Plano de teste

Números - ProBrain

1. Introdução

1.1 Propósito do sistema

O sistema “Números” foi criado com o propósito de fornecer 4 tipos de resultados para o usuário, após a informação do número:

1. Se o número digitado é par ou ímpar.
2. Se é um número primo ou não primo.
3. Se está na sequência de Fibonacci.
4. Mostrar a sequência de números onde os múltiplos de 4 recebem o adicional da palavra “pin”.

1.2 Objetivos e prioridades

Neste relatório, será documentado os erros/bugs encontrados durante o desenvolvimento do projeto, dividindo-os da seguinte forma:

1. Implementação 1: par ou ímpar.
2. Implementação 2: primo ou não primo.
3. Implementação 3: sequência Fibonacci.
4. Implementação 4: múltiplos de 4 com “pin”.

2. Ambiente de testes

2.1 Definição do ambiente de teste

Serão feitos testes manuais no navegador Chrome.
É necessário definir no arquivo ***lauch.json*** o diretório para execução correta do sistema.
Após, serão realizados testes unitários automatizados usando o framework Cypress.

3. Tipos de testes utilizados

3.1 Testes funcionais

- **Teste Unitário:** utilizaremos testes unitários automatizados para a realização deste plano, pois a identificação de possíveis erros será mais rápida e eficiente.

4. Casos de testes

Caso de Uso	ID	Passos	Resultado
CT01 – Informar um número (1-1000)	1	Acessar a página inicial do sistema.	Página inicial será exibida.
	2	Digitar o número escolhido dentro da definição (1-1000).	Campo permite a entrada apenas de números, caracteres especiais e letras não são permitidos.
	3	Digitar caractere especial (!, @, #, $,...) ou letras.	Será exibido um pop-up com um aviso para digitar apenas números de (1-1000).

Caso de Uso	ID	Passos	Resultado
CT02 – Exibir resultados	1	Clicar em “Verificar”.	Serão exibidos todos os resultados das 4 implementações definidas (pag.2).

5. Erros e melhorias

Aqui serão detalhados os erros/bugs encontrados ao longo do desenvolvimento do projeto, juntamente com as melhorias. Este trecho será divido entre as 4 implementações (pag.2).

Antes de focar individualmente em cada implementação, listarei os erros gerais e melhorias implementadas.

5.1 Erros gerais e melhorias

1. **Permissão para digitar caracteres especiais e letras:** no início do desenvolvimento, o usuário poderia digitar qualquer caractere no input, isso causou um erro nos cálculos das implementações, no qual não eram mostrados os resultados. Foi implantado um

trecho no código JS que impede a digitação de qualquer caractere especial e letra, assim, corrigindo essa questão.

2. **Digitação de números fora do padrão definido (1-1000):** o projeto informado especifica que seja permitida a entrada de número entre 1 – 1000 para a realização dos cálculos. Com isso, foi incluída uma ***function*** chamada ***"verificaNumero"*** que verifica se o número digitado pelo usuário está dentro do limite definido. Se não estiver, o sistema dará um aviso (pop-up) e limpará o input para que ele digite novamente um número válido.
3. **Descrição do projeto:** no pedido inicial do projeto, é informado de que o usuário poderá escolher uma implementação, porém, mais à frente no pedido, também diz que **todos** os resultados deverão ser exibidos para o usuário. Sendo assim, não foi incluído uma aba de opções para o usuário escolher, já que todos os resultados deverão ser exibidos na tela de qualquer forma.

5.2 Implementação 1: Par ou Ímpar

Durante o desenvolvimento dessa implementação, foi constatado um erro de definições de "par" e "ímpar".

```
function parOuImpar(numero) {
    return numero % 2 === 0 ? "Ímpar" : "Par";
}
```

Acima podemos ver que as ***strings "Ímpar"*** e ***"Par"*** estão trocadas. Com isso, na hora de exibir o resultado para o usuário se o número for par, será exibido o resultado ***"Ímpar"*** e se o número for ímpar, será exibido o resultado ***"Par"***. Isso porquê o trecho: "***numero % 2 === 0***", verifica se o resto da divisão de numero por 2 é igual a 0, o que indica que "***numero"*** é par."

A correção é apenas trocar novamente as ***strings***, assim, mostrará o resultado correto.

```
function parOuImpar(numero) {
    return numero % 2 === 0 ? "Par" : "Ímpar";
}
```

5.3 Implementação 2: Primo ou Não Primo

Na primeira versão dessa implementação, fio adicionado um "***else***" dentro do loop "***for***". Este bloco "***else***" sempre será executado após a primeira interação do loop, independentemente do resultado da condição "***if***", com isso, a função sempre retornaria "Primo" como resultado final. Segue abaixo o código citado:

```
function numPrimoOuNao(numero) {
    if (numero < 2) return "Primo";
    for (let i = 2; i <= Math.sqrt(numero); i++) {
        if (numero % i === 0) {
            return "Não Primo";
        }
        else {
            return "Primo";
    }
    return "Primo";
}
```

A correção aplicada foi removendo o bloco "***else***", pois ela não é necessária observando a lógica do restante do código, onde o loop verifica se o número definido pelo usuário é divisível por algum número entre 2 e a raiz quadrada do próprio número. Se encontrar o divisor, a função retorna "***Não Primo***", e se não, retorna "***Primo***".

```
function numPrimoOuNao(numero) {
    if (numero < 2) return "Não Primo";
    for (let i = 2; i <= Math.sqrt(numero); i++) {
        if (numero % i === 0) {
            return "Não Primo";
        }
    }
    return "Primo";
}
```

5.4 Implementação 3: Sequência de Fibonacci

Nesta implementação, após executar o sistema e tentar verificar os resultados dos cálculos após a digitação do número no input, não foi exibido nenhum resultado na tela, por conta de um erro de sintaxe:

```
function sequenciaFibonacci(numero) {
    let a = 0, b = 1;

    while (a <=numero) {
        if (a === numero) {
            return "Está na sequência de Fibonacci"
        }
```

```
        const temp = a;
        a = b;
        b += temp;

    return "Não está na sequência de Fibonacci";
}
```

Na seta vermelha está indicando o local onde deveria haver "*}*" para fechar o loop "***while***". Para corrigir, bastou incluir esse fechamento.

5.5 Implementação 4: Múltiplos de 4 com "pin"

Na implementação 4, tivemos um erro de sintaxe e formatação, segue o código contendo os erros:

```
function multiplosDe4ComPin(numero) {
    let resultado = "";
    for (let i = 1; i <= numero; i++) {
        resultado += (i % 4 === 0 ? i + "pin, " : i + ", ");

    return resultado.slice(0, -3);
}
```

Na seta laranja, podemos ver o erro de sintaxe, onde não houve o fechamento do loop "***for***", o que faz com que o resultado final não seja apresentado corretamente. E na seta vermelha, a definição "***-3***" irá cortar um caractere a mais (ao invés de remover apenas a vírgula, irá remover também o espaço), trazendo a apresentação incorreta no resultado final.

A correção para ambos foi esta:

```
function multiplosDe4ComPin(numero) {
    let resultado = "";
    for (let i = 1; i <= numero; i++) {
        resultado += (i % 4 === 0 ? i + "pin, " : i + ", ");
    }
    return resultado.slice(0, -2);
}
```

5.6 Ajustes no arquivo style.css

Durante o desenvolvimento do projeto, também foi necessário ajustar o layout geral dele, para deixa-lo mais acessível e também agradável de utilizar.

Foi incluído dois ***class*** no arquivo HTML para os ajustes no layout, um chamado ***container*** e outro ***container__resultado***, seguindo as boas práticas de HTML.

Primeiro foi incluído um ***:root*** para definir a fonte e cor usada através de variáveis:

```
:root {
    --cor-texto: rgb(0, 0, 0);
    --fonte: "Montserrat";
}
```

Com isso, as informações ficaram mais “agradáveis aos olhos”.

Também foi incluído no ***body*** mais ajustes na centralização da tela e padrão para a exibição das informações, assim, é tudo ajustado conforme o zoom-in e zoom-out do usuário.

```
body {
    color: var(--cor-texto);
    font-family: var(--fonte);
    display: flex;
    justify-content: center;
    padding: 20%;
}
```

No primeiro momento, foi observado que os textos estavam muito “colados”, não havia um espaçamento adequado entre as linhas, por isso, dentro dos containers foi adicionado esse espaçamento, também facilitando a leitura e separação de cada resultado exibido na tela:

```
.container {
    text-align: center;
    line-height: 3em;
}

.container__resultado {
    line-height: 6mm ;
}
```

6. Instruções e execução de testes com Cypress

6.1 Requisitos

Certifique-se de que o Node.js e o npm (Node Package Manager) estão instalados em seu sistema. Se necessário, você pode instalar o Cypress globalmente executando o seguinte comando no terminal:

"***npm install – g cypress***"

Os testes estão estruturados dentro do diretório "***cypress***" no projeto. Dentro desse diretório, encontra-se o subdiretório "***e2e***" onde estão localizados os códigos.

Certifique-se de instalar todas as dependências necessárias para os testes, execute o seguinte comando no terminal para instalar:

"***npm install***"

6.2 Executando os testes

Para executar os testes localmente, utilize o seguinte comando para abrir a interface gráfica do Cypress:

"***npx cypress open***"

De forma alternativa, também é possível executar o teste em forma de linha de comando, gerando também os vídeos playback dos testes, usando o seguinte comando:

"***npx cypress run***"

Dessa forma, irá exibir todos os testes e resultados no terminal, gerando um breve relatório do que foi feito.

6.3 Execução do front

Para a execução do front-end do projeto, utilizamos o ***Vercel***. Essa, é uma plataforma que permite a hospedagem de aplicações de forma gratuita. Assim, facilitando o acesso ao projeto criado.

Dessa forma, é possível acessar a aplicação por meio da url: https://numeros-pro-brain.vercel.app