


Acceso a Base de Datos con Java



© JMA 2016. All rights reserved

INTRODUCCIÓN

© JMA 2016. All rights reserved

Introducción

- Cuando nos ponemos a desarrollar, en el código final lo que tenemos que hacer a veces es mucho.
- Deberemos desarrollar código para:
 - Lógica de la aplicación
 - Acceso a Base de Datos
 - EJB para modularizar las aplicaciones.
- Deberemos también programar en diferentes Lenguajes:
 - Java (mayoritariamente)
 - SQL
 - HTML, etc
- "La vida es corta, dedique menos tiempo a escribir código para la unión BBDD-Aplicación JAVA y más tiempo añadiendo nuevas características"

© JMA 2016. All rights reserved

Introducción a Hibernate

- Las bases de datos relacionales son indiscutiblemente el centro de la empresa moderna.
- Los datos de la empresa se basan en entidades que están almacenadas en ubicaciones de naturaleza relacional. (Base de Datos)
- Los actuales lenguajes de programación, como Java, ofrecen una visión intuitiva, orientada a objetos de las entidades de negocios a nivel de aplicación.
- Se han realizado mucho intentos para poder combinar ambas tecnologías (relacionales y orientados a objetos), o para reemplazar uno con el otro, pero la diferencia entre ambos es muy grande.

© JMA 2016. All rights reserved

Discrepancia del Paradigma

- Problemas de identidad
 - Objetos Java definen dos nociones diferentes de identidad:
 - Identidad de objeto o referencia (equivalente a la posición de memoria, comprobar con un `==`).
 - La igualdad como determinado por la aplicación de los métodos `equals()`.
 - La identidad de una fila de base de datos se expresa como la clave primaria.
 - Ni `equals()` ni `==` es equivalente a la clave principal.
- Problemas de Asociaciones
 - El lenguaje Java representa a las asociaciones mediante utilizar referencias a objetos
 - Las asociaciones entre objetos son punteros unidireccionales.
 - Relación de pertenencia → Elementos contenidos
 - Modelo de composición (colecciones) → Modelo jerárquico
 - Las asociaciones en BBDD están representados mediante la migración de claves.
 - Todas las asociaciones en una base de datos relacional son bidireccional

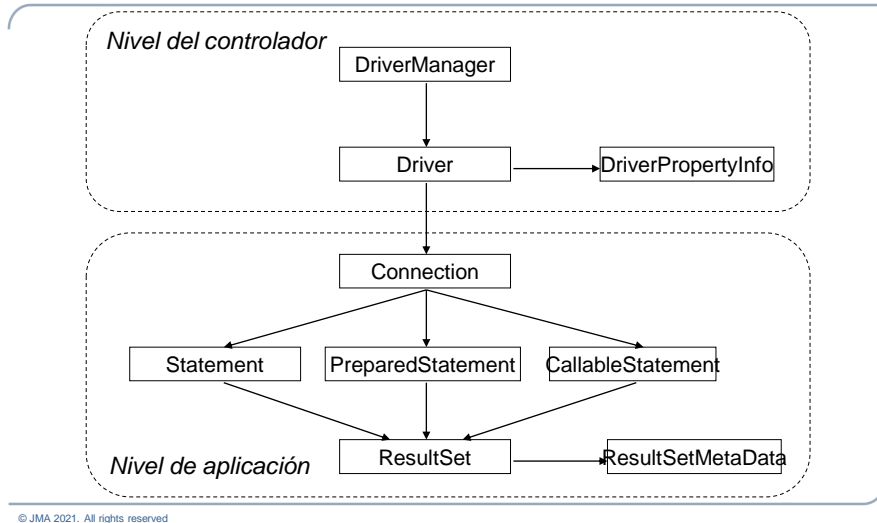
© JMA 2016. All rights reserved

Paquete `java.sql`

JDBC (JAVA DATABASE CONNECTIVITY)

© JMA 2021. All rights reserved

Arquitectura de JDBC



© JMA 2021. All rights reserved

Esquema de JDBC

El trabajo con datos de una base de datos sigue siempre los mismos pasos.

Pasos a seguir:

- Cargar el controlador.
- Establecer una conexión.
- Operaciones con datos:
 - Preparar la sentencia.
 - Opcionalmente: pasar parámetros.
 - Ejecutar la sentencia.
 - Si recupera datos:
 - Recorrer el conjunto de resultados.
 - Realizar su tratamiento.
 - Opcionalmente: confirmar o deshacer los cambios.
- Opcionalmente: confirmar o deshacer los cambios.
- Cerrar la conexión.

© JMA 2021. All rights reserved

Carga del controlador

La primera operación que debemos realizar para disponer de acceso a la base de datos es cargar la clase del controlador con el método:

```
Class.forName("<NombreClase>");
```

La clase cargada se encarga de registrarse de forma automática en el DriverManager utilizando el método:

```
DriverManager.registerDriver(this);
```

La clase puente JDBC-ODBC es:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

© JMA 2021. All rights reserved

Tipos de controladores

Hay cuatro tipos de controladores definidos por JDBC:

- Clase 1: Puente JDBC-ODBC
 - La clase del controlador utiliza las funciones nativas estandarizadas de ODBC. El cliente debe disponer de ODBC y conexión a la base de datos.
- Clase 2: JDBC a API nativa
 - La clase del controlador utiliza las funciones nativas suministradas por el fabricante del SGDB.
- Clase 3: Protocolo de red, todo Java
 - La clase del controlador se implementa en Java y se comunica en red con un servidor de aplicaciones mediante un protocolo de red estándar, que a su vez es el encargado de comunicarse con el servidor de base de datos. Es el más flexible.
- Clase 4: Protocolo nativo, todo Java
 - La clase del controlador se implementa en Java y se comunica directamente con el SGDB a través de un protocolo nativo de servidor.

© JMA 2021. All rights reserved

Tipos de Drivers / Controladores

Driver JDBC-ODBC:

- `sun.jdbc.odbc.JdbcOdbcDriver` (incluido en la API de JDBC de la plataforma J2SE)

Driver MySQL:

- `com.mysql.jdbc.Driver` (no incluido)

Driver Oracle:

- `oracle.jdbc.driver.OracleDriver` (no incluido)

Driver PostgreSQL

- `org.postgresql.Driver`

Driver DB2:

- `com.ibm.db2.jdbc.app.DB2Driver` (no incluido)

© JMA 2021. All rights reserved

Conexión

Para realizar una conexión invocamos el método:

```
Connection con =
DriverManager.getConnection("<CadenaConexión>", "<Usuario>",
"<Clave>");
```

Donde:

- CadenaConexión: Es la cadena de conexión a la base de datos siguiendo el formato establecido por el fabricante. Por regla general siguen el siguiente formato:
 - `jdbc:<TipoDeControlador>:<BaseDeDatos>`
- Usuario: Cadena con el nombre del usuario registrado en el gestor de la base de datos.
- Clave: Cadena con la clave que autentica al usuario.

© JMA 2021. All rights reserved

Conexión

Para realizar la Conexión necesitaremos:

- URL de la base de datos o dirección IP
- Nombre de Usuario
- Contraseña de autenticación en la Base de Datos
- Puerto (oracle)
- Nombre de la Base de Datos (oracle)

ORACLE

```
String cadena="jdbc:oracle:thin:System/oracle@Localhost:1521:xe"
Connection conexion= DriverManager.getConnection(cadena);
```

ODBC

```
cadena = "jdbc:odbc:Tutorial";
Connection conexion= DriverManager.getConnection(cadena, " ", " ");
```

MYSQL

```
cadena = "jdbc:mysql://localhost:3306/sakila";
Connection con=DriverManager.getConnection(cadena, "root", "mimono");
```

© JMA 2021. All rights reserved

Conexión

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public final class JDBCbridged {
```

```
    static {
        try {
            Class.forName("oracle.jdbc.OracleDriver");
        } catch (ClassNotFoundException e) {
            System.err.println("No encuentro el driver");
        }
    }
}
```

```
    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:xe", "usuario", "contraseña");
    }
}
```

© JMA 2021. All rights reserved

Conexión: Métodos

<code>close();</code>	Se encarga de cerrar la conexión con la Base de Datos.
<code>setAutocommit(false/true);</code>	Activa o desactiva el método autocommit. Después de cada transacción se realiza un commit.
<code>commit();</code>	Todos los cambios realizados desde el último commit se vuelven permanentes.
<code>rollback();</code>	Desecha todos los cambios realizados desde el último commit o roolback
<code>createStatement()</code>	Crea un objeto Statement que ejecutará sentencias SQL.
<code>prepareStatement(String)</code>	Crea un objeto PreparedStatement, para realizar llamadas a sentencias SQL que contengan parámetros de entrada.
<code>prepareCall(String)</code>	Crea un objeto CallableStatement, para ejecutar llamada a Stored Procedures.

© JMA 2021. All rights reserved

Ejecución de sentencias

Una vez que se ha establecido la conexión a la base de datos, podemos enviar sentencias SQL al Gestor.

Las sentencias SQL se pueden enviar al gestor de tres modos diferentes:

- Statement: creación de consultas de estructura estática.
- PreparedStatement: creación de consultas de estructura dinámica.
- CallableStatement: ejecución de procedimientos almacenados en el gestor.

Pueden ser ejecutadas múltiples tipos de instrucciones SQL:

- DQL (Select)
- DML (Insert, Update, Delete)
- DDL (Create, Drop)

© JMA 2021. All rights reserved

Statement

Se utilizan instancias de Statement para utilizar sentencias SQL. Es necesario crear una instancia:

```
Statement stmt = con.createStatement();
```

Dentro de la creación del método createStatement, se puede indicar las características del objeto que devolverá Statement en su ejecución (ResultSet):

- ResultSet.TYPE_FORWARD_ONLY: Solo recorrido adelante
- ResultSet.TYPE_SCROLL_INSENSITIVE: Recorrido Delante-Atrás sin tener en cuenta los cambios producidos
- ResultSet.CONCUR_READ_ONLY: Solo de lectura
- ResultSet.CONCUR_UPDATABLE: Lectura / Escritura

© JMA 2021. All rights reserved

Statement

Se pueden realizar tres tipos de operaciones:

- Consultas de datos: Ejecutar una selección y recuperar un conjunto de resultados:
 - `ResultSet rs = stmt.executeQuery("SELECT ...");`
- Modificaciones: Ejecutar una sentencia que modifique los datos: inserciones, modificaciones y borrado de filas de una tabla, instrucciones DDL y DCL.
 - `[int NumFilas =] stmt.executeUpdate("<CadenaSQL>");`
- General: Se usa cuando no se sabe el tipo de la sentencia o puede devolver varios conjuntos de resultados y/o recuentos. Los resultados se recuperan con los métodos `getResultSet()`, `getMoreResult()` y `getUpdateCount()`.
 - `stmt.execute("<CadenaSQL>");`

© JMA 2021. All rights reserved

PreparedStatement

Las mayoría de los SGBD permiten precompilar las sentencias SQL que van a ser utilizadas reiteradamente, optimizado de esta forma los accesos.

JDBC dispone del interface PreparedStatement (subclase de Statement) para utilizar dicho mecanismo. Es necesario crear una instancia:

```
PreparedStatement prStmt = con.prepareStatement("<SQL>");
```

La cadena SQL puede contar con parámetros, se representan con ? en la posición donde debe ir el argumento.

Es necesario introducir los parámetros antes de ejecutar la sentencia:

```
set<Tipo>(<NúmeroParámetro>, <Valor>);
```

- donde <Tipo> es el tipo del valor introducido y el número (empezando de 1) identifica el orden del parámetro.

Se ejecuta la sentencia con el correspondiente método executeQuery(), executeUpdate() o execute().

```
PreparedStatement pstmt = con.prepareStatement("SELECT * FROM productos WHERE id=? and fecha=?" );
```

```
pstmt.setInt(1,"10");
```

```
pstmt.setDate(2,"03/09/2001");
```

```
ResultSet rs = pstmt.executeQuery();
```

© JMA 2021. All rights reserved

CallableStatement

Como en el caso anterior, la mayoría de SGBD permiten trabajar con procedimientos almacenados, la interface CallableStatement (subclase de PreparedStatement). Se crean las instancias:

```
CallableStatement callStmt = con.prepareCall("<Llamada>");
```

Donde la cadena de invocación al procedimiento sigue el formato:

- Devuelve un valor: {?= call <Procedimiento>[<arg1>,<arg2>, ...]}
- No devuelve valor: {call <Procedimiento>[<arg1>,<arg2>, ...]}

Los parámetros de entrada son manejados de la misma forma que en PreparedStatement.

Los parámetros de salida han de ser registrados para poder recuperarlos:

- callStmt.registerOutParameter(<NúmeroParámetro>, <Tipo>);
- Se ejecuta el procedimiento almacenado con el correspondiente método executeQuery(), executeUpdate() o execute().
- <Tipo> get<Tipo>(<NúmeroParámetro>);

© JMA 2021. All rights reserved

Transacciones

El JDBC soporta el manejo de transacciones aportando los métodos:

- `setTransactionIsolation(<Nivel>)`: Fija el nivel de aislamiento.
 - `TRANSACTION_NONE`
 - No puede realizar transacciones.
 - `TRANSACTION_READ_COMMITTED`
 - Solo accede a data con transacciones terminadas.
 - `TRANSACTION_READ_UNCOMMITTED`
 - Permite consultar los valores aunque no haya terminado la transacción.
 - `TRANSACTION_REPEATABLE_READ`
 - Los datos leídos repetidamente conservan el valor inicial hasta que termine la transacción.
 - `TRANSACTION_SERIALIZABLE`
 - Los datos mantienen el valor inicial hasta que termine la transacción.
- `getTransactionIsolation()`: Devuelve el nivel de aislamiento utilizado.
- `commit()`, `rollback()`: Confirma o descarta la transacción.
- `setAutoCommit()`, `getAutoCommit()` : Fija o indica si se realiza un commit automáticamente después de cada transacción.

© JMA 2021. All rights reserved

ResultSet

Contiene un cursor con los resultados de una consulta o un procedimiento almacenado. El cursor devuelto se encuentra posicionado delante de la primera fila.

Métodos:

- `next()`: Posiciona el cursor en la siguiente fila, devolverá false cuando no tenga mas filas.
- `close()`: cierra el cursor.
- `getMetaData()`: Devuelve una instancia de `ResultSetMetaData` con la estructura de columnas y tipos.

Las columnas deben ser recuperadas de izquierda a derecha y una vez consultado su valor lo pierden. La primera columna tiene el número 1. Se utilizan los métodos:

- `<Tipo> get<Tipo>(<NúmeroColumna> o "<NombreColumna>")`;

© JMA 2021. All rights reserved

ResultSet

El tratamiento del objeto ResultSet depende de las características del mismo:

- Solo lectura
- Lectura y Escritura
- Solo movimiento hacia delante
- Movimiento adelante y atrás

Para obtener los diferentes resultados el cursor es recorrido de forma similar a los objetos Iterator, Enumeration, foreach

```
ResultSet resultado = stm.executeQuery(sql);
while ( resultado.next() ){
    System.out.println(" id producto--> "+resultado.getInt("id"));
    System.out.println(" producto--> "+resultado.getString("nombre"));
}
```

© JMA 2021. All rights reserved

ResultSet: Métodos

getArray(columna)	Devuelve el valor actual de la columna como un array de Java
getByte(columna)	Devuelve el valor actual de la columna como un byte de Java
getDate(columna)	Devuelve el valor actual de la columna como una Fecha de Java
getDouble(columna)	Devuelve el valor actual de la columna como un Double de Java
getInt(columna)	Devuelve el valor actual de la columna como un Int de Java
getString(columna)	Devuelve el valor actual de la columna como un String de Java

absolute(int)	Mueve el cursor a la fila indicada.
beforeFirst()	Mueve el cursor delante de la primera fila del objeto
first()	Mueve el cursor a la primera fila del objeto
previous()	Mueve el cursor a la anterior fila del objeto
next()	Mueve el cursor a la siguiente fila del objeto
afterLast()	Mueve el cursor detrás de la última fila del objeto

cancelRowUpdate()	Cancela los cambios realizados en la fila actual de este objeto
deleteRow()	Elimina la fila actual de este objeto y de la base de datos

© JMA 2021. All rights reserved

Recuperación Datos. Equivalencias

Tabla de equivalencia entre tipos de datos SQL y JAVA

SQL	Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

© JMA 2021. All rights reserved

ResultSetMetaData

Esta clase está preparada para obtener información acerca de las propiedades de cada una de las columnas, que conforman el cursor/tabla, que fue obtenida en la ejecución de una consulta.

El objeto `ResultSetMetaData` se obtiene a partir del objeto `ResultSet` obtenido en la ejecución de la consulta

```
ResultSetMetaData rsmd = ResultSet.getMetaData();
```

El objeto `ResultSetMetaData` dispone de métodos para obtener:

- `getColumnCount()`: Devuelve el número de columnas del `ResultSet`
- `getTableName(columna)`: Devuelve el nombre de la tabla de la columna indicada
- `columnName(columna)`: Devuelve el nombre de la columna indicada
- `getColumnDisplaySize (columna)`: Devuelve el tamaño de la columna indicada
- `getColumnType(columna)`: Devuelve el tipo de la columna como un TIPO Java
- `isNullable(columna)`: Indica si la columna permite nulos

© JMA 2021. All rights reserved

ResultSetMetaData

```

ResultSet conjuntoResultados = instruccion.executeQuery("SELECT nombre,
apellido FROM TEMPLA");
StringBuffer resultados = new StringBuffer();
ResultSetMetaData metaDatos = conjuntoResultados.getMetaData();
int numeroDeColumnas = metaDatos.getColumnCount();
for(int i = 1; i <= numeroDeColumnas; i++)
    resultados.append(metaDatos.getColumnName(i)+"\t");
resultados.append("\n");
while(conjuntoResultados.next()) {
    for(int i = 1; i <= numeroDeColumnas; i++)
        resultados.append(conjuntoResultados.getObject(i) + "\t");
    resultados.append("\n");
}
System.out.println(resultados.toString());

```

© JMA 2021. All rights reserved

DatabaseMetaData

Hay casos en los que se requiere conocer la estructura de una base de datos (nombre y diseño de las tablas, tipos de los campos, etc.).

Los datos que describen la estructura de las bases de datos es lo que se conoce como metadatos.

Los metadatos se obtienen utilizando el método getMetaData de la clase Connection.

```
DatabaseMetaData metadatos=con.getMetaData();
```

Una vez obtenido el objeto se pueden utilizar métodos para obtener información sobre la base de datos.

- Nombre de la Base de Datos
- Tablas de la BBDD
- Esquemas, etc

© JMA 2021. All rights reserved

DatabaseMetaData

- `getDatabaseProductName()`: Devuelve el nombre comercial del sistema gestor de base de datos en uso
- `getDatabaseProductVersion()`: Devuelve la versión de producto del sistema de base de datos en uso
- `getDriverName()`: Devuelve el nombre del driver JDBC en uso
- `getUserName()`: Devuelve el nombre de usuario actual del gestor de bases de datos.
- `getSchemas()`: Devuelve los esquemas de la Base de Datos.
- `getTables(String catálogo, String Esquema, String Tabla, String tipos[])`: Obtiene una tabla de resultados (ResultSet) en la que cada fila es una tabla del catálogo.

© JMA 2021. All rights reserved

DatabaseMetaData

```
DatabaseMetaData dmd = conn.getMetaData();
ResultSet rs1 = dmd.getSchemas();
while (rs1.next()) {
    String ss = rs1.getString(1);
    ResultSet rs2 = dmd.getTables(null, ss, "%", null);
    while (rs2.next())
        System.out.println(rs2.getString(3) + " " +
            rs2.getString(4));
}
conn.close();
```

© JMA 2021. All rights reserved

Procesos por Lotes

Los procesos por lotes permiten recopilar y lanzar en una sola consulta un conjunto de instrucciones.

Sólo se permiten colocar instrucciones SQL DML y DDL (UPDATE, INSERT,DELETE, CREATE TABLE,..).

Esto se realizar mediante los métodos de la clase Statement:

- addBatch: Permite añadir nuevas instrucciones al proceso por lotes.
- executeBatch: Lanza las instrucciones almacenadas como una única consulta. El resultado de executeBatch es un array de enteros donde cada elemento es el número de filas modificadas por la acción lanzada correspondiente.

```
Statement stat=con.createStatement();
stat.addBatch("CREATE TABLE.....
stat.addBatch("INSERT INTO....
...
int cuentaFilas[]=stat.executeBatch();
```

© JMA 2021. All rights reserved

Auto cierre (v7)

```
try (Connection con = JDBCbridged.getConnection()) {
    String sql = "SELECT country_id, country_name FROM hr.countries WHERE country_name LIKE ?";
    try (PreparedStatement cmd = con.prepareStatement(sql)) {
        cmd.setString(1, "U%");
        try (ResultSet rs = cmd.executeQuery()) {
            while (rs.next()) {
                System.out.println(rs.getString("country_id") + " " + rs.getString(2));
            }
        } catch (SQLException e) {
            System.out.println("ERROR: " + e.getMessage());
        }
    } catch (SQLException e) {
        System.out.println("ERROR: " + e.getMessage());
    }
} catch (SQLException e) {
    System.out.println("ERROR: " + e.getMessage());
}
```

© JMA 2021. All rights reserved



<https://mybatis.org/mybatis-3/>

© JMA 2016. All rights reserved

Contenidos

- Introducción
- Configuración
- Arquitectura
- Consultas
- Mapeo de resultados
- DML
- SQL Dinámico
- Anotaciones

© JMA 2016. All rights reserved

INTRODUCCIÓN

© JMA 2016. All rights reserved

Introducción

- MyBatis es un framework de persistencia que soporta SQL, procedimientos almacenados y mapeos avanzados. MyBatis elimina casi todo el código JDBC, el establecimiento manual de los parámetros y la obtención de resultados. MyBatis puede configurarse con XML o anotaciones y permite mapear diccionarios (mapas) y POJOs (Plain Old Java Objects) con registros de base de datos.
 - A diferencia de las herramientas ORM, MyBatis no mapea objetos Java a tablas de base de datos sino métodos a sentencias SQL, objetos a parámetros y resultados a objetos.
 - MyBatis es software libre y se ha desarrollado bajo Licencia Apache 2.0
 - MyBatis es una bifurcación de Apache iBATIS 3.0 (ahora obsoleto) y es mantenido por un equipo que incluye a los creadores originales de iBATIS.
-

© JMA 2016. All rights reserved

Instalación

- Para usar MyBatis sólo tienes que incluir el fichero mybatis-x.x.x.jar en el classpath.
 - <https://github.com/mybatis/mybatis-3/releases>
- Si usas Maven añade esta dependencia en tu pom.xml:

```
<dependency>  
  <groupId>org.mybatis</groupId>  
  <artifactId>mybatis</artifactId>  
  <version>x.x.x</version>  
</dependency>
```
- Plugin:
 - MyBatis: Eclipse plugin for MyBatis
 - MyBatis Generator: Code generator for MyBatis and iBATIS
 - MyBatis Dynamic SQL: SQL Generator for MyBatis and Spring JDBC Templates

© JMA 2016. All rights reserved

Arquitectura

- El interfaz principal para trabajar con MyBatis es el `SqlSession`. A través de este interfaz puedes ejecutar comandos, obtener mappers y gestionar transacciones.
- Las `SqlSessions` se crean por una instancia de `SqlSessionFactory`. La `SqlSessionFactory` contiene métodos para crear instancias de `SqlSessions` de distintas formas.
- La `SqlSessionFactory` en si misma se crea por la `SqlSessionFactoryBuilder` que puede crear una `SqlSessionFactory` a partir de XML, anotaciones o un objeto `Configuration` creado por código.

© JMA 2016. All rights reserved

Arquitectura

- La potencia de MyBatis reside en los Mapped Statements. Ahí es donde está la magia. Para lo potentes que son, los ficheros XML de mapeo son relativamente simples. Sin duda, si los comparas con el código JDBC equivalente comprobarás que ahorras el 95% del código.
- Desde sus comienzos, MyBatis ha sido siempre un framework XML. La configuración se basa en XML y los mapped statements se definen en XML. MyBatis 3 ofrece una nueva configuración basada en anotaciones. Las anotaciones simplemente ofrecen una forma más sencilla de implementar los mapped statements sin introducir un montón de sobrecarga.

© JMA 2016. All rights reserved

Configuración

- El `SqlSessionFactoryBuilder` tiene métodos `build()` para construir una `SqlSessionFactory` desde un documento XML o un objeto `Configuration` creado por código.

```
String resource = "com/example/mybatis-config.xml";
InputStream inputStream =
    Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new
    SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

© JMA 2016. All rights reserved

Configuración

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <settings>
    <setting name="mapUnderscoreToCamelCase" value="true" />
  </settings>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC" />
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/sakila" />
        <property name="username" value="root" />
        <property name="password" value="root" />
      </dataSource>
    </environment>
  </environments>
  <mappers>
    :
  </mappers>
</configuration>
```

© JMA 2016. All rights reserved

settings

- Son muy importantes para definir cómo se comporta MyBatis en ejecución.

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25"/>
  <setting name="defaultFetchSize" value="100"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>
</settings>
```

© JMA 2016. All rights reserved

mappers

- Hay que indicarle a MyBatis dónde encontrar los SQL mapped statements (sentencias SQL mapeadas). Java no ofrece muchas posibilidades de auto-descubrimiento así que la mejor forma es simplemente decirle a MyBatis donde encontrar los ficheros de mapeo:

```
<mappers>
  <!-- Using classpath relative resources -->
  <mapper resource="com/example/mappers/AuthorMapper.xml"/>
  <!-- Using url fully qualified paths -->
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <!-- Using mapper interface classes -->
  <mapper class="com.example.mappers.PostMapper"/>
  <!-- Register all interfaces in a package as mappers -->
  <package name="com.example.mappers"/>
</mappers>
```

© JMA 2016. All rights reserved

Configuración por código

- La clase Configuration contiene todo lo que posiblemente necesites conocer de la instancia de SqlSessionFactory.

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment = new Environment("development",
transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

© JMA 2016. All rights reserved

SqlSessionFactory

- SqlSessionFactory tiene métodos que se usan para crear instancias de SqlSession en función a si quieres usar un ámbito transaccional para esta sesión o utilizar auto-commit (lo cual equivale a no usar transacción en la mayoría de las bases de datos y/o JDBC drivers), quieres que MyBatis obtenga una conexión de un datasource o quieres proporcionar tu propia conexión, quieres que MyBatis reúse PreparedStatements y/o haga batch updates (incluyendo inserts y deletes).
- Por defecto: se arranca una transacción (sin auto-commit), se obtiene una nueva conexión con el nivel de aislamiento por defecto de la base de datos y no se reutilizaran PreparedStatements y ni actualizaciones batch.

```
try (SqlSession session = factory.openSession()) { // auto-close
```

© JMA 2016. All rights reserved

SqlSession

- La instancia de SqlSession es la clase más potente de MyBatis. Es donde encontrarás todos los métodos para ejecutar sentencias, hacer commit o rollback de transacciones y obtener mappers.
- Métodos de ejecución de sentencias
 - `<T> T selectOne(String statement, Object parameter)`
 - `<E> List<E> selectList(String statement, Object parameter)`
 - `<T> Cursor<T> selectCursor(String statement, Object parameter)`
 - `<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey)`
 - `int insert(String statement, Object parameter)`
 - `int update(String statement, Object parameter)`
 - `int delete(String statement, Object parameter)`

© JMA 2016. All rights reserved

Control de transacción

- Por defecto MyBatis no hace un commit a no ser que haya detectado que la base de datos ha sido modificada por una insert, update o delete. Si has realizado cambios sin llamar a estos métodos, entonces puedes pasar true en al método de commit() y rollback() para asegurar que se realiza el commit (ten en cuenta que aun así no puedes forzar el commit() en modo auto-commit o cuando se usa un gestor de transacciones externo). La mayoría de las veces no tendrás que llamar a rollback() dado que MyBatis lo hará por ti en caso de que no hayas llamado a commit().
 - void commit()
 - void commit(boolean force)
 - void rollback()
 - void rollback(boolean force)

© JMA 2016. All rights reserved

Mapped statements

- Los mapped statements pueden establecerse en ficheros XML, varios por fichero pero todos con un identificador único respecto al namespace (espacio de nombres) del fichero:


```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```
- Para realizar una llamada hay que especificar el nombre completo (fully qualified):


```
Blog blog = session.selectOne("com.example.BlogMapper.selectBlog",
101);
```

© JMA 2016. All rights reserved

Mapped statements

- Los mapped statements pueden establecerse mediante anotaciones en interfaces, tantos como se desee. Las anotaciones son mucho más claras para sentencias sencillas, sin embargo, las anotaciones java son limitadas y más complicadas de usar para sentencias complejas:

```
package com.example.mappers;
public interface BlogMapper {
    @Select("SELECT * FROM blog WHERE id = #{id}")
    Blog selectBlog(int id);
}
```

- Para obtener la implementación del mapped statements se solicita al `SqlSession`:

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

© JMA 2016. All rights reserved

Ciclo de vida

- El `SqlSessionFactoryBuilder` puede instanciarse, usarse y desecharse. No es necesario mantenerla una vez que ya has creado la `SqlSessionFactory`.
- Una vez creado, el `SqlSessionFactory` debería existir durante toda la ejecución de tu aplicación, no debería haber ningún o casi ningún motivo para eliminarlo o recrearlo. El mejor ámbito para el `SqlSessionFactory` es el de aplicación, usando el patrón Singleton o el Static Singleton.
- Las instancias de `SqlSession` deben seguir un patrón Unit of Work, el ámbito adecuado es el de petición (request), método o transacción. No son seguras para multiprocesos y no deben ser compartidas. Cerrar la `SqlSession` es muy importante.
- Los mappers son interfaces que creas como enlace con los mapped statements. Las instancias de mappers se obtienen de una `SqlSession` y por tanto, su ámbito no puede exceder el de la `SqlSession` de la que fueron creados.

© JMA 2016. All rights reserved

CONSULTAS

© JMA 2016. All rights reserved

select

- El select statement es uno de los elementos que más utilizarás en MyBatis. No es demasiado útil almacenar datos en la base de datos si no puedes leerlos, de hecho las aplicaciones suelen leer bastantes más datos de los que modifican. Por cada insert, update o delete posiblemente haya varias selects.
 - Este es uno de los principios básicos de MyBatis y la razón por la que se ha puesto tanto esfuerzo en las consultas y el mapeo de resultados.
-

© JMA 2016. All rights reserved

select

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
```

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

© JMA 2016. All rights reserved

Tipos

- Los tipos de datos debe ser nombres de clase cualificados (fully qualified): paquete.clase.
- Un type alias es simplemente un alias (un nombre más corto) para un tipo Java. Solo es importante para la configuración XML y existe para reducir la cantidad de texto al teclear nombres de clase cualificados:

```
<typeAliases>
  <typeAlias alias="Author" type="com.example.Author"/>
  <typeAlias alias="Blog" type="com.example.Blog"/>
  <typeAlias alias="Post" type="com.example.Post"/>
</typeAliases>
```

© JMA 2016. All rights reserved

Tipos

- Los tipos de datos debe ser nombres de clase cualificados (fully qualified): paquete.clase.
- Un type alias es simplemente un alias (un nombre más corto) para un tipo Java. Solo es importante para la configuración XML y existe para reducir la cantidad de texto al teclear nombres de clase cualificados:


```
<typeAliases>
  <typeAlias alias="Author" type="com.example.Author"/>
  <typeAlias alias="Blog" type="com.example.Blog"/>
  <typeAlias alias="Post" type="com.example.Post"/>
</typeAliases>
```
- Hay muchos type aliases pre construidos. No son sensibles a mayúsculas/minúsculas.

© JMA 2016. All rights reserved

Tipos

Alias	Tipo mapeado	Alias	Tipo mapeado
• <i>_byte</i>	<i>byte</i>	• double	Double
• <i>_long</i>	<i>long</i>	• float	Float
• <i>_short</i>	<i>short</i>	• boolean	Boolean
• <i>_int</i>	<i>int</i>	• date	Date
• <i>_integer</i>	<i>int</i>	• decimal	BigDecimal
• <i>_double</i>	<i>double</i>	• bigdecimal	BigDecimal
• <i>_float</i>	<i>float</i>	• object	Object
• <i>_boolean</i>	<i>boolean</i>	• map	Map
• string	String	• hashmap	HashMap
• byte	Byte	• list	List
• long	Long	• arraylist	ArrayList
• short	Short	• collection	Collection
• int	Integer	• iterator	Iterator
• integer	Integer		

© JMA 2016. All rights reserved

Parameters

- Los parámetros son elementos muy potentes en MyBatis. La sintaxis `#{}` hace que MyBatis genere parámetros de PreparedStatement (?) y que asigne los valores a los parámetros de forma segura.

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```
- Los tipos primitivos y los tipos de datos simples como Integer o String no tienen propiedades relevantes y por tanto el parámetro será reemplazado por su valor.
- En un objeto complejo el comportamiento es distinto, se buscarán en las propiedades los parámetros.

```
<select id="authUsers" parameterType="User">
  select count(id)
  from users
  where username = #{usr} and password = #{pwd}
</select>
```

© JMA 2016. All rights reserved

Parameters

- Acompañando al nombre del parámetro se puede especificar:
 - el tipo de Java (`javaType`) y el tipo JDBC (`jdbcType`)
 - para los tipos numéricos, `numericScale` permite especificar cuantas posiciones decimales son relevantes.
 - se puede indicar un `TypeHandler` específico (o un alias) para la conversión del tipo.
 - `mode` permite especificar parámetros IN, OUT o INOUT. Si el `mode=OUT` (o INOUT) y el `jdbcType=CURSOR`, se debe especificar un `resultMap`.

```
#{height, javaType=double, jdbcType=NUMERIC,
  numericScale=2}
```

© JMA 2016. All rights reserved

Sustitución de Strings

- Por defecto, usar la sintaxis `#{}` hace que MyBatis genere propiedades de `PreparedStatement`, esto es más seguro, más rápido y casi siempre la opción adecuada.
- En algunos casos se quiere inyectar un trozo de texto sin modificaciones dentro de la sentencia SQL, para lo que se utiliza `${}` como marcador.

```
SELECT * FROM user WHERE ${column} = #{value}
ORDER BY ${columnName}
```
- No es seguro pasar un texto introducido por el usuario a una sentencia SQL, esto permite ataques de inyección de SQL y se debe sanear antes de interpolar.

© JMA 2016. All rights reserved

Fragmentos

- Este elemento se utiliza para definir un fragmento reusable de código SQL que puede ser incluido en otras sentencias. Puede parametrizarse estáticamente (durante la fase de carga). Los diferentes valores de propiedad pueden variar en las instancias incluidas.

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password
</sql>
```
- Puede ser incluido en otras sentencias:

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias"
      value="t1"/></include>,
    <include refid="userColumns"><property name="alias"
      value="t2"/></include>
  from some_table t1 cross join some_table t2
</select>
```

© JMA 2016. All rights reserved

MAPEO DE RESULTADOS

© JMA 2016. All rights reserved

Auto mapeó

- MyBatis puede auto mapear los resultados por ti y en el resto de los casos es posible que tengas que crear un ResultMap. También se puede combinar ambas estrategias.
- Al auto mapear resultados, MyBatis obtiene el nombre de columna y busca una propiedad con el mismo nombre sin tener en cuenta las mayúsculas.
- Normalmente las columnas de base de datos se nombran usando mayúsculas y separando las palabras con un subrayado, mientras que las propiedades java se nombran habitualmente siguiendo la notación tipo camelcase. Para habilitar el auto-mapeo entre ellas hay que establecer el parámetro de configuración `mapUnderscoreToCamelCase` a `true`.
- El auto-mapeo funciona incluso cuando hay un ResultMap específico. Cuando esto sucede, para cada result map, todas las columnas que están presentes en el ResultSet y que no tienen un mapeo manual se auto-mapearán. Posteriormente se procesarán los mapeos manuales.
- Hay tres niveles de auto mapeo:
 - NONE: desactiva el auto mapeo. Solo las propiedades mapeadas manualmente se informarán.
 - PARTIAL: auto mapea todos los resultados que no tienen un mapeo anidado definido en su interior (joins).
 - FULL: lo auto mapea todo.

© JMA 2016. All rights reserved

Auto mapeo

- Para utilizar el auto mapeo, se pueden solucionar las discrepancias entre columnas y propiedades reescribiendo las consultas asignando a cada columna un alias que se corresponda con la propiedad:

```
<select id="selectUsers" resultType="com.example.User">
  select
    user_id as "id",
    user_name as "name",
    hashed_password as "password"
  from some_table
  where user_id = #{id}
</select>
```

© JMA 2016. All rights reserved

Mapeo manual

- El elemento resultMap es el elemento más importante y potente de MyBatis. Permite eliminar el 90% del código que requiere el JDBC para obtener datos de ResultSets, y en algunos casos incluso permite hacer cosas que no están siquiera soportadas en JDBC. En realidad, escribir un código equivalente para realizar algo similar a un mapeo para un statement complejo podría requerir cientos de líneas de código.
- El diseño de los ResultMaps es tal, que los statements simples no requieren un resultMap explícito (auto-mapeo), y los statements más complejos requieren sólo la información imprescindible para describir relaciones.

© JMA 2016. All rights reserved

Result Maps

- El elemento resultMap permite crear mapeadores de resultados reutilizables. Se debe indicar un identificador único (id) dentro del namespace que se utiliza para identificar el result map, el nombre completamente cualificado de la clase o el alias del parámetro que se pasará al statement (type) y, opcionalmente, autoMapping que sobrescribe el parametro global autoMappingBehavior.


```
<resultMap id="detailedBlogResultMap" type="Blog">
</resultMap>
```
- Dentro del elemento se define la estructura del mapeo.

© JMA 2016. All rights reserved

Result Maps

- id: result ID; marcar los results con ID mejora el rendimiento
- result: un resultado normal inyectado en un campo o una propiedad de un JavaBean
- constructor: usado para inyectar resultados en el constructor de la clase durante la instanciación
 - idArg: argumento ID; marcar el argumento ID mejora el rendimiento
 - arg: un resultado normal inyectado en el constructor
- association: una asociación con un objeto complejo; muchos resultados acabarán siendo de este tipo
 - result mapping anidado: las asociaciones son resultMaps en sí mismas o pueden apuntar a otro resultMap
- collection: una colección de tipos complejos
 - result mapping anidado: las colecciones son resultMaps en sí mismas o pueden apuntar a otro resultMap
- discriminator: utiliza un valor del resultado para determinar qué resultMap utilizar
 - case: un resultMap basado en un valor concreto
 - result mapping anidado – un case es un resultMap en sí mismo y por tanto puede contener a su vez elementos propios de un resultMap o bien apuntar a un resultMap externo.

© JMA 2016. All rights reserved

Mapecto simple

- Estos son los mapeadores más sencillos. Ambos id y result mapean una columna con una propiedad o campo de un tipo de dato simple (String, int, double, Date, etc.). Opcionalmente se puede indicar el tipo de Java (javaType), el tipo JDBC (jdbcType) y un TypeHandler específico (o un alias) para la conversión del tipo.
- La única diferencia entre ambos es que id marca que dicho resultado es un identificador y dicha propiedad se utilizará en las comparaciones entre instancias de objetos. Esto mejora el rendimiento global y especialmente el rendimiento de la cache y los mapeos anidados (ej. mapeo de joins).

```
<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>
```

© JMA 2016. All rights reserved

constructor

- Por defecto, MyBatis usa el constructor sin parametros para crear una instancia y los setter para rellenar la propiedades.
- La inyección en el constructor permite informar valores durante la instanciación de la clase, sin necesidad de exponer métodos públicos.

```
<constructor>
  <idArg column="id" javaType="_int" name="id" />
  <arg column="age" javaType="int" name="age" />
  <arg javaType="String" name="username" />
</constructor>
```

© JMA 2016. All rights reserved

association

- El elemento `association` trata las relaciones de tipo “tiene-un”, relaciones 1 a 1 y n a 1. Convierte la clave ajena en un objeto. MyBatis puede hacerlo de dos formas distintas:
 - Nested Select: Ejecutando otra select que devuelve el tipo complejo deseado.


```
<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author"
    select="selectAuthor"/>
</resultMap>
```
 - Nested Results: Usando un `ResultMap` anidado que trata con los datos repetidos de resultsets provenientes de joins.


```
<association property="author" javaType="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
</association>
```
- Por defecto MyBatis sólo crea objetos hijos si al menos una de las columnas mapeadas a las propiedades de dicho objeto es no nula. El atributo `columnPrefix` permite mapear columnas de un join a un `Result Map` externo.

© JMA 2016. All rights reserved

collection

- El elemento `collection` trata las relaciones de tipo “tiene-varios”, relaciones 1 a n. Funciona de una forma muy similar al `association`.
- La principal diferencia es que la propiedad debe ser una lista o un conjunto.


```
<collection property="posts" javaType="ArrayList"
  column="id" ofType="Post" select="selectPostsForBlog"/>
```
- Se puede mapear sobre un join de forma anidada o utilizar un prefijo para indicar las columnas adecuadas:


```
<collection property="posts" ofType="Post"
  resultMap="blogPostResult" columnPrefix="post_"/>
```

© JMA 2016. All rights reserved

discriminator

- El elemento discriminator permite mapear un conjunto de resultados a distintos tipos de datos o a una jerarquía de herencia de clases. El discriminador funciona muy parecido la sentencia switch de Java.
- Una definición de discriminator especifica los atributos column, que contiene los valores que actúan como selectores, y javaType, para asegurar que se utiliza el tipo de comparación adecuada, por defecto cadena.
- Esta compuesta por un conjuntos de elementos case, y cada uno establece el valor que debe tener la columna selectora para que se aplique su mapeo.
- Si coincide con alguno de los casos del discriminator entonces usará el resultMap especificado en cada caso.
- Si no coincide ninguno de los casos se utilizará solo el resultMap definido fuera del bloque discriminator en caso de que existiera.

© JMA 2016. All rights reserved

discriminator

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin"/>
  <result property="year" column="year"/>
  <result property="make" column="make"/>
  <result property="model" column="model"/>
  <result property="color" column="color"/>
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultMap="carResult"/>
    <case value="2" resultMap="truckResult"/>
    <case value="3" resultMap="vanResult"/>
    <case value="4" resultMap="suvResult"/>
  </discriminator>
</resultMap>
```

© JMA 2016. All rights reserved

DML

© JMA 2016. All rights reserved

insert

```
<insert
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<insert id="insertAuthor">
  insert into Author (id,username,password,email,bio)
  values ({#id},{#username},{#password},{#email},{#bio})
</insert>
```

© JMA 2016. All rights reserved

update

```
<update
  id="updateAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<update id="updateAuthor">
  update Author set
    username = #{username}, password = #{password},
    email = #{email}, bio = #{bio}
  where id = #{id}
</update>
```

© JMA 2016. All rights reserved

delete

```
<delete
  id="deleteAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete id="deleteAuthor">
  delete from Author where id = #{id}
</delete>
```

© JMA 2016. All rights reserved

Procedimientos almacenados

```
<select id="selectBlog" resultSets="blogs,authors"
      resultMap="blogResult"
      statementType="CALLABLE">
  { call getBlogsAndAuthors(
    #{id, jdbcType=INTEGER, mode=IN},
    #{count, jdbcType=INTEGER, mode=OUT},
    )
  }
</select>
```

© JMA 2016. All rights reserved

SQL DINÁMICO

© JMA 2016. All rights reserved

SQL Dinámico

- Una de las características más potentes de MyBatis ha sido siempre sus capacidades de SQL dinámico. MyBatis aporta un lenguaje de SQL dinámico potente que puede usarse en cualquier mapped statement.
- Los elementos de SQL dinámico deberían ser familiares a aquel que haya usado JSTL o algún procesador de texto basado en XML. En versiones anteriores de MyBatis había un montón de elementos que conocer y comprender. MyBatis 3 mejora esto y ahora hay algo menos de la mitad de esos elementos con los que trabajar. MyBatis emplea potentes expresiones OGNL para eliminar la necesidad del resto de los elementos:
 - if
 - choose (when, otherwise)
 - trim (where, set)
 - foreach

© JMA 2016. All rights reserved

if

```
<select id="findActiveBlogLike"
      resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

© JMA 2016. All rights reserved

choose, when, otherwise

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

© JMA 2016. All rights reserved

where

- El elemento where sabe que debe insertar la “WHERE” solo si los tags internos devuelven algún contenido. Más aun, si el contenido comienza con “AND” o “OR”, sabe cómo eliminarlo.

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
      state = #{state}
    </if>
    <if test="title != null">
      AND title like #{title}
    </if>
  </where>
</select>
```

© JMA 2016. All rights reserved

set

- El elemento set se puede usar para incluir dinámicamente columnas a modificar y dejar fuera las demás: prefijará dinámicamente con SET y eliminará todas las comas sobrantes que pudieran quedar tras las asignaciones de valor después de que se hayan aplicado las condiciones.

```
<update id="updateAuthorIfNecessary">
  update Author
  <set>
    <if test="username != null">username=#{username},</if>
    <if test="password != null">password=#{password},</if>
    <if test="email != null">email=#{email},</if>
    <if test="bio != null">bio=#{bio}</if>
  </set>
  where id=#{id}
</update>
```

© JMA 2016. All rights reserved

trim

- El elemento trim prefijará dinámicamente con el valor del atributo prefix si se cumple alguna de las condiciones contenidas. Del resultado se elimina cualquier cosa que se haya especificado en el atributo prefixOverrides, y que se inserta todo lo incluido en with. El atributo prefixOverrides acepta una lista de textos delimitados por el carácter "|" donde el espacio en blanco es relevante. where y set son especializaciones de trim:

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

© JMA 2016. All rights reserved

foreach

- Otra necesidad común del SQL dinámico es iterar sobre una colección, habitualmente para construir una condición IN.

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
    open="(" separator="," close=")">
    #{item}
  </foreach>
</select>
```

© JMA 2016. All rights reserved

bind

- El elemento bind permite crear una variable a partir de una expresión OGNL y asociarla al contexto.

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value=
    "'%' + _parameter.getTitle() + '%'" />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

© JMA 2016. All rights reserved

selectKey

- MyBatis puede tratar automáticamente las claves autogeneradas. Cuando las bases de datos que no soportan columnas autogeneradas o el driver JDBC no haya incluido aun dicho soporte, se puede utilizar selectKey.

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from
    SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author(id, username, password)
  values ({id}, #{username}, #{password})
</insert>
```
- Si order es BEFORE, entonces la obtención de la clave se realizará primero, se informará el campo indicado en keyProperty y se ejecutará la insert. Si es AFTER se ejecuta primero la insert y después la selectKey.

© JMA 2016. All rights reserved

Dialectos de bases de datos

- Si se ha configurado un databaseIdProvider, la variable "_databaseId" estará disponible en el código dinámico, de esta forma se pueden ajustar las sentencias a las diferencias del SQL según el fabricante de la base de datos.

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1
    </if>
  </selectKey>
  insert into users values ({id}, #{name})
</insert>
```

© JMA 2016. All rights reserved

SQL Builder

- Una de las cosas más tediosas que un programador Java puede llegar a tener que hacer es incluir código SQL en código Java. Normalmente esto se hace cuando es necesario generar dinámicamente el SQL – de otra forma podrías externalizar el código en un fichero o un procedimiento almacenado. MyBatis tiene una respuesta potente a la generación dinámica de SQL mediante las capacidades del mapeo XML. Sin embargo, en ocasiones se hace necesario construir una sentencia SQL dentro del código Java.
- MyBatis 3 introduce un concepto un tanto distinto para tratar con el problema, con la clase SQL, se puede crear una sentencia SQL en un sólo paso invocando a sus métodos.

© JMA 2016. All rights reserved

SQL Builder

```
String sql =
" SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME, " +
" P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +
" FROM PERSON P, ACCOUNT A " +
" INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +
" INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +
" WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +
" OR (P.LAST_NAME like ?) " +
" GROUP BY P.ID " +
" HAVING (P.LAST_NAME like ?) " +
" OR (P.FIRST_NAME like ?) " +
" ORDER BY P.ID, P.FULL_NAME";
```

© JMA 2016. All rights reserved

SQL Builder

```
private String selectPersonSql() {  
    return new SQL() {{  
        SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");  
        SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");  
        FROM("PERSON P");  
        FROM("ACCOUNT A");  
        INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");  
        INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");  
        WHERE("P.ID = A.ID");  
        WHERE("P.FIRST_NAME like ?");  
        OR();  
        WHERE("P.LAST_NAME like ?");  
        GROUP_BY("P.ID");  
        HAVING("P.LAST_NAME like ?");  
        OR();  
        HAVING("P.FIRST_NAME like ?");  
        ORDER_BY("P.ID");  
        ORDER_BY("P.FULL_NAME");  
    }}.toString();  
}
```

© JMA 2016. All rights reserved

ANOTACIONES

© JMA 2016. All rights reserved

Interface

@Mapper

```
public interface UserMapper {
    @Select("SELECT id, name FROM users WHERE name = #{name}")
    User selectById(@Param("name") String value);
    @MapKey("id")
    @Select("SELECT id, name FROM users WHERE name LIKE #{name} || '%'")
    Map<Integer, User> selectByStartingWithName(String name);
    @Options(useGeneratedKeys = true, keyProperty = "id")
    @SelectKey(statement = "SELECT identity('users')", keyProperty = "id", before =
        true, resultType = int.class)
    @Insert("INSERT INTO users (id, name) VALUES(#{id}, #{name})")
    void insert(User user);
    @Update("UPDATE users SET name = #{name} WHERE id = #{id}")
    boolean update(User user);
    @Delete("DELETE FROM users WHERE id = #{id}")
    boolean deleteById(int id);
}
```

© JMA 2016. All rights reserved

Maapeo de resultados

```
@Results(id="UserResult", value = {
    @Result(property = "id", column = "id", id = true),
    @Result(property = "name", column = "name"),
    @Result(property = "email" column = "id", one =
        @One(select = "selectUserEmailById", fetchType =
            FetchType.LAZY)),
    @Result(property = "telephoneNumbers" column = "id",
        many = @Many(select =
            "selectAllUserTelephoneNumberById", fetchType =
                FetchType.LAZY))
})
```

```
@ResultMap("UserResult")
```

© JMA 2016. All rights reserved

Mapeo de resultados

```
@ConstructorArgs({
    @Arg(column = "id", javaType = int.class, id = true),
    @Arg(column = "name", javaType = String.class),
    @Arg(javaType = UserEmail.class, select = "selectUserEmailById", column = "id")
})

@AutomapConstructor
public User(int id, String name) { ... }

@TypeDiscriminator(column = "type", javaType = String.class,
    cases = {
        @Case(value = "1", type = PremiumUser.class),
        @Case(value = "2", type = GeneralUser.class),
        @Case(value = "3", type = TemporaryUser.class)
    }
})
```

© JMA 2016. All rights reserved

SQL dinámico

```
@Update({"<script>",
    "update Author",
    " <set>",
    " <if test='user != null'>username=#{user},</if>",
    " <if test='pwd != null'>password=#{pwd},</if>",
    " <if test='email != null'>email=#{email},</if>",
    " <if test='bio != null'>bio=#{bio}</if>",
    " </set>",
    "where id=#{id}",
    "</script>"}
void updateAuthorValues(Author author);
```

© JMA 2016. All rights reserved

ARQUITECTURA

© JMA 2020. All rights reserved

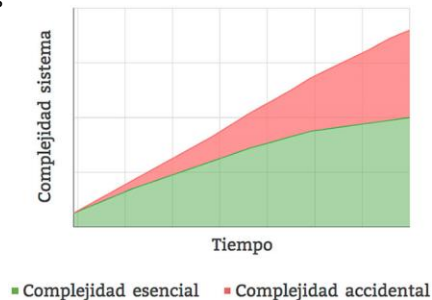
Arquitectura ágil

- Es económicamente factible realizar cambios radicales si la estructura del software separa sus aspectos de forma eficaz y dispone de las pruebas unitarias adecuadas de la arquitectura del sistema.
 - Es posible iniciar un proyecto de software con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura.
 - Los estándares facilitan la reutilización de ideas y componentes, reclutar personas con experiencia relevante, encapsulan buenas ideas y conectan componentes. Sin embargo, el proceso de creación de estándares a veces puede llevar demasiado tiempo para que la industria espere, y algunos estándares pierden contacto con las necesidades reales de los adoptantes a los que están destinados a servir. Hay que usar estándares cuando añadan un valor demostrable.
-

© JMA 2020. All rights reserved

Complejidad

- La complejidad esencial está causada por el problema a resolver y nada puede eliminarla; si los usuarios quieren que un programa haga 30 cosas diferentes, entonces esas 30 cosas son esenciales y el programa debe hacer esas 30 cosas diferentes.
- La complejidad accidental se relaciona con problemas que los desarrolladores crean y pueden solucionar; por ejemplo, los detalles de escribir y optimizar el código o las demoras causadas por el procesamiento por lotes.
- Con el tiempo, la complejidad accidental ha disminuido sustancialmente, los programadores de hoy dedican la mayor parte de su tiempo a abordar la complejidad esencial.



© JMA 2020. All rights reserved

Arquitectura Limpia

- En los últimos años, hemos visto una amplia gama de ideas sobre la arquitectura de los sistemas. Éstos incluyen:
 - Arquitectura Hexagonal (también conocida como Puertos y Adaptadores) por Alistair Cockburn y adoptada por Steve Freeman y Nat Pryce en su libro Growing Object Oriented Software
 - Arquitectura de cebolla (Onion Architecture) de Jeffrey Palermo
 - Screaming Architecture de Robert Martin
 - DCI de James Coplien y Trygve Reenskaug.
 - BCE por Ivar Jacobson de su libro Ingeniería de software orientada a objetos: un enfoque basado en casos de uso.
- Aunque todas estas arquitecturas varían algo en sus detalles, son muy similares. Todos tienen el mismo objetivo, que es la separación de conceptos. Todos logran esta separación dividiendo el software en capas. Cada uno tiene al menos una capa para las reglas de negocio y otra para las interfaces.

© JMA 2020. All rights reserved

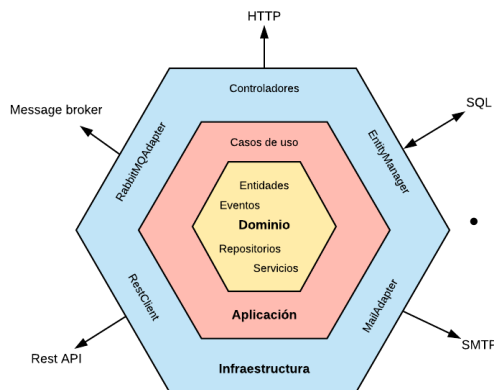
Arquitecturas multicapas

- Cada capa se apoya en la capa subsiguiente (capas horizontales) que depende de las capas debajo de ella, que a su vez dependerá de alguna infraestructura común y servicios públicos.
- El gran inconveniente de esta arquitectura en capas de arriba hacia abajo es el acoplamiento que crea.
- Hay sistemas estrictos (strict layered systems) y relajados (relaxed layered systems) que determinarán el nivel de acoplamiento de las dependencias entre capas.
- Las preocupaciones transversales provocan la aparición de capas verticales.
- Todo esto crea una complejidad accidental innecesaria.



© JMA 2020. All rights reserved

Arquitectura Hexagonal

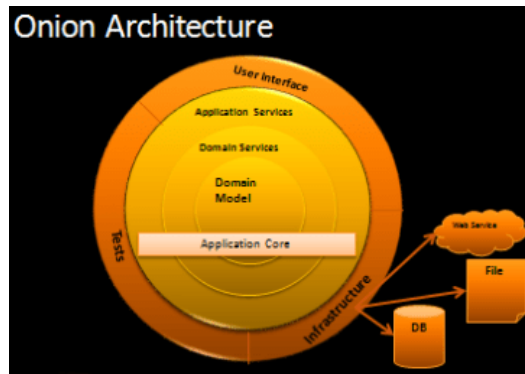


- Propone que nuestro dominio sea el núcleo de las capas y que este no se acople a nada externo. Mediante el principio de inversión de dependencias, en vez de hacer uso explícito, nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas.
- También llamada arquitectura de puertos y adaptadores:
 - Puerto: definición de una interfaz pública.
 - Adapter: especialización de un puerto para un contexto concreto.

© JMA 2020. All rights reserved

Onion Architecture

El termino Onion Architecture o Arquitectura cebolla fue acuñada por Jeffrey Palermo, es un patrón arquitectónico basado en capas concéntricas, de fuera a dentro (núcleo), y en mecanismos de inyección de dependencias, para disminuir el acoplamiento entre las capas.



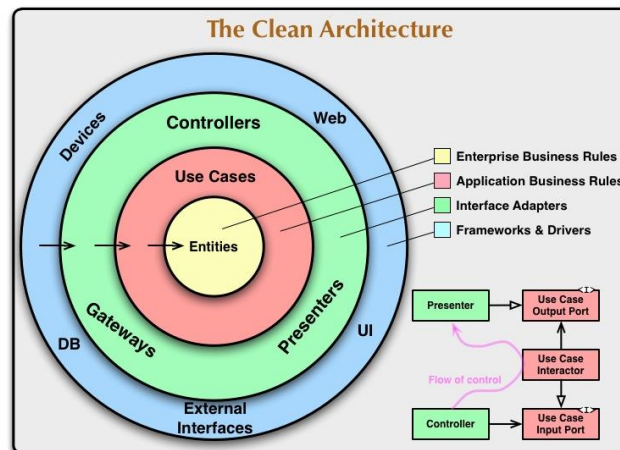
© JMA 2020. All rights reserved

Arquitectura Limpia

- Independiente de los marcos: La arquitectura no depende de la existencia de alguna biblioteca de software cargado de funciones. Esto permite utilizar dichos marcos como herramientas, en lugar de tener que ajustar el sistema a las limitaciones impuestas.
- Comprobable: Las reglas de negocio se pueden probar sin la interfaz de usuario, la base de datos, el servidor web o cualquier otro elemento externo.
- Independiente de la IU: La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario web podría reemplazarse por una interfaz de usuario de consola, por ejemplo, sin cambiar las reglas de negocio.
- Independiente de la base de datos: Puede cambiar Oracle o SQL Server por Mongo, BigTable, CouchDB u otra cosa. Las reglas de negocio no están vinculadas a la base de datos.
- Independiente de cualquier dependencia externa: De hecho, las reglas de negocio simplemente no saben nada sobre el mundo exterior.

© JMA 2020. All rights reserved

Arquitectura Limpia



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

© JMA 2020. All rights reserved

La regla de la dependencia

- El diagrama presenta una arquitectura basada en “capas de cebolla”, enfoque propuesto por Jeffrey Palermo.
- Los círculos concéntricos representan diferentes áreas del software. En general, cuanto más avanza, mayor nivel se vuelve el software. Los círculos exteriores son mecanismos. Los círculos internos son políticas.
- La regla primordial que hace que esta arquitectura funcione es la regla de dependencia. Esta regla dice que las dependencias del código fuente solo pueden apuntar hacia adentro. Nada en un círculo interno puede saber nada sobre algo en un círculo externo. En particular, el nombre de algo declarado en un círculo exterior no debe ser mencionado por el código en un círculo interior. Eso incluye funciones, clases, variables o cualquier otra entidad de software nombrada.
- Del mismo modo, los formatos de datos utilizados en un círculo exterior no deben ser utilizados por un círculo interior, especialmente si esos formatos son generados por un marco en un círculo exterior. No queremos que nada en un círculo externo impacte en los círculos internos.

© JMA 2020. All rights reserved

Entidades

- Las entidades encapsulan los conceptos del negocio. Una entidad puede ser un objeto con métodos o puede ser un conjunto de funciones y estructuras de datos. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.
- Estas entidades son los objetos de dominio de la aplicación. Encapsulan las reglas más generales y de alto nivel.
- Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, no esperaríamos que estos objetos se vieran afectados por un cambio en la navegación de la página o la seguridad.
- Ningún cambio operativo en una aplicación en particular debería afectar la capa de la entidad.

© JMA 2020. All rights reserved

Casos de uso

- Esta capa contiene las reglas de negocio específicas de la aplicación. Encapsula e implementa todos los casos de uso del sistema. Éstos orquestan el flujo de datos hacia-desde las entidades y hacen que las entidades usen su lógica de negocio para conseguir el objetivo del caso de uso.
- No esperamos que los cambios en esta capa afecten a las entidades. Tampoco esperamos que esta capa se vea afectada por cambios en las externalidades como la base de datos, la interfaz de usuario o cualquiera de los marcos comunes. Esta capa está aislada de tales preocupaciones.
- Sin embargo, si es esperable que los cambios en el funcionamiento de la aplicación o en las reglas del negocio vayan a afectar a los casos de uso y, por lo tanto, a esta capa. Es decir, si los detalles de un caso de uso cambian, algún código de esta capa se verá afectado y habrá que cambiarlo.

© JMA 2020. All rights reserved

Adaptadores de interfaz

- Esta capa contiene un conjunto de adaptadores que convierten los datos desde el formato más conveniente para el caso de uso y entidades al formato más conveniente o aceptado por elementos externos como la base de datos o la UI. Por ejemplo, esta capa es la que contendrá por completo la arquitectura MVC de una GUI.
- Los presentadores, vistas y controladores pertenecen a esta capa. Los modelos son sólo estructuras que se pasan desde los controladores a los casos de uso y de vuelta desde los casos de uso a los presentadores y/o vistas.
- Del mismo modo, los datos son convertidos en esta capa, desde la forma de las entidades y casos de uso a la forma conveniente para la herramienta de persistencia de datos.
- En este círculo se encuentran también los demás adaptadores encargados de convertir datos obtenidos de una fuente externa, como un servicio externo, al formato interno usado por los casos de uso y las entidades.

© JMA 2020. All rights reserved

Frameworks y drivers

- El círculo más externo se compone generalmente de frameworks y herramientas como la base de datos, el framework web, etc.
- Generalmente en esta capa sólo se escribe código que actúa de “pegamento” con los círculos interiores.
- En esta capa es donde van todos los detalles. La web es un detalle. La base de datos es un detalle. Se mantienen estas cosas afuera, donde no pueden hacer mucho daño.

© JMA 2020. All rights reserved

Cruzando fronteras

- En la parte inferior derecha del diagrama hay un ejemplo de cómo se cruzan las fronteras del círculo. Muestra cómo los controladores y presentadores se comunican con los casos de uso del círculo interno.
- El control del flujo empieza en el controlador, atraviesa el caso de uso y finaliza en el presentador. La dependencia a nivel de código fuente apunta hacia adentro, hacia los casos de uso.
- Se resuelve esta supuesta contradicción usando el principio de inversión de dependencia, es decir, desarrollando interfaces y relaciones de herencia de tal manera que las dependencias de código fuente se oponen al flujo de control en sólo algunos puntos a través de la frontera.
- Considerando que los casos de uso necesitan llamar al presentador, esta llamada no debe ser directa ya que se violaría el principio de la regla de dependencia. Ninguna implementación de un círculo exterior puede ser llamado por un círculo interior. En esta situación el caso de uso llama a una interfaz definida en el círculo interno y hace que el presentador implemente dicha interfaz en el círculo exterior.

© JMA 2020. All rights reserved

Datos que cruzan los límites

- Las estructuras de datos que se pasan a través de las fronteras deben ser simples.
- No se debería pasar entidades o filas de base de datos. Muchos frameworks de base de datos devuelve los datos en el formato de respuesta de una query. Por ejemplo, un RowStructure. Pasar los datos en ese formato violaría la regla de dependencia obligando a un círculo interno saber algo sobre un círculo exterior.
- Las estructuras de datos que tienen cualquier tipo de dependencia con capas exteriores viola la regla de dependencia.
- Los datos que cruzan las fronteras deben ser estructuras de datos simples. Se puede utilizar estructuras básicas u objetos Data Transfer. O los datos pueden ser argumentos en las llamadas a funciones. O bien, almacenar en un diccionario o hash. Cuando se pasan datos a través de una frontera, siempre debe ser en la forma más conveniente para el círculo interior.

© JMA 2020. All rights reserved

Beneficios

- Cumplir con estas simples reglas no es difícil y ahorrará muchos dolores de cabeza en el futuro.
- Al separar el software en capas y cumplir con la regla de dependencia, obliga a que nuestro dominio no esté acoplado a nada externo mediante el uso de interfaces propias del dominio que son implementadas por elementos externos (Alta reutilización: Principio de Responsabilidad Única (SRP) de SOLID).
- Esto creará un sistema que es intrínsecamente comprobable, con todos los beneficios que ello implica (Alta testabilidad: Aplicación del Principio de Inversión de Dependencias (DIP) de SOLID para la interacción del dominio con el resto de elementos)
- Permite estar preparado para cambiar los detalles de implementación, cuando alguna de las partes externas del sistema se vuelve obsoleta, como la base de datos o el marco web, se pueden reemplazar con un mínimo de esfuerzo (Alta tolerancia al cambio: Principio de Abierto/Cerrado (OCP) de SOLID derivado de la aplicación del DIP).
- Permite postergar decisiones importantes.

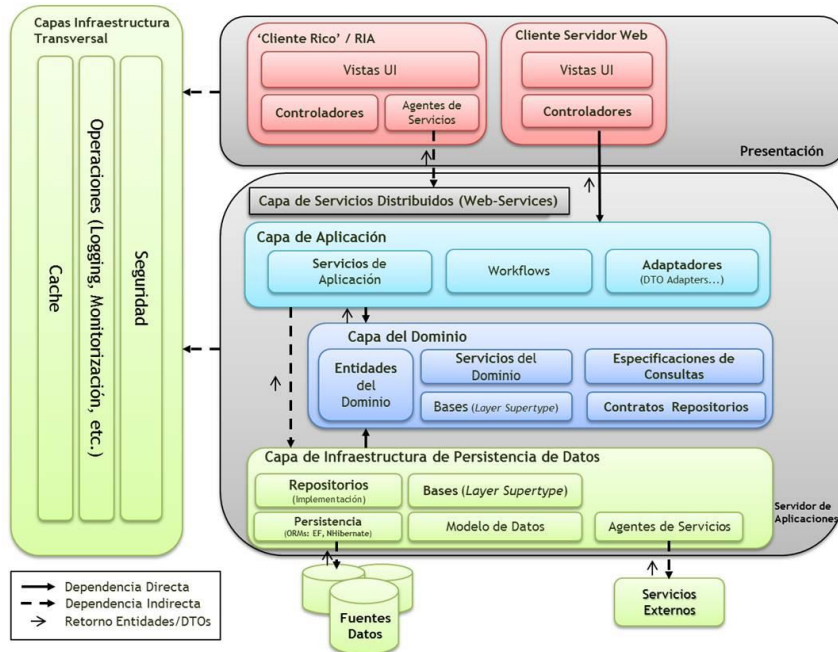
© JMA 2020. All rights reserved

Postergar decisiones importantes

- Una buena arquitectura nos permite postergar decisiones importantes, esto no significa que estemos obligados a hacerlo. Sin embargo, al poder postergarlas tenemos muchísima flexibilidad.
- Cuanto más tarde tomemos una decisión, más conocimiento tendremos del negocio y del entorno técnico:
 - Soluciones más simples/sencillas
 - Soluciones más pequeñas y manejables
 - Mayor productividad, minimiza el hacer cosas que no aporte valor real al final, retrabajos y adaptaciones
- Nuestro objetivo es tener mucho juego de cintura, reaccionar bien y rápido a cualquier cosa y sin echarnos las manos a la cabeza. Cuanto más postponemos, añadimos menos cosas innecesarias, menos llenamos la mochila y es más difícil coger lastre, por lo que podemos viajar ligeros. Como cualquiera sabe: una mudanza se hace fácil si tenemos pocas cosas.

© JMA 2020. All rights reserved

Arquitectura N-Capas con Orientación al Dominio



Capas de DDD

- **Interface de usuario (User Interface)**
 - Responsable de presentar la información al usuario, interpretar sus acciones y enviarlas a la aplicación.
- **Aplicación (Application)**
 - Responsable de coordinar todos los elementos de la aplicación. No contiene lógica de negocio ni mantiene el estado de los objetos de negocio. Es responsable de mantener el estado de la aplicación y del flujo de esta.
- **Dominio (Domain)**
 - Contiene la información sobre el Dominio. Es el núcleo de la parte de la aplicación que contiene las reglas de negocio. Es responsable de mantener el estado de los objetos de negocio. La persistencia de estos objetos se delega en la capa de infraestructura.
- **Infraestructura (Infrastructure)**
 - Esta capa es la capa de soporte para el resto de capas. Provee la comunicación entre las otras capas, implementa la persistencia de los objetos de negocio y las librerías de soporte para las otras capas (Interface, Comunicación, Almacenamiento, etc..)
- Dado que son capas conceptuales, su implementación puede ser muy variada y en una misma aplicación, tendremos partes o componentes que formen parte de cada una de estas capas.

Entidades de Dominio

- Una entidad es cualquier objeto del dominio que mantiene un estado y comportamiento más allá de la ejecución de la aplicación y que necesita ser distinguido de otro que tenga las mismas propiedades y comportamientos.
- Es un tipo de clase dedicada a representar un modelo de dominio persistente que:
 - Debe ser publica (no puede ser estar anidada ni final o tener miembros finales)
 - Deben tener un constructor público sin ningún tipo de argumentos.
 - Para cada propiedad que queramos persistir debe haber un método get/set asociado.
 - Debe tener una clave primaria
 - Debería sobrescribir equals para la identidad

© JMA 2020. All rights reserved

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de entidades asociadas que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
 - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
 - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

© JMA 2020. All rights reserved

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red o de diferentes capas.
- Su finalidad es:
 - Desacoplar del nivel de servicio de la capa de base de datos.
 - Quitar las referencias circulares.
 - Ocultar determinadas propiedades que los clientes no deberían ver.
 - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
 - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
 - Evitar el "exceso" y las vulnerabilidades por publicación.

© JMA 2020. All rights reserved

Repositorio

- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

© JMA 2020. All rights reserved

Servicio

- Los servicios representan operaciones, acciones o actividades que no pertenecen conceptualmente a ningún objeto de dominio concreto. Los servicios no tienen ni estado propio ni un significado más allá que la acción que los definen.
- Podemos dividir los servicios en tres tipos diferentes:
 - Domain services
 - Son responsables del comportamiento más específico del dominio, es decir, realizan acciones que no dependen de la aplicación concreta que estemos desarrollando, sino que pertenecen a la parte más interna del dominio y que podrían tener sentido en otras aplicaciones pertenecientes al mismo dominio.
 - Application services
 - Son responsables del flujo principal de la aplicación, es decir, son los casos de uso de nuestra aplicación. Son la parte visible al exterior del dominio de nuestro sistema, por lo que son el punto de entrada-salida para interactuar con la funcionalidad interna del dominio. Su función es coordinar entidades, value objects, domain services e infrastructure services para llevar a cabo una acción.
 - Infrastructure services
 - Declaran comportamiento que no pertenece realmente al dominio de la aplicación pero que debemos ser capaces de realizar como parte de este.

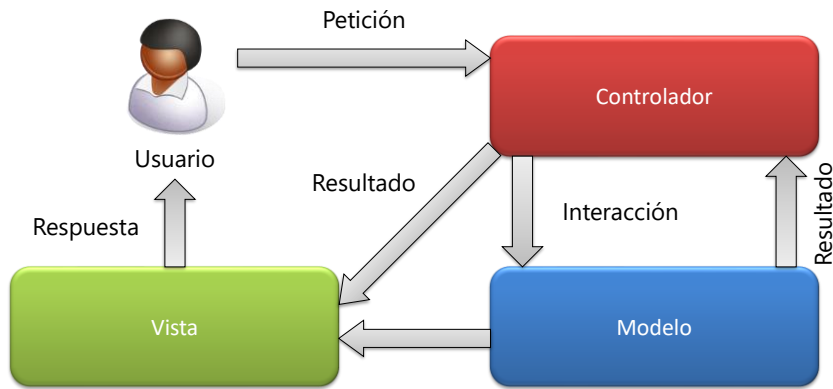
© JMA 2020. All rights reserved

El patrón MVC

- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, ...) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net ...)

© JMA 2020. All rights reserved

El patrón MVC



© JMA 2020. All rights reserved

El patrón MVC



- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**



- **Interfaz** de usuario
- Incluye elementos de **interacción**

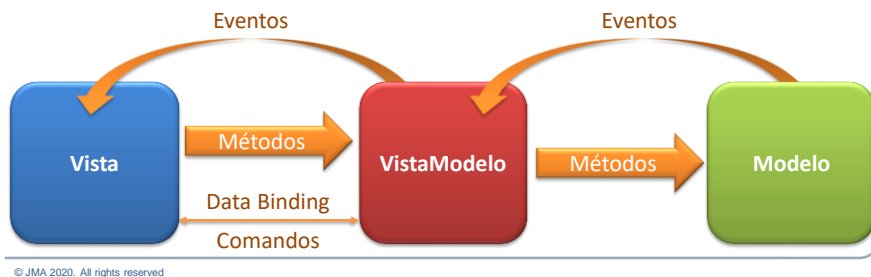


- **Intermediario** entre Modelo y Vista
- **Mapa acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

© JMA 2020. All rights reserved

Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaModelo** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



¿Cuáles son los beneficios del patrón MVVM?

- Separación de vista / presentación.
- Permite las pruebas unitarias: como la lógica de presentación está separada de la vista, podemos realizar pruebas unitarias sobre la VistaModelo.
- Mejora la reutilización de código.
- Soporte para manejar datos en tiempo de diseño.
- Múltiples vistas: la VistaModelo puede ser presentada en múltiples vistas, dependiendo del rol del usuario por ejemplo.

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto con unas técnicas de programación:
 - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
 - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir. Técnicas de implementación:
 - Service Locator: es un componente (contenedor) que contiene referencias a los servicios y encapsula la lógica que los localiza dichos servicios.
 - Inyección de dependencias.

© JMA 2020. All rights reserved

Inyección de Dependencias

- Las dependencias son expresadas en términos de interfaces en lugar de clases concretas y se resuelven dinámicamente en tiempo de ejecución.
- La Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java o .NET, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del Reflection y crear una instancia de dicha clase inyectándole sus dependencias.

© JMA 2020. All rights reserved

<http://spring.io>

SPRING CON SPRING BOOT

© JMA 2016. All rights reserved

Spring

- Inicialmente era un ejemplo hecho para el libro “J2EE design and development” de Rod Johnson en 2003, que defendía alternativas a la “visión oficial” de aplicación JavaEE basada en EJBs.
- Actualmente es un framework open source que facilita el desarrollo de aplicaciones java JEE & JSE (no está limitado a aplicaciones Web, ni a java pueden ser .NET, Silverlight, Windows Phone, etc.)
- Provee de un contenedor encargado de manejar el ciclo de vida de los objetos (beans) para que los desarrolladores se enfoquen a la lógica de negocio. Permite integración con diferentes frameworks.
- Surge como una alternativa a EJB's
- Actualmente es un framework completo compuesto por múltiples módulos/proyectos que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día
 - MVC
 - Negocio (donde empezó originalmente)
 - Acceso a datos

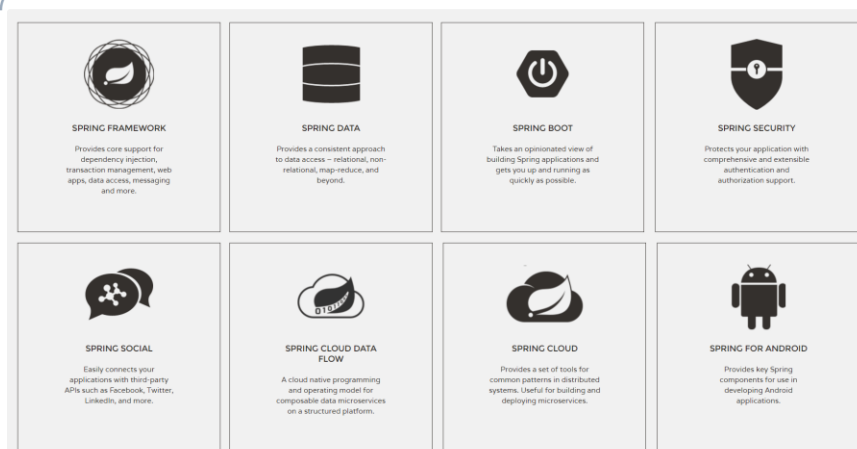
© JMA 2016. All rights reserved

Características

- **Ligero**
 - No se refiere a la cantidad de clases sino al mínimo impacto que se tiene al integrar Spring.
- **No intrusivo**
 - Generalmente los objetos que se programan no tienen dependencias de clases específicas de Spring
- **Flexible**
 - Aunque Spring provee funcionalidad para manejar las diferentes capas de la aplicación (vista, lógica de negocio, acceso a datos) no es necesario usarlo para todo. Brinda la posibilidad de utilizarlo en la capa o capas que queramos.
- **Multiplataforma**
 - Escrito en Java, corre sobre JVM

© JMA 2016. All rights reserved

Proyectos



© JMA 2016. All rights reserved

Módulos necesarios

- Spring Framework
 - Spring Core
 - Contenedor IoC (inversión de control) - inyector de dependencia
 - Spring MVC
 - Framework basado en MVC para aplicaciones web y servicios REST
- Spring Data
 - Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube
- Spring Boot
 - Simplifica el desarrollo de Spring: inicio rápido con menos codificación

© JMA 2016. All rights reserved

Spring Boot

- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.
 - Configuración: Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
 - Resolución de dependencias: Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
 - Despliegue: Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.
 - Métricas: Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Bean's creado por la aplicación, controles para el prendido y apagado, etc.
 - Extensible: Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
 - Productividad: Herramientas de productividad para desarrolladores como LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

© JMA 2016. All rights reserved

Dependencias: starters

- Los starters son un conjunto de descriptores de dependencias convenientes (versiones compatibles, ya probadas) que se pueden incluir en su aplicación.
- Se obtiene una ventanilla única para el módulo de Spring y la tecnología relacionada que se necesita, sin tener que buscar a través de códigos de ejemplo y copiar/pegar cargas de descriptores de dependencia.
- Por ejemplo, si desea comenzar a utilizar Spring con JPA para un acceso CRUD a base de datos, basta con incluir la dependencia spring-boot-starter-data-jpa en el proyecto.

© JMA 2016. All rights reserved

Spring Tool

- <https://spring.io/tools>
- Spring Tool Suite
 - IDE gratuito, personalización del Eclipse
- Plug-in para Eclipse (VSCode, Atom)
 - Help → Eclipse Marketplace ...
 - Spring Tools 4 for Spring Boot

© JMA 2016. All rights reserved

Crear proyecto

- Desde web:
 - <https://start.spring.io/>
 - Descomprimir en el workspace
 - Import → Maven → Existing Maven Project
- Desde Eclipse:
 - New Project → Sprint Boot → Spring Started Project
- Dependencias
 - Web
 - JPA
 - JDBC (o proyecto personalizado)

© JMA 2016. All rights reserved

Application

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(ApiHrApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        // Opcional: Procesar los args una vez arrancado SprintBoot
    }

}
```

© JMA 2016. All rights reserved

Configuración

- **@Configuration**: Indica que esta es una clase usada para configurar el contenedor Spring.
- **@ComponentScan**: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado, ellos son, las clases que utilizan las siguientes anotaciones: **@Component**, **@Service**, **@Controller**, **@Repository**.
- **@EnableAutoConfiguration**: Habilita la configuración automática, esta herramienta analiza el classpath y el archivo `application.properties` para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados, por ejemplo: al encontrar el motor de bases de datos H2 la aplicación se configura para utilizar este motor de datos, al encontrar Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.
- **@SpringBootApplication**: Es el equivalente a utilizar las anotaciones: **@Configuration**, **@EnableAutoConfiguration** y **@ComponentScan**

© JMA 2016. All rights reserved

Configuración

- Editar `src/main/resources/application.properties`:

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

# MySQL settings
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
logging.level.org.hibernate.SQL=debug

server.port=${PORT:8080}
```
- Repetir con `src/test/resources/application.properties`
- Eclipse: Run Configurations → Arguments → VM Arguments: `-DPORT=8888`

© JMA 2016. All rights reserved

Instalación con Docker

- Docker Toolbox
 - Windows 10 Pro ++: <https://docs.docker.com/docker-for-windows/install/>
 - Otras: <https://github.com/docker/toolbox/releases>
- Ejecutar Docker QuickStart
- Para crear el contenedor de MySQL con la base de datos Sakila:
 - `docker run -d -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 --name mysql-sakila 1maa/sakila:latest`
- Para crear el contenedor de MongoDB:
 - `docker run -d --name mongodb -p 27017:27017 mongo`
- Para crear el contenedor de Redis:
 - `docker run --name redis -p 6379:6379 -d redis`
- Para crear el contenedor de RabbitMQ:
 - `docker run -d --hostname rabbitmq --name rabbitmq -p 4369:4369 -p 5671:5671 -p 5672:5672 -p 15671:15671 -p 15672:15672 -p 25672:25672 rabbitmq:3-management`

© JMA 2016. All rights reserved

<https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm>

IOC CON SPRING CORE

© JMA 2016. All rights reserved

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto a unas técnicas de programación:
 - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
 - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

© JMA 2016. All rights reserved

Inyección de Dependencias

- Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del API Reflection de Java y crear una instancia de dicha clase inyectándole sus dependencias.

© JMA 2016. All rights reserved

Inyección

- La Inyección de Dependencias proporciona:
 - Código es más limpio
 - Desacoplamiento es más eficaz, pues los objetos no deben de conocer donde están sus dependencias ni cuales son.
 - Facilidad en las pruebas unitaria e integración

© JMA 2016. All rights reserved

Introducción

- Spring proporciona un contenedor encargado de la inyección de dependencias (Spring Core Container).
- Este contenedor nos posibilita inyectar unos objetos sobre otros.
- Para ello, los objetos deberán ser simplemente JavaBeans.
- La inyección de dependencias será bien por constructor o bien por métodos setter.
- La configuración podrá realizarse bien por anotaciones Java o mediante un fichero XML (XMLBeanFactory).
- Para la gestión de los objetos tendrá la clase (BeanFactory).
- Todos los objetos serán creados como singletons sino se especifica lo contrario.

© JMA 2016. All rights reserved

Modulo de dependencias

- Se crea el fichero de configuración applicationContext.xml y se guarda en el directorio src/META-INF.

```
<beans xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd">
  <context:component-scan base-package="es.miEspacio.ioc.services">
  </context:component-scan>
</beans>
```

© JMA 2016. All rights reserved

Beans

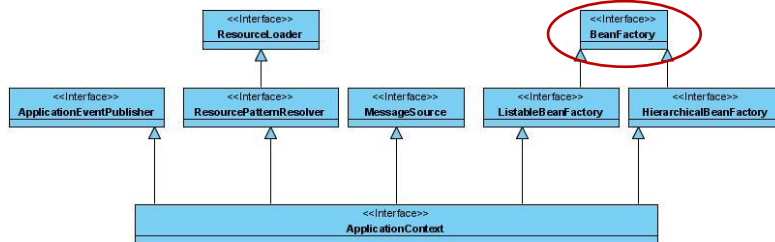
- Los beans se corresponden a los objetos reales que conforman la aplicación y que requieren ser inyectables: los objetos de la capa de servicio, los objetos de acceso a datos (DAO), los objetos de presentación (como las instancias Action de Struts), los objetos de infraestructura (como Hibernate SessionFactories, JMS Queues), etc.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  <!-- services -->
  <bean id="petStore" class="com.samples.PetStoreServiceImpl">
    <property name="accountDao" ref="accountDao"/>
    <property name="itemDao" ref="itemDao"/>
    <!-- additional collaborators and configuration for this bean go here -->
  </bean>
  <!-- more bean definitions for services go here -->
</beans>
```

© JMA 2016. All rights reserved

Bean factory

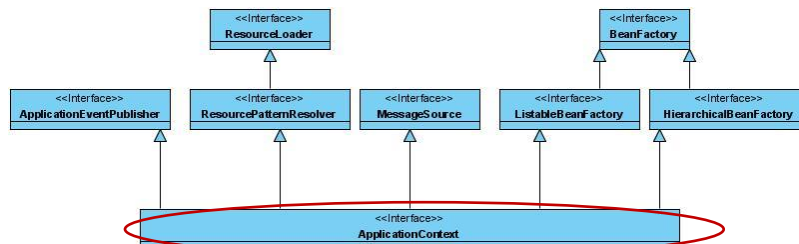
- Denominamos Bean Factory al contenedor Spring.
- Cualquier Bean Factory permite la configuración y la unión de objetos mediante la inyección de dependencia.
- Este Bean Factory también permite una gestión del ciclo de vida de los beans instanciados en él.
- Todos los contenedores Spring (Bean Factory) implementan el interface BeanFactory y algunos sub-interfaces para ampliar funcionalidades



© JMA 2016. All rights reserved

Application Context

- Spring también soporta una "fábrica de beans" algo más avanzado, llamado contexto de aplicación.
- Application Context, es una especificación de Bean Factory que implementa la interface ApplicationContext.
- En general, cualquier cosa que un Bean Factory puede hacer, un contexto de aplicación también lo puede hacer.



© JMA 2016. All rights reserved

Uso de la inyección de dependencias

- Se crea un inyector partiendo de un módulo de dependencias.
- Se solicita al inyector las instancias para que resuelva las dependencias.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("META-INF/applicationContext.xml");
    BeanFactory factory = context;
    Client client = (Client )factory.getBean("ID_Cliente");
    client.go();
}
```

- Muestra:
 - Este es un servicio...

© JMA 2016. All rights reserved

Anotaciones IoC

- | | |
|--|---|
| <ul style="list-style-type: none"> • Autodescubrimiento <ul style="list-style-type: none"> – @Component – @Repository – @Service – @Controller – @Scope • Personalización <ul style="list-style-type: none"> – @Configuration – @Bean | <ul style="list-style-type: none"> • Inyección <ul style="list-style-type: none"> – @Autowired (@Inject) – @Qualifier (@Named) – @Value – @PropertySource – @Required – @Resource • Otras <ul style="list-style-type: none"> – @PostConstruct – @PreDestroy |
|--|---|

© JMA 2016. All rights reserved

Esterotipos

- Spring define un conjunto de anotaciones core que categorizan cada uno de los componentes asociándoles una responsabilidad concreta.
 - `@Component`: Es el estereotipo general y permite anotar un bean para que Spring lo considere uno de sus objetos.
 - `@Repository`: Es el estereotipo que se encarga de dar de alta un bean para que implemente el patrón repositorio que es el encargado de almacenar datos en una base de datos o repositorio de información que se necesite. Al marcar el bean con esta anotación Spring aporta servicios transversales como conversión de tipos de excepciones.
 - `@Service`: Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y aglutina llamadas a varios repositorios de forma simultánea. Su tarea fundamental es la de agregador.
 - `@Controller`: El último de los estereotipos que es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y el aplicativo. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas.
 - `@RestController` que es una especialización de controller que contiene las anotaciones `@Controller` y `@ResponseBody` (escribe directamente en el cuerpo de la respuesta en lugar de la vista).

© JMA 2016. All rights reserved

Alcance

- Un aspecto importante del ciclo de vida de los Beans es si el contenedor creará una única instancia o tantas como ámbitos sean necesarios.
 - `prototype`: No reutiliza instancias, genera siempre una nueva instancia. `@Scope("prototype")`
 - `singleton`: (Predeterminado) Instancia única para todo el contenedor Spring IoC. `@Scope("singleton") @Singleton`
 - Adicionalmente, en el contexto de un Spring Web Application Context: `@RequestScope @SessionScope @ApplicationScope`
 - `request`: Instancia única para el ciclo de vida de una sola solicitud HTTP. Cada solicitud HTTP tiene su propia instancia única.
 - `session`: Instancia única para el ciclo de vida de cada HTTP Session.
 - `application`: Instancia única para el ciclo de vida de un ServletContext.
 - `websocket`: Instancia única para el ciclo de vida de un WebSocket.

© JMA 2016. All rights reserved

Inyección

- La inyección se realiza con la anotación `@Autowired`:
 - En atributos:


```
@Autowired
private MyBeans myBeans;
```
 - En propiedades (setter):


```
@Autowired
public void setMyBeans(MyBeans value) { ... }
```
 - En constructores
- Por defecto la inyección es obligatoria, se puede marcar como opcional en cuyo caso si no encuentra el Bean inyectará un null.


```
@Autowired(required=false) private MyBeans myBeans;
```
- Se puede completar `@Autowired` con la anotación `@Lazy` para inyectar un proxy de resolución lenta.

© JMA 2016. All rights reserved

Inyección

- Con `@Qualifier` (`@Named`) se pueden agrupar o cualificar los beans asociándoles un nombre:


```
public interface MyInterface { ... }
```

```
@Component
@Qualifier("old")
public class MyInterfaceImpl implements MyInterface { ... }
```

```
@Qualifier("new")
public class MyNewInterfaceImpl implements MyInterface { ... }
```

```
@Autowired(required=false)
@Qualifier("new")
private MyInterface srv;
```

© JMA 2016. All rights reserved

Acceso a ficheros de propiedades

- Localización (fichero .properties, .yaml, .xml):
 - Por defecto: src/main/resources/application.properties
 - En la carpeta de recursos src/main/resources:
@PropertySource("classpath:my.properties")
 - En un fichero local:
@PropertySource("file://c:/cng/my.properties")
 - En una URL:
@PropertySource("http://myserver/application.properties")
- Acceso directo:
@Value("\${spring.datasource.username}") private String name;
- Acceso a través del entorno:
@Autowired private Environment env;
env.getProperty("spring.datasource.username")

© JMA 2016. All rights reserved

Ciclo de Vida

- Con la inyección el proceso de creación y destrucción de las instancias de los beans es administrada por el contenedor.
- Para poder intervenir en el ciclo para controlar la creación y destrucción de las instancias se puede:
 - Implementar las interfaces InitializingBean y DisposableBean de devoluciones de llamada
 - Sobrescribir los métodos init() y destroy()
 - Anotar los métodos con @PostConstruct y @PreDestroy.
- Se pueden combinar estos mecanismos para controlar un bean dado.

© JMA 2016. All rights reserved

Configuración por código

- Hay que crear una (o varias) clase anotada con `@Configuration` que contendrá un método por cada clase/interfaz (sin estereotipo) que se quiera tratar como un Bean inyectable.
- El método ira anotado con `@Bean`, se debería llamar como la clase en notación Camel y devolver del tipo de la clase la instancia ya creada. Adicionalmente se puede anotar con `@Scope` y con `@Qualifier`.

```
public class MyBean { ... }

@Configuration
public class MyConfig {
    @Bean
    @Scope("prototype")
    public MyBean myBean() { ... }
```

© JMA 2016. All rights reserved

Doble herencia

- Se crea el interfaz con la funcionalidad deseada:

```
public interface Service {
    public void go();
}
```

- Se implementa la interfaz en una clase (por convenio se usa el sufijo `Impl`):

```
import org.springframework.stereotype.Service;
@Service
@Singleton
public class ServiceImpl implements Service {
    public void go() {
        System.out.println("Este es un servicio...");
    }
}
```

© JMA 2016. All rights reserved

Cliente

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("ID_Cliente")
public class Client {
    private final Service service;

    @Autowired
    public void setService(Service service){
        this.service = service;
    }

    public void go(){
        service.go();
    }
}

@Autowired establece que deben resolverse los parámetros mediante DI.
```

© JMA 2016. All rights reserved

Anotaciones estándar JSR-330

- A partir de Spring 3.0, Spring ofrece soporte para las anotaciones estándar JSR-330 (inyección de dependencia). Esas anotaciones se escanean de la misma manera que las anotaciones de Spring.
- Cuando trabaje con anotaciones estándar, hay que tener en cuenta que algunas características importantes no están disponibles.

Anotaciones Spring	Anotaciones Estándar (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

© JMA 2016. All rights reserved

<https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc>

SPRING FRAMEWORK

© JMA 2016. All rights reserved

Introducción

- Spring Framework se encarga de todos los detalles de bajo nivel que pueden hacer de JDBC una API tan tediosa.
- Se puede elegir entre varios enfoques para formar la base del acceso JDBC a la base de datos o combinar varios.
 - JdbcTemplate: es el enfoque clásico y más popular de Spring JDBC. Este es el enfoque de "nivel más bajo" y todos los demás utilizan un JdbcTemplate bajo cubierta.
 - NamedParameterJdbcTemplate: envuelve a JdbcTemplate para proporcionar parámetros con nombre en lugar de los tradicionales ? Como marcadores de posición de JDBC . Este enfoque proporciona una mejor documentación y facilidad de uso cuando se tienen varios parámetros para una declaración SQL.
 - SimpleJdbcInsert y SimpleJdbcCall: optimizar los metadatos de la base de datos para limitar la cantidad de configuración necesaria. Este enfoque simplifica la codificación, basta con proporcionar el nombre de la tabla o procedimiento y un mapa de parámetros que coincidan con los nombres de las columnas. Esto solo funciona si la base de datos proporciona los metadatos adecuados, en caso contrario, se debe proporcionar una configuración explícita de los parámetros.
 - Los objetos RDBMS, incluidos MappingSqlQuery, SqlUpdate y StoredProcedure, requieren que se creen objetos reutilizables y seguros para subprocesos durante la inicialización de la capa de acceso a datos. Este enfoque se basa en JDO Query, en el que define su cadena de consulta, declara parámetros y compila la consulta. Una vez hecho esto, los métodos execute(...), update(...), y findObject(...) se pueden llamar varias veces con diferentes valores en los parámetros.

© JMA 2016. All rights reserved

Conexiones a la base de datos

- Spring obtiene una conexión a la base de datos a través de un DataSource. El DataSource es parte de la especificación JDBC y es una fábrica de conexiones generalizada. Permite que un contenedor o un marco oculten la agrupación de conexiones y los problemas de gestión de transacciones del código de la aplicación.

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/sakila"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

- Con Sprint Boot, en application.properties:
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root

© JMA 2016. All rights reserved

Conexiones a la base de datos

```
@Configurable
public class AppConfig {
    @Bean(destroyMethod = "close")
    DataSource dataSource(Environment env) {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/sakila");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }

    @Bean
    JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    DataSourceTransactionManager dataSourceTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

© JMA 2016. All rights reserved

JdbcTemplate

- JdbcTemplate es la clase central del paquete principal de JDBC.
- Maneja la creación y liberación de recursos, lo que le ayuda a evitar errores comunes, como olvidar cerrar la conexión.
- Realiza las tareas básicas del flujo de trabajo principal de JDBC (como la creación y ejecución de declaraciones), dejando que el código de la aplicación proporcione SQL y extraiga los resultados.
- La clase JdbcTemplate:
 - Ejecuta consultas SQL
 - Actualiza declaraciones y llamadas a procedimientos almacenados
 - Realiza iteraciones sobre las instancias ResultSet y la extracción de valores de devueltos.
 - Detecta las excepciones de JDBC y las traduce a la jerarquía de excepciones genérica y más informativa definida en el paquete `org.springframework.dao`.

© JMA 2016. All rights reserved

Consultas

- La siguiente consulta obtiene un valor escalar:


```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor",
Integer.class);
```
- La siguiente consulta usa una consulta parametrizada:


```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
    "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```
- Para objetos complejos se requiere un mapeador:


```
Actor actor = jdbcTemplate.queryForObject(
    "select first_name, last_name from t_actor where id = ?",
    (resultSet, rowNum) -> {
        Actor newActor = new Actor();
        newActor.setFirstName(resultSet.getString("first_name"));
        newActor.setLastName(resultSet.getString("last_name"));
        return newActor;
    }, 1L);
```

© JMA 2016. All rights reserved

Consultas

- Si el mapeador se reutiliza, se crea como una clase que implementa :

```
class ActorRowMapper implements RowMapper<Actor> {
    @Override
    public Actor mapRow(ResultSet resultSet, int rowNum) throws SQLException {
        Actor newActor = new Actor();
        newActor.setFirstName(resultSet.getString("first_name"));
        newActor.setLastName(resultSet.getString("last_name"));
        return newActor;
    }
}
```
- La siguiente consulta obtiene una lista de entidades:

```
List<Actor> actors = this.jdbcTemplate.query(
    "select first_name, last_name from t_actor",
    new ActorRowMapper());
```
- Existen versiones especializadas: `queryForList`, `queryForMap`, `queryForRowSet` o `queryForStream`.

© JMA 2016. All rights reserved

Modificaciones

- Se puede utilizar el método `update(..)` para realizar operaciones de inserción, actualización y eliminación. Los valores de los parámetros se proporcionan normalmente como argumentos variables o, alternativamente, como una matriz de objetos.

```
jdbcTemplate.update(
    "insert into t_actor (first_name, last_name) values (?, ?)",
    "Leonor", "Watling");
jdbcTemplate.update(
    "update t_actor set last_name = ? where id = ?",
    "Banjo", 5276L);
jdbcTemplate.update(
    "delete from t_actor where id = ?", 5276L);
```

© JMA 2016. All rights reserved

Otras operaciones

- El método `execute(..)` se utiliza para ejecutar cualquier SQL arbitrario. En consecuencia, el método se usa a menudo para declaraciones DDL. Está muy sobrecargado con variantes que aceptan interfaces de devolución de llamada, vinculando matrices de variables, etc.

```
jdbcTemplate.execute("create table mytable (id integer,
name varchar(100))");
```

- Para invoca un procedimiento almacenado:

```
jdbcTemplate.update(
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
    Long.valueOf(unionId));
```

© JMA 2016. All rights reserved

NamedParameterJdbcTemplate

- La clase `NamedParameterJdbcTemplate` agrega soporte para programar declaraciones JDBC usando parámetros con nombre, en vez de programar declaraciones JDBC usando el clásico comodín '?' como marcador de posición de argumentos.

```
String sql = "select count(*) from T_ACTOR where first_name = :first_name";
int count = this.namedParameterJdbcTemplate.queryForObject(sql, new
    MapSqlParameterSource("first_name", firstName), Integer.class);
```

- Alternativamente, se pueden pasar los parámetros en un Map con los nombre y sus valores correspondientes:

```
int count = this.namedParameterJdbcTemplate.queryForObject(sql,
    Collections.singletonMap("first_name", firstName), Integer.class);
```

- Otra opción es `BeanPropertySqlParameterSource` clase, que envuelve un `JavaBean` arbitrario y utiliza las propiedades del `JavaBean` envuelto como fuente de valores de parámetros con nombre.

```
int count = this.namedParameterJdbcTemplate.queryForObject(sql, new
    BeanPropertySqlParameterSource(actor), Integer.class);
```

© JMA 2016. All rights reserved

SimpleJdbcInsert

- La clase SimpleJdbcInsert proporciona una configuración simplificada para las inserciones al aprovechar los metadatos de la base de datos que se pueden recuperar a través del controlador JDBC.

```
SimpleJdbcInsert insertActor = new
    SimpleJdbcInsert(dataSource).withTableName("t_actor");
Map<String, Object> parameters = new HashMap<String,
    Object>(3);
parameters.put("id", actor.getId());
parameters.put("first_name", actor.getFirstName());
parameters.put("last_name", actor.getLastName());
insertActor.execute(parameters);
```

© JMA 2016. All rights reserved

SimpleJdbcInsert

- Se puede BeanPropertySqlParameterSource utilizar para obtener los parámetros.
- Se puede limitar las columnas para una inserción especificando una lista de nombres de columna con el método usingColumns:

```
insertActor = new SimpleJdbcInsert(dataSource)
    .withTableName("t_actor")
    .usingColumns("first_name", "last_name")
    .usingGeneratedKeyColumns("id");
Map<String, Object> parameters = new HashMap<String, Object>(2);
parameters.put("first_name", actor.getFirstName());
parameters.put("last_name", actor.getLastName());
Number newId = insertActor.executeAndReturnKey(parameters);
actor.setId(newId.longValue());
```

© JMA 2016. All rights reserved

SimpleJdbcCall

- La clase SimpleJdbcCall usa los metadatos de la base de datos para buscar los parámetros in y out de un procedimiento almacenado para que no se tenga que declararlos explícitamente.

```
SimpleJdbcCall procReadActor = new SimpleJdbcCall(dataSource)
    .withProcedureName("read_actor");
SqlParameterSource in = new MapSqlParameterSource()
    .addValue("in_id", id);
Map out = procReadActor.execute(in);
Actor actor = new Actor();
actor.setId(id);
actor.setFirstName((String) out.get("out_first_name"));
actor.setLastName((String) out.get("out_last_name"));
```

© JMA 2016. All rights reserved

GESTIÓN DE DATOS

© JMA 2016. All rights reserved

Spring Data

- Spring Framework ya proporcionaba soporte para JDBC, Hibernate, JPA o JDO, simplificando la implementación de la capa de acceso a datos, unificando la configuración y creando una jerarquía de excepciones común para todas ellas.
- Spring Data es un proyecto (subproyectos) de SpringSource cuyo propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.
- Spring Data viene a cubrir el soporte necesario para distintas tecnologías de bases de datos NoSQL integrándolas con las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.
- Con cada tipo de tecnología de persistencia, los DAOs (Data Access Objects) ofrecen las funcionalidades típicas de CRUD para objetos de dominio propios, métodos de búsqueda, ordenación y paginación. Spring Data proporciona interfaces genéricas para estos aspectos (CrudRepository, PagingAndSortingRepository) e implementaciones específicas para cada tipo de tecnología de persistencia.

© JMA 2016. All rights reserved

Entidades

- Las entidades encapsulan los conceptos del negocio. Una entidad puede ser un objeto con métodos o puede ser un conjunto de funciones y estructuras de datos. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.
- Estas entidades son los objetos de dominio de la aplicación. Encapsulan las reglas más generales y de alto nivel.
- Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, no esperaríamos que estos objetos se vieran afectados por un cambio en la navegación de la página o la seguridad.
- Ningún cambio operativo en una aplicación en particular debería afectar la capa de la entidad.

© JMA 2016. All rights reserved

Entidades de Dominio

- Una entidad es cualquier objeto del dominio que mantiene un estado y comportamiento más allá de la ejecución de la aplicación y que necesita ser distinguido de otro que tenga las mismas propiedades y comportamientos.
- Es un tipo de clase dedicada a representar un modelo de dominio persistente que:
 - Debe ser publica (no puede ser estar anidada ni final o tener miembros finales)
 - Deben tener un constructor público sin ningún tipo de argumentos.
 - Para cada propiedad que queramos persistir debe haber un método get/set asociado.
 - Debe tener una clave primaria
 - Debería sobrescribir los métodos equals y hashCode
 - Debería implementar el interfaz Serializable para utilizar de forma remota

© JMA 2016. All rights reserved

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de entidades asociadas que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
 - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
 - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

© JMA 2016. All rights reserved

Anotaciones JPA

Anotación	Descripción
@Entity	<ul style="list-style-type: none"> - Se aplica a la clase. - Indica que esta clase Java es una entidad a persistir.
@Table(name="Tabla")	<ul style="list-style-type: none"> - Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. - Es opcional si el nombre de la clase coincide con el de la tabla.
@Id	<ul style="list-style-type: none"> - Se aplica a una propiedad Java e indica que este atributo es la clave primaria.
@Column(name="Id")	<ul style="list-style-type: none"> - Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. - Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos.
@Column(...)	<ul style="list-style-type: none"> - name: nombre - length: longitud - precision: número total de dígitos - scale: número de dígitos decimales - unique: restricción valor único - nullable: restricción valor obligatorio - insertable: es insertable - updatable: es modificable
@Transient	<ul style="list-style-type: none"> - Se aplica a una propiedad Java e indica que este atributo no es persistente

© JMA 2016. All rights reserved

Asociaciones

- **Uno a uno (Unidireccional)**
 - En la entidad fuerte se anota la propiedad con la referencia de la entidad.
 - **@OneToOne(cascade=CascadeType.ALL):**
 - Esta anotación indica la relación uno a uno de las 2 tablas.
 - **@PrimaryKeyJoinColumn:**
 - Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.
- **Uno a uno (Bidireccional)**
 - Las dos entidades cuentan con una propiedad con la referencia a la otra entidad.

© JMA 2016. All rights reserved

Asociaciones

- Uno a Muchos
 - En Uno
 - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos:
 - List: Ordenada con repetidos
 - Set: Desordenada sin repetidos
 - @OneToMany(mappedBy="propEnMuchos", cascade= CascadeType.ALL)
 - mappedBy: contendrá el nombre de la propiedad en la entidad muchos con la referencia a la entidad uno.
 - @IndexColumn (name="idx")
 - Opcional. Nombre de la columna que en la tabla muchos para el orden dentro de la Lista.
 - En Muchos
 - Dispone de una propiedad con la referencia de la entidad uno.
 - @ManyToOne
 - Esta anotación indica la relación de Muchos a uno
 - @JoinColumn (name="idFK")
 - Indicaremos el nombre de la columna que en la tabla muchos contiene la clave ajena a la tabla uno.

© JMA 2016. All rights reserved

Asociaciones

- Muchos a muchos (Unidireccional)
 - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
 - @ManyToMany(cascade=CascadeType.ALL):
 - Esta anotación indica la relación muchos a muchos de las 2 tablas.
- Muchos a muchos (Bidireccional)
 - La segunda entidad también dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
 - @ManyToMany(mappedBy="propEnOtroMuchos"):
 - mappedBy: Propiedad con la colección en la otra entidad para preservar la sincronización entre ambos lados

© JMA 2016. All rights reserved

Cascada

- El atributo cascade se utiliza en los mapeos de las asociaciones para indicar cuando se debe propagar la acción en una instancia hacia la instancias relacionadas mediante la asociación.
- Enumeración de tipo CascadeType:
 - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
 - DETACH (Separar)
 - MERGE (Modificar)
 - PERSIST (Crear)
 - REFRESH (Releer)
 - REMOVE (Borrar)
 - NONE
- Acepta múltiples valores:
 - @OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})

© JMA 2016. All rights reserved

Mapeo de Herencia

- Tabla por jerarquía de clases
 - Padre:
 - @Table("Account")
 - @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
 - @DiscriminatorColumn(name="PAYMENT_TYPE")
 - Hija:
 - @DiscriminatorValue(value = "Debit")
- Tabla por subclases
 - Padre:
 - @Table("Account")
 - @Inheritance(strategy = InheritanceType.JOINED)
 - Hija:
 - @Table("DebitAccount")
 - @PrimaryKeyJoinColumn(name = "account_id")
- Tabla por clase concreta
 - Padre:
 - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
 - Hija:
 - @Table("DebitAccount")

© JMA 2016. All rights reserved

Repositorio

- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

© JMA 2016. All rights reserved

Repositorio

- Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para las entidades se simplifica porque solo vamos a necesitar un interfaz que extienda uno de los siguientes interfaces:
 - `CrudRepository<T,ID>`
 - `count`, `delete`, `deleteAll`, `deleteById`, `existsById`, `findAll`, `findAllById`, `findById`, `save`, `saveAll`
 - `PagingAndSortingRepository<T,ID>`
 - `findAll(Pageable pageable)`, `findAll(Sort sort)`
 - `JpaRepository<T,ID>`
 - `deleteAllInBatch`, `deleteInBatch`, `flush`, `getOne`, `saveAll`, `saveAndFlush`
 - `MongoRepository<T,ID>`
 - `findAll`, `insert`, `saveAll`
- En el proceso de inyección Spring implementa la interfaz antes de inyectarla:


```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {}
@Autowired
private ProfesorRepository repository;
```

© JMA 2016. All rights reserved

Repositorio

- El interfaz puede ser ampliado con nuevos métodos que serán implementados por Spring:
 - Derivando la consulta del nombre del método directamente.
 - Mediante el uso de una consulta definida manualmente.
- La implementación se realizará mediante la decodificación del nombre del método, dispone de una sintaxis específica para crear dichos nombre:

```
List<Profesor> findByNombreStartingWiths(String nombre);
List<Profesor> findByApellido1AndApellido2OrderByEdadDesc( String
    apellido1, String apellido2);
List<Profesor> findByTipoIn(Collection<Integer> tipos);

int deleteByEdadGreaterThan(int valor);
```

© JMA 2016. All rights reserved

Repositorio

- Prefijo consulta derivada:
 - find (read, query, get), count, delete
- Opcionalmente, limitar los resultados de la consulta:
 - Distinct, TopNumFilas y FirstNumFilas
- Expresión de propiedad: *ByPropiedad*
 - Operador (Between, LessThan, GreaterThan, Like, ...) por defecto equal.
 - Se pueden concatenar varias con And y Or
 - Opcionalmente admite el indicador IgnoreCase y AllIgnoreCase.
- Opcionalmente, *OrderByPropiedadAsc* para ordenar,
 - se puede sustituir Asc por Desc, admite varias expresiones de ordenación.
- Parámetros:
 - un parámetro por cada operador que requiera valor y debe ser del tipo apropiado
- Parámetros opcionales:
 - Sort → Sort.by("nombre", "apellidos").descending()
 - Pageable → PageRequest.of(0, 10, Sort.by("nombre"))

© JMA 2016. All rights reserved

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname, findByFirstnames, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	... where x.age >= ?1

© JMA 2016. All rights reserved

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parámetro enlazado con % anexado)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1(parámetro enlazado con % antepuesto)

© JMA 2016. All rights reserved

Repositorio

P. Clave	Muestra	Fragmento de JPQL
Containing	findByFirstnameContaining	... where x.firstname like ?1(parámetro enlazado entre %)
OrderBy	findByAgeOrderByLastNameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastNameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

© JMA 2016. All rights reserved

Repositorio

- Valor de retorno de consultas síncronas:
 - find, read, query, get:
 - List<Entidad>
 - Stream<Entidad>
 - Optional<T>
 - count, delete:
 - long
- Valor de retorno de consultas asíncronas (deben ir anotadas con @Async):
 - Future<Entidad>
 - CompletableFuture<Entidad>
 - ListenableFuture<Entidad>

© JMA 2016. All rights reserved

Repositorio

- Mediante consultas JPQL:


```
@Query("from Profesor p where p.edad > 67")
List<Profesor> findJubilados();

@Modifying
@Query("delete from Profesor p where p.edad > 67")
List<Profesor> deleteJubilados();
```
- Mediante consultas SQL nativas:


```
@Query("select * from Profesores p where p.edad between
?1 and ?2", nativeQuery=true)
List<Profesor> findActivos(int inicial, int final);
```

© JMA 2016. All rights reserved

Transacciones

- Por defecto, los métodos CRUD en las instancias del repositorio son transaccionales. Para las operaciones de lectura, el indicador `readOnly` de configuración de transacción se establece en `true` para optimizar el proceso. Todos los demás se configuran con un plano `@Transactional` para que se aplique la configuración de transacción predeterminada.
- Cuando se van a realizar varias llamadas al repositorio o a varios repositorios se puede anotar con `@Transactional` el método para que todas las operaciones se encuentren dentro de la misma transacción.


```
@Transactional
public void create(Pago pago) { ... }
```
- Para que los métodos de consulta sean transaccionales:


```
@Override
@Transactional(readOnly = false)
public List<User> findAll();
```

© JMA 2016. All rights reserved

Validaciones

- Desde la versión 3, Spring ha simplificado y potenciado en gran medida la validación de datos, gracias a la adopción de la especificación JSR 303. Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Esto nos facilita la validación de los datos enviados antes de llegar al controlador REST. (**Dependencia: Starter I/O > Validation**)
- Las anotaciones se pueden establecer a nivel de clase, atributo y parámetro de método.
- Se puede exigir la validez mediante la anotación `@Valid` en el elemento a validar.

```
public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
```
- Para realizar la validación manualmente:

```
@Autowired // Validation.buildDefaultValidatorFactory().getValidator()
private Validator validator;
Set<ConstraintViolation<Persona>> constraintViolations = validator.validate(
    persona );
Set<ConstraintViolation<Persona>> constraintViolations =
    validator.validateProperty( persona, "nombre" );
```

© JMA 2016. All rights reserved

Validaciones (JSR 303)

- `@Null` : Comprueba que el valor anotado es null
- `@NotNull` : Comprueba que el valor anotado no sea null
- `@NotEmpty` : Comprueba si el elemento anotado no es nulo ni está vacío
- `@NotBlank` : Comprueba que la secuencia de caracteres anotados no sea nula y que la longitud recortada sea mayor que 0. La diferencia `@NotEmpty` es que esta restricción solo se puede aplicar en secuencias de caracteres y que los espacios en blanco finales se ignoran.
- `@AssertFalse` : Comprueba que el elemento anotado es falso.
- `@AssertTrue` : Comprueba que el elemento anotado es verdadero

© JMA 2016. All rights reserved

Validaciones (JSR 303)

- @Max(value=) : Comprueba si el valor anotado es menor o igual que el máximo especificado
- @Min(value=) : Comprueba si el valor anotado es mayor o igual que el mínimo especificado
- @Negative : Comprueba si el elemento es estrictamente negativo. Los valores cero se consideran inválidos.
- @NegativeOrZero : Comprueba si el elemento es negativo o cero.
- @Positive : Comprueba si el elemento es estrictamente positivo. Los valores cero se consideran inválidos.
- @PositiveOrZero : Comprueba si el elemento es positivo o cero.
- @DecimalMax(value=, inclusive=) : Comprueba si el valor numérico anotado es menor que el máximo especificado, cuando inclusive= falso. De lo contrario, si el valor es menor o igual al máximo especificado.
- @DecimalMin(value=, inclusive=) : Comprueba si el valor anotado es mayor que el mínimo especificado, cuando inclusive= falso. De lo contrario, si el valor es mayor o igual al mínimo especificado.

© JMA 2016. All rights reserved

Validaciones (JSR 303)

- @Digits: El elemento anotado debe ser un número cuyo valor tenga el número de dígitos especificado.
- @Past : Comprueba si la fecha anotada está en el pasado
- @PastOrPresent : Comprueba si la fecha anotada está en el pasado o en el presente
- @Future : Comprueba si la fecha anotada está en el futuro.
- @FutureOrPresent : Comprueba si la fecha anotada está en el presente o en el futuro
- @Email : Comprueba si la secuencia de caracteres especificada es una dirección de correo electrónico válida.
- @Pattern(regex=, flags=) : Comprueba si la cadena anotada coincide con la expresión regular regex considerando la bandera dadamatch.
- @Size(min=, max=) : Comprueba si el tamaño del elemento anotado está entre min y max (inclusive)

© JMA 2016. All rights reserved

Validaciones (Hibernate)

- @CreditCardNumber(ignoreNonDigitCharacters=): Comprueba que la secuencia de caracteres pasa la prueba de suma de comprobación de Luhn.
- @Currency(value=): Comprueba que la unidad monetaria de un `javax.money.MonetaryAmount` forma parte de las unidades monetarias especificadas.
- @DurationMax(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento `java.time.Duration` no sea mayor que el construido a partir de los parámetros de la anotación.
- @DurationMin(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento `java.time.Duration` no sea menor que el construido a partir de los parámetros de anotación.
- @EAN: Comprueba que la secuencia de caracteres sea un código de barras EAN válido
- @ISBN: Comprueba que la secuencia de caracteres sea un ISBN válido.

© JMA 2016. All rights reserved

Validaciones (Hibernate)

- @Length(min=, max=): Valida que la secuencia de caracteres esté entre min e max incluidos
- @CodePointLength(min=, max=, normalizationStrategy=): Valida que la longitud del punto de código de la secuencia de caracteres esté entre min e max incluidos.
- @LuhnCheck(startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos de la secuencia de caracteres pasan el algoritmo de suma de comprobación de Luhn.
- @Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo genérico de suma de comprobación mod 10.
- @Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo de suma de comprobación mod 11.

© JMA 2016. All rights reserved

Validaciones (Hibernate)

- @Normalized(form=)**: Valida que la secuencia de caracteres anotados se normalice de acuerdo con lo dado form.
- @Range(min=, max=)**: Comprueba si el valor anotado se encuentra entre (inclusive) el mínimo y el máximo especificados
- @SafeHtml(additionalTags=, additionalTagsWithAttributes=, baseURI=, whitelistType=)**: Comprueba si el valor anotado contiene fragmentos potencialmente maliciosos como `<script>`.
- @ScriptAssert(lang=, script=, alias=, reportOn=)**: Comprueba si la secuencia de comandos proporcionada se puede evaluar correctamente con el elemento anotado.
- @UniqueElements**: Comprueba que la colección solo contiene elementos únicos.
- @URL(protocol=, host=, port=, regexp=, flags=)**: Comprueba si la secuencia de caracteres anotada es una URL válida según RFC2396.

© JMA 2016. All rights reserved

Validaciones personalizadas

- Anotación personalizada para la validación:


```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = NifValidator.class)
@Documented
public @interface NIF {
    String message() default "{validation.NIF.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

© JMA 2016. All rights reserved

Validaciones personalizadas

- Clase del validador


```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class NifValidator implements ConstraintValidator<NIF, String> {
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if(value == null) return true;
        value = value.toUpperCase();
        if(!value.matches("^\\d{1,8}[A-Z]$")) return false;
        return "TRWAGMYFPDXBNJZSQVHLCKE".charAt(Integer.parseInt(
            value.substring(0, value.length() - 1)) % 23)
            == value.charAt(value.length() - 1);
    }
}
```
- En el fichero ValidationMessages.properties


```
validation.NIF.message=${validatedValue} no es un NIF válido.
```

© JMA 2016. All rights reserved

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
 - Desacoplar del nivel de servicio de la capa de base de datos.
 - Quitar las referencias circulares.
 - Ocultar determinadas propiedades que los clientes no deberían ver.
 - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
 - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
 - Evitar el "exceso" y las vulnerabilidades por publicación.

© JMA 2016. All rights reserved

Lombok

<https://projectlombok.org/>

- En las clases Java hay mucho código que se repite una y otra vez: constructores, equals, getters y setters. Métodos que quedan definidos una vez que dicha clase ha concretado sus propiedades, y que salvo ajustes menores, serán siempre sota, caballo y rey.
- Project Lombok es una biblioteca de java que se conecta automáticamente al editor y crea herramientas que automatizan la escritura de java.
- Mediante simples anotaciones ya nunca mas vuelves a escribir otro método get o equals.

```
@Data @AllArgsConstructor @NoArgsConstructor public class MyDTO {
    private long id;
    private String name;
}
```
- La anotación @Value (no confundir con la de Spring) crea la versión de solo lectura.
- Es necesario agregar las bibliotecas al proyecto y configurar el entorno.

© JMA 2016. All rights reserved

ModelMapper

<http://modelmapper.org/>

- Las aplicaciones a menudo contienen modelos de objetos similares pero diferentes, donde los datos en dos modelos pueden ser similares pero la estructura y las responsabilidades de los modelos son diferentes. El mapeo de objetos facilita la conversión de un modelo a otro, permitiendo que los modelos separados permanezcan segregados.
- ModelMapper facilita el mapeo de objetos, al determinar automáticamente cómo se mapea un modelo de objeto a otro, de acuerdo con las convenciones, de la misma forma que lo haría un ser humano, al tiempo que proporciona una API simple y segura de refactorización para manejar casos de uso específicos.

```
ModelMapper modelMapper = new ModelMapper();
OrderDTO orderDTO = modelMapper.map(order, OrderDTO.class);
```

© JMA 2016. All rights reserved

Proyecciones

- Los métodos de consulta de Spring Data generalmente devuelven una o varias instancias de la raíz agregada administrada por el repositorio. Sin embargo, a veces puede ser conveniente crear proyecciones basadas en ciertos atributos de esos tipos. Spring Data permite modelar tipos de retorno dedicados, para recuperar de forma más selectiva vistas parciales de los agregados administrados.
- La forma más sencilla de limitar el resultado de las consultas solo a los atributos deseados es declarar una interfaz o DTO que exponga los métodos de acceso para las propiedades a leer, que deben coincidir exactamente con las propiedades de la entidad:

```
public interface NamesOnly {
    String getNombre();
    String getApellidos();
}
```

- El motor de ejecución de consultas crea instancias de proxy de esa interfaz en tiempo de ejecución para cada elemento devuelto y reenvía las llamadas a los métodos expuestos al objeto de destino.

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
    List<NamesOnly> findByNombreStartingWith(String nombre);
}
```

© JMA 2016. All rights reserved

Proyecciones

- Las proyecciones se pueden usar recursivamente.

```
interface PersonSummary {
    String getNombre();
    String getApellidos();
    DireccionSummary getDireccion();
    interface DireccionSummary {
        String getCiudad();
    }
}
```

- En las proyecciones abiertas, los métodos de acceso en las interfaces de proyección también se pueden usar para calcular nuevos valores:

```
public interface NamesOnly {
    @Value("#{args[0] + ' ' + target.nombre + ' ' + target.apellidos}")
    String getNombreCompleto(String tratamiento);
    default String getFullName() {
        return getNombre.concat(" ").concat(getApellidos());
    }
}
```

© JMA 2016. All rights reserved

Proyecciones

- Se puede implementar una lógica personalizada mas compleja en un bean de Spring y luego invocarla desde la expresión SpEL:

```
@Component
class MyBean {
    String getFullName(Person person) { ... }
}
interface NamesOnly {
    @Value("#{@myBean.getFullName(target)}")
    String getFullName();
    ...
}
```

- Las proyecciones dinámicas permiten utilizar genéricos en la definición del repositorio para resolver el tipo de devuelto en el momento de la invocación:


```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
    <T> List<T> findByNombreStartingWith(String prefijo, Class<T> type);
}
dao.findByNombreStartingWith("J", ProfesorShortDTO.class)
```

© JMA 2016. All rights reserved

Serialización Jackson

- Jackson es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y des serializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos-JSON.
- Jackson es bastante “inteligente” y sin decirle nada es capaz de serializar y des serializar bastante bien los objetos. Para ello usa básicamente la reflexión de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la des serialización buscará un método “setName(String s)”.


```
ObjectMapper objectMapper = new ObjectMapper();
String jsonText = objectMapper.writeValueAsString(person);
Person person = new ObjectMapper().readValue(jsonText, Person.class);
```
- El proceso de serialización y des serialización se puede controlar declarativamente mediante anotaciones:

<https://github.com/FasterXML/jackson-annotations>

© JMA 2016. All rights reserved

Serialización Jackson

- **@JsonProperty**: indica el nombre alternativo de la propiedad en JSON.

```
@JsonProperty("name") public String getTheName() { ... }
@JsonProperty("name") public void setTheName(String name) { ... }
}
```
- **@JsonFormat**: especifica un formato para serializar los valores de fecha/hora.

```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy hh:mm:ss")
public Date eventDate;
```
- **@JsonIgnore**: marca que se ignore una propiedad (nivel miembro).

```
@JsonIgnore public int id;
```

© JMA 2016. All rights reserved

Serialización Jackson

- **@JsonIgnoreProperties**: marca que se ignore una o varias propiedades (nivel clase).

```
@JsonIgnoreProperties({ "id", "ownerName" })
@JsonIgnoreProperties(ignoreUnknown=true)
public class Item {
```
- **@JsonInclude**: se usa para incluir propiedades con valores vacíos/nulos/ predeterminados.

```
@JsonInclude(Include.NON_NULL)
public class Item {
```
- **@JsonAutoDetect**: se usa para anular la semántica predeterminada qué propiedades son visibles y cuáles no.

```
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
public class Item {
```

© JMA 2016. All rights reserved

Serialización Jackson

- **@JsonView**: permite indicar la Vista en la que se incluirá la propiedad para la serialización / deserialización.

```
public class Views {
    public static class Partial {}
    public static class Complete extends Partial {}
}

public class Item {
    @JsonView(Views.Partial.class)
    public int id;
    @JsonView(Views.Partial.class)
    public String itemName;
    @JsonView(Views.Complete.class)
    public String ownerName;
}
```

```
String result = new ObjectMapper().writerWithView(Views.Partial.class)
    .writeValueAsString(item);
```

© JMA 2016. All rights reserved

Serialización Jackson

- **@JsonFilter**: indica el filtro que se utilizará durante la serialización (es obligatorio suministrarlo).

```
@JsonFilter("ItemFilter")
public class Item {
    public int id;
    public String itemName;
    public String ownerName;
}
```

```
FilterProvider filters = new SimpleFilterProvider().addFilter("ItemFilter",
    SimpleBeanPropertyFilter.filterOutAllExcept("id", "itemName"));
MappingJacksonValue mapping = new
    MappingJacksonValue(dao.findAll());
mapping.setFilters(filters);
return mapping;
```

© JMA 2016. All rights reserved

Serialización Jackson

- **@JsonManagedReference** y **@JsonBackReference**: se utilizan para manejar las relaciones maestro/detalle marcando la colección en el maestro y la propiedad inversa en el detalle (múltiples relaciones requieren asignar nombres únicos).

```
@JsonManagedReference
public User owner;
@JsonBackReference
public List<Item> userItems;
```
- **@JsonIdentityInfo**: indica la identidad del objeto para evitar problemas de recursión infinita.

```
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "id")
public class Item {
    public int id;
```

© JMA 2016. All rights reserved

Serialización XML (JAXB)

- JAXB (Java XML API Binding) proporciona a una manera rápida, conveniente de crear enlaces bidireccionales entre los documentos XML y los objetos Java. Dado un esquema, que especifica la estructura de los datos XML, el compilador JAXB genera un conjunto de clases de Java que contienen todo el código para analizar los documentos XML basados en el esquema. Una aplicación que utilice las clases generadas puede construir un árbol de objetos Java que representa un documento XML, manipular el contenido del árbol, y regenerar los documentos del árbol, todo ello en XML sin requerir que el desarrollador escriba código de análisis y de proceso complejo.
- Los principales beneficios de usar JAXB son:
 - Usa tecnología Java y XML
 - Garantiza datos válidos
 - Es rápida y fácil de usar
 - Puede restringir datos
 - Es personalizable
 - Es extensible

© JMA 2016. All rights reserved

Anotaciones principales (JAXB)

- Para indicar a los formateadores JAXB como transformar un objeto Java a XML y viceversa se puede anotar (`javax.xml.bind.annotation`) la clases `JavaBean` para que JAXP infiera el esquema de unión.
- Las principales anotaciones son:
 - `@XmlElement(namespace = "namespace")`: Define la raíz del XML.
 - `@XmlElement(name = "newName")`: Define el elemento de XML que se va usar.
 - `@XmlAttribute(required=true)`: Serializa la propiedad como un atributo del elemento.
 - `@XmlID`: Mapea un propiedad `JavaBean` como un XML ID.
 - `@XmlType(propOrder = { "field2", "field1", .. })`: Permite definir en que orden se van escribir los elementos dentro del XML.
 - `@XmlElementWrapper`: Envuelve en un elemento los elementos de una colección.
 - `@XmlTransient`: La propiedad no se serializa.

© JMA 2016. All rights reserved

Jackson DataFormat XML

- Extensión de formato de datos para Jackson (<http://jackson.codehaus.org>) para ofrecer soporte alternativo para serializar POJOs como XML y deserializar XML como pojos. Soporte implementado sobre la API de Stax (`javax.xml.stream`), mediante la implementación de tipos de API de Streaming de Jackson como `JsonGenerator`, `JsonParser` y `JsonFactory`. Algunos tipos de enlace de datos también se anularon (`ObjectMapper` se clasificó como `XmlMapper`).
- Dependencia:


```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

© JMA 2016. All rights reserved