ÍNDICE

La **interacción** es junto a la inclusión de elementos multimedia una de las claves del avance del desarrollo web.

La interacción es el proceso que establece un usuario con un dispositivo, sistema u objeto determinado. Es una acción recíproca entre el elemento con el que se interacciona y el usuario.

"Un elemento interactivo es aquel que cambia cuando el usuario interactúa con él"

Un ejemplo se puede observar al interactuar con la guitarra que desarrolló Google Logos en honor al nacimiento de Les Paul: http://www.google.com/logos/2011/lespaul.html

Este Logo interactivo fue desarrollado con HTML5, CSS3 y JavaScript, reemplazando a los archivos .swf ya obsoletos realizados con Adobe Flash.

Uno de los aspectos más importantes de la interactividad en el ámbito de esta unidad y módulo son los efectos y animaciones.

Como se mencionaba anteriormente, ha habido diferentes tendencias antes de los lenguajes HTML, CSS y JavaScript. Incluso en la tecnología JavaScript se han empleado diversas utilidades que a día de hoy ya no son necesarias, ya que este lenguaje va evolucionando para que de forma nativa se puedan incorporar utilidades relativas a efectos y animaciones.

A continuación se explican en detalle los métodos que JavaScript nos proporciona para crear efectos y animaciones.

1. Las funciones setTimeout y clearTimeout

El manejo de tiempos es primordial cuando se trata de añadir dinamismo a la página web.

La función **setTimeout**_establece un temporizador que ejecuta una función o una porción de código después de que transcurre un tiempo establecido.

El formato más típico de esta función es: `setTimeout(nombreFuncion, milisegundos)`. Cada parámetro tiene el significado que se explica debajo:

- nombreFunción: El código que se ejecuta. Se puede definir en el propio parámetro.
- Milisegundos: El tiempo en milisegundos que el temporizador debe esperar antes de ejecutar la función o el código. Si se omite este parámetro se usa el valor 0.

Un ejemplo sencillo es el código de debajo que muestra un alert tras esperar 3 segundos (3000 ms).

```
var myVar;

function myFunction() {
   myVar = setTimeout(alertFunc, 3000);
}

function alertFunc() {
   alert("Hello!");
}
```

La función **setTimeout** devuelve un ID de proceso**.** El uso más común es parar una animación asociada al ID si lo consideramos necesario. Esto se lleva a cabo con `clearTimeout(ID)`

```
var myVar;

function myFunction() {
   myVar = setTimeout(() => alert("Hello"), 3000);
}

function myStopFunction() {
   clearTimeout(myVar);
}
```

2. Las funciones setInterval y clearInterval

Ejecuta una función o un fragmento de código de forma repetitiva cada vez que termina el periodo de tiempo determinado

La sintaxis de **setInterval** es muy parecida a setTimeout, es decir, `setInterval(nombreFuncion, milisegundos)`

De la misma manera, setInterval devuelve un ID de proceso que se puede pasar como parámetro a **clearInterval**, que se encarga de detener la animación actual.

Un ejemplo de ambas funciones podría ser el siguiente, que inicia una animación hasta que se desplaza 350px y en ese momento se para con clearInterval.

```
function myMove() {
   let elem = document.getElementById("animate");
   let pos = 0;
   var id = setInterval(frame, 5);

   // Cuando se mueven 350 píxeles se para la animación
   function frame() {
      if (pos == 350) {
         clearInterval(id);
      } else {
         pos++;
         elem.style.top = pos + "px";
         elem.style.left = pos + "px";
      }
   }
}
```

3. Las funciones requestAnimationFrame y cancelAnimationFrame

Las funciones setTimeout y setInterval como en el código anterior pueden llegar a resultar ineficientes a la hora de realizar animaciones, ya que requieren que el navegador compruebe continuamente la cantidad de segundos que han pasado y tenga que renderizar la página más de lo debido. Esto puede tener como consecuencia que las animaciones no se ejecuten dentro de los tiempos requeridos.

El método **requestAnimationFrame** informa al navegador sobre la realización de una animación y solicita que el navegador programe el renderizado de la ventana para el próximo ciclo de animación. El método acepta un único argumento: una función callback que recibe un parámetro que indica el tiempo (en formato timestamp con milisegundos) en el que está programado que se ejecute el ciclo de animación.

Veamos un ejemplo para aclarar el funcionamiento.

```
var start = null;
var element = document.getElementById('SomeElementYouWantToAnimate');

function step(timestamp) {
   if (!start){
      start = timestamp;
      var progress = timestamp - start;
      element.style.transform = 'translate(' + Math.min(progress / 10, 200) + 'px, ' +
          Math.min(progress / 10, 200) + 'px)';
      if (progress < 2000) {
         window.requestAnimationFrame(step);
      }
   }
}

window.requestAnimationFrame(step);
```

Como se puede observar, para evitar que la animación se detenga hay que llamar a su vez a requestAnimationFrame() desde el callback hasta que se cumpla la condición adecuada, en este caso que pasen dos segundos.

En definitiva la estructura sería siempre así:

```
function repetirVariasVeces() {
   // Realizar acciones durante la animación

   requestAnimationFrame(repetirVariasVeces);
}

// Primera llamada y luego se hacen llamadas recursivas en el propio callback
requestAnimationFrame(repetirVariasVeces);
```

De esta forma, se produce el renderizado del componente en el navegador y esto puede ocurrir hasta 60 veces por segundo en pestañas en primer plano, pero tiene una serie de mecanismos para optimizar el rendimiento en pestañas que están inactivas, a diferencia de los métodos anteriores.

Al igual que setTimeout y setInterval, la función requestAnimationFrame devuelve un ID de proceso que sirve a una función análoga denominada **cancelAnimationFrame** para detener la animación actual, como se especifica en las partes del código en negrita.

```
var start = null;
var element = document.getElementById('SomeElementYouWantToAnimate');
var myRequest;

function step(timestamp) {
   if (!start){
      start = timestamp;
      var progress = timestamp - start;
      element.style.transform = 'translate(' + Math.min(progress / 10, 200) + 'px, ' +
          Math.min(progress / 10, 200) + 'px)';
      if (progress < 2000) {
         myRequest = window.requestAnimationFrame(step);
      }
   }
}

myRequest = window.requestAnimationFrame(step);

window.cancelAnimationFrame(myReq);
```

Por tanto, en el código de arriba no se llegaría a ejecutar la animación porque se detiene antes.

4. La propiedad transition

Las transiciones no son objeto de estudio de esta unidad, pero podrían ser de utilidad en algún momento.

Las transiciones CSS, parte del borrador de la especificación CSS3, proporcionan una forma de animar los cambios de las propiedades CSS, en lugar de que los cambios surtan efecto de manera instantánea. Por ejemplo, si se cambia el color de un elemento de blanco a negro, normalmente el cambio es instantáneo. Al habilitar las transiciones CSS, el cambio sucede en un intervalo de tiempo que puedes especificar, siguiendo una curva de aceleración personalizable.

A continuación se explica a través de un ejemplo su funcionamiento.

```
<style>
   div.recuadro {
      width: 100px;
      height: 100px;
      background: red;
      transition: width 2s, height 4s;
   }
</style>
<script type="text/javascript">
   function redimensionar(ancho, alto){
      document.querySelector(".recuadro").style.width = ancho;
      document.querySelector(".recuadro").style.height = alto;
   }
</script>
```

La propiedad abreviada **transition** permite especificar entre comas tantas propiedades como queramos cambiar. En este caso el ancho se modifica en 2 segundos y el alto en 4 segundos.

Como se decía anteriormente, también es posible personalizar la curva de aceleración añadiendo un tercer parámetro denominado **transition-timing-function**. La estructura en este caso podría ser `transition: width 2s ease, height 4s ease-in;`

Ya veremos más adelante que las transiciones nos permiten ahorrar código de animaciones. Puedes encontrar más información en la especificación de la W3C: https://www.w3.org/TR/css-transitions-1/

5. Promesas de JavaScript

Hasta ahora hemos visto cómo crear animaciones. De alguna manera podemos controlar cuando terminan. Pero, ¿qué ocurre si queremos encadenar varios efectos?

Por un lado, el siguiente código en realidad ejecuta los tres efectos casi a la vez porque las instrucciones se interpretan de arriba a abajo y no podemos controlar cuando termina cada uno.

```
setTimeout(1000, hazAlgo);
setTimeout(1000, hazAlgoMas);
setTimeout(1000, hazLaTerceraCosa);
```

Podríamos hacer que se encadenaran varios **callback** de los diferentes **setTimeout** de la siguiente manera:

```
setTimeout(1000, function () {
   setTimeout(1000, function () {
      setTimeout(1000, hazLaTerceraCosa);
   });
});
```

Esto genera el llamado código **spaguetti** que en realidad resulta muy complicado manejar. Veamos otro enfoque.

```
hazAlgo().then(function (resultado) {
   return hazAlgoMas(resultado);
})
.then(function (nuevoResultado) {
   return hazLaTerceraCosa(nuevoResultado);
})
.then(function (resultadoFinal) {
   console.log('Obtenido el resultado final: ' + resultadoFinal);
})
```

La idea de la estructura anterior es que las diferentes funciones que vamos llamando (hazAlgo, hazAlgoMas y hazLaTerceraCosa) permiten indicar al desarrollador cuando ha terminado el código, que resulta especialmente útil en animaciones con una duración de segundos o cuando hay que cargar algún recurso en el navegador. Siguiendo el enfoque anterior podemos añadir tantas funciones con **then** como secuencias queramos encadenar

5.1. Creación y secuencias con then

En un código de programación estándar las instrucciones se ejecutan secuencialmente. Hay certeza de que cada una va detrás de la otra. Sin embargo, en computación asíncrona si una animación dura 2 segundos, si no lo controlamos bien, el resto del código se sigue ejecutando.

Una promesa es un objeto que representa la terminación o el fracaso de una operación asíncrona. Representa un valor que puede estar disponible ahora, en el futuro, o nunca.

Las promesas se definen como se indica en el siguiente fragmento de código real:

```
function decirGanador(numero) {
   return new Promise((resolve, reject) => {
      setTimeout(function () {
         if (numero % 2 == 0) {
            resolve("¡Éxito, es un número par!");
         } else {
            reject("¡Error, es un número impar!");
         }
       }, 5000);
   });
}
```

El objeto promesa recibe dos parámetros. Cada uno de ellos es una función, lo que puede resultar un poco abstracto y complejo. La función **resolve** representa la terminación con éxito, mientras que **reject** se llama en caso de error. Cada una de estas funciones reciben como parámetro un tipo de datos que puede ser cualquiera y se recogerá al lanzar la secuencia la promesa. Los métodos **resolve** y **reject** se encargan de finalizar la promesa automáticamente con error o éxito según el caso.

En el ejemplo anterior creamos una función que recibe un número. Esta función crea una promesa que en 5 segundos dice al usuario si es par (éxito) o impar (fracaso). Como la función ya devuelve una promesa en sí, podemos hacer referencia al resultado con el método **then** descrito anteriormente de la siguiente manera.

```
decirGanador(10)
   .then(data => alert(data))
   .catch(error => alert(error));
```

El método **then** de los objetos **Promise** se ejecuta solo cuando la promesa que devuelve la función decirGanador se ha generado con éxito. Este método recibe como parámetro la función **resolve** tal cual como se generó, es decir, hay que pasar esta función al then con un parámetro que se corresponde a aquel que se envió cuando se resolvió la promesa. El parámetro de **resolve** es lo que mostramos en el alert del código de ejemplo (el mensaje "¡Éxito, es un número par!").

Por el contrario, si ha habido un error se puede encadenar la función **catch** que solo se alcanza si la promesa terminó con **reject.** De manera análoga a **then**, se define con la misma estructura que reject: una

función con un único parámetro que mostramos en el alert (cuyo resultando en este caso sería "¡Error, es un número impar!").

El método **then** ofrece una sintaxis alternativa y puede recibir directamente dos parámetros que se corresponden con las funciones **resolve** y **reject** tal y como se indica debajo.

```
decirGanador(5).then(data => alert(data)), error => alert(error));
```

En realidad es muy parecido al enfoque con **then/catch**, aunque en este caso el método then se encarga de manejar el error sin necesidad de llamar a **catch**.

Como se indicaba en la definición de promesas, lo realmente interesante de este tipo de objetos es encadenar efectos. Esto se puede hacer de manera sencilla, ya que cada método **then** puede devolver una promesa que se puede manejar posteriormente con tantos **then** y **catch** como queramos.

```
decirGanador(10)
.then(data1 => {
   alert(data1);
   return decirGanador(5);
})
.then(data2 => alert(data1))
.catch(error => alert(error));
```

Es decir, en el código anterior el primer **then** devuelve otra promesa a través del método decirGanador. Esta promesa se recoge encadenando otro **then** y esto se puede hacer tantas veces como promesas queramos secuenciar. Además, el **catch** al final se encarga de manejar cualquier error en las promesas anteriores.

Ten cuidado con alternar el **catch** entre los dos **then**.

```
decirGanador(10)
.then(data1 => {
   alert(data1);
   return decirGanador(5);
})
.catch(error => alert(error));
.then(data2 => alert(data1))
```

En este caso se ejecutaría el código asociado al método **catch** si la primera promesa “decirGanador(10)” se rechaza. Podríamos pensar que el segundo **then** no se ejecuta nunca si ha habido un error en el primero. Sin embargo, como veremos posteriormente los métodos **then** y **catch** devuelven automáticamente un **resolve** o **reject**, aunque no se especifique de manera explícita con una promesa. Por tanto, el segundo **then** se ejecutaría después del **catch** con el parámetro “data2” a **undefined**, ya que nunca le llega el resultado del primer **then** (el return decirGanador con parámetro 5) pero sí que le llega un **Promise.resolve** del catch (esto lo veremos con más detalle en los próximos párrafos).

Por último, puede que te hayas dado cuenta que una desventaja del uso de promesas es que tenemos que crear manualmente este tipo de objeto para poder controlar el código asíncrono. Esto no siempre es necesario y JavaScript nos facilita el trabajo con los métodos **Promise.resolve** y **Promise.reject**. Estos métodos se encargan de generar automáticamente lo que sería el resultado de una promesa con **resolve** y **reject** respectivamente. De esta manera, podemos llamar directamente al **then** o **catch** según el caso, tal y como se puede observar a continuación.

```
Promise.resolve("Prueba thenable").then(prueba1 => alert(prueba1));
```

Aquí realmente no tiene mucho sentido emplear **Promise.resolve** salvo que queramos forzar la ejecución de alguna sentencia en un código asíncrono. No obstante, cuando encadenamos varios **then** ocurre lo siguiente.

```
Promise.resolve("Prueba thenable")
.then(prueba1 => {
   alert(prueba1);
   return "Otra prueba";
})
.then(prueba2 => alert(prueba2))
.then(() => alert("Envié un último mensaje"));
```

Si te fijas en el código anterior, la única promesa que se genera es el **resolve** que forzamos para que devuelva el mensaje "Prueba thenable". En el primer **then** se devuelve otro mensaje, pero no es una promesa. Sin embargo, JavaScript convierte cualquier **return** dentro de un then automáticamente en un **Promise.resolve** (como ocurría en el catch anterior que tenía un método then detrás)**.** De esta manera, el segundo then recibe el método **resolve** con el parámetro "Otra prueba" como si lo hubiera generado una promesa.

Por otro lado, el segundo **then** no devuelve nada, por tanto el último método **then** recibiría **undefined**, pero se ejecuta igualmente y en este caso le pasamos una función sin parámetros y generamos un alert con un texto que no se recibe como parámetro.

Si quisiéramos generar un mensaje de error sin necesidad de crear otra promesa, podemos llamar a `return Promise.reject(...)` y en este caso manejarlo a través del segundo parámetro de **then** o directamente con **catch** como hemos visto anteriormente.

En cualquier caso, lo importante es que nos basta con generar una única promesa al principio de la secuencia. El resto del código ya se encarga JavaScript automáticamente de convertirlo al tipo de datos adecuado para manejar tantas secuencias como queramos.

5.2. Promise.all y Promise.race

Hasta ahora hemos visto cómo encadenar varias promesas y que cada una se ejecute justo después de la otra.

Existen dos métodos que pueden recibir un array con promesas y generar el **then** cuando se cumpla una condición común a todas ellas.

Por un lado, tenemos **Promise.all** que solo genera el **then** cuando todas las promesas del array que reciben como parámetro hayan finalizado con **resolve**. Lo veremos mejor con un ejemplo:

```
var p1 = new Promise((resolve, reject) => { setTimeout(resolve, 4000, 'una'); });
var p2 = new Promise((resolve, reject) => { setTimeout(resolve, 1000, 'dos'); });
var p3 = new Promise((resolve, reject) => { setTimeout(resolve, 3000, 'tres'); });
var p4 = new Promise((resolve, reject) => { setTimeout(resolve, 2000, 'cuatro'); });
var p5 = new Promise((resolve, reject) => { setTimeout(resolve, 5000, 'cinco'); });

Promise.all([p1, p2, p3, p4, p5])
.then(values => { alert(values)}, reason => { alert( reason)});
```

En el código del cuadro de arriba el **then** solo se ejecutará si terminan con **resolve** las cinco promesas anteriores. El método **then** recibe como parámetro un array con los objetos que ha ido generando cada **resolve** (en este caso un array con los mensajes ['una', dos', 'tres', cuatro', 'cinco']).

Si tuviéramos el siguiente escenario…

```
var p1 = new Promise((resolve, reject) => { setTimeout(resolve, 4000, 'una'); });
var p2 = new Promise((resolve, reject) => { setTimeout(resolve, 1000, 'dos'); });
var p3 = new Promise((resolve, reject) => { setTimeout(resolve, 3000, 'tres'); });
var p4 = new Promise((resolve, reject) => { setTimeout(resolve, 2000, 'cuatro'); });
var p5 = new Promise((resolve, reject) => { reject("promesa 5 fallida"); });

Promise.all([p1, p2, p3, p4, p5])
.then(values => { alert(values)}, reason => { alert( reason)});
```

En este caso se ejecuta siempre el segundo parámetro del **then** que maneja el error, cuyo mensaje será el correspondiente al **reject** que ha ocasionado que no todas las promesas finalicen con **resolve**.

El método **Promise.all** es especialmente interesante cuando queremos ejecutar varios efectos o instrucciones asíncronas y esperar a que finalicen todas.

De manera análoga, **Promise.race** tiene una estructura similar, pero en este caso el método **then** optará por la promesa que ha finalizado antes.

```
var p1 = new Promise((resolve, reject) => { setTimeout(resolve, 4000, 'una'); });
var p2 = new Promise((resolve, reject) => { setTimeout(resolve, 1000, 'dos'); });
var p3 = new Promise((resolve, reject) => { setTimeout(resolve, 3000, 'tres'); });
var p4 = new Promise((resolve, reject) => { setTimeout(resolve, 2000, 'cuatro'); });
var p5 = new Promise((resolve, reject) => { setTimeout(resolve, 5000, 'cinco'); });

Promise.race([p1, p2, p3, p4, p5])
.then(value => { alert(value)}, reason => { alert( reason)});
```

Es decir, la primera promesa que termina en este caso es p2, ya que el `setTimeout` finaliza en solo un segundo. El resultado que recibe **then** es aquel **resolve** de la promesa que haya finalizado primero.

Si se diera el siguiente caso:

```
var p5 = new Promise((resolve, reject) => { setTimeout(resolve, 500, "cinco"); });
var p6 = new Promise((resolve, reject) => { setTimeout(reject, 100, "seis"); });

Promise.race([p5, p6])
.then(value => { alert(value); }), reason => { alert(value));
```

Como el **reject** se genera primero, se ejecuta directamente el código del segundo parámetro del **then** para manejar el error.

En esta unidad estamos viendo promesas asociadas a efectos y animaciones, pero las promesas también se pueden emplear para cargar ficheros u otros recursos. Si tenemos varias URL de servidores, se puede emplear el fichero que carga más rápido. De ahí una de las mayores utilidades de **Promise.race**.

6. Uso de async/await

El uso de promesas con **then** y **catch** nos evita la pirámide de callbacks que veíamos al comienzo del apartado anterior. Sin embargo, el código puede resultar difícil de gestionar cuando encadenamos varios **then**.

Para facilitar el desarrollo, el estándar ECMAScript2017 ha incluido una nueva sintaxis para manejar promesas a través de los modificadores **async/await**. Básicamente la idea es el código "espere" a que haya finalizado cada promesa sin apenas necesidad de crear bloques adicionales. Veamos un ejemplo.

```
function miPrimeraPromesa() {
   return new Promise((resolve, reject) => {
      setTimeout(() => resolve("¡Éxito!"), 2000);
   });
}

async function generarAsync() {
   console.log("Antes de la promesa");

   // miPrimeraPromesa es una función que devuelve un objeto promesa
   let resultado = await miPrimeraPromesa();

   console.log("Después de la promesa y devuelve: ${resultado}");
}

generarAsync();
```

Lo que hace el código anterior es esperar a que finalice el **setTimeout** de la promesa. El resto del código queda "bloqueado" hasta que termina la ejecución y sin necesidad de crear bloques **then/catch**. La variable "resultado" contiene el **resolve** que ha generado la promesa. El equivalente al **then** se consigue añadiendo el modificador **await** por delante de la llamada a la función que devuelve una promesa.

La gran desventaja de este enfoque es que todo el código asíncrono y la llamada a promesas con **await** se debe ejecutar dentro de una función con el modificador **async**. No obstante, en muchos casos se simplifica el código respecto a la secuencia de **then** y **catch**.

¿Y qué ocurre en caso de error? Vamos a ver el código de ejemplo del apartado anterior, ahora con **async** y **await**. En su momento teníamos una función que devolvía una promesa con un **resolve** si el parámetro era un número par y un **reject** si era impar. Recordemos el código:

```
function decirGanador(numero) {
   return new Promise((resolve, reject) => {
      setTimeout(function () {
         if (numero % 2 == 0) {
            resolve("¡Éxito, es un número par!");
         } else {
            reject("¡Error, es un número impar!");
         }
       }, 5000);
   });
}
```

Ahora nos creamos una función con **async** para generar la secuencia.

```
async function procesarJuego(numero) {
   let mensaje;
   try {
      mensaje = await decirGanador(numero);

      alert(mensaje);
   } catch (error) {
      alert(error);
   }
}
```

Como se refleja en el código, si la promesa devuelve **reject** podemos crear un bloque **try/catch** y manejar el error como si fuera otro cualquiera del código JavaScript. En definitiva, el código espera 5 segundos con await y en función de si se llama a **resolve** o **reject** seguirá en el try o saltará al catch.

Si quisiéramos encadenar varias "partidas", necesitamos crear otra función con **async** y un **await** por cada llamada para garantizar que las promesas se ejecutan en el orden establecido (procesarJuego es la función definida en el bloque anterior y que llama a la función que devuelve la promesa).

```
async function probar() {
   await procesarJuego(10);
   await procesarJuego(5);
}

probar();
```

En definitiva, la estructura del código con **async/await** toma el siguiente patrón (en este caso creamos una variable por cada resultado, pero podríamos crear un array o similares):

```
async function generarSecuencia() {
   const results1 = await generarPromesa('Parámetro 1');
   const results2 = await generarPromesa('Parámetro 2');
   const results3 = await generarPromesa('Parámetro 3');
   const results4 = await generarPromesa('Parámetro 4');
   const results5 = await generarPromesa('Parámetro 5');
   const finalResults = await generarPromesa('Parámetro 6');

   return finalResults;
}

generarSecuencia();
```

7. Otras herramientas

Hay librerías que han estado presentes durante años y nos han permitido ahorrar código y que además sea compatible en todos los navegadores. Una de ellas es **jQuery**, que tiene una amplia gama de efectos para mostrar y ocultar información o realizar animaciones complejas sin apenas emplear código.

Sin embargo, las últimas especificaciones de CSS y JavaScript eliminan la necesidad de emplear librerías externas. Por ejemplo, el código siguiente permite mostrar un div inicialmente oculto con un evento de desvanecimiento

```
$("button").click(function () {
    $("div:nth-of-type(1)").fadeIn();
)};
```

Sí que es cierto que no hay nada en JavaScript que permita implementar este efecto, no obstante, hoy en día se puede hacer con tan pocas líneas de código que en la mayoría de ocasiones no interesa importar una librería que añade una gran cantidad de funcionalidades que no vamos a necesitar, ya que ralentiza el rendimiento de la aplicación y del navegador.

Otras librerías como **animate.css, move.js** o **velocity.js** están más especializadas en efectos y animaciones y se pueden incluir sin consumir tantos recursos.

No obstante, cuanto más se pueda implementar de forma nativa sin librerías más eficiente será el desarrollo. De hecho, hay una tecnología en desarrollo aún experimental en algunos navegadores que nos permitirá crear animaciones solamente con JavaScript. Aquí se puede encontrar la especificación: https://www.w3.org/TR/web-animations-1/