Projeto custos-transporte

1. Para este projeto, de nome **custos-transporte**, criado para atender ao **Exercício 1** da avaliação, foi utilizado a linguagem **Java,** o **Spring Boot**, o seu servidor **Tomcat** embarcado e o banco de dados em memória **H2**. Esse framework foi utilizado por criar toda a infraestrutura necessário, possibilitando que o meu foco fosse somente na lógica da aplicação.

2. Como foi utilizado o **Spring Boot** e **Maven** para criar esse projeto, os passos necessários para executar a aplicação são os seguintes:

   2.1. Fazer o clone do git ou download dos fontes do sistema no meu repositório GitHub:
   https://github.com/Paulo-E-F-Fernandes/custos-transporte

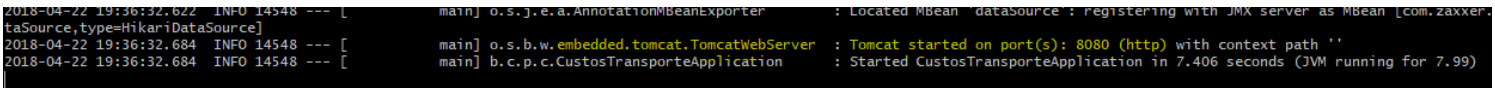
   2.2. Comando para o clone do repositório pelo git:
   git clone https://github.com/Paulo-E-F-Fernandes/custos-transporte.git

   2.3. Acessar o diretório raiz do projeto, de nome **custos-transporte** e nesse diretório será possível ver o **pom.xml**. Utilizar o comando **Maven** para compilar o projeto:
   mvn clean install

   2.4. Após a compilação, será criado no diretório **target** o seguinte arquivo:
   custos-transporte-0.0.1-SNAPSHOT.jar

   2.5. Pelo terminal, acessar o diretório **target** e executar o comando abaixo. Sugiro digitar o comando, pois ao copiar pode ser levada alguma sujeira ou carácter escondido do visualizador e não executar o comando.
   java -jar custos-transporte-0.0.1-SNAPSHOT.jar

   2.6. Após verificar no terminal que o **Spring Boot** subiu o servidor, é possível acessar o sistema através da URL **http://127.0.0.1:8080/simulacao** ou qualquer outra porta que o **Spring Boot** indicar, conforme a imagem abaixo:



3. Para o projeto, no **back-end**, além de **Java,** do **Spring Boot** com **Tomcat** embarcado e banco de dados **H2** que foram mencionados acima, utilizei também o **Spring-Data-JPA**, os design patterns **Template Method** e **Chains of Responsability**.

   3.1. O patterns **Template Method** foi utilizado na classe abstrata **CustoTemplateMethod** que implementa o método **executar** da **Interface IOperacao**. Dentro do método **executar** é chamado o método abstrato **calcularCusto** que é implementado por quem herdar de **CustoTemplateMethod** e após o processamento do **calcularCusto** é feita a chamada da execução do cálculo do próximo custo da cadeia.

```java
public abstract class CustoTemplateMethod implements IOperacao {

    protected final MathContext mc = new MathContext(10, RoundingMode.HALF_UP);
    protected IOperacao proximo;

    protected abstract void calcularCusto(Orcamento orcamento);

    @Override
    public void executar(Orcamento orcamento) {
        calcularCusto(orcamento);
        proximo.executar(orcamento);
    }

    @Override
    public void setProximo(IOperacao proximo) {
        this.proximo = proximo;
    }

}
```

3.2. O **Chains of Responsability** foi utilizado para criar uma cadeia de fluxo de cálculo dos custos, sendo que se algum outro custo é necessário ser adicionado, só precisarei criar uma classe que estenderá a classe **CustoTemplateMethod** e implementará a lógica de calculo no método **calcularCusto** e por fim, adicionar a instância dessa nova classe na cadeia. Com isso a ampliação do sistema ficou muito simples. A última classe que deve ser adicionada na cadeia é **FinalizaProcessamento** que não possui um próximo definido, encerrando o processamento do custo do transporte.

4. Para o **Front-end**, foi utilizado o **Thymeleaf**, **Bootstrap 4** para a estilização e uma lib js (jquery.masknumber.min.js) para aplicação de máscara nos campos do formulário. Os arquivos **app.js** e **style.css** possui as poucas customizações que foram necessárias fazer.