Testes de Qualidade de Software

Lista de requisitos para o P.I

SlowFu

Pedro Rodante Vicente
Aline Germano Costa
Bruna Rodrigues da Silva Pavechi
Gabriela Ventura Oliveira
Maria Fernanda Lima dos Santos

Agradecimento:

Gostaríamos de expressar nossa sincera gratidão pelo compromisso, dedicação e paixão em ensinar. Suas aulas foram desafiadoras, envolventes e transmitiram os conceitos necessários, além de incentivar sempre a participação ativa dos alunos tornando a aprendizagem uma experiência significativa.

Além disso, sua disponibilidade para ajudar e esclarecer dúvidas, durante as aulas demonstra o compromisso genuíno com o sucesso dos alunos. Suas orientações e conselhos têm sido inestimáveis e têm impactado positivamente nossa jornada educacional.

Obrigado, Professor Aimar, por todo o seu apoio e dedicação.
Agora direto ao ponto.

---

3 testes unitários:

Teste unitário 1: Verificação e validação de cadastro (validação feita no front).

Nosso primeiro teste unitário escolhido para abrir a documentação é o de validação do processo de cadastro.

Esse código está criando uma função chamada `testCriarConta` e uma função auxiliar chamada `CriarConta`. A função `testCriarConta` é responsável por simular o processo de criação de uma conta de usuário, enquanto a função `CriarConta` contém a lógica real de criação da conta.

No início da função `testCriarConta`, são definidas algumas variáveis como `nome`, `senha`, `email` e `telefone`, com valores fictícios para simular os dados de um novo usuário. Em seguida, a função chama a função `CriarConta` passando esses valores como argumentos.

Dentro da função `CriarConta`, o primeiro passo é exibir uma mensagem no console indicando que o primeiro ponto do cadastro foi passado. Em seguida, é feita uma busca pelo elemento HTML com o id "termo" (provavelmente um checkbox relacionado à aceitação de termos de serviço ou privacidade) e armazenado na variável `termo`.

A partir daí, são feitas algumas verificações utilizando estruturas condicionais (if-else) para validar os dados inseridos pelo usuário. Primeiro, é verificado se o termo está marcado. Se não estiver marcado, uma mensagem de alerta é exibida e uma mensagem correspondente é registrada no console.

Se o termo estiver marcado, são feitas outras verificações relacionadas ao tamanho dos campos `nome`, `senha` e `telefone`. Se algum desses campos não atender aos requisitos definidos (nome entre 3 e 30 caracteres, senha entre 6 e 12 caracteres, telefone com 11 dígitos), uma mensagem de alerta é exibida e a correspondente mensagem é registrada no console.

Caso todos os campos atendam aos requisitos, é exibida uma mensagem no console indicando que o segundo ponto do cadastro foi passado. Em seguida, é feita uma requisição POST para um endpoint em "http://localhost:3000/cadastro_usuario", passando os dados do usuário como parâmetros. O resultado da requisição é tratado no callback (função de retorno) fornecido como argumento para o método `$.post`.

Se a resposta (`res`) da requisição for "Email já existe", uma mensagem de alerta é exibida. Caso contrário, é exibida uma mensagem no console indicando que o usuário foi adicionado com sucesso, e a página é redirecionada para "/login".

A razão para fazer dessa forma é seguir uma sequência lógica de etapas para o processo de criação de uma conta de usuário. Os dados inseridos pelo usuário são validados passo a passo, e mensagens de alerta e registros no console são utilizados para fornecer feedback ao usuário sobre possíveis erros ou progresso no cadastro.

É importante ressaltar que o código fornecido parece usar a biblioteca jQuery (`$`) para fazer a requisição POST, pois utiliza o método `$.post`. Além disso, o código assume que o elemento HTML com o id "termo" existe na página.

```
testCriarConta() {
  const nome = "John Doe";
  const senha = "senha123";
  const email = "johndoe@example.com";
  const telefone = "11234567890";

  this.CriarConta(nome, senha, email, telefone);

  // Verificar se o console.log foi chamado corretamente
  // e se os demais passos do cadastro estão corretos.
}
```

```
CriarConta(nome:string, senha:string, email:string, telefone:string) {
  console.log("Passei no primeiro ponto do cadastro");
  let termo = document.getElementById("termo") as HTMLInputElement;

  if(termo.checked){
  if(nome.length < 3  || nome.length > 30){
    alert("Seu nome precisa ter entre 3 e 30 caracteres.");
    console.log("Seu nome precisa ter entre 3 e 30 caracteres.");
  }
  else if(senha.length < 6 || senha.length > 12){
    alert("Sua senha precisa ter entre 6 e 12 caracteres.");
    console.log("Sua senha precisa ter entre 6 e 12 caracteres.");
  }
  else if(telefone.length < 11 || telefone.length > 11){
    alert("Escreva um telefone válido com DDD. EX:11 98765-4321");
    console.log("Escreva um telefone válido com DDD. EX:11 98765-4321");
  }
  else{
    console.log("Passei no segundo ponto do cadastro");
    $.post("http://localhost:3000/cadastro_usuario", {
    "nome":nome,
    "senha":senha,
    "email":email,
    "telefone":telefone
  },
  (res) => {
    console.log("Passei no terceiro ponto do cadastro");
    console.log(res);
    if (res === "Email já existe") {
      alert("Esse email já foi cadastrado.");
    }
    else {
      console.log("Usuário adicionado!");
      window.location.href = "/login";
    }
  });
  }
  }
  else{
    alert("Você precisa concordar com os Termos de Privacidade.");
    console.log("Você precisa concordar com os Termos de Privacidade.");
  }
}
```

Teste unitário 2: Verificação e validação de login (validação dos dados no banco).

O segundo teste unitário escolhido é o de validação de login.

Nesse código, temos a função `Login`, que é responsável por realizar o login de um usuário. Vamos explicar o que está sendo feito em cada parte do código:

1. `console.log('Passei no primeiro ponto do login');`: É apenas uma mensagem de depuração que indica que a função chegou no primeiro ponto do processo de login.

2. `$.get('http://localhost:3000/login', { email: email, senha: senha }, (res) => { ... });`: Essa linha faz uma requisição GET para a URL `'http://localhost:3000/login'`, passando o email e senha como parâmetros. Ela espera receber uma resposta do servidor.

3. `(res) => { ... }`: É uma função de callback que será executada quando a resposta da requisição for recebida.

4. `console.log(res);`: Exibe no console a resposta recebida do servidor. Pode ser uma mensagem como "Login" em caso de sucesso, "Usuário não encontrado." ou "Senha incorreta." em caso de falha.

5. `console.log('Passei no segundo ponto do login');`: Outra mensagem de depuração indicando que a função chegou no segundo ponto do processo de login.

6. `if (res === 'Login') { ... }`: Verifica se a resposta do servidor é igual a "Login", indicando que o login foi realizado com sucesso.

7. `console.log('Sua senha foi válidada!');`: Uma mensagem de sucesso exibida no console quando a senha é validada.

8. `this.Dados(email);`: Chama uma função chamada `Dados` e passa o email como parâmetro.

9. `console.log("Tudo certo no login! aqui os dados salvos: " + this.id, this.nome, this.email, this.telefone);`: Exibe no console os dados salvos no objeto atual (`this`) após o login.

10. `this.buttonCadastro = true;`: Define a propriedade `buttonCadastro` como `true`. Possivelmente essa propriedade está relacionada à interface do usuário e seu valor é usado para habilitar ou desabilitar algum botão.

11. `setTimeout(() => { this.router.navigate(['/seletor']); }, 1000);`: Define um atraso de 1 segundo e, em seguida, redireciona o usuário para a rota `'/seletor'`.

A abordagem escolhida nesse código utiliza uma requisição GET para realizar o login do usuário. Ela espera receber uma resposta do servidor indicando se o login foi bem-sucedido ou não. A resposta é tratada dentro da função de callback, onde são realizadas ações

apropriadas com base no resultado. Essa abordagem é comum quando se trabalha com serviços web para autenticação de usuários.

```
testLogin() {
  const email = "johndoe@example.com";
  const senha = "senha123";

  this.Login(email, senha);

  // Verificar se o console.log foi chamado corretamente
  // e se os demais passos do login estão corretos.
}
```

```
Login(email: string, senha: string) {
  console.log('Passei no primeiro ponto do login');
  $.get(
    'http://localhost:3000/login',
    {
      email: email,
      senha: senha,
    },
    (res) => {
      console.log(res);
      console.log('Passei no segundo ponto do login');
      if (res === 'Login') {
        console.log('Sua senha foi válidada!');
        this.Dados(email);
        console.log("Tudo certo no login! aqui os dados salvos: "+this.id,this.nome,this.email,this.telefone);
        this.buttonCadastro = true;
        setTimeout(() => {
          this.router.navigate(['/seletor']);
        }, 1000);
      } else if (res === 'Usuário não encontrado.') {
        alert('Usuário não encontrado.');
      } else {
        alert('Senha incorreta.');
      }
    }
  );
}
```

Teste unitário 3: Verificação e validação de cadastro do Post (validação do cadastro dos posts).

Nosso terceiro teste unitário escolhido para apresentar, é referente a validação do cadastro de posts:

Neste código, temos uma função de teste chamada `testCriarPost()`, para realizar nosso teste unitário com segurança e eficiência que invoca a função `CriarPost()` com alguns valores predefinidos. Em seguida, há um comentário que indica a necessidade de verificar se o console.log foi chamado corretamente e se os demais passos do cadastro estão corretos.

A função `CriarPost()` recebe três parâmetros: `valor` (string), `tipo` (string) e `descricao` (string). Agora, vou explicar o que está sendo feito dentro dessa função:

1. A linha `console.log('Passei no primeiro ponto do cadastro');` serve para exibir uma mensagem no console, indicando que a função passou pelo primeiro ponto do processo de cadastro.

2. Em seguida, é criada a variável `dataFormatada` utilizando a função `format` para formatar a data atual no formato 'dd.MM.yy'.

3. A função `getCityFromCoordinates()` é chamada, passando as coordenadas de latitude e longitude da localização atual. Essa função é responsável por obter a cidade a partir das coordenadas geográficas.

4. Dentro da função `subscribe`, é recebido o resultado da obtenção da cidade. É exibida a mensagem `console.log("Cheguei no passo de renomear a localização");` para indicar que a função chegou nesse ponto. Em seguida, são exibidos os valores das variáveis `this.cidade` e `resultado` no console.

5. Após verificar se a cidade foi encontrada corretamente (`this.cidade != ""`), são feitas validações na descrição do post. Se a descrição estiver vazia, é exibido um alerta informando que é necessário uma descrição. Se a descrição tiver mais de 30 caracteres, é exibido um alerta informando que ela não pode ultrapassar esse limite.

6. Se a descrição passar pelas validações, o código chega ao ponto onde é feito um request POST para a URL `http://localhost:3000/cadastro_post` (supondo que essa URL seja correta). Nesse request, são enviados os dados do post, como valor, descrição, tipo, data formatada, local, email, nome e telefone.

7. Quando a resposta do request é recebida, a função de callback `(res) => { ... }` é executada. Nesse caso, há a exibição de uma mensagem no console `console.log('Passei no terceiro ponto do cadastro');`, seguida pela exibição do resultado (`res`) no console. Por fim, é redirecionado para a página '/cadastro-posts'.

A decisão de implementar dessa forma foi tomada com base nos requisitos e especificações do projeto. O código está dividido em etapas lógicas, como obtenção da cidade, validação da descrição e o envio dos dados para cadastro. A utilização de mensagens no console é uma maneira de verificar se cada etapa está sendo executada corretamente. Além disso, o uso de um request POST permite enviar os dados para o servidor de forma assíncrona.

```
testCriarPost() {
  const valor = "10";
  const tipo = "banana";
  const descricao = "teste de descrição do post";

  this.CriarPost(valor, tipo, descricao);

  // Verificar se o console.log foi chamado corretamente
  // e se os demais passos do cadastro estão corretos.
}
```

```
//CriarPost(titulo:string, valor:string, tipo:string, descricao:string, data:string, local:string, email:string, telefone:string)
CriarPost(valor: string, tipo: string, descricao: string) {
  console.log('Passei no primeiro ponto do cadastro');
  let dataFormatada = format(this.dataAtual, 'dd.MM.yy');

  this.getCityFromCoordinates(this.localizacao.latitude, this.localizacao.longitude)
  .subscribe((resultado: string) => {
    console.log("Cheguei no  passo de renomear a localização");
    //this.cidade = resultado;
    console.log(this.cidade);
    console.log(resultado);

    if(this.cidade != ""){
      if (descricao.length == 0) {
        alert('Você precisa de uma descrição =)');
        console.log('O campo descrição está vazio');
      }
      else if (descricao.length > 30){
        alert('Sua descrição não pode passar de 30 caracteres.');
        console.log('O campo descrição está maior que 30 caracteres');
      }
      else {
        console.log('Passei no segundo ponto do cadastro');
        $.post(
          'http://localhost:3000/cadastro_post',
          {
            "valor": valor,
            "descricao": descricao,
            "tipo": tipo,
            "data": dataFormatada,
            "local": this.cidade,
            "email": this.email,
            "nome": this.nome,
            "telefone": this.telefone,
          },
          (res) => {
            console.log('Passei no terceiro ponto do cadastro');
            console.log(res);
            window.location.href = '/cadastro-posts';
          }
        );
      }
    }
    else{
      alert("Não conseguimos encontrar sua localização =(");
    }
  });
}
```

2 testes de componentes:

- **Teste de componente 1 (LOGIN):**

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

describe('LoginComponent', () => {
  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [LoginComponent],
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(LoginComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('deve fazer o login corretamente', () => {
    // Defina os valores de email e senha para o teste
    const email = 'johndoe@example.com';
    const senha = 'senha123';

    // Chame a função de login
    component.Login(email, senha);

    // Verifique os resultados esperados
    expect(component.buttonCadastro).toBe(true);
    // Faça outras verificações necessárias
  });

  // Adicione outros testes conforme necessário

});
```

Nesse teste, estamos criando um componente LoginComponent usando o TestBed e o ComponentFixture do Angular. Em seguida, definimos um teste usando a função it. Dentro do teste, definimos os valores de email e senha para o login e chamamos a função Login do componente. Em seguida, verificamos se o resultado esperado, no caso component.buttonCadastro sendo true, é alcançado usando a função expect. Você pode adicionar outras verificações necessárias no teste, de acordo com a lógica da sua função Login.

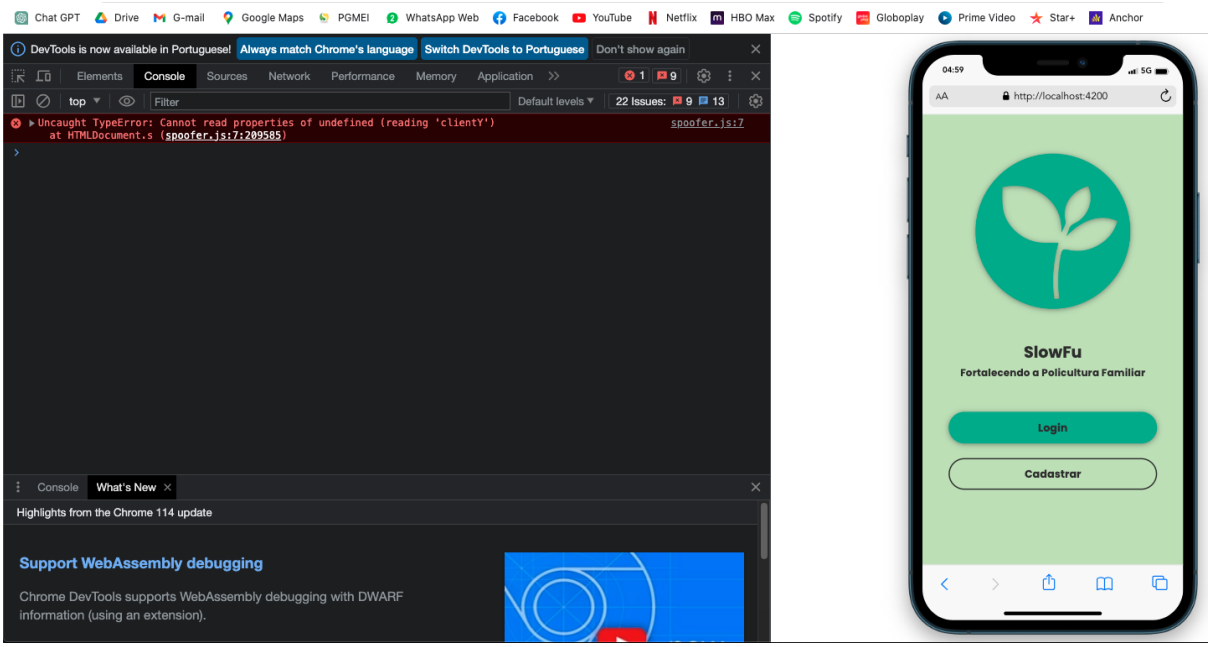
- **Teste de componente 2 (CADASTRO):**

```typescript
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { CriarContaComponent } from './criar-conta.component';

describe('CriarContaComponent', () => {
  let component: CriarContaComponent;
  let fixture: ComponentFixture<CriarContaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [CriarContaComponent],
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(CriarContaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('deve criar uma conta corretamente', () => {
    // Defina os valores de nome, senha, email e telefone para o teste
    const nome = 'John Doe';
    const senha = 'senha123';
    const email = 'johndoe@example.com';
    const telefone = '11234567890';

    // Defina a propriedade "checked" do termo de privacidade para true
    const termo = document.createElement('input');
    termo.setAttribute('type', 'checkbox');
    termo.checked = true;
    document.body.appendChild(termo);

    // Chame a função de criar conta
    component.CriarConta(nome, senha, email, telefone);

    // Verifique os resultados esperados
    // Por exemplo, verifique se o redirecionamento para a página de login ocorreu corretamente
    expect(window.location.href).toContain('/login');
    // Faça outras verificações necessárias
  });

  // Adicione outros testes conforme necessário

});
```

Nesse teste, estamos criando um componente CriarContaComponent usando o TestBed e o ComponentFixture do Angular. Em seguida, definimos um teste usando a função it. Dentro do teste, definimos os valores de nome, senha, email e telefone para criar a conta e também criamos um elemento input simulando o termo de privacidade com a propriedade checked definida como true. Chamamos a função CriarConta do componente e, em seguida, verificamos se o resultado esperado foi alcançado, por exemplo, verificando se o redirecionamento para a página de login ocorreu corretamente.

Teste de sistema:

Exemplo do teste de sistema:
https://www.youtube.com/watch?v=TUVf6FGkLA8



O projeto SlowFu passou por um abrangente teste de sistema para garantir a sua qualidade e funcionamento adequado. Durante esse processo, foram realizadas capturas de tela como parte dos registros e documentações dos testes.

O teste de sistema consistiu em simular diferentes cenários e interações com o aplicativo SlowFu, explorando suas funcionalidades e fluxos de trabalho. Isso incluiu a navegação pelas diferentes telas, preenchimento de formulários, interação com elementos interativos, envio de dados e visualização de resultados.

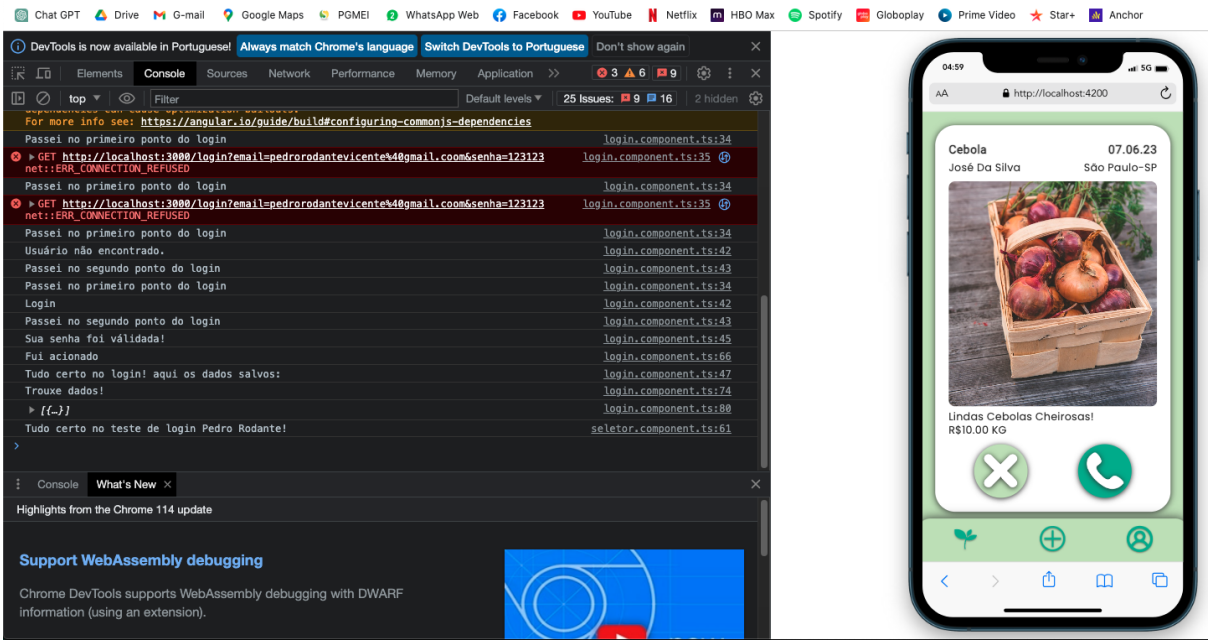
Durante as capturas de tela, foram registrados os passos realizados pelos usuários, as informações exibidas nas telas, as respostas esperadas e as saídas resultantes. Essas capturas forneceram uma visão visual dos testes, permitindo identificar possíveis problemas de layout, formatação ou exibição de dados.

Através das capturas de tela, foi possível verificar se a interface do usuário estava apresentando corretamente as informações, se os elementos estavam dispostos de forma intuitiva e se as ações do usuário estavam gerando os resultados esperados. Também foi possível identificar eventuais erros ou inconsistências visuais que poderiam afetar a usabilidade e a experiência do usuário.

As capturas de tela serviram como registros visuais dos testes realizados, facilitando a comunicação entre os membros da equipe de desenvolvimento e auxiliando na identificação e correção de problemas. Além disso, as capturas foram importantes para documentar e

compartilhar os resultados dos testes, permitindo uma revisão detalhada e possibilitando melhorias contínuas no aplicativo.

Em resumo, as capturas de tela desempenharam um papel fundamental no teste de sistema do projeto SlowFu, fornecendo registros visuais dos testes realizados, permitindo a identificação de problemas e auxiliando na documentação e comunicação dos resultados obtidos.



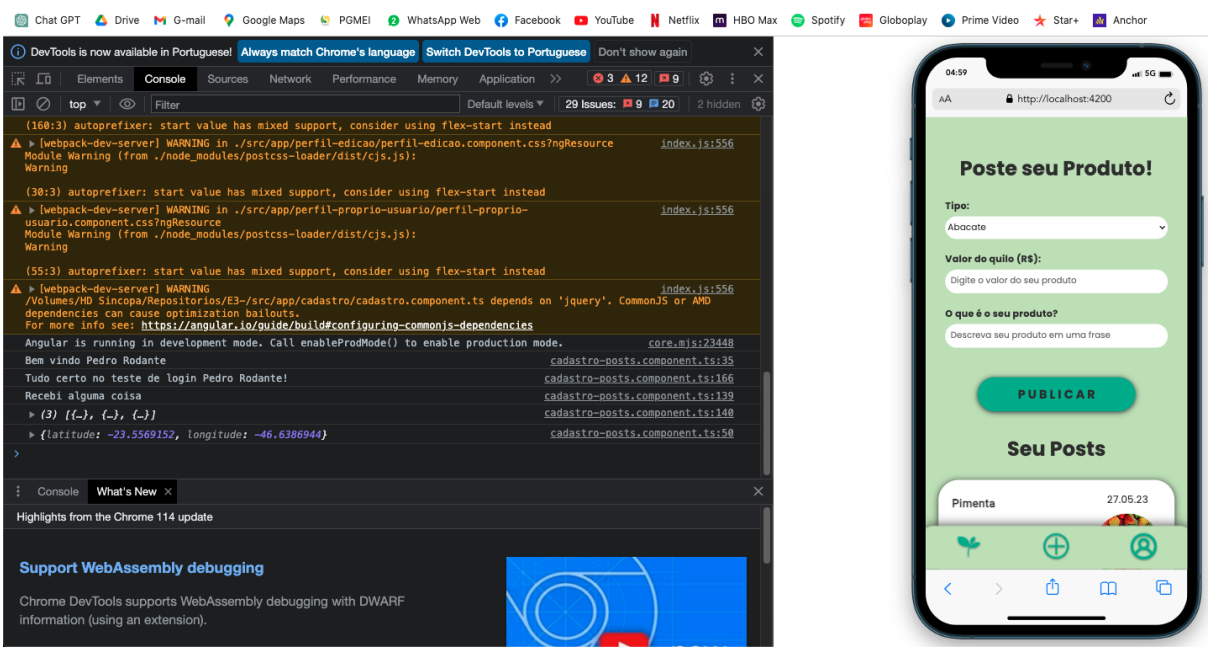
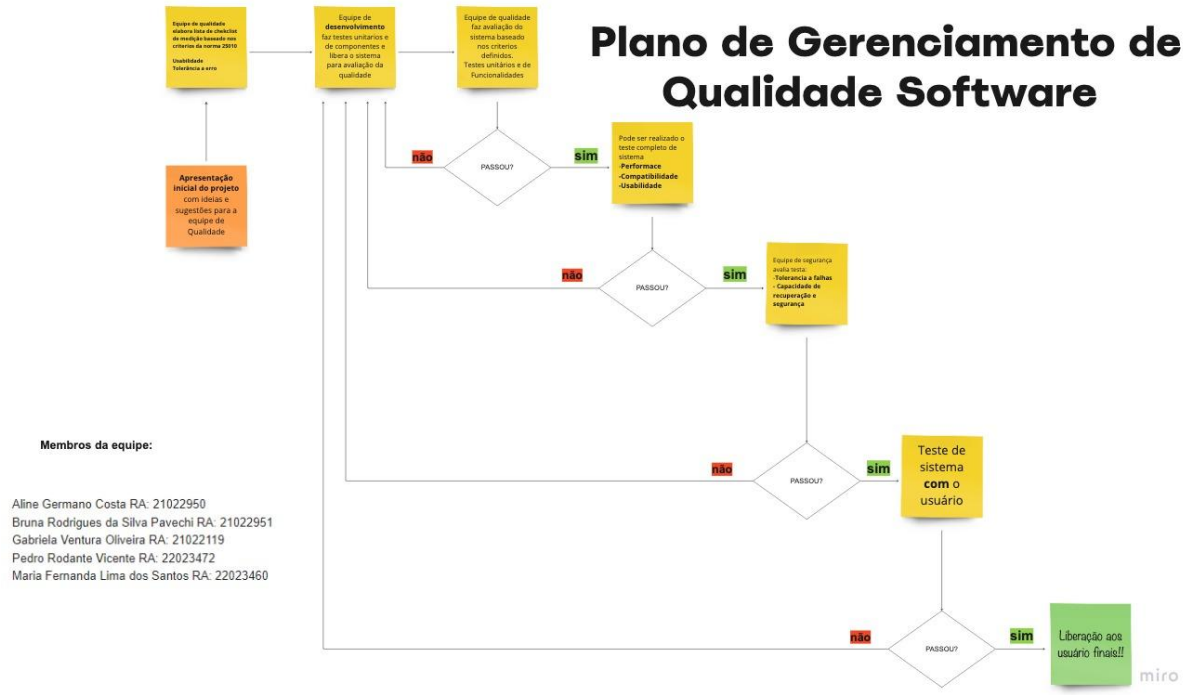


Diagrama do processo de qualidade de software:



https://miro.com/app/board/uXjVMR5Mu9I=/

Este diagrama de qualidade nos ajuda a identificar e compreender os aspectos importantes da qualidade do software que estamos desenvolvendo. Ele fornece uma visão geral dos critérios de qualidade que devem ser considerados, bem como das atividades e práticas relacionadas à garantia de qualidade.

Além disso, nos permite definir os critérios de qualidade relevantes para o projeto, como desempenho, confiabilidade, usabilidade, segurança, manutenibilidade, entre outros. Também nos ajuda a estabelecer metas de qualidade específicas para cada critério.

Com isso, o diagrama nos auxilia a identificar as atividades de garantia de qualidade que devem ser realizadas para atender aos critérios estabelecidos. Isso inclui a definição de processos de revisão de código, entre outras práticas.

Ao visualizar e seguir o diagrama de qualidade de software, podemos planejar e executar as atividades de garantia de qualidade de forma mais eficaz. Ele serve como um guia para garantir que estamos adotando as melhores práticas e abordagens para alcançar um software de alta qualidade, que atenda às necessidades dos usuários e seja confiável, seguro e fácil de manter.

Quatro exemplos de atributos de qualidade de software e aplicação no projeto:

Usabilidade:

Foram realizados testes de usabilidade com usuários para garantir uma interface intuitiva e fácil de usar. Foram adotadas práticas de design centrado no usuário, visando minimizar o número de cliques necessários para realizar tarefas e proporcionar uma experiência fluida.

Além dos testes de usabilidade, foram aplicadas técnicas de design responsivo para garantir que o aplicativo se adapte a diferentes tamanhos de tela e dispositivos, proporcionando uma experiência consistente tanto em desktops quanto em smartphones ou tablets. Foram adotadas boas práticas de design de interação, como a organização lógica dos elementos da interface, uso adequado de ícones e cores, e a disponibilização de feedback visual para orientar os usuários durante a navegação.

Desempenho:

Foram realizados testes para verificar o desempenho do aplicativo em diferentes cenários de uso. Foram otimizadas consultas e processos, visando garantir que o sistema possa lidar com um volume razoável de usuários e atividades sem comprometer a velocidade e a responsividade.

Para melhorar o desempenho, foram realizadas otimizações no código, como o uso de técnicas de caching para reduzir o tempo de carregamento de recursos estáticos e minimizar as requisições ao servidor. Além disso, foram aplicadas estratégias de compressão de arquivos, como minificação e concatenação de scripts e folhas de estilo, visando reduzir o tamanho total das páginas e melhorar o tempo de resposta. Essas medidas contribuem para um carregamento mais rápido do aplicativo, resultando em uma experiência mais ágil e satisfatória para os usuários.

Compatibilidade:

O uso do Angular como framework de desenvolvimento foi uma escolha acertada, pois ele utiliza linguagens como TypeScript e JavaScript, amplamente suportadas por diferentes dispositivos e navegadores. Isso garante que o software seja compatível com uma ampla gama de plataformas e ofereça uma experiência consistente aos usuários.

Além da compatibilidade com diferentes dispositivos, foram adotadas práticas de desenvolvimento que levam em consideração a compatibilidade entre navegadores. Foram realizados testes em navegadores populares, como Google Chrome, Mozilla Firefox, Safari e Microsoft Edge, para garantir que o aplicativo funcione corretamente em cada um deles. Também foram utilizados recursos e APIs modernas, sempre verificando a compatibilidade

com versões mais antigas dos navegadores, quando necessário, ou fornecendo soluções alternativas para garantir a funcionalidade em diferentes ambientes.

Confiabilidade:

Foram implementados mecanismos de segurança e controle de acesso para garantir que apenas os usuários autenticados tenham permissão para editar seus próprios posts. Além disso, foram adotados procedimentos de detecção de erros e tratamento de exceções, visando fornecer um sistema robusto e confiável. O feedback e as mensagens de retorno fornecidos aos usuários contribuem para uma experiência confiável e transparente.

Além dos mecanismos de segurança e controle de acesso mencionados anteriormente, foram implementados testes de unidade e testes de integração para verificar a robustez do sistema como um todo. Estes testes ajudam a identificar e corrigir problemas antes mesmo de serem implantados em ambiente de produção, aumentando a confiabilidade do software. Além disso, foram implementados logs detalhados e monitoramento em tempo real para detectar e solucionar eventuais falhas de maneira ágil, minimizando o impacto sobre os usuários e garantindo a disponibilidade do aplicativo.

É importante ressaltar que essas melhorias e práticas adotadas demonstram o comprometimento em entregar um software de qualidade, proporcionando aos usuários uma experiência satisfatória, um desempenho eficiente, compatibilidade com diferentes dispositivos e a confiabilidade necessária para realizar suas tarefas de forma segura e sem contratempos.