# 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>

    )

}