Artigo
Danusa Calixto · Set. 19, 2022 12min de leitura

Angular. Componentes de contêiner e apresentação (burro-inteligente)

Olá! Hoje, eu quero falar sobre um dos padrões arquiteturais mais importantes no Angular.

O próprio padrão não está diretamente relacionado ao Angular, mas, como o Angular é um framework baseado em componentes, esse padrão é um dos mais essenciais para desenvolver aplicativos Angular modernos.

Padrão contêiner-apresentação

Acredita-se que bons componentes devem ser pequenos, focados, independentes, testáveis e, acima de tudo, reutilizáveis.

Se o componente está fazendo chamadas de servidor, contém lógica de negócio, está estreitamente acoplado a outros componentes, sabe muito sobre o funcionamento interno de outros componentes ou serviços, então ele fica maior e mais difícil de testar, ampliar, reutilizar e modificar. Para resolver esses problemas, existe o padrão "contêiner-apresentação".

Geralmente, todos os componentes podem ser divididos em dois grupos: contêiner (inteligente) e apresentação (burro).

Os **componentes contêiner** podem receber dados de serviços (mas não devem chamar APIs diretamente), contêm lógica de negócio e veiculam dados para serviços ou componentes filhos. Frequentemente, os componentes contêiner são aqueles que especificamos como componentes roteados na configuração de roteamento (mas, claro, nem sempre).

Os **componentes de apresentação** só podem receber dados e mostrar de alguma maneira na tela. Eles podem reagir com base nas entradas do usuário, mas só mudando o estado isolado local. Todas as comunicações com o resto do app devem ser feitas ao emitir eventos personalizados. Esses componentes são altamente reutilizáveis.

Para ilustrar, vou nomear alguns exemplos de componentes contêiner e de apresentação:

Contêiner: AboutPage, UserPage, AdminPanel, OrderPage, etc.

Apresentação: Button, Calendar, Table, ModalDialog, TabView, etc.

Exemplo de um botão maluco

Vamos analisar um péssimo exemplo de uso inadequado da abordagem de componentes que encontrei em um projeto real.

```
@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 ...
  }
}
```

Simplifiquei para melhorar a legibilidade, mas, na realidade, estava muito pior. Esse é um componente de botão que contém todas as ações possíveis que o usuário pode invocar ao clicar no botão — realizar chamadas de API, validar formulários, buscar informações de serviços e muito mais. Você pode imaginar a rapidez com que esse componente pode se tornar um inferno até em um aplicativo relativamente pequeno. O código desse componente de botão que encontrei (e depois refatorei) tinha mais de 2 mil linhas. Insano!

Quanto eu perguntei ao desenvolvedor que escreveu o código por qual motivo ele decidiu colocar toda essa lógica em um único componente, ele disse que era um "encapsulamento"

Vamos lembrar das qualidades que um bom componente deve ter:

Pequeno - esse botão com mais de 2 mil linhas de código não é pequeno. Além disso, ele aumentará sempre que alguém precisar de outro botão para uma ação diferente.

Focado - esse botão faz várias coisas sem qualquer relação e não pode ser chamado de focado.

Independente - esse botão é estreitamente acoplado a vários serviços e formulários, e qualquer mudança neles afetará o botão.

Testável - sem comentários.

Reutilizável - ele não é nada reutilizável. Você precisará modificar o código do componente sempre que quiser usá-lo para uma ação que ele não tiver e lidará com todas as ações desnecessárias e dependências desse botão.

Além disso, esse componente de botão oculta o botão HTML nativo, bloqueando o acesso do desenvolvedor às propriedades. Esse é um ótimo exemplo de como não escrever o código de componentes.

Vamos ver um exemplo bastante simples que usa esse componente de botão e tentar refatorá-lo com o padrão contêiner-apresentação.

```
@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) {}
}
```

Você pode ver que não há lógica nesse componente — o formulário está armazenado em um serviço, e o botão contém toda a invocação da lógica ao clicar nele. Portanto, agora, nosso botão tem toda a lógica não relacionada ao comportamento e é mais inteligente do que o componente pai, que está diretamente relacionado às ações processadas pelo botão.

Refatorar componente de botão com padrão contêiner-apresentação

Vamos separar as funções desses dois componentes. O botão deve ser um componente de apresentação — pequeno e reutilizável. O formulário de registro que contém o botão pode ser um componente contêiner com toda a lógica de negócio e as comunicações com a camada de serviços.

Não abordarei a parte com o botão nativo visível (provavelmente em um artigo futuro), mas focarei principalmente no aspecto arquitetural da relação entre esses dois componentes.

Componente de botão refatorado (apresentação)

```
@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();
```

```
  }
}
```

Como você pode ver, nosso botão refatorado é bastante simples. Só possui duas entradas e uma saída. As entradas são usadas para o recebimento de dados do componente pai e a exibição deles para o usuário (modifique a aparência do botão com classes e exiba o rótulo do botão). A saída é usada para o evento personalizado que será acionado sempre que o usuário clicar no nosso botão.

Esse componente é pequeno, focado, independente, testável e reutilizável. Ele não contém lógica não relacionada ao comportamento do próprio componente. Ele não tem ideia do funcionamento interno do aplicativo e pode ser importado com segurança e usado em qualquer parte do aplicativo ou até em outros aplicativos.

Componente de formulário de registro refatorado (contêiner)

```
@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);
  }
}
```

Você pode ver que nosso formulário de registro agora usa botões e reage ao evento de clique com o método de chamada registerUser. A lógica desse método está estreitamente relacionada a esse formulário, então é recomendável incluir aqui.

Esse é um exemplo bastante simples e a árvore de componentes só possui dois níveis. **Esse padrão apresenta alguns perigos quando a árvore de componentes tiver mais níveis.**

Exemplo mais sofisticado

Esse não é um exemplo do mundo real, mas espero que ajude a entender possíveis problemas com esse padrão.

Imagine uma árvore de componentes desta forma (de cima para baixo):

user-orders - componente de nível superior. É o componente contêiner que fala com a camada de serviços, recebe

os dados sobre o usuário e as ordens, transmite adiante na árvore e renderiza a lista de ordens.

user-orders-summary - componente de nível médio. É o componente de apresentação que renderiza a barra acima da lista de ordens do usuário com o número total de ordens.

cashback - componente de nível inferior (folha). É o componente de apresentação que exibe o valor total de cashback do usuário e tem um botão para transferir para a conta bancária.

Componente de contêiner de nível superior

Vamos analisar nosso componente de contêiner user-orders de nível superior.

```
@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>
```

Como você pode ver, o componente user-orders define 2 observables: user$ e orders$, usando async pipe no modelo para fazer a inscrição. Além de transmitir os dados para o componente de apresentação user-orders-summary, renderiza uma lista de ordens. Também se comunica com a camada de serviço reagindo ao evento personalizado requestCashbackWithdrawal emitido de user-orders-summary.

Componente de apresentação de nível médio

```
@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();
    }
}
```

Esse componente é projetado de maneira bastante semelhante ao componente de botão refatorado. Ele renderiza os dados recebidos pelas entradas e emite o evento personalizado com base em uma ação do usuário. Ele não chama serviços nem contém lógica de negócio. Portanto, é um componente de apresentação puro que usa outro componente de apresentação de cashback.

Componente de apresentação de nível inferior

```
@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();
    }
}
```

Esse é outro componente de apresentação que só recebe dados por entrada e gera eventos com saída. Bastante simples e reutilizável, **mas há alguns problemas na árvore de componentes**.

Você provavelmente percebeu que o componente user-orders-summary e cashback têm entradas semelhantes (cashbackBalanace e balance) e a mesma saída (requestCashbackWithdrawal). Isso ocorre porque nosso componente contêiner está muito longe do componente de apresentação mais profundo. Quanto mais níveis de

árvore com esse design, pior será o problema. Vamos analisar os problemas mais a fundo.

Problema 1 - Propriedades extrínsecas em componentes de apresentação de nível médio

O user-orders-summary recebe a entrada cashbackBalanace para transmitir à parte inferior da árvore, mas não usa ela sozinha. Se você se deparar com essa situação, esse é um dos indicadores de que você provavelmente tem um design de árvore de componentes com falhas. Componentes na vida real podem ter várias entradas e saídas e com esse design você terá várias entradas de "proxy", que deixará os componentes de nível médio ainda menos reutilizáveis (conforme você une a componentes filhos), e repetições de código.

Problema 2 - Bubbling de eventos personalizados de componentes de nível inferior a superior

Esse problema é muito parecido com o anterior, mas está relacionado às saídas do componente. Como você pode ver, o evento personalizado requestCashbackWithdrawal está repetido nos componentes cashback e user-orders-summary. Novamente, isso ocorre porque o componente contêiner está muito longe do componente de apresentação mais profundo. Isso também impede que o componente médio seja reutilizado sozinho.

Existem pelo menos duas soluções possíveis para esses problemas.

1º – torne os componentes de nível médio mais agnósticos a conteúdo usando ngTemplateOutlet e revele os componentes mais profundos diretamente aos componentes contêiner. Pularemos isso hoje, já que merece um artigo separado.

2º – reformule a árvore de componentes.

Refatorando a árvore de componentes

Vamos refatorar nosso código para ver como podemos resolver os problemas com as propriedades extrínsecas e o bubbling de eventos no componente de nível médio.

Componente de nível superior refatorado

```
@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>
```

Removemos o observable user$ e o método onRequestCashbackWithdrawal() do componente contêiner de nível superior. Está muito mais simples agora e só transmite os dados necessários para renderizar o próprio componente user-orders-summary, mas não o componente filho cashback.

Componente de nível médio refatorado

```
@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[];
}
```

Também está bastante simplificado. Agora, só tem uma entrada e renderiza o número total de ordens.

Componente de nível inferior refatorado

```
@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 */);
    }
  }
```

```
}
```

Uau. Como você pode ver, **não é mais de apresentação**. Agora, é bastante semelhante ao componente de nível superior, então o componente contêiner está na parte inferior da árvore. Essa refatoração permitiu simplificar o design inteiro da árvore de componentes e as APIs e a lógica dos nossos dois componentes no topo da árvore.

E quanto à reusabilidade do novo componente cashback? Ainda é reutilizável, já que contém apenas a lógica relacionada ao próprio componente, então permanece independente.

O novo design da nossa árvore de componentes parece ser mais fácil de manter, mais otimizada e atomizada. Não há mais bubbling de eventos e entradas repetidas na árvore de componentes, e o design geral está muito mais simples. Conseguimos isso ao colocar o componente contêiner adicional na parte inferior da árvore de componentes. Esse método pode ser usado para simplificar o design da sua árvore de componentes, mas você precisa ter uma boa compreensão de quais componentes na árvore podem ser contêineres sem grande perda na reusabilidade e quais devem ser apenas componentes de apresentação. Essa é sempre uma questão de equilíbrio e escolhas design ao criar a arquitetura do app.

É muito fácil entender errado o padrão contêiner-apresentação e pensar que componentes contêiner só podem ser de nível superior (intuitivamente, eles contêm todos os outros componentes na árvore de componentes local). No entanto, esse não é o caso. Os componentes contêiner podem estar em qualquer nível da árvore de componentes e, como você viu, até no nível da folha. Gosto de chamá-los de componentes **inteligentes**, porque, para mim, é muito claro que eles terão lógica de negócio e poderão estar em qualquer lugar da árvore de componentes.

Palavras finais

Espero que agora você tenha uma visão melhor do padrão contêiner-apresentação e dos possíveis problemas da implementação.

Tentei manter o mais simples possível, mas há muitas informações disponíveis relacionadas a esse padrão.

Em caso de dúvidas ou observações, não hesite em entrar em contato nos comentários.

O próximo artigo será sobre changeDetectionStrategy no Angular (que tem grande relação com esta postagem).

Até mais!

#Angular #Angular2 #Desenvolvimento de IU (Interface do Usuário) #Frontend #Guia de Codificação #Outro

---

URL de origem:https://pt.community.intersystems.com/post/angular-componentes-de-cont%C3%AAiner-e-apresenta%C3%A7%C3%A3o-burro-inteligente