

FIAP

FIAP

SLIDER ▢■◀

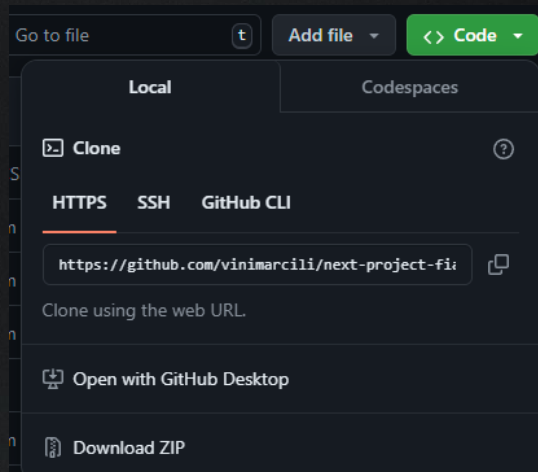


FormData / HooksForm

Preparativos

A partir da aula de hoje vamos construir uma aplicação juntos. Vamos criar um formulário de login, cadastro e exibir informações em uma área autenticada. Vamos configurar o nosso ambiente

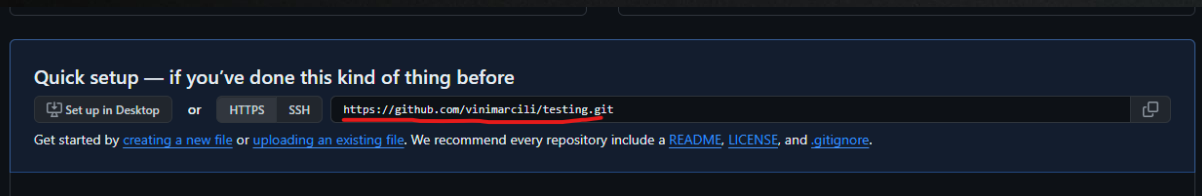
1. Abrir a URL do repositório com a base do projeto (<https://github.com/vinimarcili/react-project-fiap>)
2. Copiar a URL para clonar o repositório
3. Digitar o comando `git clone URL-COPIADA`
4. Criar um novo repositório na sua conta para o projeto
5. Abrir o projeto clonado no VSCODE



Preparativos

A partir da aula de hoje vamos construir uma aplicação juntos. Vamos criar um formulário de login, cadastro e exibir informações em uma área autenticada. Vamos configurar o nosso ambiente

6. Copiar a URL do seu repositório



7. Abrir o terminal integrado e digitar o comando `git remote set-url origin NOVA-URL-COPIADA`

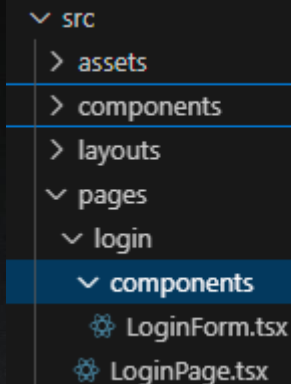
8. Digitar o comando `npm install` e testar a aplicação com o `npm run dev`

9. Digitar o comando `git push origin master` para enviar o projeto para o seu GitHub

Criando o Login

Vamos montar a página de login

1. Vamos criar um component LoginForm.tsx
2. Dentro do nosso component vamos criar um component de client com email e senha para login
3. Vamos estilizar com algumas classes do Tailwind



```
src
├── assets
├── components
├── layouts
├── pages
├── login
│   └── components
│       ├── LoginForm.tsx
│       └── LoginPage.tsx
```

```
1  import Button from "@/components/Button"
2  import Input from "@/components/Input"
3
4
5  const LoginForm = () => {
6    return (
7      <form
8        className="w-full flex flex-col gap-2"
9        noValidate
10      >
11        <Input
12          label='E-mail'
13          type='email'
14          name='email'
15          id='email'
16          placeholder='E-mail'
17          required
18        />
19        <Input
20          label='Senha'
21          type='password'
22          name='password'
23          id='password'
24          placeholder='Senha'
25          minLength={6}
26          required
27        />
28        <Button type='submit'>
29          Entrar
30        </Button>
31      </form>
32    )
33  }
34
35  export default LoginForm
```


Criando o Login

Vamos montar a página de login

5. Agora vamos chamar na nossa LoginPage

6. Estilizar com algumas classes do Tailwind e colocar um maxWidth para o nosso form

```
import LoginForm from "../components/LoginForm"

const LoginPage = () => {
  return <div className='p-3 bg-white rounded shadow mx-auto w-full' style={{ maxWidth: '300px' }}>
    <h2 className="text-center mb-2">
      Login
    </h2>
    <LoginForm />
  </div>
}

export default LoginPage
```

Criando o Login

Resultado

Bem-vindo ao FIAP Next.js

Login

E-mail

Please fill out this field.

Senha

Entrar

Hook de Forms

Como já vimos nas aulas de React, podemos criar qualquer custom hook para nós auxiliar. Vamos criar um custom Hook de React para nós ajudar a lidar com formulários.

1. Vamos criar uma pasta hooks, e dentro uma pasta use-form e o seu respectivo tsx dentro.
2. Vamos definir o que nosso Hook vai ter de entrada, vamos também definir as interfaces
 - O elemento HTML do formulário, para validarmos globalmente no Hook
 - O estado inicial, ou seja, as variáveis que nosso formulário precisa enviar (no nosso caso, email e senha)
 - Uma função de callback para quando o formulário for submetido com sucesso
 - Uma função de callback para quando acontecer algum erro de validação do formulário ou de execução (opcional)
 - Uma função que trata erros customizados não padronizados pelo HTML (como uma regra específica de negócio, como uma data de fim menor que uma de início, por exemplo) e devolve um objeto com os erros em forma de Objeto (opcional)

Hook de Forms

```
3 // Define a interface para o estado do formulário, permitindo qualquer chave com valores de qualquer tipo
4 ...
5 export interface FormState {
6   [key: string]: any
7 }
8 // Define a interface para o estado dos erros do formulário, onde cada chave é um campo e o valor é uma mensagem de erro
9 ...
10 interface ErrorsState {
11   [key: string]: string
12 }
13 // Define o tipo de função que configura erros personalizados, recebendo um evento de formulário e retornando um objeto ErrorsState
14 type SetCustomErrorsFunction = (target: HTMLFormElement) => ErrorsState
15
16 // Define o tipo de função de callback para submissão, que recebe os valores do formulário e opcionalmente o evento do formulário, retornando uma Promise
17 type SubmitCallbackFunction = (values: FormState, target?: FormEvent<HTMLFormElement>) => Promise<void>
18
19 // Hook personalizado useForm para gerenciar formulários
20 const useForm = (
21   formRef: RefObject<HTMLFormElement>, // Elemento do formulário
22   initialState: FormState, // Estado inicial do formulário
23   submitCallback: SubmitCallbackFunction, // Função de callback executada na submissão do formulário
24   errorCallback?: (error: Error) => Promise<void>, // Função opcional para lidar com erros
25   setCustomErrors?: SetCustomErrorsFunction, // Função opcional para definir erros personalizados
26 ) => {
27
28 }
29
30 export default useForm // Exporta o hook personalizado useForm
31
32
```

Hook de Forms

3. Agora vamos definir alguns estados:

- Vamos precisar de um estado para controlar os nossos dados
- Um estado para armazenarmos os erros de formulário
- Um estado para armazenar a contagem de erros (numérico)
- Um estado para controlar o loading do nosso formulário, se ele já foi submetido ou não
- Também vamos criar uma variável simples para o form, para pegar o **current** da nossa referência

```
3 // Hook personalizado useForm para gerenciar formulários
3 const useForm = (
1 |   formRef: RefObject<HTMLFormElement>, // Elemento do formulário
2   initialState: FormState, // Estado inicial do formulário
3   submitCallback: SubmitCallbackFunction, // Função de callback executada na submissão do formulário
4   errorCallback?: (error: Error) => Promise<void>, // Função opcional para lidar com erros
5   setCustomErrors?: SetCustomErrorsFunction, // Função opcional para definir erros personalizados
5 ) => {
7   const [loading, setLoading] = useState(false) // Estado de loading do formulário, inicializado como false
3   const [data, setData] = useState<FormState>(initialState) // Estado dos dados do formulário, inicializado com o estado inicial
3   const [errors, setErrors] = useState<ErrorsState>({}) // Estado dos erros do formulário, inicializado como um objeto vazio
3   const [errorsCount, setErrorsCount] = useState(0) // Contador de erros do formulário, inicializado como 0
1 |   const form = formRef.current // Obtém o elemento do formulário a partir da referência
2 }
```

Hook de Forms

4. Vamos pensar no que precisamos retornar para quem for usar o nosso Hook, vamos aproveitar para criar as duas funções que precisamos retornar, a handleSubmit e a handleChange

- Os dados do formulário que vão ser atualizados por um metodo interno do Hook quando o usuário fizer alguma alteração no campo (**data**)
- O objeto de erro (**errors**)
- A contagem dos erros (**errorsCount**)
- O **loading**
- A função de Submit que deve ser fornecida ao Formulário na hora de enviar a requisição (**handleSubmit**)
- A função que faz a alteração do estado dentro do Hook, que deve ser usada pelos inputs (**handleChange**)

Hook de Forms

```
19 // Hook personalizado useForm para gerenciar formulários
20 const useForm = (
21   formRef: RefObject<HTMLFormElement>, // Elemento do formulário
22   initialState: FormState, // Estado inicial do formulário
23   submitCallback?: SubmitCallbackFunction, // Função de callback executada na submissão do formulário
24   errorCallback?: (error: Error) => Promise<void>, // Função opcional para lidar com erros
25   setCustomErrors?: SetCustomErrorsFunction, // Função opcional para definir erros personalizados
26 ) => {
27   const [loading, setLoading] = useState(false) // Estado de loading do formulário, inicializado como false
28   const [data, setData] = useState<FormState>(initialState) // Estado dos dados do formulário, inicializado com o estado inicial
29   const [errors, setErrors] = useState<ErrorsState>({}) // Estado dos erros do formulário, inicializado como um objeto vazio
30   const [errorsCount, setErrorsCount] = useState(0) // Contador de erros do formulário, inicializado como 0
31   const form = formRef.current // Obtém o elemento do formulário a partir da referência
32
33   // Função para lidar com a submissão do formulário
34   const handleSubmit = useCallback(async (e: FormEvent<HTMLFormElement>) => {
35     e.preventDefault() // Previne o comportamento padrão de submissão do formulário
36
37   }, [])
38
39   // Função para lidar com mudanças nos campos do formulário
40   const handleChange = useCallback(async (e: ChangeEvent<HTMLInputElement>) => {
41
42   }, [])
43
44   // Retorna os estados e funções do hook
45   return {
46     data, // Dados do formulário
47     errors, // Erros do formulário
48     errorsCount, // Contador de erros
49     loadingSubmit: loading, // Estado de loading do formulário
50     handleChange, // Função para lidar com mudanças nos campos
51     handleSubmit // Função para lidar com a submissão do formulário
52   }
53 }
54
55 export default useForm // Exporta o hook personalizado useForm
```

Hook de Forms

5. Na nossa função de submit vamos precisar fazer varias coisas:

- Validar e alterar o estado do loading
- Validar os erros padrão
- Chamar a função de validação de erro externo (setCustomErrors)
- Em caso de erro chamar o callback de erro (errorCallback)
- Em caso de sucesso, chamar o callback de sucesso (submitCallback)

Hook de Forms

6. Fazendo controle do loading

```
// Função para lidar com a submissão do formulário
const handleSubmit = useCallback(async (e: FormEvent<HTMLFormElement>) => {
  e.preventDefault() // Previne o comportamento padrão de submissão do formulário

  if (loading) { // Se já estiver carregando, não faz nada
    return
  }

  setLoading(true) // Define o estado de loading como true

  setLoading(false) // Define o estado de loading como false após a submissão
}, [loading])
```

Hook de Forms

7. Vamos criar uma função para fazer a validação padrão dos erros para que o código da `handleSubmit` não fique muito grande. A função vai ser chamar `validateDefault`

```
// Função para validar os campos do formulário
const validateDefault = useCallback(() => {
  // Se o formulário não existir, retorna um objeto vazio
  if (form === null) {
    return {}
  }
  const formData = new FormData(form) // Cria um objeto FormData com os dados do formulário
  const isValid = form.checkValidity() // Verifica se o formulário é válido

  const newErrors: ErrorsState = {} // Objeto para armazenar novos erros
  if (!isValid) {
    for (const [name] of formData) { // Itera sobre os campos do formulário
      const element = form.elements.namedItem(name) // Obtém o elemento do campo pelo nome
      if (element instanceof HTMLInputElement) {
        newErrors[name] = element.validationMessage // Armazena a mensagem de validação do campo no objeto de erros
      }
    }
  }

  return newErrors // Retorna os novos erros
}, [form])
```

Hook de Forms

8. Vamos criar uma função que conta os erros do nosso Hook e atualiza o estado

```
const countErrors = (errorsObject: ErrorsState) => {  
  const count = Object.keys(errorsObject).length // Conta o número de erros  
  setErrorsCount(count) // Atualiza o contador de erros  
  return count // Retorna o número de erros  
}
```

Hook de Forms

9. Juntar tudo em uma função única para lidar com os erros além dos padrão, os custom erros também, vamos chamar de **handleErrors**

- Vamos pegar os erros do validateDefault
- Pegar os erros do setCustomErrors
- Juntar ambos em um único objeto
- Contar a quantidade de total de erros e atualizar o estado
- Então vamos retornar os erros e a contagem

Hook de Forms

```
const handleErros = useCallback(async () => {
  if (form === null) { // Se o formulário não existir, não faz nada
    return { validationErrors: {}, count: 0 }
  }
  const newErrors = validateDefault() // Valida os campos do formulário e obtém os erros
  const customErrors = setCustomErrors?.(form) // Obtém erros personalizados, se a função for fornecida
  const validationErrors = {
    ...newErrors, // Erros padrão
    ...customErrors // Erros personalizados
  }
  setErrors(validationErrors) // Atualiza o estado dos erros com os novos erros
  const count = countErrors(validationErrors) // Conta o número de erros

  // Retorna os erros e o contador de erros
  return {
    validationErrors,
    count
  }
}, [validateDefault, setCustomErrors, form])
```


Hook de Forms

10. Agora vamos fazer a lógica de tratamento de erros.

- Vamos pegar os erros do `validateDefault`
- Pegar os erros do `setCustomErrors`
- Juntar ambos em um único objeto
- Contar a quantidade de total de erros e atualizar o estado
- Validar se tem erro e se tiver, chamar o callback de erro
- Vamos jogar tudo isso em uma função chamada **`handleErrors`**
- Então vamos retornar os erros e a contagem

```
const handleErrors = useCallback(async (e: FormEvent<HTMLFormElement>) => {  
  const newErrors = validateDefault(e) // Valida os campos do formulário  
  const customErrors = setCustomErrors?.(e) // Obtém erros personalizados, se a função for fornecida  
  const validationErrors = {  
    ...newErrors, // Erros padrão  
    ...customErrors // Erros personalizados  
  }  
  const countErrors = Object.keys(validationErrors).length // Conta o número de erros  
  
  setErrorsCount(countErrors) // Atualiza o contador de erros  
  
  if (countErrors) { // Se houver erros  
    if (errorCallback instanceof Function) {  
      await errorCallback(new Error('Invalid Form'), {  
        cause: {  
          ...validationErrors // Passa os erros para a função onError, se fornecida  
        }  
      })  
    }  
  }  
  
  // Retorna os erros e o contador de erros  
  return {  
    error: validationErrors,  
    count: countErrors  
  }  
}, [validateDefault, errorCallback, setCustomErrors])
```

Hook de Forms

11. Vamos chamar o `handleErrors` dentro do `handleSubmit`, pegar a contagem e se for maior que zero, executar o nosso `errorCallback`

12. Agora por fim, vamos chamar a função de sucesso `handleSubmit`. Feito isso nosso hook está pronto para o uso

```
// Função para lidar com a submissão do formulário
const handleSubmit = useCallback(async (e: FormEvent<HTMLFormElement>) => {
  e.preventDefault() // Previne o comportamento padrão de submissão do formulário

  if (loading) { // Se já estiver carregando, não faz nada
    return
  }

  setLoading(true) // Define o estado de loading como true

  const { count, validationErrors } = await handleErrors() // Valida os campos do formulário e obtém os erros
  if (count) {
    setLoading(false)
    if (errorCallback instanceof Function) {
      await errorCallback(new Error('Invalid Form', {
        cause: {
          ...validationErrors // Passa os erros para a função onError, se fornecida
        }
      })))
    }
    return // Se houver erros, não faz nada
  }

  // Chama a função de callback de submissão passando os dados do formulário
  await submitCallback(data, e)

  setLoading(false) // Define o estado de loading como false após a submissão
}, [loading, handleErrors, submitCallback, data, errorCallback])
```

Hook de Forms

13. Agora, atualizaremos o handleChange para verificar os erros sempre que mudar algo no formulário e para atualizar os nossos valores com o setData

```
// Função para lidar com mudanças nos campos do formulário
const handleChange = useCallback(async (e: ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target // Extrai o nome e o valor do campo alterado
  setData((oldData) => {
    return {
      ...oldData, // Mantém os dados antigos
      [name]: value // Atualiza o campo alterado
    }
  })

  await handleErros() // Valida os campos do formulário e obtém os erros
}, [handleErros])
```

Hook de Forms

14. Vamos usar o `useEffect` para atualizar os erros toda vez que nosso elemento HTML mudar, por qualquer motivo que seja

```
// Efeito para lidar com mudanças no Elemento HTML do formulário
useEffect(() => {
  handleErros()
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [form]) // Executa o efeito quando o formulário mudar, o comentário acima desabilita o aviso de dependências faltantes
```

Usando o Hook

Vamos voltar ao nosso formulário e usar o que acabamos de criar

1. Vamos criar uma função para lidar com os erros `submitErrorCallback`. Por enquanto só vamos logar os erros no console
2. Vamos criar uma função para lidar com o sucesso `submitCallback`. Vamos logar os dados no console e criar uma “fake Promise” para demorar 5 segundos, apenas para conseguir testar o nosso loading

```
async function submitErrorCallback(error: Error) {  
  // TODO: Tratar erros  
  console.log(error.cause)  
}  
  
async function submitCallback(values: FormState) {  
  console.log(values)  
  
  // TODO: Envie os dados do formulário para a API  
  
  // DO fake request to take 5s  
  await new Promise((resolve) => setTimeout(resolve, 5000))  
}
```


Usando o Hook

3. Agora sim vamos invocar o nosso Hook e passar os parametros obrigatórios, vamos criar o nosso objeto de estado inicial com email e password, e pegar os valores que o hook retorna.

```
const LoginForm = () => {  
  const formRef = useRef<HTMLFormElement>(null)  
  const initialLoginForm = {  
    email: '',  
    password: ''  
  }  
  const {  
    data: {  
      email,  
      password  
    },  
    loadingSubmit, // You, 4 days ago • feat:  
    handleChange,  
    handleSubmit,  
    errorsCount  
  } = useForm(  
    formRef,  
    initialLoginForm,  
    submitCallback,  
    submitErrorCallback  
  )  
}
```

Usando o Hook

4. Vamos editar nosso HTML no JSX colocando as funções corretas, além disso vamos colocar uma mensagem de “Carregando” no botão quando o loading for verdadeiro

```
return (  
  <form className="w-full flex flex-col gap-2">  
    <Input  
      label='E-mail'  
      type='email'  
      name='email'  
      id='email'  
      placeholder='E-mail'  
      required  
    />  
    <Input  
      label='Senha'  
      type='password'  
      name='password'  
      id='password'  
      placeholder='Senha'  
      required  
    />  
    <Button type='submit'>  
      Entrar  
    </Button>  
  </form>  
)
```

```
return (  
  <Form  
    className="w-full flex flex-col gap-2"  
    onSubmit={handleSubmit}  
    ref={formRef}  
    noValidate  
  >  
    <Input  
      label='E-mail'  
      type='email'  
      name='email'  
      id='email'  
      placeholder='E-mail'  
      value={email}  
      handleChange={(_, e) => handleChange(e)}  
      readOnly={loadingSubmit}  
      required  
    />  
    <Input  
      label='Senha'  
      type='password'  
      name='password'  
      id='password'  
      placeholder='Senha'  
      minLength={6}  
      value={password}  
      handleChange={(_, e) => handleChange(e)}  
      readOnly={loadingSubmit}  
      required  
    />  
    <Button type='submit' disabled={loadingSubmit || !!errorsCount || !formRef.current}>  
      {  
        loadingSubmit  
        ? 'Carregando...'  
        : 'Entrar'  
      }  
    </Button>  
  </Form>  
)
```

Usando o Hook

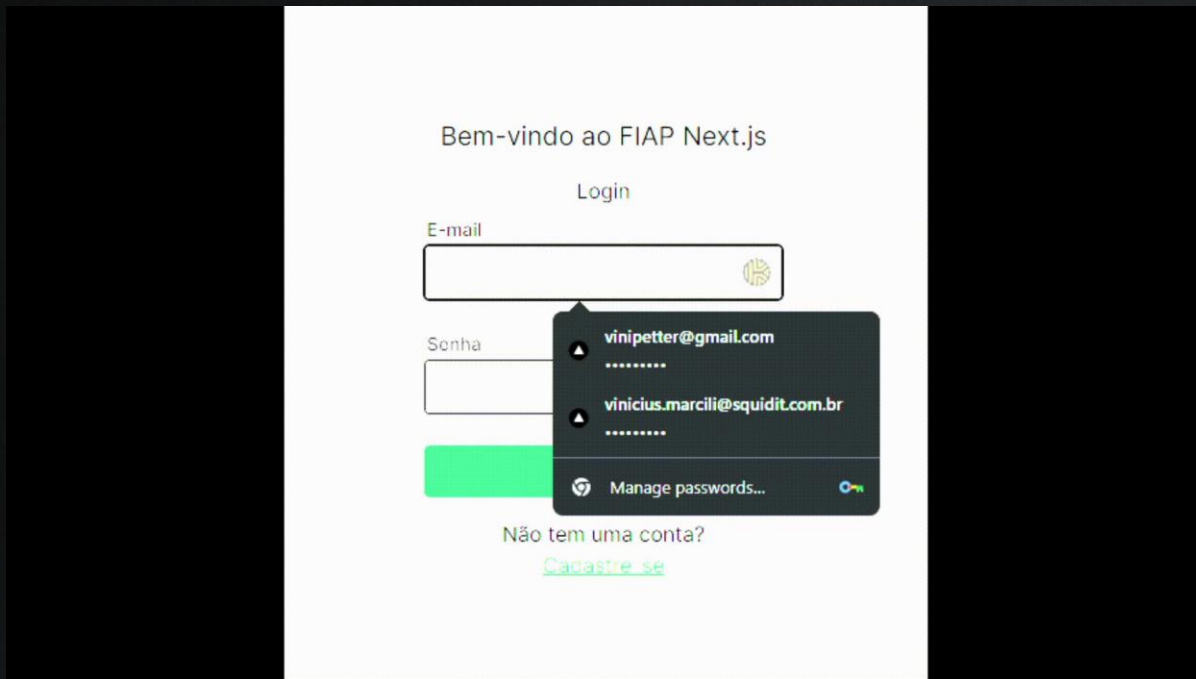
5. Pra finalizar vamos criar um footer para levar o nosso usuário para a página de cadastro

```
const LoginPage = () => {  
  return <div className='p-3 bg-white rounded shadow mx-auto w-full' style={{ maxWidth: '300px' }}>  
    <h2 className="text-center mb-2">  
      Login  
    </h2>  
    <LoginForm />  
    <footer className="text-center mt-4">  
      Não tem uma conta?<br />  
      <Link className="text-green-500 underline" href="/login/register">Cadastre-se</Link>  
    </footer>  
  </div>  
}
```

Usando o Hook

Resultado

Código da aula: <https://github.com/vinimarcili/react-project-fiap/tree/step-1>



Exercício

Agora complete a `registerPage`, faça um formulário de cadastro usando o hook que acabamos de criar. Também coloque um botão para o usuário voltar para a tela de login caso desista de se cadastrar.

Faça um cadastro com os campos:

- Nome
- E-mail
- CPF
- Data de nascimento
- Senha
- Confirmar senha

Faça validações customizadas (lembre-se de usar a função de erro customizado do hook)

- Se a senha e confirmar senha são iguais
- Se o usuário é maior de 18 anos

Dúvidas, críticas ou sugestões?

FIAP