

# Drag and drop: InteractJS

Interacción gestual en Interfaces Web

---

## El unicornio del dragdrop

---

Desde hace bastantes años había **bastantes interfaces diseñadas con el patrón de arrastrar y soltar**.

En el caso de la web se introduce el drag drop con la inclusión HTML5, aunque previamente con jQueryUI o Dojo o alguna librería más basada en jQuery.

Esto era posible gracias a **complejas detecciones de mouseEvents internas** y aplicaciones de los movimientos resultantes de estos eventos a **transformaciones CSS** o a modificaciones de coordenadas top, bottom, left, right, en posicionamiento absoluto.

Era **bastante complicado de programar** y no nos asegurábamos que fuera a funcionar bien en todos los navegadores.



# Librerías gestuales para Drag and Drop

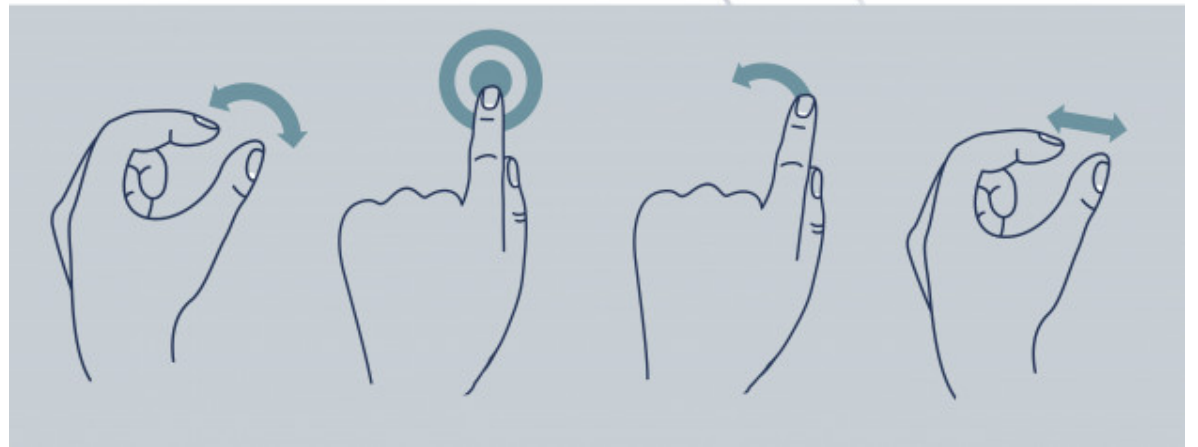
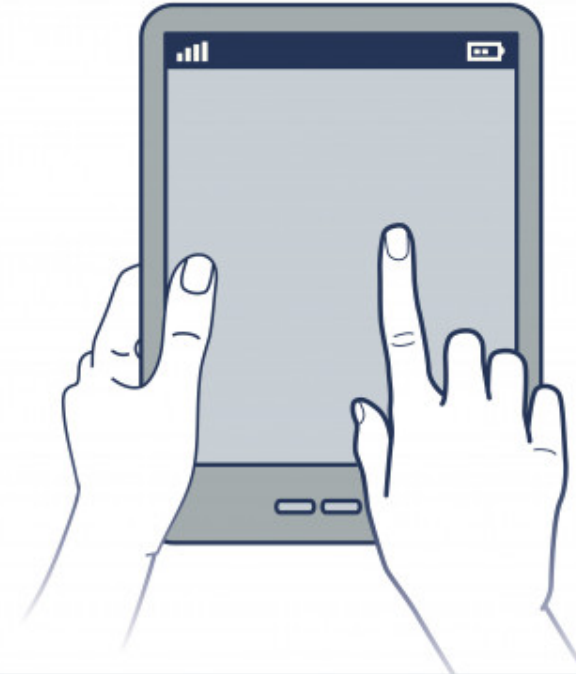
---

InteractJS soluciona este problema.

Es básicamente una librería de drag, drop, resize y gestos en el navegador que funciona desde IE9 en adelante y que tiene una API sencilla e intuitiva.

Se puede instalar desde NPM o yarn en caso de usar módulos, o si no desde un CDN como pone en la página de GitHub del proyecto.

<https://github.com/taye/interact.js>



Vector hand icons - touchscreen interface  
illustration Premium Vector  
venimo

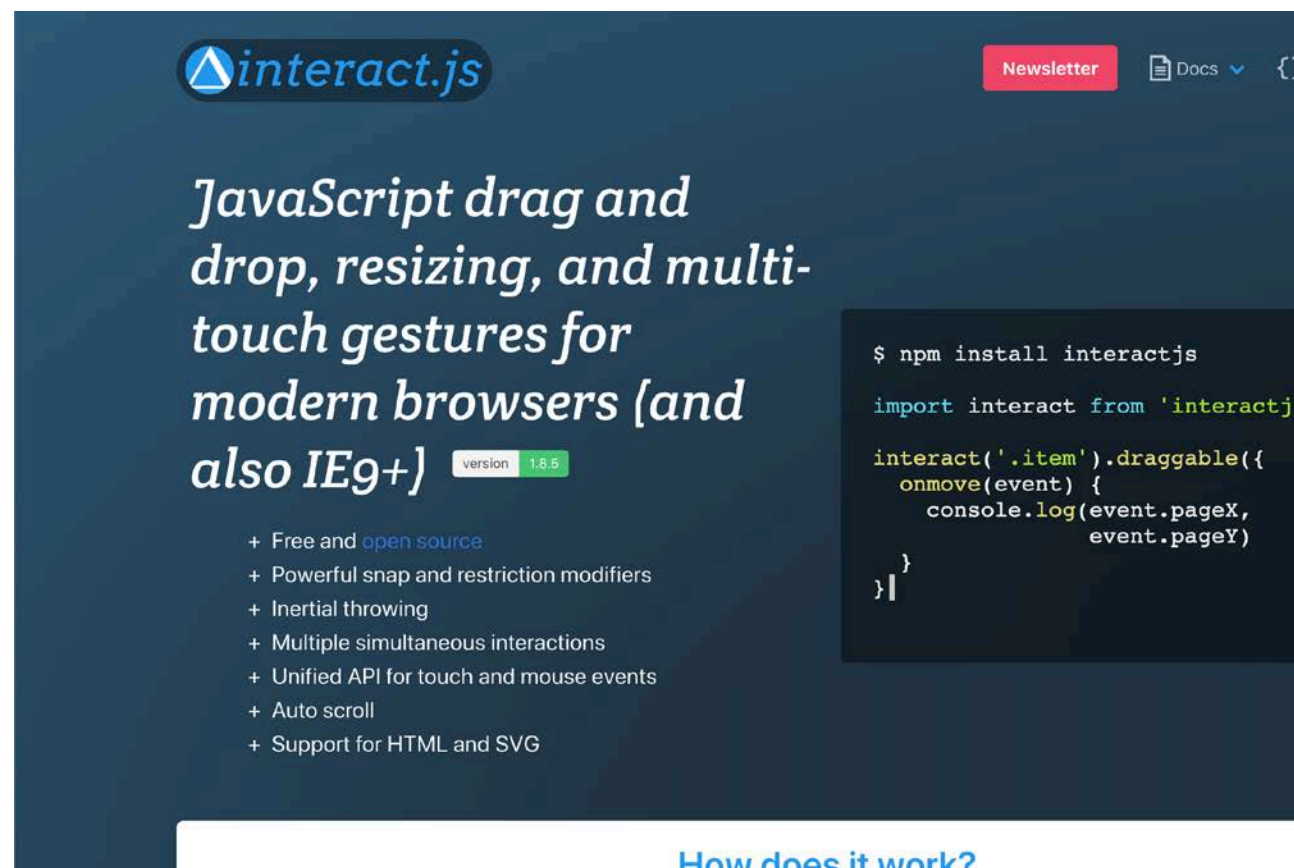
## Iniciar Interact

Para iniciar Interact tenemos que hacerlo con la función `interact()`, donde pasamos como argumento el selector del elemento que queremos que tenga las acciones de interact.

**En principio vamos a trabajar con un elemento. Para trabajar con varios, recomiendo pasar a un sistema de clases.** Luego vemos eso.

Para crear un elemento interactable. Podemos hacerlo con string selector, o con un `HTMLElement`.

```
// Interactable  
let dragElement = interact('.drag_element');
```



The screenshot shows the official website for Interact.js. At the top, there is a navigation bar with the Interact.js logo, a 'Newsletter' button, and links to 'Docs' and a GitHub repository. The main heading reads 'JavaScript drag and drop, resizing, and multi-touch gestures for modern browsers (and also IE9+)'. Below this, a list of features is provided: '+ Free and open source', '+ Powerful snap and restriction modifiers', '+ Inertial throwing', '+ Multiple simultaneous interactions', '+ Unified API for touch and mouse events', '+ Auto scroll', and '+ Support for HTML and SVG'. To the right, a code block shows the installation and usage of the library: `$ npm install interactjs`, `import interact from 'interactjs'`, and `interact('.item').draggable({ onmove(event) { console.log(event.pageX, event.pageY) } })`. At the bottom right, a link 'How does it work?' is visible.

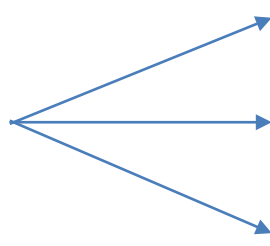
# Eventos y modificadores

---

Una vez que hemos creado el elemento interactable, podemos hacer dos cosas:

A) **Indicar el tipo de interacción** que queremos hacer **con algunas configuraciones** interesantes,

B) y **añadir eventos** en función del tipo de interacción.



Objeto draggable

Objeto dropzone

Objeto resizable

Dragstart, dragend, dragmove...

Dropmove, drop...

Resizestart...

Importante recalcar:

Interact no es plug and play, es decir, no es conectar la librería y todo funciona, si no que hay que trabajar un poquito. Pero por eso quizá es más interesante.

## Objeto Draggable

---

Es la interacción más simple. Pero tiene algunos puntos interesantes que contar.

Para ello, vamos a hacer un ejemplo que vamos a ir complicando para tener una perspectiva de aquellas cosas que podemos configurar en un objeto (u objetos) draggable.

## Iniciar Draggable

---

Para inicializar un objeto Draggable, simplemente le pasamos el método draggable y un objeto de configuración como argumento:

Así como en el **HTML** crear el elemento correspondiente. Y en **CSS** la librería nos hace una recomendación para mejorar la UI.

El mero hecho de hacer estas operaciones, hace que el sistema de dragging ya “funcione”. Si nos fijamos en el navegador, ya cambia el cursor cuando pasamos ratón por encima, etc..

Al intentar arrastrar el elemento, nos salta un warning en consola diciendo que “no puede hacer nada si no ponemos listeners”. **Vamos a ver qué es eso de los listeners.**

```
<div class="draggable">drag</div>
```

```
let dragElement = interact('.draggable');  
dragElement.draggable({});
```

```
.draggable {  
  //...  
  touch-action: none;  
  user-select: none;  
}
```

## Iniciar eventos

---

Para que el sistema de drag drop funcione, a parte de inicializar la librería, tenemos que hacer que el elemento tenga listeners a los eventos de dragstart, dragmove y dragend.

Observamos en consola como ahora sí tenemos eventos que funcionan cuando hacemos dragstart, dragmove o dragend en nuestro objeto.

```
let dragElement = interact('.draggable');
dragElement.draggable({
  listeners: {
    start(ev) {
      console.log(ev);
    },
    move(ev) {
      console.log(ev);
    },
    end(ev) {
      console.log(ev);
    }
  }
});
```



## Objeto evento – ev.target

---

Pasando el objeto evento por consola, podemos ver algunas propiedades interesantes:

Como `ev.target`, que nos indica el elemento que estamos arrastrando.

```
dragElement.draggable({  
  listeners: {  
    move(ev) {  
      console.log(ev.target);  
    }  
  }  
});
```

## Objeto evento – ev.client

---

Ev.client, que es un objeto que tiene la posición x e y de dónde se encuentra nuestro puntero arrastrando en el viewport, y que podemos usar para hacer nuestra primera transformación.

Vemos que haciendo esto, podemos actualizar al propiedad CSS Transform, y hacer un arrastre, no muy elaborado, pero es un inicio.

```
dragElement.draggable({  
  listeners: {  
    move(ev) {  
      console.log(ev.client);  
      ev.target.style.transform = `translate(${ev.client.x}px, ${ev.client.y}px)`;  
    }  
  }  
});
```

## Objeto evento – ev.delta

Para mejorar esto, vamos a usar un concepto que se usa mucho en eventos continuos como el de arrastre (la gente que diseña videojuegos conoce muy bien este concepto), que son las deltas.

La delta es la distancia en x e y que recorre el puntero desde que se lanza un evento, hasta el siguiente.

Si en lugar de usar client.x o client.y, lo que hacemos es sumar a la transformación las delta.x y delta.y correspondientemente, conseguimos un arrastre más natural, porque los movimientos no se calculan en absoluto desde el viewport, si no en relativo desde el elemento actualizando su posición en función del movimiento de elemento apuntador (dedo, ratón, etc...).

```
let dragElement = interact('.draggable');
let dPos = { x: 0, y: 0 };

dragElement.draggable({
  listeners: {
    move(ev) {
      dPos.x += ev.delta.x;
      dPos.y += ev.delta.y;
      ev.target.style.transform = `
        translate(${dPos.x}px, ${dPos.y}px)
      `;
      console.log(ev.delta);
    }
  }
});
```

## Objeto evento – ev.delta

El uso de deltas, nos obliga a crear un objeto que almacene la posición acumulada de deltas. Esto es, necesitamos un objeto que guarde la transformación anterior, para sumarle la nueva. En este caso es dPos.

De este modo hemos visto algunas de las propiedades del objeto evento en draggable.

```
let dragElement = interact('.draggable');
let dPos = { x: 0, y: 0 };

dragElement.draggable({
  listeners: {
    move(ev) {
      dPos.x += ev.delta.x;
      dPos.y += ev.delta.y;
      ev.target.style.transform = `
        translate(${dPos.x}px, ${dPos.y}px)
      `;
      console.log(ev.delta);
    }
  }
});
```

## Dragging con varios elementos

---

Lo que hemos hecho vale para un elemento. Si ponemos varios:

Observamos que vamos a tener un arrastre raro, ya que al comenzar a arrastrar un elemento, el objeto dPos ya tiene una información que corresponde con el último elemento arrastrado.

Esto se puede salvar poniendo dos objetos dPos: dPos1 y dPos2, y haciendo algo así:

```
<div class="app">  
  <div class="draggable">drag</div>  
  <div class="draggable">drag</div>  
</div>
```

## Dragging con varios elementos

```
let dragElement = interact('.draggable');
let dPos1 = { x: 0, y: 0 };
let dPos2 = { x: 0, y: 0 };

dragElement.draggable({
  listeners: {
    move(ev) {
      if (ev.target.classList.contains('d1')) {
        dPos1.x += ev.delta.x;
        dPos1.y += ev.delta.y;
        ev.target.style.transform = `
          translate(${dPos1.x}px, ${dPos1.y}px)`;
      } else if (ev.target.classList.contains('d2')) {
        dPos2.x += ev.delta.x;
        dPos2.y += ev.delta.y;
        ev.target.style.transform = `
          translate(${dPos2.x}px, ${dPos2.y}px)`;
      }
    }
  }
});
```

Esto salvaría el día, pero no es escalable, porque si tenemos un número  $n$  de elementos cambiante, no podemos hacer nada.

## Dragging con varios elementos

---

Una opción sería almacenar cada dPosN como atributos en el elemento HTML, como atributos custom o como micro data, y guardar las transformaciones en esos atributos.

```
<div class="app">  
  <div class="draggable" xpos="0" ypos="0">drag</div>  
  <div class="draggable" xpos="0" ypos="0">drag</div>  
</div>
```

# Dragging con varios elementos

---

De este modo:

```
dragElement.draggable({  
  listeners: {  
    move(ev) {  
      let x = Number(ev.target.getAttribute('xpos')) + ev.delta.x;  
      ev.target.setAttribute('xpos', x);  
  
      let y = Number(ev.target.getAttribute('ypos')) + ev.delta.y;  
      ev.target.setAttribute('ypos', y);  
  
      ev.target.style.transform = `translate($ {x}px, $ {y}px)`  
    }  
  }  
});
```

Esta manera sería correcta, pero quizá un poco antigua.

Otra más nueva, sería usando clases.



## Dragging de varios elementos usando clases

```
class DraggingElement {  
  constructor(el) {  
    this.el = el;  
    this.x = 0;  
    this.y = 0;  
    this.interactElement = interact(this.el);  
    this.interactElement.draggable({  
      listeners: {  
        move: (ev) => {  
          this.x += ev.delta.x;  
          this.y += ev.delta.y;  
          this.el.style.transform = `  
            translate(${this.x}px, ${this.y}px)  
          `;  
        }  
      }  
    });  
  }  
}
```

De este modo podemos crear una clase que implemente la funcionalidad completa, guardando las transformaciones delta, y lo que necesitamos.

Sería la manera más interesante de hacer esto.

## Dragging de varios elementos usando clases

---

Y luego instanciamos cada elemento con su selector correspondiente.

```
let de1 = new DraggingElement('.draggable.d0');  
let de2 = new DraggingElement('.draggable.d1');
```

U otra opción sería hacerlo más genérico todavía, que cree instancias en función de la cantidad de elementos que encuentre.

```
let d = Array.from(document.querySelectorAll('.draggable'));  
let dElements = d.map(dItem => new DraggingElement(dItem));
```

## Dragstart y dragend

Aunque lógicamente el evento más importante en el drag, es dragmove. **Dragstart y end, se usan para, entre otras cosas, hacer algún cambio de clase, o alguna modificación interesante como por ejemplo que cambie el estilo cuando comienza el arrastre y lo cambie al terminar de nuevo.**

O que actualice el z-index de los elementos de modo que el elemento arrastrado sea el que esté más arriba:

```
class DraggingElement {  
  // ...  
  this.interactElement.draggable({  
    listeners: {  
      start: () => {  
        // ...  
      },  
      move: (ev) => { // ... },  
      end: () => {  
        // ...  
      },  
    },  
  });  
}
```

## Más opciones para el Objeto draggable

---

El objeto draggable tiene varias opciones muy interesantes que no vamos a comentar por falta de tiempo, que nos permiten hacer snap -que el arrastre no vaya continuo si no en cantidades de píxeles, o que la capacidad de arrastre esté restringida al elemento padre, o que tenga inercia al arrastrar.

Todas estas opciones, que tienen bastante que explicar, vienen bien ejemplificadas en la documentación oficial:

<https://interactjs.io/docs/modifiers>

```
class DraggingElement {  
  // ...  
  this.interactElement.draggable({  
    inertia: true,  
    modifiers: [  
      interact.modifiers.restrictRect({  
        restriction: 'parent',  
        endOnly: true  
      })  
    ],  
    listeners: {  
      // ...  
    }  
  })  
}
```

## Objeto dropzone

---

Habitualmente, aquellos elementos que arrastramos en una UI, es porque queremos interactuar de modo que sucedan cosas, al soltar estos elementos en algún área en concreto.

```
<div class="app">
  <div class="draggable" xpos="0" ypos="0">
    drag
  </div>
  <div class="draggable" xpos="0" ypos="0">
    drag
  </div>
  <div class="droppable">
    drop
  </div>
</div>
```

## Objeto dropzone

---

Habitualmente, aquellos elementos que arrastramos en una UI, es porque queremos interactuar de modo que sucedan cosas, al soltar estos elementos en algún área en concreto.

Enter hace referencia al evento de entrar en el área arrastrando algo, leave lo contrario, abandonar el área, move es arrastrar algo en la dropzone, y drop, es cuando soltamos algo en la dropzone.

```
class DroppinElement {
  constructor(el) {
    this.el = el;
    this.interactElement = interact(this.el);
    this.interactElement.dropzone({
      listeners: {
        enter: (ev) => {
          console.log(ev);
        },
        move: (ev) => {
          console.log(ev);
        },
        leave: (ev) => {
          console.log(ev);
        },
        drop: (ev) => {
          console.log(ev);
        }
      }
    });
  }
}
```

## Objeto dropzone

---

Hay algunas opciones como `overlap` – que hace referencia a la cantidad de elemento arrastrado que tiene que entrar en la dropzone para lanzar el evento `enter`, o `accept` o `checker` que nos permiten manejar bien qué elementos dejamos hacer drop y cuales ignoramos.

La información la encontramos en este link de la documentación oficial:

<https://interactjs.io/docs/dropzone>

```
class DroppingElement {
  constructor(el) {
    this.el = el;
    this.interactElement = interact(this.el);
    this.interactElement.dropzone({
      listeners: {
        enter: (ev) => {
          console.log(ev);
        },
        move: (ev) => {
          console.log(ev);
        },
        leave: (ev) => {
          console.log(ev);
        },
        drop: (ev) => {
          console.log(ev);
        }
      }
    });
  }
}
```

## Hay algunas opciones más en esta librería

---

Resizable: Nos permite controlar eventos de resize sobre elementos (sidebars, widgets), que nos permite rehacer a nuestro gusto el layout de una UI. Es recomendable ver la documentación:

<https://interactjs.io/docs/resizable>

Gesturable: Que nos permite controlar de manera bastante precisa gestos sobre un dispositivo táctil, que daría quizá para otro video.

Os recomiendo que sigáis trabajando en esta librería, que aunque sea un poco compleja, es bastante completa y permite hacer “casi todo” que tenga que ver con eventos de interacción con una pantalla más allá de click o el teclado.