Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Padrões em Front-ends

Francisco Eugênio Wernke

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: Prof. Dr. Fabio Kon
Cossupervisor: Me. Renato Cordeiro Ferreira
Cossupervisor: Me. João Lino Daniel

São Paulo
2023

Resumo

Francisco Eugênio Wernke. **Padrões em Front-ends**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Padrões de software são conhecidos e utilizados há mais de 40 anos. Os padrões ajudam engenheiros de software a compilar conhecimentos a partir de experiências vividas ao longo de suas carreiras, compartilhando esse conhecimento. Por isso, padrões são um importante ferramental para novos programadores. Os padrões são constituídos de três componentes: contexto, problema e solução. Eles podem ser usados para solucionar diversos problemas de design. Cada área da engenharia de software, como o back-end e o front-end, tem seus próprios padrões e respectivas variantes. Dado este cenário, esta pesquisa pretende amadurecer a comunidade de desenvolvimento front-end a partir do uso de padrões. Para isso, dez documentações sobre os padrões mais relevantes para a comunidade front-end foram criadas. A pesquisa começou separando vinte e um padrões da literatura associados ao desenvolvimento front-end. Para filtrar os dez mais relevantes, foram feitas dez entrevistas com programadores com cinco anos ou mais de experiência em desenvolvimento front-end. A partir dos dados dessas entrevistas, coletados por formulários que avaliavam cada padrão em cinco tópicos, foi feita a escolha dos dez mais bem avaliados. Com essa lista de padrões, foram feitas as documentações propostas contendo: 1. As informações básicas sobre os padrões como nome, problema de design que se propõe a resolver e solução; 2. Um diagrama de funcionamento do padrão contendo seus componentes e a relação entre eles; 3. Uma implementação em código do mesmo, quando fizesse sentido. Essa documentação foi disponibilizada publicamente em: https://franwernke.github.io/TCC-Design-Patterns. A pesquisa gerou uma coletânea de padrões relevantes para desenvolvedores front-end, bem como um material de fácil entendimento para iniciantes da área. Essa documentação pode ser usada para se produzir um curso completo de padrões em desenvolvimento front-end.

Palavras-chave: Padrões. Padrões de Projeto. Padrões Arquiteturais. Expressões Linguísticas. Desenvolvimento Web. Front-End. Back-End.

Abstract

Francisco Eugênio Wernke. **Front-end Patterns**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Software patterns have been known and used for over 40 years. Patterns help software engineers compile knowledge from experiences acquired throughout their careers, sharing this knowledge with their peers. Therefore, patterns are an important tool for new programmers. Patterns consist of three components: context, problem, and solution. They can be used to solve various design problems. Each area of software engineering, such as back-end and front-end, has its own patterns and respective variants. Given this scenario, this research aims to mature the front-end development community through the use of patterns. To achieve this, ten documentations on the most relevant patterns for the front-end community were created. The research began by identifying twenty-one patterns from the literature associated with front-end development. To filter the top ten patterns, ten interviews were held with programmers with five or more years of experience in front-end development. Using the data from these interviews, collected through forms that evaluated each pattern on five topics, the top ten patterns were chosen. With this list of patterns, the proposed documentations were created containing: 1. basic information about the patterns, such as name, design problem they aim to solve, and solution; 2. a working diagram of the pattern containing its components and the relationship between them; 3. an implementation in code when it made sense. This documentation was made publicly available at: https://franwernke.github.io/TCC-Design-Patterns. This research has generated a collection of relevant patterns for front-end developers, as well as an easily understandable material for beginners in the field. This documentation can be used to create a complete course on patterns in front-end development.

Keywords: Patterns. Design Patterns. Architectural Patterns. Idioms. Web Development. Front End. Back End.

Lista de tabelas

Lista de figuras

Sumário

Apêndices

Capítulo 1

Introdução

Para criarmos produtos digitais, precisamos formar vários profissionais qualificados. No ano de 2021, existiam cerca de 25,6 milhões de desenvolvedores web no mundo. Espera-se que, até o final de 2024, existam 28,7 milhões (Vailshery 2023). Estes profissionais precisam aprender boas práticas de mercado para aumentarem sua produtividade e então se destacarem.

Nesse cenário competitivo, algumas técnicas da Ciência da Computação utilizadas por profissionais, podem ser uma vantagem para os recém-formados. Este estudo foca no uso de padrões de projeto, padrões arquiteturais e expressões idiomáticas como alavancas para a produção de software com alta qualidade bem como o aumento de produtividade de desenvolvedores.

O ensino de padrões é um caminho rápido para o aprendizado das boas práticas de engenharia de software, trazendo consigo a experiência de inúmeros desenvolvedores (Shvets 2021b). Dessa forma, o objetivo desta pesquisa é **facilitar o acesso da comunidade iniciante em front-end a padrões, bem como melhorar a compreensão desses, por meio da criação de um catálogo de padrões relevantes aos estágios iniciais do desenvolvimento front-end**.

Esta monografia conta com cinco capítulos, sendo o Capítulo 1 a introdução, além de um anexo. Nos Capítulos 2 e 3, será explicitado o fundamento teórico do projeto, explicando as definições de padrões adotadas, explicitando o que são front-ends e o que são padrões de front-end. No Capítulo 3, será explicitada a metodologia do trabalho, definições sobre as entrevistas, métodos para a classificação dos padrões e a base para a confecção das documentações. No Capítulo 4, mostram-se os resultados das entrevistas e a documentação produzida. No Capítulo 5, termina-se o texto com as conclusões do trabalho e possíveis

trabalhos futuros. Por fim, o Anexo A é o protocolo de entrevista usado pela pesquisa.

Capítulo 2

Front-Ends

Este capítulo traz uma visão geral sobre o que é e como funciona o desenvolvimento front-end hoje, incluind quais tecnologias são centrais para essa área.

2.1 História do Desenvolvimento Web

No fim dos anos 1980, o avanço da tecnologia dos computadores possibilitou uma revolução no compartilhamento de informações entre universidades. A criação da primeira especificação da Web em 1989 pelo cientista Tim Berners-Lee possibilitou várias implementações de sistemas que serviam hipertextos via Internet. Dessa forma, os cientistas podiam escrever artigos em uma linguagem de hipertextos e disponibilizá-los mantendo versões confiáveis do conhecimento adquirido (Berners-Lee 1989).

A primeira proposta da Web levava em conta **servidores** que produziriam hipertexto capaz de ser lido e traduzido em artigos, gráficos, imagens e diversos elementos visuais no computador que receberia esse hipertexto. O software local que exibiria o artigo a partir do hipertexto foi chamado de **navegador** e requeria um formato padronizado de hipertexto.

A partir dessa definição, a criação de navegadores para a leitura de hipertextos cresceu rapidamente. A criação do primeiro navegador, "WorldWideWeb" (WWW), permitia a exibição de hipertextos no padrão desenvolvido por Tim Berners-Lee e o hipertexto era entregue por um servidor web também desenvolvido por ele. O WWW permitia a edição desses hipertextos e era a principal ferramenta de atualização das documentações feitas especificamente para o contexto da CERN.

O processo de criação da Web definiu uma divisão entre o software que funcionaria do

lado do servidor e o software que funcionaria no lado do cliente. Nas palavras do próprio Tim Berners-Lee: "*A única maneira pela qual podemos encorporar flexibilidade suficiente [ao sistema] é separando o software de armazenamento de informação do software de exibição, com uma interface bem definida entre eles*" (Berners-Lee 1989). Dessa forma, a separação entre software de exibição (front-ends) e software de armazenamento (back-ends) era intrínseca a Web desde seus primórdios.

2.2 O que são Front-Ends?

Existem algumas definições para o conceito de front-end (Cambridge 2023, AWS 2023, Souto 2023). Em desenvolvimento web, **front-ends** são componentes de software que lidam com a interface do usuário. Esses componentes constroem e organizam as interfaces, além de lidar com interações do usuário. Eles também conversam com os componentes do back-end que lidam com o armazenamento de dados e implementam regras de negócio da aplicação.

Front-ends possuem três grandes responsabilidades:

1. Criar uma interface que forneça informações sobre os dados de interesse do usuário de forma fácil e rápida. As interfaces devem criar uma experiência de uso que satisfaça as necessidades do usuário sem obstáculos, ter elegância e simplicidade (Norman e Nielsen 2023).

2. Estruturar a apresentação dos dados para facilitar o consumo destes pelos usuários. Diferentes front-ends podem consumir a mesma fonte de dados, mas terem propósitos diferentes. Dessa forma, é importante que se criem diferentes organizações para os mesmos dados.

3. Prover formas de entrada de dados como botões, formulários, listas, entre outros, para que os usuários possam modificar os dados apresentados. Para criar uma experiência amigável ao usuário e simplificar os back-ends, os front-ends atuam como uma camada de abstração às operações do back-end e outros mecanismos de mudança de estado do modelo.

Dadas as necessidades específicas dos front-ends, existem padrões que são mais comumente utilizados dentro do contexto do desenvolvimento front-end.

2.3 Cenário atual do desenvolvimento front-end

No desenvolvimento de front-ends, existem algumas ferramentas que se destacam como as principais do mercado. O *Hypertext Markup Language* (HTML), o *Cascading Style Sheets* (CSS) e o JavaScript (implementação do padrão ECMAScript) são tecnologias pilares para criar front-ends (Sobczak 2023). HTML e CSS são, respectivamente, linguagens de marcação (Docs 2020b) e de folha de estilos (Docs 2020a), enquanto Javascript é uma linguagem de programação.

2.3.1 HTML

A primeira versão do HTML foi criada logo depois da Web por Tim Berners-Lee em 1991. O propósito do HTML era ser uma linguagem para a criação de páginas Web, definindo o conteúdo da página. O HTML funciona por meio de marcações nomeadas que estruturam trechos do conteúdo (tags), transformando-os em um hipertexto. Atualmente, a versão mais usada do HTML é a 5, que apresenta inúmeras melhorias em relação às versões anteriores.

O Programa 2.1 mostra um exemplo de HTML.

Programa 2.1 Exemplo de arquivo HTML.

```
1  <div>
2    Aqui começa uma divisão
3    <h1> Isto eh um título </h1>
4    <p> Isto eh um parágrafo </p>
5    A tag com "/" termina a marcação
6  </div>
```

2.3.2 CSS

O CSS foi desenvolvido em 1995 por Håkon Wium Lie, que também trabalhava na CERN (Bos 2016). O objetivo principal do CSS era possibilitar a adição de estilos a páginas Web de forma simples, considerando as limitações dos dispositivos que apresentavam as interfaces. Apesar de existirem algumas implementações diferentes dessa ideia, o CSS tinha como ideal mediar as intenções do autor da página web com as preferências do leitor da página. Assim, o arquivo CSS era criado com os requisitos do autor mas poderia produzir diferentes resultados caso as limitações do navegador ou preferências do leitor da página divergissem (Bos 2016).

Atualmente, a versão do CSS mais utilizada é a 3 que é compatível com todos os principais navegadores (Wells 2018).

2.3.3 JavaScript

Criado em 1995 por Brendan Eich, que trabalhava na empresa Netscape, JavaScript (JS) foi uma das primeiras iniciativas para alocar parte do processamento das páginas Web no lado do cliente. Até então, somente os servidores Web tinham controle sobre o conteúdo da página. Uma vez gerado o hipertexto, não era possível alterá-lo até que outra página fosse requisitada ao servidor. Dessa forma, o JS possibilitou a adição de comportamento às páginas no navegador do usuário.

Dois anos depois da criação do JavaScript, em 1997, a Netscape começou a trabalhar junto da ECMA International, organização sem fins lucrativos europeia que define padrões para sistemas de software. Dessa colaboração, surgiu o ECMAScript, padrão de interfaces e design de linguagens que poderia ser usado por navegadores como referência para a implementação de seus interpretadores. Com essa padronização, surgiram algumas outras implementações do ECMAScript, como o JScript da Microsoft e o ActionScript da MacroMedia - que seria posteriormente adquirida pela Adobe Systems (Rauschmayer 2014).

Devido à sua criação ter sido amplamente relacionada ao Java, a sintaxe do JS é bastante similar ao Java. Porém, várias decisões estruturais de design das linguagens são diferentes. O JS, apesar de ser uma linguagem multiparadigma, é uma linguagem mais voltada a programação funcional que a orientação a objetos. A tipagem dinâmica, a reflexão de objetos irrestrita e a falta de tratamento de exceções nativo, dentre outros, são características que tornam o JS uma linguagem de rápido aprendizado, mas que não reforça boas práticas. Por isso, a adoção de padrões é extremamente importante.

2.4 JavaScript nesta Pesquisa

Por ser central no desenvolvimento front-end, esta pesquisa focou na implementação dos padrões em JavaScript, trazendo expressões idiomáticas dessa linguagem. Isso não significa que os padrões (arquiteturais, de projeto e até mesmo variações das expressões idiomáticas) apresentados aqui não podem ser implementados em outras linguagens.

A trindade da Web - HTML, CSS e JavaScript - está no centro de todas as tecnologias de desenvolvimento front-end. Porém, a partir da criação de bibliotecas e arcabouços para o desenvolvimento de interfaces, é comum que não seja necessário escrever diretamente nessas linguagens. A biblioteca *React.js* foi a mais popular no ecossistema JavaScript para front-ends em 2022 (JavaScript 2022). Ela auxilia na criação de HTML no JavaScript (seguindo o formato JSX), sem necessidade de escrevê-lo diretamente.

Capítulo 3

Padrões

Este capítulo explora as definições de padrão e as diretrizes para a documentação de um.

3.1 Analogia com Feng Shui

O Feng Shui é uma pseudociência inventada na China antiga que prega um conjunto de regras para organizar casas, vilarejos e até mesmo tumbas. As regras estabelecidas pela técnica não eram testadas cientificamente. No entanto, serviam para auxiliar a construção de grandes estruturas, convencendo os arquitetos a seguirem um caminho único. O Feng Shui absorvia a experiência coletiva dos arquitetos chineses e os ajudava a ter um sistema único que, frequentemente, funcionava (He e Luo 2000).

De forma similar, padrões auxiliam na construção de software, como já foi averiguado em alguns casos na construção de interfaces com boa experiência de usuário (Nuval 2020). Os padrões não são soluções prontas, como algoritmos ou bibliotecas, mas sim guias para soluções existentes testadas através dos anos por engenheiros de software Buschmann 1996. Padrões captam algumas características da solução que vão além da corretude, passando também pela reusabilidade, manutenibilidade, facilidade de entendimento, entre outras.

3.2 Definição Adotada

Esta pesquisa adota a seguinte definição: "*padrões descrevem um problema recorrente de projeto que surge de contextos específicos e apresentam um esquema genérico e experimentado de uma solução*" (Buschmann 1996). Essa definição encaixa diversas estruturas comuns no

desenvolvimento de software, que são criadas até mesmo sem a consciência do autor do código. Alguns exemplos de padrões são:

1. Model-View-Controller
2. Model-View-ViewModel
3. Builder
4. Singleton

Um padrão é composto de três componentes: problema, contexto e solução. Enquanto um padrão descreve a solução para um problema, ele deve considerar o contexto em que o problema é encontrado. O contexto apresenta requisitos funcionais que são denominados "forças" (Buschmann 1996). Todas as forças de um problema-contexto devem ser balanceadas, de forma a criar uma solução específica e detalhada.

3.3 Documentação de um padrão

Existem diversas formas de descrever um padrão, uma delas é descrevendo as seguintes características do padrão:

- **Nome:** Todo padrão possui um conjunto de nomes que o identificam. A nomenclatura é um dos pontos mais relevantes do padrão, pois é por ela que se comunica a estrutura declarada pelo padrão.
- **Exemplo:** Um exemplo prático do problema que o padrão descreve, destacando a necessidade de um padrão que o resolva.
- **Contexto:** Situações genéricas em que o padrão pode ser aplicado.
- **Problema:** O problema que o padrão descreve. Aqui, apresentam-se as forças que podem estar envolvidas e os pontos nos quais o problema atinge o software.
- **Solução:** A solução proposta pelo padrão, destacando como ela balanceia as forças apresentadas no problema e sua estrutura geral em software. Incluem-se também esquemas que explicam a dinâmica entre os componentes da solução e suas relações.
- **Implementação:** Um guia de como implementar o padrão de forma concreta. Pode ser uma implementação real em código ou um guia representando sua estrutura em código. É importante ressaltar que essa estrutura pode variar dependendo do contexto do problema.

- **Variantes:** Algumas variantes do padrão que podem ser refinamentos (para contextos mais complexos) ou mudanças estruturais no padrão (para servir contextos diferentes).

3.3.1 Exemplo completo: Singleton

Nome: Singleton

Exemplo: Em sistemas onde registram-se informações sobre os processos que estão acontecendo, geralmente são usados registros em texto (*logs*) para anotar acontecimentos. Usualmente, cria-se um arquivo no sistema operacional no qual todos os registros do sistema são centralizados. Dessa forma, não é interessante que vários objetos interajam com o arquivo, pois, isso poderia gerar problemas de concorrência, levando ao registro da sequência errada dos fatos e potencial duplicidade de informações.

Contexto: Classes que tenham a restrição de ter somente uma instância ou que precisem de um ponto único de acesso global àquela instância.

Problema: Uma classe que realiza uma função única dentro do sistema precisa ser instanciada diversas vezes em vários fluxos diferentes. Porém, essas instâncias precisam organizar e comunicar mudanças de estado entre si. Dessa forma, é criada uma rede de informações complexa entre as instâncias. Além disso, se essa classe controla um recurso limitado, como um banco de dados ou um arquivo, deve-se controlar o acesso de cada instância a esse recurso para não ocorrer a sobrescrita de alterações.

Solução: A limitação das instâncias de uma classe permite um controle maior sobre as ações possíveis para as instâncias. A classe implementa um método especial para a instanciação que sempre retorna a mesma instância em vez de criar uma nova. Dessa forma, todas as outras classes trabalham com referências para a mesma instância e o estado dela é comunicado a todos os usuários. Assim, o acesso aos recursos da instância é controlado via um ponto de acesso único a ela.

Implementação: Para implementar o Singleton, devemos criar método estático **createInstance**. Esse método deve criar uma instância nova caso nenhuma exista, ou devolver um referência para a pré-existente caso contrário. O construtor da classe deve ser privado. Após a criação do método, definem-se atributos e métodos que serão a interface da instância.

Um exemplo de implementação em JavaScript pode ser encontrado no Programa 3.1.

Variantes: O padrão Module pode ser implementado como diversos Single-

Programa 3.1 Implementação de Singleton em JavaScript.

```
const Logger {
  public const numberOfLogsCreated;

  private constructor() {
    numberOfLogsCreated = 0;
  }

  let instance: Logger;

  static function createInstance() {
    if (instance === undefined) {
      return new Logger();
    }
    return instance;
  }

  function info() {
    //...escreve para um arquivo ou qualquer implementação
    numberOfLogsCreated += 1;
  }
};
```

TONS.

3.4 Escopo de Padrões

Ao definir um padrão, também deve-se pensar no escopo dele no projeto e qual o seu nível de influência sobre o software. Alguns padrões, se implementados, terão impacto sobre toda a estrutura do sistema, definindo relações macroscópicas entre componentes ou grupos de componentes. Outros terão efeito somente em um componente e sua estrutura, impactando de forma granular o sistema.

Neste viés, os padrões podem ser classificados em três grandes grupos:

- **Padrões Arquiteturais**

Um padrão arquitetural define uma organização estrutural fundamental para o sistema. Ele provém um conjunto de subsistemas pré-definidos, especifica as responsabilidades deles e inclui regras e guias para as relações entre eles (Buschmann 1996).

Exemplos: Model-View-Controller, Model-View-ViewModel, Layers e Distributed Systems ou Microsserviços.

- **Padrões de Projeto**

Um padrão de projeto define um esquema para refinar subsistemas ou componentes de

um sistema ou as relações entre eles. Ele descreve uma estrutura para resolver problemas de implementação dentro de um contexto particular do sistema. (Buschmann 1996)

Exemplos: Observer, Constructor, Adapter, Proxy.

- **Expressões Idiomáticas ou Linguísticas (*Idioms*)**

Uma expressão idiomática é um padrão de baixo nível de abstração que geralmente é implementado dentro de uma linguagem de programação ou arcabouço (Shvets 2021a). Estes padrões descrevem detalhes sobre componentes e relações entre subsistemas usando funcionalidades da própria linguagem (Buschmann 1996).

Exemplos: Counted Pointer (em C++), Short-circuit Operators (em JavaScript).

Apesar de existir um grande universo de padrões e variantes, alguns conjuntos nasceram para resolver problemas de um domínio específico. Um exemplo são padrões de microsserviços (Richardson 2015). Seguindo essa linha, esta seção explora o que são os padrões de front-end, tanto aqueles específicos para o domínio, quanto outros mais genéricos que se encaixam nesse domínio.

3.5 Padrões de Front-end

Os padrões descritos pelo livro *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma *et al.* 1994) podem ser utilizados em inúmeros contextos diferentes. Os problemas e soluções descritas pelos quatro autores, conhecidos como a "Gangue dos Quatro" (em inglês *Gang of Four*, GoF), dão uma base geral para a construção de software orientado a objetos, mas não entram em domínios específicos como a construção de interfaces (Gamma *et al.* 1994, Guru 2023).

Dessa forma, os padrões GoF também podem ser usados para front-ends. A diferença é que há um contexto mais específico para esses padrões. Adotam-se, então, variações dos padrões originais que melhor se encaixem para front-ends.

Por outro lado, dentro do domínio de construção de interfaces, existem algumas vertentes que lidam com diferentes preocupações do front-end. Para experiência do usuário, por exemplo, o livro *Designing interfaces* (Tidwell 2011) descreve padrões na construção visual de interfaces que maximizam a simplicidade e usabilidade da tela. Esse livro contribuiu para a ideia inicial de uma interface e seu design.

Nesta pesquisa, as questões de *User Experience* (UX) não serão tratadas. Abordam-se mais extensamente dois tipos de padrões: organização de código e arquitetura dos dados

dentro do front-end.

Na categoria de padrões de front-end, temos alguns padrões que já surgiram com contextos e problemas específicos. Um exemplo é o padrão Back-end For Front-end. Esse padrão arquitetural surgiu como resposta a um problema gerado pelo estilo arquitetural de microsserviços usado no back-end. Como surgiram diversas APIs que o front-end tinha de buscar informações, o uso de banda para buscar informações no back-end ficou muito alto, devido as diversas requests necessários. Além disso, mudanças de contrato nos vários back-ends refletiam diretamente no front-end. Nesse ponto, surgiu o Back-end For Front-end, que atua como um intermediário entre os diversos serviços e o front-end (Calçado 2015).

Ainda dentro dos padrões que surgiram no contexto do front-end, há algumas expressões linguísticas específicas de arcabouços destinados ao desenvolvimento front-end. Na biblioteca React.js, por exemplo, existem alguns padrões próprios, tais como os padrões useContext e query observer.

Dessa forma, podemos concluir que existem padrões que são mais específicos e relevantes para o desenvolvimento front-end, por: ou terem nascido do contexto de desenvolvimento de interfaces, ou serem específicos de arcabouços usados em front-ends, ou mesmo serem variantes pensadas para front-ends. Esta pesquisa foca nesses padrões, cuja definição será discutida no Capítulo 4.

Capítulo 4

Metodologia

Este capítulo descreve quais objetivos específicos foram traçados para cumprir o objetivo explicitado no Capítulo 1. Estes objetivos específicos levaram ao desenvolvimento de uma série de passos, técnicas usadas e decisões tomadas para que esta pesquisa os realizasse. O objetivo deste capítulo é tornar possível a reprodução desta pesquisa.

4.1 Objetivos específicos

O objetivo do projeto é **facilitar o acesso da comunidade iniciante em front-end a padrões, bem como melhorar a compreensão desses, por meio da criação de um catálogo de padrões relevantes aos estágios iniciais do desenvolvimento front-end**. Para atingí-lo, , foram traçados alguns objetivos menores e específicos que devem ser alcançados, tais como:

1. definir critérios que possam avaliar a relevância do padrão para a comunidade desenvolvedora, principalmente para profissionais mais experientes;
2. criar uma lista de padrões a partir de referências como livros, blogs e documentações;
3. entrevistar profissionais experientes com front-ends para determinar quais padrões melhor preenchem os critérios definidos no Item 1;
4. criar uma documentação, com um molde bem definido, sobre cada padrão de uma lista selecionada;
5. disponibilizar esta documentação publicamente.

Ao alcançar esses objetivos, esta pesquisa contribui para o amadurecimento e aprendizado da comunidade de desenvolvimento front-end por meio de padrões.

4.2 Critérios para Avaliação dos Padrões

Um padrão tem várias características que são derivadas do problema e do contexto que o definem. Porém, é importante notar que nenhum padrão resolve qualquer problema. Além disso, contextos mais restritivos podem tornar o padrão pouco útil, adicionando complexidade excessiva ou desbalanceando suas forças. Dessa forma, avaliar a relevância e utilidade de padrões em uma escala única é uma tarefa complicada.

Contudo, dentro do contexto de padrões de front-end, especificado na Seção 3.5, é possível construir uma noção mais restrita sobre o que é um padrão, definir características que indicam relevância e utilidade. Assim, um padrão pode ser classificado como relevante ou não para desenvolvedores front-end.

Nesta pesquisa, as métricas escolhidas para avaliar se um padrão é relevante para o desenvolvimento front-end são:

- **Conhecimento do padrão:** Essa métrica avalia a popularidade do padrão. A expectativa é que a maioria dos padrões sejam conhecidos pois os padrões pré-selecionados já tem certa relevância para a área.
- **Frequência na literatura:** Essa métrica avalia quais padrões podem ser facilmente encontrados na literatura. A ideia é que padrões que são frequentemente documentados, seja pela literatura cinzenta (como blogs e vídeos) ou acadêmica (como artigos e livros), são importantes para os desenvolvedores. Além disso, se um desenvolvedor procura por literatura sobre esses padrões, a indústria de software atualmente exige a implementação desses padrões.
- **Simplicidade de implementação:** Essa métrica avalia o quão rápido chega-se do problema à solução utilizando esse padrão. Essa métrica é representada pela simplicidade de implementação, pois se o padrão resolve o problema e pode ser facilmente implementado, então o tempo entre a concepção e solução é pequeno. Com essa métrica, é possível separar os padrões mais eficazes e esses deveriam aparecer na documentação gerada pela pesquisa.
- **Facilidade de manutenção:** Essa métrica avalia o quanto um padrão ajuda na organização do código e, consequentemente, em um código fácil de ser estendido e modificado conforme surgem novas necessidades. A facilidade de manutenção avalia características centrais do padrão como o apoio a experiência do desenvolvedor.
- **Frequência de uso em projetos:** Essa métrica avalia a frequência com que o padrão é implementado. Ela também pode ser estendida para avaliar se a pessoa

que implementou viu resultados positivos na sua implementação e decidiu utilizar o padrão novamente.

4.3 Padrões Pré-Selecionados

A lista pré-selecionada de padrões considerou padrões que, na literatura, são relacionados à construção de interfaces. Dentre eles, é comum que alguns sejam relacionados a bibliotecas e arcabouços populares no front-end (como o *React.js*) devido à importância desse arcabouço no cenário atual de desenvolvimento front-end.

Para cada padrão, serão apresentados um nome (podem existir variantes), uma breve descrição do problema e da solução. Essa descrição é a mesma que foi apresentada aos entrevistados. A lista completa se encontra na Tabela 4.1.

4.4 Entrevistas

As entrevistas são uma parte importante desta pesquisa. Esta seção destrincha o método de entrevista e seus objetivos.

4.4.1 Escolha dos Convidados e Realização das Entrevistas

Devido à importância das entrevistas na pesquisa, os convidados foram selecionados para que as respostas sejam condizentes com a realidade atual do desenvolvimento front-end.

Sendo assim, a experiência dos entrevistados em desenvolvimento front-end foi priorizada na escolha. Os convidados têm no mínimo cinco anos de experiência em front-end. Entre eles, existem diferentes contextos de desenvolvimento, tais como: sites para navegadores comuns, apps para Android e IOS, interfaces para desktops, entre outros.

Depois da escolha dos profissionais, a entrevista foi conduzida de forma padronizada para todos os entrevistados. O protocolo completo da entrevista, incluindo fotografias do formulário, pode ser encontrado no Capítulo 6.

As diretrizes para a entrevista pretendiam classificar cada padrão da lista pré-selecionada dentro dos cinco tópicos destrinchados no Seção 4.2. Para isso, foi criado um formulário pela plataforma *Google Forms*, que permite a coleta e armazenamento as

Nome	Descrição
Back-end for Front-end	Criação de serviços back-end específicos para interfaces front-end, permitindo uma melhor adaptação e controle das necessidades do front-end
Centralized State Management	Cria uma store centralizada para manter o estado da aplicação
Module pattern	Um padrão que encapsula funcionalidades em módulos independentes, protegendo o escopo de variáveis e evitando poluição do escopo global
MVC	Divide a aplicação em Model, View e Controller com papeis bem definidos
MVVM	Separa a interface do usuário (View) dos dados (Model) com um intermediário chamado ViewModel
Micro Frontends	Divide uma aplicação front-end em módulos menores, independentes e autônomos que criam a interface em conjunto
Provider	Um padrão que fornece dados ou funcionalidades a componentes front-end, muitas vezes utilizado em contextos de gerenciamento de estado
Smart-dumb components	Distingue entre componentes inteligentes (smart) que têm lógica e componentes burros (dumb) que são puramente de apresentação
Proxy	Age como intermediário entre objetos, permitindo controlar o acesso a eles
Observer	Define uma relação de um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus observadores são notificados e atualizados automaticamente
Prototype	Permite criar objetos a partir de protótipos, economizando recursos de criação de objetos semelhantes
Adapter	Permite a interface de uma classe ser compatível com outra, facilitando a integração de diferentes sistemas
Decorator	Adiciona funcionalidades a objetos existentes de forma dinâmica, sem modificar sua estrutura básica
Singleton	Garante que uma classe tenha apenas uma única instância e fornece um ponto de acesso global para essa instância

Command	Encapsula uma solicitação como um objeto, permitindo a parametrização de clientes com solicitações, fila de comandos e execução assíncrona
Chain of Responsibility	Permite passar solicitações ao longo de uma cadeia de manipuladores, cada um decidindo se processa a solicitação ou a passa para o próximo na cadeia
useContext	Padrão usado em React para compartilhar dados entre componentes sem passar as propriedades manualmente por todos os níveis da hierarquia de componentes
Data Binding	Automatiza a sincronização de dados entre a camada de modelo e a interface do usuário, mantendo-os automaticamente atualizados
Query observer	Gerencia o estado e os dados de consultas em aplicações React
Promise based async	Uso de Promises para lidar com operações assíncronas de forma mais clara e estruturada
Render Prop	Uso de uma propriedade especial em componentes React para passar funções que retornam elementos JSX, permitindo a reutilização de lógica de renderização

Tabela 4.1: *Lista dos padrões pré-selecionados*

respostas dos entrevistados. Depois das entrevistas, a plataforma gera gráficos para a visualização dos dados, expostos no Capítulo 5.

4.4.2 Compilação dos Dados

Para a análise dos dados gerados pelas entrevistas, foi escolhida a plataforma Jupyter hospedada no Google Colab. A ferramenta usa Python como linguagem de programação. Essa escolha se deu pela facilidade e experiência do pesquisador com o ecossistema.

Após as entrevistas, os dados coletados via formulário, armazenados no Google Forms, foram baixados e armazenados localmente. A partir de então, os seguintes passos foram realizados para a análise destes dados:

1. exportar os dados do Google Forms para um objeto Python por meio da biblioteca Pandas;
2. categorizar os dados por métrica e padrão, sendo as colunas as métricas e as linhas os padrões;

3. normalizar as colunas para que todas as métricas tenham o mesmo impacto sobre o resultado final;

4. somar as pontuações normalizadas em cada métrica para cada padrão, resultando em um número, a nota do padrão;

5. ordenar as notas para que os dez primeiros padrões sejam escolhidos para a lista final.

4.5 Modelo de Documentação

As documentações produzidas foram agrupadas pelo escopo do padrão sendo as categorias: “Padrão Arquitetural”, “Padrão de Projeto” e “Expressões Linguísticas”.

Para cada padrão, há uma página HTML que descreve suas características, explicitadas na Seção 3.2. Com essas características, é possível implementar o padrão onde for necessário e definir para que problemas ele pode ser usado.

A documentação possui uma coluna somente com os seguintes tópicos em ordem:

- Nome
- Problema
- Exemplo
- Contexto
- Solução
- Diagrama da solução
- Código da implementação em JS (Caso faça sentido para o padrão)
- Variantes (Caso existam e sejam relevantes)

Os exemplos em código são implementados em JavaScript seguindo o ECMAScript 2022 (ES2022), versão mais recente do ECMAScript durante o desenvolvimento desta pesquisa (International 2023). Alguns padrões arquiteturais podem ser complexos demais para a implementação em código ou serem melhor aproveitados como conceitos de organização. Nesses casos, apenas o diagrama da estrutura do código será apresentado.

4.6 Disponibilização das Documentações

As documentações serão disponibilizadas, temporariamente, no mesmo domínio que a página desta pesquisa, em https://franwernke.github.io/TCC-Design-Patterns.

A organização da página tem dois níveis: escopo do padrão e nome do padrão. Dentro de cada escopo, existem todos os padrões documentados. Além disso, foram adicionados marcadores às páginas, que ajudam na pesquisa tanto via buscador da própria página quanto via buscadores da Web.

Capítulo 5

Resultados

Neste capítulo são apresentados os resultados alcançados pela pesquisa. Primeiro, o capítulo introduz resultados das entrevistas realizadas e como eles implicam na lista de padrões selecionados. Depois, apresenta as documentações geradas são apresentadas em sua íntegra.

5.1 Entrevistas

Após as entrevistas, a plataforma Google Forms gerou do Figura 5.1 ao Figura 5.5 para cada métrica e seu resultado. No eixo Y, temos todos os padrões da lista pré-selcecionada. Já no eixo X, temos a quantidade de respondentes que afirmaram que o padrão em Y é bem avaliado naquela métrica, seguido pela porcentagem dos respondentes que marcaram aquela opção.

5.2 Padrões Selecionados

Para a normalização dos dados, foram escolhidas duas técnicas diferentes: o Z-Score e o Min-Max. Os dois métodos tiveram resultados próximos. No entanto, perto da média, eles selecionaram padrões diferentes para a lista final. Como a distribuição do Z-Score foi maior e o Min-Max tinha passos muito grandes entre um padrão e outro, o Z-Score foi escolhido.

Após a coleta, a normalização dos dados resultou na Tabela 5.1.

A soma dos valores de cada uma das métricas e a ordenação pelo melhor score resultou na Tabela 5.2.

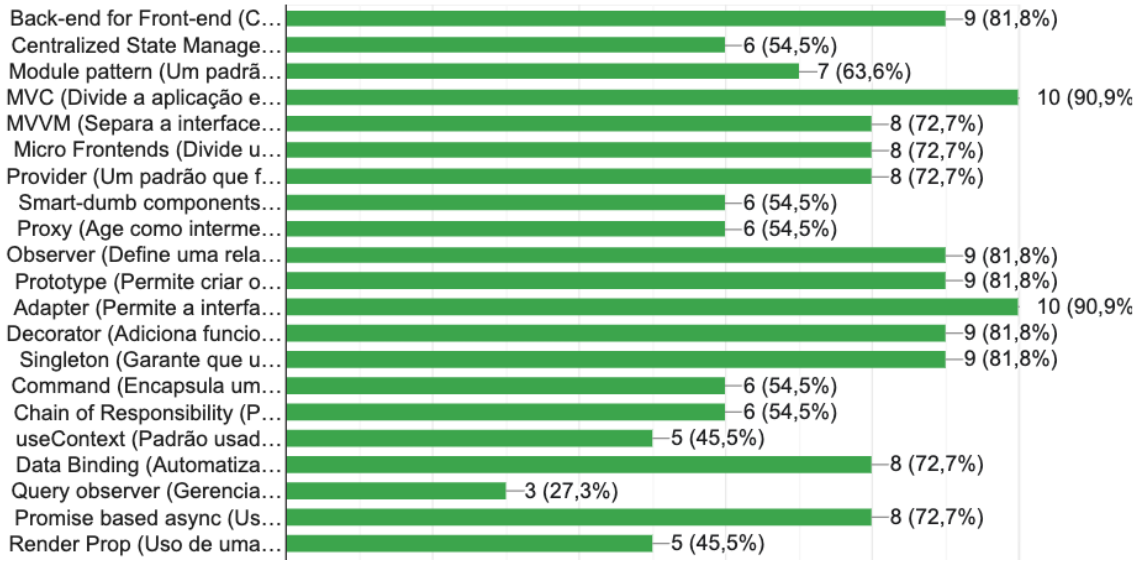


Figura 5.1: *Resultado - Popularidade do padrão*

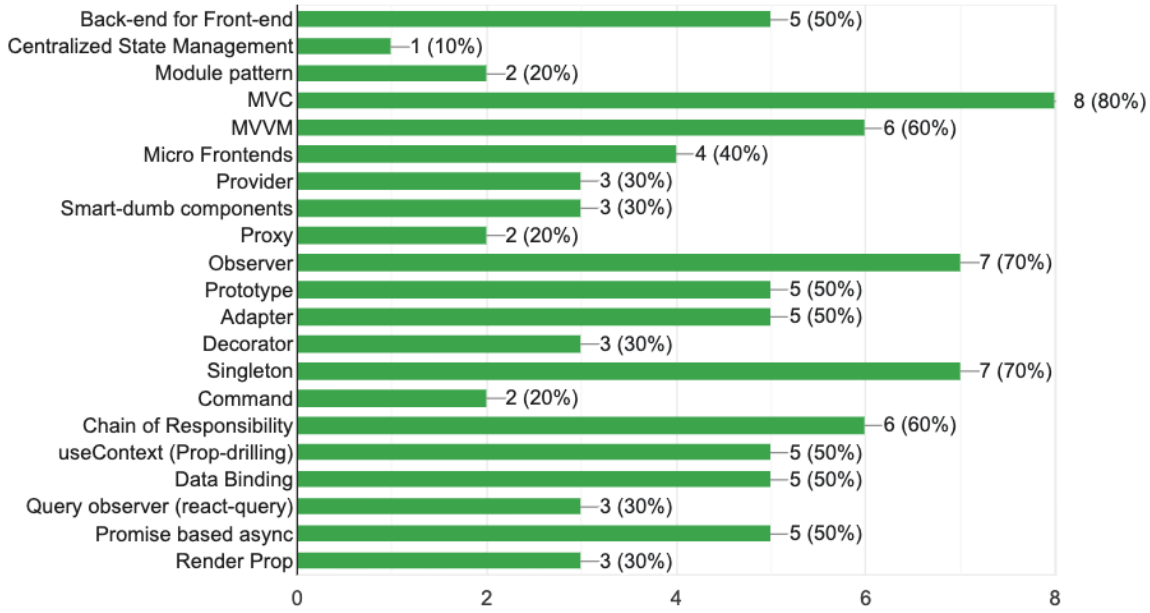


Figura 5.2: *Resultado - Frequência na literatura*

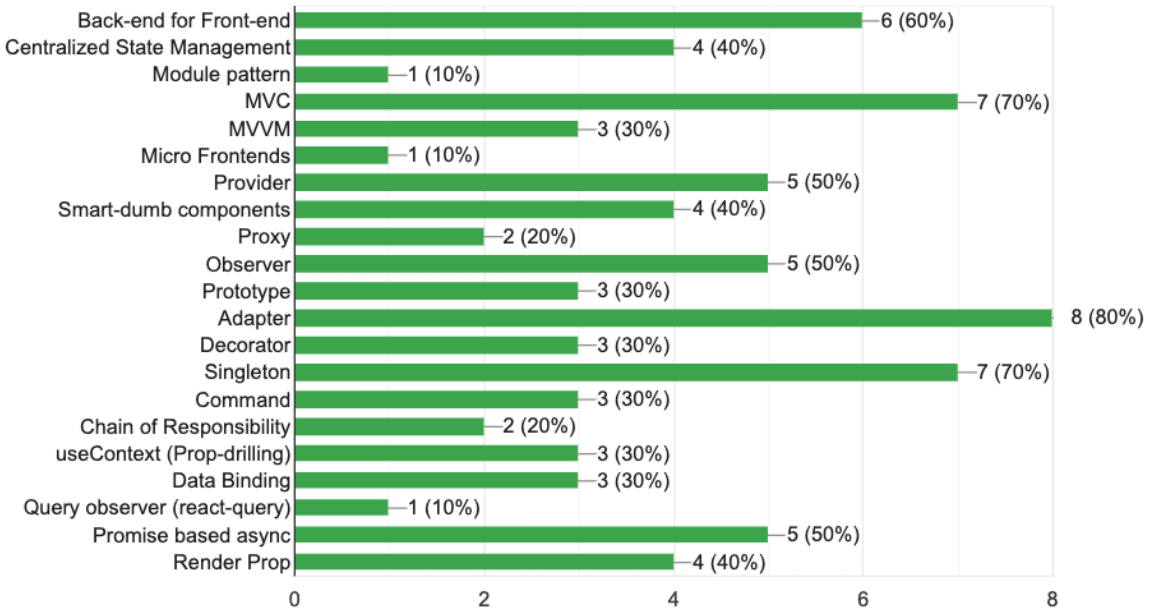


Figura 5.3: *Resultado - Simplicidade de implementação*

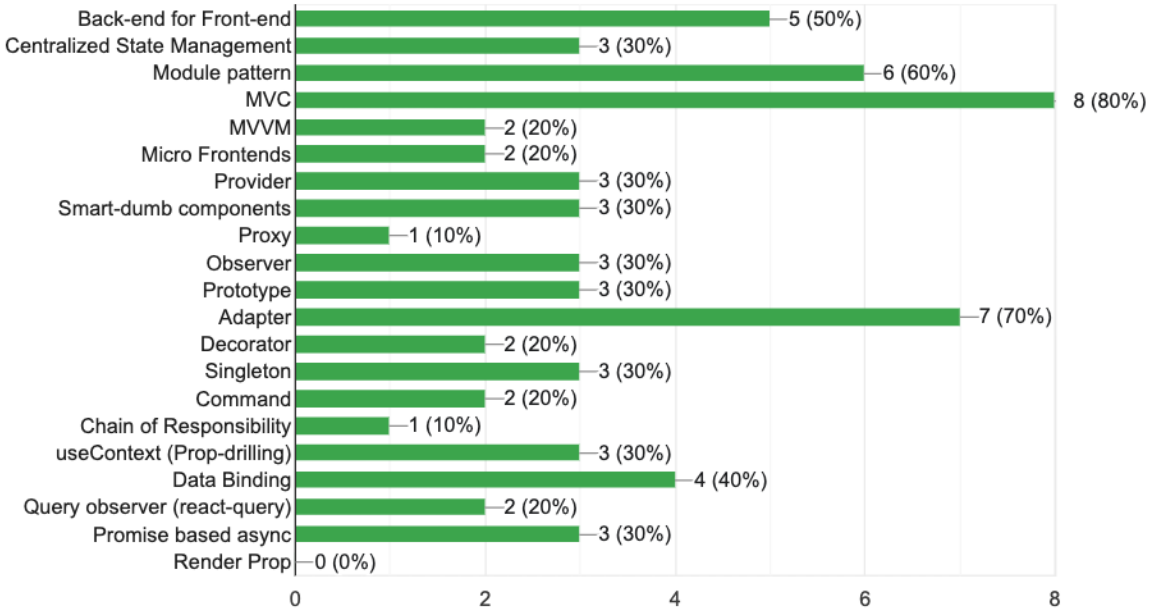


Figura 5.4: *Resultado - Facilidade de manutenção*

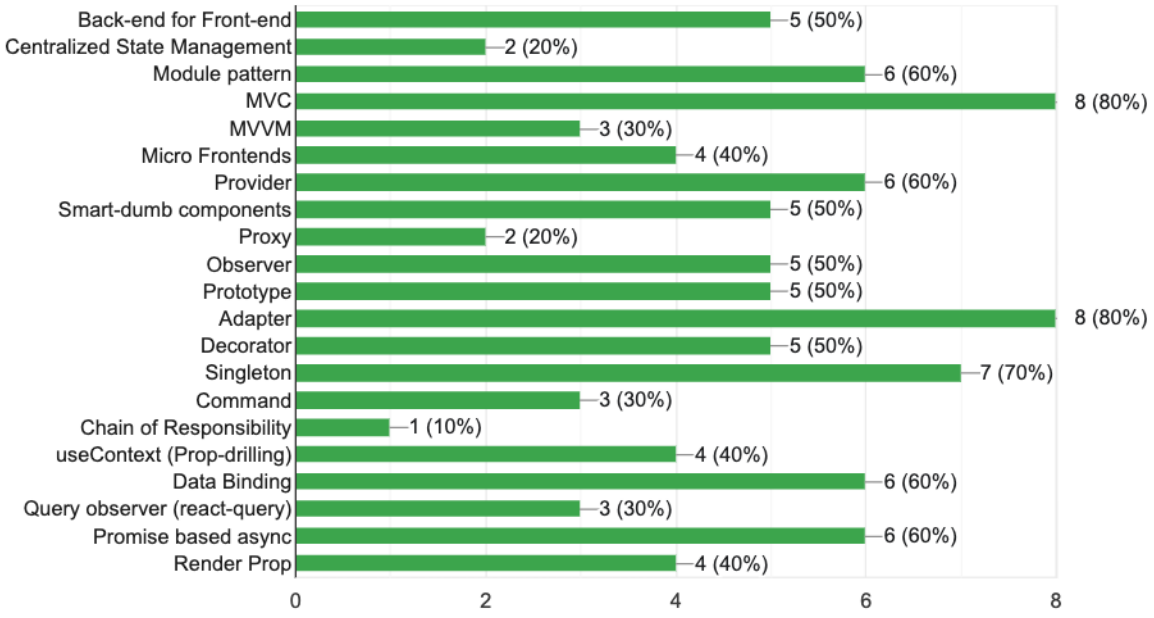


Figura 5.5: *Resultado - Frequência de uso em projetos*

	popularity	literatureFrequency	simplicityOfImplementation	manutenanceEasiness	frequencyOfUse
Back-end for Front-end	0.893500	0.384995	1.113700	0.972572	0.179193
Centralized State Management	-0.762103	-1.770978	0.096844	-0.074813	-1.433543
Module pattern	-0.210235	-1.231985	-1.428442	1.496264	0.716772
MVC	1.445368	2.001975	1.622129	2.543649	1.791929
MVVM	0.341632	0.923989	-0.411585	-0.598506	-0.895964
Micro Frontends	0.341632	-0.153998	-1.428442	-0.598506	-0.358386
Provider	0.341632	-0.692991	0.605272	-0.074813	0.716772
Smart-dumb components	-0.762103	-0.692991	0.096844	-0.074813	0.179193
Proxy	-0.762103	-1.231985	-0.920013	-1.122198	-1.433543
Observer	0.893500	1.462982	0.605272	-0.074813	0.179193
Prototype	0.893500	0.384995	-0.411585	-0.074813	0.179193
Adapter	1.445368	0.384995	2.130557	2.019956	1.791929
Decorator	0.893500	-0.692991	-0.411585	-0.598506	0.179193
Singleton	0.893500	1.462982	1.622129	-0.074813	1.254350
Command	-0.762103	-1.231985	-0.411585	-0.598506	-0.895964
Chain of Responsibility	-0.762103	0.923989	-0.920013	-1.122198	-1.971122
useContext (Prop-drilling)	-1.313971	0.384995	-0.411585	-0.074813	-0.358386
Data Binding	0.341632	0.384995	-0.411585	0.448879	0.716772
Query observer (react-query)	-2.417706	-0.692991	-1.428442	-0.598506	-0.895964
Promise based async	0.341632	0.384995	0.605272	-0.074813	0.716772
Render Prop	-1.313971	-0.692991	0.096844	-1.645890	-0.358386

Tabela 5.1: *Resultados - Dados normalizados*

	Padrão	Score
0	MVC	9.405050
1	Adapter	7.772805
2	Singleton	5.158148
3	Back-end for Front-end	3.543960
4	Observer	3.066134
5	Promise based async	1.973858
6	Data Binding	1.480694
7	Prototype	0.971290
8	Provider	0.895871
9	Decorator	-0.630389
10	MVVM	-0.640434
11	Module pattern	-0.657626
12	Smart-dumb components	-1.253871
13	useContext (Prop-drilling)	-1.773759
14	Micro Frontends	-2.197699
15	Chain of Responsibility	-3.851448
16	Command	-3.900143
17	Render Prop	-3.914395
18	Centralized State Management	-3.944594
19	Proxy	-5.469842
20	Query observer (react-query)	-6.033609

Tabela 5.2: *Nota dos padrões pré-selecionados*

Por fim, usando somente os dez padrões mais bem avaliados na soma dos cinco tópicos, definiu-se a lista final de padrões mostrada na Tabela 5.3.

Nome	Nível de escopo
MVC	Arquitetural
Back-end for Front-end	Arquitetural
Adapter	De projeto
Singleton	De projeto
Observer	De projeto
Prototype	De projeto
Provider	De projeto
Decorator	De projeto
Promise based async	Idiom
Data Binding	Idiom

Tabela 5.3: *Lista de padrões selecionados*

5.3 Documentação sobre os Padrões Selecionados

Após a seleção dos padrões, restou somente confeccionar as documentações sobre cada um deles. A Figura 5.6 tem um exemplo de documentação. Print-screens de todas as outras documentações podem ser encontradas no Apêndice B.

5.4 Discussões

O processo de seleção dos padrões mostrou que existem várias formas diferentes de se implementar cada padrão. Vários entrevistados constataram que usavam estruturas de código descritas pelos padrões sem ter conhecimento de que eram padrões. As entrevistas também ajudaram a constatar a importância desta pesquisa para a área de front-end, que ainda é pobre na documentação de padrões de projeto. Além disso, alguns candidatos desconheciam o conceito de padrões. Por isso, foi importante que as entrevistas fossem conduzidas por uma pessoa que acompanhasse e explicasse os padrões. Se fosse usado o método de compartilhamento em massa do formulário da pesquisa, a validade dos dados seria comprometida.

Por esse aspecto de selecionar e apoiar os entrevistados pessoalmente, a pesquisa reuniu somente dez amostras de profissionais de front-end. A dificuldade de achar pessoas focadas em front-ends, com vários anos de experiência disponíveis para entrevistas foi um fator relevante. Essa limitação pode comprometer a confiabilidade da análise estatística feita dos resultados do formulário. Contudo, espera-se que a seleção dos convidados e as respostas

Back-end for Front-end Padrão arquitetural

Problema

Ao aplicarmos o estilo arquitetural de microsserviços nos nossos sistemas de software, geralmente acabamos com informações espalhadas por vários serviços diferentes. Para que front-ends consigam apresentar essas informações sobre o negócio, é necessário que o front-end conheça grande parte da estrutura dos microsserviços. Isso pode levar a quebra do princípio da responsabilidade única dos componentes.

Além disso, podemos ter front-ends para mobile, para aplicativos de computador e para navegadores. Cada front-end apresenta as informações de forma diferente e portanto também precisa de formatos diferentes do modelo. Isso pode ser resolvido usando diferentes *endpoints* para cada front-end, porém, aumentamos a complexidade dos back-ends e acoplamento entre os serviços de front e back-end.

Exemplo

Um grupo de desenvolvedores tem um grande serviço back-end que recebe todas as requisições de um software front-end para navegadores. Depois de um tempo, o back-end fica grande demais e o grupo resolve dividi-lo em vários serviços menores, adotando o estilo arquitetural de Microsserviços. Com o crescimento da empresa, também foi necessário criar um aplicativo para celulares, ou seja, um front-end mobile.

Como as visualizações são diferentes, cada serviço tem de saber se está respondendo requisições do mobile ou do navegador. Assim, criamos endpoints duplicados para cada dado e os front-ends tem de saber quais endpoints acessarem.

Contexto

- Um sistema de software implementado com microsserviços e um ou mais front-ends que consomem informações desses microsserviços.
- Existência de endpoints duplicados ou consumindo informações sobre a natureza do front-end (se é navegador, mobile, híbrido, aplicativo desktop).
- Lógica complexa dentro dos front-ends para reunir informações dos diversos serviços, por vezes, tendo conhecimento desnecessário da estrutura dos dados no back-end.

Solução

O Back-end for Front-end (BFF) separa a lógica de requisição dos dados da lógica de exibição e estado do front-end. Para o front-end, teremos apenas um serviço de back-end que fornece todos os dados que ele precisa. Já o BFF, expõe as rotas necessárias para o front-end e contém a lógica de agrupamento dos dados, reunindo informações de diversos back-ends para entregar a visualização que o front-end espera.

Neste padrão, cada front-end tem o seu próprio BFF. Dessa forma, podemos lidar com as diferentes necessidades de cada front-end dentro do BFF, evitando esses detalhes no back-end e evitando mudanças desnecessárias na estrutura dos dados armazenados.

Também dentro dessa estrutura, mudanças nos enpoints dos back-ends só resultam em mudanças no BFF, não sendo necessária a alteração do front-end. Isso facilita a manutenção dos back-ends, apesar de ser um padrão pensado principalmente para o front-end.

Diagrama

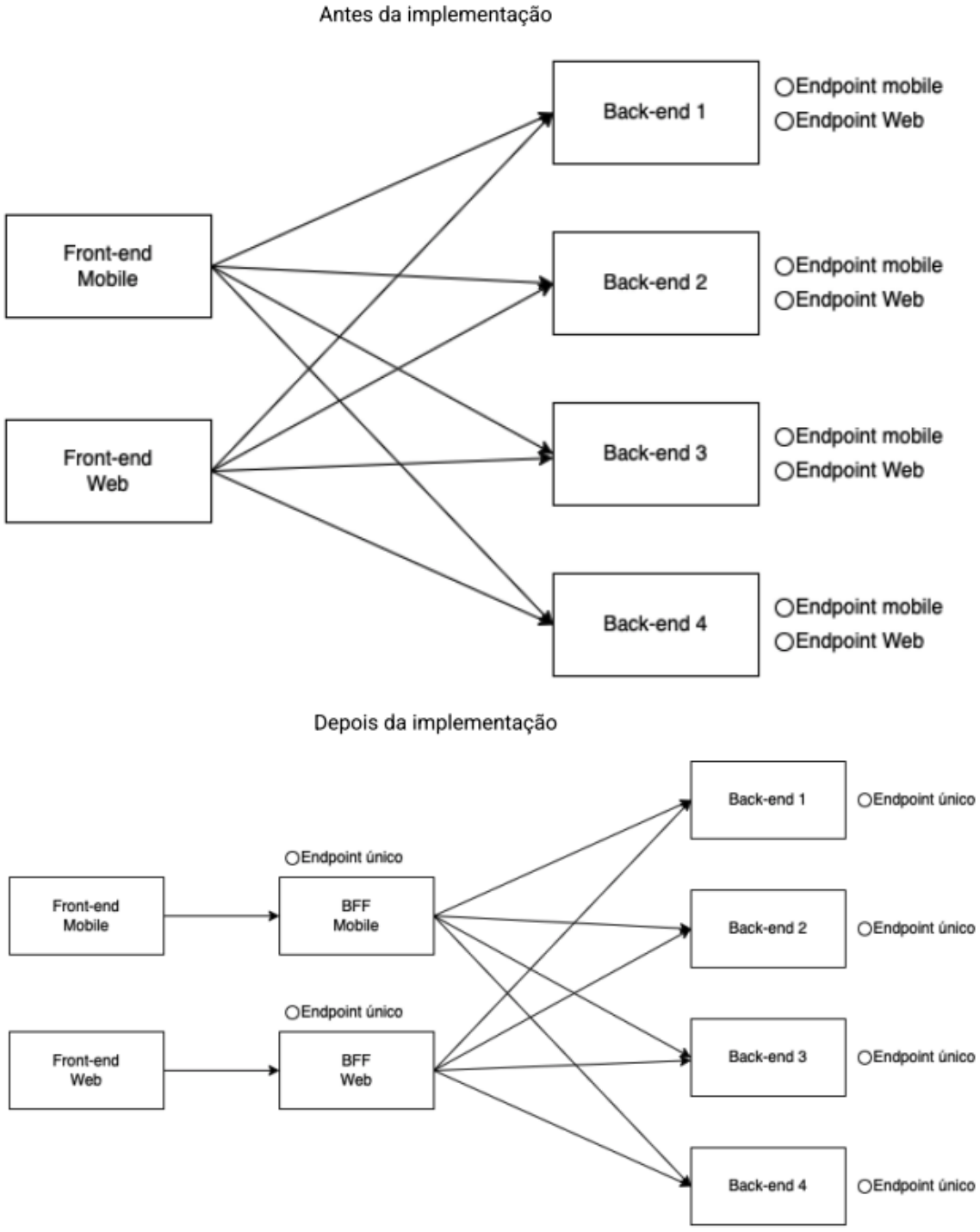


Figura 5.6: *Back-end for Front-end*

(tanto quantitativas coletadas pelo formulário quanto qualitativas coletadas no fim das entrevistas) tenham compensado a quantidade de amostras pela verossimilhança.

Capítulo 6

Conclusões

Esta pesquisa pretendia agilizar o processo de formação de desenvolvedores de software por meio da divulgação de padrões para front-ends. Para cumprir esse objetivo, foram coletados vinte e um padrões associados a front-ends. Dentre eles, era necessário saber quais padrões eram realmente usados na área e se realmente representavam um salto qualitativo para os softwares que os usavam. Dessa forma, foram entrevistados vários convidados com experiência em front-ends. A análise dos dados coletados revelou uma lista de dez padrões com grande relevância para front-ends. Para cada um dos padrões dessa lista, foi feita uma documentação contendo o passo-a-passo de identificação do problema de design associado ao padrão, bem como uma implementação de referência dele.

As documentações tem grande relevância para a comunidade front-end. Portanto, devem ser distribuídas e divulgadas para toda a comunidade. Nesta pesquisa, não foi feita uma estratégia de divulgação. Assim, para ter o impacto esperado, é necessário propagar e disponibilizar facilmente essas documentações em sites comumente visitados por desenvolvedores front-end, como por exemplo a documentação da Mozilla (MDN Web Docs). Além disso, ainda existe espaço para documentar vários padrões que podem ser também relevantes para desenvolvedores front-end.

Na produção das documentações, foram coletadas várias fontes que podem servir de base para uma revisão de estudos sobre front-ends. Como uma coletânea de padrões, esta pesquisa organiza diversos padrões que antes eram encontrados somente de forma espalhada e, por vezes, difusa. Assim, é possível lecionar sobre padrões no desenvolvimento front-end a partir das documentações, sendo uma extensão natural deste trabalho.

Esta pesquisa concluiu seus objetivos de duas formas: conscientizando os participantes das entrevistas e disponibilizando as documentações sobre os padrões. A partir das conver-

sas com desenvolvedores front-end, foi possível amadurecer os participantes no assunto desta pesquisa, além de conscientizar a importância de padrões para o desenvolvimento. Finalmente, as documentações tem um papel importante de consulta que pode perdurar por anos dentro do desenvolvimento front-end.

Apêndice A

Protocolo de Entrevista

A.1 Objetivo

O objetivo deste estudo é classificar padrões de front-end pré-selecionados em cinco parâmetros:

- frequência em bibliografias,
- reconhecimento por parte de profissionais da área,
- simplicidade de implementação,
- facilidade de manutenção, e
- frequência de uso em projetos.

Para isto, vamos usar entrevistas com profissionais da área de Front-end.

A.2 Termo de Consentimento e Confidencialidade

Ao convidar o entrevistado para participar da pesquisa, envie o seguinte Termo de Consentimento e Confidencialidade para o aceite verbal, a ser coletado no início de cada entrevista.

Termo de Consentimento para Entrevista Online e Gravação

Título do Estudo: Padrões Arquiteturais para Front-end

Entrevistador: Francisco Eugênio Wernke

Data da Entrevista: [Data da Entrevista]

Local da Entrevista: Google Meets

Eu, [Nome do Participante], neste ato denominado “Participante”, declaro que fui informado(a) sobre os detalhes do estudo intitulado “Padrões Arquiteturais para Front-end”, conduzido pelo pesquisador “Francisco Eugênio Wernke” orientado pelo professor “Fábio Kon” e pelos aluno de doutorado “João Francisco Daniel” e “Renato Cordeiro Ferreira”. Fui devidamente esclarecido(a) sobre os objetivos, procedimentos e uso da gravação desta entrevista, realizada de forma online.

Objetivo do Estudo: Estou ciente de que a finalidade desta entrevista é contribuir para a pesquisa mencionada acima, visando coletar percepções sobre o uso e conhecimento de padrões no cenário atual de desenvolvimento front-end.

Procedimentos da Entrevista: Compreendo que a entrevista será realizada de forma online, através da plataforma Google Meets, e terá a duração aproximada de 45 minutos. Durante a entrevista, serão abordados tópicos relacionados ao estudo, e terei a oportunidade de compartilhar minhas opiniões, experiências e perspectivas.

Gravação e Uso da Entrevista: Estou ciente de que a entrevista será gravada para fins de documentação e análise pela equipe de pesquisa. Entendo que a gravação será utilizada exclusivamente pelo grupo de pesquisa responsável pelo estudo e que, se a entrevista for usada para artigos ou outros trabalhos no futuro, ela será submetida a um processo de anonimização para proteger minha identidade. A gravação será armazenada de forma segura e confidencial.

Direito de Retirada: Entendo que minha participação é voluntária e que tenho o direito de retirar meu consentimento a qualquer momento, sem a necessidade de justificar minha decisão. Minha retirada não terá nenhum impacto sobre minha relação com o grupo de pesquisa ou quaisquer serviços futuros.

Declaro que li este termo de consentimento, compreendi seu conteúdo e concordo em participar da entrevista online e autorizar a gravação, de acordo com os termos aqui descritos.

A.3 Roteiro

O roteiro foi simplificado em uma apresentação de slides para ser acompanhada durante a entrevista. Esta é a versão mais completa que pode ser usada para reproduzir a entrevista.

A.3.1 Objetivo do Projeto

Explique para o entrevistado o objetivo do projeto como um todo, com as seguintes frases como base:

- encontrar os padrões mais importantes para o desenvolvimento front-end e como eles são usados por profissionais da indústria, e
- produzir uma documentação sobre os padrões selecionados que seja de fácil entendimento para novatos na área.

A.3.2 Definições do Projeto

Nesta seção, foi explicado qual é conceito de padrão utilizado no projeto.

- "Nesta pesquisa estamos usando o conceito de padrão como uma tríade de problema, solução e contexto. Os padrões descrevem dentro de um contexto, como um problema pode ser resolvido. Chamamos as necessidades que partem do problema e de seu contexto de 'Forças'. Essas forças devem ser compreendidas e balanceadas pela solução. Um padrão é, ao mesmo tempo, o problema, o processo e o resultado em software da solução do problema."

- "Dividimos os padrões em 3 categorias: padrões de design, padrões arquiteturais e expressões idiomáticas, dependendo da sua aplicabilidade e impacto no software. Padrões de design são aplicáveis a subsistemas, componentes de um sistema ou mecanismos que relacionam dois componentes. Ex: Mediador, Adaptador, Singleton. padrões arquiteturais tem um contexto mais amplo, define a base sobre a qual um sistema é construído, organiza um grupo de componentes. Exs: MVC, Layered, Pipes-and-Filters. Já as expressões idiomáticas são padrões que existem dentro de linguagens de programação específicas, de uso mais limitado e específico. Exs: useContext no React, useReducer em vez de vários useStates, Singletons em C++."

A.3.3 Apresentação dos Padrões Pré-Selecionados

Depois da introdução e alguns exemplos que situam o entrevistado no tema, comece a apresentação da lista não-filtrada de padrões. Ela contém vinte e um padrões expostos em um formulário do Google Forms. O formulário continha cinco perguntas:

1. Dos padrões citados, quais deles você já ouviu falar?
2. Quais você mais encontra na literatura?
3. Quais destes você considera que são simples de implementar?

4. Quais deles você considera que auxiliam na manutenção do software ou componente?
5. Quais destes você já implementou?

A Figura A.1 mostra uma das cinco perguntas do formulário que pode ser acessado aqui: Link do formulário. As demais perguntas seguem o mesmo padrão. Contudo, a primeira pergunta contém uma descrição de cada um dos padrões, como mostrado na Figura A.2. Use essa descrição para explicar o padrão caso o candidato tenha dúvidas.

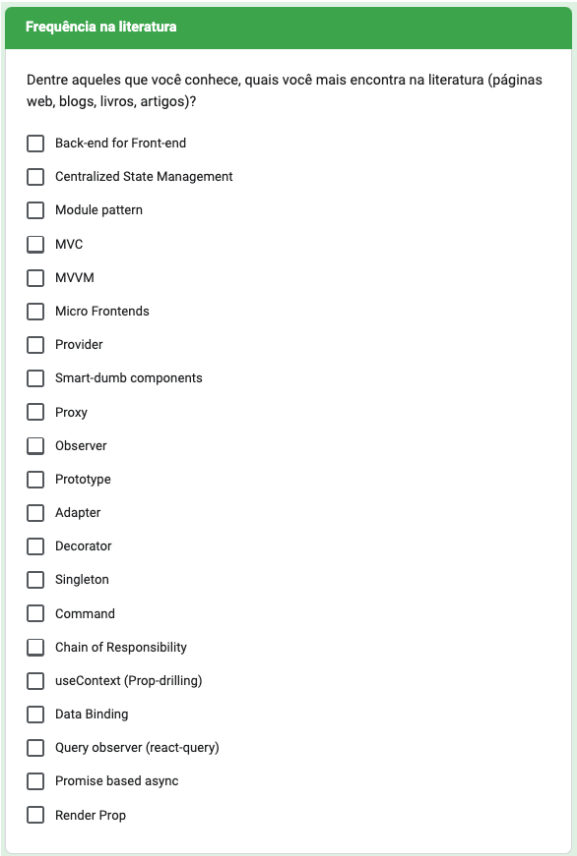


Figura A.1: *Pergunta sobre simplicidade de implementação do formulário*

A.3.4 Perguntas Abertas

Por fim, aplique duas perguntas abertas sobre a experiência da pessoa com padrões. As perguntas são:

- Você gostaria de compartilhar alguma experiência relacionada?
- Como você conheceu os padrões relatados?

Estas perguntas fornecem um material valioso para entender o contexto que o entrevistado tinha previamente com padrões e como os entrevistados aprenderam estes padrões.

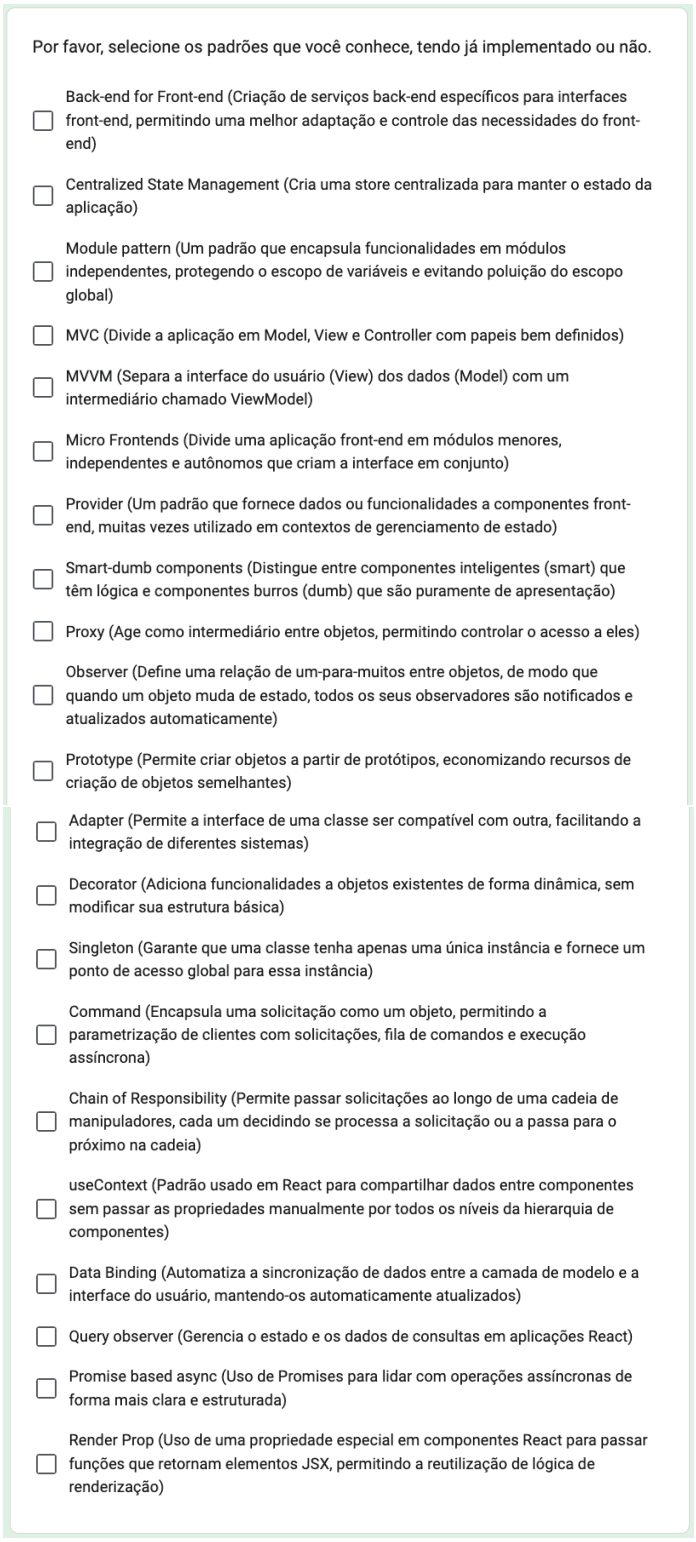


Figura A.2: *Pergunta sobre conhecimento do padrão*

Apêndice B

Documentações

Neste apêndice estão todas as documentações criadas.

Back-end for Front-end Padrão arquitetural

Problema

Ao aplicarmos o estilo arquitetural de microsserviços nos nossos sistemas de software, geralmente acabamos com informações espalhadas por vários serviços diferentes. Para que front-ends consigam apresentar essas informações sobre o negócio, é necessário que o front-end conheça grande parte da estrutura dos microsserviços. Isso pode levar a quebra do princípio da responsabilidade única dos componentes.

Além disso, podemos ter front-ends para mobile, para aplicativos de computador e para navegadores. Cada front-end apresenta as informações de forma diferente e portanto também precisa de formatos diferentes do modelo. Isso pode ser resolvido usando diferentes *endpoints* para cada front-end, porém, aumentamos a complexidade dos back-ends e acoplamento entre os serviços de front e back-end.

Exemplo

Um grupo de desenvolvedores tem um grande serviço back-end que recebe todas as requisições de um software front-end para navegadores. Depois de um tempo, o back-end fica grande demais e o grupo resolve dividi-lo em vários serviços menores, adotando o estilo arquitetural de Microsserviços. Com o crescimento da empresa, também foi necessário criar um aplicativo para celulares, ou seja, um front-end mobile.

Como as visualizações são diferentes, cada serviço tem de saber se está respondendo requisições do mobile ou do navegador. Assim, criamos endpoints duplicados para cada dado e os front-ends tem de saber quais endpoints acessarem.

Contexto

- Um sistema de software implementado com microsserviços e um ou mais front-ends que consomem informações desses microsserviços.
- Existência de endpoints duplicados ou consumindo informações sobre a natureza do front-end (se é navegador, mobile, híbrido, aplicativo desktop).
- Lógica complexa dentro dos front-ends para reunir informações dos diversos serviços, por vezes, tendo conhecimento desnecessário da estrutura dos dados no back-end.

Solução

O Back-end for Front-end (BFF) separa a lógica de requisição dos dados da lógica de exibição e estado do front-end. Para o front-end, teremos apenas um serviço de back-end que fornece todos os dados que ele precisa. Já o BFF, expõe as rotas necessárias para o front-end e contém a lógica de agrupamento dos dados, reunindo informações de diversos back-ends para entregar a visualização que o front-end espera.

Neste padrão, cada front-end tem o seu próprio BFF. Dessa forma, podemos lidar com as diferentes necessidades de cada front-end dentro do BFF, evitando esses detalhes no back-end e evitando mudanças desnecessárias na estrutura dos dados armazenados.

Também dentro dessa estrutura, mudanças nos enpoints dos back-ends só resultam em mudanças no BFF, não sendo necessária a alteração do front-end. Isso facilita a manutenção dos back-ends, apesar de ser um padrão pensado principalmente para o front-end.

Diagrama

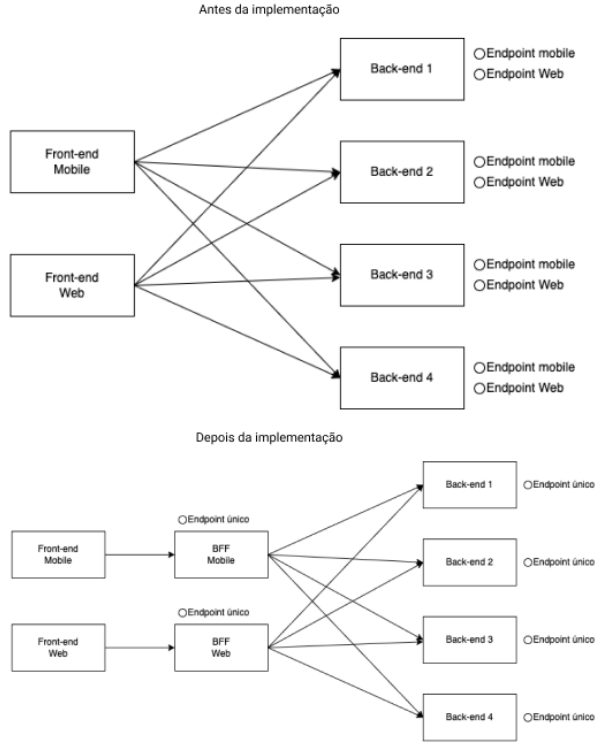


Figura B.1: *Back-end for Front-end*

Model-View-Controller Padrão arquitetural

Problema

Organizar uma aplicação é sempre uma tarefa complicada. Porém, não ter nenhuma direção para a arquitetura de um sistema, é a receita para a desorganização em pouquíssimo tempo. Dessa forma, precisamos de uma guia para entendermos em que parte do software cada parte da lógica fica. Assim, conseguimos ler o código como uma linha de produção, onde cada subsistema tem seu papel bem-definido e regido por regras claras.

Apesar de existirem diversos padrões arquiteturais que possam ser seguidos, é muito importante que saibamos qual a intenção daquele padrão. Diferentemente de escolhas pontuais sobre o projeto que são tomadas de acordo com novas funcionalidades que são adicionadas, o padrão arquitetural tem efeitos duradouros e vai influenciar a construção do projeto do início ao fim.

Exemplo

Você precisa construir um novo front-end para apresentar dados internos sobre os usuários do seu produto. Esse front-end apresentará diversas telas, uma para cada entidade da empresa, com relações entre algumas delas. Essas telas serão construídas por várias equipes, cada uma terá seu contexto e entidade bem-definidas, de forma que uma equipe tenha que ter o mínimo de interação possível com outra equipe.

No começo, os projetos funcionam bem e as equipes conseguem navegar na complexidade de cada entidade. Porém, devido a um corte de verbas, algumas equipes são desfeitas e agora sua equipe deve dar manutenção a algumas outras entidades além da originalmente planejada para vocês.

Como não foi pensado nenhum padrão para a construção da tela de cada entidade, os softwares contruídos pelas equipes são profundamente diferentes. Agora, navegar pela complexidade de outras telas é uma tarefa complicada e dar manutenção é custoso.

Contexto

- O software a ser construído deve permitir apresentar e modificat um modelo de dados de uma entidade
- O software deve ser extensível a novas funcionalidades
- A relação entre a tela, os dados que estamos trabalhando e as operações possíveis sobre eles é bem clara

Solução

O Model-View-Controller é um dos padrões arquiteturais mais antigos de toda a computação. Ele define a aplicação como três componentes básicos: Modelo, Visualização e Controles. Esta separação guia os programas para que saibamos onde encontrarmos cada tipo de lógica. Para front-ends, temos uma definição específica de cada um desses componentes:

Modelo: É uma representação das entidades do negócio geralmente armazenadas nos back-ends. Apesar do front-end não controlar diretamente o modelo, ainda é necessário que exista a sincronização do modelo interno do front-end com o modelo do back-end.

Visualizações: É o componente que estrutura as páginas e produz de fato o HTML e o CSS. Os componentes da aplicação se encontram aqui e eles devem conhecer o modelo para organizá-lo em páginas. A lógica destes elementos deve se limitar a apresentação e a modificação dos dados deve ser bastante restrita. Geralmente deixa-se uma camada intermediária entre os modelos e visualizações que permite a adaptação destes dados (vide padrão Adapter).

Controles: É o componente que estipula as ações possíveis sobre o modelo. Aqui, encontramos funções que executam chamadas para back-ends para alterarmos informações, também invalidando o modelo local e forçando sincronizações. Em aplicações modernas, é comum vermos os Controles muito próximos as visualizações (Se são unidos, podemos chamar de uma arquitetura MVVM).

Diagrama

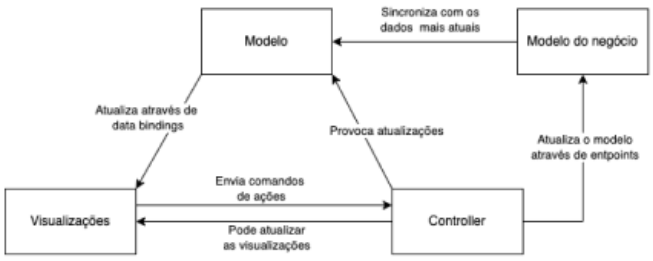


Figura B.2: *MVC*

Adapter Padrão de Projeto

Problema

Em front-ends, é comum que tenhamos que utilizar serviços back-end que fornecem uma série de dados de acordo com sua estrutura interna. Contudo, nem sempre a estrutura dos dados representa o melhor formato para ser usado nos componentes. Muitas vezes, precisamos trabalhar nessa estrutura antes dela chegar aos componentes. Isso ajuda na separação entre a lógica de apresentação e tratamento dos dados. Além disso, mudanças na estrutura dos dados que vem dos back-ends nem sempre devem alterar os componentes e, consequentemente, apresentação destes dados.

Exemplo

Você desenvolve um front-end para um serviço de hotelaria digital. Para que o usuário crie reservas nos hotéis, é importante que você disponibilize o número de telefone desses hotéis. O serviço Back-end enviava os números de telefone em uma lista que associava cada hotel a um número de telefone diferente. Porém, devido a mudanças na organização do back-end, o endpoint passou a enviar um dicionário que continha os nomes dos hotéis como chaves e os números de telefone como valores. Dessa forma, você precisa alterar todos os componentes que usavam números de telefone para se adaptarem ao novo modelo.

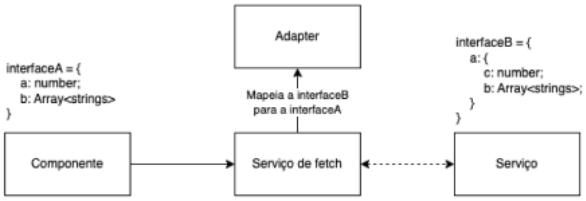
Contexto

- Mudanças no retorno de endpoints causam grandes alterações nos componentes que os consomem
- Acoplamento entre a estrutura de dados enviada pelo back-end e a estrutura de dados interna do front-end
- Lógica de mapeamento de dados espalhada pelos componentes

Solução

O padrão Adapter utiliza de interfaces e mapeadores para facilitar e centralizar a lógica de mapeamento entre estruturas de dados. Ao criarmos um adapter para cada estrutura de dados que temos dentro do nosso front-end, podemos padronizar a expectativa dos componentes com relação aos dados que serão recebidos por eles. Além disso, ainda diminuímos o acoplamento entre o serviço back-end e o front-end, tornando mais fácil a mudança do retorno dos back-ends sem impactar os front-ends. Esse padrão é muito utilizado em conjunto com os Back-ends for Front-ends, que são uma expansão desse desacoplamento.

Diagrama



Implementação

```
type Telefone = {
  numero: string;
  hotelId: number;
}

function adaptarTelefones (telefone: Record<any, string>): Telefone[] {
  const telefones: Telefone[] = [];

  for (const [key, value] of Object.entries(telefone)) {
    telefones.push({
      numero: value,
      hotelId: parseInt(key)
    });
  }

  return telefones;
}

async function fetchTelefones() {
  const telefones = await fetch('http://hoteis.com/telefones');
  const body = await telefones.text();
  return adaptarTelefones(JSON.parse(body));
}

export const componenteB = () => {
  const [telefones, setTelefones] = React.useState<Telefone[]>([]);

  React.useEffect(() => {
    fetchTelefones().then(telefones => setTelefones(telefones));
  })

  return <>
    <ul>
      {telefones.map(telefone => <li key={telefone.hotelId}>{telefone.numero}</li>)}
    </ul>
  </>
};
```

Figura B.3: *Adapter*

Singleton Padrão de Projeto

Problema

É comum que, ao implementar front-ends, algumas funcionalidades precisem ser acessadas de diversos lugares. Temos diversas maneiras de compartilhar lógica entre componentes mas algumas vezes é necessário que estabeleçamos um controle especial sobre como cada componente vai acessar aquela funcionalidade. Além disso, alguns recursos só podem ser acessados uma vez, como arquivos e conexões perenes com APIs.

Exemplo

Imaginemos que você possui um front-end complexo que é utilizado por vários usuários. Porém, a equipe de produto da sua empresa quer que você produza dados sobre a utilização de algumas funcionalidades, para controlar quais funcionalidades devem ser evoluídas ou descartadas. Para gerar e armazenar essas métricas, você usa um provedor externo que distribui uma biblioteca cliente que precisa conectar com o servidor de métricas. Esta conexão precisa continuar ativa durante todo o funcionamento da aplicação pois criá-la é demorado e custoso para a aplicação.

Contexto

- Temos um recurso único dentro da aplicação, como uma conexão com banco de dados ou uma conexão perene com uma API
- A funcionalidade precisa ser acessível a várias partes do sistema
- A inicialização do objeto que gerencia o recurso é custosa

Solução

O padrão Singleton possibilita limitar a quantidade de instâncias de uma classe, para uma ou mais instâncias, dependendo da necessidade. Esse controle permite um contrato especializado para lidarmos com um recurso ou simplesmente para disponibilizar a funcionalidade com estado para todo o sistema. Para criarmos esta instância, podemos colocar essa inicialização no começo do ciclo de vida do front-end ou inicializarmos somente quando for necessário o uso da funcionalidade.

O Singleton pode ser considerado um padrão controverso. Se utilizado com muita frequência, ele pode dificultar a manutenção do sistema, pois cada Singleton é uma variável global com estado exposto a todos os componentes.

Diagrama

Implementação

```
async function inicializarConexao() {
  const conexao: any = await criarConexao('http://metricasSA/conexao', SingletonDeMetricas.chaveDeConexao);
  return conexao.cliente;
}

class SingletonDeMetricas {
  private static instancia: SingletonDeMetricas;

  private cliente: any;
  private constructor() {};

  public static chaveDeConexao = '123';

  public static getInstancia(): SingletonDeMetricas {
    if (!SingletonDeMetricas.instancia) {
      SingletonDeMetricas.instancia = new SingletonDeMetricas();
      SingletonDeMetricas.instancia.cliente = inicializarConexao();
    }

    return SingletonDeMetricas.instancia;
  }

  public async enviarMetrica(metrica: string) {
    this.cliente.postMetricaParaServidor(metrica);
  }
}

export const componenteA = () => {
  const metricas = SingletonDeMetricas.getInstancia();

  return <>
    <button onClick={() => metricas.enviarMetrica('botaoA')}>Botão A</button>
  </>
};
```

Figura B.4: *Singleton*

Observer Padrão de Projeto

Problema

Ao elaborarmos a passagem de informação de um componente ao outro dentro de um sistema, é comum que vários componentes precisem trocar informações dadas mudanças de estado internas. É comum também, que essas trocas de informações comecem a ficar cada vez mais complexas, exigindo um alto acoplamento entre diferentes partes do sistema. O acoplamento pode deixar mudanças no comportamento dos componentes, uma tarefa árdua que envolve analisar todos os componentes que dependem de cada estado.

Exemplo

Você está implementando um e-commerce em que vários produtos entram e saem de promoção rapidamente, de forma que o usuário poderia entrar em um produto e a promoção não estar mais ativa depois de poucos segundos. Nesse sistema, é imprescindível que atualizações nos preços dos produtos gerem novas renderizações em cada produto exibido na tela. Para isso, a aplicação busca constantemente no serviço de back-end, por atualizações de preço. Contudo, estamos interessados em atualizações em tempo real somente de produtos que estejam sendo exibidos na tela.

Ao implementar um esquema de eventos para alguns produtos, você rapidamente percebe que a lógica do software que busca informações no back-end constantemente ficará bastante complexa. E essa complexidade cresce na mesma proporção que a quantidade de produtos na loja.

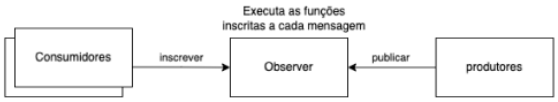
Contexto

- Grande quantidade de dados que precisam ser repassados para componentes
- Informações que precisam ser entregues imediatamente para os componentes, para não atrapalhar a experiência do usuário
- Diversos consumidores de informação para um número baixo de produtores de dados.

Solução

O padrão Observer, vastamente usado em sistemas Back-end, tem papel importante na atualização de dados em tempo real no front-end também. Esta padrão estabelece uma interface para consumidores se inscreverem a eventos ou tópicos e também para produtores poderem postar novos eventos que serão consumidos por todos os interessados naqueles eventos. A inversão de dependência dos produtores para com os consumidores é a peça principal neste padrão. Como os consumidores agora tem total controle sobre quais eventos eles desejam e o produtor não se importa mais com isso, a complexidade é pulverizada pelos componentes, tornando-os mais simples de serem mantidos.

Diagrama



Implementação

```
class Observer {
  constructor() {
    this.topicos = {};
  }

  subscribe(fn, topico) {
    if (this.topicos[topico]) {
      this.topicos[topico].push(fn);
      return;
    }
    this.topicos[topico] = [fn];
  }

  unsubscribe(fn, topico) {
    if (topico) {
      this.topicos[topico] = this.topicos[topico].filter(subscriber => subscriber !== fn);
    } else {
      Object.keys(this.topicos).forEach((key) => {
        this.topicos[key] = this.topicos[key].filter(subscriber => subscriber !== fn);
      });
    }
  }

  postarEvento(mensagem, topico) {
    if (this.topicos[topico]) {
      this.topicos[topico].forEach((subscriber) => {
        subscriber(mensagem);
      });
    } else {
      throw new Error('Topico não existe');
    }
  }
}

class Produto {
  constructor(nome, preco, observer) {
    this.nome = nome;
    this.preco = preco;
    this.observer = observer;
    this.observer.subscribe(this.alterarPreco.bind(this), nome);
  };

  alterarPreco(mensagem) {
    this.preco = mensagem;
    console.log(`O produto ${this.nome} teve seu preço alterado para ${mensagem}`);
  }
}

class Loja {
  constructor(observer, servicoDePreco) {
    this.observer = observer;
    this.servicoDePreco = servicoDePreco;
  }

  consultarPrecoProduto(produto) {
    const preco = this.servicoDePreco.consultarPrecoProduto(produto);
    this.observer.postarEvento(preco, produto);
  }
}
```

Figura B.5: *Observer*

Protoype Padrão de Projeto

Problema

Ao criarmos instâncias de uma classe, atribuímos a elas valores para cada um dos atributos. Se pensamos em objetos simples de serem criados, é comum instânciarmos várias vezes a classe e atribuirmos manualmente os atributos. COntudo, em alguns casos, a instanciação de um objeto é algo trabalhoso e pode gerar diferentes erros no caminho.

Exemplo

Você está implementando um sistema e precisa testar uma classe. Para cada caso de teste, você precisará instânciar a classe que quer testar e realizar alguns processos para certificar que ela se comporta da forma desejada. Porém, se a classe que você precisa testar é particularmente complexa, cada caso de teste demorará um pouco para instanciar a classe. Dessa forma, executar esses testes se torna demorado, oque pode dificultar a produção do código.

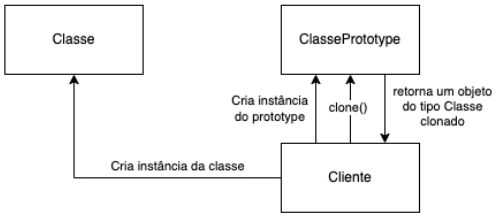
Contexto

- Temos uma classe que tem um processo de instanciação complexo
- Precisamos copiar um objeto de forma exata com todos os seus atributos

Solução

O padrão Prototype é usado para copiarmos objetos de forma idêntica, com todos os atributos iguais. Esse padrão utiliza uma segunda classe que tem um método que conhece todos os atributos da classe a ser copiada e copia os atributos do objeto passado para um novo objeto que é retornado. Dessa forma, podemos resolver o problema do exemplo criando uma classe Prototype que copia todos os atributos da classe sendo testada para um novo objeto. Assim, podemos alterar o objeto clonado para todos os casos de teste sem perda do estado original da instância.

Diagrama



Implementação

```
class ClasseSendoTestada {
  constructor (atrA, atrB) {
    this.atrA = atrA;
    this.atrB = atrB;
    this.atrC = this.tarefaComplicada();
  }

  public atrA: number;
  public atrB: number;
  public atrC: number;

  tarefaComplicada() {
    // ... código que demora a ser executado
    return 2;
  }

  soma() {
    return this.atrA + this.atrB;
  }
}

class ClasseSendoTestadaPrototype {
  clone(proto: ClasseSendoTestada) {
    return {
      atrA: proto.atrA,
      atrB: proto.atrB,
      atrC: proto.atrC,
      soma: proto.soma,
      tarefaComplicada: proto.tarefaComplicada,
    }
  }
}

class Teste {
  constructor() {
    this.classeSendoTestada = new ClasseSendoTestada(1, 2);
    this.classeClonada = new ClasseSendoTestadaPrototype();
  }

  public classeSendoTestada: ClasseSendoTestada;
  public classeClonada: ClasseSendoTestadaPrototype;

  testarSoma() {
    const objInstanciado = this.classeClonada.clone(this.classeSendoTestada);
    const resultado = objInstanciado.soma();
    if (resultado !== 3) {
      throw new Error('Resultado inválido');
    }
  }

  testarSomaEmOutroCenario() {
    const objInstanciado = this.classeClonada.clone(this.classeSendoTestada);
    objInstanciado.atrA = 2;
    const resultado = objInstanciado.soma();
    if (resultado !== 4) {
      throw new Error('Resultado inválido');
    }
  }
}
```

Figura B.6: *Prototype*

Provider Padrão de Projeto

Problema

A passagem de contexto de um componente para o outro pode ser feita de várias formas. É comum que precisemos criar estruturas para comunicar esse contexto da forma mais simples possível. Uma forma de provermos informações é usar a passagem de propriedades de um componente para o outro, prática bastante comum na maioria dos frameworks de front-end. Contudo, essa prática pode se tornar problemática a partir do momento que passamos a ter diversos níveis de componentes e várias propriedades.

Além disso, é comum termos de disponibilizar funções e funcionalidades para os componentes que precisam manter um determinado estado. É o caso de alertas e notificações que precisam ser disparadas por diversos componentes

Exemplo

Você está criando um aplicativo de leitura no qual podemos definir diversos aspectos do texto via configurações do usuário. As configurações atuais do aplicativo são definidas em uma tela de configurações que disparava eventos para diversos componentes que se atualizam de acordo com a mudança na configuração. Porém, com o tempo, foram criadas muitas configurações e portanto muitos componentes consomem esses eventos. Além disso, os novos componentes de imagem eram renderizados somente em algumas páginas e precisavam guardar estes eventos para retomarem as suas configurações mesmo quando não estavam aparecendo na tela.

Contexto

- Diversas informações de estado sendo armazenadas de forma descentralizada.
- Criação de múltiplos eventos para lidar com a troca de informação.
- Perca de informação ao renderizar e desaparecer da tela.

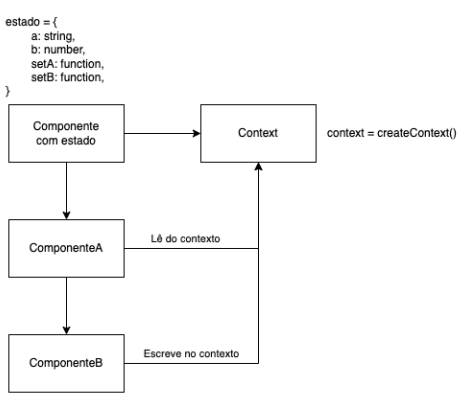
Solução

O padrão Provider envolve uma parte da árvore de componentes e disponibiliza o seu estado interno como uma série de informações sobre o estado da aplicação. Além disso, podemos controlar a renderização de componentes por meio de funções que modificam o estado interno do Provider.

Os Providers podem ser usados em diversas partes do nosso código e o estado de cada um deles é independente. Dessa forma, o Provider é altamente reutilizável e pode ser usado de forma a dar significado aos dados. Podemos usar o Provider por exemplo para: Configurações do aplicativo, sessão do usuário, contexto do navegador, entre outros.

Providers existem em várias linguagens e frameworks e geralmente usam-se bibliotecas que entregam esta solução pronta. No React, é comum usarmos o useContext. O exemplo a seguir utilizará desta funcionalidade padrão do React.

Diagrama



Implementação

```jsx
import { useContext, useState } from "react";

const ConfigContext = React.createContext(null);

export function App({ texto, numeroDaPagina }) {
  const [tamanhoDaFonte, setTamanhoDaFonte] = useState(12);
  const [mostrarNumeroDaPagina, setMostrarNumeroDaPagina] = useState(false);
  const [darkMode, setDarkMode] = useState(false);

  return (
    <ConfigContext.Provider value={{
      tamanhoDaFonte,
      mostrarNumeroDaPagina,
      darkMode,
      setTamanhoDaFonte,
      setMostrarNumeroDaPagina,
      setDarkMode,
    }}>
      <PaginaDeConfiguracao/>
      <PaginaParaLeitura texto={texto} numeroDaPagina={numeroDaPagina}/>
    </ConfigContext.Provider>
  )
}

export function PaginaDeConfiguracao() {
  const {
    tamanhoDaFonte,
    mostrarNumeroDaPagina,
    darkMode,
    setTamanhoDaFonte,
    setMostrarNumeroDaPagina,
    setDarkMode
  } = useContext(ConfigContext);

  return (
    <div>
      <input
        type="range"
        min="12"
        max="24"
        value={tamanhoDaFonte}
        onChange={(e) => setTamanhoDaFonte(e.target.value)}
      />
      <input
        type="checkbox"
        value={mostrarNumeroDaPagina}
        onChange={(e) => setMostrarNumeroDaPagina(e.target.value)}
      />
      <input
        type="checkbox"
        value={darkMode}
        onChange={(e) => setDarkMode(e.target.value)}
      />
    </div>
  );
}

export function PaginaParaLeitura({ texto, numeroDaPagina }) {
  const { tamanhoDaFonte, mostrarNumeroDaPagina, darkMode } = useContext(ConfigContext);

  return (
    <>
      {mostrarNumeroDaPagina && <p>{numeroDaPagina}</p>}
      <div style={{ backgroundColor: darkMode ? 'black' : 'white' }}>
        <p style={{ fontSize: `${tamanhoDaFonte}px`, color: darkMode ? 'white' : 'black' }}>
          {texto}
        </p>
      </div>
    </>
  );
}
```

Figura B.7: *Provider*

Promise based async Expressão linguística

Problema

O JavaScript é uma linguagem de somente uma *thread*. Isto significa que todas as operações são executadas em um único laço, que chamamos de Laço de Eventos (*Event Loop*). Enquanto isso facilita o entendimento do código, pois cada operação será executada até o fim sem impedimentos, isso também pode gerar alguns problemas quando precisamos executar operações que dependem de respostas de APIs. Por exemplo, quando vamos executar um *fetch*, o JS chama algumas funções do navegador, que por sua vez devem executar e adicionar uma nova operação ao laço para que o JS colha os dados.

Exemplo

Você está criando um programa que usará 3 APIs. Essas APIs retornarão dados necessários para apresentar uma tela para seu usuário. Porém, enquanto o usuário espera essas chamadas serem completas, gostaríamos que ele visse uma animação acontecendo na tela, que é controlada também pelo JavaScript. Além disso, gostaríamos de ter um botão na tela que o usuário pudesse cancelar essas chamadas. Contudo, ao começarmos as chamadas as APIs, a animação trava e o botão se torna irresponsivo.

Contexto

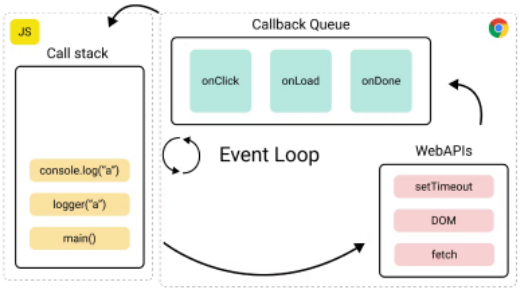
- Estamos executando processos que demoram um tempo indeterminado
- Os processos que estamos executando dependem de APIs terceiras, não controladas pelo nosso programa
- O laço de eventos do JavaScript não pode parar para esperar esses processos

Solução

A solução implementada pelo JavaScript é permitir que tarefas sejam agendadas para serem executadas somente quando algum processo terceiro responder. Dessa forma, podemos conferir se uma tarefa esta pronta para ser processada e só executá-la no momento certo, não impedindo outras tarefas no processo. A prioridade do laço de eventos, no Javascript, é sempre manter a tela do usuário atualizada, processando mudanças no HTML e no CSS a todo momento. Só depois de garantir os processos que envolvem a tela que ele processa tarefas secundárias que foram envolvidas em Promises.

A expressão linguística Promise que é implementada pelo JS para permitir a assincronia, disponibiliza uma interface padrão para atrasarmos a execução de tarefas. A partir do ES 2017, podemos simplificar o uso de Promises com as palavras reservadas **async** e **await.**

Diagrama



Referência: https://subhra.hashnode.dev/all-about-javascript-event-loop

Implementação

```
// forma anterior ao ES2017

function fetchDados() {
  fetch('http://site.com/dados').then(response => {
    response.json().then(dados => {
      console.log(dados);
    })
  })
}

// forma atual
async function fetchDadosAsync() {
  const response = await fetch('http://site.com/dados');
  const dados = await response.json();
  console.log(dados);
}

// Usando uma Promise para atrasar a execução de uma função

function esperar(ms) {
  // Aqui, usamos o parâmetro resolve para indicar quando a Promise foi resolvida
  // quando chamamos resolve(), o Laço de Eventos do JavaScript termina a tarefa
  function minhaFuncao(resolve) {
    console.log('executando minhaFuncao');
    setTimeout(resolve, ms)
    console.log('terminei!');
  }

  return new Promise(resolve => minhaFuncao(resolve));
}
```

Figura B.8: *Promise based async*

Decorator Padrão de Projeto

Problema

Ao implementar um componente, é comum que tenhamos que adicionar diversas regras para a visualização. Contudo, os componentes podem acumular responsabilidades demais e se tornarem complexos. Dessa forma, precisamos arranjar maneiras de separar diferentes lógicas para reaproveitá-las melhor. Apesar de podermos compor componentes com Providers, por vezes é interessante que possamos fazer essas mudanças de forma dinâmica.

Exemplo

Você precisa construir um componente que apresenta textos em diferentes formatos para um blog. Este componente já contém diversas regras de apresentação que foram implementadas nele. Com o tempo, pedem para que os textos possam ser exibidos em várias línguas, para atender um novo público de clientes que falam outras línguas.

Inserir essa nova regra dentro do componente exigirá a mudança de várias partes dele e ele terá mais responsabilidades que no começo. Isso pode torná-lo difícil de ler e também de manter.

Contexto

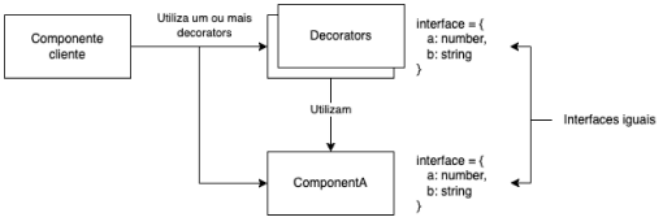
- Precisamos adicionar funcionalidades sem alterar suas regras internas
- A regra de exibição depende de alguns fatores externos como a linguagem do usuário
- Ainda precisamos do componente original em algumas situações

Solução

O padrão Decorator oferece uma forma simples de adicionar novas funcionalidades a componentes sem alterar a estrutura interna deles e promovendo o reuso do componente. O padrão funciona a partir de funções que criam novos componentes a partir de componentes criados anteriormente.

Podemos compor mais de um Decorator no mesmo componente, desde que as interfaces entre os decorators usados sejam compatíveis. Os componentes que serão "decorados" tem uma lógica mínima para trabalhar com os Decorators e podem até mesmo não serem modificados, mantendo o princípio de propósito único do componente.

Diagrama



Implementação

```
export const PostDoBlog = ({ titulo, subtitulo, corpoDoTexto }) => (
  <>
    <bold>{autor}</bold>
    <h1>{titulo}</h1>
    <h3>{subtitulo}</h3>
    <p>{corpoDoTexto}</p>
  </>
);

export const DecoratorTradutorParaOIngles = (ComponentePostDoBlog) => {
  const tradutor = useTradutor('en-us');

  const componenteTraduzido = ({ titulo, subtitulo, corpoDoTexto }) => (
    <ComponentePostDoBlog
      titulo={tradutor.traduzir(titulo)}
      subtitulo={tradutor.traduzir(subtitulo)}
      corpoDoTexto={tradutor.traduzir(corpoDoTexto)}
    />
  );

  return componenteTraduzido;
};

export const PaginaDoBlog = (linguagemDoUsuario) => {
  if (linguagemDoUsuario === 'pt-br') {
    return (
      <PostDoBlog
        titulo="Título do post"
        subtitulo="Subtítulo do post"
        corpoDoTexto="Corpo do post"
      />
    );
  } else if (linguagemDoUsuario === 'en-us') {
    const PostDoBlogTraduzido = DecoratorTradutorParaOIngles(PostDoBlog);
    return (
      <PostDoBlogTraduzido
        titulo="Título do post"
        subtitulo="Subtítulo do post"
        corpoDoTexto="Corpo do post"
      />
    );
  }
};
```

Figura B.9: *Decorator*

Data Binding Expressão linguística

Problema

Ao criarmos estado em aplicações front-end, é importante que a atualização dos dados seja refletida por toda a aplicação. Em JavaScript puro, essa atualização deve ser declarada a todos os momentos. É comum que usemos eventos para notificar a mudança de dados e cada componente deve consumir esse evento e atualizar-se conforme sua regra própria. Contudo, essa abordagem facilmente se torna complexa e pouco reutilizável entre componentes, sendo necessário implementar o mesmo método de escuta para cada dado modificável.

Além disso, por vezes a modificação do dado pode acontecer em vários locais diferentes. Tanto o modelo de dados quanto os inputs da aplicação podem ter controle sobre o dado, gerando uma via de mão dupla de atualizações.

Exemplo

Imagine que você esteja implementando um site que monitora dados de bolsas de valores, como valores atuais de ações e outras métricas personalizadas. Em determinado momento, você precisa dar suporte a contrução de gráficos personalizados. Esses gráficos devem mostrar uma pré-visualização de como ficarão em tempo real. Para implementar esta lógica, o gráfico de pré-visualização deve consumir alterações nos dados originais do gráfico em tempo real.

Contexto

- Temos um dado que deve ser visto da mesma forma por todos os componentes
- Modificações neste dado devem ser notificadas a todos os componentes que o utilizam
- Tanto a origem do dado quanto os seus consumidores devem poder alterar o dado caso seja necessário

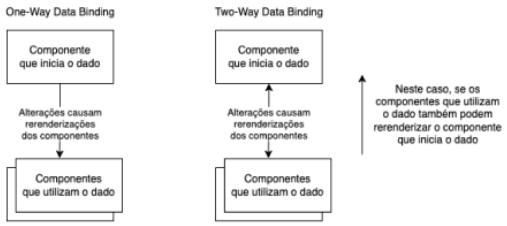
Solução

Data Binding é uma técnica comum em front-ends para sincronizar variáveis diferentes que representam o mesmo dado. É possível estabelecer uma analogia de ponteiros, comumente usados em C ou C++, e o Data Binding. Ambas as técnicas entregam uma variável que pode ser modificada em vários lugares do código. Contudo, o Data Binding entrega, além disso, uma forma de reagirmos a atualizações da mesma forma que o padrão Observer. O Data Binding é implementado pela maioria dos frameworks JavaScript dada a sua recorrente necessidade na construção de UIs.

O Data Binding pode ser implementado de duas formas diferentes: *One-way* ou *Two-way* data binding. Na primeira forma, somente a origem do dado pode modificá-lo e podemos ter qualquer quantidade de consumidores, de forma muito similar ao padrão Observer. Já na segunda forma, todos aqueles que utilizam o dado podem modificá-lo, dessa forma, todas as variáveis geradas a partir daquele dado tem o mesmo poder sobre ele.

Diferentemente de frameworks como o Angular em que o data binding deve ser declarado literalmente, o React implementa diversas abstrações para que não tenhamos que lidar diretamente com o data binding. Por esse motivo, o React é geralmente considerado uma framework declarativa.

Diagrama



Implementação

```jsx
// One-Way Data Binding

export function GraficoDeValoresDaBolsa () {
  // o useState cria um dado que é observado por todos os componentes que o utilizam
  const [valores] = useState([1, 2, 3, 4, 5, 6, 7, 8, 9]);

  return (
    <div>
      {/* Aqui, associamos o componente Grafico ao valor de "valores" */}
      <Grafico valores={valores} />
    </div>
  )
}

export function Grafico({ valores }) {
  // Toda vez que o valor de "valores" mudar, o componente será re-renderizado
  // Assim, a variável "valores" (que é uma variável nova aqui)
  // é um Data Binding do dado "valores" do componente pai
  return (
    <div>
      {valores.map((valor) => (
        <ComponenteVisual valor={valor} />
      ))}
    </div>
  )
}

// Two-Way Data Binding

export function Contador () {
  // Com o setValor, podemos alterar o "valor"
  const [valor, setValor] = useState(0);

  return (
    <div>
      <p>{valor}</p>
      {/*
        Aqui, novamente associamos o componente a variável valor
        Porém, agora o componente também pode alterar o valor
        Dessa forma, temos um Two-Way Data Binding
      */}
      <BotaoDeIncremento valor={valor} setValor={setValor} />
    </div>
  )
}

export function BotaoDeIncremento({ valor, setValor }) {
  return (
    // Chamando a função setValor, alteramos o dado tanto aqui quanto no componente pai
    <button onClick={() => setValor(valor + 1)}>Incrementar</button>
  )
}
```

Figura B.10: *Data binding*

Referências

[AWS 2023] AWS. *Qual é a diferença entre front-end e back-end no desenvolvimento de aplicações?* 2023. URL: https://aws.amazon.com/pt/compare/the-difference-between-frontend-and-backend/ (acesso em 23/10/2023).

[Berners-Lee 1989] Timothy J Berners-Lee. "Information management: a proposal". *CERN* (1989).

[Bos 2016] Bert Bos. *A brief history of CSS until 2016*. 2016. URL: https://www.w3.org/Style/CSS20/history.html (acesso em 15/11/2023).

[Buschmann 1996] Frank et al Buschmann. *Pattern-Oriented Sofware Architecture*. 1ª ed. John Wiley Sons Ltd, 1996.

[Calçado 2015] Phil Calçado. *The Back-end for Front-end Pattern (BFF)*. 2015. URL: https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html (acesso em 07/10/2023).

[Cambridge 2023] University Press Cambridge. *Meaning of front end in English*. 2023. URL: https://dictionary.cambridge.org/us/dictionary/english/front-end (acesso em 23/10/2023).

[Docs 2020a] MDN Web Docs. *CSS: Cascading Style Sheets*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (acesso em 21/11/2023).

[Docs 2020b] MDN Web Docs. *HTML: HyperText Markup Language*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (acesso em 21/11/2023).

[Gamma *et al.* 1994] Erich Gamma, Richard Helm, Ralph Johnson e John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1ª ed. Addison-Wesley Professional, 1994.

[Guru 2023] Spring Framework Guru. *Gang of Four Design Pattern*. 2023. URL: https://springframework.guru/gang-of-four-design-patterns/ (acesso em 25/10/2023).

[He e Luo 2000] Xiaoxin He e Jun Luo. "Fengshui and the environment of southeast china". *Worldviews* 4.3 (2000), pp. 213–234. ISSN: 13635247, 15685357. URL: http://www.jstor.org/stable/43809172 (acesso em 17/10/2023).

[International 2023] ECMA International. *ECMAScript® 2023 language specification*. 2023. URL: https://ecma-international.org/publications-and-standards/standards/ecma-262/ (acesso em 20/11/2023).

[JavaScript 2022] State of JavaScript. *Front-end Frameworks*. 2022. URL: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/ (acesso em 25/10/2023).

[Norman e Nielsen 2023] Don Norman e Jakob Nielsen. *The Definition of User Experience (UX)*. 2023. URL: https://www.nngroup.com/articles/definition-user-experience/ (acesso em 23/10/2023).

[Nuval 2020] Charlene Nuval. *3 ways UX Design and Feng Shui Overlap*. 2020. URL: https://uxplanet.org/3-ways-ux-design-and-feng-shui-overlap-c93eace46012 (acesso em 30/10/2023).

[Rauschmayer 2014] Dr. Axel Rauschmayer. *Speaking Javascript*. 1ª ed. O'Reilly Media, Inc., 2014.

[Richardson 2015] Chris Richardson. *A pattern language for microservices*. 2015. URL: https://microservices.io/patterns/index.html (acesso em 07/10/2023).

[Shvets 2021a] Alexander Shvets. *Classificação dos padrões*. 2021. URL: https://refactoring.guru/pt-br/design-patterns/classification (acesso em 23/10/2023).

[Shvets 2021b] Alexander Shvets. *Por que devo aprender padrões?* 2021. URL: https://refactoring.guru/pt-br/design-patterns/why-learn-patterns (acesso em 29/10/2023).

[Sobczak 2023] Michał Sobczak. *24 Top Frontend Technologies to Use in 2023*. 2023. URL: https://www.netguru.com/blog/front-end-technologies (acesso em 24/10/2023).

[Souto 2023] Mario Souto. *Front-end, Back-end e Full Stack*. 2023. URL: https://www.alura.com.br/artigos/o-que-e-front-end-e-back-end (acesso em 23/10/2023).

[Tidwell 2011] Jenifer Tidwell. *Designing interfaces*. 3ª ed. O'Reilly, 2011.

[Vailshery 2023] Lionel Sujay Vailshery. *Number of software developers worldwide in 2018 to 2024*. 2023. URL: https://www.statista.com/statistics/627312/worldwide-developer-population/ (acesso em 24/10/2023).

[Wells 2018] Christopher J. Wells. *A brief history of CSS until 2016*. 2018. URL: https://www.technologyuk.net/website-development/introduction-to-css/introduction.shtml (acesso em 15/11/2023).