

# Tópicos em Qualidade e Testes de Software

**Contribuições e debates  
promovidos na FATEC Araras**

**Edição primeiro semestre - 2024**

**NILTON CESAR SACCO (organizador)**  
**ORLANDO SARAIVA JÚNIOR (organizador)**

**Paulo Cesar Petruz Junior**  
**Klayvert Ryan Alves**  
**Paula Nascimento Silva de Melo**  
**Denis Lopes Ferreira dos Santos**  
**Diego Eduardo Wiltler**  
**Marius Jorge Pessi**  
**Jair Lopes Junior**  
**Antonio Luis Pereira Candioto**  
**Gleison Rodrigo Moura da Silva**  
**André Felipe de Paula**  
**Joice Fernanda Rodrigues**

**ISBN: 978-65-01-07165-7**

# O curso Desenvolvimento em Software Multiplataforma

Caro leitor,

Este livro é diferente dos que você costuma ler. Ele representa a entrega e a evidência do trabalho realizado por nós, da Fatec Araras, durante o primeiro semestre de 2024, especificamente na disciplina "Qualidade e Testes de Software".

Para contextualizar, é importante explicar o curso "Desenvolvimento em Software Multiplataforma". De acordo com o projeto pedagógico, o curso visa formar profissionais habilitados a desenvolver software para diversas plataformas, como Web, Desktop, Mobile, em Nuvem e Internet das Coisas, utilizando conceitos de Segurança da Informação e Inteligência Artificial. Além disso, o curso tem como objetivo especializar profissionais para trabalhar com metodologias ágeis de gestão de projetos, versionamento, integração e entrega contínua de software, visando desenvolver soluções que atendam aos critérios de qualidade exigidos pelo mercado.

Nossos alunos são incentivados a se familiarizar e aplicar as tecnologias mais modernas. Por exemplo, já no primeiro semestre, cada aluno cria seu próprio portfólio na plataforma Github. Além disso, a entrega dos trabalhos em grupo ocorre em um repositório onde todos são incentivados a realizar *pull requests*.

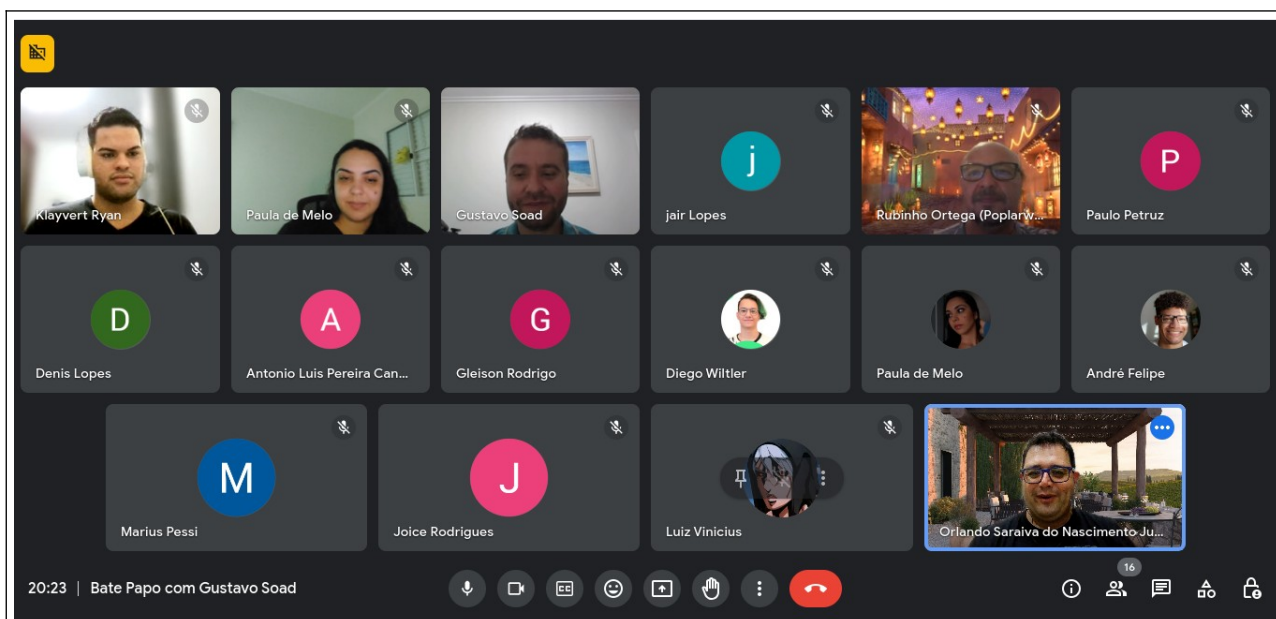
Entre as várias disciplinas que compõem a formação deste profissional qualificado, destaca-se a disciplina "Qualidade e Testes de Software".

## A disciplina Qualidade e Testes de Software

A disciplina Qualidade e Testes de Software é ministrada no último semestre de forma remota. A missão do docente desta disciplina é empregar metodologias que visem garantir critérios de qualidade no desenvolvimento de uma solução computacional.

E nesta segunda edição, somando os aprendizados e aos *feedbacks* dos alunos que cursaram essa disciplina no segundo semestre de 2023 e as ideias recebidas de profissionais da área e docentes de outras Fatecs e Etecs, acredito que conseguimos avançar em tornar essa disciplina ainda mais atrativa aos nossos alunos.

Uma das práticas realizadas, com a vênua da coordenação, foi convidar um especialista externo para um bate papo sobre Qualidade e Testes de Software. Gostaríamos de expressar nosso sincero agradecimento ao convidado externo, Gustavo Soad, que participou de uma de nossas aulas via Meet. Sua generosidade em compartilhar sua vasta experiência na área de testes foi de grande valor para todos nós.



**Figura 1** : Print do encontro virtual com o especialista *Gustavo Soad*

Outra prática pedagógica que realizamos foi a organização de sessões de Coding Dojo de forma remota, integrando ativamente os alunos na prática colaborativa de programação e resolução de problemas reais dentro da disciplina.

Apresentamos um projeto com o framework Django com testes unitários, [[https://github.com/orlandosaraivajr/Pratica\\_TDD\\_2](https://github.com/orlandosaraivajr/Pratica_TDD_2) ] e dentro da aula expositiva, realizamos alterações para atender a novos requisitos. Nestas aulas, tivemos a oportunidade de aplicar o desenvolvimento orientado à testes (TDD).

Também apresentamos e exploramos a ferramenta Pytest. E a partir daqui, cada um teve tempo para pesquisar, estudar e apresentar uma ferramenta alinhada ao seu interesse profissional. A experiência foi incrível, e o resultado do *workshop* promovido dentro da disciplina tornou-se este livro.

Cada capítulo é independente do capítulo anterior e do capítulo posterior. Você pode ler individualmente, para conhecer a perspectiva do aluno ao tema estudado e apresentado dentro dos seminários da disciplina. Foram várias as experiências e aprendizados nesta jornada. E o conteúdo apresentado nas próximas páginas pelos próprios alunos é uma amostra de que estamos conseguindo realizar nossa missão como docentes do Centro Paula Souza: Promover a educação pública profissional e tecnológica dentro de referenciais de excelência, visando o desenvolvimento tecnológico, econômico e social do Estado de São Paulo.

Orlando Saraiva do Nascimento Júnior  
Docente da disciplina

Nilton Cesar Sacco  
Coordenador do Curso

# Capítulo 01

# Capítulo 01

Paulo Cesar Petruz Junior  
paulo.petrusz@fatec.sp.gov.br

*"Testes de software podem ser um processo doloroso, mas não testar pode ser fatal."*

*Boris Beizer*

## Introdução

Neste capítulo, falaremos sobre os impactos na ausência de teste de software, que podem trazer consequências desastrosas para qualquer organização que desenvolva softwares, os testes trazem a garantia de que funcione corretamente e ajudam a identificar e corrigir os erros antes de chegar ao usuário final.

## Conceitos básicos sobre a utilização de testes

Temos uma categorização de testes por tipos, onde cada um será realizado em um momento diferente do projeto com o objetivo de abranger do início ao fim do desenvolvimento. Alguns dos testes mais famosos são:

**Testes unitários:** realiza uma verificação sobre cada componente desenvolvido individualmente no código, como funções, métodos ou classes. Esse tipo de teste tem como objetivo testar cada componente de forma isolada, para que facilite a detecção de erros no desenvolvimento dos estágios iniciais do projeto.

**Testes de Integração:** faz uma comparação entre módulos ou componente do sistema com o objetivo de garantir que funcionem corretamente quando em conjuntos e identificando problemas na interface entre os módulos.

**Testes de Sistema:** busca testar o sistema como um todo garantindo que funcione corretamente os componentes em conjunto conforme esperado. Neste tipo de teste consegue assegurar que os requisitos funcionais e não funcionais sejam atendidos.

**Testes de Aceitação:** garantem que o software cumpre todos os requisitos do usuário e está pronto para ser entregue atendendo as expectativas dos usuários.

**Testes de Regressão:** em caso de novas mudanças ou adições ao código esse tipo de teste assegura que não será introduzido novos bugs, assim mantendo a estabilidade do sistema ao garantir que não terá impacto nas funcionalidades já existente.

**Testes de Estresse:** Coloca o sistema para atuar em situações extremas assim conseguindo determinar os seus limites. Durante esse tipo de teste podemos avaliar a robustez e a capacidade de recuperação do sistema sob carga extrema.

**Testes de Segurança:** identifica as possíveis vulnerabilidades e garantem que o sistema está protegido contra-ataques protegendo os dados sensíveis.

# Importância da utilização dos testes junto ao desenvolvimento

Uma pergunta frequente de estudantes e profissionais recém-inseridos na área de programação é: por que devo realizar o desenvolvimento dos testes junto com o desenvolvimento do meu projeto? A resposta simples é que desenvolver testes em paralelo com o código evita retrabalho e aumenta a eficiência tanto para a empresa quanto para o desenvolvedor. Quando os testes são criados simultaneamente ao código, é mais fácil detectar e corrigir erros nas fases iniciais do desenvolvimento, economizando tempo e recursos a longo prazo.

Com a utilização do *Test-Drive Development* conseguimos uma gama de benefícios, sendo eles, prevenção de retrabalho, qualidade, confiança e facilidade de manutenção, gostaria de ressaltar sobre a qualidade e confiança, o TDD traz o incentivo para a escrita de código de qualidade desde o início do desenvolvimento, desta forma, cada funcionalidade é testada quando é desenvolvida resultando em um software mais robusto e confiável.

## Síntese

Os testes de software são fundamentais para garantir que o produto tenha qualidade, segurança e seja totalmente funcional. Eles permitem identificar e corrigir erros em várias etapas do desenvolvimento, assegurando que o software atenda às expectativas dos usuários e requisitos especificados. Diferentes tipos de testes, como unitários, de integração, de sistema, de aceitação, de regressão, de estresse e de segurança, desempenham papéis complementares, cobrindo desde a verificação de componentes individuais até a avaliação do sistema como um todo sob condições extremas.

## Conclusão

A ausência de testes de software pode resultar em problemas graves, como falhas críticas, vulnerabilidades de segurança e insatisfação do usuário final. Integrar os testes ao processo de desenvolvimento é uma prática essencial que não apenas previne retrabalho, mas também promove a confiança e a qualidade do software entregue. Adotar metodologias como o Test-Driven Development (TDD) pode ser particularmente eficaz, garantindo que cada funcionalidade seja rigorosamente testada e que o código desenvolvido seja robusto e sustentável.

Em resumo, investir em testes de software é investir na saúde e no sucesso do projeto. Organizações que priorizam a qualidade e a confiabilidade em seus processos de desenvolvimento certamente colherão os frutos de um software eficiente, seguro e alinhado às necessidades dos usuários.

## Bibliografia

[1]<https://www.linkedin.com/pulse/falhas-famosas-de-programação-que-entraram-para-história-pacheco/> Acesso em 27 mai. 2024.

[2]<https://www.alura.com.br/artigos/tipos-de-testes-principais-por-que-utiliza-los> Acesso em 11/06/2024

# Capítulo 02

# Capítulo 02

Klayvert Ryan Alves  
klayvert.ryan16@gmail.com

## Java Code Coverage(JaCoCo)

Este capítulo tem como objetivo explorar em detalhes a integração do TDD com o JaCoCo(Java Code Coverage), uma ferramenta de cobertura de código amplamente utilizada na linguagem Java, e verificar como a utilização e integração entre esses dois conceitos podem ser benéficos no desenvolvimento de software.

Veremos que a combinação destas metodologias não apenas eleva a qualidade do código, mas também aprimora a eficiência do processo de desenvolvimento, oferecendo aos desenvolvedores uma abordagem estruturada e sistemática para a criação de software de alta qualidade.

### Objetivo e Importância

A cobertura de código é uma métrica que indica a extensão do código-fonte que é executado durante a execução dos testes. O objetivo do JaCoCo é fornecer insights sobre quais partes do código estão sendo testadas e quais não estão, ajudando os desenvolvedores a identificar áreas que necessitam de mais testes e, assim, melhorar a qualidade do software.

### Funcionamento

JaCoCo funciona instrumentando o bytecode do aplicativo Java. Quando os testes são executados, o JaCoCo coleta informações sobre quais partes do código foram executadas. Essas informações são então usadas para gerar relatórios de cobertura.

### Componentes Principais

**Agente JaCoCo:** Um agente Java que é anexado à JVM (Java Virtual Machine) para instrumentar o bytecode. Este agente coleta dados de execução enquanto o código é executado.

**Ant Task e Maven Plugin:** Integrações com ferramentas de construção como Apache Ant e Apache Maven, permitindo a fácil execução do JaCoCo como parte do processo de build.

**Relatórios:** JaCoCo gera relatórios em vários formatos (HTML, XML, CSV), proporcionando uma visualização detalhada da cobertura do código.

### Tipos de Cobertura

- 1- Cobertura de linha: Percentual de linhas de código que foram executadas.
- 2- Cobertura de ramo: Percentual de branches (ramificações de decisão, como if-else) que foram executadas.
- 3- Cobertura de método: Percentual de métodos que foram executados.
- 4- Cobertura de classe: Percentual de classes que foram executadas.
- 5- Integração com Ferramentas de Build e CI/CD



JaCoCo pode ser facilmente integrado em sistemas de build e pipelines de CI/CD (Continuous Integration/Continuous Deployment). Ferramentas populares como Jenkins, GitLab CI, e Travis CI suportam JaCoCo, facilitando a automação da coleta de cobertura de código como parte do fluxo de desenvolvimento contínuo.

## Benefícios

**Melhoria da Qualidade do Código:** Ao identificar áreas não cobertas pelos testes, os desenvolvedores podem adicionar testes adicionais, resultando em um código mais robusto e menos propenso a bugs.

**Feedback Imediato:** Integrações com CI/CD permitem que os desenvolvedores recebam feedback imediato sobre a cobertura de código após cada build.

**Decisões Informadas:** Dados de cobertura ajudam na tomada de decisões sobre onde focar esforços de teste e refatoração.

## JaCoCo na Prática Adicionando as Dependências

No exemplo a seguir será mostrado como adicionar as dependências da biblioteca JaCoCo utilizando maven(um dos gerenciadores de pacotes mais utilizados em Java):

```
<dependencies>
  <dependency>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.11</version>
  </dependency>
</dependencies>
```

Para melhor performance da biblioteca, podemos adicionar também seu plugin da seguinte maneira:

```

<plugins>
  <plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.11</version>
    <executions>
      <execution>
        <id>prepare-agent</id>
        <goals>
          <goal>prepare-agent</goal>
        </goals>
      </execution>
      <execution>
        <id>report</id>
        <phase>test</phase>
        <goals>
          <goal>report</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>

```

Desta forma, ao executar os testes com sucesso utilizando os comandos “mvn clean test” o próprio JaCoCo irá se encarregar de escanear todo o projeto e nos informar a cobertura dos testes em comparação aos nossos modelos de objetos, nossas classes e dos métodos como mostra o exemplo abaixo:

Exemplo do relatório em formato HTML gerado pela biblioteca.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.pipa.PipaAPI.rest.dto	<div><div></div></div>	26%	<div><div></div></div>	5%	354	469	85	260	84	199	2	9
com.pipa.PipaAPI.domain.entity	<div><div></div></div>	35%	<div><div></div></div>	27%	360	490	25	93	125	245	6	14
com.pipa.PipaAPI.domain.models.auth	<div><div></div></div>	13%	<div><div></div></div>	0%	86	102	9	21	41	57	2	6
com.pipa.PipaAPI.service	<div><div></div></div>	61%	<div><div></div></div>	34%	40	81	92	245	31	68	2	8
com.pipa.PipaAPI.configuration	<div><div></div></div>	0%	<div><div></div></div>	0%	26	26	65	65	18	18	5	5
com.pipa.PipaAPI.rest.controller	<div><div></div></div>	0%	<div><div></div></div>	n/a	36	36	56	56	36	36	7	7
com.pipa.PipaAPI.domain.enums	<div><div></div></div>	91%	<div><div></div></div>	n/a	3	9	3	24	3	9	0	3
com.pipa.PipaAPI	<div><div></div></div>	0%	<div><div></div></div>	n/a	2	2	3	3	2	2	1	1
com.pipa.PipaAPI.utils.data	<div><div></div></div>	72%	<div><div></div></div>	n/a	1	3	1	3	1	3	0	1
Total	6.175 of 9.146	32%	990 of 1.162	14%	908	1.218	339	770	341	637	25	54

Podemos analisar na imagem acima alguns detalhes interessantes como:

- 1- Element: os packages do nosso projeto
- 2- Missed Instructions: gráfico de barra mostrando quanto dos métodos de cada package tiveram suas instruções devidamente cobertas.
- 3- Cov.: A porcentagem de coverage(cobertura) do código de cada package.
- 4- Methods: A quantidade de métodos do package.
- 5- Classes: A quantidade de classes do package.
- 6- Missed: A quantidade de métodos faltantes para teste.

Para ver os detalhes de um método específico (neste caso o “findByUsername”), podemos seguir o caminho dos packages até o método em questão:

Abaixo está o nosso método que busca usuários pelo seu nome:

```

public User findByUsername(String username) {
    return this.repository.findByUsername(username)
        .orElseThrow(() -> new RuntimeException(HttpStatus.NOT_FOUND, "The user with username '%s' not found".formatted(username)));
}

```

Em seguida podemos ver o nosso teste, então de forma resumida ele cria um mock de um usuário com o nome “teste”, ou seja, ele faz um usuário que só existirá no momento do teste. Após isso ele chama o método e valida se o retorno é um valor não nulo.

```
@Test
public void findByUsername() {
    String username = "teste";

    when(this.userRepository.findByUsername(username)).thenReturn(Optional.of(new User()));

    User result = this.authenticationService.findByUsername(username);

    assertNotNull(result);
}
```

Agora partimos para o JaCoCo, onde ele irá nos mostrar a cobertura deste método com o teste criado.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
• lambda\$authenticate\$0(AuthenticationRequest)	<div><div></div></div>	0%	n/a		1 1	1 1	1 1
• authenticate(AuthenticationRequest)	<div><div></div></div>	83%	n/a		0 1	2 14	0 1
• register(RegisterRequest)	<div><div></div></div>	100%	n/a		0 1	0 14	0 1
• AuthenticationService(UserRepository, UserService, PasswordEncoder, JwtService, AuthenticationManager)	<div><div></div></div>	100%	n/a		0 1	0 7	0 1
• lambda\$findByUsername\$1(String)	<div><div></div></div>	100%	n/a		0 1	0 1	0 1
• findByUsername(String)	<div><div></div></div>	100%	n/a		0 1	0 2	0 1
Total	22 of 147	85%	0 of 0	n/a	1 6	2 37	1 6

E clicando sobre ele podemos ver nosso código original deste método e tracejado de verde a área que nosso teste cobriu.

```
75.
76. public User findByUsername(String username) {
77.     return this.repository.findByUsername(username)
78.         .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "The user with username '%s' not found".formatted(username)));
79. }
```

## Conclusão

JaCoCo é uma ferramenta poderosa e essencial para qualquer desenvolvedor Java que deseja garantir alta qualidade e confiabilidade do código através de testes eficazes. Ele oferece uma visão detalhada sobre a cobertura de código, permitindo melhorias contínuas e sustentadas no desenvolvimento de software.

## Bibliografia

- [1] - Github da biblioteca. <https://github.com/jacoco/jacoco>
- [2] - JaCoCo Java Code Coverage Library <https://www.eclemma.org/jacoco/>
- [3] - JaCoCo Documentation <https://www.jacoco.org/jacoco/trunk/doc/>
- [4] - JaCoCo Maven Repository <https://mvnrepository.com/artifact/org.jacoco/jacoco-maven-plugin>
- [5] - JUnit 5 User Guide <https://junit.org/junit5/docs/current/user-guide/>

# Capítulo 03

# Capítulo 03

Paula Nascimento Silva de Melo  
paulademelo.ti@gmail.com

## Testes Instrumentados com Espresso no Android

### O que são Testes instrumentados?

Testes instrumentados são testes automatizados que são executados em um ambiente real ou simulado para verificar a funcionalidade do aplicativo. Eles são essenciais para garantir a qualidade e a estabilidade, identificando possíveis problemas de integração e comportamento do usuário. Os testes instrumentados também permitem simular interações do usuário, como toques na tela, cliques em botões e preenchimento de formulários, proporcionando uma visão abrangente do desempenho do aplicativo em diferentes cenários.

Diferenças entre testes instrumentados e testes de unidade:

**Abrangência:** testes instrumentados testam a aplicação como um todo, enquanto testes de unidade focam em componentes isolados.

**Acesso:** testes instrumentados acessam recursos do dispositivo e do framework Android, enquanto testes de unidade não.

**Simulação:** testes instrumentados simulam interações do usuário, enquanto testes de unidade não.

**Pensando em Analogias:** Imagine uma casa em construção, onde cada parte representa um aspecto do desenvolvimento de um aplicativo Android. Nos testes instrumentados, estamos realizando uma inspeção completa da casa em construção para garantir que tudo esteja sendo construído conforme o planejado, desde a fundação até o telhado. Estamos verificando se as paredes estão no lugar certo, se os encanamentos estão funcionando corretamente e se as conexões elétricas estão seguras. Em outras palavras, estamos examinando a casa como um todo para garantir sua qualidade e estabilidade. Por outro lado, nos testes de unidade, estamos nos concentrando em componentes individuais da casa, como portas, janelas e tomadas elétricas. Estamos verificando se cada componente desempenha sua função corretamente: se as portas abrem e fecham sem problemas, se as janelas fornecem ventilação adequada e se as tomadas elétricas estão funcionando conforme o esperado. Em resumo, enquanto nos testes instrumentados estamos preocupados com a casa como um todo, nos testes de unidade estamos nos certificando de que cada parte funcione como deveria.

## Benefícios e importância dos testes instrumentados

Os testes instrumentados oferecem diversos benefícios, incluindo a detecção precoce de problemas de integração, validação do comportamento do usuário, garantia da estabilidade do aplicativo e suporte a um processo de desenvolvimento ágil. Além disso, eles ajudam a reduzir o tempo gasto com testes manuais e aumentam a confiança na qualidade do aplicativo. A importância dos testes instrumentados é evidente na melhoria da experiência do usuário, na identificação e correção de falhas e na garantia da funcionalidade adequada do aplicativo em diferentes dispositivos e situações.

## Principais funcionalidades e vantagens

**Simulação de interações do usuário:** clique em botões, digite em campos de texto, navegue em menus, etc.

**Verificação de estados:** verifique o estado da UI, textos, valores de variáveis, etc.

**Espera por eventos:** aguarde eventos como animações, carregamentos de dados, etc.

**Captura de screenshots:** capture imagens da tela para análise visual.

**Suporte a diferentes dispositivos e versões do Android:** teste em diversos dispositivos e versões do Android.

**Integração com outras ferramentas de teste:** combine o Espresso com outras ferramentas para testes mais abrangentes.

## Estrutura básica dos testes instrumentados no Android

Para compreender a estrutura que será apresentada adiante, é recomendável ter algum conhecimento prévio em linguagem de programação. Os exemplos serão escritos em Java, porém, mesmo se não estiver familiarizado com esta linguagem, não será um impedimento para continuar a leitura. Ao longo do texto, serão fornecidas explicações detalhadas sobre os principais métodos do Espresso no Android.

Antes de explorarmos a estrutura básica dos testes instrumentados no Android, é importante mencionar que o Espresso já está incluído nas dependências do Android no Gradle. Portanto, não é necessário realizar nenhuma configuração adicional para começar a utilizar o Espresso em seus projetos. Agora, vamos explorar a estrutura do teste instrumentado com Espresso.

### Exemplo prático 1

**Cenário:** Realizar o login no aplicativo.

**Objetivo do Teste:** O objetivo principal do teste é verificar se o fluxo de login funciona corretamente, garantindo que ao inserir as credenciais válidas e clicar no botão de login, o usuário seja direcionado para a tela principal e a mensagem de boas-vindas seja exibida corretamente.

O método de teste instrumentado com espresso segue a seguinte estrutura:

```
@Test
public void testLoginSuccess() {
    // Arrange
    onView(withId(R.id.usernameEditText)).perform(typeText("usuario"));
    onView(withId(R.id.passwordEditText)).perform(typeText("senha"));

    // Action
    onView(withId(R.id.loginButton)).perform(click());

    // Assert
    onView(withId(R.id.mainActivityTextView)).check(matches(withText("Bem-vindo!")));
}
```

### **Anotação @Test:**

A anotação @Test indica que o método testLoginSuccess() é um teste unitário. Ela é essencial para que o Espresso reconheça e execute o método como parte do conjunto de testes.

### **Arrange: preparando o cenário**

Nesta etapa, simulamos o estado inicial da aplicação antes da ação a ser testada. No exemplo, utilizamos o método onView() para localizar dois campos de texto (usernameEditText e passwordEditText) e inserir os valores "usuario" e "senha", respectivamente.

### **Action: executando a ação**

Simulamos a interação do usuário com a interface gráfica. No exemplo, utilizamos o método onView() para localizar o botão "Login" (loginButton) e simular um clique nele.

### **Assert: verificando o resultado esperado**

Verificamos se o resultado obtido após a execução da ação corresponde ao esperado. No exemplo, utilizamos o método onView() para localizar um TextView (mainActivityTextView) e verificar se o texto exibido nele é "Bem-vindo!".

### **Exemplo prático 2**

No universo dos testes instrumentados com Espresso, os métodos perform e matches são ferramentas essenciais para simular interações do usuário e verificar o resultado esperado. vamos explorar outro exemplo prático da aplicação desses métodos.

**Cenário:** Imagine um aplicativo de compras online onde o usuário pode adicionar itens ao carrinho e visualizar o total da compra.

**Objetivo do Teste:** Verificar se o total da compra é atualizado corretamente após a adição de um item ao carrinho.

```
@Test
public void testCartTotalUpdate() {
    // Arrange
    onView(withId(R.id.productNameEditText)).perform(typeText("Camisa"));
    onView(withId(R.id.productPriceEditText)).perform(typeText("50.00"));
    onView(withId(R.id.addButton)).perform(click());

    // Act
    onView(withId(R.id.cartButton)).perform(click());

    // Assert
    onView(withId(R.id.cartTotalTextView)).check(matches(withText("50.00")));
}
```

### **Arrange (preparando o cenário):**

`onView(withId(R.id.productNameEditText)).perform(typeText("Camisa"))` : Insere o texto "Camisa" no campo de nome do produto.

`onView(withId(R.id.productPriceEditText)).perform(typeText("50.00"))` : Insere o valor "50.00" no campo de preço do produto.

`onView(withId(R.id.addButton)).perform(click())` : Simula um clique no botão "Adicionar".

### **Action (executando a ação):**

`onView(withId(R.id.cartButton)).perform(click())` : Simula um clique no botão "Carrinho".

### **Assert (verificando o resultado esperado):**

`onView(withId(R.id.cartTotalTextView)).check(matches(withText("50.00")))` : Verifica se o texto exibido no TextView "cartTotalTextView" é "50.00", confirmando que o total da compra foi atualizado corretamente.

### **Observações:**

Neste exemplo, utilizamos o método `typeText()` para inserir valores em campos de edição e o método `click()` para simular cliques em botões. O método `matches(withText("50.00"))` verifica se o texto exibido no TextView corresponde exatamente à string "50.00". Você pode adaptar este exemplo para testar diferentes cenários, como adicionar vários itens ao carrinho ou verificar o total da compra com descontos.

### **Desvendando o método perform**

O método `perform` é a porta de entrada para a simulação de ações do usuário na interface gráfica do seu aplicativo. Ele permite que você execute diversas tarefas, como:

#### **Clicar em botões:**

`onView(withId(R.id.loginButton)).perform(click());`

#### **Inserir texto em campos de edição:**

`onView(withId(R.id.usernameEditText)).perform(typeText("usuario"));`



### **Selecionar opções em menus:**

```
onView(withId(R.id.spinner)).perform(new OnItemSelectedListener() { ... });
```

### **Realizar gestos na tela:**

```
onView(withId(R.id.imageView)).perform(swipeLeft());
```

### **Desvendando o método matcher**

O método `matches` é o guardião do resultado esperado do seu teste. Ele permite que você verifique se o estado da interface gráfica após a execução da ação corresponde ao que você espera. As principais aplicações do `matches` são:

#### **Verificar o texto exibido em um `TextView`:**

```
onView(withId(R.id.textView)).check(matches(withText("Bem-vindo!")));
```

#### **Verificar se um elemento está presente na tela:**

```
onView(withId(R.id.button)).check(matches(isDisplayed()));
```

#### **Validar o valor de um atributo de um elemento:**

```
onView(withId(R.id.checkBox)).check(matches(isChecked()));
```

## **Dicas e Boas Práticas para Escrever Testes Instrumentados com Espresso**

Ao escrever testes instrumentados com Espresso, algumas boas práticas podem melhorar a eficácia e a manutenibilidade dos seus testes. Aqui estão algumas dicas úteis:

Nomeie seus testes de forma descritiva e significativa, para que fique claro qual aspecto do aplicativo está sendo testado em cada caso.

Mantenha seus testes curtos e focados em um único comportamento ou funcionalidade. Isso torna os testes mais fáceis de entender e depurar.

Evite a dependência de dados externos ou do estado do aplicativo para garantir que seus testes sejam consistentes e independentes uns dos outros.

Use o recurso de espera explícita (`Espresso.onView().perform().check()`) para garantir que as ações do usuário tenham sido concluídas antes de verificar o resultado. Isso ajuda a evitar flutuações nos testes devido a atrasos na execução.

Regularmente revise e atualize seus testes à medida que o código do aplicativo é modificado ou atualizado. Testes desatualizados podem levar a resultados inconsistentes e podem não refletir mais o comportamento esperado do aplicativo.

Seguir estas boas práticas ajudará a manter seus testes instrumentados organizados, confiáveis e eficientes ao longo do ciclo de vida do seu projeto.

## Referências

Documentação oficial:

<https://developer.android.com/training/testing/espresso?hl=pt-br>

Exemplo básico Espresso:

<https://github.com/android/testing-samples/tree/main/ui/espresso/BasicSample>

Artigo sobre o Espresso:

<https://medium.com/android-dev-br/testes-instrumentados-tipos-de-teste-e-como-s%C3%A3o-executados-1f13a8cfe4c4>

Artigo sobre o Espresso:

<https://www.vogella.com/tutorials/AndroidTesting/article.html>

# Capítulo 04

# Capítulo 04

**Denis Lopes Ferreira dos Santos**  
**denis.santos31@fatec.sp.gov.br**

*"Por incrível que possa parecer, é mais importante o tempo que você passa estudando sozinho do que aquele que passa assistindo às aulas"*

*Prof. Pier*

## Introdução

O Pytest teve suas origens em 2003 e 2004 onde teve uma grande contribuição de Holger Krekel, com o surgimento de frameworks de testes como utest e da biblioteca std. A primeira versão estável do Pytest, o Pytest 1.0, foi lançada em 2009. Desde então, a biblioteca se tornou cada vez mais popular entre a comunidade Python, ganhando reconhecimento por sua simplicidade e efetividade.

Atualmente, o Pytest é uma das bibliotecas de teste mais utilizadas para Python, com uma comunidade ativa e em constante crescimento. Sua popularidade se deve à sua facilidade de uso, flexibilidade e à ampla gama de recursos que oferece.

Neste capítulo, falaremos sobre a ferramenta Pytest. Posteriormente, será apresentado, sua instalação e sua aplicação em testes unitários, de integração, de aceitação, demonstrando o que a torna uma ferramenta poderosa e versátil para garantir a qualidade e confiabilidade do seu código Python.

## Pytest

Pytest é uma ferramenta de teste em Python bem conhecida e eficaz, que se destaca por sua facilidade e adaptabilidade. Permite a criação de testes compreensíveis e legíveis, abrangendo desde testes unitários básicos até testes funcionais mais elaborados para programas e bibliotecas.

Suas principais características, como sua simplicidade e legibilidade no qual facilita a criação de testes de maneira clara e concisa, mesmo para desenvolvedores que estão começando com testes automatizados. Testes no pytest são geralmente escritos como funções que começam com o prefixo `test_`, e as verificações (assertivas) dos resultados são feitas utilizando a palavra-chave `assert` do Python.

Outros recursos disponíveis que o tornam uma ferramenta poderosas, como testes

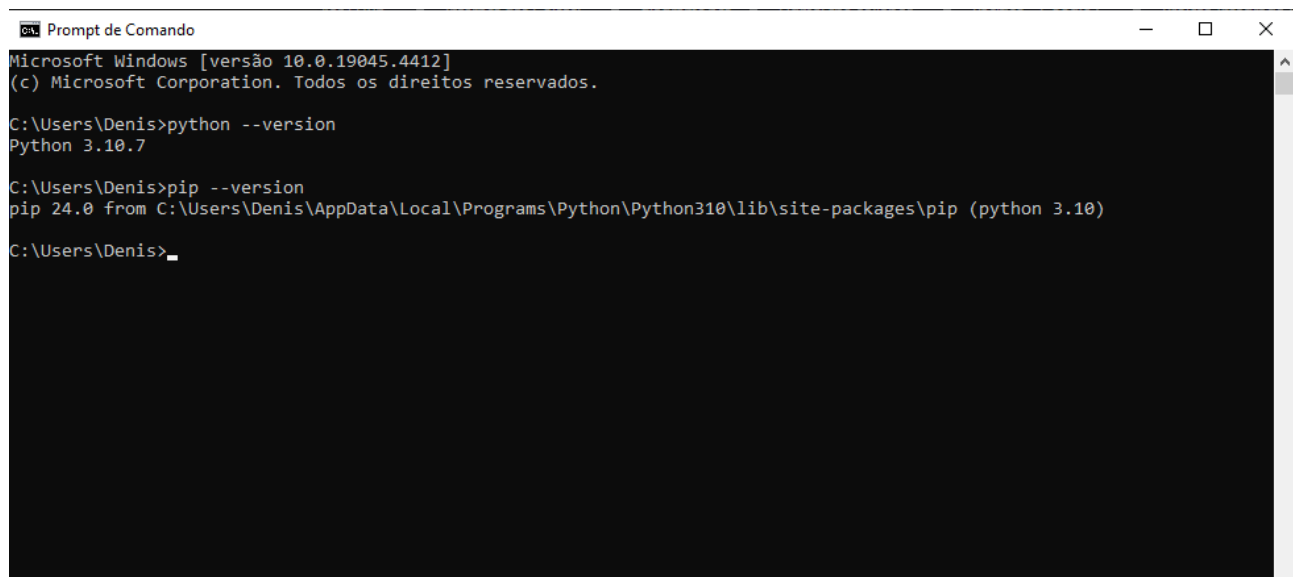
parametrizados, fixtures que são usadas para configurar o ambiente de teste, permitindo a criação de condições necessárias para os testes (como configuração de banco de dados, criação de arquivos temporários etc.), plugins e extensibilidade.

## Instalando o Pytest

Instalar o pytest é um processo simples que pode ser feito usando o pip, o gerenciador de pacotes do Python.

Certifique-se de que você tem o Python e o pip instalados:

Primeiro, verifique se você tem o Python e o pip instalados no seu sistema. Abra um terminal e digite:

A screenshot of a Windows Command Prompt window titled "Prompt de Comando". The window shows the following text: "Microsoft Windows [versão 10.0.19045.4412] (c) Microsoft Corporation. Todos os direitos reservados." followed by the command "C:\Users\Denis>python --version" and its output "Python 3.10.7". Then, the command "C:\Users\Denis>pip --version" is entered, resulting in the output "pip 24.0 from C:\Users\Denis\AppData\Local\Programs\Python\Python310\lib\site-packages\pip (python 3.10)". The prompt "C:\Users\Denis>" is shown at the bottom, ready for the next command.

```
Microsoft Windows [versão 10.0.19045.4412]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Denis>python --version
Python 3.10.7

C:\Users\Denis>pip --version
pip 24.0 from C:\Users\Denis\AppData\Local\Programs\Python\Python310\lib\site-packages\pip (python 3.10)

C:\Users\Denis>
```

Imagem realizada do processo de verificação de instalação do Python e pip realizado no prompt de comando em 21/05/2024

Caso o Python ou o pip não estiverem instalados, você precisará instalá-los primeiro. Você pode baixar o Python no site oficial na aba de downloads: <https://www.python.org/downloads/>, e o pip geralmente vem incluído com o Python, caso o pip não esteja instalado podemos o instalar utilizando o gerenciador de pacotes a partir do prompt de comando digitando o seguinte código `python -m ensurepip --default-pip`(faz que o pip seja instalado com as configurações padrões) .

Após confirmado a instalação do Python e do pip você pode instalar o pytest executando o seguinte comando no terminal: `pip install pytest` como podemos ver na imagem abaixo.

```
Prompt de Comando
Microsoft Windows [versão 10.0.19045.4412]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Denis>pip install pytest
Requirement already satisfied: pytest in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (8.1.1)
Requirement already satisfied: iniconfig in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (from pytest) (2.0.0)
Requirement already satisfied: packaging in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (from pytest) (23.1)
Requirement already satisfied: pluggy<2.0,>=1.4 in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (from pytest) (1.4.0)
Requirement already satisfied: exceptiongroup>=1.0.0rc8 in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (from pytest) (1.1.3)
Requirement already satisfied: tomli>=1 in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (from pytest) (2.0.1)
Requirement already satisfied: colorama in c:\users\denis\appdata\local\programs\python\python310\lib\site-packages (from pytest) (0.4.6)

C:\Users\Denis>
```

Imagem realizada do processo de instalação do pytest no Prompt de Comando em 21/05/2024

Pronto concluindo a instalação do pytest caso queira pode-se verificar a instalação utilizando o código `pytes --version`, a partir de agora já podemos escrever testes utilizando o pytest.

## Testes na Prática

O pytest é uma ferramenta poderosa e flexível para testar aplicações Python. Ele oferece uma sintaxe simples e uma ampla gama de funcionalidades avançadas que ajudam a manter a qualidade do código. Com o uso de fixtures, marcações e parametrização, os testes podem ser mais organizados, reutilizáveis e abrangentes. Integrando-se bem com outras ferramentas do ecossistema Python.

Para demonstrar melhor o uso do pytest neste tópico trarei alguns exemplos do uso com um simples exemplo de soma.

```
test_soma.py > ...
1  #definindo função Soma
   tabnine: test | explain | document | ask
2  def soma(a, b):
3      """Soma dois números."""
4      return a + b
5
   tabnine: test | explain | document | ask
6  def test_soma():
7      """Testa a função soma."""
8      assert soma(1, 2) == 3 # Verifica se a função retorna o valor esperado
9      assert soma(-5, 10) == 5 # Testa outro caso de teste
10
   tabnine: test | explain | document | ask
11 def test_soma_positivos():
12     """Testa a função soma com números positivos."""
13     assert soma(1, 2) == 3
14
   tabnine: test | explain | document | ask
15 def test_soma_negativos():
16     """Testa a função soma com números negativos."""
17     assert soma(-1, -2) == -3
18
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
===== test session starts =====
platform win32 -- Python 3.10.7, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\Users\Denis\Desktop\testesOrlando
plugins: anyio-3.7.1
collected 3 items

test_soma.py ... [100%]

===== 3 passed in 0.04s =====
```

Imagens tiradas de um código de operações matemáticas realizadas no Visual Studio Code em 21/05/2024

O pytest é amplamente reconhecido por seu sistema de fixtures, que facilita a configuração e a limpeza do ambiente de teste de maneira eficiente e reutilizável. Fixtures são funções que permitem preparar o estado necessário antes que os testes sejam executados e garantir que o ambiente seja limpo após a execução dos testes.

```

1 tabnine: test | explain | document | ask
2
3 @pytest.fixture
4 def lista_de_nomes_fixture():
5     """Cria uma lista de nomes para os testes."""
6     return ["Fulano", "Beltrano", "Sicrano"]
7
8 tabnine: test | explain | document | ask
9 def test_nome_na_lista(lista_de_nomes_fixture):
10     """Testa se o nome 'Fulano' está presente na lista."""
11     assert "Fulano" in lista_de_nomes_fixture
12
13 #A função de teste recebe o parâmetro lista_de_nomes_fixture.
14 #Como vimos antes, essa fixture fornece uma lista de nomes pré-criada e ordenada.
15 #Isso elimina a necessidade de criar ou ordenar a lista dentro da própria função de teste,
16 # promovendo organização e reuso de código.

```

PROBLEMS

2

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

COMMENTS

Python

+

+

+

+

+

+

```

===== test session starts =====
platform win32 -- Python 3.10.7, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\Users\Denis\Desktop\testesOrlando
plugins: anyio-3.7.1
collected 1 item

test_lista_nome_fixture.py .

1 passed in 0.04s

```

Imagens tiradas de um código de operações matemáticas realizadas no Visual Studio Code em 21/05/2024

## Síntese

Com a simplicidade e a grande comunidade crescente desta ferramenta de teste a sua utilização é crescente justificada pelas possibilidades de seu uso em diversos cenários, onde neste capítulo apresentamos alguns pontos que a torna uma ótima ferramenta para testes, no qual auxilia desenvolvedores da linguagem python.

## Bibliografia

- [1] PYTEST. Documentação oficial do Pytest. Disponível em: <https://docs.pytest.org/en/8.2.x/>. Acesso em: 04 jun. 2023.
- [2] PYTEST. Documentação oficial do Pytest. Disponível em: <https://docs.pytest.org/en/8.2.x/getting-started.html>. Acesso em: 04 jun. 2023.
- [3] MEDIUM. Documentação oficial do Pytest. Disponível em: <https://medium.com/beyn-technology/hands-on-start-testing-with-pytest-1ef39e59176a>. Acesso em: 04 jun. 2023.
- [4] AWARI. Documentação oficial do Pytest. Disponível em: <https://awari.com.br/instalar-pip-python-aprenda-como-utilizar-essa-poderosa-ferramenta-de-gerenciamento-de-pacotes-no-python> Acesso em: 28 maio 2023.

# Capítulo 05



# Capítulo 05

**Diego Eduardo Wiltler**  
**diego.eduardo.w@gmail.com**

## Introdução à Biblioteca Hypothesis

A biblioteca Hypothesis é uma ferramenta poderosa e flexível para testes de software, baseada no conceito de property-based testing. Originária do mundo da programação funcional, ela foi adaptada para Python, tornando-se uma ferramenta essencial para muitos desenvolvedores que buscam melhorar a robustez e confiabilidade de seus códigos através de testes automatizados.

A Hypothesis automatiza a criação de testes, permitindo que os desenvolvedores definam as propriedades gerais que uma função deve satisfazer, em vez de um conjunto fixo de valores. A partir daí, a biblioteca gera automaticamente casos de teste que exploram limites e condições incomuns, muitas vezes descobrindo falhas que testes manuais ou convencionais poderiam não encontrar.

## Funcionamento Básico da Hypothesis

A Hypothesis é uma biblioteca que transforma o modo como conduzimos testes de software ao implementar o conceito de property-based testing. Este método distingue-se dos testes unitários tradicionais, onde o desenvolvedor escreve testes específicos com valores pré-definidos. Em vez disso, com a Hypothesis, o desenvolvedor especifica propriedades que o código deve satisfazer e a biblioteca gera automaticamente uma vasta gama de entradas para testar essas propriedades.

### Princípios do Property-based Testing

1. **Generalização:** Ao invés de testar casos específicos, a Hypothesis testa a generalidade do código. Por exemplo, se uma função deve ordenar listas, ao invés de testar com uma lista específica, a Hypothesis gera diversas listas aleatórias para garantir que a função ordena corretamente qualquer lista.
2. **Simplificação Automática:** Quando um teste falha, a Hypothesis tenta 'simplificar' os dados que causaram a falha, identificando o menor caso possível que ainda provoca o erro. Este processo é conhecido como "shrinking", facilitando para os desenvolvedores entenderem o que deu errado.
3. **Repetibilidade:** Cada teste gerado pela Hypothesis é repetível. A biblioteca fornece um 'seed' específico usado para gerar o teste. Isso permite que os desenvolvedores reproduzam exatamente o mesmo conjunto de dados que levou a uma falha, essencial para a depuração e correção de erros.

## Como a Hypothesis Gera Testes

A geração de testes pela Hypothesis baseia-se na definição de estratégias para cada tipo de dado que sua função espera. Por exemplo, se uma função aceita um inteiro e uma string, o desenvolvedor define estratégias para gerar inteiros e strings. A Hypothesis então utiliza essas estratégias para criar combinações desses dados, tentando cobrir um espectro amplo de casos de uso, incluindo valores extremos como números muito grandes ou strings vazias.

O processo inicia-se com a função `@given`, um decorador que é usado para especificar as estratégias de entrada. Vejamos um exemplo básico:

```
from hypothesis import given
import hypothesis.strategies as st

@given(st.integers(), st.text())
def test_function(x, y):
    assert isinstance(x, int)
    assert isinstance(y, str)
```

Este exemplo simples demonstra como iniciar um teste com a Hypothesis, especificando que `x` deve ser um inteiro e `y` uma string. A partir dessas definições, a Hypothesis irá gerar múltiplos conjuntos de dados para tentar quebrar a função `test_function`.

## Configuração e Instalação da Hypothesis

Configurar a biblioteca Hypothesis em seu ambiente de desenvolvimento é um processo simples, mas fundamental para aproveitar todos os benefícios do property-based testing. Neste capítulo, abordaremos as etapas necessárias para instalar e configurar a Hypothesis, preparando seu ambiente para criar testes robustos e eficazes.

### Instalação da Hypothesis

A Hypothesis é facilmente instalável via `pip`, o gerenciador de pacotes para Python. Para instalar a versão mais recente da biblioteca, basta executar o seguinte comando no terminal:

```
pip install hypothesis
```

Esse comando instala a Hypothesis e todas as dependências necessárias. É recomendável que a instalação seja feita dentro de um ambiente virtual Python para evitar conflitos de dependências com outros projetos ou com a biblioteca do sistema.

## Configuração Inicial

Após a instalação, a Hypothesis não requer configuração adicional para começar a ser usada em testes simples. No entanto, ela oferece várias opções de configuração que permitem ajustar o comportamento da geração de testes. Essas configurações podem ser especificadas globalmente ou por teste individual, através do decorador `@settings`.

Por exemplo, para aumentar o número de exemplos gerados durante os testes, você pode ajustar o parâmetro `max_examples` como mostrado a seguir:

```
from hypothesis import given, settings
import hypothesis.strategies as st

@settings(max_examples=1000)
@given(st.integers())
def test_integers_are_non_negative(x):
    assert x >= 0
```

Este código configura o teste para que a Hypothesis gere até 1000 exemplos diferentes, aumentando as chances de descobrir falhas.

## Escrevendo Testes Básicos com Hypothesis

Após configurar a Hypothesis em seu ambiente de desenvolvimento, o próximo passo é começar a escrever testes. Neste capítulo, exploraremos como criar testes básicos utilizando a Hypothesis, introduzindo conceitos fundamentais e algumas das estratégias mais comuns para gerar dados de teste.

### Criando Seu Primeiro Teste

O cerne da Hypothesis está em seu decorador `@given`, que é usado para fornecer dados de teste à função de teste. Vamos criar um teste simples para verificar se a soma de dois números inteiros sempre resulta em um número maior ou igual a cada um dos operandos. Para isso, usaremos a estratégia `integers()`:

```
from hypothesis import given
import hypothesis.strategies as st

@given(st.integers(), st.integers())
def test_sum_of_integers(x, y):
    assert x + y >= x
    assert x + y >= y
```

Este teste verifica que a soma de dois inteiros x e y é sempre maior ou igual a x e y separadamente, o que é verdade para números inteiros em Python devido às propriedades da aritmética.

## Estratégias Básicas

A Hypothesis fornece várias estratégias pré-definidas que podem ser usadas para gerar tipos de dados simples, como inteiros, strings e listas. Aqui estão algumas das estratégias mais utilizadas:

- Integers: `st.integers()` gera números inteiros dentro de um intervalo especificado ou dentro de todo o intervalo possível se nenhum limite for especificado.
- Text: `st.text()` gera strings aleatórias baseadas em caracteres Unicode.
- Lists: `st.lists(elements=st.integers())` gera listas de inteiros. O argumento `elements` pode ser qualquer outra estratégia para gerar os elementos da lista.

## Testando Funções com Dados Complexos

À medida que você se familiariza com as estratégias básicas, pode começar a testar funções que requerem entradas mais complexas. Por exemplo, testar uma função que aceita uma lista de números e retorna a lista ordenada:

```
@given(st.lists(st.integers()))
def test_sorting(list_of_integers):
    sorted_list = sorted(list_of_integers)
    for i in range(1, len(sorted_list)):
        assert sorted_list[i] >= sorted_list[i - 1]
```

Este teste verifica se os elementos de uma lista de inteiros estão em ordem não decrescente após a aplicação da função `sorted`.

# Estratégias Avançadas na Hypothesis

Depois de dominar os fundamentos da escrita de testes básicos com a Hypothesis, você pode explorar estratégias mais avançadas que permitem uma maior customização e controle sobre os dados de teste gerados. Neste capítulo, detalhamos algumas dessas estratégias avançadas e como elas podem ser usadas para testar casos ainda mais complexos.

## Customização de Estratégias

A Hypothesis permite que você crie estratégias personalizadas para se adequar exatamente às necessidades dos seus testes. Usando a função `builds()`, você pode construir instâncias de qualquer classe Python baseando-se em estratégias para seus atributos:

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

product_strategy = builds(Product, name=text(), price=integers(min_value=1))
```

Esta estratégia personalizada `product_strategy` pode ser usada para gerar instâncias da classe `Product` com nomes e preços aleatórios, onde o preço sempre será um inteiro positivo.

## Composição de Estratégias

Além de personalizar estratégias para tipos de dados específicos, você também pode compor várias estratégias para criar entradas de teste mais complexas. Por exemplo, para testar uma função que processa listas de produtos, você pode usar:

```
from hypothesis import given
from hypothesis.strategies import lists

@given(lists(product_strategy))
def test_process_products(products):|
```

## Estratégias Condicionais

Em alguns casos, você pode querer que os dados de teste satisfaçam condições específicas. A Hypothesis oferece o `assume()` para isso, que permite incluir condições dentro de seus testes:

```
from hypothesis import given, assume
import hypothesis.strategies as st

@given(st.lists(st.integers(), min_size=1))
def test_positive_sums(numbers):
    assume(all(x > 0 for x in numbers))
    assert sum(numbers) > 0
```

Este teste gera listas de inteiros e assume que todos os números são positivos, descartando quaisquer casos que não atendam a essa condição antes de executar o teste.

### Uso de filter()

Outra maneira de controlar a geração de dados é através do uso da função filter() em uma estratégia, que exclui dados que não atendem a um critério específico:

```
positive_integers = st.integers().filter(lambda x: x > 0)
@given(positive_integers)
def test_positive_integer(x):
    assert x > 0
```

Este teste assegura que apenas inteiros positivos são usados, utilizando a função filter() para rejeitar quaisquer valores que não sejam positivos.

## Integração da Hypothesis com Outras Ferramentas de Teste

A Hypothesis pode ser integrada com várias outras ferramentas e frameworks de teste para criar um ambiente de teste mais robusto e abrangente. Este capítulo explora como a Hypothesis pode ser combinada com algumas das ferramentas mais populares para maximizar a eficácia dos testes automatizados.

### Integração com pytest

O pytest é um dos frameworks de teste mais populares no ecossistema Python. A Hypothesis funciona de maneira excelente com o pytest, permitindo a utilização de suas poderosas funcionalidades de geração de testes junto com as ricas funcionalidades de assertiva e fixação do pytest.

Para usar a Hypothesis com pytest, basta escrever seus testes com o decorador @given normalmente, e o pytest identificará e executará esses testes. Aqui está um exemplo de como isso pode ser feito:

```

from hypothesis import given
import hypothesis.strategies as st

@given(st.integers(), st.integers())
def test_sum(x, y):
    assert x + y == y + x

```

Executar este teste com pytest irá automaticamente gerar múltiplos pares de inteiros e testar a propriedade comutativa da adição.

Integração com unittest

Para projetos que utilizam o unittest, o módulo de testes padrão do Python, a Hypothesis também oferece suporte. Para integrar, você pode usar o decorador `@given` da Hypothesis diretamente nos métodos de teste do unittest.

```

import unittest
from hypothesis import given
import hypothesis.strategies as st

class TestMath(unittest.TestCase):
    @given(st.integers(), st.integers())
    def test_sum(self, x, y):
        self.assertEqual(x + y, y + x)

```

Esta integração permite que você aproveite os recursos de organização e estrutura do unittest junto com a geração de testes da Hypothesis.

Utilizando Hypothesis com outras bibliotecas de testes

A Hypothesis também pode ser usada com outras bibliotecas populares como nose2 e tox, e pode ser integrada em ambientes de teste mais complexos que utilizam contêineres e integração contínua. Isso permite que os desenvolvedores mantenham seus fluxos de trabalho de teste existentes enquanto aproveitam os benefícios da geração de testes baseados em propriedades.

## Exemplos Reais de Aplicação da Hypothesis

A aplicação prática da Hypothesis em projetos reais pode ilustrar de forma clara os benefícios desta ferramenta. Neste capítulo, exploramos casos reais onde a Hypothesis foi usada para descobrir falhas, melhorar a qualidade do código e garantir a confiabilidade de sistemas em diferentes contextos.

### Descoberta de Bugs em Bibliotecas Populares

Um dos usos mais importantes da Hypothesis é na descoberta de bugs em bibliotecas amplamente utilizadas. Por exemplo, a Hypothesis foi usada para testar a biblioteca de manipulação de datas e horas arrow. Através dos testes automatizados, foram descobertos casos onde a biblioteca falhava em converter corretamente datas em certos fusos horários, um bug crítico para qualquer aplicação que dependesse de precisão temporal.

## **Melhoria na Robustez do Software**

Em outro caso, uma equipe estava desenvolvendo um parser de XML que precisava lidar com uma grande variedade de dados mal formados de maneira segura. Usando a Hypothesis, a equipe foi capaz de gerar automaticamente inúmeros formatos malformados de XML para testar a robustez do parser. O resultado foi a identificação e correção de várias vulnerabilidades que poderiam levar a falhas de segurança sérias se não fossem tratadas.

## **Testando APIs de Redes Complexas**

A Hypothesis também foi fundamental no teste de APIs de redes complexas, onde as interações entre múltiplos sistemas podem levar a comportamentos imprevisíveis. Um projeto de rede social utilizou a Hypothesis para simular e testar as interações entre milhares de usuários simultâneos, verificando a consistência e a estabilidade da API sob condições de estresse elevado.

## **Otimização de Algoritmos**

No campo da otimização de algoritmos, a Hypothesis ajudou a testar algoritmos de ordenação e busca para garantir que funcionasse eficientemente em todos os casos. Ao gerar uma grande variedade de listas e arrays com características únicas, a Hypothesis possibilitou a identificação de padrões de dados que causavam degradação no desempenho, permitindo otimizações direcionadas e eficazes.

# **Valor e Impacto da Hypothesis no Mercado**

A utilização da Hypothesis está crescendo significativamente no mercado de desenvolvimento de software devido ao seu potencial de aumentar a eficiência e a confiabilidade dos testes automatizados. Neste capítulo, exploramos o valor agregado pela Hypothesis aos projetos e sua crescente adoção por empresas e desenvolvedores em diversos setores.

## **Aumento da Confiança no Software**

O principal valor trazido pela Hypothesis é o aumento da confiança no software desenvolvido. Empresas que adotam a Hypothesis em seus processos de teste conseguem garantir um nível mais alto de qualidade e confiabilidade, reduzindo a frequência e a gravidade dos bugs. Isso se traduz em software mais estável e, consequentemente, em uma melhor experiência para o usuário final.

## **Redução de Custos com Manutenção**

A capacidade da Hypothesis de encontrar falhas e casos de borda de maneira eficiente ajuda as organizações a reduzir significativamente os custos de manutenção de software. Ao identificar e corrigir problemas potenciais durante a fase de desenvolvimento, as empresas evitam gastos com patches de emergência e suporte técnico após o lançamento do produto.

## **Adoção em Diversos Setores**

A flexibilidade da Hypothesis permite sua adoção em uma variedade de setores, incluindo finanças, saúde, tecnologia e governo. Em cada um desses campos, a garantia de que os sistemas funcionem conforme o esperado sob diversas condições é crucial, especialmente quando falhas podem ter consequências sérias.



## **Casos de Sucesso**

Empresas de tecnologia de ponta, como Google e Amazon, têm integrado a Hypothesis em seus fluxos de trabalho de desenvolvimento. Por exemplo, a Hypothesis foi usada para testar sistemas críticos de processamento de transações financeiras, garantindo que esses sistemas pudessem lidar com uma ampla gama de cenários de entrada sem falhas.

## **Impacto na Comunidade de Desenvolvimento**

Além do impacto direto em projetos e empresas, a Hypothesis também contribui para a comunidade de desenvolvimento de software como um todo. A biblioteca é frequentemente destacada em conferências e workshops sobre testes de software, e muitos desenvolvedores compartilham suas experiências e melhores práticas em blogs e fóruns online.

# **Desafios e Limitações da Hypothesis**

Embora a Hypothesis seja uma ferramenta poderosa para testes automatizados, como qualquer tecnologia, ela possui suas limitações e desafios. Neste capítulo, discutimos algumas das dificuldades encontradas pelos usuários ao implementar e utilizar a Hypothesis, bem como as limitações inerentes à abordagem de testes baseados em propriedades.

## **Curva de Aprendizado**

Um dos maiores desafios ao adotar a Hypothesis é sua curva de aprendizado. Para desenvolvedores acostumados apenas com testes unitários tradicionais, a transição para testes baseados em propriedades pode ser intimidante. A necessidade de pensar em termos de propriedades gerais do sistema, em vez de casos específicos, exige uma mudança significativa na mentalidade de teste.

## **Complexidade na Definição de Estratégias**

Criar estratégias de teste eficazes que cubram adequadamente o espaço de problema pode ser complexo, especialmente para funções que aceitam entradas altamente estruturadas ou com muitas dependências. A dificuldade em definir estratégias adequadas pode levar a uma cobertura de teste insuficiente ou a testes que não exploram adequadamente as falhas potenciais.

## **Desempenho dos Testes**

Os testes baseados em propriedades, especialmente quando mal configurados, podem ser significativamente mais lentos do que os testes unitários convencionais. Isso ocorre porque a Hypothesis pode gerar uma grande quantidade de casos de teste para tentar quebrar as propriedades definidas. Em sistemas grandes e complexos, isso pode levar a tempos de execução prolongados, afetando a eficiência do desenvolvimento e da integração contínua.

## **Falsos Positivos e Gerenciamento de Exceções**

Outra questão que pode surgir com a Hypothesis são os falsos positivos, onde os testes falham não devido a um bug real, mas devido a inadequações nas estratégias de geração de dados ou na compreensão das propriedades testadas. Além disso, a gestão de exceções pode ser complicada, especialmente quando as exceções são um comportamento válido para algumas entradas.

## **Limitações da Geração de Dados**

A Hypothesis é limitada pelas estratégias de dados disponíveis e pela forma como essas estratégias são implementadas. Em alguns casos, pode ser desafiador gerar dados que sejam verdadeiramente representativos de todos os cenários de uso possíveis, especialmente em domínios altamente especializados ou com regras de negócio complexas.

## **Futuro da Hypothesis e Tendências em Testes Baseados em Propriedades**

À medida que o desenvolvimento de software continua a evoluir, a Hypothesis e os testes baseados em propriedades também estão se adaptando e melhorando. Este capítulo explora as tendências futuras e as inovações esperadas para a Hypothesis, fornecendo uma visão sobre como esta ferramenta pode evoluir e continuar a impactar a indústria de software.

### **Inovações na Geração de Dados**

Uma das áreas mais promissoras para inovação na Hypothesis é a melhoria das estratégias de geração de dados. Espera-se que futuras versões da biblioteca ofereçam maneiras ainda mais inteligentes e eficientes de gerar dados que desafiem os sistemas de maneiras novas e úteis. Isso pode incluir melhores algoritmos para 'shrinking', que reduzem ainda mais os casos de teste para suas formas mais simples e informativas após uma falha.

### **Integração com Inteligência Artificial**

A integração de técnicas de inteligência artificial e aprendizado de máquina com a geração de testes baseados em propriedades é outra área de desenvolvimento potencial. Isso poderia permitir que a Hypothesis aprenda com testes anteriores para melhor prever e explorar as áreas mais críticas de um código, aumentando a eficácia dos testes sem exigir mais esforço humano.

### **Expansão para Novas Linguagens e Plataformas**

Embora a Hypothesis seja mais conhecida no contexto do Python, há um interesse crescente em expandir suas capacidades para outras linguagens de programação. Isso ampliaria seu impacto, permitindo que desenvolvedores em diferentes ambientes beneficiem-se de testes baseados em propriedades. A implementação de princípios similares em ambientes como JavaScript, Go ou Rust poderia transformar os testes nesses ecossistemas.

### **Melhoria na Usabilidade e Documentação**

Para superar a curva de aprendizado associada aos testes baseados em propriedades, é provável que vejamos melhorias significativas na usabilidade e na documentação da Hypothesis. Isso incluirá tutoriais mais interativos, exemplos de código abrangentes e uma comunidade mais ativa para suportar novos usuários.

### **Colaboração e Normas Comunitárias**

À medida que mais organizações adotam a Hypothesis, pode-se esperar uma maior colaboração entre indústrias para estabelecer melhores práticas e normas em testes baseados em propriedades. Isso ajudará a padronizar abordagens e aumentar a confiança nos resultados dos testes, facilitando a adoção por uma gama ainda maior de projetos e equipes.

# Biografia e Referências sobre a Biblioteca Hypothesis

A Hypothesis é uma biblioteca de testes avançada para Python, conhecida por sua abordagem de testes baseados em propriedades, que permite uma escrita mais simples e resultados poderosos ao executar os testes. A biblioteca ajuda a encontrar casos extremos no código que geralmente não são considerados durante os testes convencionais. A Hypothesis é amplamente apreciada por sua estabilidade e facilidade de integração com suítes de testes existentes.

## Funcionalidades Principais:

- A Hypothesis oferece uma implementação completa de testes baseados em propriedades para Python, incluindo testes com o estado.
- É compatível com várias ferramentas de teste, como Pytest, Unittest, Nose e testes do Django, e suporta CPython e PyPy 3.8 e versões posteriores.
- A biblioteca é de código aberto sob a Mozilla Public License 2.0 ([Hypothesis Works](#)).
- Documentação e Comunidade:
- A documentação da Hypothesis é rica e detalhada, fornecendo guias de início rápido, detalhes sobre recursos avançados e exemplos práticos para ajudar os usuários a começar e a explorar as capacidades da biblioteca ([Hypothesis Docs](#)) ([Red Hat Developer](#)).
- A Hypothesis mantém uma comunidade ativa, com suporte contínuo através de sua documentação on-line e projetos colaborativos disponíveis no GitHub ([Hypothesis Works](#)).

## Aplicações Práticas:

- A biblioteca é eficaz na identificação de erros e na verificação de garantias de código através de exemplos que desafiam as expectativas dos desenvolvedores, como demonstrado em testes que revelam problemas ao lidar com entradas como strings vazias ou valores inesperados ([Hypothesis Docs](#)).

A Hypothesis é valorizada tanto por desenvolvedores individuais quanto por grandes empresas devido à sua capacidade de aprimorar significativamente a qualidade dos testes e descobrir vulnerabilidades que poderiam passar despercebidas em abordagens de teste tradicionais. A abordagem de testes baseados em propriedades foi popularizada pelo QuickCheck em Haskell e adaptada de forma eficaz para Python pela Hypothesis ([Red Hat Developer](#)).

Para mais informações, explore a documentação oficial da Hypothesis e sua página no PyPI:

- [Hypothesis Documentation](#)
- [Hypothesis on PyPI](#)

# Capítulo 06

# Capítulo 06

**Marius Jorge Pessi**  
**mariuspessi@yahoo.com.br**

*“Uma máquina consegue fazer o trabalho de 50 homens ordinários.  
Nenhuma máquina consegue fazer o trabalho de um homem extraordinário”*

*Elbert Hubbard, escritor.*

## Introdução

Esse capítulo discutiremos a ferramenta Selenium, uma das principais ferramentas de testes automatizados. O Selenium iniciou seu desenvolvimento em 2004 na ThoughtWorks em Chicago, com Jason Huggins construindo um script como “JavaScriptTestRunner” para um teste de um aplicativo interno de Tempo e Despesas. Os testes automatizados dos aplicativos são essenciais para a ThoughtWork, com o auxílio de Paul Gross e Jie Tina Wang no desenvolvimento.

O Selenium é uma ferramenta muito importante para automação de testes e Web Scraping. Nos testes automatizados simula um usuário, interagindo com botões, preenchendo campos de formulários. Interage com praticamente todos os navegadores hoje no mercado por meio de drivers e pode ser utilizado em diversas linguagens de programação como Javascript, Java, Ruby, C#, Python entre outros.

O Selenium trabalha enviando comandos básicos ao navegador ou solicitando informações, ele possui 8 componentes básicos:

Iniciando Sessão, agindo e solicitando informações do navegador, estabelecendo uma estratégia de espera, encontrando, agindo e solicitando informações de um elemento, encerrando a sessão.

As principais IDE's que pode ser usado para os scripts do Selenium são: [Eclipse](#), [IntelliJ IDEA](#), [PyCharm](#), [RubyMine](#), [Rider](#), [WebStorm](#) e [VS Code](#).

Esse capítulo do livro será focado no WebDriver do Selenium implementado em uma solução em JavaScript com a IDE VsCode.

Instalando a ferramenta:

Com o Node previamente instalado, deve ser usado o comando `npm install selenium-webdriver` no terminal do VsCode para a instalação da ferramenta e adicionado no Package Json o requisito na dependência "mocha": "10.4.0".

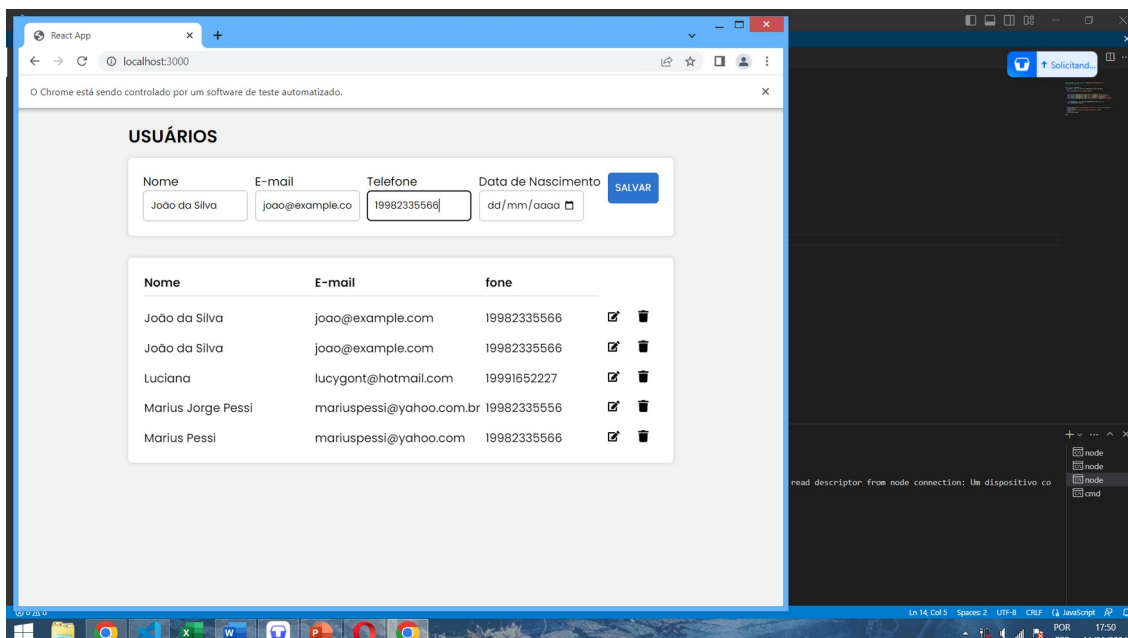
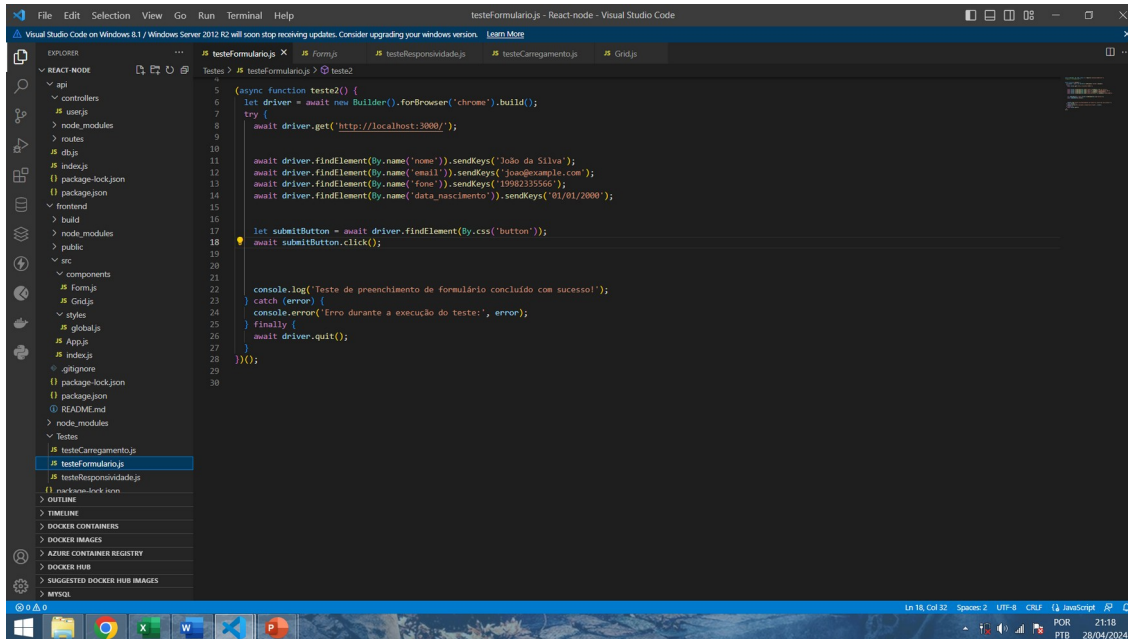
No site <https://developer.chrome.com/docs/chromedriver/downloads?hl=pt-br> devemos baixar o driver do Google Chrome pela versão que tem instalado na máquina.

Os principais testes unitários são: testes de Funcionabilidade, Regressão, Integração, Aceitação de usuário, Desempenho e Navegabilidade

## Práticas de testes

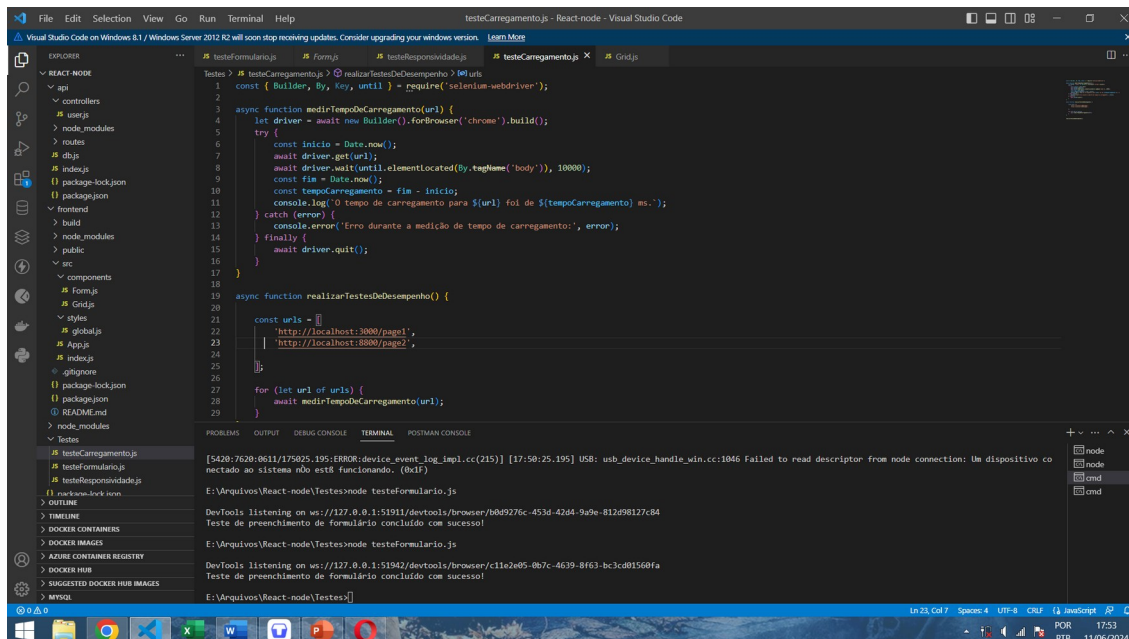
### Teste de Funcionabilidade:

O teste de Funcionabilidade o Selenium preenche os campos e pode utilizar os botões da aplicação:



## Teste de Desempenho

O teste de Desempenho não é uma das especialidades do Selenium, porém pode ser usado para medir o tempo de carregamento tanto da API e da Aplicação Front-end, verificando possíveis atrasos nos carregamentos das aplicações.



```
File Edit Selection View Go Run Terminal Help
testeCarregamento.js - React-node - Visual Studio Code

Visual Studio Code on Windows 8.1 / Windows Server 2012 R2 will soon stop receiving updates. Consider upgrading your windows version. Learn More

EXPLORER
  REACT-NODE
    api
    controllers
    users.js
    node_modules
    routes
    db.js
    index.js
    package-lock.json
    package.json
    frontend
    build
    node_modules
    public
    src
    components
    Form.js
    Grid.js
    styles
    global.js
    App.js
    index.js
    gptignore
    package-lock.json
    package.json
    README.md
    node_modules
    testes
      testeCarregamento.js
      testeFormulario.js
      testeResponsividade.js
  OUTLINE
  TIMELINE
  DOCKER CONTAINERS
  DOCKER IMAGES
  AZURE CONTAINER REGISTRY
  DOCKER HUB
  SUGGESTED DOCKER HUB IMAGES
  MYSQL

TESTES > # testeCarregamento.js > realizarTestesDeDesempenho > 10 urls
1 const { Builder, By, Key, until } = require('selenium-webdriver');
2
3 async function medirTempoDeCarregamento(url) {
4   let driver = await new Builder().forBrowser('chrome').build();
5   try {
6     const inicio = Date.now();
7     await driver.get(url);
8     await driver.wait(until.elementLocated(By.tagName('body')), 10000);
9     const fim = Date.now();
10    const tempoCarregamento = fim - inicio;
11    console.log(`O tempo de carregamento para ${url} foi de ${tempoCarregamento} ms.`);
12  } catch (error) {
13    console.error('Erro durante a medição de tempo de carregamento:', error);
14  } finally {
15    await driver.quit();
16  }
17 }
18
19 async function realizarTestesDeDesempenho() {
20   const urls = [
21     'http://localhost:3000/pagel',
22     'http://localhost:8000/page2',
23   ];
24
25   for (let url of urls) {
26     await medirTempoDeCarregamento(url);
27   }
28 }
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
255
```



```
8 console.log('Teste de responsividade para ${url} (${tamanhoDaJanela.width}x${tamanhoDaJanela.height}) concluido.');
9
10 catch (error) {
11   console.error('Erro durante o teste de responsividade:', error);
12 }
13 finally {
14   await driver.quit();
15 }
16
17 async function realizarTestesDeResponsividade() {
18
19   const urls = [
20     'http://localhost:3000',
21   ];
22
23
24
25   const tamanhosDaJanela = [
26     { width: 320, height: 480 }, // Exemplo: iPhone SE
27     { width: 768, height: 1024 }, // Exemplo: iPad
28     { width: 1024, height: 768 }, // Exemplo: Tablet
29     { width: 1440, height: 900 }, // Exemplo: Laptop
30     { width: 1920, height: 1080 }, // Exemplo: Desktop
31   ];
32
33   for (let url of urls) {
34     for (let tamanhoDaJanela of tamanhosDaJanela) {
35       await testarResponsividade(url, tamanhoDaJanela);
36     }
37   }
38
39   realizarTestesDeResponsividade();
40
41 }
42
```

```
E:\Arquivos\React-node\Testes\node testeCarregamento.js
DevTools listening on ws://127.0.0.1:52031/devtools/browser/2f95bdc-deac-48af-961d-38f199efbf90
0 tempo de carregamento para http://localhost:3000/pagel foi de 3883 ms.
DevTools listening on ws://127.0.0.1:52052/devtools/browser/2800cc5d-8e66-44af-9aeb-ef4cced08c79
0 tempo de carregamento para http://localhost:3000/pagel foi de 3273 ms.
E:\Arquivos\React-node\Testes\node testeResponsividade.js
DevTools listening on ws://127.0.0.1:52088/devtools/browser/73fb6218-d768-4218-bbe0-20b1124ca5af
Teste de responsividade para http://localhost:3000 (320x480) concluido.
DevTools listening on ws://127.0.0.1:52109/devtools/browser/67aa8b16-4eec-4430-9a8e-82fa9611432d
Teste de responsividade para http://localhost:3000 (768x1024) concluido.
DevTools listening on ws://127.0.0.1:52130/devtools/browser/fa9cca9f-8281-4e30-9081-fellde21a8f0
Teste de responsividade para http://localhost:3000 (1024x768) concluido.
DevTools listening on ws://127.0.0.1:52149/devtools/browser/cd94267-5397-4cf9-bd47-24898d1fe8c5
Teste de responsividade para http://localhost:3000 (1440x900) concluido.
DevTools listening on ws://127.0.0.1:52168/devtools/browser/6db84acd-2ed3-4ff1-b8c5-79bf8d213611
Teste de responsividade para http://localhost:3000 (1920x1080) concluido.
E:\Arquivos\React-node\Testes>
```

## Síntese

O Selenium é uma importante ferramenta no mundo das aplicações e testes por sua versatilidade de diversas linguagens e robustez nos testes, possui grande comunidade e documentação, para ser usado desde as pequenas aplicações a aplicativos complexos.



## **Bibliografia**

- [1] Selenium. Documentação Oficial. Disponível em <https://www.selenium.dev/> . Acesso em: 20 maio 2024.
- [2] Selenium. Conheça o que é e para que serve. Disponível em [https://antlia.com.br/artigos/selenium-automacao-de-processos-web/#:~:text=O%20que%20%C3%A9%20o%20Selenium,Despesas%20\(Python%2C%20Plone\)](https://antlia.com.br/artigos/selenium-automacao-de-processos-web/#:~:text=O%20que%20%C3%A9%20o%20Selenium,Despesas%20(Python%2C%20Plone).). Acesso em: 23 maio 2024.
- [3] Selenium. O que é Selenium. Disponível em [https://www.treinaweb.com.br/blog/o-que-e-selenium#google\\_vignette](https://www.treinaweb.com.br/blog/o-que-e-selenium#google_vignette). Acesso em 02 jun. 2024.

# Capítulo 07

# Capítulo 07

**Jair Lopes Junior**  
**Jair.lopes@fatec.sp.gov.br**

"Falar é fácil,  
Me mostre o código"

Linus Torvalds

## Introdução

Neste capítulo, iremos abordar o cenário dos testes unitários para aplicações Java com JUnit, tanto os conceitos teóricos quanto o prático. Será mostrado como configurar o seu projeto e demonstrado como é o funcionamento da ferramenta através de um exemplo de uma calculadora. É um exemplo que todos conhecem o funcionamento da calculadora e assim fica fácil o entendimento.[1]

## JUnit

JUnit se destaca como uma ferramenta fundamental e amplamente adotada. Criado por Kent Beck e Erich Gamma em 1997, o JUnit rapidamente se tornou um pilar no desenvolvimento de software, facilitando a criação e execução de testes automatizados.[2]

Desde suas primeiras versões, o JUnit evoluiu significativamente, incorporando diversas funcionalidades que tornam os testes mais robustos e eficazes. Suas anotações, como `@Test`, `@Before`, `@After` e `@BeforeClass`, entre outras, simplificam a organização e a execução dos testes, permitindo que os desenvolvedores escrevam código de teste de forma clara e intuitiva.

A influência do JUnit vai além do próprio framework. Ele estabeleceu padrões que foram adotados por outras ferramentas de teste e inspirou a criação de frameworks semelhantes para diferentes linguagens de programação. Sua integração com IDEs populares, sistemas de build como Maven e Gradle, e plataformas de integração contínua, reforçou ainda mais sua importância no ciclo de desenvolvimento de software.[3]

Um marco significativo na evolução do JUnit foi a introdução do JUnit 5, também conhecido como JUnit Jupiter. Esta versão trouxe uma arquitetura modular e novas funcionalidades, como a capacidade de escrever testes dinâmicos e uma API de extensão poderosa, que permitiu aos desenvolvedores personalizar e estender o comportamento dos testes de maneiras inovadoras.

## Configuração do projeto

Crie um novo projeto Java na sua IDE preferida (por exemplo, IntelliJ IDEA ou Eclipse).

Adicione a biblioteca JUnit ao projeto. Se você estiver utilizando o gerenciador de dependência Maven, adicione a seguinte dependência ao seu projeto 'pom.xml'.

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

## Implementação da Calculadora

Crie uma classe chamada 'Calculator' no pacote principal do seu projeto.

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

## Criação do Teste com JUnit

Crie uma nova classe de teste no diretório de teste do seu projeto (geralmente 'src/test/java') e nomeie a classe como 'CalculatorTest'. Importe as classes necessárias do JUnit.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
```

Escreva o teste para o método de soma.

```
public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 2);
        assertEquals(4, result, "2 + 2 deveria ser 4");
    }
}
```

## Síntese

Neste capítulo, enfatizamos a importância dos testes automatizados no desenvolvimento ágil. Os testes automatizados, incluindo os criados com JUnit, permitem que os desenvolvedores verifiquem se seu código está funcionando corretamente e evitem regressões à medida que fazem alterações ou adicionam novos recursos.

Espero que os tópicos abordados tenham sido valiosos e que essas informações abram novas possibilidades para você. Todo o código desenvolvido e utilizado neste capítulo está disponível no GitHub, permitindo que você o analise a qualquer momento e aprofunde ainda mais seu conhecimento.

## Bibliografia

[1] JAVA. Documentação oficial do Java. Disponível em: <https://docs.oracle.com/en/java/>. Acesso em: 27 mai. 2024.

[2] JUNIT. Documentação oficial do JUnit. Disponível em: <https://junit.org/junit5/>. Acesso em: 27 mai. 2024.

[3] MAVEN. Documentação oficial do Maven. Disponível em: <https://maven.apache.org/guides/index.html>. Acesso em: 27 mai. 2024.

# Capítulo 08

# Capítulo 08

**Antonio Luis Pereira Candioto**  
**antoniolpcandioto@gmail.com**

*"Quando acreditamos apaixonadamente em algo que ainda não existe, nós o criamos. O inexistente é o que não desejamos o suficiente."*

*Franz Kafka*

## Introdução

Em testes para simulações web, geralmente precisamos validar todos os fluxos da página, possibilidades de cliques, inserções, segurança e respostas, o que pode ser extremamente trabalhoso de realizar manualmente. Para solucionar esses desafios, diversas ferramentas foram desenvolvidas para realizar testes automatizados. Os testes automatizados aplicam ferramentas de software para executar processos manuais de forma automatizada, com o objetivo de validar a funcionalidade, desempenho e segurança de uma aplicação.

Neste capítulo, falaremos sobre a biblioteca Playwright e sua utilidade para executar esses testes realizando simulações em navegadores web. O Playwright busca melhorar a qualidade das aplicações web e reduzir riscos em diversos cenários de maneira personalizada, economizando tempo e reduzindo custos.

## Playwright

Tendo como principais características o seu suporte Cross-Browser, capacidade de automação completa, gerenciamento de sessões e vários outros recursos, o Playwright lhe entrega uma vasta caixa de ferramentas para interagir com sites e realizar testes. A biblioteca está disponível em Python, Java, .NET e Node.js, tornando seu uso mais abrangente e acessível para diversos tipos de desenvolvedores back-end, porém, sua baixa complexidade e a possibilidade de integração com bibliotecas de aprendizagem de máquina e web scraping em Python criam um cenário em que essa linguagem ganha destaque entre as outras.

A ferramenta foi criada em 2020, pela Microsoft e buscou superar ferramentas semelhantes como O Puppeteer e o Selenium, com uma API mais robusta e flexível e que abrangesse mais navegadores.



**Figura 1:** Playwright

## Instalação em Python

A instalação do Playwright em Python é bem simples, primeiramente usamos a instalação pelo pip para instalarmos o pacote do playwright e em seguida instalamos todas as dependências do playwright, como os navegadores e seus motores.

Install the **Pytest** plugin:

```
pip install pytest-playwright
```

Install the required browsers:

```
playwright install
```

**Figura 2:** Instalação do framework (fonte: <https://playwright.dev/python/docs/intro>)

## Realizando testes

Com a ferramenta em mãos, é só importar e começar a construir os testes, com poucas linhas já podemos testar funções básicas como uma autenticação, como vemos no exemplo abaixo, onde verificamos se a mensagem de usuário incorreto apareceu no conteúdo do site após inserirmos os dados na tela



```

teste1.py > ...
1  from playwright.sync_api import sync_playwright
2
3  def test_login():
4      with sync_playwright() as p:
5          browser = p.firefox.launch(headless=False, slow_mo=2000)
6          page = browser.new_page()
7          page.goto('https://github.com/login')
8          page.fill('input[name="login"]', 'usuario_teste')
9          page.click('input[name="commit"]')
10         assert "Incorrect username or password." in page.text_content('body')
11         browser.close()
12
13     test_login()

```

**Figura 3:** Realização de testes com Playwright usando o assert

Apresentamos um teste também com o Pytest, onde executamos os testes abaixo em uma consulta na Wikipedia.

```

import pytest
from playwright.sync_api import sync_playwright

@pytest.fixture(scope="module")
def browser():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        yield browser
        browser.close()

@pytest.fixture
def page(browser):
    page = browser.new_page()
    yield page
    page.close()

def test_wikipedia_search(page):
    page.goto('https://pt.wikipedia.org')

    page.fill('input[name="search"]', 'Python')
    page.get_by_text("Pesquisar").nth(0).click()
    page.keyboard.press("Enter")
    page.wait_for_timeout(4000)
    assert "Javascript" in page.text_content('h1[id="firstHeading"]'), "Resultados de busca incorretos ou não encontrados"

```

**Figura 4:** Realização de testes com Playwright usando o pytest e assert

Por fim, um teste de busca na Amazon, onde queremos encontrar um resultado de Notebook após pesquisar, retornando erro se não estiver correto. Podemos também ver uma função de captura de tela no código, que é uma das funcionalidades que o Playwright apresenta.

```
from playwright.sync_api import sync_playwright

def test_shopping_cart():
    with sync_playwright() as p:
        browser = p.firefox.launch(headless=False)
        page = browser.new_page()

        page.goto('https://www.amazon.com/')
        page.fill('#twotabsearchtextbox', 'notebook')
        page.press('#twotabsearchtextbox', 'Enter')
        page.click('span.a-price')

        page.click('input#add-to-cart-button')
        page.wait_for_timeout(1000)
        assert page.inner_text('span#nav-cart-count') == '1', "Teste Falhou!"
        print("O teste passou!")
        page.screenshot(path="example.png")
        page.pause()
        browser.close()

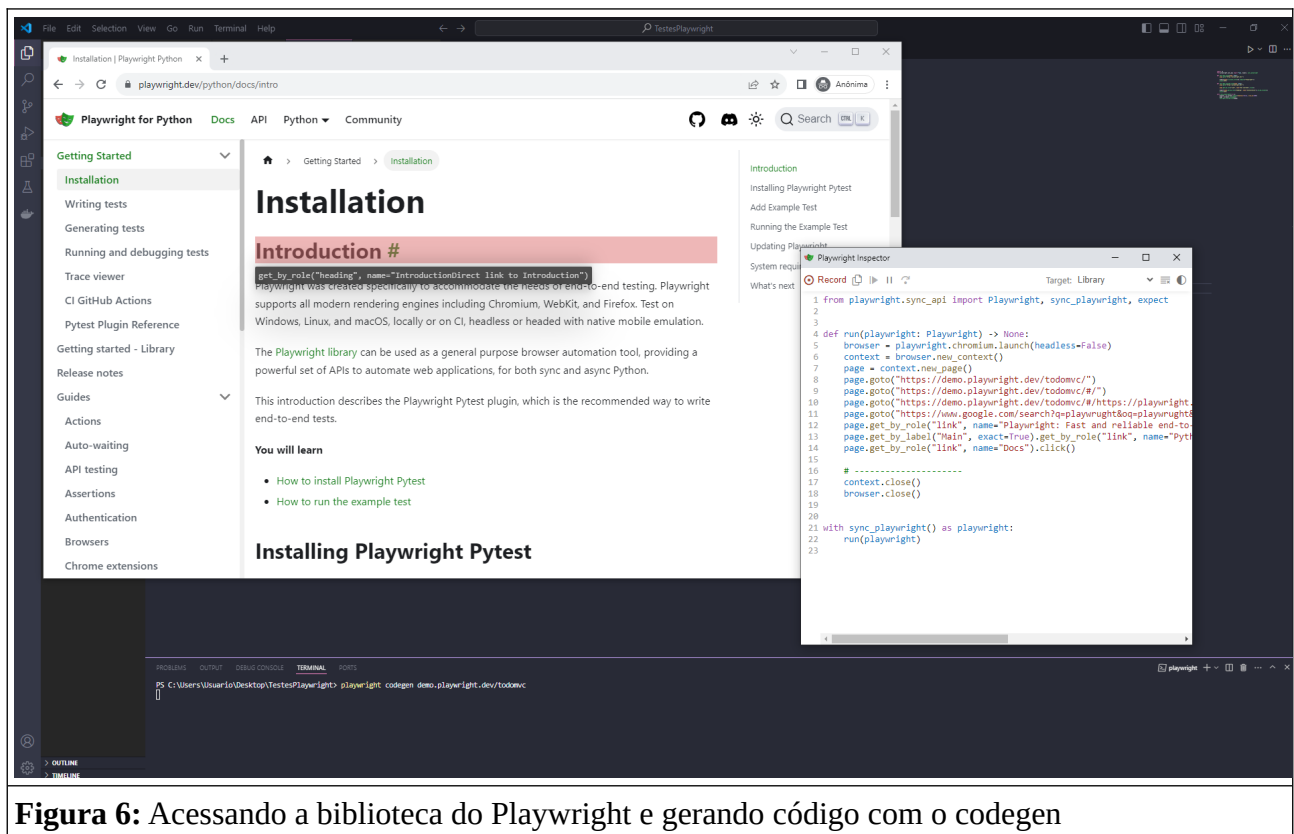
test_shopping_cart()
```

**Figura 5:** Realização de testes com Playwright com pesquisa na Amazon

## Gerando códigos com Codegen

Para quem não dominar a linguagem ou que precisa de mais pressa no Web Scraping, o Playwright também possui o comando de codegen, onde um navegador é aberto com uma pequena janela que vai gerando código de acordo com a suas interações na tela, impulsionando ainda mais o desenvolvimento.

De forma óbvia, a geração dos códigos nem sempre sairá de forma perfeita, mas é ótima para auxiliar a começar um projeto e guiar pessoas leigas na biblioteca.



**Figura 6:** Acessando a biblioteca do Playwright e gerando código com o codegen

## Síntese

É possível utilizar o Playwright nos mais diversos cenários, criando automações de forma fácil, em qualquer site e integráveis com várias outras bibliotecas, tendo assim um potencial enorme para criar testes de forma rápida e eficiente, tentando conquistar e abranger cada vez mais funcionalidades e formas de realizar testes, com suporte para várias linguagens e navegadores e até mesmo abraçando a geração de códigos automatizada por meio do codegen.

Por ser uma biblioteca recente, ainda existe um grande potencial para expansão e ainda integração com inteligência artificial e outros recursos que estão crescendo no mercado. É uma grande mão na roda para potencializar o desenvolvimento web e o cenário de automações.

## Bibliografia

REHKOPF, Max. Testes de software automatizados. Disponível em: <https://www.atlassian.com/br/continuous-delivery/software-testing/automated-testing>. Acesso em 08 jun. 2024.

PLAYWRIGHT. Documentação oficial do Playwright. Disponível em: <https://playwright.dev/python/>. Acesso em: 08 jun. 2024.

MICROSOFT. Use o Playwright para automatizar e testar no Microsoft Edge. Disponível em: <https://learn.microsoft.com/pt-br/microsoft-edge/playwright/>. Acesso em: 08 jun. 2024.

Martins, Pedro. Iniciando seus testes com Playwright. Disponível em: <https://medium.com/engenharia-arquivei/iniciando-seus-testes-com-playwright-573c0f0cbfa8>. Acesso em: 08 jun. 2024.

# Capítulo 09

# Capítulo 09

**Gleison Rodrigo Moura da Silva**  
**gleison.silva4@fatec.sp.gov.br**

## Explorando testes com o Xunit

Este capítulo tem como objetivo explorar em detalhes a integração do TDD (Desenvolvimento Orientado por Testes) com o Xunit, uma estrutura de teste unitário moderna e amplamente utilizada na plataforma .NET, e verificar como o uso e a integração entre esses dois conceitos podem ser benéficos no desenvolvimento de software.

Veremos que a combinação dessas duas metodologias não apenas eleva a qualidade do código, mas também aprimora a eficiência do processo de desenvolvimento, oferecendo aos desenvolvedores uma abordagem estruturada e sistemática para a criação de software de alta qualidade.

## Desenvolvimento Orientado a Testes (TDD)

No desenvolvimento de software, garantir a qualidade do código é crucial. O Desenvolvimento Orientado por Testes (TDD) destaca-se como uma prática essencial, promovendo a criação de software robusto desde o início.

TDD, ou *Test-Driven Development*, coloca os testes unitários no centro do desenvolvimento. Os desenvolvedores escrevem testes antes de implementar funcionalidades, resultando em um código mais estável e resistente a mudanças. Cada adição ou alteração é validada por testes automatizados, promovendo um desenvolvimento ágil e confiável.

TDD traz vários benefícios, como a promoção da confiabilidade do código e a identificação rápida de problemas. Além disso, incentiva um design modular e flexível, facilitando a manutenção e evolução do software.

## O que é C#?

Durante o desenvolvimento da plataforma .NET, as bibliotecas foram inicialmente escritas em Simple Managed C (SMC), com um compilador próprio. Em janeiro de 1999, Anders Hejlsberg foi escolhido pela Microsoft para liderar uma equipe de desenvolvimento e criar uma nova linguagem, inicialmente chamada Cool. Em 2000, na Professional Developers Conference (PDC), o projeto .NET foi apresentado ao público e a linguagem Cool foi renomeada para C#.

C# (pronunciado "C Sharp") é uma linguagem de programação muito popular, conhecida por sua baixa curva de aprendizado e simplicidade, sem perder a potência. É a principal linguagem do .NET Framework, o framework de desenvolvimento da Microsoft.

C# é uma linguagem multiparadigma, suportando orientação a objetos, com conceitos como encapsulamento, herança e polimorfismo. É fortemente tipada e case-sensitive, ou seja, diferencia entre letras maiúsculas e minúsculas. C# facilita o desenvolvimento com recursos que aumentam a produtividade dos desenvolvedores.

## Sintaxe do C#

A sintaxe do C# se destaca por sua clareza e simplicidade, tornando o aprendizado e a escrita de código uma experiência intuitiva. Um exemplo bem conhecido, é o famoso "Hello World", que oferece uma introdução à linguagem.

```
using System;

namespace HelloApp
{
    0 referências
    class Program
    {
        0 referências
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Figura 01: Exemplo de código 'Hello World' em C# utilizando a sintaxe básica do programa

## Xunit

O Xunit é um framework de testes unitários para .NET, desenvolvido para facilitar a escrita e execução de testes automatizados em projetos C#. Desenvolvido por Jim Newkirk e Brad Wilson como uma alternativa aos frameworks de testes existentes na época, como NUnit e MSTest. A primeira versão do Xunit saiu em 2007, e desde então tem sido amplamente adotada pela comunidade de .NET.

# Xunit no Desenvolvimento

Uma das principais vantagens do Xunit é sua flexibilidade. O Xunit permite que os desenvolvedores escrevam testes com uma sintaxe clara e intuitiva, e oferece suporte a várias funcionalidades avançadas, como testes parametrizados e asserções customizadas. Além disso, sua extensibilidade facilita a criação de novos tipos de asserções e configurações personalizadas de testes.

O Xunit se integra perfeitamente com ferramentas de build e pipelines de CI/CD, como Azure DevOps, Jenkins e GitHub Actions. Isso permite que os testes sejam executados automaticamente durante o processo de integração contínua, garantindo que o código permaneça robusto e livre de regressões.

Com o aumento da utilização de operações assíncronas no desenvolvimento moderno, o Xunit oferece suporte nativo para testes assíncronos. Isso significa que os desenvolvedores podem testar métodos assíncronos de forma eficiente, garantindo que o código funcione corretamente em cenários de alta concorrência e paralelismo.

## Exemplo Prático

Para ilustrar o funcionamento dos testes em um ambiente de desenvolvimento, apresentaremos a seguir um exemplo de uma aplicação de testes em um projeto.

Imagine que você precise criar uma calculadora e garantir que suas operações funcionem corretamente. Para isso, usaremos testes unitários.

Vamos iniciar este processo criando nossa aplicação. Ao abrir o Visual Studio, veremos vários tipos de projetos disponíveis, mas vamos utilizar o tipo de projeto “Biblioteca de Classes”:

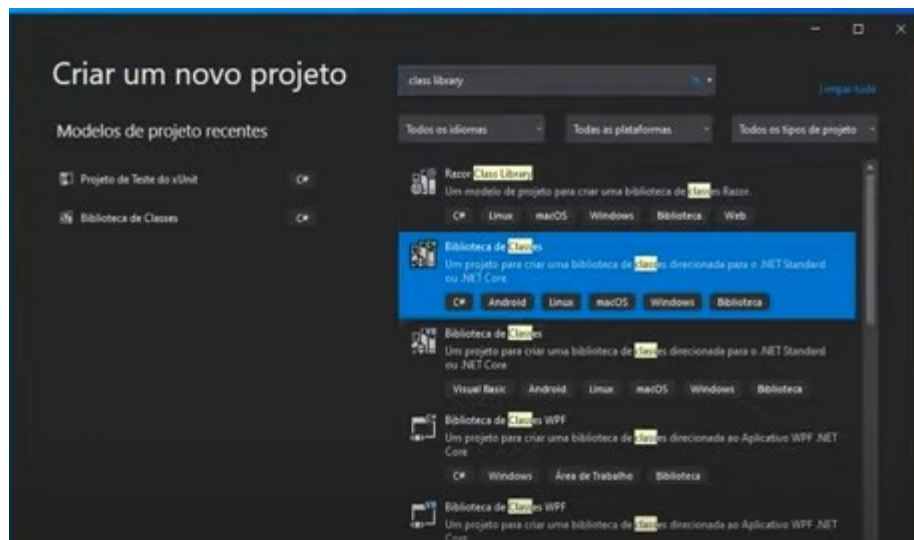


Figura 02: Criação de um novo projeto selecionando a opção “Biblioteca de Classes”

Após escolher o tipo de projeto, vamos chamá-lo de "Calculator" e selecionar a versão do .NET que utilizaremos no desenvolvimento desta aplicação (neste exemplo, será utilizada a versão .NET 5). Realizando essas configurações, o ambiente de desenvolvimento será preparado, e estaremos prontos para começar a implementar as funcionalidades da calculadora. Vamos criar os métodos em nosso projeto, contendo as quatro operações básicas, representadas pelos seguintes nomes: Addition, Subtraction, Multiplication e Division.

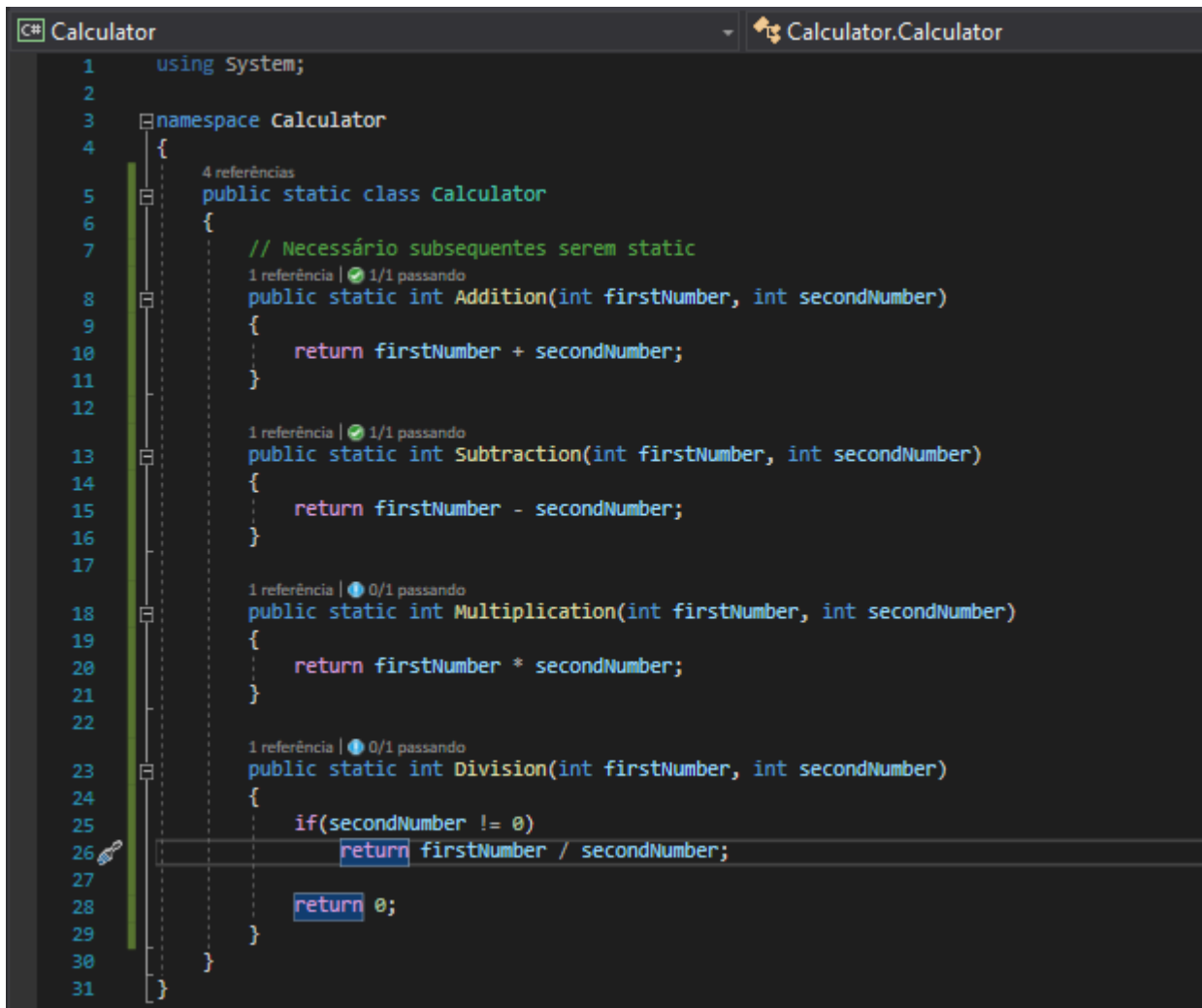


Figura 03: Métodos representando as quatro operações de Calculadora

Agora a classe “Calculator”, possui os métodos necessários para que possamos inserir os valores desejados e ocorra o processamento das informações.

Observe que há um ícone acima destes métodos, representado por uma cor azul ou verde. Ambos indicam referências a testes no projeto. O ícone azul representa que o teste ainda não foi executado, enquanto o ícone verde indica que o teste foi executado com sucesso. Em alguns casos, pode haver um ícone vermelho acima do método, indicando que o método não passou em algum teste executado anteriormente.

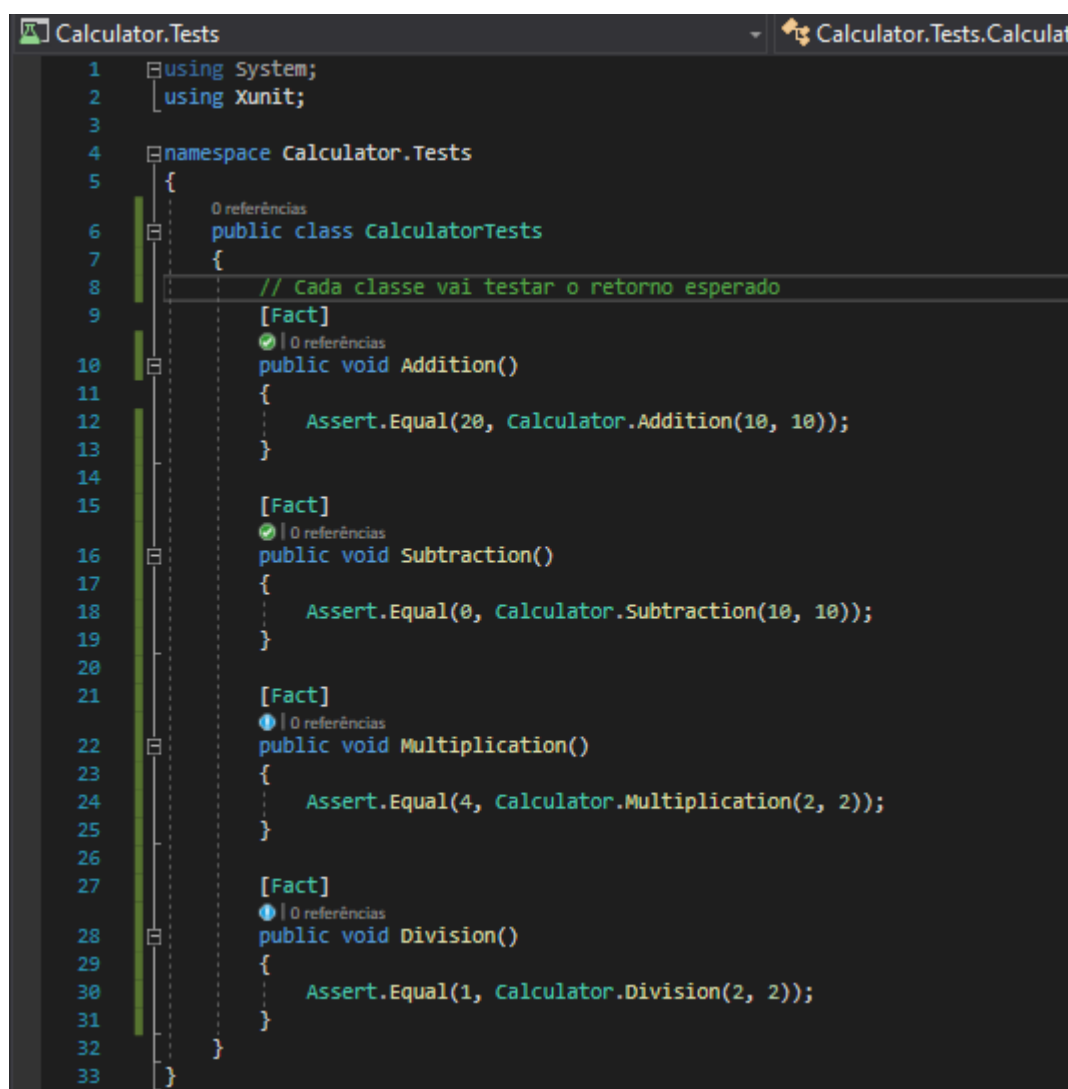
Ao criar os métodos, por padrão esses ícones não irão aparecer, pois para que isso ocorra,



precisamos criar um outro projeto que será responsável por assegurar que esses métodos estão funcionando como o esperado.

Sendo assim, o próximo passo será criar um projeto de testes. Na Solution "Calculator", clique com o botão direito e selecione a opção "Adicionar", e em seguida "Novo Projeto". Nomeie-o como "Calculator.Tests" e renomeie a classe deste novo projeto como "Calculator.Tests".

Agora podemos criar os métodos de teste que irão validar os quatro métodos que criamos em nossa calculadora anteriormente, verificando se seus tipos de retorno são válidos. Por exemplo, no método de adição, esperamos que  $10 + 10$  seja igual a 20, como mostrado a seguir:



```
1  using System;
2  using Xunit;
3
4  namespace Calculator.Tests
5  {
6      0 referências
7      public class CalculatorTests
8      {
9          // Cada classe vai testar o retorno esperado
10         [Fact]
11         0 referências
12         public void Addition()
13         {
14             Assert.Equal(20, Calculator.Addition(10, 10));
15         }
16
17         [Fact]
18         0 referências
19         public void Subtraction()
20         {
21             Assert.Equal(0, Calculator.Subtraction(10, 10));
22         }
23
24         [Fact]
25         0 referências
26         public void Multiplication()
27         {
28             Assert.Equal(4, Calculator.Multiplication(2, 2));
29         }
30
31         [Fact]
32         0 referências
33         public void Division()
34         {
35             Assert.Equal(1, Calculator.Division(2, 2));
36         }
37     }
38 }
```

Figura 04: Métodos responsáveis pelos testes unitários da aplicação “Calculator”

Para que essa classe “CalculatorTests” possa criar uma instância da classe “Calculator”, será preciso atribuir uma dependência ao nosso projeto de testes. Para fazer isso, vamos clicar com o botão direito no projeto de teste, clicar em “Dependências do Projeto” e escolher a caixa com o nome do nosso projeto da calculadora, no caso será o projeto “Calculator”.

No exemplo da imagem, estamos utilizando o atributo [Fact], que é usado para marcar um método de teste como um fato (fact), ou seja, um teste que sempre deverá ser verdadeiro. É o tipo mais simples de teste no Xunit e é ideal para testar casos simples e diretos.

Apesar de não estarmos utilizando, existe também o atributo [Theory]. Este atributo é usado para marcar métodos de teste que aceitam argumentos. Ele permite criar testes parametrizados, nos quais o mesmo teste é executado com diferentes conjuntos de dados. É útil para verificar comportamentos que devem ser validados em vários cenários.

## Testes

Para testar se os métodos estão funcionando e retornando valores corretamente, com o botão direito vamos executar o projeto “Calculator.Tests”. Após a execução, vemos o resultado em tela para cada um dos métodos testados, indicando que passaram (ícone verde):

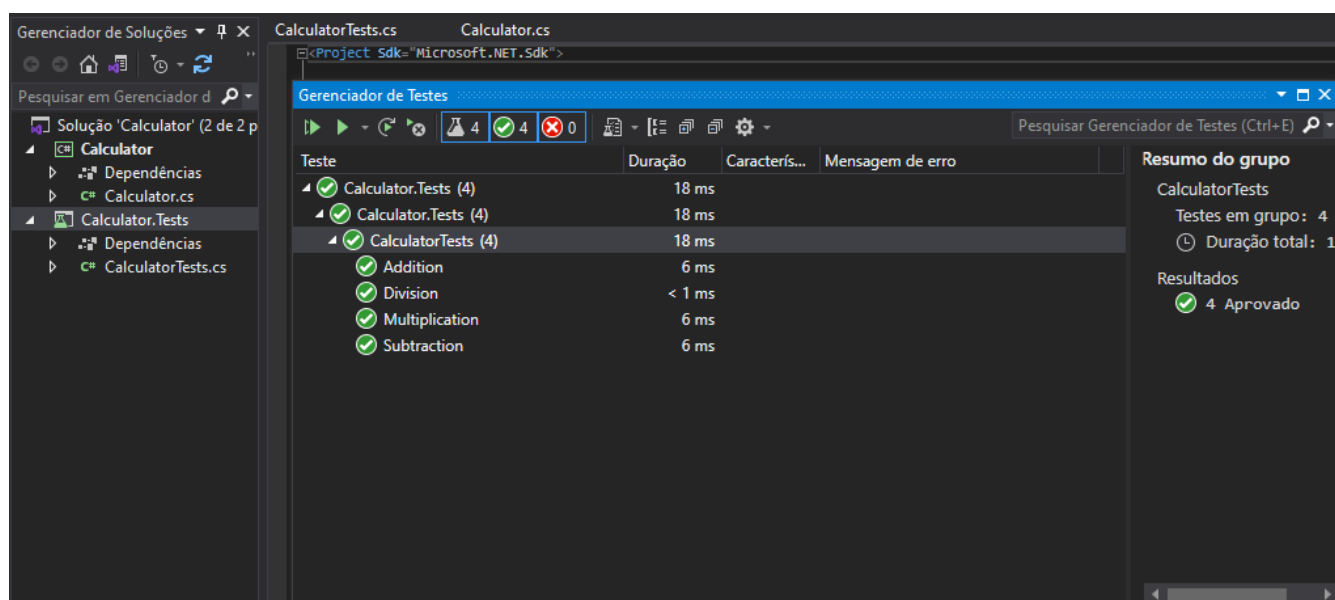


Figura 05: Resultado dos testes executados de “Calculator.Tests” sem falhas

Vamos fazer outro teste. Suponhamos que, no método de verificação Subtraction (no arquivo "Calculator.Tests"), tenhamos usado o método incorreto Calculator.Addition() em vez de Calculator.Subtraction(). Ao executar os testes novamente, perceberemos que um teste falhou (ícone vermelho), pois o resultado final foi diferente do esperado pelo teste (realizou-se uma soma em vez de uma subtração).

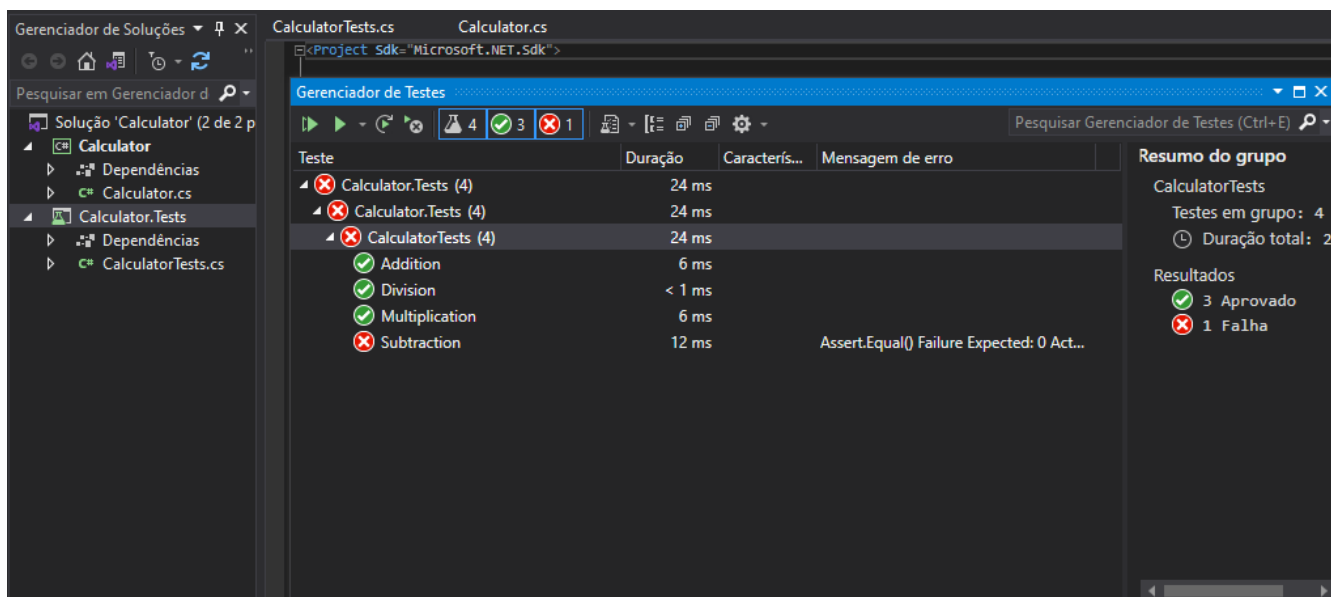


Figura 06: Resultado dos testes executados de “Calculator.Tests” com falhas

Quando os testes não passam, é importante identificar a causa do problema e corrigi-lo. Uma possível abordagem é revisar o código do método sob teste e garantir que esteja implementado corretamente, seguindo as especificações e os requisitos do sistema. Além disso, podemos verificar se os dados de entrada fornecidos aos métodos de teste estão corretos e se os resultados esperados foram definidos de maneira apropriada.

Se o problema persistir, podemos recorrer à depuração do código, utilizando ferramentas disponíveis no ambiente de desenvolvimento para identificar possíveis erros lógicos ou de implementação. Outra estratégia é revisar os testes unitários, garantindo que cubram todos os cenários relevantes e que os casos de teste sejam abrangentes o suficiente para validar o comportamento do código em diferentes situações. Ao seguir essas práticas e abordagens, podemos resolver eficientemente os problemas encontrados durante a execução dos testes unitários.

## Síntese

Neste capítulo, vimos um pouco sobre como a utilização do Xunit em conjunto com TDD promove um desenvolvimento mais ágil e confiável, permitindo que os desenvolvedores criem software de alta qualidade de maneira eficiente. A flexibilidade, extensibilidade e suporte a testes assíncronos do Xunit o tornam uma ferramenta poderosa no arsenal de qualquer desenvolvedor .NET.

## Bibliografia

- [1] C#. Documentação oficial do C#. Disponível em: <https://learn.microsoft.com/en-us/dotnet/csharp/>.
- [2] C#. Introdução ao C#. Disponível em: <https://www.devmedia.com.br/guia/linguagem-csharp/38152>
- [3] C#. Documentação do Xunit. Disponível em: <https://xunit.net/>
- [4] C#. Guia de TDD com Xunit. Disponível em: <https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test>
- [5] C#. Artigo sobre Xunit no Medium. Disponível em: <https://medium.com/@mateusquintino/escrevendo-testes-unit%C3%A1rios-em-net-com-xunit-aa1ec1d0b77b>

# Capítulo 10

# Capítulo 10

André Felipe de Paula  
andreflp066@gmail.com

## Introdução

Nesse presente capítulo, vamos aprender sobre o desenvolvimento orientado a testes em *front-end*, compreendendo sua importância no processo de desenvolvimento de aplicações. Vamos realizar a introdução a ferramenta *Cypress* e demonstraremos sua aplicabilidade por meio de um projeto de exemplo.

## Desenvolvimento de testes em Front-End

Para compreender os testes, é preciso entender o que é *front-end*. Em desenvolvimento *web*, o *front-end* é a parte “visível” de uma aplicação *web* em que os usuários podem interagir por meio de cliques de botões, preenchimento de formulários, dentre outras formas de navegação.

De uma forma resumida, o *front-end* é responsável por garantir que o usuário tenha sua experiência por meio de como os elementos são dispostos na tela, além da otimização deles conforme o dispositivo do usuário.

Escrever testes em *front-end* é essencial para que se possa garantir a qualidade das aplicações *web*. Atualmente as empresas e usuários demandam cada vez mais por experiências melhores e mais fluidas, sendo preciso garantir que tudo funcione conforme o planejado durante a navegação do usuário. Desta maneira, os testes em *front-end* não apenas auxiliam na identificação de falhas e erros com antecedência, mas também é uma forma de garantir a segurança de que as atualizações e modificações não impactem negativamente na experiência quando os usuários recebem o *software*.

Os *designers* de interface e desenvolvedores precisam trabalhar em conjunto para que o sistema seja otimizado e intuitivo para atingir os objetivos de uma boa navegação por parte do usuário, para que ele volte a utilizar o produto a partir dessa boa experiência, sem gerar frustrações.

Por fim, iremos explorar os benefícios de escrever testes no *front-end*, destacando como essa abordagem contribui para o desenvolvimento de aplicações *web* confiáveis.

## O Cypress

O *Cypress* [1] é uma ferramenta de código aberto muito utilizada para realizar testes de *front-end*. Ele foi criado com o objetivo de atender às necessidades de desenvolvedores e analistas de QA que necessitam realizar testes de maneira simples e intuitiva.

Um dos diferenciais desta ferramenta é oferecer uma experiência de criação de testes com instalação e implementação simplificadas, fornecendo recursos para facilitar esse desenvolvimento.

Além disso, o *Cypress*, também fornece a visualização de resultados em tempo real, permitindo a verificação da integridade dos elementos dispostos na tela e suas interações, como cliques em botões, navegações entre páginas e preenchimento de formulários [1].

O *Cypress* [2], também possui um destaque em seu funcionamento, tendo a capacidade de lidar com comandos nativos. É possível fazer a leitura do tráfego de rede, manipular dados enviados e recebidos, inclusive o controle de todo o processo de automação. Ademais, ele também oferece um painel de métricas que permite a visualização dos testes criados e os seus resultados.

Com o *Cypress*, é possível realizar diversos tipos de testes, incluindo *end-to-end*, componente, integração e unitários. Fazendo com que seja uma opção atrativa para equipes de desenvolvimento que querem melhorar a qualidade das aplicações *web*, já que a ferramenta promete a simplicidade no uso [2].

Isso posto, é preciso entender como a configuração do ambiente pode ser feita com ele. É importante destacar os seguintes pontos quanto a sua aplicabilidade:

- Configuração inicial: o *Cypress* pode ser instalado em projetos pré-existentes ou novos a partir do *Node.js*, utilizado para instalação de dependências. Instalando através do *Node Package Manager*, é possível começar a utilizá-lo [3];
- Integração no projeto: ele pode ser integrado com diferentes tipos de projetos *web*, como o *Angular* [4] ou *React* [5], *frameworks* populares para a construção de páginas para o *front-end*;
- Criação de testes: o *Cypress* tem uma sintaxe amigável, com o objetivo de haver simplicidade no processo de criação dos testes. Utilizando somente o *Javascript*, durante a primeira inicialização da ferramenta, ele oferece exemplos de testes que são criados automaticamente, podendo escolher o componente que deseja que o teste de exemplo seja criado [5].

O *Cypress* também possui uma comunidade ativa, em formato de *blog*, em que os desenvolvedores podem publicar novidades sobre a ferramenta, divulgar *webinars* e organizar eventos virtuais para compartilhar experiências [6]. A empresa criadora da ferramenta possui um *GitHub* em que é disponibilizado o código onde usuários podem relatar seus problemas e dar sugestões de melhorias diretamente no código fonte, por ser uma solução *open source* [7].

## Exemplificando

Para demonstrar melhor como o *Cypress* funciona, utilizaremos de um exemplo prático. Foi selecionado um projeto [8] escrito na linguagem de programação *Typescript*, com *framework React* que constitui um componente simples.

```

PS <redacted> \projeto> npm create vite@latest
Need to install the following packages:
create-vite@5.2.3
Ok to proceed? (y) y

> npx
> create-vite

✓ Project name: ... projeto_ebook
✓ Select a framework: » React
✓ Select a variant: » JavaScript

Scaffolding project in <redacted> \projeto\projeto_ebook...

```

Figura 1: Inicialização do projeto usando Vite [9]

A partir dessa criação, cria-se um componente simples: um formulário de login. Para isso, pode-se utilizar o *Typescript* e *React* [8] para estruturá-los e *CSS* para estilização. Nesse componente, tem-se dois campos: um para *email* e um para senha. Também há um botão que ao clicar irá simular um *login*.

```

Login.tsx

import { useState } from 'react'
import './Login.css'
import { BrowserRouter as Routers, Routes, Route, Navigate } from 'react-router-dom'
import HomePage from './Home'

// eslint-disable-next-line @typescript-eslint/no-unused-vars
function Login(){

  const [email, setEmail] = useState("")
  const [password, setPassword] = useState("")
  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  const [emailError, setEmailError] = useState("")
  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  const [passwordError, setPasswordError] = useState("")
  // const navigate = useNavigate();

  const onClick = () =>{
    setEmailError("")
    setPasswordError("")

    if("" === email){
      setEmailError("Please enter your email")
      return
    }

    if("" === password)
    {
      setPasswordError("Please enter a password")
      return
    }
    if(password.length<7){
      setPasswordError("password must be 8 character or longer")
      return
    }
    if(!/^[^@\s]+@([\w-]+\.)+[\w-]{2,4}$/i.test(email)){
      setEmailError("please enter a valid email address")
      return
    }

    if(email){
      return(
        <>
        {
          <Routers>
          {
            <Routes>
            <Route path="/" element={<HomePage />}>

```



```
)
}

export default Login
```

Figura 2: Criação do componente em React [8]

```

Login.css

html {
  height: 100%;
}
body {
  margin: 0;
  padding: 0;
  font-family: sans-serif;
  background: linear-gradient(#141e30, #446891);
}

.login-box {
  position: absolute;
  top: 50%;
  left: 50%;
  width: 400px;
  padding: 40px;
  .login-box .user-box label {
    position: absolute;
    top: 0;
    left: 0;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    pointer-events: none;
    transition: .5s;
  }

  .login-box .user-box input:focus ~ label,
  .login-box .user-box input:valid ~ label {
    top: -20px;
    left: 0;
    color: #03e9f4;
    font-size: 12px;
  }
}

.login-box a span {
  position: absolute;
  display: block;
}

.login-box a span:nth-child(1) {
  top: 0;
  left: -100%;
  width: 100%;
  height: 2px;
  background: linear-gradient(90deg, transparent, #03e9f4);
  animation: btn-anim1 1s linear infinite;
}

@keyframes btn-anim1 {
  0% {
    left: -100%;
  }
  50%, 100% {
    left: 100%;
  }
}

.login-box a span:nth-child(2) {
  top: -100%;
  right: 0;
  width: 2px;
  height: 100%;
  background: linear-gradient(180deg, transparent, #03e9f4);
  animation: btn-anim2 1s linear infinite;
  animation-delay: .25s;
}

@keyframes btn-anim2 {
  .mainContainer {
    flex-direction: column;
    display: flex;
    align-items: center;
    justify-content: center;
    height: 100vh;
  }

  .titleContainer {
    display: flex;
    flex-direction: column;
    font-size: 64px;
    font-weight: bolder;
    align-items: center;
    justify-content: center;
  }

  .resultContainer, .historyItem {
    flex-direction: row;
    display: flex;
    width: 400px;
    align-items: center;
    justify-content: space-between;
  }
}

```

```

.historyContainer {
  flex-direction: column;
  display: flex;
  height: 200px;
  align-items: center;
  flex-grow: 5;
  justify-content: flex-start;
}

.buttonContainer {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  height: 260px;
}

.inputContainer {
  display: flex;
  flex-direction: column;
  align-items: flex-start;
  justify-content: center;
}

.inputContainer > .errorLabel {
  color: red;
  font-size: 12px;
}

```

```

@keyframes btn-anim3 {
  0% {
    right: -100%;
  }
  50%,100% {
    right: 100%;
  }
}

.login-box a span:nth-child(4) {
  bottom: -100%;
  left: 0;
  width: 2px;
  height: 100%;
  background: linear-gradient(360deg, transparent, #03e9f4);
  animation: btn-anim4 1s linear infinite;
  animation-delay: .75s
}

@keyframes btn-anim4 {
  0% {
    bottom: -100%;
  }
  50%,100% {
    bottom: 100%;
  }
}

```

```

.mainContainer {
  flex-direction: column;
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100vh;
}

.titleContainer {
  display: flex;
  flex-direction: column;
  font-size: 64px;
  font-weight: bolder;
  align-items: center;
  justify-content: center;
}

.resultContainer, .historyItem {
  flex-direction: row;
  display: flex;
  width: 400px;
  align-items: center;
  justify-content: space-between;
}

```

```

@keyframes btn-anim3 {
  0% {
    right: -100%;
  }
  50%,100% {
    right: 100%;
  }
}

.login-box a span:nth-child(4) {
  bottom: -100%;
  left: 0;
  width: 2px;
  height: 100%;
  background: linear-gradient(360deg, transparent, #03e9f4);
  animation: btn-anim4 1s linear infinite;
  animation-delay: .75s
}

@keyframes btn-anim4 {
  0% {
    bottom: -100%;
  }
  50%,100% {
    bottom: 100%;
  }
}

```

```

.inputBox {
  height: 48px;
  width: 400px;
  font-size: large;
  border-radius: 8px;
  border: 1px solid grey;
  padding-left: 8px;
}

input[type="button"] {
  border: none;
  background: cornflowerblue;
  color: white;
  padding: 12px 24px;
  margin: 8px;
  font-size: 15px;
  border-radius: 8px;
  cursor: pointer;
}

```

Figura 3: Estilização do componente em CSS [8]

Com isso, temos essa tela de *login*:

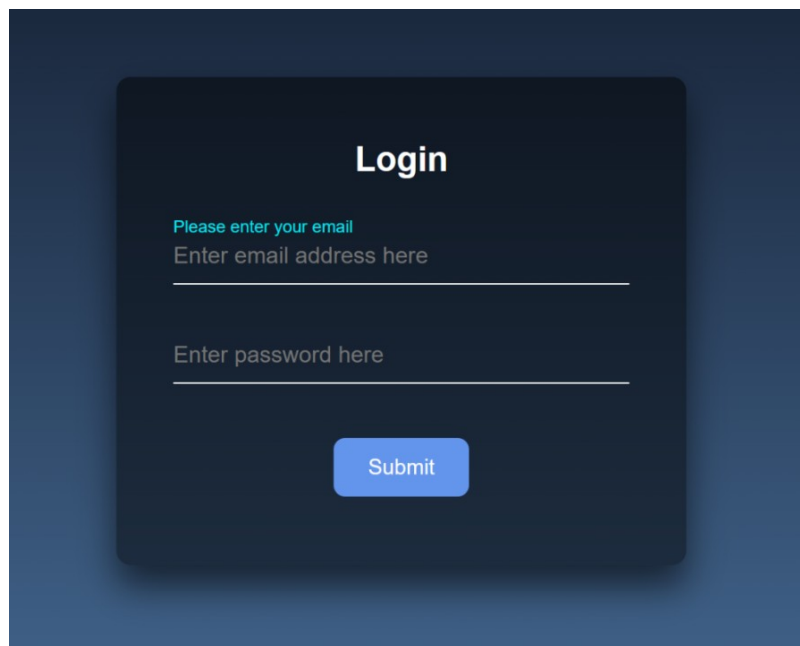


Figura 4: Tela de login [8]

Agora, temos uma estrutura de pastas com o componente em *React – Typescript* e a estilização desse componente, semelhante ao exemplo ilustrado na figura 18 [9].

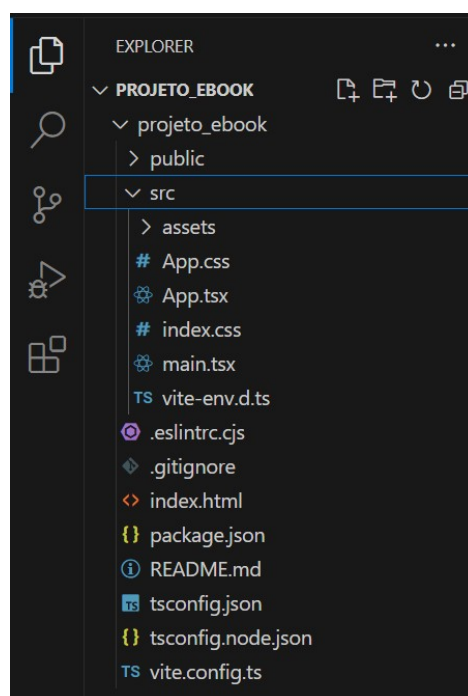


Figura 5: Exemplo de estrutura de projeto [9]

Para começar a utilizar o *Cypress*, é necessário instalar o seu pacote por meio do *NPM* [10], que se trata de uma ferramenta para gerenciar pacotes de projetos.

```
$ npm install cypress --save-dev

added 172 packages, and audited 173 packages in 51s

35 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
npm notice
npm notice New patch version of npm available! 9.6.4 -> 9.6.6
npm notice Changelog: <https://github.com/npm/cli/releases/tag/v9.6.6>
npm notice Run 'npm install -g npm@9.6.6' to update!
npm notice
```

Figura 6: Instalação do Cypress [11]

Nesse instante, é possível rodar o *Cypress* e começar a escrever os testes. Para isso, teremos que abrir a interface do *Cypress* e selecionar a opção de *Component Testing*, para testar a renderização e funcionalidade básicas do componente de formulário.

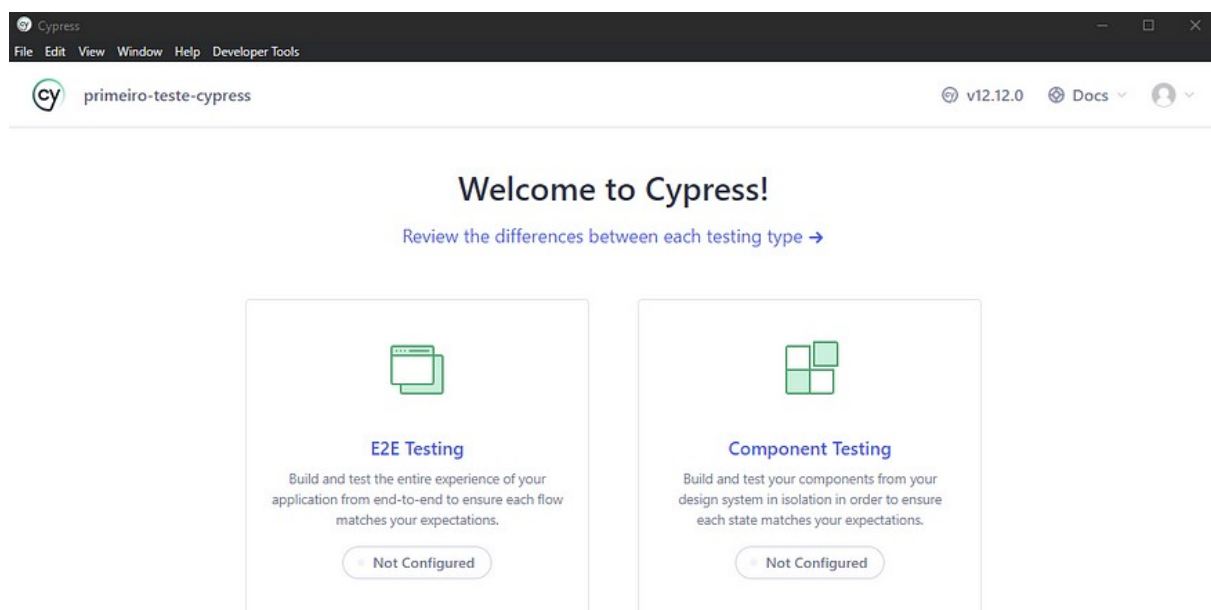


Figura 7: Tela inicial do Cypress Test Runner [11]

Vai ser realizado um teste de componente, que irá avaliar o comportamento do componente de maneira isolada. No exemplo do formulário de *login*, é preciso testar a consistência na renderização dele, assim como se os estados estão de acordo com o esperado.

Cria-se um arquivo de teste na mesma pasta que o componente e levantamos as asserções. Para que o *Cypress* reconheça os arquivos de teste, é necessário haver a extensão *(.cy)*, portanto, como o projeto está utilizando *React* e *Typescript*, é preciso utilizar a extensão *(.cy.tsx)* para que seja configurado corretamente.

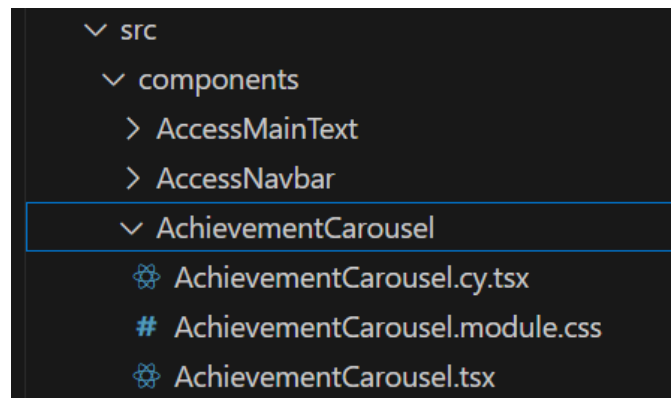


Figura 8: Estrutura de pastas com Cypress [9]

Agora, podemos começar a escrever os nossos testes. Vamos criar um teste simples, que apenas verifica se o componente está sendo renderizado em tela com as informações adequadas.

```
// monta o componente AchievementCarousel e verifica se o texto "Iniciante Ambiental" é renderizado corretamente.
describe("AchievementCarousel", () => { it("Should render the achievement Iniciante Ambiental.", () => {
  cy.mount(
    <ThemeProvider>
      <Router>
        <AchievementCarousel />
      </Router>
    </ThemeProvider>
  );

  // verifica se a conquista "Iniciante Ambiental" é renderizada.
  cy.contains("Iniciante Ambiental").should("exist");
});

// Monta o componente AchievementCarousel dentro do ThemeProvider e Router o que permite renderizar componentes de React para testes.
describe("AchievementCarousel", () => { it("Should render the achievement Ativista Compromissado.", () => {
  cy.mount(
    <ThemeProvider>
      <Router>
        <AchievementCarousel />
      </Router>
    </ThemeProvider>
  );
});
```

Figura 9: Criação do teste com Cypress [9]

Dessa maneira, já realizamos o nosso primeiro teste no Cypress. Se acessarmos a interface da ferramenta, é possível ver que o teste está funcionando e com as asserções conforme construímos na imagem anterior.

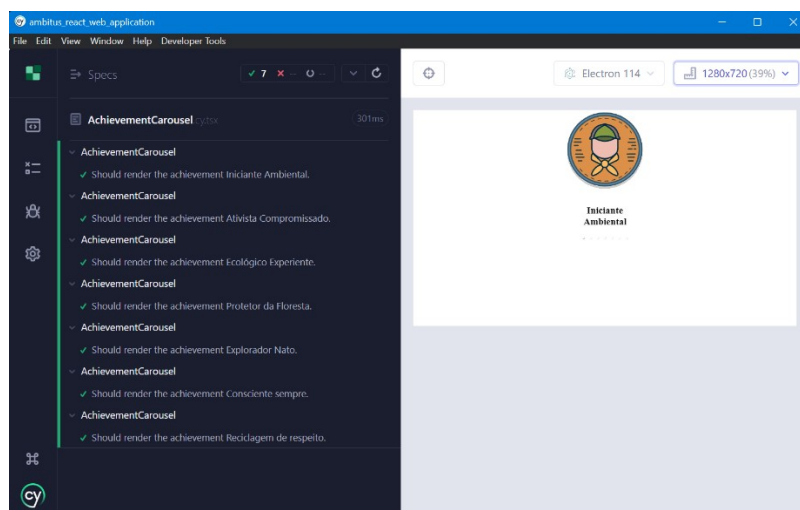


Figura 10: Resultado do teste em Cypress

Existem várias formas de fazer um teste de componente com Cypress. É possível complementar esse formulário de login e verificar visitas à página, cliques etc. Para explorar as diversas

possibilidades que o *Cypress* oferece, a documentação dele [1] traz exemplos de integração para validar componentes isoladamente, usando *Angular*, *Vue* e *Svelte*, além do *React* que foi usado de modelo nesse capítulo.

## Síntese

Nesse presente capítulo, foi destacado a importância da realização de testes em *Front-End*, que mostram aos desenvolvedores se os componentes estão agindo de acordo com o esperado isoladamente, fornecendo uma melhor experiência aos usuários finais de uma aplicação.

Em resumo, também foi exemplificado como é realizado um projeto que faz uso do *Cypress*, mostrando como uma ferramenta de testes pode ser simples e crucial, garantindo a qualidade do produto, sem gerar impacto negativo nas interações com a interface em geral.

## Bibliografia

- [1] CYPRESS. **Documentação oficial do Cypress**. Disponível em: <https://docs.cypress.io>. Acesso em: 16 maio 24.
- [2] TESTING COMPANY. **Conheça o cypress e suas vantagens para automação de testes**. Disponível em: <https://testingcompany.com.br/blog/conheca-o-cypress-e-suas-vantagens-para-automacao-de-testes>. Acesso em: 16 maio 24.
- [3] NODE.JS. **Documentação oficial do Node.js**. Disponível em: <https://nodejs.org/en/docs>. Acesso em: 17 maio 24.
- [4] ANGULAR. **Documentação oficial do Angular**. Disponível em: <https://angular.io>. Acesso em: 17 maio 24.
- [5] REACT. **Documentação oficial do React**. Disponível em: <https://react.dev>. Acesso em: 17 maio 24.
- [6] CYPRESS. **Página da comunidade e blog oficial do Cypress**. Disponível em: <https://www.cypress.io/events>. Acesso em: 20 maio 24.
- [7] CYPRESS. **Página oficial do GitHub do Cypress**. Disponível em: <https://github.com/cypress-io>. Acesso em: 20 maio 24.
- [8] TECH INSIGHTS. **Create a Login Form in React TypeScript**. Disponível em: <https://reactconf.org/create-a-login-form-in-react-typescript/>. Acesso em: 31 maio 24.
- [9] CYPRESS. **Página da comunidade e blog oficial do Cypress**. Disponível em: <https://www.cypress.io/events>. Acesso em: 20 maio 24.
- [10] NPM. **Documentação oficial do NPM**. Disponível em: <https://docs.npmjs.com>. Acesso em: 04 jun. 24.
- [11] THIAGO CASTILHO. **Introdução ao Cypress: Um Guia Passo a Passo para Testes de Aplicações Web**. Disponível em: <https://medium.com/@thiago-castilho/introducao-ao-cypress-um-guia-passo-a-passo-para-testes-de-aplicacoes-web-6be295b0557>. Acesso em: 04 jun. 24.

# Capítulo 11

# Capítulo 11

Joice Fernanda Rodrigues  
joice.rodrigues2@fatec.sp.gov.br

## Introdução

Os testes unitários são componentes essenciais do desenvolvimento de software de alta qualidade. Eles garantem que cada unidade de código funcione conforme o esperado, isolando-a de outras partes do sistema. No entanto, escrever testes unitários eficazes pode ser um desafio, especialmente quando se trata de testar código que depende de outras classes, serviços ou APIs externas.

Mocking é uma técnica de teste de software que aborda esse desafio. Ela envolve a criação de objetos simulados, também conhecidos como mocks, que imitam o comportamento de dependências reais. Isso permite que os desenvolvedores testem o código em isolamento, sem a necessidade de configurar e gerenciar dependências reais.

O Kotlin oferece diversas bibliotecas de mocking robustas, como Mockito-Kotlin e MockK, que facilitam a criação e o uso de mocks em testes unitários. Essas bibliotecas fornecem APIs intuitivas e flexíveis para criar mocks, definir seu comportamento e verificar suas interações com o código em teste.

## Benefícios do Mocking

O uso de mocks em testes unitários oferece diversos benefícios:

**Isolamento de Testes:** Mocks permitem que os testes unitários sejam executados de forma independente, sem a necessidade de configurar e gerenciar dependências reais. Isso torna os testes mais confiáveis e fáceis de manter.

**Maior Controle:** Com mocks, os desenvolvedores podem controlar o comportamento das dependências, definindo respostas específicas para diferentes cenários de teste. Isso permite que eles testem o código em uma variedade de condições, sem depender do estado de dependências reais.

**Melhoria da Cobertura de Código:** Mocks podem ser usados para aumentar a cobertura de código dos testes unitários, alcançando partes do código que seriam difíceis ou impossíveis de testar com dependências reais.

**Desenvolvimento Orientado a Testes:** O uso de mocks facilita a implementação do desenvolvimento orientado a testes (TDD), onde os testes unitários são escritos antes do código de produção. Isso promove um processo de desenvolvimento mais disciplinado e focado na qualidade.



## Tipos de Mocks

Existem diferentes tipos de mocks que podem ser usados em testes unitários:

**Mocks Estáticos:** Esses mocks são criados manualmente pelo desenvolvedor, definindo o comportamento desejado para cada método da dependência.

**Mocks Dinâmicos:** Esses mocks são criados usando bibliotecas de mocking como Mockito-Kotlin ou MockK. Elas permitem que os desenvolvedores definam o comportamento dos mocks de forma mais dinâmica e flexível.

**Mocks de Espião:** Esses mocks registram as interações com o código em teste, permitindo que os desenvolvedores verifiquem se os métodos corretos foram chamados com os argumentos corretos.

**Mocks de Partição:** Esses mocks dividem o comportamento da dependência em diferentes partições, permitindo que os desenvolvedores testem cada partição de forma independente.

**Utilize Mocks Apenas Quando Necessário:** Não use mocks em excesso, pois isso pode tornar os testes mais difíceis de entender e manter. Use-os apenas quando realmente precisar isolar uma dependência para testar o código em questão.

**Defina o Comportamento dos Mocks de Forma Clara:** Defina o comportamento dos mocks de forma clara e concisa, usando APIs intuitivas das bibliotecas de mocking. Evite definir comportamentos complexos

## Bibliotecas Populares em Kotlin

Duas bibliotecas de mocking se destacam no ecossistema Kotlin:

**Mockito-Kotlin:** Uma extensão para a biblioteca Mockito amplamente utilizada em Java. Ela fornece APIs familiares para usuários do Mockito e integra-se perfeitamente com a sintaxe concisa do Kotlin

**MockK:** Uma biblioteca de mocking nativa do Kotlin, projetada especificamente para a linguagem. Ela oferece um conjunto de recursos ricos e uma API intuitiva baseada em Kotlin.

## Exemplo com Mockito-Kotlin

Suponha que queremos testar uma classe `CarrinhoDeCompras` que depende de uma classe `Produto`:

```
class Produto(val nome: String, val preco: Double)

class CarrinhoDeCompras {

    private val produtos: MutableList<Produto> = mutableListOf()
```

```
fun adicionarProduto(produto: Produto) {  
    produtos.add(produto)  
}  
  
fun calcularTotal(): Double {  
    var total = 0.0  
    for (produto in produtos) {  
        total += produto.preco  
    }  
    return total  
}  
}
```

Para testar o método `calcularTotal` da classe `CarrinhoDeCompras`, podemos usar Mockito-Kotlin para criar um mock da classe `Produto`:

```
@Test  
fun `deve calcular o total corretamente`() {  
    val mockProduto1 = mock<Produto>()  
    val mockProduto2 = mock<Produto>()  
  
    `when`(mockProduto1.preco).thenReturn(10.0)  
    `when`(mockProduto2.preco).thenReturn(5.0)  
  
    val carrinho = CarrinhoDeCompras()  
    carrinho.adicionarProduto(mockProduto1)  
    carrinho.adicionarProduto(mockProduto2)  
  
    val totalCalculado = carrinho.calcularTotal()  
  
    assertEquals(15.0, totalCalculado)  
}
```

Neste exemplo, usamos as funções `mock` e `when` do Mockito-Kotlin para criar mocks dos produtos e definir seu comportamento. A função `thenReturn` define o valor de retorno simulado para o método `preco` dos mocks.

## Exemplo com MockK

Podemos alcançar o mesmo resultado usando a biblioteca MockK:

```
@Test
fun `deve calcular o total corretamente com MockK`() {
    val mockProduto1 = mockk<Produto>()
    val mockProduto2 = mockk<Produto>()

    every { mockProduto1.preco } returns 10.0
    every { mockProduto2.preco } returns 5.0

    val carrinho = CarrinhoDeCompras()
    carrinho.adicionarProduto(mockProduto1)
    carrinho.adicionarProduto(mockProduto2)

    val totalCalculado = carrinho.calcularTotal()

    assertEquals(15.0, totalCalculado)
}
```

Aqui, usamos as funções `mockk` e `every` do MockK para criar mocks e definir seu comportamento. A sintaxe do MockK é mais concisa e baseada em Kotlin, tornando o código mais legível.

## Conclusão

Mocks são uma ferramenta valiosa para escrever testes unitários eficazes em Kotlin. Ao usá-los com sabedoria, os desenvolvedores podem isolar o código em teste, controlar o comportamento das dependências e garantir a qualidade e confiabilidade de seu código. Este artigo forneceu uma visão geral detalhada de mocks em Kotlin, explorando seus benefícios, tipos, melhores práticas e bibliotecas populares como Mockito-Kotlin e MockK.

Além do que foi apresentado, existem mais bibliotecas de mocking, como verificação de interações, mocks de espião e mocks de partição. Isso permitirá escrever testes unitários ainda mais robustos e flexíveis.