UNIVERSIDADE CATÓLICA DO SALVADOR

Documentação do Projeto: Intellijentes-FluxRoute Sistema com Injeção de Dependências e Rotas Dinâmicas

Alison Andrade Rocha
Caio Caldas Matos de Morais
João Vitor Senhorinho Guilherme
Lucas Alves Matos Sampaio Gaspari
Marcelo Vinicius Lins Dantas Gomes
Yuri Brito de Souza
Gabriel Teixeira Cotrim Marques
Gustavo Cruz Ramos da Silva

Salvador
2024

1. Introdução

Este projeto é um sistema Java que utiliza conceitos avançados de design de software, como **injeção de dependências, reflexão**, e **padrões de projeto** (Command, Singleton e Factory Method).
Ele é projetado para oferecer flexibilidade e escalabilidade, permitindo:

- Adicionar novas **rotas** e **repositórios** de forma dinâmica, sem configurações manuais.
- Gerenciar a **injeção automática de dependências** através de reflexão e anotações personalizadas.
- Garantir **consistência de dados** em repositórios, seguindo o padrão Singleton.

Objetivos principais

- **Roteamento Dinâmico:** Um único servlet gerencia todas as requisições e despacha para os métodos apropriados com base nas rotas definidas.
- **Injeção de Dependências Automática:** Usar reflexão para injetar repositórios e serviços automaticamente.
- **Padrão Singleton**: Assegurar que os repositórios em memória sigam o padrão Singleton, evitando múltiplas instâncias e inconsistências.

Pré-requisitos

- **Java 11 ou superior;**
- **Maven 3.6 ou superior;**
- **IDE de sua preferência**, recomendamos EclipseIDE ou IntelliJ IDEA.

Configuração do Ambiente

1. Clone o repositório

git clone https://github.com/usuario/Intellijentes-FluxRoute.git

2. Abra sua IDE
3. Selecione File > Open e escolha o diretório do projeto clonado.
4. Aguarde a IDE importar o projeto e baixar as dependências Maven.

Execução do Projeto

Para compilar e executar o projeto, use os seguintes comandos Maven:

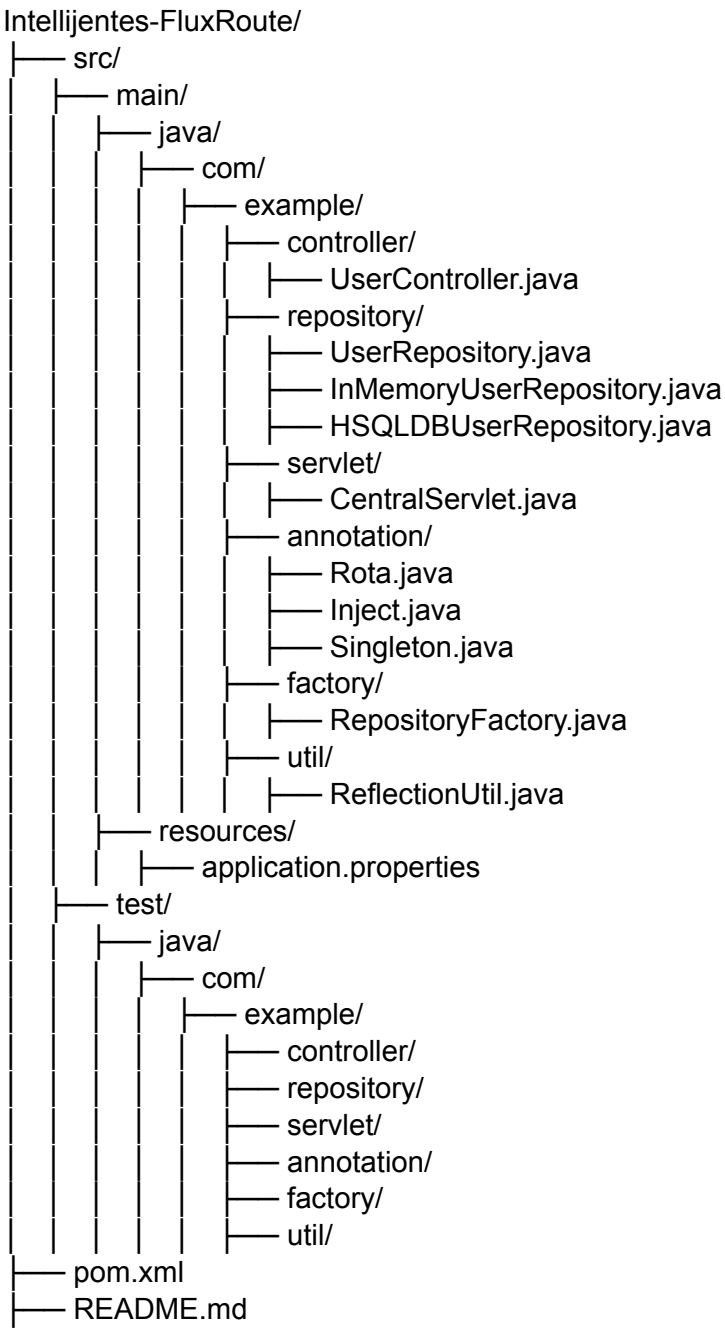
```
mvn clean install

mvn spring-boot:run
```

Licença

2. Estrutura do Projeto

O projeto segue uma estrutura modular, dividida em pacotes bem definidos. A seguir está o diretório principal e os pacotes organizados:



3. Anotações Utilizadas

3.1 Anotação @Rota

- **Propósito:** Mapear métodos de controladores a rotas HTTP específicas.
- **Funcionamento:** O sistema escaneia as classes de controle por métodos anotados com @Rota, mapeando automaticamente as rotas HTTP às suas respectivas implementações.

Exemplo de uso:

```
@Rota(path = "/adicionarProduto")
public class ProdutoAdicionarServlet implements Command {
    private static final long serialVersionUID = 1L;
```

3.2 Anotação @Inject

- **Propósito**: Automatizar a injeção de dependências, como repositórios ou serviços, em classes controladoras.

- **Funcionamento:** Através de reflexão, o sistema identifica os campos anotados com @Inject e injeta as instâncias necessárias dinamicamente.

Exemplo de uso:

```
@Inject
private ProdutoService produtoService;
```

3.3 Anotação @Singleton

- **Propósito**: Garantir que uma classe possua apenas uma instância ao longo da execução da aplicação.
- **Funcionamento:** O sistema controla a criação de instâncias das classes marcadas com @Singleton, garantindo que todos os consumidores acessem a mesma instância.

Exemplo de uso:

```
@Singleton
public class MemoriaProdutoRepository implements ProdutoRepository<Produto, Integer>{
```

3.4 Anotação @WebServlet

Propósito: Configura e mapeia uma classe como um servlet para tratar requisições HTTP.

Funcionamento: O contêiner de servlets (como Tomcat ou Jetty) identifica e registra os servlets baseados nessa anotação, associando-os às rotas configuradas.

```
@WebServlet("/view/*")
public class ProdutoController extends HttpServlet {
```

3.5 Anotação @WebListener

Propósito: Registrar classes que implementam interfaces de listener, como ServletContextListener, HttpSessionListener ou ServletRequestListener.

Funcionamento:

- O contêiner web (como Tomcat ou Jetty) detecta automaticamente as classes anotadas com @WebListener e registra os eventos para os quais elas estão configuradas.

```java
@WebListener
public class InicializadorListener implements ServletContextListener {
```

3.6 Anotação @Target

Propósito:

Definir o tipo de elementos do código (como classes, métodos, campos, etc.) aos quais uma anotação pode ser aplicada.

Funcionamento:

- A anotação @Target especifica o tipo de elemento em que uma anotação pode ser utilizada.
- No exemplo da anotação @Singleton, foi usado ElementType.TYPE, o que significa que a anotação pode ser aplicada apenas a classes.
-
```java
@Target(ElementType.TYPE)
```

3.7 Anotação @Retention

Propósito:

Especificar o momento em que a anotação deve ser mantida no código e se estará disponível para o processamento em tempo de execução.

Funcionamento:

- A anotação @Retention define se a anotação será mantida apenas no código fonte, durante a compilação, ou se estará disponível durante a execução.

```java
@Retention(RetentionPolicy.RUNTIME)
```

4. Reflexão e Injeção de Dependências

4.1 Carga Dinâmica de Rotas

Descrição:
O sistema utiliza reflexão para escanear automaticamente as classes de controle, que são anotadas com a anotação @Rota, e registrar rotas HTTP dinâmicas para os respectivos métodos. A anotação @Rota define o caminho da URL associado ao método, permitindo que o sistema registre automaticamente as rotas e associe-as aos métodos apropriados de maneira dinâmica.

Vantagem:
A carga dinâmica de rotas permite adicionar ou alterar rotas sem necessidade de configurações manuais no código. Isso torna o sistema mais flexível e escalável, uma vez que novos controladores ou métodos podem ser introduzidos sem modificar configurações centrais.

Exemplo de fluxo:

1. Ao iniciar a aplicação, o sistema escaneia as classes de controle para identificar métodos anotados com @Rota.
2. Para cada método encontrado, a rota definida no parâmetro path da anotação @Rota é registrada e associada à implementação do método correspondente.
3. Quando uma requisição HTTP é recebida, o sistema verifica a URL e despacha a execução para o método do controlador que corresponde à rota registrada.

Exemplo no código:

```
@Rota(path = "/adicionarProduto")
public class ProdutoAdicionarServlet implements Command {
 private static final long serialVersionUID = 1L;

 @Inject
 private ProdutoService produtoService;
```

4.2 Injeção de Dependências

Descrição:
A injeção de dependências é automatizada através da reflexão, facilitando a instância e a injeção de serviços ou repositórios necessários em classes controladoras. A anotação @Inject é usada para marcar os campos que devem ser automaticamente preenchidos com instâncias das dependências adequadas, como o serviço ProdutoService no exemplo abaixo.

Vantagem:
Essa abordagem reduz o acoplamento entre as classes e simplifica a criação de instâncias, permitindo que o sistema gerencie a criação e o ciclo de vida das dependências automaticamente. Além disso, facilita a modularização e o teste das classes de forma isolada.

Funcionamento:

1. Durante a inicialização do sistema, o contêiner de injeção de dependências (gerido pelo mecanismo de reflexão) identifica os campos anotados com @Inject.
2. O contêiner resolve e instancia as dependências necessárias.
3. As dependências são injetadas automaticamente nos campos anotados, evitando a necessidade de inicialização manual.

5. Padrão Factory Method e Troca de Repositórios

Descrição:
A classe PersistenciaFactory implementa o padrão de projeto **Factory Method**, que permite a criação de diferentes tipos de repositórios de maneira flexível e dinâmica. Ela decide, com base em um parâmetro fornecido, qual tipo de repositório (em memória ou HSQLDB) será utilizado para armazenar e recuperar dados. Essa abordagem possibilita a troca entre os repositórios sem que o código que utiliza o repositório precise conhecer os detalhes de implementação de cada um.

A fábrica encapsula a lógica de escolha e criação de instâncias dos repositórios, delegando essa responsabilidade à classe PersistenciaFactory. Isso torna o código mais modular, pois o tipo de persistência pode ser alterado facilmente, sem impacto no restante da aplicação.

Funcionamento:

A fábrica de repositórios possui um método estático getProdutoRepository(), que recebe um tipo de repositório como parâmetro (no caso, MEMORIA ou HSQL) e retorna a instância do repositório correspondente. Dependendo do tipo passado, o método instanciará o repositório em memória utilizando o **SingletonManager** para garantir que apenas uma instância seja criada ou criará uma nova instância de repositório HSQLDB.

1. Quando o tipo MEMORIA é informado, o repositório em memória é retornado, e sua instância é garantida como única, utilizando o padrão **Singleton**.
2. Quando o tipo HSQL é informado, um novo repositório HSQL é instanciado.
3. O padrão **Factory Method** assegura que o código cliente (que chama o método getProdutoRepository()) não precise se preocupar com a implementação específica do repositório.

Exemplo de uso:

```
public class PersistenciaFactory {


    public static final int MEMORIA = 0;
    public static final int HSQL = 1;


    public static ProdutoRepository<?, ?> getProdutoRepository(int type) {
        ProdutoRepository<?, ?> produtoRepository;

        switch (type) {
            case MEMORIA: {
            produtoRepository = SingletonManager.getInstance(MemoriaProdutoRepository.class);
            break;
        }
            case HSQL: {
            produtoRepository = new HSQLProdutoRepository();
            break;
        }
        default:
            throw new IllegalArgumentException("Unexpected value: " + type);
        }
        return produtoRepository;
    }
}
```

6. Padrão Singleton

Descrição:

O **Padrão Singleton** é um padrão de design estrutural utilizado para garantir que uma classe tenha **apenas uma instância** durante toda a execução da aplicação. No contexto deste projeto, ele é usado para garantir que os repositórios em memória, como o MemoriaProdutoRepository, possuam uma única instância, preservando a **consistência dos dados** durante o ciclo de vida da aplicação.

A anotação @Singleton é usada para marcar as classes que devem seguir o padrão Singleton. Quando uma classe é anotada com @Singleton, o sistema se assegura de que, independentemente do número de vezes que a classe for requisitada, ela sempre retornará a mesma instância.

Funcionamento:

O padrão **Singleton** controla a criação de instâncias da classe. Quando uma classe é anotada com @Singleton, o sistema gerencia sua instância de forma que apenas uma instância única da classe será criada e compartilhada em toda a aplicação.

Para garantir que a classe Singleton seja corretamente instanciada, o projeto utiliza um **SingletonManager**, que armazena a instância da classe e a fornece quando requisitada. Isso assegura que a instância seja criada apenas uma vez e reutilizada por toda a aplicação, evitando inconsistências de dados.

1. **Marca a classe como Singleton**: A anotação @Singleton é aplicada em classes que devem ter uma única instância.
2. **Gerenciamento de instância**: O **SingletonManager** é responsável por gerenciar a criação e o acesso à instância da classe marcada como Singleton.
3. **Evita múltiplas instâncias**: O sistema assegura que, em qualquer lugar do código, a mesma instância da classe será utilizada, evitando inconsistências nos dados.

Exemplo de uso:

```
@Singleton
public class MemoriaProdutoRepository implements ProdutoRepository<Produto, Integer>{
```

7. Diagramas

7.1 Diagrama de Classes

O diagrama de classes que detalha a arquitetura e os relacionamentos do sistema está disponível no Anexo [1] ao final deste documento.

7.2 Diagrama de Sequência

1. Adicionar Produto

   O fluxo para adicionar um produto é descrito:

   - O usuário acessa a página de adição de produto.
   - O usuário preenche o formulário com os detalhes do produto.
   - O usuário submete o formulário.
   - O sistema valida os dados do produto.
   - O sistema adiciona o produto ao repositório.
   - O sistema confirma a adição do produto ao usuário.

2. Listar Produtos

   O fluxo para listar produtos é descrito:

   - O usuário acessa a página de listagem de produtos.
   - O sistema recupera a lista de produtos do repositório.
   - O sistema exibe a lista de produtos ao usuário.

3. Editar Produtos

   O fluxo para editar um produto é descrito:

   - O usuário acessa a página de edição de produto.
   - O usuário seleciona o produto a ser editado.
   - O sistema exibe o formulário de edição com os detalhes do produto.
   - O usuário modifica os detalhes do produto.
   - O usuário submete o formulário.
   - O sistema valida os dados do produto.
   - O sistema atualiza os detalhes do produto no repositório.
   - O sistema confirma a atualização do produto ao usuário.

4. Excluir Produtos

   O fluxo para excluir um produto é descrito:

   - O usuário acessa a página de listagem de produtos.
   - O usuário seleciona o produto a ser excluído.
   - O sistema solicita confirmação da exclusão.
   - O usuário confirma a exclusão.
   - O sistema remove o produto do repositório.
   - O sistema confirma a exclusão do produto ao usuário.

Diagrama de Uso:

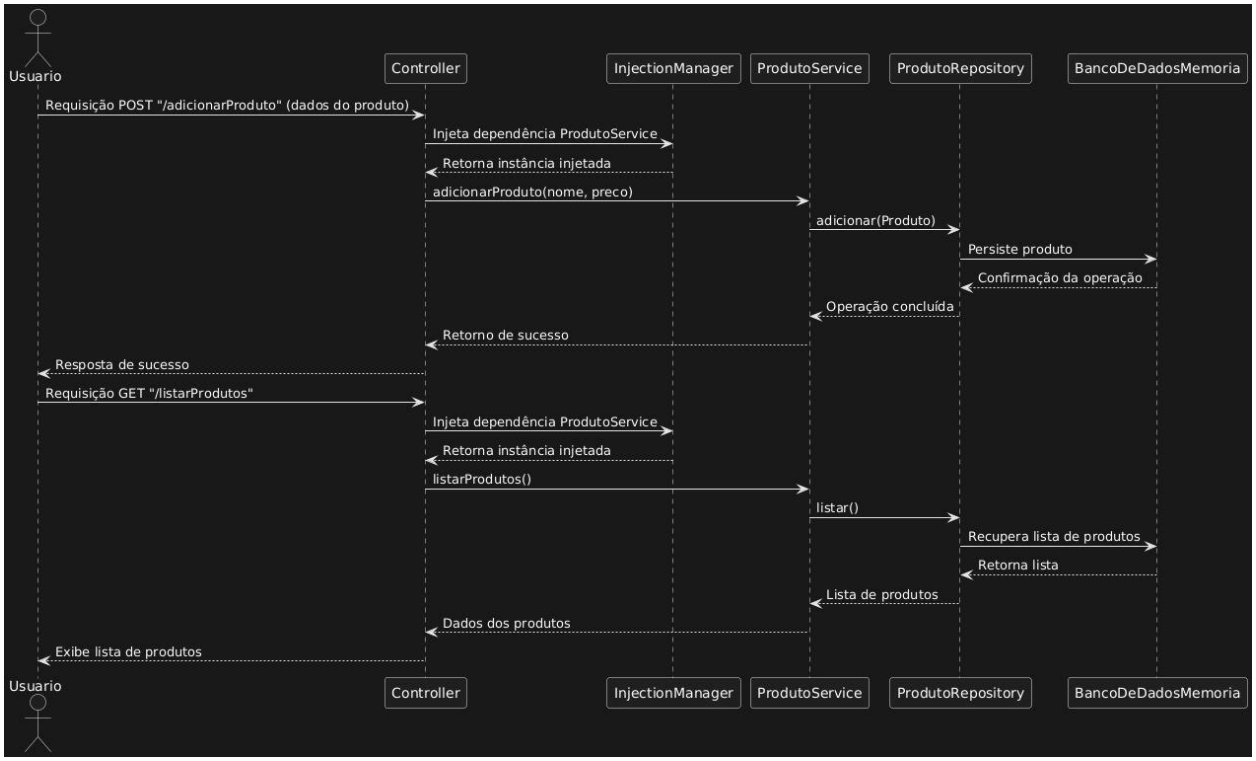


Imagem 1 - Diagrama de Uso

8. Conclusão

Este projeto demonstra conceitos modernos de design de software, como:

- **Padrões de Projeto**: Command, Singleton e Factory Method.
- **Reflexão**: Para roteamento dinâmico e injeção de dependências.
- **Anotações Personalizadas**: Simplificando a manutenção e escalabilidade.

A solução é flexível e escalável, permitindo alterações dinâmicas sem impactar a estrutura principal. Assim, ela atende a requisitos de modularidade, consistência de dados e facilidade de manutenção.

9. Anexos

Anexo 1 - Diagrama de Classes.

pkg

controller

managers

InjectManager

+ ResolveDependencias() : void
- createDependencyForField() : object
- instantiateDependency() : object

SingletonManager

+ getInstance() : generic
- createInstance() : generic

Operations

Interface **Command**
+ execute() : void

Annotation **Implementation Of**

Annotation **Inject**

Annotation **Rota**

Annotation **Singleton**

Implements

Implements

Implements

Implements

ProdutoAdicionarServlet
- serialVersionUID : longfinal = 1
- produtoService : ProdutoService

+ execute() : void
- doGet() : void
- doPost() : void

ProdutoEditarServlet
- serialVersionUID : longfinal = 1
- produtoService : int

+ execute() : void
- doGet() : void
- doPost() : void

ProdutoExcluirServlet
- serialVersionUID : longfinal = 1
- produtoService : ProdutoService

+ execute() : void

InicializadorListener
- commands : HashMap

+ contextInitialized() : void
- runOperations() : void
+ contextDestroyed() : void

ProdutoListarServlet
- serialVersionUID : longfinal = 1
- produtoService : ProdutoService

+ execute() : void

ProdutoController

service() : void

model

Produto
- id : integer
- nome : String
- preco : double

+ getId() : integer
+ setId() : void
+ getNome() : String
+ setNome() : void
+ getPreco() : double
+ setPreco() : void

persistencia

PersistenciaFactory
+ MEMORIA : intfinal = 0
+ HSQL : intfinal = 1

+ getProdutoRepository() : ProdutoRepository

HSQLProdutoRepository

+ adicionar() : void
+ remover() : void
+ listar() : list
+ atualizar() : void
+ obterPorID() : Produto

Implements

MemoriaProdutoRepository
- currentId : AtomicInteger = 1
- produtos : HashMap
- Instancia : MemoriaProdutoRepository

+ adicionar() : void
+ atualizar() : void
+ remover() : void
+ listar() : list
+ obterPorID() : Produto
+ getInstancia() : MemoriaProdutoRepository

Implements

Interface **ProdutoRepository**

+ adicionar() : void
+ remover() : void
+ atualizar() : void
+ listar() : list
+ obterporId() : Produto

Service

ProdutoService
- produtoRepository : ProdutoRepository

+ adicionarProduto() : void
+ removerProduto() : void
+ obterProdutoPorId() : Produto
+ atualizarProduto() : void
+ listarProdutos() : list

util

DatabaseInitializationListener

+ contextInitialized() : void
+ contextDestroyed() : void

DatabaseUtil
- URL : StringFinal
- USER : StringFinal
- PASSWORD : StringFinal

+ iniciarBanco() : void
+ getConnection() : Connection