AnimeJS

Introducción a la animación en JS

Algo de historia

Hasta ahora todo lo que hemos visto relacionado con animación, tiene que ver con CSS.

Haciendo historia, en realidad la animación ya se daba en la web con dos tecnologías:

SMIL que es una parte del spec de SVG para animaciones SVG declarado en el Markup (un poco en desuso hoy en día) y,

jQuery nos sorprendió cuando nació con la posibilidad de animar elementos HTML controlado por JS.

Hoy en día tenemos las transiciones y animaciones de CSS, que funcionan muy bien, se interpretan de manera nativa en el navegador y por tanto tienen bastante performance.

¿Qué librerías hay y para qué valen?

- Popmotion: Es una librería muy potente, basada en los conceptos de programación reactiva RxJS.
- GSAP: Es una librería muy potente, que mezcla una sintaxis sencilla con una gran performance, además de bastantes plugins para hacer interacciones y una buena documentación.
- AnimeJS: Es la librería que vamos a analizar en el curso. Los conceptos son bastante similares a GSAP, con una muy buena documentación y un gran soporte.
- Velocity.js: Una librería alternativa estándar con bastante experiencia y con una documentación muy extensiva.

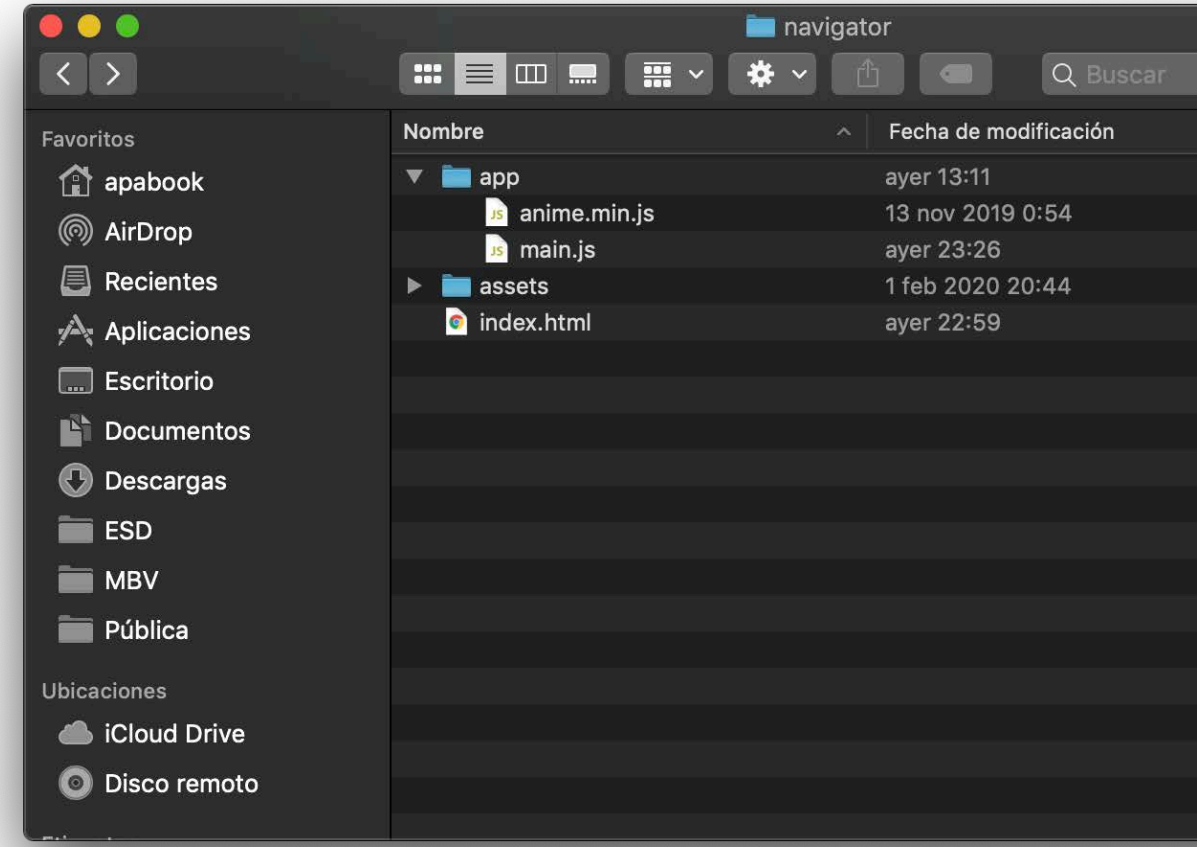
Instalación de AnimeJS

Podemos instalar animeJS por NPM si estamos usando algún bundler, que no es nuestro caso. O descargando la librería y conectándola a nuestro proyecto.

Las instrucciones de instalación las encontramos en github:
https://github.com/juliangarnier/anime/#download

Nos dan una opción de descarga manual, en caso de que no usemos NPM. Una vez esté descargado, en la carpeta raíz, seguimos la ruta /lib/anime.min.js.

Esta carpeta podemos copiarla a nuestro proyecto local, y la vinculamos en HTML justo antes del script main.js (nunca después, si no no funcionaría).



Animación básico

A la hora de hacer una animación básica, lo vamos a hacer de este modo y explicamos las propiedades.

En este objeto tenemos múltiples posibilidades, y es donde se define toda la animación, encontramos 3 tipos de propiedades importantes:

```
let animatingBlock = document.querySelector( '.block' );

let animation = anime({
    // qué elementos animamos?
    targets: [animatingBlock],

    // cómo animamos?
    duration: 1500,
    delay: 1000,
    direction: 'alternate',
    loop: 3,
    endDelay: 1000,
    easing: 'easeInOutSine',

    // qué propiedades?
    translateX: 500
});
```

Qué animamos: Targets

Son los elementos que se van a animar.

Esta propiedad permite muchos tipos de sintaxis, pero recomiendo por cuestiones de orden la expuesta.

a) Seleccionamos DOM Elements con querySelector o querySelectorAll y los almacenamos en variables,

b) Metemos estas variables en targets como array.

Hay otros métodos pero este es el más ordenado

```
let animatingBlock = document.querySelector( '.block' );

let animation = anime({
    // qué elementos animamos?
    targets: [animatingBlock],

    // cómo animamos?
    duration: 1500,
    delay: 1000,
    direction: 'alternate',
    loop: 3,
    endDelay: 1000,
    easing: 'easeInOutSine',

    // qué propiedades?
    translateX: 500
});
```

Cómo animamos

Son propiedades que indican valores como duración, retraso, easing, etc… Vamos a repasarlas:

- **Duration**: Es la duración de la animación
- **Delay**: Es el tiempo que tarda la animación en lanzarse desde la invocación de la función.
- **Direction**: Es la dirección de la animación. Tiene 3 opciones: “normal”, que es por defecto y ejecuta la animación en una dirección tal y como está programada, “reverse”, que ejecuta la animación al revés, y “alternate” que ejecuta la animación en caso de que se haga varias veces cada vez hacia una dirección.

```
let animatingBlock = document.querySelector( '.block' );

let animation = anime({
    // qué elementos animamos?
    targets: [animatingBlock],

    // cómo animamos?
    duration: 1500,
    delay: 1000,
    direction: 'alternate',
    loop: 3,
    endDelay: 1000,
    easing: 'easeInOutSine',

    // qué propiedades?
    translateX: 500
});
```

Cómo animamos

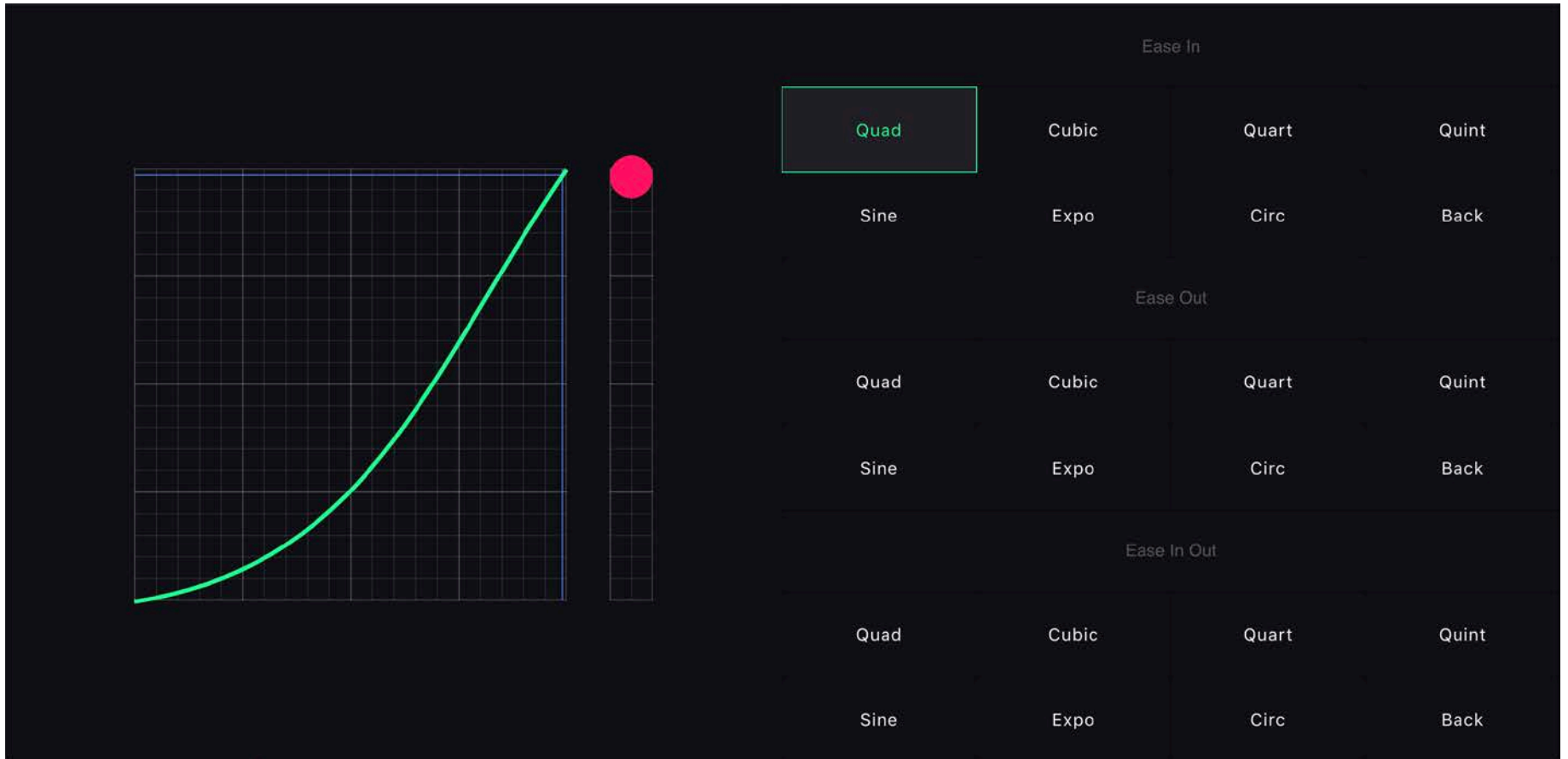
- **Loop**: Cuántas veces se ejecuta la animación. Por defecto 1, pero pueden ser más, o incluso “infinite” veces.
- **EndDelay**: Si delay es el tiempo de retardo entre la invocación de la función y la ejecución de la misma, endDelay es el tiempo de retardo entre loops.
- **Easing**: Es el tipo de timing function, que ya nos suenan de las animaciones CSS.

```
let animatingBlock = document.querySelector( '.block' );

let animation = anime({
    // qué elementos animamos?
    targets: [animatingBlock],

    // cómo animamos?
    duration: 1500,
    delay: 1000,
    direction: 'alternate',
    loop: 3,
    endDelay: 1000,
    easing: 'easeInOutSine',

    // qué propiedades?
    translateX: 500
});
```

Easings

Para analiza que tipo de timing functions tenemos disponibles, recomiendo mirar este Codepen de Julian Garnier, creador de animeJS. https://codepen.io/juliangarnier/full/mWdraw

Los sistemas de nombrado son así: Ease In + Quad, se nombra como “easeInQuad”, o por ejemplo Ease In Out + Quart se nombra “easeInOutQuart”.



Para ver los nombres bien y alguna opción más:

https://animejs.com/documentation/#linearEasing

Qué propiedades animamos

A nivel básico se puede animar cualquier propiedad CSS que sea animable (numéricas, de color, etc…), y se pueden animar algunas propiedades DOM (que esto es muy útil).

```
anime({
    // qué elementos animamos?
    // ...

    // propiedades CSS
    backgroundColor: '#333',
    color: '#fff',
    fontSize: '1.3rem',

    // transformaciones
    translateX: '40px',
    translateY: '50px',
    rotate: '2turn', // o '45deg',
    scale: '1.2',

    // DOM
    innerHTML: [0, 100],
    round: 1
});
```

Qué propiedades animamos - Propiedades CSS

Las propiedades que animamos relacionadas con CSS son las mismas que usamos en el Objeto Style de JavaScript.

Estas son las mismas que en CSS pero con un cambio de nomenclatura, por ejemplo: “background-color” no es una propiedad con un nombre permitido en JS, porque el carácter “-“ no está permitido.

De modo que cuando tenemos propiedades de dos palabras, se pasan a camelCase. “background-color” se convierte en backgroundColor, así sucesivamente.

```
anime({
    // qué elementos animamos?
    // ...

    // propiedades CSS
    backgroundColor: '#333',
    color: '#fff',
    fontSize: '1.3rem',

    // transformaciones
    translateX: '40px',
    translateY: '50px',
    rotate: '2turn', // o '45deg',
    scale: '1.2',

    // DOM
    innerHTML: [0, 100],
    round: 1
});
```

Qué propiedades animamos - Transformaciones

Aunque la propiedad transform se puede tocar desde JS, hay unos atajos que nos da AnimeJS como translateX, translateY, rotate o scale.

Si queremos mover un objeto, se recomienda tocar estos valores que están creados para hacer animaciones sin coste de performance y funcionan muy bien con SVG. Podemos usar la opacidad también, cambios de color sombra, etc..

Nunca recomendaría animar margins, paddings o tipografía, no funciona bien a nivel de animación.

```
anime({
    // qué elementos animamos?
    // ...

    // propiedades CSS
    backgroundColor: '#333',
    color: '#fff',
    fontSize: '1.3rem',

    // transformaciones
    translateX: '40px',
    translateY: '50px',
    rotate: '2turn', // o '45deg',
    scale: '1.2',

    // DOM
    innerHTML: [0, 100],
    round: 1
});
```

Varios valores

Sintaxis array

En algunos momentos queremos elaborar la animación un poco más, y queremos no sólo meter un valor al que vamos a ir, si no que queremos meter el valor desde el que vamos, o varios valores intermedios.

Sintaxis objeto

En caso de querer cambiar algo más que los valores, podemos hacer un objeto donde para esa propiedad en concreto sobreescribimos valores de duración, easing, etc…

```
let animatingBlock = document.querySelector('.block');

let animation = anime({
    targets: [animatingBlock],
    duration: 1500,
    delay: 500,
    easing: 'easeInOutSine',

    translateX: [100, 0],
    translateY: [0, 200, 100, 0],

    backgroundColor: {
        value: ['#eee', '#333', '#eee'],
        duration: 2500,
        easing: 'easeInOutQuad',
    }
});
```

Añadiendo algo de interactividad

Hasta ahora hemos visto solamente animaciones que se lanzan sólas, pero esto va a pasar no muchas veces en un proyecto real. Lo más habitual es ver cómo las animaciones se lanzan tras un evento.

Autoplay: false & play()

Esta propiedad consigue que la animación no se lance sóla. Si no que se lance cuando se invoque el método play.

```
let animatingBlock = document.querySelector('.block');

let animation = anime({
    // ...
    // control
    autoplay: false,
});

// con evento
document.addEventListener('click', () => {
    animation.play();
});
```

Añadiendo algo de interactividad

Hasta ahora hemos visto solamente animaciones que se lanzan sólas, pero esto va a pasar no muchas veces en un proyecto real. Lo más habitual es ver cómo las animaciones se lanzan tras un evento.

Autoplay: false & play()

Esta propiedad consigue que la animación no se lance sóla. Si no que se lance cuando se invoque el método play.

```
let animatingBlock = document.querySelector('.block');

let animation = anime({
    // ...
    // control
    autoplay: false,
});

// con evento
document.addEventListener('click', () => {
    animation.play();
});
```

Uso de play() y pause() con un booleano

En este ejemplo vemos como funciona un play con pause.

Lo que hacen es simplemente interrumpir la animación o continuarla.

```
let animatingBlock = document.querySelector('.block');
let animation = anime({
    // ...
});
let paused = false;

document.addEventListener('click', () => {
    if (!paused) {
        animation.play();
        paused = true;
    } else {
        animation.pause();
        paused = false;
    }
});
```

Uso de play() y reverse() – Toggling animation

En muchos casos el tipo de animación que queremos hacer con eventos, y que la podemos ver en los ejemplos, es la de un toggle o un switcher. Es decir una animación que se lanza en una dirección en un click, y que se puede cancelar en otro click revirtiendo la animación.

Para ello animeJS necesita un par de pasos previos. Comentamos el código. Tenéis bastantes ejemplos a disposición.

```
let animatingBlock = document.querySelector('.block');
let animation = anime({
    // ...
});

// con evento
document.addEventListener('click', () => {
    if (animation.began) {
        animation.completed = false;
        // prevents animation.reset()
        animation.reverse();
    }
    if (animation.paused) {
        animation.play();
    }
});
```

Uso de play() y reverse() – Toggling animation

Animation.play(): Método que lanza la animación

Animation.reverse(): Método que cambia la propiedad direction

Animation.paused: Es un valor booleano que dice que es una animación que no ha comenzado o que está pausada.

Animation.began: Es un valor booleano que dice que es una animación que ha comenzado.

Animation.completed = false. Esto lo pongo (y recomiendo ponerlo) porque cuando una animación concluye y le damos al play(), por defecto anime resetea la animación y puede dar algún resultado indeseado. (Ejemplo).

```
let animatingBlock = document.querySelector('.block');
let animation = anime({
    // ...
});

// con evento
document.addEventListener('click', () => {
    if (animation.began) {
        animation.completed = false;
        // prevents animation.reset()
        animation.reverse();
    }
    if (animation.paused) {
        animation.play();
    }
});
```

Callbacks y promesas

Hay callbacks y promesas para controlar las fases diferentes de animeJS, por si necesitamos coordinar las animaciones con otras funcionalidades en nuestro programa.

Sobre callbacks y promesas tenemos información en la web oficial, que recomiendo leer: https://animejs.com/documentation/#update

```
let animation = anime({
    // ...
    // callbacks
    begin: (anim) => {
        console.log(anim);
    },
    update: (anim) => {
        console.log(anim);
    },
    complete: (anim) => {
        console.log(anim);
    }
});

animation.finished.then(function () {
    console.log('animation concluded...')
});
```

Timelines

Los timelines son sistemas de agrupación de animaciones, donde tratamos varias animaciones como si fueran una sola.

Esto es muy útil porque podemos trabajar con animaciones secuenciadas de varios elementos, y podemos concatenarlas.

El timeline se crea con el método timeline, y se pasa un objeto con las propiedades que sean comunes a todos los keyframes.

Luego se detallan los keyframes con el término add. Este método acepta dos parámetros, uno que es el objeto con las propiedades de animación como si fuera un objeto anime normal, y un segundo llamado offset.

```
let animatingBlocks = document.querySelectorAll('.block');

let animation = anime.timeline({
    duration: 1500,
    easing: 'easeInOutSine'
});
animation
    .add({
        targets: [animatingBlocks[0]],
        translateX: [100, 500, 50, 0],
        translateY: [0, 200, 150, 0],
    })
    .add({
        targets: [animatingBlocks[1]],
        translateX: [50, 30, 200, 0],
        translateY: [0, 50, 400, 0],
        rotate: { value: '1turn', delay: 500 }
    }, '+=200')
    .add({
```

Timelines - offset

Offset puede ser un número o un string, yo recomiendo un string.

Nos permite lanzar los siguientes keyframes con un offset de tiempo.

Si es negativo, se lanza antes de acabar la animación anterior, si es positivo, es un delay entre animaciones.

```
animation
    .add({
        targets: [animatingBlocks[0]],
        translateX: [100, 500, 50, 0],
        translateY: [0, 200, 150, 0],
    })
    .add({
        targets: [animatingBlocks[1]],
        translateX: [50, 30, 200, 0],
        translateY: [0, 50, 400, 0],
        rotate: { value: '1turn', delay: 500 }
    }, '+=200')
    .add({
        targets: [animatingBlocks[2]],
        translateX: [100, 500, 50, 0],
        translateY: [0, 200, 150, 0],
        rotate: '-3turn'
    }, '-=400')
    .add({
```

Ejemplos avanzados

Dejo algunos ejemplos avanzados que recomiendo que estudiéis para la realización del ejercicio P2P de la sección.