



WACAD0014

Biblioteca Frontend React

Aula 04

Júlia Luiza

jlslc@icomp.ufam.edu.br

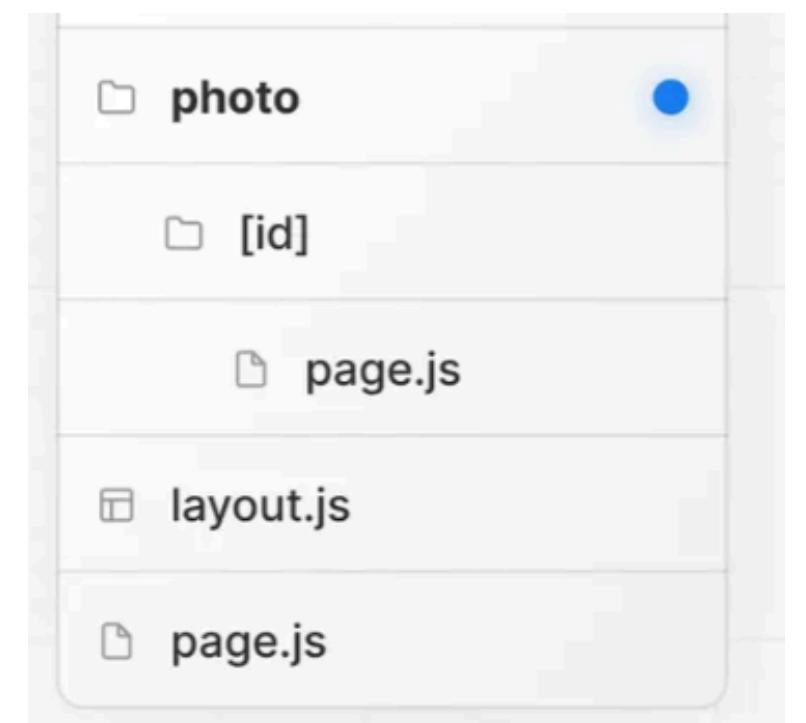
Cronograma: Aula 04

- Rotas dinâmicas no Next;
 - Atividade Prática
- Axios no React;
- Biblioteca React-query;
 - useQuery;
 - useMutation;
 - Atividade Prática
- Formulários no React;
 - Tipos de Input;
 - Validação;
 - react-hook-form;
 - Atividade Prática
- Boas práticas gerais.

Rotas dinâmicas no Next

Como já vimos nas aulas anteriores e também no exemplo de uso do hook `useParams`, o Next possui o recurso de **rotas dinâmicas**.

- De maneira geral, para criar rotas dinâmicas basta **aninhar** a criação de páginas dentro de pastas que representem então os parâmetros dinâmicos;
- Se você deseja que exista uma rota de detalhes de produto, por exemplo, onde seja possível passar qualquer 'nome' de produto, temos então um caso de rota dinâmica:
 - Nesse caso, basta decidir o nome do parâmetro que irá representar o nome do produto e estabelecer a rota:
 - i. `www.site.com/detalhes-produto/[nomeParametro]`
 - ii. `www.site.com/[nomeParametro]`
 - No projeto, a opção i iria refletir nas pastas da seguinte forma:
 - `app/detalhes-produto/[nomeParametro]/page.tsx`
 - E a opção ii ficaria assim:
 - `app/[nomeParametro]/page.tsx`



`/photos/[id] -> /photos/1`

Rotas dinâmicas no Next

Além disso, as rotas dinâmicas podem ser estendidas para abranger todos os parâmetros subsequentes adicionando reticências entre colchetes: [...folderName]

- Por exemplo, app/shop/[...slug]/page.js corresponderá a /shop/clothes, mas também a /shop/clothes/tops, /shop/clothes/tops/t-shirts e assim por diante.

Route	Example URL	params
app/shop/[...slug]/page.js	/shop/a	{ slug: ['a'] }
app/shop/[...slug]/page.js	/shop/a/b	{ slug: ['a', 'b'] }
app/shop/[...slug]/page.js	/shop/a/b/c	{ slug: ['a', 'b', 'c'] }

<https://nextjs.org/docs/app/building-your-application/routing/dynamic-routes#catch-all-segments>

Hooks do Next e rotas dinâmicas: Vamos praticar?



Agora olhando para nossa listagem de produtos, vamos adicionar uma nova funcionalidade:

1. No componente CardProduto, adicione um novo botão "ver detalhes";
 - a. [código bootstrap aqui](#)
2. Crie uma nova página com rota dinâmica, com a seguinte configuração:
 - a. diretório: app/produto/[produto]/page.tsx
 - b. rota: <http://localhost:3000/produto/notebook>
3. Na nova página, implemente um componente client side Produto, com os requisitos a seguir:
 - a. Deve possuir um estado local para **produto**;
 - b. Deve utilizar o hook `useParams()` para identificar o parâmetro 'produto' na url;
 - c. Assim que identificar, deve utilizar o `useEffect()` para realizar um fetch dos detalhes do produto, no seguinte endpoint:
<https://ranekapi.origamid.dev/json/api/produto/{nomeProduto}>, e então atualizar o estado de produto;
 - d. O componente deve exibir em tela os detalhes do produto, quando houver um. Caso contrário, deve renderizar o estado de carregamento.
 - i. [código bootstrap aqui](#)

Hooks do Next e rotas dinâmicas: Vamos praticar?



1. De volta ao componente CardProduto, agora realize as seguintes alterações:
 - a. No botão ver detalhes criado anteriormente, adicione um `onClick` para uma função chamada `verDetalhesProduto`;
 - b. Implemente então a função, que deve receber como argumento o nome do produto e realizar um `router.push()` para a rota dinâmica, passando o nome do produto como parâmetro;
 - i. Utilize o hook `useRouter()` do next/navigation
2. Lembre-se de tipar corretamente o estado de Produto e demais parâmetros

Hooks do Next e rotas dinâmicas: Vamos praticar?



Loja WA Início Carrinho Sair

Resumo do Carrinho

Quantidade total: 0

Valor total: R\$0.00

Produtos disponíveis:

A photograph of a silver notebook with a yellow floral patterned cover.

Notebook

R\$ 2300

[Adicionar no carrinho](#)

[Ver detalhes](#)

A photograph of a black smartphone lying next to a white laptop keyboard.

Smartphone

R\$ 2399

[Adicionar no carrinho](#)

[Ver detalhes](#)

A photograph of a black Canon camera with a strap.

Câmera

R\$ 2199

[Adicionar no carrinho](#)

[Ver detalhes](#)

A photograph of a black smartwatch with a leather strap.

Smartwatch

R\$ 1199

[Adicionar no carrinho](#)

[Ver detalhes](#)

[veja aqui o gif](#)

Axios no React

Até agora, implementamos o fetch de dados por meio da própria API de 'fetch' da web. Contudo, em projetos maiores usualmente torna-se necessário a utilização de bibliotecas e/ou pacotes que facilitem a camada de requisições.

- Uma das mais populares é o **Axios**;
- **"Axios é um cliente HTTP simples baseado em promessa para o navegador e node.js. Axios fornece uma biblioteca simples de usar em um pacote pequeno com uma interface muito extensível."**

A X I O S

```
import axios from "axios";
axios.get('/users')
  .then(res => {
    console.log(res.data);
});
```

<https://axios-http.com/>

Axios no React

Para implementar em um projeto React, é bem simples:

- Deve-se instalar utilizando npm:
 - `npm install axios`
- Criar uma instância do Axios, definindo a url base das requisições:

```
src > app > services > TS api.ts > ...
1 import axios from "axios";
2
3 const api = axios.create({
4   baseURL: "https://ranekapi.origamid.dev/json/api",
5 });
6
7 export default api;
```

- Substituir ou implementar o fetch de dados pelos métodos disponíveis no axios. Exemplo:

```
useEffect(() => {
  // fetch(`https://ranekapi.origamid.dev/json/api/produto/${params.produto}`)
  //   .then((response) => response.json())
  //   .then((produto) => setProduto(produto));

  api.get(`/produto/${params.produto}`).then((response) => {
    setProduto(response.data);
  });
}, [params.produto]);
```

Axios no React

Da mesma forma que com o fetch API, também podemos fazer uso da função de limpeza do useEffect para requisições realizados por meio do axios:

```
useEffect(() => {
  const controller = new AbortController();

  api
    .get(`/produto/${params.produto}`, {
      signal: controller.signal,
    })
    .then((response) => {
      setProduto(response.data);
    })
    .catch((error) => {
      console.error(error);
    });

  return () => controller.abort();
}, [params.produto]);
```

Axios no React

```
{  
  // `data` is the response that was provided by the server  
  data: {},  
  
  // `status` is the HTTP status code from the server response  
  status: 200,  
  
  // `statusText` is the HTTP status message from the server response  
  // As of HTTP/2 status text is blank or unsupported.  
  // (HTTP/2 RFC: https://www.rfc-editor.org/rfc/rfc7540#section-8.1.2.4)  
  statusText: 'OK',  
  
  // `headers` the HTTP headers that the server responded with  
  // All header names are lower cased and can be accessed using the bracket notation.  
  // Example: `response.headers['content-type']`  
  headers: {},  
  
  // `config` is the config that was provided to `axios` for the request  
  config: {},  
  
  // `request` is the request that generated this response  
  // It is the last ClientRequest instance in node.js (in redirects)  
  // and an XMLHttpRequest instance in the browser  
  request: {}  
};  
  
https://axios-http.com/docs/response\_schema
```

React Query

Mesmo que facilite a camada de requisições, há algumas outras funcionalidades importantes não cobertas pelo Axios, por isso ele costuma ser utilizado em conjunto com outras bibliotecas mais robustas e completas, como por exemplo o React Query.

- "TanStack Query (FKA React Query) é frequentemente descrito como a **biblioteca de fetch de dados necessária para aplicativos da web**, mas em termos mais técnicos, **facilita muito o fetch, o armazenamento em cache, a sincronização e a atualização do estado do servidor** em seus aplicativos da web.";
- "É sem dúvida uma das melhores bibliotecas para gerenciar o estado do servidor"
 - "Ele funciona incrivelmente bem imediatamente, sem configuração e pode ser personalizado de acordo com sua preferência à medida que seu aplicativo cresce."



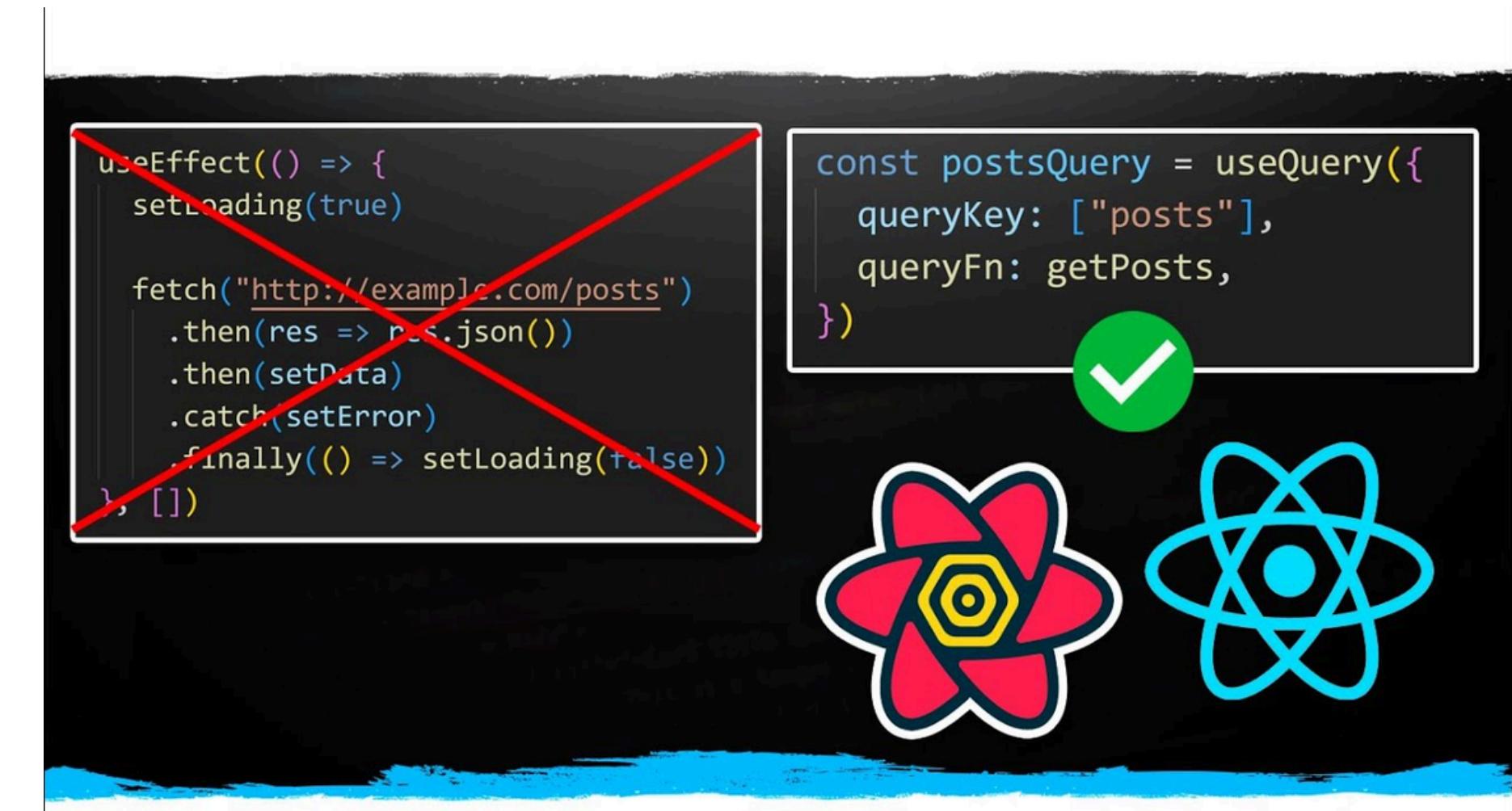
React Query

<https://tanstack.com/query/latest/docs/framework/react/overview>

React Query

Principais Recursos

- **Buscar Dados Assíncronos:** Facilita a busca de dados de APIs ou fontes externas.
- **Cache Automático:** Gerencia automaticamente o cache de dados para reduzir requisições desnecessárias.
- **Atualizações em Tempo Real:** Suporta atualizações automáticas de dados em tempo real.
- **Gestão de Estados de Carregamento:** Simplifica o tratamento de estados de carregamento e erros.
- "Ajudá-lo a remover muitas linhas de código complicado e mal compreendido de seu aplicativo e substituí-lo por apenas algumas linhas de lógica React Query."



https://www.youtube.com/watch?app=desktop&v=lVLz_ASqAio

React Query

Setup

- Antes de efetivamente implementar o react-query nas chamadas da aplicação, é necessário instanciar um 'client' do mesmo e compartilhá-lo em toda a aplicação.
 - Em outras palavras, **deve-se inicializar o react-query**;
 - Isso deve ser feito por meio da função `QueryClient()` em conjunto com o `QueryClientProvider`;
 - Em um projeto React com Next App Router, uma forma de implementar esse passo é por meio de um Client Component, parecido com o que fizemos para o Bootstrap do JS.

React Query

Setup

```
ReactQueryClient.tsx U X
src > app > components > ReactQueryClient.tsx > ...
1  "use client";
2
3  import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
4  import { useState } from "react";
5
6  export const ReactQueryClientProvider = ({  
7    children,  
8  }: {  
9    children: React.ReactNode;  
10 }) => {  
11  const [queryClient] = useState(() => new QueryClient());  
12  return (  
13    <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>  
14  );  
15};
```

```
export default function RootLayout({  
  children,  
}: Readonly<{  
  children: React.ReactNode;  
}>) {  
  return (  
    <html lang="pt-br">  
      <body>  
        <ReactQueryClientProvider>  
          <Navbar />  
          {children}  
          <BootstrapClient />  
        </ReactQueryClientProvider>  
      </body>  
    </html>  
  );  
}
```

[código aqui](#)

React Query

useQuery

- o hook **useQuery** gerencia automaticamente o carregamento, o erro e a recuperação dos dados de uma query (ou consulta);
- Para utilizar nos Componentes, deve-se chamá-lo com pelo menos:
 - Uma chave exclusiva para identificar a query;
 - Uma função assíncrona que retorna uma promessa;
- A chave exclusiva que você fornece é usada internamente para buscar novamente, armazenar em cache e compartilhar suas consultas em todo o aplicativo. Deve ser um array.
- A função deve retornar dados ou um erro.

```
export async function fetchTodoList() {
  return await api.get("/todos");
}

function Exemplo() {
  const result = useQuery({ queryKey: ["todos"], queryFn: fetchTodoList });
}
```

React Query

useQuery

- O objeto de **result** contém alguns estados muito importantes. Uma consulta só pode estar em um dos seguintes estados em um determinado momento:
 - **isPending** ou `status === 'pending'` - A consulta ainda não possui dados
 - **isError** ou `status === 'error'` - A consulta encontrou um erro
 - **isSuccess** ou `status === 'success'` - A consulta foi bem-sucedida e os dados estão disponíveis
- Além desses estados primários, mais informações estão disponíveis dependendo do estado da consulta:
 - **error** - Se a consulta estiver no estado `isError`, o erro estará disponível por meio da propriedade `error`.
 - **data** - Se a consulta estiver no estado `isSuccess`, os dados estarão disponíveis por meio da propriedade `data`.
 - **isFetching** - Em qualquer estado, se a consulta for buscada a qualquer momento (incluindo nova busca em segundo plano), `isFetching` será verdadeiro.

React Query

useQuery

- Exemplo completo de uso:

```
function Example() {
  const { isPending, error, data } = useQuery({
    queryKey: ["repoData"],
    queryFn: () =>
      fetch("https://api.github.com/repos/TanStack/query").then((res) =>
        res.json()
      ),
  });

  if (isPending) return "Carregando...";

  if (error) return "Ocorreu um erro: " + error.message;

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.description}</p>
      <strong>👀 {data.subscribers_count}</strong>{" "}
      <strong>⭐ {data.stargazers_count}</strong>{" "}
      <strong>🍴 {data.forks_count}</strong>
    </div>
  );
}
```

React Query e Axios

Com o React Query e Axios no mesmo projeto, podemos então ser mais intencionais na organização das requisições, como por exemplo:

- **Separando cada requisição em funções específicas com nomeação mais clara.** Exemplo: "getListProdutos", "getDetalhesProduto", etc.
- **Para cada função de requisição, criar um hook específico customizado** com nomeação também mais clara, e que faça uso do react-query;
 - O que torna fácil a reutilização da mesma request em Componentes diferentes;
 - Melhora a organização e facilita a manutenção;
 - Já implementa por padrão mecanismos de cache, retry, etc, que são funcionalidades do react query.

React Query e Axios

```
export default function ListagemProdutos({  
    adicionarAoCarrinho,  
}: IListagemProdutos) {  
    const { produtos, isPending, isError } = useListaProdutos();  
  
    if (isPending) return <h5>Carregando...</h5>;  
    if (isError) return <h5>Ocorreu um erro ao carregar os produtos.</h5>;  
  
    if (!produtos) return <h5>Não há produtos disponíveis no momento.</h5>;  
  
    return (  
        <>  
            <h5 className="mb-3">Produtos disponíveis:</h5>  
  
            <div className="row row-cols-1 row-cols-md-2 row-cols-lg-4 g-3">  
                {produtos.map((produto) => (  
                    <CardProduto  
                        key={produto.id}  
                        produto={produto}  
                        adicionarAoCarrinho={adicionarAoCarrinho}  
                    />  
                ))}  
            </div>  
        </>  
    );  
}
```

```
app > hooks > TS useListaProdutos.ts > ...  
import { useQuery } from "@tanstack/react-query";  
import { getListaProduto } from "../services/produtos";  
  
export function useListaProdutos() {  
    const { data, isPending, isError } = useQuery({  
        queryKey: ["listaProdutos"],  
        queryFn: () => getListaProduto(),  
    });  
  
    return { produtos: data, isPending, isError };  
}
```

```
app > services > TS produtos.ts > ...  
import api from "./api";  
  
export async function getListaProduto(): Promise<Produto[]> {  
    return api.get("/produto").then((response) => response.data);  
}
```

[código aqui](#)

React Query e Axios: Vamos praticar?



Olhando para a página de **listagem de produtos e detalhes de produto**, faça as modificações necessárias para que o fetch ocorra por meio do axios e react-query, com os seguintes requisitos:

1. Para a listagem de produtos, pode-se implementar o exemplo da página anterior da forma que foi apresentado;
2. Para detalhes de produto, realize os mesmos passos:
 - a. Crie uma função específica para realizar o fetch por meio do axios chamada "getDetalhesProduto", também em `app/services/produtos.ts`;
 - b. Depois, na pasta de hooks, crie um novo hook customizado específico chamado "useDetalhesProduto", que deve utilizar o `useQuery` para realizar a requisição de detalhes do produto;
 - c. Por fim, altere a lógica do Componente da página de detalhes do produto para utilizar o hook criado no passo anterior, e implemente uma mensagem de tratamento para o estado de **loading**, **error** e de **não haver detalhes**.

React Query e Axios: Vamos praticar?



Observações:

- Antes de implementar o uso dos hooks, realize o **setup inicial** do react-query como explicado nos slides;
- E para instalar o react-query, basta seguir a documentação:
 - `npm i @tanstack/react-query`

React Query

useMutation

- Ao contrário das consultas, as **mutações** são normalmente usadas para **criar/atualizar/excluir** dados ou executar efeitos colaterais do servidor. Para este propósito, o react-query exporta um hook **useMutation**.
 - Seria o equivalente aos métodos PUT/POST/DELETE, enquanto que o useQuery seria para o GET.

```
function App() {
  const mutation = useMutation({
    mutationFn: (newTodo) => {
      return axios.post("/todos", newTodo);
    },
  });

  return (
    <div>
      {mutation.isPending ? (
        "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>
  );
}
```

React Query

useMutation

- Uma mutação só pode estar em um dos seguintes estados em um determinado momento:
 - **isIdle** ou status === 'idle' - A mutação está atualmente inativa ou em um estado novo/redefinido
 - **isPending** ou status === 'pendente' - A mutação está em execução no momento
 - **isError** ou status === 'error' - A mutação encontrou um erro
 - **isSuccess** ou status === 'sucesso' - A mutação foi bem-sucedida e os dados da mutação estão disponíveis
- Além desses estados primários, mais informações estão disponíveis dependendo do estado da mutação:
 - **error** - Se a mutação estiver em estado de erro, o erro estará disponível por meio da propriedade error.
 - **data** - Se a mutação estiver em estado de sucesso, os dados estarão disponíveis por meio da propriedade data.

Formulários no React

Quanto a **formulários** no React, podemos utilizar uma **variedade de abordagens**, desde o **gerenciamento de estado local** até a **utilização de bibliotecas especializadas** como Formik ou React Hook Form. Considerando a abordagem de **estado local**:

- Devemos definir o estado para o `value` e a função atualizadora para o `onChange`;
- E no `form` controlamos o que acontece ao enviar o mesmo, por isso definimos uma função para lidar com o `onSubmit`, assim como fizemos em exercícios anteriores.
 - O uso de `preventDefault()` irá prevenir o comportamento padrão, que seria de atualizar a página.

```
const App = () => {
  const [nome, setNome] = React.useState("");
  function handleSubmit(event) {
    event.preventDefault();
    console.log(nome);
  }
  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="nome">Nome</label>
      <input
        type="text"
        id="nome"
        value={nome}
        onChange={(event) => setNome(event.target.value)}
      />
      <button>Enviar</button>
    </form>
  );
};
```

Formulários no React

Abordagem de estado local

- Podemos definir um **estado para cada campo**;
- Ou podemos definir um **objeto que irá conter todos os valores** dos campos do formulário.

```
const [nome, setNome] = React.useState("");
const [email, setEmail] = React.useState("")
```

```
<input
  type="text"
  id="nome"
  value={nome}
  onChange={({event}) => setNome(event.target.value)}
/>
```

```
<input
  type="email"
  id="email"
  value={email}
  onChange={({event}) => setEmail(event.target.value)}
/>
```

```
const [form, setForm] = React.useState({
  nome: "",
  email: ""
});
```

```
function handleChange({ target }) {
  const { id, value } = target;
  setForm({ ...form, [id]: value });
}
```

```
<input type="text" id="nome" value={nome} onChange={handleChange} />
<label htmlFor="email">Email</label>
<input type="email" id="email" value={email} onChange={handleChange} />
```

Exemplo com objeto "form"

Exemplo com estado para cada campo

Formulários no React

Validação

- O **onBlur** é ativado sempre que o campo fica fora de foco, momento perfeito para validarmos o dado do campo;
- A validação pode ser feita com JavaScript utilizando REGEX;
- E para exibição do erro para o usuário, pode-se utilizar um estado local.

```
<input  
  label="CEP"  
  id="cep"  
  type="text"  
  value={cep}  
  onChange={handleChange}  
  onBlur={handleBlur}  
/>  
{error && <p>{error}</p>}
```

```
const [cep, setCep] = React.useState("");  
const [error, setError] = React.useState(null);  
  
function validateCep(value) {  
  if (value.length === 0) {  
    setError("Preencha um valor");  
    return false;  
  } else if (!/^{\d{5}-?\d{3}}$/.test(value)) {  
    setError("Preencha um cep válido");  
    return false;  
  } else {  
    setError(null);  
    return true;  
  }  
}  
  
function handleBlur({ target }) {  
  validateCep(target.value);  
}
```

Formulários no React

Tipos de inputs

- Quanto aos tipos de input, no React você pode utilizar todos normalmente, podendo seguir a mesma lógica de manter um estado local para cada outro tipo de input do seu formulário, como: Textarea; Select; Radio, Checkbox, etc.
- Contudo, caso seu formulário comece a ter vários inputs do mesmo tipo, por exemplo, uma boa oportunidade de **reutilização de código** seria a criação de um componente mais genérico;

```
const Input = ({ id, label, value, type, onChange, placeholder }) => {
  return (
    <>
      <label htmlFor={id}>{label}</label>
      <input
        type={type}
        id={id}
        name={id}
        value={value}
        onChange={onChange}
        placeholder={placeholder}
      />
    </>
  );
}

export default Input;
```

Formulários no React

```
interface FormGeneratorProps<T> {  
  onSubmit: SubmitFunction<T>;  
  fields: FieldDescriptor[];  
  controlledFields?: FieldDescriptor[];  
  buttonText: MessageDescriptor;  
  children?: ReactNode;  
  onSubmitError?: (error: Error) => void;  
  onSubmitSuccess?: (result: T) => void;  
  submitErrorMessage?: string;  
  submitSuccessMessage?: string;  
  shouldDisableLoadingOnSuccess?: boolean;  
  onChange?: (field: FieldDescriptor, value: string) => void;  
  useValidators?: boolean;  
  initialValues?: FormResult;  
}  
  
function FormGenerator<T>({
```

E indo mais além, é possível também criar um Componente inteiro de Form que seja genérico.

Dessa forma, evita-se repetição de código e propaga-se boas práticas do React.

```
<FormGenerator  
  fields={['newPassword', 'confirmPassword']}  
  controlledFields={['newPassword']}
```

onSubmit={onSubmit}

```
  onSubmitSuccess={onPasswordCreated}  
  onChange={onPasswordChange}  
  buttonText={messages.submitButton}  
  submitErrorMessage={intl.formatMessage(messages.submitError)}>
```

React Hook Form

Como dito anteriormente, outra opção para construção de formulários é a utilização de bibliotecas especializadas. Uma das mais populares no ecossistema React é a **React Hook Form**.

- Essencialmente, é uma **biblioteca de gerenciamento de formulários para React**;
- "Formulários de alto desempenho, flexíveis e extensíveis com validação fácil de usar.;"
- Baseada em hooks, o que a torna simples e eficiente;
- Minimiza a necessidade de escrever código *boilerplate*:
 - *boilerplate*: seções de código que precisam ser escritas repetidamente com pouca ou nenhuma alteração em muitos lugares dentro de um projeto.



React Hook Form

<https://react-hook-form.com/>

React Hook Form

Como usar:

- **Instalação:** Instale via npm ou yarn;
 - npm install react-hook-form
- **Uso Básico:** Inicialize o formulário, defina os campos e gerencie a submissão;
 - Para inicializar um novo formulário, utilize o hook 'useForm';
 - Registre cada campo do formulário no hook acima. Isso disponibilizará seu valor tanto para validação quanto para envio do formulário;
 - Para submissão, utilize o 'handleSubmit', também do hook.
- **Validação:** Defina regras de validação usando validadores integrados ou personalizados;
 - Lista de regras de validação suportadas: obrigatório, min, máx, tamanho mínimo, tamanho máximo, pattern, validate;
- **Tratamento de Erros:** Acesse mensagens de erro detalhadas para campos inválidos.

React Hook Form

useForm

- Ao chamar useForm() é que você obtém acesso a várias funções e objetos importantes para o gerenciamento do formulário, como:
 - **register**: Esta função é usada para registrar campos de entrada no formulário. Você a utiliza atribuindo o retorno de register para o atributo **ref** de um elemento de entrada HTML;
 - **handleSubmit**: Esta função é usada para lidar com a submissão do formulário. Você a associa ao evento **onSubmit** do formulário e passa uma função de retorno de chamada que será chamada quando o formulário for submetido com sucesso e passar na validação.
 - **formState**: É um objeto que contém informações adicionais sobre o estado do formulário, como o estado de submissão, os campos que já foram alterados e também um estado global dos erros atuais, entre outras.

```
import { useForm, SubmitHandler } from "react-hook-form";

type Inputs = {
    example: string;
    exampleRequired: string;
};

export default function App() {
    const {
        register,
        handleSubmit,
        formState: { errors },
    } = useForm<Inputs>();
```

React Hook Form

Validação e Erros

- Para validação, como dito anteriormente, pode-se utilizar as nativas do HTML normalmente;
- Para cada erro de validação, haverá uma entrada no objeto 'errors' disponível no 'formState';
- Dessa forma, podemos consultá-lo para definir a mensagem de erro que será exibida para o usuário

```
<input  
  type="password"  
  className="form-control form-control-lg"  
  id="senha"  
  {...register("senha", { required: true, minLength: 6 })}  
/>  
{errors.senha?.type === "required" && (  
  <span className="text-danger">Esse campo é obrigatório</span>  
)}  
  
{errors.senha?.type === "minLength" && (  
  <span className="text-danger">Minímo de 6 (seis) caracteres </span>  
)}
```

```
errors  
  ▼ {email: {...}, senha: {...}} ⓘ  
    ▼ email:  
      message: ""  
      ▶ ref: input#email.form-control.form-control-lg  
      type: "required"  
      ▶ [[Prototype]]: Object  
    ▼ senha:  
      message: ""  
      ▶ ref: input#senha.form-control.form-control-lg  
      type: "required"  
      ▶ [[Prototype]]: Object  
      ▶ [[Prototype]]: Object
```

React Hook Form

```
import { useForm, SubmitHandler } from "react-hook-form";

type Inputs = {
  example: string;
  exampleRequired: string;
};

export default function App() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<Inputs>();
  const onSubmit: SubmitHandler<Inputs> = (data) =>
    console.log(data);

  return (
    /* "handleSubmit" will validate your inputs before invoking "onSubmit" */
    <form onSubmit={handleSubmit(onSubmit)}>
      {/* register your input into the hook by invoking the "register" function */}
      <input defaultValue="test" {...register("example")} />

      {/* include validation with required or other standard HTML validation rules */}
      <input {...register("exampleRequired", { required: true })} />
      {/* errors will return when field validation fails */}
      {errors.exampleRequired && <span>This field is required</span>}

      <input type="submit" />
    </form>
  );
}
```

<https://react-hook-form.com/get-started>

React Hook Form: Vamos praticar?



Vamos refatorar nosso pequeno formulário da página de login para fazer uso do react hook form, da seguinte forma:

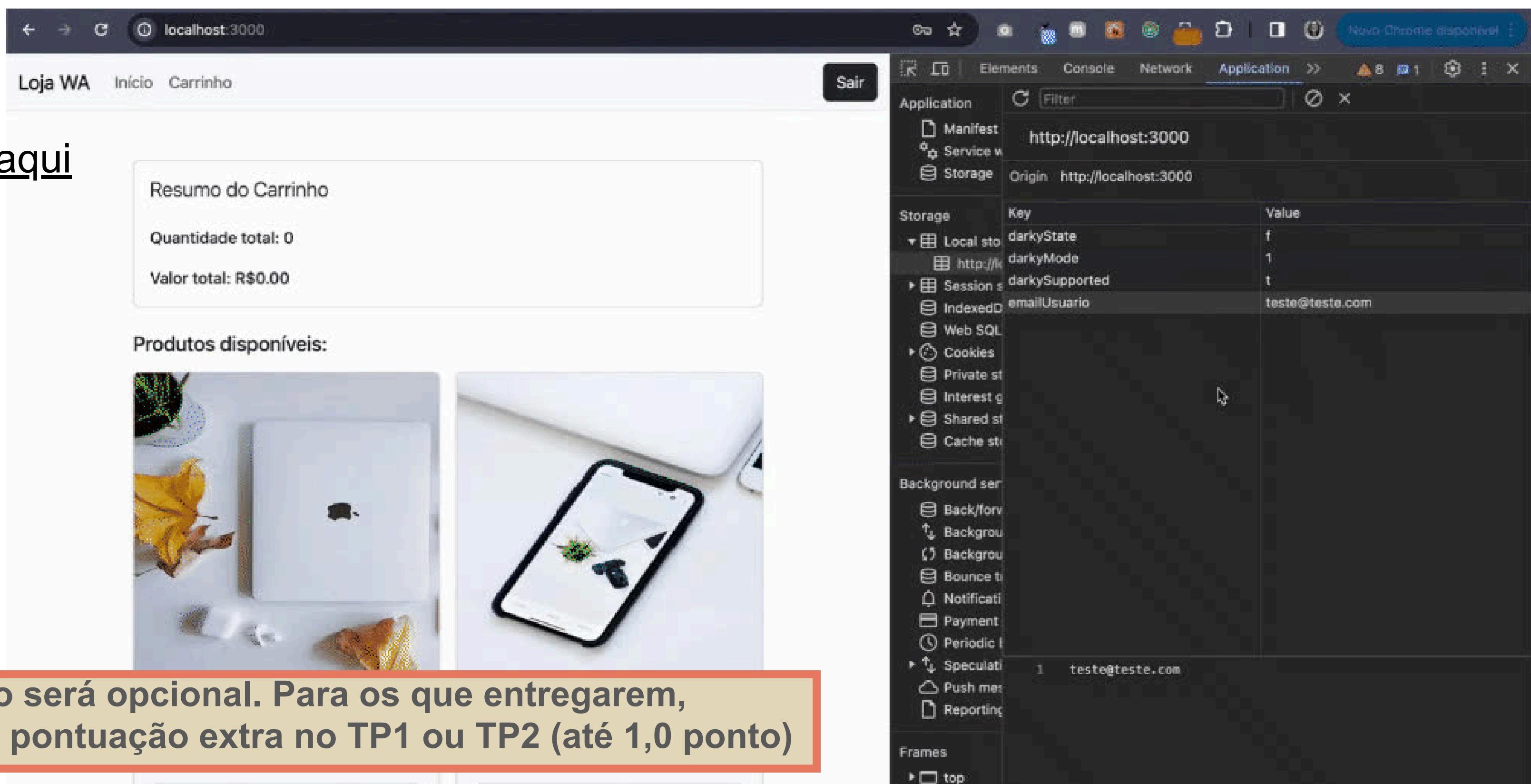
1. Remova os estados locais existentes, assim como os onChanges dos inputs, se houverem;
2. Inicie um novo formulário com o hook `useForm` do react hook form;
 - a. Não esqueça de incluir um tipo para o `useForm`, que deverá ser um objeto contendo todos os inputs do formulário e suas respectivas tipagens.
3. Por meio do `useForm`, extraia as funções '`register`', '`handleSubmit`' e o objeto de errors do objeto '`formState`';
4. Para o campo de **email**, utilize o `register` para adicionar o campo no formulário e também adicionar uma validação de "**required**";
5. Para o campo de **senha**, utilize o `register` para adicionar o campo no formulário e também adicionar duas validações: "**required**" e "**minLength**", que deve ser **6**.
 - a. Para cada erro, implemente uma mensagem personalizada e específica, que pode ser renderizada em um **span** logo abaixo do input. As mensagens devem aparecer baseado na verificação do objeto '`errors`';
6. Por fim, utilize o `handleSubmit` no `onSubmit` do form, passando a mesma função utilizada em atividades anteriores, responsável pelo `setEmailUsuario`;
 - a. A função deve ser ajustada para procurar o valor de email em '`data`' agora, e não no estado local.

React Hook Form: Vamos praticar?



Resultado esperado:

[veja o gif aqui](#)



The screenshot shows a web application for a store named "Loja WA". The main content area displays a "Resumo do Carrinho" (Cart Summary) with "Quantidade total: 0" and "Valor total: R\$0.00". Below this, there is a section titled "Produtos disponíveis:" (Available Products) showing two images: one of a yellow flower and one of a smartphone displaying a game.

The browser's developer tools are open, specifically the Application tab under the Network panel. It shows the storage data for the session. The "Storage" table lists the following items:

Storage	Key	Value
LocalStorage	darkyState	f
LocalStorage	darkyMode	1
SessionStorage	darkySupported	t
IndexedDB	emailUsuario	teste@teste.com
Web SQL		
Cookies		
Private storage		
Interest groups		
Shared storage		
Cache storage		

- Esse exercício será opcional. Para os que entregarem, contará como pontuação extra no TP1 ou TP2 (até 1,0 ponto)

Boas práticas gerais

- **Separe Componentes:**
 - Divida seu aplicativo em componentes reutilizáveis e coesos, seguindo o princípio de responsabilidade única. Isso facilita a manutenção e promove a legibilidade do código.
- **Evite Lógica no JSX:**
 - Mantenha o JSX limpo e livre de lógica complexa. Extraia lógica para funções ou hooks personalizados para promover a reutilização e a manutenção.
- **Utilize Tipagem Forte:**
 - Aproveite o poder do TypeScript para fornecer tipagem estática aos componentes, props e estados, garantindo maior segurança e facilidade na manutenção do código.
- **Padronização de Código:**
 - Mantenha um estilo de código consistente em todo o projeto, seguindo as diretrizes de codificação e as convenções de nomenclatura estabelecidas pela comunidade ou pela equipe de desenvolvimento.
- **Mantenha-se Atualizado:**
 - Acompanhe as atualizações e as melhores práticas da comunidade React e Next.js para incorporar constantemente melhorias e novos recursos em seu projeto.

Obrigada!

Dúvidas?

- Slack
- Email: jlslc@icomp.ufam.edu.br



União

