

Xseed

Curso de React _ Technisys

Clase 06 -

Lifecycle Methods, External Data Access, Presentational vs Container


by Diego Cáceres

Repaso

PropTypes (1)

```
class Users extends Component {  
  render() {  
    return (  
      <ul>{this.props.list.map((friend) => {  
        return <li>{friend}</li>  
      })}  
    </ul>  
  )  
}
```

Este componente espera una lista de nombres y realiza un `map` para mostrarlos como una lista no ordenada (``).



¿Qué pasaría si en vez de pasarle una lista en la **prop** `list`, se le pasa un string?

```
<Users list="Diego, Juan, Pedro" />
```

PropTypes (3)

El ejemplo visto anteriormente, declarando las **PropTypes** sería así:

```
import React, { Component } from "react";
import PropTypes from 'prop-types';
class Users extends Component {
  render() {
    return (
      <ul> {this.props.list.map((friend) => {
        return <li>{friend}</li>
      })} </ul>
    )
  }
}
Users.propTypes = {
  list: PropTypes.array.isRequired
}
```

Default Prop Values – Ejemplo

```
class Greeting extends React.Component {  
  render() {  
    return (  
      <h1>Hello, {this.props.name}</h1>  
    );  
  }  
}
```

// El valor por defecto:

```
Greeting.defaultProps = {  
  name: 'Stranger'  
};  
  
Greeting.propTypes = {  
  name: PropTypes.string  
};
```

Forms – Controlled Components

Para lograr esto en React se utiliza una técnica llamada **Controlled Components**.

Normalmente en HTML los elementos de formulario como `<input>`, `<textarea>` y `<select>` mantienen su propio estado y lo actualizan en función al `input` del usuario.

Esta técnica consiste básicamente en mantener al **state** de React como la **fuentes única de verdad**, por lo que es React quien se encarga de mantener el estado de estos elementos, y además de actualizarlo en función al `input` del usuario.

En la siguiente slide se verá el código anterior pero aplicando estos conceptos.

Forms – Controlled Components

```
class NameForm extends React.Component {
  constructor(props) {
    super(props); this.state = { value: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) { this.setState({ value: event.target.value }); }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value); event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label> Name:
        <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```


Forms – Manejando múltiples inputs (1)

En el caso de tener que manejar varios inputs en un componente, no es necesario crear un handler para cada uno de ellos, sino que se puede aprovechar el atributo `name`, asignándole un `name` distinto a cada elemento, y luego en el handler decidir qué actualizar en función del `event.target.name`.

```
handleInputChange(event) {  
  const target = event.target;  
  const value = target.type === 'checkbox' ? target.checked : target.value;  
  const name = target.name;  
  
  this.setState({  
    [name]: value  
  });  
}
```

Lifecycle Methods

Lifecycle Methods

Todos los componentes en React tienen un **ciclo de vida**, empezando cuando son **montados** en pantalla, cuando son **actualizados**, y cuando se **desmontan**.

Una de las diferencias entre los Class Components y los Functional Components es que en Class Components se puede sobrescribir los métodos de ciclo de vida para programar acciones que suceden en cada uno de ellos.

Esto permite, por ejemplo, ir a buscar datos a una API exactamente después de que se dibuja por primera vez el componente.

Lifecycle Methods

El equipo de React está realizando algunos cambios importantes en las nuevas versiones de la API de React (v16.3), entre ellos, 3 métodos de ciclo de vida serán gradualmente eliminados:

- `componentWillMount`
- `componentWillReceiveProps`
- `componentWillUpdate`

El motivo principal, es que se prestan fácilmente para prácticas incorrectas, y que sumados a la nueva funcionalidad que liberaran en un futuro “Async Rendering”, podrían ocasionar mayores problemas. [Blog Post](#)

Lifecycle Methods – Mounting (1)

Estos métodos se llaman cuando se crea una instancia de un componente y se inserta en el DOM:

- `constructor()`
- `componentWillMount()` – Invocado una única vez
(Será deprecated en próximas versiones)
- `render()`
- `componentDidMount()` – Invocado una única vez

Lifecycle Methods – Mounting (2)

Algunas de las cosas que se pueden resolver en estos métodos pueden ser:

- Declarar algunas **Props** por defecto.
- **Setear** el estado inicial del componente (state).
- Realizar llamadas **Ajax** para obtener determinados datos necesarios para el componente.
- Setear ***listeners*** (Ej: Websockets o Firebase).

Lifecycle Methods – Updating

Un `update` puede ser causado por cambios en **props** o en el **state** del componente. Estos métodos se llaman cuando un componente se vuelve a renderizar, pero no son llamados en el render inicial:

- `componentWillReceiveProps (nextProps)`
(Será deprecated en próximas versiones)
- `shouldComponentUpdate (nextProps, nextState)`
- `componentWillUpdate (nextProps, nextState)`
(Sera deprecated en próximas versiones)
- `render ()` – Método obligatorio definirlo.
- `componentDidUpdate (prevProps, prevState)`

Lifecycle Methods – Unmounting

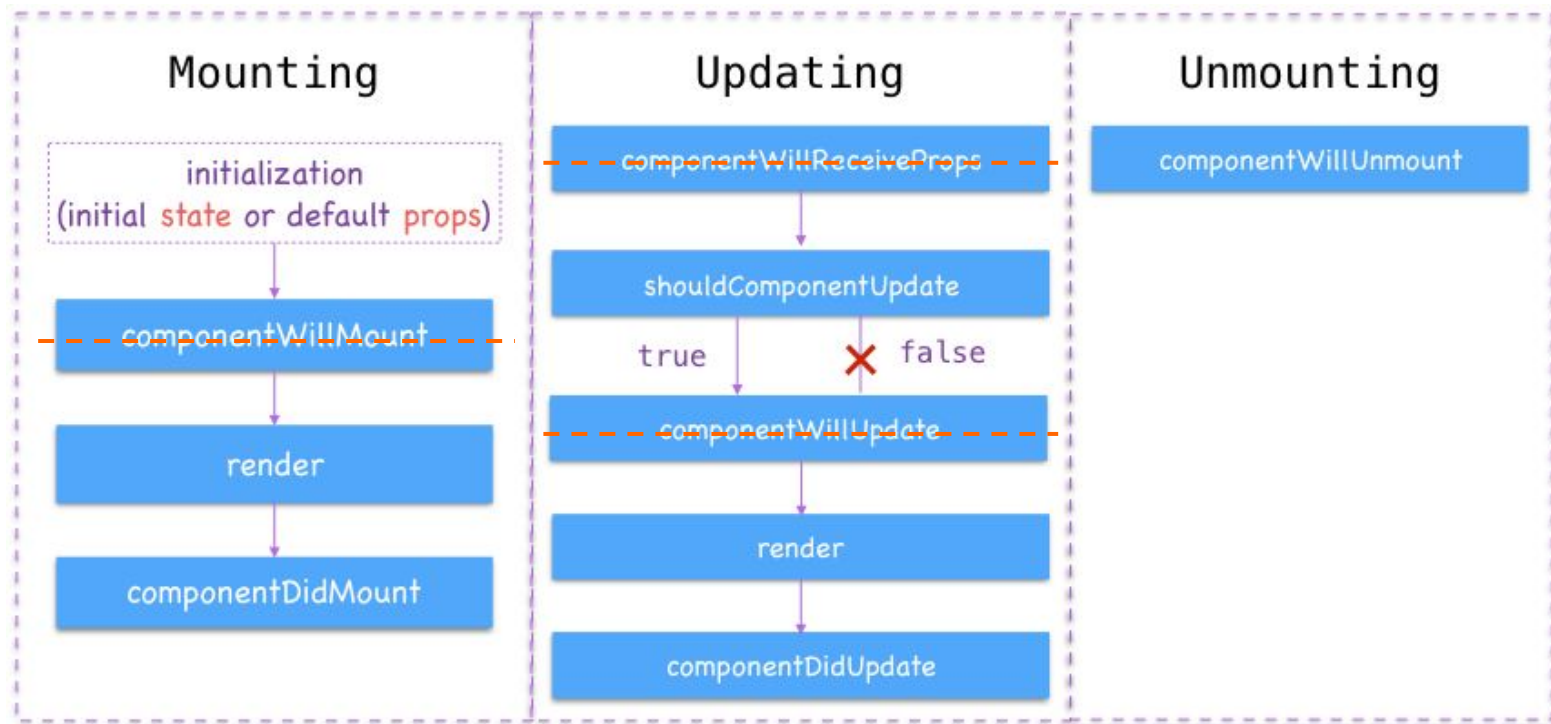
Este método es llamado cuando un componente se elimina del DOM:

- `componentWillUnmount()`

Por lo general, este método es utilizado para remover *listeners* cuando el componente se **desmonta**, aunque se le podrían encontrar otros usos.

Lifecycle Methods

Se muestran a continuación algunos diagramas para entender esto.



----- (Serán deprecated en próximas versiones)

Lifecycle Methods – Ejemplo

```
class MyComponent extends Component {  
  constructor(props) {  
    super(props);  
  }  
  componentDidMount() {  
    console.log("Se acaba de montar el componente");  
  }  
  componentWillUnmount() {  
    console.log("Se acaba de des-montar el componente");  
  }  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
      </div>  
    );  
  }  
}
```

Gradual Migrating Path

- 16.3: Introduce aliases para los métodos inseguros: `UNSAFE_componentWillMount`, `UNSAFE_componentWillReceiveProps` y `UNSAFE_componentWillUpdate` (Tanto los nombres anteriores como los nuevos aliases funcionan en esta versión)
- Una futura liberacion 16.x: Se habilitará una advertencia para `componentWillMount`, `componentWillReceiveProps` y `componentWillUpdate` (Tanto los nombres anteriores como los nuevos aliases funcionan en esta versión)
- 17.0: Se eliminarán `componentWillMount`, `componentWillReceiveProps` y `componentWillUpdate` (Sólo los nuevos comenzando en `UNSAFE_` funcionarán.)

Lifecycle Methods – A partir de v16.3 de React

Además de desincentivar el uso de estos 3 métodos de ciclo de vida, a partir de React v16.3 (29/03/2018) contamos con dos nuevos métodos:

- `getDerivedStateFromProps (nextProps, prevState)`
- `getSnapshotBeforeUpdate`

getDerivedStateFromProps

El nuevo método estático es invocado luego de que un componente es instanciado, y también cuando recibe nuevas `props`. Puede retornar un objeto para actualizar el `state`, o `null` para indicar que las nuevas `props` no requieren actualizar el `state`.

En conjunto con `componentDidUpdate` este método debería cubrir todos los usos de `componentWillReceiveProps`

```
class Example extends React.Component {  
  static getDerivedStateFromProps(nextProps, prevState) {  
    // ...  
  }  
}
```

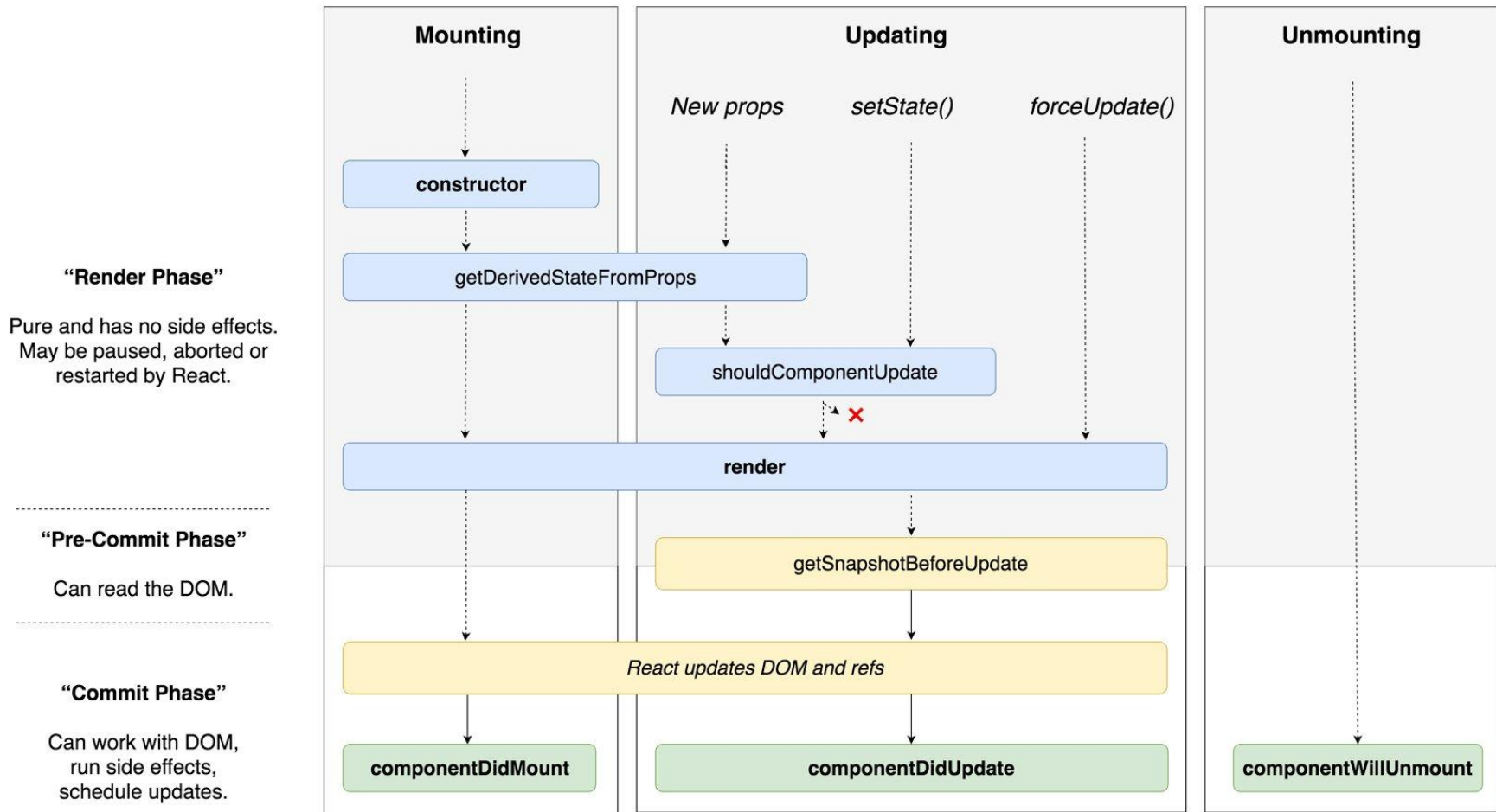
getSnapshotBeforeUpdate

Este nuevo método es llamado inmediatamente antes de que las mutaciones se realicen (ejemplo, antes que el DOM sea actualizado). Lo que este método retorna, será enviado como tercer parámetro a `componentDidUpdate`

(Este método de ciclo de vida no es usualmente utilizado, pero puede ser útil en casos como manualmente conservar la posición de scroll del usuario cuando la pantalla se vuelve a dibujar)

En conjunto con `componentDidUpdate` este método debería cubrir todos los usos de `componentWillUpdate`.

```
class Example extends React.Component {  
  getSnapshotBeforeUpdate(prevProps, prevState) {  
    // ...  
  }  
}
```



External Data Access

External Data Access

Existen varias opciones en JavaScript para resolver el acceso a datos externos.

En el curso se verá cómo utilizar la librería **Axios** que es una de las más populares.

En la [documentación de GitHub](#) se puede encontrar cómo utilizarla en la mayoría de los casos, pero aquí se verán las formas más simples.

Lo primero que se necesita es instalarla en el proyecto de React:

- `npm install --save axios`

External Data Access – Promises

Axios es una librería basada en las **Promises** de JavaScript.

La filosofía de las promesas en JavaScript está pensada para poder ejecutar código de forma **asincrónica** y poder controlar qué se ejecuta al finalizar el código asincrónico. Tienen una simple regla y es que las promesas **siempre** se van a **resolver** (éxito) o a **fallar**.

Esto quiere decir que siempre se puede esperar el resultado en la función `then` de una promesa, o los errores en la función `catch`.

Nota: Antes de existir las promesas para este tipo de cosas en JavaScript se utilizaban los **callbacks**.

External Data Access – Promises

```
var p = new Promise(function(resolve, reject) {
```

```
    // Código async, llamar a una API.
```

```
    if (/* No hubo errores */) {
```

```
        resolve('Success!');
```

```
    }
```

```
    else {
```

```
        reject('Failure!');
```

```
    }
```

```
});
```

```
p.then(function(result) {
```

```
    console.log(result); /* Success! */
```

```
}, function(err) {
```

```
    console.log(err); /* Failure! */
```

```
})
```

```
.catch(error){ /* Fallo */ }
```

Sería muy raro que se utilice el `new Promise`. Normalmente, se utilizarán APIs basadas en promesas.

External Data Access – Axios GET

Como Axios está basado en Promesas, se pueden realizar llamadas GET y programar qué hacer con el resultado en `then`, sin necesidad de utilizar callbacks.

```
axios.get('/user?ID=12345')  
  .then(function (response) {  
    return axios.get('/comment')  
  })  
  .catch(function (error) {  
    console.log(error);  
  });
```

```
axios.get('/user', {  
  params: {  
    ID: 12345  
  }  
})  
  .then(function (response) {  
    console.log(response);  
  })  
  .catch(function (error) {  
    console.log(error);  
  });
```

También se pueden pasar los parámetros como un objeto.

External Data Access – Axios POST

Para hacer una llamada de tipo `POST` simplemente se manda un segundo parámetro conteniendo un objeto con lo que se quiere que contenga el `body` de la llamada.

```
axios.post('/user', {  
  firstName: 'Diego',  
  lastName: 'Caceres'  
})  
.then(function (response) {  
  console.log(response);  
})  
.catch(function (error) {  
  console.log(error);  
});
```

```
const baseUrl = "https://myapp.com/api";  
axios.post(`${baseUrl}/user`, {  
  firstName: 'Diego',  
  lastName: 'Caceres'  
})  
.then(function (response) {  
  console.log(response);  
})  
.catch(function (error) {  
  console.log(error);  
});
```

Pasando la URL completa.

External Data Access – Axios

Para utilizar **Axios** en React, luego de instalarla simplemente se debe importar la librería en el archivo donde se vaya a utilizar.

```
import React from "react";
import axios from "axios";
class App extends React.Component {
  // Constructor
  componentDidMount() {
    axios.get("https://api.github.com/users").then(result => {
      this.setState({
        users: result.data
      });
    });
  }
  render() {
    // Render.
  }
}
```

Arrow Function. Se puede utilizar una función común pero habría que guardar `this` en una variable para acceder desde dentro del nuevo contexto de la función.

Presentational – Container

Presentational – Container (1)

Es un **patrón** muy utilizado en React ya que facilita la reutilización de componentes. También es llamado patrón **Container Component**.

La idea es bastante sencilla; un contenedor se encarga de obtener los datos y luego renderiza su correspondiente sub-componente, llamado **Presentational Component** (Componente de Presentación).

Normalmente, para nombrarlos se mantiene parte del nombre:

- `StockWidgetContainer => StockWidget`
- `TagCloudContainer => TagCloud`

Presentational – Container (2)

A continuación se muestra un componente que muestra comentarios sin aplicar el patrón.

```
class CommentList extends React.Component {  
  // constructor con this.state = { comments: [] }  
  componentDidMount() {  
    fetchSomeComments(comments => this.setState({ comments: comments }));  
  }  
  render() {  
    return (  
      <ul>  
        {this.state.comments.map(c => (  
          <li>{c.body}-{c.author}</li>  
        ))}  
      </ul>  
    );  
  }  
}
```

Presentational – Container (3)

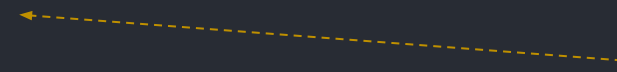
El componente es correcto y funciona, pero se pierden algunos beneficios de React.

- **Reusabilidad:** `CommentList` solo puede ser utilizado bajo las mismas circunstancias.
- **Estructura de datos:** En el `render` se utiliza una estructura para cada comentario, que idealmente se tendría que “exigir”, para lo que se puede utilizar **Prop Types**, pero en este caso no se puede. Si el *endpoint* cambia, esto va a fallar silenciosamente.

Presentational – Container (4)

El mismo ejemplo anterior pero aplicando el patrón se vería así.

```
class CommentListContainer extends React.Component {  
  // constructor con this.state = { comments: [] }  
  componentDidMount() {  
    fetchSomeComments(comments =>  
      this.setState({ comments: comments }));  
  }  
  render() {  
    return <CommentList comments={this.state.comments} />;  
  }  
}
```

```
const CommentList = props =>   
  <ul>  
    {props.comments.map(c => <li>{c.body}-{c.author}</li>)}  
  </ul>;
```

Ahora existe la posibilidad de declarar PropTypes y que si la API cambia, el error sea más visible.

Ejercicios

Ejercicios

Descargar el [zip](#) EjerciciosClase06. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
 - `$ yarn install`
 - `$ yarn start`
- Si usan npm:
 - `$ npm install`
 - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en <http://localhost:3000/>

Ejercicio 1

Editar el archivo `src/App.js`. Utilizando la API de usuarios de GitHub

(<https://api.github.com/users>) implementar dos componentes: `UserListContainer` y

`UserList` para mostrar los usuarios que devuelve la API.

- Se puede probar la llamada a la API en el navegador, ya que es un `GET`, para ver la estructura de los datos que devuelve.
- Básicamente, de cada usuario se necesita mostrar la **imagen**, el **Id**, y el **nombre** de login.

Ejercicio 2 (1)

Este ejercicio parte de la solución de un ejercicio anterior, donde se comparan dos perfiles de GitHub introduciendo el username, pero con algunas variaciones.

- Se necesita implementar el componente `src/components/ProfileDetails` para que a partir del username que recibe por props, y utilizando la API de GitHub, muestre los detalles de cada perfil. (<https://api.github.com/users/:username>)
- Luego de implementarlo, notar que si la llamada a **Axios** está solo en el método `componentDidMount`, si se realiza una nueva búsqueda los datos no cambian (ya que `componentDidMount` solo se ejecuta una vez en el ciclo de vida). Identificar qué método del ciclo de vida implementar para detectar cuando cambia el username y volver a obtener los datos.

Ejercicio 2 (2)

- Luego de completar la primer parte, separar el componente `ProfileDetails` en un componente **Container** y uno **Presentational**:
`ProfileDetailsContainer` y `ProfileDetails`.
- Definir **PropTypes** en `ProfileDetails`.
- Definir **Default Types** en `ProfileDetails` para cuando reciba alguna de las Props.

Ejercicio 3

Deben completar el ejercicio para poder tener un ABM de Productos.

- En `App.js` se maneja el estado de los productos.
- En `components/ProductList.js` se muestra la lista de componentes, y también se puede seleccionar uno para modificarlo, o borrar uno.
- En `components/ProductABM.js` se muestra el formulario, y se puede agregar o modificar un producto seleccionado.
 - Cuando le pasan un producto seleccionado, debe mostrar los datos del mismo precargados.
 - Luego de presionar submit, se deben limpiar los campos.
- Hay algunas clases en `App.css` para ayudarlos, pero pueden cambiarlas o agregar las que quieran.