

Laboratorio 7 Blue Prints + Sockets

Sebastián Cardona, Laura Gil, Zayra Gutiérrez

**Ingeniero de Sistemas
Javier Toquica Barrera**

Universidad Escuela Colombiana de ingeniería Julio Garavito

2025-1

Contenido

Introducción	3
Desarrollo del Laboratorio.....	4
Parte 1	4
Parte 2	9
Conclusiones	17

Introducción

En el desarrollo de aplicaciones web interactivas, la captura y gestión de eventos es fundamental para mejorar la experiencia del usuario y la funcionalidad de la aplicación. Este proyecto tiene como objetivo la implementación de un manejador de eventos para capturar 'clicks' en un elemento, permitiendo la manipulación de dibujos en tiempo real. A través del uso de eventos y la comunicación en tiempo real con WebSockets y STOMP, se busca desarrollar una herramienta de dibujo colaborativa. La aplicación permitirá la gestión de planos, incluyendo su creación, edición y eliminación, y la propagación de eventos en tiempo real entre distintos clientes conectados.

Se modularizará el código para garantizar la correcta separación de responsabilidades, facilitando la escalabilidad y mantenibilidad del sistema. Además, se implementará la comunicación con un servidor Spring Boot que actuará como un broker de mensajes, permitiendo la transmisión eficiente de eventos entre los clientes.

El desarrollo se llevará a cabo en varias fases, asegurando la correcta captura y replicación de eventos, el manejo de múltiples dibujos simultáneos y la construcción de polígonos de manera colaborativa. De esta manera, se busca crear una aplicación robusta y eficiente para la edición compartida de planos en tiempo real.

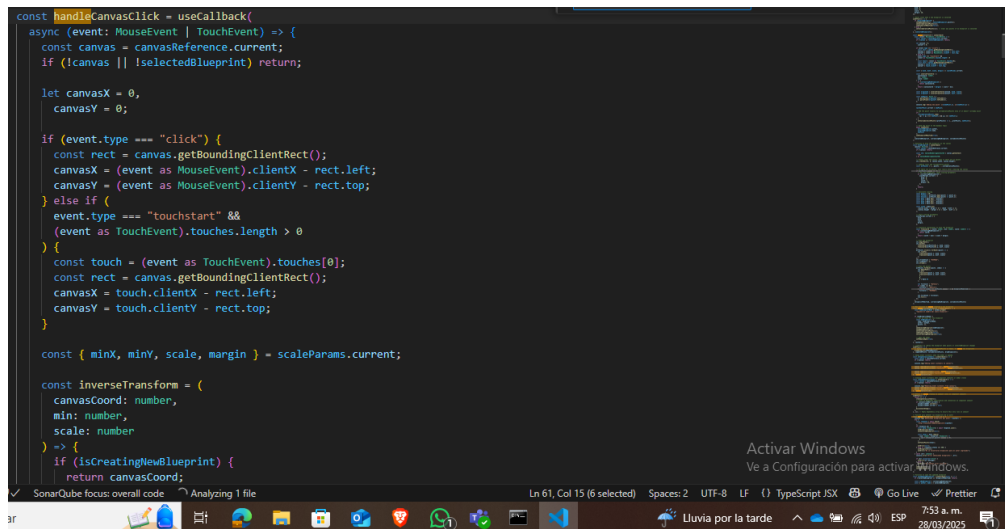
Toda la implementación del ejercicio se encuentra disponible en el siguiente repositorio de GitHub: <https://github.com/SebastianCardona-P/ARSW-Lab7.git>

Desarrollo del Laboratorio

Parte 1

1. Agregue al canvas de la página un manejador de eventos que permita capturar los 'clicks' realizados, bien sea a través del mouse, o a través de una pantalla táctil. Para esto, tenga en cuenta [este ejemplo de uso de los eventos de tipo 'PointerEvent'](#) (aún no soportado por todos los navegadores) para este fin. Recuerde que a diferencia del ejemplo anterior (donde el código JS está incrustado en la vista), se espera tener la inicialización de los manejadores de eventos correctamente modularizado, tal [como se muestra en este codepen](#).

A continuación, nuestro trabajo fue hecho en react, para desarrollar esta parte, construimos una función en el componente de Blueprint.tsx para capturar los clicks y toques de pantalla en el canvas, guardar las coordenadas, y las originales sin escalar:



```
const handleCanvasClick = useCallback(
  async (event: MouseEvent | TouchEvent) => {
    const canvas = canvasReference.current;
    if (!canvas || !selectedBlueprint) return;

    let canvasX = 0,
        canvasY = 0;

    if (event.type === "click") {
      const rect = canvas.getBoundingClientRect();
      canvasX = (event as MouseEvent).clientX - rect.left;
      canvasY = (event as MouseEvent).clientY - rect.top;
    } else if (
      event.type === "touchstart" &&
      (event as TouchEvent).touches.length > 0
    ) {
      const touch = (event as TouchEvent).touches[0];
      const rect = canvas.getBoundingClientRect();
      canvasX = touch.clientX - rect.left;
      canvasY = touch.clientY - rect.top;
    }

    const { minX, minY, scale, margin } = scaleParams.current;

    const inverseTransform = {
      canvasCoord: number,
      min: number,
      scale: number
    } => {
      if (isCreatingNewBlueprint) {
        return canvasCoord;
      }
    }
  },
  [selectedBlueprint, scaleParams]
);
```

2. Agregue lo que haga falta en sus módulos para que cuando se capturen nuevos puntos en el canvas abierto (si no se ha seleccionado un canvas NO se debe hacer nada):
 - i. Se agregue el punto al final de la secuencia de puntos del canvas actual (¡sólo en la memoria de la aplicación, AÚN NO EN EL API!).
 - ii. Se repinte el dibujo.

Para esto se guardan los nuevos puntos dentro

```
const originalX = inverseTransform(canvasX, minX, scale);
const originalY = inverseTransform(canvasY, minY, scale);

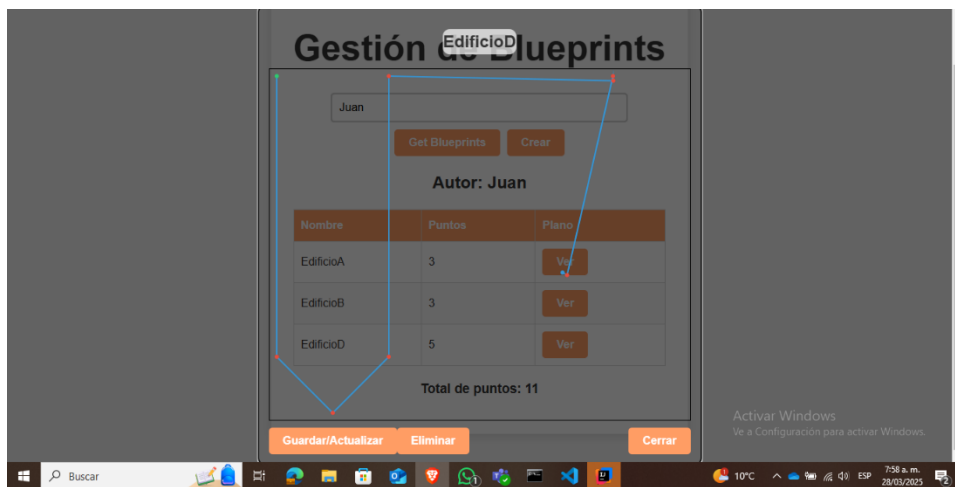
const newPoint: Point = {
  x: parseFloat(originalX.toFixed(2)),
  y: parseFloat(originalY.toFixed(2)),
};

console.log('Adding new point: x=${newPoint.x}, y=${newPoint.y}');

lastSentPoint.current = newPoint;

// Add the point locally to collaborativePoints only if it doesn't already exist
if (
  !collaborativePoints.some(
    (p) => p.x === newPoint.x && p.y === newPoint.y
  )
) {
  setCollaborativePoints((prevPoints) => [...prevPoints, newPoint]);
}

// Send the point to the dynamic topic
await sendPointToTopic(newPoint);
```



3. *Agregue el botón Save/Update. Respetando la arquitectura de módulos actual del cliente, haga que al oprimirse el botón:*
 - i. *Se haga PUT al API, con el plano actualizado, en su recurso REST correspondiente.*
 - ii. *Se haga GET al recurso /blueprints, para obtener de nuevo todos los planos realizados.*
 - iii. *Se calculen nuevamente los puntos totales del usuario.*

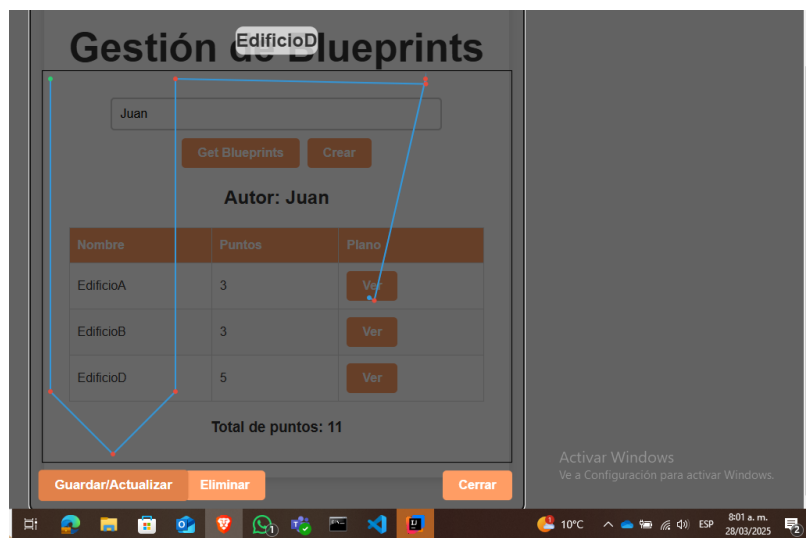
El botón fue agregado en el canvas, y su funcionalidad hace que se llame al backend para hacer la petición put, enviando los nuevos puntos guardados en memoria

```
});
} else {
  response = await fetch(
    `http://localhost:8080/blueprints/${author}/${name}`,
    {
      method: "PUT",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(blueprintToSave),
    }
  );
}

if (!response.ok) {
  throw new Error("Error saving blueprint");
}

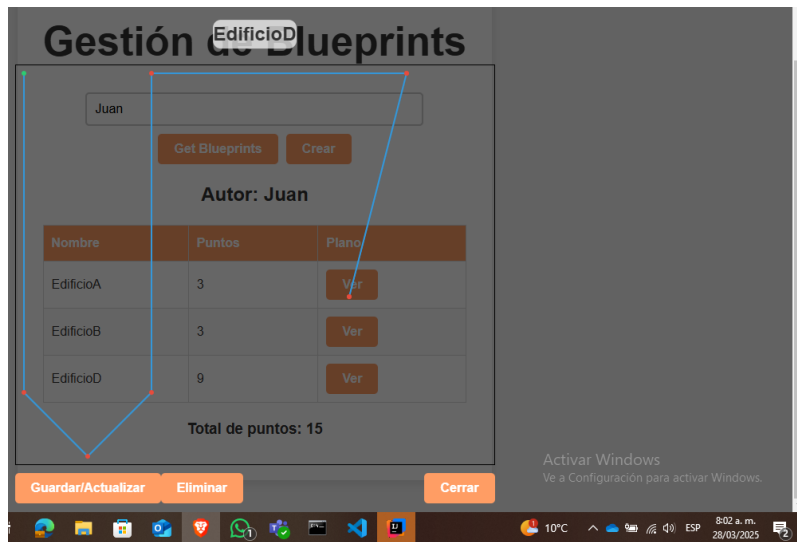
if (isNewBlueprint) {
  setBlueprints((prevBlueprints) => [...prevBlueprints, blueprintToSave]);
} else {
  const updatedBlueprints = blueprintToSave.blueprints.map((bp) => {
    bp.name === name ? { ...bp, points: blueprintToSave.points } : bp
  });
  setBlueprints(updatedBlueprints);
}

setBlueprintModified(false);
setIsCreatingBlueprint(false);
setCollaborativePoints([]);
}
```



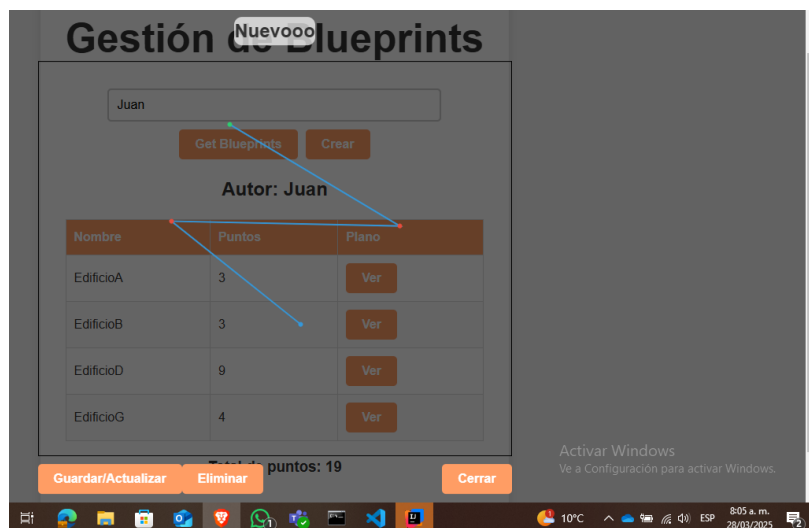
Cuando volvemos a consultar el mismo punto ya está actualizado





4. Agregue el botón 'Create new blueprint', de manera que cuando se oprima:
 - o Se borre el canvas actual.
 - o Se solicite el nombre del nuevo 'blueprint' (usted decide la manera de hacerlo).
 - iii. Hacer POST al recurso /blueprints, para crear el nuevo plano.
 - iv. Hacer GET a este mismo recurso, para actualizar el listado de planos y el puntaje del usuario.
5. Agregue el botón 'DELETE', de manera que (también con promesas):
 - o Borre el canvas.
 - o Haga DELETE del recurso correspondiente.
 - o Haga GET de los planos ahora disponibles.

El botón crear solo se habilita cuando hay un autor escrito, cuando se crea un nuevo blueprint, pedimos el nombre del blueprint, creamos un canvas vacío y esperamos que el autor haga click en el botón guardar / actualizar, para hacer la petición al backend




```
// Function to save the updated blueprint
const saveUpdatedBlueprint = useCallback(async () => {
  if (!selectedBlueprint || !blueprintModified) return;

  const isNewBlueprint = !isCreatingNewBlueprint;
  const { author, name } = selectedBlueprint;

  const blueprintToSave = {
    author: author,
    name: name,
    points: [...updatedPoints, ...collaborativePoints],
  };

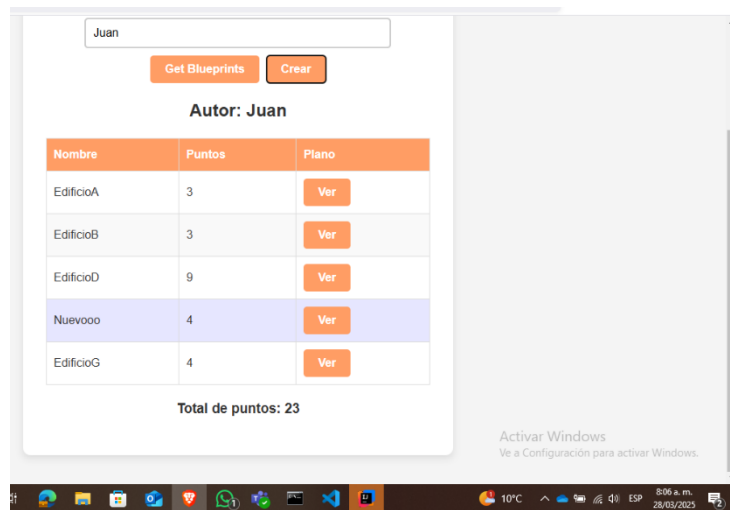
  try {
    let response;
    if (!isNewBlueprint) {
      response = await fetch('http://localhost:8080/blueprints', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(blueprintToSave),
      });
    } else {
      response = await fetch(
        `http://localhost:8080/blueprints/${author}/${name}`,
        {
          method: 'PUT',
          headers: {
            'Content-Type': 'application/json',
          },
          body: JSON.stringify(blueprintToSave),
        },
      );
    }

    if (!response.ok) throw new Error('Error saving blueprint');
  } catch (error) {
    console.error('Error saving blueprint:', error);
    setError('Error al guardar el blueprint');
  }

  // Update state
  setBlueprints((prevBlueprints) => [...prevBlueprints, blueprintToSave]);
  setBlueprintModified(false);
  setCollaborativePoints([]);
  handleSetBlueprints();
  setModelIsOpen(false);
} catch (error) {
  console.error('Error saving blueprint:', error);
  setError('Error al guardar el blueprint');
}

// Call the function
saveUpdatedBlueprint();
}, [selectedBlueprint, blueprintModified, updatedPoints, collaborativePoints, blueprints, isCreatingNewBlueprint]);
```

Cuando guardamos el blueprint, se muestra la lista actualizada



Parte 2

Para las partes I y II, usted va a implementar una herramienta de dibujo colaborativo Web, basada en el siguiente diagrama de actividades:

Haga que la aplicación HTML5/JS al ingresarle en los campos de X y Y, además de graficarlos, los publique en el tópic: `/topic/newpoint` . Para esto tenga en cuenta (1) usar el cliente STOMP creado en el módulo de JavaScript y (2) enviar la representación textual del objeto JSON (usar `JSON.stringify`).

```
/**
 * Endpoint STOMP para recibir un nuevo punto y enviarlo a todos los suscriptores
 * del topic "/topic/newpoint/{author}/{name}".
 * @param author Nombre del autor.
 * @param name Nombre del blueprint.
 * @param point Punto a enviar.
 */
@MessageMapping("/newpoint/{author}/{name}") // El destino para recibir el mensaje no usages  LaaSofia
public void handleNewPoint(
    @DestinationVariable("author") String author,
    @DestinationVariable("name") String name,
    Point point) {
    // Enviar el punto a todos los suscriptores del topic "/topic/newpoint/{author}/{name}"
    System.out.println("Enviando punto: " + point + " a /topic/newpoint/" + author + "/" + name);

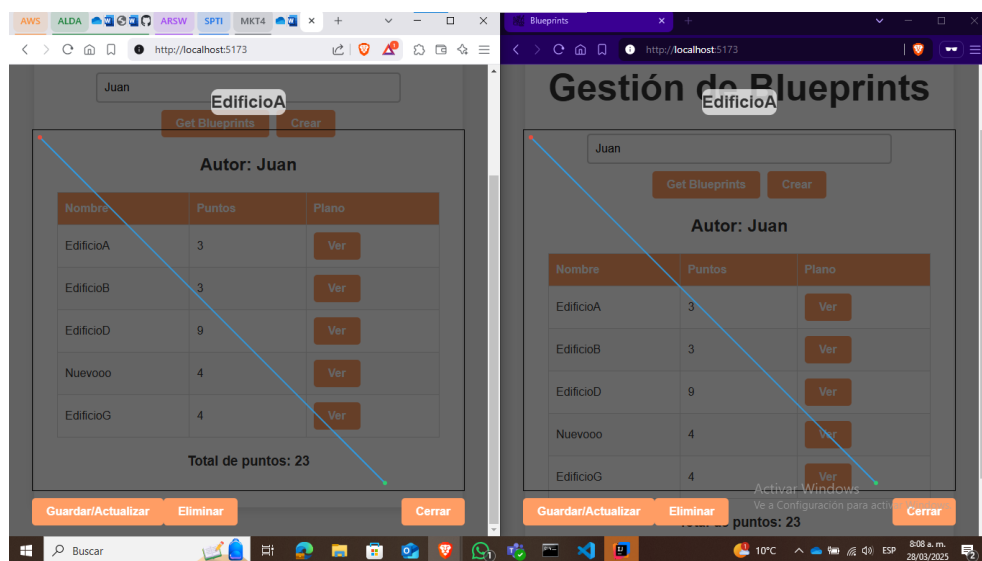
    simpMessagingTemplate.convertAndSend(destination: "/topic/newpoint/" + author + "/" + name, point);
}
```

Dentro del módulo JavaScript modifique la función de conexión/suscripción al WebSocket, para que la aplicación se suscriba al tópic `/topic/newpoint` (en lugar del tópic `/TOPICOXX`). Asocie como 'callback' de este suscriptor una función que muestre en un mensaje de alerta (`alert()`) el evento recibido. Como se sabe que en el tópic indicado se publicarán sólo puntos, extraiga el contenido enviado con el evento (objeto JavaScript en versión de texto), conviértalo en objeto JSON, y extraiga de éste sus propiedades (coordenadas X y Y). Para extraer el contenido del evento use la propiedad 'body' del mismo, y para convertirlo en objeto, use `JSON.parse`.

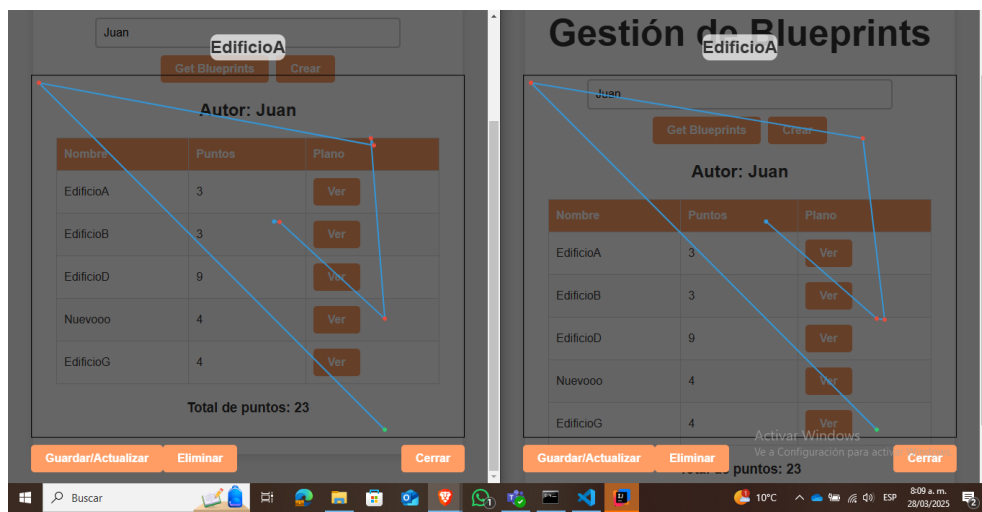
```
64 export const subscribeToNewPoints = async (
65   author: string,
66   name: string,
67   callback: (point: { x: number; y: number }) => void
68 ) => {
69   let subscription: { unsubscribe: () => void } | null = null;
70
71   const topic = `/topic/newpoint/${author}/${name}`;
72   console.log(`Attempting to subscribe to ${topic}`);
73
74   // Ensure the STOMP client is connected before subscribing
75   if (!stompClient.connected) {
76     await connectStomp();
77   }
78
79   // Now that the client is connected, subscribe
80   if (stompClient.connected && !subscription) {
81     subscription = stompClient.subscribe(topic, (message: IMessage) => {
82       console.log(`Received message on ${topic}:`, message.body);
83       try {
84         const point = JSON.parse(message.body);
85         if (point.x !== undefined && point.y !== undefined) {
86           callback(point);
87         } else {
88           console.warn("Invalid point received:", point);
89         }
89       }
90     });
91   }
92 }
```

Parte II.

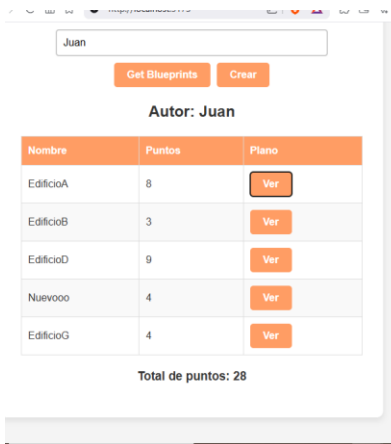
Para hacer mas útil la aplicación, en lugar de capturar las coordenadas con campos de formulario, las va a capturar a través de eventos sobre un elemento de tipo `<canvas>`. De la misma manera, en lugar de simplemente mostrar las coordenadas enviadas en los eventos a través de 'alertas', va a dibujar dichos puntos en el mismo canvas. Haga uso del mecanismo de captura de eventos de mouse/táctil usado en ejercicios anteriores con este fin. Haga que el 'callback' asociado al tópico `/topic/newpoint` en lugar de mostrar una alerta, dibuje un punto en el canvas en las coordenadas enviadas con los eventos recibidos. Para esto puede dibujar un círculo de radio 1. Ejecute su aplicación en varios navegadores (y si puede en varios computadores, accediendo a la aplicación mediante la IP donde corre el servidor). Compruebe que a medida que se dibuja un punto, el mismo es replicado en todas las instancias abiertas de la aplicación.



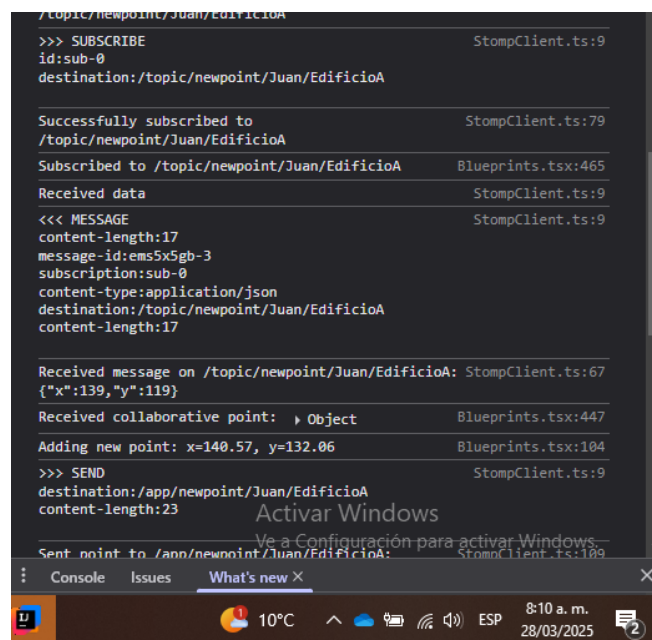
Cuando dibuje de una parte se mostrará de la otra:



Cuando de en guardar se actualiza con el total de puntos



Asi se ven los mensajes de log, identificando como se suscribe y como enviamos los puntos al servidor stomp



Parte III.

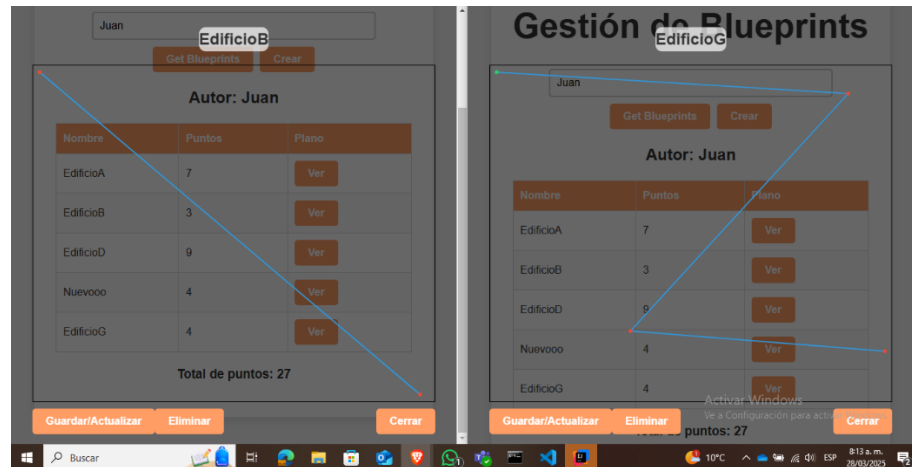
Ajuste la aplicación anterior para que pueda manejar más de un dibujo a la vez, manteniendo tópicos independientes. Para esto: Agregue un campo en la vista, en el cual el usuario pueda ingresar un número. El número corresponderá al identificador del dibujo que se creará. Modifique la aplicación para que, en lugar de conectarse y suscribirse automáticamente (en la función `init()`), lo haga a través de botón 'conectarse'. Éste, al oprimirse debe realizar la conexión y suscribir al cliente a un tópico que tenga un nombre dinámico, asociado el identificador ingresado, por ejemplo: `/topic/newpoint.25`, `topic/newpoint.80`, para los dibujos 25 y 80 respectivamente. De la misma manera, haga que las publicaciones se realicen al tópico asociado al

identificador ingresado por el usuario. Rectifique que se puedan realizar dos dibujos de forma independiente, cada uno de éstos entre dos o más clientes.

El backend modificamos la suscripción para suscribirse a topic/newpoint/autor/nombre de esta manera solo los suscritos en dicho blueprint verán los cambios

```
tsconfig.app.json package.json Blueprints.tsx App.css TS StompClient.ts App.tsx main.tsx index.html package-lock.json
> components > TS StompClient.ts > ...
8 |
9 | export const subscribeToNewPoints = async (
10 |   author: string,
11 |   name: string,
12 |   callback: (point: { x: number; y: number }) => void
13 | ) => {
14 |   let subscription: { unsubscribe: () => void } | null = null;
15 |
16 |   const topic = `topic/newpoint/${author}/${name}`;
17 |   console.log('Attempting to subscribe to ${topic}');
18 |
19 |   // Ensure the STOMP client is connected before subscribing
20 |   if (!stompClient.connected) {
21 |     await connectStomp();
22 |   }
23 |
24 |   // Now that the client is connected, subscribe
25 |   if (stompClient.connected && !subscription) {
26 |     subscription = stompClient.subscribe(topic, (message: IMessage) => {
27 |       console.log('Received message on ${topic}:', message.body);
28 |       try {
29 |         const point = JSON.parse(message.body);
30 |         if (point.x !== undefined && point.y !== undefined) {
31 |           callback(point);
32 |         } else {
33 |           console.warn('Invalid point received:', point);
34 |         }
35 |       } catch (error) {
36 |         console.error('Error parsing message body:', error);
37 |       }
38 |     });
39 |     console.log('Successfully subscribed to ${topic}');
40 |   }
41 | }
42 |
```

Se abrieron dos blueprints diferentes



Los cambios que pasan en una, no se muestran en la otra, porque ambos clientes están conectados a tópicos diferentes.

```
STOMP client is already active StompClient.ts:45
Attempting to subscribe to StompClient.ts:57
/topic/newpoint/Juan/EdificioG
>>> SUBSCRIBE StompClient.ts:9
id:sub-1
destination:/topic/newpoint/Juan/EdificioG

Successfully subscribed to StompClient.ts:79
/topic/newpoint/Juan/EdificioG
Subscribed to /topic/newpoint/Juan/EdificioG Blueprints.tsx:465
Adding event listeners to canvas Blueprints.tsx:264
Removing event listeners from canvas Blueprints.tsx:278
>>> UNSUBSCRIBE StompClient.ts:9
id:sub-1

Unsubscribed from /topic/newpoint/Juan/EdificioG StompClient.ts:89
```

```
Received message on StompClient.ts:67
/topic/newpoint/Juan/EdificioB: {"x":17,"y":82}
Received collaborative point: ▶ Object Blueprints.tsx:447
Adding new point: x=22.82, y=44.68 Blueprints.tsx:104
>>> SEND StompClient.ts:9
destination:/app/newpoint/Juan/EdificioB
content-length:21

Sent point to /app/newpoint/Juan/EdificioB: StompClient.ts:109
▶ Object

Received data StompClient.ts:9
<<< MESSAGE StompClient.ts:9
content-length:15
message-id:dgxxxxnnj-14
subscription:sub-3
content-type:application/json
destination:/topic/newpoint/Juan/EdificioB
content-length:15

Received message on StompClient.ts:67
/topic/newpoint/Juan/EdificioB: {"x":22,"y":44}
Received collaborative point: ▶ Object Blueprints.tsx:447
Obteniendo blueprints del autor: Juan Blueprints.tsx:298
```

Parte IV.

Para la parte IV, usted va a implementar una versión extendida del modelo de actividades y eventos anterior, en la que el servidor (que hasta ahora sólo fungía como Broker o MOM -Message Oriented Middleware-) se volverá también suscriptor de ciertos eventos, para a partir de los mismos agregar la funcionalidad de 'dibujo colaborativo de polígonos': Para esto, se va a hacer una configuración alterna en la que, en lugar de que se propaguen los mensajes 'newpoint.{numdibujo}' entre todos los clientes, éstos sean recibidos y procesados primero por el servidor, de manera que se pueda decidir qué hacer con los mismos. Para ver cómo manejar esto desde el manejador de eventos STOMP del servidor, revise puede revisar la documentación de Spring.

Cree una nueva clase que haga el papel de 'Controlador' para ciertos mensajes STOMP (en este caso, aquellos enviados a través de `/app/newpoint.{numdibujo}`). A este controlador se le inyectará un bean de tipo `SimpMessagingTemplate`, un Bean de Spring que permitirá publicar eventos en un determinado tópico. Por ahora, se definirá que cuando se intercepten los eventos enviados a `/app/newpoint.{numdibujo}` (que se supone deben incluir un punto), se mostrará por pantalla el punto recibido, y luego se procederá a reenviar el evento al tópico al cual están suscritos los clientes `/topic/newpoint`.

```
/**
 * Endpoint STOMP unificado para manejo de puntos y poligonos
 */
@Controller
@MessageMapping("/newpoint/{author}/{name}")
public class BlueprintController {

    @DestinationVariable("author") String author,
    @DestinationVariable("name") String name,
    Point point) {

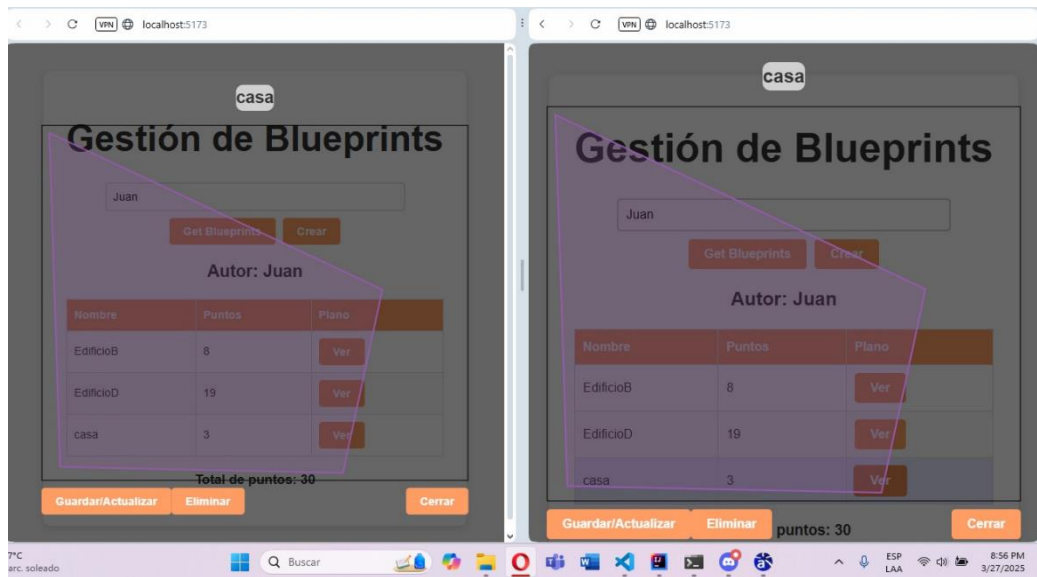
        String numdibujo = author+name;
        System.out.println(point.getX());
        System.out.println(point.getY());
        if (drawingPoints.containsKey(numdibujo)){
            drawingPoints.get(numdibujo).add(point);
        } else {
            ArrayList<Point> arrayList = new ArrayList<>();
            arrayList.add(point);
            drawingPoints.put(numdibujo, arrayList);
        }

        simpMessagingTemplate.convertAndSend(destination: "/topic/newpoint/" + author + "/" + name, point);
        if (drawingPoints.get(numdibujo).size() >= 4){
            simpMessagingTemplate.convertAndSend(destination: "/topic/newpolygon/" + author + "/" + name, drawingPoints.get(numdibujo));
            drawingPoints.put(numdibujo, new ArrayList<>());
        }
    }
}
```

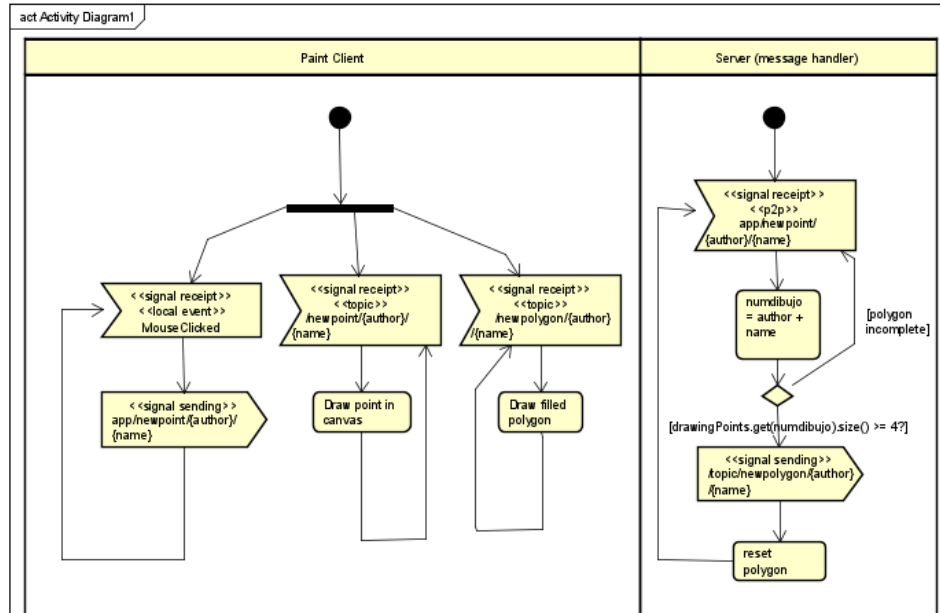
Ajuste su cliente para que, en lugar de publicar los puntos en el tópico `/topic/newpoint.{numdibujo}`, lo haga en `/app/newpoint.{numdibujo}`. Ejecute de nuevo la aplicación y rectifique que funcione igual, pero ahora mostrando en el servidor los detalles de los puntos recibidos. Una vez rectificado el funcionamiento, se quiere aprovechar este 'interceptor' de eventos para cambiar ligeramente la funcionalidad: Se va a manejar un nuevo tópico llamado `/topic/newpolygon.{numdibujo}`, en donde el lugar de puntos, se recibirán objetos javascript que tengan como propiedad un conjunto de puntos. El manejador de eventos de `/app/newpoint.{numdibujo}`, además de propagar los puntos a través del tópico `/topic/newpoints`, llevará el control de los puntos recibidos (que podrán haber sido dibujados por diferentes clientes). Cuando se completen tres o más puntos, publicará el polígono en el tópico `/topic/newpolygon`. Recuerde que esto se realizará concurrentemente, de manera que **REVISE LAS POSIBLES CONDICIONES DE CARRERA!**. También tenga en cuenta que desde el manejador de eventos del servidor se tendrán N dibujos independientes!. Verifique la funcionalidad: igual a la anterior, pero ahora dibujando polígonos cada vez que se agreguen cuatro puntos.

El cliente, ahora también se suscribirá al tópico `/topic/newpolygon`. El 'callback' asociado a la recepción de eventos en el mismo debe, con los datos recibidos, dibujar un polígono.

```
static > src > components > TS StompClients > subscribeToNewPolygons
130 export const subscribeToNewPolygons = async (
131   author: string,
132   blueprintName: string,
133   callback: (points: { x: number; y: number }[]) => void
134 ): Promise<() => void> => {
135   try {
136     await connectStomp();
137
138     if (!stompClient.connected) {
139       throw new Error("STOMP client is not connected after attempting to connect.");
140     }
141
142     const topic = `/topic/newpolygon/${author}/${blueprintName}`;
143     console.log("Subscribing to polygon topic: ${topic}");
144
145     const subscription = stompClient.subscribe(topic, (message: IMessage) => {
146       try {
147         const points = JSON.parse(message.body) as { x: number; y: number }[];
148         console.log("Received polygon with ${points.length} points");
149         callback(points);
150       } catch (error) {
151         console.error("Error parsing polygon points:", error);
152       }
153     });
154
155     return () => {
156       subscription.unsubscribe();
157       console.log("Unsubscribed from polygon topic: ${topic}");
158     };
159   } catch (error) {
160     console.error("Error setting up polygon subscription:", error);
161     return () => {};
```



A partir de los diagramas dados en el archivo ASTAH incluido, haga un nuevo diagrama de actividades correspondiente a lo realizado hasta este punto, teniendo en cuenta el detalle de que ahora se tendrán tópicos dinámicos para manejar diferentes dibujos simultáneamente.



Conclusiones

La implementación de blueprints ha permitido explorar distintas tecnologías y técnicas de desarrollo web, destacando la importancia de la captura de eventos, la comunicación en tiempo real y la modularización del código. La integración de eventos junto con WebSockets y STOMP, ha facilitado la gestión y sincronización de polígonos en múltiples clientes.

El uso de Spring Boot como servidor ha permitido implementar un broker de mensajes eficiente, mejorando la propagación de eventos y la interacción entre usuarios.

Finalmente, la evolución del proyecto desde la captura de puntos hasta la generación de polígonos ha demostrado la relevancia de manejar correctamente las condiciones de concurrencia y la suscripción a tópicos dinámicos. Este proyecto no solo proporciona una base para futuras mejoras, sino que también ilustra el potencial de las aplicaciones web colaborativas en entornos de edición y diseño en tiempo real.