Guia completo para validação com Spring Boot

Bean Validation é o padrão de fato para implementar a lógica de validação no ecossistema Java. Está bem integrado com Spring e Spring Boot.

No entanto, existem algumas armadilhas. Este tutorial aborda todos os principais casos de uso de validação e exemplos de código de esportes para cada um.

Exemplo de Código

Este artigo é acompanhado por um exemplo de código funcional no GitHub .

Configurando a Validação

O suporte para validação de feijão do Spring Boot vem com o iniciador de validação, que podemos incluir em nosso projeto (notação Gradle):

```
implementation('org.springframework.boot:spring-boot-starter-validation')
```

Não é necessário adicionar o número da versão, pois o plugin Spring Dependency Management Gradle faz isso para nós. Se não estiver usando o plugin, você pode encontrar a versão mais recente aqui .

No entanto, se também incluímos o iniciador da web, o iniciador de validação é gratuito:

```
implementation('org.springframework.boot:spring-boot-starter-web')
```

Observe que o iniciador de validação não faz mais do que adicionar uma dependência a uma versão compatível do validador do **hibernate**, que é a implementação mais amplamente usada da especificação Bean Validation.

Bean Validation Basics

Basicamente, o Bean Validation funciona definindo restrições aos campos de uma classe, anotando-os com certas anotações .

Em seguida, você passa um objeto dessa classe para um validador que verifica se as restrições foram satisfeitas.

Veremos mais detalhes nos exemplos abaixo.

Validando a entrada para um controlador Spring MVC

Digamos que implementamos um controlador Spring REST e queremos validar a entrada que foi passada por um cliente. Existem três coisas que podemos validar para qualquer solicitação HTTP recebida:

- o corpo do pedido,
- variáveis dentro do caminho (por exemplo, `id` em `/foos/{id}`) e,
- parâmetros de consulta.

Vamos examinar cada um deles com mais detalhes.

Validando um corpo de solicitação

Em solicitações POST e PUT, é comum passar uma carga JSON dentro do corpo da solicitação. Spring mapeia automaticamente o JSON de entrada para um objeto Java. Agora, queremos verificar se o objeto Java de entrada atende aos nossos requisitos.

Esta é nossa classe de carga útil de entrada:

```
class Input {

  @Min(1)
  @Max(10)
  private int numberBetweenOneAndTen;

  @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
  private String ipAddress;

  // ...
}
```

Temos um `int` campo que deve ter um valor entre 1 e 10, inclusive, e um `String` campo que deve conter um endereço IP (a regex, na verdade, ainda permite endereços IP inválidos com octetos maiores que 255, mas vamos consertar isso posteriormente no tutorial)

Este é o controlador REST que pega um `Input` objeto no corpo da solicitação e o valida:

```
@RestController
class ValidateRequestBodyController {

 @PostMapping("/validateBody")
 ResponseEntity<String> validateBody(@Valid @RequestBody Input input) {
  return ResponseEntity.ok("valid");
 }

}
```

Simplesmente adicionamos a `@Valid` anotação ao `Input` parâmetro, que também é anotado com `@RequestBody` para marcar que deve ser lido no corpo da solicitação. Fazendo isso, estamos dizendo ao Spring para passar o objeto para a `Validator` antes de fazer qualquer outra coisa.

Use `@Valid` em tipos complexos

Se a `Input` classe contém um campo com outro tipo complexo que deve ser validado, esse campo também precisa ser anotado com `@Valid` .

Se a validação falhar, ele irá disparar um `MethodArgumentNotValidException` . Por padrão, o Spring irá traduzir essa exceção para um status HTTP 400 (Bad Request).

Podemos verificar esse comportamento com um teste de integração:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(controllers = ValidateRequestBodyController.class)
class ValidateRequestBodyControllerTest {

 @Autowired
 private MockMvc mvc;

 @Autowired
 private ObjectMapper objectMapper;

 @Test
 void whenInputIsInvalid_thenReturnsStatus400() throws Exception {
  Input input = invalidInput();
  String body = objectMapper.writeValueAsString(input);

  mvc.perform(post("/validateBody")
        .contentType("application/json")
        .content(body))
        .andExpect(status().isBadRequest());
 }
}
```

Você pode encontrar mais detalhes sobre o teste de controladores Spring MVC em meu artigo sobre a `@WebMvcTest` anotação .

Validando Variáveis de Caminho e Parâmetros de Solicitação

A validação de variáveis de caminho e parâmetros de solicitação funciona de maneira um pouco diferente.

Não estamos validando objetos Java complexos neste caso, uma vez que variáveis de caminho e parâmetros de solicitação são tipos primitivos como `int` ou seus objetos correspondentes como `Integer` ou `String`.

Em vez de anotar um campo de classe como acima, estamos adicionando uma anotação de restrição (neste caso `@Min`) diretamente ao parâmetro do método no controlador Spring:

```
@RestController
@Validated
class ValidateParametersController {

 @GetMapping("/validatePathVariable/{id}")
 ResponseEntity<String> validatePathVariable(
    @PathVariable("id") @Min(5) int id) {
  return ResponseEntity.ok("valid");
 }

 @GetMapping("/validateRequestParameter")
 ResponseEntity<String> validateRequestParameter(
    @RequestParam("param") @Min(5) int param) {
  return ResponseEntity.ok("valid");
 }
}
```

Observe que temos que adicionar a `@Validated` anotação do Spring ao controlador no nível da classe para dizer ao Spring para avaliar as anotações de restrição nos parâmetros do método.

A `@Validated` anotação só é avaliada em nível de classe neste caso, embora seja permitida para uso em métodos (aprenderemos por que é permitida em nível de método ao discutir grupos de validação mais tarde).

Em contraste com a validação do corpo da solicitação, uma validação com falha acionará um em `ConstraintViolationException` vez de um `MethodArgumentNotValidException`. O Spring não registra um manipulador de exceção padrão para esta exceção, portanto, por padrão, causará uma resposta com status HTTP 500 (Erro Interno do Servidor).

Se quisermos retornar um status HTTP 400 em vez disso (o que faz sentido, já que o cliente forneceu um parâmetro inválido, tornando-o uma solicitação incorreta), podemos adicionar um manipulador de exceção personalizado ao nosso controlador:

```
@RestController
@Validated
class ValidateParametersController {

 // request mapping method omitted

 @ExceptionHandler(ConstraintViolationException.class)
 @ResponseStatus(HttpStatus.BAD_REQUEST)
 ResponseEntity<String> handleConstraintViolationException(ConstraintViolationException e) {
  return new ResponseEntity<>("not valid due to validation error: " + e.getMessage(), HttpStatus.BAD_REQUEST);
 }

}
```

Posteriormente neste tutorial, veremos como retornar uma resposta de erro estruturada que contém detalhes sobre todas as validações com falha para o cliente inspecionar.

Podemos verificar o comportamento de validação com um teste de integração:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(controllers = ValidateParametersController.class)
class ValidateParametersControllerTest {

 @Autowired
 private MockMvc mvc;

 @Test
 void whenPathVariableIsInvalid_thenReturnsStatus400() throws Exception {
  mvc.perform(get("/validatePathVariable/3"))
        .andExpect(status().isBadRequest());
 }

 @Test
 void whenRequestParameterIsInvalid_thenReturnsStatus400() throws Exception {
  mvc.perform(get("/validateRequestParameter")
        .param("param", "3"))
        .andExpect(status().isBadRequest());
 }

}
```

Validando a entrada para um método de serviço Spring

Em vez de (ou adicionalmente) validar a entrada no nível do controlador, também podemos validar a entrada para qualquer componente do Spring. Para isso, usamos uma combinação das anotações `@Validated` e `@Valid`:

```
@Service
@Validated
class ValidatingService{

  void validateInput(@Valid Input input){
    // do something
  }

}
```

Novamente, a `@Validated` anotação só é avaliada no nível da classe, portanto, não a coloque em um método neste caso de uso.

Aqui está um teste para verificar o comportamento de validação:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ValidatingServiceTest {

 @Autowired
 private ValidatingService service;

 @Test
 void whenInputIsInvalid_thenThrowsException(){
  Input input = invalidInput();

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateInput(input);
  });
 }

}
```

Validando Entidades JPA

A última linha de defesa para validação é a camada de persistência. Por padrão, Spring Data usa o Hibernate por baixo, que oferece suporte à Validação de Bean fora da caixa.

A camada de persistência é o lugar certo para validação?

Normalmente não queremos fazer a validação tão tarde quanto na camada de persistência, porque isso significa que o código de negócios acima trabalhou com objetos potencialmente inválidos que podem levar a erros imprevistos. Mais sobre este tópico em meu artigo sobre anti-padrões de validação de feijão .

Digamos que você queira armazenar objetos de nossa `Input` classe no banco de dados. Primeiro, adicionamos a anotação JPA necessária `@Entity` e adicionamos um campo de ID:

```
@Entity
public class Input {

  @Id
  @GeneratedValue
  private Long id;

  @Min(1)
  @Max(10)
  private int numberBetweenOneAndTen;

  @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
  private String ipAddress;

  // ...

}
```

Em seguida, criamos um repositório Spring Data que nos fornece métodos para persistir e consultar os `Input` objetos:

```
public interface ValidatingRepository extends CrudRepository<Input, Long> {}
```

Por padrão, sempre que usarmos o repositório para armazenar um `Input` objeto cujas anotações de restrição sejam violadas, obteremos um, `ConstraintViolationException` como este teste de integração demonstra:

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
class ValidatingRepositoryTest {

  @Autowired
  private ValidatingRepository repository;

  @Autowired
  private EntityManager entityManager;

  @Test
  void whenInputIsInvalid_thenThrowsException() {
    Input input = invalidInput();

    assertThrows(ConstraintViolationException.class, () -> {
      repository.save(input);
      entityManager.flush();
    });
  }

}
```

Você pode encontrar mais detalhes sobre o teste de repositórios Spring Data em meu artigo sobre a `@DataJpaTest` anotação .

Observe que a Validação do Bean só é disparada pelo Hibernate depois que o Bean `EntityManager` é liberado. O Hibernate libera isso `EntityManager` automaticamente em certas circunstâncias, mas no caso do nosso teste de integração, temos que fazer isso manualmente.

Se por algum motivo quisermos desabilitar a Validação do Bean em nossos repositórios Spring Data, podemos definir a propriedade Spring Boot `spring.jpa.properties.javax.persistence.validation.mode` como `none`.

Implementando um validador personalizado

Se as anotações de restrição disponíveis não forem suficientes para nossos casos de uso, talvez queiramos criar uma.

Na `Input` classe acima, usamos uma expressão regular para validar que uma String é um endereço IP válido. No entanto, a expressão regular não está completa: ela permite octetos com valores maiores que 255 (ou seja, “111.111.111.333” seria considerado válido).

Vamos corrigir isso implementando um validador que implementa essa verificação em Java em vez de uma expressão regular (sim, eu sei que poderíamos apenas usar uma expressão regular mais complexa para obter o mesmo resultado, mas gostamos de implementar validações em Java, don não é?).

Primeiro, criamos a anotação de restrição personalizada `IpAddress`:

```
@Target({ FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = IpAddressValidator.class)
@Documented
public @interface IpAddress {

  String message() default "{IpAddress.invalid}";

  Class<?>[] groups() default { };

  Class<? extends Payload>[] payload() default { };

}
```

Uma anotação de restrição personalizada precisa de todos os seguintes:

- o parâmetro `message`, apontando para uma chave de propriedade em `ValidationMessages.properties`, que é usada para resolver uma mensagem em caso de violação,
- o parâmetro `groups`, permitindo definir em quais circunstâncias esta validação deve ser acionada (falaremos sobre grupos de validação mais tarde),
- o parâmetro `payload`, permitindo definir uma carga útil a ser passada com esta validação (uma vez que este é um recurso raramente usado, não o abordaremos neste tutorial), e

- uma `@Constraint` anotação apontando para uma implementação da `ConstraintValidator` interface.

A implementação do validador é semelhante a esta:

```
class IpAddressValidator implements ConstraintValidator<IpAddress, String> {

 @Override
 public boolean isValid(String value, ConstraintValidatorContext context) {
  Pattern pattern =
   Pattern.compile("^([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})$");
  Matcher matcher = pattern.matcher(value);
  try {
   if (!matcher.matches()) {
    return false;
   } else {
    for (int i = 1; i <= 4; i++) {
     int octet = Integer.valueOf(matcher.group(i));
     if (octet > 255) {
      return false;
     }
    }
    return true;
   }
  } catch (Exception e) {
   return false;
  }
 }
}
```

Agora podemos usar a `@IpAddress` anotação como qualquer outra anotação de restrição:

```
class InputWithCustomValidator {

 @IpAddress
 private String ipAddress;

 // ...

}
```

Validando de forma programática

Pode haver casos em que desejamos invocar a validação programaticamente, em vez de confiar no suporte integrado à Validação de Bean do Spring.

Nesse caso, **podemos apenas criar um `Validator` manualmente** e invocá-lo para acionar uma validação:

```
class ProgrammaticallyValidatingService {

  void validateInput(Input input) {
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();
    Set<ConstraintViolation<Input>> violations = validator.validate(input);
    if (!violations.isEmpty()) {
      throw new ConstraintViolationException(violations);
    }
  }

}
```

Isso não requer nenhum suporte Spring.

No entanto, Spring Boot nos fornece uma `Validator` instância pré-configurada . Podemos injetar essa instância em nosso serviço e usá-la em vez de criar uma manualmente:

```
@Service
class ProgrammaticallyValidatingService {

  private Validator validator;

  ProgrammaticallyValidatingService(Validator validator) {
    this.validator = validator;
  }

  void validateInputWithInjectedValidator(Input input) {
    Set<ConstraintViolation<Input>> violations = validator.validate(input);
    if (!violations.isEmpty()) {
      throw new ConstraintViolationException(violations);
    }
  }
}
```

Quando este serviço for instanciado pelo Spring, ele terá automaticamente uma `Validator` instância injetada no construtor.

O seguinte teste de unidade prova que os dois métodos acima funcionam conforme o esperado:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ProgrammaticallyValidatingServiceTest {

 @Autowired
 private ProgrammaticallyValidatingService service;

 @Test
 void whenInputIsInvalid_thenThrowsException(){
  Input input = invalidInput();

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateInput(input);
  });
 }

 @Test
 void givenInjectedValidator_whenInputIsInvalid_thenThrowsException(){
  Input input = invalidInput();

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateInputWithInjectedValidator(input);
  });
 }

}
```

Usando grupos de validação para validar objetos de maneira diferente para diferentes casos de uso

Freqüentemente, certos objetos são compartilhados entre diferentes casos de uso.

Vejamos as operações CRUD típicas, por exemplo: o caso de uso “Criar” e o caso de uso “Atualizar” muito provavelmente tomarão o mesmo tipo de objeto como entrada. No entanto, pode haver validações que devem ser acionadas em diferentes circunstâncias:

- apenas no caso de uso “Criar”,
- apenas no caso de uso “Atualizar”, ou
- em ambos os casos de uso.

O recurso Bean Validation que nos permite implementar regras de validação como essa é chamado de “Grupos de Validação” .

Já vimos que todas as anotações de restrição devem ter um `groups` campo. Isso pode ser usado para passar quaisquer classes em que cada uma defina um determinado grupo de validação que deve ser acionado.

Para nosso exemplo CRUD, simplesmente definimos duas interfaces de marcador `OnCreate` e `OnUpdate` :

```
interface OnCreate {}

interface OnUpdate {}
```

Podemos então usar essas interfaces de marcador com qualquer anotação de restrição como esta:

```
class InputWithGroups {

  @Null(groups = OnCreate.class)
  @NotNull(groups = OnUpdate.class)
  private Long id;

  // ...

}
```

Isso garantirá que o ID esteja vazio em nosso caso de uso “Criar” e que não esteja vazio em nosso caso de uso “Atualizar”.

Spring oferece suporte a grupos de validação com a `@Validated` anotação:

```
@Service
@Validated
class ValidatingServiceWithGroups {

   @Validated(OnCreate.class)
   void validateForCreate(@Valid InputWithGroups input){
     // do something
   }

   @Validated(OnUpdate.class)
   void validateForUpdate(@Valid InputWithGroups input){
     // do something
   }

}
```

Observe que a `@Validated` anotação deve ser aplicada novamente a toda a classe. Para definir qual grupo de validação deve estar ativo, ele também deve ser aplicado no nível do método.

Para ter certeza de que o acima funciona conforme o esperado, podemos implementar um teste de unidade:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ValidatingServiceWithGroupsTest {

 @Autowired
 private ValidatingServiceWithGroups service;

 @Test
 void whenInputIsInvalidForCreate_thenThrowsException() {
  InputWithGroups input = validInput();
  input.setId(42L);

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateForCreate(input);
  });
 }

 @Test
 void whenInputIsInvalidForUpdate_thenThrowsException() {
  InputWithGroups input = validInput();
  input.setId(null);

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateForUpdate(input);
  });
 }

}
```

Cuidado com os grupos de validação

Usar grupos de validação pode facilmente se tornar um antipadrão, já que estamos misturando questões. Com grupos de validação, a entidade validada deve conhecer as regras de validação para todos os casos de uso (grupos) em que é usada. Mais sobre este tópico em meu artigo sobre **antipadrões de Validação de Bean** .

Retorno de respostas de erro estruturadas

Quando uma validação falha, queremos retornar uma mensagem de erro significativa ao cliente. Para permitir que o cliente exiba uma mensagem de erro útil, **devemos retornar uma estrutura de dados que contenha uma mensagem de erro para cada validação que falhou** .

Primeiro, precisamos definir essa estrutura de dados. Vamos chamá-lo `ValidationErrorResponse` e ele contém uma lista de `Violation` objetos:

```
public class ValidationErrorResponse {

  private List<Violation> violations = new ArrayList<>();

  // ...
}

public class Violation {

  private final String fieldName;

  private final String message;

  // ...
}
```

Em seguida, criamos um global `ControllerAdvice` que lida com todas as `ConstraintViolationExceptions` bolhas até o nível do controlador. Para detectar erros de validação também para corpos de solicitação , também lidaremos com `MethodArgumentNotValidExceptions` :

```
@ControllerAdvice
class ErrorHandlingControllerAdvice {

  @ExceptionHandler(ConstraintViolationException.class)
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ResponseBody
  ValidationErrorResponse onConstraintValidationException(
      ConstraintViolationException e) {
    ValidationErrorResponse error = new ValidationErrorResponse();
    for (ConstraintViolation violation : e.getConstraintViolations()) {
      error.getViolations().add(
        new Violation(violation.getPropertyPath().toString(), violation.getMessage()));
    }
    return error;
  }

  @ExceptionHandler(MethodArgumentNotValidException.class)
  @ResponseStatus(HttpStatus.BAD_REQUEST)
  @ResponseBody
  ValidationErrorResponse onMethodArgumentNotValidException(
      MethodArgumentNotValidException e) {
    ValidationErrorResponse error = new ValidationErrorResponse();
    for (FieldError fieldError : e.getBindingResult().getFieldErrors()) {
      error.getViolations().add(
        new Violation(fieldError.getField(), fieldError.getDefaultMessage()));
    }
    return error;
  }

}
```

O que estamos fazendo aqui é simplesmente ler informações sobre as violações das exceções e traduzi-las em nossa `ValidationErrorResponse` estrutura de dados.

Observe a `@ControllerAdvice` anotação que torna os métodos de tratamento de exceções disponíveis globalmente para todos os controladores dentro do contexto do aplicativo.

Conclusão

Neste tutorial, examinamos todos os principais recursos de validação de que podemos precisar ao construir um aplicativo com Spring Boot.

Se você quiser sujar as mãos no código de exemplo, dê uma olhada no repositório github .

Histórico de atualização

- **25/10/2018:** adicionado uma palavra de cautela sobre o uso de validação de bean na camada de persistência (veja este tópico no Twitter).

Siga-me no Twitter para obter mais dicas sobre como se tornar um desenvolvedor de software melhor.

Tom Hombergs

Como engenheiro de software profissional, consultor, arquiteto e solucionador de problemas em geral, venho praticando a arte do software há mais de dez anos e ainda estou aprendendo algo novo a cada dia. Adoro compartilhar as coisas que aprendi, para que você (e eu no futuro) tenha uma vantagem inicial.