



Laboratorio 7 - BluePrints parte 24

Andrés Felipe Arias Ajiaco

Cesar David Amaya Gómez

Johan Sebastián Gracia Martínez

Sebastián David Blanco Rodríguez

Universidad Escuela Colombiana de ingeniería Julio Gravito

Arquitecturas de software

Ing. Javier Iván Toquica Barrera

22 de marzo de 2024

Introducción

El objetivo de este laboratorio "Blueprints" radica en su capacidad para simplificar el proceso de la gestión de planos arquitectónicos, permitiendo a los usuarios crear nuevos planos de manera intuitiva y eficaz. Con un enfoque centrado en el usuario, se desarrollaron funcionalidades clave que incluyen la creación de nuevos planos con atributos como autor, nombre y puntos definitorios en la parte del frontend. Además, se implementaron características que facilitan la actualización y eliminación de planos, brindando un control total sobre el proceso de diseño y gestión.

Para potenciar la funcionalidad de la aplicación, estableceremos conexión con un backend que proporciona una API REST. Esta API nos permitirá almacenar, obtener, actualizar y eliminar los planos arquitectónicos de manera segura y eficiente. Exploraremos cómo configurar Angular para interactuar con esta API, gestionar las solicitudes HTTP y presentar los datos obtenidos de manera coherente y atractiva en nuestra interfaz de usuario.

Al integrar Java con Spring Boot en el lado del servidor y Angular en el cliente, "Blueprints" aseguraremos una sincronización eficiente entre el frontend y el backend, ofreciendo una experiencia de usuario ágil y receptiva. Esta arquitectura distribuida permite un desempeño óptimo y una escalabilidad sin contratiempos, fundamentales para manejar cargas de trabajo variables y soportar un crecimiento continuo.

Desarrollo del Laboratorio

1. Lo primero que haremos es implementar la funcionalidad para actualizar un blueprint para esto agregaremos en get-blueprints component una variable nueva para almacenar los puntos que el usuario quiere actualizar del plano.

```
export class GetBlueprintsComponent{  
  blueprints:Blueprint[];  
  flagDiv:Boolean = false;  
  author:string;  
  amountOfPoints:number;  
  name:string;  
  blueprintName:string;  
  blueprint:Blueprint;  
  points: { x: number, y: number }[];
```

- Para lograr la implementación de esta funcionalidad crearemos un método el cual se ejecutará al momento de que el usuario haga click en el botón de Save/Update, el cual será para obtener los puntos.

```
//Method to get the points  
askForPoints() {  
  const points = window.prompt("Please enter the points in this way (x,y),(x,y):");  
  if (points !== null && this.blueprintName!="") {  
    const parsedPoints = this.parsePoints(points);  
    if(parsedPoints){  
      this.points = parsedPoints;  
      this.updateBlueprint();  
    }  
  }  
}
```

Para lograr que el usuario tenga una mejor experiencia al usar la aplicación se implementó un `window.prompt` el cual permite una interacción directa con el usuario mediante la presentación de mensajes claros. Se hizo la solicitud de los nuevos puntos del plano indicándole la forma correcta de escribirlos ya que a su vez estos se filtrarán por un método el cual los ajustará al formato correcto del Blueprint y después se hará la petición para actualizar este.

- El método que se implementó por el cual pasaran los puntos para ajustarlos al formato correcto del Blueprint quita los paréntesis y los separa por las comas permitiendo que el formato sea de tipo de la variable declarada en el componente inicialmente.

```
// Method to analyze and validate the points entered in the format ((x),y))
private parsePoints(pointsInput: string): { x: number, y: number }[] | null {
  try {
    // Remove the parentheses and divide the string by commas
    const pointsArray = pointsInput.replace(/(\(|\)|,)/g, '').split(',');
    // Validate if there are an even number of values
    if (pointsArray.length % 2 !== 0) {
      return null;
    }
    const parsedPoints: { x: number, y: number }[] = [];

    for (let i = 0; i < pointsArray.length; i += 2) {
      const x = parseFloat(pointsArray[i]);
      const y = parseFloat(pointsArray[i + 1]);

      // Verify the values are numbers
      if (!isNaN(x) && !isNaN(y)) {
        parsedPoints.push({ x, y });
      } else {
        return null;
      }
    }
    return parsedPoints;
  } catch (error) {
    return null;
  }
}
```

- Posteriormente se llama al método de updateBlueprint el cual actualizara el Blueprint con los puntos nuevos dados por el usuario haciendo la petición al servidor.

```
//Method to update the blueprint
updateBlueprint(){
  try{
    this.blueprintServices.updateBlueprint(this.author,this.blueprintName,this.points).subscribe(data => {
      this.clearCanvas();
      this.getBlueprintsByAuthor();
      this.getBlueprintByNameandAuthor();
    });
  }catch (error) {
    console.log('Error in updateBlueprint:', error);
  }
}
```

Para lograr la correcta actualización del Blueprint y de la experiencia del usuario, Se creo el servicio que realiza la petición al servidor para actualizar este, se creó un método que limpia el canvas y posteriormente se ejecutaron los métodos realizados en el laboratorio anterior para hacer el get de los planos del usuario y para visualizar el cambio en el canvas.

- Creación del servicio:

```
//Method to update the blueprint
updateBlueprint(author:string,nameBlueprint:string,points: { x: number, y: number }[]):Observable<Object>{
  return this.httpClient.put(`${this.URL}/${author}/${nameBlueprint}`,points);
}
```

- Método que limpia el canvas:

```
//Clean canvas
private clearCanvas() {
  const canvas = document.getElementById('Canvas') as HTMLCanvasElement;
  if (canvas) {
    const context = canvas.getContext("2d");
    if (context) {
      context.clearRect(0, 0, canvas.width, canvas.height);
    } else {
      console.error('Could not get 2D context from canvas');
    }
  } else {
    console.error('The canvas element could not be found');
  }
}
```

- HTML y CSS:

```
<div *ngIf="flagDiv">
  <button id="save-update" (click)="askForPoints()">Save/Update</button>
```

```
#save-update{
  position: absolute;
  left: 800px;
  top: 750px;
  width: 120px;
  background-color: green;
}

#save-update:hover{
  background-color: lightgreen;
}
```

- Funcionalidad:

Blueprints

Author:

andres's blueprints: Current blueprint: andres_p1

Name	Amount of Points
andres_p1	2
andres_p2	2

Total user points : 4

Please enter the points in this way (x,y),(x,y):

En esta parte se hace la petición de los planos de andres se abre el plano con el nombre de andres_p1 y se le da click al botón de Save/Update para actualizar los puntos, en la tabla nos damos cuenta que el plano tiene 2 puntos, y la suma total de puntos es 4.

Blueprints

Author: Get Blueprints

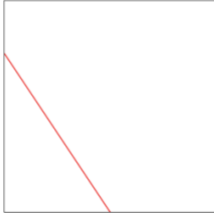
Create new blueprint

andres's blueprints:

Name	Amount of Points	Open
andres_p1	3	<button>Open</button>
andres_p2	2	<button>Open</button>

Total user points : 5

Current blueprint: andres_p1



Save/Update Delete

Al darle ok, podemos observar que la cantidad de puntos del plano cambio a 3 y el total de puntos a 5, a su vez la gráfica en el canvas cambio.

- Ahora realizaremos la implementación para eliminar un Blueprint, para esta implementamos en get-blueprints component el método que se ejecutara cuando el usuario le de eliminar a algún plano.

```
//Method to delete a Blueprint
deleteBlueprint(){
  try{
    this.blueprintServices.deleteBlueprint(this.author,this.blueprintName).subscribe(data => {
      this.clearCanvas();
      this.blueprintName = '';
      this.onSubmit();
    });
  }catch (error) {
    console.log('Error in createNewBlueprint:', error);
  }
}
```

Para lograr la correcta implementación de la funcionalidad implementamos el servicio que realiza la petición al servidor para eliminar el blueprint tanto en el frontend como en el Backend, posteriormente limpiaremos el canva y el nombre del Blueprint actual, después de esto ejecutaremos el método onSubmit el cual ejecutar el método de getBlueprintsByAuthor lo que permitirá revisar si el autor tiene más planos para

actualizar la vista.

- Creación del servicio en el Frontend:

```
//Method to delete the blueprint
deleteBlueprint(author:string,nameBlueprint:string):Observable<Object>{
  return this.httpClient.delete(`${this.URL}/${author}/${nameBlueprint}`);
}
```

- Creación del servicio en el Backend

- Controlador:

```
± Andres Felipe
@RequestMapping(value = "${Author}/{bpname}",method = RequestMethod.DELETE)
public ResponseEntity<?> deleteBlueprint(@PathVariable String Author, @PathVariable String bpname) {
    try {
        blueprintsServices.deleteBlueprint(Author,bpname);
        return new ResponseEntity<>(HttpStatus.OK);
    } catch (Exception ex) {
        Logger.getLogger(BlueprintAPIController.class.getName()).log(Level.SEVERE, msg: null, ex);
        return new ResponseEntity<>( body: "Error" + ex.getMessage(), HttpStatus.FORBIDDEN);
    }
}
```

- Servicio:

```
2 usages ± Andres Felipe
public void deleteBlueprint(String author, String name) throws BlueprintNotFoundException {
    bpp.deleteBlueprint(author,name);
}
```

- Persistencia:

```
2 usages ± Andres Felipe
@Override
public void deleteBlueprint(String author, String blueprintName) throws BlueprintNotFoundException {
    Tuple<String, String> tuple = new Tuple<>(author,blueprintName);
    blueprints.remove(tuple);
}
```

- HTML Y CSS:

```
<button id="delete" (click)="deleteBlueprint()">Delete</button>
```

```
#delete{
  position: absolute;
  left: 1050px;
  top: 750px;
  background-color: red;
}

#delete:hover{
  background-color: lightcoral;
}
```

- Funcionalidad:

Siguiendo el ejemplo anterior para actualizar el plano procederemos a eliminar este y tendría que verse reflejado que solo le queda un plano al autor, al igual que ya no hay ningún plano seleccionado ni pintado.

Blueprints

Author: Get Blueprints

Create new
blueprint

andres's blueprints:

Name	Amount of Points	Open
andres_p2	2	<button>Open</button>

Current blueprint:



Total user points : 2

Save/Update

Delete

3. Por último realizaremos la implementación para Crear un nuevo Blueprint, para esto crearemos un nuevo componente con el comando `ng g c create-blueprint`, en este tendremos como variable nueva un blueprint y el input de los puntos del plano en string ya que al igual para actualizar los puntos de un plano se pasarán por el mismo filtro.

```
export class CreateBlueprintComponent {  
  
  blueprint: Blueprint = new Blueprint();  
  pointsInput: string;  
  
  constructor(private blueprintServices:BlueprintService,private router:Router){  
  
  }  
}
```

En el constructor de este componente usaremos el servicio de blueprints para poder hacer la petición al servidor y un router para poder navegar dinámicamente en la

aplicación web.

- Posteriormente realizaremos los métodos necesarios para lograr una correcta implementación de la funcionalidad, en el archivo `create-blueprint.component.ts`. Crearemos un método para que al momento de enviar el formulario obtengamos correctamente los datos del Blueprint y pasemos por el filtro los puntos del plano para ajustarlos al correcto formato.

```
//Method to send the petition for getting blueprints
onSubmit(){
  //Convert the points
  const parsedPoints = this.parsePoints(this.pointsInput);
  if (parsedPoints) {
    this.blueprint.points = parsedPoints;
    this.blueprint.amountOfPoints = parsedPoints.length;
    this.createNewBlueprint();
    this.router.navigate(['/blueprints']);
  } else {
    console.log("Error: points are invalid.");
  }
}
```

En este método configuramos los atributos del plano según lo que el usuario Digito, pasando por el filtro los puntos para configurarle estos al Blueprint, Estableciendo la cantidad de puntos que este tiene, llamando al método que Ejecuta el servicio que realiza la petición al servidor y posteriormente navegar A la parte principal de la aplicación que es donde se puede evidenciar la correcta creación de este por medio del router.

- Llamada al servicio:

```
private createNewBlueprint() {
  try {
    this.blueprintServices.createNewBlueprint(this.blueprint).subscribe(data => {
      console.log(data);
    });
  } catch (error) {
    console.log('Error in createNewBlueprint:', error);
  }
}
```

- Creación del servicio en blueprint.service.ts:

```
//Service to create a new blueprint
createNewBlueprint(blueprint:Blueprint):Observable<Object>{
  return this.httpClient.post(`${this.URL}/agregar`,blueprint);
}
```

- HTML y CSS:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Blueprints</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <h2>
    Create New Blueprint
  </h2>
  <form (ngSubmit)="onSubmit()">
    <div>
      <label for="author">Authors:</label>
      <input type="text" id="author" name="author" [(ngModel)]="blueprint.author" required>
    </div>
    <div>
      <label for="blueprintName">Blueprint Name:</label>
      <input type="text" id="blueprintName" name="blueprintName" [(ngModel)]="blueprint.name" required>
    </div>
    <div>
      <label for="points">Points:</label>
      <textarea id="points" name="points" [(ngModel)]="pointsInput" required placeholder="Ingresa los pun
    </div>
    <button type="submit">Create</button>
  </form>
</body>
</html>
```

```
body, html {
  height: 100%;
  margin: 0;
  display: flex;
  justify-content: center;
  align-items: center;
}
form {
  margin-top: 150px;
  width: 800px;
  padding: 20px;
  border: 1px solid black;
  border-radius: 5px;
  background-color: #f2f2f2;
}
label {
  display: block;
  margin-bottom: 10px;
  font-style: italic;
  font-size: 25px;
}
input[type="text"],
textarea {
  width: 100%;
  padding: 8px;
  margin-bottom: 10px;
  border: 1px solid black;
  border-radius: 4px;
  box-sizing: border-box;
}
button[type="submit"] {
  background-color: #4CAF50;
  color: white;
  padding: 10px 20px;
}
```

- Funcionalidad:

Para revisar la funcionalidad e este caso daremos click al botón de Create new

Blueprint y aparecerá:

Blueprints

Create New Blueprint

Author:

Blueprint Name:

Points:

Ingrese los puntos en formato (x,y),(x,y), separados por comas. Por ejemplo: (1,2),(3,4),(5,6)

Create

Crearemos un plano con los siguiente atributos : AUTHOR: ivan , BLUEPRINT

NAME: ivan_p1 y POINTS: (0,100),(100,200),(400,500),(500,600),(600,0) y damos

a créate lo cual nos redireccionara a la página principal en donde podemos comprobar

la creación de este:

Blueprints

Author:

Get Blueprints

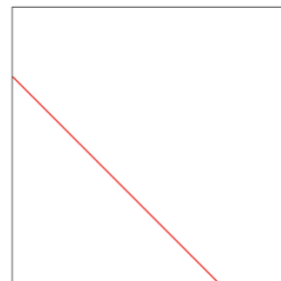
Create new
blueprint

ivan's blueprints:

Name	Amount of Points	Open
ivan_p1	5	Open

Total user points : 5

Current blueprint: ivan_p1



Save/Update

Delete

Conclusiones

1. **Integración eficaz de tecnologías modernas:** La combinación de Java con Spring Boot en el backend y Angular en el frontend demuestra una integración efectiva de tecnologías modernas para desarrollar una aplicación robusta y escalable. Esta elección de tecnologías sugiere una cuidadosa consideración de la eficiencia y la capacidad de expansión del sistema.
2. **Enfoque centrado en el usuario:** La implementación de funcionalidades específicas, como la creación, actualización y eliminación de planos, indica un enfoque centrado en el usuario para garantizar una experiencia de usuario fluida y satisfactoria.

Bibliografia

- <http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/>
- https://www.youtube.com/watch?v=o_HV_FCs-Z0&t=1952s