ChatGPT

Tipos de validação em requisições HTTP com Spring Boot 3.4.5 (Java 21)

Introdução

Quando uma API recebe dados de clientes ou de outras aplicações, é essencial garantir que esses dados estejam no formato esperado antes de prosseguir com o processamento. O **Jakarta Bean Validation** (JSR 380) e sua implementação de referência, o **Hibernate Validator**, são as bibliotecas mais comuns para essa finalidade em aplicações Spring Boot. A validação utiliza anotações que definem restrições diretamente nos campos dos objetos, reduzindo o código repetitivo e melhorando a manutenção da aplicação.

O Spring Boot 3.4.5 (compatível com Java 21) integra automaticamente o Hibernate Validator. Basta adicionar as anotações corretas aos modelos de dados (DTOs, VOs ou entidades) que o framework valida as entradas e retorna erros de forma padronizada. Além das restrições padrão da especificação, é possível criar **restrições personalizadas** para validar regras de negócio específicas (por exemplo, verificar se um valor pertence a um determinado enum ou validar CPF e CNPJ).

Teoria das validações e anotações

Restrições padrão do Bean Validation

A tabela a seguir resume algumas anotações importantes disponíveis em `jakarta.validation.constraints` e `org.hibernate.validator.constraints`. O Bean Validation especifica, por exemplo, restrições de formato, tamanho e intervalo numérico. Já o Hibernate Validator fornece restrições adicionais (como validação de CPF e CNPJ).

Anotação	Descrição/Exemplo (citações)
@NotNull, **@NotEmpty** e **@NotBlank**	Garantem que o valor não seja `null` ou vazio. A documentação do Bean Validation mostra que `@NotNull` indica que o valor de um campo não deve ser `null` [1]. `@NotEmpty` e `@NotBlank` (fornecidas pelo Hibernate Validator) garantem que Strings ou coleções não estejam vazias [2].
@Size(min, max)	Valida o tamanho de uma String, coleção ou array. A especificação esclarece que `@Size` avalia o tamanho da string ou coleção e permite especificar limites mínimo e máximo [3].
@Pattern(regexp)	Obriga que uma string siga uma expressão regular. A documentação mostra que o valor deve corresponder ao padrão definido no atributo `regexp` [4].

Anotação	Descrição/Exemplo (citações)
@Min / @Max e **@DecimalMin / @DecimalMax**	Impõem limites mínimo/máximo para números inteiros (`@Min` , `@Max`) ou decimais (`@DecimalMin` , `@DecimalMax`). O guia mostra que `@DecimalMin("0.0")` e `@DecimalMax("99999.99")` podem ser usados para validar um `BigDecimal` representando preço [5] .
@Digits(integer, fraction)	Garante que o valor numérico contenha um número máximo de dígitos inteiros e fracionários. O mesmo guia explica que `@Digits(integer=6,fraction=2)` limita a parte inteira a 6 dígitos e a parte decimal a 2 dígitos [6] .
@Positive / @PositiveOrZero / @Negative	Verificam se o número é positivo, positivo ou zero, ou negativo. A especificação mostra que `@Positive` exige um número positivo [7] .
@Past / @Future / @PastOrPresent / @FutureOrPresent	Valida se uma data está no passado ou futuro. O guia do Terasoluna indica que `@Future` e `@Past` são utilizados para datas futuras ou passadas [8] .
@DateTimeFormat (Spring)	Para strings com datas/horas, o Spring oferece `@DateTimeFormat` . O guia recomenda usar essa anotação em vez de validar datas com regex, pois é mais simples [9] .
@Email e **@URL**	Validam formatos de e-mail ou URL. O guia do Terasoluna lista `@Email` e `@URL` como restrições do Hibernate Validator para validar endereços RFC 2822 e URLs válidas [10] .
@CreditCardNumber	Verifica números de cartão de crédito pelo algoritmo de Luhn [11] .
@CPF e **@CNPJ**	Restrições específicas do Hibernate Validator para validar CPFs e CNPJs. O arquivo de mensagens padrão mostra as mensagens `invalid Brazilian corporate taxpayer registry number (CNPJ)` e `invalid Brazilian individual taxpayer registry number (CPF)` para as anotações `org.hibernate.validator.constraints.br.CNPJ` e `org.hibernate.validator.constraints.br.CPF` [12] , indicando que essas anotações verificam a validade de números de CNPJ e CPF.
@Valid	Aplicada em propriedades que são outros objetos; permite validar recursivamente os campos do objeto aninhado [13] .
@AssertTrue / @AssertFalse	Exigem que o valor de um campo booleano seja respectivamente `true` ou `false` [14] .

Observação sobre formato de data

Para validar o formato de datas enviadas como strings (por exemplo, "2025-07-31"), o Spring aconselha usar a anotação `@DateTimeFormat` . O guia explica que usar `@DateTimeFormat` é recomendado em vez de `@Pattern` , pois este último exige criar uma expressão regular complicada e não verifica a existência da data [9] . Com `@DateTimeFormat` , o Spring tenta converter a string para

`java.time.LocalDate` ou outro tipo data/hora; se o formato for inválido, uma mensagem de erro de `TypeMismatchException` será gerada.

Criação de validações personalizadas

Nem todas as regras de negócio podem ser implementadas com anotações padrão. Nestes casos, é possível criar **anotações personalizadas** combinando restrições existentes ou implementando um validador próprio. O guia detalha duas estratégias:

1. **Combinar regras existentes** – Quando uma mesma combinação de restrições é reutilizada em vários pontos (por exemplo, strings alfanuméricas ou IDs com tamanho específico), é útil criar uma nova anotação que reúna essas regras. O guia mostra um exemplo de `@AlphaNumeric`, que usa `@Pattern` e consolida as mensagens de erro [15], e de `@NotNegative`, baseado em `@Min` para impedir números negativos [16].
2. **Implementar um** `ConstraintValidator` – Para regras que não podem ser compostas com as restrições existentes (por exemplo, validar ISBN-13, CPF, CNPJ ou um valor pertencente a um enum), é necessário criar uma anotação e uma classe que implemente `ConstraintValidator`. O guia informa que qualquer regra pode ser criada implementando `javax.validation.ConstraintValidator` e vinculando essa implementação a uma anotação [17]. O exemplo apresentado implementa `ISBN13Validator` com método `isValid`, retornando `true` quando o valor está no formato correto [18].

Validação de valores de enumeração

Nas imagens fornecidas pelo usuário, existe um exemplo de anotação personalizada chamada `@Enumeration`. Essa anotação recebe uma classe de enum (`enumClass`) e utiliza um `ConstraintValidator` chamado `EnumerationValidator`. A lógica do validador converte as constantes do enum em uma lista de valores aceitos e, no método `isValid`, verifica se o valor recebido (convertido para upper case) está presente nessa lista. Esse padrão segue as recomendações do Bean Validation para criar restrições personalizadas: a anotação é marcada com `@Constraint(validatedBy = ...)` e o validador implementa `ConstraintValidator` [18].

Validação de CPF e CNPJ

O Hibernate Validator já oferece anotações para CPF e CNPJ (`@CPF` e `@CNPJ`). A lista de mensagens padrão mostra que essas anotações verificam se um número é um “Brazilian individual taxpayer registry number (CPF)” ou “Brazilian corporate taxpayer registry number (CNPJ)” válido [12]. Para utilizá-las, é necessário adicionar a dependência `hibernate-validator` e `hibernate-validator-annotation-processor` (ou `hibernate-validator-cdi`) no projeto Spring Boot e anotar o campo `String` correspondente com `@CPF` ou `@CNPJ`. É possível combinar `@NotBlank` para garantir que o valor não esteja vazio.

Implementação prática com Spring Boot 3.4.5

A seguir, apresentamos um exemplo simplificado de serviço REST em Spring Boot 3.4.5 (Java 21) que demonstra o uso das validações comentadas. O exemplo utiliza uma classe `SimulacaoVO` (VO de simulação) e uma classe aninhada `MultaVO`. Os campos são validados por meio de anotações padrão, restrições do Hibernate Validator e uma anotação personalizada para enumeração.

Dependências Maven (trecho)

Para usar as anotações do Bean Validation e as validações de CPF/CNPJ, inclua as seguintes dependências no `pom.xml` (Spring Boot já traz a maioria, mas é bom ilustrar):

```
<dependencies>
    <!-- Spring Boot starter web inclui o hibernate-validator -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Lombok para gerar getters/setters/constructores -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <!-- Dependência explícita do hibernate-validator para acesso a @CPF/
@CNPJ (opcional) -->
    <dependency>
        <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator</artifactId>
    </dependency>
</dependencies>
```

Anotação `@Enumeration`

```
package com.example.demo.validation;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * Valida se o valor de uma string pertence ao enum especificado.
 */
@Documented
@Constraint(validatedBy = { EnumerationValidator.class })
@Target({ FIELD, PARAMETER })
@Retention(RUNTIME)
public @interface Enumeration {
    Class<? extends Enum<?>> enumClass();
    String message() default "deve ser um valor do enum {enumClass}";
```

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

```
package com.example.demo.validation;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * Implementação do validador para a anotação {@link Enumeration}.
 */
public class EnumerationValidator implements
ConstraintValidator<Enumeration, CharSequence> {
    private List<String> acceptedValues;

    @Override
    public void initialize(Enumeration annotation) {
        // Coleta os nomes das constantes do enum e converte para maiúsculo
        acceptedValues = Stream.of(annotation.enumClass().getEnumConstants())
                               .map(Enum::name)
                               .map(String::toUpperCase)
                               .collect(Collectors.toList());
    }

    @Override
    public boolean isValid(CharSequence value, ConstraintValidatorContext
context) {
        if (value == null) {
            return true; // valor nulo é considerado válido; use @NotNull
para exigir valor
        }
        return acceptedValues.contains(value.toString().toUpperCase());
    }
}
```

Classe de enumeração `TipoServicoEnum`

```
package com.example.demo.enums;

import lombok.Getter;
import lombok.RequiredArgsConstructor;

@Getter
@RequiredArgsConstructor
public enum TipoServicoEnum {
```

```
    MULTA_RENAINF("MULTA_RENAINF", "Multa de trânsito do RENAINF"),
    LICENCIAMENTO("LICENCIAMENTO", "Licenciamento do veículo"),
    IPVA("IPVA", "Imposto sobre a Propriedade de Veículos Automotores"),
    MULTA_TRANSITO("MULTA_TRANSITO", "Multa de trânsito");

    private final String code;
    private final String description;
}
```

Modelo de dados (VO) com validações

```
package com.example.demo.vo;

import com.example.demo.enums.TipoServicoEnum;
import com.example.demo.validation.Enumeration;
import jakarta.validation.Valid;
import jakarta.validation.constraints.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.validator.constraints.br.CNPJ;
import org.hibernate.validator.constraints.br.CPF;
import org.springframework.format.annotation.DateTimeFormat;

import java.math.BigDecimal;
import java.time.LocalDate;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class SimulacaoVO {
    @NotBlank
    private String renavam;

    @NotBlank
    @Pattern(regexp = "[A-Z]{3}[0-9A-Z]{4}", message = "Placa deve estar no
formato ABC1D23")
    private String placa;

    // Validação de tipo de serviço usando a anotação personalizada
    @NotNull
    @Enumeration(enumClass = TipoServicoEnum.class)
    private String tipoServico;

    @NotBlank
    @Pattern(regexp = "[A-Z]{2}", message = "UF deve ter duas letras
maiúsculas")
    private String uf;

    // Valor total dos débitos, aceitando números positivos com 2 casas
```

```
decimais
    @NotNull
    @DecimalMin(value = "0.0", inclusive = true, message = "Valor deve ser
positivo")
    @Digits(integer = 10, fraction = 2, message = "Valor deve ter no máximo
10 dígitos inteiros e 2 decimais")
    private BigDecimal valorTotalDebitos;

    @Valid
    private MultaVO multa;

    // Campo opcional para CPF ou CNPJ do proprietário
    @CPF(message = "CPF inválido")
    private String cpf;

    @CNPJ(message = "CNPJ inválido")
    private String cnpj;

    // Data de emissão da simulação
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    @PastOrPresent(message = "Data de emissão deve ser no passado ou
presente")
    private LocalDate dataEmissao;

    @Data
    @NoArgsConstructor
    @AllArgsConstructor
    public static class MultaVO {
        @NotNull
        @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
        @Past(message = "Data da multa deve estar no passado")
        private LocalDate dataMulta;

        @NotBlank
        private String descricao;

        @NotNull
        @DecimalMin(value = "0.0", inclusive = true)
        @Digits(integer = 10, fraction = 2)
        private BigDecimal valor;
    }
}
```

Controlador REST

```
package com.example.demo.controller;

import com.example.demo.vo.SimulacaoVO;
import jakarta.validation.Valid;
import org.springframework.http.HttpStatus;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/simulacoes")
public class SimulacaoController {

    @PostMapping
    public ResponseEntity<String> criar(@Valid @RequestBody SimulacaoVO vo) {
        // Em uma aplicação real, persistiríamos a simulação aqui
        return ResponseEntity.status(HttpStatus.CREATED).body("Simulação
criada com sucesso");
    }
}
```

Explicação

- **BigDecimal / valores monetários** – os campos `valorTotalDebitos` e `valor` são validados com `@DecimalMin` e `@Digits`. O guia demonstra que `@DecimalMin("0.0")` e `@DecimalMax` podem ser usados para limitar o valor de um `BigDecimal` [5], e `@Digits` controla o número de dígitos inteiros e fracionários [6].
- **Data** – `dataEmissao` e `dataMulta` são mapeadas como `LocalDate` e anotadas com `@DateTimeFormat` (para converter a string da requisição) e `@Past` ou `@PastOrPresent`. O guia recomenda usar `@DateTimeFormat` em vez de regex para checar formato de data [9], e explica que `@Past` e `@PastOrPresent` verificam se a data está no passado ou no passado/presente [8].
- **Enumeração** – o campo `tipoServico` usa a anotação personalizada `@Enumeration`, que carrega os nomes constantes do enum `TipoServicoEnum` e verifica se o valor enviado é um desses nomes. A criação de anotações personalizadas segue as orientações de implementar `ConstraintValidator` e associar a anotação ao validador [18].
- **CPF e CNPJ** – os campos `cpf` e `cnpj` usam as anotações `@CPF` e `@CNPJ` do Hibernate Validator, cuja lista de mensagens indica que elas validam os números de CPF e CNPJ [12].
- **Placa e UF** – `placa` e `uf` usam `@Pattern` para validar o formato. O Bean Validation indica que `@Pattern` verifica se uma string coincide com a expressão regular fornecida [4].

Conclusão

O Bean Validation oferece um conjunto de anotações para validar rapidamente entradas de usuário e outras informações recebidas por um serviço REST. Ao combinar restrições padrão como `@NotNull`, `@Size`, `@Pattern`, `@Digits` e `@Past` com anotações do Hibernate Validator (`@Email`, `@URL`, `@CPF`, `@CNPJ`), grande parte das validações comuns pode ser declarada de forma simples. Para regras de negócio não cobertas pelas restrições existentes – como verificar se uma string corresponde a um determinado enum ou validar um formato específico – é possível criar anotações personalizadas implementando a interface `ConstraintValidator` [18]. Com essas práticas, as aplicações Spring Boot 3.4.5 em Java 21 permanecem seguras, legíveis e de fácil manutenção.

Considerações adicionais e boas práticas

Em projetos maiores é comum que as mesmas combinações de restrições e mensagens sejam repetidas em vários campos. Para evitar duplicação e centralizar as mensagens de erro, podemos:

1. **Centralizar mensagens em** `messages.properties` – Em vez de definir a string literal da mensagem no atributo `message` das anotações, informe a chave da mensagem. O Spring resolve essa chave no `messages.properties` (ou `ValidationMessages.properties`) utilizando `ResourceBundleMessageInterpolator`. Por exemplo:

```
@NotBlank(message = "{simulacao.renavam.obrigatorio}")
@Pattern(regexp = "[A-Z]{3}[0-9A-Z]{4}", message =
"{simulacao.placa.invalida}")
private String placa;
```

No arquivo de mensagens (por exemplo, `messages.properties`), defina:

```
simulacao.renavam.obrigatorio = O renavam é obrigatório.
simulacao.placa.invalida = Placa inválida: o formato deve ser ABC1D23.
```

Dessa forma, as mensagens ficam concentradas num único arquivo e podem ser traduzidas ou alteradas sem modificar o código.

1. **Criar anotações compostas com** `@ReportAsSingleViolation` – Para reutilizar várias restrições em um único campo, podemos criar uma anotação que combine as restrições e defina apenas uma mensagem. O exemplo a seguir cria uma anotação `@DataObrigatoria` que verifica se a data está no passado ou presente e converte o erro em uma única mensagem. A anotação é marcada com `@ReportAsSingleViolation` para que apenas a mensagem dessa anotação apareça em caso de erro:

```
package com.example.demo.validation;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import jakarta.validation.ReportAsSingleViolation;
import jakarta.validation.constraints.PastOrPresent;
import org.springframework.format.annotation.DateTimeFormat;
import java.lang.annotation.*;

@Documented
@Constraint(validatedBy = {})
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@ReportAsSingleViolation
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
@PastOrPresent
public @interface DataObrigatoria {
    String message() default "{data.obrigatoria}";
```

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

Em `messages.properties`, podemos parametrizar a mensagem para incluir o nome do campo usando variáveis de interpolação:

```
data.obrigatoria = {campo} inválida ou ausente.
```

Para que `{campo}` seja substituído dinamicamente pelo nome do campo, o validador pode personalizar a mensagem via `ConstraintValidatorContext`. Por exemplo:

```
public class DataObrigatoriaValidator implements
ConstraintValidator<DataObrigatoria, LocalDate> {
    @Override
    public boolean isValid(LocalDate value, ConstraintValidatorContext
context) {
        if (value == null || value.isAfter(LocalDate.now())) {
            // personaliza o atributo {campo}
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate(

context.getDefaultConstraintMessageTemplate().replace("{campo}", "data de
pagamento")
            ).addConstraintViolation();
            return false;
        }
        return true;
    }
}
```

Assim, ao aplicar `@DataObrigatoria(campo="data de pagamento")` em diferentes campos, a mensagem "data de pagamento inválida ou ausente" será mostrada conforme o campo.

1. **Utilizar constantes para chaves de mensagem** – Para evitar digitar manualmente as chaves das mensagens no código, é comum criar uma classe de constantes com os nomes das chaves. Exemplo:

```
public final class Mensagens {
    public static final String RENAVAM_OBRIGATORIO =
"simulacao.renavam.obrigatorio";
    public static final String PLACA_INVALIDA = "simulacao.placa.invalida";
    private Mensagens() {}
}
// Uso:
@NotBlank(message = "{" + Mensagens.RENAVAM_OBRIGATORIO + "}")
```

Essa prática reduz o risco de erros de digitação e facilita a localização das mensagens utilizadas.

Segurança e prevenção de injeções e XSS

Validações de formato e tipo não são suficientes para impedir ataques de injeção. Um atacante pode enviar **scripts** ou caracteres maliciosos em campos de texto livre, tentar **SQL Injection** ou explorar vulnerabilidades de **Cross-Site Scripting (XSS)** em aplicações web. Para mitigar esses riscos, é necessário aplicar estratégias complementares de validação, escape e codificação.

Permitir apenas o que é aceitável

O **OWASP Input Validation Cheat Sheet** alerta que filtrar entradas pela negação de padrões comuns (`denylist`) — por exemplo, rejeitar a string `1=1` ou a tag `<script>` — é falho, pois atacantes contornam facilmente essas verificações [19]. Em vez disso, recomenda-se a estratégia de **allowlist**: definir exatamente quais caracteres ou padrões são permitidos e rejeitar todo o resto [20]. Para campos estruturados (datas, CPFs, CNPJs, placas), isso já é feito com expressões regulares e anotações específicas. Para texto livre, é possível permitir letras, números e pontuações específicas com regex ou sanitizar o conteúdo.

Escape e codificação de saída

A prevenção de XSS baseia-se no conceito de **resistência perfeita à injeção**, que exige validar todas as variáveis e em seguida **escapá-las ou sanitizá-las** antes de enviá-las para o cliente [21]. O OWASP reforça que os frameworks modernos (React, Angular, Thymeleaf) oferecem codificação automática, mas não são perfeitos; por isso, usar bibliotecas de codificação e sanitização ajuda a preencher lacunas [22]. Por exemplo, ao escrever dados em um contexto HTML, as entidades especiais (`&`, `<`, `>`, `&quot;`, `&#x27;`) devem ser substituídas por seus equivalentes codificados (`&amp;`, `&lt;`, `&gt;`, `&quot;`, `&#x27;`) [23]. Em contextos de atributo, script, CSS ou URLs é preciso aplicar a codificação específica para cada tipo de conteúdo [24].

O artigo de Ben Hoyt alerta que **sanitizar entradas** retirando caracteres perigosos pode quebrar dados legítimos e gerar falsa sensação de segurança [25]. A abordagem correta é **armazenar a entrada original** e **escapar o valor na saída** conforme o contexto [26]. Por exemplo, nomes de usuário devem ser exibidos com HTML escaping, e consultas ao banco devem usar consultas parametrizadas em vez de concatenar strings [26].

Uso de APIs seguras

Injeções em consultas ocorrem quando dados não confiáveis são enviados diretamente para interpretadores (SQL, LDAP, XPath, SO). O OWASP Injection Prevention Cheat Sheet lista três regras principais: (1) realizar validação adequada, (2) utilizar APIs seguras com **consultas preparadas ou parametrizadas**, e (3) escapar os dados conforme o contexto [27]. Em aplicações Spring, utilizar Spring Data JPA, DynamoDB ou repositórios do Spring Cloud AWS garante que as consultas sejam preparadas. **Nunca concatene parâmetros diretamente em strings de consulta**.

Sanitização de HTML e strings

Caso a API permita texto rico ou HTML (por exemplo, uma descrição de multa), é necessário higienizar o HTML. Bibliotecas recomendadas incluem:

- **OWASP Java HTML Sanitizer** (`org.owasp.html`): permite definir uma *Safelist* e remove tags não permitidas.
- **jsoup** (`org.jsoup:jsoup`): fornece o método `Jsoup.clean(input, Safelist.basic())`, que retira tags perigosas e atributos maliciosos.
- **Encoder do OWASP** (`org.owasp.encoder:encoder`): oferece métodos como `Encode.forHtml()` e `Encode.forJava()` para codificar saídas em diferentes contextos.

Podemos criar uma anotação personalizada `@SafeHtml` que utiliza `jsoup` para limpar o conteúdo antes de persistir ou validar:

```
package com.example.demo.validation;

import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.*;

@Documented
@Constraint(validatedBy = SafeHtmlValidator.class)
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface SafeHtml {
    String message() default "{campo} contém conteúdo HTML perigoso";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

```
package com.example.demo.validation;

import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;
import org.jsoup.Jsoup;
import org.jsoup.safety.Safelist;

public class SafeHtmlValidator implements ConstraintValidator<SafeHtml,
String> {
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context)
{
        if (value == null) return true;
        // Limpa usando uma safelist básica (permite tags simples como <b>,
<i>, <p>)
        String sanitized = Jsoup.clean(value, Safelist.basic());
        // Se o conteúdo original difere do sanitizado, havia HTML perigoso
        boolean valido = sanitized.equals(value);
```

```
        return valido;
    }
}
```

No arquivo `messages.properties` podemos definir a mensagem usando a interpolação de campo, de modo que a mensagem final seja "Descrição contém conteúdo HTML perigoso".

Filtrando requisições globalmente

Para proteger a API contra XSS em todos os campos de entrada, podemos registrar um filtro que intercepta as requisições e sanitiza parâmetros e corpo JSON. Uma abordagem simples é usar o `OncePerRequestFilter` do Spring:

```
package com.example.demo.config;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.jsoup.Jsoup;
import org.jsoup.safety.Safelist;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import javax.servlet.http.HttpServletRequestWrapper;
import java.io.IOException;

@Component
public class XssFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        XssRequestWrapper sanitizedRequest = new XssRequestWrapper(request);
        filterChain.doFilter(sanitizedRequest, response);
    }

    // Wrapper que sanitiza parâmetros
    private static class XssRequestWrapper extends HttpServletRequestWrapper
{
        public XssRequestWrapper(HttpServletRequest request) {
            super(request);
        }
        @Override
        public String getParameter(String name) {
            String value = super.getParameter(name);
            return clean(value);
        }
        @Override
        public String[] getParameterValues(String name) {
```

```
            String[] values = super.getParameterValues(name);
            if (values == null) return null;
            String[] sanitized = new String[values.length];
            for (int i = 0; i < values.length; i++) {
                sanitized[i] = clean(values[i]);
            }
            return sanitized;
        }
        private String clean(String value) {
            return value == null ? null : Jsoup.clean(value,
Safelist.basic());
        }
    }
}
```

Essa abordagem garante que qualquer campo de entrada contendo scripts ou HTML perigoso seja sanitizado antes de chegar ao controlador. Combine com as validações específicas (regex, `@Pattern`, `@CPF/@CNPJ`) para reforçar a segurança.

Cabeçalhos e políticas de segurança

Além das validações, adicionar cabeçalhos HTTP de segurança ajuda a proteger as APIs. O Spring Security pode ser configurado para incluir automaticamente:

- **Content-Security-Policy (CSP)**: define políticas para scripts e conteúdos carregados pelo navegador.
- **X-Content-Type-Options: nosniff** e **X-Frame-Options**: reduzem ataques de clickjacking e mímica de tipo de conteúdo.
- **X-XSS-Protection**: ativa mecanismos de detecção de XSS no navegador (embora obsoleto em alguns navegadores).

A maioria desses cabeçalhos é configurada pelo Spring Security ao usar o `spring-boot-starter-security`, mas eles também podem ser customizados em uma classe de configuração `SecurityFilterChain`.

Resumo

Para tornar a API segura contra ataques de injeção e XSS:

1. **Valide todas as entradas** utilizando anotações de Bean Validation e estratégias de allowlist; não confie em filtrar padrões maliciosos [19].
2. **Use consultas preparadas** ou APIs seguras (Spring Data, Spring Cloud AWS) e evite concatenar parâmetros em strings de consulta [27].
3. **Escape e codifique as saídas** de acordo com o contexto; utilize bibliotecas de codificação para HTML, atributos, scripts, CSS e URLs [23] [24].
4. **Sanitize HTML ou texto rico** usando bibliotecas como jsoup ou OWASP Java HTML Sanitizer; crie anotações como `@SafeHtml` para reutilização.
5. **Adote filtros globais** para higienizar parâmetros de requisição e configurar cabeçalhos de segurança via Spring Security.

Assim, combinando validação, codificação, sanitização e APIs seguras, a aplicação Spring Boot reduz significativamente o risco de injeções e ataques XSS.

Proposta de VO genérico melhorado

Considerando as melhorias acima, podemos refatorar o `SimulacaoVO` para reduzir repetição e externalizar as mensagens. Criaremos anotações compostas para:

- `@Renavam` – combina `@NotBlank` e um `@Pattern` com regex de RENAVAM (11 dígitos) e usa uma mensagem unificada.
- `@Placa` – valida placas no padrão Mercosul ou antigo, reunindo as regras em uma anotação reutilizável.
- `@Uf` – exige dois caracteres maiúsculos.
- `@ValorMonetario` – combina `@DecimalMin` e `@Digits` para valores positivos com escala de duas casas decimais.
- `@DataObrigatoria` – já descrita acima.

Além disso, as mensagens são externalizadas em `messages.properties` e referenciadas por chave. Assim, a classe VO fica enxuta e genérica:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class SimulacaoVO {
    @Renavam(message = "{simulacao.renavam.obrigatorio}")
    private String renavam;

    @Placa(message = "{simulacao.placa.invalida}")
    private String placa;

    @NotNull(message = "{simulacao.tipoServico.obrigatorio}")
    @Enumeration(enumClass = TipoServicoEnum.class, message =
"{simulacao.tipoServico.invalido}")
    private String tipoServico;

    @Uf(message = "{simulacao.uf.invalida}")
    private String uf;

    @ValorMonetario(message = "{simulacao.debitos.invalido}")
    private BigDecimal valorTotalDebitos;

    @Valid
    private MultaVO multa;

    @CPF(message = "{simulacao.cpf.invalido}")
    private String cpf;

    @CNPJ(message = "{simulacao.cnpj.invalido}")
    private String cnpj;
```

```
    @DataObrigatoria(message = "{simulacao.dataEmissao.invalida}")
    private LocalDate dataEmissao;
    // ...
}
```

Desse modo, adiciona-se novas validações criadas uma vez e reaproveitáveis em todo o projeto. As mensagens são parametrizadas via arquivo `messages.properties` e podem ser personalizadas conforme o campo, como "data de pagamento inválida" ou "valor da multa inválido". Essa abordagem melhora a manutenibilidade e evita duplicação.

Fontes das anotações e dependências

Para implementar as validações em um projeto Spring Boot é importante saber de onde vêm cada anotação e qual dependência Maven a torna disponível:

Anotação	Pacote para importação	Observação/Deper
`@NotNull`, `@NotBlank`, `@Size`, `@Pattern`, `@DecimalMin`, `@Digits`, `@Positive`, `@Past`, `@PastOrPresent`	`jakarta.validation.constraints.*`	Disponíveis na esp Jakarta Bean Valida inclusas no `sprin starter-validat` (presente em `spr starter-web`).
`@DateTimeFormat`	`org.springframework.format.annotation.DateTimeFormat`	Faz parte do módu Spring Framework strings em tipos de
`@Email`, `@URL`, `@CreditCardNumber`, `@NotEmpty`, `@NotBlank` (versões da Hibernate)	`org.hibernate.validator.constraints.*`	Fornecidas pelo Hibernate Validato adicionar a depend `hibernate-vali`
`@CPF` e `@CNPJ`	`org.hibernate.validator.constraints.br.CPF` e `org.hibernate.validator.constraints.br.CNPJ`	Validador de CPF e numéricos. Necess `hibernate-vali` (versão ≥ 6) como
`@Enumeration`, `@DataObrigatoria`, `@Renavam`, `@Placa`, `@Uf`, `@ValorMonetario`	Pacote customizado do projeto, por exemplo `com.example.demo.validation`	Devem ser implem conforme mostrad relatório usando `@` e, quando aplicáve `@ReportAsSingl`
Lombok (`@Data`, `@NoArgsConstructor`, `@AllArgsConstructor`)	`lombok.*`	Requer a dependê Lombok no `pom.x` configuração do pl para gerar código compilação.

Dependências Maven adicionais

Para além do `spring-boot-starter-web`, inclua explicitamente o `hibernate-validator` (para garantir a presença das anotações de CPF/CNPJ e outras restrições da Hibernate) e o Lombok se utilizado:

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

Considerações sobre a validação de documentos brasileiros

As anotações `@CPF` e `@CNPJ` do Hibernate Validator validam os formatos numéricos atualmente utilizados pelo fisco brasileiro para pessoas físicas e jurídicas. Estas anotações verificam os dígitos de controle e a formatação (14 dígitos para CNPJ, 11 dígitos para CPF). Entretanto, notícias indicam que, a partir de julho de 2026, os novos CNPJs poderão conter letras, pois a Receita Federal planeja ampliar a combinação de caracteres. As versões atuais da biblioteca ainda não contemplam esse formato alfanumérico. Caso a legislação mude, há duas alternativas:

1. **Aguardar a atualização do Hibernate Validator** – A comunidade frequentemente atualiza as anotações de `org.hibernate.validator.constraints.br` conforme a Receita Federal publica as regras definitivas. Verifique a documentação da versão em uso.
2. **Criar uma validação customizada** – Para antecipar o suporte ao formato alfanumérico, você pode implementar uma anotação personalizada, como `@CNPJAlfanumerico`, que combine `@Pattern` para aceitar letras e números no padrão futuro e um validador que calcule os dígitos de controle (caso continuem exigidos). Essa anotação deve ser implementada com um `ConstraintValidator` semelhante ao exemplo de `ISBN13Validator` mencionado no guia [18]. O validador pode delegar a uma biblioteca de mercado (por exemplo, a biblioteca "cpf-cnpj-validator" disponível no Maven Central) ou implementar o algoritmo de cálculo.

Da mesma forma, caso exista a necessidade de validar **RENAVAM**, **placas** ou outros documentos veiculares de acordo com as normas mais recentes do DENATRAN, recomenda-se:

- Criar anotações específicas (`@Renavam` , `@Placa`) usando regex atualizados e/ou algoritmos de verificação de dígito. A anotações podem referenciar chaves de mensagem no arquivo de mensagens.
- Manter os padrões (regex) em constantes ou arquivos de configuração, facilitando futuras atualizações.
- Aplicar **princípios de boas práticas**, como nomes de classes/anotações claros (`@DocumentoNacional` , `@DocumentoVeicular`), imutabilidade dos objetos de valor e testes unitários cobrindo os formatos válidos e inválidos.

Anotações que suportam múltiplos formatos (legado e novo)

Quando um documento possui mais de um formato válido (por exemplo, CNPJ numérico antigo e CNPJ alfanumérico futuro), a anotação personalizada precisa aceitar ambas as variantes. Uma estratégia é criar um **validador que reconheça diferentes padrões**:

- Defina a anotação customizada sem se prender a um único formato. Exemplo: `@CNPJValido`.
- Implemente um `ConstraintValidator` que verifique se o valor atende a **um dos** formatos válidos. Por exemplo, verifique se o valor é numérico com 14 dígitos e possui dígitos verificadores corretos **ou** se o valor segue a estrutura alfanumérica do novo CNPJ e passa pelos algoritmos de validação que a Receita Federal vier a adotar.
- Inclua opções na anotação para ativar ou desativar formatos específicos (por exemplo, `boolean allowLegacy() default true; boolean allowAlphanumeric() default true;`). Isso permite que a mesma anotação seja reaproveitada em diferentes contextos.

O mesmo conceito se aplica a outras validações. Se placas de veículos possuem formato antigo (AAA-1111) e formato Mercosul (BRA2A22), uma anotação `@VehiclePlate` pode validar ambos os padrões usando regex diferentes no validador. A classe `ConstraintValidator` receberia a anotação e testaria o valor contra todos os padrões permitidos.

Exemplo simplificado de validador para CNPJ com suporte a formatos futuros

```
@Documented
@Constraint(validatedBy = CnpjValidator.class)
@Target({ ElementType.FIELD, ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface CNPJValido {
    String message() default "{documento.cnpj.invalido}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    boolean allowLegacy() default true;
    boolean allowAlphanumeric() default true;
}

public class CnpjValidator implements ConstraintValidator<CNPJValido,
String> {
    private boolean allowLegacy;
    private boolean allowAlphanumeric;

    @Override
    public void initialize(CNPJValido annotation) {
        this.allowLegacy = annotation.allowLegacy();
        this.allowAlphanumeric = annotation.allowAlphanumeric();
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context)
{
        if (value == null || value.isBlank()) return true;
        // remove máscara
        String normalized = value.replaceAll("[^A-Za-z0-9]", "");
```

```
        boolean legacyOk = allowLegacy && normalized.matches("\\d{14}") &&
validaCnpjNumerico(normalized);
        boolean alphaOk = allowAlphanumeric && normalized.matches("[A-Za-
z0-9]{14}") && validaCnpjAlfanumerico(normalized);
        return legacyOk || alphaOk;
    }

    private boolean validaCnpjNumerico(String cnpj) {
        // implementar algoritmo de cálculo de dígito verificador
        return true;
    }

    private boolean validaCnpjAlfanumerico(String cnpj) {
        // implementar a regra oficial quando disponível
        return true;
    }
}
```

Esse exemplo demonstra como encapsular a lógica para múltiplos formatos em uma única anotação. Para documentos como CPF, RENAVAM ou placas, a abordagem é semelhante: determinar todos os padrões aceitos e validar se o valor enviado se enquadra em pelo menos um deles.

Convenções de nomenclatura

Ao nomear anotações e classes, procure separar aquilo que é específico do Brasil daquilo que é genérico:

- Use **nomes em português** para validações de documentos nacionais exclusivos (ex.: `@CPF`, `@CNPJ`, `@Renavam`), pois esses termos fazem parte do domínio brasileiro e não têm equivalente direto em inglês.
- Use **nomes em inglês** para validações genéricas que podem ser aplicadas em qualquer contexto (ex.: `@MonetaryAmount` ou `@Money` para valores monetários, `@VehiclePlate` para placas de veículos, `@StateCode` para sigla de estado). Isso facilita a compreensão por equipes internacionais.
- Siga convenções de nomenclatura do Java: inicie anotações e classes com maiúscula e use camel case (`@VehiclePlate`, `@CnpjValido`). Evite abreviações excessivas que dificultem a leitura.

Com essas estratégias, é possível criar validações flexíveis e expressivas que suportam formatos antigos e novos e mantêm o código organizado e claro.

Exemplo de aplicação completa com DynamoDB, cache e Swagger

Para ilustrar as validações e princípios descritos até aqui, segue um projeto de exemplo utilizando **Spring Boot 3.4.5** (Java 21), **AWS DynamoDB** como banco de dados, **caching com Caffeine** e documentação da API via **SpringDoc/Swagger**. O objetivo é ser simples e performático: existe um único recurso `Simulacao` que pode ser criado e consultado por ID. O exemplo utiliza as anotações de validação apresentadas, incluindo as personalizadas (`@Enumeration`, `@DataObrigatoria`, `@ValorMonetario`, etc.).

Dependências Maven

Para aproveitar o ecossistema Spring e evitar código de baixo nível do AWS SDK, utilizaremos o **Spring Cloud AWS** para integrar o DynamoDB. Conforme o tutorial de Spring Cloud AWS, recomenda-se importar o BOM `spring-cloud-aws-dependencies` e usar o starter específico para DynamoDB [28]. O trecho a seguir mostra as dependências principais no `pom.xml`:

```
<dependencyManagement>
  <dependencies>
    <!-- Bill of materials da Spring Cloud AWS -->
    <dependency>
      <groupId>io.awspring.cloud</groupId>
      <artifactId>spring-cloud-aws-dependencies</artifactId>
      <version>3.0.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
    <!-- Starter Web: inclui validação padrão -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Starter DynamoDB via Spring Cloud AWS -->
    <dependency>
        <groupId>io.awspring.cloud</groupId>
        <artifactId>spring-cloud-aws-starter-dynamodb</artifactId>
    </dependency>
    <!-- Lombok para gerar código boilerplate -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <!-- Hibernate Validator com anotações BR (CPF, CNPJ) -->
    <dependency>
        <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator</artifactId>
    </dependency>
    <!-- Cache com Caffeine -->
    <dependency>
        <groupId>com.github.ben-manes.caffeine</groupId>
        <artifactId>caffeine</artifactId>
    </dependency>
    <!-- Resilience4j para resiliência -->
    <dependency>
        <groupId>io.github.resilience4j</groupId>
```

```
        <artifactId>resilience4j-spring-boot3</artifactId>
    </dependency>
    <!-- Actuator e Micrometer para métricas -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-datadog</artifactId>
    </dependency>
    <!-- Documentação OpenAPI/Swagger -->
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    </dependency>
</dependencies>
```

Estrutura de pacotes sugerida

```
com.example.demo
├── DemoApplication.java        // classe principal
├── controller
│   └── SimulacaoController.java
├── service
│   └── SimulacaoService.java
├── repository
│   └── SimulacaoRepository.java
├── model
│   └── Simulacao.java          // entidade para DynamoDB
└── validation
    ├── Enumeration.java
    ├── EnumerationValidator.java
    ├── DataObrigatoria.java
    ├── ValorMonetario.java
    └── ...                     // demais anotações customizadas
```

Classe principal

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@EnableCaching // habilita o cache do Spring
public class DemoApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Entidade e DTO para DynamoDB

Utilizaremos o **DynamoDB Enhanced Client** do AWS SDK, que mapeia classes POJO para tabelas. O objeto `Simulacao` será a entidade salva na tabela. Para simplificar, reusaremos a mesma classe para entrada e persistência; em cenários reais, convém separar DTO e entidade.

```
package com.example.demo.model;

import com.example.demo.enums.TipoServicoEnum;
import com.example.demo.validation.*;
import jakarta.validation.Valid;
import jakarta.validation.constraints.*;
import lombok.*;
import org.hibernate.validator.constraints.br.CNPJ;
import org.hibernate.validator.constraints.br.CPF;
import org.springframework.format.annotation.DateTimeFormat;
import
software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.DynamoDbBean;
import
software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.DynamoDbPartitionKey;

import java.math.BigDecimal;
import java.time.LocalDate;

@Data
@DynamoDbBean
@NoArgsConstructor
@AllArgsConstructor
public class Simulacao {
    private String id;

    @Renavam(message = "{simulacao.renavam.obrigatorio}")
    private String renavam;

    @Placa(message = "{simulacao.placa.invalida}")
    private String placa;

    @NotNull(message = "{simulacao.tipoServico.obrigatorio}")
    @Enumeration(enumClass = TipoServicoEnum.class, message =
"{simulacao.tipoServico.invalido}")
    private String tipoServico;

    @Uf(message = "{simulacao.uf.invalida}")
    private String uf;
```

```
    @ValorMonetario(message = "{simulacao.debitos.invalido}")
    private BigDecimal valorTotalDebitos;

    @Valid
    private Multa multa;

    @CPF(message = "{simulacao.cpf.invalido}")
    private String cpf;

    @CNPJ(message = "{simulacao.cnpj.invalido}")
    private String cnpj;

    @DataObrigatoria(message = "{simulacao.dataEmissao.invalida}")
    private LocalDate dataEmissao;

    @DynamoDbPartitionKey
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    // classe interna para a multa
    @Data
    @NoArgsConstructor
    @AllArgsConstructor
    public static class Multa {
        @DataObrigatoria(message = "{simulacao.multa.data.invalida}")
        private LocalDate dataMulta;

        @NotBlank(message = "{simulacao.multa.descricao.obrigatoria}")
        private String descricao;

        @ValorMonetario(message = "{simulacao.multa.valor.invalido}")
        private BigDecimal valor;
    }
}
```

Repositório DynamoDB

O repositório utiliza o `DynamoDbTemplate` fornecido pelo Spring Cloud AWS, que simplifica as operações CRUD. O `DynamoDbTemplate` é automaticamente configurado ao incluir o starter de DynamoDB. O exemplo abaixo mostra como salvar e buscar uma simulação:

```
package com.example.demo.repository;

import com.example.demo.model.Simulacao;
import io.awspring.cloud.dynamodb.DynamoDbTemplate;
```

```
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Repository;

@Repository
@Slf4j
@RequiredArgsConstructor
public class SimulacaoRepository {
    private final DynamoDbTemplate dynamoDbTemplate;

    public Simulacao save(Simulacao simulacao) {
        log.info("Salvando simulação no DynamoDB: {}", simulacao);
        dynamoDbTemplate.save(simulacao);
        return simulacao;
    }

    public Simulacao findById(String id) {
        log.info("Buscando simulação {} no DynamoDB", id);
        return dynamoDbTemplate.load(Simulacao.class, id);
    }
}
```

Nessa implementação, o Spring injeta automaticamente o `DynamoDbTemplate` configurado com as credenciais e região definidos em `application.yml`. Isso elimina a necessidade de manipular diretamente o `DynamoDbEnhancedClient` e reduz a quantidade de código de infraestrutura.

Serviço com cache Caffeine

O serviço orquestra a validação (via `@Valid` no controlador), gera o identificador da simulação, persiste e faz cache. O cache é configurado pelo Spring automaticamente com Caffeine.

```
package com.example.demo.service;

import com.example.demo.model.Simulacao;
import com.example.demo.repository.SimulacaoRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.cache.annotation.CachePut;
import org.springframework.stereotype.Service;

import java.util.UUID;

@Service
@Slf4j
@RequiredArgsConstructor
public class SimulacaoService {
    private final SimulacaoRepository repository;

    @CachePut(cacheNames = "simulacoes", key = "#result.id")
```

```
    public Simulacao create(Simulacao simulacao) {
        simulacao.setId(UUID.randomUUID().toString());
        log.info("Criando simulação de id {}", simulacao.getId());
        return repository.save(simulacao);
    }

    @Cacheable(cacheNames = "simulacoes", key = "#id")
    public Simulacao findById(String id) {
        log.info("Solicitando simulação {}", id);
        return repository.findById(id);
    }
}
```

Configuração de cache Caffeine

```
package com.example.demo.config;

import com.github.benmanes.caffeine.cache.Caffeine;
import org.springframework.cache.CacheManager;
import org.springframework.cache.caffeine.CaffeineCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.concurrent.TimeUnit;

@Configuration
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager manager = new
CaffeineCacheManager("simulacoes");
        manager.setCaffeine(Caffeine.newBuilder()
                .expireAfterWrite(10, TimeUnit.MINUTES)
                .maximumSize(1000));
        return manager;
    }
}
```

Controlador REST com Swagger

O controlador expõe endpoints para criar e recuperar simulações. Os parâmetros são validados automaticamente e os logs documentam cada etapa. A presença do `springdoc-openapi` gera a documentação Swagger em `/swagger-ui.html`.

```
package com.example.demo.controller;

import com.example.demo.model.Simulacao;
import com.example.demo.service.SimulacaoService;
import io.swagger.v3.oas.annotations.Operation;
```

```java
import io.swagger.v3.oas.annotations.tags.Tag;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/simulacoes")
@RequiredArgsConstructor
@Slf4j
@Tag(name = "Simulacoes", description = "Operações de criação e consulta de
simulações")
public class SimulacaoController {
    private final SimulacaoService service;

    @Operation(summary = "Cria uma nova simulação")
    @PostMapping
    public ResponseEntity<Simulacao> create(@Valid @RequestBody Simulacao
simulacao) {
        log.info("Requisição de criação de simulação recebida");
        Simulacao created = service.create(simulacao);
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }

    @Operation(summary = "Obtém uma simulação pelo id")
    @GetMapping("/{id}")
    public ResponseEntity<Simulacao> getById(@PathVariable String id) {
        log.info("Requisição de busca de simulação {} recebida", id);
        Simulacao simulacao = service.findById(id);
        if (simulacao == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(simulacao);
    }
}
```

Observações sobre o uso de DynamoDB e Swagger

- **DynamoDB** – O exemplo utiliza o *starter* `spring-cloud-aws-starter-dynamodb`, que registra automaticamente o cliente e o `DynamoDbTemplate`. A classe `Simulacao` continua anotada com `@DynamoDbBean` e `@DynamoDbPartitionKey`, e o repositório injeta o `DynamoDbTemplate` para operações CRUD. Para funcionamento correto é necessário configurar as credenciais e a região no `application.yml` (como mostrado no exemplo). Em ambientes locais, a propriedade `endpoint` pode apontar para o DynamoDB Local.
- **Cache** – O cache com Caffeine melhora a performance de leituras repetidas. O método `create` atualiza o cache via `@CachePut`, enquanto `findById` consulta o cache antes de acessar o banco.

- **Swagger/OpenAPI** – Com a dependência `springdoc-openapi-starter-webmvc-ui`, a documentação é gerada automaticamente. As anotações `@Operation` e `@Tag` enriquecem a descrição dos endpoints. Personalizações avançadas (como UI customizada) podem ser implementadas se desejado, mas o exemplo usa a configuração padrão.

Com esse projeto de exemplo, todas as validações discutidas anteriormente são aplicadas em um serviço real, permitindo inclusão e consulta de registros no DynamoDB com desempenho melhorado por cache e documentação automática via Swagger. As boas práticas de nomeação, mensagens externas e anotações customizadas garantem que o código seja robusto e extensível.

Melhorias adicionais: resiliência, escalabilidade e observabilidade

Além da implementação básica, aplicações de produção exigem atenção a aspectos como monitoramento, desempenho e tolerância a falhas. Seguem algumas técnicas e bibliotecas que podem ser incorporadas ao projeto para torná-lo mais robusto e escalável:

Actuator e Micrometer para monitoramento

O **Spring Boot Actuator** expõe endpoints de saúde, métricas, logs e rastreamento. Para utilizá-lo, inclua a dependência `spring-boot-starter-actuator` e configure os endpoints que deseja expor. Em conjunto, o **Micrometer** coleta métricas e as envia para ferramentas como Datadog, Prometheus ou New Relic. Para enviar métricas ao Datadog, adicione `micrometer-registry-datadog` e configure a API key, conforme abaixo:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-datadog</artifactId>
</dependency>
```

Em `application.yml`, especifique quais endpoints do Actuator serão expostos (por exemplo, `/health`, `/info`, `/metrics`) e configure o exportador do Datadog. O exemplo abaixo expõe apenas os endpoints essenciais e envia métricas a cada 30 segundos:

```
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus
  metrics:
    export:
      datadog:
        enabled: true
```

```
        api-key: ${DATADOG_API_KEY}
        step: 30s # intervalo entre envios de métricas
```

Com essas configurações, a aplicação disponibiliza endpoints de observabilidade e envia métricas de CPU, memória, taxas de requisição e latência para o Datadog. É possível criar dashboards e alertas com base nessas métricas para acompanhar a saúde do serviço.

Virtual Threads e alto desempenho com JDK 21

O Java 21 traz suporte a **Virtual Threads** (Project Loom), que possibilitam lidar com milhares de requisições concorrentes sem a sobrecarga das threads de sistema. O Spring Boot 3.2+ oferece suporte experimental a virtual threads via propriedade `spring.threads.virtual.enabled`. Basta ativar no `application.yml`:

```
spring:
  threads:
    virtual:
      enabled: true
```

Para que o servidor web utilize *virtual threads* no processamento de requisições, basta habilitar a propriedade `spring.threads.virtual.enabled=true`. O Spring Boot criará *virtual threads* para cada requisição, economizando recursos do sistema. Além disso, para tarefas assíncronas fora do contexto web, é possível definir um executor baseado em virtual threads. O exemplo a seguir demonstra como habilitar as virtual threads no `application.yml` e registrar um executor personalizado:

```
spring:
  threads:
    virtual:
      enabled: true # habilita virtual threads para requisições HTTP
```

```
@Configuration
public class AsyncConfig {
    // Executor assíncrono com virtual threads
    @Bean
    public Executor taskExecutor() {
        return Executors.newVirtualThreadPerTaskExecutor();
    }
}
```

Com esse executor, qualquer método anotado com `@Async` ou qualquer uso manual de `taskExecutor.submit(...)` utilizará *virtual threads*, liberando as threads de sistema enquanto espera I/O. Isso é especialmente útil para chamadas a DynamoDB, HTTP ou serviços externos.

Resiliência com Resilience4j

Para lidar com falhas de serviço (por exemplo, indisponibilidade do DynamoDB), utilize **Resilience4j**. Ele fornece circuit breakers, retry e bulkhead. Exemplo de dependência:

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot3</artifactId>
</dependency>
```

Com a dependência adicionada, é possível anotar métodos que chamam serviços externos com `@Retry`, `@CircuitBreaker` ou `@Bulkhead`. O Spring cria automaticamente os *aspects* que interceptam as chamadas e aplicam as políticas configuradas. No exemplo abaixo, o serviço `SimulacaoService` tenta recuperar a simulação do banco até 3 vezes em caso de erro, com um tempo de espera entre tentativas definido no `application.yml`:

```
import io.github.resilience4j.retry.annotation.Retry;

@Service
@Slf4j
@RequiredArgsConstructor
public class SimulacaoService {
    private final SimulacaoRepository repository;

    @Retry(name = "buscarSimulacao", fallbackMethod = "fallbackBuscar")
    public Simulacao findById(String id) {
        log.info("Solicitando simulação {}", id);
        return repository.findById(id); // esta chamada será reexecutada em
caso de exceção
    }

    // método de fallback em caso de falha após todas as tentativas
    public Simulacao fallbackBuscar(String id, Throwable t) {
        log.error("Falha ao buscar simulação {}: {}", id, t.getMessage());
        return null;
    }
}
```

O comportamento do `@Retry` é configurado no `application.yml`. Por exemplo, o trecho a seguir define que a política `buscarSimulacao` deve tentar até 3 vezes com intervalo inicial de 200 ms e aumento exponencial:

```
resilience4j:
  retry:
    instances:
      buscarSimulacao:
        max-attempts: 3
        wait-duration: 200ms
```

```
        enable-exponential-backoff: true
        exponential-backoff-multiplier: 2.0
```

Da mesma forma, é possível aplicar `@CircuitBreaker` para abrir um circuito após repetidas falhas, impedindo chamadas subsequentes até que o serviço externo se recupere. A configuração dos circuit breakers é análoga e permite ajustar o *failure rate threshold*, a janela de amostragem e o tempo de espera para o meio-fechado.

Observabilidade distribuída com Datadog APM ou OpenTelemetry

Além de métricas, é recomendável instrumentar a aplicação com **tracing distribuído**. O Spring Boot oferece integração com **OpenTelemetry** via `spring-boot-starter-otel-autoconfigure`. Os traces podem ser enviados ao Datadog APM configurando o exportador OTLP. Isso ajuda a rastrear chamadas entre serviços e identificar gargalos.

Imagem Docker otimizada

Para construir a aplicação em um container, utilize uma imagem oficial da Amazon com JDK 21, como `amazoncorretto:21-alpine`, que é enxuta e otimizada para a AWS. Um exemplo de `Dockerfile` multi-stage:

```
FROM maven:3.9-amazoncorretto-21 AS build
WORKDIR /workspace
COPY pom.xml .
COPY src ./src
RUN mvn -B package -DskipTests

FROM amazoncorretto:21-alpine
WORKDIR /app
COPY --from=build /workspace/target/demo-*.jar app.jar
ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Graceful shutdown em contêineres e EKS

Quando a aplicação é executada em contêineres (por exemplo, pods de Kubernetes ou ECS/EKS), é importante garantir que ela finalize as requisições em andamento antes de encerrar o processo. O Spring Boot 3 habilita o **graceful shutdown**: ao receber um sinal de término (SIGTERM), o servidor HTTP para de aceitar novas conexões e aguarda a conclusão das requisições ativas dentro de um tempo limite configurável. Para ativar esse comportamento, defina `server.shutdown=graceful` no arquivo de propriedades. Em um tópico de discussão do Stack Overflow, um usuário incluiu a propriedade `server.shutdown=graceful` em `application.properties` para habilitar o desligamento gracioso [29].

Em aplicações que rodam em EKS, é recomendável combinar esse recurso com os *hooks* de ciclo de vida do Kubernetes. Um pod pode definir um `preStop` e um tempo de

`terminationGracePeriodSeconds` para garantir que o load balancer pare de encaminhar tráfego antes que o processo finalize. A sequência recomendada é:

1. O Kubernetes envia o sinal SIGTERM ao processo do Java quando o pod deve ser encerrado (por exemplo, durante um deploy).
2. O Spring Boot, com `server.shutdown=graceful`, deixa de aceitar novas requisições e aguarda até que as requisições atuais terminem ou até expirar o tempo limite de shutdown.
3. O load balancer remove o pod da lista de *endpoints* ao receber a atualização de *readiness probe*, evitando novas conexões.

Para ajustar o tempo limite de shutdown, use a propriedade `spring.lifecycle.timeout-per-shutdown-phase`, que define quanto tempo o Spring esperará em cada fase (por padrão, 30 segundos). Exemplo de configuração em `application.yml`:

```
server:
  shutdown: graceful

spring:
  lifecycle:
    timeout-per-shutdown-phase: 30s # aguarda até 30 segundos para concluir
requisições
```

Se a aplicação manipular *executors* próprios (por exemplo, processamentos assíncronos), certifique-se de configurar os executores para aguardar a conclusão das tarefas, chamando `setWaitForTasksToCompleteOnShutdown(true)` no caso do `ThreadPoolTaskExecutor`. O post do Stack Overflow mostra um exemplo em que o autor configurou `setWaitForTasksToCompleteOnShutdown(true)` para permitir que tarefas assíncronas terminem antes de finalizar a aplicação [29]. Quando associadas aos *virtual threads* e aos mecanismos de *retry* (Resilience4j), essas práticas resultam em desligamento seguro e previsível.

A seguir um exemplo de configuração de um `ThreadPoolTaskExecutor` que espera os workers terminarem antes do shutdown. Essa configuração pode ser adicionada em uma classe `@Configuration` da aplicação:

```
@Configuration
public class ExecutorConfig {
    @Bean(name = "applicationTaskExecutor")
    public ThreadPoolTaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(10);
        executor.setMaxPoolSize(20);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("app-exec-");
        // aguarda a conclusão das tarefas no shutdown
        executor.setWaitForTasksToCompleteOnShutdown(true);
        executor.setAwaitTerminationSeconds(30);
        executor.initialize();
        return executor;
```

```
    }
}
```

Com `setWaitForTasksToCompleteOnShutdown(true)` e `setAwaitTerminationSeconds(30)`, o Spring aguardará até 30 segundos para que as tarefas em execução terminem antes de encerrar o processo. Essa configuração complementa o comportamento de graceful shutdown do servidor HTTP.

Outras melhorias possíveis

- **Validação e sanitização de entradas** – Adicionar bibliotecas de segurança (OWASP ESAPI ou Spring Security) para sanitizar entradas e evitar injeção.
- **Controle de versionamento de API** – Usar versionamento no caminho (`/api/v1/simulacoes`) ou via cabeçalhos para permitir evolução sem quebrar clientes antigos.
- **Internacionalização (i18n)** – Suportar múltiplos idiomas no `messages.properties` e no Swagger, definindo locale por `Accept-Language`.
- **Monitoramento de logs estruturados** – Configurar o Logback para gerar logs JSON com campos de contexto, facilitando integração com plataformas como Datadog e ELK.
- **Testes e CI/CD** – Incluir testes unitários e de integração (por exemplo, usando Testcontainers para DynamoDB local) e configurar um pipeline CI/CD que faça build e push da imagem Docker para ECR, seguido de deploy em ECS ou EKS.

Arquivo `application.yml` completo (exemplo)

Para reunir as configurações apresentadas ao longo deste relatório em um único lugar, segue um exemplo de arquivo `application.yml` que habilita o graceful shutdown, virtual threads, cache Caffeine, Resilience4j, Actuator com exportador Datadog e configurações de tempo de vida:

```yaml
# Configura o shutdown gracioso do servidor
server:
  shutdown: graceful

# Timeout de cada fase de shutdown (30 s)
spring:
  lifecycle:
    timeout-per-shutdown-phase: 30s
  # habilita virtual threads para requisições HTTP
  threads:
    virtual:
      enabled: true
  cache:
    cache-names: simulacoes
    caffeine:
      spec: maximumSize=1000,expireAfterWrite=10m

  # Configurações do Spring Cloud AWS para DynamoDB
  cloud:
    aws:
      region:
        static: sa-east-1
```

```
      credentials:
        access-key: ${AWS_ACCESS_KEY_ID}
        secret-key: ${AWS_SECRET_ACCESS_KEY}
      dynamodb:
        endpoint: http://localhost:8000
# use a endpoint local para testes ou deixe vazio para AWS
        table-name-prefix: simulacoes-

# Configurações de Retentativas (Resilience4j)
resilience4j:
  retry:
    instances:
      buscarSimulacao:
        max-attempts: 3
        wait-duration: 200ms
        enable-exponential-backoff: true
        exponential-backoff-multiplier: 2.0

# Actuator e Micrometer com exportação para Datadog
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus
  metrics:
    export:
      datadog:
        enabled: true
        api-key: ${DATADOG_API_KEY}
        step: 30s

logging:
  level:
    root: INFO
    com.example.demo: DEBUG
```

Este arquivo centraliza as propriedades necessárias para habilitar todas as funcionalidades descritas (exceto as chaves específicas do DynamoDB, que devem ser configuradas via `application.yml` ou variáveis de ambiente). É uma base para iniciar o projeto com boas práticas de shutdown, escalabilidade, cache, resiliência e observabilidade.

Conclusão

Ao longo deste relatório apresentamos uma visão abrangente das boas práticas de **validação** em aplicações Spring Boot 3.4.5 (Java 21), desde restrições padrão do Bean Validation até a criação de anotações personalizadas para cenários específicos como enumerações e documentos brasileiros. Mostramos como **externalizar mensagens**, **criar anotações compostas** e **nomear cuidadosamente** as validações para manter o código limpo e reutilizável. Em seguida, desenvolvemos um **exemplo prático** que integra DynamoDB, cache com Caffeine e documentação via Swagger, demonstrando como aplicar as validações em um serviço REST.

Para levar a aplicação a um nível de produção, exploramos melhorias de **resiliência** (Resilience4j), **observabilidade** (Actuator, Micrometer, Datadog APM), **escalabilidade** (virtual threads e configuração de executores), **desligamento gracioso** em ambientes de contêiner (Kubernetes/EKS) e otimização de containers com **imagens multi-stage**. Também dedicamos uma seção à **segurança contra injeções e XSS**, abordando estratégias de validação, uso de consultas preparadas, codificação de saída, sanitização de HTML e filtros globais, conforme orientações da OWASP [19] [27].

Essas práticas combinadas permitem construir serviços modernos que são resilientes a falhas, escaláveis sob carga, observáveis e seguros contra injeção de código. O uso de validação robusta, codificação correta, sanitização e APIs seguras garante que a aplicação possa atender às demandas atuais e futuras de ambientes exigentes de nuvem.

---

[1] [3] [4] [7] Using Bean Validation Constraints
https://javaee.github.io/tutorial/bean-validation002.html

[2] [5] [6] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] 4.1. Input Validation — TERASOLUNA Server Framework for Java (5.x) Development Guideline 5.2.1.RELEASE documentation
https://terasolunaorg.github.io/guideline/5.2.1.RELEASE/en/ArchitectureInDetail/WebApplicationDetail/Validation.html

[19] [20] Input Validation - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

[21] [22] [23] [24] Cross Site Scripting Prevention - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[25] [26] Don’t try to sanitize input. Escape output.
https://benhoyt.com/writings/dont-sanitize-do-escape/

[27] Injection Prevention - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html

[28] DynamoDB with Spring Cloud AWS DynamoDbTemplate
https://howtodoinjava.com/spring-boot/spring-cloud-aws-dynamodbtemplate/

[29] Spring boot graceful shutdown not waiting for loop to finish - Stack Overflow
https://stackoverflow.com/questions/79173415/spring-boot-graceful-shutdown-not-waiting-for-loop-to-finish