Herramientas para Hibernate

Bases de Datos - 38210

Master Oficial en Desarrollo de Aplicaciones y Servicios Web

Miquel Esplà Gomis
Armando Suarez Cueto

Bean Validation

Enlace al manual

Bean Validator 2.0

- Bean Validation 2.0 (JSR 380): marco estándar para validación
- Anotación del tipo de la usada en JPA para validar los contenidos de campos de clases
- Para Hibernate proporciona una doble ventaja:
  - mayor consistencia y control de los contenidos de las clases
  - menos accesos a la base de datos (se validan los datos antes de acceder a la BD)

```
@NotNull(message="Name cannot be null”)
@Length(min=1, max=75, message="Name must have length between 1 and 75.")
String name;
```

Validaciones más frecuentes

- ***@NotNull***: el valor no puede ser nulo
- ***@NotEmpty***: el valor no ser vacío (por ejemplo, una cadena vacía)
- ***@AssertTrue @AssertFalse***: comprueba que el valor es *true* o *false*, respectivamente
- ***@Size***: comprueba que el tamaño de la propiedad està entre los valores de los atributos de tipo String, Collection, Map, etc.
- ***@Min @Max:*** comprueba que el valor es, como mínimo o máximo, respectivamente, el que se especifica como valor

Validaciones más frecuentes

- ***@Email***: Valida el formato de un correo electrónico
- ***@Past @Future***: Comprueba que una fecha sea anterior o posterior a la actual, respectivamente
- ***@Positive @Negative***: Comprueba que un número es positivo o negativo
- ***@Pattern***: Permite definir una expresión regular para validar una propiedad
- Más etiquetas de validación en la sección correspondiente del manual

Dependencias para utilizar validación

Dependencias para la definición de restricciones de validación y gestión de los mensajes de error:

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.9.Final</version>
</dependency>

<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.el</artifactId>
    <version>3.0.0</version>
</dependency>
```

Gestión de violaciones

- Si se viola alguna de las condiciones a la hora de persistir una entidad, se produce un error de ejecución
- También se puede gestionar la validación manualmente:

```
ValidatorFactory  factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Student student = new Student();
...
Set<ConstraintViolation<Student>> violations = validator.validate(student);
```

Hibernate Search

Enlace al manual

¿Para qué sirve Hibernate Search?

- Proyecto semi-independiente de Hibernate
- Integra Lucene (o ElasticSearch) para búsquedas basadas en texto y no en SQL
- Proporciona una API similar a las de las consultas en Hibernate

Configuración de Hibernate Search

Tan solo hay que añadir una dependencia al pom.xml:

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-orm</artifactId>
   <version>5.9.1.Final</version>
</dependency>
```

Y la dirección donde Lucene debe guardar el índice de palabras:

```
<property name="hibernate.search.default.directory_provider">filesystem</property>
<property name="hibernate.search.default.indexBase">/var/lucene/indexes</property>
```

Configuración de Hibernate Search

```
@Entity
@Indexed
@Table(name = "Subject")
public class Subject {
    @Id
    private int id;

    @Field(termVector = TermVector.YES)
    private String name;
    ...
```

Hay que anotar los elementos sobre los que se realizarán búsquedas:

- las entidades con `@Indexed`
- las propiedades con `@Field`

Inicializando el índice de texto

- Cuando lancemos nuestra aplicación debemos inicializar el índice de términos y documentos
- A partir de este momento, Hibernate Search se encargará de mantenerlo actualizado

```
FullTextSession fullTextSession = Search.getFullTextSession(session);

fullTextSession.createIndexer().startAndWait();
```

El gestor de búsquedas de texto

- El objeto `FullTextSession` hace de interfaz con el índice de Lucene
- El objeto `QueryBuilder` permite crear objetos de consulta sobre una entidad

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction()

// Primero definimos nuestra consulta
QueryBuilder queryBuilder = fullTextSession.getSearchFactory().buildQueryBuilder()
  .forEntity(Subject.class)
  .get();
Query query= ...
// La convertimos en una consulta de Hibernate y la ejecutamos
List<Subject> result = fullTextSession.createFullTextQuery(query, Subject.class).list();

tx.commit()
```

Tipos de consulta

Coincidencia exacta de palabras:

```
org.apache.lucene.search.Query query = queryBuilder
  .keyword()
  .onField("name")
  .matching("Programación")
  .createQuery();
```

Tipos de consulta

Coincidencia parcial (*fuzzy*):

```
org.apache.lucene.search.Query query = queryBuilder
  .keyword()
  .fuzzy()
  .withEditDistanceUpTo(4)
  .onField("name")
  .matching("Programar")
  .createQuery();
```

La opción `withEditDistanceUpTo()` permite poner un límite de similitud entre palabras (si la distancia de edición es mayor que 4 los resultados no se mostrarán)

Tipos de consulta

Uso de caracteres comodín:

```
org.apache.lucene.search.Query query = queryBuilder
  .keyword()
  .wildcard()
  .onField("name")
  .matching("P*")
  .createQuery();
```

Todas las asignaturas que empiezan por P

Tipos de consulta

Búsquedas multipalabra:

```
org.apache.lucene.search.Query query = queryBuilder
  .phrase()
  .onField("nombre")
  .sentence("Programación 1")
  .createQuery();
```

Tipos de consulta

Búsqueda de rangos:

```
org.apache.lucene.search.Query query = queryBuilder
  .range()
  .onField("id")
  .from(2).to(25)
  .createQuery();
```

Tipos de consulta

Búsqueda de entidades similares:

```
org.apache.lucene.search.Query query = queryBuilder
  .moreLikeThis()
  .comparingField("name").boostedTo(10f)
  .andField("description").boostedTo(1f)
  .toEntity(asignatura)
  .createQuery();
```

Busca las asignaturas que se parecen a *asignatura* dando 10 veces más peso a las similitudes en el título (`boostedTo`) que en la descripción.

Tipos de consulta

Búsquedas combinadas:

```
org.apache.lucene.search.Query query = queryBuilder
  .bool()
  .must(queryBuilder.keyword()
  .onField("name").matching("Programación")
  .createQuery())
  .should(queryBuilder.range()
  .onField("id").from(5).to(25)
  .createQuery()).createQuery();
```

Combina dos consultas, una sobre los campos name y id. Must define una condicion imprescindible, mientras que should define una condición recomendable pero prescindible

Multi-tenancy

Enlace al manual

¿Qué es la multi-tenancy?

- Si nuestra aplicación está disponible para diversos propietarios (usuarios individuales, clientes, instituciones, empresas, etc.) y necesitamos que los datos sean específicos para cada uno (habitual en SaaS, por ejemplo: *Slack.com*) necesitamos una estrategia de multi-tenancy
- Ayuda al cumplimiento de la Ley de Protección de Datos, aumentando la seguridad en el acceso a los datos de cada propietario
- Posibles estrategias de multi-tenancy:
    - una base de datos para cada uno....
    - un esquema de la base de datos para cada uno...
    - particionar los datos dentro de la base de datos...

Estrategias multi-tenancy

- **Una base de datos para propietario**: más caro, más requisitos de espacio y gestión más cara por parte de la herramienta de gestión de la bases de datos
- **Un esquema por propietario en la misma base de datos**: algo menos caro, la gestión de la separación de los datos recae en la herramienta de gestión
- **Particionamiento de los datos**: menos recursos invertidos, pero requiere de la implementación ad-hoc a través del uso, por ejemplo, de columnas adicionales en las tablas donde se guarde el identificador de cada propietario

Estrategias multi-tenancy

- **Una base de datos para propietario**: más caro, más requisitos de espacio y gestión más cara por parte de la herramienta de gestión de la bases de datos
- **Un esquema por propietario en la misma base de datos**: algo menos caro, la gestión de la separación de los datos recae en la herramienta de gestión
- ~~**Particionamiento de los datos:** menos recursos invertidos, pero requiere de la implementación ad-hoc a través del uso, por ejemplo, de columnas adicionales en las tablas donde se guarde el identificador de cada propietario~~

Multi-tenancy en Hibernate

- Para implementar estrategias multi-tenancy en Hibernate se debe implementar dos interfaces:
  - `AbstractMultiTenantConnectionProvider`: gestiona las conexiones a la base de datos: cuando se crea una sesión se accede a la base de datos o esquema deseados
  - `CurrentTenantIdentifierResolver`: permite saber el identificador del propietario actual, de forma que la clase `AbstractMultiTenantConnectionProvider` sepa a qué base de datos o esquema acceder
- Ejemplo de implementación aquí

Multi-tenancy sin Hibernate

- Con las clases anteriores, Hibernate se encarga de realizar la conexión adecuada al llamar al método `getCurrentSession()` de la `SessionFactory`
- Sin este módulo, la tarea se debería realizar a mano, lo que requeriría:
  - obtener el ID del propietario (*tenant*) antes de realizar una conexión a la base de datos
  - mantener manualmente la información de cada propietario para poder crear las conexiones
  - realizar la conexión a mano, especificando la información de conexión a la base de datos
  - realizar las comprobaciones de seguridad pertinentes, como que el propietario que accede a la base de datos en una misma sesión no ha cambiado