
spring
Boot

Introducción al desarrollo de microservicios

Los microservicios son una arquitectura de desarrollo de software que se ha vuelto muy popular en los últimos años. En lugar de construir una única aplicación monolítica, los microservicios dividen una aplicación en componentes más pequeños y autónomos, llamados "microservicios". Cada microservicio tiene la capacidad de realizar una función o tarea específica dentro de la aplicación y puede ser desarrollado, probado, implementado y escalado de forma independiente. Algunos conceptos clave relacionados con los microservicios son:

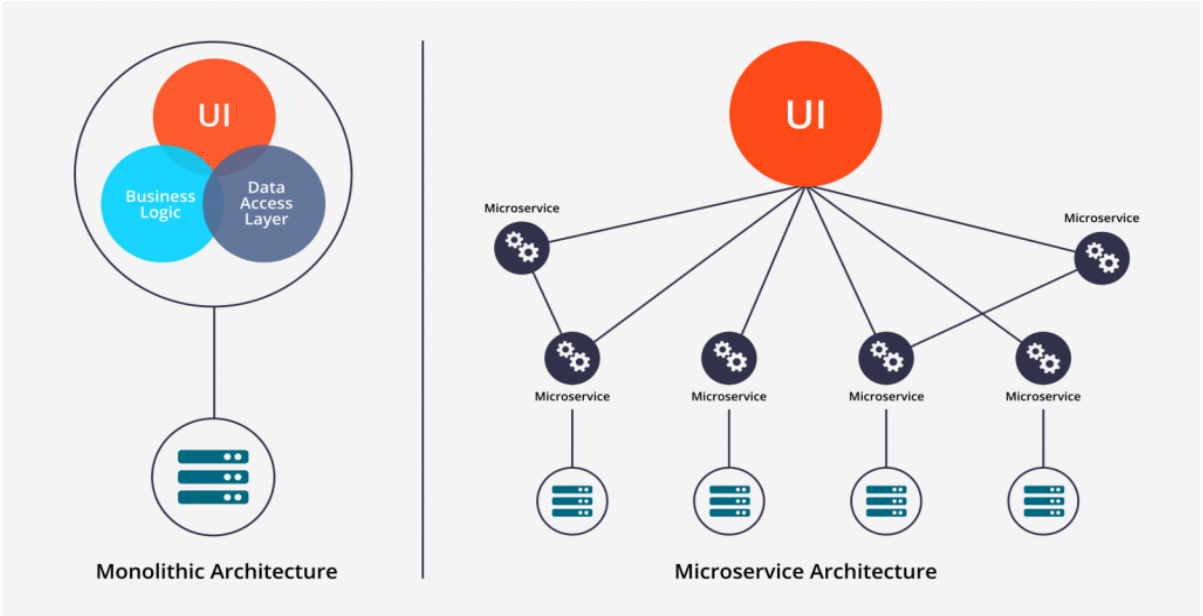
- **Independencia y desacoplamiento**: Cada microservicio opera de manera independiente, lo que significa que puede ser desarrollado por equipos separados, utilizando diferentes tecnologías y bases de datos si es necesario. Esto fomenta el desacoplamiento y reduce la interdependencia entre los componentes del sistema.
- **Comunicación entre microservicios**: Los microservicios se comunican entre sí a través de protocolos como HTTP, gRPC o mensajería, lo que permite que trabajen juntos para brindar una funcionalidad completa. Esto se logra a través de llamadas a API o servicios de mensajería.
- **Escalabilidad**: Los microservicios se pueden escalar de forma independiente, lo que significa que puedes asignar más recursos a un microservicio específico cuando experimenta una mayor carga de trabajo, sin afectar a otros microservicios.
- **Facilidad de mantenimiento y despliegue**: Dado que los microservicios son unidades independientes, es más fácil mantener y desplegar cambios en una parte de la aplicación sin afectar al resto. Esto facilita la implementación continua (CI/CD) y la corrección de errores.
- **Resistencia a fallos**: Los microservicios pueden diseñarse para ser resistentes a fallos, lo que significa que si un microservicio falla, no afecta necesariamente a la funcionalidad global de la aplicación. Se pueden implementar estrategias de recuperación, como la reinternación automática, para garantizar la disponibilidad.
- **Documentación y descubrimiento de servicios**: Cada microservicio debe estar bien documentado, y existen herramientas y estándares para facilitar el descubrimiento de servicios en un entorno de microservicios.
- **Seguridad**: La seguridad es un aspecto crítico en la arquitectura de microservicios. Cada microservicio debe estar protegido y se deben establecer mecanismos de autenticación y autorización.
- **Gestión de versiones**: Dado que los microservicios se pueden desarrollar y escalar de manera independiente, es importante gestionar las versiones de la API para garantizar la compatibilidad entre los servicios.

Los microservicios ofrecen flexibilidad y escalabilidad, pero también presentan desafíos, como la complejidad en la gestión de la comunicación entre servicios y la necesidad de una infraestructura adecuada para orquestar y administrar los microservicios. En el manual, se explorarán estas cuestiones en profundidad y se proporcionarán ejemplos y buenas prácticas para diseñar, desarrollar y desplegar aplicaciones basadas en microservicios con Java y Spring Boot.

Arquitecturas monolíticas vs Microservicios

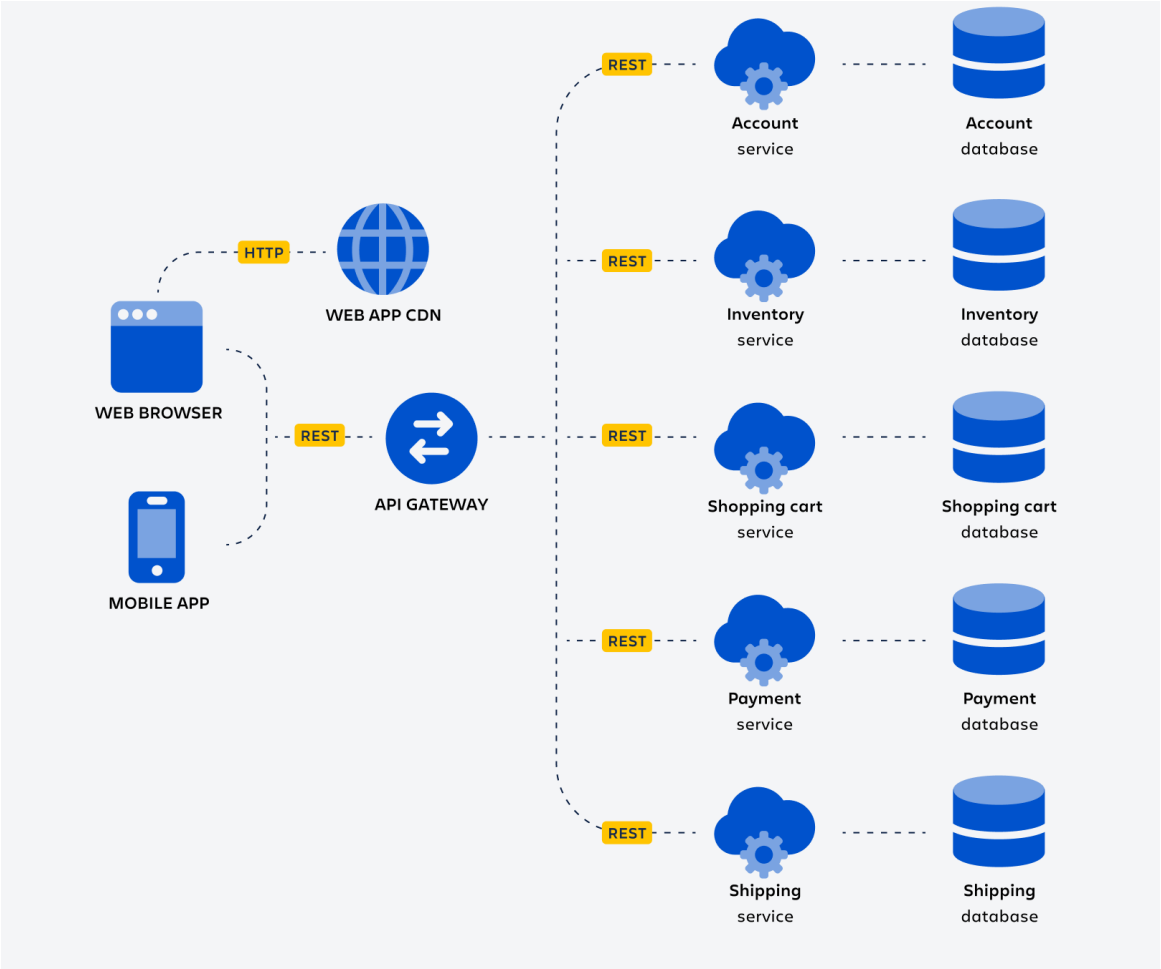
Una arquitectura monolítica es un modelo tradicional de un programa de software que se compila como una unidad unificada y que es autónoma e independiente de otras aplicaciones. Una arquitectura de microservicios es el concepto opuesto al de la arquitectura monolítica, ya que es un método que se basa en una serie de servicios que se pueden implementar de forma independiente. La arquitectura monolítica puede resultar práctica al principio de un proyecto para aliviar la sobrecarga cognitiva de la gestión de código, así como la implementación. Pero una vez que una aplicación monolítica se vuelve grande y compleja, resulta difícil escalarla, la implementación continua pasa a ser un desafío y las actualizaciones pueden resultar complicadas.

Si bien una aplicación monolítica se crea como una sola unidad indivisible, los microservicios dividen esa unidad en una colección de unidades independientes que contribuyen a un todo más amplio. Una aplicación se construye como una serie de servicios que se pueden implementar de forma independiente, están descentralizados y se desarrollan de forma autónoma.



Los microservicios entran en la categoría de sistema distribuido. Un sistema distribuido es un conjunto de programas informáticos que utilizan recursos computacionales en varios nodos de cálculo distintos para lograr un objetivo compartido común. Estos sistemas consiguen mayor fiabilidad, rendimiento y facilidad de escalabilidad.

Los nodos de un sistema distribuido ofrecen redundancia, de modo que, si un nodo falla, hay otros que pueden sustituirlos y reparar el error. Cada nodo se puede escalar de forma horizontal y vertical, lo que mejora el rendimiento. Si un sistema se somete a una carga extensiva, pueden añadirse nodos adicionales para ayudar a absorber dicha carga.



Spring Boot

Spring Boot es un potente framework de desarrollo de aplicaciones Java que ha revolucionado la forma en que se construyen y despliegan aplicaciones empresariales. Diseñado para simplificar drásticamente el desarrollo de aplicaciones, Spring Boot se ha convertido en una opción popular para desarrolladores y equipos de desarrollo que buscan crear aplicaciones Java eficientes, escalables y de alto rendimiento.

En esencia, Spring Boot es una extensión del ampliamente utilizado framework Spring, pero con un enfoque singular en la simplicidad y la productividad. La filosofía central de Spring Boot es reducir la complejidad de configuración y proporcionar una serie de convenciones y características listas para usar que permiten a los desarrolladores centrarse en la lógica de negocio en lugar de preocuparse por la configuración y la infraestructura.

A continuación, destacamos algunas de las características y ventajas más importantes de Spring Boot:

1. Configuración Simplificada: Spring Boot utiliza configuración basada en anotaciones, lo que significa que puedes definir la configuración directamente en tus clases Java, eliminando la necesidad de archivos XML de configuración. Esto simplifica significativamente la gestión de la configuración de tu aplicación.

2. Autoconfiguración Inteligente: Spring Boot incorpora un potente mecanismo de autoconfiguración. Detecta automáticamente las bibliotecas y las dependencias que estás utilizando en tu proyecto y configura la aplicación en consecuencia. Esto reduce la necesidad de configuraciones manuales y aumenta la eficiencia.

3. Starter POMs: Spring Boot proporciona Starter POMs (Project Object Models) preconfigurados que agrupan las dependencias comunes para tareas específicas, como desarrollo web, acceso a bases de datos, seguridad, y más. Esto hace que la gestión de dependencias sea sencilla y permite a los desarrolladores comenzar rápidamente con proyectos específicos.

4. Contenedores Integrados: Spring Boot incluye contenedores integrados para servidores web, como Tomcat, Jetty o Undertow. Esto significa que puedes empaquetar tu aplicación en un archivo JAR autocontenido que incluye su propio servidor web, evitando la necesidad de desplegar en servidores externos.

5. Soporte para Microservicios: Spring Boot es una elección popular para el desarrollo de microservicios, ya que facilita la creación de servicios independientes y escalables. La facilidad de implementación y la comunicación eficiente entre servicios lo convierten en una opción atractiva en el mundo de la arquitectura de microservicios.

6. Spring Boot Actuator: Esta extensión de Spring Boot proporciona características de monitorización y administración que facilitan el seguimiento del estado de una aplicación. Permite acceder a métricas, información sobre la salud de la aplicación, información sobre los beans de Spring, entre otros.

7. Desarrollo rápido y pruebas unitarias: Spring Boot facilita el desarrollo rápido de aplicaciones y promueve las pruebas unitarias. Los desarrolladores pueden crear controladores y servicios de manera eficiente, y se pueden escribir pruebas unitarias utilizando las características de Spring Boot para simular el entorno de la aplicación.

8. Integración con Spring Ecosystem: Spring Boot se integra perfectamente con otros proyectos del ecosistema Spring, como Spring Data, Spring Security, Spring Cloud, entre otros. Esto facilita la construcción de aplicaciones completas y escalables.

9. Facilita la implementación continua (CI/CD): La estructura de proyectos y la facilidad de configuración en Spring Boot hacen que sea más fácil implementar aplicaciones en entornos de integración continua y entrega continua.

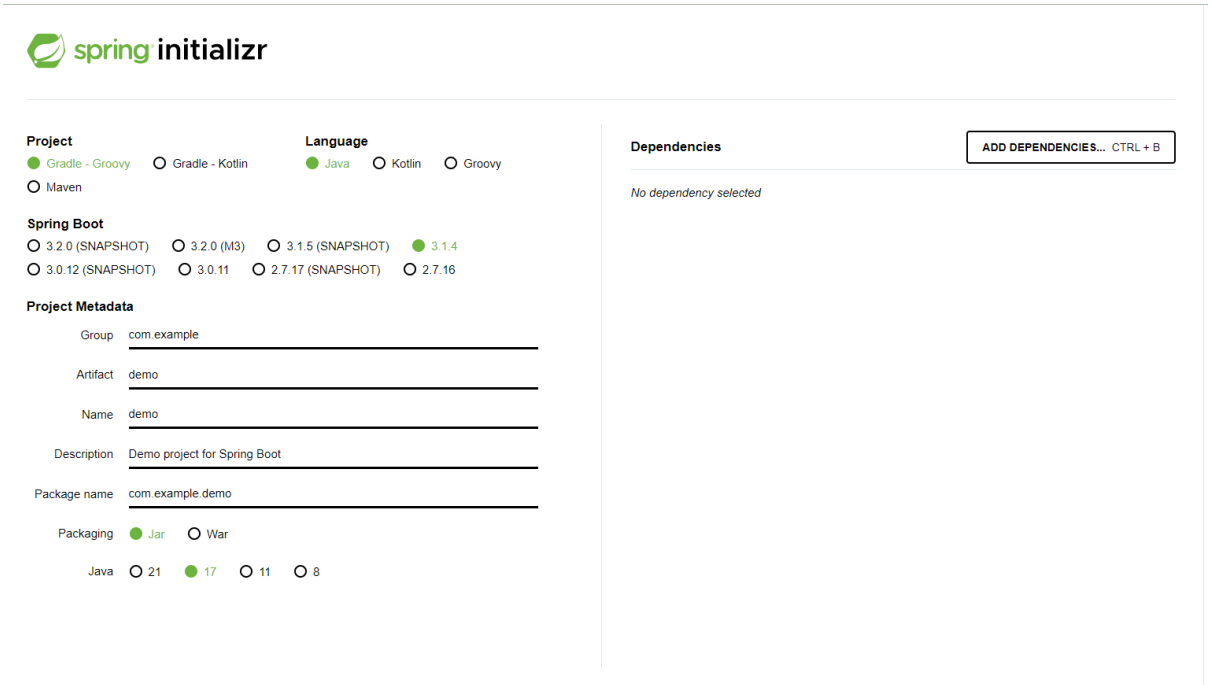
10. Documentación y comunidad activa: Spring Boot cuenta con una amplia documentación oficial y una comunidad activa que proporciona recursos, tutoriales y soporte a los desarrolladores.

Configuración y creación de un proyecto con Spring Boot

La configuración y creación de un proyecto con Spring Boot es un proceso relativamente sencillo gracias a las herramientas proporcionadas por Spring Initializr y las convenciones de Spring Boot. Aquí te explico cómo configurar y crear un proyecto básico con Spring Boot:

Paso 1: Acceder a Spring Initializr

Visita el sitio web de Spring Initializr en https://start.spring.io/.



Paso 2: Configurar el Proyecto

En la página inicial de Spring Initializr, verás un formulario que te permite configurar tu proyecto. Aquí tienes algunas de las configuraciones más comunes:

- **Project**: Selecciona el tipo de proyecto que deseas crear. Por ejemplo, puedes elegir "Maven Project" o "Gradle Project" como sistema de gestión de proyectos.
- **Language**: Selecciona el lenguaje de programación, que será Java en la mayoría de los casos.
- **Spring Boot**: Selecciona la versión de Spring Boot que deseas utilizar. La última versión estable es una elección común.
- **Group**: Es el paquete raíz de tu proyecto, generalmente se usa la notación de paquete inversa del dominio de tu empresa (por ejemplo, com.ejemplo).

- **Artifact**: Es el nombre de tu proyecto.
- **Description**: Proporciona una descripción breve de tu proyecto.
- **Packaging**: Puedes elegir "Jar" o "War" según tus necesidades. "Jar" es común para aplicaciones independientes.
- **Java**: Selecciona la versión de Java que deseas utilizar.
- **Dependencies**: Agrega las dependencias específicas que necesitas para tu proyecto. Por ejemplo, "Spring Web" para aplicaciones web, "Spring Data JPA" para acceso a bases de datos, etc.

Una vez que hayas configurado las opciones, puedes hacer clic en el botón "Generate" para generar el proyecto base.

Paso 3: Descargar el Proyecto Base

Spring Initializr generará un archivo ZIP que contiene tu proyecto base con la configuración especificada. Descarga este archivo ZIP a tu sistema local.

Paso 4: Descomprimir el Proyecto

Descomprime el archivo ZIP descargado en la ubicación deseada en tu sistema.

Paso 5: Importar el Proyecto en tu IDE

Abre tu Entorno de Desarrollo Integrado (IDE) preferido. La mayoría de los IDE populares, como IntelliJ IDEA, Eclipse o Visual Studio Code, admiten el desarrollo de aplicaciones con Spring Boot.

Importa el proyecto recién creado en tu IDE. Cada IDE tiene su propio proceso de importación, pero generalmente, puedes seleccionar la opción "Importar proyecto" o "Abrir proyecto" y seleccionar la carpeta raíz de tu proyecto.

Paso 6: Iniciar el Desarrollo

Una vez que hayas importado el proyecto en tu IDE, estarás listo para comenzar a desarrollar. Spring Boot proporciona una estructura de proyecto predefinida que incluye directorios para controladores, servicios, modelos y más. Además, las dependencias que seleccionaste en Spring Initializr ya están configuradas y listas para usar.

Comienza a desarrollar tu aplicación, escribiendo tus controladores, servicios y configuración según sea necesario. Puedes utilizar anotaciones de Spring, como @Controller, @Service y @Repository, para definir tus componentes.

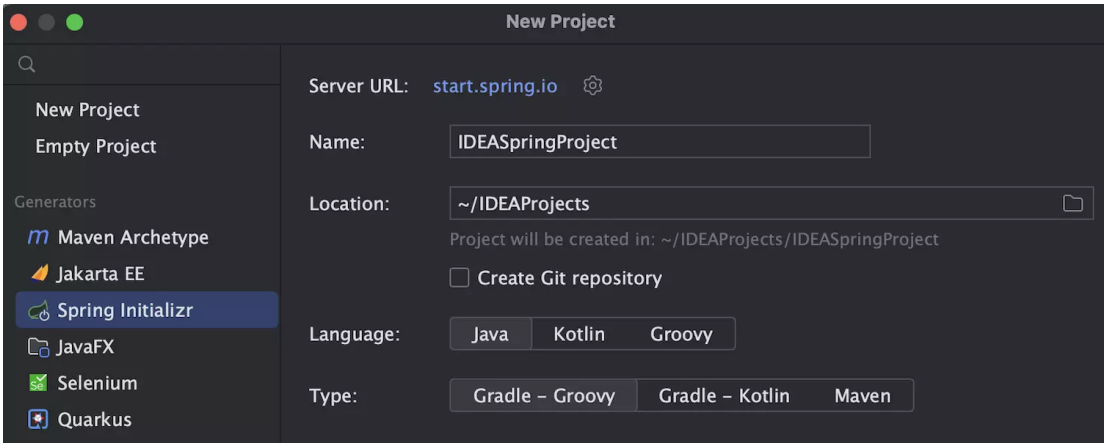
Paso 7: Ejecución de la Aplicación

Para ejecutar tu aplicación Spring Boot, busca la clase principal que contiene el método main() (por lo general, llamada Application o similar) y ejecútala.

Spring Boot iniciará automáticamente el servidor web incorporado y tu aplicación estará en funcionamiento.

Eso es todo. Has configurado y creado un proyecto básico con Spring Boot. Ahora puedes seguir desarrollando tu aplicación y aprovechar las características y la simplicidad que Spring Boot proporciona. Recuerda que puedes personalizar y extender tu proyecto agregando más dependencias y ajustando la configuración según tus necesidades específicas.

Cabe destacar que algunos IDEs como Intellij, incluyen Spring Initializr en el propio entorno de desarrollo, por lo que dicho proceso se realiza de manera similar en el entorno, facilitando su gestión y creación.



Gestión de dependencias

En las dependencias que se muestran a continuación se contará con el gestor de dependencias Maven seleccionado para nuestro proyecto.

Básicas

- **Spring Boot Starter**: Esta es la dependencia principal de Spring Boot que proporciona un conjunto de funcionalidades básicas para iniciar una aplicación Spring Boot.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```

- **Spring Boot Starter Data JPA**: Esta dependencia agrega soporte para JPA (Java Persistence API) y simplifica la configuración de la capa de acceso a datos.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- **Hibernate**: Agrega Hibernate como el proveedor de persistencia.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
</dependency>
```

- **Driver de Base de Datos**: Debes agregar la dependencia correspondiente al controlador (driver) de la base de datos que planeas utilizar. Por ejemplo, si estás usando MySQL, agregarías:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Opcionales

- **HikariCP (Pool de Conexiones):** Es una buena práctica configurar un pool de conexiones para administrar las conexiones a la base de datos de manera eficiente. HikariCP es una elección común en combinación con Spring Boot**.**
- **Lombok:** Lombok es una biblioteca que simplifica la creación de clases Java reduciendo la necesidad de escribir código repetitivo, como métodos getters y setters.
- **Spring Boot DevTools:** Facilita el desarrollo y la recarga en caliente de la aplicación durante el proceso de desarrollo. Esto puede mejorar la productividad al evitar la necesidad de reiniciar la aplicación manualmente después de cada cambio.
- **Spring Data REST**: Esta dependencia permite exponer automáticamente tus repositorios JPA como servicios RESTful. Facilita la creación de una API REST completa a partir de tus repositorios JPA sin tener que escribir mucho código adicional.
- **Spring Security**: Si necesitas implementar autenticación y autorización en tu aplicación, Spring Security es una dependencia recomendada que facilita la configuración de seguridad.
- **MapStruct**: Esta biblioteca es útil para mapear objetos DTO (Data Transfer Objects) a entidades JPA y viceversa. Simplifica la conversión entre diferentes tipos de objetos.
- **Springfox Swagger2**: Agregar esta dependencia permite documentar tu API REST de manera automática y proporciona una interfaz de usuario para explorar la documentación de la API.
- **Spring Cloud Sleuth**: Si planeas implementar una arquitectura de microservicios, Spring Cloud Sleuth proporciona rastreo distribuido para ayudar a depurar y monitorizar las solicitudes a través de múltiples servicios.
- **Spring Data Redis**: Si necesitas una caché distribuida o una base de datos en memoria, Spring Data Redis es una opción popular para la integración de Redis.
- **Spring Retry**: Esta dependencia es útil si necesitas agregar lógica de reintento a operaciones que pueden fallar, como llamadas a servicios externos.
- **Thymeleaf**: Si estás desarrollando una aplicación web y necesitas un motor de plantillas, Thymeleaf es una excelente opción que se integra bien con Spring Boot.

Patrón de Diseño

Basado en el clásico patrón MVC:

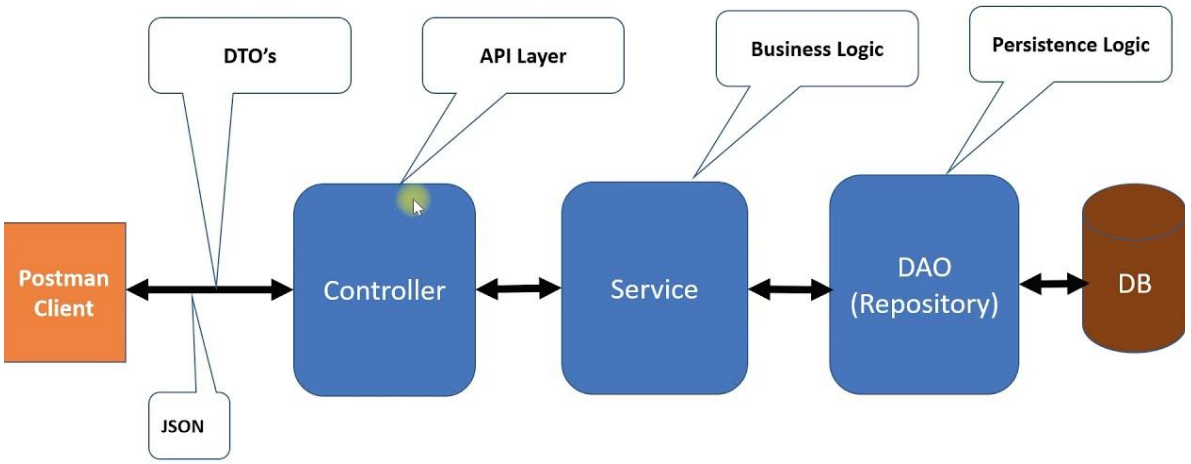
Modelo-Vista-Controlador (MVC):

Modelo: Representa los datos y la lógica de la aplicación. En el contexto de una aplicación Spring Boot con JPA, el modelo a menudo se asocia con las entidades de la base de datos, que son objetos Java anotados con JPA (por ejemplo, @Entity), que representan tablas en la base de datos.

Vista: Es la capa de presentación de la aplicación que se encarga de mostrar los datos al usuario y de capturar las interacciones del usuario. En una aplicación web, las vistas suelen ser páginas HTML, plantillas Thymeleaf, o componentes front-end como React o Angular. En el contexto de una API REST, las vistas son las respuestas JSON generadas por los controladores.

Controlador: Actúa como intermediario entre el Modelo y la Vista. Los controladores manejan las solicitudes de los usuarios y toman decisiones sobre qué datos se deben mostrar y cómo se debe mostrar. En una aplicación Spring Boot, los controladores suelen ser clases anotadas con @Controller para páginas web o @RestController para API REST.

A continuación se muestra un esquema de cómo se representa este modelo en Spring Boot:



Repositorios JPA(DAO):
Los repositorios JPA proporcionan una forma eficiente de interactuar con la base de datos a través de objetos Java. En Spring Boot, los repositorios JPA se crean mediante interfaces que extienden JpaRepository o una interfaz personalizada que extiende CrudRepository. Estas interfaces se encargan de proporcionar métodos para realizar operaciones de CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades relacionadas con la base de datos.

Services:
Los servicios son componentes que encapsulan la lógica de negocio de la aplicación. Están diseñados para manejar la lógica entre los controladores y los repositorios. Los servicios utilizan los repositorios JPA para acceder y manipular datos y aplican la lógica de negocio según sea necesario. En Spring Boot, los servicios suelen estar anotados con @Service.

RestControllers:
Los RestControllers son controladores especializados en la construcción de API REST. Están diseñados para recibir solicitudes HTTP (como GET, POST, PUT, DELETE) y devolver respuestas en formato JSON o XML. Estos controladores se anotan con @RestController y exponen endpoints que son accesibles a través de URL.

La **secuencia típica de una solicitud en una aplicación Spring Boot** basada en este patrón de diseño sería la siguiente:

1. El usuario realiza una solicitud HTTP, por ejemplo, una solicitud GET a una URL específica.
2. El controlador correspondiente recibe la solicitud y decide qué servicio invocar.
3. El servicio se comunica con los repositorios JPA para acceder a los datos necesarios.
4. El servicio procesa los datos según la lógica de negocio.
5. El servicio devuelve los datos al controlador.
6. El controlador convierte los datos en una respuesta JSON y la envía de vuelta al usuario.

Este patrón de diseño facilita la modularidad y el mantenimiento de la aplicación al separar claramente las responsabilidades de cada componente. Además, Spring Boot proporciona una integración sencilla de estos componentes a través de la inversión de control y la inyección de dependencias, lo que hace que el desarrollo sea más eficiente y escalable.

Programación de CRUD completo

A continuación vamos a explicar paso a paso cómo se programaría una funcionalidad completa en Spring Boot , basándonos en el modelo arquitectónico y de diseño previamente citado. Para ello partiremos de una estructura previa de tablas de base de datos y llegaremos desde el modelo al controlador pasando por todas las capas.

Base de datos

Partiremos de la siguiente estructura de tablas en nuestra base de datos:

Tabla de Base de Datos: **Libro**

```
CREATE TABLE libro (
    id INT PRIMARY KEY,
    titulo VARCHAR(100),
    isbn VARCHAR(13),
    anyo_publicacion INT,
    editorial VARCHAR(50),
    id_autor INT,
    FOREIGN KEY (autor_id) REFERENCES Autor(id)
);
```

Tabla de Base de Datos: **Autor**

```
CREATE TABLE autor (
    id INT PRIMARY KEY,
    nombre VARCHAR(50),
    nacionalidad VARCHAR(30),
    fecha_nacimiento DATE
);
```

Modelos

Nuestro primer paso es convertir estas tablas en clases Java, que actuarán como modelos persistentes, siendo Entidades JPA. A continuación se detallan las normas que seguiremos a la hora de transformar nuestra tabla de base de datos en modelo.

- Cada tabla de nuestra base de datos se convierte en una entidad JPA, a excepción de tablas relación que no tengan columnas más allá de las respectivas columnas que representan las foreign key.
- Para representar la entidad se crea una clase Java con el nombre de nuestra tabla, y se le incluyen las anotaciones:
    - @Entity : para declararla como entidad persistente.
    - @Table : para enlazar nuestra entidad con la tabla de base de datos correspondiente.
- Cada columna de nuestra tabla de base de datos se convierte en un atributo de la clase. Los atributos deben tener un tipo similar al de base de datos, por lo que se usarán tipos Java acordes a los mismos. A cada atributo se le pone la anotación @Column para enlazarlo con la columna de base de datos que representa. Con el atributo que representa a lo columna id, hay usar también la anotación @Id , y si va a ser un campo autogenerado @GeneratedValue, con la estrategia de generación que vayamos a usar.
- Para las relaciones seguiremos las siguientes reglas: Para representar las relaciones desde una entidad hacia una cardinalidad “N” (Many) se utiliza un atributo Set<OtraEntidad>, y para representar de una entidad hacia otra con cardinalidad 1 se utiliza un objeto del tipo de la “OtraEntidad”. Y las anotaciones correspondientes a cada relación de las que mostramos a continuación:
    - **@OneToOne:** Anota un campo en una entidad para establecer una relación uno a uno con otra entidad. Esto significa que un objeto de una entidad está relacionado con un solo objeto de otra entidad.
    - **@OneToMany:**Anota un campo en una entidad para establecer una relación uno a muchos con otra entidad. Esto implica que un objeto de una entidad puede estar relacionado con varios objetos de otra entidad.
    - **@ManyToOne:**Anota un campo en una entidad para establecer una relación muchos a uno con otra entidad. Esto significa que varios objetos de la entidad actual están relacionados con un solo objeto de otra entidad.
    - **@ManyToMany:** Anota un campo en una entidad para establecer una relación muchos a muchos con otra entidad. En este tipo de relación, varios objetos de una entidad están relacionados con varios objetos de otra entidad.
    - **@JoinTable**:Usada en conjunto con @ManyToMany, esta anotación define una tabla intermedia para representar la relación muchos a muchos.

Puedes especificar el nombre de la tabla intermedia y las columnas de unión utilizando @JoinTable.

- **@JoinColumn**: Permite especificar la columna que se utilizará como clave foránea (FK) en una relación entre entidades. Esto es útil cuando deseas personalizar el nombre de la columna en lugar de utilizar el nombre predeterminado.
- **@OneToMany(mappedBy = "nombreCampo")**:
  Se utiliza en la entidad inversa de una relación bidireccional @OneToMany para indicar cuál campo de la entidad principal se asocia con esta entidad. Esto se utiliza para definir la inversa de una relación uno a muchos.
- **@ManyToOne(fetch = FetchType.LAZY)**:Controla cómo se carga la entidad relacionada. FetchType.LAZY significa que la entidad relacionada se cargará sólo cuando se acceda a ella explícitamente. FetchType.EAGER cargará la entidad relacionada automáticamente cuando se recupere la entidad principal.
- **@Cascade:** Se utiliza para especificar las operaciones de cascada que deben aplicarse a la entidad relacionada cuando se realiza una operación en la entidad principal. Por ejemplo, @Cascade(CascadeType.PERSIST) significa que cuando se guarda la entidad principal, se guardará automáticamente la entidad relacionada.
- **@Fetch:** Controla cómo se carga la entidad relacionada en una consulta. Puedes especificar @Fetch(FetchMode.JOIN) para cargar la entidad relacionada mediante una consulta JOIN, o @Fetch(FetchMode.SELECT) para cargarla mediante una consulta SELECT.

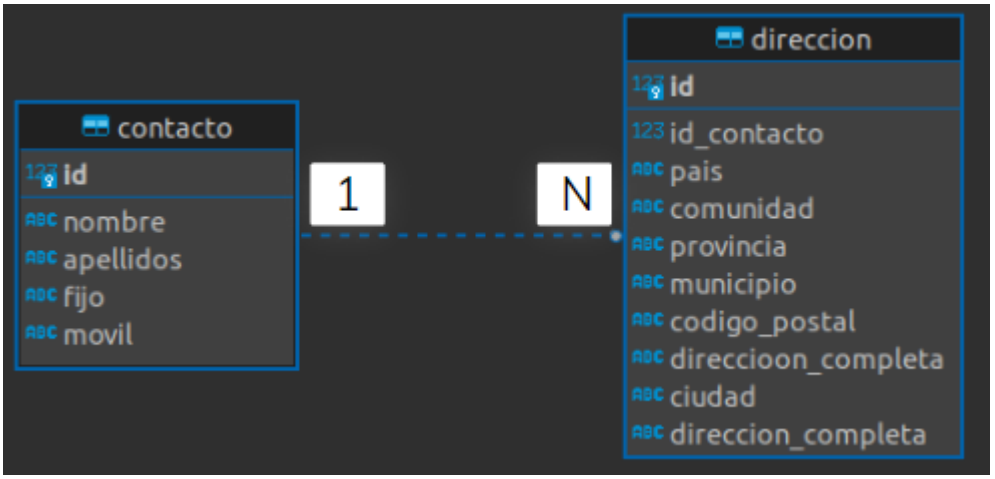
A continuación se muestra como quedarían los modelos siguiente estás reglas y basándonos en las tablas de base de datos que tomamos como punto de partida:

```java
@Entity
@Table(name = "libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "titulo", length = 100, nullable = false)
    private String titulo;

    @Column(name = "isbn", length = 13, unique = true, nullable = false)
    private String isbn;

    @Column(name = "anyo_publicacion")
    private int anyoPublicacion;

    @Column(name = "editorial", length = 50)
    private String editorial;

    @OneToOne @JoinColumn(name = "id_autor")
    private Autor autor;
    // Getters y Setters
}
```

```java
@Entity
@Table(name = "autor")
public class Autor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nombre", length = 50, nullable = false)
    private String nombre;

    @Column(name = "nacionalidad", length = 30)
    private String nacionalidad;

    @Column(name = "fecha_nacimiento")
    private LocalDate fechaNacimiento;

    @OneToMany(mappedBy = "autor")
    private Set<Libro> libros = new HashSet<>();
    // Getters y Setters
}
```

Y también mostraremos ejemplos de cómo se implementarían los tres tipos básicos de relaciones , con sus respectivos ejemplos de base de datos:

One to Many (1:N)



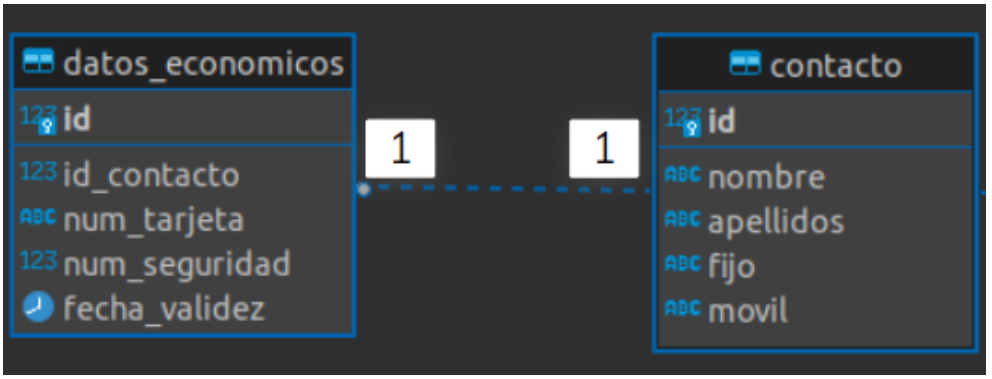

```
@OneToMany(cascade = CascadeType.ALL , mappedBy = "contacto" , fetch = FetchType.LAZY)
private Set<Direccion> direcciones = new HashSet<>( initialCapacity: 0);
```

En la clase Contacto

```
@ManyToOne
@JoinColumn(name = "id_contacto")
private Contacto contacto;
```

En la clase Dirección

One to One(1:1)



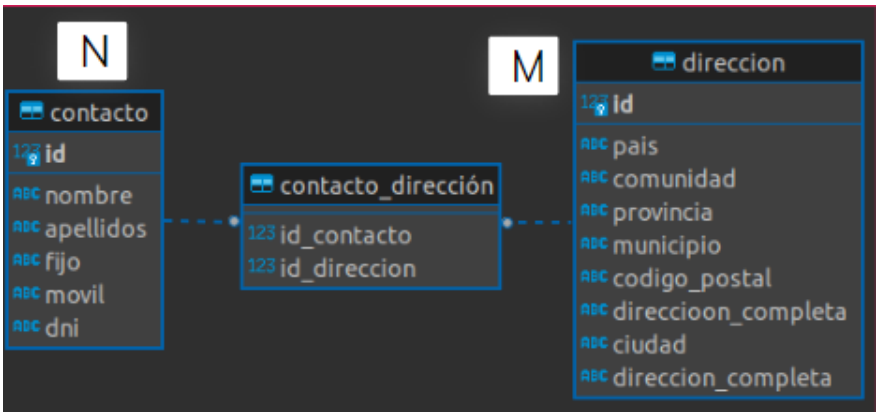

```
@OneToOne(cascade = CascadeType.ALL , mappedBy = "contacto" , fetch = FetchType.LAZY)
private DatosEconomicos datosEconomicos;
```

En la clase Contacto

```
@OneToOne
@JoinColumn(name = "id_contacto", nullable = false)
private Contacto contacto;
```

En la clase DatosEconomicos

Many To Many (N:M)




```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinTable(name = "contacto_direccion",
        joinColumns ={@JoinColumn(name = "id_contacto",nullable = false)},
        inverseJoinColumns = {@JoinColumn(name = "id_direccion", nullable = false)})
@JsonManagedReference
private Set<Direccion> direcciones = new HashSet<>( initialCapacity: 0);
```

En la clase Contacto

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinTable(name = "contacto_direccion",
        joinColumns ={@JoinColumn(name = "id_direccion",nullable = false)},
        inverseJoinColumns = {@JoinColumn(name = "id_contacto", nullable = false)})
@JsonManagedReference
private Set<Contacto> contactos = new HashSet<>( initialCapacity: 0);
```

En la clase Dirección

Repositorios

Los repositorios en Spring Boot son los encargados de gestionar la capa de acceso a datos, proporcionan una interfaz extensa de métodos para realizar consultas, inserciones , modificaciones y eliminaciones.

Uno de los tipos de Repositorios más utilizados son los JPA Repositories, proporcionados por Spring Data JPA. Los JPA Repositories son interfaces que proporcionan un conjunto de métodos predefinidos para realizar operaciones de persistencia en una base de datos utilizando JPA. Estos repositorios facilitan la gestión de entidades JPA y simplifican la escritura de consultas personalizadas.

Los JPA Repositories son especialmente útiles en el contexto de aplicaciones Spring Boot, ya que simplifican en gran medida el acceso y manipulación de datos en la base de datos. Algunas de las ventajas que proporcionan son las siguientes:

- **Operaciones CRUD Predefinidas**: Los JPA Repositories proporcionan métodos predefinidos para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) en las entidades JPA. Estos métodos incluyen save, findById, findAll, delete, entre otros.
- **Generación de Consultas Automáticas**: Los JPA Repositories utilizan la convención de nombres de métodos para generar consultas SQL automáticamente a partir de los nombres de los métodos y los atributos de la entidad. Esto significa que puedes realizar consultas sin necesidad de escribir SQL manualmente.
- **Consultas Personalizadas**: Además de las operaciones CRUD predefinidas, puedes definir tus propios métodos en los repositorios JPA para realizar consultas personalizadas utilizando la anotación @Query. Esto te permite realizar consultas más complejas y adaptadas a tus necesidades específicas.
- **Paginación y Ordenamiento**: Los JPA Repositories facilitan la paginación y el ordenamiento de los resultados de las consultas. Puedes especificar el número de elementos por página y el orden de los resultados de manera sencilla.
- **Soporte para Transacciones**: Los JPA Repositories se integran perfectamente con el sistema de transacciones de Spring, lo que garantiza que las operaciones en la base de datos se realicen dentro de una transacción, lo que es esencial para mantener la consistencia de los datos.
- **Abstracción de la Base de Datos**: Los JPA Repositories ocultan la complejidad subyacente de la base de datos, permitiéndote interactuar con las entidades JPA de manera más orientada a objetos en lugar de trabajar directamente con SQL.

Se creará un repositorio por cada Entity haciendo que el repositorio extienda de JPARepository, indicando la Entity que cubre el repositorio y el tipo de dato que sea el id de esa Entity.

Para los repositorios en lugar de una clase , usaremos una interfaz. Ya que no implementaremos métodos , sino que usaremos la propia interfaz que nos proporciona JPA junto con las anotaciones de Spring.

En la cabecera de la interfaz utilizaremos la notación @Repository.

Así quedarían los repositorios para las Entities que generamos anteriormente.

```
@Repository
public interface LibroRepository extends JpaRepository<Libro, Long> {
}
```

```
@Repository
public interface AutorRepository extends JpaRepository<Autor, Long> {
}
```

Como hemos comentado uno de los objetivos de los JPA Repositories es proporcionar una interfaz de acceso a datos. En cuanto a consultas se refiere, contamos con tres opciones de métodos de consulta:

Consultas JPA:

Las consultas JPA se basan en los métodos de repositorio que siguen la convención de nombres proporcionada por Spring Data JPA. Estos métodos generan consultas SQL automáticamente a partir de los nombres de los métodos y los atributos de la entidad. Por ejemplo, **findByTitulo**(String titulo) generará una consulta SQL para buscar libros por su título.

Estas consultas son simples de usar y adecuadas para consultas comunes y sencillas.

A continuación mostramos algunos ejemplos:

```java
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface LibroRepository extends Repository<Libro, Long> {

    List<Libro> findByEditorialAndAnyooPublicacion(String editorial, int anyoPublicacion);
    List<Libro> findDistinctByEditorialOrTitulo(String editorial, String titulo);
    List<Libro> findDistinctByEditorialOrTitulo(String editorial, String titulo);
    List<Libro> findByEditorialIgnoreCase(String editorial);
    List<Libro> findByEditorialAndTituloAllIgnoreCase(String editorial, String titulo);
    List<Libro> findByEditorialOrderByTituloAsc(String editorial);
    List<Libro> findByEditorialOrderByTituloDesc(String editorial);
}
```

En este ejemplo:

- **findByEditorialAndAnyoPublicacion**: busca libros por editorial y año de publicación.
- **findDistinctByEditorialOrTitulo** y **findDistinctByEditorialOrTitulo** realizan búsquedas por editorial o título, habilitando la opción distinct para evitar duplicados.
- **findByEditorialIgnoreCase** busca libros por editorial sin importar mayúsculas y minúsculas.
- **findByEditorialAndTituloAllIgnoreCase** busca libros por editorial y título, sin importar mayúsculas y minúsculas en ambos campos.
- **findByEditorialOrderByTituloAsc** y **findByEditorialOrderByTituloDesc** permiten ordenar los resultados por título en orden ascendente o descendente.

Para más información aquí dejamos varios enlaces de páginas de documentación:

- https://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
- https://www.baeldung.com/spring-data-jpa-query

Consultas JPQL (Java Persistence Query Language):

JPQL es un lenguaje de consulta orientado a objetos similar a SQL, pero que se utiliza para consultar objetos en lugar de tablas de base de datos. Puedes definir tus propias consultas JPQL en anotaciones @Query en tus repositorios JPA.

Las consultas JPQL te permiten realizar consultas más complejas y personalizadas, así como un mayor control sobre los resultados de las consultas.

```
@Query("SELECT l FROM Libro l WHERE l.editorial = :editorial")
public List<Libro> findLibrosPorEditorial(@Param("editorial") String
editorial);
```

Aquí dejamos también un enlace con la documentación del lenguaje JPQL:

- https://docs.oracle.com/cd/E11035_01/kodo41/full/html/ejb3_langref.html

Consultas Nativas:

Las consultas nativas son consultas SQL directas que se ejecutan en la base de datos. Puedes utilizar consultas nativas en repositorios JPA mediante la anotación @Query con la opción nativeQuery = true.

Las consultas nativas son útiles cuando necesitas ejecutar consultas SQL específicas de la base de datos que no se pueden lograr fácilmente con JPQL o cuando deseas un mayor rendimiento en consultas complejas.

```
@Query(value = "SELECT * FROM Libro WHERE editorial = :editorial",
nativeQuery = true)
public List<Libro> findLibrosPorEditorialNativa(@Param("editorial")
String editorial);
```

Servicios

En una arquitectura típica de una aplicación Spring Boot con Spring Data JPA, los servicios desempeñan un papel fundamental. Los servicios actúan como intermediarios entre los controladores (que manejan las solicitudes HTTP) y los repositorios (que interactúan con la base de datos). Los servicios contienen la lógica de negocio de la aplicación y coordinan las operaciones en la base de datos a través de los repositorios.

A continuación vemos de forma detallada sus funciones:

1. **Lógica de Negocio Centralizada**: los servicios actúan como un componente intermedio entre los controladores y los repositorios de datos. Su principal función es centralizar y encapsular la lógica de negocio de la aplicación. Esto significa que los servicios contienen todas las reglas y operaciones específicas de la aplicación que no están relacionadas directamente con el acceso a la base de datos.
2. **Coordinación de Repositorios**: los servicios coordinan y gestionan las operaciones de acceso a la base de datos. Esto implica la utilización de repositorios JPA para realizar operaciones CRUD en la base de datos. Los servicios utilizan métodos proporcionados por los repositorios para buscar, guardar, actualizar y eliminar registros en la base de datos.
3. **Abstracción de la Capa de Controladores**: una de las ventajas clave de los servicios es que separan la capa de controladores de la lógica de negocio. Esto permite que los controladores se centren en el manejo de las solicitudes HTTP, la validación de entrada y la gestión de las respuestas, mientras que la lógica de negocio se encarga de manera centralizada en los servicios.
4. **Reutilización de Lógica**: los servicios promueven la reutilización de la lógica de negocio en toda la aplicación. Esto significa que una operación de negocio que se necesita en varios lugares de la aplicación se implementa una vez en un servicio y se puede utilizar en varios controladores.
5. **Flexibilidad y Mantenibilidad**: al separar la lógica de negocio en servicios, se logra una mayor flexibilidad y mantenibilidad. Los cambios en las reglas de negocio se realizan en un solo lugar, lo que facilita la modificación y actualización de la lógica de la aplicación sin afectar a los controladores ni a los repositorios.
6. **Pruebas Unitarias**: los servicios son una unidad ideal para la realización de pruebas unitarias. Al encapsular la lógica de negocio en servicios, se pueden escribir pruebas unitarias para garantizar que la lógica de la aplicación funcione correctamente sin depender de la capa de controladores ni de la base de datos.
7. **Transacciones y Seguridad**: los servicios son el lugar adecuado para gestionar transacciones y seguridad. Puedes anotar métodos de servicio con anotaciones de transacción para asegurarse de que las operaciones se ejecuten dentro de una transacción. Además, puedes implementar lógica de seguridad en los servicios para restringir el acceso a ciertas operaciones de negocio.

Aquí mostramos ejemplos de cómo podrían implementarse los servicios en base a los modelos que hemos desarrollado anteriormente:

```java
@Service
public class LibroService {

    @Autowired
    private LibroRepository libroRepository;

    public List<Libro> buscarLibrosPorEditorial(String editorial) {
        return libroRepository.findByEditorial(editorial);
    }

    public void guardarLibro(Libro libro) {
        libroRepository.save(libro);
    }

    public void eliminarLibro(Long id) {
        libroRepository.deleteById(id);
    }
}
```

```java
@Service
public class AutorService {
    @Autowired
    private AutorRepository autorRepository;

    public List<Autor> buscarAutoresPorNacionalidad(String nacionalidad)
{
        return autorRepository.findByNacionalidad(nacionalidad);
    }

    public void guardarAutor(Autor autor) {
        autorRepository.save(autor);
    }

    public void eliminarAutor(Long id) {
        autorRepository.deleteById(id);
    }
}
```

Controladores

Los controladores en una aplicación Spring Boot desempeñan un papel fundamental en la gestión de las solicitudes HTTP y la presentación de respuestas al cliente. Son responsables de recibir las solicitudes HTTP, interactuar con los servicios para realizar operaciones y devolver las respuestas adecuadas al cliente. Aquí te proporcionaré una explicación detallada de la parte de los controladores en esta arquitectura:

1. **Gestión de Solicitudes HTTP**: los controladores actúan como puertas de entrada para las solicitudes HTTP entrantes. Cada método en un controlador puede ser mapeado a una ruta URL específica utilizando anotaciones como @RequestMapping, @GetMapping, @PostMapping, etc. Estos métodos se ejecutan cuando se accede a la URL correspondiente, lo que permite gestionar las solicitudes entrantes.
2. **Validación de Entrada:** antes de procesar las solicitudes, los controladores pueden realizar validaciones de entrada para garantizar que los datos proporcionados por el cliente sean correctos y seguros. Pueden usar anotaciones de validación, como @RequestParam, @PathVariable, o @RequestBody, para recibir datos y validarlos.
3. **Llamadas a Servicios**: una de las principales funciones de los controladores es interactuar con los servicios de la aplicación. Los controladores llaman a métodos en los servicios para realizar operaciones de negocio. Esto incluye la creación, lectura, actualización y eliminación de datos en la base de datos, así como la ejecución de cualquier lógica de negocio relacionada.
4. **Conversión y Transformación de Datos**: los controladores también pueden ser responsables de convertir y transformar datos para adaptarlos a las necesidades del cliente. Esto puede incluir la conversión de objetos de dominio en DTOs (Objetos de Transferencia de Datos) que se envían al cliente, o la transformación de respuestas en diferentes formatos, como JSON o XML.
5. **Gestión de Respuestas**: los controladores son responsables de devolver respuestas adecuadas al cliente. Esto implica la creación de objetos de respuesta que pueden incluir datos y metadatos, como códigos de estado HTTP, encabezados y el cuerpo de la respuesta. Los controladores pueden devolver diferentes tipos de respuestas, como vistas HTML, JSON, XML, etc.
6. **Enrutamiento y Gestión de Vistas**: en el caso de aplicaciones web, los controladores también pueden gestionar la navegación y las vistas. Pueden devolver vistas HTML o redirigir a otras rutas en función de la lógica de la aplicación y las acciones del usuario.
7. **Excepciones y Control de Errores**: los controladores deben manejar excepciones y errores de manera adecuada. Pueden utilizar manejo de excepciones personalizado para responder a situaciones de error de manera significativa y proporcionar mensajes de error al cliente.

8. **Interacción con la Capa de Presentación**: en una arquitectura de aplicación típica, los controladores no deben contener lógica de negocio ni acceso directo a la base de datos. En cambio, interactúan con la capa de servicios para separar la lógica de negocio de la gestión de solicitudes y respuestas.
9. **Pruebas Unitarias**: los controladores son unidades ideales para la realización de pruebas unitarias. Se pueden probar por separado utilizando marcos de pruebas como JUnit o TestNG para garantizar que respondan adecuadamente a las solicitudes y generen las respuestas correctas.

Aquí mostramos ejemplos para los modelos que teníamos

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/libros")
public class LibroController {

    @Autowired
    private LibroService libroService;

    // Endpoint para obtener todos los libros
    @GetMapping
    public List<Libro> obtenerLibros() {
        return libroService.obtenerTodosLosLibros();
    }

    // Endpoint para obtener un libro por su ID
    @GetMapping("/{id}")
    public Libro obtenerLibroPorId(@PathVariable Long id) {
        return libroService.obtenerLibroPorId(id);
    }

    // Endpoint para crear un nuevo libro
    @PostMapping
    public Libro crearLibro(@RequestBody Libro libro) {
        return libroService.crearLibro(libro);
    }

    // Endpoint para actualizar un libro por su ID
    @PutMapping("/{id}")
    public Libro actualizarLibro(@PathVariable Long id, @RequestBody Libro libro) {
        return libroService.actualizarLibro(id, libro);
    }

    // Endpoint para eliminar un libro por su ID
    @DeleteMapping("/{id}")
    public void eliminarLibro(@PathVariable Long id) {
        libroService.eliminarLibro(id);
    }
}
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/autores")
public class AutorController {

    @Autowired
    private AutorService autorService;

    // Endpoint para obtener todos los autores
    @GetMapping
    public List<Autor> obtenerAutores() {
        return autorService.obtenerTodosLosAutores();
    }

    // Endpoint para obtener un autor por su ID
    @GetMapping("/{id}")
    public Autor obtenerAutorPorId(@PathVariable Long id) {
        return autorService.obtenerAutorPorId(id);
    }

    // Endpoint para crear un nuevo autor
    @PostMapping
    public Autor crearAutor(@RequestBody Autor autor) {
        return autorService.crearAutor(autor);
    }

    // Endpoint para actualizar un autor por su ID
    @PutMapping("/{id}")
    public Autor actualizarAutor(@PathVariable Long id, @RequestBody Autor autor) {
        return autorService.actualizarAutor(id, autor);
    }

    // Endpoint para eliminar un autor por su ID
    @DeleteMapping("/{id}")
    public void eliminarAutor(@PathVariable Long id) {
        autorService.eliminarAutor(id);
    }
}
```

DTOs

Los DTO (Data Transfer Objects) son objetos utilizados para transferir datos entre diferentes capas de una aplicación, especialmente entre la capa de controladores y la capa de presentación (por ejemplo, la interfaz de usuario). Es altamente recomendable a la hora de trabajar con APIs que la transmisión de información en los cuerpos de las peticiones se realice mediante DTOs en lugar de Entities, de esta manera se devuelven y reciben los datos específicos necesarios para cada petición o función completa, pudiendo personalizar nuestro DTO según nuestras necesidades en todo momento.

Utilidad de los DTOs:

- **Separación de Responsabilidades**: los DTOs ayudan a separar las responsabilidades entre la capa de controladores y la capa de servicios. Los controladores se encargan de manejar las solicitudes y respuestas HTTP, mientras que los servicios se centran en la lógica de negocio y la interacción con las entidades de dominio.
- **Optimización de Datos**: los DTOs permiten enviar únicamente la información necesaria desde la capa de servicios hasta la capa de presentación. Esto puede reducir la cantidad de datos transferidos a través de la red y acelerar la respuesta de la aplicación.
- **Seguridad y Privacidad**: al utilizar DTOs, puedes ocultar ciertos atributos de las entidades de dominio que no deben ser accesibles públicamente a través de las API. Esto mejora la seguridad y la privacidad de los datos de tu aplicación.
- **Versionado de API**: los DTOs también son útiles cuando necesitas realizar cambios en las estructuras de datos sin afectar a las versiones anteriores de tu API. Puedes crear nuevos DTOs con los cambios necesarios sin alterar las estructuras de entidades de dominio subyacentes.

Mejores Prácticas para Usar DTOs:

- **Nombre Descriptivo**: da a tus DTOs nombres descriptivos que reflejen su propósito y los datos que contienen. Esto facilita la comprensión de su uso en la aplicación.
- **Evita la Lógica de Negocio en DTOs**: los DTOs deben contener exclusivamente datos y no deben incluir lógica de negocio. Deja la lógica de negocio en la capa de servicios.
- **Conversión entre Entidades y DTOs**: Utiliza métodos de conversión o bibliotecas de mapeo (por ejemplo, ModelMapper o MapStruct) para convertir entre entidades de dominio y DTOs de manera eficiente.
- **Validación de Datos**: Asegúrate de que los datos en los DTOs sean válidos antes de procesarlos en la capa de servicios. Puedes utilizar anotaciones de validación,

como las proporcionadas por Hibernate Validator, para verificar la integridad de los datos.

- **Documentación**: documenta tus DTOs adecuadamente para que otros desarrolladores comprendan su estructura y uso. Esto es especialmente importante cuando se trabaja en equipos.
- **Pruebas Unitarias**: también puedes escribir pruebas unitarias para los métodos que involucran la conversión entre entidades y DTOs para garantizar que los datos se manejen correctamente.

Ejemplos:

```
import lombok.Getter;
import lombok.Setter;

@Data
@Builder
public class LibroDTO {
    private Long id;
    private String titulo;
    private String isbn;
    private int anyoPublicacion;
    private String editorial;
    private AutorDTO autor;

}
```

Como podemos apreciar, el DTO suele contener los datos correspondiente a nuestra entity, pero sin obligación de que aparezcan todos ellos , ni de que necesariamente se usen los mismos tipos de datos.

Por ejemplo un cambio en los tipos de datos en cuanto a DTO y Entity habitual es el cambio de las fechas, en las entities suelen ser del tipo “LocalDate” o “LocalDateTime” sin embargo en los DTO se suele usar el tipo “String” , transformando la fecha al formato usado por el usuario.

Otro cambio habitual es el de las relaciones, que se representan en las Entity mediante un Objeto o colección de objetos, sin embargo en los DTO se suelen incluir solo alguno de sus campos y habitualmente para representarlos se suele usar un DTO dentro de otro. En algunos casos incluso sólo se indica el identificador de la relación en sí porque lo único que queramos hacer sea insertar/relacionar/asignar información.

Para el paso de DTO a Entity y viceversa en lugar de hacer las conversiones de los datos de manera manual , se suelen utilizar mapeadores. En la siguiente sección explicaremos cómo se utilizan y las diferentes opciones con las que contamos.

Conversores/Mapeadores

Un "mapper" (mapeador) en el contexto de la programación se refiere a un componente o función que se utiliza para realizar mapeos o conversiones entre diferentes tipos de datos o estructuras. La idea principal detrás de un mapper es tomar datos en un formato o estructura y transformarlos en otro formato o estructura, de manera que los datos sean compatibles con el uso previsto.

En el contexto de mapeo de objetos, un "mapper" generalmente se utiliza para convertir datos de un objeto de un tipo a otro. Por ejemplo, puedes tener un objeto de tipo A y desear convertirlo en un objeto de tipo B, y un "mapper" se encargaría de realizar esa conversión.

Para nuestras aplicaciones Spring Boot los usaremos principalmente para transformar de DTO a Entity y viceversa.

Algunos ejemplos de situaciones en las que se utilizan mappers incluyen:

- **Mapeo de objetos a bases de datos**: Convertir objetos de tu aplicación a registros de una base de datos y viceversa.
- **Serialización y deserialización**: Convertir objetos Java en formatos de datos como JSON o XML y viceversa.
- **Integración de servicios web**: Convertir datos entre los formatos utilizados por una API web y los objetos de tu aplicación.
- **Mapeo entre modelos de datos**: Convertir datos entre diferentes modelos de datos dentro de una aplicación.

Los mappers son especialmente útiles en situaciones en las que los datos deben viajar entre diferentes partes de una aplicación o entre diferentes sistemas. Ayudan a garantizar que los datos se transformen de manera correcta y coherente, lo que facilita la interoperabilidad y la comunicación entre componentes o sistemas heterogéneos.

Los mappers pueden ser implementados manualmente escribiendo código para realizar las conversiones, o puedes utilizar bibliotecas o herramientas específicas, como MapStruct en Java o AutoMapper en C#, que simplifican la implementación de mappers y generan automáticamente código de mapeo basado en convenciones o anotaciones. Estas herramientas ahorran tiempo y reducen la posibilidad de errores humanos en las conversiones de datos.

A continuación hablaremos de MapStruct, herramienta que utilizaremos en nuestra aplicación Java para la conversión de la que venimos hablando.

MapStruct

MapStruct es una biblioteca de generación de código en Java que se utiliza para simplificar el proceso de mapeo entre objetos Java. Su objetivo principal es automatizar la generación de código de mapeo eficiente en tiempo de compilación, lo que ahorra tiempo de desarrollo y reduce la probabilidad de errores en las conversiones de datos.

Las principales características y ventajas de MapStruct son las siguientes:

- **Generación de código en tiempo de compilación**: MapStruct genera automáticamente implementaciones de mapeo en tiempo de compilación, lo que significa que no se incurre en sobrecarga de reflexión o rendimiento en tiempo de ejecución al realizar conversiones de datos.
- **Facilidad de uso**: Utiliza anotaciones en interfaces Java para definir cómo deben realizarse las conversiones de datos entre diferentes tipos de objetos. Esto hace que la configuración sea clara y legible.
- **Tipo de datos seguros**: MapStruct ofrece seguridad de tipos durante las conversiones, lo que ayuda a reducir los errores en tiempo de compilación.
- **Mapeo personalizado**: Aunque MapStruct genera código de mapeo automáticamente en la mayoría de los casos, también permite la personalización mediante métodos definidos manualmente en las interfaces de mapeo.
- **Soporte para múltiples formatos de datos**: MapStruct se utiliza comúnmente para mapear entre objetos Java y formatos de datos como JSON, XML, bases de datos y otros.
- **Integración con herramientas de compilación**: Puedes configurar fácilmente tu proyecto para que MapStruct genere las implementaciones de mapeo durante el proceso de compilación, lo que facilita su uso en proyectos Java estándar.

El uso de MapStruct puede ser especialmente beneficioso en aplicaciones donde el mapeo de datos es una tarea común y donde se necesita un alto rendimiento en las conversiones. Al generar código de mapeo en tiempo de compilación, MapStruct ayuda a evitar errores sutiles y mejora la eficiencia de las conversiones de datos, lo que lo convierte en una herramienta valiosa en el desarrollo de software en Java.

A continuación se muestran los pasos que deberíamos realizar en nuestro proyecto para el uso simple de MapStruct.

Paso 1: Agregar Dependencias

Agrega las dependencias necesarias de MapStruct a tu proyecto. En tu archivo pom.xml (si usas Maven) o build.gradle (si usas Gradle), incluye las dependencias de MapStruct y el procesador de anotaciones:

```
<dependencies>
    <!-- Otras dependencias -->
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>{version}</version>
    </dependency>
</dependencies>
```

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>{version}</version>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>{version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Paso 2: Realización de Mapeadores

Construye los mapeadores mediante interfaces Java, indicando los métodos de conversión que necesites, posteriormente en tiempo de compilación MapStruct se encargará de implementarlos y generarlos automáticamente guiándose por las anotaciones que utilices.

Por ejemplo, supongamos que tienes una entidad Libro y un DTO LibroDTO. Puedes usar las anotaciones @Mapper y @Mapping para definir un mapeador personalizado entre ellos:

```
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;

@Mapper
public interface LibroMapper {

    @Mapping(target = "titulo", source = "nombre")
    LibroDTO libroToLibroDTO(Libro libro);
}
```

Este es un ejemplo simple, pero se nos puede complicar si nos encontramos con atributos que se llaman diferente en DTO y Entity , necesiten una transformación de tipos o algún método específico de conversión.
Aquí un ejemplo:

```
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.Mappings;

@Mapper
public interface LibroMapper {

    @Mappings({
        @Mapping(target = "titulo", source = "nombre"),
        @Mapping(target = "fechaPublicacion", dateFormat = "dd-MM-yyyy", source = "fechaPublicacion"),
        @Mapping(target = "cantidadPaginas", source = "libro", qualifiedByName = "calcularPaginas")
    })
    LibroDTO libroToLibroDTO(Libro libro);

    @Mappings({
        @Mapping(target = "nombre", source = "titulo"),
        @Mapping(target = "fechaPublicacion", dateFormat = "dd-MM-yyyy", source = "fechaPublicacion"),
        @Mapping(target = "libro", qualifiedByName = "calcularPaginasInverso")
    })
    Libro libroDTOToLibro(LibroDTO libroDTO);

    default String calcularPaginas(Libro libro) {
        return "Este libro tiene " + libro.getPaginas() + " páginas.";
    }

    default String calcularPaginasInverso(LibroDTO libroDTO) {
        // Implementa la lógica inversa para calcular las páginas de un libro a partir del DTO
        // Esto dependerá de tus requerimientos específicos.
        return null;
    }
}
```

En este ejemplo:

- Usamos @Mappings para agrupar las anotaciones @Mapping y definir múltiples mapeos de una vez.
- Convertimos el atributo nombre de la entidad Libro en el atributo título del DTO.
- Formateamos la fecha de publicación en el formato "dd-MM-yyyy" tanto para la entidad Libro como para el DTO.
- Utilizamos el método por defecto calcularPaginas para calcular la cantidad de páginas del libro en el DTO. Este método se llama automáticamente al mapear un Libro a un LibroDTO.
- También definimos un método por defecto inverso calcularPaginasInverso que se utilizará al realizar la conversión inversa de LibroDTO a Libro.

En el caso de tener que utilizar services la cosa cambia, ya que para realizar la inyección del servicio en nuestro constructor deberemos usar clases abstractas y convertir nuestros métodos a implementar por mapstruct en abstractos también.

Supongamos que tienes una entidad Libro que tiene una referencia al autor mediante su ID y un DTO LibroDTO que contiene el ID del autor. Aquí está un ejemplo de cómo configurar el mapeador LibroMapper para recuperar automáticamente los datos del autor del servicio y realizar la conversión:

```
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.Named;
import org.springframework.beans.factory.annotation.Autowired;

@Mapper(componentModel = "spring")
public abstract class LibroMapper {

    @Autowired
    private AutorService autorService;

    @Mapping(target = "autor", source = "id_autor", qualifiedByName = "mapToAutor")
    public abstract Libro libroDTOToLibro(LibroDTO libroDTO);

    @Named("mapToAutor")
    public Autor mapToAutor(LibroDTO libroDTO) {
        if (libroDTO.getIdAutor() != null) {
            return autorService.obtenerAutorPorId(libroDTO.getIdAutor());
        }
        return null;
    }
}
```

En este ejemplo:

- Hemos definido un método con nombre mapToAutor y lo hemos calificado con @Named("mapToAutor"). Este método toma un objeto LibroDTO y utiliza el servicio AutorService para obtener el autor correspondiente.
- En el método libroDTOToLibro, hemos utilizado @Mapping para especificar que el atributo autor en la entidad Libro debe obtenerse llamando al método con nombre mapToAutor. La anotación qualifiedByName se utiliza para indicar el nombre del método de mapeo que debe ejecutarse.

Este enfoque permite que MapStruct utilice el método con nombre mapToAutor para convertir el atributo autor en la entidad Libro basándose en la información proporcionada en el LibroDTO.

Asegúrate de que la configuración del componente Spring (componentModel = "spring") esté en su lugar para habilitar la inyección de dependencias en la clase LibroMapper.

Librerías Recomendadas

Lombok

Project Lombok es una biblioteca de Java que se utiliza para simplificar el desarrollo de software al reducir la necesidad de escribir cierto código boilerplate (código repetitivo y de uso común). Lombok se integra con el compilador de Java y ofrece anotaciones que permiten generar automáticamente métodos y constructores, entre otros elementos, en tiempo de compilación.
Las ventajas de usar Project Lombok incluyen:

- **Reducción de código boilerplate:** Lombok ayuda a reducir la cantidad de código repetitivo que normalmente se encuentra en las clases Java, lo que puede hacer que el código sea más limpio y fácil de mantener.
- **Mejora la legibilidad del código:** Al eliminar código redundante, el código resultante es más conciso y legible, lo que facilita su comprensión.
- **Facilita el desarrollo rápido:** La generación automática de métodos y constructores puede acelerar el desarrollo al eliminar la necesidad de escribir manualmente estos fragmentos de código.
- **Integración sencilla:** Lombok se integra fácilmente con entornos de desarrollo y herramientas de compilación sin requerir configuraciones adicionales complejas.

A continuación mostramos algunas de las anotaciones más importantes de Lombok:

Anotación	Función
`@Getter/@Setter`	Simula los métodos GET y SET de todos los atributos de la clase.
`@ToString`	Simula el método TOSTRING tomando todos los atributos de la clase.
`@NoArgsConstructor` `@RequiredArgsConstructor` `@AllArgsConstructor`	Simula los constructores de la clase, en función a la anotación que se elija toma unos u otros parámetros.
`@EqualsAndHashCode`	Simula los métodos Equals y HashCode tomando todos los atributos de la clase (incluidos atributos de relaciones)
`@Data`	Simula todas las etiquetas anteriores a la vez. Tomando el constructor @RequiredArgsConstructor.

Lombok cuenta con más anotaciones que se pueden consultar en la web oficial de Lombok es su apartado de “features”:

https://projectlombok.org/features/

Para usar Lombok, basta con incluir sobre las cabeceras de nuestras clases las anotaciones que necesitemos , así como sobre las cabeceras de métodos , variables , etc. Por ejemplo podemos añadir las anotaciones @Getter y @Setter sobre la cabecera de una clase para que implemente dichos métodos para todos sus atributos, o bien ubicar dichas anotaciones sobre los atributos que deseemos que se implementen.

Para utilizar Lombok lo primero que deberás hacer es agregar su dependencia en tu proyecto, aquí mostramos un ejemplo de sería usando Maven:

```
<dependencies>
	<dependency>
		<groupId>org.projectlombok</groupId>
		<artifactId>lombok</artifactId>
		<version>1.18.30</version>
		<scope>provided</scope>
	</dependency>
</dependencies>
```

A continuación mostramos ejemplos de cómo incluiríamos Lombok en nuestro proyecto:

```
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class Persona {
    private String nombre;
    private int edad;
}
// Uso
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.setNombre("Juan");
        persona.setEdad(25);

        System.out.println("Nombre: " + persona.getNombre());
        System.out.println("Edad: " + persona.getEdad());
    }
}
```

```java
import lombok.ToString;

@ToString
public class Persona {
    private String nombre;
    private int edad;
}

// Uso
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.setNombre("Ana");
        persona.setEdad(30);

        System.out.println(persona); // Imprimirá automáticamente el
resultado de toString()
    }
}
```

```java
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;

@AllArgsConstructor
@NoArgsConstructor
public class Persona {
    private String nombre;
    private int edad;
}

// Uso
public class Main {
    public static void main(String[] args) {
        Persona persona1 = new Persona("Carlos", 28);
        Persona persona2 = new Persona(); // Se genera automáticamente
un constructor sin argumentos

        // ...
    }
}
```

```java
import lombok.Builder;

@Builder
public class Persona {
    private String nombre;
    private int edad;
}

// Uso
public class Main {
    public static void main(String[] args) {
        Persona persona = Persona.builder()
                                 .nombre("Luisa")
                                 .edad(22)
                                 .build();

        // ...
    }
}
```