になっています。
これは、コンポーネントツリーの下にコンテナコンポーネントを追加することで達成しました。 この手法は、コンポーネントツリーの設計を単純化するために使用できますが、再利用可能性を大幅に失うことなく、ツリー内のどのコンポーネントをコンテナにするのが適しており、どれが純粋なプレゼンテーションコンポーネントであるべきかを十分に理解しておくことが必要です。 これはバランスと設計の選択に関する課題であり、アプリのアーキテクチャを作成するときには必ず発生するものです。

「コンテナ・プレゼンテーション」パターンは非常に誤解しやすいもので、コンテナコンポーネントは必ずトップレベルコンポーネントであるべきだと考えがちです（直感的に、コンテナはローカルコンポーネントツリー内のその他すべてのコンポーネントを格納するものです）。 しかし、そうではありません。コンテナコンポーネントはコンポーネントツリーのあらゆるレベルに配置することができます。上記で見たように、リーフレベルにも配置可能です。
私はコンテナコンポーネントを**スマート**
（賢い）コンポーネントと呼んでいます。私にとって、これらのコンポーネントがビジネスロジックを含み、コンポーネントツリーのあらゆる場所に配置できるということが非常に明白であるためです。

後書き

ここまでで、「コンテナ・プレゼンテーション」パターンの概要とその実装における潜在的な問題について理解できたことと思います。

できるだけ単純に維持するよう努めましたが、このパターンに関しては非常に多くの情報が存在します。

ご質問やメッセージがございましたら、お気軽にコメント欄に投稿してください。

次回の記事では、Angular における changeDetectionStrategy
についてお話しします（この記事に非常に関連する内容です）。

それではまた！

#Angular #Angular2 #UI 開発 #コーディングのガイドライン #フロントエンド #その他

ソースURL:
https://jp.community.intersystems.com/post/angular-%E3%82%B3%E3%83%B3%E3%83%86%E3%83%8A%E3%81%A8%E3%83%97%E3%83%AC%E3%82%BC%E3%83%B3%E3%83%86%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3%E3%82%B3%E3%83%B3%E3%83%9D%E3%83%BC%E3%83%8D%E3%83%B3%E3%83%88%EF%BC%88%E5%88%A5%E5%90%8D-smart-dumb%EF%BC%89