[Radix-ui 1](#_Toc182724783)

[Dialog 1](#_Toc182724784)

[Radio Group 2](#_Toc182724785)

[Json Server 3](#_Toc182724786)

[Instalação do json-server 3](#_Toc182724787)

[Problema no comando -d “delay” 4](#_Toc182724788)

[Configurar um comando para rodar no json-server 4](#_Toc182724789)

[Requisitar arquivo da API (json server) 4](#_Toc182724790)

[Funcionamento do fetch API 4](#_Toc182724791)

[Context 7](#_Toc182724792)

[Criando context 7](#_Toc182724793)

[Utilizar o context 8](#_Toc182724794)

[Formatando data e valor BR 9](#_Toc182724795)

[Hooks 10](#_Toc182724796)

[Criar um Hook 10](#_Toc182724797)

[Utilizar o hook 11](#_Toc182724798)

[React Hook Form 12](#_Toc182724799)

[Instalando react hook form 12](#_Toc182724800)

[Utilizando react hook form 12](#_Toc182724801)

[Simular um delay na função handleSubmit 13](#_Toc182724802)

[Fazer o botão aguardar o retorno da API para poder clicar novamente 13](#_Toc182724803)

[Utilizando campos controled no react hook form 14](#_Toc182724804)

[Iniciar com um campo do radio selecionado pelo controller 16](#_Toc182724805)

[Aprimorando requisições 16](#_Toc182724806)

[Cambo de Busca 16](#_Toc182724807)

[Configurando Axios 18](#_Toc182724808)

[Cadastrando novo item (Transação) 18](#_Toc182724809)

# Radix-ui

## Dialog

npm install @radix-ui/react-dialog

No componente que vai chamar o modal (dialog):

import \* as Dialog from '@radix-ui/react-dialog'

Envolva o botão que vai chamar o modal e onde ele ser colocado com **<Dialog.Root>:**

<Dialog.Root>

<NewTransactionButton>Nova transação</NewTransactionButton>

</Dialog.Root>

Envolva o botão que vai chamar o modal com **<Dialog.Trigger>:**

**<Dialog.Trigger>** é o botão da biblioteca especifico para chamar o modal, para não criar um botão dentro do outro passe a propriedade **asChild**, isso faz com que o **Trigger** não seja renderizado na tela e o botão comum funcione no lugar do **Trigger**.

<Dialog.Root>

<Dialog.Trigger asChild>

<NewTransactionButton>Nova transação</NewTransactionButton>

</Dialog.Trigger>

</Dialog.Root>

Abaixo do <Dialog.Trigger> coloque o **<Dialog.Portal>** que faz com que o dialog não faça parte de um componente específico, mas que ele seja um elemento a parte.

<Dialog.Root>

<Dialog.Trigger asChild>

<NewTransactionButton>Nova transação</NewTransactionButton>

</Dialog.Trigger>

<Dialog.Portal></Dialog.Portal>

</Dialog.Root>

Dentro do **<Dialog.Portal>** coloque um **<Dialog.Overlay>** que é o fundo preto um pouco opaco, e por fim coloque o **<Dialog.Content>:**

<Dialog.Root>

<Dialog.Trigger asChild>

            <NewTransactionButton>Nova transação</NewTransactionButton>

</Dialog.Trigger>

<Dialog.Portal>

<Dialog.Overlay />

<Dialog.Content></Dialog.Content>

</Dialog.Portal>

</Dialog.Root>

Dentro do modal podem também ser colocadas algumas tags especificas do dialog:

<Dialog.Content>

<Dialog.Title>Nova transação</Dialog.Title>

<Dialog.Close />

</Dialog.Content>

## Radio Group

Criar radio group  
npm install @radix-ui/react-radio-group

<RadioGroup.Root defaultValue="1" name="example">

<RadioGroup.Item value="1">Default</RadioGroup.Item>

<RadioGroup.Item value="2">Comfortable</RadioGroup.Item>

<RadioGroup.Item value="3">Compact</RadioGroup.Item>

</RadioGroup.Root>

# Json Server

## Instalação do json-server

Json server simula uma API no front-end

npm install json-server

Na raiz crie um arquivo chamado **server.json** em seguido digite no console:

npx json-server server.json

Caso de **erro**, pode ser que ele está direcionando o server para a mesma porta do front-end. Para configurar outra porta digite:

npx json-server server.json -p 3333

Para testar crie uma tabela no arquivo **server.json:**

{

    "transactions": [

        {

            "id": 1,

            "description": "Desenvolvimento de site",

            "type": "income",

            "category": "Venda",

            "price": 14000,

            "createdAt": "2024-11-10T12:03:35.747Z"

        }

    ]

}

Dica para adicionar a data:

No console.log do navegador digite **new Date().toISOString()**

Por padrão o json-server **não fica assistindo as alterações do server.json** para isso digite -w na frente do comando de inicialização do json-server, exemplo:

npx json-server server.json -w

ou

npx json-server server.json -p 3333 -w  
  
Para configurar um delay “neste caso 500 milissegundos” como se fosse uma requisição de uma API vindo de um servidor real faça:

npx json-server server.json -w -d 500

ou

npx json-server server.json -p 3333 -w -d 500

## Problema no comando -d “delay”

Caso **o delay não funcione**, pode ser que a versão que está sendo instalada atualmente não tenha a função -d pois ela foi removida a partir da versão @0.17.3, então desinstale o json-server:

npm uninstall json-server

E o instale novamente com o seguinte comando:

npm i [json-server@0.17.3](mailto:json-server@0.17.3) -D

## Configurar um comando para rodar no json-server

No arquivo package.json configure o seguinte comando:

"scripts": {

    "dev:server": "npx json-server server.json -w -d 500",

  },

# Requisitar arquivo da API (json server)

## Funcionamento do fetch API

Para requisitar arquivos sem uso de biblioteca utilizamos a API **fetch** do navegador passando a rota do servidor:

fetch('http://localhost:3000/transactions').then(response => {

        console.log(response)

    })

fetch('http://localhost:3000/transactions')

Como a requisição deve ser carregada apenas uma vez, utilizamos o **useEffect( )** sem as dependências.

    useEffect(() => {

        fetch('http://localhost:3000/transactions').then(response => {

            console.log(response)

        })

    }, [])

Para visualizarmos estes dados como json utilizamos um **response.json:**

    useEffect(() => {

        fetch('http://localhost:3000/transactions').then(response => {

            response.json().then(data => {

                console.log(data)

            })

        })

    }, [])

Para evitar uma promisse (.then) dentro de outra promisse, podemos aninha-los da seguinte forma:

useEffect(() => {

        fetch('http://localhost:3000/transactions')

            .then(response => {response.json()

            .then(data => {

                console.log(data)

            })

        })

    }, [])

O useEffect não aceita **async** ser uma função assíncrona diretamente, para transformar isso em uma função assíncrona:

async function loadTransactions() {

    const response = await fetch('http://localhost:3000/transactions')

    const data = await response.json()

    console.log(data)

}

useEffect(() => {

    loadTransactions()

}, [])

Para mostrar as informações em tela utilizamos o **useState( )** ecriamos uma interface para tipar os dados do estado.

interface Transaction {

    id: number;

    description: string;

    type: 'income' | 'outcome';

    price: number;

    category: string;

    createdAt: string;

}

export function Transactions() {

const [transactions, setTransactions] = useState<Transaction[]>([])

async function loadTransactions() {

    const response = await fetch('http://localhost:3000/transactions')

    const data = await response.json()

    setTransactions(data)

}

useEffect(() => {

    loadTransactions()

}, [])

Agora temos que utilizar um **map**, chamando o estado dentro do componente:

<tbody>

   {transactions.map(transaction => {

       return (

          <tr key={transaction.id}>

              <td width="50%">{transaction.description}</td>

               <td>

                    <PriceHighLight variant={transaction.type}>

                         {transaction.price}

                    </PriceHighLight>

               </td>

               <td>{transaction.category}</td>

               <td>{transaction.createdAt}</td>

          </tr>

      )

   })}

</tbody>

# Context

## Criando context

Crie uma pasta chamada contexts dentro da pasta src, com o arquivo **context.tsx**. Exemplo: TransactionsContext.tsx

import { createContext, ReactNode, useEffect, useState } from "react";

interface Transaction {

    id: number;

    description: string;

    type: 'income' | 'outcome';

    price: number;

    category: string;

    createdAt: string;

}

interface TransactionContextType {

    transactions: Transaction[];

}

interface TransactionsProviderProps {

    children: ReactNode

}

export const TransactionsContext = createContext({} as TransactionContextType);

export function TransactionsProvider({ children }: TransactionsProviderProps) {

    const [transactions, setTransactions] = useState<Transaction[]>([])

    async function loadTransactions() {

        const response = await fetch('http://localhost:3000/transactions')

        const data = await response.json()

        setTransactions(data)

    }

    useEffect(() => {

        loadTransactions()

    }, [])

    return (

        <TransactionsContext.Provider value={{ transactions }}>

            {children}

        </TransactionsContext.Provider>

    )

}

## Utilizar o context

Utilize o **provider** para encapsular o componente pai de todos os componentes que utilizaram os atributos enviados pelo context.

import { TransactionsProvider } from './contexts/TransactionsContext'

export function App() {

  return (

    <ThemeProvider theme={defaultTheme}>

      <GlobalStyle />

      <TransactionsProvider>

        <Transactions />

      </TransactionsProvider>

    </ThemeProvider>

  )

}

No componente que utilizará as propriedades utilize o useContext

import { TransactionsContext } from "../../contexts/TransactionsContext";

export function Transactions() {

    const { transactions } = useContext(TransactionsContext)

return (

<TransactionsTable>

          <tbody>

              {transactions.map(transaction => {

                   return (

                       <tr key={transaction.id}>

                            <td width="50%">{transaction.description}</td>

                            <td>

                                <PriceHighLight variant={transaction.type}>

                                    {transaction.price}

                                </PriceHighLight>

                             </td>

                             <td>{transaction.category}</td>

                             <td>{transaction.createdAt}</td>

                         </tr>

                     )

                )}

            </tbody>

        </TransactionsTable>

    )

}

# Formatando data e valor BR

Crie uma pasta utils na pasta src e crie um arquivo chamado **formater.ts**

export const dateFormatter = new Intl.DateTimeFormat('pt-BR')

export const priceFormatter = new Intl.NumberFormat('pt-BR', {

    style: 'currency',

    currency: 'BRL',

})

<TransactionsTable>

    <tbody>

       {transactions.map(transaction => {

           return (

               <tr key={transaction.id}>

                    <td width="50%">{transaction.description}</td>

                    <td>

                        <PriceHighLight variant={transaction.type}>

                            {transaction.type === 'outcome' && '- '}

                            {priceFormatter.format(transaction.price)}

                        </PriceHighLight>

                    </td>

                    <td>{transaction.category}</td>

                    <td>{dateFormatter.format(new Date(transaction.createdAt))}</td>

                </tr>

            )

        })}

    </tbody>

</TransactionsTable>

# Hooks

## Criar um Hook

Crie uma pasta **hook** dentro da pasta src, em seguida crie o arquivo de hook.ts. Exemplo: useSummary.ts

import { useContext } from "react"

import { TransactionsContext } from "../contexts/TransactionsContext"

export function useSummary() {

    const { transactions } = useContext(TransactionsContext)

    const summary = transactions.reduce(

        (acc, transaction) => {

            if (transaction.type === 'income') {

                acc.income += transaction.price

                acc.total += transaction.price

            } else {

                acc.outcome += transaction.price

                acc.total -= transaction.price

            }

            return acc;

        },

        {

            income: 0,

            outcome: 0,

            total: 0

        }

    )

    return summary

}

## Utilizar o hook

import { useSummary } from "../../hooks/userSummary";

export function Summary() {

    const summary = useSummary()

    return(

        <SummaryContainer>

            <SummaryCard>

                <header>

                    <span>Entradas</span>

                    <ArrowCircleUp size={32} color="#00b37e"/>

                </header>

                <strong>{priceFormatter.format(summary.income)}</strong>

            </SummaryCard>

            <SummaryCard>

                <header>

                    <span>Saídas</span>

                    <ArrowCircleDown size={32} color="#f75a68"/>

                </header>

                <strong>{priceFormatter.format(summary.outcome)}</strong>

            </SummaryCard>

            <SummaryCard variant="green">

                <header>

                    <span>Total</span>

                    <CurrencyDollar size={32} color="#fff"/>

                </header>

                <strong>{priceFormatter.format(summary.total)}</strong>

            </SummaryCard>

        </SummaryContainer>

    )

}

# React Hook Form

## Instalando react hook form

npm i react-hook-form zod

npm i @hookform/resolvers

## Utilizando react hook form

Importe o **zodResolver** crie uma variavel para o schema e como tipagem aproveite o mesmo:

const newTransactionFormSchema = z.object({

    description: z.string(),

    price: z.number(),

    category: z.string(),

    //type: z.enum(['income', 'outcome']),

})

type NewTransactionFormInputs = z.infer<typeof newTransactionFormSchema>

Crie uma variável para o **useForm:**

const { register, handleSubmit } = } = useForm<NewTransactionFormInputs>({

     resolver: zodResolver(newTransactionFormSchema)

})

Atribua o register para os campos do formulário nomeando-os:

<input

    type="text"

    placeholder="Description"

    required

    {...register('description')}

/>

<input

    type="number"

    placeholder="Preço"

    required

    {...register('price', { valueAsNumber: true})}

/>

Crie uma função para o **handleSubmit:**

function handleCreateNewTransaction(data: NewTransactionFormInputs) {

    console.log(data)

}

Como este console é possível visualizar os dados dos campos do formulário no navegador após passar essa função para o form.

Agora passe o **handleSubit** ao form:

 <form onSubmit={handleSubmit(handleCreateNewTransaction)}>

## Simular um delay na função handleSubmit

async function handleCreateNewTransaction(data: NewTransactionFormInputs) {

    await new Promise(resolve => setTimeout(resolve, 2000))

    console.log(data)

}

## Fazer o botão aguardar o retorno da API para poder clicar novamente

Adicione o atributo **formState** com o parâmetro **isSubmitting** ao **useForm:**

const {

    register,

    handleSubmit,

    formState: { isSubmitting }

} = useForm<NewTransactionFormInputs>({

     resolver: zodResolver(newTransactionFormSchema)

})

Passe o **isSubmitting** na propriedade **disabled** do botão **submit:**

<button type="submit" disabled={isSubmitting}>

    Cadastrar

</button>

Feito isso podemos até mesmo **estilizar este botão quando estiver disabled:**

button[type="submit"] {

    height: 58px;

    border: 0;

    background: ${props => props.theme['green-500']};

    color: ${props => props.theme.white};

    font-weight: bold;

    padding: 0 1.25rem;

    border-radius: 6px;

    margin-top: 1.5rem;

    cursor: pointer;

    &:disabled {

    opacity: 0.6;

    cursor: not-allowed;

}

    &&:not(:disabled):hover {

        background: ${props => props.theme['green-700']};

        transition: background-color 0.2s;

    }

}

## Utilizando campos controled no react hook form

Adicione a importação **Controller**:

import { Controller, useForm } from 'react-hook-form';

Passe a propriedade **control** a variável **useForm**:

const {

    control,

    register,

    handleSubmit,

    formState: { isSubmitting }

} = useForm<NewTransactionFormInputs>({

    resolver: zodResolver(newTransactionFormSchema)

})

Utillize o componente <**Controler />** com a propriedade **control** passando o control do useForm, utilize a propriedade **name** com o campo criado no eschema com os valores. Neste caso é o type que ficou comentado:

const newTransactionFormSchema = z.object({

    description: z.string(),

    price: z.number(),

    category: z.string(),

    type: z.enum(['income', 'outcome']),

})

<Controller

    control={control}

    name='type'

/>

A ultima propriedade é o **render,** que é a função que retorna o html com os campos controled.

<Controller

    control={control}

    name='type'

    render={() => {

       return ()

    }}

/>

Adicionamos o html com os campos do formulario e podemos testar todas as props que o render traz de dentro da função no console.log: **field, fieldState, formState**.

* formState – traz informações gerais do formulário
* fieldState – traz informações sobre o campo que passamos para o controller
* field – traz os eventos do campo que passamos para o formulário, **onChange, onBlur, value.**

<Controller

    control={control}

    name='type'

    render={(props) => {

        console.log(props)

        return (

            <TransactionType>

                <TransactionTypeButton

                    variant='income'

                    value='income'

                >

                <ArrowCircleUp size={24}/>

                    Entrada

                </TransactionTypeButton>

                <TransactionTypeButton

                     variant='outcome'

                     value='outcome'

                >

                    <ArrowCircleDown size={24}/>

                     Saída

                </TransactionTypeButton>

            </TransactionType>

        )

    }}

/>

Então desestruturamos as props, utilizando apenas o field para pegar o **onChange** e o **value** do controller e passar os mesmos para o html. Neste caso estou usando um RadioGroup no componente, que utiliza uma propriedade chamada **onValueChange no lugar de onChange** nativo do html, veja exemplo:

 <Controller

    control={control}

    name='type'

    render={({ field }) => {

console.log(field)

        return (

            <TransactionType

                onValueChange={field.onChange}

                value={field.value}

> Restante do código .....

)}}/>

## Iniciar com um campo do radio selecionado pelo controller

Como passamos a propriede value com o field.value, podemos passar o defaulValues para a variável useForm:

 const {

    control,

    register,

    handleSubmit,

    formState: { isSubmitting }

} = useForm<NewTransactionFormInputs>({

    resolver: zodResolver(newTransactionFormSchema),

    defaultValues: {

        type: 'income'

    }

})

# Aprimorando requisições

## Campo de Busca

Com o useEffect a função de **loadTransactions** carrega apenas uma vez, para ela carregar no momento da busca vamos enviar ela pelo contexto, então no arquivo context faça:

1. Mudar o nome **de loadTransactions para fetchTransactions**
2. Passar a tipagem
3. Enviar como **value do provider**
4. Passar um **query** de busca opcional como uma **string**
5. Criar uma variável com a url da tabela
6. Criar um ternário para o searchParams
7. Passar a variável url para o fetch

**2**

interface TransactionContextType {

    transactions: Transaction[];

    fetchTransactions:(query?: string) => Promise<void>

}

**3**

return (

    <TransactionsContext.Provider value={{

        transactions,

        fetchTransactions,

    }}>

        {children}

    </TransactionsContext.Provider>

)

**4, 5, 6**

async function fetchTransactions(query?: string) {

    const url = new URL('http://localhost:3000/transactions')

    if (query) {

        url.searchParams.append('q', query)

    }

    const response = await fetch(url)

    const data = await response.json()

    setTransactions(data)

}

No componente de busca pegamos a função **fetchTransactions do contexto.**

Criamos a função **async handleSearch** para o handleSubmit e passamos a **fetchTransactions** como await dentro dela recebendo **data.query.**

const searchFromSchema = z.object({

    query: z.string(),

})

type SearchFormInputs = z.infer<typeof searchFromSchema>

export function SearchForm() {

    const { fetchTransactions } = useContext(TransactionsContext)

    const {

        register,

        handleSubmit,

        formState: { isSubmitting }

    } = useForm<SearchFormInputs>({

        resolver: zodResolver(searchFromSchema)

    })

    async function handleSearchTransactions(data: SearchFormInputs) {

        await fetchTransactions(data.query)

    }

    return (

        <SearchFormContainer onSubmit={handleSubmit(handleSearchTransactions)}>

            <input type="text"

            placeholder="Busque por transações"

            {...register('query')}

        />

            <button type="submit" disabled={isSubmitting}>

                <MagnifyingGlass size={20}/>Buscar

            </button>

        </SearchFormContainer>

    )

}

## Configurando Axios

Instalando o Axios:

npm i axios

Criar uma **pasta lib** dentro da **pasta src** com um arquivo chamado **axios.ts**

Dentro do arquivo passe a **url base** do servidor.

import axios from "axios";

export const api = axios.create({

    baseURL: 'http://localhost:3000'

})

Voltando no contexto, altere a função de requisição:

import { api } from "../lib/axios";

async function fetchTransactions(query?: string) {

    const response = await api.get('transactions', {

        params: {

            q: query,

        }

    })

    setTransactions(response.data)

}

useEffect(() => {

    fetchTransactions()

}, [])

## Cadastrando novo item (Transação)

Crie uma variável salvando os campos do **data.**

Crie um método post, passando os campos da variável **data**.

async function handleCreateNewTransaction(data: NewTransactionFormInputs) {

    const { description, price, category, type } = data

    await api.post('transactions', {

        description,

        price,

        category,

        type,

        createdAt: new Date(),//campo não necessário com backend real

    })

}

Isso já faz salvar os dados, mas não reseta o modal, para isso configure um **reset** no **useForm** e passe o para o método post.

export function NewTransactionModal() {

    const {

        control,

        register,

        handleSubmit,

        formState: { isSubmitting },

        reset

    } = useForm<NewTransactionFormInputs>({

        resolver: zodResolver(newTransactionFormSchema),

        defaultValues: {

            type: 'income'

        }

    })

    async function handleCreateNewTransaction(data: NewTransactionFormInputs) {

        const {

            description,

            price,

            category,

            type

        } = data

        await api.post('transactions', {

            description,

            price,

            category,

            type,

            createdAt: new Date(),//campo não necessário com backend real

        })

        reset()

    }

## Ordenar campos por data de criação

Adicione **\_sort** e **\_order** na função de requisição no contexto

async function fetchTransactions(query?: string) {

    const response = await api.get('transactions', {

        params: {

            \_sort: 'createdAt',

            \_order: 'desc',

            q: query,

        } //sort e order ordena os campos da tabela pela ordem

    })

setTransactions(response.data)}

## Atualizar campos na tela no momento da criação

**Api.post já retorna** os campos atualizados da tabela, para testar basta:

const response = await api.post('transactions', {

        description,

        price,

        category,

        type,

        createdAt: new Date(),

    })

console.log(response.data)

Então podemos criar uma função **dentro do contexto** que cria uma nova transação atualizando o estado destas transações:

**1 –** Criar a interface

interface CreateTransactionInput {

    description: string;

    price: number;

    category: string;

    type: 'income' | 'outcome';

}

**2 –** Criar a função repassando o **api.post** para ela atualizando o estado

async function createTransaction(data: CreateTransactionInput) {

    const { description, price, category, type } = data

    const response = await api.post('transactions', {

        description,

        price,

        category,

        type,

        createdAt: new Date(),//campo não necessário com backend real

    })

    setTransactions(state => [response.data, ...state])

}

**3 –** Adicionar a função a interface do contexto

interface TransactionContextType {

    transactions: Transaction[];

    fetchTransactions:(query?: string) => Promise<void>

    createTransaction: (data: CreateTransactionInput) => Promise<void>

}

**4 –** Enviar a função

return (

    <TransactionsContext.Provider value={{

        transactions,

        fetchTransactions,

        createTransaction,

    }}>

        {children}

    </TransactionsContext.Provider>

)

Voltando para o componente que recebe a função, utilizaremos o **useContext** trazendo a função para criação:

const { createTransaction } = useContext(TransactionsContext)

async function handleCreateNewTransaction(data: NewTransactionFormInputs) {

    const { description, price, category, type } = data

    await createTransaction({

        description,

        price,

        category,

        type,

    })

    reset()

}