

Manejo de imágenes con Spring Boot y React

Índice

- 01** **Presentación**
- 02** Implementación en Spring Boot
- 03** Manejo de imágenes en Spring Boot
- 04** Nuestro frontend con React
- 05** Implementando Tailwind CSS
- 06** Consumiendo la API de Spring Boot
- 07** Preguntas ?



01

Presentación



Profesora

Elena Fischietto

Apodo: Ele

Hola! Soy desarrolladora web full stack. Soy graduada de CTD y he realizado cursos en Platzi y plataformas similares.

Me encuentro estudiando Ingeniería en Inteligencia Artificial en la Universidad de Palermo.

Las tecnologías que utilizo son: React, Typescript, Redux, Node, PHP, Java, Bases de datos SQL, entre otras.

Trabajo en el área IT de una startup argentina donde buscamos integrar la tecnología para mejorar la experiencia de los usuarios.



Profesor

José Miranda

Apodo: Jose

Hola! Soy desarrollador web full stack. Graduado del Curso Intensivo Full Stack.

Hace 2 años trabajo en DH, como profesor del Curso Intensivo Full Stack, en el equipo de Contenidos y actualmente estoy trabajando Junto Con Ele y Humber como Community Managers Técnicos en la modalidad Ondemand de CTD.

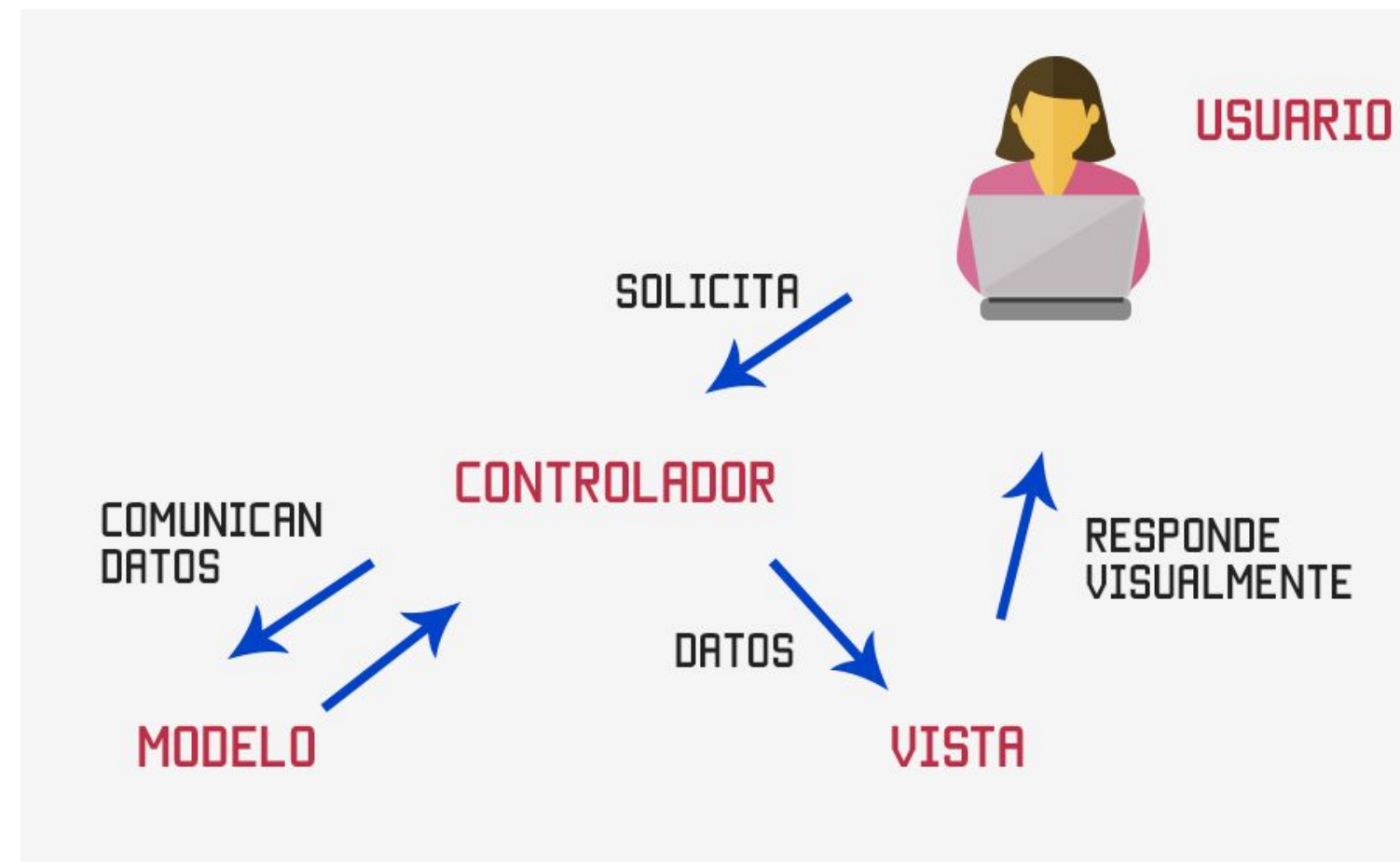
Las tecnologías que utilizo son Js, Ts, como React, Next, Express, Bases de datos Relacionales y no Relacionales, Java, entre otros.

02

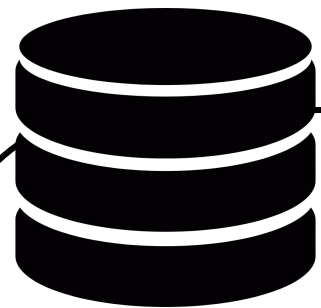
Implementación en Spring Boot

Modelo Vista Controlador (MVC)

Patrón de diseño que separa, por un lado, la estructura y manipulación de datos, y por otro, la forma en que se muestran.

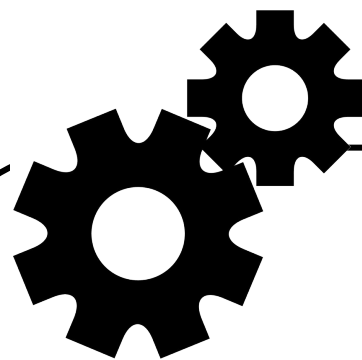


Compuesto por



Modelo

Representa los datos y la lógica de negocio. Responsable de manejar la información, realizar validaciones, y responder a las consultas de la vista o el controlador.



Controlador

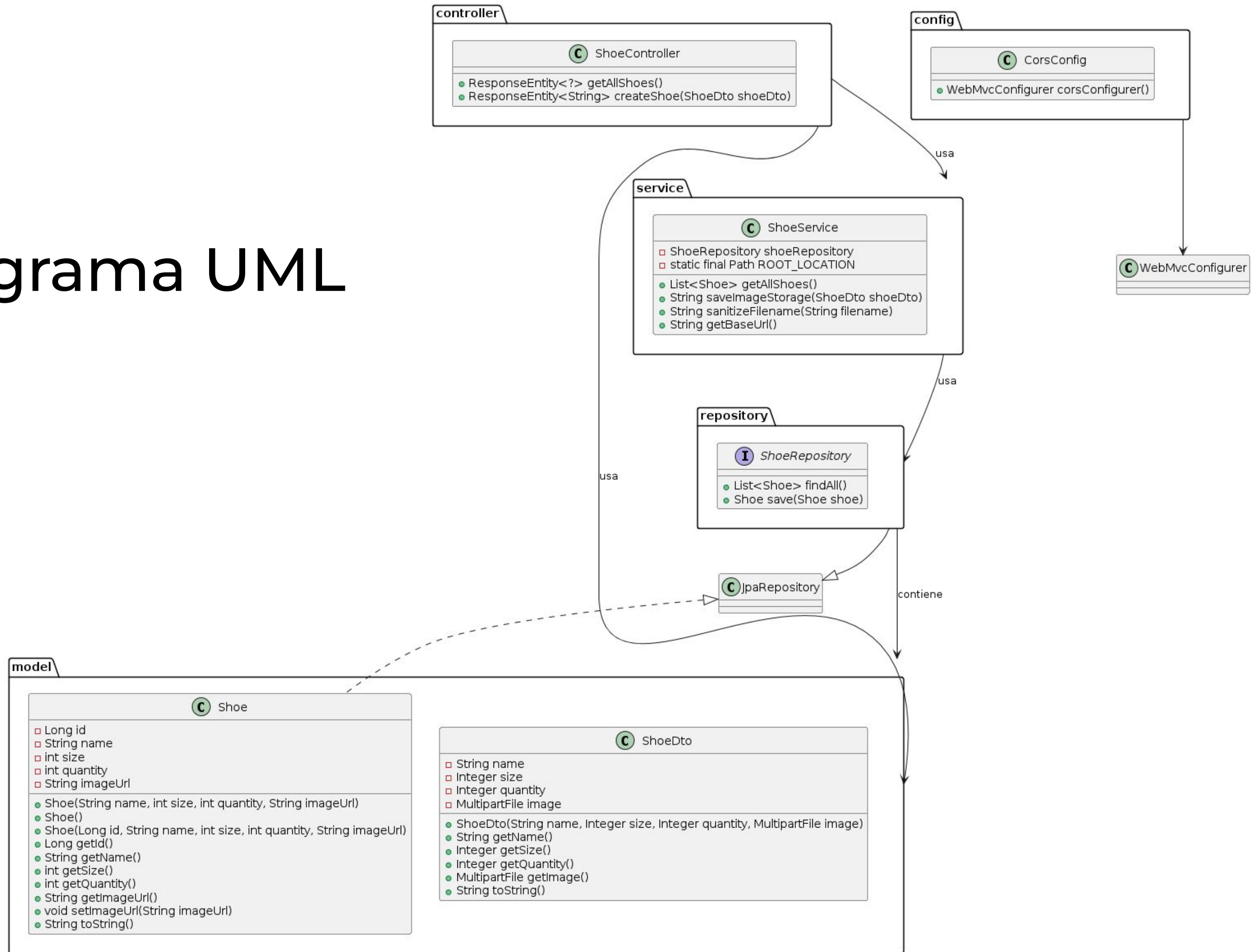
Actúa como intermediario entre el modelo y la vista. Responde a las interacciones del usuario, maneja las solicitudes de datos del modelo y actualiza la vista.



Vista

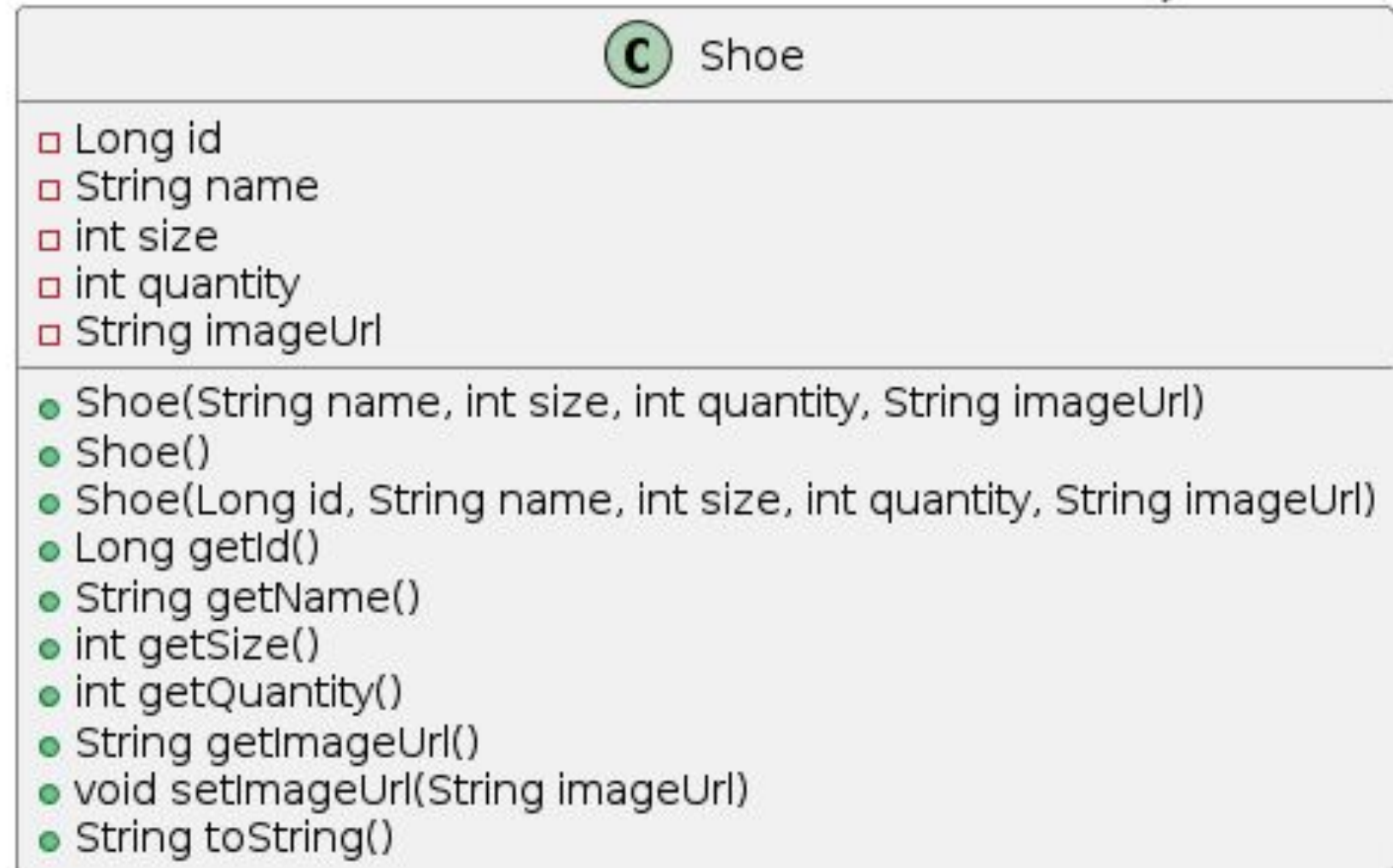
Representación visual de los datos contenidos en el modelo. Muestra la información al usuario de manera adecuada (por ejemplo una api).

Diagrama UML



Modelo

Define las entidades de datos que representan objetos de dominio en la aplicación.



DTO

El objeto de transferencia de datos es una clase que sirve para representar un modelo y sirve para transferir datos entre las distintas capas del proyecto.

C ShoeDto	
□ String name	
□ Integer size	
□ Integer quantity	
□ MultipartFile image	
● ShoeDto(String name, Integer size, Integer quantity, MultipartFile image)	
● String getName()	
● Integer getSize()	
● Integer getQuantity()	
● MultipartFile getImage()	
● String toString()	

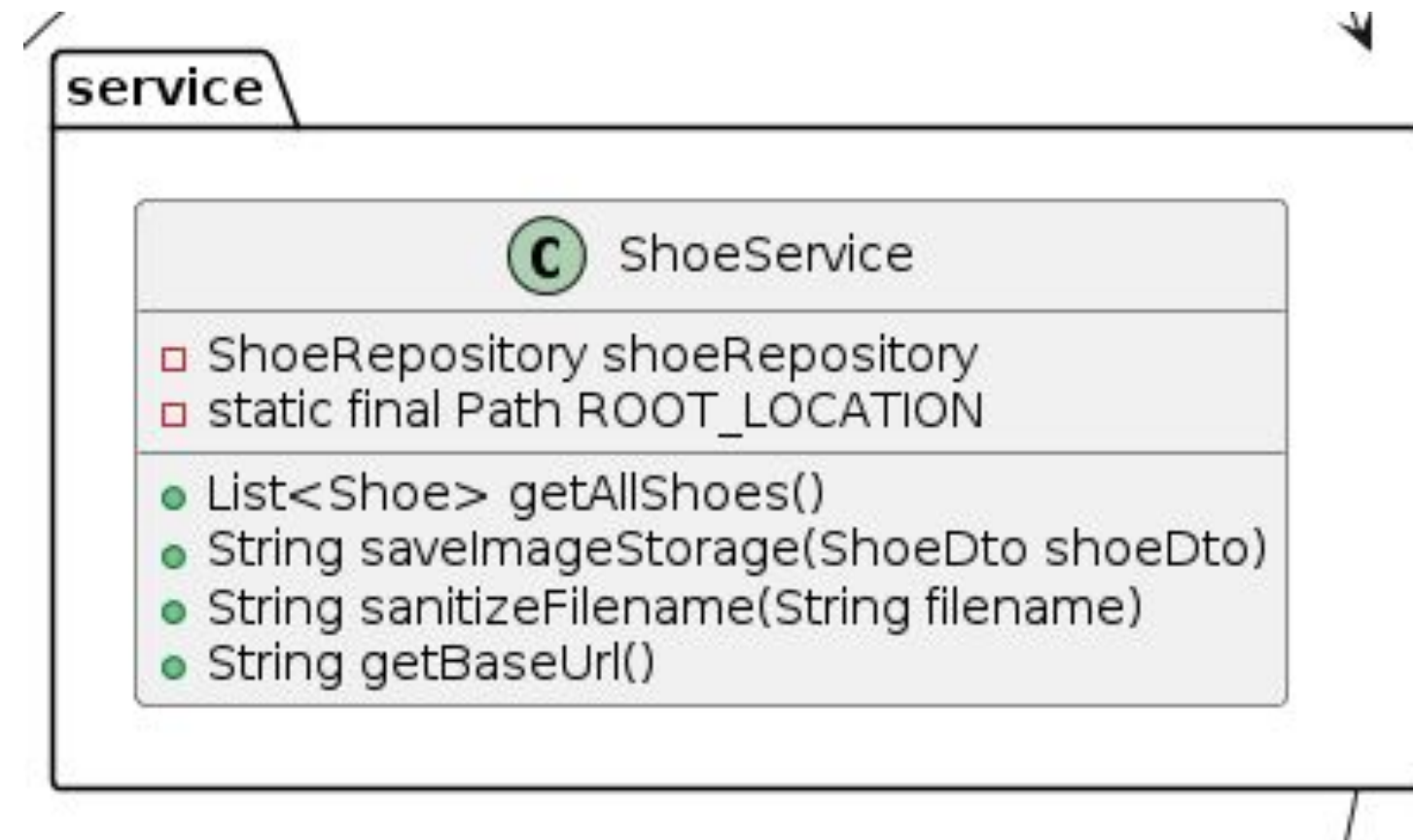
Repositorio

Contiene operaciones para acceder y manipular datos persistentes en la base de datos.



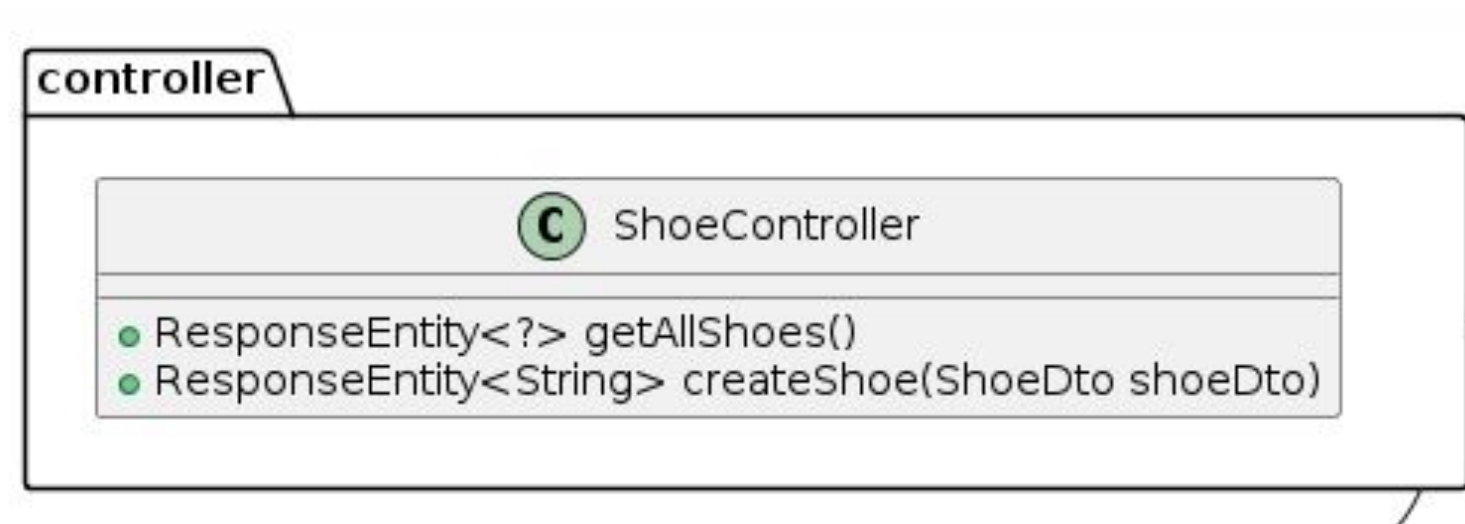
Service

Contiene la lógica de negocio de la aplicación y sirve como intermediario entre el controlador y el repositorio.



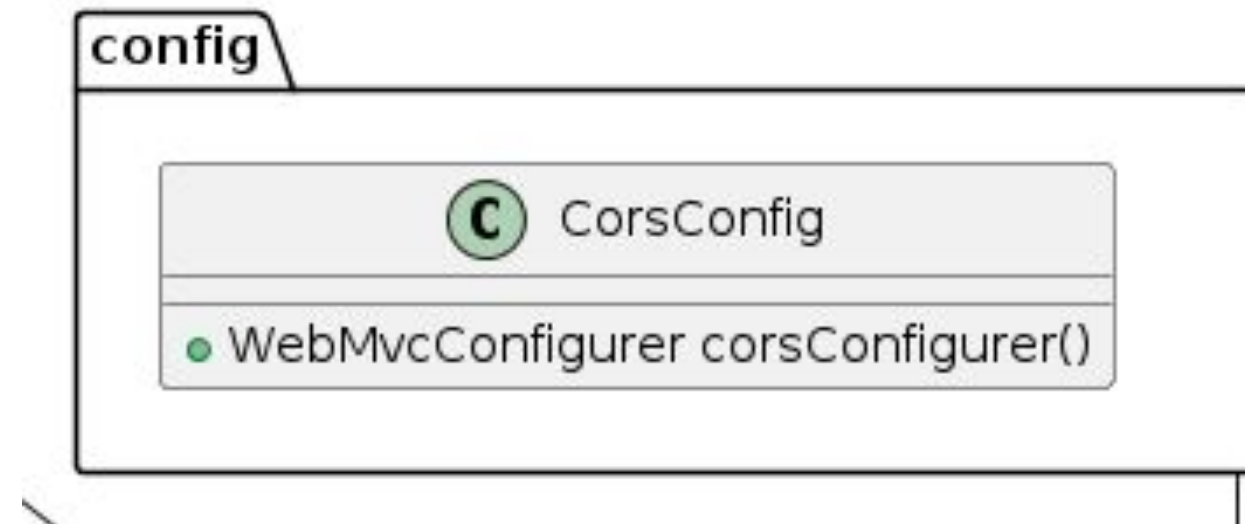
Controller

Contiene los accesos que gestionan las solicitudes HTTP y las respuestas para interactuar con los clientes.



Config

Se agrupan las configuraciones específicas de la aplicación, por ejemplo los CORS.



CORS es una política de seguridad presente en los navegadores que limita las solicitudes HTTP realizadas desde una página web a un servidor con dominios (origin) distintos.

03

Manejo de imágenes en Spring Boot

Qué es una imagen y cómo se almacena

- Una imagen es un archivo binario (compuesto de 0 y 1).
- Debido a su estructura compleja, suelen ser pesados y sería costoso en términos de recursos almacenarla en una base de datos (db) directamente.
- Por esto mismo en que en una db solo se guardará la url de acceso a donde esté alojado el recurso, por ejemplo un bucket de AWS o, en nuestro caso, un directorio local de nuestra máquina.



Por lo que en nuestro proyecto vamos a preparar dos clases en el package model:

- La **clase Shoe** que va a tener un string que contendrá la url de la imagen. Se guardará en la base de datos.
- La clase **ShoeDto** que contiene un atributo de tipo **MultipartFile**, y sirve para representar archivos enviados desde un formulario. Nos servirá para manipular la imagen hasta el momento de almacenarla y obtener el path.

```
@Entity
public class Shoe {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private int size;
    private int quantity;
    private String imageUrl;
```

```
public class ShoeDto {

    private String name;
    private Integer size;
    private Integer quantity;
    private MultipartFile image;
```

En el service vamos a tener dos métodos. El primero será para **guardar una imagen nueva**, que realiza una serie de pasos que veremos a continuación.

```
public String saveImageStorage(ShoeDto shoeDto) throws IOException {

    MultipartFile file = shoeDto.getImage();

    if (file.isEmpty()) {
        throw new IOException("Failed to store empty file.");
    }

    if (!Files.exists(ROOT_LOCATION)) {
        Files.createDirectories(ROOT_LOCATION);
    }

    String originalFilename = file.getOriginalFilename();
    if (originalFilename == null) {
        throw new IOException("Name of file invalid");
    }

    String sanitizedFilename = sanitizeFilename(originalFilename);
    String fileName = UUID.randomUUID().toString() + "_" + sanitizedFilename;
    Path destinationFile = ROOT_LOCATION.resolve(Paths.get(fileName));

    try {
        Files.copy(file.getInputStream(), destinationFile);
    } catch (IOException e) {
        throw new IOException("Failed to store file " + fileName, e);
    }

    String fullPath = destinationFile.toAbsolutePath().toString();

    try {
        Shoe shoe = new Shoe(shoeDto.getName(), shoeDto.getSize(), shoeDto.getQuantity(), fileName);
        shoeRepository.save(shoe);
    } catch (Exception e) {
        throw new IOException("Failed to save image URL to database: " + e.getMessage(), e);
    }

    return fullPath;
}
```


Del DTO recibido va a extraer el archivo de imagen. Si el archivo está vacío va a retornar un error.

```
public String saveImageStorage(ShoeDto shoeDto) throws IOException {  
  
    MultipartFile file = shoeDto.getImage();  
  
    if (file.isEmpty()) {  
        throw new IOException("Failed to store empty file.");  
    }  
  
    if (!Files.exists(ROOT_LOCATION)) {  
        Files.createDirectories(ROOT_LOCATION);  
    }  
  
    String originalFilename = file.getOriginalFilename();  
    if (originalFilename == null) {  
        throw new IOException("Name of file invalid");  
    }  
  
    String sanitizedFilename = sanitizeFilename(originalFilename);  
    String fileName = UUID.randomUUID().toString() + "_" + sanitizedFilename;  
    Path destinationFile = ROOT_LOCATION.resolve(Paths.get(fileName));  
  
    try {  
        Files.copy(file.getInputStream(), destinationFile);  
    } catch (IOException e) {  
        throw new IOException("Failed to store file " + fileName, e);  
    }  
  
    String fullPath = destinationFile.toAbsolutePath().toString();  
  
    try {  
        Shoe shoe = new Shoe(shoeDto.getName(), shoeDto.getSize(), shoeDto.getQuantity(), fileName);  
        shoeRepository.save(shoe);  
    } catch (Exception e) {  
        throw new IOException("Failed to save image URL to database: " + e.getMessage(), e);  
    }  
  
    return fullPath;  
}
```

Verificamos que el directorio en que queremos guardar la imagen exista (indicado en la constante), y si no existe lo crea.

```
public String saveImageStorage(ShoeDto shoeDto) throws IOException {

    MultipartFile file = shoeDto.getImage();

    if (file.isEmpty()) {
        throw new IOException("Failed to store empty file.");
    }

    if (!Files.exists(ROOT_LOCATION)) {
        Files.createDirectories(ROOT_LOCATION);
    }

    String originalFilename = file.getOriginalFilename();
    if (originalFilename == null) {
        throw new IOException("Name of file invalid");
    }

    String sanitizedFilename = sanitizeFilename(originalFilename);
    String fileName = UUID.randomUUID().toString() + "_" + sanitizedFilename;
    Path destinationFile = ROOT_LOCATION.resolve(Paths.get(fileName));

    try {
        Files.copy(file.getInputStream(), destinationFile);
    } catch (IOException e) {
        throw new IOException("Failed to store file " + fileName, e);
    }

    String fullPath = destinationFile.toAbsolutePath().toString();

    try {
        Shoe shoe = new Shoe(shoeDto.getName(), shoeDto.getSize(), shoeDto.getQuantity(), fileName);
        shoeRepository.save(shoe);
    } catch (Exception e) {
        throw new IOException("Failed to save image URL to database: " + e.getMessage(), e);
    }

    return fullPath;
}
```


Va a extraer el nombre del archivo, lo sanitiza y agrega un id aleatorio para evitar nombres duplicados.

```
public String saveImageStorage(ShoeDto shoeDto) throws IOException {

    MultipartFile file = shoeDto.getImage();

    if (file.isEmpty()) {
        throw new IOException("Failed to store empty file.");
    }

    if (!Files.exists(ROOT_LOCATION)) {
        Files.createDirectories(ROOT_LOCATION);
    }

    String originalFilename = file.getOriginalFilename();
    if (originalFilename == null) {
        throw new IOException("Name of file invalid");
    }

    String sanitizedFilename = sanitizeFilename(originalFilename);
    String fileName = UUID.randomUUID().toString() + "_" + sanitizedFilename;
    Path destinationFile = ROOT_LOCATION.resolve(Paths.get(fileName));

    try {
        Files.copy(file.getInputStream(), destinationFile);
    } catch (IOException e) {
        throw new IOException("Failed to store file " + fileName, e);
    }

    String fullPath = destinationFile.toAbsolutePath().toString();

    try {
        Shoe shoe = new Shoe(shoeDto.getName(), shoeDto.getSize(), shoeDto.getQuantity(), fileName);
        shoeRepository.save(shoe);
    } catch (Exception e) {
        throw new IOException("Failed to save image URL to database: " + e.getMessage(), e);
    }

    return fullPath;
}
```

Guardamos la imagen en el directorio destino y, si falla, retorna un error.

```
public String saveImageStorage(ShoeDto shoeDto) throws IOException {

    MultipartFile file = shoeDto.getImage();

    if (file.isEmpty()) {
        throw new IOException("Failed to store empty file.");
    }

    if (!Files.exists(ROOT_LOCATION)) {
        Files.createDirectories(ROOT_LOCATION);
    }

    String originalFilename = file.getOriginalFilename();
    if (originalFilename == null) {
        throw new IOException("Name of file invalid");
    }

    String sanitizedFilename = sanitizeFilename(originalFilename);
    String fileName = UUID.randomUUID().toString() + "_" + sanitizedFilename;
    Path destinationFile = ROOT_LOCATION.resolve(Paths.get(fileName));

    try {
        Files.copy(file.getInputStream(), destinationFile);
    } catch (IOException e) {
        throw new IOException("Failed to store file " + fileName, e);
    }

    String fullPath = destinationFile.toAbsolutePath().toString();

    try {
        Shoe shoe = new Shoe(shoeDto.getName(), shoeDto.getSize(), shoeDto.getQuantity(), fileName);
        shoeRepository.save(shoe);
    } catch (Exception e) {
        throw new IOException("Failed to save image URL to database: " + e.getMessage(), e);
    }

    return fullPath;
}
```


Obtiene la ruta de la imagen y, junto con los datos obtenidos del DTO, crea un objeto Shoe para guardar en la DB.
Retorna el path de la imagen.

```
public String saveImageStorage(ShoeDto shoeDto) throws IOException {

    MultipartFile file = shoeDto.getImage();

    if (file.isEmpty()) {
        throw new IOException("Failed to store empty file.");
    }

    if (!Files.exists(ROOT_LOCATION)) {
        Files.createDirectories(ROOT_LOCATION);
    }

    String originalFilename = file.getOriginalFilename();
    if (originalFilename == null) {
        throw new IOException("Name of file invalid");
    }

    String sanitizedFilename = sanitizeFilename(originalFilename);
    String fileName = UUID.randomUUID().toString() + "_" + sanitizedFilename;
    Path destinationFile = ROOT_LOCATION.resolve(Paths.get(fileName));

    try {
        Files.copy(file.getInputStream(), destinationFile);
    } catch (IOException e) {
        throw new IOException("Failed to store file " + fileName, e);
    }

    String fullPath = destinationFile.toAbsolutePath().toString();

    try {
        Shoe shoe = new Shoe(shoeDto.getName(), shoeDto.getSize(), shoeDto.getQuantity(), fileName);
        shoeRepository.save(shoe);
    } catch (Exception e) {
        throw new IOException("Failed to save image URL to database: " + e.getMessage(), e);
    }

    return fullPath;
}
```


El segundo método será para obtener todas las imágenes en la db:

- Utilizamos la función findAll que se obtiene del repositorio.
- Le agregamos la ruta absoluta de la imagen para nuestra máquina.
- Retorna una lista de Shoes.

```
public List<Shoe> getAllShoes() throws Exception {  
    try {  
        List<Shoe> shoes = shoeRepository.findAll();  
        shoes.forEach(shoe -> {  
            String baseUrl = getBaseUrl();  
            String imageUrl = baseUrl + "/images/" + shoe.getImageUrl();  
            shoe.setImageUrl(imageUrl);  
        });  
        return shoes;  
    } catch (Exception e) {  
        throw new Exception("Failed to retrieve shoes: " + e.getMessage(), e);  
    }  
}
```

De esta forma en el controller solo nos ocuparemos de llamar al método correspondiente y manejar errores, aislando la lógica de negocio.

```
public class ShoeController {

    @Autowired
    private ShoeService shoeService;

    @GetMapping
    public ResponseEntity<?> getAllShoes() {
        try {
            List<Shoe> shoes = shoeService.getAllShoes();
            return ResponseEntity.ok(shoes);
        } catch (Exception e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }

    @PostMapping
    public ResponseEntity<String> createShoe(@ModelAttribute ShoeDto shoeDto
    ) {
        try {
            return ResponseEntity.ok(shoeService.saveImageStorage(shoeDto));
        } catch (Exception e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }
}
```



@ModelAttribute se usa para asociar datos enviados por un cliente a un DTO.

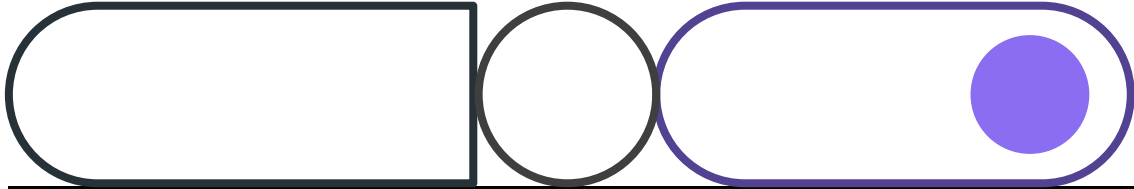
Si al recibir el path de una imagen simplemente intentamos acceder desde el navegador, Spring Boot nos va a informar que la ruta no existe, y esto es porque tenemos que configurar el manejo de archivos estáticos.

Este fragmento lo agregaremos en el archivo de configuración de CORS, y especifica que para todas las solicitudes a las rutas que comienzan con **/images**, se van a buscar archivos dentro del directorio `/images` en la raíz del proyecto.

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/images/**")
        .addResourceLocations("file:./images/");
}
```

04

Nuestro frontend con React



Elegimos React debido a su eficiencia y flexibilidad. Nos permite construir interfaces de usuario interactivas y reutilizables, lo que facilita el mantenimiento y escalabilidad del código. Su enfoque basado en componentes y su virtual DOM optimizan el rendimiento.



Vite como herramienta de construcción

Vamos a crear un entorno de desarrollo **rápido** y eficiente para aplicaciones web, utilizando una configuración mínima y aprovechando la capacidad de los módulos de ES nativos del navegador para ofrecer una recarga en caliente ultra rápida y una compilación optimizada.



05

Implementando Tailwind CSS

Qué es Tailwind CSS

Tailwind CSS es un framework de utilidad para CSS que permite crear diseños personalizados de manera rápida y eficiente mediante clases predefinidas. Para este ejemplo vamos a seguir la [documentación oficial](#) para la instalación de un proyecto nuevo con Vite



Tailwind CSS

Pasos para la instalación de nuestro Frontend

Como primer paso necesitamos crear el proyecto en sí, vamos a utilizar Vite y Tailwind

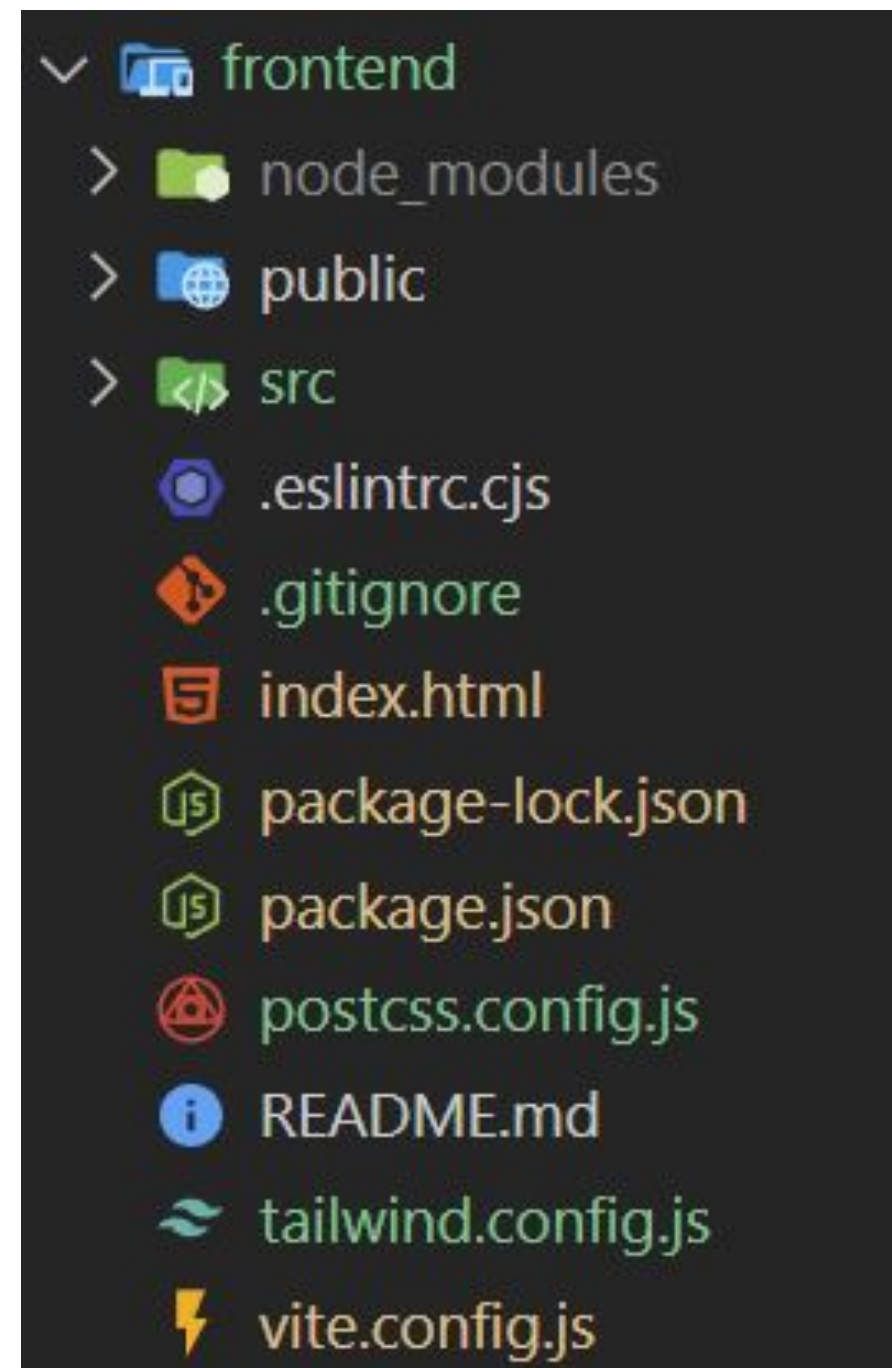
npm create vite frontend --template react

Nos posicionamos en la carpeta con CD

cd frontend

Luego instalamos Tailwind CSS con los siguientes comandos

npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p



Vamos a detallar un poco de lo que se instala

tailwindcss: Instalamos el framework de CSS que nos van a brindar todas las clases para aplicar estilos dentro del elemento.

postcss: Una herramienta que se utiliza para procesar el CSS y aplicar transformaciones automáticas, como optimizaciones y modificaciones que mejoran el rendimiento y la compatibilidad

autoprefixer: Un plugin de PostCSS que añade prefijos específicos del navegador al CSS para asegurar que funcione en todos los navegadores.

Post-Procesado

```
.button {  
  display: flex;  
  user-select: none;  
  transition: transform 0.2s;  
}
```

Partimos de unos estilos simples que requieren prefijos de compatibilidad para los navegadores

```
.button {  
  display: -webkit-box;  
  display: -ms-flexbox;  
  display: flex;  
  -webkit-user-select: none;  
    -moz-user-select: none;  
    -ms-user-select: none;  
      user-select: none;  
  -webkit-transition: -webkit-transform 0.2s;  
  transition: -webkit-transform 0.2s;  
  transition: transform 0.2s;  
  transition: transform 0.2s, -webkit-transform 0.2s;  
}
```

Luego del post-procesado obtenemos la compatibilidad de los navegadores automáticamente.

Luego de ejecutar el comando de inicialización (**`npx tailwindcss init -p`**) debemos configurar una serie de archivos.



Configurar la detección de los archivos

Debemos asegurarnos de que Tailwind CSS detecte todos los archivos donde vamos a usar clases de Tailwind, por lo tanto vamos a modificar el archivo **tailwind.config.js**

```
/** @type {import('tailwindcss').Config} */  
export default {  
  content: [  
    "./index.html",  
    "./src/**/*.{js,ts,jsx,tsx}",  
  ],  
  theme: {  
    extend: {},  
  },  
  plugins: [],  
}
```

Configurar las directivas necesarias

Un último paso, previo a correr el servidor, sería configurar las directivas necesarias para que Tailwind procese y genere las clases que utilizaremos en el proyecto. Para esto, dentro de nuestro **index.css** debemos agregar las siguientes líneas de código.

Con **@tailwind base**: importamos los estilos base de Tailwind CSS que normalizan el estilo entre navegadores. Incluye estilos como reset de márgenes, padding, etc.

Con **@tailwind components**: importamos los estilos de los componentes predefinidos de Tailwind CSS, como botones, inputs, y otros componentes UI.

Con **@tailwind utilities**: importamos todas las clases de

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;
```


Una vez configurado nuestro proyecto podemos comenzar a utilizar las clases que nos ofrece Tailwind sin problemas



Crear variables de entorno en React+Vite



.ENV

Vite tiene por defecto una configuración para integrar las variables de entorno. Para crearlas necesitamos un archivo `.env` donde deben llevar el prefijo **VITE_** para que puedan utilizarse por defecto.

En nuestro caso utilizamos **VITE_URL_API** como se puede observar a continuación en su definición y uso para servir la URL de la api creada en la etapa anterior

`.env`

```
VITE_URL_API=http://localhost:8080/shoes
```

`Component.jsx`

```
fetch(import.meta.env.VITE_URL_API)
  .then((res) => res.json())
  .then((data) => {
    setShoes(data);
  })
```


Obteniendo datos de una API

En este ejemplo obtenemos mediante una variable de entorno el endpoint para acceder al Listado de productos que nos devuelve la API utilizando el método GET.

Esta data sería lista para actualizar el estado que guarda el Array del listado de Zapatos para luego ser iterada para imprimir cada una de las tarjetas o cual sea su objetivo final.

```
fetch(import.meta.env.VITE_URL_API)
  .then((res) => res.json())
  .then((data) => {
    setShoes(data);
  })
  .catch((error) => {
    console.error("Error al obtener los productos:", error);
  });
```

Enviando Info a la API

En este ejemplo mediante los respectivos inputs actualizamos el estado de cada uno de los campos para luego generar ese formData que va a ser enviado por POST al endpoint de nuestro servidor que recibe toda la información para crear un nuevo producto y a su vez procesa la imagen para guardarla de forma local en nuestro proyecto.

```
<label className="block mb-2">Quantity:</label>
<input
  type="number"
  value={quantity}
  onChange={(e) => setQuantity(e.target.value)}
/>
```

```
const handleSubmit = async (e) => {
  e.preventDefault();

  const formData = new FormData();
  formData.append("name", name);
  formData.append("size", size);
  formData.append("quantity", quantity);
  formData.append("image", image);

  fetch(import.meta.env.VITE_URL_API, {
    method: "POST",
    body: formData,
  })
    .then((response) => {
      if (response.ok) {
        console.log("Shoe created successfully");
        setCanGetShoesToTrue();
      } else {
        console.error("Failed to create shoe");
      }
    })
    .catch((error) => {
      console.error("Error creating shoe:", error);
    });
};
```

Muchas Gracias