3.7 Estado en React

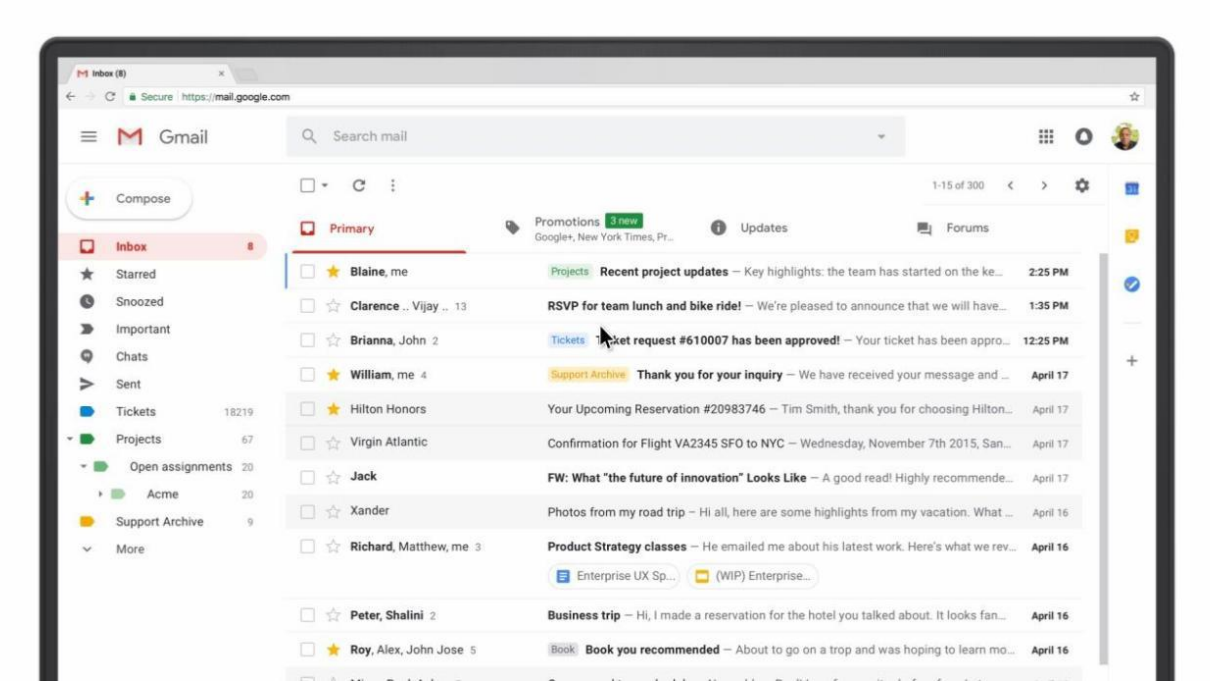
Contenidos

Introducción

Hasta ahora hemos usado los componentes prácticamente como plantillas HTML que podemos personalizar con `props`. Declarábamos los componentes y se pintaban. Aunque también hemos visto cómo pueden reaccionar a eventos, en esta sesión iremos un paso más allá y veremos cómo cada componente puede tener una pequeña memoria que le permitirá tener actividad por sí mismo.

¿Para qué sirve lo que vamos a ver en esta sesión?

El **estado** de una interfaz son sencillamente los datos que necesitamos para representarla. Por ejemplo, vamos a fijarnos en la interfaz de entrada a GMail. Los datos que necesitamos para poder pintarla son muchos más de los que pensamos a priori. Por un lado, necesitamos los datos concretos de email: el nombre de remitente, asunto, el texto inicial del email y la fecha. Pero también necesitamos conocer si un correo está marcado como favorito para mostrar la estrella rellana o vacía. O, al marcar algún correo con un check, hay opciones nuevas que aparecen. Por tanto, todos estos datos que necesito para poder pintar la interfaz son el estado.



New GMail cover

React es especialmente bueno manejando los pequeños cambios que se necesitan en cada uno de los componentes de una web. Cada instancia de un componente tiene un **estado** que refleja cada uno de esos pequeños cambios. React encapsula toda la complejidad y la distribuye en este sistema predecible. Saber utilizar los estados nos permitirá definir cómo se comportarán los componentes en cada momento y declarar *react-ciones* más elaboradas a según qué interacción con el usuario.

Manejo del estado en un componente de React

El estado de un componente es la memoria **en cada momento** que tiene la instancia de un componente que se está mostrando en pantalla. Se trata de un atributo de la instancia parecido a las `props`, al que podremos acceder con `this.state`. Al contrario que las `props`, el **estado varía** durante el tiempo en que el componente aparece pintado en la pantalla. Es decir, las `props` no podemos cambiarlas, pero los estados, sí, aunque de cierta manera. En la siguiente sección (*¿Qué pasa cuando hay un cambio de estado?*) veremos cómo afecta esto a nuestro componente.

Por defecto, el estado de un componente está vacío. Hay dos momentos y maneras distintas de darle valor: en el `constructor()` del componente, y en cualquier otro momento con `setState()`.

Asignar el estado inicial en el `constructor()`

En el `constructor()` podemos definir el estado que tendrá el componente en el primer momento en que se muestre en pantalla. En otras palabras, el estado inicial del componente. Lo haremos asignando a `this.state` un objeto con los valores iniciales de **todos los estados** que vaya a tener el componente:

```
const generateRandomInteger = maxValue => Math.floor(Math.random() * maxValue);

class RandomInteger extends React.Component {
  constructor(props) {
    super(props);
    const maxValue = props.maxValue;

    this.state = {
      number: generateRandomInteger(maxValue)
    };
  }

  render() {
    return <span>{this.state.number}</span>;
  }
}
```

> Nota: El `constructor` es el **único lugar** donde podremos asignar directamente un valor a `this.state`. En el resto, debemos usar `setState()`

Como en este ejemplo el estado nunca cambia más, puede parecer que esto no sirve de mucho comparado con las `props`, pero en realidad ya hemos delegado la responsabilidad de generar el número aleatorio al propio componente. El componente ahora es independiente y ningún componente padre o madre debe mandarle qué número mostrar. Hemos declarado un componente con identidad propia (y quizá adolescente, :P).

Actualizar el estado: el método `setState()`

Ahora bien, hemos dicho que podemos cambiar el estado. Sin embargo, tenemos que hacerlo de cierta manera. Sabemos que los componentes en React son declarativos: no decimos *cómo* se hace un componente, sino el *qué*, y React es quien se encarga del *cómo*. A la hora de cambiar los valores del estado, **no podremos asignar directamente los valores a `this.state`** o

cualquiera de sus propiedades, sino que utilizaremos el método `setState()` del componente: React se encargará del resto.

Podemos llamar a `setState()` de varias maneras. La más común y sencilla de ellas es pasarle un **objeto literal** con las claves (nombres) del estado que queremos cambiar y sus valores. Es decir, si tenemos tres estados `a`, `b` y `c`, pero solo queremos cambiar el valor de `c`, pasaremos un objeto `{ c: 'nuevo valor' }`, sin incluir `a` ni `b`. React lo mezclará con el estado actual y solo cambiará las propiedades que deba cambiar.

```
class BipolarButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      styling: 'info'
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // Nuestra función escuchadora del evento click
    this.setState({
      styling: 'danger'
    });
  }

  render() {
    return (
      <button
        className={`btn btn-${this.state.styling}`}
        onClick={this.handleClick}
      >
        {this.props.label}
      </button>
    );
  }
}
```

▶ `setState()` de objeto literal en Codepen

Utilizaremos la forma del objeto literal cuando el nuevo valor no dependa del anterior o de ningún otro estado del componente actual. Como React no asegura que los cambios de estado

se ejecuten en el momento (los agrupa en lotes), si usamos el valor de un estado actual para calcular otro podemos estar usando un estado que no tiene el valor que pretendemos. Eso es una fuente de errores. Para esos casos existe otra manera de llamar a `setState()`, esta vez le pasamos una función, un `callback`. El `callback` recibe como parámetros el estado que modificaremos (`prevState`), y las `props` del componente, y devolverá un objeto literal con las claves (nombres) de los estados que queremos cambiar. Es más fácil verlo que contarlo:

```
class BipolarButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      styling: 'info'
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState((prevState, props) => {
      let nextStyling;
      if (prevState.styling === 'info') {
        nextStyling = 'danger';
      } else {
        nextStyling = 'info';
      }

      return {
        styling: nextStyling
      };
    });
  }

  render() {
    return (
      <button
        className={`btn btn-${this.state.styling}`}
        onClick={this.handleClick}
      >
        {this.props.label}
      </button>
    );
  }
}
```

También se puede usar una *arrow function* más corta y, en este caso, un operador ternario o "`if` corto":

```
this.setState((prevState, props) => ({
  styling: prevState.styling === 'info' ? 'danger' : 'info'
})); // este doble paréntesis se suele olvidar de primeras
```

▶ `setState()` con función en Codepen

EJERCICIO 1

Mostrando info relacionada

Vamos a partir de una web sencilla con un input de tipo texto y un párrafo vacío. ¿Seremos capaces de hacer que con React y el estado, cuando modificamos el input aparezca el texto en el párrafo?

EJERCICIO 2

El cuadrado parpadeante

Vamos a crear una página con una cuadrado de tamaño fijo (por ejemplo un `div`) con un color de fondo azul. Vamos a hacer que al hacer clic sobre el cuadrado, su color de fondo cambie a rojo. Si volvemos a hacer clic, pasa de nuevo a azul, y así sucesivamente. Vamos a implementar este cuadrado parpadeante usando el estado del componente (podemos almacenar el color o un booleano).

> PISTA: Al escuchar el evento de clic para comprobar de qué color estaba anteriormente el cuadrado, usaremos la versión de `setState` que toma como parámetro el `prevState`

EJERCICIO 3

Qué hora será

Hace unas semanas, la empresa Time2Sleep nos encargó una página que mostrase sus ejercicios de relajación orientados a agilizar el sueño. Después de publicarla, recibieron *feedback* de sus usuarios: se quedaban tan profundamente dormidos que, al despertar, no recordaban ni su nombre. Como quedaron muy contentos con el trabajo (literalmente: *"¡cómo nos flipan estas adalabers!"*), ahora nos han pedido una evolutiva, que es como se llama a las

funcionalidades que se añaden a un proyecto ya hecho, para que añadamos un reloj a la web. Así sus usuarios sabrán al menos qué hora es.

Vamos a crear un componente reloj (`Clock`) que nos muestre la hora en cada momento. Tendrá un método `updateClock()` en el componente para actualizar el estado con `setState(/* objeto */)` , que actualizará la hora con `new Date()` . En el constructor del componente declararemos un `setInterval()` que ejecute `updateClock` cada segundo.

> PISTA: para obtener la información de la hora con un objeto de tipo fecha, podemos usar los métodos `getHours` , `getMinutes` y `getSeconds` como se explica en la página de MDN

EJERCICIO 4

Contador de ovejas

Definitivamente, Time2Sleep es fan de Adalab. Esta vez, basándose en unos novedosos estudios científicos de alguna famosísima universidad que dice que al contar ovejas nos aburrimos tanto que caemos dormidos, nos han encargado que hagamos un contador de ovejas digital.

Crearemos un componente cuentaovejas (`SheepCounter`) que mostrará un número en grande y un botón. El botón tendrá asignado un escuchador al evento `click` que aumentará el contador. Actualizaremos el valor del contador con `setState(/* función */)` .

¿Qué pasa cuando hay un cambio de estado?

Como hemos visto varias veces arriba, React no asegura que los cambios de estado se ejecuten al momento. Vamos a ver cómo hace React los cambios de estado, de manera simplificada.

Un cambio de estado no solo cambia los valores de `this.state` . Lo más importante que hace es **volver a llamar al método `render()`** del componente después. Recordemos que podemos usar valores del estado en nuestro JSX como texto, para calcular otros valores que utilizaremos o como `props` para otros componentes hijo, por ejemplo. Cuando el estado cambia, el componente se tiene que re- `render` izar de nuevo, y si las `props` de los hijos cambian, esos componentes hijos también tienen que re- `render` izarse, y sus hijos a su vez. Así que cambiar el estado es costoso, porque hace que vuelvan a `render` izarse varios componentes **en cadena**.

> **NOTA**: a partir de ahora tenemos que recordar que, para que un componente vuelva a pintarse, es decir, se ejecute su método `render` puede ser por 2 motivos: 1) por un cambio en el estado (`this.state`) o 2) por un cambio en las `props` que le llegan del componente padre.

Ahora podemos entender por qué React no asegura que los cambios de estado ocurran al momento, aunque sean bastante rápidos. Cuando llamamos a `setState()` en cualquiera de sus formas, React registra esa **petición** de cambio de estado y la añade a una cola de tareas por hacer. Para no tener que re- `render` izar componentes demasiadas veces, es posible que agrupe en lotes (*batches*) algunos cambios de estado a la vez y los procese juntos para mejorar con eso el rendimiento. Esto significa que tendremos que pensar **las llamadas a `setState()` como llamadas asíncronas**.

El `callback` de `setState()`

Si usásemos un valor de `this.state` después de llamar a `setState()` , podría no tener el valor actualizado. Por eso, `setState()` acepta un `callback` como último parámetro, como cualquier función asíncrona.

```
this.setState(
  {
    mensaje: 'nuevo mensaje'
  },
  () => {
    console.log(this.state.mensaje); // 'nuevo mensaje'
  }
);
```

El `callback` se ejecutará justo después de que el cambio de estado haya tenido lugar, así que se pueden usar los nuevos valores de `this.state` sin problema.

EJERCICIO 5

Contador de ovejas avanzado

Sobre el componente cuentaovejas (`SheepCounter`) del ejercicio anterior, añadimos la funcionalidad de que, además de mostrar el número de ovejas, muestra también la imagen de una oveja. Por ejemplo, si el contador está en 6, además de aparecer el número 6 veremos 6 imágenes de ovejas.

> Podéis usar esta imagen por ejemplo:
> http://www.clker.com/cliparts/e/4/8/7/13280460782141411990Cartoon Sheep.svg.hi.png

Spread operator

El operador *spread* (`...`) convierte un array o un objeto en el conjunto de valores que contiene, por lo que nos permite usarlos como si estuvieran escritos en el propio código. Una de las ventajas que nos ofrece el operador *spread* es que no tenemos por qué saber qué hay en el array u objeto en cada momento.

Spreading de array

Veamos un par de ejemplos con arrays. Tenemos un array y queremos añadirle un nuevo valor:

```
const names = ['Smith', 'White', 'Black', 'Pinkman'];

const newNames = [...names, 'Williams'];

console.log(newNames)); // ['Smith', 'White', 'Black', 'Pinkman', 'Williams']
```

Ahora pongamos que queremos mezclar dos arrays distintos que tenemos:

```
const myBooks = ['1984', 'Brave New World'];
const myBrotherBooks = ['We', 'Fahrenheit 451'];

const books = [...myBooks, ...myBrotherBooks];

console.log(books); // [`1984`, 'Brave New World', 'We', 'Fahrenheit 451']
```

Spreading de objeto

Podemos usar el operador *spread* también con las propiedades de los objetos. Por ejemplo, para añadir una propiedad nueva o sobreescribirla si ya existe. En este ejemplo copiamos el objeto `person` y lo guardamos en `twinSister` . El objeto `person` sigue existiendo y los dos son independientes:

```
const person = {
  name: 'Marie',
  lastName: 'Smith',
  age: 39
};

const twinSister = { ...person, name: 'Juliette' };

console.log(twinSister); // { name: 'Juliette', lastName: 'Smith', age: 39 }
```

Copia vs referencia

Recordemos que en el módulo de javascript aprendimos que si teniamos un array o un objeto literal en una variable, al guardarlo en otra, en realidad guardábamos una *referencia* del array/objeto.

De manera que si modificábamos o mutábamos una de las dos variables, por ejemplo con un `.push` para el caso del array o cambiando el valor de una propiedad para el caso del objeto, estos cambios también sucedían en la otra variable, ya que el array u objeto contenido en ambas era el mismo.

El operador *spread* nos permite hacer una copia de un objeto o array en un momento dado. De manera que si cambiamos o mutamos, por ejemplo, el original, la copia no se ve afectada, ya que son independientes y no comparten referencia.

Actualizando un valor del estado a partir del segundo nivel

Como hemos visto, si tenemos en el estado un objeto con tres claves, por ejemplo:

```
this.state = {
  selectedTab: 4,
  theme: 'dark',
  userData: {
    age: 30,
    eyeColor: 'soft-caramel',
    mostHatedFruit: 'banana'
  }
};
```

Podemos actualizarlas con `this.setState()`, pasándole como primer parámetro **un objeto o una funcion que debe devolver un objeto**

```
this.setState({ selectedTab: 2 });
```

o

```
this.setState(prevState => {
  return { selectedTab: 2 };
});
```

Este objeto debía tener, como mínimo, una de las propiedades que se encuentran en el primer nivel del estado (`selectedTab, theme, userData`) y React se encarga de que las propiedades hermanas no presentes no se vean afectadas.

Pero, ¿y si queremos actualizar una propiedad que no está a primer nivel, por ejemplo `age` ?. En estos casos tenemos que identificar la propiedad madre en el primer nivel, en nuestro ejemplo sería `userData`, y es la que tenemos que modificar.

Las propiedades hijas de `userData` (`eyeColor, mostHatedFruit...`) ya no se encuentran a primer nivel y React no las controla, es nuestra responsabilidad que al modificar una las otras no se vean afectadas.

En este punto, si tenéis que modificar la clave `age`, es posible que penséis en algo así:

```
this.setState(prevState => {
  prevState.userData.age = 33;
  return { userData: prevState.userData };
});
```

Pero a React **NO le gusta que mutemos arrays y objetos anidados, mejor si creamos una copia**, ¡operador `spread` al rescate!:

```
this.setState(prevState => {
  return {
    userData: {
      ...prevState.userData,
      age: 33
    }
```

```
  };
});
```

En este ejemplo creamos una copia del objeto `userData` usando spread y después actualizamos la propiedad `age` al nuevo valor (33). Ten en cuenta que el orden es importante, y si hay una propiedad repetida, prevalece la de más abajo (como en la cascada de CSS).

EJERCICIO 6

Info del usuario

Vamos a partir de un objeto con información de un usuario que tenemos en el estado de nuestro componente. Lo vamos a inicializar a este valor directamenete en el constructor.

```
{
  firstName: 'Ada',
  lastName: 'Lovelace',
  birdthDate: {
    day: 10,
    month: 'diciembre',
    year: 1815
  }
}
```

Vamos a crear un formulario donde vamos a poder modificar estos campos del estado.

> NOTA: Cuidado al modificar los campos anidados dentro del objeto `birdthDate` ; recuerda que para modificarlos es muy útil usar en el `setState` el operador spread `...` para mantener el resto de datos de ese objeto. Por ejemplo:

```
this.setState(prevState => {
  return {
    birthDate: {
      ...prevState.birthDate,
      day: 8
    }
  };
});
```

BONUS

EJERCICIO BONUS 7

Fruta fresca

Vamos a hacer una lista de frutas populares, que nos permita añadir y quitar elementos.

```
this.state = {
  popularFruits: ['kiwi', 'pinneaple', 'strawberry'],
  newFruit: ''
};
```

1. Pintar el listado a partir de la clave del estado `popularFruits` .
2. Pintar un input de texto y un botón con el texto 'Añadir'
3. Cada vez que el input cambie, hay que actualizar la clave del estado `newFruit`
4. Cuando se pulse el botón 'Añadir' hay que:
   - actualizar la clave del estado `popularFruits` con el valor de `newFruit` . Ojo, no vale mutar el array contenido en `popularFruits` con un push. Usaremos `spread` o el método de array `.concat` para generar un nuevo array.
   - actualizar el valor de `newFruit` cn comitas vacias para limpiar el input.
5. Ahora vamos a añadir un botón 'Eliminar' junto a cada fruta, a este botón tenemos que añadirle un atributo `value` o `data-fruit` con el nombre de la fruta junto a la que se encuentra como valor.
6. Cuando se pulse el botón tenemos que recoger la fruta que queremos eliminar y actualizar la clave del estado `popularFruits` con un nuevo array que no contenga dicha fruta. El método `.filter` de array que devuelve una copia nueva puede ayudaros con esta tarea.

Al turrón!

Spread y parámetros de funciones

El operador *spread* también nos puede ser útil para pasar todos los valores de un array como parámetros a una función:

```
const vowels = ['a', 'e', 'i', 'o', 'u'];

console.log(...vowels);
```

Esto sería equivalente a:

```
console.log(vowels[0], vowels[1], vowels[2], vowels[3], vowels[4]);
```

Rest parameters

Cuando declaramos una función propia también nos sirve para almacenar los parámetros sobrantes (*rest parameters*) en una sola variable:

```
function showFavoriteFruits(first, ...rest) {
  const restOfFruits = rest.join(' and ');
  console.log(
    `My favourite fruit is the ${first}, although I like ${restOfFruits} also.`
  );
}

const myFavoriteFruits = ['orange', 'banana', 'pear'];
showFavoriteFruits(...myFavoriteFruits); // 'My favourite fruit is the orange,
```

Recursos externos

Documentación de React

Documentación oficial de React (en inglés).

- Estado en los componentes

Egghead

Serie de clases en vídeo que introduce y explora los fundamentos básicos de React (en inglés).

- Gestión de los estados en los componentes