Usuario colocando email e senha correto e vai logar na plataforma

Teste negativo

Quero que minha tela de login tenha opção de esquece minha senha

Erro de tentativas apos 3 vezes e ser bem intuitivo

Test TDD

PRE condição: cliente necessita de email e senha valida

Colocar email valido e senha clicar no botão entrar

1. Resultado esperado : efetuar autenticação com sucesso na plataforma
2.

    Test TDD
    NEgativo

    Pre condição: cliente com senha e email invalido

    Colocar email e senha invalida clicar no botão

    2 resultado esperado: mensagem de erro para usuario que email e senha incorreto

MELHORADO

Cenário: Login de Usuário na Plataforma

Teste Positivo

Pré-condição: O cliente deve possuir um e-mail e uma senha válidos cadastrados.

1. Inserir um **e-mail válido** e a **senha correta**.
2. Clicar no botão **"Entrar"**.

Resultado esperado:

- O sistema deve autenticar o usuário com sucesso e redirecioná-lo para a plataforma.

Teste Negativo: Login com Dados Inválidos

Pré-condição: O cliente utiliza um e-mail e/ou senha inválidos.

1. Inserir um **e-mail inválido** ou uma **senha incorreta**.
2. Clicar no botão **"Entrar"**.

Resultado esperado:

- O sistema exibe uma mensagem de erro clara:

"E-mail ou senha incorretos. Por favor, tente novamente."

Requisitos Adicionais para a Tela de Login

1. **Funcionalidade de "Esqueceu a Senha":**
    a. Adicionar uma opção visível para que o usuário possa recuperar a senha.
    b. Redirecionar para um fluxo de redefinição de senha ao clicar nessa opção.
2. **Bloqueio após Tentativas Mal-Sucedidas:**
    a. Após **3 tentativas consecutivas** de login com credenciais inválidas, exibir uma mensagem clara para o usuário:

"Muitas tentativas inválidas. Por favor, aguarde 5 minutos ou redefina sua senha."

    b. Garantir que o sistema bloqueie o login por um período de tempo configurável.

3. **Interface Intuitiva:**
    a. Garantir que os campos de **e-mail** e **senha** estejam claramente identificados.
    b. Adicionar feedback visual (bordas vermelhas ou mensagens contextuais) ao detectar campos inválidos.

Nota: Implementar os testes acima seguindo os princípios do TDD (Test-Driven Development). As validações devem ser unitárias e integradas para garantir uma experiência robusta e amigável ao usuário.

1. Testes Manuais

O QA verifica manualmente se as funcionalidades do software estão funcionando conforme o esperado.

- **Testes Exploratórios**:
    - O QA explora o sistema sem roteiros pré-definidos, buscando comportamentos inesperados.
    - Exemplo: Tentar inserir um e-mail inválido em um formulário de cadastro.
- **Testes de Regressão (Manual)**:
    - Garantem que novas alterações ou correções não quebraram funcionalidades existentes.
    - Exemplo: Após corrigir um bug no login, verificar se a página inicial ainda carrega corretamente.

2. Testes Automatizados

O QA usa ferramentas para criar scripts que testam funcionalidades repetitivamente, economizando tempo.

- **Testes de Regressão (Automatizados)**:

    - Automatizam a execução de testes antigos após cada alteração no código.
- **Testes Funcionais**:
    - Verificam se funcionalidades específicas estão operando corretamente.
    - Exemplo: Testar automaticamente se o botão "Adicionar ao Carrinho" funciona.
- **Testes de UI (Interface do Usuário)**:
    - Garantem que elementos visuais (botões, links, etc.) estejam interativos e exibidos corretamente.

3. Testes Funcionais

Focados no comportamento do software de acordo com os requisitos definidos.

- Exemplo: Verificar se, ao inserir as credenciais corretas, o usuário consegue fazer login.

4. Testes Não Funcionais

Avaliam aspectos além da funcionalidade básica.

- **Testes de Performance**:
    - Verificam como o sistema se comporta sob carga.
    - Exemplo: Testar quantos usuários simultâneos um site consegue suportar sem falhar.
- **Testes de Segurança**:
    - Identificam vulnerabilidades no sistema.
    - Exemplo: Verificar se o sistema protege dados sensíveis, como senhas.
- **Testes de Usabilidade**:
    - Avaliam se o sistema é intuitivo e fácil de usar.
    - Exemplo: Garantir que o processo de compra seja simples e rápido.

5. Testes de Integração

Verificam se diferentes módulos ou sistemas trabalham bem juntos.

- Exemplo: Confirmar que o módulo de pagamento processa transações corretamente quando conectado ao módulo de carrinho de compras.

6. Testes de Aceitação

O QA valida se o sistema atende aos requisitos do cliente antes de ser liberado.

- **Testes UAT (User Acceptance Testing)**:
    - Simulam o uso do sistema pelo cliente ou usuário final.

7. Testes de Mobile

Para aplicativos móveis, os QAs realizam:

- **Testes de Responsividade**:
    - Garantem que o aplicativo funciona bem em diferentes tamanhos de tela.
- **Testes de Compatibilidade**:
    - Verificam se o aplicativo funciona em várias versões do sistema operacional.

8. Testes Exploratórios e Ad-hoc

- **Exploratório**:
    - Sem roteiros, explorando funcionalidades com base na experiência do QA.
- **Ad-hoc**:
    - Testes informais, sem planejamento ou documentação, para encontrar falhas rapidamente.

9. Testes de APIs

- Verificam se a comunicação entre sistemas (APIs) está funcionando corretamente.
- Exemplo: Usar ferramentas como Postman para testar endpoints REST.

10. Testes de Stress e Carga

- **Stress**: Testam o limite máximo do sistema.
    - Exemplo: Simular milhares de usuários acessando ao mesmo tempo.
- **Carga**: Avaliam o desempenho em condições normais de uso.

11. Testes de Localização

- Garantem que o software é adaptado para diferentes idiomas, moedas e padrões culturais.

Resumo

O QA é essencial para assegurar a qualidade, confiabilidade e satisfação do usuário com o software. Ele combina testes manuais, automatizados e específicos (como segurança e performance) para entregar um produto impecável.

Se precisar de detalhes sobre um tipo de teste específico ou ferramentas que o QA usa, posso aprofundar! 😊

BDD (Behavior-Driven Development)

Conceito:

O BDD foca no comportamento do sistema em linguagem natural, aproximando desenvolvedores, testadores e stakeholders. Os cenários são descritos com estruturas como **Given-When-Then** para detalhar o comportamento esperado.

Cenário 1: Login com Credenciais Válidas

Descrição no BDD:

Given: O usuário possui um e-mail e senha válidos cadastrados.

When: Ele insere o e-mail e senha corretamente e clica no botão "Entrar".

Then: Ele é autenticado com sucesso e redirecionado para a plataforma.

O **desk check** é uma técnica de verificação manual de código-fonte, onde o programador ou a equipe de desenvolvimento revisa o código de forma detalhada, linha por linha, simulando a execução do programa mentalmente para identificar possíveis erros ou falhas lógicas.

É um processo informal e não automatizado, geralmente realizado antes de testes formais, com o objetivo de identificar problemas de implementação, entender o fluxo do código e garantir que ele cumpra os requisitos definidos. O desk check é uma forma de revisão de código mais acessível e eficaz nas primeiras fases de desenvolvimento.

Vantagens do Shift-Left Testing:

1. **Detecção precoce de falhas** – Identificar e corrigir problemas no início do ciclo de vida do software.
2. **Redução de custos** – Corrigir erros mais rapidamente quando são encontrados, reduzindo o custo de correção em fases posteriores.
3. **Melhor colaboração** – Desenvolvedores e testadores trabalham juntos desde o início, o que resulta em maior qualidade.
4. **Entrega mais rápida** – O tempo total de desenvolvimento pode ser reduzido, pois os erros são tratados rapidamente.

Essa abordagem envolve a automação de testes, integração contínua e a utilização de ferramentas para realizar testes em cada estágio do desenvolvimento, incluindo testes de unidade, integração e até testes de aceitação.

Test Driven Development (TDD) é uma abordagem de desenvolvimento de software onde os testes são escritos antes do código da funcionalidade. O ciclo básico do TDD é conhecido como **Red-Green-Refactor**, e envolve três etapas principais:

1. **Red**: O primeiro passo é escrever um teste automatizado para uma funcionalidade que ainda não foi implementada. Como o código ainda não existe, o teste falhará, o que gera a cor "vermelha" (indicando falha).
2. **Green**: A segunda etapa é escrever o código necessário para passar no teste. O objetivo aqui é garantir que o código seja suficiente para fazer o teste passar, sem se preocupar com a perfeição ou otimização.

3. **Refactor**: Após o teste passar, a próxima etapa é refatorar o código, melhorando-o sem alterar sua funcionalidade. O código pode ser limpo, otimizado ou reorganizado, e os testes continuam garantindo que tudo ainda funcione conforme esperado.

Vantagens do TDD:

- **Qualidade do código**: O TDD leva a um código mais testável, modular e com menos bugs.
- **Design orientado a testes**: Fornece uma base sólida para o design de software, pois os desenvolvedores precisam pensar em como a funcionalidade será testada antes de escrevê-la.
- **Documentação viva**: Os testes funcionais servem como uma documentação prática, descrevendo como o sistema deve se comportar.
- **Facilidade para refatoração**: Como há uma suíte de testes automatizados, mudanças no código podem ser feitas com mais segurança, pois qualquer erro será rapidamente detectado pelos testes.

Desvantagens do TDD:

- **Curva de aprendizado**: Para equipes novas, pode levar algum tempo para se acostumar com a prática de escrever testes antes do código.
- **Custo inicial**: Escrever testes primeiro pode ser visto como um esforço adicional, o que pode aumentar o tempo inicial de desenvolvimento.
- **Não substitui outros testes**: Embora o TDD seja eficaz para detectar erros, não deve ser a única técnica de teste usada; testes de integração e de sistema também são essenciais.

O TDD é um dos pilares do desenvolvimento ágil e se integra bem com práticas como **Continuous Integration (CI)** e **Continuous Delivery (CD)**.

ehavior Driven Development (BDD) é uma técnica de desenvolvimento ágil que foca na colaboração entre desenvolvedores, testadores e stakeholders não técnicos (como analistas de negócios) para definir e entender o comportamento esperado do software. O BDD tem como objetivo tornar os requisitos de software mais claros e compreensíveis para todos os envolvidos no projeto, usando uma linguagem mais natural e acessível.

Ao contrário do **TDD** (Test Driven Development), que é centrado em testes unitários e foca no código, o BDD concentra-se no comportamento da aplicação do ponto de vista do usuário, utilizando cenários de testes baseados em histórias de usuários.

O ciclo do BDD é baseado em **Gherkin**, uma linguagem de comportamento que utiliza a estrutura **Given-When-Then** para descrever os requisitos de uma funcionalidade. O formato é simples, legível por humanos e permite que todos, desde desenvolvedores até stakeholders não técnicos, participem da criação dos requisitos e do teste da aplicação.

Estrutura Given-When-Then:

- **Given** (Dado): O estado inicial ou contexto.
- **When** (Quando): A ação ou evento que ocorre.
- **Then** (Então): O resultado esperado ou comportamento do sistema.

Exemplo de um cenário de BDD:

```
gherkin
Copiar código
Cenário: Usuário realiza login com sucesso
  Dado que o usuário está na página de login
  Quando ele insere seu nome de usuário e senha corretos
  E clica no botão de login
  Então ele deve ser redirecionado para a página inicial
  E uma mensagem de boas-vindas deve ser exibida
```

Vantagens do BDD:

1. **Colaboração**: Facilita a comunicação entre desenvolvedores, testadores e stakeholders não técnicos, garantindo que todos estejam alinhados com os objetivos do projeto.
2. **Entendimento claro dos requisitos**: Usar uma linguagem natural e simples ajuda a esclarecer os requisitos e as funcionalidades para todos os membros da equipe.
3. **Testes automatizados e verificáveis**: O BDD incentiva a criação de testes automatizados que validam o comportamento do sistema, proporcionando uma cobertura de testes mais abrangente e útil.
4. **Documentação viva**: Os cenários escritos em Gherkin servem como documentação de como o sistema deve funcionar, que está sempre atualizada e pode ser entendida por qualquer pessoa envolvida no projeto.

Ferramentas com suporte a BDD:

- **Cucumber**: A ferramenta mais popular para implementar testes BDD, que suporta Gherkin.
- **SpecFlow**: Similar ao Cucumber, mas focada em .NET.
- **JBehave**: Outra ferramenta BDD para Java.
- **Behat**: Focada em PHP.

Desvantagens do BDD:

1. **Curva de aprendizado**: Pode ser necessário um período de adaptação para equipes novas que não estão acostumadas com a linguagem Gherkin ou a abordagem colaborativa.
2. **Overhead inicial**: Escrever cenários de comportamento pode adicionar uma sobrecarga no início do projeto, especialmente em projetos pequenos ou com requisitos bem definidos.
3. **Manutenção de cenários**: Se os requisitos mudam com frequência, a manutenção dos cenários de BDD pode se tornar trabalhosa.

O BDD é ideal para equipes ágeis que buscam alinhar os requisitos de negócios e de software de forma colaborativa, garantindo que as funcionalidades atendam às expectativas dos usuários.

Acceptance Test Driven Development (ATDD) é uma abordagem de desenvolvimento ágil que se concentra na criação de testes de aceitação antes do desenvolvimento de funcionalidades. O ATDD envolve a colaboração entre desenvolvedores, testadores e stakeholders (como analistas de negócios e clientes) para definir os critérios de aceitação de uma funcionalidade antes de ela ser implementada.

O objetivo principal do ATDD é garantir que o software atenda às necessidades dos usuários e aos requisitos de negócios desde o início do desenvolvimento. Isso é feito escrevendo os testes de aceitação que descrevem o comportamento esperado do sistema, com base em cenários definidos em colaboração com todos os envolvidos. Esses testes, uma vez implementados, servem como uma validação de que a funcionalidade foi construída corretamente e atende aos requisitos.

Diferença entre ATDD, BDD e TDD:

- **ATDD**: Foca na criação de testes de aceitação para validar se o sistema atende aos requisitos do cliente, garantindo que a funcionalidade esteja conforme o esperado.

- **BDD**: Foca na definição do comportamento da aplicação e na colaboração entre todos os membros da equipe, mas com uma ênfase maior na linguagem natural e nos testes de comportamento (com cenários como Given-When-Then).
- **TDD**: Foca em testes de unidade, onde o desenvolvedor escreve testes pequenos para validar o código enquanto ele é desenvolvido.

Ciclo de ATDD:

1. **Definição dos critérios de aceitação**: A equipe de desenvolvimento, testadores e stakeholders definem, juntos, os requisitos e critérios de aceitação da funcionalidade.
2. **Escrita dos testes de aceitação**: Com base nos critérios definidos, são escritos os testes de aceitação, que especificam como a funcionalidade deve se comportar em diferentes cenários.
3. **Desenvolvimento da funcionalidade**: Os desenvolvedores implementam o código necessário para passar nos testes de aceitação.
4. **Execução dos testes de aceitação**: Uma vez que a funcionalidade é implementada, os testes de aceitação são executados para validar que a funcionalidade atende aos critérios definidos.
5. **Refatoração (se necessário)**: Caso os testes não passem ou identifiquem problemas, a funcionalidade é ajustada até que os testes sejam bem-sucedidos.

Exemplos de testes de aceitação:

Se você estiver implementando uma funcionalidade de login em uma aplicação, os critérios de aceitação poderiam ser:

- **Critério 1**: O sistema deve permitir que um usuário faça login com sucesso usando um nome de usuário e senha válidos.
- **Critério 2**: O sistema deve exibir uma mensagem de erro se o usuário tentar fazer login com uma senha incorreta.
- **Critério 3**: O sistema deve bloquear o acesso após 5 tentativas de login falhas.

Vantagens do ATDD:

1. **Melhor alinhamento entre stakeholders**: Como os testes de aceitação são definidos junto com os requisitos do cliente, garante-se que todos os envolvidos no projeto tenham uma visão clara e comum do que deve ser entregue.

2. **Maior foco nos requisitos de negócios**: Ao escrever os testes de aceitação baseados nos requisitos de negócios, a equipe garante que o sistema atenderá às expectativas do cliente.
3. **Redução de retrabalho**: Como os critérios de aceitação são definidos antes da implementação, há menos chance de rework ou alterações significativas após o desenvolvimento.
4. **Documentação clara**: Os testes de aceitação servem como uma documentação clara e prática de como a funcionalidade deve se comportar.

Desvantagens do ATDD:

1. **Demora inicial para definição de critérios**: Pode ser mais demorado no início do projeto, pois exige que todos os stakeholders participem ativamente da definição dos critérios de aceitação.
2. **Exige colaboração constante**: Para ser bem-sucedido, o ATDD depende de uma colaboração contínua entre a equipe de desenvolvimento, testadores e stakeholders, o que pode ser desafiador em equipes grandes ou distribuídas.
3. **Manutenção de testes**: À medida que os requisitos do projeto mudam, os testes de aceitação também podem precisar ser atualizados, o que pode aumentar a carga de manutenção.

Ferramentas para ATDD:

- **FitNesse**: Uma ferramenta de ATDD que permite que os testes de aceitação sejam escritos em uma linguagem simples e executados diretamente.
- **Cucumber**: Usado principalmente para BDD, mas também pode ser aplicado em ATDD, especialmente quando os critérios de aceitação são definidos em uma linguagem natural.

O **ATDD** é uma prática poderosa para equipes ágeis, pois assegura que o desenvolvimento esteja alinhado com as necessidades reais dos usuários, além de melhorar a comunicação e o entendimento entre todas as partes envolvidas no projeto.

Perguntas

- **Qual abordagem é a mais adequada a ser utilizada em diferentes cenários?**

- **Quando estamos trabalhando de forma manual, como podemos aplicar essas abordagens de desenvolvimento (BDD, TDD, etc.) de maneira eficiente?**