

ESCUELA POLITÉCNICA NACIONAL
CONSTRUCCIÓN Y EVOLUCIÓN DE SOFTWARE
PROYECTO SEGUNDO BIMESTRE

Integrantes:

- Averos David
- Campaña Gary
- Torres David

Documentación del Proyecto “Registro de Acciones”

Descripción del proyecto:

El proyecto consiste en la creación de una aplicación web que permita al usuario poder realizar las operaciones basicas "CRUD" en un registro de Acciones, donde se puede visualizar, crear, editar y eliminar las acciones que el usuario desee.

TECNOLOGÍAS IMPLEMENTADAS:

Framework Para El FRONT-END

ANGULAR

- Angular CLI es una herramienta de desarrollo de código abierto para facilitar la creación de aplicaciones Angular, así como para realizar múltiples tareas relacionadas con el desarrollo (como pruebas, empaquetado y despliegue). Angular CLI es una herramienta de línea de comandos que puede usar para inicializar, desarrollar, mantener y solucionar problemas aplicaciones Angular. También puede usar Angular CLI para ejecutar pruebas unitarias y de extremo a extremo, agregar características y desplegar aplicaciones.
- [Angular CLI](#)

Bootstrap 4

- Bootstrap 4 es la última versión del framework de diseño más popular del mundo. Bootstrap es un framework de código abierto que facilita el diseño web y el desarrollo de aplicaciones. Contiene plantillas de diseño basadas en HTML y CSS para tipografía, formularios, botones, tablas, navegación, modales, carruseles de imágenes y muchos otros componentes, así como complementos de JavaScript opcionales. Bootstrap también le da la capacidad de crear diseños web receptivos con mucho menos esfuerzo.

Framework Para El BACK-END

SPRING BOOT

- Springboot es un framework que permite crear aplicaciones de manera rápida y sencilla, con el objetivo de reducir la complejidad de la configuración y el despliegue de aplicaciones Spring. Spring Boot es una herramienta que permite crear aplicaciones de manera rápida y sencilla, con el objetivo de reducir la complejidad de la configuración y el despliegue de aplicaciones Spring. Spring Boot es una herramienta que permite crear aplicaciones de manera rápida y sencilla, con

el objetivo de reducir la complejidad de la configuración y el despliegue de aplicaciones Spring.

- [Spring Boot](#)

PERSISTENCIA (BASE DE DATOS)

Motor XAMPP

- XAMPP es un paquete de software libre, que consiste principalmente en el sistema de gestión de bases de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script: PHP y Perl. El nombre proviene del acrónimo de X (para cualquiera de los diferentes sistemas operativos), Apache, MySQL, PHP, Perl. El programa es liberado bajo la licencia GNU y actúa como un servidor web libre, fácil de usar y capaz de interpretar páginas dinámicas.

PHPMYADMIN

- phpMyAdmin es una herramienta de software libre escrita en PHP, destinada a manejar la administración de MySQL a través de la Web. phpMyAdmin admite una amplia gama de operaciones con MySQL. Las operaciones más frecuentes son compatibles, y es posible administrar bases de datos, tablas, columnas, relaciones, índices, usuarios, permisos, etc. Además, phpMyAdmin es capaz de manejar múltiples servidores MySQL.

MYSQL

- MYSQL es un sistema de gestión de bases de datos relacional, que permite la creación de bases de datos, tablas, procedimientos almacenados, funciones, entre otros. MYSQL es un sistema de gestión de bases de datos relacional, que permite la creación de bases de datos, tablas, procedimientos almacenados, funciones, entre otros. MYSQL es un sistema de gestión de bases de datos relacional, que permite la creación de bases de datos, tablas, procedimientos almacenados, funciones, entre otros.
- [MYSQL](#)

Descarga del Proyecto

Pasos de instalación

1. Clonar el repositorio:

git clone https://github.com/SrKarma07/Registro_De_Acciones.git

Definición de la Metodología Implementada para la realización del proyecto

La metodología XP (Extreme Programming) se implementó en el proyecto

"Registro_De_Acciones" siguiendo sus principios fundamentales de comunicación, simplicidad, retroalimentación, valentía y respeto, adaptándose a un equipo de tres personas. A continuación, se detalla cómo se aplicaron los roles y actividades dentro de este marco:

Roles y Actividades Definidas:

1. **Coach, Tester y Programador 1 (Desarrollador Front-end)**

- **Rol:** Responsable del desarrollo de la interfaz de usuario, asegurando que sea intuitiva y cumpla con los requisitos funcionales.
- **Actividades:**
 - Diseño de UI/UX utilizando herramientas como Angular.
 - Implementación de llamadas al backend mediante servicios REST.
 - Pruebas unitarias y de interfaz para garantizar la usabilidad y funcionalidad.

2. **Cliente y Programador 2 (Desarrollador Back-end)**

- **Rol:** Encargado de la lógica de negocio, la base de datos y la API que conecta el frontend con el backend.
- **Actividades:**
 - Diseño y normalización de la base de datos en MySQL.
 - Desarrollo de endpoints REST en Spring Boot.
 - Implementación de pruebas unitarias y de integración para los servicios desarrollados.

3. **Tracker, BigBoss y Programador 3 (Tester/Integrador)**

- **Rol:** Asegura la calidad del software mediante pruebas continuas y gestiona la integración de los módulos desarrollados por el equipo.
- **Actividades:**
 - Desarrollo y ejecución de planes de pruebas exhaustivas.
 - Coordinación de sesiones de programación en pareja para identificar y corregir errores.
 - Gestión de la integración continua y despliegue del software.

Implementación de la Metodología XP:

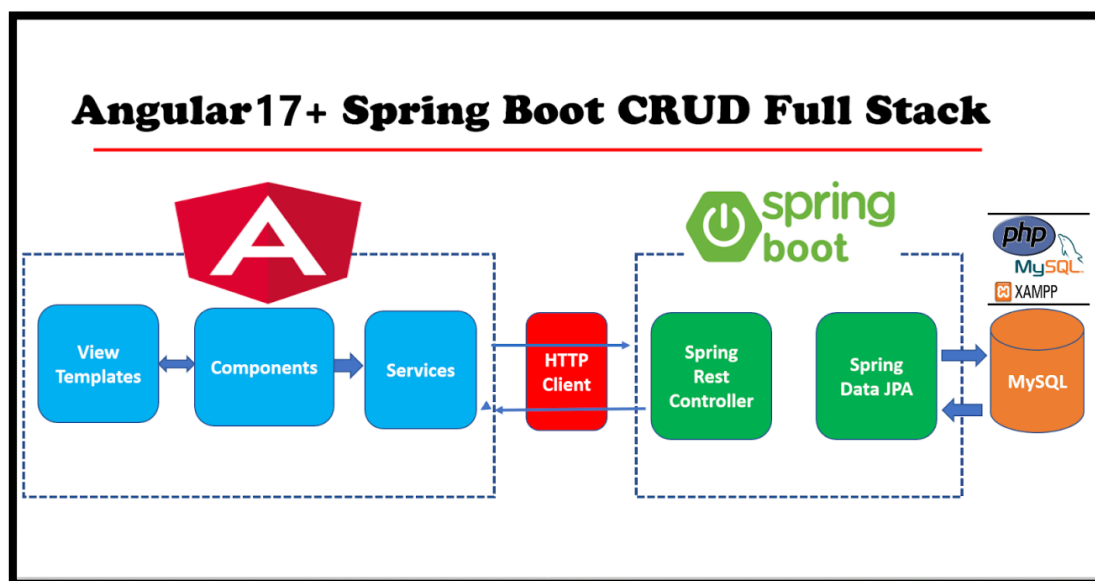
- **Planificación:** El equipo comenzó con una reunión de planificación para definir historias de usuario basadas en los requisitos del proyecto. Se estimaron tareas y se asignaron roles específicos considerando la experiencia y habilidades de cada miembro.
- **Diseño Simple:** Se optó por un diseño que satisfacía los requisitos actuales sin sobrecomplicar el sistema, dejando espacio para futuras extensiones y mantenimiento.
- **Desarrollo Iterativo y Entregas Frecuentes:** El proyecto se dividió en iteraciones cortas, al final de las cuales se entregaba un incremento funcional del software. Esto permitió obtener retroalimentación temprana y continua del cliente.

- **Programación en Parejas:** Aunque el equipo se dividió principalmente en frontend y backend, se organizaron sesiones donde los miembros programaban en parejas, rotando los roles. Esto mejoró la calidad del código y facilitó la transferencia de conocimientos dentro del equipo.
- **Integración Continua:** Desde el inicio, se estableció un flujo de trabajo que permitía integrar y probar el software frecuentemente. Cada integración era una oportunidad para detectar y resolver conflictos tempranamente.
- **Refactorización:** El equipo se comprometió a mejorar continuamente la calidad del código, realizando refactorizaciones regulares para optimizar el diseño y la implementación sin alterar la funcionalidad.
- **Propiedad Colectiva del Código:** Todos los miembros del equipo eran responsables del código, lo que fomentó un enfoque colaborativo y redujo los cuellos de botella en el desarrollo.
- **Retroalimentación del Cliente:** Se mantuvo una comunicación constante con el cliente para asegurar que el producto final cumpliera con sus expectativas, ajustando el rumbo del proyecto según fuera necesario.

La implementación de XP en el proyecto "Registro_De_Acciones" permitió al equipo trabajar de manera eficiente y adaptativa, manteniendo un alto nivel de calidad del software y satisfaciendo las necesidades del cliente.

Definición de la Arquitectura del proyecto Registro_De_Acciones

El proyecto "Registro_De_Acciones" implementó una arquitectura MVC (Modelo-Vista-Controlador), que es un patrón de diseño utilizado para desarrollar interfaces de usuario de manera que se separe la lógica de negocio de la interfaz de usuario. Este enfoque facilita la gestión del código, su mantenimiento y la escalabilidad del proyecto. A continuación, se detallan los componentes de esta arquitectura y cómo se implementaron utilizando Angular CLI y Bootstrap para el frontend, Spring Boot para el backend, y MySQL con XAMPP para la base de datos.



Frontend (Vista)

- **Tecnologías Utilizadas:** Angular CLI y Bootstrap.
- **Implementación:**
 - **Angular CLI:** Se utilizó para estructurar la aplicación siguiendo el patrón MVC, donde Angular actúa principalmente como la capa de "Vista" pero también facilita la integración del "Controlador" a través de sus componentes y servicios.
 - **Bootstrap:** Proporcionó los estilos y componentes UI para crear una interfaz de usuario atractiva y responsiva. Se usaron modales, botones, formularios y otros elementos de Bootstrap para mejorar la experiencia de usuario.
 - La comunicación con el backend se realizó a través de servicios Angular que hacían peticiones HTTP, manejando la lógica de presentación y la interacción con el usuario.

Backend (Controlador y Modelo)

- **Tecnología Utilizada:** Spring Boot.
- **Implementación:**
 - **Controlador:** Se implementaron controladores REST utilizando Spring Boot, que actuaron como intermediarios entre la vista (frontend) y el modelo (base de datos). Estos controladores procesaban las solicitudes HTTP, ejecutaban la lógica de negocio necesaria y devolvían las respuestas adecuadas al cliente.
 - **Modelo:** Se definieron entidades JPA que representaban las tablas de la base de datos, permitiendo a Spring Boot interactuar con la base de datos MySQL de manera eficiente a través del ORM (Object-Relational Mapping).
 - Se utilizó la inyección de dependencias de Spring para manejar las instancias de los objetos necesarios, lo que facilitó la desacoplación y la reutilización del código.

Base de Datos (Persistencia)

- **Tecnología Utilizada:** MySQL con XAMPP.
- **Implementación:**
 - Se utilizó XAMPP como entorno de desarrollo para manejar el servidor Apache y el servidor de base de datos MySQL.
 - Se crearon esquemas y tablas en MySQL para almacenar la información de las acciones, incluyendo datos como el nombre de la acción, fecha de compra, cantidad, entre otros.
 - La configuración de la conexión a la base de datos se realizó en el archivo **application.properties** de Spring Boot, especificando la URL de la base de datos, el nombre de usuario y la contraseña.

Integración de la Arquitectura MVC

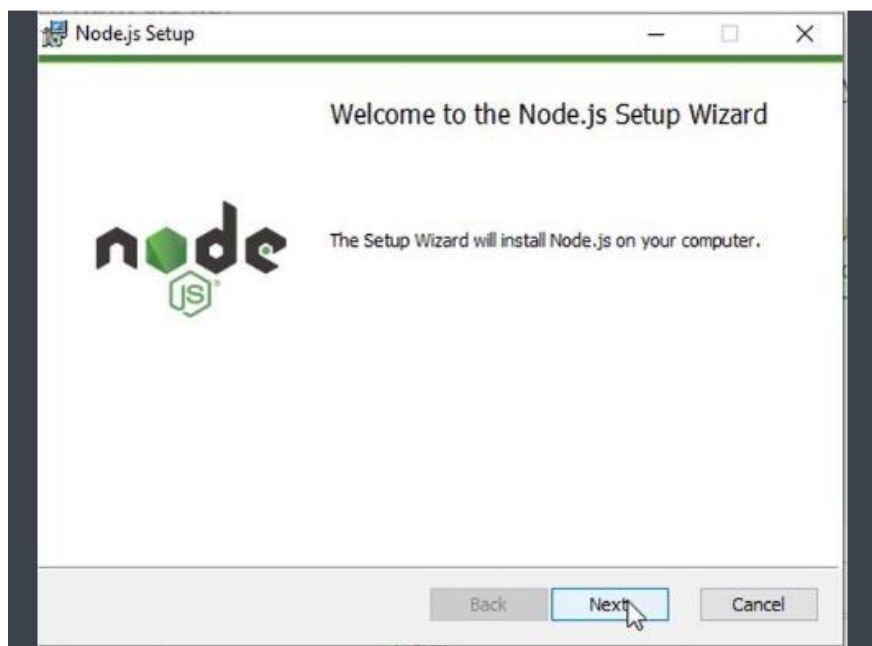
- La integración entre la vista (Angular), el controlador y el modelo (Spring Boot) se realizó a través de API REST, permitiendo una comunicación asíncrona y un acoplamiento débil entre el frontend y el backend.
- Angular envió solicitudes HTTP al backend, que a su vez interactuó con la base de datos para realizar operaciones CRUD (crear, leer, actualizar, eliminar). Los resultados se enviaban de vuelta al frontend para ser presentados al usuario.
- Se implementaron prácticas de seguridad y manejo de errores tanto en el frontend como en el backend para asegurar la integridad y la confiabilidad de la aplicación.

Esta arquitectura MVC permitió al equipo desarrollar, probar y mantener cada parte de la aplicación de manera independiente, mejorando la eficiencia del desarrollo y la calidad del software en el proyecto "Registro_De_Acciones".

FRONT END – Angular CLI - NodeJs

Instalación de NodeJs

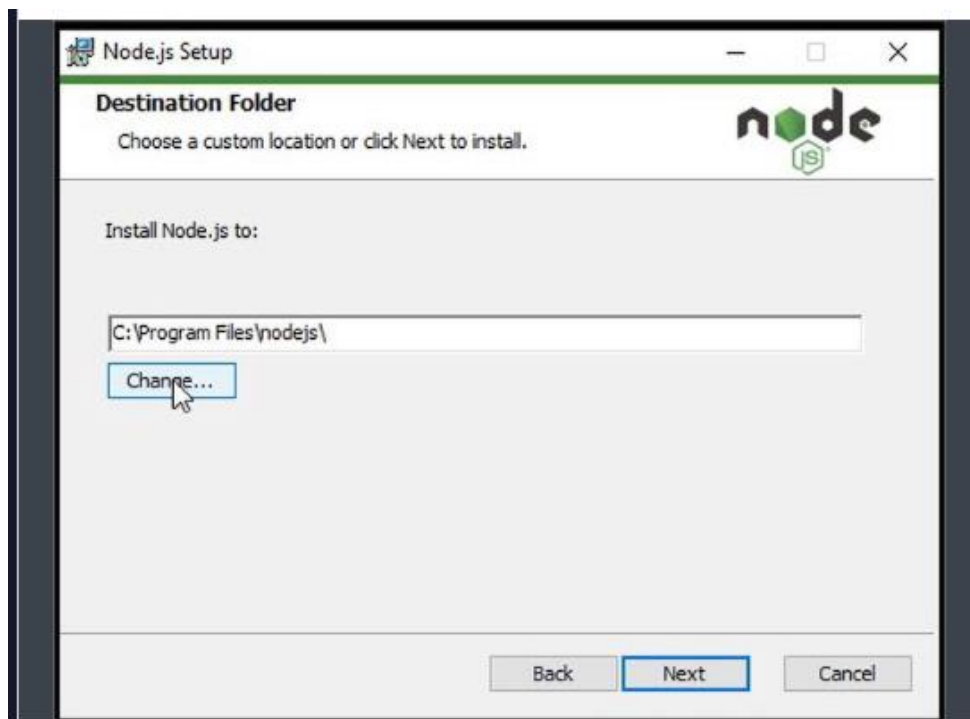
Para iniciar la instalación y ejecutar el asistente de instalación, únicamente debes abrir el archivo recién descargado con extensión .msi. Este archivo abrirá una ventana como la que ves a continuación:



Esta es la pantalla de bienvenida, donde aparece un botón con la leyenda "Next" que debes seleccionar para iniciar el proceso de instalación, luego de eso, verás la pantalla para aceptar la licencia de uso de Node.js, en donde deberás activar la casilla con la leyenda "I accept the terms in the License Agreement" si aceptas los terminos y condiciones que aparecen en la misma ventana. Tal como se muestra a continuación:

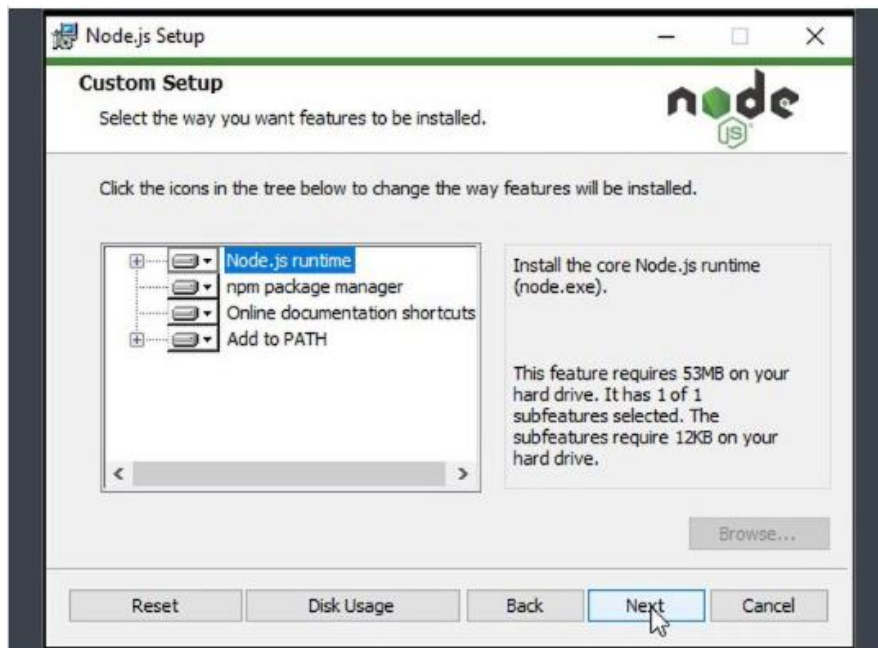


Luego de haber aceptado los términos y condiciones, selecciona el botón con la leyenda "Next" para avanzar con la instalación. En la siguiente imagen, el asistente te permite seleccionar la ruta en tu computadora donde quieres que se instalen los archivos de Node.js, si no tienes una preferencia para modificar esta ruta, te recomiendo que dejes la recomendada.

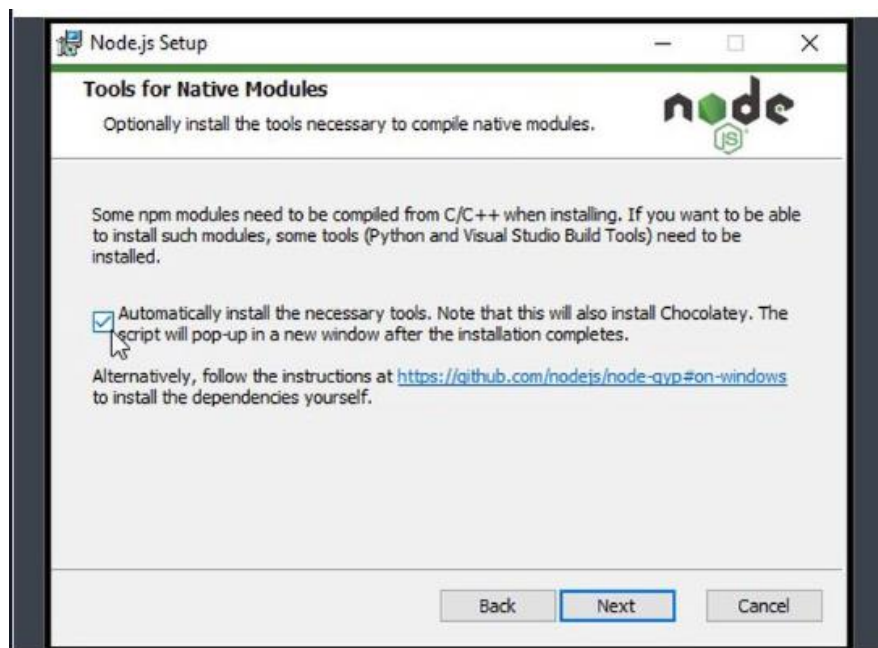


Luego de seleccionar la ruta, o si deseas dejar la que aparece por defecto, presiona el botón con la leyenda "Next". La siguiente pantalla te permite configurar la instalación de Node.js, a menos de que tengas experiencia con Node.js y sepas específicamente qué cosas

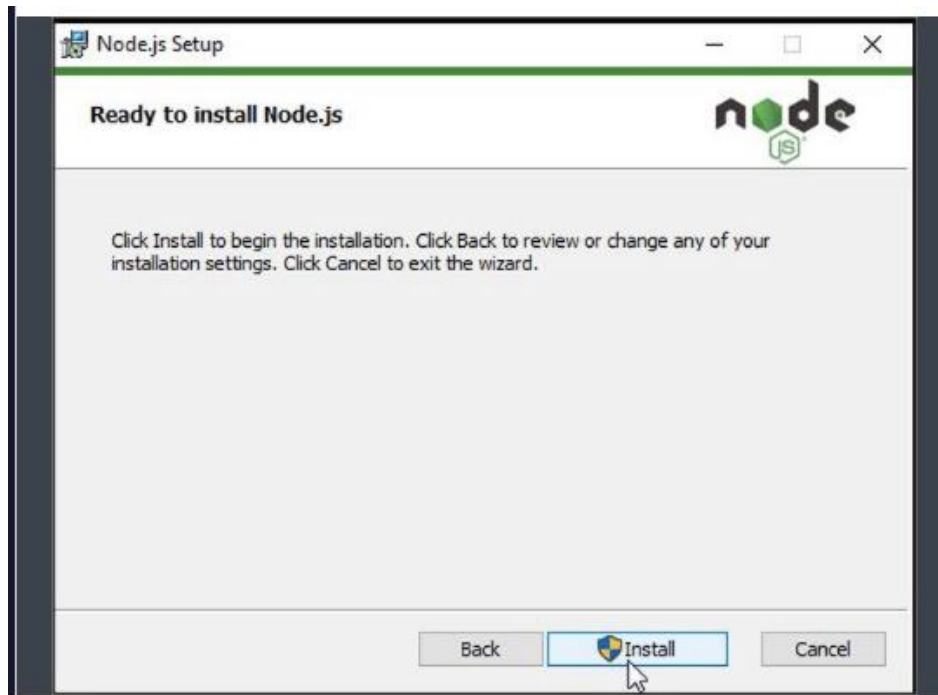
necesitas y qué cosas no, te recomiendo dejar la instalación tal y como está, sin hacer configuraciones adicionales:



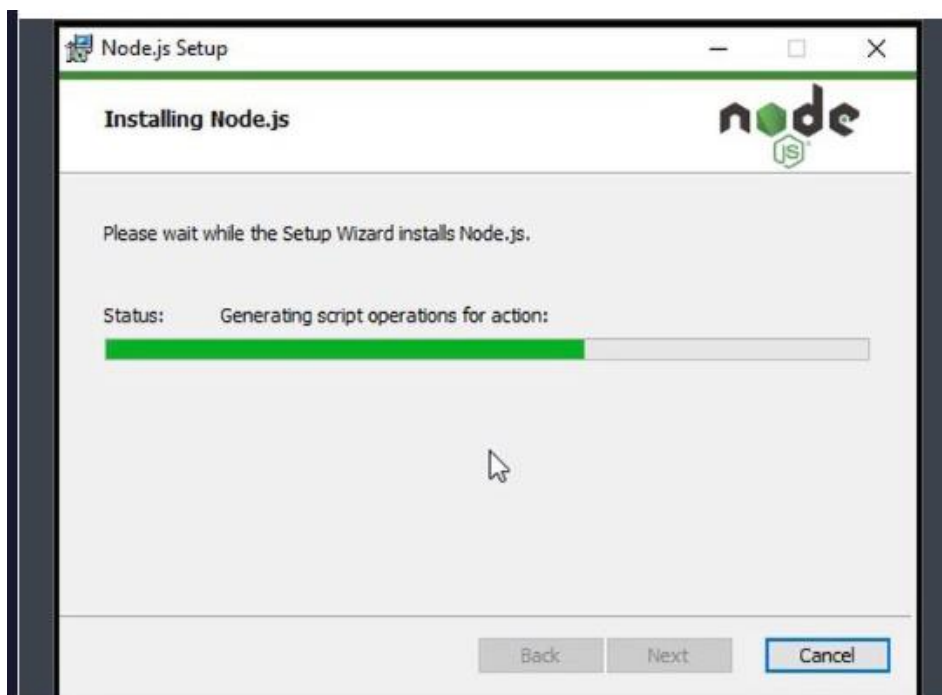
Asegúrate de presionar el botón con la leyenda "Next" para continuar. La siguiente pantalla solicita tu consentimiento para instalar herramientas adicionales a Node.js que son necesarias para que Node.js funcione, incluido el manejador de paquetes Chocolatey que puede servirte para instalar otras dependencias de desarrollo en tu computadora.



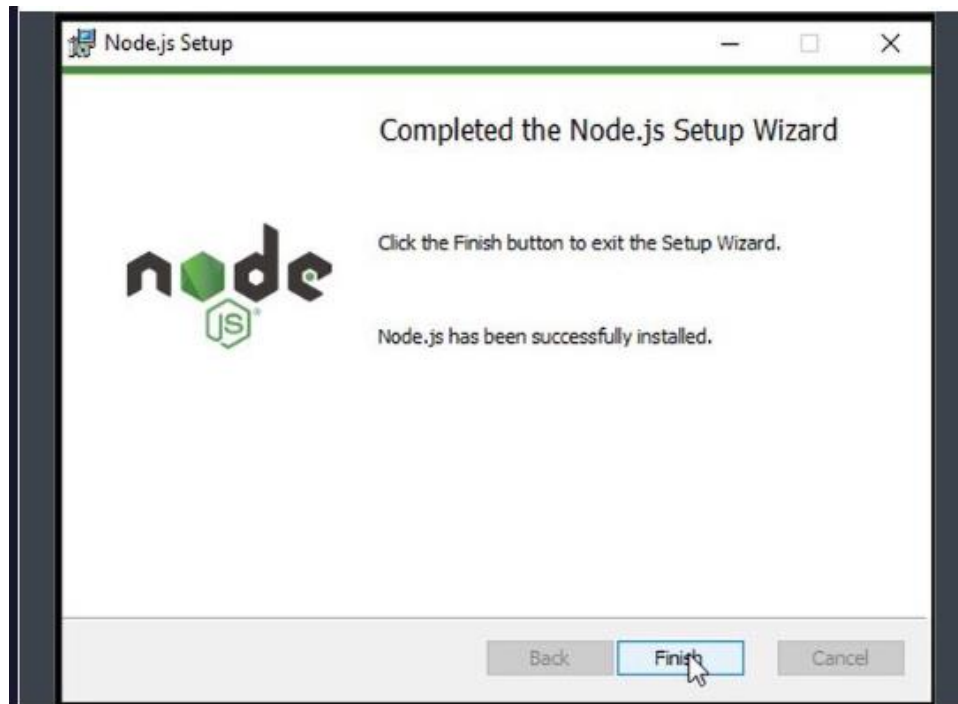
Para la última pantalla únicamente debemos presionar el botón con la leyenda "Install" para iniciar el proceso de instalación. Ten en cuenta que este botón puede solicitar que habilites permisos de Administrador.



La siguiente pantalla que verás contiene una barra de progreso con el status de la instalación, únicamente debes esperar a que esta barra se complete:



Por último verás una pantalla notificando que la instalación se ha completado exitosamente. Si la instalación no se ha completado exitosamente, deja un comentario en la sección de dudas. Haz clic en el botón con la leyenda "Finish" para continuar.



Para poder visualizar la version de NodeJs instalado en nuestro ordenador, escribimos el siguiente comando en la terminal CMD:

```
C:\> node -v
v20.11.1
```

En este caso, la version de NodeJs con la que trabajaremos será la v20.11.1

Una vez instalado NodeJs en nuestro ordenador procederemos con la verificación de la instalación de npm con el siguiente comando.

```
C:\> node -v
v20.11.1

C:\> npm --version
10.5.0

C:\>
```

En este caso, la version de NodeJs con la que trabajaremos será la v10.5.0

Instalación de Angular CLI

Para instalar Angular en tu sistema local, necesitas lo siguiente:

- Node.js

Angular requiere una versión [actual](#), [LTS activa](#) o [LTS de mantenimiento](#) de Node.js.

Utilizarás la CLI de Angular para crear proyectos, generar código de aplicaciones y bibliotecas, y realizar una variedad de tareas de desarrollo, como pruebas, agrupación e implementación.

Para instalar CLI de Angular, abre una terminal y ejecuta el siguiente comando:

```
npm install -g @angular/cli
```

```
C:\>ng version

Angular CLI
Angular CLI: 17.2.2
Node: 20.11.1
Package Manager: npm 10.5.0
OS: win32 x64

Angular:
...

Package                                Version
-----                                -
@angular-devkit/architect              0.1702.2 (cli-only)
@angular-devkit/core                   17.2.2 (cli-only)
@angular-devkit/schematics             17.2.2 (cli-only)
@schematics/angular                   17.2.2 (cli-only)
```

Creación del espacio de trabajo y nuestra aplicación inicial.

Para crear un nuevo espacio de trabajo y una aplicación inicial:

1. Ejecuta el comando CLI `ng new` y proporciona el nombre `my-app`, como se muestra aquí:

```
ng new my-app
```

2. El comando `ng new` te solicitará información sobre las funciones que debe incluir en la aplicación inicial. Acepta los valores predeterminados presionando la tecla `Enter` o `Return`.

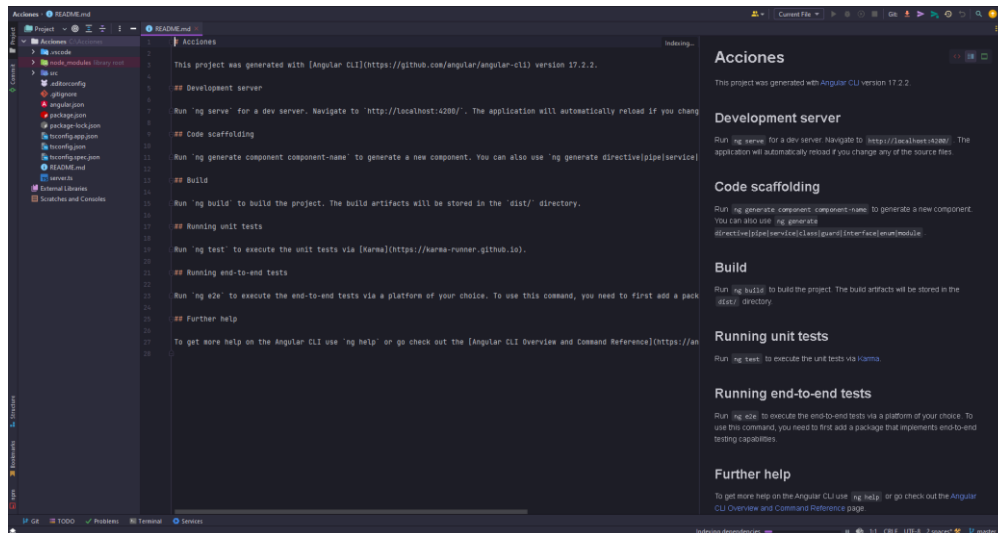
La CLI de Angular instala los paquetes npm de Angular necesarios y otras dependencias. Esto puede tardar unos minutos.

La CLI crea un nuevo espacio de trabajo y una aplicación de bienvenida simple, lista para ejecutarse.

En la Terminal CMD nos ubicamos en la carpeta donde se desea crear el proyecto

```
npm install
C:\>ng new Acciones
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? Yes
CREATE Acciones/angular.json (2848 bytes)
CREATE Acciones/package.json (1269 bytes)
CREATE Acciones/README.md (1089 bytes)
CREATE Acciones/tsconfig.json (936 bytes)
CREATE Acciones/.editorconfig (290 bytes)
CREATE Acciones/.gitignore (590 bytes)
CREATE Acciones/tsconfig.app.json (342 bytes)
CREATE Acciones/tsconfig.spec.json (287 bytes)
CREATE Acciones/server.ts (1759 bytes)
CREATE Acciones/.vscode/extensions.json (134 bytes)
CREATE Acciones/.vscode/launch.json (490 bytes)
CREATE Acciones/.vscode/tasks.json (980 bytes)
CREATE Acciones/src/main.ts (256 bytes)
CREATE Acciones/src/favicon.ico (15086 bytes)
CREATE Acciones/src/index.html (307 bytes)
CREATE Acciones/src/styles.css (81 bytes)
CREATE Acciones/src/main.server.ts (271 bytes)
CREATE Acciones/src/app/app.component.html (20239 bytes)
CREATE Acciones/src/app/app.component.spec.ts (951 bytes)
CREATE Acciones/src/app/app.component.ts (317 bytes)
CREATE Acciones/src/app/app.component.css (0 bytes)
CREATE Acciones/src/app/app.config.ts (330 bytes)
CREATE Acciones/src/app/app.routes.ts (80 bytes)
CREATE Acciones/src/app/app.config.server.ts (361 bytes)
CREATE Acciones/src/assets/.gitkeep (0 bytes)
\ Installing packages (npm)...
```

Proyecto Abierto en el editor de código WEBSTORM.



Ejecutamos el Proyecto con el comando ng serve en la terminal dentro de la carpeta donde se encuentra el Proyecto

```
Server bundles
Initial chunk files | Names | Raw size
chunk-CIHI3CEQ.mjs | - | 1.69 MB |
polyfills.server.mjs | polyfills.server | 555.05 kB |
main.server.mjs | main.server | 214.75 kB |
chunk-VPSODEBW.mjs | - | 2.51 kB |
render-utils.server.mjs | render-utils.server | 423 bytes |

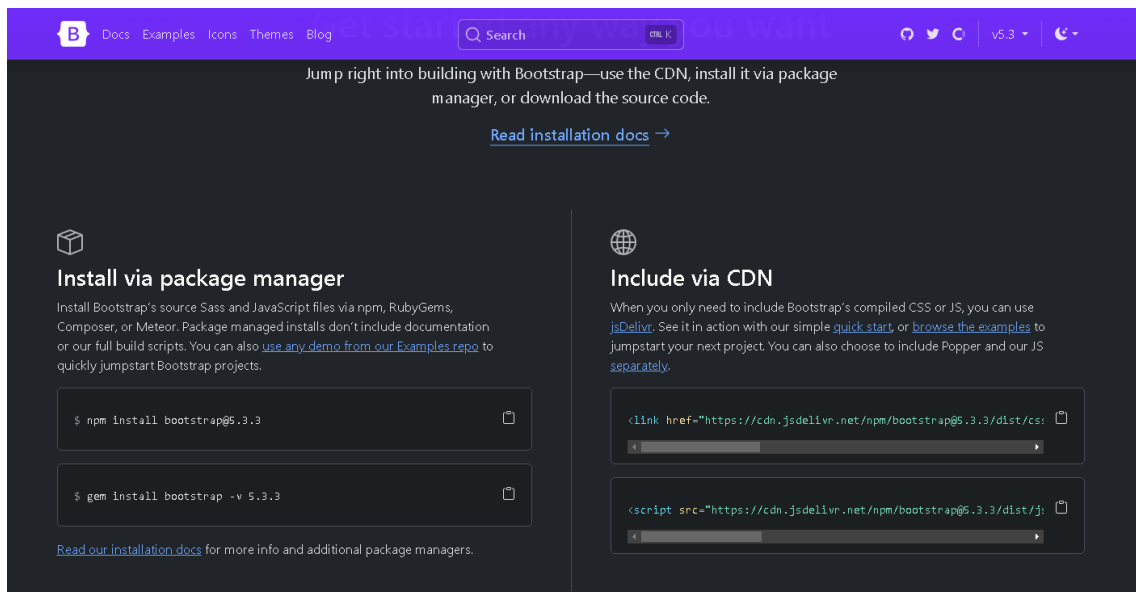
Lazy chunk files | Names | Raw size
chunk-OTT6LQ5K.mjs | xhr2 | 39.10 kB |
Application bundle generation complete. [8.315 seconds]
Watch mode enabled. Watching for file changes...
→ Local: http://localhost:4200/
→ press h + enter to show help
```

En este caso, nuestro servidor integrado nos asigna el puerto 4200 del localhost para poder acceder a nuestro proyecto desde un navegador web.

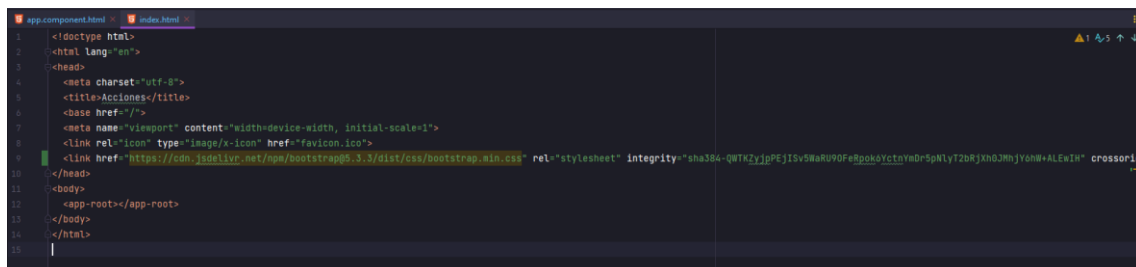
Este es el Proyecto visto desde un navegador



Añadiremos a nuestro proyecto via CDN las librerías de BOOTSTRAP, para ello nos dirigimos a la página oficial de Bootstrap y copiamos el link del cdn.



Una vez hecho esto lo agregaremos en el fichero 'index.html' de nuestro proyecto



Para crear los componentes que necesitamos para nuestra hoja de acciones, en la terminal ejecutaremos los comandos que se ven en la imagen

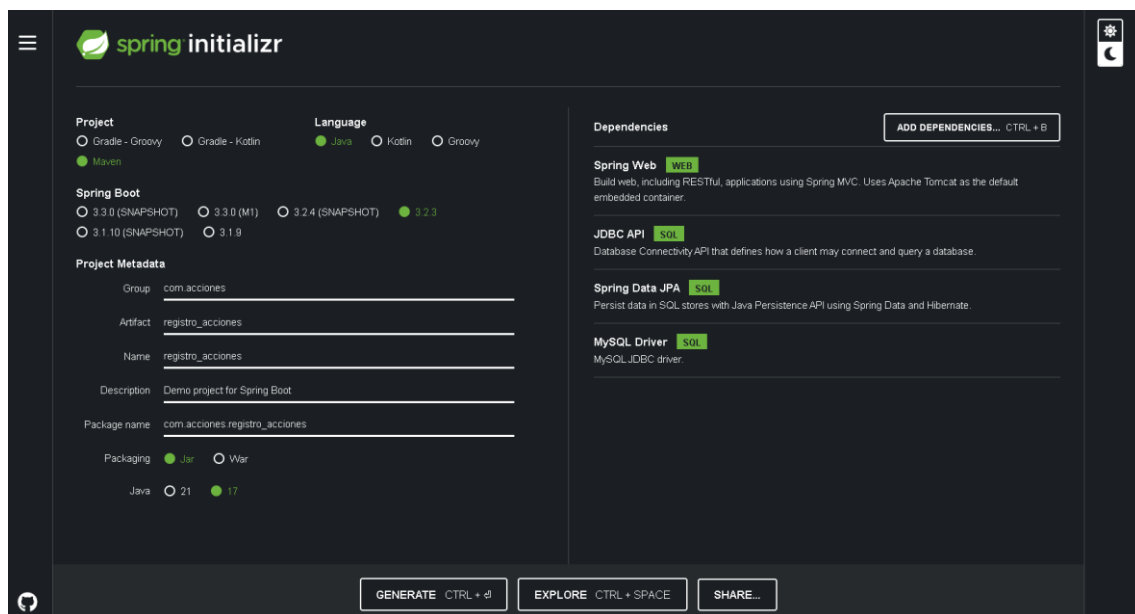
```
Terminal: Local x + ↗
PS C:\Acciones> ng g c Accion/listar
CREATE src/app/Accion/listar/listar.component.html (22 bytes)
CREATE src/app/Accion/listar/listar.component.spec.ts (619 bytes)
CREATE src/app/Accion/listar/listar.component.ts (246 bytes)
CREATE src/app/Accion/listar/listar.component.css (0 bytes)
PS C:\Acciones> 
```

Una vez hecho tendremos listo nuestro entorno de trabajo con Angular CLI.

BACK END – SpringBoot

Para nuestro entorno de trabajo de backend hemos seleccionado el framework de SpringBoot, para poder empezar a trabajar en este entorno nos dirigiremos a la pagina de SpringInitializr para poder crear nuestro proyecto tipo Maven con las especificaciones y librerías necesarias para poder trabajar.

En esta imagen podemos ver todas las configuraciones de nuestro proyecto Maven para el backend, una vez seleccionado todo esto, procedemos a descargar el proyecto y abrirlo en el editor de código de JAVA “Apache Netbeans”.



Una vez abierto nuestro proyecto en NetBeans podemos empezar a trabajar.

Documentación de la Aplicación Web Registro de Acciones

Codificación y Configuración del BackEnd – SpringBoot

Clase Acción.java (Model)

Código

```
package com.acciones.registro_acciones.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
```

```
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "acciones")
public class Accion {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idAccion")
    private int idAccion;

    @Column(name = "NombreAccion")
    private String nombreAccion;

    @Column(name = "FechaCompra")
    private String fechaCompra;

    @Column(name = "PrecioCompraAccion")
    private int precioCompraAccion;

    @Column(name = "CantidadAcciones")
    private int cantidadAcciones;

    @Column(name = "CostoTotalCompra")
    private int costoTotalCompra;

    public int getPrecioCompraAccion() {
        return precioCompraAccion;
    }

    public int getIdAccion() {
        return idAccion;
    }

    public void setIdAccion(int idAccion) {
        this.idAccion = idAccion;
    }

    public String getNombreAccion() {
        return nombreAccion;
    }

    public void setNombreAccion(String nombreAccion) {
        this.nombreAccion = nombreAccion;
    }

    public String getFechaCompra() {
        return fechaCompra;
    }
}
```

```

    }

    public void setFechaCompra(String fechaCompra) {
        this.fechaCompra = fechaCompra;
    }

    public int getCantidadAcciones() {
        return cantidadAcciones;
    }

    public void setCantidadAcciones(int cantidadAcciones) {
        this.cantidadAcciones = cantidadAcciones;
    }

    public int getCostoTotalCompra() {
        return costoTotalCompra;
    }

    public void setCostoTotalCompra(int costoTotalCompra) {
        this.costoTotalCompra = costoTotalCompra;
    }
}

```

Documentación de Accion.java (Clase Modelo)

La clase **Accion** es un modelo dentro de un proyecto Spring Boot Maven que representa una transacción de acciones en el mercado de valores. Está anotada con anotaciones de JPA (Java Persistence API) para mapearla a una tabla de base de datos llamada "acciones". Esta clase encapsula los datos y la lógica de negocio de las transacciones de acciones.

Anotaciones de Clase:

- **@Entity**: Especifica que la clase es una entidad. Esta anotación marca la clase como una clase Java persistente.
- **@Table(name = "acciones")**: Especifica la tabla en la base de datos con la que esta entidad está mapeada. El nombre de la tabla se define como "acciones".

Campos:

- **idAccion**: El identificador único para cada acción. Está marcado con **@Id** para denotarlo como una clave primaria y **@GeneratedValue** para especificar que este campo se autoincrementa.
- **nombreAccion**: Representa el nombre de la acción o stock. Se almacena en la columna "NombreAccion" de la base de datos.
- **fechaCompra**: Almacena la fecha de compra de la acción. Es un string que representa la fecha en que se compró la acción.
- **precioCompraAccion**: Representa el precio de compra por acción en el momento de la compra.
- **cantidadAcciones**: Indica la cantidad de acciones compradas en esta transacción.

- **costoTotalCompra:** Representa el costo total de la compra, calculado a partir del precio de compra por acción y la cantidad de acciones.

Métodos:

- Métodos **get** y **set** para cada campo: Estos métodos permiten acceder y modificar los valores de los campos de la entidad. Proporcionan una interfaz para interactuar con los atributos de la entidad de manera segura.

Esta clase es fundamental para el manejo de datos relacionados con las transacciones de acciones dentro de la aplicación, permitiendo realizar operaciones de persistencia como creación, lectura, actualización y eliminación de registros de acciones en la base de datos.

Clase AccionServicio.java

Código

```
package com.acciones.registro_acciones;

import com.acciones.registro_acciones.model.Accion;
import java.util.List;

public interface AccionServicio {
    List<Accion>listar();
    Accion listarId(int idAccion);
    Accion agregar(Accion p);
    Accion editar(Accion p);
    Accion eliminar(int idAccion);
}
```

Documentación de AccionServicio.java (Interfaz de Servicio)

La interfaz **AccionServicio** define los métodos esenciales para el manejo de operaciones relacionadas con las acciones en el mercado de valores dentro de un proyecto Spring Boot Maven. Esta interfaz establece el contrato que los servicios de acciones deben implementar, proporcionando una abstracción para operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y otras lógicas de negocio asociadas con las acciones.

Métodos Definidos:

- **List<Accion> listar():** Devuelve una lista de todas las acciones disponibles en la base de datos. Este método es utilizado para obtener un conjunto completo de registros de acciones para su visualización o procesamiento adicional.
- **Accion listarId(int idAccion):** Busca y devuelve una acción específica por su identificador único (**idAccion**). Este método es útil para operaciones que requieren manipulación o visualización de detalles de una acción específica.
- **Accion agregar(Accion p):** Añade una nueva acción a la base de datos y devuelve la acción agregada. Este método toma una instancia de **Accion** como parámetro, la persiste en la base de datos y luego retorna la entidad persistida, lo que es útil para confirmar la inclusión y obtener cualquier dato generado automáticamente, como el ID.

- **Accion editar(Accion p):** Actualiza los detalles de una acción existente en la base de datos. Este método acepta una instancia de **Accion** modificada y aplica los cambios en la base de datos, asegurando que la información de la acción esté actualizada.
- **Accion eliminar(int idAccion):** Elimina una acción de la base de datos basándose en su identificador. Este método es crucial para remover registros de acciones que ya no son necesarios o relevantes.

Uso:

La interfaz **AccionServicio** es un componente clave en la arquitectura de la aplicación, facilitando la separación de las operaciones de negocio de la lógica de controladores y otros componentes. Al definir una interfaz de servicio, se promueve la desacoplamiento y la flexibilidad en el diseño del sistema, permitiendo, por ejemplo, la implementación de diferentes versiones del servicio sin alterar el resto de la aplicación.

Esta interfaz debe ser implementada por una clase de servicio concreta (**AccionServicioImp**), la cual proporcionará la lógica de negocio específica para interactuar con la base de datos a través del repositorio de acciones.

Clase AccionServicioImp.java

Código

```
package com.acciones.registro_acciones;

import com.acciones.registro_acciones.model.Accion;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AccionServicioImp implements AccionServicio{
    @Autowired
    private AccionRepositorio repositorio;

    @Override
    public List<Accion> listar() {
        return (List<Accion>) repositorio.findAll();
    }

    @Override
    public Accion listarId(int idAccion) {
        return repositorio.findById(idAccion).orElseThrow(() -> new
        RuntimeException("Accion no encontrada"));
    }

    @Override
    public Accion agregar(Accion p) {
        return repositorio.save(p);
    }
}
```

```
@Override
public Accion editar(Accion p) {
    return repositorio.save(p);
}

@Override
public Accion eliminar(int id) {
    return repositorio.findById(id).map(p -> {
        repositorio.delete(p);
        return p;
    }).orElseThrow(() -> new RuntimeException("Accion no encontrada con id: " + id));
}
}
```

Documentación de AccionServicioImp.java (Implementación de Servicio)

La clase **AccionServicioImp** implementa la interfaz **AccionServicio**, proporcionando una implementación concreta de los métodos definidos para interactuar con la base de datos de acciones. Utiliza el patrón de diseño de inyección de dependencias para acceder al repositorio asociado con las entidades de acción. Esta clase es anotada con **@Service** para indicar que es un componente de servicio en la arquitectura de Spring Boot.

Atributos:

- **@Autowired private AccionRepositorio repositorio:** Inyección automática de la instancia del repositorio de acciones, que permite realizar operaciones CRUD en la base de datos.

Métodos Implementados:

- **List<Accion> listar():** Retorna todas las acciones presentes en la base de datos, haciendo uso del método **findAll()** proporcionado por el repositorio de Spring Data JPA.
- **Accion listarId(int idAccion):** Busca una acción específica por su ID. Si la acción no se encuentra, se lanza una excepción de tiempo de ejecución.
- **Accion agregar(Accion p):** Guarda una nueva acción en la base de datos y retorna la entidad guardada, permitiendo el acceso a cualquier dato generado durante la persistencia, como el ID autogenerado.
- **Accion editar(Accion p):** Actualiza una acción existente en la base de datos. Si la acción ya existe (basado en su ID), se actualizan sus datos; de lo contrario, se crea una nueva entrada.
- **Accion eliminar(int id):** Elimina una acción por su ID. Si la acción existe, se elimina; si no, se lanza una excepción indicando que la acción no se encontró.

Funcionalidad y Uso:

Esta clase sirve como intermediario entre el controlador y el repositorio, encapsulando la lógica de negocio y las operaciones de base de datos para las entidades de acción. Al

separar las responsabilidades de esta manera, se mejora la organización del código y se facilita su mantenimiento y escalabilidad.

La anotación **@Service** permite que Spring reconozca automáticamente esta clase como un bean y la gestione dentro de su contenedor de inversión de control (IoC), facilitando la inyección de dependencias donde sea necesario.

Clase **AccionRepositorio.java** (interface)

Código
<pre>package com.acciones.registro_acciones; import com.acciones.registro_acciones.model.Accion; import java.util.List; import java.util.Optional; import org.springframework.data.jpa.repository.JpaRepository; public interface AccionRepositorio extends JpaRepository<Accion, Integer>{ List<Accion> findAll(); Optional<Accion> findById(Integer idAccion); Accion save(Accion p); void delete(Accion p); }</pre>

Documentación de **AccionRepositorio.java** (Interfaz de Repositorio)

La interfaz **AccionRepositorio** extiende **JpaRepository**, proporcionando una serie de métodos para realizar operaciones CRUD sobre la entidad **Accion**. Esta interfaz facilita la interacción con la base de datos para la gestión de acciones, aprovechando las funcionalidades proporcionadas por Spring Data JPA.

Métodos Heredados de **JpaRepository**:

- **List<Accion> findAll():** Obtiene todas las acciones almacenadas en la base de datos.
- **Optional<Accion> findById(Integer idAccion):** Busca una acción por su identificador único (**idAccion**). Retorna un **Optional<Accion>**, lo que permite manejar de forma elegante la posibilidad de que una acción no exista.
- **Accion save(Accion p):** Guarda o actualiza una acción en la base de datos. Si **p** tiene un **id** nulo, se crea una nueva acción; si **p** tiene un **id** no nulo, la acción existente con ese **id** se actualiza.
- **void delete(Accion p):** Elimina la acción proporcionada de la base de datos.

Funcionalidad y Uso:

La interfaz **AccionRepositorio** actúa como un puente entre la lógica de la aplicación y la base de datos, abstrayendo las operaciones de base de datos para que puedan ser realizadas de manera sencilla y declarativa. La herencia de **JpaRepository** proporciona una

rica colección de métodos CRUD y facilita la implementación de operaciones personalizadas si fuera necesario.

Ventajas:

- **Abstracción de la Capa de Datos:** Permite a los desarrolladores centrarse en la lógica de negocio sin preocuparse por los detalles de la implementación de acceso a datos.
- **Manejo Elegante de Entidades:** Utiliza **Optional** para el manejo de casos donde una entidad pueda no estar presente, reduciendo el riesgo de **NullPointerException**.
- **Integración con el Ecosistema Spring:** Al extender **JpaRepository**, se integra a la perfección con otras características de Spring, como las transacciones declarativas y la inyección de dependencias.

Clase Controlador.java (Controller)

Código

```
package com.acciones.registro_acciones.controller;
import com.acciones.registro_acciones.AccionServicio;
import com.acciones.registro_acciones.model.Accion;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@CrossOrigin(origins = "http://localhost:4200")
@RestController
@RequestMapping("/{acciones}")
public class Controlador {

    @Autowired
    AccionServicio servicio;

    @GetMapping
    public List<Accion>listar(){
        return servicio.listar();
    }

    @PostMapping
```

```

public Accion agregar(@RequestBody Accion p){
    Accion accionCreada = servicio.agregar(p);
    return accionCreada;
}

@GetMapping (path =("/{id}")
public Accion listarId(@PathVariable("id")int id){
    return servicio.listarId(id);
}

@PutMapping (path =("/{id}")
public Accion editar(@RequestBody Accion p,@PathVariable("id")int id){
    p.setIdAccion(id);
    return servicio.editar(p);
}

@DeleteMapping (path =("/{id}")
public Accion delete(@PathVariable("id")int id){
    return servicio.eliminar(id);
}
}

```

Documentación de Controlador.java (Controlador)

La clase **Controlador** se encarga de manejar las solicitudes HTTP para la gestión de acciones. Utiliza **AccionServicio** para interactuar con la capa de servicio y realizar operaciones CRUD. Está anotada con **@RestController**, indicando que es un controlador de Spring MVC que devuelve objetos y datos directamente al cuerpo de la respuesta.

Anotaciones y su Significado:

- **@CrossOrigin(origins = "http://localhost:4200")**: Permite las solicitudes de origen cruzado desde **http://localhost:4200**, generalmente utilizado para permitir que el frontend (Angular) se comuniquen con el backend.
- **@RestController**: Indica que la clase es un controlador donde cada método devuelve un objeto de dominio en lugar de una vista.
- **@RequestMapping("/{acciones"})**: Mapea las solicitudes a la ruta **/acciones** para este controlador.

Métodos del Controlador:

- **public List<Accion> listar()**: Retorna todas las acciones disponibles. Mapeado a una solicitud GET.
- **public Accion agregar(@RequestBody Accion p)**: Recibe una acción para agregar a través del cuerpo de la solicitud y la agrega. Mapeado a una solicitud POST.
- **public Accion listarId(@PathVariable("id") int id)**: Retorna una acción específica por su ID. Mapeado a una solicitud GET con una variable de ruta.

- **public Accion editar(@RequestBody Accion p, @PathVariable("id") int id):** Actualiza una acción existente basándose en el ID proporcionado y los datos de la acción en el cuerpo de la solicitud. Mapeado a una solicitud PUT.
- **public Accion delete(@PathVariable("id") int id):** Elimina una acción basándose en el ID proporcionado. Mapeado a una solicitud DELETE.

Inyección de Dependencias:

- **@Autowired:** Inyecta la dependencia de **AccionServicio**, permitiendo al controlador interactuar con la capa de servicio para realizar operaciones de negocio.

Clase CorsConfig.java

Código

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {

                registry.addMapping("/registro_acciones/**").allowedOrigins("http://localhost:4200")
                    .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
                    .allowedHeaders("*")
                    .allowCredentials(true);

            }
        };
    }
}
```

Documentación de CorsConfig.java

La clase **CorsConfig** se encarga de la configuración de CORS (Cross-Origin Resource Sharing) para la aplicación Spring Boot. Esta configuración es crucial para permitir que el frontend, alojado en un dominio diferente, interactúe con el backend sin infringir la política de mismo origen. Está marcada con la anotación **@Configuration**, lo que indica que la clase se utiliza para definir beans y configuraciones de Spring.

Anotaciones Utilizadas:

- **@Configuration:** Indica que la clase es utilizada por Spring IoC container como una fuente de definiciones de beans y configuraciones.

- **@Bean**: Indica que el método anotado produce un bean que será gestionado por el contenedor de Spring. En este caso, el bean es un **WebMvcConfigurer** que personaliza la configuración de CORS.

Método corsConfigurer:

- Este método devuelve una instancia de **WebMvcConfigurer**, que se utiliza para configurar CORS globalmente en la aplicación.

Configuración de CORS:

Dentro del método **corsConfigurer**, se personaliza la configuración de CORS mediante el objeto **CorsRegistry**. Se especifican las siguientes configuraciones:

- **registry.addMapping("/registro_acciones/**")**: Aplica la configuración de CORS a todas las rutas que coincidan con el patrón **/registro_acciones/****. Esto significa que cualquier solicitud que coincida con este patrón estará sujeta a las políticas de CORS definidas aquí.
- **.allowedOrigins("http://localhost:4200")**: Especifica los orígenes que están permitidos para acceder a los recursos. En este caso, solo las solicitudes provenientes de **http://localhost:4200** están permitidas. Esto es típico en el desarrollo local donde el frontend Angular se sirve desde este origen.
- **.allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")**: Define los métodos HTTP permitidos para las solicitudes CORS. Esto incluye los métodos más comunes utilizados en las API REST.
- **.allowedHeaders("*")**: Permite todas las cabeceras en las solicitudes CORS. Esto es útil para permitir cabeceras personalizadas o específicas necesarias para la aplicación.
- **.allowCredentials(true)**: Habilita el envío de credenciales (como cookies HTTP y headers de autenticación) en las solicitudes CORS. Esto es importante para las solicitudes que requieren autenticación.

Ejemplo de Uso:

Esta configuración permite que una aplicación frontend alojada en **http://localhost:4200** interactúe con el backend de Spring Boot sin problemas de CORS, siempre que las solicitudes coincidan con el patrón **/registro_acciones/**** y cumplan con los métodos, orígenes y cabeceras permitidas definidas.

Clase Application.properties

Código
<pre>server.servlet.context-path=/registro_acciones server.port=8082 spring.datasource.url=jdbc:mysql://localhost:3307/registro_acciones spring.datasource.username=root spring.datasource.password= spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver spring.jpa.show-sql=true</pre>


```
spring.jpa.hibernate.naming.physical-  
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl  
logging.level.org.hibernate.SQL=DEBUG  
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Documentación de application.properties

El archivo **application.properties** en una aplicación Spring Boot es utilizado para configurar aspectos variados de la aplicación, incluyendo el servidor web, conexiones a la base de datos, y más. A continuación, se describe el propósito de cada configuración especificada en el archivo proporcionado:

- **server.servlet.context-path=/registro_acciones:** Establece el contexto base (path) de la aplicación. Todas las rutas de la aplicación serán relativas a este base path. En este caso, las rutas empezarán con **/registro_acciones**.
- **server.port=8082:** Define el puerto en el que se ejecutará el servidor de aplicaciones Spring Boot. Aquí se configura para usar el puerto 8082.
- **spring.datasource.url=jdbc:mysql://localhost:3307/registro_acciones:** Configura la URL de conexión a la base de datos. Indica que se utiliza MySQL corriendo localmente en el puerto 3307, y la base de datos específica es **registro_acciones**.
- **spring.datasource.username=root:** Define el nombre de usuario para conectarse a la base de datos MySQL. En este caso, es el usuario **root**.
- **spring.datasource.password=:** Establece la contraseña para el usuario especificado para la conexión a la base de datos. Aquí se deja en blanco, lo que implica que no hay contraseña.
- **spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver:** Especifica el nombre de la clase del driver JDBC de MySQL que se utilizará para las conexiones a la base de datos.
- **spring.jpa.show-sql=true:** Habilita la visualización de las consultas SQL generadas por Hibernate/JPA en la consola. Es útil para fines de depuración y entendimiento de cómo se traducen las operaciones de JPA a SQL.
- **spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl:** Define la estrategia de nomenclatura física para Hibernate. La estrategia especificada aquí es la implementación estándar, que no aplica ninguna transformación a los nombres de las entidades y columnas.
- **logging.level.org.hibernate.SQL=DEBUG:** Configura el nivel de logging de las sentencias SQL generadas por Hibernate. El nivel **DEBUG** permite que se registren detalladamente.
- **logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE:** Establece el nivel de logging para el enlace de parámetros SQL de Hibernate. El nivel **TRACE** es aún más detallado que **DEBUG**, mostrando exactamente qué datos se enlazan a cada parámetro SQL en las consultas.

Resumen

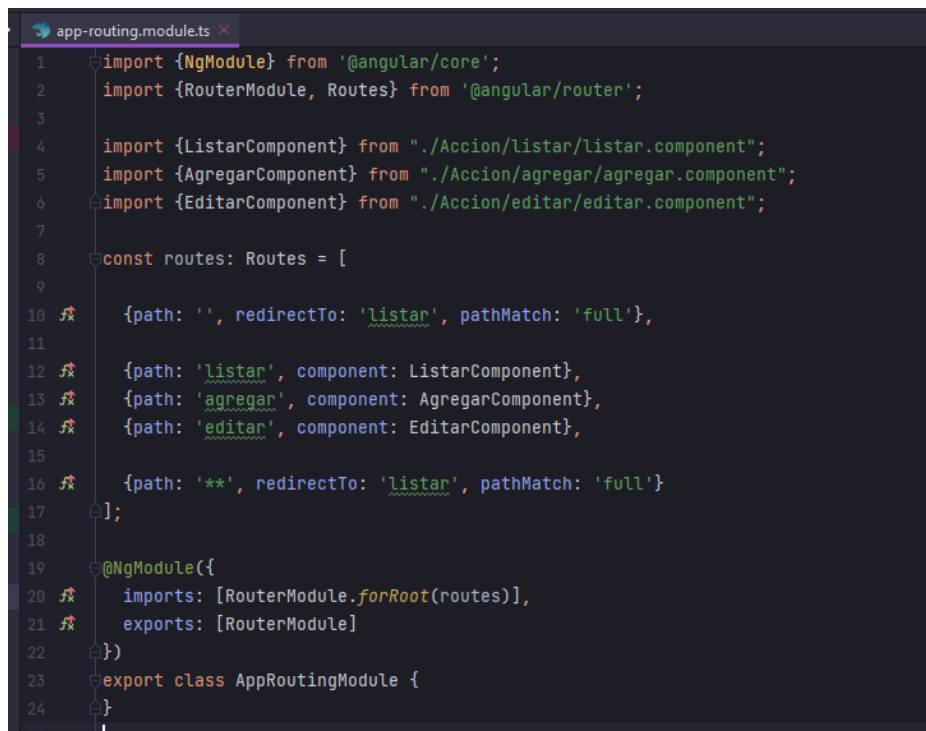
Este archivo **application.properties** configura una aplicación Spring Boot para correr en el puerto 8082 con un contexto base **/registro_acciones**, conectarse a una base de datos MySQL local en el puerto 3307 usando el usuario **root** sin contraseña, y habilita logs detallados para las operaciones SQL y de Hibernate. Esto facilita el desarrollo y la depuración al proporcionar visibilidad sobre las interacciones con la base de datos y el comportamiento de la ORM.

Conexión del BackEnd SpringBoot en Netbeans con la base de datos.



Documentación del Front-End con Angular 17 y Bootstrap 4

Componente App-routing.module.ts



```
1 import {NgModule} from '@angular/core';
2 import {RouterModule, Routes} from '@angular/router';
3
4 import {ListarComponent} from './Accion/listar/listar.component';
5 import {AgregarComponent} from './Accion/agregar/agregar.component';
6 import {EditarComponent} from './Accion/editar/editar.component';
7
8 const routes: Routes = [
9
10   {path: '', redirectTo: 'listar', pathMatch: 'full'},
11
12   {path: 'listar', component: ListarComponent},
13   {path: 'agregar', component: AgregarComponent},
14   {path: 'editar', component: EditarComponent},
15
16   {path: '**', redirectTo: 'listar', pathMatch: 'full'}
17 ];
18
19 @NgModule({
20   imports: [RouterModule.forRoot(routes)],
21   exports: [RouterModule]
22 })
23 export class AppRoutingModule {
24 }
25
```

Documentación de AppRoutingModuleModule en Angular

El módulo **AppRoutingModule** en Angular es responsable de definir las rutas y la navegación dentro de la aplicación. Se utiliza para mapear las URL a componentes específicos, permitiendo la carga dinámica de contenido según la ruta a la que accede el usuario. A continuación, se detalla la configuración de rutas definida en el archivo **app-routing.module.ts**:

Importaciones

- **NgModule:** Permite definir un módulo que encapsula componentes, directivas, servicios, etc.
- **RouterModule, Routes:** Herramientas de Angular Router que permiten configurar las rutas y realizar operaciones de enrutamiento.
- Componentes **ListarComponent**, **AgregarComponent**, **EditarComponent:** Componentes importados que serán asociados a rutas específicas.

Configuración de Rutas

Las rutas se definen en el arreglo **routes**, donde cada objeto de ruta puede tener las siguientes propiedades:

- **path:** La URL en el navegador que se asocia con el componente.
- **component:** El componente que se cargará cuando se navegue a la ruta especificada.
- **redirectTo:** Una URL a la que se redirigirá al usuario.
- **pathMatch:** Define cómo se debe hacer coincidir la URL. El valor **full** significa que toda la URL debe coincidir.

Rutas Definidas

1. **Ruta Raíz (''):** Redirige a **'listar'** inmediatamente, asegurando que la aplicación tenga una página de inicio definida.
2. **Ruta 'listar':** Asocia la URL **/listar** con el componente **ListarComponent**, que se encarga de mostrar la lista de acciones.
3. **Ruta 'agregar':** Asocia la URL **/agregar** con el componente **AgregarComponent**, utilizado para añadir nuevas acciones.
4. **Ruta 'editar':** Asocia la URL **/editar** con el componente **EditarComponent**, destinado a la edición de acciones existentes.
5. **Ruta Comodín ('**'):** Redirige a **'listar'** para cualquier otra ruta no especificada, actuando como una especie de mecanismo de "página no encontrada".

Decorador @NgModule

El decorador **@NgModule** se utiliza para definir el módulo de enrutamiento, especificando las importaciones necesarias y las exportaciones. En este caso, importa **RouterModule.forRoot(routes)**, que inicializa el enrutador con las rutas definidas anteriormente, y exporta **RouterModule** para hacerlo accesible en toda la aplicación.

Resumen

AppRoutingModule organiza la navegación en la aplicación Angular, definiendo rutas claras para listar, agregar y editar acciones, así como una ruta por defecto y un mecanismo de redirección para manejar URLs no reconocidas. Esto facilita la gestión de la navegación y la organización del contenido en la aplicación.

Componente app.component.ts

```
app.component.ts
1  import {Component} from '@angular/core';
2  import {Router} from '@angular/router';
3
4  @Component({
5    selector: 'app-root',
6    templateUrl: './app.component.html',
7    styleUrls: ['./app.component.css']
8  })
9  export class AppComponent {
10    title = 'Acciones1';
11
12    constructor(private router: Router) {
13    }
14
15    Listar() {
16      this.router.navigate(commands: ["listar"]);
17    }
18
19    Agregar() {
20      this.router.navigate(commands: ["agregar"]);
21    }
22  }
23
```

Documentación del Componente AppComponent en Angular

El componente **AppComponent** actúa como el componente raíz de la aplicación Angular, desde donde se gestionan las operaciones y navegaciones principales. A continuación, se desglosan los elementos clave del código proporcionado para **app.component.ts**:

Importaciones

- **Component**: Decorador que permite definir un componente y sus metadatos.
- **Router**: Servicio que permite manipular la navegación y la URL del navegador.

Decorador @Component

Define la configuración del componente **AppComponent**, incluyendo:

- **selector**: Identificador del componente que se usa en el HTML para renderizar el componente.
- **templateUrl**: Ruta al archivo HTML que define la vista del componente.
- **styleUrl**: Ruta al archivo CSS que contiene los estilos específicos del componente.

Propiedades del Componente

- **title**: Una propiedad de tipo **string** que almacena el título de la aplicación, en este caso, 'Acciones1'.

Constructor

El constructor del componente inyecta el servicio **Router**, proporcionando al componente la capacidad de realizar navegaciones programáticas.

Métodos

- **Listar()**: Método que utiliza el servicio **Router** para navegar a la ruta 'listar'. Este método es llamado para mostrar la lista de acciones dentro de la aplicación.

- **Agregar():** Método similar al anterior pero para navegar a la ruta '**agregar**', utilizado para mostrar el formulario que permite añadir una nueva acción.

Vista del Componente listar.component.html



Vista del Componente editar.component.html



Vista del Componente agregar.component.html



Persistencia (Base de Datos) Xampp

Query para creación de la base de datos

```
-- phpMyAdmin SQL Dump
-- version 5.2.1
-- https://www.phpmyadmin.net/
--
-- Servidor: 127.0.0.1
-- Tiempo de generación: 24-02-2024 a las 19:18:04
-- Versión del servidor: 10.4.32-MariaDB
-- Versión de PHP: 8.2.12

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Base de datos: `registro_acciones`
--
-----

--
-- Estructura de tabla para la tabla `acciones`
--

CREATE TABLE `acciones` (
  `idAccion` int(11) NOT NULL,
  `NombreAccion` varchar(5) NOT NULL,
  `FechaCompra` date NOT NULL,
  `PrecioCompraAccion` int(3) NOT NULL,
  `CantidadAcciones` int(3) NOT NULL,
  `CostoTotalCompra` int(25) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

--
-- Volcado de datos para la tabla `acciones`
--

INSERT INTO `acciones` (`idAccion`, `NombreAccion`, `FechaCompra`,
`PrecioCompraAccion`, `CantidadAcciones`, `CostoTotalCompra`) VALUES
(1, 'AAPL', '2023-01-10', 150, 10, 1500),
(2, 'MSFT', '2023-02-15', 250, 5, 1250),
(3, 'TSLA', '2023-03-20', 700, 2, 1400);

--
```

```

-- Disparadores `acciones`
--
DELIMITER $$
CREATE TRIGGER `resultado2` BEFORE UPDATE ON `acciones` FOR EACH ROW BEGIN
    SET NEW.CostoTotalCompra = NEW.PrecioCompraAccion * NEW.CantidadAcciones;
END
$$
DELIMITER ;
DELIMITER $$
CREATE TRIGGER `resultado3` BEFORE INSERT ON `acciones` FOR EACH ROW BEGIN
    SET new.CostoTotalCompra = new.PrecioCompraAccion * new.CantidadAcciones;
END
$$
DELIMITER ;

--
-- Índices para tablas volcadas
--
--
-- Indices de la tabla `acciones`
--
ALTER TABLE `acciones`
    ADD PRIMARY KEY (`idAccion`);

--
-- AUTO_INCREMENT de las tablas volcadas
--
--
-- AUTO_INCREMENT de la tabla `acciones`
--
ALTER TABLE `acciones`
    MODIFY `idAccion` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=5;
COMMIT;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;

```

Documentación del Script para la creación de la base de datos

Este script SQL se utiliza para crear y configurar una base de datos llamada **registro_acciones** que almacena información sobre acciones compradas, tales como el nombre de la acción, la fecha de compra, el precio por acción, la cantidad de acciones y el costo total de la compra. A continuación, se detalla la estructura y propósito de cada parte del script:

Configuración Inicial

- Establece el modo SQL para evitar la asignación automática de valores a las columnas con **AUTO_INCREMENT**.
- Inicia una transacción para agrupar varias operaciones SQL como una sola unidad de trabajo.
- Configura la zona horaria a UTC (**+00:00**).
- Ajusta el conjunto de caracteres y collation a **utf8mb4** y **utf8mb4_general_ci** respectivamente, lo que permite el uso de emojis y caracteres especiales.

Creación de la Base de Datos y la Tabla

- Crea la base de datos **registro_acciones**.
- Define la estructura de la tabla **acciones** con las columnas:
 - **idAccion**: Un identificador único para cada acción (clave primaria).
 - **NombreAccion**: El nombre o símbolo de la acción.
 - **FechaCompra**: La fecha en que se compró la acción.
 - **PrecioCompraAccion**: El precio de compra por acción.
 - **CantidadAcciones**: La cantidad de acciones compradas.
 - **CostoTotalCompra**: El costo total de la compra, calculado como el precio por acción multiplicado por la cantidad de acciones.

Inserción de Datos

- Inserta registros de ejemplo en la tabla **acciones**, con datos de acciones de empresas como Apple (AAPL), Microsoft (MSFT), y Tesla (TSLA).

Disparadores (Triggers)

- Crea dos disparadores, **resultado2** y **resultado3**, que se ejecutan antes de las operaciones de actualización e inserción, respectivamente. Estos disparadores calculan automáticamente el **CostoTotalCompra** multiplicando el **PrecioCompraAccion** por la **CantidadAcciones**.

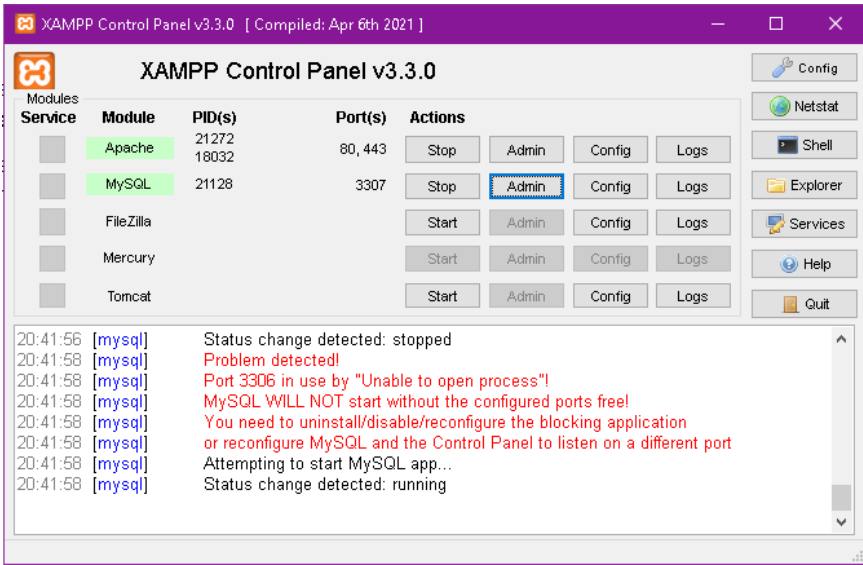
Índices y Auto Incremento

- Establece la columna **idAccion** como clave primaria.
- Configura **idAccion** para que se autoincrementa, asegurando que cada nueva acción insertada tenga un identificador único.

Finalización

- Aplica las configuraciones iniciales de conjunto de caracteres y collation.
- Finaliza la transacción con un **COMMIT**, asegurando que todas las operaciones previas se completen satisfactoriamente.

Este script SQL es fundamental para la creación y configuración inicial de la base de datos del proyecto, permitiendo almacenar y gestionar datos de acciones de manera eficiente y automática.



Examinar

Estructura

SQL

Buscar

Insertar

Exportar

Importar

Privilegios

Operaciones

Mostrando filas 0 - 1 (total de 2, La consulta tardó 0,0004 segundos.)

SELECT * FROM `acciones`

Perfilando

Editar en línea

Editar

Explicar SQL

Crear código PHP

Actualizar

Mostrar todo

Número de filas: 25

Filtrar filas: Buscar en esta tabla

Ordenar según la clave: Ninguna

Opciones extra

				idAccion	NombreAccion	FechaCompra	PrecioCompraAccion	CantidadAcciones	CostoTotalCompra
<input type="checkbox"/>	Editar	Copiar	Borrar	2	MSFT	2024-02-15	250	6	1500
<input type="checkbox"/>	Editar	Copiar	Borrar	3	TSLA	2024-03-20	700	2	1400

Seleccionar todo

Para los elementos que están marcados:

Editar

Copiar

Borrar

Exportar

Mostrar todo

Número de filas: 25

Filtrar filas: Buscar en esta tabla

Ordenar según la clave: Ninguna

Documento de Análisis de Cambio

Para la realización de este proyecto, se tuvo que hacer un diagnóstico y análisis en base al proyecto anteriormente creado, y teniendo tantos puntos a considerar como lo fueron: el lenguaje de programación, el entorno de trabajo y la calidad del producto software, como grupo nos vimos en la necesidad de utilizar un nuevo entorno de trabajo y reestructurar toda la arquitectura del proyecto, optando por un framework mejor desarrollado y distribuido que en la anterior entrega.

Siendo así definiremos los principales cambios adoptados para la creación del proyecto.

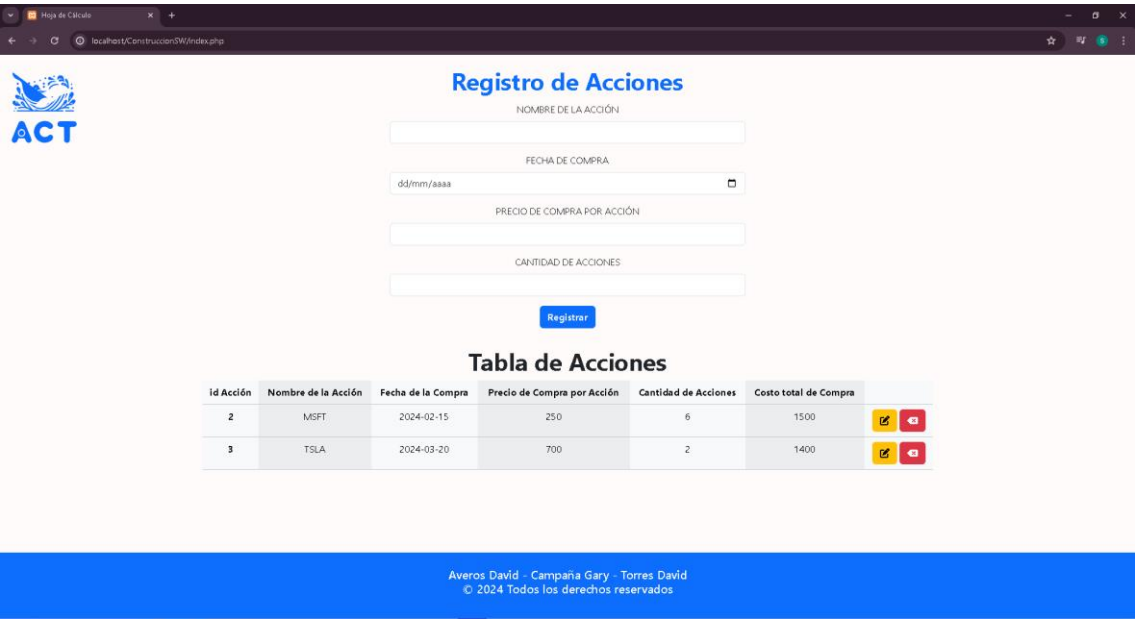
Diferencias	
Proyecto Primer Bimestre	Proyecto Segundo Bimestre
Lenguajes de programación	
PHP, HTML y CSS	TypeScript, Java, HTML y CSS
Entorno de trabajo (Framework)	
PHPStorm & Xampp	Front-End -> Angular & Bootstrap Back-End -> SpringBoot
Conexión con la Base De Datos	
Clase conexión.php	JPA & JDBC Drivers (SpringBoot)
Servidor Web del Front-End	
Servidor Apache (XAMPP)	Servidor de Angular CLI (ng serve)
Servidor de Back-End	
Servidor Apache (XAMPP)	Servidor de SpringBoot (Tomcat)
Vista	
PHP & Html (index.php servidor apache)	TypeScript & Html (Angular CLI)
Controlador	
Clases PHP (delete, register y modify)	Clase Controlador (BackEnd – SpringBoot)
Modelo	
X	Clase Acción.java (SpringBoot)
Comunicación Front con Back-End	
X	Clase application.properties (SpringBoot)
Comunicación Back-End con la persistencia	
X	Proyecto de SpringBoot tipo Maven
Control de Solicitudes HTTP	
X	Clase CorsConfig.java

Como Podemos observar, el nuevo proyecto cumple con especificación más completas y técnicas que el de primer bimestre.

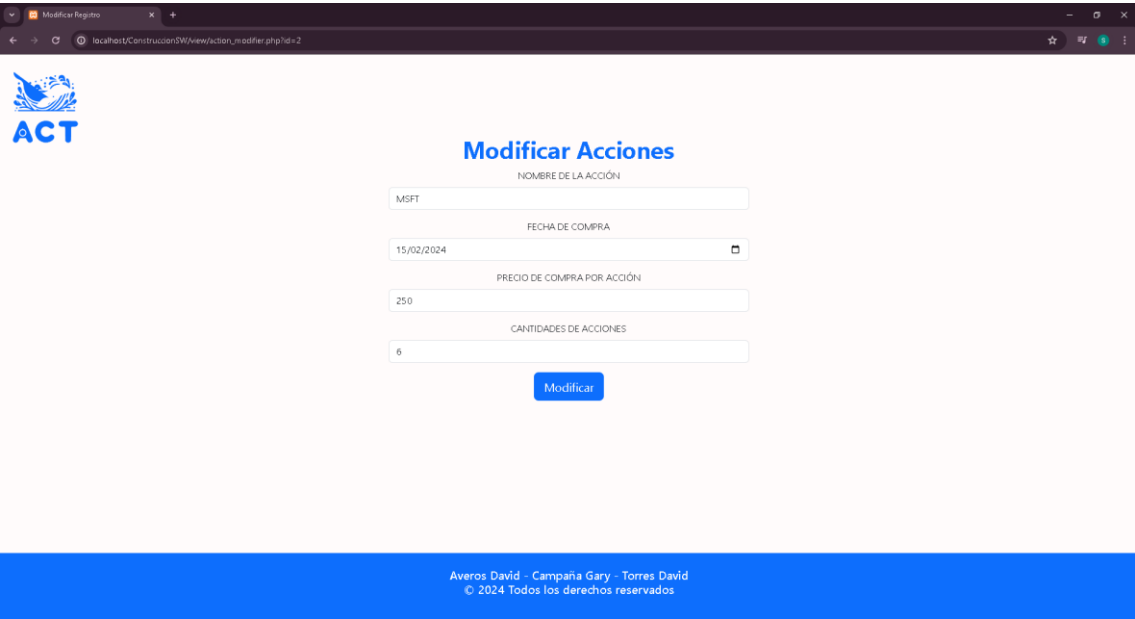
A continuación, se podrá visualizar la diferencia entre las vistas de ambos proyectos:

VISTAS PROYECTO PRIMER BIMESTRE

Vista index.php



Vista ModificarAcción



VISTAS PROYECTO SEGUNDO BIMESTRE

Vista del Componente listar.component.html



Vista del Componente editar.component.html



Vista del Componente agregar.component.html



En la parte visual, se adopto conservar tanto la tonalidad de colores así como los componentes del footer y la imagen del Logo.

Donde nos encontramos con la siguiente tabla para observar las similitudes o componentes que se reutilizaron en ambos proyectos:

Semejanzas		
	Proyecto Primer Bimestre	Proyecto Segundo Bimestre
Base de Datos	X	X
Tonalidad de Colores (Vistas)	X	X
Pie de Página (Footer)	X	X
Servidor de Persistencia (Apache - XAMPP)	X	X
Funciones CRUD	X	X