📚 Guia Didático de Desenvolvimento de Features - Sistema Financeiro

Este guia explica, de forma prática e detalhada, como criar uma feature no sistema financeiro, usando como exemplo a implementação de transações. Siga este roteiro para entender o processo e aplicar em qualquer nova funcionalidade.

1. Como pensar para iniciar uma nova feature?

Sempre comece pelo domínio! Pergunte-se:

- Qual problema de negócio quero resolver?
- Quais entidades e regras existem?
- Quais dados são essenciais?

No caso de transações:

- O que é uma transação? Uma movimentação financeira (entrada ou saída de dinheiro).
- Quais campos são obrigatórios? Valor, data, descrição, categoria, usuário.
- Quais regras? Valor positivo, data obrigatória, categoria válida.

2. Passo a passo da implementação

Passo 1: Modelagem da Entidade de Domínio

Crie a classe principal do negócio em `domain/entities/Transacao.java`:

```java
@Entity
@Table(name = "transacoes")
public class Transacao {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
    private BigDecimal valor;
    private LocalDate data;
    private String descricao;
    @ManyToOne
    private Categoria categoria;
    @ManyToOne
    private Usuario usuario;

    public void validarValor() {
        if (valor.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Valor deve ser positivo");
        }
    }
}
```

Por que começar aqui?

- Define o modelo central do negócio
- Regras de negócio ficam encapsuladas
- Não depende de tecnologia

Passo 2: Interface do Repositório

Crie a interface em `application/repositories/TransacaoRepository.java` :

```java
public interface TransacaoRepository {
    Transacao save(Transacao transacao);
    Optional<Transacao> findById(UUID id);
    List<Transacao> findByUsuario(Usuario usuario);
    List<Transacao> findByPeriodo(LocalDate inicio, LocalDate fim);
    void delete(Transacao transacao);
}
```

Por que interface?

- Facilita testes
- Permite trocar implementação
- Segue o princípio da inversão de dependência

Passo 3: Serviço de Aplicação (Casos de Uso)

Implemente a lógica de negócio em `application/services/TransacaoService.java` :

```
@Service
public class TransacaoService {
    private final TransacaoRepository transacaoRepository;
    private final CategoriaRepository categoriaRepository;
    private final UsuarioRepository usuarioRepository;

    public Transacao criarTransacao(UUID usuarioId, UUID categoriaId, BigDecimal valor,
        Usuario usuario = usuarioRepository.findById(usuarioId).orElseThrow();
        Categoria categoria = categoriaRepository.findById(categoriaId).orElseThrow();
        Transacao transacao = new Transacao();
        transacao.setUsuario(usuario);
        transacao.setCategoria(categoria);
        transacao.setValor(valor);
        transacao.setData(data);
        transacao.setDescricao(descricao);
        transacao.validarValor();
        return transacaoRepository.save(transacao);
    }

    public BigDecimal calcularSaldo(UUID usuarioId) {
        List<Transacao> transacoes = transacaoRepository.findByUsuario(usuarioId);
        BigDecimal receitas = transacoes.stream().filter(t -> t.getCategoria().getTipo(
        BigDecimal despesas = transacoes.stream().filter(t -> t.getCategoria().getTipo(
        return receitas.subtract(despesas);
    }
}
```

Por que aqui?

- Centraliza regras de negócio

- Orquestra entidades e repositórios
- Facilita testes e manutenção

Passo 4: Implementação do Repositório (Infraestrutura)

Implemente a persistência em `infrastructure/persistence/JpaTransacaoRepository.java`:

```
@Repository
public interface SpringDataTransacaoRepository extends JpaRepository<Transacao, UUID> {
    List<Transacao> findByUsuarioId(UUID usuarioId);
    @Query("SELECT t FROM Transacao t WHERE t.data BETWEEN :inicio AND :fim")
    List<Transacao> findByPeriodo(LocalDate inicio, LocalDate fim);
}

@Repository
public class TransacaoRepositoryImpl implements TransacaoRepository {
    private final SpringDataTransacaoRepository jpaRepository;
    // ...implementação dos métodos...
}
```

Por que separar?

- Mantém o domínio desacoplado da tecnologia
- Facilita migração futura

Passo 5: DTOs (Data Transfer Objects)

Crie DTOs em `presentation/dto/TransacaoRequest.java` e `TransacaoResponse.java`:

```java
public class TransacaoRequest {
    @NotNull
    @Positive
    private BigDecimal valor;
    @NotNull
    private LocalDate data;
    private String descricao;
    @NotNull
    private UUID categoriaId;
}

public class TransacaoResponse {
    private UUID id;
    private BigDecimal valor;
    private LocalDate data;
    private String descricao;
    private CategoriaResponse categoria;
    private String tipo;
}
```

Por que DTO?

- Não expõe entidades diretamente
- Centraliza validações de entrada
- Facilita evolução da API

Passo 6: Controller REST

Implemente o endpoint em `presentation/controllers/TransacaoController.java` :

```java
@RestController
@RequestMapping("/api/transacoes")
public class TransacaoController {
    private final TransacaoService transacaoService;

    @PostMapping
    public ResponseEntity<TransacaoResponse> criar(@Valid @RequestBody TransacaoRequest
        Transacao transacao = transacaoService.criarTransacao(usuario.getId(), request.
        return ResponseEntity.status(HttpStatus.CREATED).body(TransacaoResponse.from(tr
    }

    @GetMapping("/saldo")
    public ResponseEntity<BigDecimal> saldo(@AuthenticationPrincipal Usuario usuario) {
        BigDecimal saldo = transacaoService.calcularSaldo(usuario.getId());
        return ResponseEntity.ok(saldo);
    }
}
```

Boas práticas:

- Versionamento da API
- Documentação Swagger
- Tratamento de erros
- Segurança com autenticação

3. Ordem recomendada para criar uma feature

1. Entidade de domínio (domain/entities)
2. Interface do repositório (application/repositories)
3. Serviço/casos de uso (application/services)
4. Implementação do repositório (infrastructure/persistence)
5. DTOs (presentation/dto)
6. Controller REST (presentation/controllers)

4. Decisões de design importantes

- **UUID para IDs:** Segurança e escalabilidade
- **BigDecimal para valores:** Precisão monetária
- **Hard Delete para transações:** LGPD, direito ao esquecimento
- **Soft Delete para categorias:** Preserva histórico
- **Validação em múltiplas camadas:** DTO, Service, Entity

5. Checklist para novas features

- ☐ Modelei a entidade de domínio?
- ☐ Defini a interface do repositório?
- ☐ Implementei os casos de uso no service?
- ☐ Criei a implementação do repositório?
- ☐ Criei DTOs de request/response?
- ☐ Implementei o controller REST?
- ☐ Adicionei validações?
- ☐ Documentei com Swagger?
- ☐ Escrevi testes unitários?
- ☐ Tratei exceções adequadamente?

6. Dicas finais para desenvolvimento

- Comece sempre pelo domínio
- Use interfaces para abstrações
- Valide em múltiplas camadas
- Documente com Swagger
- Pense em casos extremos (null, vazio, limites)
- Escreva testes primeiro
- Mantenha métodos pequenos e nomes descritivos

7. Notas pessoais de desenvolvimento

Este espaço é para suas anotações pessoais, insights e aprendizados durante o desenvolvimento.