

## **Actividad 2 - Diseño Web y Conexión**

### **Desarrollo de Sistemas Web II**

### **Ingeniería en Desarrollo de Software**

**Tutor: Aarón Iván Salazar Macías**

**Alumno: Jusi Ismael Linares Gutiérrez**

**Fecha: 17/02/2024**

## Índice

Introducción .....	3
Descripción.....	3
Justificación .....	3
Desarrollo: .....	4
Diseño del proyecto .....	4
Codificación .....	8
Conexión a la BD .....	24
Conclusión.....	26
GitHub .....	26

## Introducción

En esta actividad, se aborda el desarrollo y diseño de un servicio web utilizando el lenguaje de programación Java junto con el framework Spring Boot. La solicitud proviene de la tienda Sara, que busca una interfaz de tienda en línea fácil de entender y agradable a la vista para sus usuarios. Se espera la creación de distintas interfaces, como la página de galería de productos, descripción del producto, y la página de pago del carrito.

El enfoque principal recae en la implementación de una API REST que retorne un formato JSON con los productos almacenados en la base de datos creada en la actividad anterior. Esta conexión con la base de datos es esencial para garantizar que la información de productos esté siempre actualizada y disponible para los usuarios. El entorno de desarrollo seleccionado es Visual Studio Code, un entorno ampliamente utilizado y versátil para proyectos de desarrollo. Este ejercicio no solo permite aplicar conocimientos teóricos sobre bases de datos y desarrollo web, sino también la práctica directa en la creación de servicios web funcionales y eficientes.

## Descripción

La tienda Sara ha avanzado en su proceso de transformación digital al solicitar el diseño y desarrollo de un sitio web para su tienda en línea. La elección de utilizar Java y Spring Boot para construir una API REST demuestra la intención de utilizar tecnologías modernas y eficientes en el desarrollo del servicio web. La solicitud de interfaces específicas, como la página de galería de productos, la descripción del producto seleccionado y la página de pago del carrito, resalta la importancia de proporcionar a los usuarios una experiencia intuitiva y completa durante su interacción en línea.

La estructuración de la galería de productos según categorías, la presentación de información clave como nombre, imagen y precio, así como la inclusión de un filtro de búsqueda, demuestran la preocupación por la usabilidad y accesibilidad del sitio. La conexión con la base de datos creada en la actividad anterior es esencial para garantizar que la información del catálogo de productos esté siempre actualizada y disponible para los usuarios.

## Justificación

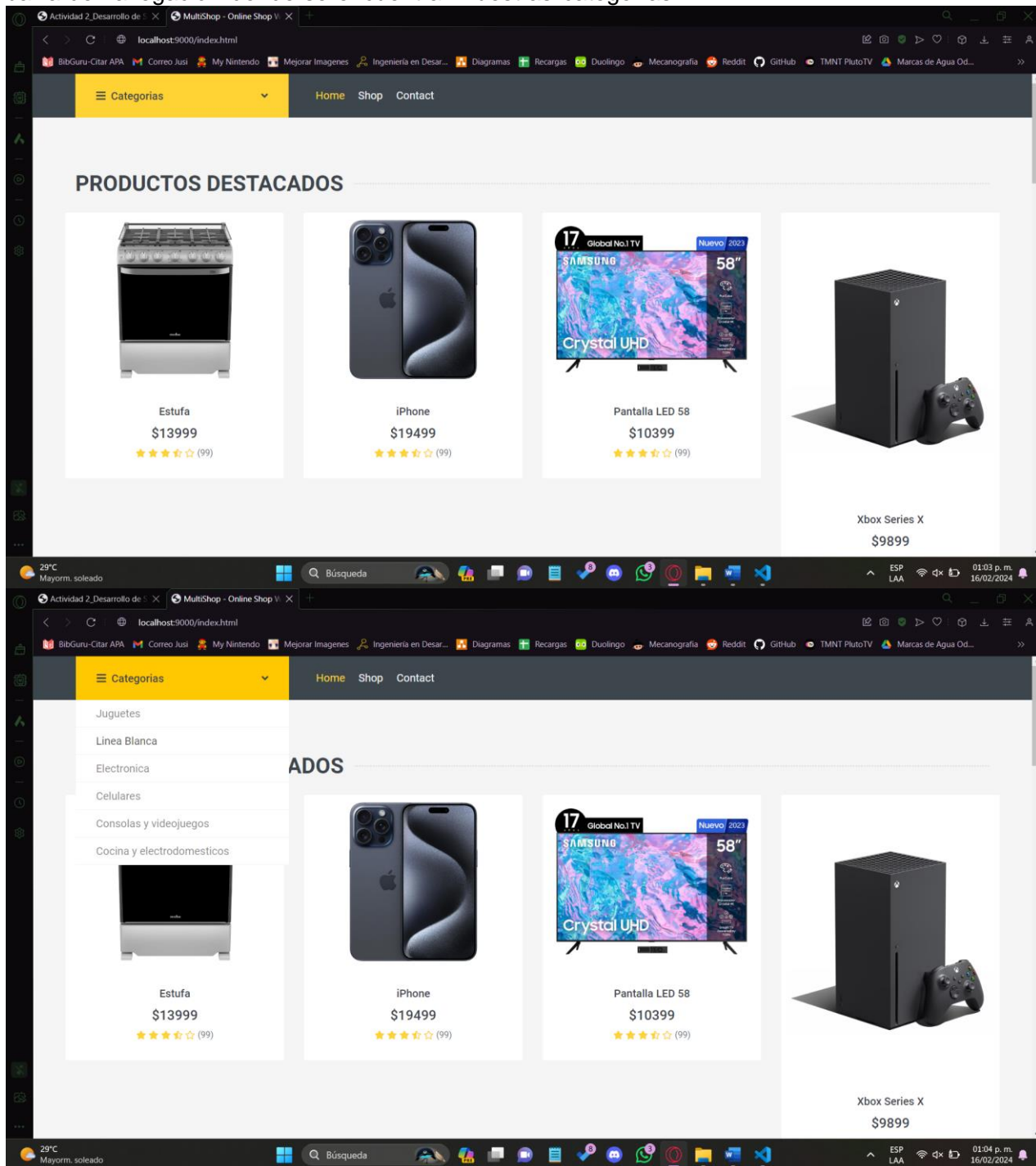
El uso de Java y Spring Boot para desarrollar un servicio web con una API REST para el sistema de la tienda Sara es una elección acertada por varias razones. En primer lugar, Java es un lenguaje de programación confiable y robusto, ampliamente utilizado en el desarrollo de aplicaciones empresariales y sistemas web. Spring Boot, por su parte, simplifica el desarrollo de aplicaciones basadas en Java al proporcionar un marco de trabajo que facilita la configuración y el desarrollo rápido.

La elección de una API REST asegura que el sistema pueda interactuar de manera eficiente con el front-end y otros servicios web, proporcionando una comunicación flexible y escalable entre los componentes del sistema. El formato JSON para la transmisión de datos es ligero y fácil de entender, lo que mejora la eficiencia de la transmisión de información entre el servidor y el cliente. Además, Visual Studio Code es un entorno de desarrollo popular y versátil que facilita la codificación y ofrece herramientas útiles para depurar y administrar proyectos.

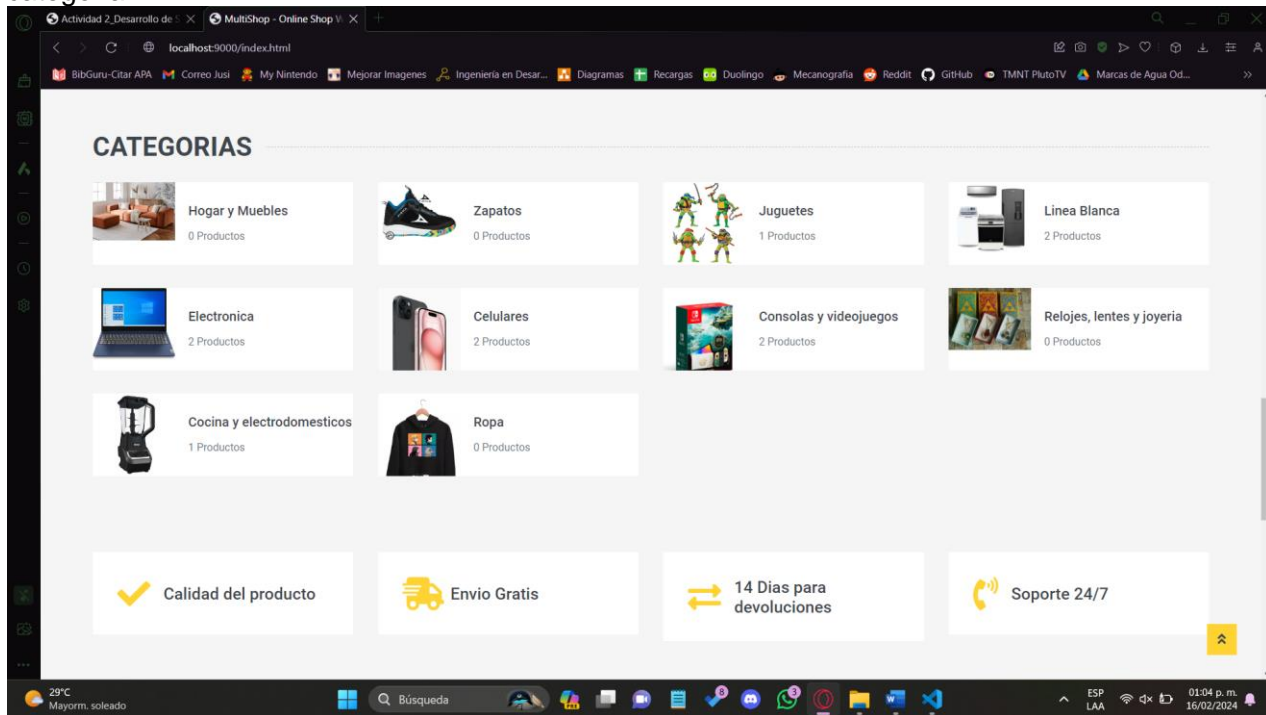
Desarrollo:

## Diseño del proyecto

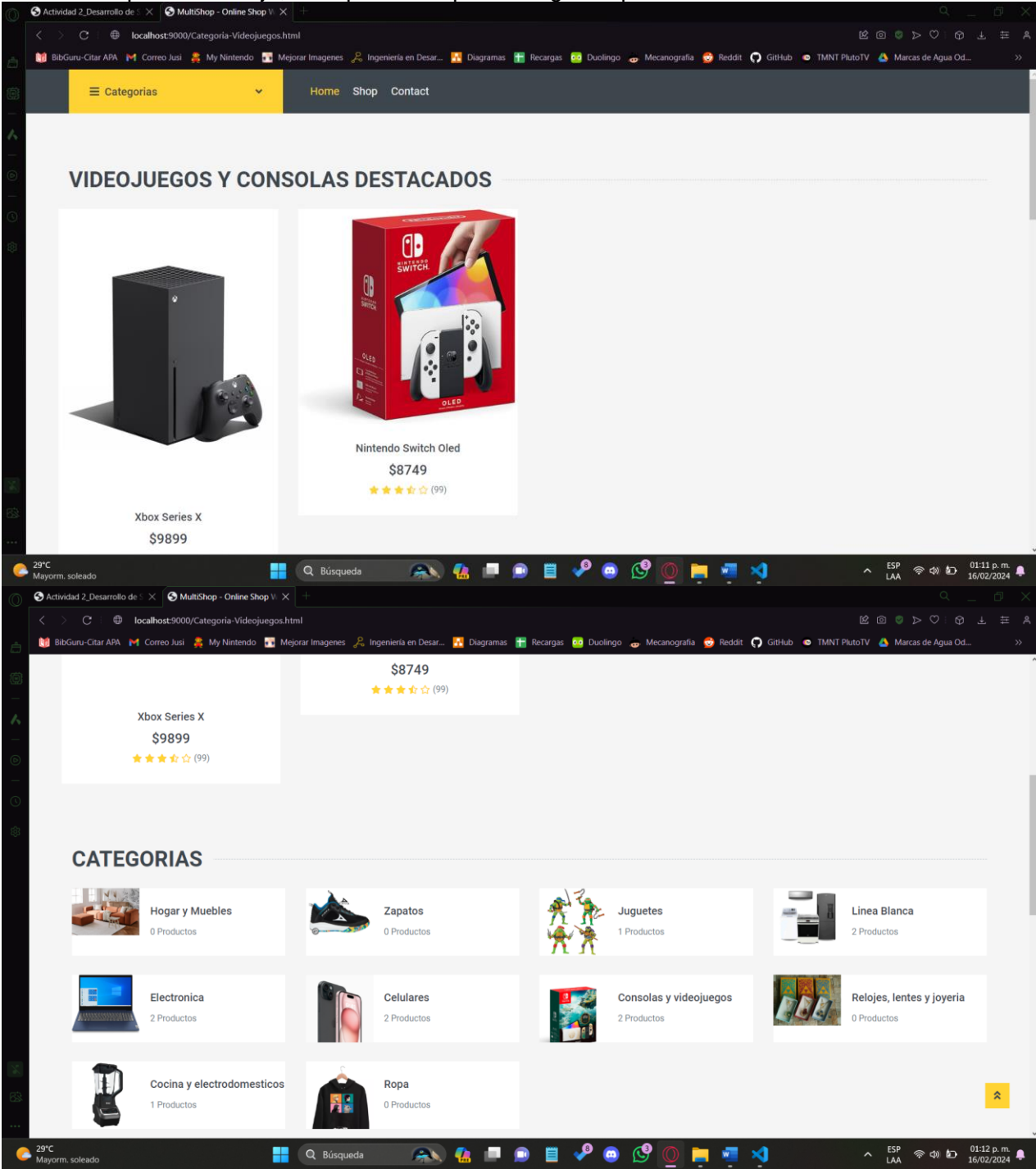
En el index lo primero que tenemos es algunos de nuestros productos, al igual que una pequeña barra de navegación donde se encuentran nuestras categorías



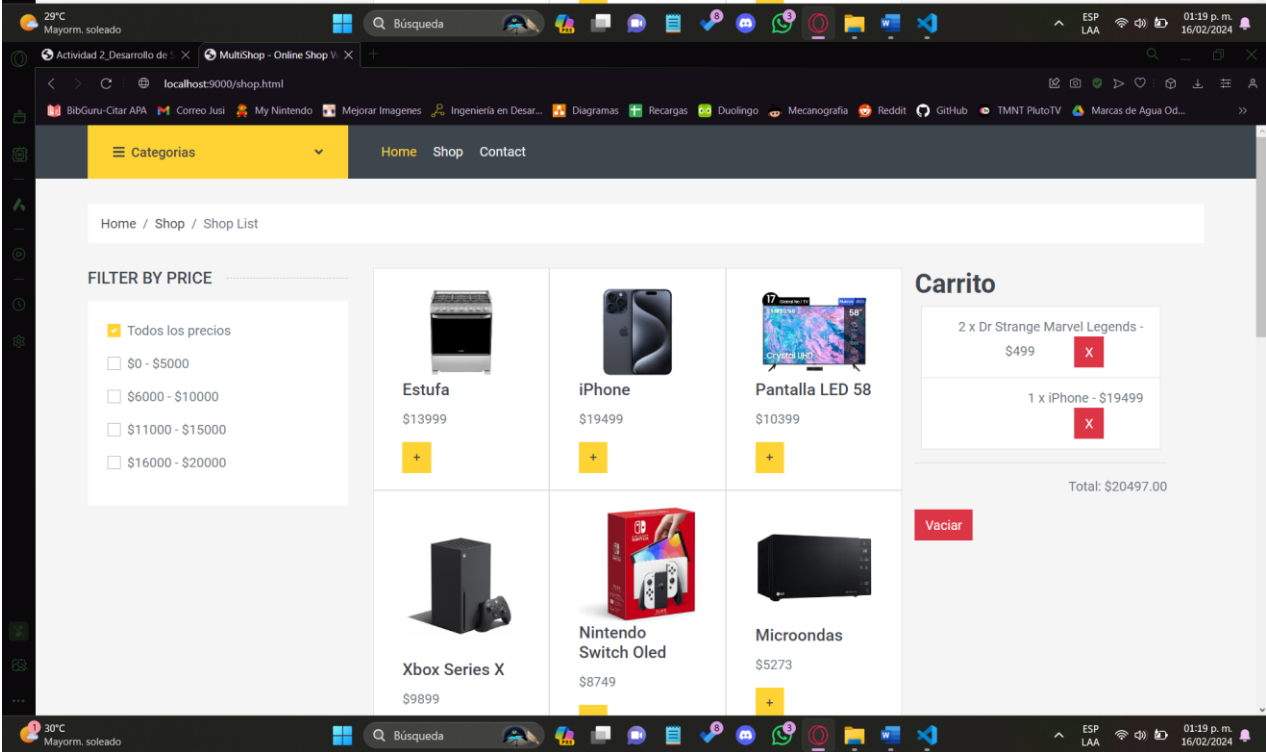
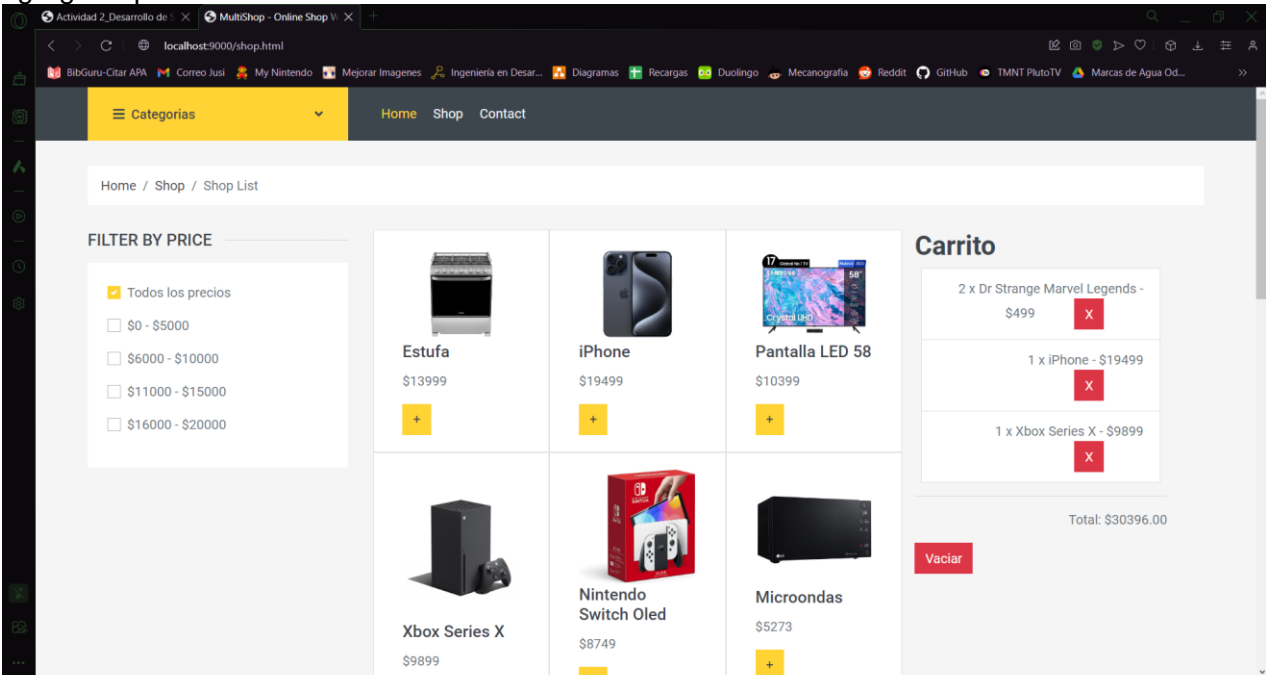
Si bajamos un poco mas igualmente tenemos un menú de categorías donde además de la categoría se nos indica la cantidad de productos pertenecientes a cada categoría, ya sea dando clic en este menú o en el de la barra de navegación se nos redirigirá hacia la pagina de dicha categoría



A partir de aquí también podemos movernos a otras categorías desde la barra de navegación o el menú de la parte de abajo, esto para cualquier categoría que seleccionemos



En cuanto al apartado de todos nuestros productos sin importar la categoría podemos ir realizando nuestras compras al anexarlos al carrito, igualmente el carrito ira realizando la suma de nuestros artículos seleccionados, dando clic en la x podemos eliminar cualquier articulo que hayamos agregado por error



## Codificación

En cuanto a la codificación el primer archivo que generamos es el de la conexión con la BD, pero por el momento lo omitiremos puesto que lo veremos más adelante de forma más detallada.

Tras crear nuestro archivo de conexión creamos la interfaz `IProductoRepository` donde se guardaran todos los métodos que usaremos a lo largo de la creación de nuestro programa.

`List<Producto> findAll()`: Este método devuelve una lista de todos los productos en la base de datos.

`int save(Producto producto)`: Este método guarda un nuevo producto en la base de datos.

`int update(Producto producto)`: Este método actualiza la información de un producto existente en la base de datos.

`int deleteById(int ID)`: Este método elimina un producto de la base de datos utilizando su ID

`String findDescriptionById(int id)`: Este método encuentra y devuelve la descripción(nombre del artículo) de un producto utilizando el ID.

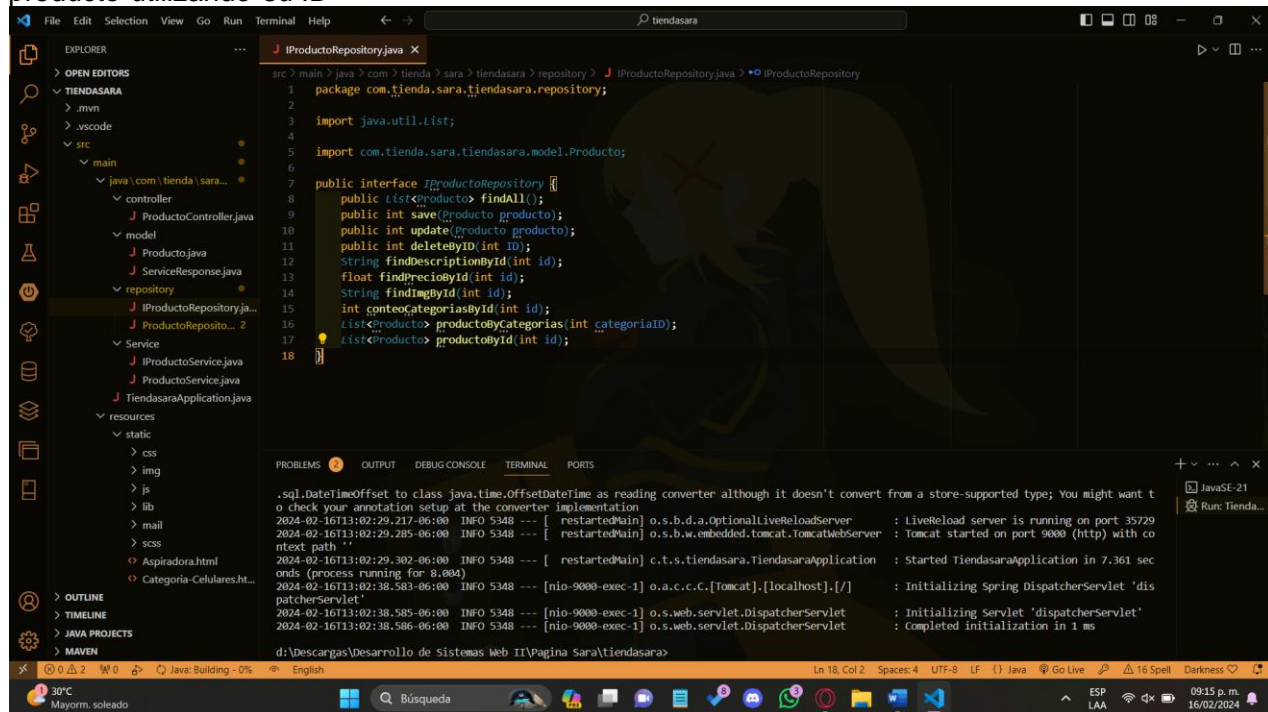
`float findPrecioById(int id)`: Este método encuentra y devuelve el precio de un producto utilizando el ID.

`String findImgById(int id)`: Este método encuentra y devuelve la URL de la imagen de un producto utilizando el ID.

`int conteoCategoriasById(int id)`: Este método encuentra y devuelve la cantidad de categorías a las que pertenece un producto utilizando el ID.

`List<Producto> productoByCategorias(int categoriaID)`: Este método devuelve una lista de productos que pertenecen a una categoría en base a su `idCategoria`

`List<Producto> productoById(int id)`: Este método encuentra y devuelve todos los datos de un producto utilizando su ID



```
src > main > java > com > tienda > sara > tiendasara > repository > IProductoRepository.java > IProductoRepository
1 package com.tienda.sara.tiendasara.repository;
2
3 import java.util.List;
4
5 import com.tienda.sara.tiendasara.model.Producto;
6
7 public interface IProductoRepository {
8     public List<Producto> findAll();
9     public int save(Producto producto);
10    public int update(Producto producto);
11    public int deleteById(int id);
12    String findDescriptionById(int id);
13    float findPrecioById(int id);
14    String findImgById(int id);
15    int conteoCategoriasById(int id);
16    List<Producto> productoByCategorias(int categoriaID);
17    List<Producto> productoById(int id);
18 }
```

```
.sql.DateOffset to class java.time.OffsetDateTime as reading converter although it doesn't convert from a store-supported type; You might want to
check your annotation setup at the converter implementation
2024-02-16T13:02:29.217-06:00 INFO 5348 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-02-16T13:02:29.285-06:00 INFO 5348 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 9000 (http) with co
ntext path ''
2024-02-16T13:02:29.302-06:00 INFO 5348 --- [ restartedMain] c.t.s.tiendasara.TiendasaraApplication : Started TiendasaraApplication in 7.361 sec
onds (process running for 8.004)
2024-02-16T13:02:38.583-06:00 INFO 5348 --- [nio-9000-exec-1] o.a.c.c.c.[Tomcat].[/] : Initializing Spring DispatcherServlet 'dis
patcherServlet'
2024-02-16T13:02:38.585-06:00 INFO 5348 --- [nio-9000-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-02-16T13:02:38.586-06:00 INFO 5348 --- [nio-9000-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
d:\Descargas\Desarrollo de Sistemas Web II\Pagina Sara\tiendasara>
```



Posteriormente creamos la clase java Producto donde solamente definiremos el tipo de dato de cada columna en nuestra tabla Producto, es importante resaltar que los tipos de datos entre este archivo y la base de datos deben ser iguales para evitar errores

```
Producto.java
src > main > java > com > tienda > sara > tiendasara > model > Producto.java > Producto
1 package com.tienda.sara.tiendasara.model;
2
3 import lombok.Data;
4
5 @Data
6 public class Producto {
7     int ID;
8     String Descripcion;
9     float Precio;
10    int Cantidad;
11    int idCategorias;
12    int idMarcas;
13    String img;
14 }
```

Posteriormente crearemos nuestra clase ProductoRepository donde escribiremos las consultas SQL que se ejecutaran en nuestra base de datos junto con los datos que requieran para ejecutarla correctamente (como el ID)

```
ProductoRepository.java
src > main > java > com > tienda > sara > tiendasara > repository > ProductoRepository.java > Language Support for Java(TM) by Red Hat > ProductoRepository > deleteById(int)
1 package com.tienda.sara.tiendasara.repository;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.jdbc.core.BeanPropertyRowMapper;
7 import org.springframework.jdbc.core.JdbcTemplate;
8 import org.springframework.stereotype.Repository;
9
10 import com.tienda.sara.tiendasara.model.Producto;
11
12 @Repository
13 public class ProductoRepository implements IProductoRepository {
14
15     @Autowired
16     private JdbcTemplate jdbcTemplate;
17
18     @Override
19     public int deleteById(int ID) {
20         String SQL = "DELETE * FROM Productos WHERE ID = ?";
21         return jdbcTemplate.update(SQL, new Object[] { ID });
22     }
23
24     @Override
25     public List<Producto> findAll() {
26         String SQL = "SELECT * FROM Productos";
27         return jdbcTemplate.query(SQL, BeanPropertyRowMapper.newInstance(Producto.class));
28     }
29
30     @Override
31     public int save(Producto producto) {
32         String SQL = "INSERT INTO Productos VALUES(?,?,?,?,?,?)";
33         return jdbcTemplate.update(SQL,
34             new Object[] { producto.getID(), producto.getDescripcion(), producto.getPrecio(),
35                 producto.getCantidad(), producto.getIdCategorias(), producto.getIdMarcas() });
36     }
37 }
```

```

38     @Override
39     public int update(Producto producto) {
40         String SQL = "UPDATE Productos SET ID=?, Descripcion=?, Precio=?, Cantidad=?, idCategorias=?, idMarcas=?, img=? WHERE ID=?";
41         return jdbcTemplate.update(SQL,
42             new Object[] { producto.getID(), producto.getDescripcion(), producto.getPrecio(),
43                 producto.getCantidad(), producto.getIdCategorias(), producto.getIdMarcas() });
44     }
45
46     @Override
47     public String findDescriptionById(int id) {
48         String SQL = "SELECT Descripcion FROM Productos WHERE ID=?";
49         return jdbcTemplate.queryForObject(SQL, String.class, id);
50     }
51
52     @Override
53     public float findPrecioById(int id) {
54         String SQL = "SELECT precio FROM Productos WHERE ID=?";
55         return jdbcTemplate.queryForObject(SQL, float.class, id);
56     }
57
58     @Override
59     public String findImgById(int id) {
60         String SQL = "SELECT img FROM Productos WHERE ID=?";
61         return jdbcTemplate.queryForObject(SQL, String.class, id);
62     }
63
64     @Override
65     public int conteoCategoriasById(int id) {
66         String SQL = "SELECT COUNT(idCategorias) FROM productos WHERE idCategorias=?";
67         return jdbcTemplate.queryForObject(SQL, int.class, id);
68     }
69
70     @Override
71     public List<Producto> productoByCategorias(int categoriaID) {
72         String SQL = "SELECT * FROM Productos WHERE idCategorias=?";
73         return jdbcTemplate.query(SQL, new Object[] { categoriaID }, BeanPropertyRowMapper.newInstance(Producto.class));
74     }
75
76     @Override
77     public List<Producto> productoByCategorias(int categoriaID) {
78         String SQL = "SELECT * FROM Productos WHERE idCategorias=?";
79         return jdbcTemplate.query(SQL, new Object[] { categoriaID }, BeanPropertyRowMapper.newInstance(Producto.class));
80     }
81
82     @Override
83     public List<Producto> productoById(int id) {
84         String SQL = "SELECT * FROM Productos WHERE ID=?";
85         return jdbcTemplate.query(SQL, new Object[] { id }, BeanPropertyRowMapper.newInstance(Producto.class));
86     }
87 }

```

La interfaz `IProductoService` define los servicios que una clase puede proporcionar relacionados con los productos de una tienda.

`findAll()`: Obtiene todos los productos de la tienda.



`save(Producto producto)`: Guarda un nuevo producto en la tienda.


`update(Producto producto)`: Actualiza la información de un producto existente en la tienda.

`deleteById(int ID)`: Elimina un producto de la tienda según su ID.

`findByCategoria(int categoriaID)`: Busca todos los productos que pertenecen a una categoría específica.

`findById(int categoriaID)`: Busca un producto específico por su ID.

 `IProductoService.java` 

```
src > main > java > com > tienda > sara > tiendasara > Service >  IProductoService.java
1  package com.tienda.sara.tiendasara.Service;
2
3  import java.util.List;
4
5  import com.tienda.sara.tiendasara.model.Producto;
6
7  public interface IProductoService {
8      public List<Producto> findAll();
9      public int save(Producto producto);
10     public int update(Producto producto);
11     public int deleteById(int ID);
12     public List<Producto> findByCategoria(int categoriaID);
13     public List<Producto> findById(int categoriaID);
14 }
```

Para ProductService proporcionamos los métodos para las operaciones CRUD que realizaremos en la siguiente actividad.

@Service: Esta anotación de Spring indica que la clase es un servicio y debe ser manejada por el contenedor de Spring.

@Autowired: Esta anotación de Spring se utiliza para realizar la inyección de dependencias automáticamente en los campos de la clase.

Manejo de excepciones: Cada método envuelve las llamadas al repositorio (iProductoRepository) en un bloque try-catch para manejar cualquier excepción que pueda ocurrir durante la ejecución.

```
J ProductService.java X
src > main > java > com > tienda > sara > tiendasara > Service > J ProductService.java > Language Support for J
12 public class ProductService implements IProductoService {
39     @Override
40     public int save(Producto producto) {
41         int row;
42         try {
43             row = iProductoRepository.save(producto);
44         } catch (Exception e) {
45             throw e;
46         }
47         return row;
48     }
49
50     @Override
51     public int update(Producto producto) {
52         int row;
53         try {
54             row = iProductoRepository.update(producto);
55         } catch (Exception e) {
56             throw e;
57         }
58         return row;
59     }
60
61     @Override
62     public List<Producto> findByCategoria(int categoriaID) {
63         List<Producto> list;
64         try {
65             list = iProductoRepository.productoByCategorias(categoriaID);
66         } catch (Exception e) {
67             throw e;
68         }
69         return list;
70     }
71
72     @Override
73     public List<Producto> findById(int id) {
74         List<Producto> list;
```

J ProductoService.java X

src > main > java > com > tienda > sara > tiendasara > Service > J ProductoService.java > Language Support for Java(TM) I

```
1 package com.tienda.sara.tiendasara.Service;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import com.tienda.sara.tiendasara.model.Producto;
9 import com.tienda.sara.tiendasara.repository.IProductoRepository;
10
```

```
11 @Service
12 public class ProductoService implements IProductoService {
13
```

```
14     @Autowired
15     private IProductoRepository iProductoRepository;
16
```

```
17     @Override
18     public int deleteById(int ID) {
19         int row;
20         try {
21             row = iProductoRepository.deleteById(ID);
22         } catch (Exception e) {
23             throw e;
24         }
25         return row;
26     }
27
```

```
28     @Override
29     public List<Producto> findAll() {
30         List<Producto> list;
31         try {
32             list = iProductoRepository.findAll();
33         } catch (Exception e) {
34             throw e;
35         }
36         return list;
37     }
38
```

```
72     @Override
73     public List<Producto> findById(int id) {
74         List<Producto> list;
75         try {
76             list = iProductoRepository.productoById(id);
77         } catch (Exception e) {
78             throw e;
79         }
80         return list;
81     }
82 }
```

Para nuestro ProductoController, definiremos las rutas de nuestros métodos (donde las {} indican el dato necesario para obtener un resultado) para que más adelante, cuando sean llamadas, puedan ser accedidas de manera adecuada por nuestra aplicación. Por ejemplo, hemos definido un método que manejará las solicitudes GET enviadas a la ruta /description/{id}. Donde {id} es un marcador de posición para el ID del producto del que se desea obtener la descripción. Cuando esta ruta sea accedida con un ID específico, el método getDescriptionById será ejecutado y mostrara el resultado de nuestra consulta definida en ProductoRepository.

```
J ProductoController.java X
src > main > java > com > tienda > sara > tiendasara > controller > J ProductoController.java > Language Support for Java(TM) by Red Hat > ProductoController > getDescriptionById(int)
1 package com.tienda.sara.tiendasara.controller;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.CrossOrigin;
9 import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.PostMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RestController;
15
16 import com.tienda.sara.tiendasara.Service.IProductoService;
17 import com.tienda.sara.tiendasara.model.Producto;
18 import com.tienda.sara.tiendasara.model.ServiceResponse;
19 import com.tienda.sara.tiendasara.repository.IProductoRepository;
20
21 @RestController
22 @RequestMapping("/api/v1/producto")
23 @CrossOrigin("")
24 public class ProductoController {
25     @Autowired
26     private IProductoRepository productoRepository;
27
28     @Autowired
29     private IProductoService iProductoService;
30
31     @GetMapping("/list")
32     public ResponseEntity<List<Producto>> list() {
33         var result = iProductoService.findAll();
34         return new ResponseEntity<>(result, HttpStatus.OK);
35     }
36
37     @PostMapping("/save")
38     public ResponseEntity<ServiceResponse> save(@RequestBody Producto producto) {
39         ServiceResponse serviceResponse = new ServiceResponse();
40         int result = iProductoService.save(producto);
41         if (result == 1) {
42             serviceResponse.setMessage(message:"Item Saved with success");
43         }
44         return new ResponseEntity<>(serviceResponse, HttpStatus.OK);
45     }
46
47     @PostMapping("/update")
48     public ResponseEntity<ServiceResponse> update(@RequestBody Producto producto) {
49         ServiceResponse serviceResponse = new ServiceResponse();
50         int result = iProductoService.update(producto);
51         if (result == 1) {
52             serviceResponse.setMessage(message:"Item update with success");
53         }
54         return new ResponseEntity<>(serviceResponse, HttpStatus.OK);
55     }
56
57     @GetMapping("/delete/{id}")
58     public ResponseEntity<ServiceResponse> delete(@PathVariable int id) {
59         ServiceResponse serviceResponse = new ServiceResponse();
60         int result = iProductoService.deleteByID(id);
61         if (result == 1) {
62             serviceResponse.setMessage(message:"Item removed with success");
63         }
64         return new ResponseEntity<>(serviceResponse, HttpStatus.OK);
65     }
66 }
```

```

67  @GetMapping("/description/{id}")
68  public ResponseEntity<String> getDescriptionById(@PathVariable int id) {
69      String description = ProductoRepository.findDescriptionById(id);
70      if (description != null) {
71          return new ResponseEntity<>(description, HttpStatus.OK);
72      } else {
73          return new ResponseEntity<>("Producto no encontrado", HttpStatus.NOT_FOUND);
74      }
75  }
76
77  @GetMapping("/precio/{id}")
78  public ResponseEntity<Float> getPrecioById(@PathVariable int id) {
79      float precio = ProductoRepository.findPrecioById(id);
80      if (precio > 0) {
81          return new ResponseEntity<>(precio, HttpStatus.OK);
82      } else {
83          return new ResponseEntity<>(HttpStatus.NOT_FOUND);
84      }
85  }
86
87  @GetMapping("/img/{id}")
88  public ResponseEntity<String> getImgById(@PathVariable int id) {
89      String img = ProductoRepository.findImgById(id);
90      if (img != null) {
91          return new ResponseEntity<>(img, HttpStatus.OK);
92      } else {
93          return new ResponseEntity<>("Imagen no encontrada", HttpStatus.NOT_FOUND);
94      }
95  }
96
97  @GetMapping("/countCategorias/{id}")
98  public ResponseEntity<Integer> getCountCategoriasById(@PathVariable int id) {
99      int countCategorias = ProductoRepository.conteoCategoriasById(id);
100     if (countCategorias >= 0) {
101         return new ResponseEntity<>(countCategorias, HttpStatus.OK);
102     } else {
103         return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
104     }
105 }
106
107 @GetMapping("/productoByCategorias/{categoriaID}")
108 public ResponseEntity<List<Producto>> getProductosByCategoria(@PathVariable int categoriaID) {
109     List<Producto> productos = ProductoRepository.productoByCategorias(categoriaID);
110     if (!productos.isEmpty()) {
111         return new ResponseEntity<>(productos, HttpStatus.OK);
112     } else {
113         return new ResponseEntity<>(HttpStatus.NOT_FOUND);
114     }
115 }
116
117 @GetMapping("/productoById/{id}")
118 public ResponseEntity<List<Producto>> getProductosById(@PathVariable int id) {
119     List<Producto> productos = ProductoRepository.productoById(id);
120     if (!productos.isEmpty()) {
121         return new ResponseEntity<>(productos, HttpStatus.OK);
122     } else {
123         return new ResponseEntity<>(HttpStatus.NOT_FOUND);
124     }
125 }
126 }

```

Como extra podemos verificar el correcto funcionamiento de nuestras rutas api con ayuda de la extensión Talend API Tester, primero ingresamos nuestra api y posteriormente damos clic en send, si nuestro código es correcto debería mostrarnos los resultados de nuestras consultas

The image displays two screenshots of the Talend API Tester application, demonstrating a successful API call.

**Top Screenshot:** Shows the 'Requests' tab. The method is set to 'GET' and the URL is 'http://localhost:9000/api/v1/producto/list'. The 'Send' button is highlighted, with a tooltip indicating 'Send request (Alt + Enter)'. Below the request configuration, the 'Response' section shows a status of '200'.

**Bottom Screenshot:** Shows the 'Response' section expanded. The 'HEADERS' tab is selected, displaying the following headers:

- Vary: Origin
- Vary: Access-Control-Request-Method
- Vary: Access-Control-Request-Headers
- Content-Type: application/json
- Transfer-Encoding: chunked
- Date: Sat, 17 Feb 2024 06:03:09 GMT
- Keep-Alive: timeout=60
- Connection: keep-alive

The 'BODY' tab is also selected, showing a JSON array of product data:

```
[{"idCategorias": 4, "idMarcas": 1, "img": "https://cutt.ly/7WuZlgi", "id": 1, "precio": 13999.0, "cantidad": 100, "descripcion": "Estufa"}, {"idCategorias": 6, "idMarcas": 6, "img": "https://cutt.ly/ywWuCrAP", "id": 2, "precio": 19499.0, "cantidad": 85, "descripcion": "iPhone"}, {"idCategorias": 5, "idMarcas": 3, "img": "https://cutt.ly/hwWuCIYI", "id": 3, "precio": 130, "cantidad": 130, "descripcion": ""}]
```



Ya podemos pasar a diseñar nuestra aplicación, en mi caso utilice una plantilla por lo que principalmente me concentrare en explicar los scripts y todo lo relacionado a ello.

Para el index considero 3 partes importantes: El mostrar artículos destacados. El redireccionamiento de los artículos hacia una página con el nombre del articulo + .html y el conteo de artículos por categoría en el menú de categorías.

Empecemos con los artículos.

1. Utilizamos jQuery para esperar a que el documento este listo `$(document).ready(function () { ... }`
2. Realizamos una solicitud GET a la API en la ruta `"/api/v1/producto/list"` para obtener la lista de productos.
3. Para cada producto, generamos un fragmento de HTML que contiene la información del producto, específicamente el url de su imagen, descripción y precio, todo esto obtenido de nuestra base de datos.
4. Agregamos este fragmento de HTML al contenedor con el id `"productContainer"`.
5. Si hay algún error al obtener la lista de productos, se imprime un mensaje de error en la consola.

```
<script>
$(document).ready(function () {
    // Realizar solicitud GET a la API para obtener la lista de productos
    $.get("/api/v1/producto/list", function (data, status) {
        console.log("Response from server:", data); // Imprimir la respuesta del servidor
        console.log("Status of request:", status); // Imprimir el estado de la solicitud
        if (status === "success") {
            data.forEach(function (producto) {
                // Crear el HTML para cada producto
                var productHtml = `
<div class="col-lg-3 col-md-4 col-sm-6 pb-1">
<div class="product-item bg-light mb-4">
<div class="product-img position-relative overflow-hidden">

</div>
<div class="text-center py-4" id="productDetails${producto.id}">
<a class="h6 text-decoration-none text-truncate" href="${producto.descripcion}.html">${producto.descripcion}</a>
<div class="d-flex align-items-center justify-content-center mt-2">
<h5>${producto.precio}</h5>
</div>
<div class="d-flex align-items-center justify-content-center mb-1">
<small class="fa fa-star text-primary mr-1"></small>
<small class="fa fa-star text-primary mr-1"></small>
<small class="fa fa-star text-primary mr-1"></small>
<small class="fa fa-star-half-alt text-primary mr-1"></small>
<small class="fa fa-star text-primary mr-1"></small>
<small>(99)</small>
</div>
</div>
</div>
</div>`;
                // Agregar el HTML del producto al div
                $("#productContainer").append(productHtml);
            });
        } else {
            console.log("Error al obtener la lista de productos.");
        }
    });
});
</script>
```

Continuemos con el redireccionamiento de la descripción del artículo hacia su página de descripción.

```
<a class="h6 text-decoration-none text-truncate"
href="{producto.descripcion}.html">{producto.descripcion}</a>:
```

El fragmento de código anterior crea un enlace que muestra la descripción del producto como texto y enlaza a una página HTML que lleva el nombre de la descripción del producto obtenida en nuestra BD.

```
<div class="text-center py-4" id="productDetails{producto.id}">
  <a class="h6 text-decoration-none text-truncate" href="{producto.descripcion}.html">{producto.descripcion}</a>
  <div class="d-flex align-items-center justify-content-center mt-2">
    <h5>{producto.precio}</h5>
  </div>
  <div class="d-flex align-items-center justify-content-center mb-1">
    <small class="fa fa-star text-primary mr-1"></small>
    <small class="fa fa-star text-primary mr-1"></small>
    <small class="fa fa-star text-primary mr-1"></small>
    <small class="fa fa-star-half-alt text-primary mr-1"></small>
    <small class="fa fa-star text-primary mr-1"></small>
    <small>(99)</small>
  </div>
</div>
```

Por último tenemos el script para el conteo de artículos pertenecientes a una categoría específica en base a su ID, para este ejemplo lo haremos con el ID = 1

`$.get("/api/v1/producto/countCategorias/1", function (data, status) { ... });` Realiza una solicitud GET a la API con la ruta `/api/v1/producto/countCategorias/1` para obtener el conteo de productos pertenecientes a la categoría con ID 1 (utilizando la consulta definida en `ProductoRepository`)

`$("#categoria1 small").text(data + " Productos");` Actualiza el contenido del elemento HTML `<small>` dentro del elemento con el ID "categoria1" con el número de productos obtenido de la respuesta del servidor, seguido de la palabra "Productos". Dando como resultado el siguiente texto "0 Productos" (El número 0 puede cambiar en base a la respuesta de nuestra consulta).

Si la solicitud no se realizó con éxito, se imprime un mensaje de error en la consola del navegador.

```
<!--Conteo de productos pertenecientes a X categoría-->
<script>
  $(document).ready(function () {
    // Realizar solicitud GET a la API para obtener el conteo de artículos que pertenecen a X categoría
    $.get("/api/v1/producto/countCategorias/1", function (data, status) {
      console.log("Response from server:", data); // Imprimir la respuesta del servidor
      console.log("Status of request:", status); // Imprimir el estado de la solicitud
      if (status === "success") {
        // Actualizar el contenido del elemento <small> con el número de productos
        $("#categoria1 small").text(data + " Productos");
      } else {
        console.log("Error al obtener la información");
      }
    });
  });
</script>
```

Ahora pasemos a las paginas de descripción de los artículos, utilizaremos solo uno como ejemplo, los demás son básicamente lo mismo, pero cambiando sus respectivos ID.

Para este codigo igualmente cuento con 3 scripts importantes: La obtención de la descripción, del precio y de la url de la imagen.

Descripción:

`$.get("/api/v1/producto/description/4", function (data, status) { ... });` Realiza una solicitud GET a la API con la ruta `"/api/v1/producto/description/4"` para obtener la descripción del producto con ID 4.

`$("#productName4").text(data);` Actualiza el contenido del elemento HTML `<h3>` con el ID `"productName4"` con la descripción del producto obtenida de la respuesta de la base de datos.

```
<!--Descripcion-->
<script>
    $(document).ready(function () {
        // Realizar solicitud GET a la API para obtener la descripción del producto con ID
        $.get("/api/v1/producto/description/4", function (data, status) {
            console.log("Response from server:", data); // Imprimir la respuesta del servidor
            console.log("Status of request:", status); // Imprimir el estado de la solicitud
            if (status === "success") {
                // Actualizar el contenido del elemento <h3> con la descripción del producto
                $("#productName4").text(data);
            } else {
                console.log("Error al obtener la descripción del producto.");
            }
        });
    });
</script>
```

Precio:

`$.get("/api/v1/producto/precio/4", function (data, status) { ... });` Realiza una solicitud GET a la API con la ruta `/api/v1/producto/precio/4` para obtener el precio del producto con ID 4.

`$("#productPrice4").text("$" + data.toFixed(2));` Esta línea de código actualiza el contenido del elemento HTML con el ID `"productPrice4"` para mostrar el precio del producto, junto al símbolo de dólar y con dos decimales.

```
<!-- Precio -->
<script>
    $(document).ready(function () {
        $.get("/api/v1/producto/precio/4", function (data, status) {
            console.log("Response from server:", data); // Imprimir la respuesta del servidor
            console.log("Status of request:", status); // Imprimir el estado de la solicitud
            if (status === "success") {
                // Actualizar el contenido del elemento <h3> con el precio del producto
                $("#productPrice4").text("$" + data.toFixed(2)); // Formatear el precio
            } else {
                console.log("Error al obtener el precio del producto.");
            }
        });
    });
</script>
```

Imagen:

`$.get("/api/v1/producto/img/4", function (data, status) { ... });` Realiza una solicitud GET a la API en la ruta `/api/v1/producto/img/4` para obtener la URL de la imagen del producto con ID 4.

`$("#productImage4").attr("src", data);` Actualiza el atributo `src` del elemento HTML con el ID `"productImage4"` con la URL de la imagen del producto, obtenida de la respuesta de la API.

```
<!-- Script para obtener y mostrar la imagen del producto -->
<script>
    $(document).ready(function () {
        // Realizar solicitud GET a la API para obtener la URL de la imagen del producto con ID 1
        $.get("/api/v1/producto/img/4", function (data, status) {
            console.log("Response from server:", data); // Imprimir la respuesta del servidor
            console.log("Status of request:", status); // Imprimir el estado de la solicitud
            if (status === "success") {
                // Actualizar el atributo src del elemento <img> con la URL de la imagen del producto
                $("#productImage4").attr("src", data);
            } else {
                console.log("Error al obtener la imagen del producto.");
            }
        });
    });
</script>
```

El funcionamiento de mis paginas de categorías es básicamente igual al del index, solamente se modifica la ruta API para cada pagina de categorías (/api/v1/producto/productoByCategorias/{categorialD}). La API se encarga de obtener todos los productos que pertenezcan a una categoría específica en base al idCategoría (SELECT \* FROM Productos WHERE idCategorias = ?)

```
<script>
$(document).ready(function () {
    // Realizar solicitud GET a la API para obtener la lista de productos
    $.get("/api/v1/producto/productoByCategorias/7", function (data, status) {
        console.log("Response from server:", data); // Imprimir la respuesta del servidor
        console.log("Status of request:", status); // Imprimir el estado de la solicitud
        if (status === "success") {
            // Recorrer la lista de productos
            data.forEach(function (producto) {
                // Crear el HTML para cada producto
                var productHtml = `
<div class="col-lg-3 col-md-4 col-sm-6 pb-1">
    <div class="product-item bg-light mb-4">
        <div class="product-img position-relative overflow-hidden">
            
        </div>
        <div class="text-center py-4" id="productDetails${producto.id}">
            <a class="h6 text-decoration-none text-truncate" href="${producto.descripcion}.html">${producto.descripcion}</a>
            <div class="d-flex align-items-center justify-content-center mt-2">
                <h5>${producto.precio}</h5>
            </div>
            <div class="d-flex align-items-center justify-content-center mb-1">
                <small class="fa fa-star text-primary mr-1"></small>
                <small class="fa fa-star text-primary mr-1"></small>
                <small class="fa fa-star text-primary mr-1"></small>
                <small class="fa fa-star-half-alt text-primary mr-1"></small>
                <small class="far fa-star text-primary mr-1"></small>
            </div>
        </div>
    </div>
</div>`;
                // Agregar el HTML del producto al div
                $("#productContainer").append(productHtml);
            });
        } else {
            console.log("Error al obtener la lista de productos.");
        }
    });
});
</script>
```

Para la pagina shop.html que muestra todos los artículos el script y la parte mas importante es todo lo que hace funcionar al carrito de compras

Realizamos una solicitud GET a la API en la ruta `"/api/v1/producto/list"` para obtener una lista de productos. Utiliza la función ``fetch`` para realizar la solicitud y la función ``response.json()`` para convertir la respuesta en formato JSON. Los datos obtenidos se almacenan en la variable ``baseDeDatos``.

Iteramos sobre los datos obtenidos y creamos elementos HTML para cada producto, incluyendo su imagen, descripción, precio y un botón para añadir al carrito. Estos elementos se agregan al DOM para que sean visibles.

Cuando hacemos clic en el botón de un producto, añadimos el ID del producto al array ``carrito``. Luego llamamos a la función ``renderizarCarrito()`` para actualizar la visualización de nuestro carrito en la página.

Mostramos cada producto en una lista junto con su cantidad y precio. También proporcionamos un botón de eliminar para cada producto en el carrito. Calculamos y mostramos el precio total del carrito.

Cuando hacemos clic en el botón de eliminar de un producto en el carrito, eliminamos ese producto del array `"carrito"` y actualizamos la visualización del carrito en la página.

Al hacer clic en el botón "Vaciar carrito", vaciamos completamente el array `"carrito"` actualizamos la visualización del carrito en la página.

```
<script>
  document.addEventListener('DOMContentLoaded', () => {
    // Variables
    let baseDeDatos = [];

    // Función para obtener los datos de la API
    async function obtenerDatosAPI() {
      try {
        const response = await fetch('/api/v1/producto/list');
        const data = await response.json();
        baseDeDatos = data;
        renderizarProductos();
      } catch (error) {
        console.error('Error al obtener los datos de la API:', error);
      }
    }

    let carrito = [];
    const divisa = '$';
    const DOMitems = document.querySelector('#items');
    const DOMcarrito = document.querySelector('#carrito');
    const DOMtotal = document.querySelector('#total');
    const DOMbotonVaciar = document.querySelector('#boton-vaciar');

    // Funciones
    function renderizarProductos() {
      baseDeDatos.forEach((info) => {
        // Estructura
        const miNodo = document.createElement('div');
        miNodo.classList.add('card', 'col-sm-4');
        // Body
        const miNodoCardBody = document.createElement('div');
        miNodoCardBody.classList.add('card-body');
```

```

// Título
const miNodoTitulo = document.createElement('h5');
miNodoTitulo.classList.add('card-title');
miNodoTitulo.textContent = info.descripcion;
// Imagen
const miNodoImagen = document.createElement('img');
miNodoImagen.classList.add('img-fluid');
miNodoImagen.setAttribute('src', info.img);
// Precio
const miNodoPrecio = document.createElement('p');
miNodoPrecio.classList.add('card-text');
miNodoPrecio.textContent = `${divisa}${info.precio}`;
// Boton
const miNodoBoton = document.createElement('button');
miNodoBoton.classList.add('btn', 'btn-primary');
miNodoBoton.textContent = '+';
miNodoBoton.setAttribute('marcador', info.id);
miNodoBoton.addEventListener('click', anyadirProductoAlCarrito);
// Insertamos
miNodoCardBody.appendChild(miNodoImagen);
miNodoCardBody.appendChild(miNodoTitulo);
miNodoCardBody.appendChild(miNodoPrecio);
miNodoCardBody.appendChild(miNodoBoton);
miNodo.appendChild(miNodoCardBody);
DOMItems.appendChild(miNodo);
});
}

/*Evento para añadir un producto al carrito de la compra*/
function anyadirProductoAlCarrito(eyento) {
  // Añadir el nodo a nuestro carrito
  carrito.push(eyento.target.getAttribute('marcador'));
  // Actualización del carrito
  renderizarCarrito();
}

function renderizarCarrito() {
  DOMcarrito.textContent = '';
  const carritosSinDuplicados = [...new Set(carrito)];
  carritosSinDuplicados.forEach((item) => {
    const miItem = baseDeDatos.find((itemBaseDatos) => itemBaseDatos.id === parseInt(item));
    // Cuenta el número de veces que se repite el producto
    const numeroUnidadesItem = carrito.reduce((total, itemId) => {
      return itemId === item ? total += 1 : total;
    }, 0);
    const miNodo = document.createElement('li');
    miNodo.classList.add('list-group-item', 'text-right', 'mx-2');
    miNodo.textContent = `${numeroUnidadesItem} x ${miItem.descripcion} - ${divisa}${miItem.precio}`;
    // Boton de borrar
    const miBoton = document.createElement('button');
    miBoton.classList.add('btn', 'btn-danger', 'mx-5');
    miBoton.textContent = 'X';
    miBoton.style.marginLeft = '1rem';
    miBoton.dataset.item = item;
    miBoton.addEventListener('click', borrarItemCarrito);
    miNodo.appendChild(miBoton);
    DOMcarrito.appendChild(miNodo);
  });
  // Calculamos el precio total en el HTML
  DOMtotal.textContent = calcularTotal();
}

```

```

/*Borrar un elemento del carrito*/
function borrarItemCarrito(evento) {
    // obtenemos el producto ID que hay en el boton pulsado
    const id = evento.target.dataset.item;
    // Borramos todos los productos
    carrito = carrito.filter((carritoId) => {
        return carritoId !== id;
    });
    renderizarCarrito();
}

/*Calcula el precio total teniendo en cuenta los productos repetidos*/
function calcularTotal() {
    // Recorremos el array del carrito
    return carrito.reduce((total, item) => {
        const miItem = baseDeDatos.find((itemBaseDatos) => itemBaseDatos.id === parseInt(item));
        return total + miItem.precio;
    }, 0).toFixed(2);
}

/* Vacía el carrito */
function vaciarCarrito() {
    // Limpiamos los productos guardados
    carrito = [];
    renderizarCarrito();
}

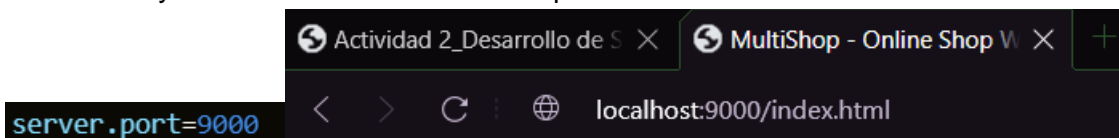
DOMbotonVaciar.addEventListener('click', vaciarCarrito);

obtenerDatosAPI();
});
</script>

```

## Conexión a la BD

Con `server.port` establecemos que el puerto en el que se ejecutara nuestra aplicación será en el 9000 como ya hemos visto en nuestras capturas anteriores



Aquí especificamos la URL de nuestra conexión a la base de datos, utilizando JDBC para conectarse a SQL Server donde la base de datos se llama TiendaSara, con `encrypt=true` indicamos que utilizaremos cifrado SSL y con `trustServerCertificate=true` indicamos que debe confiar en el servidor para evitar cualquier problema de que se rechace la conexión

```
spring.datasource.url=jdbc:sqlserver://localhost;databaseName=TiendaSara;encrypt=true;trustServerCertificate=true
```

Con esto especificamos el nombre de usuario y contraseña para la conexión con la base de datos

```
spring.datasource.username=sa
spring.datasource.password=12345678
```

Aquí habilitamos la visualización de consultas SQL generadas por hibernate

```
spring.jpa.show-sql=true
```

Especifica como hibernate debe actualizar el esquema de la base de datos. Con 'update', Hibernate creará, modificará o eliminará tablas según sea necesario para que coincidan con las definiciones de las entidades.

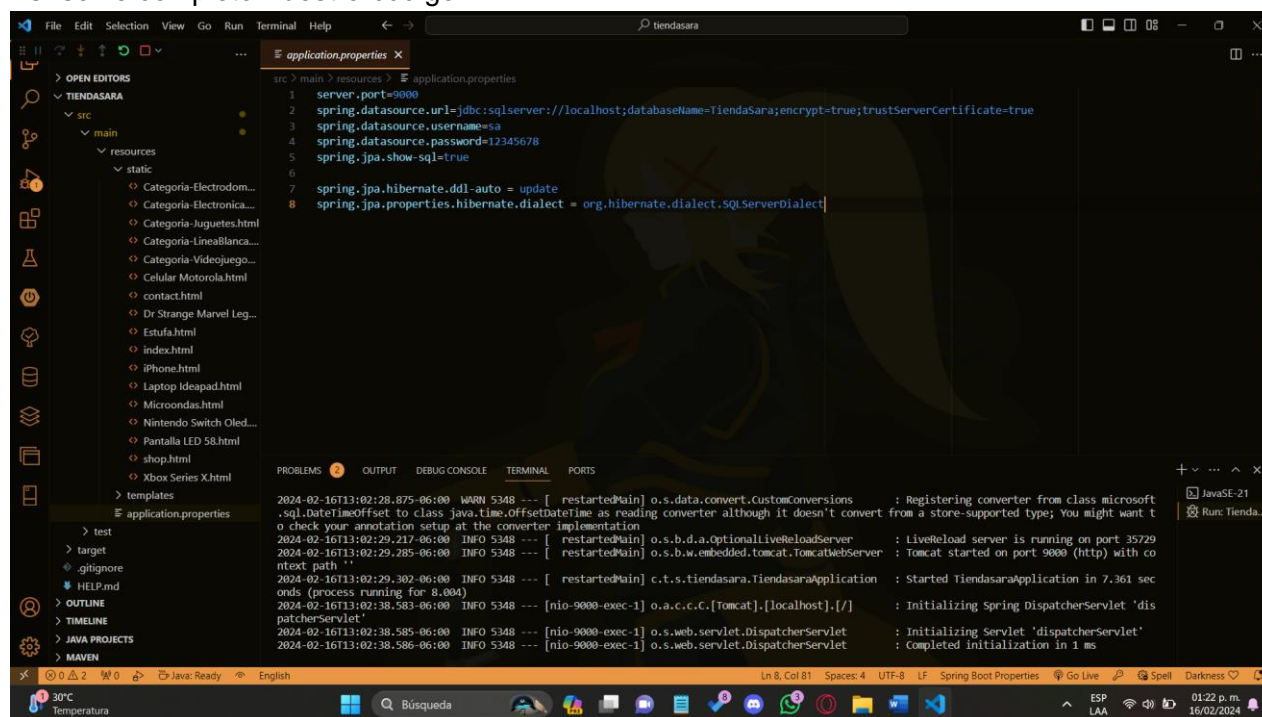
```
spring.jpa.hibernate.ddl-auto = update
```

se encarga de generar las consultas SQL específicas del motor de base de datos que se está utilizando. En este caso, se utiliza el dialecto `SQLServerDialect` para SQL Server.



```
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.SQLServerDialect
```

Así se ve completo nuestro código



```
src > main > resources > application.properties
1  server.port=9000
2  spring.datasource.url=jdbc:mysql://localhost:database=tiendaSara;encrypt=true;trustServerCertificate=true
3  spring.datasource.username=sa
4  spring.datasource.password=12345678
5  spring.jpa.show-sql=true
6
7  spring.jpa.hibernate.ddl-auto = update
8  spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect

2024-02-16T13:02:28.875-06:00 WARN 5348 --- [ restartedMain] o.s.data.convert.CustomConversions : Registering converter from class microsoft
.sql.Date to class java.time.OffsetDateTime as reading converter although it doesn't convert from a store-supported type; You might want t
o check your annotation setup at the converter implementation
2024-02-16T13:02:29.217-06:00 INFO 5348 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-02-16T13:02:29.285-06:00 INFO 5348 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 9000 (http) with co
ntext path ''
2024-02-16T13:02:29.302-06:00 INFO 5348 --- [ restartedMain] c.t.s.tiendasara.TiendasaraApplication : Started TiendasaraApplication in 7.361 sec
onds (process running for 8.004)
2024-02-16T13:02:38.583-06:00 INFO 5348 --- [nio-9000-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dis
patcherServlet'
2024-02-16T13:02:38.585-06:00 INFO 5348 --- [nio-9000-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-02-16T13:02:38.586-06:00 INFO 5348 --- [nio-9000-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
```

## Conclusión

La creación de un servicio web utilizando Java y Spring Boot con una API REST para gestionar productos de una tienda en línea es una tarea fundamental en el desarrollo de aplicaciones modernas. Esta actividad proporciona una base sólida para el desarrollo de sistemas de comercio electrónico, que son vitales en el mundo empresarial actual.

El uso de Java y Spring Boot garantiza un desarrollo eficiente y robusto, aprovechando las características de un lenguaje de programación ampliamente utilizado en el desarrollo empresarial y un marco de trabajo que simplifica el desarrollo de aplicaciones web. La API REST proporciona una forma estandarizada de comunicarse con el sistema, permitiendo la integración con otros sistemas y la construcción de interfaces de usuario flexibles.

La importancia de esta actividad radica en la capacidad de proporcionar una experiencia de usuario agradable y eficiente en la compra de productos en línea. Al implementar una galería de productos con características como filtrado por categorías y búsqueda, se mejora la usabilidad y la satisfacción del cliente. Además, la capacidad de conectar el sistema con una base de datos permite gestionar de manera efectiva el inventario de productos y realizar operaciones como agregar, actualizar y eliminar productos.

## GitHub

<https://github.com/JusiLinGu>