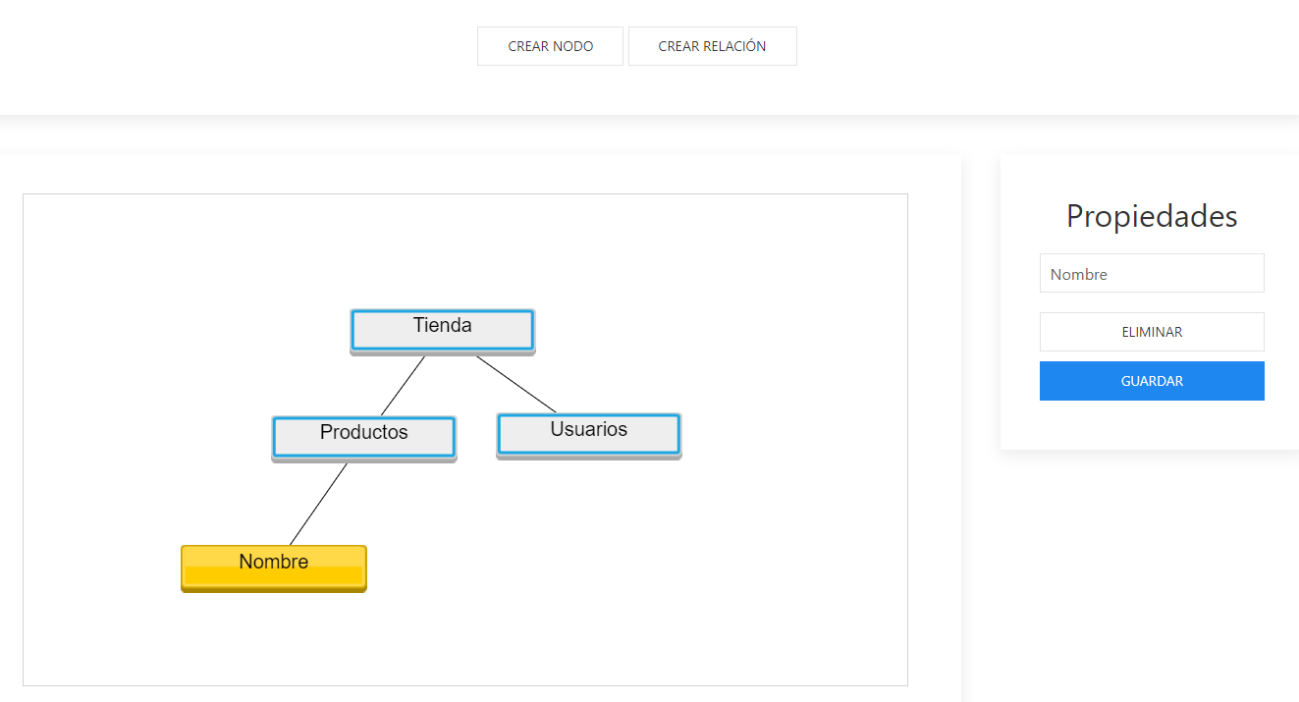
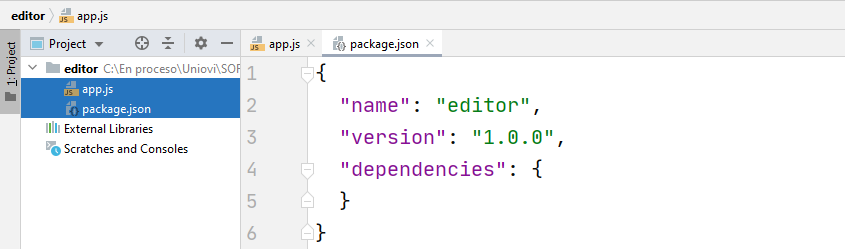
# Introducción

En esta práctica vamos a crear un editor de diagramas web muy simple, basado en el canvas de JS. El proyecto se desarrollará paso a paso y servirá como **punto de partida** para la creación de futuros editores.



# Proyecto Node

Creamos un **nuevo proyecto de Node** y un fichero **app.js** y un **package.json.**



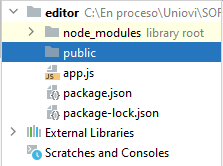
Accedemos a la línea de comandos en la ruta del proyecto e instalamos el módulo express.

|  |
| --- |
| npm install express@4.17.1 |

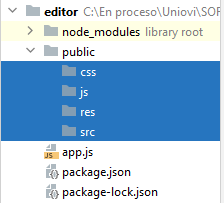
Accedemos a **app.js** y creamos una aplicación de express, con un directorio público.

|  |
| --- |
| const express = require('express'); const app = express(); const PORT = 3000;  app.use(express.static('public')); app.get('/', (req, res) => {  res.redirect("/index.html"); });  app.listen(PORT, () => console.log(`Server listening on port: ${PORT}`)); |

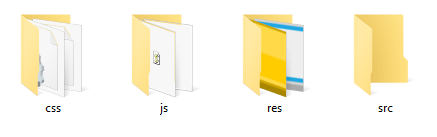
Creamos el directorio **public**, es donde se incluirán los recursos estáticos



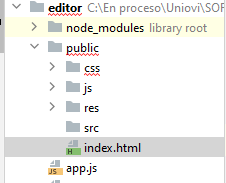
Vamos a crear dentro de **public** otros 4 directorios: **css, js, res, src.**



Descomprimimos dentro de cada carpeta los recursos descargados del campus virtual.



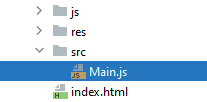
En la raíz de la carpeta public, vamos a crear el fichero **index.html**



Se trata de una vista muy simple que contiene un elemento **canvas** de html con la id **canvas**.

|  |
| --- |
| <!DOCTYPE html> <html> <head>  <meta charset="utf-8" />  <link rel="stylesheet" href="/css/uikit.css"/>  <meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate" />  <meta http-equiv="Pragma" content="no-cache" />  <meta http-equiv="Expires" content="0" />  <title>Editor</title>  <script src="/js/jquery.min.js"></script>  <script src="/js/uikit.js"></script>  <script src="/js/uikit-icons.js"></script> </head> <body>  <div class="uk-grid-small uk-child-width-expand@s uk-text-center" uk-grid>  <div>  <div class="uk-card uk-card-default uk-card-body">  *<!-- Herramientas -->* </div>  </div> </div> <div class="uk-child-width-expand@s uk-text-center" uk-grid>  <div class="uk-width-3-4">  <div class="uk-card uk-card-default uk-card-body">  *<!-- Panel de canvas -->* <canvas id="canvas" width="900" height="500" style="border:1px solid #d3d3d3;"></canvas>  </div>  </div>  <div class="uk-width-1-4">  <div class="uk-card uk-card-default uk-card-body">  *<!-- Propiedades -->* </div>  </div>  </div>   </body> </html> |

Vamos a crear el primero fichero de código, dentro de la carpeta **src**, se llamará **Main.js**

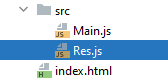


Se trata de un fichero escrito en funcional, que va a hacer de punto de unión entre todas las clases del proyecto, va a instanciar un **Editor** (clase que aun no hemos implementado)

|  |
| --- |
| *// Canvas y contexto del Canvas* var canvas = document.getElementById("canvas"); var contexto = canvas.getContext("2d");  var editor;  function *iniciar*() {  editor = new Editor();  editor.dibujar(); } |

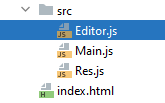
Vamos a crear un nuevo fichero **Res.js**, su misión es cargar los recursos gráficos que se usen en el editor. Los va a ir almacenando en una variable **cacheImagenes**, a partir de ahora utilizaremos esa variable cuando queramos obtener una imagen. Cuando acaba de cargar todas las foros llamamos a **iniciar()** (que se encuentra en el Main)

Es mejor cargar todos los recursos gráficos al inicio del programa que hacerlo bajo demanda, podrían tardar tiempo en cargar.



|  |
| --- |
| *// Lista de recursos a precargar* var imagenes = {  nodo : "res/nodo.png",  nodoActivo : "res/nodoActivo.png", };  var cacheImagenes = new Object(); var rutasImagenes = Object.values(imagenes); var nombresImagenes = Object.keys(imagenes); *cargarImagenes*(0);  function *cargarImagenes*(indice){  var imagenCargar = new Image();  imagenCargar.src = rutasImagenes[indice];  cacheImagenes[nombresImagenes[indice]] = imagenCargar;  imagenCargar.onload = function(){  if ( indice < rutasImagenes.length-1 ){  indice++;  *cargarImagenes*(indice);  } else {  *iniciar*();  }  } } |

Solo nos falta crear el Editor, añadimos un nuevo fichero **Editor.js**

****

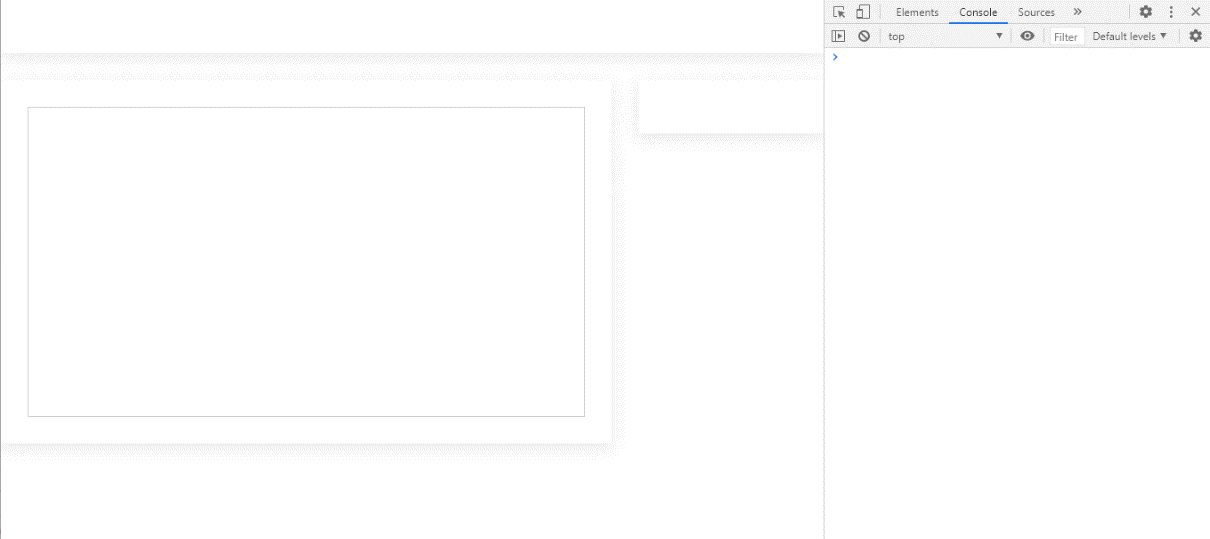
Este fichero contiene la clase **Editor**, tiene un **constructor()** y un método **dibujar().**

|  |
| --- |
| class Editor {   constructor() {   }    dibujar (){   }   } |

El último lugar hay que añadir todos los ficheros js al final del **index.html**. Hay que añadirlos en el orden correcto ya que algunos dependen de otros, por ejemplo: Main necesita que la clase Editor este cargada.

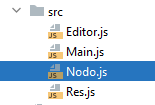
|  |
| --- |
| *<!-- Propiedades -->* </div>  </div>  </div>  <script src="src/Editor.js"></script> <script src="src/Main.js"></script> <script src="src/Res.js"></script>  </body> </html> |

Sí ejecutamos la aplicación deberíamos poder entrar en <http://localhost:3000/> y ver el canvas. Es importante comprobar que no hay errores en Consola.



# Creación de nodos

Vamos a crear un nuevo fichero **Nodo.js**



Está clase tiene un **constructor()** y un método **dibujar()**. Dibujar se encarga de la parte más importante, que es acceder al **contexto** del canvas y dibujar, primero la imagen que representa al nodo y después su texto. (Los dibujos se van colocando unos encima de otros).

|  |
| --- |
| class Nodo {   constructor(x, y) {  this.imagen = cacheImagenes["nodo"];   this.x = x;  this.y = y;  this.ancho = this.imagen.width;  this.alto = this.imagen.height;  this.texto = "nombre nodo";  }    dibujar (){  contexto.drawImage(this.imagen,  this.x - this.imagen.width/2 ,  this.y - this.imagen.height/2 );    contexto.textAlign = 'center';  contexto.font = "20px Arial";  contexto.fillText(this.texto, this.x, this.y);  }  } |

Siempre que agregamos un nuevo fichero js hay que añadirlo al **index.html**

|  |
| --- |
| </div>  <script src="src/Nodo.js"></script> <script src="src/Editor.js"></script> <script src="src/Main.js"></script> <script src="src/Res.js"></script>  </body> |

En el mismo **index.html** vamos a agregar un botón HTML que permita crear Nodos. Al pulsarlo llamará a la función **activarCrearNodo()**; Está función la vamos a implementar en el main.

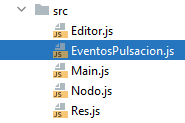
|  |
| --- |
| <title>Editor</title>  <script src="/js/jquery.min.js"></script>  <script src="/js/uikit.js"></script>  <script src="/js/uikit-icons.js"></script> </head> <body>  <div class="uk-grid-small uk-child-width-expand@s uk-text-center" uk-grid>  <div>  <div class="uk-card uk-card-default uk-card-body">  *<!-- Herramientas -->* <button id="botonCrearNodo" class="uk-button uk-button-default uk-margin-small-bottom"  onclick="activarCrearNodo()">Crear Nodo</button>  </div>  </div> </div> |

Abrimos el fichero **Main.js.** Vamos a crear una variable global llamada **acción** que sirva para que el programa sepa que acción estamos realizando, en este caso crear un nodo será **ACCION\_CREAR\_NODO.**

Aprovechamos para cambiar el estilo del botón HTML, esto es secundario.

|  |
| --- |
| var editor; var ACCION\_NO = 0; var ACCION\_CREAR\_NODO = 1; var accion = ACCION\_NO;  function *iniciar*() {  editor = new Editor();  editor.dibujar(); }  function *activarCrearNodo*(){  accion = ACCION\_CREAR\_NODO;  $('#botonCrearNodo').removeClass();  $('#botonCrearNodo').addClass('uk-button uk-button-secondary uk-margin-small-bottom'); } |

Ahora nos hace falta que el programa pueda detectar pulsaciones de ratón, creamos el fichero **EventosPulsacion** va a ser un fichero funcional donde se incluya todo lo relativo a pulsaciones.



El contenido será el siguiente. Tiene que registrar eventos de varios tipos **mousedown**, **mousemove** y **mouseup**.

**procesarCoordenadas,** este método nos va a devolver las coordenadas reales en el canvas. LAS COORDENADAS EN EL CANVAS NO SON LAS MISMAS QUE EN LA PAGINA HAY MARGENES. Además vamos a dejar implementado un sistema que facilite implementar un sistema de puede **reescalar** en el futuro. De forma que el tamaño no sea siempre el “original de 900px”.

|  |
| --- |
| $('#canvas').mousedown(function(event){  *// Calcular la posición del click en el canvas (No en la página completa)* click = *procesarCoordenadas*(event);  editor.comprobarMouseDown(click.x, click.y); })  $('#canvas').mousemove(function(event){  click = *procesarCoordenadas*(event);  editor.comprobarMouseMove(click.x, click.y); })  $('#canvas').mouseup(function(event){  click = *procesarCoordenadas*(event);  editor.comprobarMouseUp(click.x, click.y); })  function *procesarCoordenadas*(event){  var rect = canvas.getBoundingClientRect();  var escalado = canvas.clientWidth/900; *// es igual en los 2 ejes,* x = (event.clientX - rect.left)/escalado;  y = (event.clientY - rect.top)/escalado;   return { x: x, y: y} } |

Hay que registrar el nuevo fichero en el **index.html**

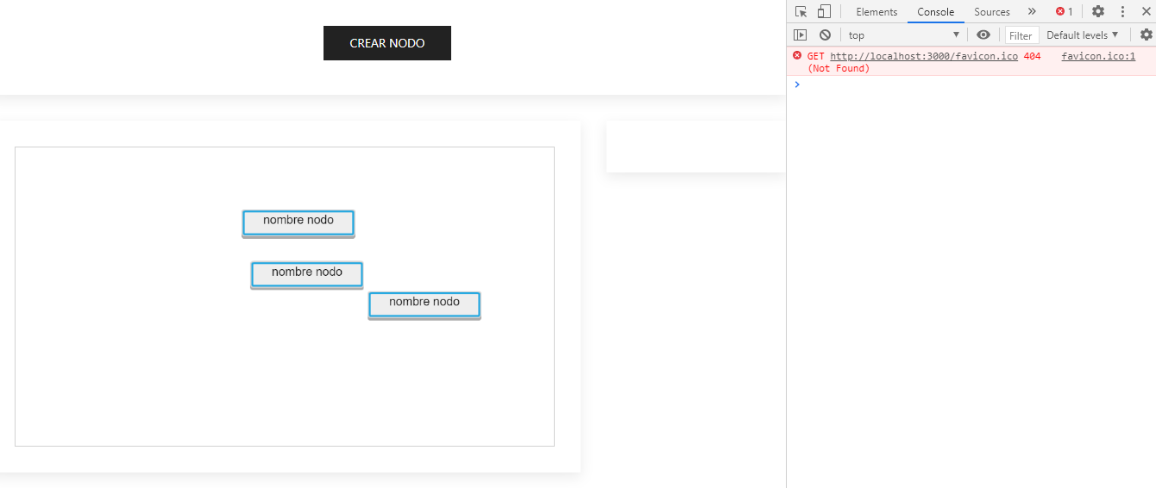
|  |
| --- |
| </div>  <script src="src/Nodo.js"></script> <script src="src/Editor.js"></script> <script src="src/Main.js"></script> <script src="src/Res.js"></script> <script src="src/EventosPulsacion.js"></script>  </body> |

Ahora nos faltan por realizar varios cambios en el **Editor.js**, tiene que:

* Gestionar los nodos, en una lista o similar
* Dibujar los nodos
* Implementar el método **comprobarMouseDown,** ver si **acción == ACCION\_CREAR\_NODO**, en ese caso crear un nuevo nodo.
* Implementar el **comprobarMouseMove** y **comprobarMouseUp**, aunque de momento no van a tener funcionalidad.

|  |
| --- |
| class Editor {   constructor() {  this.nodos = [];  }   dibujar (){  for(var i=0; i < this.nodos.length; i++) {  this.nodos[i].dibujar();  }   }   comprobarMouseDown(x, y){  *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO){  var nuevoNodo = new Nodo(x ,y);  this.nodos.push(nuevoNodo);  }  this.dibujar();  }    comprobarMouseMove(x, y){   }   comprobarMouseUp(x, y){   } } |

Ejecutamos el programa para comprobar su funcionalidad de crear nodo.



# Marcar nodo seleccionado

Actualmente podemos crear nodos, encima de otros nodos. Es decir, no detecta que el click de **comprobarMouseDown** cae dentro de un nodo existente.

Lo primero es editar **Nodo.js** para incluir un método **contienePunto(x,y),** básicamente comprueba si el punto cae el área del nodo.

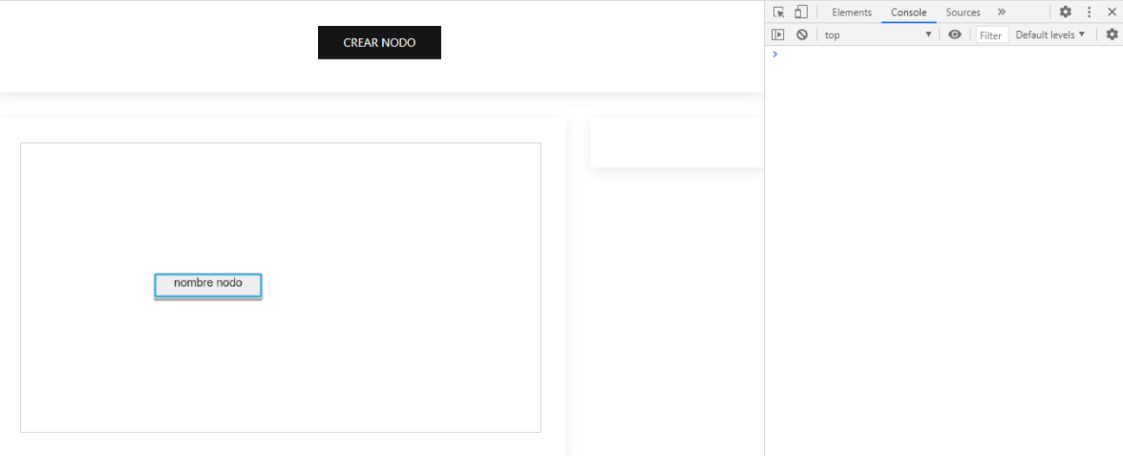
|  |
| --- |
| class Nodo {   constructor(x, y) {  this.imagen = cacheImagenes["nodo"];   this.x = x;  this.y = y;  this.ancho = this.imagen.width;  this.alto = this.imagen.height;  this.texto = "nombre nodo";  }  contienePunto(pX, pY){  if ( pY >= this.y - this.alto/2 &&  pY <= this.y + this.alto/2 &&  pX <= this.x + this.ancho/2 &&  pX >= this.x - this.ancho/2){  return true;  }  return false;  } |

Accedemos a **Editor.js** e implementamos el método **auxPulsacionEnNodo(x,y)**, sirve para calcular si una coordenada está contenida en un nodo.

Modificamos **comprobarMouseDown(),** para que calcule **nodoEnQueCaeClick**, solo crearemos un nuevo nodo si el click no cae en ningún nodo.

|  |
| --- |
| auxPulsacionEnNodo(x,y){  *// no cae en ningun nodo* for(var i=0; i < this.nodos.length; i++) {  if ( this.nodos[i].contienePunto(x,y)){  return this.nodos[i];  }  }  return null; }  comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);    *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX,y + scrollY);  this.nodos.push(nuevoNodo);  }  this.dibujar(); } |

Sí ejecutamos el programa, no nos dejará crear nodos sobre otros.



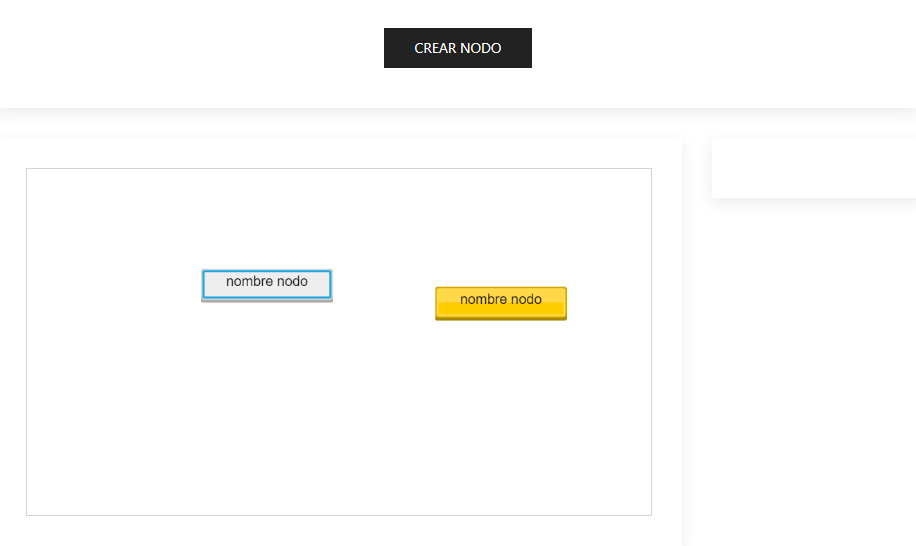
Aprovechamos que tenemos la variable **nodoEnQueCaeElClick** para hacer que si un click cae sobre un nodo este se seleccione. En **Nodo.js** agregamos un nuevo método **dibujarSeleccionado(boolean)** que le cambie la imagen dependiendo de si está seleccionado o no.

|  |
| --- |
| class Nodo {   constructor(x, y) {  this.imagen = cacheImagenes["nodo"];   this.x = x;  this.y = y;  this.ancho = this.imagen.width;  this.alto = this.imagen.height;  this.texto = "nombre nodo";  }    dibujarSeleccionado(seleccionado){  if ( seleccionado ){  this.imagen = cacheImagenes["nodoActivo"];  } else {  this.imagen = cacheImagenes["nodo"];  }  } |

Accedemos a **Editor.js,** implementamos **dibujarNodoSeleccionado(nodo),** recorre los nodos y marca como seleccionado el que recibe como parámetros. El método **comprobarMouseDown()** debe llamar al nuevo método **dibujarNodoSeleccionado(nodo).**

|  |
| --- |
| dibujarNodoSeleccionado(nodo){  *// Nodo* for(var i=0; i < this.nodos.length; i++){  this.nodos[i].dibujarSeleccionado(false);  }  if ( nodo != null ) {  nodo.dibujarSeleccionado(true);  }  this.dibujar(); }  comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX,y + scrollY);  this.nodos.push(nuevoNodo);  }  this.dibujarNodoSeleccionado(nodoEnQueCaeElClick);  ~~this.dibujar();~~ } |

Ejecutamos la aplicación, creamos varios nodos y hacemos click sobre ellos.



# Ver propiedades de nodo (editar)

Vamos a hacer que el texto del nodo seleccionado aparezca en la parte derecha de la pantalla. Lo primero será modificar el **index.html**.

Va a tener un campo de texto **nombreNodo** y un botón guardar que al pulsar ejecuta la función **clickBotonEditarPropiedadesNodo();** (aun sin implementar).

|  |
| --- |
| <div class="uk-child-width-expand@s uk-text-center" uk-grid>  <div class="uk-width-3-4">  <div class="uk-card uk-card-default uk-card-body">  *<!-- Panel de canvas -->* <canvas id="canvas" width="900" height="500" style="border:1px solid #d3d3d3;"></canvas>  </div>  </div>  <div class="uk-width-1-4">  <div class="uk-card uk-card-default uk-card-body">  *<!-- Propiedades -->* <h2>Propiedades</h2>  <div class="uk-margin">  <input class="uk-input" type="text" id="nombreNodo">  </div>  <button class="uk-button uk-button-primary uk-width-1-1 uk-margin-small-bottom"  onClick="clickBotonEditarPropiedadesNodo()">Guardar</button>  </div>  </div>  </div> |

Accedemos a **Main.js** y agregamos la función **clickBotonEditarPropiedadesNodo.** Debe comprobar que hay un nodoActivo y después cambiarle el texto por el valor del input **nombreNodo** (usamos jQuery para obtener el valor).

|  |
| --- |
| var editor; var ACCION\_NO = 0; var ACCION\_CREAR\_NODO = 1; var accion = ACCION\_NO;  var nodoActivo = null;  function *iniciar*() {  editor = new Editor();  editor.dibujar(); }  function *globalActualizarNodoActivo*(nodo){  if ( nodo != null) {  console.log("Actualiza Nodo activo "+nodo.texto);  nodoActivo = nodo;  $('#nombreNodo').val(nodo.texto);  } else {  nodoActivo = null;  $('#nombreNodo').val("");  } }  function *clickBotonEditarPropiedadesNodo*(){  if (nodoActivo != null) {  nodoActivo.texto = $('#nombreNodo').val();  editor.dibujar();  } }  function *activarCrearNodo*(){  accion = ACCION\_CREAR\_NODO;  $('#botonCrearNodo').removeClass();  $('#botonCrearNodo').addClass('uk-button uk-button-secondary uk-margin-small-bottom'); } |

Dentro de **Main.js** vamos a crear una función adicional llamada, **globalNingunaAccionSeleccionada()**, básicamente quita la acción actual y también actualiza la clase css del botón

|  |
| --- |
| function *iniciar*() {  editor = new Editor();  editor.dibujar(); }  function *globalNingunaAccionSeleccionada*(){  accion = ACCION\_NO;  $('#botonCrearNodo').removeClass();  $('#botonCrearNodo').addClass('uk-button uk-button-default uk-margin-small-bottom'); } |

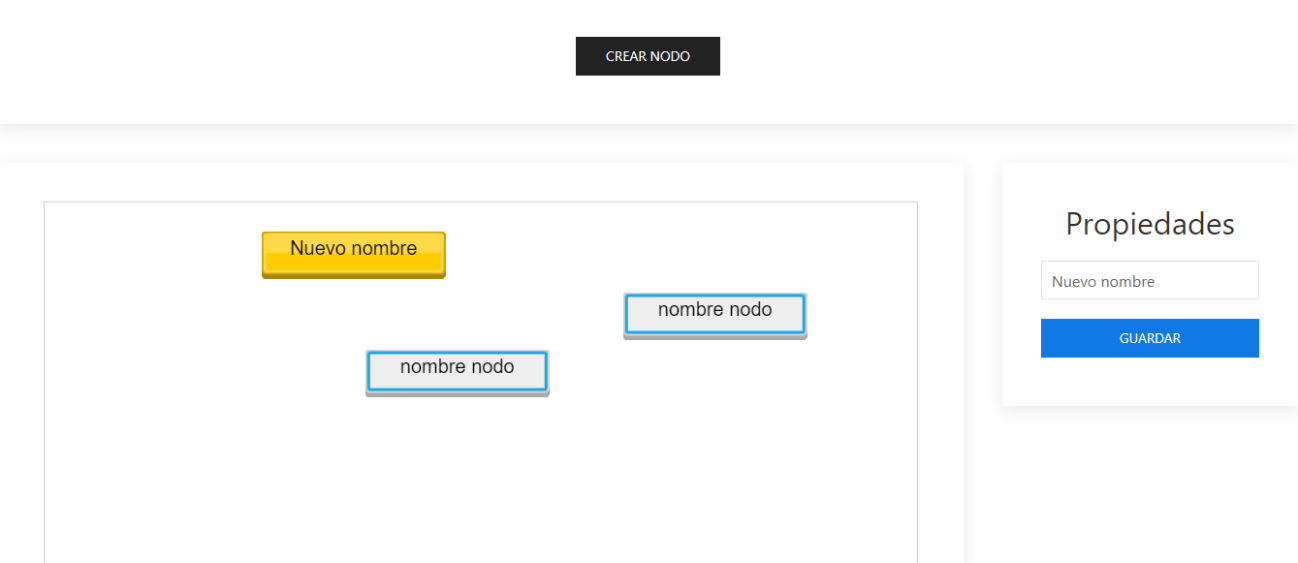
Ahora solo falta que el editor ejecute la función **globalActualizarNodoActivo(nodo),** cada vez que alguien hace click en un nodo o click en una superficie en “blanco”.

Accedemos a **Editor.js** ampliamos el método **comprobarMouseDown(x,y).**

* Si la acción es crear nodo, desactivamos las acciones **globalNingunaAccionSeleccionada();** y marcamos el nuevo nodo como activo **globalActualizarNodoActivo(nodoEnQueCaeElClick);**
* Si no hay acción y el click cae en un nodo, actualizamos el nodo activo **globalActualizarNodoActivo(nodoEnQueCaeElClick)**
* Si no hay acción y el click NO cae en un nodo, ponemos a null el nodo activo **globalActualizarNodoActivo(null);**

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x, y);  this.nodos.push(nuevoNodo);  nodoEnQueCaeElClick = nuevoNodo;  *globalNingunaAccionSeleccionada*();  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);  *// Click normal en un nodo* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick != null) {   *globalActualizarNodoActivo*(nodoEnQueCaeElClick);    *// Seleccionar una zona "libre"* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick == null ){  *globalActualizarNodoActivo*(null);  }   this.dibujarNodoSeleccionado(nodoEnQueCaeElClick);  } |

Probamos el editor, ahora debería ser posible cambiar los nombres de los nodos.



# Scroll

Ahora mismo el diagrama tiene una superficie limitada 900x500. Vamos a extenderla de forma que se pueda hacer scroll.

El canvas siempre va a medir 900x500, el scroll consiste básicamente que los nodos tengan una posición **X** e **Y** real (por ejemplo X:10000 , Y:10000 ) y una posición de pintado en el canvas (dentro de X:900 o Y:500 sí queremos que se vea)

Por lo tanto, el scroll es un valor que modifica las coordenadas **X** e **Y** reales de dibujo de los elementos.

Accedemos a **Main.js** y creamos las variables globales relativas al scroll. Vamos a tener también una nueva acción **ACCION\_SCROLL**, movernos por el scroll.

* **scrollFijado** va a contener un objeto { x , y } con las coordenadas del punto sobre el que comienza el scroll.

|  |
| --- |
| var contexto = canvas.getContext("2d");  var editor; var ACCION\_NO = 0; var ACCION\_CREAR\_NODO = 1; var ACCION\_SCROLL = 2; var accion = ACCION\_NO;  var nodoActivo = null;  var scrollX = 0; var scrollY = 0; var scrollFijado = null;  function *iniciar*() { |

El primer punto en el que vamos a comenzar a utilizar el scroll, **scrollX** y **scrollY** es el nodo.

Accedemos a **Nodo.js**, el scroll afecta al **contienePunto(pX,pY)** y **dibujar().**

* **contienePunto(pX,pY)**, la coordenada del punto viene en el sistema de coordenadas del canvas (x:0-900, y:0-500) en cambio el nodo puede tener scroll, hay que aplicar el scroll en este caso.
* **dibujar()**, el nodo puede tener scroll.

|  |
| --- |
| contienePunto(pX, pY){  if ( pY >= this.y - this.alto/2 - scrollY &&  pY <= this.y + this.alto/2 - scrollY &&  pX <= this.x + this.ancho/2 - scrollX &&  pX >= this.x - this.ancho/2 - scrollX){  return true;  }  return false; }  dibujar (){  contexto.drawImage(this.imagen,  this.x - this.imagen.width/2 - scrollX,  this.y - this.imagen.height/2 - scrollY);   contexto.textAlign = 'center';  contexto.font = "20px Arial";  contexto.fillText(this.texto, this.x - scrollX, this.y - scrollY); } |

Ahora vamos a modificar el **Editor.js**, cuando el usuario hace click **comprobarMouseDown(x,y)** en una zona “libre”, lo que quiere es comenzar a fijar el scroll.

Habrá que señalar que la acción pasa a ser **ACCION\_SCROLL** y guardar las coordenadas de donde ha hecho click en el **scrollFijado**, estas coordenadas se usan para calcular el scroll real.

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x ,y);  this.nodos.push(nuevoNodo);  nodoEnQueCaeElClick = nuevoNodo;  accion = ACCION\_NO;  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Click normal en un nodo* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick != null) {   *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Seleccionar una zona "libre"* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick == null ){  accion = ACCION\_SCROLL;  scrollFijado = {x: x, y: y};  *globalActualizarNodoActivo*(null);  }   this.dibujarNodoSeleccionado(nodoEnQueCaeElClick);  } |

La modificación del scroll comienza cuando comenzamos a mover el ratón, será implementando el método **comprobarMouseMove(x,y).**

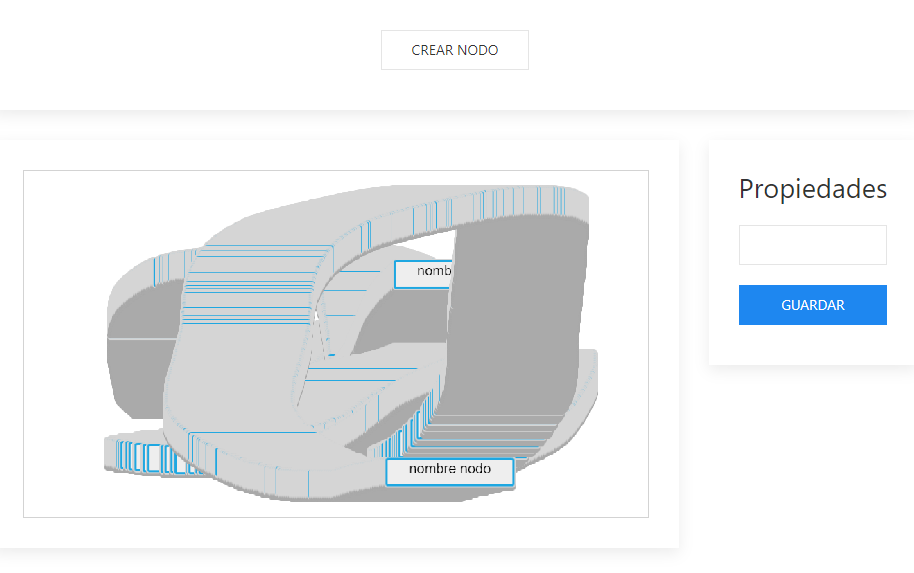
|  |
| --- |
| comprobarMouseMove(x, y){  *// Hay scroll* if ( accion == ACCION\_SCROLL && scrollFijado != null ){  scrollX += scrollFijado.x - x;  scrollY += scrollFijado.y - y;  scrollFijado = { x: x, y: y}; *// Dejar el Scroll en el nuevo punto del ratón* this.dibujar();  } } |

Solo falta hacer que el scroll termine si en algún momento levantamos el dedo del ratón, método **comprobarMouseUp(x,y)**. No conviene quitar siempre la acción, se quita solo la de Scroll, si no en el futuro podrá darnos problemas en acciones que requieren más de un click (por ejemplo, Crear una relación), al hacer UP del primer click se perdería la acción.

|  |
| --- |
| comprobarMouseUp(x, y){  scrollFijado = null;  if ( accion == ACCION\_SCROLL){  *globalNingunaAccionSeleccionada*();  } } |

Ejecutamos el editor para probar el funcionamiento del scroll hasta este punto, mantenemos el click sobre una zona libre y arrastramos el ratón. Hay un problema en el refresco del dibujado del editor.

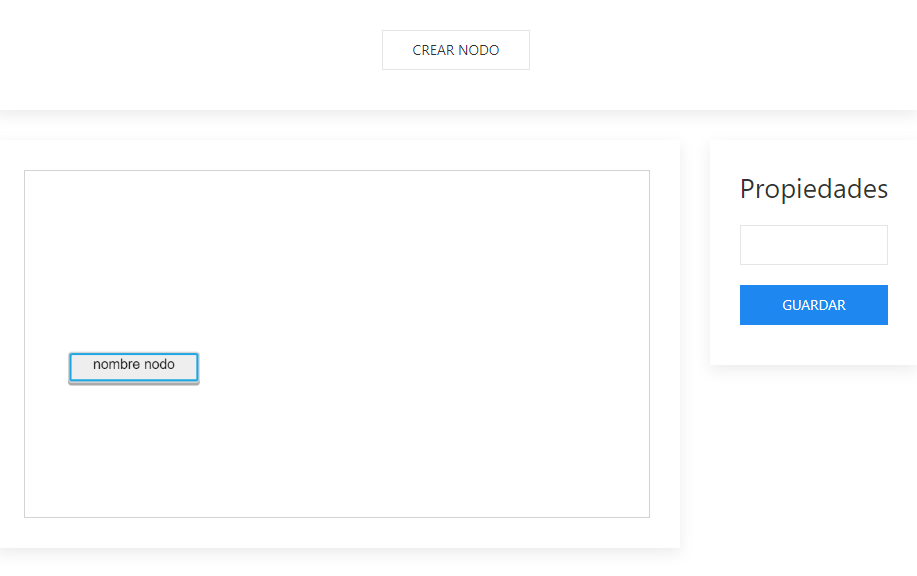
Se debe a que cuando dibujamos algo en un canvas no se “borra”, lo que había antes, solo se pinta encima, hay que buscar una forma de “borrar” el canvas antes de pintar de nuevo. Además hay otros problemas que iremos viendo poco a poco.



Accedemos a **Editor.js** nos colocamos en su método **dibujar()**, antes de comenzar a dibujar los nodos, vamos a incluir un rectángulo blanco que ocupe todo el canvas (900,500).

|  |
| --- |
| dibujar (){  *// Capa de fondo* var colorPrevio = contexto.fillStyle;  contexto.fillStyle="#fff";  contexto.fillRect(0,0,900,500);  contexto.fillStyle = colorPrevio;    for(var i=0; i < this.nodos.length; i++) {  this.nodos[i].dibujar();  } } |

Ejecutamos el editor para probar que ese problema ha desaparecido. Parece que el scroll ahora es correcto, pero tenemos otro problema, si hacemos scroll y luego creamos un nodo, este no se crea en la posición correcta.

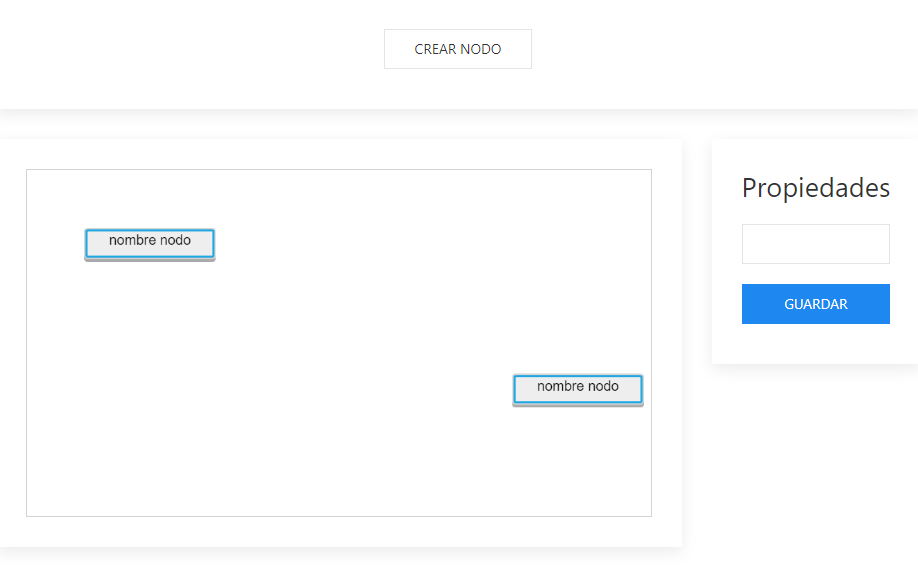


Se nos ha olvidado tener en cuenta el scroll en el momento de crear un nodo, siempre nos va a crear nodos en posiciones reales del canvas (x: 0-900, y:0-500). Eso no es correcto, si hemos desplazado el scroll antes quizá queramos crear un nodo en otra posición, por ejemplo la x:100000 , y:1000000, el sistema actual no lo permite.

Accedemos a **Editor.js** y ampliamos el **comprobarMouse(x,y),** justo en el momento de crear un nodo.

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX ,y + scrollY);  this.nodos.push(nuevoNodo); |

Tras esta modificación la acción de crear un nodo después de realizar un scroll debería funcionar correctamente.



# Mover un nodo

Nos gustaría poder mover un nodo si mantenemos el click pulsado sobre él. Esto se corresponde con una nueva acción.

Accedemos a **Main.js** y declaramos un nuevo tipo de acción.

|  |
| --- |
| var editor; var ACCION\_NO = 0; var ACCION\_CREAR\_NODO = 1; var ACCION\_SCROLL = 2; var ACCION\_MOVER\_NODO = 3; var accion = ACCION\_NO; |

Abrimos el **Editor.js** , en el método **comprobarMouseDown(x,y)** si no hay ninguna acción activa y el click cae sobre un nodo, el usuario va a poder moverlo.

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX,y + scrollY);  this.nodos.push(nuevoNodo);  nodoEnQueCaeElClick = nuevoNodo;  *globalNingunaAccionSeleccionada*();  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Click normal en un nodo* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick != null) {  accion = ACCION\_MOVER\_NODO;  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Seleccionar una zona "libre"* } |

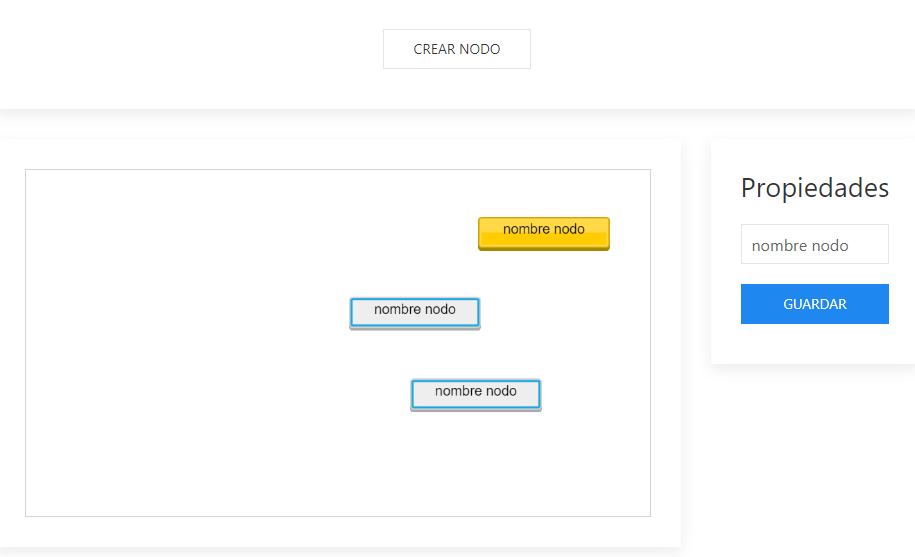
Ahora ampliamos el método **comprobarMouseMove(x,y),** los movimientos de ratón cuando la acción es ACCION\_MOVER\_NODO, deben alterar la posición X e Y del nodoActivo.

|  |
| --- |
| comprobarMouseMove(x, y){  *// Hay nodo seleccionado* if ( accion == ACCION\_MOVER\_NODO && nodoActivo != null ){  nodoActivo.x = x + scrollX;  nodoActivo.y = y + scrollY;  this.dibujar();  }  *// Hay scroll* if ( accion == ACCION\_SCROLL && scrollFijado != null ){  scrollX += scrollFijado.x - x;  scrollY += scrollFijado.y - y;  scrollFijado = { x: x, y: y}; *// Dejar el Scroll en el punto anterior para que no siga sumando* this.dibujar();  } } |

Solo falta tener en cuenta que cuando se levanta la pulsación de ratón **comprobarMouseUp(x,y)** se deja de estar en la acción **ACCION\_MOVER\_NODO**, por lo tanto tenemos que añadir esa posibilidad.

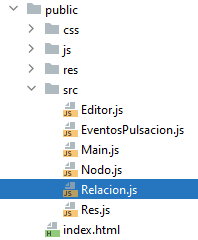
|  |
| --- |
| comprobarMouseUp(x, y){  scrollFijado = null;  if ( accion == ACCION\_MOVER\_NODO || accion == ACCION\_SCROLL){  *globalNingunaAccionSeleccionada*();  } } |

Ejecutamos la aplicación, probamos a crear varios nodos y a moverlos, también lo probamos en zona de scroll.



# Relaciones

El primer paso va a ser crear un nuevo fichero **Relacion.js** (representará el dibujo de la relación en el canvas).



La relación es un elemento un tanto especial, ya que depende del nodo. No sería nada fácil tratarla como un elemento convencional (con su forma, una línea y unas coordenadas X e Y).

Lo más efectivo es guardar una referencia a los 2 nodos que relaciona, en **dibujar()** lo que hace es crear una línea de nodo a nodo.

|  |
| --- |
| class Relacion {   constructor(nodoA, nodoB) {  this.nodoA = nodoA;  this.nodoB = nodoB;  this.color = "#000000";  }   dibujar() {  contexto.beginPath();  contexto.moveTo( this.nodoA.x - scrollX, this.nodoA.y - scrollY);  contexto.lineTo(this.nodoB.x - scrollX, this.nodoB.y - scrollY);  contexto.strokeStyle = this.color;  contexto.stroke();   }  } |

Siempre que se crea un nuevo fichero JS hay que añadirlo al **index.html.**

|  |
| --- |
| </div>  <script src="src/Relacion.js"></script> <script src="src/Nodo.js"></script> <script src="src/Editor.js"></script> <script src="src/Main.js"></script> <script src="src/Res.js"></script> <script src="src/EventosPulsacion.js"></script>  </body> </html> |

**Nodo**, de cara a facilitar la navegación la red, sería muy cómodo que el nodo también conociese sus nodos relacionados. La clase **relación** es más bien para “pintar” en el canvas, no para guardar información de la red de nodos.

Si no hubiese interfaz gráfica seguramente no hubiésemos creado la clase relación, bastaría con que cada todo tuviese una lista de nodos relacionados.

Accedemos a **Nodo.js**, agregamos un array **nodosRelacionados** y un nuevo método **agregarNodoRelacionado(nodo),** para agregar una relación entre nodos:

* No admitimos relaciones circulares con el propio nodo
* No admitimos que ya exista la relación (ni en este sentido ni en el otro), realmente nuestras relaciones son bidireccionales.

|  |
| --- |
| class Nodo {   constructor(x, y) {  this.imagen = cacheImagenes["nodo"];   this.x = x;  this.y = y;  this.ancho = this.imagen.width;  this.alto = this.imagen.height;  this.texto = "nombre nodo";   this.nodosRelacionados = [];  }   agregarNodoRelacionado(nodo){  *// No admitir relaciones consigo mismo  // Antes de agregar mirar que no tenga ya la relación* if ( nodo != this && this.nodosRelacionados.filter(n => n == nodo).length == 0) {  this.nodosRelacionados.push(nodo);  return true;  }  return false;  }   dibujarSeleccionado(seleccionado){ |

Ahora nos falta que el editor pueda gestionar y dibujar las relaciones.

Accedemos a **Editor.js** y creamos un array de **relaciones**, también las añadimos al método **dibujar()**. Las relaciones se dibujan antes de los nodos, para que la caja del nodo tape la línea que se crea en el centro del nodo.

|  |
| --- |
| class Editor {   constructor() {  this.nodos = [];  this.relaciones = [];  }   dibujar (){  *// Capa de fondo* var colorPrevio = contexto.fillStyle;  contexto.fillStyle="#fff";  contexto.fillRect(0,0,900,500);  contexto.fillStyle = colorPrevio;   for(var i=0; i < this.relaciones.length; i++) {  this.relaciones[i].dibujar();  }   for(var i=0; i < this.nodos.length; i++) {  this.nodos[i].dibujar();  }  } |

**¿Cómo se crea una relación?** vamos a incluir un botón de menú que nos permita crear relaciones entre nodo, accedemos a **index.html** y añadimos el botón HTML.

|  |
| --- |
| </head> <body>  <div class="uk-grid-small uk-child-width-expand@s uk-text-center" uk-grid>  <div>  <div class="uk-card uk-card-default uk-card-body">  *<!-- Herramientas -->* <button id="botonCrearNodo" class="uk-button uk-button-default uk-margin-small-bottom"  onclick="*activarCrearNodo*()">Crear Nodo</button>  <button id="botonCrearRelacion" class="uk-button uk-button-default uk-margin-small-bottom"  onclick="activarCrearRelacion()">Crear Relación</button>  </div>  </div> </div> |

Accedemos a **Main.js** y creamos la función **activarCrearRelacion(),** para poder crear una relación nos va a hacer falta agregar nuevas variables

* Crear la **ACCION\_CREAR\_RELACION**
* Dos variables que nos permitan saber los nodos sobre los que se hace click, **nodoRelacion1** y **nodoRelacion2**

Además creamos la función **activarCrearRelacion().**

|  |
| --- |
| var editor; var ACCION\_NO = 0; var ACCION\_CREAR\_NODO = 1; var ACCION\_SCROLL = 2; var ACCION\_MOVER\_NODO = 3; var ACCION\_CREAR\_RELACION = 4; var accion = ACCION\_NO;  var nodoRelacion1 = null; var nodoRelacion2 = null; var nodoActivo = null;  var scrollX = 0; var scrollY = 0; var scrollFijado = null;  function *iniciar*() {  editor = new Editor();  editor.dibujar(); }  function *activarCrearRelacion*(){  accion = ACCION\_CREAR\_RELACION;  editor.dibujarNodoSeleccionado(null);  $('#botonCrearRelacion').removeClass();  $('#botonCrearRelacion').addClass('uk-button uk-button-secondary uk-margin-small-bottom'); } |

Como ahora tenemos dos botones HTML de herramientas (crear nodo y crear relación) debemos ampliar la función, **globalNingunaAccionSeleccionada()** para que desactive ambos botones.

|  |
| --- |
| function *globalNingunaAccionSeleccionada*(){  accion = ACCION\_NO;  $('#botonCrearNodo').removeClass();  $('#botonCrearNodo').addClass('uk-button uk-button-default uk-margin-small-bottom');  $('#botonCrearRelacion').removeClass();  $('#botonCrearRelacion').addClass('uk-button uk-button-default uk-margin-small-bottom'); } |

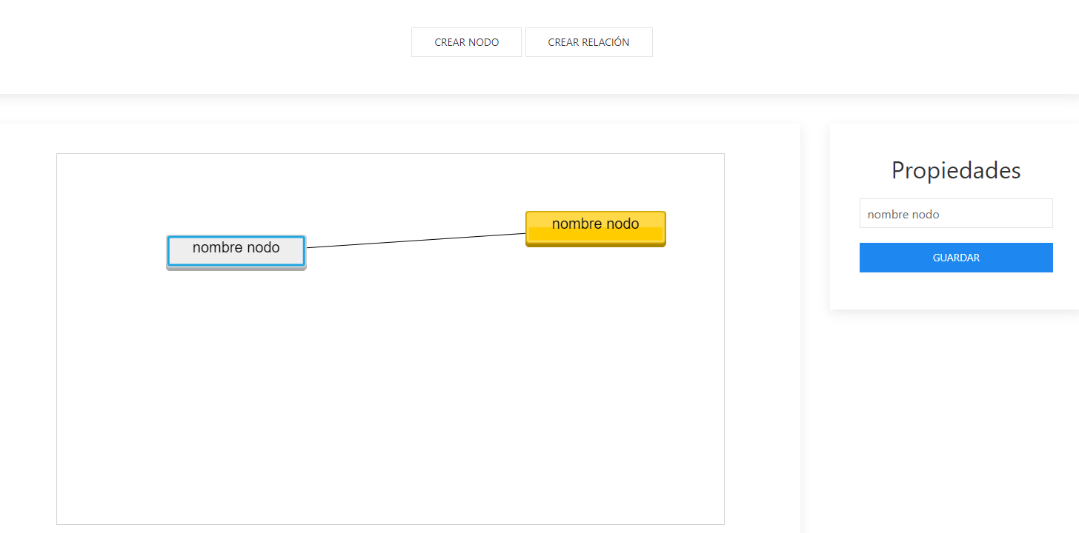
Vamos al **Editor.js**, comenzando por **comprobarMouseDown(x,y)** hay que tener en cuenta que ahora hay un nuevo tipo de acción **ACCION\_CREAR\_RELACION**, el click debe caer sobre un nodo.

El funcionamiento de esta nueva rama es la siguiente

* Si **nodoRelacion1** vale null le asignamos el nodo que acaban de seleccionar.
* Si **nodoRelacion1** No es null, analizamos **nodoRelacion2** y le asignamos el nodo que acaban de seleccionar.
  + En este caso ya podemos crear la relación y reiniciar toda la acción (borrar la acción seleccionada y los nodos)

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX,y + scrollY);  this.nodos.push(nuevoNodo);  nodoEnQueCaeElClick = nuevoNodo;  *globalNingunaAccionSeleccionada*();  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Click normal en un nodo* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick != null) {  accion = ACCION\_MOVER\_NODO;  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Seleccionar una zona "libre"* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick == null ){  accion = ACCION\_SCROLL;  scrollFijado = {x: x, y: y};  *globalActualizarNodoActivo*(null);    *// Seleccionar Nodo para crear relación* } else if ( accion == ACCION\_CREAR\_RELACION && nodoEnQueCaeElClick != null) {  if (nodoRelacion1 == null){  nodoRelacion1 = nodoEnQueCaeElClick;  } else if (nodoRelacion2 == null){  nodoRelacion2 = nodoEnQueCaeElClick;  *// Crear la relación* if (nodoRelacion1 != null && nodoRelacion2 != null){  var agregado = nodoRelacion1.agregarNodoRelacionado(nodoRelacion2);  if ( agregado ){  this.relaciones.push(new Relacion(nodoRelacion1,nodoRelacion2))  }  }  *// Reiniciarlo todo  globalNingunaAccionSeleccionada*();  nodoRelacion1 = null;  nodoRelacion2 = null;  }  }   this.dibujarNodoSeleccionado(nodoEnQueCaeElClick);  } |

Probamos la aplicación, debería dejarnos crear relaciones entre nodos. La forma que hemos utilizado para implementar las relaciones hace que estas funcionen con el scroll.



# Marcar la relación seleccionada

Lo primero que vamos a hacer es acceder a **Main.js** y crear una variable global **relacionActiva**, apuntará a la relación en la que ha pulsado el usuario.

|  |
| --- |
| var nodoRelacion1 = null; var nodoRelacion2 = null; var relacionActiva = null; var nodoActivo = null; var relacionActiva = null;  var scrollX = 0; |

Tenemos una función **globalActualizarNodo(nodo)**, que se encarga de colocar las propiedades en el HTML, vamos a ampliar la función para que también incluya las propiedades de la relación pulsada, en este caso colocamos un texto Nodo A -> Nodo B.

Solo comprobamos la relación si no hay nodo, de lo contrario si nos envían un nodo con valor y una relación null, la relación borraría la propiedad (los 2 usan el mismo campo de texto imput)

|  |
| --- |
| function *globalActualizarNodoActivo*(nodo, relacion){  if ( nodo != null) {  console.log("Actualiza Nodo activo "+nodo.texto);  nodoActivo = nodo;  $('#nombreNodo').val(nodo.texto);  } else {  console.log("Quitar Nodo activo ");  nodoActivo = null;  $('#nombreNodo').val("");  }   if (nodo == null ){  if ( relacion != null) {  relacionActiva = relacion;  $('#nombreNodo').val("r: "+relacion.nodoA.texto+" ->"+relacion.nodoB.texto);  } else {  console.log("Quitar nombre relacion ");  relacionActiva = null;  $('#nombreNodo').val("");  }  } } |

Desde el punto de vista del click, marcar la relación seleccionada no es tan sencillo como marcar un nodo. Difícilmente el click va a caer en la línea, aplicaremos una formula matemática para saber si el click está “cerca” de la línea-relación.

Accedemos a **Relación.js** e implementamos los métodos.

* **dibujarSeleccionado(boolean),** al igual que el nodo, la relación que está seleccionada se dibuja de otra forma para dar feedback al usuario, en este caso se cambia el color de la línea.
* **contienePunto(pX,pY)** al igual que en el nodo, queremos saber si el click cae sobre la relación, nos ayudaremos del calculo de la distancia entre el punto y la recta.
* **pDistance(x,y,x1,y1,x2,y2)** operación matemática calcula la distancia de un punto (x,y) sobre una recta determinada por dos puntos (x1,y1) (x2,y2).

|  |
| --- |
| class Relacion {   constructor(nodoA, nodoB) {  this.nodoA = nodoA;  this.nodoB = nodoB;  this.color = "#000000";  }   dibujar() {  contexto.beginPath();  contexto.moveTo( this.nodoA.x - scrollX, this.nodoA.y - scrollY);  contexto.lineTo(this.nodoB.x - scrollX, this.nodoB.y - scrollY);  contexto.strokeStyle = this.color;  contexto.stroke();   }   dibujarSeleccionado(seleccionado){  if ( seleccionado ){  this.color = "#FFA500";  } else {  this.color = "#000000";  }  }    contienePunto(pX, pY){  var distancia = this.pDistance(pX,pY,  this.nodoA.x - scrollX, this.nodoA.y - scrollY,  this.nodoB.x - scrollX, this.nodoB.y - scrollY);   if (distancia <= 20 ){  return true;  }  return false;  }   pDistance(x, y, x1, y1, x2, y2) {  var A = x - x1;  var B = y - y1;  var C = x2 - x1;  var D = y2 - y1;   var dot = A \* C + B \* D;  var len\_sq = C \* C + D \* D;  var param = -1;  if (len\_sq != 0) *//in case of 0 length line* param = dot / len\_sq;   var xx, yy;   if (param < 0) {  xx = x1;  yy = y1;  }  else if (param > 1) {  xx = x2;  yy = y2;  }  else {  xx = x1 + param \* C;  yy = y1 + param \* D;  }   var dx = x - xx;  var dy = y - yy;  return Math.sqrt(dx \* dx + dy \* dy);  }  } |

Accedemos al **Editor.js** para implementar el nuevo método **auxPulsacionEnRelacion(x,y);** funciona igual que el método auxiliar que analizaba si una pulsación caía en algún nodo.

|  |
| --- |
| auxPulsacionEnNodo(x,y){  *// no cae en ningun nodo* for(var i=0; i < this.nodos.length; i++) {  if ( this.nodos[i].contienePunto(x,y)){  return this.nodos[i];  }  }  return null; }  auxPulsacionEnRelacion (x,y){  *// ¿cae en alguna relacion?* for(var i=0; i < this.relaciones.length; i++) {  if(this.relaciones[i].contienePunto(x,y)){  return this.relaciones[i];  }  }  return null; } |

Ahora hay que actualizar el método **dibujarNodoSeleccionado(nodo),** hasta ahora este método solo cambiaba el estilo de dibujo del nodo seleccionado, vamos a hacer que ahora también cambie el de la **relación**.

|  |
| --- |
| dibujarNodoSeleccionado(nodo, relacion){  *// Nodo* for(var i=0; i < this.nodos.length; i++){  this.nodos[i].dibujarSeleccionado(false);  }  if ( nodo != null ) {  nodo.dibujarSeleccionado(true);  }   *// relación (solo si no hay nodo, porque a veces al pinchar  // en el nodo toca tambien la relación por proximidad* for (var i = 0; i < this.relaciones.length; i++) {  this.relaciones[i].dibujarSeleccionado(false);  }  if (relacion != null) {  relacion.dibujarSeleccionado(true);  }   this.dibujar(); } |

Revisamos el método **comprobarMouseDown(x, y),** además de seleccionar un nodo el click va a poder caer en una relación.

* Añadimos la detección de si el click cae en alguna relación. Recordamos que la selección de la relación va por “proximidad”, por lo que es técnicamente posible que el click caiga en un nodo y una relación a la vez. **Esto no nos interesa, comprobamos primero si cae en un nodo y solo si no ha caído en ningún miramos si cae en una relación**
* Selección de una zona “libre” para que sea libre además el click no ha tenido que caer en ninguna relación, por lo que se añade la condición   
  **&& relacionEnqueCaeElClick == null**
* Ahora existe una nueva posibilidad, que el usuario haga click en una relación. Esto sucede si no hay ninguna acción **accion == ACCION\_NO** y el click cae en una relación **relacionEnqueCaeElClick != null**
* Finalmente debemos recordar que el método **dibujarNodoSeleccionado(nodo,relación)** ahora también recibe la **relación** como parámetro

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);  var relacionEnqueCaeElClick = null;  if ( nodoEnQueCaeElClick == null){  relacionEnqueCaeElClick = this.auxPulsacionEnRelacion(x,y);  }   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX,y + scrollY);  this.nodos.push(nuevoNodo);  nodoEnQueCaeElClick = nuevoNodo;  *globalNingunaAccionSeleccionada*();  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Click normal en un nodo* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick != null) {  accion = ACCION\_MOVER\_NODO;  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Seleccionar una zona "libre"* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick == null && relacionEnqueCaeElClick == null ){  accion = ACCION\_SCROLL;  scrollFijado = {x: x, y: y};  *globalActualizarNodoActivo*(null);   *// Seleccionar Nodo para crear relación* } else if ( accion == ACCION\_CREAR\_RELACION && nodoEnQueCaeElClick != null) {  if (nodoRelacion1 == null){  console.log("nodoRelacion1")  nodoRelacion1 = nodoEnQueCaeElClick;  } else if (nodoRelacion2 == null){  console.log("nodoRelacion2")  nodoRelacion2 = nodoEnQueCaeElClick;  *// Crear la relación* if (nodoRelacion1 != null && nodoRelacion2 != null){  var agregado = nodoRelacion1.agregarNodoRelacionado(nodoRelacion2);  if ( agregado ){  console.log("nodoRelacion1")  this.relaciones.push(new Relacion(nodoRelacion1,nodoRelacion2))  }  }  *// Reiniciarlo todo  globalNingunaAccionSeleccionada*();  nodoRelacion1 = null;  nodoRelacion2 = null;  }  } else if (accion == ACCION\_NO && relacionEnqueCaeElClick != null){  *globalActualizarNodoActivo*(null, relacionEnqueCaeElClick);  }   this.dibujarNodoSeleccionado(nodoEnQueCaeElClick, relacionEnqueCaeElClick);  } |

Ejecutamos el programa y verificamos que podemos seleccionar una relación, debe marcarse de color naranja, también debe aparecer el nombre en las propiedades



# Borrar relaciones

Vamos a incluir un nuevo botón en el **index.html** para eliminar, estará dentro de las propiedades.

|  |
| --- |
| <div class="uk-width-1-4">  <div class="uk-card uk-card-default uk-card-body">  *<!-- Propiedades -->* <h2>Propiedades</h2>  <div class="uk-margin">  <input class="uk-input" type="text" id="nombreNodo">  </div>  <button class="uk-button uk-button-default uk-width-1-1 uk-margin-small-bottom"  onClick="clickBotonEliminarElemento()">Eliminar</button>  <button class="uk-button uk-button-primary uk-width-1-1 uk-margin-small-bottom"  onClick="*clickBotonEditarPropiedadesNodo*()">Guardar</button>  </div> </div> |

Accedemos a **Main.js** e implementamos la función **clickBotonEliminarElemento().** Va a llamar al Editor que es el que gestiona los datos y eliminar la relación, utilizamos el método **eliminarRelacion(relación)** (aun sin implementar).

|  |
| --- |
| function *iniciar*() {  editor = new Editor();  editor.dibujar(); }  function *clickBotonEliminarElemento*(){  if (relacionActiva != null) {  editor.eliminarRelacion(relacionActiva);  $('#nombreNodo').val("")  relacionActiva = null;  }  } |

Accedemos al **Editor.js** e implementamos el método **eliminarRelacion(relación).** Esta eliminación tiene dos pasos:

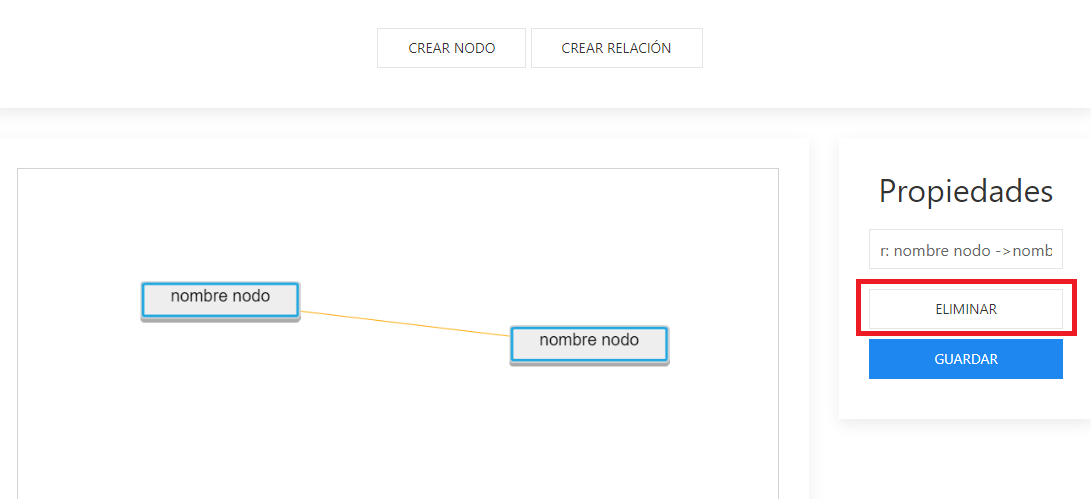
* Sacar de los nodos las relaciones, si la relación a eliminar se encuentra en alguno. Para ello se recorren todos y se prueba a llamar a eliminarRelacion (aun sin implementar)
* Sacar del array **relaciones[]** la relación a eliminar.

|  |
| --- |
| eliminarRelacion(relacionEliminar){  *// De los nodos* for (var i=0; i < this.nodos.length; i++){  this.nodos[i].eliminarRelacion(relacionEliminar);  }  *// Del dibujo* this.relaciones.splice(this.relaciones.indexOf(relacionEliminar), 1);   this.dibujar(); } |

Accedemos a **Nodo.js** e implementamos el método **eliminarRelacion(relación).** Hay que buscar si el propio nodo tiene alguna relación con el nodoB que figura en la relación, esa será la relación a eliminar

|  |
| --- |
| class Nodo {   constructor(x, y) {  this.imagen = cacheImagenes["nodo"];   this.x = x;  this.y = y;  this.ancho = this.imagen.width;  this.alto = this.imagen.height;  this.texto = "nombre nodo";   this.nodosRelacionados = [];  }   eliminarRelacion(relacion){  var nodoRelacionado = this.nodosRelacionados.filter(n => n == relacion.nodoB);  if ( nodoRelacionado != null){  this.nodosRelacionados.splice(this.nodosRelacionados.indexOf(nodoRelacionado), 1);  }  } |

Ejecutamos la aplicación, nos debería dejar seleccionar una relación y luego eliminarla.



# Borrar nodos

Borrar nodos es algo más complejo que borrar relaciones, ya que al borrar un nodo tienen que desaparecer también las relaciones en las que participa (al menos tal y como hemos planteado el editor)

Accedemos a **Main.js**, ampliamos la función **clickBotonEliminarElemento()** para que también tenga en cuenta el **nodoActivo**. Ahora debe decirle al editor que elimine el nodo y sus relaciones (método aun sin implementar).

|  |
| --- |
| function *clickBotonEliminarElemento*(){  if (relacionActiva != null) {  editor.eliminarRelacion(relacionActiva);  $('#nombreNodo').val("")  relacionActiva = null;  } else if (nodoActivo != null) {  editor.eliminarNodo(nodoActivo);  $('#nombreNodo').val("")  nodoActivo = null;  } } |

Abrimos **Editor.js** e implementamos el nuevo método **eliminarNodo(nodo)**.

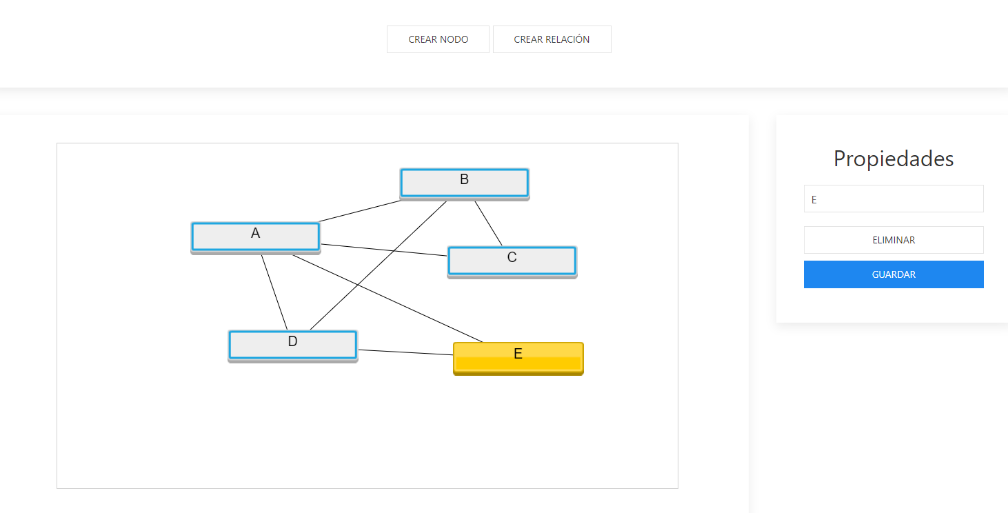
* El primer paso es sencillo sacar el nodo de **nodos[]**
* El segundo paso consiste en recorrer todos los **nodos[]** para ver si alguno tiene relación con el nodo que vamos a eliminar, implementaremos un nuevo método dentro de la clase Nodo que se llamará **eliminarRelacionesConNodo(nodo)**
* El ultimo paso consiste en eliminar las relaciones del dibujo, es decir los elementos de la colección **relaciones[]** en las que el nodo a eliminar participe de alguna manera, como inicio (nodoA) o como final (nodoB)

|  |
| --- |
| eliminarNodo(nodoEliminar){  this.nodos.splice(this.nodos.indexOf(nodoEliminar), 1);  *// ¿Otros nodos que tuviesen relaciones con ese nodo?   // Si algún nodo tiene relación con el eliminado, eliminar esas relaciones* for ( var j=0; j < this.nodos.length; j++){  this.nodos[j].eliminarRelacionConNodo(nodoEliminar);  }   *// Eliminar relaciones - dibujo, si el nodo afectado está en alguna parte* for(var i=0; i < this.relaciones.length; i++){  if ( this.relaciones[i].nodoA == nodoEliminar || this.relaciones[i].nodoB == nodoEliminar){  this.relaciones.splice(this.relaciones.indexOf(this.relaciones[i]), 1);  i--;  }  }   this.dibujar(); } |

Accedemos a **Nodo.js** e implementamos el nuevo método **eliminarRelacionConNodo(nodo)**

|  |
| --- |
| class Nodo {   constructor(x, y) {  this.imagen = cacheImagenes["nodo"];   this.x = x;  this.y = y;  this.ancho = this.imagen.width;  this.alto = this.imagen.height;  this.texto = "nombre nodo";   this.nodosRelacionados = [];  }   eliminarRelacionConNodo(nodo){  if ( nodo != null){  this.nodosRelacionados.splice(this.nodosRelacionados.indexOf(nodo), 1);  }  } |

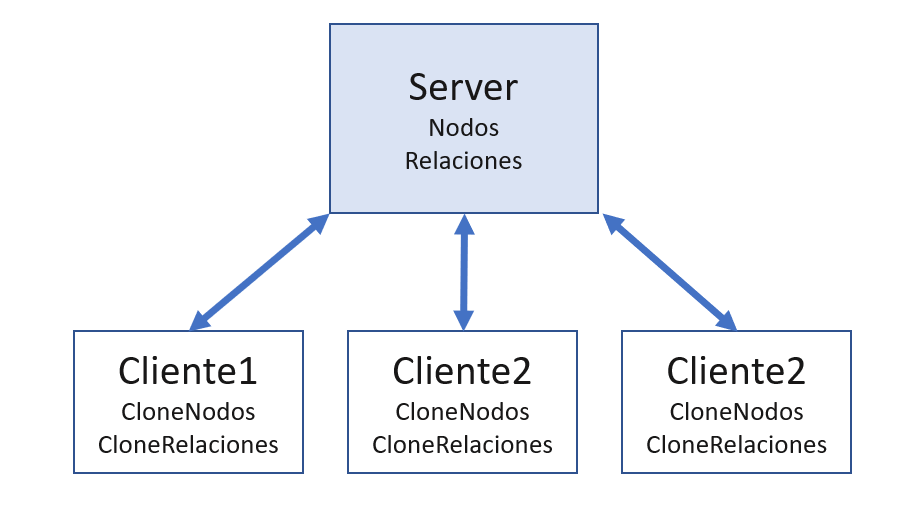
Probamos la aplicación, creamos una red de nodos con varias relaciones y probamos a eliminar uno de los nodos.



# Web Sockets (múltiples usuarios)

Actualmente todo estaba implementado en la parte del cliente (html). Vamos a modificarlo para hacer que el diagrama pueda ser compartido por varios usuarios a través de web sockets.

Los elementos del diagrama (nodos y relaciones) se van a almacenar en un servidor (backend aplicación node) , está información será compartida con los navegadores de los clientes.



Claves de la arquitectura

* Solo el **servidor** puede crear nodos y relaciones.
* Los nodos y relaciones se identifican de forma única, necesitamos **ids**.
* Los **clientes** notifican al **servidor** que quieren crear un nodo.
* El servidor notifica a los clientes los cambios en los nodos

Solo vamos a implementar una pequeña parte de la funcionalidad

Instalamos el paquete socket.io en el proyecto.

|  |
| --- |
| *npm install socket.io* |

Agregamos el socket en la parte del servidor, en el fichero **app.js.**

Guardamos el **server** en una variable, y lo utilizamos para inicializar el socket, creamos una lista **nodos** otra de **relaciones** y un contador de **id.**

|  |
| --- |
| const express = require('express'); const app = express(); const PORT = 3000;  app.use(express.static('public')); app.get('/', (req, res) => {  res.redirect("/index.html"); });  ~~app.listen(PORT, () => console.log(`Server listening on port: ${PORT}`));~~ const server = app.listen(PORT,console.log(`Server listening on port: ${PORT}`));  const io = require('socket.io')(server); var nodos = []; var relaciones = []; var id = 1; |

A continuación, inicializamos el **socket**, nos avisará con un mensaje de consola cada vez que se del evento connection (un nuevo cliente se conecta).

|  |
| --- |
| const io = require('socket.io')(server); var nodos = []; var relaciones = []; var id = 1;  *// Socket* io.on('connection', (socket) => {   *// Se acaba de conectar uno nuevo* console.log("Cliente conectado " + socket.id);  }); |

Vamos ahora a la parte del cliente, abrimos el **index.html** incluimos la librería de los sockets.

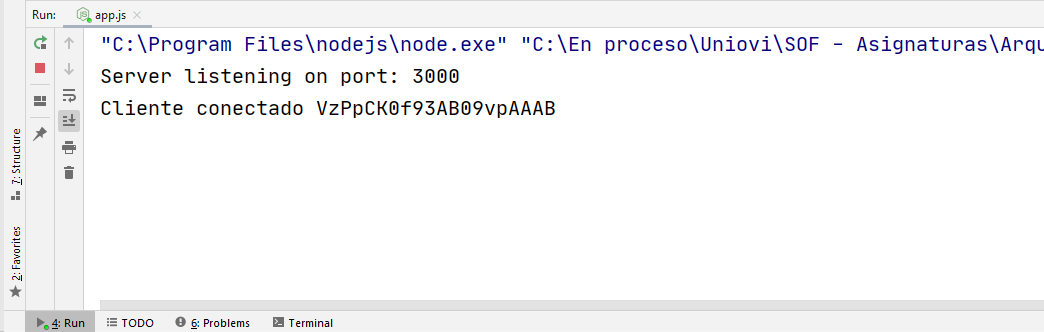
|  |
| --- |
| <!DOCTYPE html> <html> <head>  <meta charset="utf-8" />  <link rel="stylesheet" href="/css/uikit.css"/>  <meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate" />  <meta http-equiv="Pragma" content="no-cache" />  <meta http-equiv="Expires" content="0" />  <title>Editor</title>  <script src="/socket.io/socket.io.js"></script>  <script src="/js/jquery.min.js"></script>  <script src="/js/uikit.js"></script>  <script src="/js/uikit-icons.js"></script> </head> |

Continuamos en **Main.js** realizando la conexión del cliente al servidor.

|  |
| --- |
| var scrollX = 0; var scrollY = 0; var scrollFijado = null;  var socket = null;  function *iniciar*() {  editor = new Editor();  editor.dibujar();  socket = io("localhost:3000"); } |

Para verificar que funciona, ejecutamos el proyecto y abrimos en el navegador web <http://localhost:3000/>

La consola del servidor debería mostrarnos el siguiente mensaje (tantos mensajes como veces abramos la página web, por ejemplo, desde 2 navegadores).



Vamos a declarar la recepción de eventos en el servidor, abrimos el **app.js** y declaramos un evento llamado **crearNodo** , a partir de este momento los clientes pueden llamar al evento crearNodo.

Este evento recibe un objeto **msg** que contiene la información a almacenar de un nodo (x,y,texto), el servidor almacena está información en **nodos[]** y le asigna una **id.**

|  |
| --- |
| *// Socket* io.on('connection', (socket) => {   *// Se acaba de conectar uno nuevo* console.log("Cliente conectado " + socket.id);   *// Cliente nos envía mensaje* socket.on('crearNodo', (msg) => {  nodos.push( {  id: id,  x: msg.x,  y: msg.y,  texto: msg.texto  });  id++;   });  }); |

Vamos a hacer ahora que el cliente invoque este evento, entramos en **Editor.js** , el método **comprobarMouseDown(x,y**) era el que se encargaba de crear nodos. **socket.emit** nos permite enviar el dato, por el socket abierto (es decir al servidor)

El cliente no puede crear nodos por si mismo, avisa al servidor para que lo haga. Solo vamos a eliminar la parte en la que se incluye el nodo en la lista, en general este código es mejorable.

|  |
| --- |
| comprobarMouseDown(x, y){  var nodoEnQueCaeElClick = this.auxPulsacionEnNodo(x,y);  var relacionEnqueCaeElClick = null;  if ( nodoEnQueCaeElClick == null){  relacionEnqueCaeElClick = this.auxPulsacionEnRelacion(x,y);  }   *// Crear un nuevo nodo* if ( accion == ACCION\_CREAR\_NODO && nodoEnQueCaeElClick == null){  var nuevoNodo = new Nodo(x + scrollX,y + scrollY);  ~~this.nodos.push(nuevoNodo);~~  socket.emit('crearNodo', {  x: nuevoNodo.x,  y: nuevoNodo.y,  texto: nuevoNodo.texto  });  nodoEnQueCaeElClick = nuevoNodo;  *globalNingunaAccionSeleccionada*();  *globalActualizarNodoActivo*(nodoEnQueCaeElClick);   *// Click normal en un nodo* } else if (accion == ACCION\_NO && nodoEnQueCaeElClick != null) { |

La primera parte de la comunicación está implementada, los clientes notifican al servidor de que quieren crear nodos.

Ahora es el servidor el que tiene que notificar a los clientes que hay nuevos nodos, accedemos a **Main.js** y declaramos un evento, pero está ver en la parte del cliente. Este evento llamará al método **editor.socketActualizarNodos** (aun sin implementar).

|  |
| --- |
| var socket = null;  function *iniciar*() {  editor = new Editor();  editor.dibujar();  socket = io("localhost:3000");  socket.on('actualizarDiagrama', function(msg){  *// Me piden que actualice el diagrama,* console.log('actualizarDiagrama');  editor.socketActualizarNodos(msg.nodos);  }); } |

Accedemos al **Editor.js** e implementamos **socketActualizarNodos,** en este método nos va a llegar la lista actual de **nodos** (más bien su información relevante, x, y, texto, id).

* Se recorren los nodos que envía el servidor
  + ¿Existen ya en local?, coinciden las ids. En ese caso se actualiza la información
  + ¿No existen en local?, en ese caso se crean, recordar que hay que asignarle toda la información al nuevo nodo, **id** incluida.
  + No estamos contemplando aun el caso de borrar.

|  |
| --- |
| class Editor {   constructor() {  this.nodos = [];  this.relaciones = [];  }   socketActualizarNodos(socketNodos){  *// Si están los actualizo, si no los creo* for(var i=0; i < socketNodos.length ; i++){  var nodoLocal = this.nodos.find(nodo => nodo.id == socketNodos[i].id);   *// Ya estaba en el array local* if ( nodoLocal != null){  nodoLocal.x = socketNodos[i].x;  nodoLocal.y = socketNodos[i].y;  nodoLocal.texto = socketNodos[i].texto;   *// No estába en el array Local* } else {  var nuevoNodo = new Nodo(socketNodos[i].x,socketNodos[i].y);  nuevoNodo.nombre = socketNodos[i].nombre;  *// IMPORTANTISIMO para que no los replique* nuevoNodo.id = socketNodos[i].id;   this.nodos.push(nuevoNodo);  }  }  this.dibujar();  } |

Ahora nos vamos al servidor, en el fichero **app.js** vamos a hacer que se llame al evento del cliente **actualizarDiagrama.** Cada vez que se añade un nodo se lo vamos a notificar a TODOS los clientes conectados.

* **io.emit** envía la petición a todos los clientes conectados
* **socket.emit** por el contrario lo enviaría solo al socket concreto (es decir cliente que inicio la llamada)

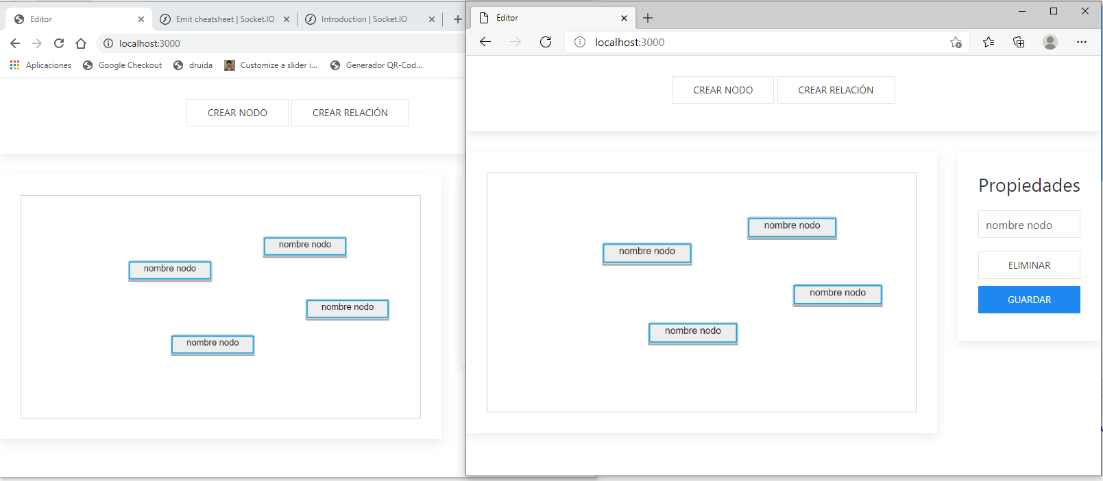
|  |
| --- |
| Existen muchas más posibilidades para enviar información (aunque no va a ser necesario utilizarlas), se pueden consultar todas aquí <https://socket.io/docs/v3/emit-cheatsheet/> |

|  |
| --- |
| *// Socket* io.on('connection', (socket) => {   *// Se acaba de conectar uno nuevo* console.log("Cliente conectado " + socket.id);   *// Cliente nos envia mensaje* socket.on('crearNodo', (msg) => {  nodos.push( {  id: id,  x: msg.x,  y: msg.y,  texto: msg.texto  });  id++;   *// io.emit se lo manda a TODOS los clientes  // socket.emit se lo mandaría solo a UNO* io.emit('actualizarDiagrama', { nodos: nodos });  }); }); |

También podemos hacer que siempre se que se conecte un nuevo cliente se envíe el diagrama actual. Solo al nuevo cliente, no hace falta enviárselo a todos.

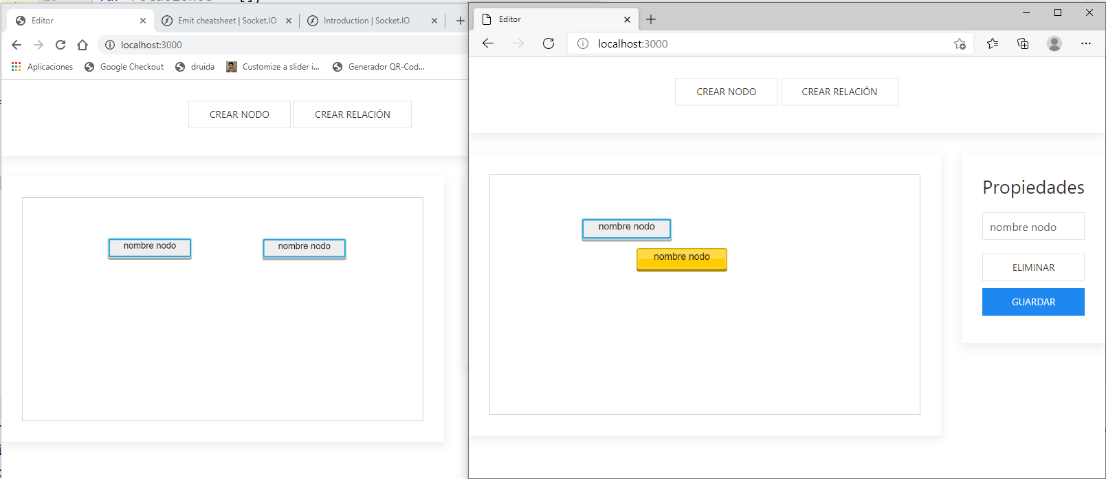
|  |
| --- |
| *// Socket* io.on('connection', (socket) => {   *// Se acaba de conectar uno nuevo* console.log("Cliente conectado " + socket.id);  socket.emit('actualizarDiagrama', { nodos: nodos });    *// Cliente nos envia mensaje* socket.on('crearNodo', (msg) => { |

Ejecutamos el servidor de nuevo y probamos a abrir dos navegadores. Los nodos que creamos en uno tienen que aparecer en otro. De momento eso es lo único que funciona.



Sí movemos los nodos (modificar su coordenada x e y), solo se modifica en la versión local.

Cuando se actualizan los nodos vuelven a su versión anterior.



Hace falta ir al servidor **app.js** e incluir un nuevo evento para modificaciones de nodos. Los clientes llamarán a este evento, para actualizar un nodo.

|  |
| --- |
| *// Socket* io.on('connection', (socket) => {   *// Se acaba de conectar uno nuevo* console.log("Cliente conectado " + socket.id);  socket.emit('actualizarDiagrama', { nodos: nodos });   *// Cliente nos envia mensaje* socket.on('modificarNodo', (msg) => {  var nodoAfectado = nodos.find(nodo => nodo.id == msg.id);  nodoAfectado.x = msg.x;  nodoAfectado.y = msg.y;  nodoAfectado.texto = msg.texto;  io.emit('actualizarDiagrama', { nodos: nodos });  });    *// Cliente nos envia mensaje* socket.on('crearNodo', (msg) => { |

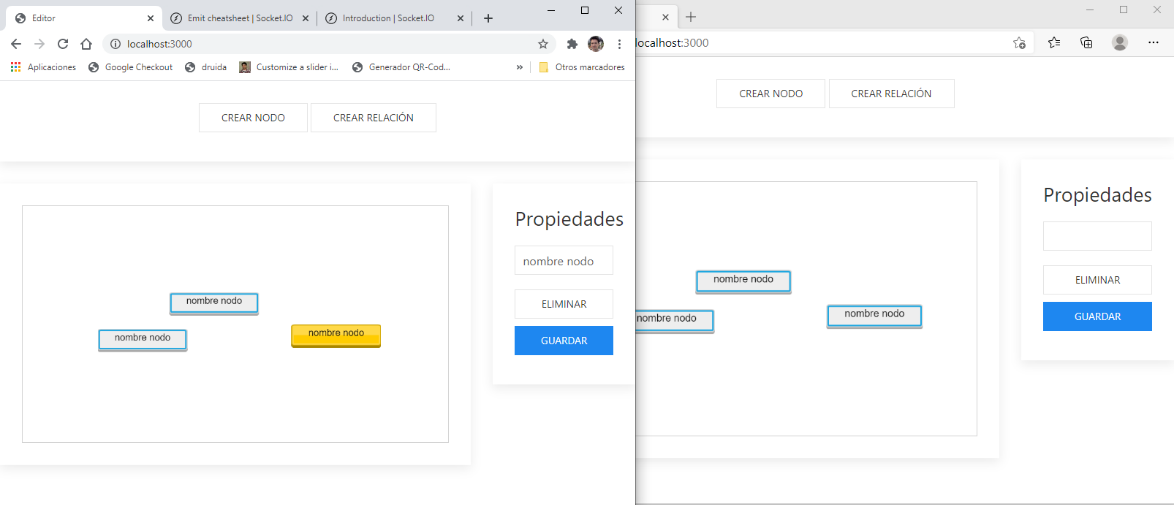
Accedemos al **Editor.js** , el método **comprobarMouseMove(x,y)** es el que se encarga de desplazar los nodos. Cada vez que uno se desplaza habría que notificárselo al servidor, para que este avise a todos los clientes.

|  |
| --- |
| comprobarMouseMove(x, y){  *// Hay nodo seleccionado* if ( accion == ACCION\_MOVER\_NODO && nodoActivo != null ){  nodoActivo.x = x + scrollX;  nodoActivo.y = y + scrollY;  socket.emit('modificarNodo', nodoActivo );  this.dibujar();  }  *// Hay scroll* if ( accion == ACCION\_SCROLL && scrollFijado != null ){  scrollX += scrollFijado.x - x;  scrollY += scrollFijado.y - y;  scrollFijado = { x: x, y: y}; *// Dejar el Scroll en el punto …* this.dibujar();  } } |

Ejecutamos la aplicación y comprobamos el funcionamiento. Funciona, pero con lag en el cliente que mueve el nodo.

Se debe a que el cliente le notifica la nueva posición al servidor, y el servidor se la vuelve a enviar para que vuelva a recolocar el nodo en la posición que ya tenía.

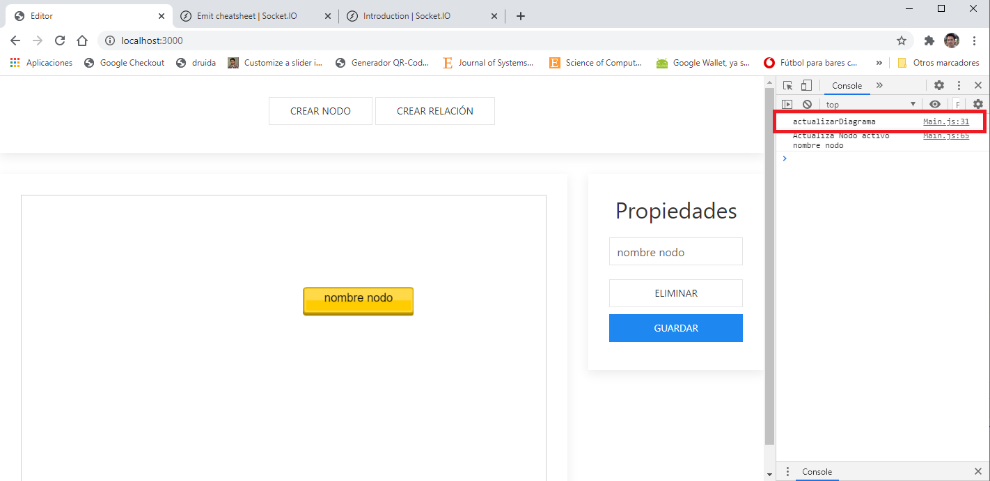
**Podemos hacer que la actualización se envíe a todos menos al cliente que ya ha movido el nodo.**



Volvemos al servidor **app.js** y cambiamos la forma de emitir el dato con **socket.broadcast.emit**, esto envía el dato a todos menos al propio cliente.

|  |
| --- |
| *// Socket* io.on('connection', (socket) => {   *// Se acaba de conectar uno nuevo* console.log("Cliente conectado " + socket.id);  socket.emit('actualizarDiagrama', { nodos: nodos });   *// Cliente nos envia mensaje* socket.on('modificarNodo', (msg) => {  var nodoAfectado = nodos.find(nodo => nodo.id == msg.id);  nodoAfectado.x = msg.x;  nodoAfectado.y = msg.y;  nodoAfectado.texto = msg.texto; socket.broadcast.emit('actualizarDiagrama', { nodos: nodos });  }); |

Ahora sí, cuando un cliente mueve un nodo no recibe actualizaciones de ese mismo movimiento. En la captura se ve que solo recibe la actualización inicial.



No vamos a acabar toda la funcionalidad, faltarían cosas como:

* Notificar la modificación de texto
* Pasar la gestión de las relaciones al servidor (muy similar a lo de los nodos)
* Eliminar nodos y relaciones