Vamos remover as coisas feitas anteriormente no App.js e vamos usar um array e iterar com map:

  constructor(props) {

    super(props);

    this.state = {

      posts: [

        {

          id: 1,

          title: 'Título 1',

          body: 'Corpo 1'

        },

        {

          id: 2,

          title: 'Título 2',

          body: 'Corpo 2 '

        },

        {

          id: 3,

          title: 'Título 3',

          body: 'Corpo 3'

        },

      ]

    };

  }

  render() {

    const { posts } = this.state;

    return (

      <div className="App">

        {/\* { posts.map(post => <h1 key={post.id}>{post.title}</h1>)} \*/

          posts.map(post => (

            <div key={post.id}>

              <h1 >{post.title}</h1>

              <p>{post.body}</p>

            </div>

          ))

        }

      </div>

    );

  }

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>

Lifecycle methods (métodos de ciclo de vida)

Agora queremos que alguma coisa aconteça na tela assim que o componente terminar de ser montado.

Só vamos colocar dentro de um setTimeout para demorar um pouco para renderizar e vermos o resultado:

  componentDidMount() {

      setTimeout(() => {

        this.setState({

          posts: [

            {

              id: 1,

              title: 'Título 1',

              body: 'Corpo 1'

            },

            {

              id: 2,

              title: 'Título 2',

              body: 'Corpo 2 '

            },

            {

              id: 3,

              title: 'Título 3',

              body: 'Corpo 3'

            },

          ]

        })

      }, 5000);

  };

Depois vamos utilizar os métodos de ciclo de vida.

Primeiro declaramos um contador no state e depois utilizamos os metodos.

Para não criarmos um looping infinito, vamos limpar o timeout ou desmontar o componente.

Vamos criar uma função para setar o estado

  handleTimeout = () => {

    const { posts, counter } = this.state;

    posts[0].title = 'Título 1 - Alterado';

   this.timeoutUpdate = setTimeout(() => {

      this.setState({ posts, counter: counter + 1 });

    }, 1000);  };

Depois vamos chamar ela quando o componente terminar de ser montado:

  componentDidMount() {

    this.handleTimeout();

  }

E a cada vez que o componente terminar de ser montado, será atualizado pela função então vamos criar um looping infinito com o método componentDidUpdate

  componentDidUpdate(){

   // clearTimeout(this.timeoutUpdate)

    this.handleTimeout();

  }

E para não ficar lixo e não causar erro, vamos limpar o timeout quando o componente for desmontado:

  componentWillUnmount() {

    clearTimeout(this.timeoutUpdate);

  }

Todo código fica assim:

import './App.css';

import { Component } from 'react';

class App extends Component {

  constructor(props) {

    super(props);

    this.state = {

      counter: 0,

      posts: [

        {

          id: 1,

          title: 'Título 1',

          body: 'Corpo 1'

        },

        {

          id: 2,

          title: 'Título 2',

          body: 'Corpo 2 '

        },

        {

          id: 3,

          title: 'Título 3',

          body: 'Corpo 3'

        },

      ]

    };

  };

  timeoutUpdate = null;

  componentDidMount() {

    this.handleTimeout();

  }

  componentDidUpdate(){

   // clearTimeout(this.timeoutUpdate)

    this.handleTimeout();

  }

  componentWillUnmount() {

    clearTimeout(this.timeoutUpdate);

  }

  handleTimeout = () => {

    const { posts, counter } = this.state;

    posts[0].title = 'Título 1 - Alterado';

   this.timeoutUpdate = setTimeout(() => {

      this.setState({ posts, counter: counter + 1 });

    }, 1000);

  };

  render() {

    const { posts, counter } = this.state;

    return (

      <div className="App">

        <h1>{counter}</h1>

        {

          posts.map(post => (

            <div key={post.id}>

              <h1 >{post.title}</h1>

              <p>{post.body}</p>

            </div>

          ))

        }

      </div>

    );

  }

};

export default App;

Podemos ver os ciclos de vida aqui:

<https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

Aula: 19. Buscando dados externos com fetch (Data fetching)

Buscar dados de fora com fetch

Vamos usar uma api para testes chamada JSON Placeholder:

<https://jsonplaceholder.typicode.com/>

Para buscar dados externos, vamos utilizar o Javascript fetch api

<https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API>

    const postsResponse = fetch('https://jsonplaceholder.typicode.com/posts');

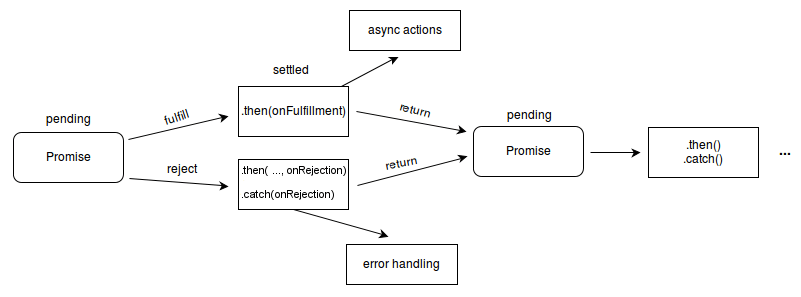
    const photosResponse = fetch('https://jsonplaceholder.typicode.com/photos');

    const [posts, photos] = await Promise.all([postsResponse, photosResponse]);

Uma promessa pode ser:

* **cumprida** (fulfilled) - a ação relativa à promessa foi bem-sucedida
* **rejeitada** (rejected) - a ação relativa à promessa falhou
* **pendente** (pending) - ainda não cumprida ou rejeitada
* **resolvida** (settled) - cumpriu ou rejeitou

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>



Vamos criar uma função loadPosts e chama-la no componentDidMount

 componentDidMount() {

    this.loadPosts();

  }

A função:

  loadPosts = async() => {

    const postsResponse = fetch('https://jsonplaceholder.typicode.com/posts');

    const [posts] = await Promise.all([postsResponse]);

    const postsJson = await posts.json();

    this.setState({ posts: postsJson });

  }

Vamos criar um css global no index.css

\* {

  margin: 0;

  padding: 0;

  box-sizing: border-box;

}

o border-box é necessário para que, ao incluirmos um padding em um elemento ele entre por dentro do tamanho e não aumente o tamanho de uma div:

<https://www.w3schools.com/css/css3_box-sizing.asp>

No App.css vamos colocar um css para a div externa a tudo com a classe posts .

- Falar que é para se comportar como grid

- Vamos dizer que é para repetir a todos elementos que é para ele auto preencher com no mínimo 280px e se não couber, poder usar um fragmento

.posts {

  display: grid;

  grid-template-columns: repeat(auto-fill, minmax(280px, 1fr));

};

Devido ao auto-fill as colunas irão se ajustando, só não teremos elementos menores que 280px.

Vamos colocar uma tag session com classe container por fora do nosso html e vamos retornar que a altura mínima será de 100vh, ou seja, 100% do viewport

o vh é um viewport-percentage length unit based on the browser default viewport size

1vh = 1% of viewport height.

.container {

  min-height: 100vh;

}

Exemplo

<https://www.w3schools.com/cssref/tryit.asp?filename=trycss_unit_vh>

O css vai ficando assim:

.container {

  min-height: 100vh;

  background-color: #eee;

}

.posts {

  display: grid;

  grid-template-columns: repeat(auto-fill, minmax(280px, 1fr));

  gap: 30px;

}

.post {

  background-color: #fff;

  border-radius: 5px;

  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

  /\* margin: 10px; \*/

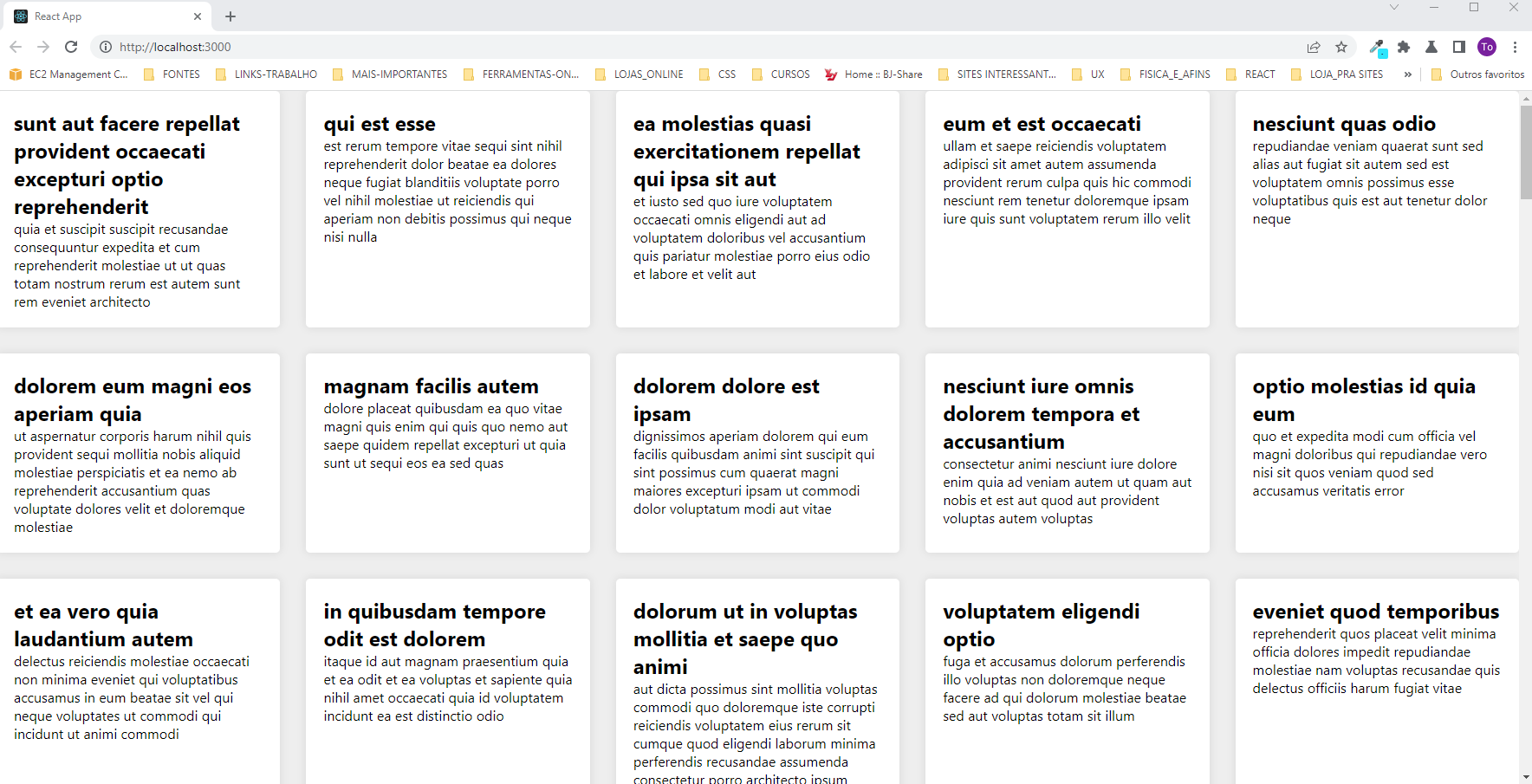
}

.post-content {

  padding: 20px;

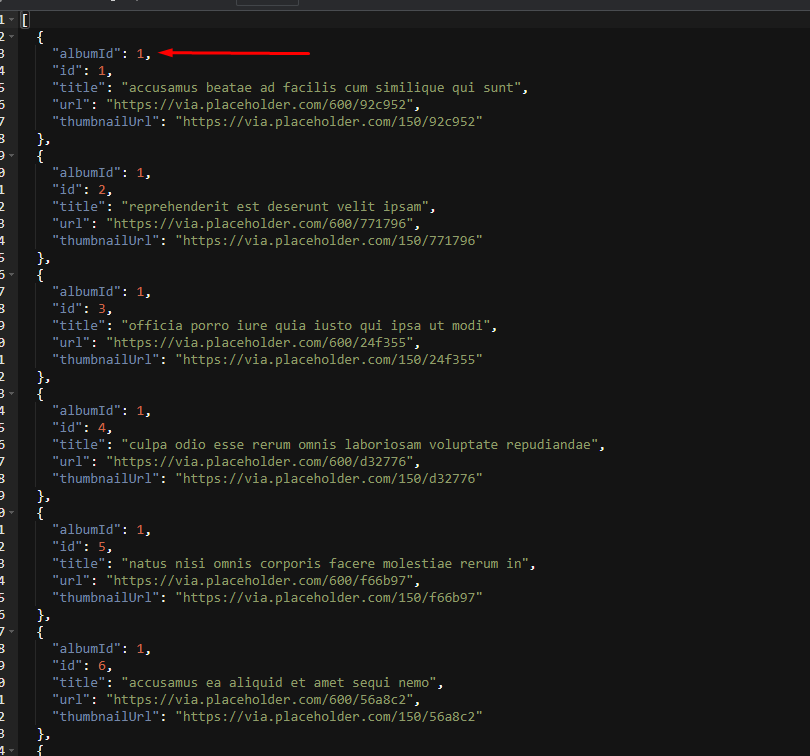
};

E o navegador totalmente responsivo assim:



Agora vamos utilizar as imagens.

Como tem muito mais imagens no endpoint de testes, vamos unir os arrays, apesar de que eu acharia melhor pegar a primeira de cada id ligando um post a cada albumId



Para unir a primeira imagem de cada objeto com o array de posts, vamos fazer o seguinte (função de zíper para unir dois arrays):

  loadPosts = async () => {

    const postsResponse = fetch('https://jsonplaceholder.typicode.com/posts')

    const photosResponse = fetch('https://jsonplaceholder.typicode.com/photos')

    const [posts, photos] = await Promise.all([postsResponse, photosResponse])

    const postsJson = await posts.json();

    const photoJson = await photos.json();

    const postsAndPhotos = postsJson.map((post, index) => {

      return { ...post, cover: photoJson[index].url }

    });

    this.setState({ posts: postsAndPhotos });

  }

A função map() vai retornar um novo array recebendo os posts com seus index então no return do map

retorna um objeto json de todos os posts combinados com o “cover” (nome qualquer) que pega o link da foto indicado no campo url

Depois só incrementar as fotos no array de posts:

  loadPosts = async () => {

    const postsResponse = fetch('https://jsonplaceholder.typicode.com/posts')

    const photosResponse = fetch('https://jsonplaceholder.typicode.com/photos')

    const [posts, photos] = await Promise.all([postsResponse, photosResponse])

    const postsJson = await posts.json();

    const photoJson = await photos.json();

    const postsAndPhotos = postsJson.map((post, index) => {

      return { ...post, cover: photoJson[index].url }

    });

    this.setState({ posts: postsAndPhotos });

  }

E o render com a imagem fica assim:

  render() {

    const { posts, counter } = this.state;

    return (

      <section className='container'>

        <div className="posts">

          {

            posts.map(post => (

              <div className="post">

                <img src={post.cover} alt={ post.title } />

                <div key={post.id} className="post-content">

                  <h1 >{post.title}</h1>

                  <p>{post.body}</p>

                </div>

              </div>

            ))

          }

        </div>

      </section>

    );

  }