React Hook Form

O que é

O react-hook-form é uma biblioteca popular para gerenciar formulários em aplicações React. Ela é conhecida por ser leve, rápida e fácil de usar. Existe mais de uma forma de implementar ela, vamos implementar para usar um funcionamento semelhante ao hook que criamos na aula anterior.

Além disso, vamos usar a biblioteca de validação yup para agilizar a validação dos nossos campos.

Preparativos

1. Vamos instalar as bibliotecas npm install react-hook-form yup @hookform/resolvers
2. Vamos criar um outro Input baseado no anterior, removendo algumas coisas para ele funcionar corretamente com o hook

Preparativos

```tsx
...
interface InputProps extends InputHTMLAttributes<HTMLInputElement> {
  handleChange?: (value: string, e: ChangeEvent<HTMLInputElement>) => void // Função opcional para lidar com mudanças no input
  label?: ReactNode // Conteúdo opcional para rótulo do input
  customError?: string | null // Mensagem de erro personalizada
}
const Input = ({ handleChange, disabled, readOnly, className = '', label = '', customError = '', ...props }: InputProps) => {
  const [error, setError] = useState<string | null>(null) // Estado para controlar mensagens de erro

  // Callback para lidar com mudanças no input
  const onHandleChange = useCallback((e: ChangeEvent<HTMLInputElement>) => {
    const { validity, validationMessage, value } = e.target
    setError(!validity.valid ? validationMessage : null)
    if (handleChange instanceof Function) {
      handleChange(value, e)
    }
  }, [handleChange]) // Dependência do callback de mudança

  // Variaveis auxiliares para controle de estado
  const errorMessage = useMemo(() => customError || error, [customError, error])
  const hasActionsState = useMemo(() => disabled || readOnly, [disabled, readOnly])
  const hasControllState = useMemo(() => hasActionsState || errorMessage, [hasActionsState, errorMessage])
  const canShowError = useMemo(() => errorMessage && !hasActionsState, [errorMessage, hasActionsState])

  return (
    <div className='w-full'>
      {label && (
        <label
          htmlFor={props.id ?? props.name ?? ''}
          className={`block font-medium mb-0.5 px-0.5 text-left text-neutral-900 text-sm`}
        >
          {label}
        </label>
      )}
      <input
        {...props} // Passa todas as outras propriedades para o input HTML
        onChange={onHandleChange} // Atribui o callback de mudança ao evento onChange
        className={`
          w-full block p-2 border rounded
          focus:outline-none focus:ring-1 ring-current border-neutral-900
          ${!hasControllState ? 'bg-white border-neutral-900' : ''}
          ${disabled ? 'bg-gray-300 border-gray-300 cursor-not-allowed' : ''}
          ${readOnly ? 'bg-gray-100 border-gray-100' : ''}
          ${canShowError ? 'border-red-500 border-2 ring-red-500' : ''}
          ${className}
        `} // Classes CSS condicionais baseadas em propriedades
        disabled={disabled} // Define se o input está desabilitado ou não
        readOnly={readOnly} // Define se o input é somente leitura ou não
      />
      {/* Exibe a mensagem de erro se houver */}
      <span className={`
        min-h-4 text-red-500 text-xs px-0.5 pt-0.5 block leading-none
        ${canShowError ? 'opacity-100 ' : 'opacity-0'}
      `}
      >
        {errorMessage}
      </span>
    </div>
  )
}
export default Input // Exporta o componente Input como padrão
```

```tsx
interface InputProps extends InputHTMLAttributes<HTMLInputElement> {
  label?: ReactNode
  customError?: string | null
}

const Input = forwardRef<HTMLInputElement, InputProps>(({ disabled, readOnly, className = "", label = "", customError = "", ...props }, ref) => {
  const errorMessage = useMemo(() => customError, [customError])
  const hasActionsState = useMemo(() => disabled || readOnly, [disabled, readOnly])
  const hasControllState = useMemo(() => hasActionsState || errorMessage, [hasActionsState, errorMessage])
  const canShowError = useMemo(() => errorMessage && !hasActionsState, [errorMessage, hasActionsState])

  return (
    <div className="w-full">
      {label && (
        <label
          htmlFor={props.id ?? props.name ?? ""}
          className="block font-medium mb-0.5 px-0.5 text-left text-neutral-900 text-sm"
        >
          {label}
        </label>
      )}
      <input
        className={`
            w-full block p-2 border rounded
            focus:outline-none focus:ring-1 ring-current border-neutral-900
            ${!hasControllState ? "bg-white border-neutral-900" : ""}
            ${disabled ? "bg-gray-300 border-gray-300 cursor-not-allowed" : ""}
            ${readOnly ? "bg-gray-100 border-gray-100" : ""}
            ${canShowError ? "border-red-500 border-2 ring-red-500" : ""}
            ${className}
          `}
        disabled={disabled}
        readOnly={readOnly}
        ref={ref}
        {...props}
      />
      <span
        className={`
            min-h-4 text-red-500 text-xs px-0.5 pt-0.5 block leading-none
            ${canShowError ? "opacity-100 " : "opacity-0"}
          `}
      >
        {errorMessage}
      </span>
    </div>
  )
}
)
Input.displayName = "Input"
export default Input
```

Usando

1. Agora vamos usar o hook, primeiro vamos definir o nosso schema da lib yup dentro do nosso login-form.tsx
2. Também vamos criar a interface

```
type LoginFormData = {
  email: string
  password: string
}

const schema = yup
  .object({
    email: yup.string().email('E-mail inválido').required('O e-mail é obrigatório'),
    password: yup.string().min(6, 'A senha deve ter no mínimo 6 caracteres').required('A senha é obrigatória'),
  })
  .required()
```

Usando

3. Vamos criar um estado para controle do loading
4. Vamos instanciar o nosso Hook

```
const LoginForm = () => {
  const formRef = useRef<HTMLFormElement>(null)
  const [loading, setLoading] = useState(false)
  const { control, handleSubmit, formState } = useForm<LoginFormData>({
    defaultValues: {
      email: '',
      password: ''
    },
    resolver: yupResolver(schema),
    mode: "onChange"
  })
  const { errors, isValid } = useMemo(() => formState, [formState])
```

Usando

No uso do useForm:

- Passamos os valores padrão
- O resolver é o que vai validar o nosso form. Usamos o validador do yup
- Alteramos o mode para onChange, assim a validação acontece sempre que algum valor mudar

Ele vai devolver alguns itens, e o que estamos usando:

- control: Variavel de controle, deve ser fornecida aos campos no HTML
- handleSubmit: vai ser mandado para o form para fazer o submit
- formState: Estado do formulário, vamos usar o isValid (valida tudo que passamos de uma vez) e os erros (assim conseguimos exibis as mensagens que passamos para o yup)

Usando

5. Agora vamos editar nossas funções submitErrorCallback e submitCallback para funcionar na nossa nova estrutura. Nós vamos tratar o erro dentro do submit comum agora.
Vamos

```
async function submitErrorCallback() {
  // TODO: Tratar erros
}

async function submitCallback(values: LoginFormData) {
  setLoading(true)

  // Verifica se o formulário é válido
  // TODO: Outros erros?
  if (!isValid) {
    await submitErrorCallback()
    setLoading(false)       You, 23 minutes ago • refactor: Upd
    return
  }

  // TODO: Envie os dados do formulário para a API
  console.log(values)

  // DO fake request to take 5s
  await new Promise((resolve) => setTimeout(resolve, 5000))
  setLoading(false)
}
```

Usando

6. Agora vamos mudar o nosso HTML. Para usar o Hook precisamos usar um componente especial da própria lib, o Controller. Ele indica para o form que estamos usando um Input custom e funcionar como um Wrapper do componente.
7. Além disso vamos alterar o onSubmit para receber o metodo que pegamos do hook passando o nosso metodo como parâmetro
8. Por ultimo vamos alterar as configurações do botão, para usar o loading novo e o isValid

```
return (
  <form
    className="w-full flex flex-col gap-2"
    onSubmit={handleSubmit(submitCallback)}
    ref={formRef}
    noValidate
  >
    <Controller
      name="email"
      control={control}
      render={({ field }) => (
        <Input
          label='E-mail'
          type='email'
          id='email'
          placeholder='E-mail'
          readOnly={loading}
          customError={errors?.email?.message}
          {...field}
        />
      )}
    />
    <Controller
      name="password"
      control={control}
      render={({ field }) => (
        <Input
          label='Senha'
          type='password'
          id='password'
          placeholder='Senha'
          minLength={6}
          readOnly={loading}
          customError={errors?.password?.message}
          {...field}
        />
      )}
    />
    <Button type='submit' disabled={!formRef.current || loading || !isValid}>
      {
        loading
          ? 'Carregando...'
          : 'Entrar'
      }
    </Button>
  </form>
)
```

Dessa forma devemos ter o mesmo resultado do que fizemos na aula passada, mas com uma implementação diferente definindo as mensagens de erros customizadas

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

Dúvidas, críticas ou sugestões?


FIAP