Gerenciando o estado em aplicativos React em escala

Parabéns!!

Se as coisas estão ficando complicadas ao gerenciar o estado de sua aplicação isso significa que a aplicação, equipe e empresa estão crescendo rapidamente!

Com algumas técnicas, podemos simplificar as coisas para que possamos concentrar nosso tempo no desenvolvimento de novas features.

Vamos nessa! 💪

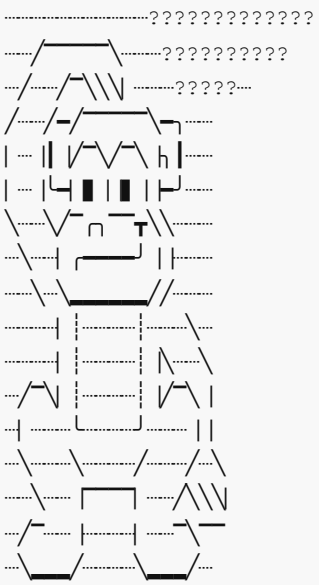
Introdução

- Lucas Feliciano 👋
- Gerente de engenharia @ Mollie.com 💳
- Focado em aplicações JavaScript de larga escala 📈
- Entusiasta de Magic the Gathering 🎲
- Guitarrista 🎸

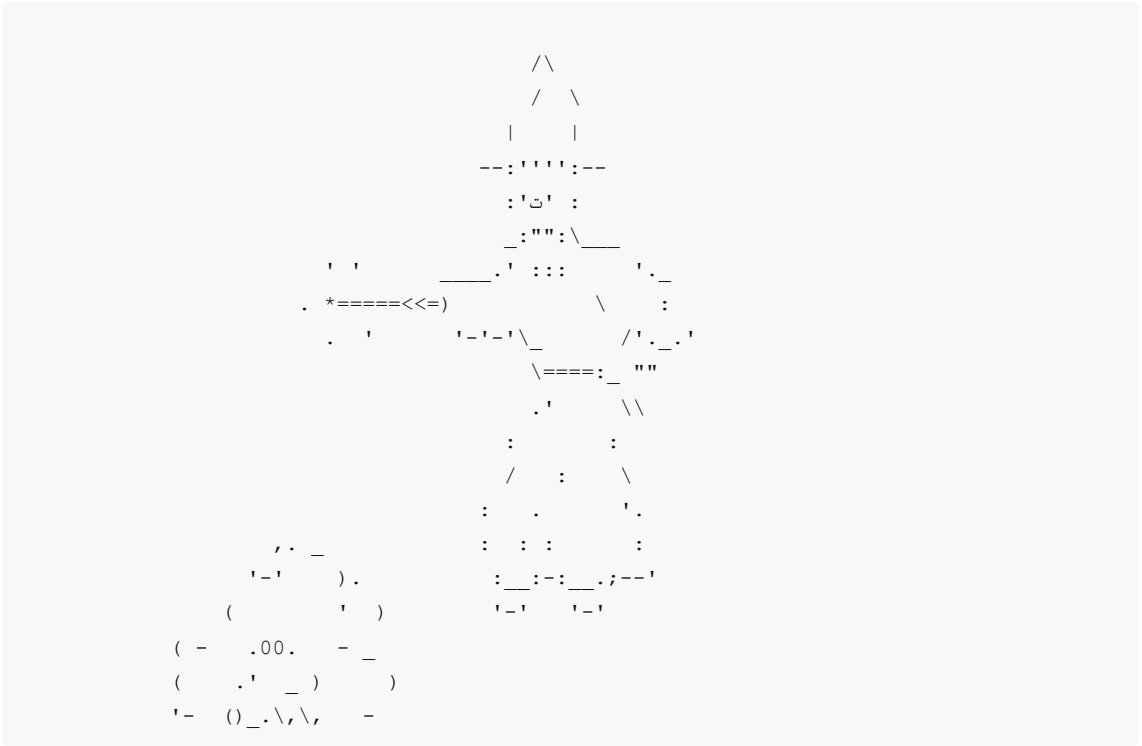
Agenda

- Tenicas de gerenciamento de estado
  - Estado local
  - Componentes contêiner e de apresentação
  - Contexto
  - Componentes compostos
  - Hook Personalizado
- Reflexão na introdução de estado
- Testes
- Demo

Local state, redux, hooks, context... Eita...



Simplificando o estado de aplicação em escala



Estado local

- Exclusivo de um único componente e é gerenciado internamente
- Não pode ser acessado ou modificado diretamente por nenhum outro componente

```
function ToggleSwitch() {
  const [isOn, setIsOn] = useState(false);

  function toggle() {
    setIsOn(!isOn)
  }

  return <div onClick={toggle}>{isOn ? 'On' : 'Off'}</div>
}
```

Componentes contêiner e componentes de apresentação

É comum separar as funções de um componente em dois tipos:

- componentes de "apresentação"
- componentes "contêiner"

Componentes de apresentação

- Responsáveis por renderizar a interface do usuário
- Frequentemente escritos como funções "puras"
- Podem receber props e retornam JSX

- Se importam com a aparência

```
function UserCard({ user }) {
  return (
    <div>
      <h3>{user.name}</h3>
      <p>{user.email}</p>
    </div>
  );
}
```

Componentes contêiner

- Responsáveis pelo gerenciamento de dados e lógica
- Frequentemente conectados ao estado do aplicativo
- Passam dados e callbacks para componentes de apresentação por props
- Se importam com a forma como as coisas funcionam

```
import React, { useState, useEffect } from 'react';

function UserDetail(props) {
  const [user, setUser] = useState({});

  useEffect(() => {
    fetch(`/api/users/${props.userId}`)
      .then((res) => res.json())
      .then((data) => setUser(data));
  }, [props.userId]);

  return <UserCard user={user} />;
}
#### Atenção com prop-drilling
- Complica a estrutura da aplicação
- Dificulta a compreensão do fluxo de dados
- Requer manutenção constante

```jsx
import React from 'react';

function Container(props) {
  const user = {
    name: 'John Doe',
    email: 'john@example.com',
  };

  return (
    <ComponentA user={user} />
  );
}

function ComponentA(props) {
```
```

```
  return (
    <div>
      <ComponentB user={props.user} />
    </div>
  );
}

function ComponentB(props) {
  return (
    <div>
      <ComponentC user={props.user} />
    </div>
  );
}

function ComponentC(props) {
  return (
    <div>
      <ComponentD user={props.user} />
    </div>
  );
}

function ComponentD(props) {
  return (
    <div>
      {props.user.name} ({props.user.email})
    </div>
  );
}
```

Contexto

- Acessível em múltiplos componentes
- Normalmente é exposto em um nível superior na aplicação
- Útil para armazenar valores compartilhados entre componentes

```
const UserContext = React.createContext(user);


  const [user, setUser] = useContext(UserContext)

  function changeUserName (name) => {
    setUser({...user, name })
  }


export const useUserContext= () =>  {
  return {
    changeUserName,
    user
```

```
  }
}

function App() {
  return (
    <UserContext.Provider value={userOne}>
      <UserContext.Provider value={userTwo}>
        <NavBar />
      </UserContext.Provider>
    </UserContext.Provider>
  );
}

function NavBar() {
  const {user, changeUserName} = useUserContext();

  return (

    <UserAvatar user={user}/>
  );
}
```

Componentes compostos

- Úteis quando você deseja criar um grupo de componentes relacionados
- Isolam estado e comportamento
- Mais fáceis de raciocinar e manter
- Requerem um pouco mais de configuração em comparação com os componentes regulares
- Casos de uso estritos

```
// containers/Form.js
import React, { useState, createContext } from 'react';

const FormContext = createContext();

function Form({ children }) {
  const [values, setValues] = useState({});

  function handleChange(event) {
    setValues({
      ...values,
      [event.target.name]: event.target.value
    });
  }

  return (
    <FormContext.Provider value={{ values, handleChange }}>
      <form>
        {children}
      </form>
    </FormContext.Provider>
```

```
  );
}

Form.Input = function FormInput({ name, ...props }) {
  const { values, handleChange } = useContext(FormContext);

  return (
    <input
      name={name}
      value={values[name] || ''}
      onChange={handleChange}
      {...props}
    />
  );
}

Form.SubmitButton = function FormSubmitButton({ children }) {
  const { values, handleChange } = useContext(FormContext);

  function handleSubmit() {
     submitRequest('/api/login', values)
  }

  return <button type="submit" onClick={handleSubmit}>{children}</button>;
}

export default Form;

// containers/Login.js
import Form from 'containers/Form';

function Login() {

  return (
    <Form>
      <Form.Input name="email" type="email" />
      <Form.Input name="password" type="password" />
      <Form.SubmitButton>Submit</FormSubmitButton>
    </Form>
  );
}
```

Hook personalizado

- Permite extrair lógica de estado de um componente
- Facilita o teste da lógica e estado isoladamente
- Permite que você reutilize a lógica de estado em sua aplicação
- Pode adicionar um nível de descontinuidade ao seu código
- Não é um substituto para redux ou outras bibliotecas de gerenciamento de estado
- Pode não ser adequado para grandes aplicações com estado complexo

```
// hooks/useFetchData.js
import { useState, useEffect } from 'react';

function useFetchData(url) {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch(url);
        const data = await response.json();
        setData(data);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    }

    fetchData();
  }, [url]);

  return { data, error, loading };
}

export default useFetchData

// containers/Products/index.js
import useFetchData from 'hooks/useFetchData';

function Products() {
  const {data, error, loading} = useFetchData('/products)

  if(loading) return 'Loading...'
  if(error) return 'Error...'

  return <ProductList products={data} />
}
```

Reflexão na introdução de estado

- Questione-se: Preciso de estado?
- Comece com o estado local
- Levante o estado para um contêiner se for necessário compartilhar estado
- Use o estado global se o valor for necessário para toda aplicação

Testes

- Garantem que o código está funcionando conforme o esperado e detecta regressões com antecedência
- Melhoram a qualidade geral e a confiabilidade do código
- Acham bugs antes de ir para produção
- Ajudam a garantir que o código seja sustentável e fácil de entender
- Documentam o código e servem como referência para desenvolvimento futuro

Testes estáticos

- Acham erros de sintaxe e tipo antes mesmo de executar seu código
- Enforçam um estilo de código consistente, tornando o código mais fácil de ler e entender
- Identificam áreas do seu código que podem estar sujeitas a erros e precisam de mais testes
- Certificam que seu código seja sustentável e escalável ao longo do tempo

Testes unitários

- Verificam a funcionalidade de um componente ou lógica específica
- Usados para garantir que o código está funcionando conforme o esperado
- São isolados do restante do aplicativo
- Focam em pedaços específicos do código

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react';

import Button from './Button';

test('Button component should render children and handle click', () => {
  const handleClick = jest.fn()
  const { getByText } = render(
    <Button onClick={handleClick}>
      Submit
    </Button>
  );

  const button = getByText('Submit');

  expect(button).toBeInTheDocument();

  fireEvent.click(button);

  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

Testes de integração

- Verificam a integração de diferentes componentes
- São usados para garantir que as diferentes partes da aplicação funcionem juntas conforme o esperado
- Podem incluir dependências externas para testar a aplicação
- Focam na integração de diferentes componentes

```
import React from 'react';
import { render, wait, fireEvent } from '@testing-library/react';
import fetchMock from 'fetch-mock';

import MyContainer from './MyContainer';

test('MyContainer', async () => {
  fetchMock
    .get('/api/greetings', { message: 'Hello John!' })
    .post('/api/consent', { message: 'Consent given!' });

  const { getByTestId, getByText } = render(
    <MyContainer />
  );

  await wait(() => expect(fetchMock.called('/api/greetings')).toBe(true));
  // Assert that the message comes from server.
  expect(getByText('Hello John!')).toBeInTheDocument();
  // Assert if Button exist and when clicked we show loading state
  fireEvent.click(getByText('Submit'));
  expect(getByText('Loading...')).toBeInTheDocument();
  // Assert that the message comes from server.
  await wait(() => expect(fetchMock.called('/api/consent')).toBe(true));
  // Assert that we render the message from the POST request.
  expect(getByText('Consent given!')).toBeInTheDocument();
});
```

Testes de ponta a ponta

- Verificam a funcionalidade de ponta a ponta de uma aplicação
- São usados para garantir que a aplicação esteja funcionando conforme o esperado do ponto de vista do usuário
- Envolvem a simulação das ações de um usuário real, como clicar em botões e preencher formulários
- Focam na funcionalidade geral da aplicação

```
describe('Contact Form', () => {
  beforeEach(() => {
    cy.visit('http://localhost:3000');
    cy.get('[data-testid="contact-form-link"]').click();
  });

  it('should submit the form, show a success message, and navigate to the homepage',
() => {
    cy.get('input[name="name"]').type('John Smith');
    cy.get('input[name="email"]').type('john@example.com');
    cy.get('textarea[name="message"]').type('Hello, I would like to know more about
your services.');
    cy.get('button[type="submit"]').click();
    cy.get('[data-testid="success-message"]').should('be.visible');
```

```
    cy.get('[data-testid="home-link"]').click();
    cy.title().should('include', 'My App');
  });

  it('should show an error message if the email is invalid', () => {
    cy.get('input[name="name"]').type('John Smith');
    cy.get('input[name="email"]').type('invalid-email');
    cy.get('textarea[name="message"]').type('Hello, I would like to know more about
your services.');
    cy.get('button[type="submit"]').click();
    cy.get('[data-testid="error-message"]').should('be.visible');
  });
});
```

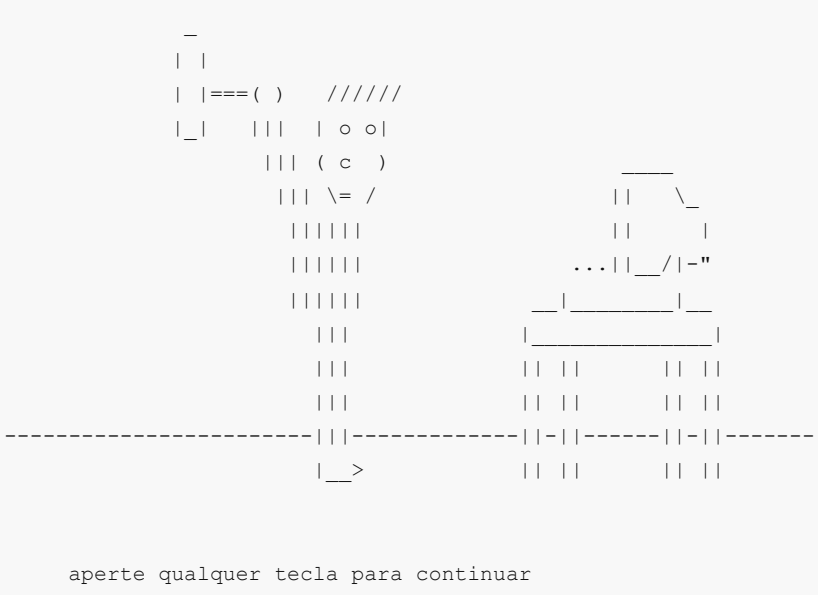
Distribuiçao de testes

- ~10%: Estático
- ~20%: Unitário
- ~65%: Integração
- ~05%: Ponta a Ponta

Revisão

- Estado nem sempre é necessário 🤷‍♂️
- Comece simples, adicione complexidade quando necessário 🤷‍♀️
- Separe as responsabilidades 🦄
- Não existe receita mágica 🪄
- Defina as regras do time 📖
- Teste sua aplicação 🟢

Demo



Obrigado

LinkedIn: Lucas Feliciano

Github: /lucasfeliciano

Twitter: @lucas_feliciano