Curso de React y React Native

Clase 04

Agenda de la clase

Agenda

- Repaso.
- Lifecycle Methods.
- External Data Access.
- Presentational – Container.
- Ejercicios.

Repaso

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 pueden programar los métodos de ciclo de vida con 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()` – (Nativo de las sintáxis de class en JS)
- `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.
- 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)`
  (Será deprecated en próximas versiones)
- `render()` – Obligatorio definirlo.
- `componentDidUpdate(prevProps, prevState)`

Lifecycle Methods – Unmounting

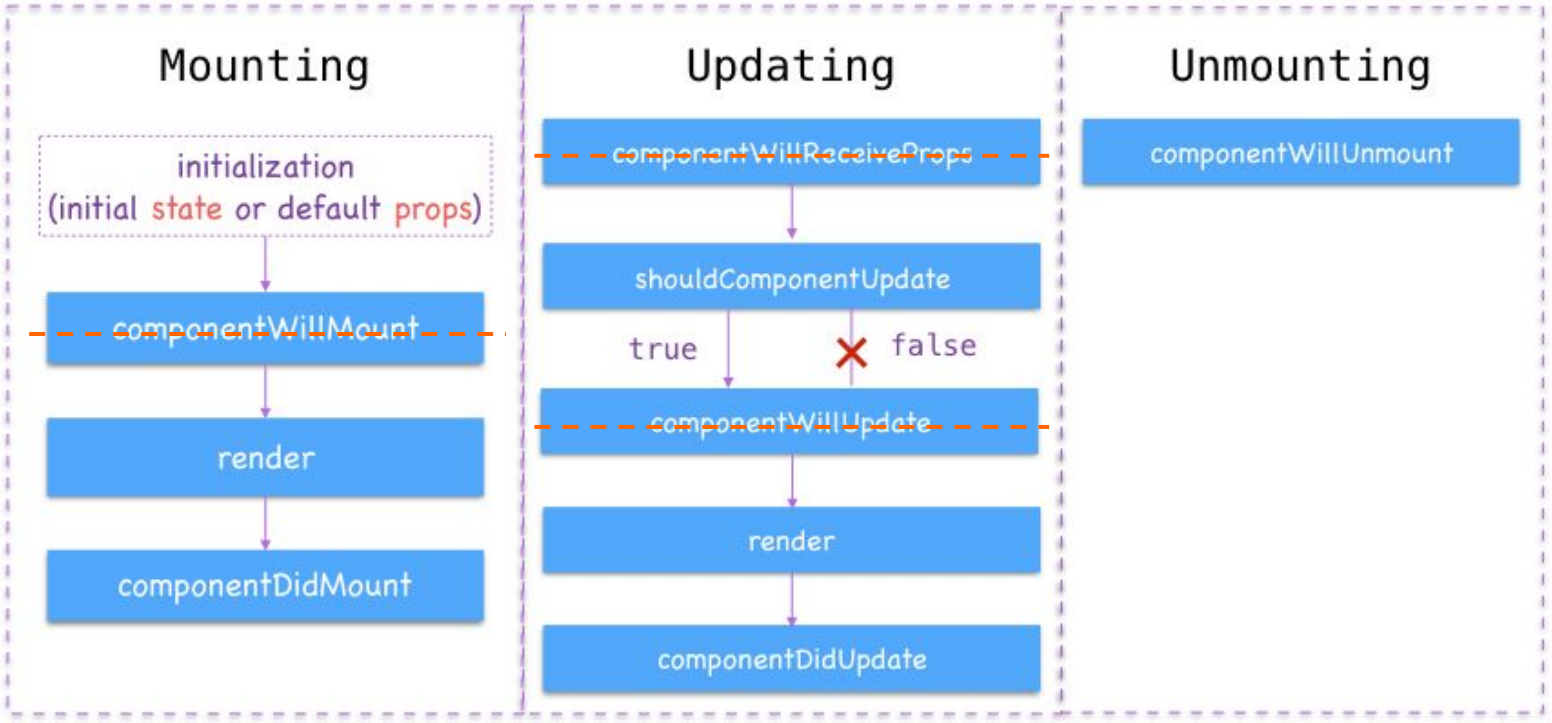
Este método es llamado cuando un componente se elimina del DOM:

- `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 desmontar 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) {
    // ...
  }
}
```

“Render Phase”

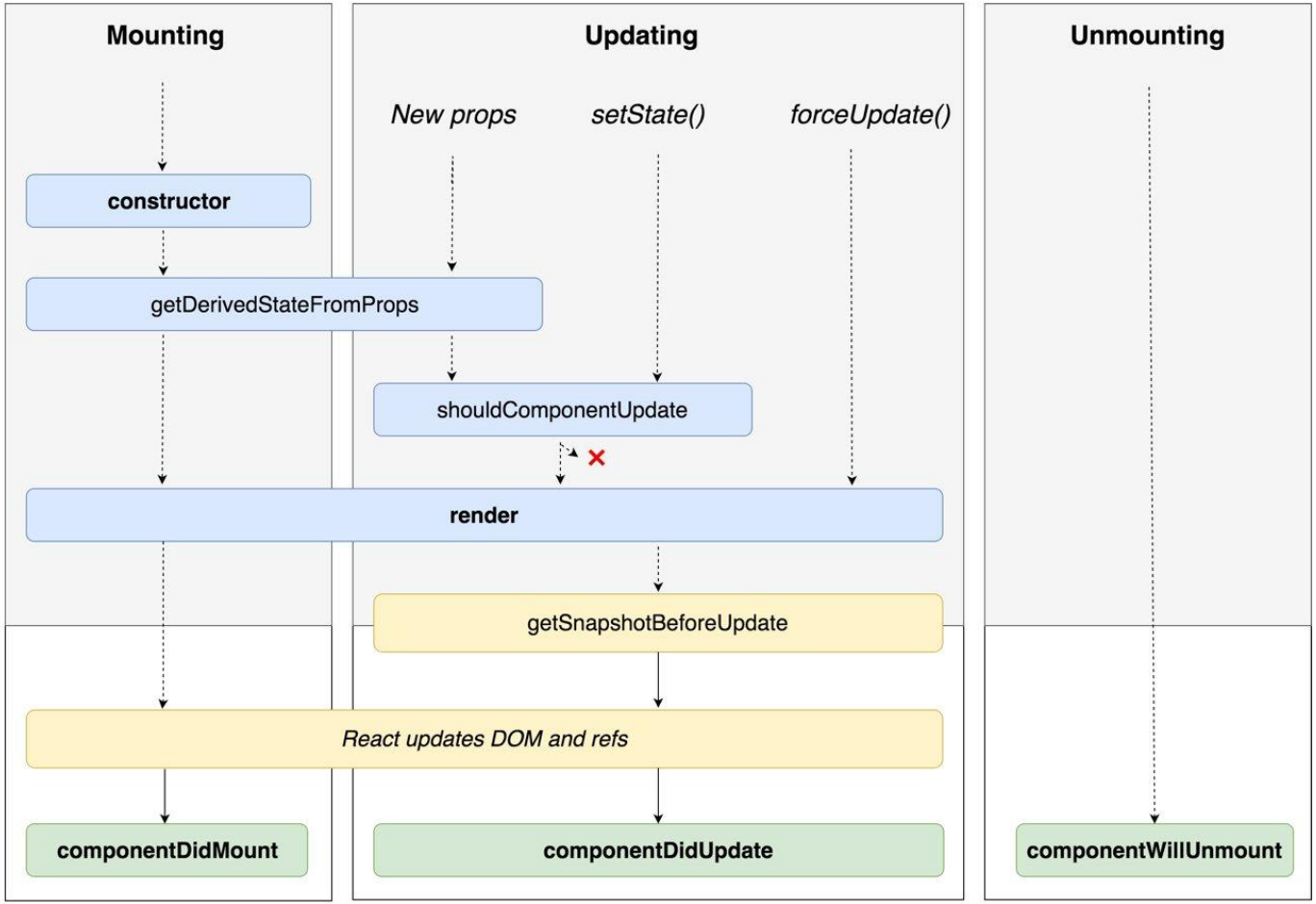
Pure and has no side effects. May be paused, aborted or restarted by React.

“Pre-Commit Phase”

Can read the DOM.

“Commit Phase”

Can work with DOM, run side effects, schedule updates.



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**.

Promesas (`Promise`) (1)

Las promesas fueron creadas para de una vez por todas dar consistencia y estandarizar el manejo de operaciones asíncronas.

Antes de ellas, se usaban callbacks, pero cada uno lo ponía como argumento en la posición que quería, con los parámetros que quería, y demás. Uno debía apoyarse constantemente en la especificación y en documentación *para cada una* de las operaciones asíncronas, lo cual no es ideal.

Promesas (`Promise`) (2)

El uso de promesas se resume a dos posibilidades:

1. Crear una promesa para exponer una funcionalidad asíncrona.
2. Consumir una promesa ya creada.

Promesas (`Promise`): Creación (1)

```
const promise = new Promise((resolve, reject) => {
    // Hacer algo, probablemente asíncrono, y luego...

    if (/* Si todo salió bien */) {
        resolve('Funcionó!');
    } else {
        reject(Error('Se rompió'));
    }
});
```

Promesas (`Promise`): Creación (2)

```
const request = require('request');

const getNodeVersions = new Promise((resolve, reject) => {
    request('http://nodejs.org/dist/index.json', (error, response, body) => {
        if (response.statusCode === 200) {
            resolve(body);
        } else {
            reject(`Status code: ${response.statusCode}`);
        }
    });
});
```

Promesas (`Promise`): Creación (3)

```
const request = require('request');

const get = url => new Promise((resolve, reject) => {
    request(url, (error, response, body) => {
        if (response.statusCode === 200) {
            resolve(body);
        } else {
            reject(`Status code: ${response.statusCode}`);
        }
    });
});
```

Promesas (`Promise`): Consumo (1)

```
get('http://nodejs.org/dist/index.json')
    .then(body => /* hacemos algo con el body de la response */)
    .catch(error => /* hacemos algo con la response */);
```

Promesas (`Promise`): Consumo (2)

```
get('http://nodejs.org/dist/index.json')
    .then(response => /* hacemos algo con la response */)
    .catch(error => /* hacemos algo con la response */);
```

Cualquier operación que retorne una promesa al ejecutarse, podemos concatenar un `.then()` y un `catch()`. No hay nada propietario que aprender. Todas se comportan igual.

Ese es el contrato al consumir una promesa, lo cual otorga la tan ansiada consistencia para consumir resultados de operaciones asíncronas.

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: 'Luís',
     lastName: 'Suárez'
})
.then(function (response) {
     console.log(response);
})
.catch(function (error) {
     console.log(error);
});
```

```
const baseUrl = "https://myapp.com/api";
axios.post(`${baseUrl}/user`, {
    firstName: 'Luís',
    lastName: 'Suárez'
})
.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 {
    componentDidMount() {
        axios.get("https://api.github.com/users").then(result => {
            this.setState({
                users: result.data
            });
        });
    }
    render() {
       // Render.
    }
}
```

Patrón 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 utiliza Presentational Components para mostrar en pantalla la información.

Presentational–Container (2)

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

```
class CommentList extends React.Component {
    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 {
    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.

Ejercicio

Ejercicio

1. Crear un proyecto de React.
2. Entrar a https://exchangeratesapi.io/, leer de qué va el sitio, su documentación y sus usos.
3. Por defecto, mostrar en pantalla el precio del EURO relativo al DÓLAR.
    - *Una buena oportunidad para crear un componente presentacional que reciba datos sobre dos divisas y las muestre en pantalla, comparándolas.*
4. Agregar dos menús `<select>` que contengan las divisas disponibles.
    - *Una buena oportunidad para crear un componente separado que reciba por props las opciones y un callback a ejecutar cuando ocurra un cambio.*
5. Cuando el usuario seleccione un valor en cualquiera de los dos select, actualizar el precio relativo entre las divisas seleccionadas.