REACT

FROM ZERO TO HERO

PROMESAS

Las promesas se crean con:

- palabra reservada 'new'
- palabra reservada Promise (la p con mayúscula)
- con un argumento adentro, que es un callback
- el callback recibe dos argumentos: resolve y reject
- se los llama asi por convención, realmente los puedes llamar como te de la gana
- `resolve` se ejecuta cuando la promesa es exitosa
- `reject` se ejecuta cuando ha habido un error
- setTimeout es una funcion que recibe dos argumentos. Un callback y el tiempo en que ejecutes ese callback.
- En este caso, el `console.log` lo hará después de 2000 milisegundos

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('2 segundos después');
  }, 2000);
});
```

Con ese código, se habrá corrido el resto de código en caso de existir, y dos segundos despues, al final, se cumple tu promesa. Ahora, como hacer para ejecutar código DESPUES de que se cumpla la promesa? Puedes dejar 'automatizado' eso con disntintos metodos:

- catch()
- finally()
- then()

then()

`then()` lo que pones dentro de then() es lo que se ejecutará una vez que se cumpla la promesa. Then() recibe como parámtro un callback

ejemplo:

```
promesa.then(() => {
    console.log('Then de la promesa)
})
```

Ese código de arriba lo escribirías después de declarar la promesa. El código total te quedaría todo así:

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('2 segundos después');
  }, 2000);
});

promesa.then(() => {
    console.log('Then de la promesa)
})
```

Hay un problema con el código de arriba, y es que no se le esta especificando al then absolutamente nada. Es por eso que al declarar la promesa, ahí tienes que ejecutar el resolve(), al cual le tienes que pasar un parámetro, y por defecto, este se lo pasa al then() que está en el bloque de código de abajo.

Asumamos que en vez de un console log, estas haciendo una solicitud a una API, y eso obviamente lo guardas en una const. En vez de hacer un console log de ese const en el código de la promesa, puedes ahi escribir un resolve, el cual recibe como parámetro, la const que contiene a esa petición de API.

Esa const es luego pasada por el resolve() al then(), el cual puede ejecutar cualquier linea de código.

ejemplo:

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    const heroe = estaEsTuPeticionApi(4);
    resolve(heroe);
  }, 2000);
});

promesa.then((heroe) => {
  console.log(heroe);
});
```

Ahora si, con ese código, estas ejecutando asincronismo 😃
Ahora, que pasa, si quieres guardar esa logica, dentro de una función? Es decir, quieres una funcion, a la cual le pasas argumento, y esa te ejecuta la logica, con ese argumento que le estás pasando? Lo tendrías que hacer así: (asumiendo que ya tienes una funcion que pide api, llamada estaEsTuPeticionApi)

```
const estaEsTuFuncionQuePediraApi = (id) => {
  const promesa = new Promise((resolve, reject) => {
    setTimeout(() => {
      const heroe = estaEsTuPeticionApi(id);
      resolve(heroe);
    }, 2000);
  });
```

```
  return promesa;
};
```

Es necesario que pongas ese return, porque si no, esa funcion lo que te devuelve por defecto es void. Tambien podrías nomas directamente escribir el return arriba:

```
const estaEsTuFuncionQuePediraApi = (id) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const heroe = estaEsTuPeticionApi(id);
      resolve(heroe);
    }, 2000);
  });
  return promesa;
};
```

Así no estarías creando una constante innecesaria, como en el penúltimo ejemplo. Esta es la manera más usada. Teniendo esa logica ya atrapada como una funcion, podemos solo escribir en una linea:

```
estaEsTuFuncionParaPedirApi(4).then((heroe) => console.log(heroe));
```

Hasta ahora, solo has trabajado con resolve, asumiendo que tu promesa siempre funciona. Para implementar el reject, y atrapar ese error con el catch, tienes que implementar un if else.

`catch()`

`cacth()` lo que pones dentro, se ejecutara cuando la promesa devuelva un error

```
const estaEsTuFuncionQuePediraApi = (id) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const heroe = estaEsTuPeticionApi(id);
      if (heroe) {
        resolve(heroe);
      } else {
        reject('Tuvimos un error');
      }
    }, 2000);
  });
  return promesa;
};

estaEsTuFuncionParaPedirApi(4).then((heroe) => console.log(heroe));
.catch(err => console.warning(err))
```

`finally()` se ejecuta despues del then o despues del catch, es decir, ya sea que se haya ejecutado cualquier de esos dos, el finally() es lo que se ejecuta despues de eso.

mini resumen:

Las promesas básicmente capturan y crean lógica, usando las palabras reservadas `resolve` y `reject`, los cuales reciben parametros, y estos mismos, mandan esos parámetros a otro bloque de lógica el cual posee `.then()` y `.catch()`. Estos últimos, ejecutan lógica con el argumento que reciben de resolve y reject.

FETCH API

La idea en este pdf, es saber básicamente como funciona fetch api y aprender sobre *promesas en cadena*

teniendo tu api_key

```
const apiKey = 'XF6VjNxOHCOOHgtNwRLAy5oftKQeWRgT';
```

Y teniendo tu endpoint, que en este caso es: api.giphy.com/v1/gifs/trending

Si tu vas a ese url por si solo, te devuelve un package json y te dice que no te puede ensenar el contenido ya que no le estas proveendo la api key. Si vas a ese url de ese endpoint, y agregas '?api_key=', seguido de tu api key, es decir: https://api.giphy.com/v1/gifs/trending?api_key=XF6VjNxOHCOOHgtNwRLAy5oftKQeWRgT Ahora si puedes ver el contenido de esa API

Ahora si, la idea es crear una función, que ejecute esa petición, usando la api key y el endpoint. Para esto usaremos fetch, lo cual, ya está instalado en tu navegador, por eso no hay que instalar nada.

Creas la const, y le metes fetch, el cual recibe el url de tu api. Como puedes ver, se lo pone en comillas francesas o backticks, ya que usarás strings y tambien tu variable. Básicamente estás concatenando tu endpoint el cual es un string, con tu const que contiene a tu api key:

```
const peticion = fetch(
  `https://api.giphy.com/v1/gifs/trending?api_key=${apiKey}`
);
```

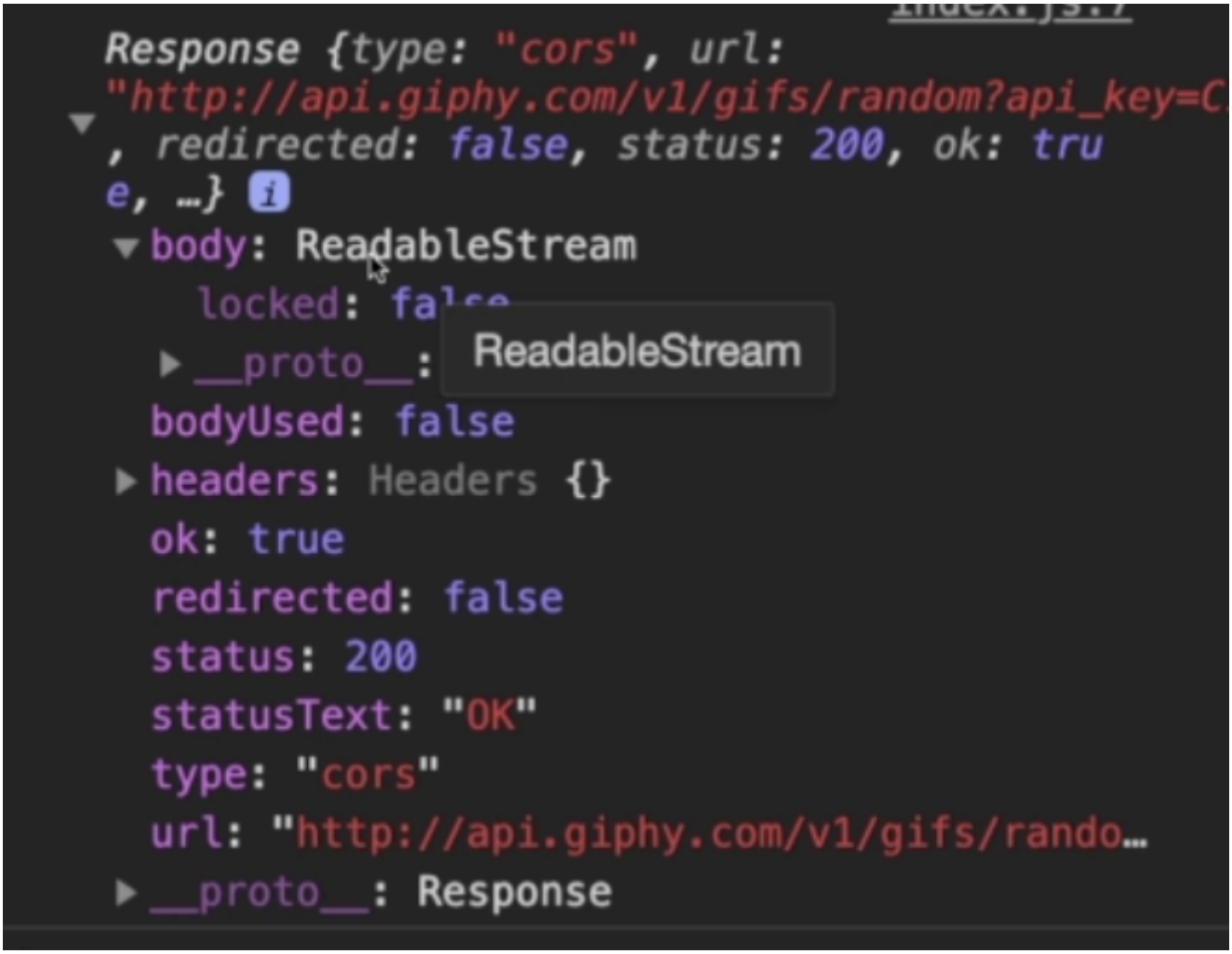
`fetch() retorna una promesa la cual se llama response`!!

y como es una promesa, quiere decir que puedes usarle .then(), etc 😃

Si tu fueses a hacer un `console.log()` a esa response, la consola te mostraría un objeto.

```
peticion.then((resp) => {
  console.log(resp);
```

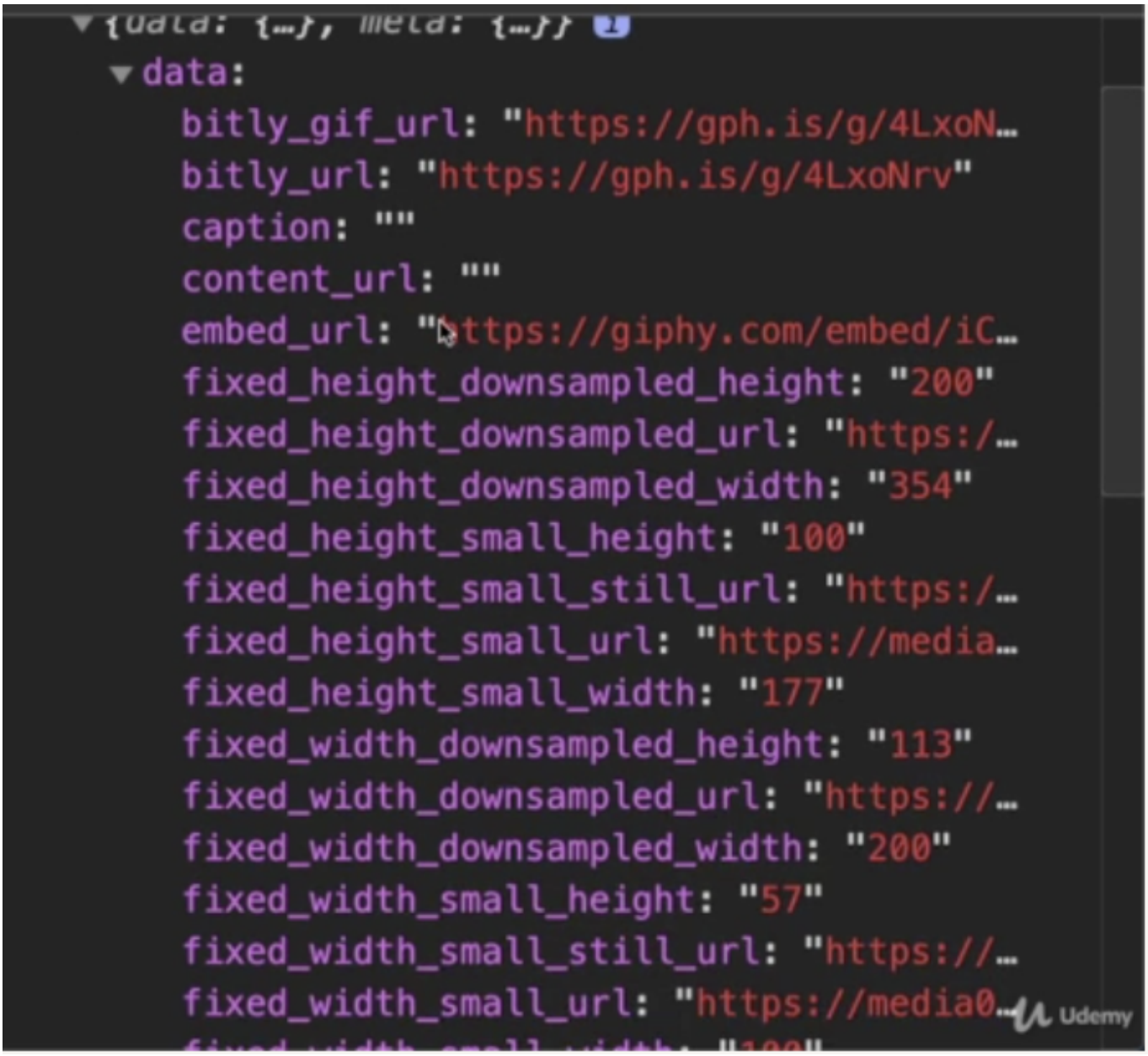
```
});
```



Lo que te importa, esta dentro de la propiedad body, la cuál aún no se puede acceder. Para acceder:

```
peticion.then((resp) => {
  resp.json().then((data) => {
    console.log(data);
  });
});
```

Ese código se ve feo, y más adelante veremos como se lo arregla, pero hasta ahora, saber que con esto, ya se muestra la data que nos interesa:



Encadenar promesas

La manera de escribir el mismo último bloque de código, pero más limpio, es usando `promesas en cadena`:

```
peticion
  .then((resp) => resp.json())
  .then((data) => {
    console.log(data);
  })
  .catch(console.warn);
```

este bloque de código, se ve mejor y hace exactamente lo mismo que este que usaste previamente:

```
peticion.then((resp) => {
  resp.json().then((data) => {
    console.log(data);
```

```
  });
});
```

Básicamente lo que pasa, es que, el resultado del primer then, es pasado al siguiente then, y así.

no hace falta escribir múltiples `catch()` ya que con uno, atrapas todos los errores

Para agregar, el url específico de una imagen, y crear una imagen en el HTML:

```
const apiKey = 'XF6VjNxOHCOOHgtNwRLAy5oftKQeWRgT';

const peticion = fetch(
  `https://api.giphy.com/v1/gifs/trending?api_key=${apiKey}`
);

peticion
  .then((resp) => resp.json())
  .then(({ data }) => {
    const { url } = data.images.original;
    const img = document.createElement('img');
    img.src = url;

    document.body.append(img);
  })
  .catch(console.warn);
```

Date cuenta como se crea una const para la url, luego creas un elemento con el método `createElement()`, y luego le agregas la url al src. Al final, usando manipulación del dom, en el body, usas el metodo `append()` para agregar esa img, a tu HTML.

Async-Await

Primero un pequeño ejemplo de una promesa:

```
const estaEsTuPromesa = () => {
    const promesa = new Promise((resolve, reject) => {
        resolve('https://cualquiercosa.com')
    })
    return promesa;
}

estaEsTuPromesa().then(console.log)
```

El siguiente paso para hacer ese código más corto:

```
const estaEsTuPromesa = () = {
    return new Promise((resolve, reject) => {
        resolve('https://cualquiercosa.com')
    })
    return promesa;
}

estaEsTuPromesa().then(console.log)
```

El siguiente:

```
const estaEsTuPromesa = () => new Promise((resolve =>
        resolve('https://cualquiercosa.com')
    )

estaEsTuPromesa().then(console.log)
```

Hasta este punto, el código está más corto, pero no necesariamente más sencillo de leer. Aquí es donde entra el `async-await`

Se puede lograr lo mismo que los códigos de arriba, de esta manera:

```
const getImage = () => {
  return 'https://cualquiercosa.com';
};

getImage();
```

Si, retorna lo mismo que el otro código, pero con una gran diferencia, esto no es aún asíncrono. Para hacerlo asíncrono:

```
const getImage = async () => {
  return 'https://cualquiercosa.com';
};

getImage().then(console.log);
```

Ahora si, lograste que sea asíncrona. Usaste dos cosas:

- la palabra reservada `async` antes de los argumentos
- `.then()` despues de convocar la función.

el async puede estar independiente, sin necesitar el await. El await siempre necesitará del async

El await nos permite trabajar nuestro código como si fuese síncrono.

La idea es hacer la siguiente linea de código (sacada de un ejercicio anterior, pero hacerla con async-await):

(sin async-await):

```
const apiKey = 'XF6VjNxOHCOOHgtNwRLAy5oftKQeWRgT';

const peticion = fetch(
  `https://api.giphy.com/v1/gifs/trending?api_key=${apiKey}`
);

peticion
  .then((resp) => resp.json())
  .then(({ data }) => {
    const { url } = data.images.original;
    const img = document.createElement('img');
    img.src = url;

    document.body.append(img);
  })
  .catch(console.warn);
```

(con async-await):

```
const getImagen = async() => {


    const apiKey='XF6VjNxOHCOOHgtNwRLAy5oftKQeWRgT';

    const peticion = await fetch(`https://api.giphy.com/v1/gifs/trending?
api_key=${apiKey})`

    const {data} = await peticion.json();.
    const {url} = data.images.original;

    const img = document.createElement('img);
    img,src = url;
    document.body.append(img);
}
```

En el async-await, para manejar errores, se usa `try` y `catch`:

```
const getImagen = async() => {
    try {
        const apiKey='XF6VjNxOHCOOHgtNwRLAy5oftKQeWRgT';

        const peticion = await
```

```
fetch(`https://api.giphy.com/v1/gifs/trending?api_key=${apiKey})`

        const {data} = await peticion.json();

        const {url} = data.images.original;

        const img = document.createElement('img');
        img.src = url;

        document.body.append(img);
    } catch(error) {
        #aquí va tu lógica del error
    }
}
```

Operación Condicional Ternario

Este código:

```
const = 'activo'
let mensaje = '';

if (activo) {
    mensaje = 'Activo';
} else {
    mensaje = 'Inactivo'
}
```

Es lo mismo que:

```
const activo = true;
const mensaje = activo ? 'Activo' : 'Inactivo';
```

En caso de querer hacer algo cuando la condición se cumpla pero no hacer nada si no se cumple:

```
const activo = true;
const mensaje = activo ? 'Activo' : null;
```

O mejor:

```
const activo = true;
const mensaje = activo && 'Activo';
```

eso ejecuta solo si la condición se cumple, si no, retorna false. Esto es bastante útil y facil en react.