Desenvolvimento para Servidores-II

Bean Validation e Lombok

Neste tópico abordaremos o uso do framework Bean Validation para validação sintática na API REST e do Lombok.

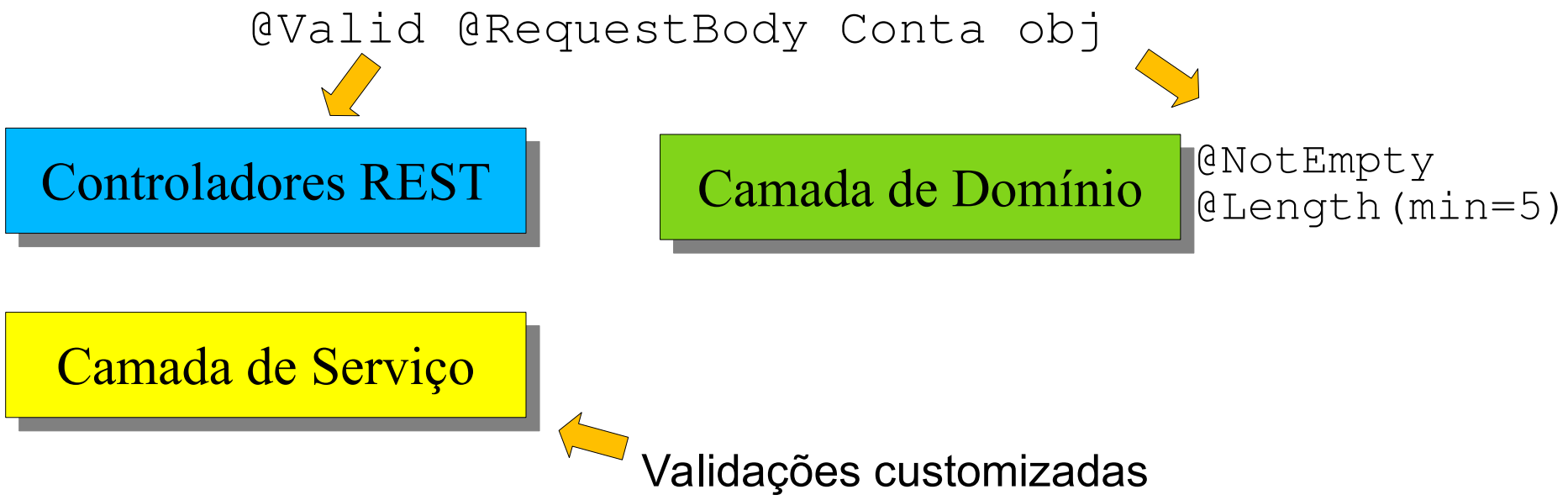
Prof. Ciro Cirne Trindade

Bean Validation

- Um *framework* de validação que permite especificar restrições de validação através de **anotações**
- Tipos de validação
  - Sem acesso a dados
    - Validação sintática: campos não nulos, valores máximos e mínimos, comprimento máximo e mínimo de strings, somente números, padrões
    - Outras: data no futuro ou passado
  - Com acesso a dados
    - CPF não pode ser repetido

Bean Validation

- As validações devem ser feitas tanto no *front-end* como no *back-end*
- Onde colocar as validações na API REST?



Bean Validation

- Implementação de referência
  - Hibernate Validator

Dependência

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

DTO (*Data Transfer Object*)

- Objeto que transmite informação entre processos
- O *pattern* DTO pode ser usado de várias formas diferentes
  - A abordagem mais comum é usá-lo para transferência de dados entre diferentes camadas de aplicações multicamadas
- Vamos usá-lo para separar a lógica de validação dos objetos transmitidos via HTTP da lógica dos objetos persistidos no banco de dados

CategoriaDTO

```
public class CategoriaDTO {
    private Long id;
    @NotBlank
    @Length(min = 4, max = 50)
    private String nome;

    public CategoriaDTO() { }
    // getters e setters
}
```

```
@Entity
@Table(name = "tb_categoria")
public class Categoria extends AbstractEntity {
        private static final long serialVersionUID = 1L;

        @Column(name = "nm_categoria", length = 50)
        private String nome;

        public Categoria() { }

         // getters/setters
}
```

Necessário fazer o mapeamento da entidade no DTO e vice-versa

- Classe CategoriaMapper.java (1/2)

```
@Component
public class CategoriaMapper {
    public CategoriaMapper() { }

    public Categoria toEntity(CategoriaDTO categoriaDTO) {
        Categoria categoria = new Categoria();
        categoria.setId(categoriaDTO.getId());
        categoria.setNome(categoriaDTO.getNome());
        return categoria;
    }

    public CategoriaDTO toDTO(Categoria categoria) {
        CategoriaDTO categoriaDTO = new CategoriaDTO();
        categoriaDTO.setId(categoria.getId());
        categoriaDTO.setNome(categoria.getNome());
        return categoriaDTO;
    }
```

Existe um framework chamado MapStruct que gera *mappers*

Necessário fazer o mapeamento da entidade no DTO e vice-versa

- Classe CategoriaMapper.java (2/2)

```
    public List<CategoriaDTO> toDTO(List<Categoria> categorias) {
        List<CategoriaDTO> categoriasDTO = new ArrayList<>();
        for (Categoria c : categorias) {
            categoriasDTO.add(toDTO(c));
        }
        return categoriasDTO;
    }

    public List<Categoria> toEntity(List<CategoriaDTO> categoriasDTO)
    {
        List<Categoria> categorias = new ArrayList<>();
        for (CategoriaDTO c : categoriasDTO) {
            categorias.add(toEntity(c));
        }
        return categorias;
    }
}
```

CategoriaService (1/2)

```
@Service
public class CategoriaService implements
                                    ServiceInterface<CategoriaDTO>{

    @Autowired
    private CategoriaRepository repository;

    @Autowired
    private CategoriaMapper mapper;

    public CategoriaService() { }

    public CategoriaDTO create(CategoriaDTO obj) {
        Categoria categoria = repository.save(mapper.toEntity(obj));
        return mapper.toDTO(categoria);
    }

    public List<CategoriaDTO> findAll() {
        return mapper.toDTO(repository.findAll());
    }
```

CategoriaService (2/2)

```
    public CategoriaDTO findById(Long id) {
        Optional<Categoria> obj = repository.findById(id);
        if (obj.isPresent())
            return mapper.toDTO(obj.get());
        return null;
    }

    public boolean update(CategoriaDTO categoriaDTO) {
        if (repository.existsById(categoriaDTO.getId())) {
            repository.save(mapper.toEntity(categoriaDTO));
            return true;
        }
        return false;
    }

    public boolean delete(Long id) {
        if (repository.existsById(id)) {
            repository.deleteById(id);
            return true;
        }
        return false;
    }
}
```

CategoriaController (1/2)

```
@RestController
@RequestMapping("/categorias")
public class CategoriaController implements
                                        ControllerInterface<CategoriaDTO>{

    @Autowired
    private CategoriaService service;

    @Override
    @GetMapping
    public ResponseEntity<List<CategoriaDTO>> getAll() {
        return ResponseEntity.ok(service.findAll());
    }

    @Override
    @GetMapping("/{id}")
    public ResponseEntity<?> get(@PathVariable("id") Long id) {
        CategoriaDTO obj = service.findById(id);
        if (obj != null) {
            return ResponseEntity.ok(obj);
        }
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }
```

CategoriaController (2/2)

```
    @Override
    @PostMapping
    public ResponseEntity<CategoriaDTO> post(@Valid
              @RequestBody CategoriaDTO obj) throws URISyntaxException {
        CategoriaDTO categoriaDTO = service.create(obj);
        URI location = ServletUriComponentsBuilder.fromCurrentRequest()
           .path("/{id}").buildAndExpand(categoriaDTO.getId()).toUri();
        return ResponseEntity.created(location).body(categoriaDTO);
    }

    @Override
    @PutMapping
    public ResponseEntity<?> put(@Valid @RequestBody CategoriaDTO obj) {
        if (service.update(obj))
            return ResponseEntity.ok(obj);
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }

    @Override
    @DeleteMapping("/{id}")
    public ResponseEntity<?> delete(@PathVariable("id") Long id) {
        if (service.delete(id))
            return ResponseEntity.ok().build();
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
    }
}
```

Validando valores numéricos mínimos e máximos

- As anotações `@Min` e `@Max` permitem validar valores mínimos e máximos de atributos numéricos
- Classe MovimentacaoDTO.java

```
@Min(1)
@Max(100000)
private Float valor;
```

Validando datas

- As anotações `@Past` e `@Future` validam se uma data está no passado e no futuro, respectivamente
- Exemplo:

```
@Past
private Date dataDeNascimento;

@Future
private Date dataDeValidade;
```

Validando e-mail

- A anotação `@Email` valida se o valor de um atributo é um e-mail válido
- Exemplo:

```
@Email
private String email;
```

Validando CPF e CNPJ

- No pacote org.hibernate.validator.constraints.br há anotações para validar o CPF e CNPJ
- Classe PessoaFisicaDTO.java

```
@CPF
private String cpf;
```

- Classe PessoaJuridicaDTO.java

```
@CNPJ
private String cnpj;
```

Lombok

- Biblioteca java para evitar código *boilerplate*
  - *getters*, *setters*, construtores, *hasCode*, *equals*, *toString*, etc.
- 2 passos
  - Baixar o Lombok e configurar a IDE
  - Colocar a dependência do pom.xml

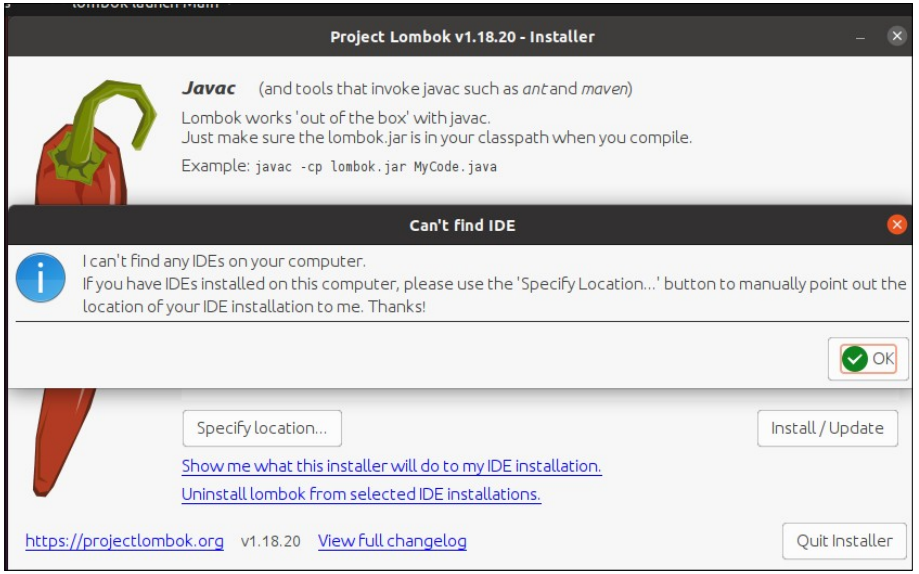
Baixar o Lombok e configurar a IDE

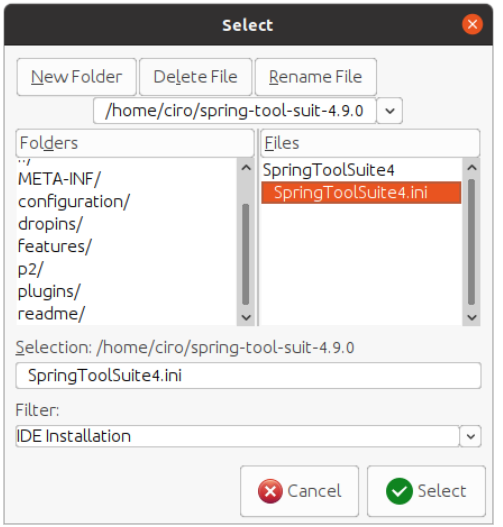
- Baixe o Lombok

- Dê um clique duplo no arquivo lombok.jar

Baixar o Lombok e configurar a IDE

- Selecione a IDE ou especifique



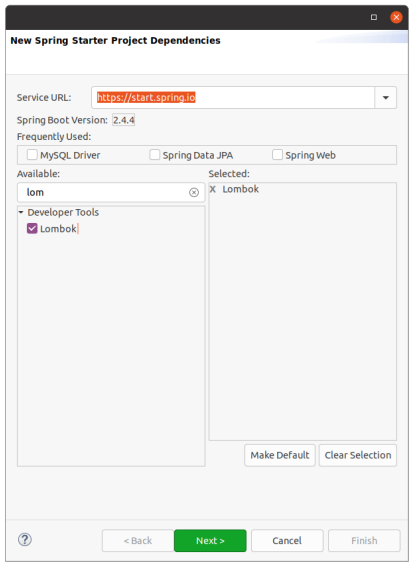


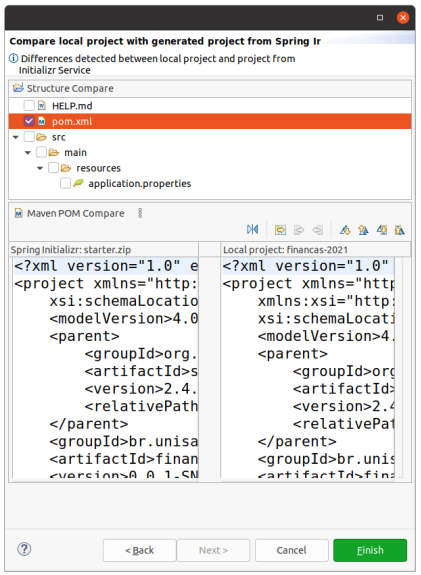
Colocar a depedência no pom.xml

- Dependência no pom.xml

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

- Via menu Spring → Add Starters





Getter e Setter

- Para gerar os método *getters* e *setters*, use as anotações `@Getter` e `@Setter`
- Podem ser usada sobre um atributo da classe ou sobre a classe inteira
  - Quando usado sobre um atributo, gera o *getter* e *setter* para esse atributo
  - Quando usado sobre a classe, gera os métodos *getters* e *setters* para **todos** os atributos da classe

Getter e Setter

- Nos atributos

```
public class Conta extends
                        AbstractEntity {
    @Getter @Setter
    @Column(name = "nr_agencia")
    private Integer agencia;

    @Getter @Setter
    @Column(name = "nm_numero",
           length = 10, unique = true)
    private String numero;

    @Getter @Setter
    @Column(name = "vl_saldo")
    private Float saldo;
}
```

- Na classe

```
@Getter
@Setter
public class Conta extends
                        AbstractEntity {

    @Column(name = "nr_agencia")
    private Integer agencia;

    @Column(name = "nm_numero",
           length = 10, unique = true)
    private String numero;

    @Column(name = "vl_saldo")
    private Float saldo;
}
```

Getter e Setter

- Por padrão os métodos *getters* e *setters* são públicos (`public`)
- Para mudar o nível de acesso, use o elemento `AccessLevel` das anotações
- Por exemplo, para mudar o nível de acesso para protegido
  - `@Getter(AccessLevel.PROTECDED)`
  - `@Setter(AccessLevel.PROTECDED)`

`AccessLevel.NONE` não gera o método *getter* ou *setter* para o atributo anotado

Getter e Setter

- Para anotar um método *getter* ou *setter*, use o elemento `onMethod = @__({@Anotacões})`
- Por exemplo:

```
@Getter(onMethod = @__(@JsonIgnore))
@Setter(onMethod = @__(@JsonProperty))
@OneToMany(cascade = CascadeType.ALL,
          orphanRemoval = true, mappedBy = "conta")
private List<Movimentacao> movimentacoes;
```

ToString

- Para gerar o método toString() para a classe, use a anotação `@ToString`
- Para excluir um atributo do método toString(), use a anotação `@ToString.Exclude`
- Exemplo:

```
@ToString
public class Conta extends AbstractEntity {
    ...
    @ToString.Exclude
    @Column(name = "vl_saldo")
    private Float saldo;
}
```

EqualsAndHashCode

- Para gerar os métodos equals() e hashCode(), use a anotação `@EqualsAndHashCode`
- Para definir esses métodos usando apenas atributos desejados, use o elemento `onlyExplicityIncluded=true` e anote esses atributos com `@EqualsAndeHashCode.Include`

Construtores

- Para gerar o construtor padrão, use a anotação `@NoArgsContructor`
- Para gerar o construtor com argumentos use a anotação `@AllArgsConstructor`
- Para gerar um construtor para os atributos anotados com `@NonNull` use a anotação `@RequiredArgsConstructor`

Data

- A anotação `@Data` é um atalho `@Getter`, `@Setter`, `@ToString`, `@EqualsAndHasCode` e `@RequiredArgsConstructor`

Builder

- A anotação `@Builder` gerar o padrão de projeto Builder para a classe
- Exemplo:

```
Conta c = Conta.builder()
                .agencia(133)
                .numero("4234-0")
                .saldo(2500.3f)
                .build();
```

Builder

- `@Builder` também pode ser usado sobre um construtor ou um método se você quiser gerar o métodos do builder para atributos selecionados

```
@Builder
public Conta(Long id, Integer agencia,
             String numero) {
    super(id);
    this.agencia = agencia;
    this.numero = numero;
}
```

Builder

- A anotação `@Singular` irá gerar 2 métodos para definir valores de atributos do tipo array ou Collection
  - Um que permite atribuir uma coleção de valores ao atributo
  - Outro que permite atribuir valores individuais ao atributo

Builder

- Exemplo do uso do `@Singular` na classe Movimentacao.java

Use o elemento `value` para definir o nome do método do builder

```
@Singular
private List<Categoria> categorias;
```

- Para adicionar uma categoria a movimentação

```
Movimentacao m = Movimentacao.builder()
    .categorias(List.of(new Categoria("Lazer"),
                new Categoria("Doméstica"))).build();
```

  - Ou

```
Movimentacao m = Movimentacao.builder()
    .categoria(new Categoria("Lazer")
    .categoria(new Categoria("Doméstica").build();
```

Referências

- ORACLE. *Introduction to Bean Validation*. Disponível em: https://docs.oracle.com/javaee/7/tutorial/bean-validation.htm
- JBOSS. Hibernate Validator. Disponível em: https://docs.jboss.org/hibernate/stable/validator/api/
- LOMBOK. Disponível em: https://projectlombok.org/