Relatório de justificativa para escolha do design do código

Visão geral

Este documento tem como fim apresentar justificativas para a escolha do design de código que foi utilizado na elaboração da proposta do projeto da startup do ramo de alimentos.

Objetivos

1. **Apresentar o problema:** Apresentar os problemas/dificuldades que culminaram na solução final.
2. **Justificar a solução:** Apresentar argumentos que deem embasamento para o design de código que foi utilizado para a solução do problema identificado.

Dificuldades/Problemas

1. Comunicação entre a o server-side e o client-side

   Separar o back-end do front-end e fazer com que ambos se comuniquem entre si

2. Separação da interface de comunicação externa do back-end, das regras de negócio e da comunicação com o banco de dados.

   Estruturar de forma adequada e escalável a interface que se comunicaria com o front-end, as regras de negócio propostas no projeto e a camada de comunicação com o banco de dados.

3. Modularizar as regras de negócio

   Separar as regras de compra de lanches do cardápio, de criação de novos lanches e aplicação de promoções para tornar fácil a manutenção e os testes automatizados.

4. Criar componentes reutilizáveis e com fácil comunicação entre si no front-end

   Criar componentes que podem ser facilmente utilizados em várias páginas do projeto e com comunicação global, para que códigos e estilos não estejam repetidamente explícitos em todas as partes do projeto

Soluções

1. Comunicação entre a o server-side e o client-side

   Para a troca dados entre as duas aplicações (client-side e server-side), foi escolhida a arquitetura REST, por conta da sua agilidade no processamento das requisições, facilidade para integrar os sistemas e flexibilidade na troca de dados, além de ser o protocolo de comunicação entre sistemas mais difundido e conhecido do mercado.

2. Separação da interface de comunicação externa do back-end, das regras de negócio e da comunicação com o banco de dados.

   Para uma boa legibilidade e arquitetura do código fonte é extremamente importante que o projeto esteja dividido em camadas bem estruturadas. A arquitetura escolhido para o sistema contém a seguinte estrutura:

   - **Endpoints:** camada que recebe as requisições do front-end e direciona para as classes de negócio.
   - **Model:** camada que contém a estrutura das tabelas do banco de dados e o mapeamento de entidades do JPA.
   - **Repository:** camada de comunicação com o banco de dados.
   - **Service:** camada que contém as regras de negócio de cada objeto e que faz a ponte entre a camada dos **Endpoints** e dos **Repositories**.

3. Modularizar as regras de negócio

   Dentro da camada de **Service** também não é ideal que todo um fluxo esteja dentro de somente um método, o que dificulta a manutenção e também a realização dos testes. Por conta disso, foram criados diversos métodos para executar funções específicas e bem definidas, tornando futuras manutenções mais fáceis e também tornando o código mais legível.

4. Criar componentes reutilizáveis e com fácil comunicação entre si no front-end

   Para a criação de componentes reutilizáveis e com comunicação global entre si, foi escolhido o **React** dentre os frameworks javascript. Com a instalação de algumas bibliotecas, o framework fornece um grande poder e maleabilidade na criação do front-end, trazendo componentes reutilizáveis, estado global para a aplicação, operações assíncronas e uma boa interação com o usuário. As bibliotecas utilizadas foram:

   - **React Bootstrap:** biblioteca que fornece componentes com a arquitetura do **React** e com design do **Bootstrap**, o que diminui o esforço e aumenta a produtividade na criação de layouts responsivos e interativos.
   - **React Router DOM:** biblioteca que fornece configurações para a navegação entre os componentes da aplicação por meio do browser.
   - **Redux / React Redux:** biblioteca que fornece a estrutura para tornar o estado da aplicação global, podendo ser acessado e modificado dentro de qualquer componente.

- **Redux Saga:** biblioteca que fornece um **middleware** para a biblioteca do **Redux**, possibilitando a criação de métodos assíncronos que façam a ponte entre o estado global da aplicação e as requisições assíncronas para o server-side, já que somente o **Redux** não é suficiente para fazer isso de forma adequada.

Com essas bibliotecas configuradas, a aplicação se estrutura da seguinte maneira:

- **Components:** componentes reutilizáveis e/ou globais dentro do projeto.
- **Configurations:** arquivos de configuração de todas as bibliotecas mencionadas acima, unindo-as entre si.
- **Reducers:** containers de estado global, cada um mantendo o estado relacionado a um objeto do banco de dados.
- **Sagas:** métodos que fazem a interligação entre os reducers e as requisições assíncronas para o server-side .
- **Services:** camada que faz as requisições para o server-side.
- **Views:** telas da aplicação.

Com esse design de código, tanto a aplicação front-end como a api no back-end se tornam escaláveis e bem arquitetadas para futuras manutenções e adições de funcionalidades.