Objetivos:

I. Introdução ao Redux;
II. Conceitos fundamentais do Redux;
III. Vite;
IV. Preparação do ambiente de desenvolvimento.

I. Introdução ao Redux

Em aplicações React, o estado é uma peça fundamental para a construção de interfaces dinâmicas. Nas aulas anteriores, aprendemos a gerenciar estados locais e globais utilizando o Context API. Embora o Context API seja uma excelente solução para compartilhar dados entre componentes, ele pode se tornar limitado em aplicações maiores, com estados mais complexos.

O Redux é uma biblioteca projetada para gerenciar estados globais de forma centralizada e previsível. Ele é baseado em três pilares principais:

1. Store (armazenamento): um único objeto JS que contém o estado global da aplicação;
2. Actions (ações): objetos que descrevem "o que aconteceu" na aplicação;
3. Reducers (redutores): funções puras que recebem o estado atual e uma ação, retornando um novo estado.

Esses princípios fazem do Redux uma ferramenta poderosa para:

- Organizar o estado de forma clara e centralizada;
- Escalar aplicações com muitos estados e interações;
- Depurar facilmente, graças a ferramentas como Redux DevTools.

Comparação entre Redux e Context API

O Context e o Redux são ambos mecanismos para gerenciar estado em aplicações React, mas possuem abordagens e aplicações diferentes:

Característica	Redux	Context
Propósito	É ideal para gerenciar estados complexos e dinâmicos, como listas de produtos, carrinhos de compras ou estados sincronizados com back-end.	Foi projetado para compartilhar dados simples (como tema, idioma ou informações do usuário logado). É uma ferramenta leve, sem sobrecarga significativa.
Estrutura	Store, actions, reducers	Provider e Consumer
Complexidade do estado	Oferece uma estrutura mais robusta, permitindo melhor organização de estados complexos por meio de reducers e middlewares.	Torna-se difícil de manter em casos de estados grandes ou muitos "dispatches" diferentes.
Escalabilidade	Permite otimizações granulares, garantindo que	Funciona bem em aplicações pequenas e médias,

	apenas os componentes afetados sejam re-renderizados.	mas pode sofrer problemas de performance em atualizações frequentes, já que a re-renderização pode afetar vários componentes.
Desempenho	Pode ter um impacto maior no desempenho em aplicações muito grandes	Geralmente tem um desempenho melhor
Aprendizado	Requer mais aprendizado	Mais fácil de aprender

II. Conceitos fundamentais do Redux

Para compreender como o Redux funciona e como ele gerencia o estado global de forma centralizada, é necessário entender os principais elementos que compõem sua arquitetura: Store, Actions, Reducers e Dispatch. Esses conceitos trabalham juntos para criar um fluxo previsível e organizado de dados na aplicação.

Store: O armazenamento central

A Store é o coração do Redux. Ela é responsável por armazenar o estado global da aplicação e agir como a única fonte de verdade. Isso significa que qualquer dado compartilhado entre componentes será mantido na Store.

Características da Store:

- Contém o estado global da aplicação;
- Permite o acesso ao estado por meio do método getState();
- Disponibiliza o método dispatch(action) para enviar ações e modificar o estado;
- Oferece o método subscribe(listener) para reagir a mudanças no estado.

Exemplo de criação de uma Store:

```
import { createStore } from 'redux';
import { rootReducer } from './reducers';

const store = createStore(rootReducer);
```

Actions: Descrevendo o que deve acontecer

As Actions são objetos simples que descrevem uma intenção de modificar o estado. Elas contêm uma propriedade obrigatória chamada type, que indica o tipo de ação a ser executada, e podem conter outros dados relacionados à ação.

Estrutura de uma Action:

```
type Action = {
  type: string;
  payload?: User; // Dados adicionais que podem ser enviados junto com a ação
};
```

Exemplo de uma Action:

```
const userAction = {
  type: "string",
```

```
  payload: {
    id: 1,
    name: "Ana",
    age: 21
  },
};
```

As Actions são apenas descrições do que deve acontecer. Elas não alteram o estado diretamente.

Reducers: Funções puras que atualizam o estado

Os Reducers são funções puras responsáveis por processar uma Action e retornar um novo estado. Eles recebem o estado atual (`state`) e a `action` como parâmetros e, com base no tipo da ação, retornam o estado atualizado.

Características dos Reducers:

- São funções puras, ou seja, não produzem efeitos colaterais;
- Não devem modificar diretamente o estado; ao invés disso, devem retornar um novo objeto com as alterações.

Exemplo de um Reducer:

```
const userSlice = createSlice({
  name: "cadastro",
  initialState: {
    users: [],
  } as UserState,
  reducers: {
    removeUser: (state, action: PayloadAction<number>) => {
      // Filtra o array e atualiza o estado
      state.users = state.users.filter((user) => action.payload !== user.id);
    },
  },
});
```

Dispatch: Disparando ações

O Dispatch é o método utilizado para enviar Actions para a Store. Quando uma Action é disparada, a Store repassa essa Action para o Reducer, que calcula o novo estado.

```
const dispatch: AppDispatch = useDispatch();
dispatch(removeUserAndCars(id));
```

Como o Redux gerencia o fluxo de dados

O Redux segue um fluxo de dados unidirecional bem definido, o que facilita a previsibilidade e o rastreamento do estado. Esse fluxo pode ser resumido em quatro etapas:

1. Disparar uma ação: um componente dispara uma action usando o método `dispatch()`;

```
const dispatch: AppDispatch = useDispatch();
dispatch(removeUserAndCars(id));
```

2. Processar no Reducer: a Store invoca o reducer, que calcula o novo estado com base na action;

```
removeUser: (state, action: PayloadAction<number>) => {
  state.users = state.users.filter((user) => action.payload !== user.id);
},
```

3. Atualizar a Store: o novo estado retornado pelo Reducer substitui o estado anterior na Store;
4. Notificar os componentes: todos os componentes que dependem do estado atualizado são notificados e re-renderizados, se necessário.

Diagrama do fluxo de dados:

Component -> dispatch(Action) -> Reducer -> Store -> Component

III. Vite

O Vite é uma ferramenta de construção de aplicações web de nova geração que se destaca pela sua velocidade e leveza. Ele oferece um servidor de desenvolvimento instantâneo e hot reloading extremamente rápido, o que torna o processo de desenvolvimento muito mais ágil e prazeroso.

- **Velocidade:** O Vite é significativamente mais rápido do que outras ferramentas de construção, como o Webpack. Ele usa módulos ES nativos para servir o código durante o desenvolvimento, o que elimina a necessidade de empacotamento.
- **Hot Reloading:** O hot reloading do Vite é extremamente rápido. As alterações no código são refletidas instantaneamente no navegador, sem a necessidade de recarregar a página.
- **Facilidade de uso:** O Vite é muito fácil de configurar e usar. Ele oferece uma experiência de desenvolvimento intuitiva e agradável.
- **Suporte a TypeScript:** O Vite oferece suporte nativo ao TypeScript, o que facilita o desenvolvimento de aplicações React com tipagem estática.

IV. Preparação do ambiente de desenvolvimento

Passo 1 - Criar um projeto React TS com Vite

Para reproduzir os exemplos, crie um projeto React TS usando a ferramenta de build Vite:

```
npm create vite@latest front -- --template react-ts
```

Passo 2 - Instalar as dependências do Redux

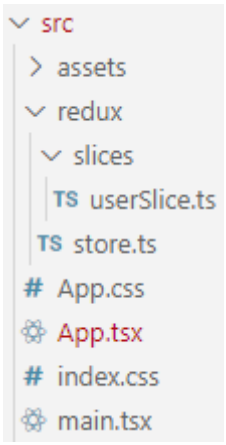
```
npm i @reduxjs/toolkit react-redux
```

Explicação dos pacotes:

- `@reduxjs/toolkit`: fornece ferramentas e convenções para simplificar o uso do Redux;

- react-redux: faz a integração entre Redux e React, permitindo que os componentes acessem o estado da Store.

Passo 3 – Criar a estrutura do Redux



Vamos organizar os arquivos do Redux em uma estrutura clara e modular. Geralmente, utilizamos uma pasta chamada store ou redux na raiz do projeto, contendo:

- store.ts: configuração da Store;
- slices: pasta para armazenar os "slices" (fatias do estado) criados com o Redux Toolkit;
- Outros arquivos: para middleware, seletores ou ações personalizadas, se necessário.

Passo 4 – Configurar a Store

A Store é o local onde o estado global da aplicação será armazenado.

Exemplo de configuração da Store (arquivo: src/redux/store.ts):

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './slices/userSlice';

export const store = configureStore({
  reducer: {
    userObject: userReducer,
  },
});

// Tipos para uso com TypeScript
export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

Explicação:

- configureStore: simplifica a criação da Store e já inclui ferramentas como Redux DevTools e middleware padrão;
- Reducers: são combinados em um objeto reducer para organizar o estado em diferentes "fatias" (slices);
- A store é criada utilizando configureStore. Nela, passamos um mapa de reducers, onde cada chave é um "slice" do estado global e cada valor é um reducer correspondente;
- reducer:
  - A chave userObject será usada como o identificador do estado controlado pelo userReducer;
  - No estado global, o Redux armazenará os dados gerenciados pelo userReducer sob a chave userObject.

Passo 5 – Criar um Slice com Redux Toolkit

Um slice é uma fatia do estado gerenciada por um Reducer e suas respectivas Actions.

Exemplo de um Slice (arquivo: src/redux/slices/useSlice.ts). O código define um slice do Redux chamado userSlice, que gerencia o estado de um cadastro de usuários:

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit";

export interface User {
  id: number;
  name: string;
  age: number;
}

interface UserState {
  users: User[];
}

const userSlice = createSlice({
  name: "cadastro",
  initialState: {
    users: [],
  } as UserState,
  reducers: {
    addUser: (state, action: PayloadAction<User>) => {
      state.users.push(action.payload);
    },
    removeUser: (state, action: PayloadAction<number>) => {
      // Filtra o array e atualiza o estado
      state.users= state.users.filter((user) => action.payload !== user.id);
    },
    incrementAge: (state, action: PayloadAction<number>) => {
      // Filtra o array, mantendo apenas os usuários com id diferente do user.id
      const user = state.users.find((user) => action.payload === user.id);
      if( user ){
        user.age = user.age + 1; // Incrementa a idade no estado
      }
    }
  },
});

export const { addUser, removeUser, incrementAge } = userSlice.actions;
export default userSlice.reducer;
```

Explicação:

- UserState: representa o estado completo gerenciado por este slice. É composto de um array chamado users, que contém objetos do tipo User;

- name: identifica o slice, neste caso "cadastro". Esse nome será usado ao criar as actions e no estado global;
- initialState: define o estado inicial do slice. Aqui, o estado começa como um array vazio de usuários (users: []);
- reducers: contém as funções (reducers) que atualizam o estado. Cada reducer representa uma operação que pode ser realizada no estado;
- userSlice.actions:
  - Gera automaticamente as actions correspondentes aos reducers definidos;
  - Cada action possui um type baseado no nome do slice e do reducer (exemplo: cadastro/addUser).
- userSlice.reducer:
  - Exporta o reducer do slice para ser utilizado na configuração da store.

Passo 6 – Integrar o Redux ao React

Com a Store configurada e os slices criados, precisamos conectar o Redux à aplicação React. Para isso, utilizamos o componente Provider do pacote react-redux.

Exemplo de integração no src/main.tsx:

```
import { createRoot } from "react-dom/client";
import App from "./App.tsx";
import { Provider } from "react-redux";
import { store } from "./redux/store.ts";

createRoot(document.getElementById("root")!).render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

Explicação:

- Provider: torna a Store disponível para todos os componentes da aplicação React;
- store: a Store configurada anteriormente é passada como propriedade.

Passo 7 – Conectar componentes ao Redux

Para acessar o estado e despachar ações dentro dos componentes, utilizamos os hooks `useSelector` e `useDispatch` do react-redux.

Exemplo de componente conectado ao Redux (arquivo: src/App.tsx):

```
import { useDispatch, useSelector } from "react-redux";
import { AppDispatch, RootState } from "./redux/store";
import { addUser, incrementAge, removeUser, User } from "./redux/slices/userSlice";
import { useState } from "react";
```

```
export default function App() {
  const users = useSelector((state: RootState) => state.userObject.users);
  const [id, setId] = useState(1);
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  const dispatch: AppDispatch = useDispatch();

  const handleSave = () => {
    const user: User = {
      id,
      name,
      age: parseInt(age),
    };
    dispatch(addUser(user));
    setId((prev: number) => prev + 1);
  };

  const handleAge = (id:number) => {
    dispatch( incrementAge(id) );
  };

  const handleRemove = (e: React.MouseEvent<HTMLButtonElement>, id: number) => {
    e.preventDefault();
    dispatch(removeUser(id));
  };

  return (
    <div>
      <div>
        <div>
          <label htmlFor="name">Nome</label>
          <input
            id="name"
            value={name}
            onChange={(e) => setName(e.target.value)}
          />
        </div>
        <div>
          <label htmlFor="age">Idade</label>
          <input
            id="age"
            value={age}
            onChange={(e) => setAge(e.target.value)}
          />
        </div>
        <div>
          <button onClick={handleSave}>Salvar</button>
```

```
        </div>
      </div>
      <ul>
        {users.map((user) => (
          <li key={user.id}>
            <span onClick={() => handleAge(user.id)} onContextMenu={(e) =>
handleRemove(e, user.id)}>
              {user.name} - {user.age}
            </span>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

V. Exercícios

Exercício 1 – Altere o código do exemplo para que o id do usuário seja criado dentro do reducer `addUser`.

Dica:

- Use o código a seguir para obter o maior id armazenado no array users do estado:

```
const newId =
  state.users.length > 0
    ? Math.max(...state.users.map((user) => user.id)) + 1
    : 1;
```

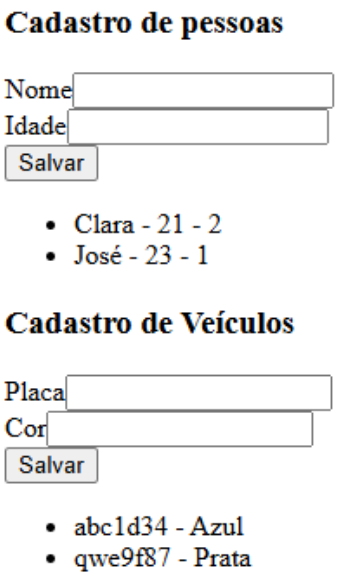
Exercício 2 – Altere o código do Exercício 1 para garantir que o array users esteja sempre ordenado por idade.

Dica:

- Use o código a seguir para ordenar o array users do estado:

```
state.users.sort((a, b) => a.age - b.age);
```

Exercício 3 – Adicione no código do Exercício 2 um slice para gerenciar o cadastro de veículos.



Requisitos:

1. O cadastro de veículos é formado por placa e cor;
2. O cadastro deverá ter as operações de inserir e remover veículos.

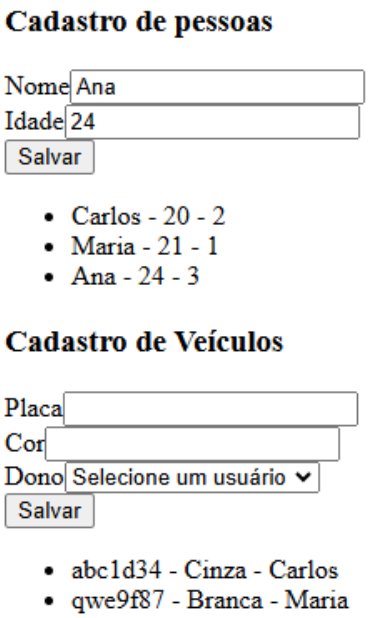
Dicas:

- Crie um slice no arquivo `src/redux/slices/carSlice.ts`;
- Crie o tipo de dado `Car`:

```
export interface Car {
  plate: string;
  color: string;
}
```

- Registre o reducer de veículos na configuração da store (`src/redux/store.ts`).

Exercício 4 – Altere o código do Exercício 3 para cada veículo estar associado a um usuário.



Requisito:

- Ao remover o usuário, devem ser removidos os veículos dele.

Dicas:

- Crie um reducer no carSlice para remover um veículo. Exemplo de nome de função reducer, removeCarsByUser;
- Mantenha os reducer do userSlice inalterados;
- Adicione o seguinte código no arquivo userSlice.ts. No componente basta usar removeUserAndCars no lugar de removeUser.

```
// Thunk para remover usuário e carros associados
export const removeUserAndCars =
  (userId: number) => (dispatch: AppDispatch) => {
    dispatch(removeUser(userId)); // Remove o usuário
    // Remove os carros associados
    dispatch(removeCarsByUser(userId));
  };
```

Thunk

Um thunk é uma função que encapsula outra função ou operação para adiar sua execução. No contexto do Redux, um thunk é uma função que pode ser despachada usando dispatch, e que, em vez de realizar diretamente a alteração do estado, pode executar lógica assíncrona ou complexa antes de despachar outras ações.

No Redux, o middleware redux-thunk nos permite escrever funções assíncronas ou com lógica de negócios avançada que despacham outras ações. Isso é especialmente útil para:

- Fazer chamadas de API antes de alterar o estado;
- Encadear múltiplas ações relacionadas;
- Combinar ações que precisam ser disparadas em sequência.

No código anterior, o reducer `removeUser` foi adaptado para que ele não apenas removesse um usuário, mas também eliminasse os carros associados a esse usuário. Para isso, tivemos de despachar duas ações:

- removeUser: remove o usuário do estado;
- removeCarsByUse: remove todos os carros associados ao usuário.

No entanto, reducers no Redux não podem despachar outras ações diretamente, pois eles precisam ser funções puras que apenas recebem o estado atual e uma ação, e retornam o novo estado. Para resolver esse problema, usamos um thunk.