DESENVOLVIMENTO *WEB* – *BACK END*

AULA 5

Prof.ª Luciane Yanase Hirabara Kanashiro

CONVERSA INICIAL

Nesta etapa de nossos estudos, será abordado um assunto vital no desenvolvimento de *software*: os testes. O teste de *software* é a espinha dorsal da qualidade, assegurando que nossas aplicações atendam não apenas aos requisitos, mas também proporcionem uma experiência sólida aos usuários, por essa razão vamos (re)ver alguns tópicos importantes sobre teste de *software*.

Ao abordar os testes funcionais com Postman, entenderemos como essa ferramenta facilita a validação de APIs, automatizando processos e garantindo que nossos serviços estejam perfeitamente alinhados. Em seguida, mergulharemos no mundo das mensagens ao usuário com Thymeleaf, explorando como essa poderosa *template engine* nos permite criar interfaces dinâmicas e envolventes.

Não deixaremos de lado a validação, abordando tanto o *back end* quanto o *front end*, em que asseguramos a integridade dos dados em ambas as extremidades da aplicação. Por fim, destacaremos a importância dos testes unitários com JUnit, construindo uma base sólida para garantir a confiabilidade e a estabilidade em cada pedaço de código. Pronto para explorar esses elementos essenciais no mundo do desenvolvimento de *software*? Vamos começar!

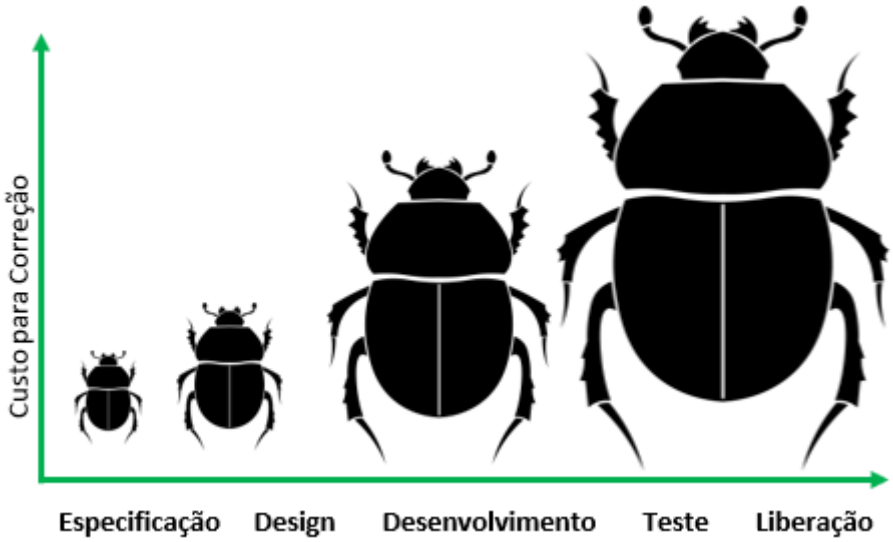
TEMA 1 – TESTE DE *SOFTWARE*

Os testes de *software* são essenciais ao longo do ciclo de vida do desenvolvimento de *software*, sendo realizados em várias fases, desde a concepção e codificação até a manutenção contínua do sistema. Recorremos à prática de testes de *software* para assegurar o adequado funcionamento do *software*, atendendo às especificações estabelecidas.

De acordo com Pressman, esse teste é parte integrante de um conceito mais abrangente, frequentemente denominado verificação e validação (V&V). A verificação compreende um conjunto de atividades cujo propósito é garantir que o *software* implemente de maneira precisa uma função específica. Por outro lado, a validação engloba uma série de tarefas destinadas a assegurar que o *software* tenha sido desenvolvido e possa ser rastreado conforme os requisitos do cliente (Pressman, 2021, p. 373).

No contexto do desenvolvimento de *software*, os custos associados à modificação de um programa crescem à medida que o projeto avança. No gráfico a seguir, podemos visualizar esse acréscimo nos custos de correção de falhas. Perceba que, conforme progredimos nas fases do desenvolvimento de *software*, os custos para a resolução de *bugs* aumentam significativamente. Adicionalmente, destaca-se que a detecção precoce dessas falhas durante os testes de *software* facilita consideravelmente sua correção.

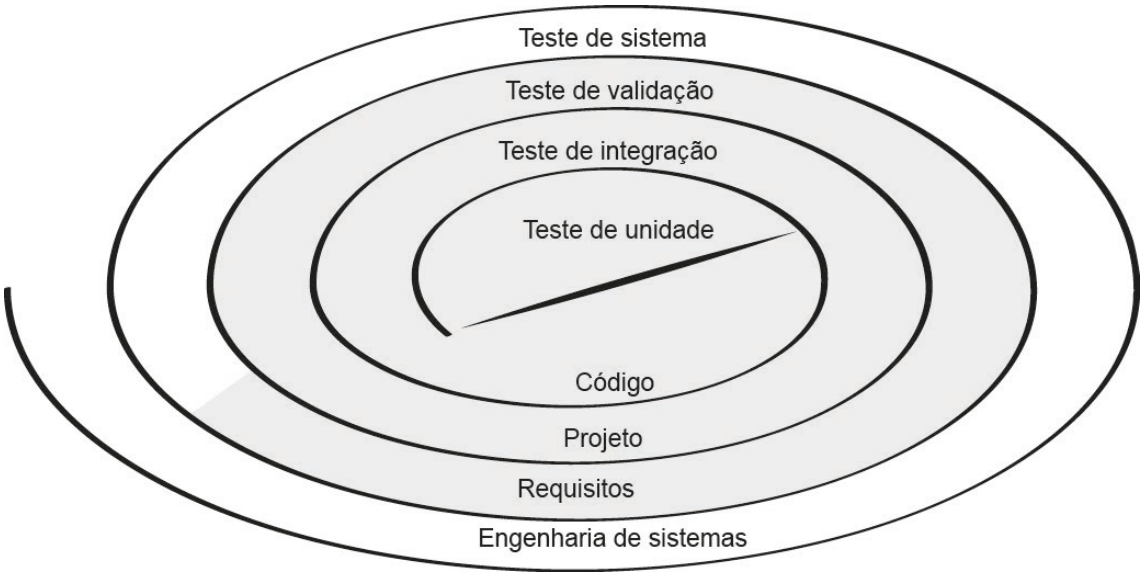
Figura 1 – Tempo em que o *bug* é encontrado



Crédito: Kholil Wahyudi/Shutterstock.

Os testes de *software* desempenham um papel crucial na garantia da qualidade do produto final. A Garantia de Qualidade de *Software* envolve um conjunto abrangente de atividades técnicas implementadas ao longo de todo o processo de desenvolvimento. Esse conjunto de atividades abarca diversos testes, visando assegurar a entrega de um *software* de alta qualidade. A figura a seguir ilustra uma estratégia de teste de *software*, integrada ao conceito de espiral proposto por Pressman.

Figura 2 – Teste de software



Fonte: Pressman, 2021.

De acordo com Pressman (2021), o teste de unidade começa no centro da espiral e se concentra em cada unidade como: componente, classe ou objeto de conteúdo de WebApp do *software*, conforme implementado no código-fonte. O teste prossegue movendo-se em direção ao exterior da espiral, passando pelo teste de integração, em que o foco está no projeto e na construção da arquitetura de *software*.

Continuando na mesma direção da espiral, encontramos o teste de validação, em que requisitos estabelecidos como parte dos requisitos de modelagem são validados em relação ao *software* criado. Por fim, chegamos ao teste do sistema, no qual o *software* e outros elementos são testados como um todo. Para testar um *software* de computador, percorre-se a espiral em direção ao seu exterior, ao longo de linhas que indicam o escopo do teste a cada volta (Pressman, 2021, p. 376).

O quadro a seguir resume o que cada teste faz.

Quadro 1 – Testes

Teste de Sistema **(Teste de interface com o Usuário)**	Compõe as atividades que verificam se todos os requisitos estão de acordo com o especificado e se foram realmente atendidos. Realizada após a revisão de todos os componentes.
Teste de Validação	Mostra se o *software* atende aos requisitos, mostra se um programa faz o que é proposto a fazer.

Teste de Integração	Verifica se as unidades se comunicam, se integram corretamente sem falhas. Realizado após serem testadas as unidades individualmente.
Teste de Unidade (Teste unitário)	Tem como objetivo testar pequenas unidades em um sistema, ou seja, verificar pequenas unidades que compõem o sistema. Geralmente de responsabilidade do próprio desenvolvedor.

TEMA 2 – TESTES FUNCIONAIS

Os **testes funcionais** em qualidade de *software* são uma categoria de testes que se concentram em avaliar se o *software* está atendendo corretamente às especificações funcionais. Esses testes buscam verificar se as funções e características do *software* estão operando conforme o esperado e se ele está cumprindo os requisitos definidos no início do processo de desenvolvimento. Os **testes funcionais** abrangem uma variedade de níveis, incluindo **testes unitários** e de i**ntegração**.

Como principais características dos testes funcionais, podemos citar a verificação das funcionalidades, verificação de conformidade com os Requisitos, exploração de cenários de Uso e verificação de entrada e saída. A seguir estão descritas essas características:

- **Verificação das funcionalidades:** os testes funcionais avaliam as diversas funcionalidades do *software* para garantir que cada uma delas esteja realizando as operações conforme o planejado. Isso inclui a interação com a interface do usuário, o processamento correto de dados, e a execução adequada de operações específicas.
- **Conformidade com os requisitos:** esses testes têm como objetivo principal verificar se o *software* está em conformidade com os requisitos especificados no início do projeto. Isso abrange tanto requisitos de negócios quanto requisitos técnicos.
- **Cenários de uso:** os testes funcionais exploram cenários de uso típicos do *software*, simulando a interação do usuário com o sistema. Isso inclui a entrada de dados, navegação pela interface e todas as ações que um usuário real poderia realizar.
- **Entrada e saída:** verificam se a entrada fornecida ao *software* produz os resultados esperados. Isso abrange a validação de dados de entrada, processamento correto e a geração de saídas esperadas.

Quando aplicável, os testes funcionais também podem avaliar a integração entre diferentes módulos ou componentes do *software* para garantir que eles trabalhem efetivamente juntos.

Importante mencionar ainda que os testes funcionais, muitas vezes, incluem a verificação da documentação associada ao *software*, garantindo que ela esteja atualizada e reflita com precisão as funcionalidades implementadas e, embora não seja exclusivamente relacionado à funcionalidade, os testes funcionais podem incluir ainda avaliações da usabilidade do *software* para garantir uma experiência positiva do usuário.

Os testes funcionais são essenciais para garantir que o software não apenas execute suas funções conforme o esperado, mas também atenda às expectativas do cliente e dos usuários finais. Eles desempenham um papel crucial na validação da qualidade funcional do *software* antes de sua entrega.

2.1 Testes funcionais com Postman[1]

Já vimos algumas características do Postman em nossos estudos, mas nesta seção abordaremos o **Postman** para utilização de testes funcionais. O Postman é uma ferramenta popular usada para realizar **testes de API** e **testes funcionais** em serviços *web*. Ele fornece uma interface gráfica amigável para criar, enviar e testar solicitações HTTP, permitindo a automação de testes e a validação de respostas.

A seguir, temos um panorama geral de como podem ser realizados testes funcionais com o Postman:

- **Criação de solicitações HTTP:** Postman pode ser utilizado para criar solicitações HTTP, como *GET*, *POST*, *PUT* ou *DELETE*, para interagir com sua API ou serviço *web*.
- **Inclusão de dados de entrada:** isso pode incluir parâmetros de consulta, cabeçalhos e corpo da solicitação, dependendo do tipo de solicitação e da API que está sendo testada.
- **Execução de testes automatizados:** o Postman permite a criação de testes automatizados associados a cada solicitação. Esses testes podem ser escritos em JavaScript e são executados após o envio da solicitação.

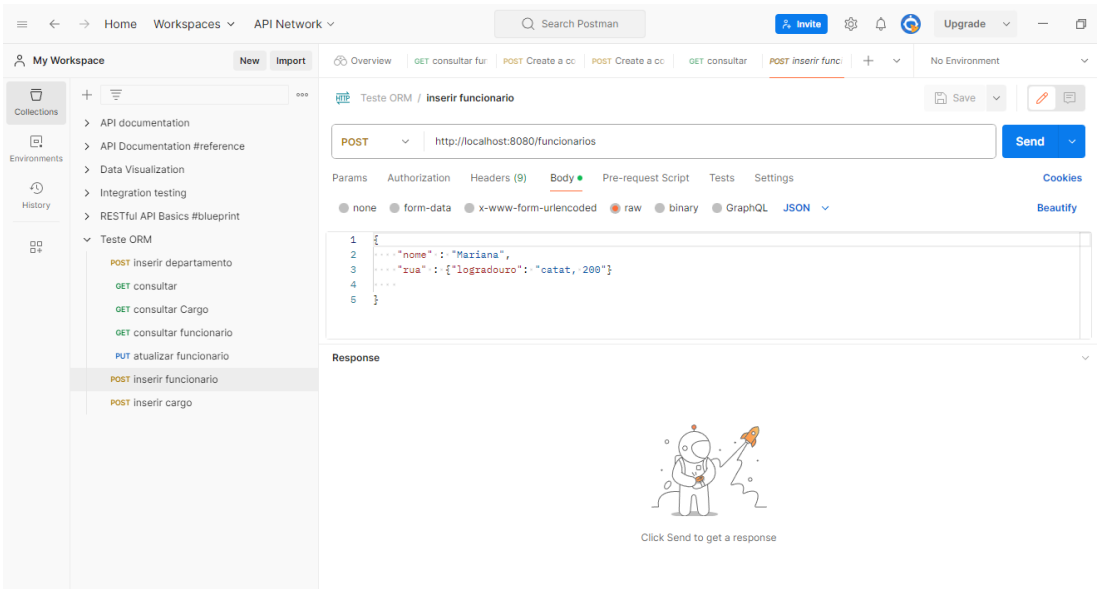
[1] O Postman está disponível para *download* gratuito no *site* oficial <https://www.postman.com/>. Acesso em: 9 maio 2024.

Eles são úteis para verificar se a resposta da API está correta, se os dados esperados estão presentes e se o formato da resposta está de acordo com as expectativas.

- **Validação de respostas:** o Postman fornece uma interface gráfica para visualizar facilmente o conteúdo das respostas, incluindo cabeçalhos, corpo e outros detalhes.
- **Gerenciamento de variáveis:** podem ser utilizadas variáveis no Postman para gerenciar dados dinâmicos, como *tokens* de autenticação ou IDs de recursos. Isso permite a reutilização de valores em várias solicitações e testes.
- **Criação de coleções de testes:** Postman permite organizar suas solicitações e testes em coleções. As coleções facilitam a execução de testes em lote e a organização de casos de teste relacionados.
- **Integração com ambientes de desenvolvimento:** o Postman suporta a criação de ambientes, permitindo que você configure variáveis específicas para diferentes ambientes (por exemplo, desenvolvimento, teste, produção).
- **Relatórios e monitoramento:** Postman oferece ainda recursos de relatórios e monitoramento para acompanhar o desempenho dos testes ao longo do tempo.

A figura a seguir mostra uma tela do Postman para uma requisição POST.

Figura 3 – Requisição POST



Ao se utilizar o Postman, é possível realizar testes funcionais eficazes em suas APIs ou serviços *web*, garantindo a qualidade e confiabilidade de suas implementações. Além do Postman, existem outras ferramentas semelhantes para testes funcionais, como o Insomnia por exemplo.

TEMA 3 – MENSAGENS AO USUÁRIO

Essas mensagens desempenham um papel importante na qualidade do *software*, pois são um meio importante de comunicação entre o sistema e o usuário. No cenário dinâmico das aplicações modernas, as **mensagens ao usuário** desempenham um papel importante na comunicação efetiva entre a aplicação e seus usuários. Essas mensagens, categorizadas como erros, avisos e informações, não são apenas elementos visuais na interface, mas, sim, uma ferramenta flexível de interação que pode moldar significativamente a experiência do usuário.

As **mensagens de erro** têm um impacto direto na experiência do usuário. Se não forem formuladas de maneira clara e útil, podem levar a frustrações e, em última instância, fazer com que o usuário desista da aplicação. Uma mensagem de erro confusa ou genérica pode representar um obstáculo para a experiência do usuário, levando à frustração e até mesmo à desistência do uso da aplicação. Sendo assim, as Mensagens de erro devem ser formuladas com extrema clareza, fornecendo informações precisas sobre o problema e sugerindo soluções tangíveis.

Já as **mensagens de avisos** fornecem informações importantes que podem orientar os usuários sobre ações ou eventos iminentes. São uma forma preventiva de comunicação, alertando sobre possíveis problemas ou fornecendo instruções para garantir uma navegação tranquila, enquanto as **mensagens informativas** oferecem um contexto valioso aos usuários, fornecendo detalhes sobre ações concluídas com sucesso, atualizações do sistema ou simplesmente dicas úteis. São uma ferramenta para empoderar os usuários com informações relevantes.

Podemos ainda afirmar que as mensagens ao usuário são um canal de comunicação flexível, permitindo uma interação mais próxima e personalizada, podendo ser adaptadas para diferentes contextos e necessidades, atendendo tanto a usuários novos quanto experientes.

No que diz respeito às mensagens ao usuário, as **mensagens de erro** com certeza representam o elo mais fraco entre uma aplicação e o usuário. Uma abordagem negligente na comunicação de problemas pode resultar em uma perda de confiança por parte do usuário. Por outro lado, mensagens de erro bem elaboradas podem transformar uma situação desafiadora em uma oportunidade de aprendizado, contribuindo para uma experiência mais positiva.

Em resumo, o cuidado e a atenção dedicados às mensagens do usuário não devem ser subestimados. Elas são a voz da sua aplicação, moldando a percepção e a interação do usuário. Ao compreender a importância de categorizar, ser flexível na comunicação e, acima de tudo, garantir a clareza nas mensagens de erro, sua aplicação pode fortalecer seu relacionamento com os usuários, proporcionando uma experiência mais fluida e satisfatória.

3.1 Mensagens ao usuário com Thymeleaf

Para **aplicações Java**, podemos utilizar o **Thymeleaf** para criar mensagens ao usuário. O Thymeleaf é um motor de *template* para aplicações Java, frequentemente utilizado com o *framework Spring*. Ele permite a incorporação de expressões e lógica de servidor diretamente nos templates HTML. Já abordamos o Thymeleaf anteriormente, mas dessa vez iremos focar na sua utilização para criar mensagens aos usuários.

Ao criar mensagens ao usuário com Thymeleaf, você pode personalizar a apresentação de informações dinâmicas de acordo com o estado da aplicação. O Thymeleaf pode ser utilizado de várias maneiras para exibir mensagens ao usuário:

- **Exibição Simples**: no HTML, você pode usar expressões Thymeleaf para exibir mensagens de maneira condicional. Por exemplo, para exibir uma mensagem de boas-vindas apenas se um usuário estiver autenticado, você pode fazer o seguinte:

```
<div th:if="${authenticated}">
    <p>Bem-vindo, usuário!</p>
</div>
```

- **Exibição de mensagens dinâmicas:** ao utilizar mensagens internacionalizadas (i18n), você pode adaptar o conteúdo das mensagens com base no idioma do usuário. Suponha que você tenha um arquivo de

mensagens messages.properties e messages_es.properties para mensagens em inglês e espanhol, respectivamente:

```
<p th:text="#{welcome.message}">Welcome!</p>
```

O Thymeleaf selecionará automaticamente a mensagem apropriada com base no idioma configurado.

- **Exibição de mensagens de erro:** ao lidar com erros, Thymeleaf pode ser usado para exibir mensagens de erro ou *feedback* ao usuário. Por exemplo:

```
<div th:if="${#fields.hasErrors('campo')}">
    <p th:errors="*{campo}">Erro no campo.</p>
</div>
```

Neste exemplo, se houver um erro no campo especificado, uma mensagem de erro será exibida.

- **Mensagens condicionalmente estilizadas:** você pode condicionalmente estilizar mensagens com base em certas condições. Por exemplo, se você quiser destacar mensagens de sucesso com uma cor verde:

```
<p th:text="${mensagem}" th:class="${sucesso} ? 'sucesso' :
'erro'"></p>
```

Aqui, a classe CSS 'sucesso' será aplicada se a variável de sucesso for verdadeira. Esses exemplos demonstram como o Thymeleaf pode ser uma ferramenta poderosa para personalizar a exibição de mensagens ao usuário em suas aplicações *web*.

TEMA 4 – VALIDAÇÃO *BACK END* E *FRONT END*

Os testes funcionais e a validação estão intrinsecamente relacionados no contexto do desenvolvimento de *software*, pois ambos têm como objetivo garantir que o *software* atenda aos requisitos especificados e funcione conforme esperado. Os testes funcionais desempenham um papel significativo na validação do *software*. Eles constituem uma parte prática da validação, pois são a implementação direta da verificação de que as funcionalidades específicas estão corretas e em conformidade com os requisitos.

A **validação de campos** em um sistema é uma prática essencial para garantir a **integridade, segurança e usabilidade** das informações manipuladas. Entre as diversas razões para validar campos, destacam-se três pontos cruciais que contribuem para a robustez e confiabilidade de uma aplicação: garantir formato correto, garantir tipo correto e prevenir a injeção de código malicioso. A seguir, estão descritos estes três pontos:

- **Garantir formato correto:** a validação de formato visa assegurar que os dados inseridos em campos específicos sigam padrões predefinidos. Por exemplo, ao coletar informações como números de telefone, *e-mails* ou CPFs, é crucial que esses dados estejam formatados de acordo com as normas estabelecidas. A validação de formato impede a entrada de dados inválidos, contribuindo para a consistência e correção das informações armazenadas.

A validação de formato abrange campos que requerem uma estrutura específica, tais como endereços de *e-mail*, números telefônicos, CEP, nomes e senhas. Ao assegurar que esses campos sigam padrões pré-definidos, como um formato de *e-mail* válido ou a presença apenas de números em um campo de telefone, a validação de formato evita inconsistências e facilita a compreensão e manipulação posterior dos dados.

- **Garantir tipo correto:** além de garantir o formato adequado, é essencial validar o tipo de dado inserido em determinados campos. Isso significa confirmar se a informação é do tipo esperado, como números, datas ou texto. A validação de tipo evita inconsistências e erros causados pela entrada de dados incompatíveis, fortalecendo a confiabilidade das operações que dependem dessas informações.
- **Prevenir injeção de código malicioso:** a segurança é uma preocupação primordial no desenvolvimento de *software*. Validar campos é uma estratégia eficaz para prevenir a injeção de código malicioso, um tipo comum de ataque cibernético. Campos de entrada que não são devidamente validados podem ser explorados por invasores para injetar *scripts* ou comandos maliciosos. A validação adequada garante que apenas dados legítimos e seguros sejam processados, protegendo o sistema contra vulnerabilidades e mantendo a integridade dos dados.

A validação de campos em aplicações *web* é uma prática fundamental para garantir a integridade e a qualidade dos dados fornecidos pelos usuários. Essa validação pode ocorrer tanto no lado cliente quanto no lado servidor, e cada abordagem desempenha um papel único no processo de garantir que as informações inseridas estejam de acordo com as expectativas e requisitos do sistema.

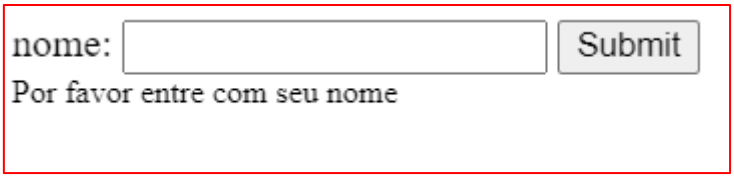
4.1 Validação *front end*

A validação de campos no lado cliente ocorre no navegador do usuário, utilizando recursos como HTML5 e JavaScript. Esse método oferece benefícios imediatos, pois permite que erros sejam detectados e corrigidos antes mesmo de os dados serem enviados para o servidor. Isso resulta em uma experiência mais ágil para o usuário, pois as respostas são instantâneas.

4.1.1 Validação com JavaScript

A validação com **JavaScript** requer codificação e **domínio da linguagem**. É uma validação feita no Lado do cliente ou seja no *Front end*. A figura a seguir ilustra um formulário html com o campo nome a ser preenchido. Caso o usuário tente submeter o campo sem digitar nada, uma mensagem é exibida (“Por favor, entre com seu nome”). Essa validação foi feita utilizando JavaScript.

Figura 3 – Formulário HTML



A seguir, estão os códigos utilizados para gerar o formulário anterior. Eles podem ser testados rapidamente no *codepen*[2] ou em qualquer outro *site* ou ferramenta similar, copiando e colando os códigos.

arquivo form_js.html

[2] Disponível em: <https://codepen.io/pen/>.

```
<html>
<head>
<link rel="stylesheet" href="estilo.css">
</head>
<form id="form">
  <label for="nome"> nome: </label>
  <input type="text" name="nome" id="nome">
  <button id="submit">Submit</button>
  <span role="alert" id="nameError" aria-hidden="true"> Por favor entre com seu nome </span>
</form>
<script type="text/javascript" src="script.js"></script>
</body>
</html>
```

Arquivo estilo.css:

```
#nameError {
  display: none;
  font-size: 0.8em;
}
#nameError.visible {
  display: block;
}
input.invalid {
  border-color: red;
}
```

Arquivo script.js

```
const submit = document.getElementById("submit");
submit.addEventListener('click', validate);
function validate(e) {
  e.preventDefault();
  const campoNome = document.getElementById("nome");
  let valid = true;
  if (!campoNome.value) {
    const nameError = document.getElementById("nameError");
    nameError.classList.add("visible");
    nameError.setAttribute('aria-hidden', false);
    nameError.setAttribute('aria-invalid', true);
  }
  return valid;
}
```

Esse código em JavaScript adiciona um evento de clique ao elemento com o id "submit" que chama a função ***validate***. A função ***validate*** é responsável por

validar um campo de formulário com o id "nome". Se o campo estiver vazio, ele exibe uma mensagem de erro e marca o campo como inválido. O **e.preventDefault()** é usado para impedir o envio do formulário caso a validação falhe.

4.1.2 Validação com HTML 5

A validação com HTML é mais simples. As validações são realizadas a partir dos atributos da *tag input*: *type*, *pattern* e *require*.

O atributo *type* permite a especificação do tipo de dado esperado para o campo. Diversos tipos estão disponíveis, como *text*, *email*, *number* e *date.* Essa simples declaração contribui para a validação básica do formato dos dados inseridos, orientando o navegador sobre o tipo de informação aguardada.

O exemplo a seguir ilustra a utilização do tipo *email* para garantir que o formato do texto digitado no campo corresponda a um endereço de *email.*

```
<form>
  <h2>Validação de e-mail</h2>
  <input type="email" value="" placeholder="nome@email.com"
required>
  <input type="submit" value="Enviar">
</form>
```

4.1.3 Atributo *pattern*

O atributo *pattern* possibilita a definição de uma expressão regular (RegEx) que o valor do campo deve atender. Expressões regulares são padrões utilizados para selecionar combinações de caracteres em uma *string*. Isso oferece uma validação mais avançada, permitindo a especificação de padrões específicos para formatos de dados complexos. O exemplo a seguir ilustra a utilização de expressões regulares para padronizar o formato do número de telefone.

```
<input type="tel" required="required" maxlength="15"
name="telefone" pattern="[0-9]{2}-[0-9]{4,6}-[0-9]{3,4}$"
/>
```

O atributo *required* indica que um campo é obrigatório e não pode ser submetido sem um valor. Isso simplifica a validação de preenchimento,

assegurando que informações essenciais sejam fornecidas antes do envio do formulário.

O exemplo a seguir mostra o código completo do formulário com a utilização do atributo *required* em conjunto com os atributos *type* e *pattern* em campos para inserir telefone.

```
<form>
  <h2>Validação de fone</h2>
 <input type="tel" required="required" maxlength="15"
name="telefone" pattern="[0-9]{2}-[0-9]{4,6}-[0-9]{3,4}$"
/>
<input type="submit" value="Enviar">
</form>
```

Uma das grandes vantagens da validação no lado cliente é que ela permite que erros sejam detectados instantaneamente, proporcionando *feedback* imediato ao usuário e melhorando a experiência de preenchimento. Outra vantagem é que, ao realizar validações no cliente, é possível reduzir a carga no servidor, uma vez que muitos erros podem ser identificados antes mesmo do envio dos dados.

Embora a validação no *front end* ofereça benefícios imediatos, como feedback instantâneo ao usuário e uma experiência mais interativa, também apresenta desvantagens que devem ser consideradas. Uma das desvantagens associadas à validação no *front end* é que a lógica de validação no *front end* é executada no navegador do cliente. Isso significa que o código responsável pela validação é visível para o usuário.

Se um usuário mal-intencionado souber como visualizar ou manipular o código-fonte da página *web*, ele pode identificar e entender as regras de validação. Isso não apenas compromete a segurança, mas também pode facilitar tentativas de manipulação ou envio de dados fraudulentos.

Outro problema é que, como a validação no *front end* ocorre no ambiente controlado pelo usuário, ela pode ser contornada. Usuários avançados ou mal-intencionados podem desativar ou modificar o código JavaScript que executa a validação. Isso permite que eles submetam dados que não atendem aos critérios de validação, comprometendo a integridade dos dados que chegam ao servidor.

Portanto, a validação no *front end* não deve ser a única linha de defesa para garantir a integridade dos dados. Ainda temos uma terceira desvantagem: o uso de recursos avançados de validação oferecidos pelo HTML5 pode ser

limitado em navegadores mais antigos que não suportam totalmente essas funcionalidades.

Embora a maioria dos navegadores modernos ofereça suporte ao HTML5, alguns usuários ainda podem estar utilizando versões desatualizadas. Isso pode resultar em uma experiência de usuário inconsistente, com validações não funcionando conforme o esperado para uma parcela dos usuários.

4.2 Validação *back end*

Uma estratégia mais segura é complementar a validação no *front end* com a validação no *back end*. Isso garante que, mesmo que a validação no *front end* seja burlada, os dados ainda serão validados no servidor antes de serem processados ou armazenados.

A validação no lado servidor, também conhecida como validação *back end*, desempenha um papel fundamental na garantia da integridade e segurança dos dados submetidos por usuários em formulários *web*. Cada linguagem de programação tem suas próprias bibliotecas e ferramentas para implementar a validação no servidor, proporcionando uma camada adicional de segurança e consistência nos dados processados.

A validação no lado servidor ocorre no ambiente do servidor *web* onde a aplicação está hospedada. Isso significa que, antes de qualquer processamento adicional, os dados enviados pelo cliente são submetidos a verificações rigorosas para garantir sua conformidade com as regras de validação previamente definidas. Isso reduz a probabilidade de dados incorretos ou maliciosos afetarem o funcionamento do sistema.

Em ambientes Java, duas ferramentas amplamente utilizadas para validação no lado servidor são o **Bean Validation** e o **Hibernate Validator**. O Bean Validation é uma especificação do Java que define um modelo de programação para validação de objetos. O Hibernate Validator, por sua vez, é uma implementação dessa especificação que oferece funcionalidades adicionais e personalizações. Ambos são recursos poderosos para garantir que objetos Java, incluindo aqueles mapeados para entidades de banco de dados, atendam aos critérios de validação necessários.

No ecossistema Spring, especificamente no framework Spring MVC, a validação é tratada através do **Spring Validator**. O Spring Validator permite que os desenvolvedores criem regras de validação personalizadas para seus objetos

de domínio. Integrado ao ciclo de vida do Spring MVC, esse mecanismo de validação é acionado automaticamente durante o processamento de formulários *web*, garantindo que apenas dados válidos sejam aceitos.

O Spring Validator é uma interface no Spring Framework, que define um contrato para validar objetos em um contexto específico, geralmente associado à validação de dados de entrada em formulários ou solicitações. Essa interface é parte do módulo org.springframework.validation e pode ser implementada pelos desenvolvedores para criar regras de validação personalizadas.

Temos ainda o termo **Spring Validation**, que é mais abrangente e refere-se ao conjunto de recursos no Spring Framework relacionados à validação de dados. Isso inclui o uso de anotações de validação, como @NotNull, @Size, @Pattern, entre outras, que são parte do módulo org.springframework.validation.annotation

Essas anotações são frequentemente usadas em conjunto com a validação baseada em anotações do Bean Validation. O **Spring Validation** é uma parte crucial do ecossistema *Spring* que oferece suporte à validação de dados em aplicações Java. Ele se baseia na **especificação de validação de objetos Java, conhecida como Bean Validation**. Essa especificação define um modelo de programação para validação de objetos, permitindo que os desenvolvedores apliquem regras de validação diretamente nas classes de domínio. As anotações da Bean Validation, como @NotNull, @Size, @Pattern, entre outras, são usadas para expressar as restrições que os campos de um objeto devem atender.

Exemplo de uso de anotações Bean Validation em uma classe Java:

```
public class Usuario {
    @NotNull
    @Size(min = 3, max = 50)
    private String nome;

    @Email
    private String email;

    // Getters e Setters
}
```

Para utilizar o Spring Validation, é necessário incluir a dependência apropriada em seu projeto. O Spring Framework fornece suporte integrado para o Bean Validation, mas a dependência específica pode variar com base no

sistema de construção de projeto que você está usando. Quando se está utilizando o Maven, pode adicionar a dependência da seguinte maneira:

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

Essa dependência inclui as bibliotecas necessárias para integrar o Bean Validation ao contexto do Spring.

O Spring Validation integra-se perfeitamente ao ciclo de vida de processamento de solicitações HTTP no Spring MVC. Quando um formulário é enviado, o Spring automaticamente aciona o processo de validação antes de encaminhar a solicitação ao controlador correspondente. As restrições definidas nas classes de domínio são verificadas, e os erros de validação são disponibilizados para os controladores.

A seguir, podemos ver um exemplo de um controlador Spring MVC utilizando validação (@Valid).

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

    @PostMapping
    public ResponseEntity<String> cadastrarUsuario(@Valid
@RequestBody Usuario usuario) {
        // Lógica para processar o usuário
        return ResponseEntity.ok("Usuário cadastrado com
sucesso!");
    }
}
```

A anotação @Valid informa ao Spring para realizar a validação do objeto Usuario. Se ocorrerem erros de validação, o Spring automaticamente os encapsula em um objeto BindingResult, que pode ser acessado para tratamento posterior.

Em resumo, o Spring Validation simplifica e padroniza a validação de dados em aplicações Java, aproveitando as capacidades da especificação Bean Validation. Adicionar a dependência apropriada ao projeto é essencial para utilizar esses recursos de validação no contexto do Spring. Isso permite que

desenvolvedores garantam a integridade dos dados, tornando as aplicações mais robustas e seguras.

Como **benefícios ou vantagens** da validação no lado servidor, temos a questão da **segurança adicional**, pois a validação no lado servidor fornece uma camada adicional de segurança, uma vez que os dados são validados no ambiente controlado do servidor antes de qualquer processamento adicional.

O segundo benefício diz respeito à **Consistência nos Dados:** ao centralizar as regras de validação no lado servidor, garante-se consistência nos dados, independentemente de como a aplicação é acessada (por exemplo, através de diferentes interfaces de usuário). E, por fim, podemos citar o **reaproveitamento de lógica:** as regras de validação no lado servidor podem ser reutilizadas em diferentes partes da aplicação, promovendo uma abordagem DRY (*Don't Repeat Yourself*) e facilitando a manutenção do código.

A validação no lado servidor é uma prática essencial para garantir a confiabilidade e a segurança dos dados recebidos em aplicações *web*. Com bibliotecas específicas para cada linguagem e *frameworks* como Bean Validation, Hibernate Validator e Spring Validator, os desenvolvedores têm à disposição ferramentas poderosas para implementar verificações robustas e personalizadas no ambiente do servidor.

Uma desvantagem significativa da validação no *back end*, especialmente em sistemas na nuvem, é o custo associado às requisições GET, PUT ou POST. Em ambientes de computação em nuvem, os provedores geralmente cobram pelos recursos consumidos, incluindo operações de leitura e gravação no banco de dados. Quando se realiza a validação no *back end*, que, muitas vezes, envolve a interação com bancos de dados ou outros serviços, cada requisição pode gerar custos adicionais.

TEMA 5 – TESTE UNITÁRIO COM JUNIT

O **JUnit** é um dos *frameworks* mais conhecidos para a execução de testes unitários em Java, desempenhando um papel crucial na promoção da qualidade e confiabilidade do código. Vamos explorar alguns pontos importantes sobre o JUnit, destacando sua simplicidade de uso e a capacidade de proporcionar um ambiente completo para a realização de testes unitários e de regressão.

O JUnit é um *framework* amplamente utilizado na comunidade Java, oferecendo uma estrutura robusta e padronizada para a criação e execução de

testes unitários. Sua popularidade decorre da simplicidade de uso, da integração eficiente com ferramentas de desenvolvimento e do suporte a práticas ágeis de desenvolvimento, como o TDD (*Test-Driven Development*). O JUnit foi projetado com a simplicidade em mente.

Sua sintaxe é clara e direta, permitindo que desenvolvedores criem testes facilmente compreensíveis. A anotação @Test é fundamental, indicando que um método específico é um caso de teste. Além disso, as assertivas oferecidas pelo JUnit, como assertEquals e assertTrue, simplificam a verificação dos resultados esperados.

O código a seguir mostra um exemplo de um teste unitário simples com JUnit:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MinhaClasseTest {

    @Test
    public void testarMetodo() {
        MinhaClasse minhaInstancia = new MinhaClasse();
        assertEquals(6, minhaInstancia.multiplicar(2, 3));
    }
}
```

A linha de código `assertEquals(5, minhaInstancia.multiplicar(2, 3));` chama o método multiplicar da instância da `MinhaClasse` com os números 2 e 3 e verifica se o resultado é igual a 6, usando o método `assertEquals`. Se o resultado da multiplicação for 6, o teste passa; caso contrário, ele falha, indicando que há um problema no método multiplicar da classe MinhaClasse.

O JUnit não apenas facilita a criação de testes unitários, mas também oferece um ambiente completo para a realização desses testes, incluindo suporte a testes de regressão. A integração com ferramentas de compilação como o Maven ou o Gradle permite a execução automatizada dos testes durante o ciclo de desenvolvimento.

Os testes unitários realizados por meio do JUnit têm como principal objetivo garantir que cada método projetado funcione conforme esperado. Eles analisam as menores partes do código, verificando se as funcionalidades específicas produzem os resultados desejados. Esse processo de verificação

granular contribui para a identificação precoce de erros e auxilia no isolamento de problemas, facilitando a manutenção do código ao longo do tempo.

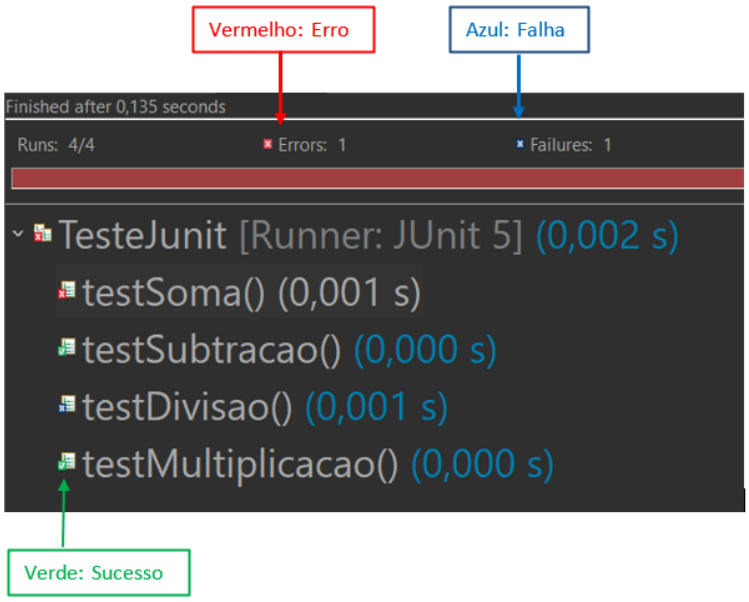
O Junit é dividido em duas partes: uma com métodos e anotações que servem para a elaboração dos casos de teste e uma outra parte que serve para realizar a execução dos casos de teste criados pela primeira parte.

Uma das características mais marcantes do JUnit é a execução automatizada dos testes. Isso significa que, uma vez que os testes são escritos e configurados, a execução pode ocorrer sem intervenção humana. Esse aspecto é crucial para a integração contínua e o desenvolvimento ágil, onde testes frequentes são necessários para garantir a estabilidade do código.

O JUnit utiliza um esquema de cores semelhante a um semáforo para indicar o resultado dos testes após a execução.

- **Verde:** indica que o teste foi executado com sucesso, sem qualquer falha ou erro. Isso significa que o comportamento observado está de acordo com o esperado no caso de teste.
- **Azul:** representa um *status* intermediário. No contexto do JUnit, o azul pode indicar que o teste foi ignorado ou que houve alguma falha de validação, como uma condição esperada que não foi atendida.
- **Vermelho:** indica que houve uma falha crítica durante a execução do teste. Isso pode ocorrer quando há exceções não tratadas ou outros erros fundamentais no código em Java testado.

Figura 4 – Exemplo de estrutura básica de teste JUnit



O JUnit é frequentemente integrado a ambientes de desenvolvimento integrados (IDEs), como o Eclipse, IntelliJ IDEA e outras. Isso simplifica ainda mais o processo de escrita, execução e interpretação dos resultados dos testes.

Algumas precauções podem ser tomadas durante o planejamento dos testes, desde a definição de um pacote em que fiquem armazenadas todos os casos de teste até a organização dos testes de forma estruturada, separando os testes unitários dos testes de integração. Os testes unitários devem se concentrar em unidades isoladas de código, enquanto os testes de integração devem validar a interação entre componentes. É importantíssimo identificar e compreender os casos de teste que sejam mais relevantes para o sistema.

FINALIZANDO

Encerramos com uma jornada abrangente pelo universo dos testes de *software*. Desde os fundamentos do teste de *software* até técnicas específicas, exploramos os testes funcionais com o Postman, destacando a importância de garantir o desempenho e a confiabilidade das APIs. Adentramos no reino das mensagens ao usuário com Thymeleaf, descobrindo como personalizar e tornar mais dinâmica a interação com os usuários em aplicações *web*. Não esquecemos a crítica validação, tanto no *back end* quanto no *front end*, ressaltando a necessidade de assegurar a integridade dos dados e proporcionar uma experiência fluida.

Finalmente, concluímos com a relevância dos testes unitários usando JUnit, uma prática fundamental para manter a estabilidade do código. Que esses conhecimentos sejam a base para o aprimoramento contínuo de nossas habilidades na criação de *software* confiável e de alta qualidade. Estamos prontos para aplicar esses aprendizados em nossos projetos futuros.

REFERÊNCIAS

BITTENCOURT, G.; TESSMANN, P. V.; SCHIRMER, T. **Teste de software**: uma introdução ao uso do JUnit em testes unitários. 2019. Disponível em: <https://www.academia.edu/17303382/Teste_de_Software_Uma_introdu%C3%A7%C3%A3o_ao_uso_do_Ju-nit_em_testes_unit%C3%A1rios?auto=download>. Acesso em: 30 abr. 2024.

GONÇALVEZ, P. F. et al. **Testes de software e gerência de configuração**. Grupo A, 2019.

MDN. **Tecnologia Web para desenvolvedores >** HTML: Linguagem de Marcação de Hipertexto> Atributo type. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML/Element/Input>. Acesso em: 30 abr. 2024.

MDN. **HTML attribute**: pattern. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/pattern>. Acesso em: 30 abr. 2024.

PATTON, R. **Software Testing**. 2. ed. Indianápolis: Sams Publishing, 2005.

POSTMAN. **What is Postman?**. Disponível em: <https://www.postman.com/product/what-is-postman/>. Acesso em: 30 abr. 2024.

PRESSMAN, R. S.; BRUCE, R. M. **Engenharia de software**. 9th. ed. [S.l.]: Grupo A, 2021.

THYMELEAF. **Tutorial**: Thymeleaf + Spring. Disponível em: <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#the-concept>. Acesso em: 30 abr. 2024.