INSTITUTO DE GESTÃO E TECNOLOGIA
DA INFORMAÇÃO

React III

Rodrigo Borba

2022

React III

Rodrigo Borba

Sumário

Capítulo 1. Introdução 6

Plano de ensino 6

Capítulo 2. Styled-components 7

Quais problemas o styled-components busca resolver? 7

Apenas o CSS crítico 7

Sem bugs por conta de nomes de classes 8

Fácil remoção de CSS 8

Estilização dinâmica de maneira fácil 8

Manutenção fácil 8

Inserção automática de prefixo de Vendor 9

Instalação 9

Sintaxe e Estilos dinâmicos baseados em propriedades 9

Template String/Template Literals 10

Definindo Estilos dinâmicos baseado em propriedades 11

Tema 13

Estendendo estilos 14

Animações 15

Pseudoelemento, pseudosseletores e nesting 15

Conclusão 16

Capítulo 3. Bibliotecas de Data Fetching 17

SWR – Stale While Revalidate 17

React Query 20

Diferenças 22

Dev tools 22

Mutation 22

Bundle Size 23

Garbage Collection 23

Capítulo 4. CSR, SPA, SSR, SSG, ISR 24

CSR – Client Side Rendering 24

SPA – Single Page Application 24

SSR – Server Side Rendering 25

SSG – Static Site Generation 26

ISR – Incremental Static Regeneration 26

Capítulo 5. NEXT.JS 27

Setup 27

Páginas 27

Pré-renderização 29

Static Generation 29

Server-side Rendering 32

Incremental Site Regeneration 33

Features mais relevantes 33

Image Optimization 34

Internacionalização 34

Fast Refresh 34

Bundles com separação de código 35

Conclusão 35

Capítulo 6. Teste de aplicações WEB 36

TDD – Test Driven Development 37

Jest 38

Instalação 38

Na prática 39

React Test Library 40

Instalação 41

Na prática 41

Referências 44

Capítulo 1. Introdução

Neste primeiro capítulo, será abordado de maneira breve todo o conteúdo que esta apostila irá cobrir, bem como as expectativas e os objetivos desta disciplina. Aqui falaremos sobre bibliotecas e Frameworks populares dentro da comunidade React, além de diferentes tipos de testes de Software e do TDD. Após a leitura desta apostila, é esperado que o aluno entenda as razões do uso de cada uma das bibliotecas/frameworks, quais problemas elas resolvem e quais são seus pontos negativos.

Plano de ensino

Esta apostila abordará os seguintes macrotópicos:

- Styled componentes;
- Bibliotecas de Data Fetching;
- Estratégias de renderização (SPA, SSR, SSG);
- Next.Js;
- Testando aplicações React;
- Deploy de aplicações.

Capítulo 2. Styled-components

"Styled-components é o resultado do questionamento de como poderíamos melhorar o CSS para estilização de componentes React. Focando em um único use-case, conseguimos otimizar a experiência para desenvolvedores, bem como o resultado para usuários finais." (styled-components).

O styled-components (SC) será a primeira biblioteca abordada por nós, e não por qualquer motivo, dado que conta com mais de 37 mil estrelas no Github e é, dentro do React, a escolha preferida da comunidade. Ela foi pensada com o intuito de resolver vários problemas que enfrentamos ou chegaremos a enfrentar em algum momento quando aplicamos estilos aos nossos componentes.

Essa biblioteca foca fortemente no conceito de modularização. Esse mesmo conceito é o que rege a ideia de componentização por trás do React. A ideia é que os estilos estejam ligados diretamente aos seus componentes. Ou seja, ao invés de escrever folhas de estilos globais, definiremos estilos para os nossos componentes em questão. Veremos mais detalhadamente nos próximos tópicos.

Quais problemas o styled-components busca resolver?

Como dito anteriormente, para entendermos de fato a razão do uso das ferramentas, precisamos entender quais os problemas que elas tentam resolver. Nesse sentido, O SC deixa claro quais são seus objetivos em sua documentação oficial. Falaremos sobre cada um dos problemas mais detalhadamente.

Apenas o CSS crítico

O styled-components consegue acompanhar quais componentes estão sendo renderizados na página e injeta apenas seus estilos, e nada mais, de maneira automática. Combinando essa funcionalidade à componentização, os usuários

carregarão o mínimo de dados possível. Essa otimização seria relativamente custosa, se fosse feita manualmente.

Os benefícios aqui são muito mais notáveis em aplicações maiores, mas ainda sim são perceptíveis com algumas dezenas de estilos definidos.

Sem bugs por conta de nomes de classes

O styled-components gera classes únicas para seus estilos. Dessa forma, não há necessidade de se preocupar com nomes de classes duplicados ou erros de escrita. Além do tempo perdido quando se está pensando em que nome dar àquela classe muito específica, que só deveria ser aplicada para um único elemento que a usa.

Fácil remoção de CSS

Pode ser bem difícil saber qual nome de classe está sendo usado em algum outro lugar do seu código, e o SC faz isso ser óbvio. Cada pedaço de estilo está amarrado a um componente específico. Se o componente não está sendo usado (situação avisada por algo como o Lint) e é removido, todo o estilo também é removido.

Estilização dinâmica de maneira fácil

A estilização pode ser adaptada baseada em propriedades ou em um tema global. Imagine que um botão pode ser azul ou vermelho baseado apenas em uma única propriedade passada na hora de usar sua tag.

Manutenção fácil

Você não precisará mais sofrer indo atrás de várias linhas de código para encontrar o estilo que está afetando indevidamente o seu componente. Com a modularização do SC, isso se torna uma atividade trivial.

Inserção automática de prefixo de Vendor

Não é necessário se preocupar em adicionar os prefixos específicos dos browsers. É preciso escrever apenas o CSS do jeito que você conhece, e o SC resolve isso para você.

Instalação

Não existe exatamente muita coisa para falar neste tópico. A instalação é bem simples. Nenhuma configuração extra é necessária.

```
# with npm
npm install --save styled-components

# with yarn
yarn add styled-components
```

A documentação oficial também recomenda o uso do resolution no package.js. Isso ajuda a evitar múltiplos problemas causados por mais de uma versão do styled-components rodando no seu projeto.

```
{
  "resolutions": {
    "styled-components": "^5"
  }
}
```

Sintaxe e Estilos dinâmicos baseados em propriedades

Chega de falar apenas da teoria. Vamos começar a ver na prática como é o uso do styled-components.

```
const wrapper = styled.div`
    padding: 18px;
    background-color: blue
`
```

"Nossa, mas que sintaxe estranha. Uma String declarada ao lado do nome *div*?"

```
const wrapper = styled.div`
    padding: 18px;
    background-color: ${(props) => props.color}
`
```

"E ainda ficou mais estranho. Agora tem uma *Arrow Function* chamada dentro da String. Mas o que está acontecendo aqui?"

Template String/Template Literals

Para entendermos essa sintaxe, antes de mais nada, precisamos dar um passo para trás e entender que, na verdade, a "string" é uma **Template String,** ou **Template Literals**. Ela foi introduzida no ES6. Com ela, conseguimos interpolar strings sem usar o "+". Sua sintaxe é composta de acentos graves. O código Javascript pode ser interpretado quando escrito entre chaves precedidas do cifrão **${**aqui algum código**}**

Um uso prático seria:

```
const nome = "Rodrigo"
console.log("Meu nome é ${nome}")
```

Talvez a maioria de vocês já conheça tudo isso. Mas o que é bem menos comum é o uso de Template Literals para a chamada de funções:

```
// essas chamada
funcao(["algum texto"]);

// é equivalente a esta
funcao`algum texto`;
```

Em suma, quando o acento grave está presente após uma função, ela é chamada passando um array de string. Se existirem palavras entre as strings, então:

```
// essas chamada
funcao(["algum texto", "resto do texto"], palavra);
// é equivalente a esta
funcao`algum texto ${palavra} resto do texto`;
```

Basicamente, é isso que acontece quando fazemos a chamada ao style.div. Dentro da função, nossa “string” é interpretada e a mágica acontece.

Definindo Estilos dinâmicos baseado em propriedades

Agora que desmistificamos a sintaxe, definir estilos simples não diferem em nada do que faríamos em uma folha de estilo comum. Porém, estamos falando de um *Css-in-Js*, e isso nos abre várias possibilidades. Uma delas é fazer com que os estilos se comportem de acordo com propriedades passadas para o nosso componente. Por exemplo, imagine que nós temos um botão estilizado:

```
const StyledButton = style.button`
      padding: 8px 12px;
      background-color: grey;
      color: white;
      border-radius: 12px;
`
```

```
<StyledButton>Este é um botão de teste</StyledButton>
```

O código acima faz com que o botão abaixo seja renderizado:


```
<StyledButton tipo={"perigo"}>
Este é um botão de teste
</StyledButton>
```

Uma vez que passamos a propriedade na chamada do nosso componente, podemos, agora, acessá-la a partir da nossa Template String e fazer com que a cor mude para vermelho em caso de tipo: perigo. Para fazer isso, usaremos a sintaxe ${...}. O SC provê em forma de callback o acesso para as propriedades passadas:

```
const StyledButton = style.button`
      padding: 8px 12px;
      background-color: ${(props) => props.tipo === "perigo" ?
"red" : "grey"};
      color: grey;
      border-radius: 12px;
`
```

Esse seria nosso resultado. Legal, né?! Um mesmo componente, mesma folha de estilo, renderizando de uma forma diferente, baseado em uma única propriedade. Pois bem, essa é uma das coisas legais do styled-components.

Tema

Também conseguimos definir temas dentro do SC. Assim, conseguimos alterar, por exemplo, a cor de todo o nosso sistema em apenas um local. Essa cor pode ser acessada de maneira parecida com a que usamos, quando acessamos uma propriedade passada ao nosso componente. Para que isso funcione, é necessário envolver os SC no ThemeProvider, fazendo com que passem a ter acesso às variáveis de tema:

```
const theme = {
  main: "blue"
};

const StyledButton = style.button`
  color: ${props => props.theme.main};
  border: 2px solid ${props => props.theme.main};
`;
```

```
<div>
 <!-- Não tem acesso -->
 <StyledButton>Normal</StyledButton>

 <ThemeProvider theme="{theme}">
   <!-- Tem acesso -->
   <StyledButton>Themed</StyledButton>
 </ThemeProvider>
```

```
</div>
```

Aconselho vocês a imprimirem a variável props para terem uma ideia melhor de como ela é estruturada.

Estendendo estilos

É legal saber também que os componentes podem ser estendidos. Vamos usar botões como exemplo mais uma vez. Imagine que você tem o seu estilo padrão para botões:

```
const StyledButtonRounded = styled.button`
      padding: 8px 12px;
      background-color: grey;
      color: white;
      border-radius: 12px;
`
```

Agora, imagine que você quer outro componente que deveria ter a aparência desse botão, porém com algumas alterações. No nosso caso, queremos um botão retangular. Sem bordas arredondadas:

```
const StyledButtonRec = styled(StyledButtonRounded)`
      border-radius: 0px;
`
```

Pronto. Agora temos um componente que carrega como base as definições do nosso componente anterior, mas que modifica seu border-radius. Isso também funciona para componentes não criados pelo styled-components.

Animações

As animações CSS com @keyFrame não possuem escopo de componente, mas, de qualquer forma, não as queremos com escopo global para evitar colisões de nomes, um dos problemas que o SC busca resolver. Para resolver essa questão, devemos criar um keyframe e exportá-lo como um helper:

```
const rotate = keyframes`
  from {
    transform: rotate(0deg);
  }

  to {
    transform: rotate(360deg);
  }
`;

const Rotate = styled.div`
  animation: ${rotate} 2s linear infinite;
`;
```

Pseudoelemento, pseudosseletores e nesting

Falaremos agora sobre como funcionam alguns recursos que temos no CSS e como eles se comportam dentro do styled-components. Não vou explicar o que são pseudoelementos, pois imagino que, se chegaram até este módulo, esse já deva ser conhecimento comum.

Se quisermos que o nosso botão altere de cor quando o cursor estiver sobre ele, podemos usar o pseudoelemento 'hover'. A sintaxe para isso está logo abaixo:

```
const StyledButton = style.button`
```

```
      color: ${props => props.theme.main};
      border: 2px solid ${props => props.theme.main};

      &:hover {
            background-color: grey;
       }
`;
```

O mesmo serve para qualquer outro pseudoelemento.

```
const StyledButton = style.button`
      color: ${props => props.theme.main};
      border: 2px solid ${props => props.theme.main};

      :before {
            content: "🦄";
       }
`;
```

Conclusão

Acredito que com isso cobrimos tudo que precisávamos para usar o styled-components de maneira satisfatória em nossa aplicação. Lembrem-se sempre de que a ideia é a modularização do CSS, de modo que os componentes contenham estritamente o código necessário para renderizá-los. Lembrem-se também de fazer uso de temas para facilitar futuras manutenções.

Capítulo 3. Bibliotecas de Data Fetching

Aplicações React não possuem uma maneira opinativa de buscar informações de um servidor. Quando falo “opinativa”, quero dizer alguma solução que tome decisões estratégicas que visem ser a escolha certa para a maioria dos casos. Bibliotecas opinativas nos poupam o tempo e a complexidade de escolher entre A ou B como melhor opção para a situação. Neste capítulo, falaremos sobre duas bibliotecas que facilitam a nossa vida quando o assunto é buscar informações remotamente por tomarem essas decisões por nós.

SWR – Stale While Revalidate

O nome “SWR” vem do termo “stale-while-revalidate”, que é uma estratégia que consiste em primeiramente retornar dados do cache (stale), solicitar os dados atualizados (revalidate) e então finalmente retornar esses dados atualizados.

Parafraseando a documentação oficial, as principais características do SWR são:

- Data fetch rápido, leve e reusável;
- Cache e prevenção de requests repetidos;
- Experiência em tempo real;
- Agnóstico quanto ao protocolo;
- Suporte a Server Side Rendering, Incremental Static Regeneration e Static Site Generation. Não se preocupem com esses termos, pois veremos mais sobre eles no capítulo do Next JS;
- Pronto para o TypeScript;

- Suporte ao React Native.

Quando desenvolvemos uma aplicação, geralmente precisamos reusar dados vindos do servidor em mais de um local/página. Com o SWR, é extremamente fácil de conseguir isso. A ideia aqui é deixar de realizar data fetching da maneira tradicional, imperativa e, em vez disso, tornar seu código mais declarativo.

Como dito antes, o SWR é agnóstico quanto à maneira de solicitar os dados, seja RESTful, seja GraphQL. Quando usado com REST, antes de mais nada, é preciso criar uma função que busque esses dados. No nosso exemplo, é uma função wrapper em cima do Fetch:

```
const fetcher = (...args) => fetch(...args).then(res =>
res.json())
```

E em nosso componente:

```
function index () {
  const { data, loading, error } =
useSWR('/api/data/algum_id', fetcher)

  if (error) return <div>failed to load</div>
  if (loading) return <div>loading...</div>

  // render data
  return <div>hello {data.name}!</div>
}
```

Aqui o SWR está:

1. Exibindo o conteúdo que, porventura, pode estar presente no cachê, exibindo algo para o usuário o mais cedo possível.

2. Solicitando novos dados.

3. Exibindo os novos dados assim que eles estiverem disponíveis, substituindo, desse modo, os dados antigos.

Como podemos observar também, o SWR, dentre outros, provê-nos três objetos que nos são úteis para a montagem da UI. Dessa forma, facilmente conseguimos acompanhar o status do request e exibir para o usuário erros e loadings.

Como já dito antes, podemos reutilizar nossos fetchers em forma de custom hooks:

```
function useProduct (id) {
    const { data, error } = useSWR(`/api/product/${id}`,
fetcher)

    return {
      product: data,
      isLoading: !error && !data,
      isError: error
    }
  }
```

E então:

```
function ProductDetails ({ id }) {
   const { product, isLoading, isError } = useProduct(id)
    if (isLoading) return <Spinner />
    if (isError) return <Error />
   return <img src={product.image} />
 }
```

Destaca-se que, se fizermos uso do useProduct várias e várias vezes, o SWR será inteligente o suficiente para realizar apenas um Request, poupando-nos, assim, dessa responsabilidade.

Podemos ir ainda além com o SRW; é possível solicitar que os dados sejam revalidados sempre que o usuário remover e voltar a adicionar o foco na nossa página:

```
useSWR(key, fetcher, {
  revalidateOnFocus: false
})
```

Ou depois de intervalo de tempo específico:

```
useSWR('/api/todos', fetcher, { refreshInterval: 1000 })
```

Ou podemos usar vários gatilhos para o revalidation:

```
useSWR(key, fetcher, {
  revalidateIfStale: true,
  revalidateOnFocus: true,
  revalidateOnReconnect: true
})
```

As opções do SWR vão muito além e seria muito massivo tratar todas nessa apostila. Acredito que tenha coberto aqui os casos básicos para que possamos seguir com nossa aplicação. Recomendo, de qualquer forma, que vocês acessem a documentação e se aprofundem no assunto: https://swr.vercel.app/docs/options.

React Query

O React Query é uma outra biblioteca de data-fetching que resolve basicamente os mesmos problemas que o SWR, porém de uma maneira ligeiramente

diferente. Ambas as bibliotecas usam cache, impedem requests duplos e permitem revalidação quando o foco é retornado, ou depois de um certo intervalo de tempo, etc.

Começando pelo setup, o React Query exige um QueryClientProvider em torno de qualquer código onde se é feito Fetch. No nosso exemplo, vamos usá-lo em torno de toda a aplicação:

```
import { QueryClient, QueryClientProvider } from "react-
query";
const queryClient = new QueryClient();

ReactDOM.render(
  <QueryClientProvider client={queryClient}>
    <App />
  </QueryClientProvider>,
  document.getElementById('root')
);
```

Pronto. Daqui em diante, a sintaxe se torna muito parecida com a do SWR. Definimos primeiro um fetcher:

```
const fetcher = (...args) => fetch(...args).then(res =>
res.json())
```

E então o usamos:

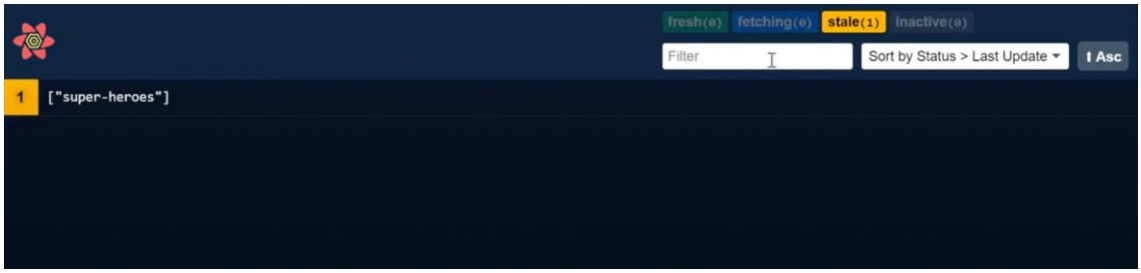
```
const component = (_ => {
   const { isLoading, error, data } = useQuery('queryId', fetcher)

   ....
}
```

Diferenças

Por mais que as duas bibliotecas sejam bastante parecidas, existem algumas diferenças importantes a serem levantadas:

Dev tools

O React-query provê um painel com informações referentes às queries que estão sendo executadas. Já o SWR não possui algo parecido.



Mutation

Diferentemente das queries, mutations são usadas para alterar informações no servidor, como operações de create, delete e update. O react-query permite essa operação de uma maneira mais fácil, provendo um objeto que possui propriedades das quais podemos verificar o estado do request:

```
function App() {
   const mutation = useMutation(newTodo => {
     return axios.post('/todos', newTodo)
   })
    return (
     <div>
       {mutation.isLoading ? (
         'Adding todo...'
       ) : (
         <>
           {mutation.isError ? (
             <div>An error occurred: {mutation.error.message}</div>
           ) : null}
            {mutation.isSuccess ? <div>Todo added!</div> : null}
```

```
            <button
              onClick={() => {
                mutation.mutate({ id: new Date(), title: 'Do Laundry' })
              }}
            >
              Create Todo
            </button>
          </>
        )}
      </div>
    )
}
```

Bundle Size

O tamanho do bundle do SWR é cerca de 15 kb, enquanto o do react-query chega a 50kb. Então, dependendo de quanto o tamanho final do bundle seja relevante para a aplicação, isso talvez pese na escolha da biblioteca.

Garbage Collection

Segundo a documentação de react-query, todas informações que estão em cache são removidas depois que qualquer query fica inativa por 5 minutos (configurável). Com o SWR, é necessário gerir o cache manualmente.

Capítulo 4. CSR, SPA, SSR, SSG, ISR

Existem algumas diferentes estratégias consolidadas pela comunidade para entregar conteúdo ao usuário através de páginas WEB. Não existe uma estratégia melhor ou pior, mas sim uma que mais se adapta a cada tipo de página web. Podemos definir a diferença entre as estratégias como **quando e onde o Javascript é executado para gerar o HTML**.

Neste capítulo, veremos mais detalhadamente o que significa cada uma dessas siglas e em quais momentos elas podem prover uma melhor experiência para os usuários.

CSR – Client Side Rendering

CSR é uma sigla para Client Side Rendering, ou Renderização no Lado do Cliente. No CSR, o **Javascript usado para gerar o HTML é executado completamente no lado do cliente**, ou seja, no browser. E isso é tudo. O conceito de CSR é exatamente este. Veremos na sequência que os conceitos podem ficar um pouco complicados.

SPA – Single Page Application

SPA é uma sigla para Single Page Application, ou Aplicação de Página Única. A ideia é trazer uma melhor experiência ao usuário após o primeiro carregamento da página. É importante ressaltar que não necessariamente aplicações SPA são compostas por apenas uma página.

O primeiro carregamento no SPA traz consigo o conteúdo necessário para permitir uma navegação praticamente instantânea, já que todo o conteúdo está presente no cliente. O que geralmente se encontra na web é o SPA em que o **Javascript usado para gerar o HTML é executado completamente no lado do cliente**, ou seja, no

browser. Porém, é importante destacar que também há SPAs que têm sua renderização no lado do servidor. Bons exemplos de SPAs são aplicações como Spotify e Gmail.

Uma das principais vantagens do SPA é a possibilidade de interação sem precisar recarregar a página. Tudo acontece de maneira praticamente instantânea. Também é possível que a carga no servidor seja reduzida, pois não existe a necessidade de chamadas subsequentes durante as interações do usuário.

Sobre as desvantagens, uma grande é que a performance pode ser imprevisível já que toda a carga é processada pelo lado do cliente e o hardware pode variar bastante. Outro ponto relevante, dependendo da aplicação, é o SEO, Search Engine Optimization ou Otimização de Motor de Busca. Para que a página esteja completamente disponível para os indexadores agirem, é necessária a execução do Javascript, e alguns indexadores não o suportam.

SSR – Server Side Rendering

SSR é uma sigla para Server Side Rendering, ou Renderização no lado do servidor. Diferentemente do SPA, o **Javascript usado para gerar o HTML é executado no lado do servidor a cada requisição.**

O SSR pode fornecer aos usuários um carregamento mais eficiente da aplicação, já que parte da renderização é feita no servidor. Isso reflete em uma menor exigência do hardware do cliente. Quanto ao SEO, o SSR colabora com o processo de indexação, uma vez que a página é entregue já renderizada. **É vital entendermos que, conceitualmente, um SSR pode prover uma SPA.** A diferença é que a página será, obviamente, renderizada no lado do servidor e as interações subsequentes no lado no cliente.

Quanto às desvantagens, pode-se dizer que o TTFB (Time To First Byte), em aplicações SSR, é maior, dado que o servidor precisa renderizar o conteúdo antes de

enviar a resposta para o cliente. O TTFB, então, se trata do tempo entre o recebimento da requisição por parte do servidor e o envio do primeiro conteúdo a ser renderizado pelo cliente. Além disso, no SSR, após o cliente receber a página e exibi-la, ela apenas se torna interativa na finalização do carregamento, o que pode causar uma certa estranheza caso exista muita demora do momento da exibição da página até a finalização do carregamento.

Frameworks como Nuxt.js, Next.js e o Angular Universal suportam Server Side Rendering.

SSG – Static Site Generation

SSG é uma sigla para Static Site Generation, ou Geração de Site Estática. O SSG se assemelha bastante ao SSR, onde o **Javascript usado para gerar o HTML é executado no lado do servidor**, porém isso acontece em **tempo de build.** Ou seja, a página é gerada apenas uma vez e mantida no cache do servidor. Todos os requests para essa página irão retornar exatamente o mesmo HTML. Isso torna a interação muito mais performática, computacionalmente barata e traz os mesmos benefícios do SSR quanto ao SEO. Essa estratégia é muito útil para páginas em que o conteúdo não é dinâmico.

ISR – Incremental Static Regeneration

ISR é uma sigla para Incremental Static Regeneration, ou Regeneração Estática Incremental. O SSG se assemelha bastante ao SSG, onde **o Javascript usado para gerar o HTML é executado no servidor em tempo de build**, porém essa página é gerada novamente continuamente em um intervalo de tempo específico. Ou seja, a página é gerada e mantida em cache e todos os requests a essa página retornam exatamente o mesmo HTML; esse processo, porém, é realizado novamente depois de algum tempo e os request subsequentes passam a retornar o novo HTML. É muito útil quando precisamos do server uma página web que não tem a necessidade de ser atualizada em um curto intervalo de tempo.

Capítulo 5. NEXT.JS

Segundo a documentação oficial, o Next.js entrega a melhor experiência de desenvolvimento com todas as features de que você precisa para produção: static híbrido/server rendering, suporte a TypeScript, build inteligente, pre-fetching de rota e mais. **Sem configurações necessárias**.

O Next.js é um framework em cima do React.js que oferece uma excelente experiência de desenvolvimento. Por ser um Framework opinativo, não precisamos resolver diversos problemas, pois o Next nos entrega soluções já validadas pela comunidade. Dessa forma, as facilidades do Next.js começam no setup do projeto, já que, após a execução de um Script, temos um projeto pronto, configurado para TypeScript e com o processo de build já pronto.

Setup

```
npx create-next-app@latest --typescript
# or
yarn create next-app --typescript
```

E pronto. Após isso, nossa aplicação está pronta para ser desenvolvida. Não são necessários passos adicionais para acessar nossa página com o boilerplate code gerado. Dependendo do seu package manager, execute **npm run dev** ou **yarn dev,** e sua aplicação será servida em: http://localhost:3000.

Páginas

O Next.js conta com um sistema muito interessante para o roteamento. Ele é baseado em diretórios. Pode ser observado na estrutura inicial de diretórios criados após o create-next-app que existe uma pasta chamada pages. Entenda essa pasta como

a raiz das rotas da sua aplicação. Por exemplo, se você criar um arquivo js/jsx dentro dessa página com o nome de **teste.tsx**:

```
function Teste() {
  return <div>This is a test</div>
}

export default Teste
```

É tudo de que precisamos para poder acessar uma nova rota. No nosso caso, http://localhost:3000/teste.

As páginas também podem ser dinâmicas. Para isso, podemos criar algo como **pages/content/[id].tsx**. Agora podemos fazer algo como: http://localhost:3000/content/1 ou http://localhost:3000/content/thisIsAnID, e o texto presente no lugar de [id] fica disponível para nossa aplicação. O conteúdo do arquivo [id].tsx seria algo como o seguinte:

```
import { useRouter } from 'next/router'

const Content = () => {
  const router = useRouter()
  const { id } = router.query

  return <p>Param content: {id}</p>
}

export default Content
```

Pré-renderização

As páginas do Next.js podem ou não ser pré-renderizadas. Existem dois tipos de pré-renderização no Next: Static Generation (**SSG**) e Server-side Rendering (**SSR**). E, como já vimos a explicação para esses termos no capítulo anterior, não nos aprofundaremos em seus conceitos aqui, e sim na sintaxe necessária para usar cada um deles.

Algo importante de pontuar é que o Next permite que você escolha o tipo de renderização para cada página. Sua aplicação pode ser um misto de páginas SSG e SSR.

Você também pode usar **CSR** junto ao SSR e ao SSG, onde algumas partes da sua página são totalmente renderizadas no servidor e outras no Frontend.

Static Generation

Por padrão, o Next.js gera **todas as páginas** como estáticas em tempo de build. Algumas dessas páginas podem ou não precisar de acessar uma API externa à nossa aplicação. Para os casos em que a página não necessita de nenhum dado, não precisamos de nenhum passo extra:

```
function About() {
  return <div>About</div>
}

export default About
```

Porém, se, mesmo em tempo de build, precisarmos acessar uma API externa:

```
export async function getStaticProps() {
  // Chama API externa para buscar a versão
  const res = await fetch('https://.../productId)
```

```
  const versionObject = await res.json()

  // Ao retornar { props: { posts } }, o componente
  // about recebe versionObject como prop
  return {
    props: {
      versionObject,
    },
  }
}


function About({ versionObject }) {
  return <div>About version: {versionObject}</div>
}

export default About
```

Perceba que implementamos uma função assíncrona especial chamada de getStaticProps(). A partir da implementação dessa função, o Next.Js identifica que essa página possui dependências e que essa função precisa ser chamada no momento em que essa página é gerada; no caso, durante o build. Entenda que aqui a página continua sendo estática, uma vez que ela é gerada apenas durante o build. O que muda apenas é que, para montá-la, o getStaticProps precisa ser executado antes. Existem algumas outras propriedades que podem ser retornadas no getStaticProps. Para conhecê-las, recomendamos a leitura da documentação oficial.

O Next permite também que, caso as rotas da sua aplicação dependam de uma informação externa, você também consiga gerar essas páginas em tempo de build. Esse conceito é um pouco mais complicado que o anterior, então leiam com atenção.

Imagine que queremos pré-renderizar as páginas referentes aos 10 produtos mais vendidos na nossa plataforma. Não seria ideal termos que manualmente criar 10

páginas diferentes. O Next.js resolve essa questão com uma função complementar à anterior: getStaticPaths(). Com o getStaticPaths, podemos acessar nossa API para receber os 10 produtos mais usados e então retornar, por exemplo, os IDs de cada item dessa listagem, de forma que cada um desses valores fique disponível na função getStaticProps. Assim, o Next.js entende que queremos que a página com rota dinâmica, /produto/[id], seja gerada para cada um dos IDs retornados em getStaticPaths.

```
// Função chamada em tempo de build
export async function getStaticPaths() {
  // Chamamos a API externa
  const res = await fetch('https://.../products?top=10')
  const posts = await res.json()

  // Gera a lista com os IDs que usaremos para gerar as
páginas estáticas
  const paths = products.map((product) => ({
    params: { id: product.id },
  }))

  // { fallback: false } significa que outras rotas (além das
retornadas no getStaticPaths devem retornar 404
  return { paths, fallback: false }
}

// Essa função também é chamada em tempo de build
export async function getStaticProps({ params }) {
  // params contém o product `id`.
  // Se a rota for como /product/1, então params.id é 1
  const res = await fetch(`https://.../products/${params.id}`)
  const post = await res.json()

  // Passa os valores do produto para a página via props
  return { props: { post } }
}
```

```
function Products({product}) {
  // Renderiza Produtos...
}

export default Products
```

Server-side Rendering

O **SSR** no Next.JS funciona de forma parecida com o Static Generation. É esperado que uma função específica seja implementada, e essa função é a getServerSideProps(). A principal diferença aqui é que essa página passa a ser gerada a cada request:

```
function Page({ data }) {
  // Renderiza data...
}

// Essa função é chamada a cada request
export async function getServerSideProps() {
  // Busca informações da API
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  // Passa as informações via props
  return { props: { data } }
}

export default Page
```

Incremental Site Regeneration

Por último, temos o **ISR.** Como explicado antes, o ISR difere do SSG pelo fato de que a página é gerada em tempo de build, porém é gerada novamente depois de um intervalo específico de tempo. Para obter esse comportamento no Next.js, basta adicionar uma propriedade a mais no objeto retornado da função getStaticProps:

```
// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// revalidation is enabled and a new request comes in
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js irá  tentar gerar novamente a página quando um
novo request chegar, porém apenas se o tempo desde o último
request for superior a 10 segundos
    revalidate: 10, // Em segundos
  }
}
```

Features mais relevantes

Além das páginas, do sistema de roteamento e das siglas SSR, SSG, ISG, o Next.js nos entrega um conjunto de features úteis. Vamos listar aquelas que, a nosso ver, são mais relevantes, e falar brevemente sobre elas.

Image Optimization

O componente de imagem do Next, next/image oferece uma variedade de otimizações quanto à performance. Segundo a documentação oficial, estas são as vantagens do next/image:

- **Performance melhorada**: sempre serve as imagens com os tamanhos corretos para cada device, usando formatos de imagem modernos.
- **Estabilidade visual**: impede “layout shifts”, que são as “quebras” no layout devido ao posterior carregamento das imagens.
- **Carregamentos mais rápidos das páginas**: imagens são carregadas apenas quando elas entram no viewport. Podemos ainda usar um efeito de blur como placeholder.
- **Flexibilidade de assets**: redimensionamento de imagens sob demanda, até para imagens armazenadas em servidores remotos.

Internacionalização

Next provê suporte para rotas internacionalizadas (i18n) a partir da versão v10.0.0. Você pode prover uma lista de locales, locale padrão e locales de domínio específico. O Next irá automaticamente cuidar das rotas.

Fast Refresh

Fast Refresh é uma feature do Next.js que entrega feedback instantâneo a alterações no código fonte do seus componentes, mantendo o estado do componente.

Bundles com separação de código

O Next.js otimiza o tamanho dos bundles enviados ao navegador com base no código que realmente está sendo usado naquele momento. Isso faz com que o carregamento das páginas aconteça de maneira muito mais rápida. Essa é uma feature chave para o Next.

Conclusão

O Next.js é um framework extremamente poderoso e que torna nossa experiência de desenvolvimento muito mais agradável. Nessa apostila, cobrimos os principais conceitos básicos. É essencial a leitura da documentação, agora que vocês possuem uma base sobre o Next.js.

Capítulo 6. Teste de aplicações WEB

Até então falamos intensamente sobre questões relacionadas a desenvolvimento. Neste capítulo, passaremos a discutir sobre aspectos de Qualidade de Software e técnicas de desenvolvimento que visam garantir um código mais seguro e documentado por meio da escrita de testes.

Obviamente não é o caso de todos, mas é muito natural encontrar uma resistência do dos desenvolvedores quando o assunto é escrita de teste. Muitas vezes, testes são tidos como algo descartável e não essencial para as aplicações. E, se você parar para refletir, essa linha de raciocínio pode até fazer um certo sentido, visto que uma aplicação funciona sem testes, mas não sem código fonte. Mas então por que ainda assim mais e mais empresas do mercado adicionam às exigências para vagas assuntos referentes a teste?

O mercado vem enxergando cada vez mais o verdadeiro valor dos testes em aplicações complexas. Como eles previnem problemas em ambiente de produção, facilitam a manutenção e tornam o código documentado. Com a rotatividade dos profissionais de T.I., é bastante provável que a próxima pessoa a dar manutenção naquele código que hoje funciona muito bem nunca o tenha visto. Nesse caso, os testes funcionam como documentação das regras de negócio daquela implementação.

Existem algumas técnicas que visam inserir os testes no processo de desenvolvimento. Logo, quanto mais cedo existirem testes, menos provável que existam problemas em ambiente de produção.

TDD – Test Driven Development

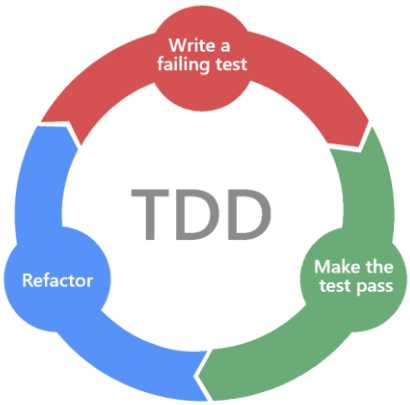
TDD é a sigla para Test Driven Development, ou Desenvolvimento Orientado por Testes. O TDD define ciclos de interações pequenos e repetidos durante o desenvolvimento da aplicação inúmeras vezes. As interações são:

- **Escrita de teste**: antes de existir a implementação da funcionalidade, o desenvolvedor deve definir testes em que fique claro que, para a entrada X, é esperada a saída Y. Obviamente esse teste irá falhar no princípio.

- **Desenvolvimento:** umas vez que temos o teste definido e falhando, o nosso objetivo nessa interação é desenvolver um código que seja capaz de fazer o teste definido anteriormente passar. Esse código deve ser o mais limpo possível, sem muita preocupação com otimização.

- **Refatoração:** durante essa interação, os testes já estão passando. Nosso objetivo é fazer com que o código esteja otimizado em questões de boas práticas, como nome de variáveis, extração de métodos etc. Obviamente essas alterações precisam ser realizadas de forma que os testes continuem passando.

Figura 1 – Logo dos frameworks/bibliotecas



Fonte: https://marsner.com/wp-content/uploads/test-driven-development-TDD.png

Nosso software é desenvolvido com base nos testes, e não o contrário. Esse é o princípio do TDD.

Jest

O Jest é um poderoso Framework de Testes em JavaScript com um foco na simplicidade. Ele é agnóstico quanto à ferramenta que está sendo usada, contanto que ela seja desenvolvida em Javascript (typescript também, obviamente). Além disso, é largamente usado em aplicações Node, React, Angular, Vue etc.

Algumas de suas características são:

- **Nenhuma configuração necessária:** Out-of-the-box pronto. Na maioria dos projetos Javascript, não é necessária nenhuma alteração de configuração adicional.
- **Testes Snapshot:** Possibilidade de testar grandes objetos com suas versões anteriores.
- **Isolamento:** Os testes rodam paralelamente em seus próprios projetos. Dessa forma, a performance é otimizada.
- **Cobertura de código:** É possível gerar relatórios de cobertura de código
- **Facilidade em mocking:** Jester provê uma maneira fácil e elegante para a criação de mocks.

Instalação

A instalação do Jest é bastante simples:

yarn add --dev jest ou npm install --save-dev jest

Na lista de scripts, no package.json:

```
{
 "scripts": {
   "test": "jest"
 }
}
```

Na prática

O Jest é extremamente simples quanto à sintaxe. Vamos definir um teste simples e executá-lo:

```
function sum(a, b) {
  return a + b
}

describe("Calculator tests", () => {
  it('should sum 1 + 1 and return 2', () => {
    expect(sum(1+1)).toBe(2);
  });
})
```

No console, em seu projeto:

```
yarn run teste
```

E então o resultado:

PASS ./sum.test.js << Nome do arquivo de teste

✓ should sum 1 + 1 and return 2 (5ms)

A função **describe** é usada para definir uma descrição para um conjunto de testes. Seu primeiro parâmetro é exatamente a descrição, e o segundo, um callback com os testes a serem executados dentro dessa descrição.

A função it (ou test) define um teste. O seu primeiro parâmetro é a descrição deste teste. É comum vermos textos como "should do something and return something", e, por esse motivo, o it é usado como alternativa ao TEST. Dessa forma, a leitura do teste fica "it should do something and return something". O seu segundo parâmetro é um callback de fato do código do teste.

Usamos o **expect** para definir o valor que usaremos com base para as comparações. O expect retorna um objeto com diversos métodos como propriedades. No nosso exemplo, usamos o **toBe,** que recebe como parâmetro o valor-alvo para a comparação. No caso, a leitura da linha do nosso teste seria: Expect sum(1, 1) to be 2 ou **espera-se que o resultado de sum(1, 1) seja 2.** O **expect** é um exemplo de Matchers. Existem vários outros como toBeNull(), toBeDefined(), toBeFalsy etc. O Jest possui uma vasta e clara documentação, o que torna inviável cobrir todos os métodos aqui. Recomendamos, então, a leitura da documentação oficial. Apresentamos aqui o básico para seguirmos com nossa aplicação.

React Test Library

O React Testing Library é uma biblioteca construída sobre a DOM Test Library. Ela provê APIs para interação com componentes React. O desenvolvimento dessa biblioteca foi motivado pelo desejo de desenvolver testes de fácil manutenção. Para isso, faz-se necessário que os detalhes de implementação tenham a menor influência possível no resultado do teste. Idealmente nossos testes devem testar apenas funcionalidade, e não implementação. Com o React Test Library, a montagem e interação com o componente que está sendo testado é feita de maneira muito simples.

Majoritariamente, no contexto do React/Next.js, o React Test Library é usado junto com o Framework Jest.

Instalação

Projetos criados com o Create React App já possuem o React Testing Library instalado. Para os demais projetos, é possível adicionar a biblioteca dessa forma:

```
npm install --save-dev @testing-library/react
```

Ou de maneira semelhante para o Yarn.

Na prática

O uso de React Test Library é bem simples e direto.

```
import { render, screen, fireEvent } from "@testing-
library/react";
import { Lista } from "../components/Lista"

it("Should contain name rodrigo borba on List", () => {
    render(<Lista contacts={undefined} />);

    expect(screen.queryByText(/rodrigo
borba/i)).not.toBeInTheDocument();
  });
```

Destrinchando o código acima, percebemos a função **render**, que é a primeira contribuição do React Testing Library com a qual estamos nos deparando. O parâmetro que estamos passando para essa função é o componente que queremos renderizar para nosso teste, poupando-nos do trabalho de implementar um mecanismo para instanciar o componente e adicioná-lo ao DOM. Também podemos perceber alguns elementos que já discutimos na seção do Jest, como o **It** e o **expect.** Também vemos agora novos

Matchers: **not** e o **toBeInTheDocument.** Como o nome da função já diz, o **toBeInTheDocument** verifica se a informação do **expect** está presente no documento. O modificador **NOT** inverte a saída. Ou seja, no caso, queremos que o nome rodrigo borba não esteja presente no documento.

Outra diferença que vemos é o uso do **screen.** Após a chamada do **render,** nosso componente é adicionado ao DOM. O screen nos permite acessar justamente os elementos do DOM. No nosso exemplo, estamos buscando elementos em que o texto é rodrigo borba. O **queryByText** é um matcher do react-testing-library. Perceba também que não estamos simplesmente passando o texto rodrigo borba, mas sim /rodrigo borba/i, que é uma **regex.** Nesse caso, estamos usando o modificador i, que torna nossa busca case-insensitive.

```
it("Should have rodrigo after click ", () => {
    render(<Filtro />);

    const button = screen.getByText("Adiciona Rodrigo");
    fireEvent.click(button);

        expect(screen
                .queryByText(/rodrigo/i))
                   .toBeInTheDocument();
  });
```

No exemplo acima, vemos outra funcionalidade do react test library, em que nos é permitida a interação com os elementos do DOM. No nosso exemplo, buscamos por um botão que possua um texto específico. Na sequência, clicamos nesse botão com o **fireEvent**. O fireEvent nos permite emular ações de um usuário em um componente. Por último, verificamos se o texto rodrigo foi de fato adicionado ao DOM.

Uma diferença que pode passar despercebida é o uso do get, query ou find. No caso, getBy…, queryBy… ou findBy…. Abaixo explicamos como cada um dos modificadores funciona.

getBy...: retorna o elemento de acordo com o parâmetro. Caso não seja achado nenhum elemento, imediatamente lança uma exceção descrevendo o erro. Caso sejam achados mais de um elemento, uma exceção também é lançada. Ou seja, esse modificador deve ser usado quando se espera que o elemento exista na tela. Caso contrário, a execução do teste será abortada, a não ser que a chamada do getBy esteja envolvida em um try…catch. Mas veremos que existem maneiras melhores de garantir que o elemento não exista.

queryBy...: retorna o elemento de acordo com o parâmetro. Caso nenhum elemento seja achado, retorna nulo. Lança uma exceção caso mais de um elemento seja achado. Esse modificador geralmente é usado para verificar se um elemento específico não existe na DOM.

findBy...: retorna uma **PROMISE** que é resolvida assim que o elemento é localizado. A promise é rejeitada caso nenhum elemento seja localizado depois de um tempo padrão de 1000ms. Caso você deseje buscar por mais de um elemento, use o **findAllBy.**

Referências

GETTING Started. **Next.js**, [*S. l.*], c2022. Disponível em: https://nextjs.org/docs. Acesso em: 15 mar. 2022.

JEST. [*S. l.*], c2022. Disponível em: https://jestjs.io/. Acesso em: 15 mar. 2022.

REACT Hooks Testing Library. [*S. l.*], c2022. Disponível em: https://react-hooks-testing-library.com/. Acesso em: 15 mar. 2022.

REACT QUERY. **Performant and powerful data synchronization for React**. [*S. l.*], c2020. Disponível em: https://react-query.tanstack.com/. Acesso em: 15 mar. 2022.

STYLED-COMPONENTS. [*S. l.*], c2022. Disponível em: https://styled-components.com/. Acesso em: 15 mar. 2022.

SWR. **React Hooks for Data Fetching**. [*S. l.*], c2022. Disponível em: https://swr.vercel.app/. Acesso em: 15 mar. 2022.

TESTING LIBRARY. **Simple and complete testing utilities that encourage good testing practices**. [*S. l.*], 2022. Disponível em: https://testing-library.com/. Acesso em: 15 mar. 2022.

WALBLER, Scott. Introduction to Test Driven Development (TDD). **Agile Data**, [*S. l.*], 2022. Disponível em: http://agiledata.org/essays/tdd.html. Acesso em: 15 mar. 2022.