Revisão

- JCP / JPA / Hibernate
- Spring Data JPA / Hibernate
  - Mapeamento Objeto Relacional
  - Independência de Banco de Dados
- Repositórios
  - Interfaces CrudRepository, PagingAndSortingRepository e JpaRepository
- Configuração
  - Dependencias
    - spring-boot-starter-data-jpa
    - Banco de dados (driver H2, Posgresql, etc…)
  - Arquivo de propriedades
    - Datasource url, username e password
    - DDL - Data definition Language
    - Show-sql
    - H2 console
- Entidades
  - @Entity, @Table, @Column, @Id, @GeneratedValue
  - Validações: @NotNull, @Size, @Future, @Past
  - Tratamento de Erro

Quem criou as tabelas????

Duas propriedades referentes a DDL (Data Definition Language do SQL - CREATE, DROP, ALTER …)

- spring.jpa.generate-ddl - indica se o spring deve ser responsável pela ddl (true ou false)
- spring.jpa.hibernate.ddl-auto - como o hibernate irá se comportar:
  - none: não realiza nenhuma operação
  - validate: valida a estrutura do banco com as entidades mapeadas, se houver diferenças
  - update: atualiza a estrutura do banco de acordo com as entidades (não exclui colunas, por segurança)
  - create: recria a estrutura do banco sempre
  - create-drop: cria a estrutura e a apaga ao final da sessão

O Spring tem um comportamento padrão diferente de acordo com o tipo de banco de dados.

- Bancos embedes (H2, HSQLDB e Derby) considera um ambiente de dev e usa create-drop por padrão
- Para outros bancos ele considera none

*Recomendação para ambiente de trabalho: deixar opções em false e none e, ou usar um script manual de banco de dados ou utilizar frameworks de gerenciamento de versão de banco, como o **liquibase** ou o **flyway***

Outras propriedades

- **spring.datasource.driverClassName** - classe do driver jdbc (org.h2.Driver ou org.postgres.Driver) - não é necessária pois o hibernate consegue identificar qual driver a partir da url, mas para alguns bancos ou situações específicas pode ser necessário configurar
- **spring.jpa.database-platform** - indica a classe do hibernate responsável por tratar o “dialeto” sql do banco
  - org.hibernate.dialect.PostgreSQLDialect
  - org.hibernate.dialect.H2Dialect
- **spring.jpa.properties.hibernate.format_sql** - se ao exibir o sql no console/log, ele deve estar formatado (identado) - (true ou false)

Validação

Incluir dependência do spring-boot-starter-validation no pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Incluir anotações de validação na entidade e no controller

```java
@Entity
@Table(name = "todo")
public class Todo {

    @Id
    private Integer id;

    @NotNull
    @Size(min = 2, max = 20)
    @Column(name = "titulo", nullable = false, length = 20)
    private String titulo;
```

```java
@GetMapping
public List<Todo> getAll() {
    return todoService.getAll();
}
@PostMapping
public Todo postTodo(@Valid @RequestBody Todo todo) {
    return todoService.addTodo(todo);
}
```

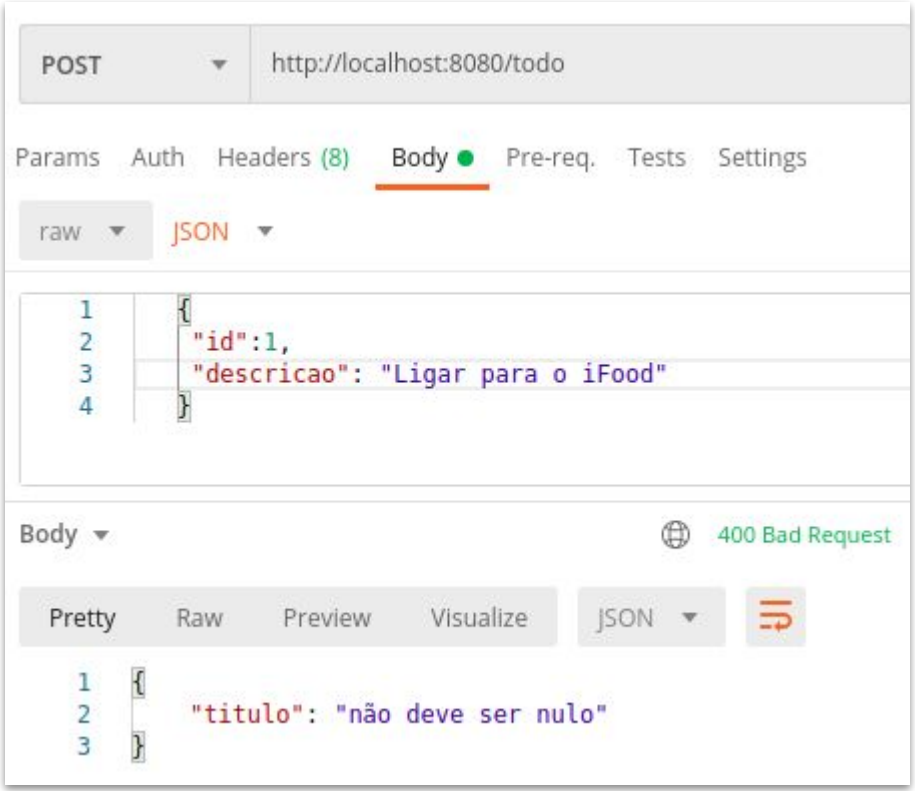
Validação

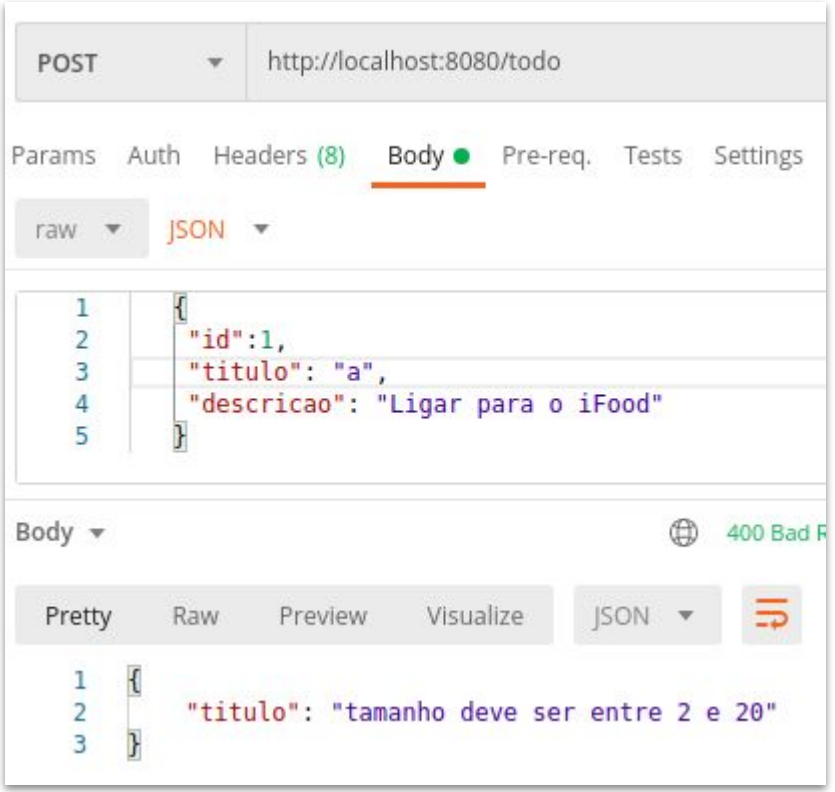
Cria o método “handler” para tratar a exceção de argumento inválido (@Valid força a validação)

```java
@RestControllerAdvice
public class ExceptionsController {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
        //cria um map para tratar os erros
        Map<String, String> errosOcorridos= new HashMap<>();
        //recupera a lista de erros
        List<ObjectError> erros = ex.getBindingResult().getAllErrors();
        for(ObjectError erro:erros) { //para cada erro
            //pega o atributo onde ocorreu o erro
            String atributo = ((FieldError) erro).getField();
            //pega a mensagem de erro
            String mensagem= erro.getDefaultMessage();
            //adiciona no map
            errosOcorridos.put(atributo, mensagem);
        }
        //retorna o map
        return errosOcorridos;
    }
```

Validação - Teste





Validação - Anotações

- Max - valor máximo inteiro (int)
- Min - valor mínimo inteiro (int)
- DecimalMax - Valor máximo numérico (BigDecimal)
  - `@DecimalMax(value = "100000.0", inclusive = true)`
  - `@DecimalMax("1.99")`
- DecimalMin - Valor mínimo numérico (BigDecimal)
  - `@DecimalMin(value = "0.0", inclusive = false)`
- Digits - numero de digitos permitidos, na parte inteira e na decimal (BigDecimal)
  - `@Digits(integer=3, faction=2)`
- Future - valida se a data é no futuro (Date)
- Past - valida se a data é no passado (Date)
- NotNull - não pode ser nulo
- Null - deve ser nulo
- Pattern - deve ser validado por expressão regular
  - `@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}")`
- Size - define os limites aceitos, (String ou Coleções)
  - `@Size(min=2, max=10)`

DTO

- ***Data Transfer Object*** (DTO) ou simplesmente ***Transfer Object*** é um padrão de projetos bastante usado em Java para o transporte de dados entre diferentes componentes de um sistema

- A ideia consiste basicamente em agrupar um conjunto de atributos numa classe simples de forma a otimizar a comunicação.

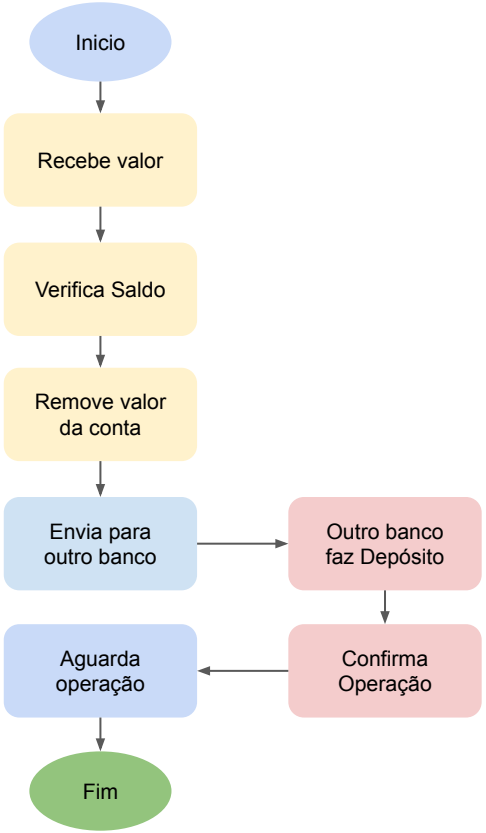
- Numa chamada remota, seria ineficiente passar cada atributo individualmente. Da mesma forma seria ineficiente ou até causaria erros passar uma entidade mais complexa.

- Além disso, muitas vezes os dados usados na comunicação não refletem exatamente os atributos do seu modelo. Então, um DTO seria uma classe que provê exatamente aquilo que é necessário para um determinado processo.

Transações

Realizar uma transferência bancária:

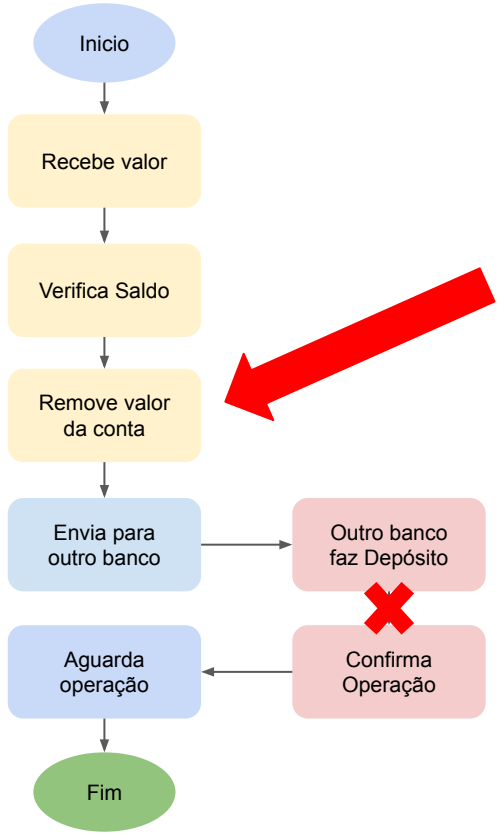
- Recebe o valor da transferência
- Verifica se há saldo na conta
- Remove o valor da conta
- Envia para o outro banco o valor
  - Faz comunicação com o banco
  - Aguarda a confirmação do outro banco
- Encerra a operação



Transações

Realizar uma transferência bancária:

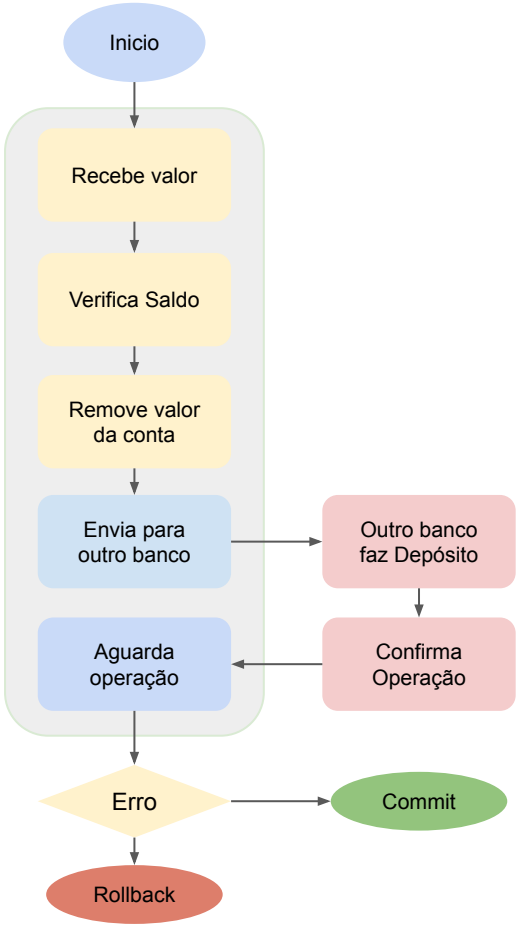
- Recebe o valor da transferência
- Verifica se há saldo na conta
- Remove o valor da conta
- Envia para o outro banco o valor
  - Faz comunicação com o banco
  - Aguarda a confirmação do outro banco
- Encerra a operação



Transações

Realizar uma transferência bancária:

- Tentar (Begin Transaction - SQL)
  - Recebe o valor da transferência
  - Verifica se há saldo na conta
  - Remove o valor da conta
  - Envia para o outro banco o valor
    - Faz comunicação com o banco
    - Aguarda a confirmação do outro banco
  - Encerra a operação
- Se não houve erro
  - Confirma (Commit - SQL)
- Se houver erro
  - Reverte (Rollback - SQL)



Transações Spring Boot

Anotação @Transacional indica que todas as operações do método serão executadas dentro de uma transação do banco

```
@Transactional
public void crudAgainstDatabase() {
    readFromDatabase();
    writeToDatabase();
}
```

IMPORTANTE: Existem duas anotações @Transactional, uma do Spring e outra do Java, utilizar a do Spring, cujo pacote e classe é:
`org.springframework.transaction.annotation.Transactional`

Mais informações https://hellokoding.com/spring-boot/transactional/

Consultas

Nos Repositórios (que são interfaces) é possível criar consultas de duas maneiras

- Derivação de nome de método

```
List<Livro> findByAutor(String autor);
```

- Anotação @Query

```
@Query("select * from Autor a where a.ativo=true")
List<Autor> buscaAutorAtivo()
```

https://www.baeldung.com/spring-data-jpa-query

Consultas - Derivacao

Nome do Método	SQL gerado
List<User> findByName(String name)	select * from user where name = :name
List<User> findByNameIsNot(String name)	select * from user where not name = :name
List<User> findByNameIsNull()	select * from user where name is null
List<User> findByActiveTrue()	select * from user where active = true
List<User> findByNameStartingWith(String prefix)	select * from user where name like :prefix + “%”
List<User> findByAgeLessThan(Integer age)	select * from user where age < :age
List<User> findByNameOrderByName(String name)	select * from user where name = :name order by name

Mais informações em https://www.baeldung.com/spring-data-derived-queries

Consultas - JPQL

JPQL - JPA Query Language
Derivada da SQL mas baseada nos objetos mapeados. Utiliza a anotação @Query em métodos da interface do Repositório

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();
```

Observação: o ***User*** na query acima se refere a classe User (entidade) e não a tabela user do sql.
O JPQL utiliza sempre o nome da classe e os atributos para definir a query.

JPQL - Ordenação

Basta adicionar um parâmetro do tipo Sort ao final do método

- Repositório

```
@Query("select u from User u where u.active=true")
List<User> listActiveUsers(Sort sort)
```

- Service

```
userRepository.listActiveUsers(Sort.by("name"))
```

JPQL - Parâmetros

Para passar parâmetros para a query, deve-se utilizar a anotação @Param antes do parâmetro no método e utilizar o mesmo nome na query, precedido de :

```
@Query("SELECT u FROM User u WHERE u.status = :status)
User findUserByStatusNamedParams(@Param("status") Integer status);
```

Podem até ser coleções para a cláusula IN do SQL

```
@Query(value = "SELECT u FROM User u WHERE u.name IN :names")
List<User> findUserByNameList(@Param("names") Collection<String> names);
```

JPQL - Nativas

Em alguns casos é possível utilizar querys utilizando o SQL nativo do banco de dados utilizado

```
@Query(value="SELECT * FROM USERS u WHERE u.status = 1", nativeQuery=true)
Collection<User> findAllActiveUsersNative();
```

Mais informações em https://thorben-janssen.com/spring-data-jpa-query-annotation/ e https://www.baeldung.com/spring-data-jpa-query e https://www.baeldung.com/spring-data-jpa-pagination-sorting e https://www.baeldung.com/spring-data-derived-queries

Exercício

Criar uma aplicação para gerenciar uma biblioteca de livros armazenando no banco de dados

- Entidade Livro
  - Id - identificador (número, sequence, chave primária)
  - titulo - titulo do livro (texto, obrigatório, minimo 5 e máximo 30 caracteres)
  - tipo - tipo do livro (fantasia, técnico, romance) (texto, obrigatório, minimo 3 máximo 20)
  - autor - nome do autor do livro (texto, obrigatorio, minimo 10, maximo 40)
  - data de publicação - data de publicação do livro (deve ser uma data anterior a data atual)
- API
  - Todas as operações CRUD no tópico /livro
- Bonus
  - No endpoint GET /livro possibilidade de passar um parâmetro ?ordem=campo para que a listagem venha ordenada por aquele campo

Pesquisar:

- Como passar a data em json para o controller (qual o formato)?
- Como ordenar no findAll do repositório