

Módulo Caixa – Documentação:

Dependências:

- **Spring Web:** Essencial para criar aplicações web e APIs REST.
- **Spring Data JPA:** Facilita a comunicação com o banco de dados de uma forma muito produtiva.
- **H2 Database:** Um banco de dados em memória. É perfeito para desenvolvimento, pois não precisamos instalar nada. Ao reiniciar a aplicação, os dados são apagados.
- **Lombok:** Uma biblioteca fantástica que reduz a quantidade de código repetitivo que precisamos escrever (como getters, setters, construtores, etc.).

1. Entidade: RegistroCaixa

A "entidade" é a representação da nossa tabela do banco de dados em formato de uma classe Java. Ela terá exatamente os atributos que você listou.

1. Dentro do seu projeto, navegue até a pasta `src/main/java/br/com/gestaofinanceira/sistema_gestao`.
2. Crie um novo pacote (package) chamado `caixa`.
3. Dentro do pacote `caixa`, crie um novo arquivo de Enum chamado `TipoMovimentacao.java`. Este arquivo garantirá que o tipo só possa ser `ENTRADA` ou `SAIDA`.

```
// Arquivo: TipoMovimentacao.java
package br.com.gestaofinanceira.sistema_gestao.caixa;

public enum TipoMovimentacao {
    ENTRADA,
    SAIDA
}
```

4. Agora, no mesmo pacote `caixa`, crie a classe principal da nossa entidade, chamada `RegistroCaixa.java`.

```
RegistroCaixa.java x RegistroCaixaController.java RegistroCaixaRepository.java RegistroCaixaService.java application.properties
1 package br.com.gestaofinanceira.sistema_gestao.caixa;
2
3 import java.math.BigDecimal;
4 import java.time.LocalDateTime;
5 import java.time.LocalDate;
6
7 import jakarta.persistence.Entity;
8 import jakarta.persistence.EnumType;
9 import jakarta.persistence.Enumerated;
10 import jakarta.persistence.GeneratedValue;
11 import jakarta.persistence.GenerationType;
12 import jakarta.persistence.Id;
13 import lombok.Data;
14
15 @Data // Anotação do Lombok: cria getters, setters, toString, etc. automaticamente. 11 usages & Lucas
16 @Entity(name = "registros_caixa") // Anotação do JPA: indica que esta classe é uma tabela no banco de dados.
17 public class RegistroCaixa {
18
19     @Id // Indica que este atributo é a chave primária da tabela.
20     @GeneratedValue(strategy = GenerationType.IDENTITY) // O banco de dados irá gerar o valor do ID automaticamente.
21     private Long id;
22
23     @Enumerated(EnumType.STRING) // Diz ao JPA para salvar o texto ("ENTRADA" ou "SAIDA") no banco.
24     private TipoMovimentacao tipo;
25
26     private String descricao;
27
28     private BigDecimal valor; // Dica de Mestre: Sempre use BigDecimal para valores financeiros!
29
30     private LocalDate data;
31
32     private LocalDateTime hora;
33 }
```

Analisando o código:

- **@Data:** Mágica do Lombok. Evita que a gente tenha que escrever getId(), setId(), getDescricao(), setDescricao(), etc.
- **@Entity:** Transforma nossa classe Java simples em uma entidade que o JPA (Spring Data JPA) pode gerenciar e salvar no banco de dados.
- **@Id e @GeneratedValue:** Configuram o id como chave primária auto-incrementável, um padrão em 99% dos casos.
- **BigDecimal:** Usamos BigDecimal para dinheiro porque tipos como double ou float podem ter problemas de arredondamento que são inaceitáveis em sistemas financeiros.

Dado que temos a "forma" dos nossos dados (a entidade RegistroCaixa), precisamos de um mecanismo para de fato salvar, buscar e manipular esses dados no banco de dados. É aqui que entra o **Repositório**.

Pense no repositório como o gerente de dados da sua entidade. Ele é a ponte direta e especializada entre seu código Java e a tabela no banco de dados. A melhor parte? Com o Spring Data JPA, 90% do trabalho já vem pronto.

2. Criando o Repositório (Repository)

Objetivo: Criar uma interface que nos dará, magicamente, todos os métodos básicos para interagir com a tabela registros_caixa.

1. **Onde criar?** Dentro do mesmo pacote onde você criou suas outras classes: `src/main/java/br/com/gestaofinanceira/sistema_gestao/caixa`.
2. **Como criar?** Crie um novo arquivo, mas desta vez, selecione **interface**, e não class. Dê a ele o nome de `RegistroCaixaRepository.java`.
3. **O Código:**



```
1 package br.com.gestaofinanceira.sistema_gestao.caixa;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface RegistroCaixaRepository extends JpaRepository<RegistroCaixa, Long> {
6 }
7
```

Analisando o código (A Mágica):

- `public interface RegistroCaixaRepository`: Estamos definindo uma interface, um "contrato", e não uma classe com lógica.
- `extends JpaRepository<RegistroCaixa, Long>`: Esta é a linha mais importante.
 - Ao estender `JpaRepository`, estamos dizendo ao Spring: "Por favor, crie uma implementação para mim que saiba como gerenciar a entidade `RegistroCaixa`".
 - O primeiro parâmetro, `RegistroCaixa`, diz **qual entidade** este repositório vai gerenciar.
 - O segundo parâmetro, `Long`, diz qual é o **tipo da chave primária (@Id)** daquela entidade.

Só por ter criado essa interface, o Spring nos presenteia com vários métodos prontos para usar, como:

- save(registro): Salva um novo registro ou atualiza um existente.
- findById(id): Busca um registro pelo seu ID.
- findAll(): Busca TODOS os registros da tabela.
- deleteById(id): Exclui um registro pelo seu ID.
- E muitos outros!

3. Configurando o Banco de Dados em Memória (H2)

Já adicionamos a dependência do H2, mas precisamos dizer ao Spring como usá-lo e habilitar um console no navegador para a gente poder visualizar os dados.

1. Vá até a pasta src/main/resources.
2. Abra o arquivo application.properties.
3. Adicione o seguinte conteúdo a ele:



```

1  # Configuracao do Banco de Dados H2 em Memoria
2  spring.datasource.url=jdbc:h2:mem:gestadb
3  spring.datasource.driverClassName=org.h2.Driver
4  spring.datasource.username=sa
5  spring.datasource.password=
6
7  # Configuracao do JPA (Hibernate)
8  spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
9  spring.jpa.hibernate.ddl-auto=update
10
11 # Mostra no console as queries SQL que o JPA esta executando
12 spring.jpa.show-sql=true
13
14 # Habilitando o Console do H2
15 spring.h2.console.enabled=true
16 spring.h2.console.path=/h2-console
  
```

Com isso, toda vez que você iniciar sua aplicação, o Spring irá:

1. Criar um banco de dados H2 em memória.
2. Ler sua classe @Entity (RegistroCaixa) e criar a tabela registros_caixa automaticamente.
3. Permitir que você acesse e visualize esse banco de dados pelo navegador.

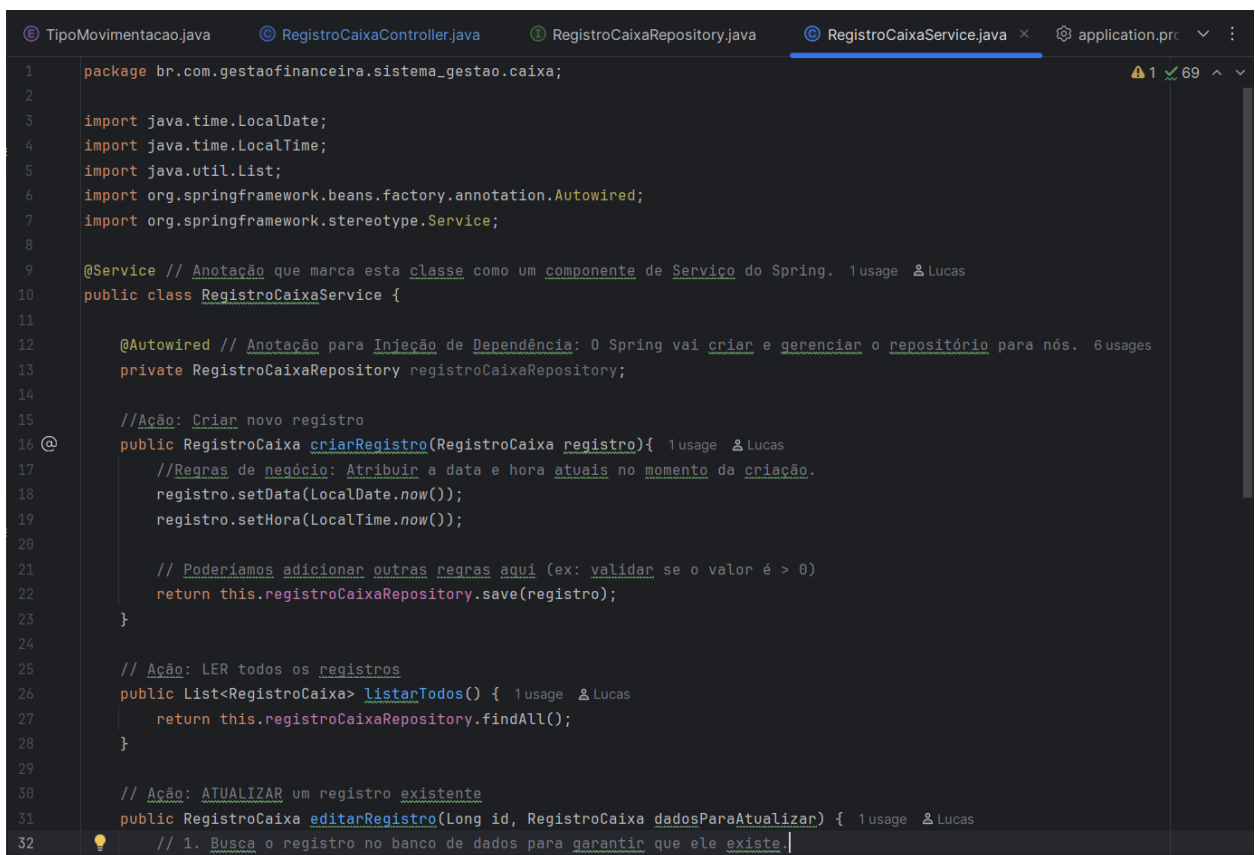
Por que a Camada de Serviço é importante? É aqui que mora a **lógica de negócio** da sua aplicação. Regras como "não é permitido registrar um valor negativo" ou "toda descrição deve

ter no mínimo 3 caracteres" são implementadas aqui. Ela orquestra as chamadas ao repositório para cumprir uma tarefa de negócio.

4: Criando a Camada de Serviço (Service)

Objetivo: Criar uma classe que centralizará nossas operações de negócio para o caixa.

1. **Onde criar?** Ainda dentro do pacote
src/main/java/br/com/gestaofinanceira/sistema_gestao/caixa.
2. **Como criar?** Crie uma nova **class** chamada RegistroCaixaService.java.
3. **O Código:**



```
1 package br.com.gestaofinanceira.sistema_gestao.caixa;
2
3 import java.time.LocalDate;
4 import java.time.LocalDateTime;
5 import java.util.List;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8
9 @Service // Anotação que marca esta classe como um componente de Serviço do Spring. 1 usage 2 Lucas
10 public class RegistroCaixaService {
11
12     @Autowired // Anotação para Injeção de Dependência: O Spring vai criar e gerenciar o repositório para nós. 6 usages
13     private RegistroCaixaRepository registroCaixaRepository;
14
15     // Ação: Criar novo registro
16     @ public RegistroCaixa criarRegistro(RegistroCaixa registro){ 1 usage 2 Lucas
17         // Regras de negócio: Atribuir a data e hora atuais no momento da criação.
18         registro.setData(LocalDate.now());
19         registro.setHora(LocalTime.now());
20
21         // Poderíamos adicionar outras regras aqui (ex: validar se o valor é > 0)
22         return this.registroCaixaRepository.save(registro);
23     }
24
25     // Ação: LER todos os registros
26     public List<RegistroCaixa> listarTodos() { 1 usage 2 Lucas
27         return this.registroCaixaRepository.findAll();
28     }
29
30     // Ação: ATUALIZAR um registro existente
31     public RegistroCaixa editarRegistro(Long id, RegistroCaixa dadosParaAtualizar) { 1 usage 2 Lucas
32         // 1. Busca o registro no banco de dados para garantir que ele existe |
```

```

10 public class RegistroCaixaService {
29
30     // Ação: ATUALIZAR um registro existente
31     public RegistroCaixa editarRegistro(Long id, RegistroCaixa dadosParaAtualizar) { 1 usage 2 Lucas
32         // 1. Busca o registro no banco de dados para garantir que ele existe.
33         return this.registroCaixaRepository.findById(id)
34             .map( RegistroCaixa registroExistente -> {
35                 // 2. Atualiza os campos do registro existente com os novos dados.
36                 registroExistente.setTipo(dadosParaAtualizar.getTipo());
37                 registroExistente.setDescricao(dadosParaAtualizar.getDescricao());
38                 registroExistente.setValor(dadosParaAtualizar.getValor());
39                 // Não atualizamos data e hora, pois geralmente são do momento da criação.
40
41                 // 3. Salva o registro atualizado de volta no banco.
42                 return this.registroCaixaRepository.save(registroExistente);
43             }).orElseThrow(() -> new RuntimeException("Registro não encontrado com o id: " + id)); // Lança um erro se o ID n
44     }
45
46     // Ação: EXCLUIR um registro
47     public void excluirRegistro(Long id) { 1 usage 2 Lucas
48         // 1. Verifica se o registro existe antes de tentar deletar.
49         if (!this.registroCaixaRepository.existsById(id)) {
50             throw new RuntimeException("Registro não encontrado com o id: " + id);
51         }
52         // 2. Deleta o registro.
53         this.registroCaixaRepository.deleteById(id);
54     }
55
56 }

```

5: Criando o Controller (Controller)

Objetivo: Expor nossa lógica de negócio para o mundo através de uma **API REST**. Uma API REST é um padrão de comunicação web que usa os métodos HTTP padrão (GET, POST, PUT, DELETE) para realizar as quatro operações do CRUD que construímos.

- POST será usado para **Criar**.
 - GET será usado para **Ler**.
 - PUT será usado para **Atualizar/Editar**.
 - DELETE será usado para **Excluir**.
1. **Onde criar?** No mesmo pacote de sempre:
src/main/java/br/com/gestaofinanceira/sistema_gestao/caixa.
 2. **Como criar?** Crie uma nova **class** chamada RegistroCaixaController.java.
 3. **O Código:**

```
SistemaGestaoApplication.java  pom.xml (sistema-gestao)  TipoMovimentacao.java  RegistroCaixaController.java  RegistroCaixaReposito
1 // Arquivo: RegistroCaixaController.java
2 package br.com.gestaofinanceira.sistema_gestao.caixa;
3
4 import java.util.List;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.ResponseEntity;
7 import org.springframework.web.bind.annotation.DeleteMapping;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.PathVariable;
10 import org.springframework.web.bind.annotation.PostMapping;
11 import org.springframework.web.bind.annotation.PutMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RestController;
15
16 @RestController // Anotação que combina @Controller e @ResponseBody. Diz ao Spring que esta classe irá lidar com requisições no usages
17 // web e retornar dados (JSON).
18 @RequestMapping("/caixa") // Define que todos os endpoints nesta classe começarão com "/caixa". Ex:
19 // http://localhost:8080/caixa
20 public class RegistroCaixaController {
21
22     @Autowired // O Spring injeta a instância do nosso Service. O Controller DELEGA a lógica para o Service. 4 usages
23     private RegistroCaixaService registroCaixaService;
24
25     // Endpoint para CRIAR um novo registro
26     // Mapeia requisições HTTP POST para /caixa
27     @PostMapping("/") no usages Lucas
28     public ResponseEntity<RegistroCaixa> criar(@RequestBody RegistroCaixa novoRegistro) {
29         var registroSalvo = this.registroCaixaService.criarRegistro(novoRegistro);
30         return ResponseEntity.ok().body(registroSalvo);
31     }
32
33     // Endpoint para LER todos os registros
34     // Mapeia requisições HTTP GET para /caixa
35     @GetMapping("/") no usages Lucas
36     public ResponseEntity<List<RegistroCaixa>> listar() {
37         var listaDeRegistros = this.registroCaixaService.listarTodos();
38         return ResponseEntity.ok().body(listaDeRegistros);
39     }
40
41     // Endpoint para EDITAR um registro
42     // Mapeia requisições HTTP PUT para /caixa/{id}
43     @PutMapping("/{id}") no usages Lucas
44     public ResponseEntity<RegistroCaixa> editar(@PathVariable Long id, @RequestBody RegistroCaixa dadosParaAtualizar) {
45         var registroAtualizado = this.registroCaixaService.editarRegistro(id, dadosParaAtualizar);
46         return ResponseEntity.ok().body(registroAtualizado);
47     }
48
49     // Endpoint para EXCLUIR um registro
50     // Mapeia requisições HTTP DELETE para /caixa/{id}
51     @DeleteMapping("/{id}") no usages Lucas
52     public ResponseEntity<?> excluir(@PathVariable Long id) {
53         this.registroCaixaService.excluirRegistro(id);
54         return ResponseEntity.ok().build(); // Retorna uma resposta de sucesso sem corpo.
55     }
56 }
```

Analizando as anotações do Controller:

- **@RestController:** Uma anotação fundamental do Spring Web. Ela transforma nossa classe em um "endpoint" web, fazendo com que os retornos dos métodos sejam convertidos para o formato JSON automaticamente.

- **@RequestMapping("/caixa")**: Mapeia um endereço base para todos os métodos dentro desta classe. Fica muito mais organizado.
- **@PostMapping, @GetMapping, @PutMapping, @DeleteMapping**: Cada uma dessas anotações mapeia um método da classe a um verbo HTTP específico, definindo a ação que o endpoint irá realizar.
- **@RequestBody**: Diz ao Spring para pegar o corpo (body) da requisição (que virá em JSON) e converter para um objeto Java RegistroCaixa. Mágica pura!
- **@PathVariable**: Pega um valor diretamente da URL (como o id em /caixa/1) e o transforma em uma variável para ser usada no método.
- **ResponseEntity**: É uma classe do Spring que representa toda a resposta HTTP. Usá-la nos dá mais controle, permitindo definir o status da resposta (ex: 200 OK) e o corpo da resposta.

6: Rodando e Testando a Aplicação

Objetivo: Iniciar o servidor Spring Boot e usar uma ferramenta de cliente HTTP para interagir com a nossa API, criando, lendo, editando e excluindo registros no caixa.

Parte 1: Rodando a Aplicação

1. Na sua IDE (IntelliJ ou VS Code), navegue no explorador de arquivos até a classe principal da sua aplicação:
src/main/java/br/com/gestaofinanceira/sistema_gestao/SistemaGestaoApplication.java.
2. Esta classe terá um método main, que é o ponto de partida de qualquer aplicação Java.
3. Clique com o botão direito do mouse dentro do código desta classe e procure pela opção **"Run 'SistemaGestaoApplication.main()'"** ou um ícone de "play" verde ao lado da declaração do método main.
4. Clique para rodar.

Se tudo der certo, o console da sua IDE começará a exibir vários logs de inicialização do Spring. A última linha deve ser algo parecido com:

... Tomcat started on port(s): 8080 (http) with context path "

... Started SistemaGestaoApplication in X.XXX seconds

Isso significa que seu servidor está no ar e pronto para receber requisições na porta 8080!

Parte 2: Ferramentas de Teste

Para testar nossa API, precisamos de um programa que "finja" ser um frontend. As ferramentas mais populares para isso são o **Postman** e o **Insomnia**. Se você não tiver um deles, baixe e instale um (são gratuitos).

Vou usar os termos do Postman/Insomnia, mas eles são muito parecidos.

Parte 3: Testando os Endpoints

Vamos testar as 4 operações que criamos.

1. CRIAR um registro (POST)

- **Método HTTP:** POST
- **URL:** `http://localhost:8080/caixa/`
- Vá para a aba **Body** (Corpo), selecione a opção **raw** e o formato **JSON**.
- Cole o seguinte JSON no corpo da requisição. Este é o dado do nosso primeiro registro:

```
{  
  "tipo": "ENTRADA",  
  "descricao": "Salário do mês",  
  "valor": 5000.00  
}
```

- Clique em **Send** (Enviar).

Resultado esperado: Você deve receber uma resposta com **Status 200 OK** e o corpo da resposta será o registro que você acabou de criar, mas agora com id, data e hora preenchidos pelo sistema!

2. LISTAR todos os registros (GET)

- **Método HTTP:** GET
- **URL:** `http://localhost:8080/caixa/`
- Não precisa de Body.

- Clique em **Send**.

Resultado esperado: Status 200 OK e o corpo da resposta será uma lista (um array JSON) com os dois registros que você criou.

3. EDITAR um registro (PUT)

Vamos supor que o valor do aluguel estava errado. Vamos corrigir o registro com id 2.

- **Método HTTP:** PUT
- **URL:** `http://localhost:8080/caixa/2` (note o /2 no final para indicar qual registro queremos editar).
- **Body (JSON):**

```
{  
  "tipo": "SAIDA",  
  "descricao": "Aluguel e condomínio",  
  "valor": 1650.00  
}
```

- Clique em **Send**.

Resultado esperado: Status 200 OK e o corpo da resposta mostrará o registro de id 2 com os dados atualizados. Se você fizer o GET novamente, verá a alteração refletida lá.

4. EXCLUIR um registro (DELETE)

Vamos excluir o registro do salário (id 1).

- **Método HTTP:** DELETE
- **URL:** `http://localhost:8080/caixa/1`
- Não precisa de Body.
- Clique em **Send**.

Resultado esperado: Status 200 OK sem nenhum corpo na resposta. Se você fizer o GET novamente, verá que agora só existe o registro do aluguel.

Extra: Visualizando o Banco de Dados

Para provar que tudo está realmente funcionando, você pode ver a tabela de dados diretamente no banco H2.

1. Abra seu navegador de internet.
2. Acesse a URL: `http://localhost:8080/h2-console`
3. Na tela de login, certifique-se que o campo **JDBC URL** está exatamente como configuramos no `application.properties`: `jdbc:h2:mem:gestaodb`.
4. Clique em **Connect**.
5. Você verá sua tabela `REGISTROS_CAIXA` na lista à esquerda. Clique nela e depois em **Run** para ver os dados.

Módulo 2. Algumas ideias poderiam ser:

- Adicionar validações (ex: não permitir descrição vazia).
- Criar um módulo de categorias para os lançamentos.
- Implementar um módulo de usuários com login e senha.
- Adicionar filtros na busca (ex: buscar registros por data ou por tipo).

