Alejandro Jurado Gutiérrez

Hibernate y Spring

Manual para programadores

Índice
Índice 2
Framework Spring 4
Introducción 4
Módulos 4
Referencias 4
Spring Core 5
Inyección de dependencia 5
Referencias 6
Spring DAO-ORM 7
Hibernate 8
Introducción 8
Referencias 9
Mapeos Hibernate 10
Estados 12
Referencias 13
Asociaciones 14
Referencias 18
Consultas 19
HQL 20
Criteria 28
SQL 33
Referencias 34
Control de transacciones 35
Spring 35
Configuración 36
Referencias 36
Spring MVC 37
Introducción 37
DispatcherServlet 37
Referencias 38
Controladores 39
Mapeos 39
Métodos 40
Referencias 40
Spring Tags 41
Formularios 41
Internacionalización 41
Referencias 42
Control de errores 43
Referencias 44
Interceptores 45
Referencias 45
Tiles framework 46
Referencias 47
Servicios Spring 48
Introducción 48
Spring Boot 48
Referencias 49
Spring REST 50

Referencias 51
Spring Boot MVC 52
Referencias 55
Spring Cloud 56
Comunicación entre servicios 56
Descubrimiento 57
Referencias 58
Spring Security 59
Configuración 60
Taglib 62
Referencias 62
Spring Mock 63
Referencias 63

Framework Spring

Introducción

Spring es un framework para el desarrollo de aplicaciones Java EE (no necesariamente aplicaciones web), que facilita la implementación de la lógica de negocio del sistema, la construcción de clases Java unitarias fácilmente testeables y que presenta un buen rendimiento de ejecución frente a otras tecnologías como EJB, que además requieren de un contenedor web para su ejecución.

Como la mayoría de los frameworks de desarrollo, posee una serie de clases para facilitar la construcción de aplicaciones, y configuración a través de ficheros XML (o anotaciones).

Posee una arquitectura por módulos de los cuales se pueden aprovechar independientemente aquellos necesarios para conseguir la funcionalidad deseada. Cada módulo define perfectamente cuál es su tarea, y además en algunos casos permiten la incorporación de forma sencilla de otras tecnologías del mundo Java EE (Hibernate o Struts entre otros).

Módulos

1. Spring Core: proporciona la capacidad de Inversión de control a través del BeanFactory.
2. Spring Context: configuración de Spring.
3. Spring AOP: programación AOP y control de transacciones.
4. Spring DAO: gestión de excepciones y conexiones a través de templates.
5. Spring ORM: gestión de la información con ORM (propios y externos).
6. Spring Web: contexto para aplicaciones web.
7. Spring MVC: framework web para Spring.

Referencias

https://es.wikipedia.org/wiki/Spring_Framework

https://projects.spring.io/spring-framework/

Spring Core

Una de las principales características de Spring es la capacidad de gestionar el acceso a la lógica de negocio del sistema a través de los siguientes elementos:

- Inyección de dependencias con el mecanismo IoC.
- Desacople de configuración de dependencias de la lógica de la aplicación.

El principal elemento que se utiliza en la configuración de Spring necesario para poder implementar las dependencias son los Beans que presentan las siguientes características:

- Representa componentes software reutilizables.
- Puede tratarse de una clase tipo JavaBean o cualquier clase Java en general.
- Cada bean se inicializa por el sistema antes de poder utilizarse y en el orden correspondiente.

Inyección de dependencia

Cada clase de la arquitectura de la aplicación debe ser lo más independiente posible de otras clases, pertenezcan o no a la misma capa lógica, para fomentar su reutilización. Cuando dos clases son lo más independientes posibles cumplen con el principio del software denominado **loose coupling**, deseable en toda clase Java junto con otro principio fundamental, la **máxima cohesión**.

Parte de la dependencia que se produce en una clase se debe a la generación de instancias para poder utilizar otras clases del sistema. Modificaciones en los constructores o forma de inicializar a dichas clases tendrán un impacto en la clase original. La solución pasa por el empleo de interfaces que determinan las operaciones que se puedan consumir desde la clase origen, y la extracción de la generación de instancias hacia otros elementos de la arquitectura como por ejemplo hacia clases de tipo **Factory**.

Para ello las dependencias se inyectan a través de los **setters** o constructores de forma que cada clase se configura y conecta con otros servicios desde fuera (**IoC**), no desde su propio código. Representa una mejora del patrón Factory para la obtención de dependencias.

Existen dos tipos de contenedores en Spring para realizar la inyección en tiempo de ejecución:

1. **BeanFactory**: proporciona soporte de inyección de dependencias, determinando la generación, inicialización, utilización y destrucción de instancias de beans. Los beans residen en un contenedor y son generados según la estrategia correspondiente: singleton (defecto), prototype, request o session.
2. **ApplicationContext**: proporciona la definición de los servicios a la aplicación. Los contextos pueden ser definidos en XML, código Java o anotaciones.

El proceso de resolución de dependencias en tiempo de ejecución se denomina **wiring**, y será proporcionado a través de clases Spring como por ejemplo:

- ClassPathXmlApplicationContext.
- XmlWebApplicationContext.

Referencias

https://en.wikipedia.org/wiki/Inversion_of_control

https://en.wikipedia.org/wiki/Dependency_injection

http://www.mkyong.com/spring3/spring-3-hello-world-example/

http://www.springbyexample.org/

Spring DAO-ORM

El patrón DAO (Data Access Object) proporciona un interfaz común de acceso a los datos independiente del sistema de almacenamiento.

http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Para poder utilizar DAO Spring proporciona múltiples opciones gracias a su sistema de templates (plantillas) que manejan distintas opciones de persistencia, desde JDBC a APIs más avanzadas como JPA o Hibernate.

La conexión a la Base de datos se simplifica mucho:

1. La gestión de acceso a datos se delega al contenedor y se inyecta directamente a los Daos.
2. Los Daos pueden emplear un template directamente para la ejecución de operaciones sin gestionar la conexión/desconexión.
3. Existen clases de soporte que evitan tener que realizar la inyección y gestión de templates desde el Dao.
4. Todos los métodos de plantilla devuelven excepciones DataAccessException, que permite afinar el motivo concreto de error en la base de datos.

Cada tipo de tecnología requiere de una configuración de acceso y trabaja con una clase de soporte distinta.

En el caso de trabajar con JDBC se emplea un DataSource directamente para indicar la conexión a la Base de datos.

En el caso de emplear Hibernate, es necesario trabajar con un SessionFactory encargado de establecer las propiedades principales del framework, al cual se le inyecta el DataSource.

Las siguientes acciones son las que soporta Spring y evita al programador:

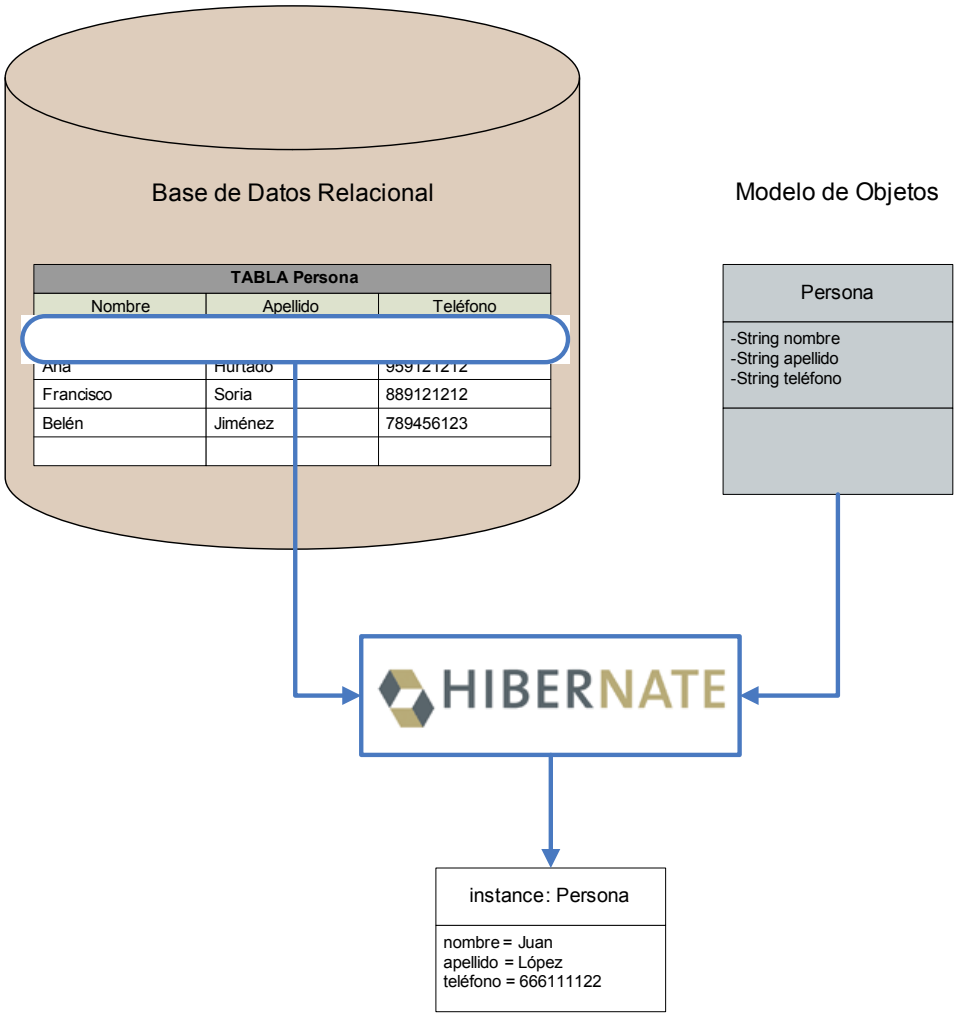
Acción	Spring	Programador
Definir parámetros de conexión		X
Abrir la conexión	X	
Inyección de parámetros de configuración	X	X
Definir operaciones a través de plantilla		X
Procesar excepciones	X	
Gestionar transacciones	X	
Cerrar la conexión	X	

Hibernate

Introducción

Hibernate es una implementación de mapeo Objeto-Relacional (**ORM** – Object-Relational Mapping). El objetivo de un ORM es hacer corresponder el modelo de datos de la aplicación con el modelo de clases de la misma, los cuales no son tan distintos como se pueda pensar.

Para ello, Hibernate realiza el mapeo de elementos (tablas, columnas) entre una Base de datos relacional y objetos de la aplicación en cuestión:



Este tipo de sistema facilita todas las operaciones de persistencia y acceso a los datos desde la aplicación al presentar las siguientes características principales:

- **Independiente de SQL**: Hibernate incorpora funcionalidades para las operaciones simples de recuperación y actualización de datos para las cuales no será necesario utilizar las sentencias SELECT, INSERT, UPDATE o DELETE habituales, siendo sustituidas por llamadas a métodos (list, save, update, delete, …). Hibernate se encarga de generar la sentencia SQL que se encargará de realizar tales operaciones. De cualquier modo, se pueden especificar consultas SQL en caso necesario o incluso consultas en el dialecto de Hibernate, el HQL, con sintaxis similar a SQL pero que establece una serie de elementos propios del sistema ORM que gestiona.
- **Independiente del SGBD**: al aislar las funciones de manipulación de datos del lenguaje SQL, se consigue que la aplicación pueda comunicarse con cualquier SGBD ya que no existirán dependencias o particularidades en las sentencias de consulta o actualización de datos que harían a la aplicación dependiente de un SGBD en cuestión. El encargado de realizar la traducción final entre las operaciones Objeto-relacionales y las sentencias SQL es Hibernate, con lo cual el problema de la compatibilidad queda solventado. Incorpora soporte para la mayoría de los sistemas de bases de datos existentes.
- **Independiente de JDBC**: Hibernate contiene una API completa que aísla a la aplicación, no solamente de las operaciones con lenguaje SQL, sino también la utilización de objetos propios de la gestión de Bases de datos JDBC (Statement, ResultSet, …). Este tipo de objetos habituales en la programación Java de acceso a Bases de datos, se sustituye por el uso de objetos mucho más sencillos como colecciones de clases tipo JavaBeans que contendrán los datos de los almacenes de datos.

Referencias

http://hibernate.org

Mapeos Hibernate

Para cada una de las clases de persistencia tiene que existir un mapeo que se encarga de definir la relación entre las tablas y columnas de éstas en la Base de datos, y las clases y sus propiedades en el modelo de objetos.

Este mapeo puede ser definido mediante anotaciones o mediante ficheros XML.

El nombre del fichero (Departamento.hbm.xml) coincidirá con el nombre de la clase que mapea y se guardará habitualmente en el mismo paquete donde se encuentra dicha clase. En este archivo XML es necesario establecer la tabla del esquema de la base de datos que se mapea con la clase Java y las columnas de ésta mapeadas con los atributos, destacando además el atributo id que corresponde con el campo clave de la tabla y que debe definir el tipo de asignación de clave en función de si se trata de una clave manual o de tipo autonumérico.

```
<hibernate-mapping>
    <class name="es.hubiqus.spr.model.Departamento"
                        table="departamentos" catalog="curso">
        <id name="numero" type="int">
            <column name="numero" />
            <generator class="assigned" />
        </id>
        <property name="nombre" type="string">
            <column name="nombre" length="50" not-null="true" />
        </property>
        <property name="ciudad" type="string">
            <column name="ciudad" length="50" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Actualmente es más habitual realizar los mapeos con anotaciones. Resulta una forma mucho más cómoda de mapeo ya que evita la generación de un archivo adicional, y en el caso de que los elementos (clase-tabla, atributos-columnas) tengan un mismo nombre no es obligatorio incluirlos.

Existen dos anotaciones clave en este tipo de mapeo, **@Entity** que define que la clase es una entidad de modelo, y **@Id** para definir el identificador de la clase. Esta última debe ir acompañada del generador de claves en el caso que corresponda.

Las anotaciones **@Table** y **@Column** se convierten en obligatorias siempre que los nombres Java difieran de los nombres de la base de datos. La anotación **@Column** de cualquier forma es habitual emplearla para indicar restricciones sobre la información contenida, estas restricciones se validan antes de lanzar la consulta SQL a la base de datos y pueden contener, entre otros, los siguientes elementos:

- **name**: nombre de la columna.
- **nullable**: **true**/false, indica si permite valores nulos.

- **unique**: true/**false**, etiqueta informativa para valores únicos.
- **insertable**: **true**/false, indica si el atributo se incluye en la sentencia INSERT.
- **updatable**: **true**/false, ídem para UPDATE.
- **length**: longitud máxima permitida.
- **precision**/**scale**: para precisión y escala de decimales.

```java
@Entity
@Table(name="departamentos")
public class Departamento {

	private Integer numero;
	private String nombre;
	private String ciudad;

	@Id
//	@GeneratedValue(strategy = GenerationType.IDENTITY)
	public Integer getNumero() {
		return numero;
	}
	public void setNumero(Integer numero) {
		this.numero = numero;
	}

	@Column(name = "nombre", nullable = false, length = 50)
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	@Column(name = "ciudad", nullable = false, length = 50)
	public String getCiudad() {
		return ciudad;
	}
	public void setCiudad(String ciudad) {
		this.ciudad = ciudad;
	}
}
```

La anotación **@Transient** indica si una columna no se mapea a ningún campo.

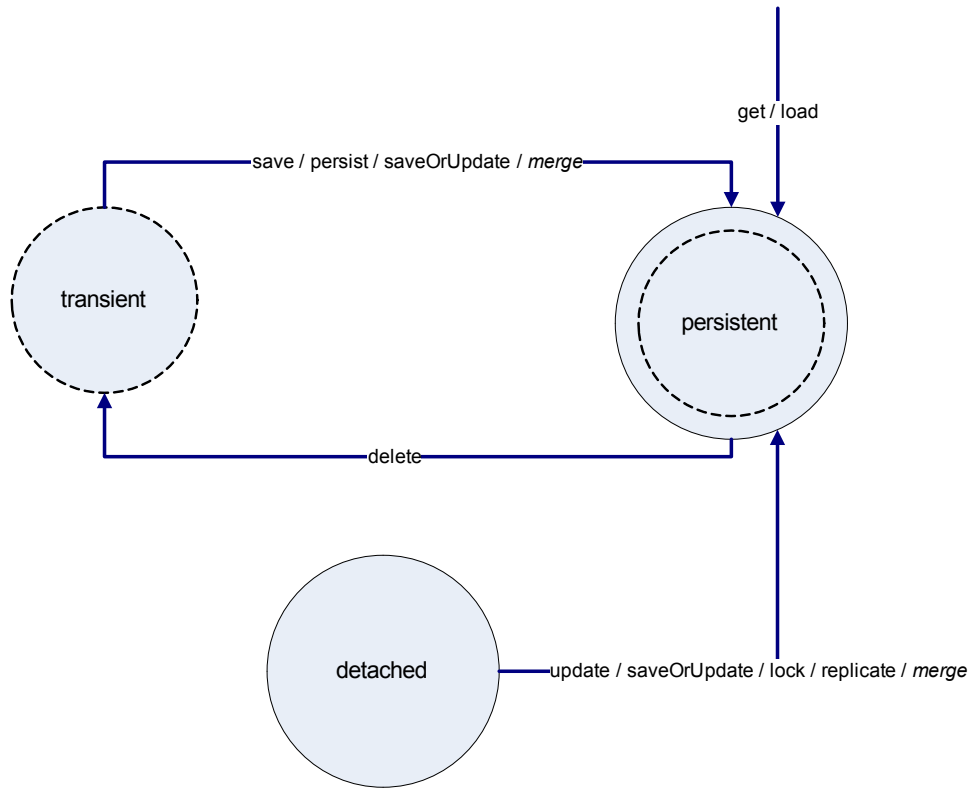
La anotación **@Temporal(TemporalType)** indica el tipo de fecha que se utiliza.

Estados

El interfaz Session es la principal vía de interacción entre la aplicación Java e Hibernate. Representa a la sesión actual, donde se producen transacciones para la interacción con la Base de datos, efectuando operaciones de lectura, creación, modificación y eliminación de instancias de clases mapeadas. Las instancias de tales clases se pueden encontrar en tres estados posibles:

- **transient**: no es persistente, no está asociada a ninguna sesión. Se pueden convertir en persistent invocando a los métodos save, persist o saveOrUpdate. Se encuentra en la aplicación pero no en la Base de datos.
- **persistent**: asociado a una única sesión. Se encuentra en la aplicación y en la Base de datos.
- **detached**: persistente previamente, no asociado con ninguna sesión. Se encuentra en la Base de datos, pero no se ha recuperado desde la aplicación.

En la siguiente ilustración aparecen estos estados, y cuáles son los métodos que provocan la transición de uno a otro.



Para poder operar con una sesión es necesario obtener una instancia de la misma y comenzar una transacción. Cualquier operación sobre dicha instancia debe contenerse dentro de una transacción.

Si se produce una excepción dentro de la transacción, ésta debe ser cancelada (rollback), si no hay que confirmarla para hacer persistentes los cambios (commit):

```
//Obtener la sesión actual
Session session = getSessionFactory().openSession();

try{
      //Comenzar la transacción
      session.beginTransaction();
      //Operaciones con session
      //Confirmar transacción
      session.getTransaction().commit();
}catch (Exception ex){
      //Deshacer transacción
      session.getTransaction().rollback();
}finally{
      if (session != null){
            session.close();
      }
}
```

Los métodos que se invocan sobre la instancia de Session producen operaciones sobre la Base de datos a través de sentencias SQL, generadas en el dialecto concreto configurado en el archivo de configuración. De esta forma, las siguientes sentencias se producen para algunos de los métodos enumerados anteriormente:

- INSERT: save, saveOrUpdate.
- UPDATE: update, saveOrUpdate.
- DELETE: delete.

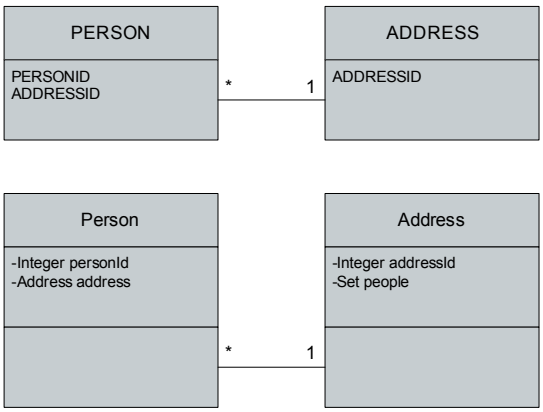
Referencias

http://docs.jboss.org/hibernate/orm/3.5/javadocs/

http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/#entity

Asociaciones

Las asociaciones en Hibernate representan las relaciones entre tablas. En un modelo ORM, las claves foráneas habitualmente no se modelan con el campo identificador únicamente, sino con instancias completas de objetos para poder aprovechar la potencia del Framework.



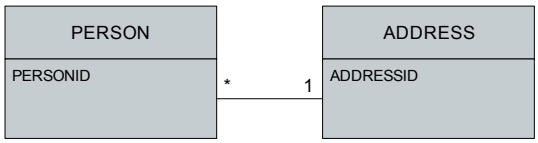
En la imagen anterior se puede observar la relación que existe entre la tabla Person (Persona) y Address (Dirección). Se trata de una relación uno a muchos donde el campo AddressId actúa como una clave foránea de la tabla Address (maestro) en la tabla Person (detalle) lo cual modela que en una Dirección pueden habitar muchas Personas.

Este tipo de relación es muy habitual en Bases de datos y como se ha comentado, se suele denominar uno a muchos (leyendo en la dirección Address → Person). En Hibernate suele leerse de forma inversa, muchos a uno (many-to-one), pero sigue tratándose de la misma relación. Esta relación a nivel de clases persistentes se modela de forma sencilla a través de un atributo address en la clase Person, de forma que si se dispone de una instancia de Person se pueden conocer todos los datos de la instancia Address donde reside esa persona, sin necesidad de realizar ningún tipo de consulta adicional a la Base de datos. En una Base de datos irremediablemente habría que realizar una consulta cruzada para recuperar tales datos.

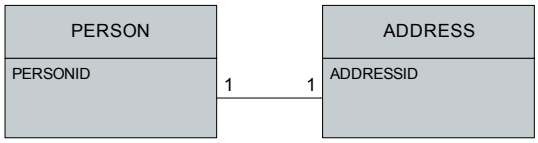
El otro atributo fruto de esta relación que aparece a nivel de clases es el conjunto people. Esto Hibernate se denomina la relación inversa, ya que las relaciones se pueden modelar en ambos sentidos. Este sentido de la relación es el que habitualmente se utiliza al realizar una consulta cruzada, es decir, cuando interesa conocer quiénes son los habitantes de una determinada dirección. De nuevo Hibernate proporciona esta información de forma sencilla sin necesidad de implementar consultas, ya que el atributo de tipo Set de la clase Address contendrá todas las instancias de la clase Person para aquellas personas que cumplan con la relación.

Para poder aprovechar estas funcionalidades es necesario tener en cuenta otros factores como la configuración de los ficheros de mapeo, o los métodos a utilizar para recuperar los valores de la Base de datos. Ambos conceptos se explican ampliamente en este capítulo y el siguiente.

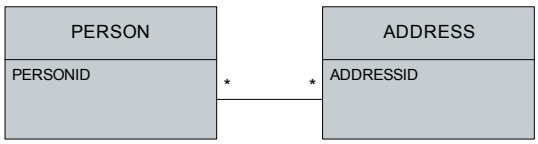
En general, en una base de datos existen tres tipos de relación:



Muchos a Uno
Una persona vive en una única dirección
En una dirección pueden vivir varias personas

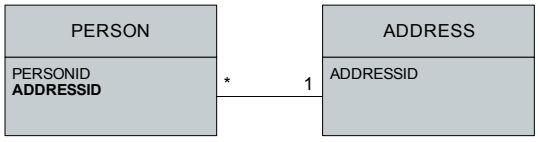


Uno a Uno
Una persona vive en una única dirección
En una dirección solo puede vivir una persona

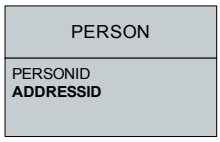


Muchos a Muchos
Una persona puede vivir en varias direcciones
En una dirección pueden vivir varias personas

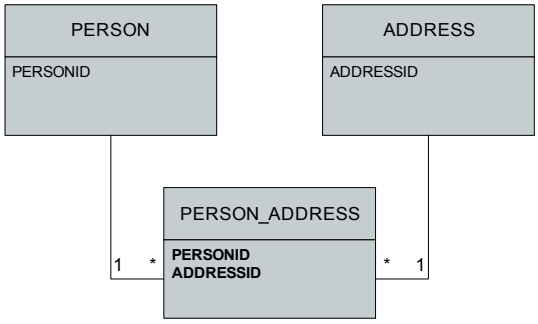
Las relaciones que existen entre las tablas dan lugar a modificaciones en las mismas a la hora de normalizar el modelo, de forma que no existan campos duplicados ni pérdidas de integridad por un mal diseño.



Muchos a Uno
La clave de la tabla Maestro pasa como clave foránea al detalle



Uno a Uno
Una de las tablas desaparece, quedando sus datos absorbidos por la tabla principal



Muchos a Muchos
Aparece una tabla intermedia cuya clave estará formada por las claves foráneas de ambas tablas

De las imágenes anteriores cabe destacar un aspecto importante, todas los tipos de relaciones que aparecían en primera instancia, se pueden reducir a un único tipo, la relación Muchos a uno, ya que:

- Las relaciones Uno a Uno se suelen eliminar
- Las relaciones Muchos a Muchos se transforman en relaciones Muchos a Uno sobre la tabla intermedia.

Este hecho va a simplificar considerablemente la gestión de asociaciones en Hibernate, ya que solamente hará falta controlar un tipo de relación para poder construir modelos ORM que representen modelos Entidad-Relación de cierta complejidad.

Los tipos de asociaciones en Hibernate que servirán para modelar las relaciones citadas es la siguiente:

- Unidireccionales: solamente se representa un extremo de la relación.
- Unidireccionales con tabla de unión.
- **Bidireccionales:** se representan ambos extremos.
- Bidireccionales con tabla de unión.

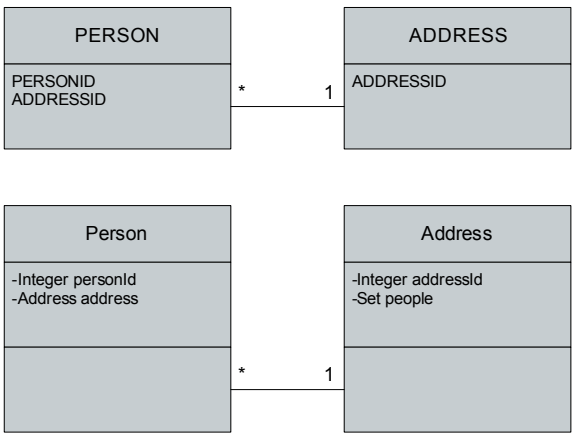
Teniendo en cuenta esos tipos de relaciones y los dos sentidos de la relación aparecen en Hibernate 12 tipos distintos de relación que se pueden modelar de forma directa. Todas estas relaciones se pueden reducir básicamente a tres tipos que serán explicados a continuación. Como se ha comentado, lo más sencillo para el desarrollo puede resultar en el diseño adecuado del modelo Entidad-Relación de la Base de datos, y simplemente en utilizar mapeos many-to-one para las relaciones entre las tablas.

Para poder modelar un sistema de Bases de datos la relación Many-to-one puede resultar la única relación imprescindible ya que como se ha explicado, el resto de ellas se pueden transformar. Esta relación normalmente se denomina Maestro-Detalle, donde existe un extremo Maestro que dispone de las claves principales, y un Detalle donde esta clave aparece como foránea.

A nivel de Hibernate, la relación bidireccional se modela de la siguiente forma:

- Many-to-one en el Detalle.
- One-to-Many en el Maestro, incorporando el atributo inverse. **No es obligatorio** modelar este sentido de la relación.

En el siguiente ejemplo aparecen los mapeos que se producirán en las clases para modelar esta relación.



```
<class name="Person">
      <id name="id" column="personId">
            <generator class="native"/>
      </id>
      <many-to-one name="address" column="addressId"/>
</class>
```

```
@ManyToOne
@JoinColumn(name="addressId")
public Address getAddress() {
      return address;
}
```

En la clase Person se mapea el sentido many-to-one que es el más sencillo:

- A la clase se agrega un atributo Address address.
- En el fichero de mapeo se agrega una etiqueta many-to-one:
  - ➢ name: nombre del atributo.
  - ➢ column: nombre de la columna de la tabla de base de datos que se corresponde con el atributo (la clave foránea).

```
<class name="Address">
      <id name="id" column="addressId">
            <generator class="native"/>
      </id>
      <set name="people" inverse="true">
            <key column="addressId"/>
            <one-to-many class="Person"/>
      </set>
</class>
```

```
@OneToMany(fetch=FetchType.EAGER, mappedBy="address")
public Set<Person> getPeople() {
      return people;
}
```

En la clase Address se mapea el sentido one-to-many, el sentido inverso:

- A la clase se agrega un atributo Set people.
- En el fichero de mapeo se agrega una etiqueta set:
  - ➢ name: nombre del atributo.
  - ➢ inverse (true): la relación es inversa.

- key:
  - ✓ column: nombre de la columna de la tabla de base de datos que actúa como clave foránea en la otra tabla.
- one-to-many
  - ✓ class: nombre de la clase que modela la otra tabla en la relación.

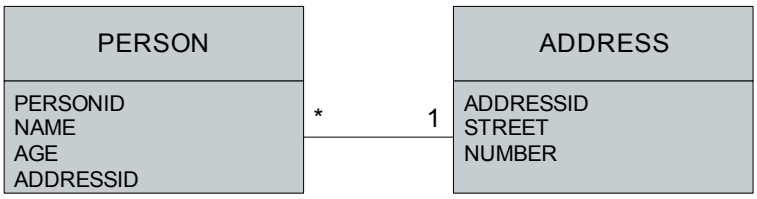
Referencias

http://docs.jboss.org/hibernate/orm/3.5/reference/en/html/associations.html

Consultas

En las bases de datos relacionales, las operaciones más frecuentes suelen ser las recuperaciones de datos de una o más tablas o vistas. Para poder realizar esta operación, en dichas bases de datos se recurre al lenguaje SQL, y más concretamente a la sentencia SELECT. De esta forma, especificando una sentencia más o menos compleja es posible obtener la información solicitada

Como se vio en el apartado anterior, la relación más habitual entre dos tablas es la relación Maestro-Detalle o many-to-one. Sobre esta relación se pueden efectuar 2 tipos de consultas cruzadas bastante habituales:



Obtener todos los datos de las personas y de la de la dirección donde viven:

```
SELECT PERSON.NAME, PERSON.AGE, ADDRESS.STREET, ADDRESS.NUMBER
FROM PERSON, ADDRESS
WHERE PERSON.ADDRESSID = ADDRESS.ADDRESSID
```

Obtener los nombres de todas las personas que viven en una dirección determinada:

```
SELECT PERSON.NAME
FROM PERSON, ADDRESS
WHERE PERSON.ADDRESSID = ADDRESS.ADDRESSID
AND ADDRESSID = 1
```

En Hibernate se podría realizar de la misma forma (SQLQuery) pero lo más habitual no es esto, sino aprovechar las características del modelo ORM para acceder a esta información de forma más eficiente. Para ello existen dos alternativas:

- Usar una consulta HQL para ejecutar la consulta, de forma que usando un lenguaje orientado a objetos pero muy similar a SQL.
- Usar un Criteria, de forma que a través de programación se puedan utilizar diversos métodos para recuperar los datos deseados de forma similar a como lo hace HQL.

Como se ha venido comentando, la relación entre las dos tablas anteriores se modela a nivel de clases con una asociación many-to-one, y los siguientes atributos relacionados:

- Address address en la clase Person, que permitirá acceder a todos los datos de la dirección de una Persona.
- Set people en la clase Address, que permitirá conocer todas las personas que habitan una misma dirección.

Lo que se modela en las clases es exactamente lo que se estaba accediendo a través de sentencias SELECT anteriores. Por tanto casi todo el trabajo ya está hecho, solamente hace falta acceder a los datos:

```
List list = session.createQuery("from Person").list();
List list = session.createCriteria(Person.class).list();
```

En ambos casos se obtiene una lista con una instancia de la clase Person por cada una de las personas de la tabla Person, en cuyo atributo address se puede consultar toda la información necesaria sobre la dirección. Sobre esta capacidad es necesario destacar que no siempre Hibernate recupera todos los datos relacionados ya que suele comportarse en modo “lazy”, es decir, solamente recupera los datos principales de la clase, mientras que los atributos de asociaciones (como address) quedan sin inicializar por motivos de eficiencia. Para forzar que estos atributos siempre se carguen se utilizan los fetch o alias a la hora de recuperar los datos, los cuales se explican con mayor detalle en los apartados posteriores.

HQL

Hibernate usa un potente lenguaje de consulta (HQL), que es similar en apariencia a SQL. En comparación con SQL, sin embargo, HQL es completamente orientado a objetos y es capaz de gestionar conceptos como herencia, polimorfismo y asociaciones.

Exceptuando los nombres de clases y atributos, el lenguaje HQL es insensible a mayúsculas (por ejemplo se admite SELECT, select o SEleCT).

La consulta HQL más simple es la siguiente:

```
from Cat
```

Devuelve todas las instancias de la clase Cat. Normalmente NO es necesario utilizar el nombre de paquete (from eg.Cat) ya que la opción auto-import es true por defecto.

A la hora de referirse a Cat desde otras partes de la consulta, normalmente se utiliza un alias a continuación de la clase. La especificación del alias puede usar la palabra clave as (as cat) o no:

```
from Cat cat
```

Los joins son la forma de realizar cruces entre tablas en Bases de datos. Habitualmente no se utilizan los joins porque la forma de escribir las consultas resulta más amigable utilizando condiciones where, de forma que es el gestor de la Base de datos quien se encarga de realizar la transformación necesaria para utilizar este tipo de unión. Las consultas siguientes son equivalentes en cuanto al resultado obtenido:

```
SELECT e.nombre, d.nombre FROM empleados e, departamentos d
WHERE d.numero = e.dnumero

SELECT e.nombre, d.nombre FROM empleados e
INNER JOIN departamentos d ON d.numero = e.dnumero
```

Existen varios tipos de JOIN posibles entre 2 tablas de la Base de datos:

- **CROSS JOIN:** unión de todos los datos de dos tablas sin especificar ningún criterio (producto cartesiano). Este tipo de JOIN es muy ineficiente.

PROPIETARIOS		
PISO	NOMBRE	GARAJE
1A	Antonio	12
1B	Manuel	
2B	Javier	10

GARAJES	
NUMERO	TAMAÑO
10	20
11	22
12	22

1A	Antonio	12	10	20
1A	Antonio	12	11	22
1A	Antonio	12	12	22
1B	Manuel		10	20
1B	Manuel		11	22
…	…	…	…	

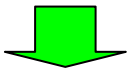
En Hibernate se especifica de forma similar a SQL:

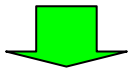
```
from Formula, Parameter
```

- **INNER JOIN** o **JOIN:** unión de los datos de dos tablas solamente en aquellos casos donde existan datos en el campo de combinación. Este tipo de JOIN es el que resulta equivalente a realizar la combinación a través de un WHERE.

PROPIETARIOS		
PISO	NOMBRE	GARAJE
1A	Antonio	12
1B	Manuel	
2B	Javier	10

GARAJES	
NUMERO	TAMAÑO
10	20
11	22
12	22





PISO	NOMBRE	NUMERO	TAMAÑO
1A	Antonio	12	22
2B	Javier	10	20

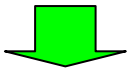
En Hibernate también se especifica de forma similar a SQL, solamente teniendo en cuenta que no se hacen uniones entre tablas propiamente dichas, sino con los atributos sobre los que se han montado asociaciones. Se puede usar con las palabras clave inner join o inner outer join, o incluso simplemente join:

```
from Cat as cat
inner join cat.mate as mate
```

- **LEFT OUTER JOIN** o **LEFT JOIN:** se seleccionan todos los datos de la primera tabla de la combinación, uniendo los datos de la segunda tabla solamente en los casos que sea posible, y poniendo null en los casos que no sea posible.

PROPIETARIOS		
PISO	NOMBRE	GARAJE
1A	Antonio	12
1B	Manuel	
2B	Javier	10

GARAJES	
NUMERO	TAMAÑO
10	20
11	22
12	22



PISO	NOMBRE	NUMERO	TAMAÑO
1A	Antonio	12	22
1B	Manuel	null	null
2B	Javier	10	20

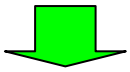
De nuevo se pueden especificar con HQL sobre los atributos sobre los que se han montado asociaciones uniones de este tipo a través de left join o left outer join:

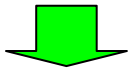
```
from Cat as cat
inner join cat.mate as mate
left outer join cat.kittens as kitten
```

- **RIGHT OUTER JOIN** o **RIGHT JOIN:** se seleccionan todos los datos de la segunda tabla de la combinación, uniendo los datos de la primera tabla solamente en los casos que sea posible, y poniendo null en los casos que no sea posible.

PROPIETARIOS		
PISO	NOMBRE	GARAJE
1A	Antonio	12
1B	Manuel	
2B	Javier	10

GARAJES	
NUMERO	TAMAÑO
10	20
11	22
12	22





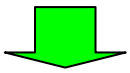
PISO	NOMBRE	NUMERO	TAMAÑO
1A	Antonio	12	22
2B	Javier	10	20
null	null	11	22

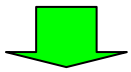
Se puede especificar con HQL con right join o right outer join.

- **FULL OUTER JOIN** o **FULL JOIN:** es una combinación de los dos anteriores.

PROPIETARIOS		
PISO	NOMBRE	GARAJE
1A	Antonio	12
1B	Manuel	
2B	Javier	10

GARAJES	
NUMERO	TAMAÑO
10	20
11	22
12	22





PISO	NOMBRE	NUMERO	TAMAÑO
1A	Antonio	12	22
1B	Manuel	null	null
2B	Javier	10	20
null	null	11	22

```
from Formula form
full join form.parameter param
```

En general, sobre cualquier tipo de join se pueden establecer condiciones:

```
from Cat as cat
left join cat.kittens as kitten
with kitten.bodyWeight > 10.0
```

Por último, es muy importante la utilización de la palabra clave fetch. Como se ha comentado, normalmente las colecciones o atributos que se refieren a asociaciones mapeadas no se inicializan debido al comportamiento “lazy” por defecto. Para asegurar la recuperación de los datos en dichas propiedades, es necesario utilizar fetch en conjunción con los joins:

```
from Cat as cat
inner join fetch cat.mate
left join fetch cat.kittens child
left join fetch child.kittens
```

Para realizar un fetch sobre todas las propiedades susceptibles de búsqueda, se puede utilizar una sentencia de este tipo:

```
from Document
fetch all properties
order by name
```

También es posible la utilización de un join implícito, es decir, no aparece la cláusula join en la consulta pero para Hibernate es necesario realizarla para resolver las condiciones impuestas. En este ejemplo no sería necesario realizar un join cat.mate si directamente se incluye como condición en la condición where, aunque se recomienda su inclusión por claridad a la hora de leer la consulta:

```
from Cat as cat
where cat.mate.name like '%s%'
```

La cláusula Select permite obtener propiedades individuales en lugar de recuperar instancias completas. Estas propiedades pueden referirse a atributos simples o a atributos mapeados. En el siguiente ejemplo se accede a la propiedad name (String) y a la propiedad mate (Cat), destacando que en este último caso se hace un join implícito (si no sería necesario incluir un join cat.mate).

```
select cat.name, cat.mate as cmate
from Cat cat
```

Dentro de las cláusulas Select se puede hacer referencia al identificador de la clase de dos formas distintas:

- Utilizando la propiedad id ya que este atributo se encuentra mapeado siempre con ese identificador (cat.id).
- Utilizando el nombre de la propiedad si ésta tiene un nombre distinto (cat.catId).

Cuando se hace una consulta HQL sin Select, a nivel de programación se recuperan las instancias concretas de la clase implicada en dicha consulta. En el siguiente fragmento de código se recuperan una colección de tipo List, donde cada elemento es una instancia de la clase Cat:

```
List list = session.createQuery("from Cat").list();
Cat cat = (Cat) list.get(0);
```

Cuando se utiliza la cláusula Select, o si se realiza un cross join de clases (from Formula, Parameter), resulta imposible para Hibernate determinar qué clase ha de emplear en la recuperación de los datos, por lo que éstos aparecen en forma de Object[]. Por tanto el conjunto de datos resultado sería una colección List de Object[]:

```
List list = session.createQuery(
                        "select cat.name, cat.mate from Cat
                        cat").list();
Object[] res = (Object[]) list.get(0);
```

En el Object[] obtenido como resultado, cada una de las posiciones es un objeto de la clase concreta a la que pertenece el atributo, ya sea porque se ha mapeado de forma explícita en el fichero hbm.xml (atributo type de la etiqueta property), o porque Hibernate lo ha resuelto de forma automática:

```
String name = (String) res[0];
Cat mate = (Cat) res[1];
```

En lugar de obtener el resultado en un Object[], se puede forzar en la consulta a obtener una colección de tipo List:

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
inner join mother.mate as mate
left outer join mother.kittens as offspr
```

También se puede obtener una colección Map, donde los alias actúan como claves y los valores son los recuperados en la consulta:

```
select new map(max(bodyWeight) as max, min(bodyWeight) as min,
count(*))
from Cat cat
```

Por último, se puede forzar a devolver el resultado en una clase concreta, cuyo constructor debe responder a la estructura utilizada en la consulta:

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
join mother.mate as mate
left join mother.kittens as offspr
```

En HQL se pueden utilizar las operaciones y expresiones habituales en SQL:

- Operadores matemáticos: +, -, *, /
- Operadores de comparación: =, >=, <=, <>, !=, like
- Operadores lógicos: and, or, not, in, is null, …
- Funciones: concat() (o ||), current_timestamp(), day(), month(), …
- …

De la misma forma, es posible utilizar funciones agregadas habituales:

- avg(), sum(), min(), max(), count()

La cláusula Where se utiliza para la especificación de condiciones en las consultas. Puede incorporar expresiones para determinar la condición, de forma similar a como se realiza en SQL, teniendo en cuenta las características orientadas a objetos de HQL:

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

Es posible comparar instancias completas, así como acceder a la propiedad class:

```
from Cat cat, Cat rival
where cat.mate = rival.mate
and cat.class = DomesticCat
```

La cláusula utilizada para ordenar el conjunto de resultados es Order By:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

La agrupación del conjunto de resultados normalmente es utilizada para la aplicación de funciones de agregado sobre los grupos. La utilización en HQL no difiere de la habitual en SQL:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

HQL también permite la utilización de consultas anidadas tanto en la cláusula Select como en las condiciones Where:

```
select cat.id, (select max(kit.weight) from cat.kitten kit) as
kitweight
from Cat as fatcat
where fatcat.weight > (
select avg(cat.weight) from DomesticCat cat
)
```

Criteria

Hibernate proporciona una API intuitiva y extensible para la construcción de consultas, la cual evita al desarrollador la necesidad de conocer HQL ni SQL para la recuperación de datos.

La forma más simple para construir un Criteria es la siguiente:

```
Criteria crit = sess.createCriteria(Cat.class);
```

El método createCriteria devuelve un objeto perteneciente al interfaz Criteria. La mayoría de los métodos que se invocan sobre un Criteria devuelven otro objeto de tipo Criteria, de forma que las llamadas se van enlazando para construir una consulta completa. A lo largo de este capítulo se explican los distintos métodos disponibles.

```
List cats = session.createCriteria(Cat.class)
.setProjection( Projections.projectionList()
.add( Projections.rowCount() )
.add( Projections.avg("weight") )
.add( Projections.max("weight") )
.add( Projections.min("weight") )
.add( Projections.groupProperty("color") )
)
.addOrder( Order.asc("color") )
.list();
```

Hay dos formas básicas de recuperar los datos de un Criteria:

- Invocando al método list, que devolverá una lista con las instancias concretas obtenidas por la consulta.

```
List cats = session.createCriteria(Cat.class).list();
Cat cat = (Cat) cats.get(0);
```

- Invocando al método uniqueResult, que devolverá un único resultado como consecuencia de la ejecución de la consulta. Este método resulta en excepción si el resultado no es un único registro.

```
Cat cat = (Cat) session.createCriteria(Cat.class)
      .add(Restrictions.eq("id", new Integer(1)))
      .uniqueResult();
```

En un Criteria, la única posibilidad de join que existe es utilizar un fetch:

```
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.like("name", "Fritz%") )
      .setFetchMode("mate", FetchMode.JOIN)
      .setFetchMode("kittens", FetchMode.JOIN)
      .list();
```

El valor FetchMode.JOIN indica que debe realizarse un inner join a la hora de realizar el fetch. En el caso que se quiera realizar otro tipo de join se utilizará el método createAlias indicando que se trata de un LEFT_JOIN, RIGHT_JOIN o FULL_JOIN:

```
List cats = sess.createCriteria(Cat.class)
      .createAlias("kittens", "kt", CriteriaSpecification.LEFT_JOIN)
      .add( Restrictions.eqProperty("kt.name", "miao") )
      .list();
```

En el siguiente ejemplo aparece la creación de un alias normal, es simplemente equivalente a la creación de un alias en HQL (as kt):

```
List cats = sess.createCriteria(Cat.class)
      .createAlias("kittens", "kt")
      .add( Restrictions.eqProperty("kt.name", "miao") )
      .list();
```

La cláusula Select de SQL o HQL permite realizar la operación denominada Proyección en álgebra relacional. El método setProjection se encarga de indicar que se van a seleccionar propiedades concretas del conjunto de resultados:

```
List results = session.createCriteria(Domestic.class, "cat")
      .createAlias("kittens", "kit")
      .setProjection( Projections.projectionList()
            .add( Property.forName("cat.name") )
            .add( Projections.property("kit.name"), "kitName" )
      )
      .list();
```

El método setProjection recibe como parámetro un ProjectionList, que se irá construyendo a través de métodos add. El método add recibe la propiedad concreta que se va a proyectar, y el segundo parámetro en caso de existir se trata del alias de dicha propiedad.

Para recuperar una propiedad a incluir en la lista de proyecciones se puede utilizar indistintamente Property.forName() o Projections.property().

Al igual que ocurre con HQL cuando se utiliza setProjection resulta imposible para Hibernate determinar qué clase ha de emplear en la recuperación de los datos, por lo que éstos aparecen en forma de Object[].

Para establecer condiciones en un Criteria se usa la clase Restrictions. A través del método add del objeto Criteria se van añadiendo las distintas restricciones, de forma que se van uniendo a través de una conjunción, es decir, a través de and. En la siguiente consulta, la condición resultante será name like "Fritz%" and weight >= 10.

```
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.like("name", "Fritz%") )
      .add( Restrictions.ge("weight", new Integer(10)) )
      .list();
```

Existen restricciones para la mayoría de los operadores habituales en consultas SQL. Para unir restricciones a través de un or se puede usar el método or() para 2 restricciones, o disjunction() para un número mayor de restricciones:

```
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.like("name", "Fritz%") )
      .add( Restrictions.or(
            Restrictions.eq( "age", new Integer(0) ),
            Restrictions.isNull("age")
      ) )
      .list();
```

```
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.in( "name", new String[] { "Fritz", "Izi",
      "Pk" } ) )
      .add( Restrictions.disjunction()
            .add( Restrictions.isNull("age") )
            .add( Restrictions.eq("age", new Integer(0) ) )
            .add( Restrictions.eq("age", new Integer(1) ) )
            .add( Restrictions.eq("age", new Integer(2) ) )
) )
.list();
```

También se pueden obtener restricciones a través de las propiedades directamente:

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.disjunction()
            .add( age.isNull() )
            .add( age.eq( new Integer(0) ) )
            .add( age.eq( new Integer(1) ) )
            .add( age.eq( new Integer(2) ) )
      ) )
      .add( Property.forName("name").in( new String[] { "Fritz",
      "Izi", "Pk" } ) )
      .list();
```

Por último, se pueden establecer restricciones SQL directamente, indicando la condición SQL, los valores a aplicar y los tipos de dichos valores:

```
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.sqlRestriction("lower({alias}.name) like
      lower(?)",
                                    "Fritz%",
                                    Hibernate.STRING) )
      .list();
```

El método utilizado para ordenar el conjunto de resultados es addOrder. Este método recibe una instancia de la clase Order que se puede obtener invocando al método asc() (orden ascendente) o desc (orden descendente) sobre la propia clase Order, o sobre una propiedad directamente:

```
List cats = sess.createCriteria(Cat.class)
      .add( Restrictions.like("name", "F%")
      .addOrder( Order.asc("name") )
      .addOrder( Property.forName("age").desc() )
      .list();
```

La agrupación del conjunto de resultados y las funciones de agregado también están permitidas sobre un Criteria. Las funciones agregadas se añaden a la lista de proyecciones del criteria de la misma forma que las propiedades:

```
List results = session.createCriteria(Cat.class)
      .setProjection( Projections.projectionList()
            .add( Projections.rowCount() )
            .add( Projections.avg("weight") )
            .add( Projections.max("weight") )
            .add( Projections.groupProperty("color") )
      )
      .list();
```

Para limitar el número de elementos devueltos por un criteria se utiliza el método limit(), conjugado con el método setFirstResult() si se quiere empezar en un registro distinto del primero (0). El siguiente Criteria obtiene una lista con 5 instancias de la clase Cat, comenzando en el registro con índice 2 del conjunto de resultados.

```
List list = sess.createCriteria(Cat.class).
      setMaxResults(5).
      setFirstResult(2).
      list();
```

SQL

También se permite la utilización de consultas en SQL nativo desde Hibernate, lo cual resulta bastante útil en algunos casos como la utilización de sentencias DDL o ejecución de procedimientos.

La principal desventaja de este método es que al utilizar el dialecto SQL de una base de datos concreta, la portabilidad de la aplicación entre distintos tipos de Bases de datos se ve comprometida.

Las consultas SQL se especifican a través del método createSQLQuery:

```
List list = sess.createSQLQuery(
                        "SELECT ID, NAME, BIRTHDATE FROM
                        CATS").list();
Object[] res = (Object[]) list.get(0);
```

La lista obtenida como resultado estará formada por un Object[] con las instancias que representan a los campos especificados en la consulta. Se pueden especificar los tipos de las instancias devueltas de forma explícita a través del método addScalar:

```
List list = sess.createSQLQuery("SELECT * FROM CATS")
      .addScalar("ID", Hibernate.LONG)
      .addScalar("NAME", Hibernate.STRING)
      .addScalar("BIRTHDATE", Hibernate.DATE);
Object[] res = (Object[]) list.get(0);
```

Se puede añadir una entidad a una consulta de forma sencilla, para que la lista de resultados sea un conjunto de instancias de la clase concreta:

```
List list = sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM
CATS").addEntity(Cat.class);
Cat cat = (Cat) list.get(0);
```

Referencias

http://docs.jboss.org/hibernate/orm/3.5/reference/en/html/queryhql.html

http://docs.jboss.org/hibernate/orm/3.5/reference/en/html/querycriteria.html

http://docs.jboss.org/hibernate/orm/3.5/reference/en/html/querysql.html

Control de transacciones

Spring

Spring proporciona una capa de control de transacciones muy flexible y potente, que se integra perfectamente con sus abstracciones de acceso a los datos.

Existe un TransactionManager para cada plataforma encargado de la gestión específica de la misma, que permite ejecutar las operaciones básicas con transacciones (commit/rollback) de forma independiente al gestor de base de datos. El transaction manager es configurable en los siguientes aspectos:

- Propagación: ámbito e interacción de la transacción. Por defecto una sola transacción, crea una nueva si no existía

  @Transactional(propagation=Propagation=REQUIRED_NEW

- Aislamiento: capacidad de una transacción de ver los datos de otra. (Por defecto el que disponga la base de datos)

  @Transactional(isolation.SERIALIZABLE

  Tipos posibles:

  - ReadUncommited: nivel más bajo, las transacciones pueden leer los datos que han sido actualizados aunque no estén confirmados aún, es posible trabajar sobre copias no definitivas
  - ReadCommited: las transacciones ven todos los cambios una vez actualizados, es posible que durante la transacción se obtengan conjuntos de datos distintos
  - RepeteableRead: las transacciones solamente ven los cambios de otras transacciones, lecturas repetidas proporcionan los mismos datos
  - Serializable: nivel más alto, las transacciones se ejecutan una tras otra

- Timeout: tiempo máximo de una transacción.

  @Transactional(timeout=60

- Sólo lectura: indica si la transacción puede modificar. Por defecto RW

  @Transactional(readOnly=false

- Gestión de excepciones: define cómo se deben gestionar las excepciones. Por defecto commit para las checked y rollback para las runtime

  @Transactional(rollbackFor=…, noRollbackFor=…

Una importante característica del control de transacciones de Spring es que habitualmente se gestiona en la capa de servicio, por lo que los Dao no deben gestionar esta capacidad. En el caso de Hibernate por tanto se recomienda la utilización de HibernateDaoSupport.

Configuración

En application-context.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans.xsd
              http://www.springframework.org/schema/tx
              http://www.springframework.org/schema/tx/spring-tx.xsd
              http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

  <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/inventario" />
       <property name="username" value="root" />
       <property name="password" value="" />
  </bean>

  <tx:annotation-driven/>

  <bean id="sessionFactory"
       class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
       <property name="dataSource">
              <ref bean="dataSource" />
       </property>

       <property name="hibernateProperties">
              <props>
                 <prop key="hibernate.dialect">
                       org.hibernate.dialect.MySQLDialect
                 </prop>
              </props>
       </property>

       <property name="annotatedClasses">
              …
       </property>
  </bean>

  <bean id="transactionManager"
      class="org.springframework.orm.hibernate4.HibernateTransactionManager">
       <property name="sessionFactory" ref="sessionFactory" />
  </bean>
  …
```

Referencias

http://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html

Spring MVC

Introducción

Módulo encargado de la gestión de aplicaciones con el patrón MVC para Spring. Se emplea habitualmente en la construcción de aplicaciones web sustituyendo la utilización de Servlets Java por controladores que facilitan la integración con la arquitectura Spring.

DispatcherServlet

Cada solicitud que se realiza a una aplicación web Spring se encuentra centralizada en el DispatcherServlet. Esta clase Spring se encarga de la detección de la petición HTTP solicitada a través del cliente (habitualmente navegador), y de la activación de la clase que atenderá dicha acción HTTP, la cual viene determinada por una trayectoria concreta (RequestMapping).

```
@Autowired
private ProductoSvc svc;

@RequestMapping(value="/listar", method=RequestMethod.GET)
public String execute(Model model){
	try {
		model.addAttribute(ATT_LISTA, svc.listar());

		return "lista";
	} catch (Exception e) {
		model.addAttribute(ATT_ERROR, e);
		return "error";
	}
}
```

La configuración del DispatcherServlet deberá encontrarse en el archivo web.xml, indicando cuáles son las trayectorias para las que actúa:

```
	<servlet>
		<servlet-name>mvc-dispatcher</servlet-name>
		<servlet-class>
			org.springframework.web.servlet.DispatcherServlet
		</servlet-class>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>mvc-dispatcher</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>
```

En una aplicación Spring MVC encontramos tres archivos de configuración:

- web.xml
- application-context.xml
- mvc-dispatcher-servlet.xml: configuración del DispatcherServlet

En la siguiente configuración se definen dos aspectos importantes:

1. El paquete base de los controladores: todas las clases marcadas como @Controller se deben encontrar en ese paquete o subpaquetes del mismo.
2. La resolución de vistas de la aplicación web: en este caso JSP.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       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
        http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="es.hubiqus.spr.controller" />
  <mvc:annotation-driven />

  <bean
   class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix">
            <value></value>
      </property>
      <property name="suffix">
            <value>.jsp</value>
      </property>
  </bean>
</beans>
```

Referencias

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

http://www.baeldung.com/spring-mvc-tutorial

Controladores

Las clases encargadas de atender las solicitudes del cliente son los controladores.

La configuración de los controladores es automática, como único requerimiento en archivo XML cabe recordar el base-package en mvc-dispatcher-servlet.xml.

Mapeos

Cada método del controlador puede atender una solicitud HTTP, y viene determinada por la URL (RequestMapping) y el método concreto. Cada mapeo se puede definir a nivel de la clase, de forma que cada método se debe encargar de un tipo de método distinto, o de una trayectoria adicional de cada uno de ellos.

```
@Controller
@RequestMapping(value = "/listar")
public class ListaController {

   //Atiende a la URL /listar, método GET
   @RequestMapping(method=RequestMethod.GET)
   public String get(Model model) {…}

   //Atiende a GET /listar - no funcionará por duplicidad de mapping
   //Existen @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping
   @GetMapping()
   public String get2(Model model) {…}

   //Atiende a la URL /listar/filtro, método GET
   @GetMapping("/filtro")
   public String get3(Model model) {…}

   //Atiende a la URL /listar/orden, método GET
   @RequestMapping(value = "/orden", method=RequestMethod.GET)
   public String get4(Model model) {…}
```

En el propio mapeo es habitual la inclusión de patrones para la obtención de parámetros (PathVariable):

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public Producto get(@PathVariable Integer id) {
      Producto res = svc.buscar(id);
      return new ResponseEntity<Producto>(res, HttpStatus.OK);
}
```

Métodos

Los métodos empleados en los controladores pueden definir cualquier tipo de parámetro necesario, ya que en tiempo de ejecución es el motor de Spring MVC quien se encarga de inyectar todos los valores necesarios.

```
@RequestMapping(method=RequestMethod.POST)
public String guardar(@ModelAttribute Producto producto, Model model)

@RequestMapping(method=RequestMethod.GET)
public String buscar(@RequestParam int id, Model model)
```

Los parámetros habituales son:

- Model o ModelAndView: objeto de Spring MVC para el almacenamiento de atributos.
- ModelAttribute: representación de un objeto de la web, se emplea en aquellos métodos cuyo destino u origen es un formulario.
- BindingResult: control de errores.
- RequestParam: parámetro de la URL, si la URL no lo contiene o difiere en el tipo de datos la trayectoria se considera no válida.

Una alternativa para la utilización de beans necesarios para los métodos es emplear la autoinyección de atributos de la clase (@Autowired).

```
@RequestMapping(value="/listar", method=RequestMethod.GET)
public String execute(Model model){
      model.addAttribute(ATT_LISTA, svc.listar());
      return "lista";
}
```

Estos métodos habitualmente devuelven un String que será tomado como el destino tras la ejecución. Si la configuración de Spring MVC es con JSP, para el ejemplo anterior el destino sería por tanto la página lista.jsp.

También es posible efectuar forwards (`forward:/listar`) o redirecciones (`redirect:/listar`).

Referencias

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-controller

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-methods

Spring Tags

Las etiquetas de Spring se emplean en las vistas para interactuar con los controladores. Las etiquetas principales pertenecen a dos taglibs:

```
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

Formularios

Las etiquetas de tipo form facilitan la construcción de formularios HTML, de hecho en tiempo de ejecución en el servidor las etiquetas se sustituyen por su equivalente en HTML estándar.

```
<form:form modelAttribute="producto" action="guardar">
  <form:hidden path="id" />
  <form:input path="nombre" />
  <form:select path="tipo.id"
     items="${lista}" itemLabel="descripcion" itemValue="id"/>
  <input type="submit">
</form:form>
```

A destacar de esta etiqueta:

- El método por defecto es POST
- modelAttribute facilita la correspondencia entre el formulario y el bean en ambos sentidos (vista - controlador - vista).
- action determina el método que se ejecutará en un controlador. Es necesario especificar un parámetro de tipo @ModelAttribute en dicho método.

Respecto a las etiquetas internas a form:

- path realiza la correspondencia entre el componente y la propiedad del mismo nombre en modelAttribute.
- Se puede especificar cualquier tipo de atributo de componente HTML aunque no pertenezca a la definición de la etiqueta (type="number", required, …)

Internacionalización

Se refiere a la capacidad de responder a distintos idiomas:

```
<spring:message code="producto.nombre"/>
```

Cuando se accede a un código de internacionalización Spring accede a un fichero **properties** con las cadenas de caracteres en el idioma solicitado de forma automática por el navegador. Para ello es necesario configurar la ruta base en application-context.xml:

```
<bean id="messageSource"
   class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basename" value="classpath:messages" />
        <property name="defaultEncoding" value="LATIN1" />
</bean>
```

Para el acceso desde código se emplea autoinyección del messageSource:

```
@RequestMapping(value="/borrar", method=RequestMethod.GET)
public String borrar(@RequestParam int id, Model model, Locale locale){
        svc.eliminar(id);
        model.addAttribute("msg",
                        messageSource.getMessage("item.borrado", null, locale));
        return "lista";
}
```

Referencias

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-view-jsp

Control de errores

El control de errores es importante en la elaboración de un proyecto con varias capas. La capa de datos habitualmente tiene incluido su propio control de errores (anotaciones @Column) para evitar operar con la base de datos si la información no es correcta. La propia base de datos siempre contendrá las validaciones de integridad y corrección.

El esfuerzo de control de errores en una aplicación web se debe centrar en las capas de vista y controlador. La vista se suele validar con HTML o JavaScript. Para el controlador se puede realizar un control de error manual, devolviendo el mensaje de error correspondiente cuando la operación no sea correcta:

```
@RequestMapping(value="/borrar", method=RequestMethod.GET)
public String borrar(@RequestParam int id, Model model, Locale locale){
   try {
        svc.eliminar(id);
        return "lista";
   } catch (Exception e) {
        model.addAttribute("msg",
                        messageSource.getMessage("error.borrar", null, locale));
        return "error";
   }
}
```

En el caso de validación de formularios será necesario incluir anotaciones de tipo Hibernate Validator que se encargan de automatizar la detección de incorrecciones de formato o elemento vacío:

```
@Entity
public class Producto {

        private Integer id;
        private String nombre;
        private Integer unidades;

        @Id
        public Integer getId() {
                return id;
        }
        @NotEmpty
        public String getNombre() {
                return nombre;
        }
        @NotNull
        @Range(min=0)
        public Integer getUnidades() {
                return unidades;
        }

     …
```

Sobre el método invocado desde el formulario se incluye el control de validación sustituyendo la anotación @ModelAttribute por @Valid:

```
@RequestMapping(value = "/guardar", method=RequestMethod.POST)
public String execute(@Valid Producto producto, BindingResult result, Model model) {
      try {
            if (result.hasErrors()){
                  return "form";
            }else{
                svc.guardar(disco);
                return "lista";
            }
      } catch (Exception ex) {
            result.rejectValue("id", MSG_ERROR);
            return "form";
      }
}
```

Los errores de validación se asocian con el atributo de la clase validada. En el formulario se pueden extraer con una etiqueta:

```
<form:form modelAttribute="producto" action="guardar">
        <form:hidden path="id"/>
        <form:input path="nombre"/>
        <form:errors path="nombre" cssClass="error" />
        <form:input path="unidades" type="number" min="0"/>
        <form:errors path="unidades" cssClass="error" />
        <input type="submit"/>
        <form:errors path="id" cssClass="error" element="div" />
</form:form>
```

Los mensajes de validación se pueden personalizar en el archivo de cadenas:

```
NotEmpty.producto.nombre=Nombre requerido
NotNull.producto.unidades=Unidades requerido
Range.producto.unidades = Debe ser un valor mayor que {2}
typeMismatch.producto.unidades = Debe ser un valor numérico
```

En la conversión de tipos entre el formulario y el modelAttribute interviene el InitBinder:

```
@InitBinder
private void initBinder(WebDataBinder binder) {
   SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
   binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
}
```

Referencias

https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc

http://hibernate.org/validator/

Interceptores

Un interceptor se ejecuta como un filtro establecido antes y/o después de la actuación del DispatcherServlet.

Un interceptor habitual es el que se establece para el control de acceso de usuarios, determinando que el usuario está autenticado siempre que encuentre un objeto asociado en la sesión:

```
public class LoginInterceptor extends HandlerInterceptorAdapter{

  @Override
  public boolean preHandle(HttpServletRequest request,
                      HttpServletResponse response, Object handler) throws Exception {
       String uri = request.getRequestURI();
       Usuario usuario = (Usuario) request.getSession().getAttribute("sessionUser");

       if(usuario == null){
              if (!uri.endsWith("/login")){  //Autorizadas sin autenticar
                     //Redirigir al inicio en caso de acceso prohibido
                     response.sendRedirect(request.getContextPath() + INDEX);
                     return false;
              }
       }else{
              //Comprobar autorización
              …
              if (!auth){
                     response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                     response.sendRedirect(request.getContextPath() + INDEX);
              }
              return auth;
       }
       return true;
  }

}
```

El interceptor se agrega a la lista de interceptores en el fichero de configuración:

```
<mvc:interceptors>
   <bean
        class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
          <property name="paramName" value="lang" />
   </bean>
   <bean class="es.hubiqus.interceptor.LoginInterceptor"/>
</mvc:interceptors>
```

Referencias

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html

http://www.baeldung.com/spring-mvc-handlerinterceptor

Tiles framework

Se trata de un framework que favorece la construcción de layouts en portales web. Evita replicar todo el contenido HTML de referencia y estructura del portal en todas las páginas que lo componen.

La configuración del proyecto requiere varias dependencias:

```
  <properties>
       <org.apache.tiles.version>2.2.2</org.apache.tiles.version>
  </properties>

  <dependencies>
    <dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-core</artifactId>
        <version>${org.apache.tiles.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-jsp</artifactId>
        <version>${org.apache.tiles.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-api</artifactId>
        <version>${org.apache.tiles.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-servlet</artifactId>
        <version>${org.apache.tiles.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-template</artifactId>
        <version>${org.apache.tiles.version}</version>
    </dependency>
  </dependencies>
```

Para definir el layout es necesaria una página donde se definen las secciones que serán completadas por el framework en tiempo de ejecución:

```
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
<html>
<head>
</head>
<body>
    <tiles:insertAttribute name="header" />
    <div class="container">
        <tiles:insertAttribute name="body" />
    </div>
    <tiles:insertAttribute name="footer" />
</body>
</html>
```

Sin emplear tiles el resultado de cada método de controlador suele ser una página (JSP). En este caso el resultado será el nombre de un tile que corresponde con una configuración concreta de layout, donde se debe indicar cómo se rellena cada una de las secciones definidas. Estas definiciones de tiles se encuentran en el archivo tiles.xml y permiten establecer relación de herencia con un layout base:

```
<tiles-definitions>
    <definition name="baseLayout" template="/tiles/layout.jsp">
        <put-attribute name="header" value="/tiles/header.jsp" />
        <put-attribute name="body" value="" />
        <put-attribute name="footer" value="/tiles/footer.jsp" />
    </definition>
    <definition name="login" extends="baseLayout">
        <put-attribute name="body" value="/form.jsp" />
    </definition>
    <definition name="inicio" extends="baseLayout">
        <put-attribute name="body" value="/inicio.jsp" />
    </definition>
</tiles-definitions>
```

La configuración del tipo de vista que se empleará en el proyecto debe encontrarse definida en el archivo mvc-dispatcher-servlet.xml:

```
<bean id="tilesConfigurer"
       class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
     <property name="definitions">
          <list>
               <value>/WEB-INF/tiles.xml</value>
          </list>
     </property>
</bean>

<bean id="tilesViewResolver"
       class="org.springframework.web.servlet.view.UrlBasedViewResolver">
     <property name="viewClass"
               value="org.springframework.web.servlet.view.tiles2.TilesView" />
</bean>
```

Cabe destacar que en web.xml hay que determinar una excepción a los archivos de estilo para que no sean tratados por DispatcherServlet:

```
<servlet-mapping>
       <servlet-name>default</servlet-name>
       <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

Referencias

https://tiles.apache.org

Servicios Spring

Introducción

Spring es un framework completo que además de una API de programación ofrece una serie de servicios para la definición de la arquitectura e infraestructura de la aplicación. Entre dichos servicios destacan los siguientes:

- Spring Boot: facilita la configuración e inicio de aplicaciones Spring.
- Spring Cloud: arquitectura de servicios y escalabilidad.
- Spring Security: seguridad integrada.
- Spring Mock: testeo de aplicaciones Spring.

Spring Boot

Una aplicación Spring Boot se construye a partir de la configuración Maven o Gradle.

Para generar un proyecto de estas características se puede emplear el inicializador web indicando las necesidades de módulos concretos: https://start.spring.io

En el caso de Maven se genera un archivo pom.xml con la siguiente configuración añadida a la configuración principal del proyecto:

```
<!-- Hereda los valores por defecto de Spring Boot -->
<parent>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-parent</artifactId>
	<version>2.0.3.RELEASE</version>
	<relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
	<!-- Dependencia web de Spring Boot -->
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-web</artifactId>
	</dependency>
	<!-- Para los tests -->
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-test</artifactId>
		<scope>test</scope>
	</dependency>
</dependencies>

<build>
	<plugins>
		<!-- Configuración de Spring Boot -->
		<plugin>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-maven-plugin</artifactId>
		</plugin>
	</plugins>
</build>
```

En el caso de una aplicación web, la principal ventaja es que no requiere de una instancia de servidor iniciado (Tomcat), sino que lo gestiona internamente para ejecución o depuración a través de una clase ejecutable:

```
@SpringBootApplication
public class SprApplication {
	public static void main(String[] args) {
		SpringApplication.run(SprApplication.class, args);
	}
}
```

Referencias

https://spring.io/projects/spring-boot

Spring REST

La generación de una API REST resulta muy sencilla en Spring, ya sea a través de aplicaciones con Spring framework o con Spring Boot. Presenta distintas alternativas en la recepción de información, tipo de respuesta, …

Para generar un servicio REST se emplea un RestController:

```
//TODO control de errores
@RestController
@RequestMapping(value = "/producto")
public class ProductoRestController {

  @Autowired
  private ProductoSvc svc;

  @RequestMapping(value="/{id}", method=RequestMethod.GET)
  public ResponseEntity<Producto> get(@PathVariable Integer id) {
      Producto res = svc.buscar(id);
      return new ResponseEntity<Producto>(res, HttpStatus.OK);
  }

  @RequestMapping(method=RequestMethod.GET)
  public ResponseEntity<List<Producto>> get() {
      List<Producto> res = svc.listar();
      return new ResponseEntity<List<Producto>>(res, HttpStatus.OK);
  }

  @RequestMapping(method=RequestMethod.POST)
  public ResponseEntity<Producto> post(@RequestBody Producto item) {
      svc.guardar(item);
      return new ResponseEntity<>(HttpStatus.OK);
  }

  @RequestMapping(value="/{id}", method=RequestMethod.PUT)
  public ResponseEntity<Producto> put(@PathVariable Integer id,
                                      @RequestBody Producto item) {
      svc.modificar(item);
      return new ResponseEntity<Producto>(item, HttpStatus.OK);
  }

  @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
  public ResponseEntity<Producto> delete(@PathVariable Integer id) {
      svc.eliminar(id);
      return new ResponseEntity<>(HttpStatus.OK);
  }

}
```

El formato de salida por defecto será JSON.

Referencias

https://spring.io/guides/gs/rest-service/

http://www.baeldung.com/rest-with-spring-series/

Spring Boot MVC

La estructura de un proyecto web con Spring Boot es distinta a la habitual, los recursos de tipo web se almacenan en una carpeta de código **/src/main/webapp**.

La configuración de Spring MVC tiene un número mayor de requerimientos:

• La clase de inicio está basada en SpringBootServletInitializer:

```
@SpringBootApplication
public class SprApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder app) {
        return app.sources(SprApplication.class);
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SprApplication.class, args);
    }
}
```

• El archivo application.properties contiene la configuración de acceso a datos.

```
# viewResolver
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp
```

• Existe una clase de tipo @Configuration encargada de la inicialización del DispatcherServlet (sustituye a mvc-dispatcher-servlet.xml).

```
@Configuration
public class SprConfiguration implements WebMvcConfigurer{

	@Override
	public void configureViewResolvers(ViewResolverRegistry registry) {
		registry.jsp("/", ".jsp");
	}

	@Override
	public void addViewControllers(ViewControllerRegistry registry) {
	    registry.addViewController("/").setViewName("forward:/index.jsp");
	}

}
```

En esta clase también se configuran por ejemplo los aspectos relacionados con el idioma de la aplicación, interceptores, …:

```
@Bean
public LocaleResolver localeResolver() {
      SessionLocaleResolver localeResolver = new SessionLocaleResolver();
      localeResolver.setDefaultLocale(new Locale("es"));
      return localeResolver;
}

@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
      LocaleChangeInterceptor interceptor = new LocaleChangeInterceptor();
      interceptor.setParamName("lang");
      return interceptor;
}

@Bean
public MessageSource messageSource() {
      ReloadableResourceBundleMessageSource messageSource =
                                    new ReloadableResourceBundleMessageSource();
      messageSource.setBasename("classpath:messages");
      messageSource.setDefaultEncoding("LATIN1");
      return messageSource;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
      registry.addInterceptor(localeChangeInterceptor());
}
```

Al eliminar los archivos de configuración estándares de Spring, los beans se definen a través de anotaciones:

- @Bean: beans de configuración.
- @Service: servicios.
- @Repository: capa DAO.

En el caso de los repositorios, la implementación de Spring Boot incluye Daos autogenerados que permiten de forma automática ejecutar las operaciones básicas.

```
public interface UsuarioDao extends CrudRepository<Usuario, Integer> {}
```

No es necesario implementar un interfaz como el anterior porque Spring lo incluye de forma autogenerada en tiempo de ejecución. Tampoco será necesario incluir operaciones de consulta adicionales si están basadas en las propiedades del Bean:

```
public Usuario findByUserAndClave(String user, String clave);
public List<Usuario> findByNombre(String nombre);
```

Los métodos derivados tienen la forma: *opByField* donde op puede ser find, count o delete entre otros y Field cualquiera de los atributos de la clase (unidos por And o Or). También se puede incluir Distinct como parte del criterio de nombrado. Ejemplos: findByTituloAndAutor, findDistinctByAutor.

En el caso de necesitar operaciones adicionales la práctica habitual es generar un interfaz personalizado:

```
public interface UsuarioDaoCustom {
      public List<Usuario> buscarAnteriores(Date fechaCreacion);
}

public class UsuarioDaoImpl implements UsuarioDaoCustom {
      public List<Usuario> buscarAnteriores(Date fechaCreacion) {}
}

public interface UsuarioDao
     extends CrudRepository<Usuario, Integer>, UsuarioDaoCustom {
}
```

Dependencias adicionales del proyecto:

```
        <!-- Servidor incrustado -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-jasper</artifactId>
            <scope>provided</scope>
        </dependency>
        <!-- Hibernate/JPA -->
        <dependency>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-starter-data-jpa</artifactId>
         </dependency>
         <!-- Conector MySQL -->
         <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
             <groupId>javax.servlet</groupId>
             <artifactId>jstl</artifactId>
         </dependency>
```

Referencias

https://spring.io/guides/gs/serving-web-content/

http://www.baeldung.com/spring-mvc-tutorial

https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html

https://www.concretepage.com/spring-boot/spring-boot-crudrepository-example

Spring Cloud

Con Spring Cloud se pueden desarrollar y desplegar microservicios de forma sencilla.

Un microservicio (API Rest normalmente) realiza una operación bien definida y necesaria por diversos sistemas empresariales de distinta naturaleza. Una aplicación puede ser construida gracias al **descubrimiento** y utilización de estos microservicios, frente a la arquitectura monolítica habitual donde un mismo despliegue contiene todo el funcionamiento de la aplicación (independientemente del hecho de que ésta se encuentre construida en diversas capas).

Los microservicios presentan ventajas en la agilidad de desarrollo, despliegue, testeo… debido a tratarse de unidades de trabajo de mucha menor entidad. Las necesidades en este caso entonces se centran en los siguientes puntos, que deben estar disponibles de forma sencilla o incluso **automática**:

- Configuración: configuración bajo arquitecturas complejas. Ej: Feign.
- Visibilidad: los servicios deben ser localizables. Ej: Eureka, Zipkin.
- Depuración: depuración más compleja por el uso de entidades remotas. Ej: Hystrix.
- Escalabilidad: se adaptan a las necesidades del sistema. Ej: Ribbon.

Todas las características y tecnologías nombradas se pueden trabajar globalmente con Spring Cloud.

Comunicación entre servicios

Cuando un servicio existe en ejecución es necesario comunicarse con él. Un microservicio puede ser enlazado de múltiples formas:

- Conexión HTTP: emplear cualquier tipo de clase Java habilitada para la construcción de un cliente REST.

```
Map<String, String> uriVariables = new HashMap<>();
uriVariables.put("id", "17");

ResponseEntity<Producto> responseEntity = new RestTemplate().getForEntity(
                                "http://localhost:8000/api-productos/producto/{id}",
                                  Producto.class, uriVariables);
Producto res = responseEntity.getBody();
```

- Cliente automatizado: no necesario escribir el código de enlace en los métodos.

```
@FeignClient(name="productos-api", url="localhost:8000")
public interface ProductoSvcProxy {

  @RequestMapping(value="/producto/{id}", method=RequestMethod.GET)
  public Producto buscar(@PathVariable("id") Integer id);
}
```

- Localización automatizada: no resulta necesario especificar la URL del servidor (o servidores) donde actúa un microservicio, y además automatiza balance de carga.

```
//Con ribbon la lista de servidores se configura en application.properties
//Ribbon automatiza el balance de carga
@FeignClient(name="productos-api")
@RibbonClient(name="productos-api")
public interface ProductoSvcProxy {

  @RequestMapping(value="/producto/{id}", method=RequestMethod.GET)
  public Producto buscar(@PathVariable("id") Integer id);
}


productos-api.ribbon.listOfServers=localhost:8000,localhost:8001
```

Configuración adicional del proyecto:

```
        <dependencyManagement>
            <dependencies>
                <!-- Dependencia general para Spring Cloud -->
                 <dependency>
                     <groupId>org.springframework.cloud</groupId>
                     <artifactId>spring-cloud-dependencies</artifactId>
                     <version>${spring-cloud.version}</version>
                     <type>pom</type>
                     <scope>import</scope>
                 </dependency>
            </dependencies>
        </dependencyManagement>

        <dependencies>
                <!-- Dependencia web de Spring Boot -->
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
                <!-- Dependencia de Spring Cloud para Feign -->
                <dependency>
                      <groupId>org.springframework.cloud</groupId>
                      <artifactId>spring-cloud-starter-openfeign</artifactId>
                </dependency>
                 <!-- Dependencia de Spring Cloud para Ribbon -->
                 <dependency>
                      <groupId>org.springframework.cloud</groupId>
                      <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
                 </dependency>
        </dependencies>
```

Descubrimiento

Para no depender de una configuración específica de servidor y puerto, los servicios no deben ser accedido por URL directamente. La solución para esto es establecer un servicio de descubrimiento, donde los clientes de los micro servicios acuden para acceder a los mismos.

La generación de un servicio de descubrimiento con Eureka es muy sencilla:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

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

```
spring.application.name=netflix-eureka-naming-server
server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

La idea es que los servicios se registren de forma automática en Eureka, y sean localizados de la misma manera. Posee un panel de gestión en http://localhost:8761

Los clientes que se registran con Eureka simplemente tienen que incluir una línea en la clase ejecutable, tanto para registro como para descubrimiento:

```
@SpringBootApplication
@EnableDiscoveryClient
public class SvcApplication {

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

```
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Referencias

http://www.springboottutorial.com

http://projects.spring.io/spring-cloud/

https://cloud.spring.io/spring-cloud-pipelines/

Spring Security

Se trata de un framework para el control de autenticación y autorización de sistemas.

El control de autenticación se puede implementar de forma automática, sin ni siquiera especificar la vista de login.

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
               .antMatchers("/").permitAll() //Rutas permitidas
               .anyRequest().authenticated() //El resto no permitido sin autentificar
               .and()
            .formLogin() //Muestra formulario de inicio por defecto
                .permitAll() //Acceso a todo el mundo
                .and()
            .logout() //Acción de salida
                .permitAll(); //Acceso a todo el mundo
    }

    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        UserDetails user =
                 User.withDefaultPasswordEncoder() //Sustituir por autenticación JPA
                    .username("a")
                    .password("a")
                    .roles("USER")
                    .build();

         return new InMemoryUserDetailsManager(user);
    }
}
```

La clase anterior contiene la configuración de seguridad básica donde se especifican las rutas públicas, las rutas que requieren autenticación, reglas para roles o usuarios concretos…

El método userDetailsService se encarga de controlar los usuarios permitidos y los roles de acceso.

Configuración adicional del proyecto:

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

Configuración

Resulta habitual establecer una vista de login personalizada en la configuración de seguridad:

```
http
.formLogin()
.loginPage("/login") //Página de login personalizada
.permitAll()
```

La autenticación seguirá estando gestionada por Spring si se emplea la URL **login** y los nombres de campos **username** y **password**.

```
<form action="/login" method="post">
     <%-- Necesario para login Spring --%>
     <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
     <input type="text" name="username"/><br>
     <input type="password" name="password"/><br>
     <input type="submit"/><br>
</form>
```

Es necesario gestionar la configuración de la vista para Spring Mvc:

```
@Configuration
public class SprConfiguration implements WebMvcConfigurer{

       @Override
       public void configureViewResolvers(ViewResolverRegistry registry) {
              registry.jsp("/", ".jsp");
       }

       @Override
       public void addViewControllers(ViewControllerRegistry registry) {
           registry.addViewController("/").setViewName("forward:/index.jsp");
           registry.addViewController("/login").setViewName("login");
       }

}
```

La autenticación contra base de datos requiere definir una clase para contener los detalles del usuario y la programación de cómo se debe producir dicho control:

```
public class UsuarioDetails extends User implements UserDetails{

       public UsuarioDetails(Usuario usuario) {
              this.usuario = usuario;
       }

…
```

```
public class UsuarioSvcImpl implements UserDetailsService{

	@Autowired
	private UsuarioDao dao;

	@Override
	public UserDetails loadUserByUsername(String username)
				throws UsernameNotFoundException {
	    Usuario usuario = dao.findByUser(username);
	    if (usuario == null) {
		  throw new UsernameNotFoundException(username);
	    }

	    return new UsuarioDetails(usuario);
	}

}
```

Cabe destacar que para que el funcionamiento sea adecuado la base de datos debe contener claves de usuario encriptadas con el mismo método de comprobación (PasswordEncoder).

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

	@Autowired
	private UsuarioSvcImpl userDetailsService;

	@Bean
	public PasswordEncoder passwordEncoder() {
	    return new BCryptPasswordEncoder();
	}

	@Bean
	public DaoAuthenticationProvider authenticationProvider() {
	    DaoAuthenticationProvider authProvider =
					new DaoAuthenticationProvider();
	    authProvider.setUserDetailsService(userDetailsService);
	    authProvider.setPasswordEncoder(passwordEncoder());
	    return authProvider;
	}

	@Override
	protected void configure(HttpSecurity http) throws Exception {
	    http
		  .authenticationProvider(authenticationProvider())
		  .formLogin()
		  .loginPage("/login")
		  .permitAll();
	}

}
```

Taglib

Las etiquetas de Spring Security taglib se pueden emplear para acceder a los valores de seguridad que se establecen al emplear dicho módulo.

```
<%@taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>

<sec:authorize access="isAuthenticated()">
      <form action="logout" method="post">
         <%-- Necesario para logout Spring --%>
         <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
         <input type="submit"/>
      </form>
</sec:authorize>

<sec:authentication property="principal.username" />

<sec:authorize access="hasRole('Admin')">
      <a class="nav-link" href="../venta/listar">Ventas</a>
</sec:authorize>
```

Configuración adicional:

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

Referencias

https://spring.io/projects/spring-security

Spring Mock

Módulo Spring para el testeo de aplicaciones, que facilita:

- El acceso a la configuración de la aplicación desde los tests
- La generación de llamadas y chequeos de resultados en aplicaciones web
- Integración con autenticación

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class SprApplicationTests {

	@Autowired
	private MockMvc mockMvc;

	@Test
	public void index() throws Exception {
		mockMvc.perform(get("/")).andExpect(status().isOk());
	}

	@Test
	public void accesoSeguro() throws Exception {
		mockMvc.perform(get("/listar"))
		  .andExpect(status().is3xxRedirection())
		  .andExpect(redirectedUrlPattern("**/login"));
	}

	@Test
	public void accesoAutenticado() throws Exception {
		FormLoginRequestBuilder login = formLogin().user("a").password("a");
		mockMvc.perform(login).andExpect(authenticated().withUsername("a"));
	}

}
```

Se ejecutan como JUnit. Configuración adicional del proyecto:

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

Referencias

https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html