

**Descrição da atividade:** fazer dois aplicativos web para mostrar a previsão do tempo.

**Data de entrega:** presencialmente na aula de 06/junho (terça-feira).

**Forma de entrega:** individual.

**Objetivos:**

- React Redux;
- Context e hooks;
- Requisições HTTP;
- CSS e Styled-components.

**Requisitos funcionais:**

- A aplicação deverá exibir a previsão do tempo da cidade fornecida pelo usuário, assim como é mostrado ao lado;
- A aplicação deverá consumir os serviços da API de previsão do tempo do CPTEC INPE (<http://servicos.cptec.inpe.br/XML>).

A previsão é obtida em dois passos:

Primeiro: é necessário obter o código da cidade:

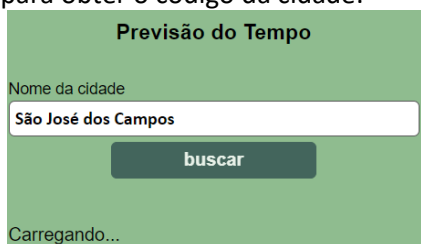
<http://servicos.cptec.inpe.br/XML/listaCidades?city=sao%20jose%20dos%20campos>

Segundo: obter a previsão usando o código da cidade:

<http://servicos.cptec.inpe.br/XML/cidade/4963/previsao.xml>

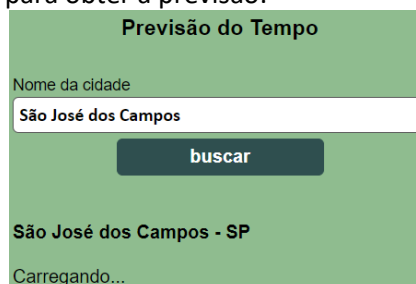
- A aplicação deverá exibir o estado da requisição:

Enquanto é processada a requisição para obter o código da cidade:

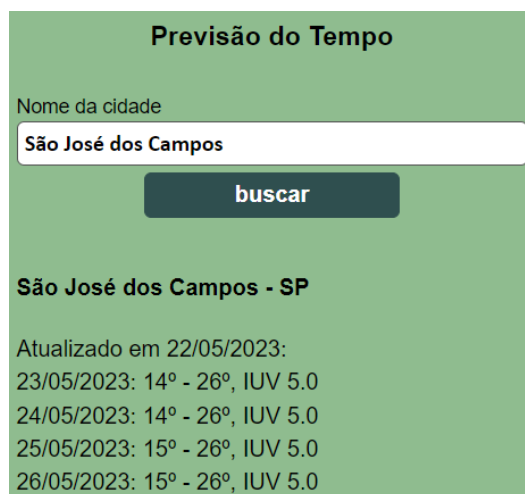


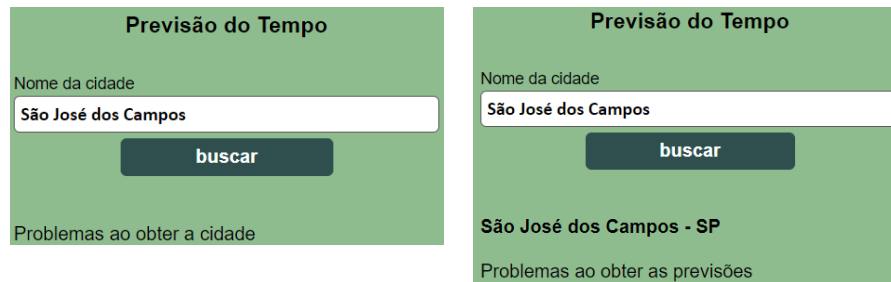
Erro na requisição para obter o código da cidade:

Enquanto é processada a requisição para obter a previsão:



Erro na requisição para obter a previsão:



**Requisitos não funcionais:**

- Deverá ser utilizado como base o código disponível em <https://github.com/arleysouza/atividade-base>. Os pacotes components, services e types não poderão ser alterados;
- Deverá ser codificada duas aplicações distintas, uma usando Context e Hooks e outra usando Redux.

**Redux:**

O Redux é uma biblioteca de gerenciamento do objeto state através da árvore de componentes, assim como faz o Context. O seu uso é aconselhável em situações específicas:

- Existem propriedades do state que precisam ser usadas em diferentes componentes do aplicativo. O Redux é uma alternativa ao objeto Context;
- O estado é atualizado com frequência;
- A lógica para atualizar o estado pode ser complexa;
- O aplicativo tem uma base de código de médio ou grande porte e pode ser trabalhado por muitas pessoas;
- É necessário saber como o estado está sendo atualizado ao longo do tempo.

Para mais detalhes acesse <https://redux.js.org/faq/general#when-should-i-use-redux>.

Crie um projeto React TS usando:

```
npx create-react-app exemplo --template typescript
```

Adicione os pacotes do react-redux e @reduxjs/toolkit (RTK – Redux Toolkit):

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

O código usado como exemplo está disponível em <https://github.com/arleysouza/redux-example>.

Os principais elementos do Redux são:

- actions** (ações): descrevem eventos que acontecem na aplicação. A única forma de mudar o estado é criando uma ação e despachá-la (dispatch). No exemplo a seguir, a ação é implementada usando a função dispatch para acionar o reducer setA para alterar o estado da propriedade a:

```
<button onClick={()=>dispatch(setA(2))}>setA</button>
```

- **dispatch** (despachar): é quem presta atenção em eventos dentro da aplicação (trata-se de um event listener ou "ouvinte de eventos" do Redux). Isto é, quando um evento for chamado - um clique em botão, por exemplo -, ele executa um reducer com a devida action;
- **reducers** (redutores): são funções responsáveis por alterar o estado da aplicação. Cada reducer funciona como se fosse um "event listener", ou seja, ele fica "escutando" quando uma ação é disparada e, se for necessário, ele retorna um novo estado conforme as informações recebidas pela action. A principal regra para os reducers é que eles precisam ser funções puras, ou seja, não podem gerar nenhum efeito colateral, como lógicas assíncronas;
- **store** (armazenamento): é o local onde todo o estado fica armazenado, ou seja, é onde todas as informações da aplicação ficam centralizadas. A Store é responsável por gerenciar as atualizações do estado, através dos Reducers. Ela também fica responsável por disponibilizar o método dispatch, pelo qual as actions são chamadas. A store é definida usando a função configureStore do RTK:

```
export const store = configureStore({  
  reducer: exemploSlice  
});
```

A função createSlice cria o estado na propriedade initialState e define a lógica de atualização na propriedade reducers. Neste exemplo, o estado possui apenas as propriedades a e b e as funções seta, setB e incA para modificar o estado.

A função createSlice recebe uma string (name), um objeto (com as propriedades e valores iniciais do estado) e um objeto com as funções reducers (<https://redux-toolkit.js.org/api/createslice>)

```
const exemploSlice = createSlice({  
  name: "exemplo", //o name do slice é obrigatório  
  initialState: {  
    a: 0,  
    b: 0  
  },  
  reducers: {  
    setA: (state, action: PayloadAction<number>) => {  
      state.a = state.a + action.payload;  
    },  
    setB: (state, action: PayloadAction<number>) => {  
      state.b = state.b + action.payload;  
    },  
    incA: (state) => {  
      state.a += 1;  
    }  
  }  
});
```

A store é propagada na árvore de componentes usando a propriedade `store` do componente `Provider` (<https://react-redux.js.org/api/provider>).

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import { Provider } from "react-redux";
import { store } from "../features/store";

const root = ReactDOM.createRoot(
  document.getElementById("root") as HTMLElement
);
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

A única forma de mudar o estado é criando uma ação (action – um objeto que descreve o que aconteceu) e despachá-lo (dispatch). No código a seguir a função `dispatch` chama a função `setA` do reducer, o parâmetro `2` estará disponível na propriedade `action.payload` do reducer.

```
function App() {
  const dispatch = useDispatch<AppDispatch>();
  const a = useSelector( (state: RootState) => state.a);
  const b = useSelector( (state: RootState) => state.b);

  return (
    <div>
      <div>A:{a} B:{b}</div>
      <button onClick={()=>dispatch(setA(2))}>setA</button>
      <button onClick={()=>dispatch(setB(2))}>setB</button>
      <button onClick={()=>dispatch(set(3))}>set</button>
    </div>
  );
}
```

Um reducer não pode chamar outro reducer. No exemplo a seguir a função `incA()` não será invocada:

```
setB: (state, action: PayloadAction<number>) => {
  state.b = state.b + action.payload;
  incA(); // não funciona
},
```

Para resolver este problema temos de colocar a chamada dos reducers dentro de uma função thunk:

```
export const set =
  (nro: number): AppThunk<void> =>
```

```
async (dispatch, getState) => {  
  dispatch(setB(nro));  
  dispatch(incA());  
};
```

A palavra "thunk" é um termo de programação que significa "um pedaço de código que faz algum trabalho atrasado". Em vez de executar alguma lógica agora, podemos escrever um corpo de função ou código que pode ser usado para executar o trabalho posteriormente.

Para o Redux especificamente, "thunks" são um padrão de escrita de funções com lógica interna que podem interagir com os métodos `dispatch` e `getState`.

Thunks são uma abordagem padrão para escrever lógica assíncrona em aplicativos Redux e são comumente usados em operações de request de dados. No entanto, eles podem ser usados para uma variedade de tarefas e podem conter lógica síncrona e assíncrona. As funções thunk recebem como parâmetro as funções `dispatch` e `getState` para interagir com os reducers e estado da store

As funções thunk não são chamadas diretamente pelo código do aplicativo. Em vez disso, elas são passadas para o `dispatch()`, assim como fizemos em:

```
<button onClick={()=>dispatch(set(3))}>set</button>
```

Para mais detalhes <https://redux.js.org/usage/writing-logic-thunks>.