



# **Desarrollo de Aplicaciones Web I**

---

<b>Curso</b>	Desarrollo de Aplicaciones Web I (0265/4694)
<b>Formato</b>	Manual de curso
<b>Autor Institucional</b>	Cibertec
<b>Páginas</b>	102 p.
<b>Elaborador</b>	Cenas Vasquez, Dany
<b>Revisor de Contenidos</b>	Morales Flores, Gustavo

# Índice

Presentación	5
Red de contenidos	6

## UNIDAD DE APRENDIZAJE 1: JAVA PERSISTENCE API

<b>1.1 Tema 1 : API de persistencia JPA</b>	<b>8</b>
1.1.1 : Gestión y monitoreo de conexiones	8
1.1.2 : Tipos e identificación de mapeo	15
1.1.3 : Entidades	26
1.1.4 : Herencia	32
<b>1.2 Tema 2 : JPA e Hibernate avanzado</b>	<b>37</b>
1.2.1 : Uso de flushing, batching y fetching	37
1.2.2 : Caching	40
1.2.3 : Control de concurrencia	41

## UNIDAD DE APRENDIZAJE 2: SPRING FRAMEWORK

<b>2.1. Tema 3 : Fundamentos</b>	<b>45</b>
2.1.1 : Introducción	45
2.1.2 : Spring Initializr	47
2.1.3 : Spring Boot Starters	48
2.1.4 : Uso de DevTools	49
2.1.5 : Lombok	50
<b>2.2. Tema 4 : Spring Data JPA</b>	<b>54</b>
2.2.1 : Introducción	54
2.2.2 : Consultas JPQL	55
2.2.3 : Uso de parámetros	57
2.2.4 : Paginación y ordenamiento	59
<b>2.3. Tema 5 : Spring Web MVC</b>	<b>64</b>
2.3.1 : Arquitectura	64
2.3.2 : Ciclo de vida	64
2.3.3 : Configuración	66
2.3.4 : Uso de estereotipos Spring	70
<b>2.4 Tema 6 : Spring Security</b>	<b>73</b>
2.4.1 : Introducción	73
2.4.2 : Configuración	74
2.4.3 : Almacenamiento en memoria	74
2.4.4 : Personalización de la autenticación	75

<b>2.5 Tema 7 : Spring RESTful</b>	<b>80</b>
2.5.1 : Introducción	80
2.5.2 : Configuración	81

### **UNIDAD DE APRENDIZAJE 3: ANGULAR**

<b>3.1 Tema 8 : Angular</b>	<b>85</b>
3.1.1 : Introducción	85
3.1.2 : Aplicaciones SPA	85
3.1.3 : Características	86

<b>3.2 Tema 9 : TypeScript</b>	<b>88</b>
3.2.1 : Introducción	88
3.2.2 : Tipos de datos	88
3.2.3 : Clases e Interfaces	89
3.2.4 : Clases e Interfaces	91
3.2.5 : Estructuras de control	92
3.2.6 : Operadores	93
3.2.7 : Promesas	94

<b>3.3 Tema 10 : Componentes Angular</b>	<b>96</b>
3.3.1 : Introducción	96
3.3.2 : Configuración	96
3.3.3 : Ensamblaje	96

<b>3.4 Tema 11 : Servicios Angular</b>	<b>98</b>
3.4.1 : Introducción	98
3.4.2 : Configuración	98
3.4.3 : Uso de Http	99

<b>Bibliografía</b>	<b>102</b>
---------------------	------------

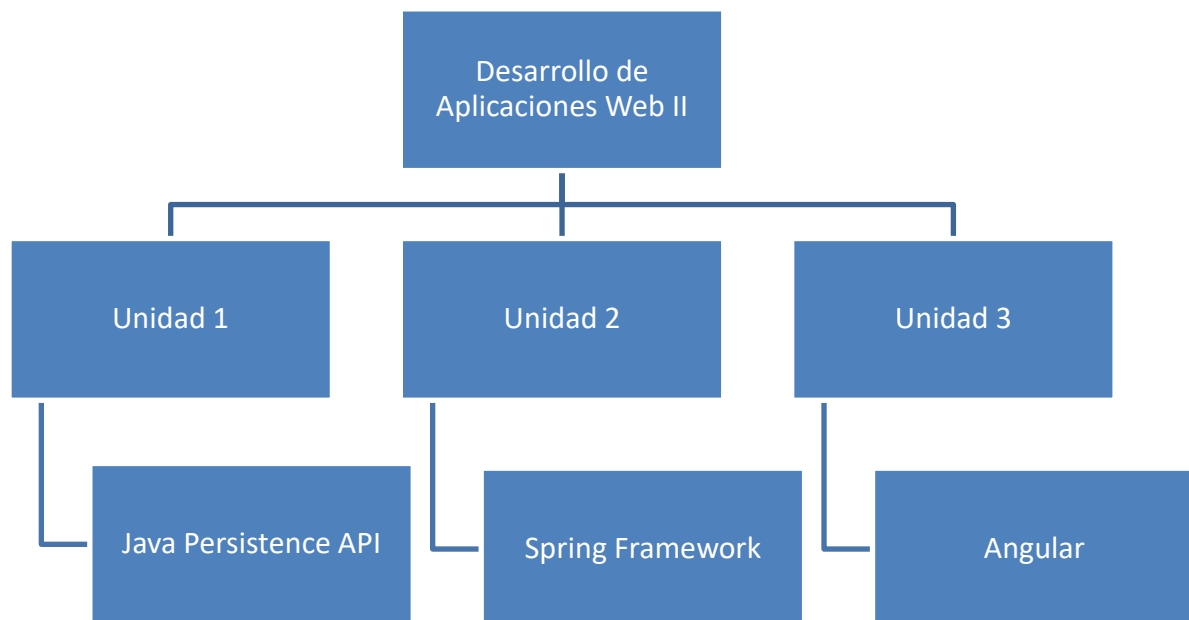
# Presentación

El curso de “Desarrollo de Aplicaciones Web II” pertenece a la línea de Programación dentro de la Carrera de Computación e Informática y brinda un conjunto de conocimientos y herramientas que permitirán a los alumnos poder desarrollar aplicaciones web de n-capas utilizando los frameworks Java: Java Persistence API (JPA ), Spring Framework y Angular.

El manual del curso ha sido diseñado bajo la modalidad de Unidades de Aprendizaje, las que desarrollan determinados temas a lo largo de las semanas establecidas para el dictado del curso. Cada unidad del manual indica los temas a ser tratados, los logros que se deben alcanzar y los contenidos que se deben desarrollar. Finalmente, se encontrará las actividades recomendadas que el alumno deberá desarrollar para reforzar lo trabajado y aprendido en la clase. Se incluye bibliografía y recursos de internet que puede colaborar en el logro de un autoaprendizaje efectivo.

El curso es eminentemente práctico, pero requiere horas adicionales de investigación y práctica por parte del alumno. Se inicia con un los conceptos de “OR-Mapping” con la especificación JPA (Java Persistence API) y su implementación en Hibernate: se abordan las anotaciones, mapeo y relaciones entre entidades así como los fundamentos básicos de JPQL para la construcción de consultas. Luego, la segunda unidad del manual aborda Spring Framework tratando de abarcar gran parte de la funcionalidad que proporciona. Finalmente, se aborda en la tercera unidad el framework de Angular, para la creación de Web Components.

# Red de contenidos



## UNIDAD

## 1

# JAVA PERSISTENCE API

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno puede realizar transacciones y consultas usando el API JPA a nivel empresarial.

**TEMARIO**

- 1.1 Tema 1 : API de persistencia JPA**
  - 1.1.1 : Gestión y monitoreo de conexiones
  - 1.1.2 : Tipos e identificación de mapeo
  - 1.1.3 : Entidades
  - 1.1.4 : Herencia
- 1.2 Tema 2 : JPA e Hibernate avanzado**
  - 1.2.1 : Uso de flushing, batching y fetching
  - 1.2.2 : Caching
  - 1.2.3 : Control de concurrencia

**ACTIVIDADES PROPUESTAS**

- Actividad 1: Aplicación de Java Persistence API.
- Actividad 2: Uso de Flushing, Batching y Fetching.

## 1.1. API DE PERSISTENCIA JPA

### 1.1.1. Gestión y monitoreo de conexiones

En el mundo de Java, existen varios frameworks y especificaciones para lograr un mapeo relacional de objetos, como Hibernate, TopLink o Java Data Objects (JDO), pero Java Persistence API (JPA) es la tecnología preferida.

La API de persistencia de Java (JPA) es una especificación de Java que gestiona objetos almacenados en una base de datos relacional. JPA le da al desarrollador una vista orientada a objetos para utilizar entidades de forma transparente en lugar de tablas. También viene con un lenguaje de consulta (Java Persistence Query Language , o JPQL), lo que permite consultas complejas sobre objetos.

JPA es solo una especificación que forma parte de Java EE y se rige por el JCP (Java Community Process). Luego es implementado por frameworks como EclipseLink, Hibernate u OpenJPA.

### JPA 2.2

JPA 2.2 se describe en JSR 338 y se lanzó en 2017. Se envió en 2017 con Java EE 8 pero era solo una versión de mantenimiento, lo que significa que usa el mismo JSR que JPA 2.1 (aún JSR 338) . Una versión de mantenimiento significa que una especificación no evoluciona mucho y, por lo tanto, no necesita un nuevo JSR. Los cambios realizados en JPA 2.2 son:

- Agrega compatibilidad con Java SE 8 Lambdas y Streams.
- Agrega @Repetible meta- anotación a las anotaciones JPA.
- Soporte para inyección de CDI en clases AttributeConverter.
- Soporte para el mapeo de los nuevos tipos java.time (por ejemplo, LocalDate, LocalTime, etc.).
- Agrega el método Stream getResultStream() predeterminado a las interfaces Query y TypedQuery.

Por razones de rendimiento, es mejor reutilizar las conexiones de base de datos. Debido a que los proveedores de JPA generan declaraciones SQL en nombre de los usuarios, es muy importante monitorear este proceso y reconocer su resultado. Esta unidad explica el mecanismo del proveedor de conexión de Hibernate y las formas de monitorear la ejecución de declaraciones.

### Gestión de conexiones JPA

Como toda la suite Java EE, la especificación JPA 1.0 estaba muy ligada a los servidores de aplicaciones empresariales. En un contenedor Java EE, todas las conexiones de la base de datos son administradas por el servidor de aplicaciones, que proporciona funciones de agrupación de conexiones, supervisión y JTA. Una vez configurado, el origen de datos Java EE se puede ubicar a través de JNDI.

En el archivo de configuración persistence.xml, el desarrollador de la aplicación debe proporcionar el nombre JNDI del origen de datos JTA o RESOURCE\_LOCAL asociado. El atributo de tipo de transacción también debe coincidir con las capacidades de transacción de la fuente de datos.

Una transacción RESOURCE\_LOCAL debe utilizar una non-jta-data-source DataSource.



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence-unit name="persistenceUnit" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <non-jta-data-source>java:/comp/env/jdbc/hsqldb</non-jta-data-source>
</persistence-unit>
```

Mientras que para una aplicación Java EE, está perfectamente bien confiar en el servidor de aplicaciones para proporcionar una referencia de DataSource con todas las funciones, las aplicaciones independientes generalmente se configuran usando inyección de dependencia en lugar de JNDI.

Desde la perspectiva de implementación de la APP, el origen de datos se puede configurar de forma externa o por el propio proveedor de JPA. La mayoría de las veces, la configuración de una fuente de datos externa sigue siendo la alternativa, ya que da más flexibilidad en la decoración del mecanismo de suministro de conexión (p. ej. registro, seguimiento).

Los proveedores de JPA pueden obtener conexiones a través del controlador JDBC ya que JPA 2.0 ha estandarizado las propiedades de configuración de la conexión de la base de datos:

Property	Description
javax.persistence.jdbc.driver	Driver full class name (e.g. org.hsqldb.jdbc.JDBCDriver )
javax.persistence.jdbc.url	Driver Url (e.g. jdbc:hsqldb:mem:test )
javax.persistence.jdbc.user	Database user's name
javax.persistence.jdbc.password	Database user's password

Figura 1: Propiedades de conexión JPA  
Fuente.- Tomado de Mihalcea, V. (2020)

Desafortunadamente, estas propiedades por sí solas no son suficientes porque la mayoría de las aplicaciones empresariales necesitan capacidades de monitoreo y agrupación de conexiones de todos modos. Por esta razón, la administración de conexiones JPA sigue siendo un tema específico de implementación, y las próximas secciones se sumergen en el mecanismo del proveedor de conexiones empleado por Hibernate.

### Proveedores de conexión Hibernate

Hibernate necesita funcionar tanto en Java EE como en entornos independientes, y la configuración de la conectividad de la base de datos se puede realizar de forma declarativa o programática. Para acomodar las conexiones del Driver JDBC, así como las configuraciones RESOURCE\_LOCAL y JTA DataSource, Hibernate define su propia abstracción de fábrica de conexiones, representada por la interfaz org.hibernate.engine.jdbc.connections.spi.ConnectionProvider:

```
public interface ConnectionProvider extends Service, Wrapped {
  public Connection getConnection() throws SQLException;
  public void closeConnection(Connection connection) throws SQLException;
```

```
public boolean supportsAggressiveRelease();  
}
```

Dado que el proveedor de conexión puede influir en el tiempo de respuesta de la transacción, cada proveedor se analiza desde la perspectiva del sistema OLTP de alto rendimiento.

### DriverManagerConnectionProvider

Hibernate elige este proveedor cuando se le dan las propiedades de conexión JPA 2.0 antes mencionadas o la contraparte de configuración específica de Hibernate:

- hibernate.connection.driver\_class
- hibernate.connection.url
- hibernate.connection.username
- hibernate.connection.password

### C3P0ConnectionProvider

c3p0 es una solución de agrupación de conexiones madura que ha demostrado su eficacia en muchos entornos de producción y, utilizando las propiedades de conexión JDBC subyacentes, Hibernate puede reemplazar la agrupación de conexiones incorporada con un c3p0DataSource. Para activar este proveedor, el desarrollador de la aplicación debe proporcionar al menos una propiedad de configuración que comience con el prefijo hibernate.c3p0:

```
<propertyname="hibernate.c3p0.max_size"value="5"/>
```

C3p0 (lanzado en 2001) y Apache DBCP2 (lanzado en 2002) son las soluciones de agrupación de conexiones Java independientes más antiguas y más implementadas. Más tarde, en 2010, BoneCP3 surgió como una alternativa de alto rendimiento para c3p0 y Apache DBCP. Hoy en día, la página de BoneCP GitHub dice que ha quedado obsoleta en favor de HikariCP4.

### HikariCPConnectionProvider

Hibernate 5 admite HikariCP (uno de los grupos de conexiones más rápidos) a través de la siguiente dependencia:

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-hikaricp</artifactId>  
  <version>${hibernate.version}</version>  
</dependency>
```

Al especificar la propiedad hibernate.connection.provider\_class, el desarrollador de la aplicación puede anular el mecanismo del proveedor de conexión predeterminado:

```
<propertyname="hibernate.connection.provider_class"
value="org.hibernate.hikaricp.internal.HikariCPConnectionProvider"/>
```

A diferencia de `DriverManagerConnectionProvider` o `C3P0ConnectionProvider`, `HikariCP` requiere propiedades de configuración específicas:

Property	Description
<code>hibernate.hikari.dataSourceClassName</code>	Driver full class name
<code>hibernate.hikari.dataSource.url</code>	Driver Url
<code>hibernate.hikari.dataSource.user</code>	Database user's name
<code>hibernate.hikari.dataSource.password</code>	Database user's password
<code>hibernate.hikari.maximumPoolSize</code>	Maximum pool size

Figura 2: Propiedades de la conexión HikariCP  
Fuente.- Tomado de Mihalcea, V. (2020)

### DatasourceConnectionProvider

Este proveedor se elige cuando el archivo de configuración JPA define un elemento `non-jta-data-source` o `jta-data-source`, o cuando se suministra una propiedad de configuración `hibernate.connection.datasource`.

### Modos de liberación de conexión

Hibernate pospone la adquisición de la conexión de la base de datos hasta que la transacción actual tiene que ejecutar su primera instrucción SQL (ya sea activada por una operación de lectura o escritura). Esta optimización permite a Hibernate reducir el intervalo de transacciones físicas, aumentando así la posibilidad de obtener una conexión del grupo. La estrategia de liberación de la conexión se controla a través de la propiedad `hibernate.connection.release_mode` que puede tomar los siguientes valores:

Value	Description
<code>after_transaction</code>	Once acquired, the database connection is released only after the current transaction either commits or rolls back.
<code>after_statement</code>	The connection is released after each statement execution and reacquired prior to running the next statement. Although not required by either JDBC or JTA specifications, this strategy is meant to prevent application servers from <a href="#">mistakenly detecting</a> <sup>7</sup> a connection leak between successive EJB (Enterprise Java Beans) calls
<code>auto</code>	This is the default value, and for <code>RESOURCE_LOCAL</code> transactions, it uses the <code>after_transaction</code> mode, while for JTA transactions it falls back to <code>after_statement</code> .

Figura 3: Modos de liberación de conexión  
Fuente.- Tomado de Mihalcea, V. (2020)

Para las transacciones JTA, el modo predeterminado puede ser demasiado estricto, ya que no todos los servidores de aplicaciones Java EE exhiben el mismo comportamiento para administrar los recursos transaccionales. De esta manera, es importante verificar si las conexiones de la base de datos se pueden cerrar fuera del componente EJB que desencadenó el evento de adquisición de la conexión. Los sistemas empresariales basados en Spring no usan Enterprise Java Beans e, incluso cuando se usa un administrador de transacciones JTA independiente, el modo de liberación de conexión `after_transaction` puede estar bien.

Es de alguna manera intuitivo que el modo `after_statement` incurre en alguna penalización de rendimiento asociada con los frecuentes ciclos de conexión de adquisición / liberación. Por esta razón, la siguiente prueba mide la sobrecarga de adquisición de conexión cuando se usa Bitronix en un contexto de aplicación Spring. Cada transacción ejecuta la misma declaración (obteniendo la marca de tiempo actual) durante un número determinado de veces (representado en el eje x). El eje y captura los tiempos de respuesta de transacción registrados para los modos de liberación de conexión `after_statement` y `after_transaction`.

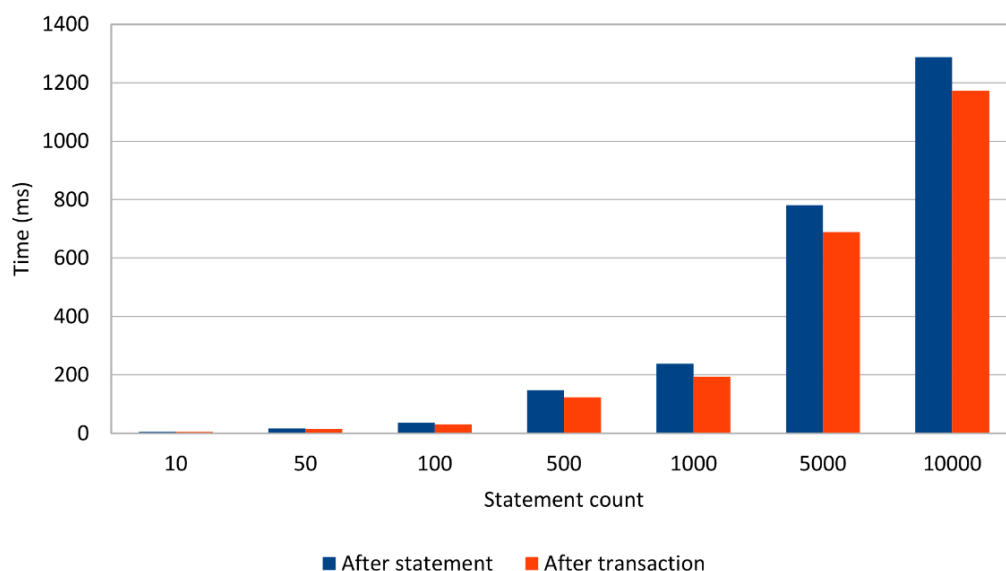


Figura 4: Modos de liberación de conexión

Fuente.- Tomado de Mihalcea, V. (2020)

Cuantas más declaraciones ejecute una transacción, mayor será la penalización de volver a adquirir la conexión de base de datos asociada del grupo de conexiones subyacente. Para visualizar mejor la sobrecarga de adquisición de la conexión, la prueba ejecuta hasta 10000 declaraciones, incluso si este número es probablemente demasiado alto para la transacción OLTP típica.

Idealmente, las transacciones de la base de datos deben ser lo más cortas posible y la cantidad de declaraciones tampoco debe ser demasiado alta. Este requisito se debe al hecho de que el número de conexiones agrupadas es limitado y es mejor liberar los bloqueos cuanto antes.

### **Monitoreo de conexiones**

Como se concluyó anteriormente, se prefiere el uso de una fuente de datos configurada externamente porque la fuente de datos real puede decorarse con capacidades de agrupación de conexiones, monitoreo y registro. Debido a que así es exactamente cómo funciona FlexyPool, el siguiente diagrama captura el mecanismo de exposición del origen de datos.

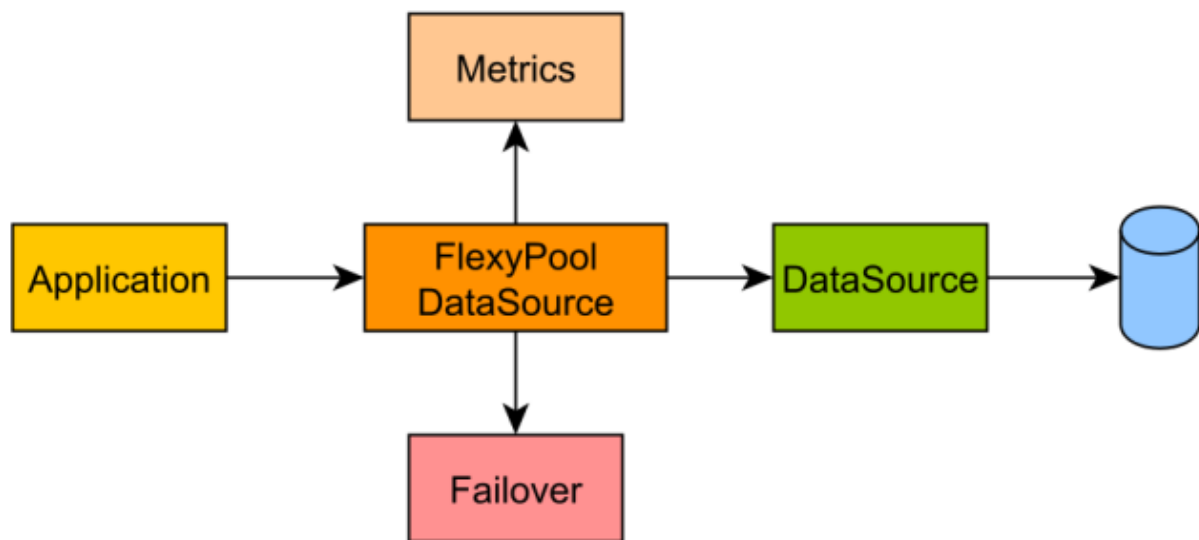


Figura 5: Arquitectura FlexyPool  
Fuente.-Tomado de Mihalcea, V. (2020)

En lugar de obtener la instancia actual de DataSource, la capa de acceso a datos obtiene una referencia de proxy. El proxy intercepta las solicitudes de adquisición y liberación de conexión y, de esta manera, puede monitorear su uso.

Cuando se usa Spring, configurar FlexyPool es bastante fácil porque la aplicación tiene control total.

En Java EE, las conexiones de la base de datos siempre deben obtenerse de un DataSource administrado, y una forma simple de integrar FlexyPool es extender el predeterminado DataSourceConnectionProviderImpl y sustituir el original DataSource con el FlexyPoolDataSource.

### **Estadísticas de Hibernate**

Hibernate tiene un recopilador de estadísticas incorporado que recopila notificaciones relacionadas con conexiones de bases de datos, transacciones de sesión e incluso el uso de almacenamiento en caché de segundo nivel.

La interfaz StatisticsImplementor define el contrato para interceptar varios eventos internos de Hibernate.

StatisticsImplementor	
openSession()	void
closeSession()	void
flush()	void
connect()	void
prepareStatement()	void
closeStatement()	void
endTransaction(boolean)	void
loadEntity(String)	void
fetchEntity(String)	void
updateEntity(String)	void
insertEntity(String)	void
deleteEntity(String)	void
optimisticFailure(String)	void
loadCollection(String)	void
fetchCollection(String)	void
updateCollection(String)	void
recreateCollection(String)	void
removeCollection(String)	void
secondLevelCachePut(String)	void
secondLevelCacheHit(String)	void
secondLevelCacheMiss(String)	void
naturalIdCachePut(String)	void
naturalIdCacheHit(String)	void
naturalIdCacheMiss(String)	void
naturalIdQueryExecuted(String, long)	void
queryCachePut(String, String)	void
queryCacheHit(String, String)	void
queryCacheMiss(String, String)	void
queryExecuted(String, int, long)	void
updateTimestampsCacheHit()	void
updateTimestampsCacheMiss()	void
updateTimestampsCachePut()	void

Figura 6: Interface Hibernate StatisticsImplementor

Fuente.-Tomado de Mihalcea, V. (2020)

Existe una gran variedad de métricas que Hibernate puede recopilar en nombre del usuario, pero, por razones de rendimiento, el mecanismo de estadísticas está deshabilitado de forma predeterminada.

Para habilitar el mecanismo de recopilación de estadísticas, primero se debe configurar la siguiente propiedad:

```
<propertyname="hibernate.generate_statistics"value="true"/>
```

Una vez recopiladas las estadísticas, para imprimirlas en el registro de la aplicación actual, la siguiente configuración del registrador debe configurarse:

```
<loggername="org.hibernate.engine.internal.StatisticalLoggingSessionEventListener"level="info"/>
```

Con estas dos configuraciones, cada vez que finaliza una sesión Hibernate(Persistence Context), el siguiente informe se muestra en el log que se está ejecutando actualmente.

```
37125102 nanoseconds spent acquiring 10000 JDBC connections;  
25521714 nanoseconds spent releasing 10000 JDBC connections;  
95242323 nanoseconds spent preparing 10000 JDBC statements;  
923615040 nanoseconds spent executing 10000 JDBC statements;
```

El recopilador de estadísticas predeterminado solo cuenta el número de veces que se llamó a un determinado método de devolución de llamada y, si eso no es satisfactorio, el desarrollador de la aplicación puede proporcionar su propia implementación de estadísticas personalizada.

### Statement logging

Una herramienta ORM puede generar automáticamente declaraciones DML, y es responsabilidad del desarrollador de la aplicación validar tanto su eficacia como su impacto general en el rendimiento. Aplazar la validación de la declaración SQL hasta que la capa de acceso a los datos comience a mostrar problemas de rendimiento es arriesgado e incluso puede afectar el costo de desarrollo. Por esta razón, el registro de declaraciones SQL se vuelve relevante desde las primeras etapas de la aplicación.

Aunque JPA 2.1 no cuenta con una propiedad de configuración estándar para registrar declaraciones SQL, la mayoría de las implementaciones de JPA admiten esta característica a través de configuraciones específicas del marco. Para ello, Hibernate define las siguientes propiedades de configuración:

Property	Description
<code>hibernate.show_sql</code>	Prints SQL statements to the console
<code>hibernate.format_sql</code>	Formats SQL statements before being logged or printed to the console
<code>hibernate.use_sql_comments</code>	Adds comments to the automatically generated SQL statement

Figura 7: Modos de liberación de conexión

Fuente.-Tomado de Mihalcea, V. (2020)

Usar la consola del sistema para el registro es una mala práctica, un marco de registro (por ejemplo, Logback o Log4j) es una mejor alternativa ya que admite appenders configurables y niveles de registro.

### 1.1.2. Tipos e identificadores de mapeo

JPA aborda la discrepancia de objeto/relación al asociar tipos de objetos de Java a estructuras de bases de datos. Suponiendo que hay una tabla de base de datos de tareas que tiene cuatro columnas (por ejemplo, id, created\_by, created\_on y status), el proveedor de JPA debe asignarlo al modelo de dominio que consta de dos clases (por ejemplo, Tarea y Cambio).

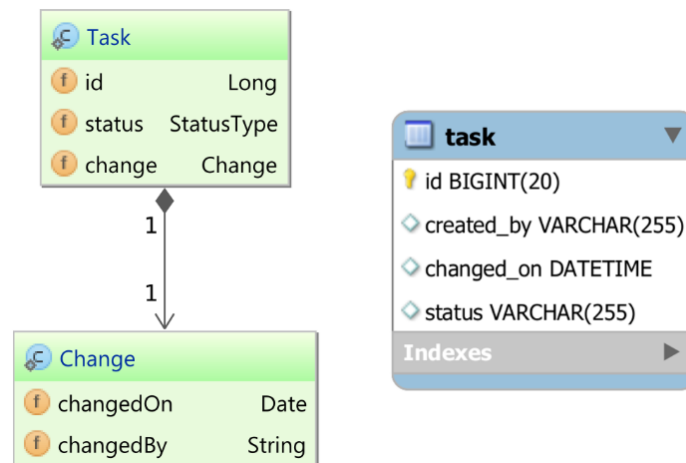


Figura 8: Tipos de Mapeo

Fuente.-Tomado de Mihalcea, V. (2020)

JPA utiliza tres elementos principales de mapeo relacional de objetos: tipos, incrustables y entidades. En el diagrama anterior, el objeto Task es una entidad, mientras que el objeto Change es un tipo incrustable.

Tanto la entidad como el grupo embeddable agrupan múltiples atributos del modelo de dominio confiando en HibernateType(s) para asociar los tipos de columnas de la base de datos con los objetos de valor de Java (por ejemplo, String, Integer, Date) . La principal diferencia entre una entidad y un incrustable es la presencia de un identificador, que se utiliza para asociar una clave única de la tabla de la base de datos (normalmente la clave primaria) con un atributo de objeto del modelo de dominio.

Los identificadores son obligatorios para los elementos de la entidad, y un tipo incrustable no puede tener una identidad propia. Al conocer la tabla de la base de datos y la columna que identifica de forma única una fila determinada, Hibernate puede correlacionar las filas de la base de datos con las entidades del modelo de dominio. Un tipo incrustable agrupa múltiples atributos en un solo componente reutilizable.

```
@Embeddable
public class Change {
    @Column(name = "changed_on")
    private Date changedOn;
    @Column(name = "created_by")
    private String changedBy;
}
```

La asociación de composición, definida por UML, es la analogía perfecta para la relación entre una entidad y un incrustable. Cuando una entidad incluye un tipo incrustable, todos sus atributos pasan a formar parte de la entidad propietaria.

El objeto incrustable no puede definir su propio identificador porque, de lo contrario, la entidad tiene más de una identidad. Al carecer de un identificador, el objeto incrustable no puede ser administrado por un contexto de persistencia y su estado está controlado por su entidad principal.



## Tipos

Para cada tipo de base de datos soportado, JDBC define un enumeration `java.sql.JDBCType`. Dado que se construye sobre JDBC, Hibernate realiza el mapeo entre los tipos JDBC y sus contrapartes Java asociadas (primitivas u objetos).

Hibernate type	JDBC type	Java type
<code>BooleanType</code>	BIT	<code>boolean</code> , <code>Boolean</code>
<code>NumericBooleanType</code>	INTEGER (e.g. 0, 1)	<code>boolean</code> , <code>Boolean</code>
<code>TrueFalseType</code>	CHAR (e.g. 'F', 'f', 'T', 't')	<code>boolean</code> , <code>Boolean</code>
<code>YesNoType</code>	CHAR (e.g. 'N', 'n', 'Y', 'y')	<code>boolean</code> , <code>Boolean</code>
<code>ByteType</code>	TINYINT	<code>byte</code> , <code>Byte</code>
<code>ShortType</code>	SMALLINT	<code>short</code> , <code>Short</code>
<code>CharacterType</code>	CHAR	<code>char</code> , <code>Character</code>
<code>CharacterNCharType</code>	NCHAR	<code>char</code> , <code>Character</code>
<code>IntegerType</code>	INTEGER	<code>int</code> , <code>Integer</code>
<code>LongType</code>	BIGINT	<code>long</code> , <code>Long</code>
<code>FloatType</code>	FLOAT	<code>float</code> , <code>Float</code>
<code>DoubleType</code>	DOUBLE	<code>double</code> , <code>Double</code>
<code>CharArrayType</code>	VARCHAR	<code>char[]</code> , <code>Character[]</code>

Figura 9: Tipos primitivos

Fuente.-Tomado de Mihalcea, V. (2020)

De un sistema de base de datos a otro, el tipo booleano se puede representar como un tipo de base de datos BIT, BYTE, BOOLEAN o CHAR, por lo que define cuatro tipos para resolver el tipo primitivo boolean.

## Tipos String

Una cadena de Java puede consumir tanta memoria como el montón de Java tenga disponible. Por otro lado, los sistemas de bases de datos definen tanto tipos de tamaño limitado (VARCHAR y NVARCHAR) como ilimitados (TEXT, NTEXT, BLOB y NCLOB).

Para adaptarse a esta discrepancia de mapeo, Hibernate define los siguientes tipos:

Hibernate type	JDBC type	Java type
<code>StringType</code>	VARCHAR	<code>String</code>
<code>StringNvarcharType</code>	NVARCHAR	<code>String</code>
<code>TextType</code>	LONGVARCHAR	<code>String</code>
<code>NTextType</code>	LONGNVARCHAR	<code>String</code>
<code>MaterializedClobType</code>	CLOB	<code>String</code>
<code>MaterializedNClobType</code>	NCLOB	<code>String</code>

Figura 10: Tipos String

Fuente.-Tomado de Mihalcea, V. (2020)

## Tipos de fecha y hora

Cuando se trata de tiempo, existen múltiples representaciones de bases de datos o Java, lo que explica la gran cantidad de tipos de Hibernate relacionados con el tiempo.

Hibernate type	JDBC type	Java type
DateType	DATE	Date
TimeType	TIME	Time
TimestampType	TIMESTAMP	Timestamp, Date
DbTimestampType	TIMESTAMP	Timestamp, Date
CalendarType	TIMESTAMP	Calendar, GregorianCalendar
CalendarDateType	DATE	Calendar, GregorianCalendar
CalendarTimeType	TIME	Calendar, GregorianCalendar
TimeZoneType	VARCHAR	TimeZone

Figura 11: Tipos fecha y hora

Fuente.-Tomado de Mihalcea, V. (2020)

## Tipos numéricos

Oracle puede representar números de hasta 38 dígitos, por lo tanto, solo cabe en un BigInteger o un BigDecimal (java.lang.Long y java.lang.Double sólo pueden almacenar hasta 8 bytes).

Hibernate type	JDBC type	Java type
BigIntegerType	NUMERIC	BigInteger
BigDecimalType	NUMERIC	BigDecimal

Figura 12: Tipos numéricos

Fuente.-Tomado de Mihalcea, V. (2020)

## Tipos binarios

Para los tipos binarios, la mayoría de los sistemas de bases de datos ofrecen múltiples opciones de almacenamiento (por ejemplo, RAW, VARBINARY, BYTEA, BLOB, CLOB). En Java, la capa de acceso a datos puede usar una matriz de bytes, un JDBC Blob o Clob, o incluso un tipo serializable, si el objeto Java fue calculado antes de ser guardado en la base de datos.

Hibernate type	JDBC type	Java type
BinaryType	VARBINARY	byte [], Byte []
BlobType	BLOB	Blob
ClobType	CLOB	Clob
NClobType	NCLOB	Clob
MaterializedBlobType	BLOB	byte [], Byte []
ImageType	LONGVARBINARY	byte [], Byte []
SerializableType	VARBINARY	Serializable
SerializableToBlobType	BLOB	Serializable

Figura 13: Tipos binarios

Fuente.-Tomado de Mihalcea, V. (2020)

## Tipos UUID

Hay varias formas de conservar un UUID (Universally Unique Identifier) de Java y, según la huella de memoria, los tipos de almacenamiento más eficientes son los tipos de columna UUID específicos de la base de datos.

Hibernate type	JDBC type	Java type
UUIDBinaryType	BINARY	UUID
UUIDCharType	VARCHAR	UUID
PostgresUUIDType	OTHER	UUID

Figura 14: Tipos UUID

Fuente.-Tomado de Mihalcea, V. (2020)

## Otros tipos

Hibernate también puede mapear Java Enum(s), Class, URL, Locale y Currency también.

Hibernate type	JDBC type	Java type
EnumType	CHAR, LONGVARCHAR, VARCHAR	Enum
	INTEGER, NUMERIC, SMALL INT, TINYINT, BIGINT, DECIMAL, DOUBLE, FLOAT	
ClassType	VARCHAR	Class
CurrencyType	VARCHAR	Currency
LocaleType	VARCHAR	Locale
UrlType	VARCHAR	URL

Figura 15: Otros tipos  
Fuente.-Tomado de Mihalcea, V. (2020)

## Tipos personalizados

No solo tiene un conjunto muy rico de tipos de datos, sino que PostgreSQL también permite agregar tipos personalizados (usando la declaración CREATE DOMAINDDL). La elección del tipo de base de datos adecuado para cada campo del modelo de dominio puede marcar la diferencia en términos de rendimiento de acceso a los datos. Aunque existe una gran variedad de tipos integrados, el desarrollador de aplicaciones no se limita a los disponibles en el mercado, y se pueden agregar nuevos tipos sin demasiado esfuerzo.

En el siguiente ejemplo, la lógica empresarial requiere el acceso de supervisión a una aplicación empresarial. Para este propósito, la capa de acceso a datos almacena las direcciones IP (Protocolo de Internet) de cada usuario registrado.

Suponiendo que esta aplicación interna utiliza solo el protocolo IPv4, las direcciones IP se almacenan en el formato de enrutamiento entre dominios sin clases (por ejemplo, 192.168.123.231 / 24). PostgreSQL puede almacenar direcciones IPv4 en un tipo cidr o inet, o puede usar un tipo de columna VARCHAR (18).

La columna VARCHAR (18) requiere 18 caracteres y, asumiendo una codificación UTF-8, cada dirección IPv4 necesita como máximo 18 bytes. La dirección de tamaño más pequeño (por ejemplo, 0.0.0.0 / 0) que toma 9 caracteres, el enfoque VARCHAR (18) requiere entre 9 y 18 caracteres para cada dirección IPv4.

El tipo inet está especialmente diseñado para direcciones de red IPv4 e IPv6, y también admite varios operadores específicos de direcciones de red (por ejemplo, <, >, &&), así como otras funciones de transformación de direcciones (por ejemplo, host (inet), netmask (inet)). A diferencia del enfoque VARCHAR (18), el tipo inet requiere solo 7 bytes para cada dirección IPv4.

Dado que tiene un tamaño más compacto (el índice puede caber mejor en la memoria) y admite muchos operadores específicos, el tipo inet es una opción mucho más atractiva. Aunque, de forma predeterminada, Hibernate no admite tipos inet, agregar un tipo de Hibernate personalizado es una tarea sencilla. La dirección IPv4 está encapsulada en su propio contenedor, que también puede definir varias funciones de manipulación de direcciones.

```
public class IPv4 implements Serializable {

    private final String address;

    public IPv4(String address) {
```

```

        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        return Objects.equals(address, IPv4.class.cast(o).address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(address);
    }

    public InetAddress toInetAddress() throws UnknownHostException {
        return InetAddress.getByAddress(address);
    }
}

```

Cuando una entidad quiere cambiar un campo IPv4, debe proporcionar una nueva instancia de objeto. Un tipo inmutable es mucho más fácil de manejar ya que su estado interno no cambia en el contexto de persistencia que se está ejecutando actualmente.

Todos los tipos personalizados deben implementar la interfaz UserType y, dado que ImmutableType se encarga de la mayoría de los detalles de implementación de UserType, IPv4Type puede centrarse en la lógica de conversación específica del tipo.

```

public class IPv4Type extends ImmutableType<IPv4> {

    public IPv4Type() {
        super(IPv4.class);
    }

    @Override
    public int[] sqlTypes() {
        return new int[]{Types.OTHER};
    }

    @Override
    public IPv4 get(ResultSet rs, String[] names,
        SharedSessionContractImplementor session, Object owner) throws SQLException {
        String ip = rs.getString(names[0]);
        return (ip != null) ? new IPv4(ip) : null;
    }

    @Override
    public void set(PreparedStatement st, IPv4 value, int index,
        SharedSessionContractImplementor session) throws SQLException {
        if (value == null) {

```

```

        st.setNull(index, Types.OTHER);
    } else {
        PGobject holder = new PGobject();
        holder.setType("inet");
        holder.setValue(value.getAddress());
        st.setObject(index, holder);
    }
}
}
}

```

El método `get()` se usa para mapear el campo `inet` a una instancia de objeto `IPv4`, mientras que `set()` se usa para transformar el objeto `IPv4` en el equivalente del controlador JDBC de PostgreSQL.

```

public abstract class ImmutableType<T> implements UserType {

    private final Class<T> clazz;

    protected ImmutableType(Class<T> clazz) {
        this.clazz = clazz;
    }

    @Override
    public Object nullSafeGet(ResultSet rs, String[] names,
        SharedSessionContractImplementor session, Object owner) throws SQLException {
        return get(rs, names, session, owner);
    }

    @Override
    public void nullSafeSet(PreparedStatement st, Object value, int index,
        SharedSessionContractImplementor session) throws SQLException {
        set(st, clazz.cast(value), index, session);
    }

    protected abstract T get(ResultSet rs, String[] names,
        SharedSessionContractImplementor session, Object owner) throws SQLException;

    protected abstract void set(PreparedStatement st, T value, int index,
        SharedSessionContractImplementor session) throws SQLException;

    @Override
    public Class<T> returnedClass() {
        return clazz;
    }

    @Override
    public boolean equals(Object x, Object y) {
        return Objects.equals(x, y);
    }

    @Override
    public int hashCode(Object x) {
        return x.hashCode();
    }

    @Override
    public Object deepCopy(Object value) {

```

```

        return value;
    }

    @Override
    public boolean isMutable() {
        return false;
    }

    @Override
    public Serializable disassemble(Object o) {
        return (Serializable) o;
    }

    @Override
    public Object assemble(Serializable cached, Object owner) {
        return cached;
    }

    @Override
    public Object replace(Object o, Object target, Object owner) {
        return o;
    }
}

```

La anotación `@Type` indica a Hibernate que use el tipo de `IPv4` para mapear el campo `IPv4`.

```

@Entity
public class Event {

    @Id
    @GeneratedValue
    private Long id;

    @Type(type = "com.vladmihalcea.book.hppj.hibernate.type.IPv4Type")
    @Column(name = "ip", columnDefinition = "inet")
    private IPv4 ip;

    public Long getId() {
        return id;
    }

    public IPv4 getIp() {
        return ip;
    }

    public void setIp(String address) {
        this.ip = new IPv4(address);
    }
}

```

## Identificadores

Todas las tablas de la base de datos deben tener una columna de clave principal, por lo que cada fila se puede identificar de forma única (la clave principal debe ser ÚNICA y NO NULA).

La clave primaria puede tener un significado en el mundo real, en cuyo caso es una clave natural, o puede generarse sintéticamente, en cuyo caso se denomina un identificador sustituto.

Para las claves naturales, la unicidad se aplica mediante un generador de secuencia único del mundo real (p. Ej. Números de identificación nacional, Números de seguridad social, Números de identificación de vehículos). En realidad, los números únicos naturales pueden plantear problemas cuando las restricciones de unicidad ya no son válidas. Por ejemplo, un número de identificación nacional puede generar números únicos, pero si el sistema empresarial debe acomodar a usuarios que vienen de varios países, es posible que dos países diferentes asignen el mismo identificador.

La clave natural puede estar compuesta por una o varias columnas. Las claves naturales compuestas pueden suponer una penalización de rendimiento adicional porque las uniones de varias columnas son más lentas que las de una sola columna, y los índices de varias columnas también tienen una mayor huella de memoria.

Las claves naturales deben ser lo suficientemente largas para acomodar tantos identificadores como el sistema necesite durante su ciclo de vida. . Debido a que las claves primarias a menudo se indexan, cuanto más larga es la clave, más memoria requiere una entrada de índice. Cada tabla unida incluye una clave externa que refleja la clave principal, y las claves externas también se indexan con frecuencia.

### **Identificadores UUID**

Sin embargo, algunos sistemas empresariales utilizan claves primarias UUID, por lo que vale la pena saber qué tipos de Hibernate funcionan mejor para esta tarea. La clave UUID puede ser generada por la aplicación usando la clase `java.util.UUID` o puede ser asignada por el sistema de base de datos.

### **El generador asignado**

Simplemente omitiendo la anotación `@GeneratedValue`, Hibernate recurre al identificador asignado, que permite que la capa de acceso a los datos controle el proceso de generación del identificador. El siguiente ejemplo asigna un identificador `java.util.UUID` a un tipo de columna `BINARY(16)`:

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    @Column(columnDefinition = "BINARY(16)")
    private UUID id;

    public Post() {
    }

    public Post(UUID id) {
        this.id = id;
    }
}
```

## El generador de UUID heredado

El generador hexadecimal de UUID se registra con el nombre `uuid` y genera representaciones de cadenas de UUID hexadecimales. Con un diseño de dígitos hexadecimales de 8-8-4-8-4, el generador hexadecimal de UUID no cumple con el estándar RFC 4122, que utiliza un formato de dígitos hexadecimales 8-4-4-4-12. El siguiente fragmento de código muestra el mapeo `UUIDHexGenerator` y la declaración de inserción asociada.

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    @Column(columnDefinition = "CHAR(32)")
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid")
    private String id;
}
```

## El generador de UUID más nuevo

El generador de UUID más reciente es compatible con RFC 4122 (variante 2) y está registrado bajo el nombre `uuid2` (trabajando con los tipos de objetos `java.lang.UUID`, `byte []` y `StringDomain Model`). En comparación con el caso de uso anterior, el mapeo y el caso de prueba tienen el siguiente aspecto:

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    @Column(columnDefinition = "BINARY(16)")
    @GeneratedValue(generator = "uuid2")
    @GenericGenerator(name = "uuid2", strategy = "uuid2")
    private UUID id;
}
```

## Identificadores numéricos

Como se explicó anteriormente, generalmente se prefiere una clave sustituta numérica ya que ocupa menos espacio y los índices funcionan mejor con identificadores secuenciales. Para generar identificadores numéricos, la mayoría de los sistemas de bases de datos ofrecen columnas de identidad (o `auto_increment`) u objetos de secuencia. JPA define la enumeración `GenerationType` para todos los tipos de generadores de identificadores admitidos:

- `IDENTITY` es para mapear el identificador de la entidad a una columna de identidad de la base de datos.
- `SEQUENCE` asigna identificadores llamando a una secuencia de base de datos determinada.
- `TABLE` es para bases de datos relacionales que no admiten secuencias (por ejemplo, MySQL 5.7), el generador de tablas emula una secuencia de base de datos utilizando una tabla separada.



- AUTO decide la estrategia de generación de identificadores en función del dialecto de la base de datos actual.

### Generador de identidad

El tipo de columna de identidad (incluido en el estándar SQL: 2003) es compatible con Oracle 12c, SQL Server y MySQL (AUTO\_INCREMENT), y permite que una columna INTEGER o BIGINT se incremente automáticamente bajo demanda. El proceso de incremento es muy eficiente ya que utiliza un mecanismo de bloqueo liviano, a diferencia de los bloqueos transaccionales más pesados. El único inconveniente es que el valor recién asignado solo se puede conocer después de ejecutar la instrucción de inserción real.

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

### Generador de secuencia

Una secuencia es un objeto de base de datos que genera números consecutivos. Definidas por el estándar SQL:2003, las secuencias de bases de datos son compatibles con Oracle, SQL Server 2012 y PostgreSQL y, en comparación con las columnas de identidad, las secuencias ofrecen las siguientes ventajas:

- La misma secuencia se puede utilizar para completar varias columnas, incluso entre tablas.
- Los valores se pueden preasignar para mejorar el rendimiento.
- Al permitir pasos incrementales, las secuencias pueden beneficiarse de las técnicas de optimización a nivel de aplicación.
- Debido a que la llamada de secuencia se puede desacoplar de la declaración de inserción real, Hibernate no deshabilita las actualizaciones por lotes de JDBC.

Para demostrar la diferencia entre la identidad y los generadores de identificador de secuencia, el ejemplo anterior se cambia para usar una secuencia de base de datos esta vez.

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
}
```

### Generador de tablas

Debido a la falta de coincidencia entre el generador de identificadores y el caché transaccional de escritura diferida, JPA ofrece un generador alternativo similar a una secuencia que funciona incluso cuando las secuencias no son compatibles de forma nativa.

Se utiliza una tabla de base de datos para contener el último valor de secuencia, y se emplea el bloqueo a nivel de fila para evitar que dos conexiones simultáneas adquieran el mismo valor de identificador.

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLEs)
    private Long id;

}
```

### 1.1.3. Entidades

En una base de datos relacional, las asociaciones se forman correlacionando filas que pertenecen a diferentes tablas. Se establece una relación cuando una tabla secundaria define una clave externa que hace referencia a la clave principal de su tabla principal. Cada asociación de base de datos se construye sobre claves externas, lo que da como resultado tres tipos de relación de tabla:

- one-to-many es la relación más común y asocia una fila de una tabla principal a varias filas en una tabla secundaria.
- one-to-one requiere que la clave principal de la tabla secundaria se asocie mediante una clave externa con la columna clave principal de la tabla principal.
- many-to-many requiere una tabla de vínculos que contenga dos columnas de clave externa que hagan referencia a las dos tablas principales diferentes.

El siguiente diagrama muestra estas tres relaciones de tabla:

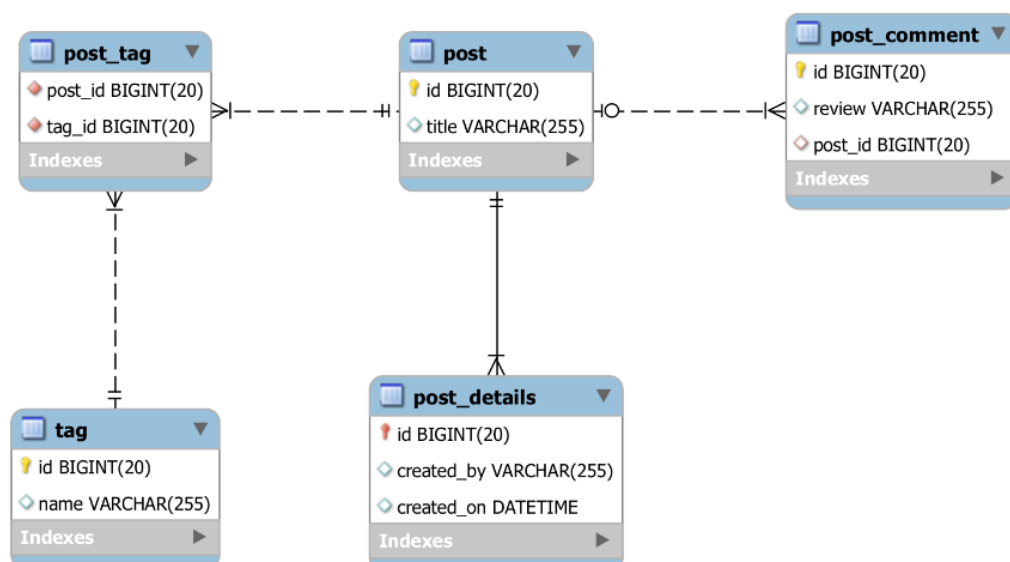


Figura 16: Relaciones de tabla  
Fuente.-Tomado de Mihalcea, V. (2020)

La tabla de post tiene una relación de uno a varios con la tabla post\_comment porque es posible que varios comentarios hagan referencia a una fila de publicaciones. La relación de uno a varios se establece a través de la columna post\_id que tiene una clave foránea que hace referencia a la clave primaria de la tabla de post. Debido a que un post\_comment no puede existir sin un post, el post es el lado principal mientras que el post\_comment es el lado secundario.

La tabla de post tiene una relación de uno a uno con post\_details. Al igual que la asociación uno a varios, la relación uno a uno implica dos tablas y una clave foránea. La clave foránea tiene una restricción de unicidad, por lo que solo una fila secundaria puede hacer referencia a un registro principal.

Post y tag son tablas independientes y ninguna es hija de la otra. Un post puede incluir varios tags, mientras que un tag también se puede asociar con varios posts. Ésta es una asociación típica de muchos a muchos y requiere una tabla de unión para resolver el lado secundario de estas dos entidades principales. La tabla de unión requiere dos claves externas que hagan referencia a las dos tablas principales.

En una base de datos relacional, la clave foránea está asociada solo con el lado secundario. Por esta razón, el lado principal no tiene conocimiento de ninguna relación secundaria asociada y, desde una perspectiva de mapeo, las relaciones de tabla son siempre unidireccionales (la clave externa foránea hace referencia a la clave primaria principal).

### Relación @ManyToOne

La relación @ManyToOne es la asociación JPA más común y se asigna exactamente a la relación de tabla de uno a muchos. Cuando se usa una asociación @ManyToOne, la clave externa subyacente está controlada por el lado secundario, sin importar que la asociación sea unidireccional o bidireccional. Esta sección se enfoca solo en las relaciones unidireccionales @ManyToOne, el caso bidireccional se discute más con la relación @OneToMany. En el siguiente ejemplo, la entidad post representa el lado principal, mientras que PostComment es el lado secundario. Como ya se mencionó, el diagrama de relación de entidades de JPA coincide exactamente con la relación de tabla uno a varios.

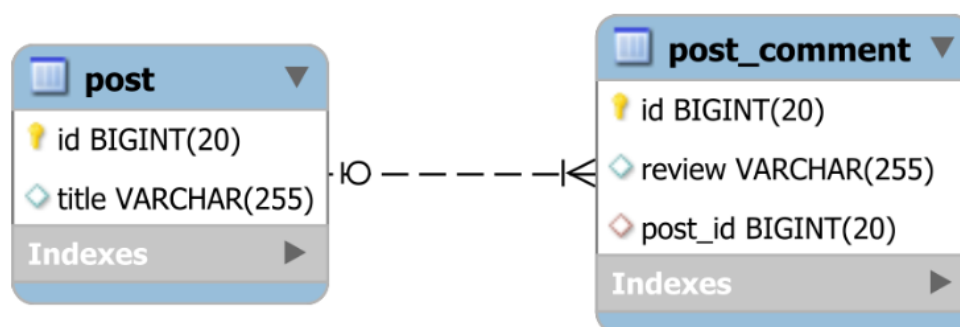


Figura 17: La relación de tablas de uno a muchos

Fuente.-Tomado de Mihalcea, V. (2020)

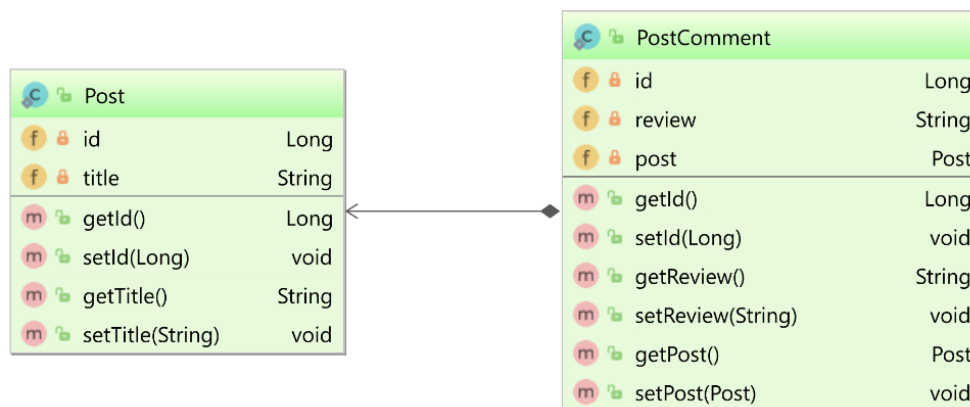


Figura 18: Relación @ManyToOne  
Fuente.-Tomado de Mihalcea, V. (2020)

### Relación @OneToMany

Si bien la asociación @ManyToOne es el mapeo más natural de la relación de tabla uno a muchos, la asociación @OneToMany también puede reflejar esta relación de base de datos, pero solo cuando se usa como mapeo bidireccional. Una asociación @OneToMany unidireccional usa una tabla de unión adicional, que ya no se ajusta a la semántica de relación de tabla uno a muchos.

### Bidireccional @OneToMany

La asociación bidireccional @OneToMany tiene un mapeo del lado secundario @ManyToOne coincidente que controla la relación de tabla subyacente de uno a muchos. El lado principal se asigna como una colección de entidades secundarias.

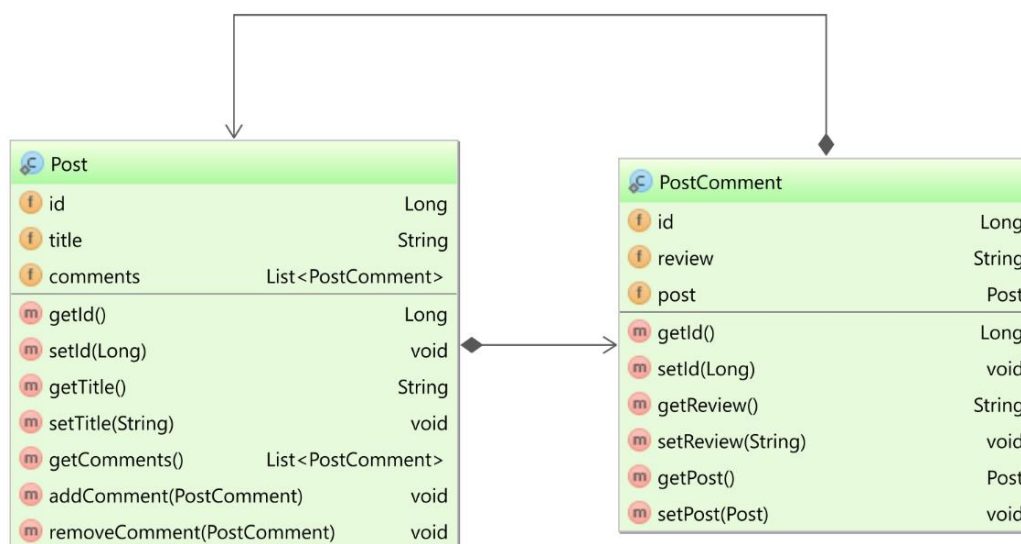


Figura 19: Relación bidireccional @OneToMany  
Fuente.-Tomado de Mihalcea, V. (2020)

### Unidireccional @OneToMany

La asociación unidireccional @OneToMany es muy tentadora porque el mapeo es más simple que su contraparte bidireccional. Debido a que solo hay un lado a tener en cuenta, no hay necesidad de métodos auxiliares y el mapeo tampoco presenta un atributo mappedBy.

@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)

```
private List<PostComment> comments = new ArrayList<>();
```

Desafortunadamente, a pesar de su simplicidad, la asociación `@OneToMany` unidireccional es menos eficiente que la asociación `@ManyToOne` unidireccional o la asociación `@OneToMany` bidireccional.

La asociación unidireccional `@OneToMany` no se asigna a una relación de tabla de uno a muchos. Debido a que no hay un lado `@ManyToOne` para controlar esta relación, Hibernate usa una tabla de unión separada para administrar la asociación entre una fila principal y sus registros secundarios.



Figura 20: La relación de la tabla `@OneToMany`

Fuente.-Tomado de Mihalcea, V. (2020)

## Relación `@OneToOne`

Desde la perspectiva de una base de datos, la asociación uno a uno se basa en una clave externa que está restringida a ser única. De esta forma, solo un registro secundario puede hacer referencia a una fila principal como máximo.

En JPA, la relación `@OneToOne` puede ser unidireccional o bidireccional.

### Unidireccional `@OneToOne`

En el siguiente ejemplo, la tabla `Post` representa el lado principal, mientras que `PostDetails` es el lado secundario de la asociación uno a uno. Como ya se mencionó, el diagrama de relación de entidades de JPA coincide exactamente con la relación de tabla uno a uno.

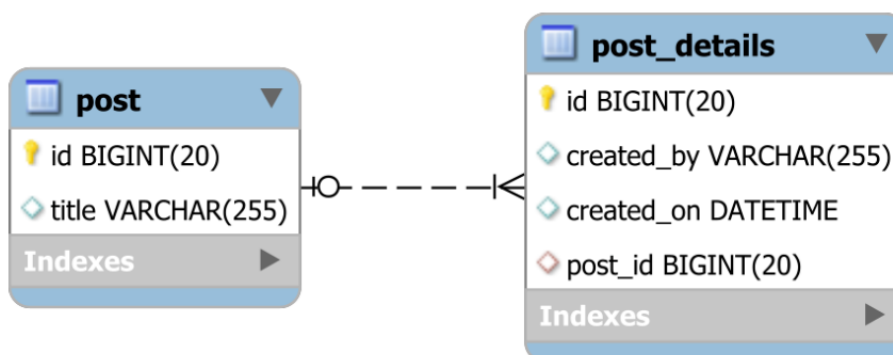


Figura 21: La relación de tabla uno a uno

Fuente.-Tomado de Mihalcea, V. (2020)

Incluso desde el lado del modelo de dominio, la relación unidireccional `@OneToOne` es sorprendentemente similar a la asociación unidireccional `@ManyToOne`.

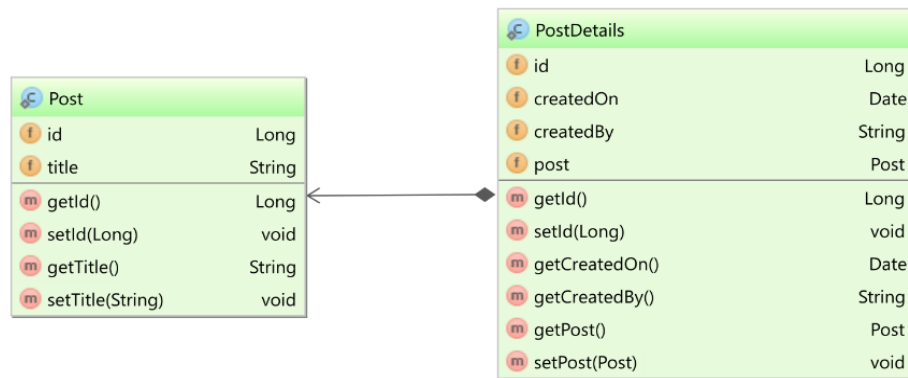


Figura 22: La relación unidireccional @OneToOne  
Fuente.-Tomado de Mihalcea, V. (2020)

El mapeo se realiza a través de la anotación `@OneToOne`, que, al igual que el mapeo de `@ManyToOne`, también puede tomar una `@JoinColumn`.

```
@OneToOne
@JoinColumn(name = "post_id")
private Post;
```

### Bidirectional @OneToOne

Una asociación bidireccional `@OneToOne` permite que la entidad principal mapee también el lado secundario:

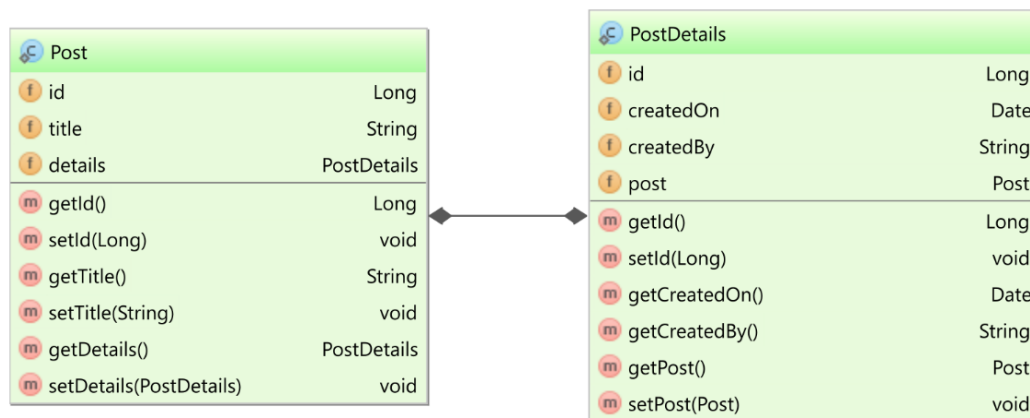


Figura 23: La relación bidireccional @OneToOne  
Fuente.-Tomado de Mihalcea, V. (2020)

El lado principal define un atributo `mappedBy` porque el lado secundario (que aún puede compartir la clave principal con su principal) todavía está a cargo de esta relación JPA:

```
@OneToOne(
    mappedBy = "post",
    cascade = CascadeType.ALL,
    fetch = FetchType.LAZY
)
private PostDetails details;
```

## Relación @ManyToMany

Desde la perspectiva de la base de datos, la anotación @ManyToMany refleja una relación de tabla de muchos a muchos:



Figura 24: La relación de tablas de muchos a muchos  
Fuente.-Tomado de Mihalcea, V. (2020)

Como las relaciones @OneToMany y @OneToOne, la asociación @ManyToMany puede ser unidireccional o bidireccional.

### Unidireccional @ManyToMany List

En nuestro caso, si necesitamos mapear una asociación @ManyToMany unidireccional entre las entidades Post y Tag, tiene sentido que la entidad Post mapee la relación @ManyToMany ya que no hay mucha necesidad de navegar por esta asociación desde el lado del Tag.

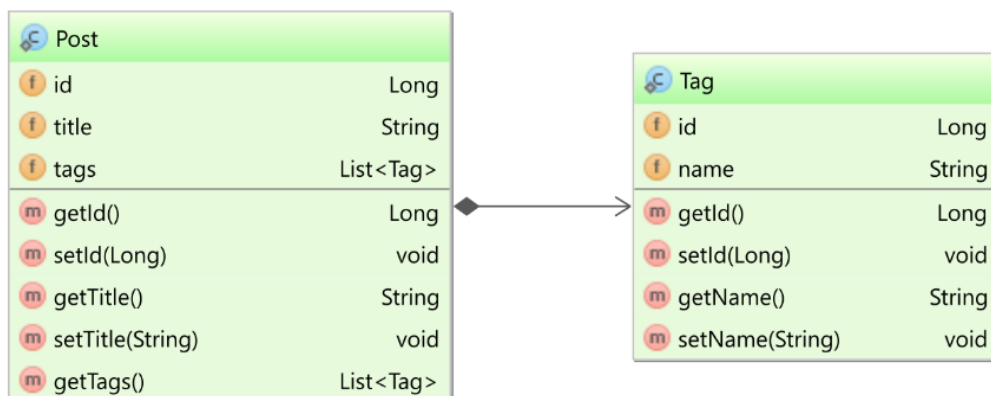


Figura 25: La relación unidireccional @ManyToMany  
Fuente.-Tomado de Mihalcea, V. (2020)

### Unidirectional @ManyToMany Set

Cuando se usa una Lista, la asociación @ManyToMany se comporta como una bolsa y, las bolsas Hibernate no son muy eficientes. Por lo tanto, cambiemos la colección de etiquetas para usar Set:

```
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(
    name = "post_tag",
    joinColumns = @JoinColumn(name = "post_id"),
    inverseJoinColumns = @JoinColumn(name = "tag_id")
)
private Set<Tag> tags = new HashSet<>();
```

## Bidirectional @ManyToMany

La relación bidireccional @ManyToMany se puede navegar tanto desde el lado de la publicación como de la etiqueta.

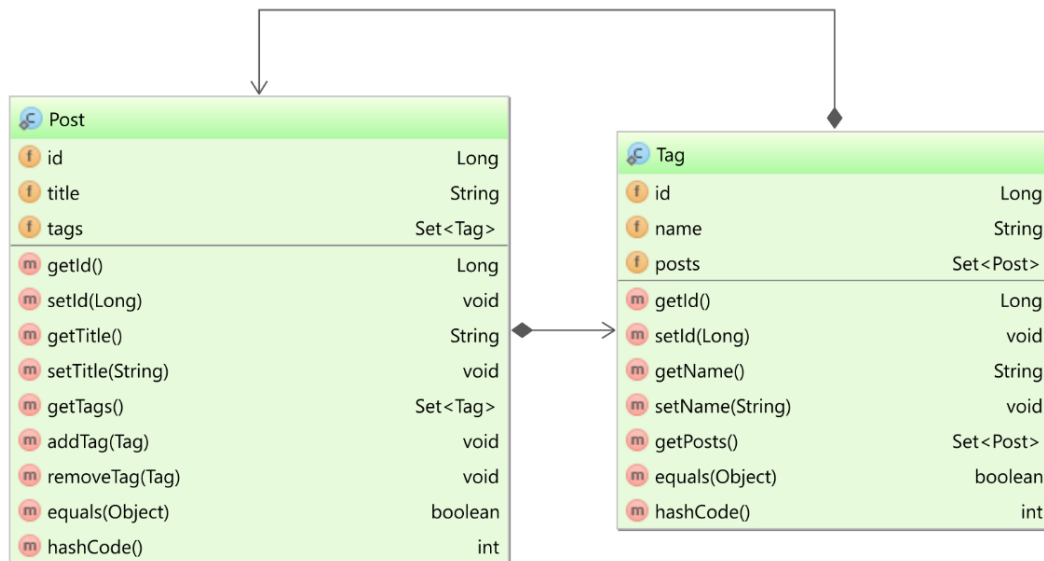


Figura 26: La relación bidireccional @ManyToMany  
Fuente.-Tomado de Mihalcea, V. (2020)

Mientras que en las asociaciones de uno a muchos y de muchos a uno, el lado secundario es el que tiene la clave externa, para una relación de tabla de muchos a muchos, ambos extremos son lados principales y la tabla de unión juega el papel del lado secundario.

Debido a que la tabla de unión está oculta cuando se usa el mapeo @ManyToMany predeterminado, el desarrollador de la aplicación debe elegir un lado propietario y un lado mapeado.

La entidad Post usa el mismo mapeo que se muestra en la sección unidireccional @ManyToMany:

```

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
@JoinTable(
    name = "post_tag",
    joinColumns = @JoinColumn(name = "post_id"),
    inverseJoinColumns = @JoinColumn(name = "tag_id")
)
private Set<Tag> tags = new HashSet<>();
  
```

mientras que la entidad Tag agrega una asociación @ManyToMany que tiene el siguiente aspecto:

```

@ManyToMany(mappedBy="tags")
private Set<Post> posts = new HashSet<>();
  
```

El atributo mappedBy le dice a Hibernate que la entidad Post es el lado propietario de esta asociación. En nuestro caso, el atributo mappedBy de la anotación @ManyToMany en la entidad Tag apunta a la colección de Tags de la entidad Post, lo que significa que la entidad Post va para propagar los cambios de estado de asociación a la tabla de unión subyacente.



### 1.1.4. Herencia

Java, como cualquier otro lenguaje de programación orientado a objetos, hace un uso intensivo de la herencia y el polimorfismo. La herencia permite definir jerarquías de clases que ofrecen diferentes implementaciones de una interfaz común. Conceptualmente, el modelo de dominio define tanto los datos (por ejemplo, entidades persistentes) como el comportamiento (lógica de negocios). Sin embargo, la herencia es más útil para variar el comportamiento que para reutilizar datos (la composición es mucho más adecuada para compartir estructuras). Incluso si los datos (entidades persistentes) y la lógica empresarial (servicios transaccionales) están desacoplados, la herencia aún puede ayudar a variar la lógica empresarial (por ejemplo, patrón Visitor).

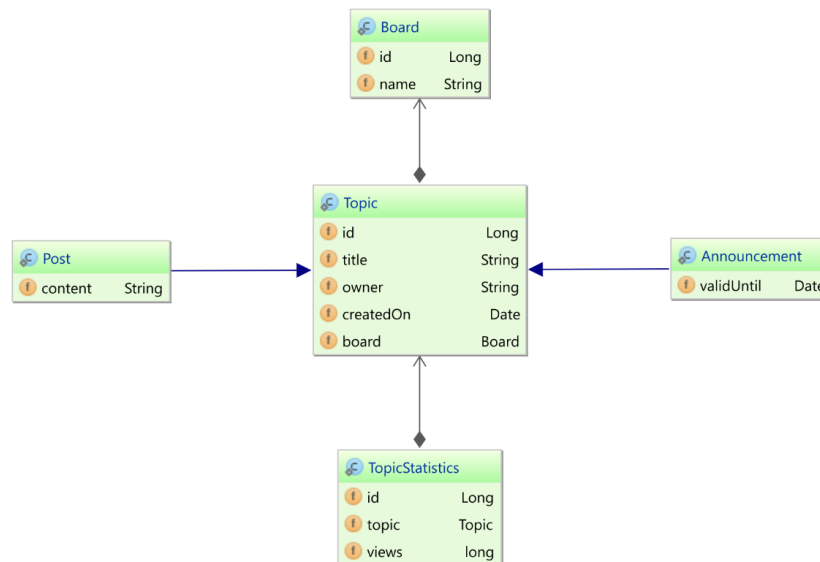


Figura 27: Herencia del modelo de dominio

Fuente.-Tomado de Mihalcea, V. (2020)

La entidad raíz de este modelo de dominio es la entidad Board porque, ya sea directa o indirectamente, todas las demás entidades están asociadas a ella.

```

@Entity
@Table(name = "board")
public class Board {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

}
  
```

El usuario final puede enviar un Post o un Announcement en un Board en particular. Debido a que el Post y el Announcement comparten la misma funcionalidad (difieren sólo en los datos), ambos heredan de una clase base Topic.

La clase Topic define una relación con una entidad Board, por lo tanto, las entidades Board y Announcement también se pueden asociar con una instancia Board.

```

@Entity
  
```

```
@Table(name = "topic")
public class Topic {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String owner;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdOn = new Date();

    @ManyToOne(fetch = FetchType.LAZY)
    private Board;

}
```

Tanto las entidades Post como Announcement amplían la clase Topic y definen sus propios atributos específicos.

```
@Entity
@Table(name = "post")
public class Post extends Topic {

    private String content;

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

}

@Entity
@Table(name = "announcement")
public class Announcement extends Topic {

    @Temporal(TemporalType.TIMESTAMP)
    private Date validUntil;

    public Date getValidUntil() {
        return validUntil;
    }

    public void setValidUntil(Date validUntil) {
        this.validUntil = validUntil;
    }

}
```

TopicStatistics se encuentra en la parte inferior de este modelo de dominio, ya que solo se necesita para fines de monitoreo, sin estar directamente asociado con la lógica comercial principal. Debido a que las estadísticas son necesarias para las entidades Post y Announcement, TopicStatistics define una asociación de entidad Topic.

```
@Entity
@Table(name = "topic_statistics")
public class TopicStatistics {

    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name = "id")
    @MapsId
    private Topic;
    private long views;
}
```

# Resumen

1. La API de persistencia de Java (JPA) es una especificación de Java que gestiona objetos almacenados en una base de datos relacional.
2. JPA es solo una especificación que forma parte de Java EE y se rige por el JCP (Java Community Process). Luego es implementado por frameworks como EclipseLink, Hibernate u OpenJPA.
3. JPA aborda la discrepancia de objeto/relación al asociar tipos de objetos de Java a estructuras de bases de datos.
4. Java, como cualquier otro lenguaje de programación orientado a objetos, hace un uso intensivo de la herencia y el polimorfismo.

# Resumen

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://www.objectdb.com/java/jpa>
- <https://thorben-janssen.com/tutorials/>

## 1.2. JPA E HIBERNATE AVANZADO

### 1.2.1. Uso de flushing, batching y fetching

#### Flushing

El contexto de persistencia actúa como caché de escritura diferida transaccional. La sesión de Hibernate se conoce comúnmente como la caché de primer nivel, ya que cada entidad administrada se almacena en un mapa y, una vez que se carga una entidad, cualquier solicitud sucesiva la sirve desde la caché, evitando así un viaje de ida y vuelta a la base de datos.

Sin embargo, además de almacenar en caché las entidades, el contexto de persistencia actúa como un búfer de transición de estado de entidad. Tanto el EntityManager como la sesión de Hibernate exponen varios métodos de administración de estados de entidades:

- El método de persist toma una entidad transitoria y la hace administrada.
- El método de merge copia el estado interno de una entidad separada en una instancia de identidad recién cargada.
- Hibernate también admite la reconexión de instancias de entidad (por ejemplo, update, saveOrUpdate o lock) que, a diferencia de merge, no requiere buscar una nueva entidad copia de referencia.
- Para eliminar una entidad, el EntityManager define el método remove, mientras que Hibernate ofrece un método delete.

Como cualquier caché de escritura diferida, el contexto de persistencia requiere un vaciado para sincronizar el estado persistente en memoria con la base de datos subyacente. En el momento de la descarga, Hibernate puede detectar si una entidad administrada ha cambiado desde que se cargó y desencadenar una actualización de la fila de la tabla. Este proceso se llama verificación dirty y simplifica enormemente las operaciones de la capa de acceso a los datos.

Por lo tanto, cuando se usa JPA, el desarrollador de la aplicación puede concentrarse sobre los cambios de estado de la entidad, y el contexto de persistencia se encarga de las declaraciones DML subyacentes. De esta manera, la lógica de la capa de acceso a datos se expresa mediante transiciones de estado del modelo de dominio, en lugar de insertar, actualizar o eliminar instrucciones SQL.

Este enfoque es muy conveniente por varias razones:

- Los cambios de estado de la entidad se almacenan en búfer, el contexto de persistencia puede retrasar su ejecución y, por lo tanto, minimiza el intervalo de bloqueo a nivel de fila, asociado con cada operación de escritura de la base de datos.
- Al ejecutarse a la vez, el contexto de persistencia puede usar actualizaciones por batch de JDBC para evitar ejecutar cada declaración en un viaje de ida y vuelta a la base de datos independiente.

#### Modos Flush

El contexto de persistencia puede vaciarse manual o automáticamente. Tanto el EntityManager como la interfaz de sesión nativa de Hibernate definen el método flush() para desencadenar la sincronización entre el modelo de dominio en memoria y las estructuras de la base de datos subyacente. Aun así, sin

un mecanismo de descarga automática, el desarrollador de la aplicación tendría que recordar realizar la descarga antes de ejecutar una consulta o antes de un compromiso de transacción.

Activar una descarga antes de ejecutar una consulta garantiza que los cambios en la memoria sean visibles para la consulta que se está ejecutando actualmente, evitando así la lectura de problemas de coherencia de sus propias escrituras.

Limpiar el contexto de persistencia justo antes de la confirmación de una transacción garantiza que los cambios en la memoria sean duraderos. Sin esta sincronización, las transiciones de estado de entidad pendientes se perderían una vez que se cierre el contexto de persistencia actual.

Para este propósito, JPA e Hibernate definen modos de descarga automática que, desde la perspectiva de la operación de acceso a datos, son más convenientes que la alternativa de descarga manual.

JPA define dos tipos de modo flush:

- FlushModeType.AUTO es el modo predeterminado y activa una descarga antes de la ejecución de cada consulta (JPQL o consulta SQL nativa) y antes de realizar una transacción.
- FlushModeType.COMMIT solo activa una descarga antes de una transacción confirmada.

Hibernate define cuatro tipos de modos flush:

- FlushMode.AUTO es el mecanismo de descarga de la API de Hibernate predeterminado y, aunque descarga el contexto de persistencia en cada confirmación de transacción, no necesariamente desencadena un flush antes de cada ejecución de consulta.
- FlushMode.ALWAYS limpia el contexto de persistencia antes de cada consulta (HQL o SQLquery nativo) y antes de un compromiso de transacción.
- FlushMode.COMMIT desencadena un vaciado de contexto de persistencia solo cuando se confirma la transacción en ejecución.
- FlushMode.MANUAL desactiva el modo de vaciado automático, y el contexto de persistencia solo se puede vaciar manualmente.

## Batching

Las actualizaciones por batch de JDBC y agrupar varias declaraciones puede reducir el número de viajes de ida y vuelta a la base de datos y, por lo tanto, reducir el tiempo de respuesta de las transacciones.

Cuando se trata de traducir transiciones de estado de entidad, Hibernate usa solo PreparedStatement(s) para las operaciones DML de inserción, actualización y eliminación generadas automáticamente. De esta manera, la aplicación está protegida contra ataques de inyección de SQL, y la capa de acceso a datos puede aprovechar mejor el procesamiento por lotes de JDBC y el almacenamiento en caché de declaraciones.

Con JDBC simple, las actualizaciones por lotes requieren una configuración programática porque, en lugar de llamar a executeUpdate, el desarrollador de la aplicación debe usar los métodos addBatch y executeBatch. Desafortunadamente, el ajuste del rendimiento a veces se realiza solo después de que la aplicación se implementa en producción, y el cambio a las declaraciones JDBC por lotes requiere cambios de código significativos.

De forma predeterminada, Hibernate no usa actualizaciones por lotes de JDBC, por lo que al insertar entidades de publicación:

```
for (int i = 0; i < 3; i++) {  
    entityManager.persist(new Post(String.format("Post no. %d", i + 1)));  
}
```

A diferencia de JDBC, Hibernate puede cambiar a `batchedPreparedStatement` (s) con solo una propiedad de configuración, y no se requiere ningún cambio de código:

```
<propertyname="hibernate.jdbc.batch_size" value="5"/>
```

## Fetching

Mientras que en SQL los datos se representan como tuplas, el modelo de dominio orientado a objetos utiliza gráficos de entidades. Hibernate se encarga de la discrepancia de impedancia relacional de objetos, permitiendo que las operaciones de acceso a datos se expresen en forma de transiciones de estado de entidad.

Definitivamente es mucho más conveniente operar en gráficos de entidad y dejar que Hibernate traduzca modificaciones de estado a declaraciones SQL, pero la conveniencia tiene su precio. Todas las sentencias SQL generadas automáticamente deben validarse, no solo por su eficacia, sino también para garantizar su eficiencia.

Con JDBC, el desarrollador de la aplicación tiene control total sobre las sentencias SQL subyacentes y la cláusula `select` dicta la cantidad de datos que se obtienen. Debido a que Hibernate oculta la generación de declaraciones SQL, la eficiencia de búsqueda no es tan transparente como con JDBC. Además, Hibernate hace que sea muy fácil obtener un gráfico de entidad completo con una sola consulta, y la búsqueda excesiva es uno de los problemas de rendimiento más comunes relacionados con JPA.

Para empeorar las cosas, muchos problemas de rendimiento pueden pasar desapercibidos durante el desarrollo porque el conjunto de datos de prueba puede ser demasiado pequeño en comparación con los datos de producción reales. Para este propósito, esta unidad pasa por varias estrategias de búsqueda de persistencia de Java, y explica cuáles son adecuadas para una aplicación basada en datos de alto rendimiento.

La obtención de demasiadas filas o columnas puede tener un gran impacto en el rendimiento de la capa de acceso a datos, e Hibernate no es diferente.

Hibernate puede obtener un conjunto de resultados dado ya sea como una proyección o como un gráfico de entidades. El primero es similar a JDBC y permite transformar el `ResultSet` en una lista de DTO (Data Transfer Objects). Este último es específico de las herramientas ORM, y aprovecha el mecanismo de persistencia automática.

Desafortunadamente, con demasiada frecuencia, esta distinción se olvida y las proyecciones de datos se reemplazan innecesariamente por consultas de entidades. Hay varias razones por las que las consultas de entidad no son una solución universal para leer datos:

1. Si se ha cargado un gráfico de entidades, pero solo se utiliza un subconjunto de todo el gráfico durante la representación de la interfaz de usuario, los datos no utilizados se convertirán en un desperdicio de recursos de la base de datos, ancho de banda de la red y recursos del

servidor (los objetos deben ser creados y luego recuperados por Java Garbage Recopilador sin servir para ningún propósito).

2. Las consultas de entidades son más difíciles de paginar, especialmente si contienen colecciones secundarias.
3. La verificación automática sucia y los mecanismos de bloqueo optimistas solo son relevantes cuando los datos deben modificarse.

Por lo tanto, las proyecciones son ideales al representar subconjuntos de datos, mientras que las consultas de entidad son útiles cuando el usuario desea editar las entidades recuperadas.

Al cargar un gráfico de entidades, el desarrollador de la aplicación debe prestar atención a la cantidad de datos que se obtienen y también al número de declaraciones que se ejecutan.

### 1.2.2. Caching

El almacenamiento en caché está en todas partes. Por ejemplo, la CPU tiene varias capas de almacenamiento en caché para disminuir la latencia asociada con el acceso a los datos de la memoria principal. Al estar cerca de la unidad de procesamiento, el caché de la CPU es muy rápido. Sin embargo, en comparación con la memoria principal, la memoria caché de la CPU es muy pequeña y solo puede almacenar datos a los que se accede con frecuencia.

Para acelerar la lectura y escritura en la unidad de disco subyacente, el sistema operativo también utiliza la memoria caché. Los datos se leen en páginas que se almacenan en caché en la memoria principal, por lo que los datos a los que se accede con frecuencia se brindan desde los búferes del sistema operativo en lugar de la unidad de disco. El caché de disco también mejora las operaciones de escritura porque las modificaciones se pueden almacenar en búfer y vaciar a la vez, mejorando así el rendimiento de escritura.

Dado que los índices y bloques de datos se sirven mejor desde la memoria, la mayoría de los sistemas de bases de datos relacionales emplean una capa de almacenamiento en caché interno.

Entonces, incluso sin configurar explícitamente una solución de almacenamiento en caché, una aplicación empresarial ya usa varias capas de almacenamiento en caché. Sin embargo, el almacenamiento en caché empresarial es a menudo una necesidad y existen varias soluciones que se pueden utilizar para este propósito.

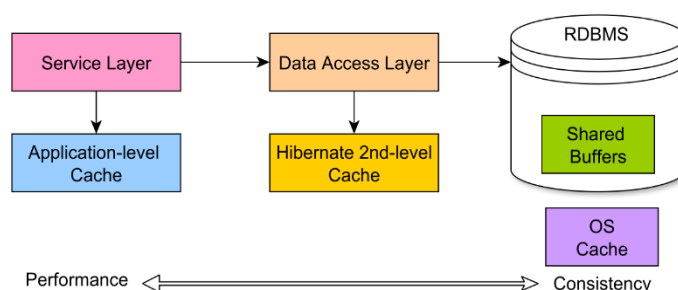


Figura 28: Capas de almacenamiento en caché empresarial

Fuente.-Tomado de Mihalcea, V. (2020)

Como se ilustra en el diagrama anterior, el almacenamiento en caché implica una compensación. Por un lado, eludir la capa de acceso a datos subyacente puede acelerar las lecturas y escrituras. Sin embargo, cuanto más lejos esté situada la solución caché, más difícil le resultará mantener la coherencia con el sistema de base de datos subyacente.



### 1.2.3. Control de concurrencia

En las transacciones JDBC, cada instrucción SQL se ejecuta dentro del alcance de una transacción de base de datos. Para evitar conflictos, los motores de base de datos emplean bloqueos a nivel de fila. Los bloqueos físicos de la base de datos se pueden adquirir implícita o explícitamente. Siempre que se cambia una fila, la base de datos relacional adquiere un bloqueo exclusivo implícito en el registro antes mencionado para evitar conflictos de write-write.

Los bloqueos también se pueden adquirir explícitamente, en cuyo caso el mecanismo de control de concurrencia se denomina bloqueo pesimista. Los bloqueos exclusivos se pueden adquirir explícitamente en la mayoría de los sistemas de bases de datos, mientras que los bloqueos compartidos no son compatibles de forma universal.

El bloqueo pesimista se ocupa de los conflictos de simultaneidad mediante la prevención, lo que puede afectar el rendimiento y la escalabilidad de la aplicación. Por esta razón, para aumentar el rendimiento de las transacciones y al mismo tiempo garantizar una sólida coherencia, muchos marcos de acceso a datos también brindan soporte de bloqueo optimista.

#### Bloqueo optimista de Hibernate

Incluso si el bloqueo pesimista solo se ha agregado en JPA 2.0, el bloqueo optimista se ha admitido desde la versión 1.0. Al igual que el bloqueo pesimista, el mecanismo de control de concurrencia optimista se puede usar implícita o explícitamente.

#### El mecanismo de bloqueo optimista implícito

Para habilitar el mecanismo de bloqueo optimista implícito, la entidad debe proporcionar un atributo @version:

```
@Entity
@Table(name = "post")
public class Post {

    @Id
    private Long id;

    private String title;

    @Version
    private int version;

}
```

# Resumen

1. La sesión de Hibernate se conoce comúnmente como la caché de primer nivel, ya que cada entidad administrada se almacena en un mapa y, una vez que se carga una entidad, cualquier solicitud sucesiva la sirve desde la caché.
2. Los datos se leen en páginas que se almacenan en caché en la memoria principal, por lo que los datos a los que se accede con frecuencia se brindan desde los búferes del sistema operativo.
3. En las transacciones JDBC, cada instrucción SQL se ejecuta dentro del alcance de una transacción de base de datos.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://www.javatpoint.com/jpa-tutorial>
- <https://www.javaguides.net/p/jpa-tutorial-java-persistence-api.html>

## UNIDAD

## 2

## SPRING FRAMEWORK

### LOGRO DE LA UNIDAD DE APRENDIZAJE

Al término de la unidad, el alumno, construye una aplicación web utilizando el patrón de arquitectura MVC, provista por el framework Spring además de integrar JPA para el acceso a datos.

### TEMARIO

- 2.1. Tema 3 : Fundamentos**
  - 2.1.1 : Introducción
  - 2.1.2 : Spring Initializr
  - 2.1.3 : Spring Boot Starters
  - 2.1.4 : Uso de DevTools
  - 2.1.5 : Lombok
- 2.2. Tema 4 : Spring Data JPA**
  - 2.2.1 : Introducción
  - 2.2.2 : Consultas JPQL
  - 2.2.3 : Uso de parámetros
  - 2.2.4 : Paginación y ordenamiento
- 2.3. Tema 5 : Spring Web MVC**
  - 2.3.1 : Arquitectura
  - 2.3.2 : Ciclo de vida
  - 2.3.3 : Configuración
  - 2.3.4 : Uso de estereotipos Spring
- 2.4 Tema 6 : Spring Security**
  - 2.4.1 : Introducción
  - 2.4.2 : Configuración
  - 2.4.3 : Almacenamiento en memoria
  - 2.4.4 : Personalización de la autenticación
- 2.5 Tema 7 : Spring RESTful**
  - 2.5.1 : Introducción
  - 2.5.2 : Configuración

### ACTIVIDADES PROPUESTAS

- Actividad 3: Funcionamiento de Spring Container
- Actividad 4: Uso de Spring Data

- Actividad 5: Uso de Spring MVC
- Actividad 6: Uso de Spring Security
- Actividad 7: Creación de servicio Rest con Spring RESTful

## 2.1. FUNDAMENTOS

### 2.1.1. Introducción

Aunque el filósofo griego Heráclito no era muy conocido como desarrollador de software, parecía tener un buen manejo del tema. Se le ha citado diciendo: "La única constante es el cambio". Esa declaración captura una verdad fundamental del desarrollo de software. La forma en que desarrollamos aplicaciones hoy en día es diferente de lo que era hace un año, hace 5 años, hace 10 años y ciertamente hace 15 años, cuando se introdujo una forma inicial de Spring Framework en el libro de Rod Johnson, *Expert One-on-One J2EE Design and Development* (Wrox, 2002, <http://mng.bz/oVjy>).

En aquel entonces, los tipos más comunes de aplicaciones desarrolladas eran aplicaciones web basadas en navegador, respaldadas por bases de datos relacionales. Si bien ese tipo de desarrollo sigue siendo relevante y Spring está bien equipado para ese tipo de aplicaciones, ahora también estamos interesados en desarrollar aplicaciones compuestas por microservicios destinados a la nube que conservan datos en una variedad de bases de datos. Y un nuevo interés en la programación reactiva tiene como objetivo proporcionar una mayor escalabilidad y un rendimiento mejorado con operaciones sin bloqueo.

A medida que el desarrollo de software evolucionó, Spring Framework también cambió para abordar las preocupaciones de desarrollo moderno, incluidos los microservicios y la programación reactiva. Spring también se propuso simplificar su propio modelo de desarrollo mediante la introducción de Spring Boot. Ya sea que esté desarrollando una aplicación web simple respaldada por una base de datos o construyendo una aplicación moderna construida alrededor de microservicios, Spring es el framework que lo ayudará a lograr sus objetivos. Esta unidad es su primer paso en un viaje a través del desarrollo de aplicaciones modernas con Spring.

#### ¿Qué es Spring?

Cualquier aplicación no trivial está compuesta por muchos componentes, cada uno responsable de su propia parte de la funcionalidad general de la aplicación, coordinándose con los demás elementos de la aplicación para hacer el trabajo. Cuando se ejecuta la aplicación, esos componentes de alguna manera deben crearse e introducirse entre sí.

En esencia, Spring ofrece un container, a menudo denominado Spring ApplicationContext, que crea y administra componentes de la aplicación. Estos componentes, o beans, están conectados entre sí dentro del contexto de la aplicación Spring para hacer una aplicación completa, al igual que los ladrillos, el mortero, la madera, los clavos, la plomería y el cableado están unidos para hacer una casa. El acto de conectar los beans se basa en un patrón conocido como Inyección de Dependencia (en inglés Dependency Injection, DI). En lugar de que los componentes creen y mantengan el ciclo de vida de otros beans que dependen, una aplicación inyectada de dependencia se basa en una entidad separada (el container) para crear y mantener todos los componentes e inyectarlos en los beans que los necesitan. Esto se hace normalmente a través de argumentos de constructor o métodos de acceso.

Por ejemplo, suponga que entre los muchos componentes de una aplicación, hay dos que abordará: un servicio de inventario (para obtener niveles de inventario) y un servicio de producto (para proporcionar información básica del producto). El servicio del producto depende del servicio de inventario para poder proporcionar un conjunto completo de información sobre los productos. La Figura 30 ilustra las relaciones entre estos beans y el Spring ApplicationContext.

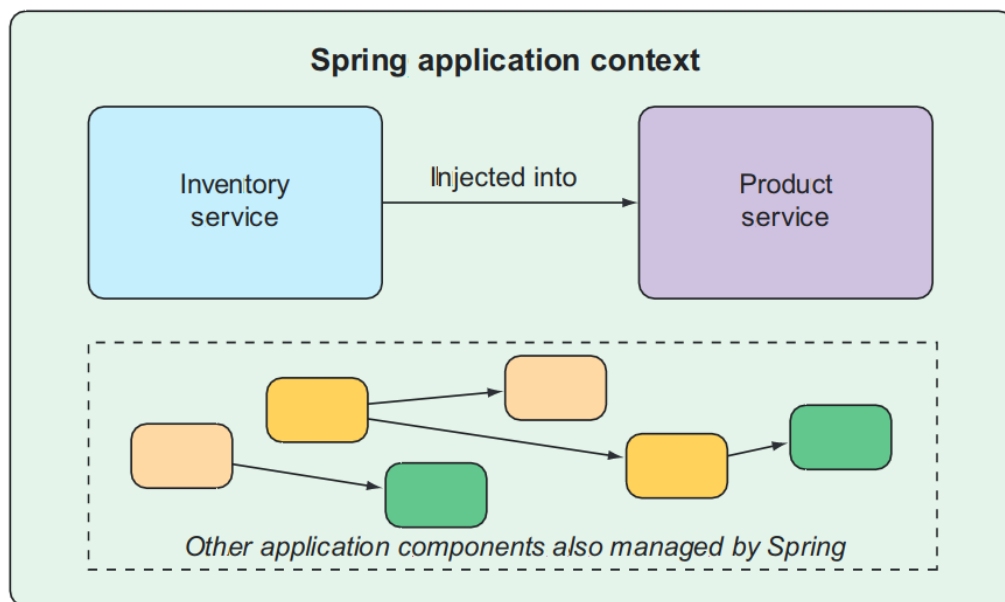


Figura 29: Los componentes de la aplicación son administrados e inyectados entre sí por el Spring ApplicationContext.

Fuente.-Tomado de Mihalcea, V. (2020)

Además de su container principal, Spring y una cartera completa de bibliotecas relacionadas ofrecen un framework web, una variedad de opciones de persistencia de datos, un framework de seguridad, integración con otros sistemas, monitoreo del tiempo de ejecución, soporte de microservicios, un modelo de programación reactiva, y muchas otras características necesarias para el desarrollo de aplicaciones modernas.

Históricamente, la forma en que guiaba el Spring ApplicationContext para conectar beans era con uno o más archivos XML que describían los componentes y su relación con otros componentes. Por ejemplo, el siguiente XML declara dos beans, un bean `InventoryService` y un bean `ProductService`, y conecta el bean `InventoryService` a `ProductService` mediante un argumento de constructor:

```
<bean id="inventoryService" class="com.example.InventoryService" />

<bean id="productService" class="com.example.ProductService" >
  <constructor-arg ref="inventoryService" />
</bean>
```

En versiones recientes de Spring, sin embargo, una configuración basada en Java es más común. La siguiente clase de configuración basada en Java es equivalente a la configuración XML:

```
@Configurationpublic class ServiceConfiguration {
    @Bean public InventoryService inventoryService() {
        return new InventoryService();
    }
    @Bean public ProductService productService() {
        return new ProductService(inventoryService());
    }
}
```

La anotación `@Configuration` le indica a Spring que esta es una clase de configuración que proporcionará beans al Spring ApplicationContext. Los métodos de la clase de configuración están anotados con `@Bean`, lo que indica que los objetos que devuelven deben agregarse como beans en el ApplicationContext (donde, de forma predeterminada, sus respectivos ID de bean serán los mismos que los nombres de los métodos que los definen).

La configuración basada en Java ofrece varios beneficios sobre la configuración basada en XML, incluida una mayor seguridad de tipos y una refactorabilidad mejorada. Aun así, la configuración explícita con Java o XML solo es necesaria si Spring no puede configurar automáticamente los componentes.

La configuración automática tiene sus raíces en las técnicas Spring conocidas como autowiring y component scanning. Con el component scanning, Spring puede descubrir automáticamente componentes de la ruta de clase de una aplicación y crearlos como beans en el Spring ApplicationContext. Con el autowiring, Spring inyecta automáticamente los componentes con los otros beans de los que dependen.

Más recientemente, con la introducción de Spring Boot, la configuración automática ha ido mucho más allá del component scanning y el autowiring. Spring Boot es una extensión de Spring Framework que ofrece varias mejoras de productividad. La más conocida de estas mejoras es la configuración automática, en la que Spring Boot puede hacer conjeturas razonables sobre qué componentes deben configurarse y conectarse entre sí, según las entradas en el classpath, las variables de entorno y otros factores.

La configuración automática de Spring Boot ha reducido drásticamente la cantidad de configuración explícita (ya sea con XML o Java) necesaria para construir una aplicación. De hecho, cuando termine el ejemplo de esta unidad, tendrá una aplicación Spring en funcionamiento que solo tiene una línea de código de configuración Spring.

Spring Boot mejora tanto el desarrollo de Spring que es difícil imaginar el desarrollo de aplicaciones Spring sin él. Por esa razón, este libro trata a Spring y Spring Boot como si fueran lo mismo. Usaremos Spring Boot tanto como sea posible, y la configuración explícita solo cuando sea necesario. Y, debido a que la configuración de Spring XML es la forma tradicional de trabajar con Spring, nos centraremos principalmente en la configuración basada en Java de Spring.

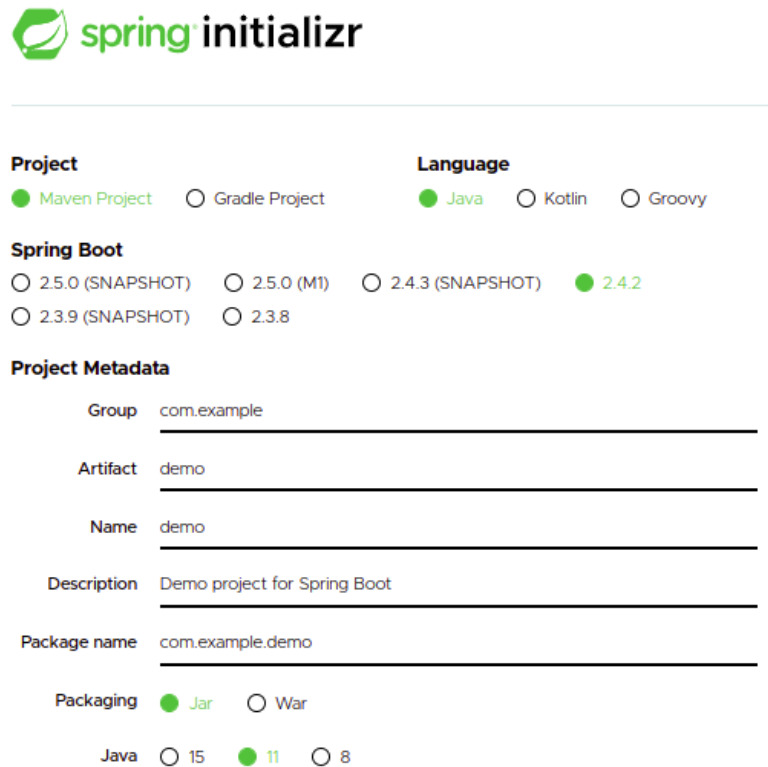
Pongámonos en marcha y podrá empezar a escribir su primera aplicación con Spring.

### 2.1.2. Spring Initializr

Spring Initializr es tanto una aplicación web basada en navegador como una API REST, que puede producir una estructura esquemática de proyecto Spring que puede desarrollar con cualquier funcionalidad que desee. A continuación, se muestran varias formas de usar Spring Initializr:

- Desde la aplicación web en <http://start.spring.io>.
- Desde la línea de comando usando el comando `curl`.
- Desde la línea de comando usando la interfaz de línea de comando de Spring Boot.
- Al crear un nuevo proyecto con Spring Tool Suite.
- Al crear un nuevo proyecto con IntelliJ IDEA.
- Al crear un nuevo proyecto con NetBeans.

En esta unidad, y a lo largo de esta guía, les mostraré cómo crear un nuevo proyecto usando Spring Tool Suite. Spring Tool Suite es un fantástico entorno de desarrollo Spring. Pero también ofrece una práctica función Spring Boot Dashboard que no está disponible en ninguna de las otras opciones de IDE.



The image shows the Spring Initializr web form. At the top is the 'spring initializr' logo. Below it, there are three sections: 'Project', 'Language', and 'Spring Boot'. The 'Project' section has radio buttons for 'Maven Project' (selected), 'Gradle Project', and 'Groovy Project'. The 'Language' section has radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for '2.5.0 (SNAPSHOT)', '2.5.0 (M1)', '2.4.3 (SNAPSHOT)', '2.4.2' (selected), '2.3.9 (SNAPSHOT)', and '2.3.8'. Below these is the 'Project Metadata' section with input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). At the bottom, there are radio buttons for 'Packaging' (Jar selected, War) and a 'Java' version selector with radio buttons for 15, 11 (selected), and 8.

Figura 30: Spring Initializr  
Fuente. - Elaboración propia

### 2.1.3. Spring Boot Starter

Los starters son un conjunto de descriptores de dependencia convenientes que puede incluir en su aplicación. Obtiene una ventanilla única para todas las tecnologías de Spring y relacionadas que necesita sin tener que buscar código de muestra y copiar y pegar un montón de descriptores de dependencia. Por ejemplo, si desea comenzar a usar Spring y JPA para el acceso a la base de datos, incluya la dependencia spring-boot-starter-data-jpa en su proyecto.

Los principiantes contienen muchas de las dependencias que necesita para poner en marcha un proyecto rápidamente y con un conjunto coherente y compatible de dependencias transitivas gestionadas.

Todos los starters oficiales siguen un patrón de denominación similar; spring-boot-starter-\*, donde \* es un tipo particular de aplicación. Esta estructura de nomenclatura está destinada a ayudar cuando necesite encontrar un starter. La integración de Maven en muchos IDE le permite buscar dependencias por nombre. Por ejemplo, con el complemento Eclipse o STS apropiado instalado, puede presionar ctrl-espacio en el editor POM y escribir "spring-boot-starter" para obtener una lista completa.

Los starters de terceros no deben comenzar con spring-boot, ya que está reservado para los artefactos oficiales de Spring Boot. Más bien, un iniciador de terceros generalmente comienza con el nombre del



proyecto. Por ejemplo, un proyecto de inicio de terceros llamado `thirdpartyproject` normalmente se llamaría `thirdpartyproject-spring-boot-starter`.

Spring Boot proporciona los siguientes starters de aplicaciones en el grupo `org.springframework.boot`:

Name	Description
<code>spring-boot-starter</code>	Core starter, including auto-configuration support, logging and YAML
<code>spring-boot-starter-activemq</code>	Starter for JMS messaging using Apache ActiveMQ
<code>spring-boot-starter-amqp</code>	Starter for using Spring AMQP and Rabbit MQ
<code>spring-boot-starter-aop</code>	Starter for aspect-oriented programming with Spring AOP and AspectJ
<code>spring-boot-starter-artemis</code>	Starter for JMS messaging using Apache Artemis
<code>spring-boot-starter-batch</code>	Starter for using Spring Batch
<code>spring-boot-starter-cache</code>	Starter for using Spring Framework's caching support
<code>spring-boot-starter-data-cassandra</code>	Starter for using Cassandra distributed database and Spring Data Cassandra
<code>spring-boot-starter-data-cassandra-reactive</code>	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive
<code>spring-boot-starter-data-couchbase</code>	Starter for using Couchbase document-oriented database and Spring Data Couchbase

Figura 31: Starters de Spring Boot

Fuente.- Tomado de <https://docs.spring.io/spring-boot/docs/2.4.2/reference/htmlsingle/#using-boot-starter>

#### 2.1.4. Uso de DevTools

DevTools proporciona a los desarrolladores de Spring algunas herramientas útiles para el tiempo de desarrollo. Entre ellos se encuentran

- Reinicio automático de la aplicación cuando cambia el código.
- Actualización automática del navegador cuando cambian los recursos destinados al navegador (como plantillas, JavaScript, hojas de estilo, etc.).
- Desactivación automática de cachés de plantillas.
- Consola H2 integrada si la base de datos H2 está en uso.

Es importante comprender que DevTools no es un complemento IDE, ni requiere que use un IDE específico. Funciona igualmente bien en Spring Tool Suite, IntelliJ IDEA y NetBeans. Además, debido a que solo está diseñado para fines de desarrollo, es lo suficientemente inteligente como para deshabilitarse cuando se implementa en un entorno de producción. Por ahora, centrémonos en las características más útiles de Spring Boot DevTools, comenzando con un reinicio automático de la aplicación.

##### Reinicio automático de la aplicación

Con DevTools como parte de su proyecto, podrá realizar cambios en el código Java y los archivos de propiedades en el proyecto y ver esos cambios aplicados después de un breve momento.

DevTools monitorea los cambios, y cuando ve que algo ha cambiado, reinicia automáticamente la aplicación.

Más precisamente, cuando DevTools está en juego, la aplicación se carga en dos cargadores de clases separados en la máquina virtual Java(JVM). Un cargador de clases se carga con su código Java, archivos de propiedades y prácticamente cualquier cosa que esté en la ruta `src/main/`. Estos son elementos que probablemente cambien con frecuencia. El otro cargador de clases está cargado con bibliotecas de dependencia, que no es probable que cambien con tanta frecuencia.

Cuando se detecta un cambio, DevTools recarga solo el cargador de clases que contiene el código de su proyecto y reinicia el contexto de la aplicación Spring, pero deja el otro cargador de clases y la JVM intactos. Aunque sutil, esta estrategia permite una pequeña reducción del tiempo que lleva iniciar la aplicación.

La desventaja de esta estrategia es que los cambios en las dependencias no estarán disponibles en los reinicios automáticos. Eso se debe a que el cargador de clases que contiene las bibliotecas de dependencia no se recarga automáticamente. Esto significa que cada vez que agregue, cambie o elimine una dependencia en su especificación de compilación, deberá reiniciar la aplicación para que esos cambios surtan efecto.

### **Actualización automática del navegador y deshabilitar cache de plantilla**

De forma predeterminada, las opciones de plantilla como Thymeleaf y FreeMarker están configuradas para comprobar los resultados del análisis de la plantilla, de modo que no sea necesario volver a analizar las plantillas con cada solicitud que atiendan. Esto es excelente en la producción, ya que genera un pequeño beneficio de rendimiento.

Las plantillas en caché, sin embargo, no son tan buenas en el momento del desarrollo. Las plantillas en caché hacen que sea imposible realizar cambios en las plantillas mientras se ejecuta la aplicación y ver los resultados después de actualizar el navegador. Incluso si ha realizado cambios, la plantilla almacenada en caché seguirá en uso hasta que reinicie la aplicación.

DevTools soluciona este problema desactivando automáticamente todo el almacenamiento en caché de plantillas. Realice todos los cambios que desee en sus plantillas y sepa que está a solo una actualización del navegador de ver los resultados.

Cuando DevTools está en ejecución, habilita automáticamente un servidor LiveReload junto con su aplicación. Por sí solo, el servidor LiveReload no es muy útil. Pero cuando se combina con un complemento de navegador LiveReload correspondiente, hace que su navegador se actualice automáticamente cuando se realizan cambios en plantillas, imágenes, hojas de estilo, JavaScript, etc., de hecho, casi todo lo que termina siendo servido en su navegador. LiveReload tiene complementos de navegador para los navegadores Google Chrome, Safari y Firefox.

### **Consola construida H2**

Si elige usar la base de datos H2 para el desarrollo, DevTools también habilitará automáticamente una Consola H2 a la que puede acceder desde su navegador web. Solo necesita apuntar su navegador web a <http://localhost:8080/h2-console> para obtener información sobre los datos con los que está trabajando su aplicación.

### 2.1.5. Lombok

Como puede ver, esta es una clase de dominio Java común y corriente, que define las tres propiedades necesarias para describir un Ingredient. Quizás lo más inusual de la clase Ingredient es que parece faltar los métodos getter y setter, por no mencionar métodos útiles como equals(), hashCode(), toString() y otros.

```
import lombok.Data;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class Ingredient {

    private final String id;

    private final String name;

    private final Type;

    public static enum Type {
        WRAP,
        PROTEIN,
        VEGGIES,
        CHEESE,
        SAUCE
    }
}
```

No los ve en la lista en parte para ahorrar espacio, pero también porque está utilizando una biblioteca increíble llamada Lombok para generar automáticamente esos métodos en tiempo de ejecución. De hecho, Lombok proporciona la anotación @Data a nivel de clase y le dice a Lombok que genere todos los métodos que faltan, así como un constructor que acepta todas las propiedades finales como argumentos. Al usar Lombok, puede mantener el código para Ingredient sencillo y corto.

Lombok no es una biblioteca de Spring, pero es tan increíblemente útil que resulta difícil desarrollar sin ella. Es un salvavidas cuando se necesita mantener ejemplos de código en un formato breve y sencillo.

Para usar Lombok, deberá agregarlo como una dependencia en su proyecto. Si está utilizando Spring Tool Suite, hacer clic derecho en el archivo pom.xml y seleccionar Editar Starters en la opción del menú contextual de Spring. Aparecerá la misma selección de dependencias que se le proporcionó al crear un proyecto, lo que le dará la oportunidad de agregar o cambiar las dependencias seleccionadas. Busque la opción Lombok, asegúrese de que esté marcada y haga clic en Aceptar; Spring Tool Suite lo agregará automáticamente a la especificación de su construcción.

Alternativamente, puede agregarlo manualmente con la siguiente entrada en pom.xml:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
```

`</dependency>`

Esta dependencia le proporcionará anotaciones de Lombok (como `@Data`) durante el desarrollo y la generación automática de métodos en el tiempo de ejecución. Pero también deberá agregar Lombok como una extensión en su IDE, o su IDE mostrará errores sobre métodos faltantes y propiedades finales que no se están configuradas. Visite <https://projectlombok.org/> para averiguar cómo instalar Lombok en su IDE de elección.

# Resumen

1. Spring ofrece un container, a menudo denominado Spring ApplicationContext, que crea y administra componentes de la aplicación.
2. Spring Inicializa una aplicación web basada en navegador como una API REST, que puede producir una estructura esquemática de proyecto Spring que puede desarrollar con cualquier funcionalidad que desee.
3. Los starters son un conjunto de descriptores de dependencia convenientes que puede incluir en su aplicación.
4. DevTools monitorea los cambios, y cuando ve que algo ha cambiado, reinicia automáticamente la aplicación.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://docs.spring.io/spring-framework/docs/current/reference/html/index.html>
- <https://www.educba.com/software-development/software-development-tutorials/spring-tutorial/>

## 2.2. SPRING DATA JPA

### 2.2.1. Introducción

El proyecto Spring Data es un proyecto general bastante grande que consta de varios subproyectos, la mayoría de los cuales se centran en la persistencia de datos con una variedad de diferentes tipos de bases de datos. Algunos de los proyectos de Spring Data más populares incluyen estos:

- Spring Data JPA-JPA persistencia contra una base de datos relacional.
- Spring Data MongoDB-Persistence a una base de datos de documentos Mongo.
- Spring Data Neo4j-Persistence a una base de datos gráfica de Neo4j.
- Spring Data Redis-Persistence a un almacén de valores-clave de Redis.
- Spring Data Cassandra-Persistence a una base de datos de Cassandra.

Una de las características más interesantes y útiles proporcionadas por Spring Data para todos estos proyectos es la capacidad de crear repositorios automáticamente, basados en una interfaz de especificación de repositorios.

Spring Data JPA está disponible para aplicaciones Spring Boot con el starter JPA. Esta dependencia de inicio no solo trae Spring Data JPA, sino que también incluye transitivamente Hibernate como la implementación de JPA:

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

Si desea utilizar una implementación de JPA diferente, deberá, al menos, excluir la dependencia de Hibernate e incluir la biblioteca JPA de su elección. Por ejemplo, para usar EclipseLink en lugar de Hibernate, necesitará modificar la compilación de la siguiente manera:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
  <exclusions>
    <exclusion>
      <artifactId>hibernate-entitymanager</artifactId>
      <groupId>org.hibernate</groupId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>2.5.2</version>
</dependency>
```

Tenga en cuenta que pueden ser necesarios otros cambios, dependiendo de su elección de implementación de JPA. Consulte la documentación de la implementación de JPA elegida para obtener más detalles.

```
import javax.persistence.Entity;
import javax.persistence.Id;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access = AccessLevel.PRIVATE, force = true)
@Entity
public class Ingredient {
    @Id
    private final String id;
    private final String name;
    private final Type;

    public static enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }
}
```

### Declarar repositorios JPA

CrudRepository declara métodos para operaciones CRUD (crear, leer, actualizar, eliminar). Tenga en cuenta que está parametrizado, siendo el primer parámetro el tipo de entidad en el que el repositorio debe persistir y el segundo parámetro es el tipo de propiedad de ID de entidad. Para IngredientRepository, los parámetros deben ser Ingredient y String.

```
import org.springframework.data.repository CrudRepository;

public interface IngredientRepository extends CrudRepository <Ingredient, String> {

}
```

Cuando se inicia la aplicación, Spring Data JPA genera automáticamente una implementación sobre la marcha. Esto significa que los repositorios están listos para usar desde el principio. Simplemente inyéctelos en los controladores y listo. Los métodos proporcionados por CrudRepository son excelentes para la persistencia de entidades de propósito general.

### 2.2.2. Consultas JPQL

Imagine que, además de las operaciones CRUD básicas proporcionadas por CrudRepository, también necesita buscar todas las orders entregadas a un código Zip determinado. Resulta que esto se puede solucionar fácilmente agregando la siguiente declaración de método a OrderRepository.

```
List<Order> findByDeliveryZip(String deliveryZip);
```

Al generar la implementación del repositorio, Spring Data examina cualquier método en la interfaz del repositorio, analiza el nombre del método e intenta comprender el propósito del método en el contexto del objeto persistente (una Order, en este caso). En esencia, Spring Data define una especie de lenguaje específico de dominio en miniatura(DSL) donde los detalles de persistencia se expresan en firmas de métodos de repositorio.

Spring Data sabe que este método está destinado a encontrar orders, porque ha parametrizado CrudRepository con Order. El nombre del método, findByDeliveryZip(), deja en claro que este método debe encontrar todas las entidades Order al hacer coincidir su propiedad deliveryZip con el valor pasado como parámetro al método.

El método findByDeliveryZip() es bastante simple, pero Spring Data también puede manejar nombres de métodos aún más interesantes. Los métodos de repositorio se componen por un verbo, un sujeto opcional, la palabra By y un predicado. En el caso de findByDeliveryZip(), el verbo es encontrar y el predicado es DeliveryZip; el asunto no se especifica y se supone que es una orden.

Consideremos otro ejemplo más complejo. Suponga que necesita consultar todos los pedidos entregados a un código ZIP determinado dentro de un rango de fechas determinado. En ese caso, el siguiente método, cuando se agrega a OrderRepository, puede resultar útil:

```
List<Order> readOrdersByDeliveryZipAndPlacedAtBetween(String deliveryZip, Date startDate, Date endDate);
```

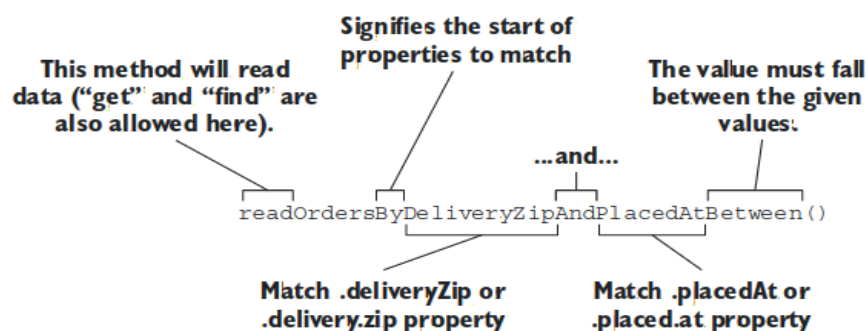


Figura 32: Spring Data analiza las firmas de métodos de repositorio para determinar la consulta que se debe realizar.

Fuente.-Tomado de Craig, W. (2020)

Aunque el tema del método es opcional, aquí dice Orders. Spring Data ignora la mayoría de las palabras en un tema, por lo que podría nombrar el método `readPuppiesBy ...` y aún encontraría entidades Order, ya que ese es el tipo con el que CrudRepository está parametrizado.

El predicado sigue a la palabra By en el nombre del método y es la parte más interesante de la firma del método. En este caso, el predicado se refiere a dos propiedades de Order: `deliveryZip` y `placedAt`. La propiedad `deliveryZip` debe ser igual al valor pasado al primer parámetro del método. La palabra clave `Between` indica que el valor de `deliveryZip` debe estar entre los valores pasados a los dos últimos parámetros del método.

Además de una operación `Equals` implícita y la operación `Between`, las firmas del método SpringData también pueden incluir cualquiera de estos operadores:

- `IsAfter`, `After`, `IsGreaterThan`, `GreaterThan`
- `IsGreaterThanEqual`, `GreaterThanEqual`
- `IsBefore`, `Before`, `IsLessThan`, `LessThan`



- IsLessThanEqual, LessThanEqual
- IsBetween, Between
- IsNull, Null
- IsNotNull, NotNull
- IsIn, In
- IsNotIn, NotIn
- IsStartingWith, StartingWith, StartsWith
- IsEndingWith, EndingWith, EndsWith
- IsContaining, Containing, Contains
- IsLike, Like
- IsNotLike, NotLike
- IsTrue, True
- IsFalse, False
- Is, Equals
- IsNot, Not
- IgnoringCase, IgnoresCase

Como alternativas para IgnoringCase e IgnoresCase, puede colocar AllIgnoringCase o AllIgnoresCase en el método para ignorar el caso de todas las comparaciones de cadenas. Por ejemplo, considere el siguiente método:

```
List<Order> findByDeliveryToAndDeliveryCityAllIgnoresCase(String deliveryTo, String deliveryCity);
```

Finalmente, también puede colocar OrderBy al final del nombre del método para ordenar los resultados por una columna específica. Por ejemplo, para ordenar por la propiedad deliveryTo:

```
List<Order> findByDeliveryCityOrderByDeliveryTo(String city);
```

Aunque la convención de nomenclatura puede ser útil para consultas relativamente simples, no hace falta mucha imaginación para ver que los nombres de los métodos pueden salirse de control para consultas más complejas. En ese caso, siéntase libre de nombrar el método como desee y anótelos con @Query para especificar explícitamente la consulta que se realizará cuando se llame al método, como muestra este ejemplo:

```
@Query("Order o where o.deliveryCity='Seattle'")  
List<Order> readOrdersDeliveredInSeattle();
```

En este uso simple de @Query, solicita todas las orders entregadas en Seattle. Pero puede usar @Query para realizar prácticamente cualquier consulta que pueda imaginar, incluso cuando sea difícil o imposible lograr la consulta siguiendo la convención de nomenclatura.

### 2.2.3. Uso de parámetros

Las aplicaciones para empresas desarrolladas con Spring a menudo necesitan ejecutar consultas complejas en la base de datos. En tal escenario, debe informar a Spring Data JPA sobre las consultas que debe ejecutar. Lo haces usando la anotación @Query.

```
import lombok.AllArgsConstructor;  
import lombok.Builder;
```

```
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
@Entity(name = "Book")
@Builder
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(unique=true)
    private String isbn;
    private String title;
    private String author;
    private boolean status;
}
```

### JPQL Select @Query con parámetros de índice

Una forma de pasar parámetros de método a una consulta es mediante un índice.

Definamos una consulta personalizada utilizando Java Persistence Query Language (JPQL) para encontrar un Book para un title y author determinados.

Para consultar un Book con parámetros de índice usando JPQL ejecute:

```
@Query("SELECT b FROM Book b WHERE b.title = ?1 and b.author = ?2")
Book findBookByTitleAndAuthorIndexJpql(String title, String authorName);
```

En el código anterior, el parámetro del método de title se asignará al parámetro de consulta con índice 1. De manera similar, el authorName se asignará al parámetro de consulta con índice 2.

Es importante tener en cuenta que el orden de los índices de los parámetros de consulta y los parámetros del método deben ser el mismo.

### SQL nativo Select @Query con parámetros de índice

Para una consulta SQL nativa personalizada, debe establecer el indicador nativeQuery en true. El código para usar SQL nativo es este.

```
@Query(value = "SELECT * FROM Book WHERE title = ?1 and author = ?2",
        nativeQuery = true)
Book findBookByTitleAndAuthorIndexNative(String title, String authorName);
```

### JPQL @Query con parámetros con nombre

Otro enfoque para pasar parámetros a una consulta es con parámetros con nombre. En este enfoque, pasa los valores de los parámetros del método a los parámetros de enlace de la consulta. Para hacerlo, use la anotación @Param con el nombre del parámetro de vinculación en la definición del método.

```
@Query("SELECT b FROM Book b WHERE b.title = :title and b.author= :author")
```

```
Book findBookByTitleAndAuthorNamedJpql(@Param("title") String title, @Param("author") String author);
```

La anotación @Param en el código anterior vincula los nombres de los parámetros de consulta con los parámetros del método.

### SQL @Query nativo con parámetros con nombre

Consultar con parámetros con nombre utilizando SQL nativo es similar a JPQL.

```
@Query(value = "SELECT * FROM Book WHERE title = :title and author= :author", nativeQuery = true)
Book findBookByTitleAndAuthorNamedNative(@Param("title") String title, @Param("author") String author);
```

### 2.2.4. Paginación y ordenamiento

La paginación offset es muy popular y Spring Boot (más precisamente, Spring Data Commons) brinda soporte a través de las API de Page y Slice. Sin embargo, mientras su proyecto evoluciona y los datos se acumulan, depender de la paginación offset puede generar penalizaciones en el rendimiento, incluso si no fue un problema al inicio del proyecto.

Tratar con la paginación offset significa que puede ignorar la penalización de rendimiento inducida al desechar *n* registros antes de alcanzar el desplazamiento deseado. Una *n* mayor conduce a una penalización significativa del rendimiento. Otra penalización es el SELECT adicional necesario para contar el número total de registros (especialmente si necesita contar cada página recuperada). Si bien la paginación keyset(búsqueda) puede ser el camino a seguir (como un enfoque diferente), la paginación offset se puede optimizar para evitar este SELECT adicional.

Con conjuntos de datos relativamente pequeños, el offset y keyset proporcionan casi el mismo rendimiento. Pero, ¿puede garantizar que el conjunto de datos no crecerá con el tiempo o puede controlar el proceso de crecimiento? La mayoría de las empresas comienzan con una pequeña cantidad de datos, pero cuando el éxito se acelera, la cantidad de datos también puede aumentar muy rápidamente.

#### Escaneo de índice en offset y keyset

El escaneo de índice en desplazamiento atravesará el rango de índices desde el principio hasta el desplazamiento especificado. Básicamente, el desplazamiento representa el número de registros que se deben omitir antes de incluirlos en el resultado, lo que significa que también se debe incluir un recuento.

En compensación, dependiendo de la cantidad de datos que se deban obtener y omitir (y tenga en cuenta que las tablas generalmente "crecen" rápidamente), este enfoque puede conducir a una degradación significativa del rendimiento. El enfoque de desplazamiento atravesará los registros ya mostrados.

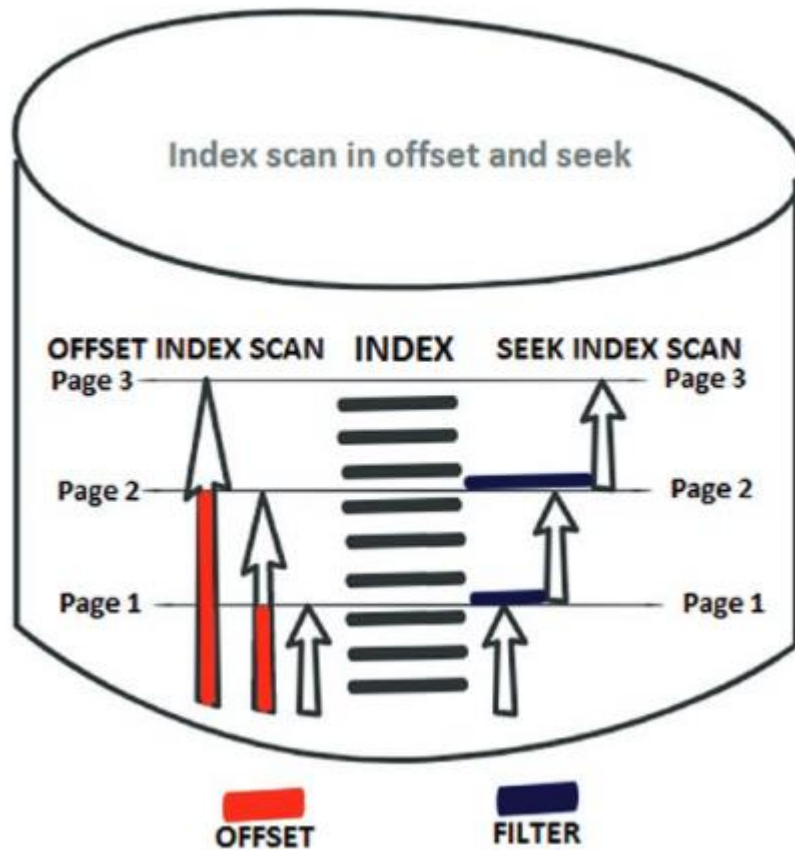


Figura 33: Escaneo de índice en offset vs paginación keyset.

Fuente.- Tomado de Leonard, A. (2020)

Por otro lado, el escaneo de índice en keyset recorrerá solo los valores requeridos, comenzando con el último valor anterior (omite los valores hasta el último valor obtenido previamente). En keyset, el rendimiento permanece aproximadamente constante en relación con el aumento de los registros de la tabla.

### Pros y contras de la paginación offset

Tenga en cuenta los siguientes pros y contras de la paginación offset.

Contras:

- Las inserciones pueden hacer que las páginas se muevan.
- Cada vez, numera las filas desde el principio.
- Aplica un filtro para eliminar las filas innecesarias.
- Si el desplazamiento es mayor que el número de filas en los resultados ordenados, no se devuelven filas

Pros:

- Puede buscar páginas arbitrarias.

## Paginación offset

Spring Boot proporciona soporte integrado para la paginación offset a través de la API Page. Considere la tabla de autor correspondiente a la entidad Author en la Figura 34.



Figura 34: La tabla de entidad de Author.

Fuente.- Tomado de Leonard, A. (2020)

Los siguientes ejemplos se basan en la entidad Author y el repositorio AuthorRepository para dar forma a una forma sencilla de implementar la paginación. Para empezar, existen al menos cinco formas de obtener el conjunto de resultados, de la siguiente manera:

- Invocar findAll(Pageable) sin un ordenamiento explícito (no recomendado):

```
authorRepository.findAll(PageRequest.of(page, size));
```

- Invocar findAll(Pageable) con ordenamiento:

```
authorRepository.findAll(PageRequest.of(page, size, Sort.by(Sort.Direction.ASC, "price")));
```

- Utilice el mecanismo Spring Data Query Builder para definir nuevos métodos en su repositorio:

```
Page < Author > findByName(String name, Pageable);
Page < Author > queryFirst10ByName(String name, Pageable);
```

- Use JPQL y @Query con y sin SELECT COUNT explícito:

```
@Query(value = "SELECT a FROM Author a WHERE a.genre = ?1", countQuery = "SELECT COUNT(*) FROM Author a WHERE a.genre = ?1")
public Page < Author > fetchByGenreExplicitCount(String genre, Pageable pageable);
```

```
@Query("SELECT a FROM Author a WHERE a.genre = ?1") public Page < Author > fetchByGenre(String genre, Pageable pageable);
```

- Use consulta nativa y @Query con y sin SELECT COUNT explícito:

```
@Query(value = "SELECT * FROM author WHERE genre = ?1", countQuery = "SELECT COUNT(*) FROM author WHERE genre = ?1", nativeQuery = true)
public Page < Author > fetchByGenreNativeExplicitCount(String genre, Pageable pageable);
```

```
@Query(value = "SELECT * FROM author WHERE genre = ?1", nativeQuery = true)
```

```
public Page < Author > fetchByGenreNative(String genre, Pageable pageable);
```

Además, el repositorio clásico necesario para admitir la paginación de Author extenderá PagingAndSortingRepository, de la siguiente manera:

```
@Repository
```

```
public interface AuthorRepository extends PagingAndSortingRepository < Author, Long > {}
```

A continuación, un método de servicio puede buscar páginas de Author en orden ascendente por edad, de la siguiente manera:

```
public Page < Author > fetchNextPage(int page, int size) {  
    return authorRepository.findAll(PageRequest.of(page, size, Sort.by(Sort.Direction.ASC, "age")));  
}
```

# Resumen

1. Spring Data es un proyecto que consta de varios subproyectos, la mayoría de los cuales se centran en la persistencia de datos con una variedad de diferentes tipos de bases de datos.
2. Spring Data tiene la capacidad de crear repositorios automáticamente, basados en una interfaz de especificación de repositorios.
3. Spring Data JPA está disponible para aplicaciones Spring Boot con el starter JPA.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://docs.spring.io/spring-data/jpa/docs/2.4.3/reference/html/>
- <https://www.javaguides.net/p/spring-data-jpa-tutorial.html>

## 2.3. SPRING WEB MVC

### 2.3.1. Arquitectura

Spring Web MVC es un framework para crear servicios web o aplicaciones web y, a menudo, se abrevia a Spring MVC o simplemente MVC. MVC significa Model-View-Controller y es uno de los patrones de diseño comunes en la programación OO (orientada a objetos).

Spring MVC está diseñado con el principio de open-close en mente (abierto para extensión, cerrado para modificación). `DispatcherServlet` es el núcleo de Spring MVC y contiene un `Servlet` `WebApplicationContext` (que contiene los controladores, `ViewResolver`, `HandlerMapping` y otros resolutores) que delega a un `Root WebApplicationContext` (que contiene los beans de servicio y repositorio de la aplicación).

Detecta los siguientes tipos de beans y los usa si los encuentra; de lo contrario, se utilizan los valores predeterminados: `HandlerMapping`, `HandlerAdapter`, `HandlerExceptionResolver`, `ViewResolver`, `LocaleResolver`, `ThemeResolver`, `MultipartResolver` y `FlashMapManager`.

Puede integrarse directamente con una tecnología de visualización como JSP, Velocity o FreeMarker, o puede devolver una respuesta serializada como JSON a través de un `ViewResolver` o un mapeador de objetos incorporado utilizando DTO (objetos de transferencia de datos) o cualquier POJO.

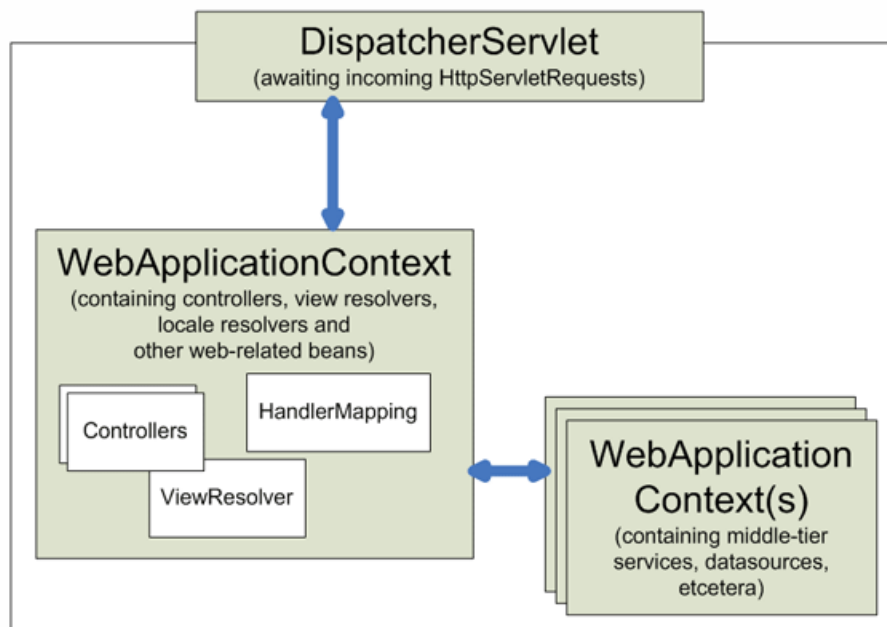


Figura 35: Contexto en Spring MVC

Fuente: - Tomado de <https://docs.spring.io/spring-framework/docs/current/reference/html/web.html>

### 2.3.2. Ciclo de vida

Cada vez que un usuario hace clic en un enlace o envía un formulario en su navegador web, se activa una solicitud. La descripción del trabajo de una solicitud es la de un mensajero. Al igual que un cartero o un repartidor de FedEx, una solicitud vive para llevar información de un lugar a otro. La solicitud es una criatura ocupada. Desde el momento en que sale del navegador hasta que regresa con una



respuesta, realiza varias paradas, dejando cada vez un poco de información y recogiendo algo más. La Figura 36 muestra todas las paradas que hace la solicitud a medida que viaja a través de Spring MVC.

Cuando la solicitud sale del navegador**(1)**, lleva información sobre lo que solicita el usuario. Como mínimo, la solicitud llevará la URL solicitada. Pero también puede contener datos adicionales, como la información enviada en un formulario por el usuario.

La primera parada en los viajes de la solicitud es en DispatcherServlet de Spring. Como la mayoría de los marcos web basados en Java, Spring MVC canaliza las solicitudes a través de un solo servidor de controlador frontal. Un controlador frontal es un patrón de aplicación web común en el que un solo servlet delega la responsabilidad de una solicitud a otros componentes de una aplicación para realizar el procesamiento real. En el caso de Spring MVC, DispatcherServlet es el controlador frontal.

El trabajo de DispatcherServlet es enviar la solicitud a un controlador Spring MVC. Un controlador es un componente de Spring que procesa la solicitud. Pero una aplicación típica puede tener varios controladores, y DispatcherServlet necesita ayuda para decidir a qué controlador enviar la solicitud. Por lo tanto, DispatcherServlet consulta una o más asignaciones de controladores**(2)** para averiguar dónde será la próxima parada de la solicitud. El handler mapping presta especial atención a la URL transportada por la solicitud al tomar su decisión.

Una vez que se ha elegido un controlador apropiado, DispatcherServlet envía la solicitud en su camino alegre al controlador elegido**(3)**. En el controlador, la solicitud deja su carga útil (la información enviada por el usuario) y espera pacientemente mientras el controlador procesa esa información. (En realidad, un controlador bien diseñado realiza poco o ningún procesamiento por sí mismo y, en cambio, delega la responsabilidad de la lógica de negocios a uno o más objetos de servicio).

La lógica realizada por un controlador a menudo da como resultado cierta información que debe llevarse al usuario y mostrarse en el navegador. Esta información se conoce como modelo. Pero enviar información sin procesar al usuario no es suficiente, debe formatearse en un formato fácil de usar, generalmente HTML. Para eso, la información debe entregarse a una vista, generalmente una página JavaServer (JSP).

Una de las últimas cosas que hace un controlador es empaquetar los datos del modelo e identificar el nombre de una vista que debería representar la salida. Luego envía la solicitud, junto con el modelo y el nombre de la vista, de regreso al DispatcherServlet**(4)**. Para que el controlador no se acople a una vista en particular, el nombre de la vista transferido a DispatcherServlet no identifica directamente una JSP específica. Ni siquiera sugiere necesariamente que la vista sea una JSP. En cambio, solo lleva un nombre lógico que se usará para buscar la vista real que producirá el resultado. DispatcherServlet consulta a un solucionador de vistas **(5)** para asignar el nombre de la vista lógica a una implementación de vista específica, que puede ser o no una JSP.

Ahora que DispatcherServlet sabe qué vista generará el resultado, el trabajo de la búsqueda casi ha terminado. Su última parada es en la implementación de la vista**(6)**, típicamente una JSP, donde entrega los datos del modelo. El trabajo de la solicitud finalmente está hecho. La vista utilizará los datos del modelo para generar la salida que será devuelta al cliente por el objeto de respuesta**(7)**.

Como puede ver, una solicitud pasa por varios pasos a lo largo de su camino para producir una respuesta para el cliente. La mayoría de estos pasos tienen lugar dentro del marco de Spring MVC, en

los componentes que se muestran en la Figura 36.

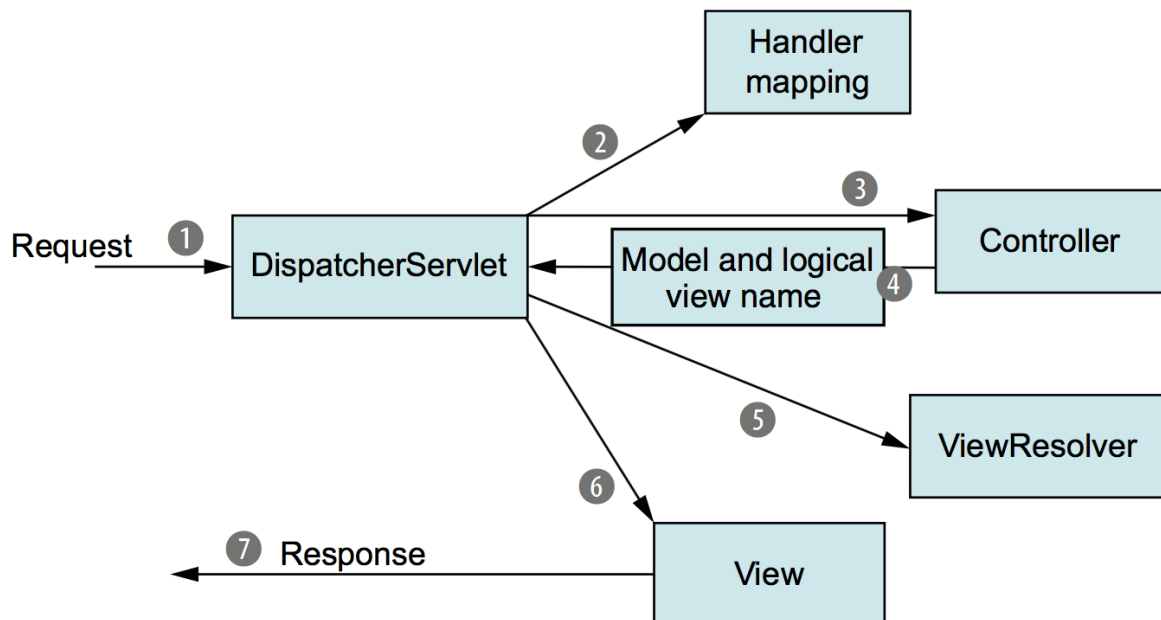


Figura 36: DispatcherServlet

Fuente.- Tomado de Adam, L. (2020).

Según la Figura 36, parece que hay muchas partes móviles que configurar. Afortunadamente, gracias a algunos avances en las versiones más recientes de Spring, es fácil comenzar con Spring MVC.

### 2.3.3. Configuración

Primero, agregue la dependencia a su proyecto.

```

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

```

Luego, habilite Web MVC en el proyecto, por ejemplo:

```

import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.context.annotation.Configuration;

@EnableWebMvc
public class WebConfig {

}

```

Asegúrese de usar `@ComponentScan` o `@Bean` para definir sus Spring Beans. También puede tener servicios y repositorios.

## Controllers

También asegúrese de usar `@ComponentScan` o `@Bean` o definiciones de bean XML para definir sus Spring Beans. En una aplicación típica, también puede tener servicios y repositorios, pero para este capítulo, nos centraremos solo en los componentes Spring Web MVC.

Anote una clase con la anotación `@Controller` para marcarla como controlador. El uso de `@RestController` es similar, pero se usa para los controladores de servicios web RESTful: se supone que el valor devuelto de cada método se convierte en un valor de retorno, como JSON (similar a cuando el método se anota con `@ResponseBody`). El uso de cualquiera de estas anotaciones permitirá que su clase sea detectada por el escaneo de componentes.

También puede anotar la clase con `@RequestMapping` para establecer un prefijo de URL que se aplicará a todos los métodos de la clase. Por ejemplo, anotar una clase de controlador con `@RequestMapping("/api/v1/")` agregará el prefijo `"/api/v1/"` al mapeo de URL de cada método dentro del controlador.

## Request Mappings

Anote un método con `@RequestMapping` o una de las anotaciones de tipo de método HTTP correspondientes, como `@GetMapping`. Cada asignación de solicitud de método debe coincidir con una solicitud entrante específica. Si más de un método coincidiera con la misma solicitud HTTP, Spring arrojará un error al inicializar el controlador (generalmente en el momento del inicio).

Cada método de la clase debe estar anotado con uno de los siguientes para asignarlo a una ruta URL correspondiente:

- `@RequestMapping`: debe establecer el método HTTP y las propiedades de la ruta, por ejemplo,  
`@RequestMapping(method = RequestMethod.PUT, path = "/courses/{id}")`
- `@GetMapping("/{path}")`: se asigna a HTTP GET.
- `@PostMapping("/{path}")`: se asigna a HTTP POST.
- `@DeleteMapping("/{path}")`: se asigna a HTTP DELETE.
- `@PutMapping("/{path}")`: se asigna a HTTP PUT.
- `@PatchMapping("/{path}")`: se asigna a HTTP PATCH.

Puede proporcionar una URL con valores incrustados en ella para definir variables de ruta que se pueden asignar a parámetros. Por ejemplo, en la URL `"/images/{filename}/raw"`, el nombre del archivo corresponde a una variable de ruta:

```
@GetMapping(value = "/images/{filename}/raw", produces = MediaType.IMAGE_JPEG_VALUE)
public void getImage(@PathVariable String filename, OutputStream output) {

}
```

En este ejemplo, dado el parámetro `OutputStream` se puede utilizar para proporcionar los datos de salida (imagen en este caso). Puede usar `produces` para establecer el tipo de contenido de la respuesta (`"image/jpeg"` en este caso). También puede anotar un método con `@ResponseStatus` para cambiar el estado HTTP exitoso a otro que no sea el predeterminado (200). Por ejemplo, lo siguiente cambiaría el código de estado de respuesta a 201:

```
@ResponseStatus(HttpStatus.CREATED)
```

```
@PostMapping(value = "/courses", consumes = MediaType.APPLICATION_JSON_VALUE)
public void create(@RequestBody final CourseDto course) {
}
```

También puede especificar parámetros de solicitud o valores de encabezado para hacer una asignación de solicitud aún más específica. Por ejemplo, `@PostMapping (value = "/courses", params = "lang = java", headers = "X-custom-header")` solo coincidiría con las solicitudes POST con un parámetro de consulta llamado "lang" con el valor de "java" y un encabezado llamado X-custom-header presente.

### Path Regular Expressions

También puede utilizar una expresión regular dentro de las definiciones de variables de ruta para limitar la coincidencia de rutas. Por ejemplo, lo siguiente solo coincidiría con las rutas que terminen con un número:

```
@GetMapping("/courses/{id:\\d+}")
public CourseDto course(@PathVariable final Long id) {

}
```

### Mapping Method Parameters

Las anotaciones válidas para los parámetros de un método mapeado en un controlador son las siguientes:

- `@RequestParam`: un parámetro de consulta.
- `@PathVariable`: una parte de la ruta.
- `@MatrixVariable`: estas variables pueden aparecer en cualquier parte de la ruta, y el carácter igual ("=") se utiliza para dar valores y el punto y coma (";") para delimitar cada variable de matriz. En la misma ruta, también podemos repetir el mismo nombre de variable o separar diferentes valores usando el carácter coma (",").
- `@RequestHeader`: un encabezado HTTP de la solicitud.
- `@CookieValue`: un valor de una cookie.
- `@RequestPart`: anotación que se puede usar para asociar la parte de una solicitud "multipart/form-data" con un argumento de método. Los tipos de argumentos de método admitidos incluyen `MultipartFile` junto con la abstracción `MultipartResolver` de Spring y `javax.servlet.http.Part` junto con las solicitudes de varias partes de Servlet 3.0, o de lo contrario, para cualquier otro argumento de método, el contenido de la parte se pasa a través de un `HttpMessageConverter` teniendo en cuenta el Encabezado "Content-Type" de la parte de la solicitud.
- `@ModelAttribute`: se puede usar para obtener acceso a un objeto desde el modelo. Por ejemplo, `public String handleCustomer(@ModelAttribute("customer") Customer customer)` obtendría el objeto `Customer` usando la clave "customer".
- `@SessionAttribute`: atributo de la sesión.
- `@RequestAttribute`: atributo de la solicitud.

## Response Body

Añote un método con `@ResponseBody` para decirle a Spring que use el valor de retorno del método como el cuerpo de la respuesta HTTP.

Alternativamente, si anota la clase con `@RestController`, esto implica que el cuerpo de la respuesta es el valor de retorno para cada método.

Spring convertirá automáticamente la respuesta al valor adecuado utilizando implementaciones de `HttpMessageConverter`. Spring MVC viene con convertidores integrados. Otros tipos de respuesta permitidos son:

- `HttpEntity`
- `ResponseEntity<T>` - Contiene una entidad para ser serializada por la lógica de conversión de Spring y valores HTTP, como un estado HTTP.
- `HttpHeaders`
- `String` (nombre de la vista para resolver)
- `View`
- `Map` or `Model`
- `ModelAndView`
- `DeferredResult<V>`, `Callable<V>`, `ListenableFuture<V>`, o `CompletableFuture<V>` - Resultados asíncronos.
- `ResponseBodyEmitter`
- `SseEmitter`
- `StreamingResponseBody`
- Reactive types like Flux

## Views

Spring Web MVC incluye soporte para varios renderizadores de vista diferentes, como JSP, FreeMarker, Groovy templates y Velocity. Según la tecnología de vista que se elija, el `ViewResolver` seleccionado expondrá el modelo, la sesión y los atributos de solicitud de manera adecuada. Spring MVC también incluye una biblioteca de etiquetas JSP para ayudar en la creación de páginas JSP. Aquí hay un diagrama general que resume cómo funciona Spring MVC, con algunas detalles que faltan, como el manejo de excepciones (cubriremos esto más adelante):

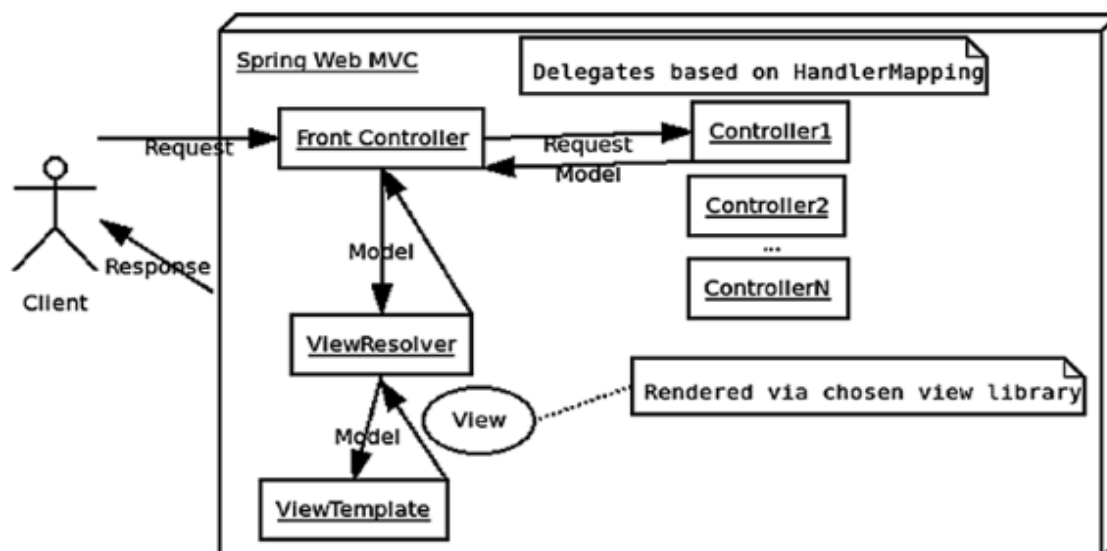


Figura 37: Spring Web MVC request/response

Fuente. – Tomado de Adam, L. (2020).

### 2.3.4. Uso de estereotipos Spring

La anotación principal en Spring es el `@Component` del paquete `org.springframework.stereotype`. Esta anotación marca una clase a partir de la cual se creará un bean. Dichas clases se recogen automáticamente mediante la configuración basada en anotaciones (clases anotadas con `@Configuration`) y el escaneo de classpath (habilitado al anotar una clase de configuración con `@ComponentScan`). En la Figura 2-12, se muestran las anotaciones más importantes utilizadas en este libro y se agrupan por su propósito. (Las anotaciones de Spring Boot no forman parte de este diagrama).

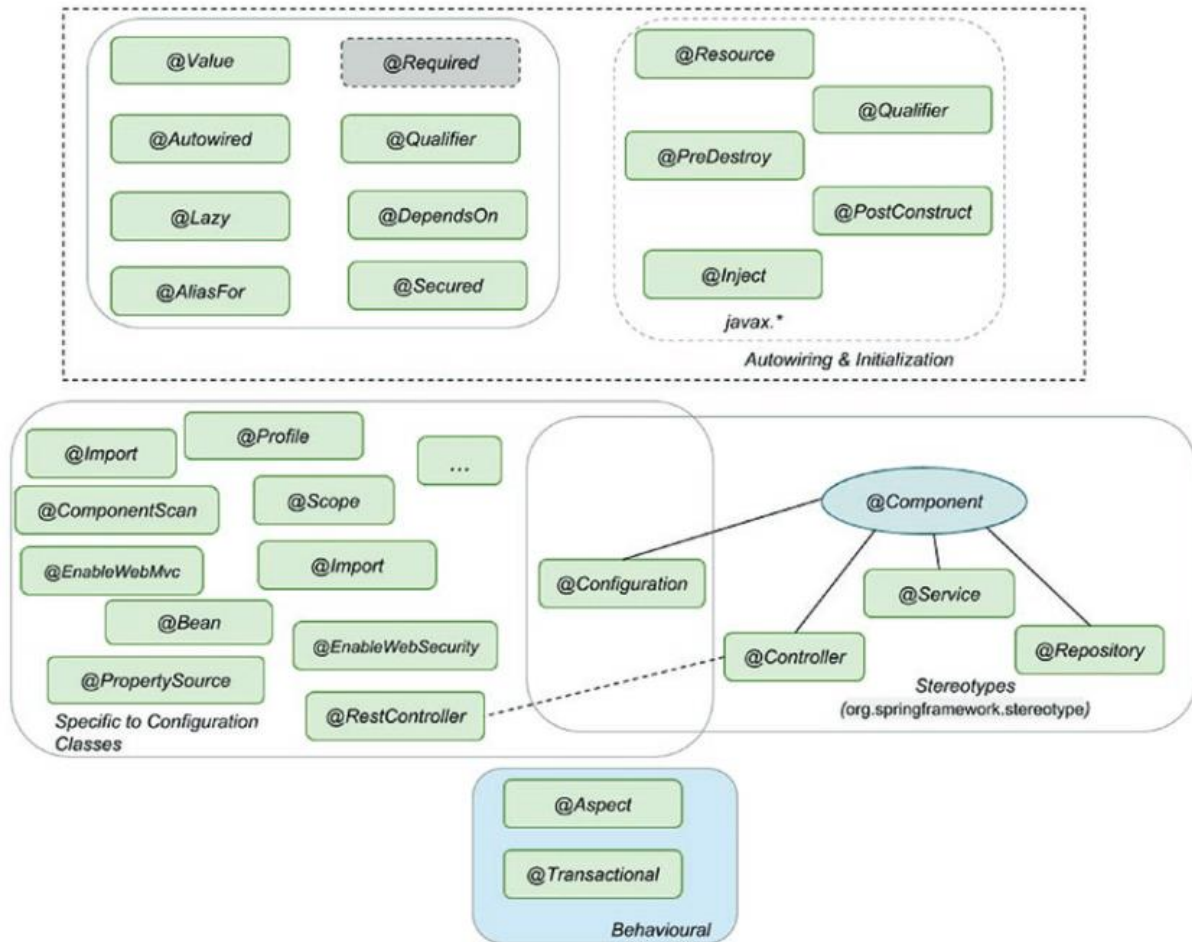


Figura 38: Anotaciones más importantes utilizadas

Fuente.- Tomado de Cosmina, L. (2020)

- Las anotaciones de estereotipos se utilizan para marcar las clases de acuerdo con su propósito.
- `@Component`: plantilla para cualquier componente administrado por Spring (bean).
- `@Repository`: plantilla para un componente utilizado para proporcionar acceso a datos, especialización de la anotación `@Component` para la capa DAO.
- `@Service`: plantilla para un componente que brinda ejecución de servicio, especialización de la anotación `@Component` para la capa de Servicio.

- `@Controller`: plantilla para un componente web, especialización de la anotación `@Component` para la capa web.
- `@Configuration`: clase de configuración que contiene definiciones de bean (métodos anotados con `@Bean`).
- Las anotaciones de autowiring y de inicialización se utilizan para definir qué dependencia se inyecta y cómo se ve el bean. Por ejemplo.
  - `@Autowired`: anotación principal para este grupo; se utiliza en dependencias para indicarle a Spring IoC que se encargue de inyectarlas. Se puede usar en campos, constructores, setters e incluso métodos. Use `@Qualifier` de Spring para especificar el nombre del bean para inyectar.
  - `@Inject`: anotación equivalente a `@Autowired` del paquete `javax.inject`. Úselo con `@Qualifier` de `javax.inject` para especificar el nombre del bean a inyectar.
  - `@Resource`: anotación equivalente a `@Autowired` del paquete `javax.annotation`. Proporciona un atributo de nombre para especificar el nombre del bean que se inyectará.
  - `@Required`: anotación de Spring que marca una dependencia como obligatoria. Se puede usar en métodos de setter, pero en Spring 5.1 quedó en desuso en favor del uso de inyección de constructor para la configuración requerida.
  - `@Lazy`: la dependencia se inyectará la primera vez que se use. Aunque esta anotación existe, evite usarla si es posible. Cuando se inicia una aplicación Spring, las implementaciones de `ApplicationContext` crean y configuran con entusiasmo todos los beans singleton como parte del proceso de inicialización, esto es útil porque los errores de configuración en la configuración o el entorno de soporte (por ejemplo, la base de datos) se pueden detectar rápidamente. Cuando se utiliza `@Lazy`, la detección de estos errores puede retrasarse.
- Las anotaciones que aparecen solo en (y en) clases anotadas con `@Configuration` son específicas de la infraestructura; definen los recursos de configuración, los componentes y su alcance.
- Las anotaciones de comportamiento son anotaciones que definen el comportamiento de un bean. También podrían denominarse anotaciones de proxy, porque implican la creación de proxies para interceptar solicitudes a los beans que se configuran con ellos.

# Resumen

1. Spring Web MVC es un framework para crear servicios web o aplicaciones web y, a menudo, se abrevia a Spring MVC o simplemente MVC.
2. Spring MVC está diseñado con el principio de open-close en mente (abierto para extensión, cerrado para modificación).
3. La anotación principal en Spring es `@Component` del paquete `org.springframework.stereotype`.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://docs.spring.io/spring-framework/docs/current/reference/html/web.html>
- <https://www.baeldung.com/spring-boot>



## 2.4. SPRING SECURITY

### 2.4.1. Introducción

La información es probablemente el elemento más valioso que tenemos ahora; los delincuentes buscan formas de robar nuestros datos e identidades infiltrándose en aplicaciones no seguras. Como desarrolladores de software, debemos tomar medidas para proteger la información que reside en nuestras aplicaciones. Ya sea una cuenta de correo electrónico protegida con un par de nombre de usuario y contraseña o una cuenta de corretaje protegida con un PIN comercial, la seguridad es un aspecto crucial de la mayoría de las aplicaciones.

#### Habilitar Spring Security

El primer paso para proteger su aplicación Spring es agregar la dependencia starter de seguridad Spring Boot a su compilación. En el archivo pom.xml del proyecto, agregue la siguiente dependencia:

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

Esta dependencia es lo único que se requiere para asegurar una aplicación. Cuando se inicia la aplicación, la autoconfiguración detectará que Spring Security está en la ruta de clases y establecerá alguna configuración de seguridad básica.

Si desea probarlo, inicie la aplicación e intente visitar la página de inicio (o cualquier página para el caso). Se le solicitará la autenticación con un cuadro de diálogo de autenticación básica HTTP. Para superarlo, deberá proporcionar un nombre de usuario y una contraseña. El nombre de usuario es user. En cuanto a la contraseña, se genera aleatoriamente y se escribe en el archivo log de la aplicación. La entrada del log se verá así:

Using default security password: 087cfc6a-027d-44bc-95d7-cbb3a798a1ea

Suponiendo que ingrese el nombre de usuario y la contraseña correctamente, se le otorgará acceso a la aplicación.

Parece que proteger las aplicaciones de Spring es un trabajo bastante sencillo. Pero antes de adelantarnos, consideremos qué tipo de configuración automática de seguridad ha proporcionado. Al no hacer nada más que agregar el starter de seguridad a la compilación del proyecto, obtiene las siguientes características de seguridad:

- Todas las rutas HTTP requieren autenticación.
- No se requieren roles o authorities específicos.
- No hay una página de inicio de sesión.
- La autenticación se solicita con HTTP basic authentication.
- Solo hay un usuario; el nombre de usuario es user.

Este es un buen comienzo, pero creo que las necesidades de seguridad de la mayoría de las aplicaciones serán bastante diferentes de estas características de seguridad básicas.

Tiene más trabajo que hacer si va a proteger adecuadamente una aplicación. Deberá al menos configurar Spring Security para hacer lo siguiente:

- Solicitar autenticación con una página de inicio de sesión, en lugar de un cuadro de diálogo básico HTTP.
- Proporcione múltiples users y habilite una página de registro para que los nuevos puedan registrarse.
- Aplique diferentes reglas de seguridad para diferentes rutas de solicitud. La página de inicio y las páginas de registro, por ejemplo, no deberían requerir autenticación en absoluto.

Para satisfacer sus necesidades de seguridad, tendrá que escribir alguna configuración explícita, anulando lo que le ha proporcionado la autoconfiguración.

### 2.4.2. Configuración

A lo largo de los años, ha habido varias formas de configurar Spring Security, incluida una configuración larga basada en XML. Afortunadamente, varias versiones recientes de Spring Security han admitido la configuración basada en Java, que es mucho más fácil de leer y escribir.

Antes de que termine esta unidad, habrá configurado todas sus necesidades de seguridad en la configuración de Spring Security basada en Java. Pero para empezar, podrá acceder fácilmente escribiendo la clase de configuración básica que se muestra a continuación.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

¿Qué hace esta configuración de seguridad básica? Bueno, no mucho, pero te acerca un paso más a la funcionalidad de seguridad que necesitas. Si intenta volver a acceder a la página de inicio, se le pedirá que inicie sesión. Pero en lugar de que se le solicite un cuadro de diálogo básico HTTP, se le mostrará un formulario de inicio de sesión.

Esta es una pequeña mejora: solicitar el inicio de sesión con una página web (incluso si su apariencia es bastante simple) siempre es más fácil de usar que un cuadro de diálogo básico HTTP. Sin embargo, la tarea actual es configurar un almacén de usuarios que pueda manejar más de un usuario.

Resulta que Spring Security ofrece varias opciones para configurar una tienda de usuarios, incluidas estas:

- Un almacenamiento de usuario en memoria.
- Un almacenamiento de usuarios basada en JDBC.
- Un almacenamiento de usuarios respaldada por LDAP.
- Un servicio personalizado de detalles de usuario.

### 2.4.3. Almacenamiento en memoria

Un lugar donde se puede guardar la información del usuario es la memoria. Suponga que solo tiene un puñado de usuarios, ninguno de los cuales probablemente cambiará. En ese caso, puede ser bastante sencillo definir a esos usuarios como parte de la configuración de seguridad. Por ejemplo, la siguiente lista muestra cómo configurar dos usuarios, "buzz" y "woody", en un almacenamiento de usuarios en memoria.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
```

```

        .withUser("buzz")
        .password("infinity")
        .authorities("ROLE_USER")
    .and()
    .withUser("woody")
    .password("bullseye")
    .authorities("ROLE_USER");
}

```

Como puede ver, AuthenticationManagerBuilder emplea una API de estilo constructor para configurar los detalles de autenticación. En este caso, una llamada al método inMemoryAuthentication() le brinda la oportunidad de especificar la información del usuario directamente en la configuración de seguridad.

Cada llamada a withUser() inicia la configuración de un usuario. El valor dado a withUser() es el nombre de usuario, mientras que la contraseña y las autorizaciones concedidas se especifican con los métodos password() y authorities(). Como se muestra, ambos usuarios tienen la autoridad ROLE\_USER. El buzz del usuario está configurado para tener infinito como contraseña. Asimismo, la contraseña de woody es la bullseye.

El almacenamiento de usuarios en memoria es conveniente para realizar pruebas o para aplicaciones muy simples, pero no permite la edición fácil de los usuarios. Si necesita agregar, eliminar o cambiar un usuario, deberá realizar los cambios necesarios y luego reconstruir y volver a implementar la aplicación.

#### 2.4.4. Personalización de la autenticación

Tiene sentido conservar los datos del usuario en base de datos. Usaremos Spring Data JPA para que los datos persistan en una base de datos relacional. Para aprovechar mejor el repositorio de Spring Data creamos un objeto de dominio y la interfaz de repositorio que representa y conserva la información del usuario.

##### Definiendo el dominio y la persistencia del usuario

Para capturar toda esa información, crear una clase de User de la siguiente manera.

```

@Entity
@Data
@NoArgsConstructor(access = AccessLevel.PRIVATE, force = true)
@RequiredArgsConstructor
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private final String username;
    private final String password;
    private final String fullname;
    private final String street;
    private final String city;
    private final String state;
}

```

```
private final String zip;
private final String phoneNumber;

@Override
public Collection < ? extends GrantedAuthority > getAuthorities() {
    return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}
}
```

Sin duda habrá notado que la clase `User` está un poco más involucrada que cualquiera de las otras entidades definidas. Además de definir varios atributos, `User` también implementa la interfaz `UserDetails` de Spring Security.

Las implementaciones de `UserDetails` proporcionarán información esencial del usuario al framework, como qué authorities se otorgan al usuario y si la cuenta del usuario está habilitada o no.

El método `getAuthorities()` debería devolver una colección de authorities otorgadas al usuario. Los diversos métodos `is___Expired()` devuelven un booleano para indicar si la cuenta del usuario está habilitada o vencida.

Para la entidad `User`, el método `getAuthorities()` simplemente devuelve una colección que indica que todos los usuarios tendrán la autoridad `ROLE_USER`. Con la entidad `User` creada, ahora puede crear la interfaz del repository:

```
public interface UserRepository extends CrudRepository <User, Long> {
    User findByUsername(String username);
}
```

Además de las operaciones CRUD proporcionadas al extender `CrudRepository`, `UserRepository` define un método `findByUsername()` que usará en el `UserDetailsService` para buscar un usuario por su nombre. Spring Data JPA generará automáticamente la implementación de esta interfaz en tiempo de ejecución. Por lo tanto, ahora está listo para escribir un `UserDetailsService` personalizado que use este repository.

`UserDetailsService` de Spring Security es una interfaz bastante sencilla:

### Crear UserDetailsService

UserDetailsService de Spring Security es una interfaz bastante sencilla:

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;  
}
```

Como puede ver, las implementaciones de esta interfaz reciben el nombre de un usuario y se espera que devuelvan un objeto UserDetails o arrojen una UsernameNotFoundException si el nombre de usuario dado no muestra ningún resultado.

Debido a que su clase User implementa UserDetails y debido a que UserRepository proporciona un método findByUsername(), son perfectamente adecuados para usar en una implementación personalizada de UserDetailsService.

```
@Service  
public class UserRepositoryUserDetailsService implements UserDetailsService {  
  
    private UserRepository userRepo;  
  
    @Autowired  
    public UserRepositoryUserDetailsService(UserRepository userRepo) {  
        this.userRepo = userRepo;  
    }  
  
    @Override  
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
        User = userRepo.findByUsername(username);  
        if (user != null) {  
            return user;  
        }  
        throw new UsernameNotFoundException("User '" + username + "' not found");  
    }  
}
```

UserRepositoryUserDetailsService se inyecta con una instancia de UserRepository a través de su constructor. Luego, en su método loadByUsername(), llama a findByUsername() en el UserRepository para buscar un usuario.

El método loadByUsername() tiene una regla simple: nunca debe devolver null. Por lo tanto, si la llamada a findByUsername() devuelve null, loadByUsername() arrojará una excepción UsernameNotFoundException. De lo contrario, se devolverá el Usuario que se encontró.

Notarás que UserRepositoryUserDetailsService está anotado con @Service. Esta es otra de las anotaciones de estereotipo de Spring que lo marcan para su inclusión en el escaneo de componentes de Spring, por lo que no hay necesidad de declarar explícitamente esta clase como un bean. Spring lo descubrirá automáticamente e instanciarlo como un bean.

Sin embargo, aún necesita configurar el UserDetailsService personalizado con Spring Security. Por lo tanto, volverá al método configure() una vez más:

```
@Autowired  
private UserDetailsService;
```

```
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
    auth.userDetailsService(userDetailsService);  
}
```

Esta vez, simplemente realiza una llamada al método `userDetailsService()`, pasando la instancia `UserDetailsService` que se ha conectado automáticamente a `SecurityConfig`.

# Resumen

1. Spring Security es un framework de autenticación y control de acceso potente y altamente personalizable. Es el estándar de facto para proteger las aplicaciones basadas en Spring.
2. Spring Security es un framework que se centra en proporcionar autenticación y autorización a las aplicaciones Java. Como todos los proyectos de Spring, el verdadero poder de Spring Security se encuentra en la facilidad con la que se puede extender para cumplir con los requisitos personalizados.
3. Spring Security ofrece protección contra ataques como session fixation, clickjacking, cross site request forgery, etc.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://docs.spring.io/spring-security/site/docs/current/reference/html5/>
- <https://www.javaguides.net/p/spring-security-tutorial.html>

## 2.5. SPRING RESTFUL

### 2.5.1. Introducción

En los temas anteriores, los objetos manejados por Spring fueron creados en una JVM y fueron accedidos y manipulados indirectamente por el usuario usando el protocolo HTTP. En este caso, todo lo que el usuario final necesita para acceder a los objetos en la JVM es la interfaz web de la aplicación. El usuario final envía comandos mediante métodos HTTP y envía datos mediante parámetros de solicitud, encabezados HTTP o el contenido del cuerpo de la solicitud. Los resultados de estos comandos incluyen recuperar objetos, crear un objeto nuevo o modificar o eliminar un objeto existente. Hay otros tipos de aplicaciones que admiten acceso remoto, que deben mencionarse primero.

Los servicios web y remotos son formas de comunicarse entre aplicaciones. Las aplicaciones pueden ejecutarse en la misma computadora, en diferentes computadoras o en diferentes redes, y pueden estar escritas en diferentes lenguajes (una aplicación Python puede consumir un servicio web proporcionado por una aplicación Java, por ejemplo). En la comunicación remota, las aplicaciones que se comunican se conocen entre sí. Hay una aplicación de servidor y una aplicación de cliente, y para garantizar la seguridad, cada una está configurada con la ubicación y otros datos para identificar a la otra. Debido a esto, admite opciones de administración de estado, puede correlacionar varias llamadas del mismo cliente y admitir devoluciones de llamada.

En la aplicación cliente, se crea un proxy del objeto de destino del servidor y se usa para acceder al objeto en el servidor. La comunicación remota se puede utilizar en cualquier protocolo, pero no funciona bien con firewalls. La comunicación remota se basa en la existencia de ensamblados de Common Language Runtime que contienen información sobre los tipos de datos. La instancia del cliente y del servidor manejan los mismos tipos de datos. Esto limita la información que debe pasarse sobre un objeto y permite que los objetos pasen por valor o por referencia. La comunicación se realiza en formato binario, XML o JSON. Estas limitaciones hicieron que la comunicación remota fuera obsoleta cuando se escribió este libro. Spring Remoting no es parte del examen de certificación y fue reemplazado por servicios web.

Los servicios web son métodos de comunicación entre procesos multiplataforma que utilizan estándares comunes y funcionan a través de cortafuegos. Trabajan con mensajes, no con objetos. Básicamente, el cliente envía un mensaje y se devuelve un mensaje de respuesta. Los servicios web funcionan en un entorno sin estado donde cada mensaje da como resultado un nuevo objeto creado para atender la solicitud. Los servicios web admiten la interoperabilidad entre plataformas y son buenos para entornos heterogéneos. Exponen sus propios conjuntos arbitrarios de operaciones, como a través de WSDL y SOAP.

REST, o REpresentational State Transfer, es un servicio web basado en HTTP para la comunicación entre aplicaciones. REST es actualmente la forma más popular en que las aplicaciones se comunican entre sí. Los servicios RESTful se pueden implementar usando Spring Boot, que es el tema central de este capítulo. ¿Por qué? Debido a que Spring Boot es muy adecuado para construir microservicios y REST facilita que los microservicios trabajen juntos, ser capaz de entender REST es práctico. Los servicios REST permiten el acceso y la manipulación de representaciones textuales de recursos web utilizando un conjunto uniforme y predefinido de operaciones sin estado.



### 2.5.2. Configuración

En el lado del servidor, el componente principal para el soporte REST es Spring MVC. Un controlador que fue diseñado para manejar solicitudes web también puede manejar solicitudes REST con algunas pequeñas modificaciones. En este proyecto tenemos dos controladores que administran instancias de Person. Uno de ellos se llama MultiplePersonController porque sus métodos de manejo completan modelos con múltiples instancias de Person y el otro se llama SinglePersonController porque llena un modelo con una sola instancia de Person. Este es el cuerpo del método inicial MultiplePersonController.list().

```
package com.apress.cems.rest.controllers;
...
@Controller
@RequestMapping("/persons")
public class MultiplePersonController {
...
@RequestMapping(path = "/list", method = RequestMethod.GET)
public String list(Model model) {
    List<Person> persons =
        personService.findAll();
    persons.sort(COMPARATOR_BY_ID);
    model.addAttribute("persons", persons);
    return "persons/list";
}
}
```

Las anotaciones compuestas se introdujeron en Spring 4.3. Reemplazan la anotación @RequestMapping con un determinado método HTTP. Están agrupados en el paquete org.springframework.web.bind.annotation. Las anotaciones de Spring y sus equivalentes se enumeran a continuación:

HTTP Method	@RequestMapping	Spring 4.3 Annotation
GET	@RequestMapping(method=RequestMethod.GET)	@GetMapping
POST	@RequestMapping(method=RequestMethod.POST)	@PostMapping
DELETE	@RequestMapping(method=RequestMethod.DELETE)	@DeleteMapping
PUT	@RequestMapping(method=RequestMethod.PUT)	@PutMapping

Figura 39: Anotaciones de métodos HTTP de Spring  
Fuente.- Tomado de Cosmina, L. (2020)

El método anterior también se puede escribir así:

```
package com.apress.cems.rest.controllers;
import org.springframework.web.bind.annotation.GetMapping;
...
@Controller
@RequestMapping("/persons")
public class MultiplePersonController {
...
}
```

```
@GetMapping (path = "/list")
public String list(Model model) {
    List<Person> persons =
        personService.findAll();
    persons.sort(COMPARATOR_BY_ID);
    model.addAttribute("persons", persons);
    return "persons/list";
}
```

# Resumen

1. Los servicios web y remotos son formas de comunicarse entre aplicaciones. Las aplicaciones pueden ejecutarse en la misma computadora, en diferentes computadoras o en diferentes redes, y pueden estar escritas en diferentes lenguajes.
2. El componente principal para el soporte REST es Spring MVC.
3. REST, o REpresentational State Transfer, es un servicio web basado en HTTP para la comunicación entre aplicaciones.
4. Los servicios REST permiten el acceso y la manipulación de representaciones textuales de recursos web utilizando un conjunto uniforme y predefinido de operaciones sin estado.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://spring.io/guides/tutorials/rest/>
- [https://www.tutorialspoint.com/spring\\_boot/spring\\_boot\\_building\\_restful\\_web\\_services.htm](https://www.tutorialspoint.com/spring_boot/spring_boot_building_restful_web_services.htm)

## UNIDAD

## 3

# ANGULAR

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad el alumno implementa una aplicación web utilizando Angular, integrada con Spring Framework.

**TEMARIO****3.1 Tema 8 : Angular**

- 3.1.1 : Introducción
- 3.1.2 : Aplicaciones SPA
- 3.1.3 : Características

**3.2 Tema 9 : TypeScript**

- 3.2.1 : Introducción
- 3.2.2 : Tipos de datos
- 3.2.3 : Clases e Interfaces
- 3.2.4 : Clases e Interfaces
- 3.2.5 : Estructuras de control
- 3.2.6 : Operadores
- 3.2.7 : Promesas

**3.4 Tema 10 : Componentes angular**

- 3.3.1 : Introducción
- 3.3.2 : Configuración
- 3.3.3 : Ensamblaje

**3.4 Tema 11 : Servicios angular**

- 3.4.1 : Introducción
- 3.4.2 : Configuración
- 3.4.3 : Uso de Http

**ACTIVIDADES PROPUESTAS**

- Actividad 8: Creación de Web Components.
- Actividad 9: Uso de Typescript.
- Actividad 10: Crear componentes en Angular.
- Actividad 11: Crear servicios en Angular.

## 3.1. ANGULAR

### 3.1.1. Introducción

Angular es una plataforma que permite desarrollar aplicaciones web en la sección cliente utilizando HTML y JavaScript para que el cliente asuma la mayor parte de la lógica y descargue al servidor con la finalidad de que las aplicaciones ejecutadas a través de Internet sean más rápidas. El hecho de estar mantenido por Google, así como una serie de innumerables razones técnicas, ha favorecido su rápida adopción por parte de la comunidad de desarrolladores.

Permite la creación de aplicaciones web de una sola página (SPA: single-page application) realizando la carga de datos de forma asíncrona.

Además de mejorar el rendimiento de las aplicaciones web, su utilización en dispositivos móviles está optimizada ya que, en ellos, los ciclos de CPU y memoria son críticos para su óptimo funcionamiento. Ello permite el desarrollo de aplicaciones móviles híbridas con Ionic 2.

Gracias al uso de componentes, se puede encapsular mejor la funcionalidad facilitando el mantenimiento de las aplicaciones.

Parecería ser la continuación de AngularJS, pero, en realidad, más que una nueva versión es realmente un framework o plataforma diferente. El hecho de utilizar componentes como concepto único en lugar de controladores, directivas y servicios de forma específica, como sucedía en AngularJS, simplifica mucho las cosas.

Angular está orientado a objetos, trabaja con clases y favorece el uso del patrón MVC (Modelo - Vista - Controlador).

Permite el uso de TypeScript (lenguaje desarrollado por Microsoft) con las ventajas que supone poder disponer de un tipado estático y objetos basados en clases. Todo ello, gracias a la especificación ECMAScript 6, que es la base sobre la que se apoya TypeScript. Gracias a un compilador (transpilador) de TypeScript, el código escrito en este lenguaje se traducirá a JavaScript original.

### 3.1.2. Aplicaciones SPA

Angular es un framework principalmente enfocado a la creación de aplicaciones web de tipo single-page application (SPA). En este capítulo haremos un breve repaso a los distintos tipos de aplicación web y, a continuación, analizaremos con más detalle en qué consisten las aplicaciones single-page application (SPA).

Una aplicación web la podemos definir como toda aquella aplicación proporcionada por un servidor web y utilizada por los usuarios a través de un cliente web (browsers o navegadores). En otras palabras, son aquellas aplicaciones codificadas en un lenguaje soportado por los navegadores web para que puedan ejecutarse desde allí.

Toda aplicación web tiene una arquitectura básica de tipo cliente-servidor. Esto es así porque toda aplicación web sigue un modelo de aplicación distribuida en la que, por una parte, está el servidor que provee de recursos y servicios y, por la otra, está el cliente que los demanda. A partir de aquí, según lo estática o dinámica que sea cada parte, tenemos distintos tipos de aplicaciones web:

- Cliente y servidor estáticos: no hay ningún tipo de dinamismo y el servidor siempre devuelve los mismos recursos sin ningún tipo de cambio. Más que de aplicaciones web, aquí hablaríamos de páginas web.
- Cliente estático y servidor dinámico: el servidor devuelve recursos dinámicos, por ejemplo, páginas web con el resultado de consultas a base de datos, etc.
- Cliente/servidor dinámicos: el cliente es dinámico porque las páginas web recibidas del servidor incluyen JavaScript, que se ejecuta en el propio navegador dando todo tipo de funcionalidades diversas.

### 3.1.3. Características

AngularJS fue desarrollado en 2009 por Miško Hevery y Adams Abrons. Originalmente era el software detrás de un servicio de almacenamiento online de archivos JSON, pero, poco tiempo después, se abandonó el proyecto y se liberó AngularJS como una biblioteca de código abierto. Adams Abrons dejó el proyecto, pero Miško Hevery, como empleado de Google, continuó el desarrollo y mantenimiento del framework.

Desde entonces se han ido lanzando muchas versiones de AngularJS, pero, sobre todas ellas, cabe destacar la salida de la versión 2.0 por los siguientes motivos:

- Se rediseñó por completo todo el framework (nueva arquitectura basada en componentes, etc.).
- Se introdujo el uso de TypeScript de Microsoft (superconjunto de JavaScript) como lenguaje de programación.
- Se cambió el nombre del framework. Pasó de llamarse AngularJS a Angular.
- Los desarrolladores anunciaron que darían soporte y mantenimiento tanto para AngularJS (versiones 1.X.Y) como para Angular (versiones superiores), pero que una y otra seguirían ciclos de vida independientes. Actualmente, AngularJS se encuentra en la versión 1.6.4, mientras que Angular se halla en la versión 4.0.

# Resumen

1. Angular está orientado a objetos, trabaja con clases y favorece el uso del patrón MVC (Modelo - Vista - Controlador).
2. Angular es un framework principalmente enfocado a la creación de aplicaciones web de tipo single-page application (SPA).
3. Toda aplicación web tiene una arquitectura básica de tipo cliente-servidor. Esto es así porque toda aplicación web sigue un modelo de aplicación distribuida en la que, por una parte, está el servidor que provee de recursos y servicios y, por la otra, está el cliente que los demanda.
4. Angular usa TypeScript de Microsoft (superconjunto de JavaScript) como lenguaje de programación.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://angular.io/guide/architecture>
- <https://adamofig.medium.com/angular-fundamentos-te%C3%B3ricos-parte-1-b9b785f18a6a>

## 3.2. TYPESCRIPT

### 3.2.1. Introducción

TypeScript es un lenguaje de programación orientado a objetos fuertemente tipado que se traduce a JavaScript añadiéndole características que no posee. La operación de traducir TypeScript a JavaScript se conoce como transpilación. Gracias al uso de TypeScript, es posible localizar errores de sintaxis antes incluso de su ejecución. De ahí que haya ganado mucha aceptación entre los desarrolladores del mundo web.

### 3.2.2. Tipos de datos

#### Tipo de dato Null

Al igual que en JavaScript, el tipo de datos null en TypeScript solo puede tener un valor válido: null. Una variable null no puede contener otros tipos de datos como número y cadena de texto. Establecer una variable a null borrará su contenido si tuviese alguno.

Recuerde que cuando el indicador strictNullChecks se configura como true en tsconfig.json, solo el valor null se puede asignar a las variables con tipo null. Este indicador está desactivado por defecto, lo que significa que también puede asignar el valor null a variables con otros tipos como number o void.

```
// With strictNullChecks set to true
let a: null = null; // Ok
let b: undefined = null; // Error
let c: number = null; // Error
let d: void = null; // Error
// With strictNullChecks set to false
let a: null = null; // Ok
let b: undefined = null; // Ok
let c: number = null; // Ok
let d: void = null; // Ok
```

#### Tipo de dato Undefined

Cualquier variable cuyo valor no haya especificado se establece en undefined. Sin embargo, usted también puede establecer explícitamente el tipo de una variable como indefinida, como se ve en el siguiente ejemplo.

Tenga en cuenta que una variable con type configurado como undefined solo puede tener undefined como su valor. Si la opción strictNullChecks está configurada como false, también podrá asignar undefined a variables de tipo numérico y cadenas de texto, etc.

```
// With strictNullChecks set to true
let a: undefined = undefined; // Ok
let b: undefined = null; // Error
let c: number = undefined; // Error
let d: void = undefined; // Ok
```



```
// With strictNullChecks set to false
let a: undefined = undefined; // Ok
let b: undefined = null; // Ok
let c: number = undefined; // Ok
let d: void = undefined; // Ok
```

### Tipo de dato Void

El tipo de datos void se usa para indicar la falta de un type para una variable. Establecer variables para que tengan un tipo void puede no ser muy útil, pero usted puede establecer el tipo de retorno de las funciones que no devuelven nada a void . Cuando se usa con variables, el tipo void solo puede tener dos valores válidos: null y undefined.

```
// With strictNullChecks set to true
let a: void = undefined; // Ok
let b: void = null; // Error
let c: void = 3; // Error
let d: void = "apple"; // Error
// With strictNullChecks set to false
let a: void = undefined; // Ok
let b: void = null; // Ok
let c: void = 3; // Error
let d: void = "apple"; // Error
```

### Tipo de dato Boolean

A diferencia de los tipos de datos number y string , boolean solo tiene dos valores válidos.

Solo puedes establecer su valor en true o false . Estos valores se usan mucho en las estructuras de control donde una pieza de código se ejecuta si una condición es true y otra parte de código se ejecuta si una condición es false.

Aquí hay un ejemplo muy básico para declarar variables booleanas:

```
let a: boolean = true;
let b: boolean = false;
let :2 boolean = 23; // Error
let d: boolean = "blue"; // Error
```

## 3.2.3. Clases e Interfaces

### Clases

Las clases son una parte fundamental de la programación orientada a objetos. Usas clases para representar cualquier entidad que tenga algunas propiedades y funciones que puedan actuar sobre propiedades determinadas. TypeScript le brinda control total sobre las propiedades y funciones que son accesibles dentro y fuera de su propia clase contenedora. Aquí hay un ejemplo muy básico de crear una clase de Person.

```
class Person {  
    name: string;  
    constructor(theName: string) {  
        this.name = theName;  
    }  
    introduceSelf() {  
        console.log("Hi, I am " + this.name + "!");  
    }  
}  
  
let personA = new Person("Sally");  
personA.introduceSelf();
```

El código anterior crea una clase muy simple llamada Person. Esta clase tiene una propiedad llamada name y una función llamada introduceSelf . La clase también tiene un constructor, que también es básicamente una función. Sin embargo, los constructores son especiales porque se llaman cada vez que creamos una nueva instancia de nuestra clase. También puede pasar parámetros a los constructores para inicializar diferentes propiedades. En nuestro caso, estamos usando el constructor para inicializar el nombre de la persona que estamos creando usando la clase Person . La función introduceSelf es un método de la clase Person, y la estamos usando aquí para imprimir el nombre de la persona en la consola. Todas estas propiedades, métodos y el constructor de una clase se denominan colectivamente miembros de la clase.

Debe tener en cuenta que la clase Person no crea automáticamente una persona por sí misma. Actúa más como un plano con toda la información sobre los atributos que una persona debería haber creado una vez. Con eso en mente, creamos una nueva persona y la llamamos Sally. Al llamar al método introduceSelf en esta persona, se imprimirá la línea "Hola, soy Sally". a la consola.

## Interfaces

Digamos que tienes un objeto de lago en el código y usarlo para almacenar información sobre algunos de los lagos más grande por área en el mundo. Este objeto de lago tiene propiedades como el nombre del lago, su área, longitud, profundidad y los países en que existe ese lago.

Los nombres de los lagos se guardará como una cadena. Las longitudes de estos lagos serán en kilómetros y serán las áreas en kilómetros cuadrados, pero ambas de estas propiedades serán almacenadas como números. Las profundidades de los lagos será en metros, y esto también podría ser un flotador.

Puesto que todos estos lagos son muy grandes, sus costas generalmente no se limitan a un solo país. Se utilizará una matriz de cadenas para almacenar los nombres de todos los países en la orilla de un lago particular. Un valor Boolean puede utilizarse para especificar si el lake es agua salada o agua dulce. El siguiente fragmento de código crea una interfaz para el objeto de nuestro lago.

```
interface Lakes {  
    name: string,  
    area: number,
```

```

    length: number,
    depth: number,
    isFreshwater: boolean,
    countries: string[]
  }

```

### 3.2.4. Uso de decoradores

Para poder utilizar los decoradores tenemos que permitirlo en nuestro tsconfig y escribir lo siguiente:

```
"experimentalDecorators": true
```

La sintaxis de los decoradores es:

```
@miDecorador
```

```

function log(constructor: Function): void {
    console.log('Registered Class: ' + constructor['name'] + ' at ' + Date.now());
}

function logm < T > (target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor < T >): void {
    console.log('Registered Method: ' + propertyKey.toString() + ' at ' + Date.now());
}

function logparam(target: Object, propertyKey: string | symbol, parameterIndex: number): void {
    console.log('Registered Parameter: ' + propertyKey.toString() + '-' + parameterIndex + ' at ' + Date.now());
}

// Así se utiliza un decorador
@log
class MyClass {
    public name: string = 'name';
    constructor() {
        console.log('constructor');
    }
    @logm
    public myMethod() {
        console.log('method')
    }
    @logm
    public myMethod2(param1: number, @logparam param2: boolean) {
        console.log('method2')
    }
}

var myClass = new MyClass();
myClass.myMethod();
myClass.myMethod2(1, false);
// ----- COMO SE USAN LOS DECORADORES
// type ClassDecorator = <TFunction extends Function>(target: TFunction): TFunction | void;
// type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor

```

```
: TypedPropertyDescriptor < T > ): TypedPropertyDescriptor < T > | void;  
// type PropertyDecorator = (target: Object, propertyKey: string | symbol): void;  
// type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex: number): void;
```

### 3.2.5. Estructuras de control

#### Condicionales

Los condicionales son expresiones que nos permiten ejecutar una secuencia de instrucciones u otra diferente dependiendo de lo que estemos comprobando. Permiten establecer el flujo de ejecución de los programas de acuerdo a determinados estados.

#### Asignación condicional

Este tipo de asignación es también conocido como el If simplificado u operador ternario. Sirve para asignar en una sola línea un valor determinado si la condición que se evalúa es true u otro si es false. La sintaxis es la siguiente:

condición ? valor\_si\_true : valor\_si\_false

Si la condición devuelve true, retornará el valor de valor\_si\_true, y si es false el valor devuelto será el de valor\_si\_false. Veamos unos ejemplos:

```
(true) 5 : 2; // Devuelve 5  
(false) 5 : 2; // Devuelve 2
```

#### Sentencia IF

Como hemos visto antes, dependiendo del resultado de una condición, obtenemos un valor u otro. Si el resultado de la condición requiere más pasos, en lugar de utilizar la asignación condicional, es mejor emplear la sentencia if. Tenemos 3 formas de aplicarlo:

if simple

```
if (condicion) {  
    bloque_de_codigo  
}
```

if/else

```
if (condicion) {  
    bloque_de_codigo_1  
} else {  
    bloque_de_codigo_2  
}
```

if/else if

```
if (condicion_1) {
```

```
        bloque_1
    } else if (condicion_2) {
        bloque_2
    } else if (condicion_3) {
        bloque_3
    } else {
        bloque_4
    }
}
```

### Sentencia Switch

Con Switch, podemos sustituir un conjunto de sentencias if-else de una manera más legible. Se comprueba la condición, y según el caso que devuelva, ejecutará un bloque u otro. Para poder separar los bloques, se utiliza la palabra break que permite salir de toda la sentencia. Tiene un bloque default que se ejecuta en el caso de que no se cumpla ningún caso. Veamos un ejemplo, esto sería switch siguiendo el ejemplo anterior del if-else:

```
switch (condicion) {
    case condicion_1:
        bloque_1
        break;
    case condicion_2:
        bloque_2
        break;
    case condicion_3:
        bloque_3
        break;
    default:
        bloque_4
}
```

El bloque default no es obligatorio.

### 3.2.6. Operadores

Estos operadores permiten formar expresiones. Las más comunes son las operaciones aritméticas.

- Suma de números:  $5 + 2$
- Resta:  $5 - 2$
- Operaciones con paréntesis:  $(3 + 2) - 5$
- Divisiones:  $3 / 3$
- Multiplicaciones:  $6 * 3$

El operador suma + también puede usarse para concatenar strings de la siguiente manera: "Hola " + "mundo" + "!" tendrá como resultado "Hola mundo!".

También posee los operadores post y pre incremento y decremento que añaden uno o restan uno a la variable numérica en la que se aplican. Dependiendo si son pre o post, la variable es autoincrementada o decrementada antes o después de la sentencia. Veamos un ejemplo:

```
var x = 1; // x=1
var y = ++x; // x=2, y=2
```

```
var z = y++ + x; // x=2, y=3, z=4
```

### 3.2.7. Promesas

Una Promesa, es un objeto que sirve para reservar el resultado de una operación futura. Este resultado llega a través de una operación asíncrona como puede ser una petición HTTP o una lectura de ficheros, que son operaciones no instantáneas, que requieren un tiempo, aunque sea pequeño, para ejecutarse y finalizar.

Para entender mejor el concepto de Promesas, veremos primero como funciona la asincronía con la típica función de callback.

```
function loadCSS(url, callback) {  
    var elem = window.document.createElement('link');  
    elem.rel = "stylesheet";  
    elem.href = url;  
    window.document.head.appendChild(elem);  
    callback();  
}  
loadCSS('styles.css', function() {  
    console.log("Estilos cargados");  
});
```

En este ejemplo tenemos una función llamada loadCSS a la que pasamos una url, presumiblemente que apunte a un fichero .css, y una función de callback como parámetros, la función básicamente crea un elemento link y lo añade al final de la <head>.

Cuando ejecutamos esta función, le pasamos la url de styles.css y una función anónima que será el callback. Lo que hará será imprimir por la consola Estilos cargados cuando finalice la carga.

Este es un ejemplo básico de una función asíncrona con callbacks. Si queremos reproducir este mismo comportamiento usando Promesas, sería de la siguiente manera:

```
// Asumamos que loadCSS devuelve una promesa  
var promise = loadCSS('styles.css');  
promise.then(function() {  
    console.log("Estilos cargados");  
});  
promise.catch(function(err) {  
    console.log("Ocurrió un error: " + err);  
});
```

# Resumen

1. TypeScript es un lenguaje de programación orientado a objetos fuertemente tipado que se traduce a JavaScript añadiéndole características que no posee.
2. El tipo de datos null en TypeScript solo puede tener un valor válido: null.
3. El tipo de datos void se usa para indicar la falta de un type para una variable.
4. Los condicionales son expresiones que nos permiten ejecutar una secuencia de instrucciones u otra diferente dependiendo de lo que estemos comprobando.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>
- <https://www.campusmvp.es/recursos/post/typescript-contra-javascript-cual-deberias-utilizar.aspx>

## 3.3. COMPONENTES ANGULAR

### 3.3.1. Introducción

El componente es el elemento básico del desarrollo en Angular. Una aplicación suele constar de una especie de árbol de componentes de varios niveles donde un componente puede llamar a sus componentes hijos y, a su vez, estos llamar a sus propios hijos (o nietos del primero), y así sucesivamente. Un componente, básicamente, es una clase acompañada del decorador `@Component` y se encarga de controlar lo que podríamos llamar Vista o zona de pantalla. A grosso modo, un componente está formado por un template, una metadata y una clase.

### 3.3.2. Configuración

Al crear una aplicación con Angular CLI, se crea un módulo por defecto y, dentro del mismo, se crea también un componente denominado `app.component`. El contenido por defecto del mismo es el siguiente:

```
import {  
    Component  
} from '@angular/core';  
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
    title = 'app works!';  
}
```

### 3.3.3. Ensamblaje

La composición de un componente es la siguiente:

- **Import:** permite importar otros componentes a utilizarse dentro del componente que se está definiendo. Como mínimo, importamos `Component` de `@angular/core`.
- **Decorador:** función de tipo decorador que posee un objeto con los metadatos que indican a Angular cómo se crea, compila y ejecuta el componente.
- **Export class:** exporta la clase asociada al componente para hacerla visible a otros componentes, de forma que puedan usar sus propiedades y métodos dependiendo a su vez de la visibilidad establecida a los mismos. Hace las veces de controlador en la clásica arquitectura MVC (Modelo-Vista- Controlador).



# Resumen

1. El componente es el elemento básico del desarrollo en Angular.
2. Un componente está formado por un template, una metadata y una clase.
3. Al crear una aplicación con Angular CLI , se crea un módulo por defecto y, dentro del mismo, se crea también un componente denominado app.component.
4. Decorador: función de tipo decorador que posee un objeto con los metadatos que indican a Angular cómo se crea, compila y ejecuta el componente.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://angular.io/guide/architecture-components>
- <https://www.acontracorrientech.com/entendiendo-los-componentes-en-angular/>

## 3.4. SERVICIOS ANGULAR

### 3.4.1. Introducción

Un servicio Angular es una clase que encapsula algún tipo de funcionalidad común entre los diferentes componentes de la aplicación, como podría ser, por ejemplo, el acceso a datos. La extracción de funcionalidades comunes de los componentes para crear servicios supone toda una serie de ventajas:

- Evitamos la repetición innecesaria de código en los componentes, por lo que su código acaba siendo más ligero y centrado en el soporte a la vista.
- Podemos crear servicios mock (de prueba) que faciliten el análisis de los componentes que los usan.
- Mejoramos el control y mantenimiento de esas funcionalidades comunes.

Vamos a ver cómo sería el formato estándar de cualquier servicio a través de un ejemplo:

```
import {  
    Injectable  
} from '@angular / core';  
@Injectable()  
export class LoggerService {  
    log(message: string) {  
        console.log(message);  
    }  
    constructor() {}  
}
```

Como vemos no es más que una clase estándar donde se le ha añadido el decorador `@Injectable()`.

### 3.4.2. Configuración

Los servicios son suministrados a los componentes mediante el patrón de diseño inyección de dependencias. Angular incorpora un framework al respecto. Básicamente, este framework nos facilita la gestión de dependencias mediante inyectores (injectors) y proveedores (providers). Estos elementos los podríamos definir de la siguiente manera:

- Un inyector mantiene una colección de servicios previamente instanciados. Si se requiere un servicio aun no instanciado, el inyector crea una instancia mediante un proveedor y lo agrega a la colección.
- Un proveedor es algo que puede crear o devolver una instancia de servicio. Normalmente es la clase del servicio en sí. Para que el inyector pueda hacer uso de los proveedores, estos deben estar registrados. Estos registros pueden hacerse a nivel de módulo o componente, dependiendo de la disponibilidad que se le quiera dar al servicio. En el diagrama general de la arquitectura Angular puede observar dónde se situaría el inyector y su colección de servicios previamente instanciados.

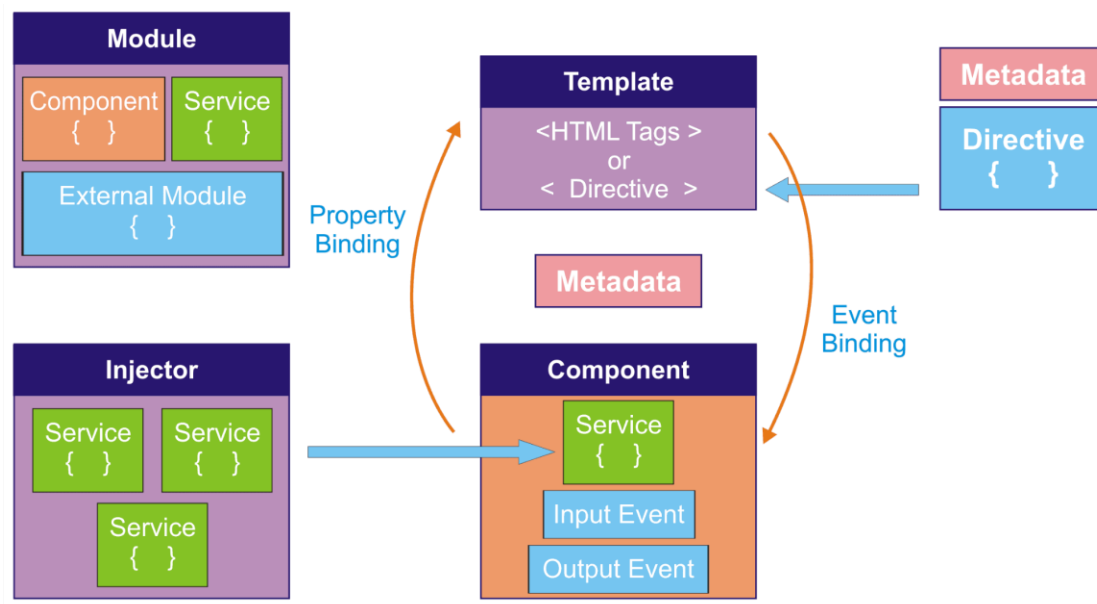


Figura 40: Angular Architecture

Fuente.- Tomado de <https://www.ngdevelop.tech/angular/architecture/>

### 3.4.3. Uso de Http

Una aplicación web puede dividirse en dos grandes bloques: Front-end y Back-end. El Front-end es la parte que se muestra a través del navegador web, o sea, la parte que se encarga de interactuar con el usuario para recoger datos e información. El Back-end, por su parte, se ejecuta en un servidor y recoge a través de una API de servicios toda esa información para gestionarla (guardarla en una base de datos, realizar comprobaciones, etc.).

En la mayoría de casos, la comunicación entre aplicaciones Front-end y servicios Back-end se realiza mediante el protocolo HTTP. Para realizar este tipo de comunicación, las aplicaciones se apoyan en API como la interfaz XMLHttpRequest y la API fetch(), que son soportadas por la mayoría de navegadores web y permiten al Front-end realizar peticiones HTTP.

Angular, por su parte, nos ofrece el servicio HttpClient. HttpClient es un servicio que proporciona una API simplificada, construida sobre la interfaz XMLHttpRequest, para realizar peticiones HTTP y procesar sus respuestas. En este sentido, el servicio ofrece toda una serie de métodos para realizar los distintos tipos de peticiones HTTP que existen:

Method	Meaning
GET	Read data
POST	Insert data
PUT or PATCH	Update data, or insert if a new id
DELETE	Delete data

Figura 41: Métodos Http

Fuente.- Tomado de <https://www.devopsschool.com/blog/understanding-rest-http-method-get-post-put-head-delete/>

El servicio también aporta otros beneficios muy interesantes como son el soporte a la realización de test, soporte a la intercepción de peticiones y respuesta, una mejor gestión de errores, etc. HttpClient forma parte del módulo HttpClientModule del paquete `@angular/common/http` que tendremos que incluir en nuestra aplicación Angular. Hay que tener en cuenta que HttpClient está disponible en Angular 4.3.X y versiones posteriores. Para versiones anteriores existía un servicio parecido que se llamaba simplemente HTTP.

# Resumen

1. Un servicio Angular es una clase que encapsula algún tipo de funcionalidad común entre los diferentes componentes de la aplicación.
2. Los servicios son suministrados a los componentes mediante el patrón de diseño inyección de dependencias.
3. Angular, por su parte, nos ofrece el servicio HttpClient. HttpClient es un servicio que proporciona una API simplificada, construida sobre la interfaz XMLHttpRequest, para realizar peticiones HTTP y procesar sus respuestas.

# Recursos

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- <https://angular.io/guide/architecture-services>
- <https://codingpotions.com/angular-servicios-llamadas-http>

# Bibliografía

- Adam L. (2020) *Spring Quick Reference Guide*. New York: Apress.
- Boada, M. & Gómez J. (2019) *El gran libro de angular*. Madrid: Alfaomega.
- Cosmina L. (2020) *Pivotal Certified Professional Core Spring 5 Developer Exam: A Study Guide Using Spring Framework 5*. New York: Apress.
- Janssen, Thorben (2021) *Tutorials. Persistence with JPA and Hibernate*. Recuperado de <https://thorben-janssen.com/tutorials/>
- Javarevisited (2021) *5 Best Kotlin Online Courses for Java and Android Developers [UPDATED]*. Recuperado de <https://javarevisited.blogspot.com/#axzz6pww30VZx>
- Keith, Mike (2013) *Pro JPA 2 : mastering the Java Persistence API*. 2nd ed. New York: Apress.  
Centro de Información: Código 005.133J KEIT 2013
- Leonard, Anghel (2020) *Spring Boot Persistence Best Practices*. New York: Apress.
- Mihalcea, Vlad (2020) *High-Performance Java Persistence*. Cluj-Napoca: Vlad Mihalcea.
- Sznajdleder, Pablo Augusto (2016) *Java a fondo: curso de programación*. 3a ed. ampliada. Buenos Aires: Alfaomega.  
Centro de Información: Código 005.133J KEIT 2013
- VMware, Inc. (2021) *Spring. Guides*. Recuperado de <https://spring.io/guides>
- Walls, Craig (2015) *Spring*. 4a ed. Madrid: Anaya Multimedia.  
Centro de Información: Código 005.133J WALL/ES
- Walls, Craig (2019) *Spring in Action*. Greenwich, CT: Manning Publications Co.