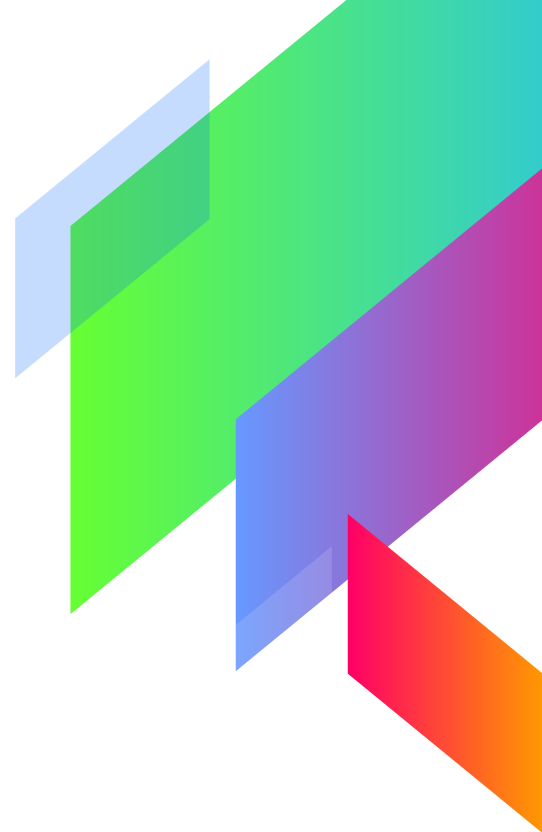




REACT.js

CLASE 5



The image features abstract geometric shapes in the corners. The top-left corner has a light blue parallelogram and a green parallelogram. The top-right corner has a light blue parallelogram, a green parallelogram, and a purple parallelogram. The bottom-left corner has a red parallelogram, an orange parallelogram, and a purple parallelogram. The bottom-right corner has a red parallelogram, an orange parallelogram, and a purple parallelogram.

The children prop



Se considera “**children**” a todo lo que se encuentre entre el tag de apertura y el tag de cierre de un componente

```
...  
<MiComponente>  
  <h1>Yo soy un hijo de MiComponente</h1>  
  <p>Y yo otro hijo</p>  
</MiComponente>  
...
```

En el ejemplo, “MiComponente” posee 2 hijos. El <h1> y el <p>

Un componente de React puede tener varios hijos, un hijo, o ningún hijo. Es decir, todo lo que se encuentre entre su tag de apertura y su tag de cierre, se considera hijo, y podremos acceder desde **this.props.children**

MiComponente.js

```
...  
class MiComponente extends Component {  
  render() {  
    return (  
      <div className="mi-componente-wrapper">  
        {this.props.children}  
      </div>  
    )  
  }  
}  
...
```

El mismo componente stateless sería...

```
const MiComponente = (props) => (  
  <div className="mi-componente-wrapper">  
    {props.children}  
  </div>  
)
```

Y destrutturando las props...

```
const MiComponente = ({ children }) => (  
  <div className="mi-componente-wrapper">  
    {children}  
  </div>  
)
```

Ejemplo pasando más de un hijo a "MiComponente"...

```
...  
<MiComponente>  
  <h1>Yo soy un hijo de MiComponente</h1>  
  <p>Y yo otro hijo</p>  
  <OtroCompoente algunaProp={10} />  
</MiComponente>  
...
```

Ejemplo pasando sólo un hijo a “MiComponente”...

```
...  
<MiComponente>  
  <h1>Yo soy el único hijo de MiComponente</h1>  
</MiComponente>  
...
```


Ejemplo de “MiComponente” sin pasarle hijos...

```
...  
<MiComponente />  
...
```

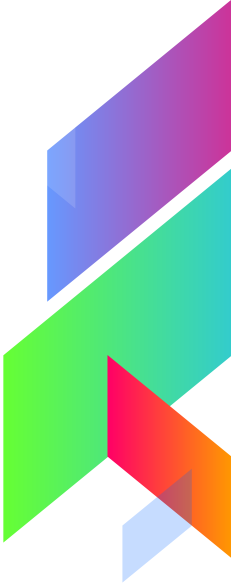


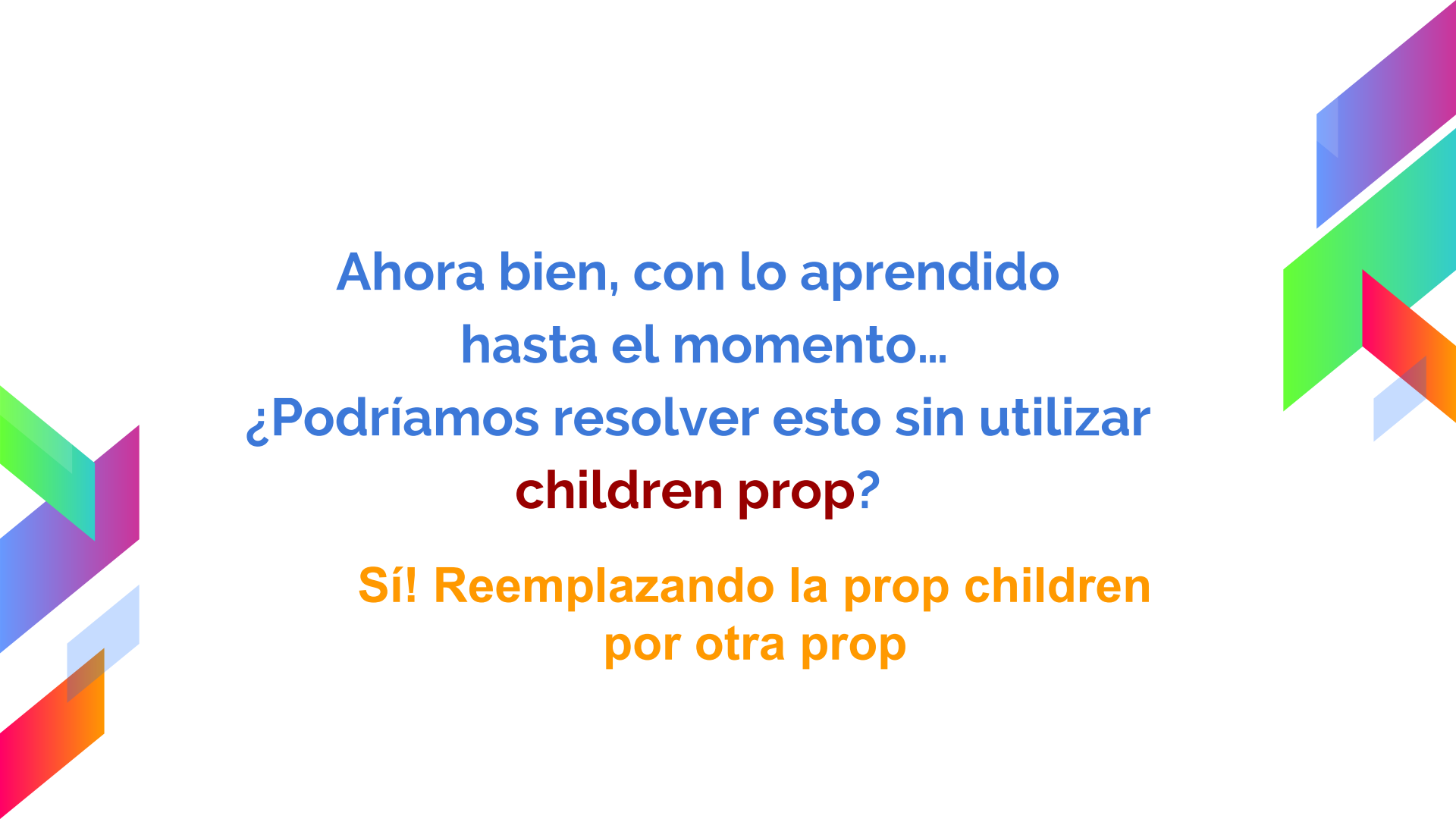

Vemos un ejemplo en pantalla?





Ahora bien, con lo aprendido
hasta el momento...
¿Podríamos resolver esto sin utilizar
children prop?



The slide features decorative geometric shapes in the corners. On the left, there are overlapping triangles in green, purple, blue, and orange. On the right, there are overlapping triangles in purple, green, red, and blue.

Ahora bien, con lo aprendido
hasta el momento...
¿Podríamos resolver esto sin utilizar
children prop?

**Sí! Reemplazando la prop children
por otra prop**

Cuando utilicemos “MiComponente” sin children...

```
...  
<MiComponente  
  content={<h1>Yo soy el único hijo de MiComponente</h1>}  
>  
...
```

Ejemplo de “MiComponente” con la prop content en lugar de children...

```
const MiComponente = ({ content }) => (  
  <div className="mi-componente-wrapper">  
    {content}  
  </div>  
)
```

La desventaja la tendremos cuando necesitemos tener más de un hijo, y tengamos que utilizar un “wrapper” para pasar a todos los hijos en la prop content

```
...  
<MiComponente  
  content={<div>  
    <h1>Yo soy un hijo de MiComponente</h1>  
    <p>Y yo otro hijo</p>  
    <OtroCompoente algunaProp={10} />  
  </div>}  
</>  
...
```

Retomando los beneficios...

El uso de la **children** prop nos permite crear “Interfaces” amigables para reutilizar bloques de código.

Por ejemplo, si creamos un componente “Alert” que utilice la gráfica del alert de Bootstrap...

```
const Alert = ({ children }) => (  
  <div className="alert alert-primary">  
    {children}  
  </div>  
)
```

Y el uso sería...

```
...  
<Alert>Soy un alert de Bootstrap!!!</Alert>  
...
```

En resumen, la children prop nos permite pasar elementos y componentes como hijos, anidándolos tal cual anidabamos los elementos html comunes y sin necesidad de contenerlos dentro de un wrapper.





Ejercicio 1

Tiempo de práctica

The image features a white background with decorative geometric shapes in the corners. These shapes are composed of overlapping translucent polygons in various colors: light blue, green, teal, purple, magenta, red, orange, and brown. The shapes are arranged in a way that they appear to be floating or layered, creating a modern, abstract aesthetic.

React.Children.map

En caso de que necesitemos recorrer y manipular los hijos de un componente dentro del método render, contamos con el método `React.Children.map` para iterarlos y manipularlos



Por ejemplo, si tuviésemos un componente que sea un listado de links y que se utilizara de esta forma...

```
<ListadoLinks>  
  <a href="https://www.digitalhouse.com/">Digital House</a>  
  <a href="https://www.google.com.ar/">Google</a>  
  <a href="https://es-la.facebook.com/">Facebook</a>  
</ListadoLinks>
```

Podríamos hacer que se renderice como una lista `` y que a cada hijo, lo inserte dentro de un ``

ListadoLinks.js

```
const ListadoLinks = ({ children }) => (  
  <ul className="listado-links">  
    {React.Children.map(children, child => (  
      <li className="listado-links-item">{child}</li>  
    ))}  
  </ul>  
);
```

También tenemos la posibilidad de recorrer los hijos y cambiarle props...

ListadoLinks.js

```
const ListadoLinks = ({ children, selectedHref }) => (  
  <ul className="listado-links">  
    {React.Children.map(children, child => (  
      <li className="listado-links-item">  
        {React.cloneElement(child, {  
          className: child.props.href === selectedHref ? 'active' : null  
        })}  
      </li>  
    ))}  
  </ul>  
);
```

También tenemos la posibilidad de recorrer los hijos y cambiarle props...

ListadoLinks.js

```
const ListadoLinks = ({ children, selectedHref }) => (  
  <ul className="listado-links">  
    {React.Children.map(children, child => (  
      <li className="listado-links-item">  
        {React.cloneElement(child, {  
          className: child.props.href === selectedHref ? 'active' : null  
        })}  
      </li>  
    ))}  
  </ul>  
);
```

Clonamos el hijo que estamos recorriendo

También tenemos la posibilidad de recorrer los hijos y cambiarle props...

ListadoLinks.js

```
const ListadoLinks = ({ children, selectedHref }) => (  
  <ul className="listado-links">  
    {React.Children.map(children, child => (  
      <li className="listado-links-item">  
        {React.cloneElement(child, {  
          className: child.props.href === selectedHref ? 'active' : null  
        })}  
      </li>  
    ))}  
  </ul>  
);
```

Y como segundo parámetro de React.cloneElement pasamos un objeto literal con las props que queremos cambiarle o crearle al hijo

También tenemos la posibilidad de recorrer los hijos y cambiarle props...

ListadoLinks.js

```
const ListadoLinks = ({ children, selectedHref }) => (  
  <ul className="listado-links">  
    {React.Children.map(children, child => (  
      <li className="listado-links-item">  
        {React.cloneElement(child, {  
          className: child.props.href === selectedHref ? 'active' : null  
        })}  
      </li>  
    ))}  
  </ul>  
);
```

Si la prop "href" del hijo es igual a la prop "selectedHref" del padre, le pondremos la clase de CSS "active" al hijo

El uso de nuestro componente quedaría así...

```
<ListadoLinks selectedHref="https://www.digitalhouse.com/">
  <a href="https://www.digitalhouse.com/">Digital House</a>
  <a href="https://www.google.com.ar/">Google</a>
  <a href="https://es-la.facebook.com/">Facebook</a>
</ListadoLinks>
```

Y el html generado será...

```
<ul class="listado-links">
  <li class="listado-links-item">
    <a href="https://www.digitalhouse.com/" class="active">Digital House</a>
  </li>
  <li class="listado-links-item">
    <a href="https://www.google.com.ar/">Google</a>
  </li>
  <li class="listado-links-item">
    <a href="https://es-la.facebook.com/">Facebook</a>
  </li>
</ul>
```






Ejercicio 2

Tiempo de práctica

Abstract geometric shapes in the top-left corner, including a green parallelogram, a light blue parallelogram, a brown parallelogram, a red parallelogram, and a purple parallelogram.

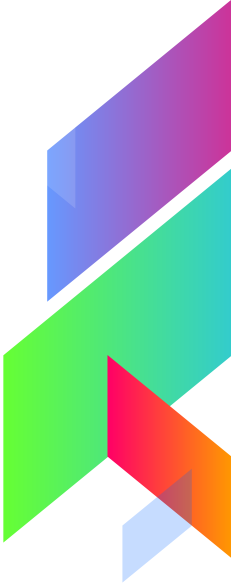
API

Abstract geometric shapes in the top-right corner, including a light blue parallelogram, a green parallelogram, a purple parallelogram, and a red parallelogram.



A la hora de utilizar una api externa, a la cual haremos varias llamadas distintas, y que necesitará de alguna configuración inicial, es buena práctica crear una clase que se encargue de todo esto.

De esta forma, nos permite encapsular la manera de llamar al servicio externo y su configuración, y en caso de que algo cambie, solamente realizaremos el refactor en la clase de la api, y no en todos los lugares donde la utilizamos.



Encapsular, componentizar, reutilizar...



```
import axios from 'axios'

const API_KEY = 'nuestra-api-key';

class TheMovieDbApi {
  constructor() {
    this.axios = axios.create({
      baseURL: 'https://api.themoviedb.org/3',
      params: {
        api_key: API_KEY,
        language: 'es-AR',
      },
    })
  }

  getPopularMovies = (page = 1) => (
    this.axios.get(`/movie/popular`, {
      params: {page: page}
    })
  );
}

Export default TheMovieDbApi;
```



```
import axios from 'axios'
```

```
const API_KEY = 'nuestra-api-key';
```

```
class TheMovieDbApi {  
  constructor() {  
    this.axios = axios.create({  
      baseURL: 'https://api.themoviedb.org/3',  
      params: {  
        api_key: API_KEY,  
        language: 'es-AR',  
      },  
    })  
  }  
  
  getPopularMovies = (page = 1) => (  
    this.axios.get(`/movie/popular`, {  
      params: {page: page}  
    })  
  );  
}
```

```
Export default TheMovieDbApi;
```

Seteamos nuestra API_KEY en una constante



```
import axios from 'axios'
```

```
const API_KEY = 'nuestra-api-key';
```

```
class TheMovieDbApi {
```

```
  constructor() {
```

```
    this.axios = axios.create({  
      baseUrl: 'https://api.themoviedb.org/3',
```

```
      params: {  
        api_key: API_KEY,  
        language: 'es-AR',
```

```
      },
```

```
    })
```

```
  }
```

```
  getPopularMovies = (page = 1) => (  
    this.axios.get(`/movie/popular`, {  
      params: {page: page}
```

```
    })
```

```
  );
```

```
}
```

```
Export default TheMovieDbApi;
```

Creamos una instancia de axios con una configuración custom

```
import axios from 'axios'

const API_KEY = 'nuestra-api-key';

class TheMovieDbApi {
  constructor() {
    this.axios = axios.create({
      baseUrl: 'https://api.themoviedb.org/3',
      params: {
        api_key: API_KEY,
        language: 'es-AR',
      },
    })
  }

  getPopularMovies = (page = 1) => (
    this.axios.get(`/movie/popular`, {
      params: {page: page}
    })
  );
}
```

Todas las llamadas comenzarán
con esta URL

```
Export default TheMovieDbApi;
```

```
import axios from 'axios'


const API_KEY = 'nuestra-api-key';

class TheMovieDbApi {
  constructor() {
    this.axios = axios.create({
      baseURL: 'https://api.themoviedb.org/3',
      params: {
        api_key: API_KEY,
        language: 'es-AR',
      },
    })
  }

  getPopularMovies = (page = 1) => (
    this.axios.get(`/movie/popular`, {
      params: {page: page}
    })
  );
}
```

```
Export default TheMovieDbApi;
```

Y a todas las llamadas le pasaremos estos parámetros por default, para evitar escribirlos en cada una



```
import axios from 'axios'

const API_KEY = 'nuestra-api-key';

class TheMovieDbApi {
  constructor() {
    this.axios = axios.create({
      baseURL: 'https://api.themoviedb.org/3',
      params: {
        api_key: API_KEY,
        language: 'es-AR',
      },
    })
  }

  getPopularMovies = (page = 1) => (
    this.axios.get(`/movie/popular`, {
      params: {page: page}
    })
  );
}

export default TheMovieDbApi;
```

De esta manera obtendremos las películas más populares llamando a la api de TMDb

El uso en nuestro componente quedaría así...

```
...  
class App extends Component {  
  constructor() {  
    super()  
    this.api = new TheMovieDbApi()  
  }  
  
  componentDidMount() {  
    this.api.getPopularMovies().then(res => {  
      console.log(res.data.results)  
    })  
  }  
  
  render() {  
    return (  
      <div>Nuestro Render</div>  
    )  
  }  
}  
...
```

El uso en nuestro componente quedaría así...

```
...  
class App extends Component {  
  constructor() {  
    super()  
    this.api = new TheMovieDbApi()  
  }  
  
  componentDidMount() {  
    this.api.getPopularMovies().then(res => {  
      console.log(res.data.results)  
    })  
  }  
  
  render() {  
    return (  
      <div>Nuestro Render</div>  
    )  
  }  
}  
...
```

Instanciamos nuestra clase de Api

El uso en nuestro componente quedaría así...

```
...  
class App extends Component {  
  constructor() {  
    super()  
    this.api = new TheMovieDbApi()  
  }  
  
  componentDidMount() {  
    this.api.getPopularMovies().then(res => {  
      console.log(res.data.results)  
    })  
  }  
  
  render() {  
    return (  
      <div>Nuestro Render</div>  
    )  
  }  
}  
...
```

Y al montarse el componente,
realizo el llamado a la API

Cuando trabajamos con cambios asincrónicos en nuestros datos, como el uso de llamados AJAX a una API externa, nos cruzamos con algunos estados del componente que antes no necesitábamos

- Loading... (mientras se realiza la consulta AJAX)
- No se encontraron resultados
- Ocurrió un error en la consulta

Todos estos estados, tienen que ser comunicados al usuario mediante interfaces gráficas e irán mostrándose u ocultándose dependiendo de la respuesta que nos dé el llamado a la API

```
class App extends Component {  
  constructor() {  
    super();  
    this.state = {  
      loading: true,  
      items: [],  
      error: null  
    };  
    this.api = new TheMovieDbApi()  
  }  
  
  componentDidMount() {...}  
  
  render() {...}  
}
```

```
class App extends Component {
  constructor() {...}

  componentDidMount() {...}

  render() {
    return (
      <div>
        {this.state.loading && <div>Loading...</div>}
        {!!this.state.error && <div>{this.state.error}</div>}
        {!!this.state.items.length && <div>Muestro los items...</div>}
      </div>
    )
  }
}
```

```
class App extends Component {
  constructor() {...}

  componentDidMount() {
    this.setState({ loading: true, error: null });
    this.api.getPopularMovies().then(res => {
      this.setState({
        loading: false,
        error: null,
        items: res.data.results
      });
    }).catch((error) => {
      this.setState({ loading: false, error: error, items: [] });
    })
  }

  render() {...}
}
```

```
class App extends Component {  
  constructor() {...}  
  
  componentDidMount() {  
    this.setState({ loading: true, error: null });  
    this.api.getPopularMovies().then(res => {  
      this.setState({  
        loading: false,  
        error: null,  
        items: res.data.results  
      });  
    }).catch((error) => {  
      this.setState({ loading: false, error: error, items: []});  
    })  
  }  
  
  render() {...}  
}
```

Hacemos que se muestre
el loading y reseteamos
el error

```
class App extends Component {  
  constructor() {...}
```

```
  componentDidMount() {  
    this.setState({ loading: true, error: null });  
    this.api.getPopularMovies().then(res => {
```

```
      this.setState({  
        loading: false,  
        error: null,  
        items: res.data.results  
      });
```

```
    }).catch((error) => {  
      this.setState({ loading: false, error: error, items: []});  
    })  
  }
```

```
  render() {...}  
}
```

En caso de que la API nos responda 200, ocultamos el loading, reseteamos el error y llenamos los ítems

```
class App extends Component {
  constructor() {...}

  componentDidMount() {
    this.setState({ loading: true, error: null });
    this.api.getPopularMovies().then(res => {
      this.setState({
        loading: false,
        error: null,
        items: res.data.results
      });
    }).catch((error) => {
      this.setState({ loading: false, error: error, items: []});
    })
  }

  render() {...}
}
```

En caso de error, oculto el
loading y muestro el mensaje
de error



Ejercicio 3

Tiempo de práctica



Gracias!

¿Preguntas?

