Kostenloses eBook

LERNEN

hibernate

Free unaffiliated eBook created from
Stack Overflow contributors.

#hibernate

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: hibernate

It is an unofficial and free hibernate ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official hibernate.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit dem Winterschlaf

Bemerkungen

Die `SessionFactory` Bean ist für das Erstellen, `SessionFactory` , Schließen und Leeren aller Datenbanksitzungen verantwortlich, die der `TransactionManager` zur Erstellung `SessionFactory` . Deshalb automatisieren wir die SessionFactory `SessionFactory` in `SessionFactory` und führen alle Abfragen durch.

Eine der größten Fragen, die neue Benutzer von Hibernate stellen, lautet: "Wann werden meine Änderungen festgeschrieben?" Die Antwort ist sinnvoll, wenn Sie `SesisonFactory` , wie der `TransactionManager` mit der `SesisonFactory` . Ihre Datenbankänderungen werden gelöscht und festgeschrieben, wenn Sie die mit `@Transactional` kommentierte `@Transactional` . Der Grund dafür ist, dass eine Transaktion eine einzelne "Einheit" der ununterbrochenen Arbeit darstellen soll. Wenn mit dem Gerät etwas schief geht, wird davon ausgegangen, dass das Gerät ausgefallen ist und alle Änderungen rückgängig gemacht werden sollten. Die `SessionFactory` und `SessionFactory` die Sitzung, wenn Sie die ursprünglich aufgerufene Service-Methode `SessionFactory` .

Das bedeutet nicht, dass die Sitzung nicht gelöscht und gelöscht wird, während Ihre Transaktion läuft. Wenn ich beispielsweise eine Servicemethode aufrufe, um eine Sammlung von 5 Objekten hinzuzufügen und die Gesamtanzahl der Objekte in der Datenbank zurückgeben, würde die `SessionFactory` erkennen, dass die Abfrage (`SELECT COUNT(*)`) einen aktualisierten Status erfordert, und Dadurch würden die 5 Objekte vor dem Ausführen der Count-Abfrage gelöscht. Die Ausführung könnte ungefähr so aussehen:

Versionen

Ausführung	Dokumentationslink	Veröffentlichungsdatum
4.2.0	http://hibernate.org/orm/documentation/4.2/	2013-03-01
4.3.0	http://hibernate.org/orm/documentation/4.3/	2013-12-01
5.0.0	http://hibernate.org/orm/documentation/5.0/	2015-09-01

Examples

Verwenden der XML-Konfiguration zum Einrichten des Ruhezustands

Ich erstelle eine Datei namens `database-servlet.xml` irgendwo im Klassenpfad.

Anfangs sieht Ihre Konfigurationsdatei folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
tx-3.2.xsd">

</beans>
```

Sie werden feststellen, dass ich die Namespaces `tx` und `jdbc` Spring importiert habe. Das liegt daran, dass wir sie in dieser Konfigurationsdatei ziemlich stark verwenden werden.

Zuerst möchten Sie die Annotations-basierte Transaktionsverwaltung (`@Transactional`) `@Transactional` . Der Hauptgrund, aus dem die Benutzer den Ruhezustand im Frühling verwenden, besteht darin, dass Frühling alle Ihre Transaktionen für Sie verwaltet. Fügen Sie Ihrer Konfigurationsdatei die folgende Zeile hinzu:

```
<tx:annotation-driven />
```

Wir müssen eine Datenquelle erstellen. Die Datenquelle ist im Wesentlichen die Datenbank, in der Hibernate Ihre Objekte persistiert. Im Allgemeinen verfügt ein Transaktionsmanager über eine Datenquelle. Wenn Sie möchten, dass Hibernate mit mehreren Datenquellen kommuniziert, verfügen Sie über mehrere Transaktionsmanager.

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="" />
    <property name="url" value="" />
    <property name="username" value="" />
    <property name="password" value="" />
</bean>
```

Die Klasse dieser Bean kann alles sein, was `javax.sql.DataSource` implementiert, damit Sie Ihre eigene schreiben können. Diese Beispielklasse wird von Spring bereitgestellt, verfügt jedoch nicht über einen eigenen Thread-Pool. Eine beliebte Alternative ist die Apache Commons `org.apache.commons.dbcp.BasicDataSource` , aber es gibt noch viele andere. Ich werde jede der Eigenschaften unten erklären:

- *driverClassName* : Der Pfad zu Ihrem JDBC-Treiber. Dies ist eine **datenbankspezifische** JAR, die in Ihrem Klassenpfad verfügbar sein sollte. Stellen Sie sicher, dass Sie über die aktuellste Version verfügen. Wenn Sie eine Oracle-Datenbank verwenden, benötigen Sie einen Oracle-Treiber. Wenn Sie eine MySQL-Datenbank haben, benötigen Sie einen MySQLDriver. Sehen Sie hier , ob Sie den benötigten Treiber finden, aber ein schneller Google-Befehl sollte Ihnen den richtigen Treiber geben.

- *URL* : Die URL zu Ihrer Datenbank. Normalerweise handelt es sich dabei um `jdbc\:oracle\:thin\:\path\to\your\database` oder `jdbc:mysql://path/to/your/database` . Wenn

Sie nach dem Standardspeicherort der von Ihnen verwendeten Datenbank suchen, sollten Sie in der Lage sein, herauszufinden, wie dies aussehen soll. Wenn Sie eine `HibernateException` mit der Nachricht `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not set` und Sie diesem Leitfaden folgen. Es besteht eine Wahrscheinlichkeit von 90%, dass Ihre URL falsch ist, eine Chance von 5% dass Ihre Datenbank nicht gestartet ist und eine 5% ige Chance besteht, dass Ihr Benutzername / Passwort falsch ist.

- *Benutzername* : Der Benutzername, der bei der Authentifizierung bei der Datenbank verwendet werden soll.
- *Kennwort* : Das Kennwort, das bei der Authentifizierung bei der Datenbank verwendet werden soll.

Als nächstes müssen Sie die `SessionFactory` . Dies ist die Sache, die Hibernate verwendet, um Ihre Transaktionen zu erstellen und zu verwalten und tatsächlich mit der Datenbank zu kommunizieren. Es gibt einige Konfigurationsoptionen, die ich im Folgenden erläutern möchte.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="au.com.project />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.use_sql_comments">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
        </props>
    </property>
</bean>
```

- *dataSource* : Ihre Datenquellen-Bean. Wenn Sie die ID der dataSource geändert haben, legen Sie sie hier fest.
- *packagesToScan* : Die zu durchsuchenden Pakete, um Ihre mit JPA gekennzeichneten Objekte zu finden. Dies sind die Objekte, die von der Sitzungsfabrik verwaltet werden müssen. Sie werden im Allgemeinen als `@Entity` und mit `@Entity` . Weitere Informationen zum Einrichten von Objektbeziehungen in Hibernate finden Sie hier .
- *annotatedClasses* (nicht gezeigt): Sie können auch eine Liste von Klassen bereitstellen, die Hibernate durchsuchen soll, wenn sie nicht alle im selben Paket enthalten sind. Sie sollten entweder `packagesToScan` oder `annotatedClasses` jedoch nicht beides. Die Deklaration sieht folgendermaßen aus:

```
<property name="annotatedClasses">
    <list>
        <value>foo.bar.package.model.Person</value>
        <value>foo.bar.package.model.Thing</value>
    </list>
</property>
```

- *hibernateProperties* : Es gibt unzählige davon, die hier liebevoll dokumentiert sind . Die

wichtigsten, die Sie verwenden werden, lauten wie folgt:

- *hibernate.hbm2ddl.auto* : Eine der heißesten Hibernate-Fragen beschreibt diese Eigenschaft. Weitere Informationen finden Sie hier . Im Allgemeinen verwende ich validate und konfiguriere meine Datenbank entweder mit SQL-Skripts (für einen In-Memory) oder erstelle die Datenbank zuvor (vorhandene Datenbank).
- *hibernate.show_sql* : Boolean-Flag, wenn true, wenn der Wert für "Hibernate" alle SQL- `stdout` die von ihm generiert werden. Sie können Ihren Logger auch so konfigurieren, dass er Ihnen die Werte `log4j.logger.org.hibernate.type=TRACE` , die an die Abfragen gebunden sind, indem Sie `log4j.logger.org.hibernate.type=TRACE log4j.logger.org.hibernate.SQL=DEBUG` in Ihrem Protokollmanager (ich verwende log4j).
- *hibernate.format_sql* : Boolean-Flag, bewirkt, dass Hibernate Ihre SQL-Datei ziemlich stdout ausgibt.
- *hibernate.dialect* (aus gutem Grund nicht gezeigt): Viele alte Tutorials zeigen Ihnen, wie Sie den Hibernate-Dialekt einstellen, den er für die Kommunikation mit Ihrer Datenbank verwendet. Der Ruhezustand **kann** basierend auf dem von Ihnen verwendeten JDBC-Treiber automatisch erkennen, welcher Dialekt verwendet werden soll. Da es ungefähr 3 verschiedene Oracle-Dialekte und 5 verschiedene MySQL-Dialekte gibt, überlasse ich diese Entscheidung dem Hibernate. Eine vollständige Liste von Dialekten finden Sie hier .

Die letzten 2 Beans, die Sie deklarieren müssen, sind:

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"
    id="PersistenceExceptionTranslator" />

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

Der `PersistenceExceptionTranslator` übersetzt datenbankspezifische `HibernateException` oder `SQLExceptions` in Spring-Ausnahmen, die vom Anwendungskontext verstanden werden können.

Die `TransactionManager` Bean steuert sowohl die Transaktionen als auch das Rollback.

Hinweis: Sie sollten Ihre `SessionFactory` Bean automatisch in Ihre `SessionFactory` übernehmen.

XML-lose Hibernate-Konfiguration

Dieses Beispiel wurde von genommen hier

```
package com.reborne.SmartHibernateConnector.utils;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class LiveHibernateConnector implements IHibernateConnector {

    private String DB_DRIVER_NAME = "";
    private String DB_URL = "jdbc:h2:~/liveDB;MV_STORE=FALSE;MVCC=FALSE";
```

```
    private String DB_USERNAME = "sa";
    private String DB_PASSWORD = "";
    private String DIALECT = "org.hibernate.dialect.H2Dialect";
    private String HBM2DLL = "create";
    private String SHOW_SQL = "true";

    private static Configuration config;
    private static SessionFactory sessionFactory;
    private Session session;

    private boolean CLOSE_AFTER_TRANSACTION = false;

    public LiveHibernateConnector() {

        config = new Configuration();

        config.setProperty("hibernate.connector.driver_class",         DB_DRIVER_NAME);
        config.setProperty("hibernate.connection.url",                 DB_URL);
        config.setProperty("hibernate.connection.username",         DB_USERNAME);
        config.setProperty("hibernate.connection.password",         DB_PASSWORD);
        config.setProperty("hibernate.dialect",                     DIALECT);
        config.setProperty("hibernate.hbm2dll.auto",                 HBM2DLL);
        config.setProperty("hibernate.show_sql",                    SHOW_SQL);

        /*
         * Config connection pools
         */

        config.setProperty("connection.provider_class",
"org.hibernate.connection.C3P0ConnectionProvider");
        config.setProperty("hibernate.c3p0.min_size", "5");
        config.setProperty("hibernate.c3p0.max_size", "20");
        config.setProperty("hibernate.c3p0.timeout", "300");
        config.setProperty("hibernate.c3p0.max_statements", "50");
        config.setProperty("hibernate.c3p0.idle_test_period", "3000");


        /**
         * Resource mapping
         */

//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);

        sessionFactory = config.buildSessionFactory();
    }


    public HibWrapper openSession() throws HibernateException {
        return new HibWrapper(getOrCreateSession(), CLOSE_AFTER_TRANSACTION);
    }


    public Session getOrCreateSession() throws HibernateException {
        if (session == null) {
            session = sessionFactory.openSession();
        }
        return session;
    }
```

```
    public void reconnect() throws HibernateException {
        this.sessionFactory = config.buildSessionFactory();
    }


}
```

Bitte beachten Sie, dass dieser Ansatz mit dem neuesten Hibernate nicht gut funktioniert (Hibernate 5.2-Release lässt diese Konfiguration immer noch zu.)

Einfaches Beispiel für den Ruhezustand mithilfe von XML

Zum Einrichten eines einfachen Hibernate-Projekts mit XML für die Konfigurationen benötigen Sie 3 Dateien, hibernate.cfg.xml, einen POJO für jede Entität und eine EntityName.hbm.xml für jede Entität. Hier ist ein Beispiel für die Verwendung von MySQL:

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/DBSchemaName
   </property>
   <property name="hibernate.connection.username">
      testUserName
   </property>
   <property name="hibernate.connection.password">
      testPassword
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="HibernatePractice/Employee.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

DBSchemaName, testUserName und testPassword würden alle ersetzt. Stellen Sie sicher, dass Sie den vollständigen Ressourcennamen verwenden, wenn er sich in einem Paket befindet.

Mitarbeiter.java

```
package HibernatePractice;
```

```
public class Employee {
    private int id;
    private String firstName;
    private String middleName;
    private String lastName;

    public Employee(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getMiddleName(){
        return middleName;
    }
    public void setMiddleName(String middleName){
        this.middleName = middleName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Employee.hbm.xml

```
<hibernate-mapping>
   <class name="HibernatePractice.Employee" table="employee">
      <meta attribute="class-description">
         This class contains employee information.
      </meta>
      <id name="id" type="int" column="empolyee_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="middleName" column="middle_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Wenn sich die Klasse in einem Paket befindet, verwenden Sie erneut den vollständigen Klassennamen packageName.className.

Nachdem Sie diese drei Dateien erworben haben, können Sie den Ruhezustand in Ihrem Projekt verwenden.

Erste Schritte mit dem Winterschlaf online lesen:

https://riptutorial.com/de/hibernate/topic/907/erste-schritte-mit-dem-winterschlaf

Kapitel 2: Assoziationszuordnungen zwischen Entitäten

Examples

OneToMany-Verein

Um die Beziehung OneToMany zu veranschaulichen, benötigen wir 2 Einheiten, z. B. Land und Stadt. Ein Land hat mehrere Städte.

In der CountryEntity unterhalb definieren wir eine Menge von Städten für Land.

```
@Entity
@Table(name = "Country")
public class CountryEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "COUNTRY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           countryId;

   @Column(name = "COUNTRY_NAME", unique = true, nullable = false, length = 100)
   private String            countryName;

   @OneToMany(mappedBy="country", fetch=FetchType.LAZY)
   private Set<CityEntity> cities = new HashSet<>();

   //Getters and Setters are not shown
}
```

Nun die Stadteinheit.

```
@Entity
@Table(name = "City")
public class CityEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "CITY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           cityId;

   @Column(name = "CITY_NAME", unique = false, nullable = false, length = 100)
   private String            cityName;

   @ManyToOne(optional=false, fetch=FetchType.EAGER)
   @JoinColumn(name="COUNTRY_ID", nullable=false)
   private CountryEntity country;

   //Getters and Setters are not shown
```

```
}
```

Eine bis viele Assoziationen mit XML

Dies ist ein Beispiel dafür, wie Sie eine Eins-zu-viele-Zuordnung mithilfe von XML durchführen würden. Wir verwenden Autor und Buch als unser Beispiel und nehmen an, dass ein Autor möglicherweise viele Bücher geschrieben hat, aber jedes Buch wird nur einen Autor haben.

Autorenklasse:

```
public class Author {
    private int id;
    private String firstName;
    private String lastName;

    public Author(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Buchunterricht:

```
public class Book {
    private int id;
    private String isbn;
    private String title;
    private Author author;
    private String publisher;

    public Book() {
        super();
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getIsbn() {
```

```
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public Author getAuthor() {
        return author;
    }
    public void setAuthor(Author author) {
        this.author = author;
    }
    public String getPublisher() {
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
}
```

Author.hbm.xml:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
   <class name="Author" table="author">
      <meta attribute="class-description">
         This class contains the author's information.
      </meta>
      <id name="id" type="int" column="author_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Book.hbm.xml:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
    <class name="Book" table="book_title">
      <meta attribute="class-description">
         This class contains the book information.
      </meta>
        <id name="id" type="int" column="book_id">
            <generator class="native"/>
```

```
        </id>
        <property name="isbn" column="isbn" type="string"/>
        <property name="title" column="title" type="string"/>
         <many-to-one name="author" class="Author" cascade="all">
             <column name="author"></column>
         </many-to-one>
        <property name="publisher" column="publisher" type="string"/>
    </class>
</hibernate-mapping>
```

Die eins zu viele Verbindung besteht darin, dass die Book-Klasse einen Author enthält und die XML-Datei das <Many-to-one> -Tag enthält. Mit dem Kaskadenattribut können Sie festlegen, wie die untergeordnete Entität gespeichert / aktualisiert wird.

Assoziationszuordnungen zwischen Entitäten online lesen:
https://riptutorial.com/de/hibernate/topic/6165/assoziationszuordnungen-zwischen-entitaten

Kapitel 3: Benutzerdefinierte Benennungsstrategie

Examples

Erstellen und Verwenden eines benutzerdefinierten ImplicitNamingStrategy

Durch das Erstellen eines benutzerdefinierten `ImplicitNamingStrategy` können Sie optimieren, wie Hibernate nicht explizit benannten `Entity` , einschließlich Fremdschlüsseln, eindeutigen Schlüsseln, Bezeichnerspalten, Basisspalten usw., Namen zuweist.

Beispielsweise generiert Hibernate standardmäßig Fremdschlüssel, die gehasht werden und ähnlich aussehen:

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

Obwohl dies häufig kein Problem ist, möchten Sie vielleicht, dass der Name aussagekräftiger ist, z. B .:

```
FK_asset_tenant
```

Dies kann leicht mit einer benutzerdefinierten `ImplicitNamingStrategy` .

In diesem Beispiel wird `ImplicitNamingStrategyJpaCompliantImpl` . Sie können jedoch auch `ImplicitNamingStrategy` implementieren, wenn Sie möchten.

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.ImplicitForeignKeyNameSource;
import org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl;

public class CustomNamingStrategy extends ImplicitNamingStrategyJpaCompliantImpl {

    @Override
    public Identifier determineForeignKeyName(ImplicitForeignKeyNameSource source) {
        return toIdentifier("FK_" + source.getTableName().getCanonicalName() + "_" +
source.getReferencedTableName().getCanonicalName(), source.getBuildingContext());
    }

}
```

Um Hibernate mitzuteilen, welche `ImplicitNamingStrategy` verwendet werden soll, definieren Sie die Eigenschaft `hibernate.implicit_naming_strategy` in Ihrer Datei `persistence.xml` oder `hibernate.cfg.xml` wie folgt:

```
<property name="hibernate.implicit_naming_strategy"
                  value="com.example.foo.bar.CustomNamingStrategy"/>
```

Oder Sie können die Eigenschaft in der Datei `hibernate.properties` wie folgt angeben:

```
hibernate.implicit_naming_strategy=com.example.foo.bar.CustomNamingStrategy
```

In diesem Beispiel erhalten alle Fremdschlüssel, die keinen explizit definierten `name` haben, jetzt ihren Namen von `CustomNamingStrategy` .

Benutzerdefinierte physikalische Benennungsstrategie

Bei der Zuordnung unserer Entitäten zu Namen von Datenbanktabellen verlassen wir uns auf eine `@Table` Annotation. Wenn wir jedoch eine Namenskonvention für die Namen der Datenbanktabellen haben, können wir eine benutzerdefinierte Strategie für die physische Benennung implementieren, um dem Hibernate mitzuteilen, dass Tabellennamen basierend auf den Namen der Entitäten berechnet werden, ohne diese Namen explizit mit der `@Table` Annotation `@Table` . Gleiches gilt für die Zuordnung von Attributen und Spalten.

Unser Entitätsname lautet beispielsweise:

```
ApplicationEventLog
```

Und unser Tabellenname lautet:

```
application_event_log
```

Unsere physikalische Benennungsstrategie muss von Entitätsnamen, bei denen es sich um Kamel-Großbuchstaben handelt, in unsere Db-Tabellennamen, bei denen es sich um Schlangen handelt, konvertieren. Wir können dies erreichen, indem Sie `PhysicalNamingStrategyStandardImpl` von hibernate erweitern:

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;

public class PhysicalNamingStrategyImpl extends PhysicalNamingStrategyStandardImpl {

    private static final long serialVersionUID = 1L;
    public static final PhysicalNamingStrategyImpl INSTANCE = new
PhysicalNamingStrategyImpl();

    @Override
    public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    @Override
    public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    protected static String addUnderscores(String name) {
        final StringBuilder buf = new StringBuilder(name);
        for (int i = 1; i < buf.length() - 1; i++) {
```

```
            if (Character.isLowerCase(buf.charAt(i - 1)) &&
                    Character.isUpperCase(buf.charAt(i)) &&
                    Character.isLowerCase(buf.charAt(i + 1))) {
                buf.insert(i++, '_');
            }
        }
        return buf.toString().toLowerCase(Locale.ROOT);
    }
}
```

Wir überschreiben das Standardverhalten der Methoden `toPhysicalTableName` und `toPhysicalColumnName` , um unsere `toPhysicalColumnName` anzuwenden.

Um unsere benutzerdefinierte Implementierung verwenden zu können, müssen wir die Eigenschaft `hibernate.physical_naming_strategy` definieren und der Klasse `PhysicalNamingStrategyImpl` den Namen geben.

```
hibernate.physical_naming_strategy=com.example.foo.bar.PhysicalNamingStrategyImpl
```

Auf diese Weise können wir unseren Code aus den `@Table` und `@Column` Annotationen `@Column` , so dass unsere Entitätsklasse:

```
@Entity
public class ApplicationEventLog {
    private Date startTimestamp;
    private String logUser;
    private Integer eventSuccess;

    @Column(name="finish_dtl")
    private String finishDetails;
}
```

wird korrekt auf db table abgebildet:

```
CREATE TABLE application_event_log (
  ...
  start_timestamp timestamp,
  log_user varchar(255),
  event_success int(11),
  finish_dtl varchar(2000),
  ...
)
```

Wie im obigen Beispiel gezeigt, können wir den Namen des db-Objekts immer noch explizit `@Column(name="finish_dtl")` wenn es aus irgendeinem Grund nicht unserer allgemeinen Namenskonvention entspricht: `@Column(name="finish_dtl")`

Benutzerdefinierte Benennungsstrategie online lesen:
https://riptutorial.com/de/hibernate/topic/3051/benutzerdefinierte-benennungsstrategie

Kapitel 4: Caching

Examples

Aktivieren des Hibernate-Caching in WildFly

Um die Zwischenspeicherung der zweiten Ebene für den Ruhezustand in WildFly zu aktivieren, fügen Sie der `persistence.xml` Datei diese Eigenschaft hinzu:

```
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

Sie können Query Caching auch mit dieser Eigenschaft aktivieren:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

Für WildFly ist es nicht erforderlich, einen Cache-Anbieter zu definieren, wenn der Second Level-Cache von Hibernate aktiviert wird, da Infinispan standardmäßig verwendet wird. Wenn Sie möchten , dass ein alternativen Cache - Provider verwenden, jedoch können Sie dies mit der `hibernate.cache.provider_class` Eigenschaft.

Caching online lesen: https://riptutorial.com/de/hibernate/topic/3462/caching

Kapitel 5: Faules Laden gegen eifriges Laden

Examples

Faules Laden gegen eifriges Laden

Das Abrufen oder Laden von Daten kann hauptsächlich in zwei Typen unterteilt werden: eifrig und faul.

Um den Ruhezustand zu verwenden, fügen Sie die neueste Version zum Abschnitt "Abhängigkeiten" Ihrer Datei "pom.xml" hinzu:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.1.Final</version>
</dependency>
```

1. Eifriges Laden und faules Laden

Das erste, was wir hier besprechen sollten, ist, was faules Laden und eifriges Laden sind:

Eager Loading ist ein Entwurfsmuster, bei dem die Dateninitialisierung vor Ort erfolgt. Dies bedeutet, dass Sammlungen zum Zeitpunkt des Abrufs des übergeordneten Elements vollständig abgerufen werden (sofort abrufen).

Lazy Loading ist ein Entwurfsmuster, mit dem die Initialisierung eines Objekts bis zu dem Punkt verschoben wird, an dem es benötigt wird. Dies kann effektiv zur Leistung der Anwendung beitragen.

2. Verwenden der verschiedenen Arten des Ladens

Lazy Loading kann mit folgenden XML-Parametern aktiviert werden:

```
lazy="true"
```

Lassen Sie uns in das Beispiel eintauchen. Zuerst haben wir eine Benutzerklasse:

```
public class User implements Serializable {

    private Long userId;
    private String userName;
    private String firstName;
    private String lastName;
    private Set<OrderDetail> orderDetail = new HashSet<>();

    //setters and getters
    //equals and hashcode
    }
```

Schauen Sie sich den Satz von orderDetail an, den wir haben. Schauen wir uns nun die **OrderDetail-Klasse an** :

```
public class OrderDetail implements Serializable {

    private Long orderId;
    private Date orderDate;
    private String orderDesc;
    private User user;

    //setters and getters
    //equals and hashcode
}
```

Der wichtige Teil, der zum Einstellen des Lazy-Ladens in der `UserLazy.hbm.xml` :

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

So wird das Lazy Loading aktiviert. Um das Lazy-Loading zu deaktivieren, können Sie einfach Folgendes verwenden: `lazy = "false"` und dies wird das eifrige Laden ermöglichen. Das folgende Beispiel zeigt das Einrichten des Ladevorgangs in einer anderen Datei User.hbm.xml:

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="false" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

Umfang

Für diejenigen, die nicht mit diesen beiden Designs gespielt haben, liegt der Geltungsbereich von Faulheit und Eifer innerhalb einer bestimmten *SessionFactory*- **Sitzung** . *Eager* lädt alles sofort, dh Sie müssen nichts aufrufen, um es abzurufen. Faules Abrufen erfordert jedoch normalerweise eine Aktion, um eine zugeordnete Sammlung / ein Objekt abzurufen. Dies ist manchmal problematisch, wenn Sie sich außerhalb der *Sitzung* faul holen. Zum Beispiel haben Sie eine Ansicht, die die Details eines bestimmten POJOs zeigt.

```
@Entity
public class User {
    private int userId;
    private String username;
    @OneToMany
    private Set<Page> likedPage;

    // getters and setters here
}
```

```
@Entity
public class Page{
    private int pageId;
    private String pageURL;

    // getters and setters here
}

public class LazzyTest{
    public static void main(String...s){
        SessionFactory sessionFactory = new SessionFactory();
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        User user = session.get(User.class, 1);
        transaction.commit();
        session.close();

        // here comes the lazy fetch issue
        user.getLikedPage();
    }
}
```

Wenn Sie versuchen, **faul** außerhalb der *Sitzung* **abgerufen zu** werden, erhalten Sie die *lazyinitializeException* . Dies ist darauf zurückzuführen, dass die Abrufstrategie für alle oneToMany oder andere Beziehungen standardmäßig *faul ist* (Aufruf der DB bei Bedarf). Wenn Sie die Sitzung geschlossen haben, können Sie nicht mit der Datenbank kommunizieren. Unser Code versucht also, eine Sammlung von *gefallenen Seiten abzurufen,* und es wird eine Ausnahme *ausgelöst,* da für das Rendern der *Datenbank* keine zugeordnete Sitzung vorhanden ist.

Lösung hierfür ist zu verwenden:

1. Sitzung in Ansicht öffnen - Hier halten Sie die Sitzung auch in der gerenderten Ansicht offen.
2. `Hibernate.initialize(user.getLikedPage())` vor dem Schließen der Sitzung - Dies weist den Hibernate an, die Auflistungselemente zu initialisieren

Faules Laden gegen eifriges Laden online lesen:
https://riptutorial.com/de/hibernate/topic/7249/faules-laden-gegen-eifriges-laden

Kapitel 6: Hibernate Entity Relationships mit Anmerkungen

Parameter

Anmerkung	Einzelheiten
`@OneToOne`	Gibt eine Eins-zu-Eins-Beziehung zu einem entsprechenden Objekt an.
`@OneToMany`	Gibt ein einzelnes Objekt an, das vielen Objekten zugeordnet ist.
`@ManyToOne`	Gibt eine Auflistung von Objekten an, die einem einzelnen Objekt zugeordnet sind.
`@Entity`	Gibt ein Objekt an, das einer Datenbanktabelle zugeordnet ist.
`@Table`	Gibt an, welche Datenbanktabelle dieses Objekt ebenfalls abbildet.
`@JoinColumn`	Gibt an, in welcher Spalte ein Vorderschlüssel gespeichert ist.
`@JoinTable`	Gibt eine Zwischentabelle an, in der Fremdschlüssel gespeichert werden.

Examples

Bidirektional viele bis viele mit vom Benutzer verwaltetem Join-Tabellenobjekt

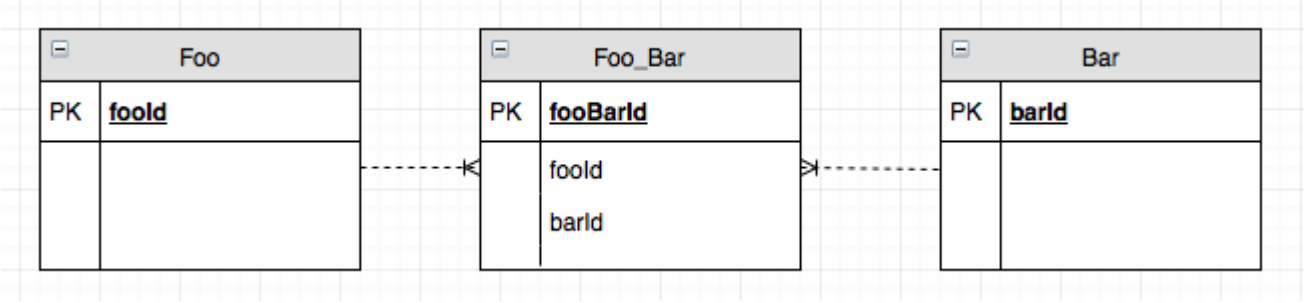
```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<FooBar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany(mappedBy = "foo")
    private List<FooBar> foos;
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;
```

```
    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId")
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

Gibt eine bidirektionale Beziehung zwischen vielen `Foo` Objekten und vielen `Bar` Objekten an, wobei eine vom Benutzer verwaltete Zwischenverbindungstabelle verwendet wird.

Die `Foo` Objekte werden als Zeilen in einer Tabelle namens `FOO` gespeichert. Die `Bar` Objekte werden als Zeilen in einer Tabelle namens `BAR` gespeichert. Die Beziehungen zwischen `Foo` und `Bar` Objekten werden in einer Tabelle mit dem Namen `FOO_BAR` gespeichert. Es gibt ein `FooBar` Objekt als Teil der Anwendung.

Wird häufig verwendet, wenn Sie zusätzliche Informationen zum Join-Objekt speichern möchten, z. B. das Datum, an dem die Beziehung erstellt wurde.



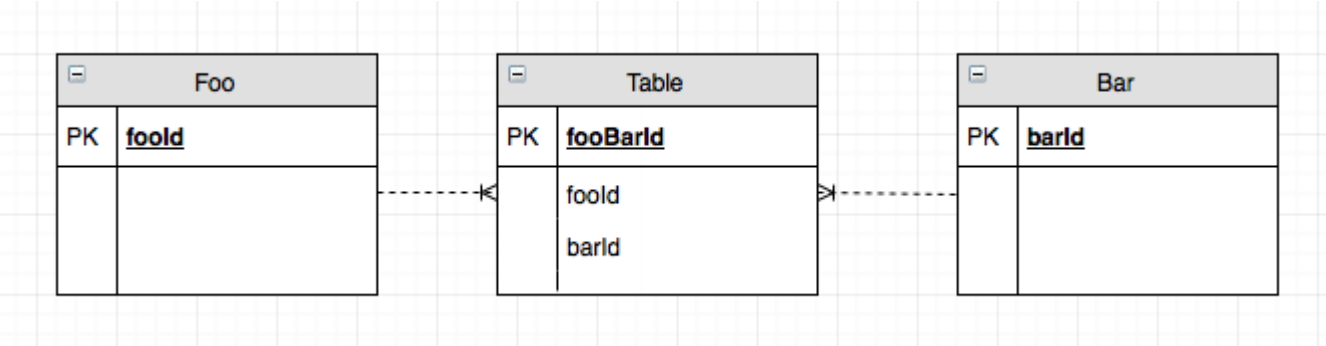
Bidirektional viele bis viele mit verwalteter Hibernate-Join-Tabelle

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId"))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="barId"),
        inverseJoinColumns = @JoinColumn(name="fooId"))
    private List<Foo> foos;
```

```
}
```

Gibt eine Beziehung zwischen vielen `Foo` Objekten zu vielen `Bar` Objekten mithilfe einer zwischengeschalteten Verknüpfungstabelle an, die Hibernate verwaltet.

Die `Foo` Objekte werden als Zeilen in einer Tabelle namens `FOO` gespeichert. Die `Bar` Objekte werden als Zeilen in einer Tabelle namens `BAR` gespeichert. Die Beziehungen zwischen `Foo` und `Bar` Objekten werden in einer Tabelle mit dem Namen `FOO_BAR` gespeichert. Dies impliziert jedoch, dass es kein `FooBar` Objekt als Teil der Anwendung gibt.

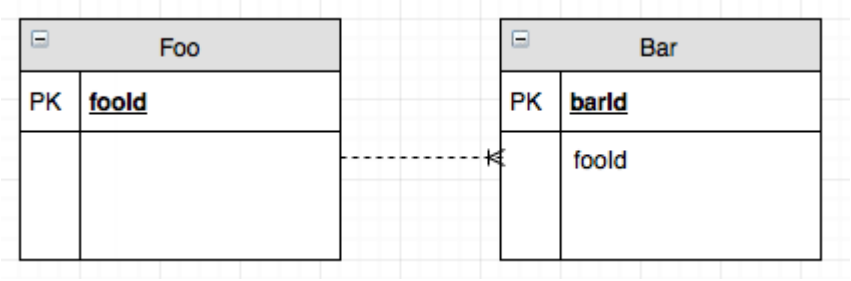


Bidirektionale Eins-zu-Viele-Beziehung unter Verwendung der Fremdschlüsselzuordnung

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;
}
```

Gibt eine bidirektionale Beziehung zwischen einem `Foo` Objekt und vielen `Bar` Objekten mithilfe eines Fremdschlüssels an.

Die `Foo` Objekte werden als Zeilen in einer Tabelle namens `FOO` gespeichert. Die `Bar` Objekte werden als Zeilen in einer Tabelle namens `BAR` gespeichert. Der Fremdschlüssel wird in der `BAR` Tabelle in einer Spalte mit der Bezeichnung `fooId` .



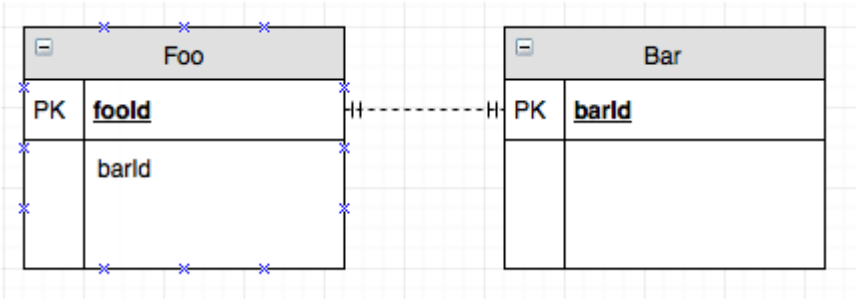
Bidirektionale Eins-zu-Eins-Beziehung, verwaltet von Foo.class

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "barId")
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToOne(mappedBy = "bar")
    private Foo foo;
}
```

Gibt eine bidirektionale Beziehung zwischen einem `Foo` Objekt und einem `Bar` Objekt mithilfe eines Fremdschlüssels an.

Die `Foo` Objekte werden als Zeilen in einer Tabelle namens `FOO` gespeichert. Die `Bar` Objekte werden als Zeilen in einer Tabelle namens `BAR` gespeichert. Der Fremdschlüssel wird in der `FOO` Tabelle in einer Spalte mit der Bezeichnung `barId` .

Beachten Sie, dass der Wert für `mappedBy` der Feldname im Objekt ist, nicht der Spaltenname.



Unidirektionale Eins-zu-Viele-Beziehung unter Verwendung der vom Benutzer verwalteten Join-Tabelle

```
@Entity
@Table(name="FOO")
public class Foo {
```

```
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId", unique=true))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    //No Mapping specified here.
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId", unique = true)
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

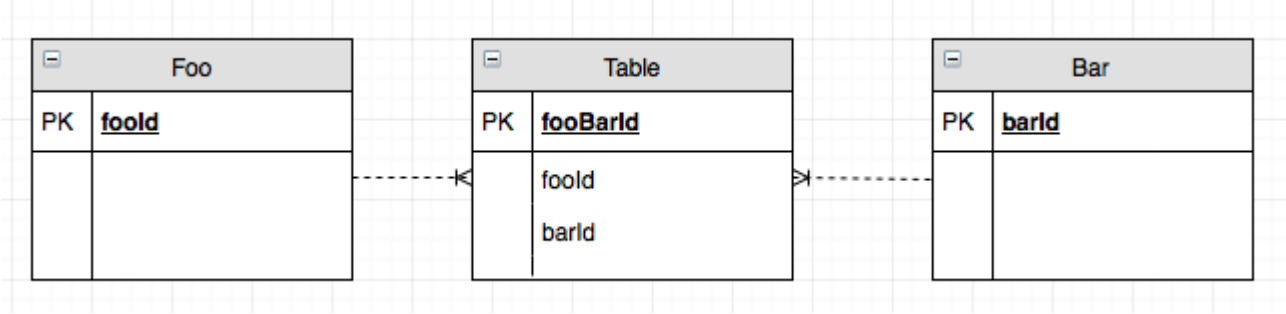
Gibt eine unidirektionale Beziehung zwischen einem `Foo` Objekt und vielen `Bar` Objekten mithilfe einer vom Benutzer verwalteten Zwischenverknüpfungstabelle an.

Dies ähnelt einer `ManyToMany` Beziehung. Wenn Sie dem Ziel-Fremdschlüssel jedoch eine `unique` Einschränkung hinzufügen, können Sie erzwingen, dass es sich um `OneToMany` .

Die `Foo` Objekte werden als Zeilen in einer Tabelle namens `FOO` gespeichert. Die `Bar` Objekte werden als Zeilen in einer Tabelle namens `BAR` gespeichert. Die Beziehungen zwischen `Foo` und `Bar` Objekten werden in einer Tabelle mit dem Namen `FOO_BAR` gespeichert. Es gibt ein `FooBar` Objekt als Teil der Anwendung.

Beachten Sie, dass es keine Zuordnung von `Bar` Objekten zu `Foo` Objekten gibt. `Bar` können frei bearbeitet werden, ohne die `Foo` Objekte zu beeinflussen.

Wird in Spring Security häufig verwendet, wenn Sie ein `User` einrichten, das über eine Liste von `Role` , die von ihm ausgeführt werden können. Sie können einem Benutzer Rollen hinzufügen und entfernen, ohne sich um das Löschen von `Role` zu kümmern.



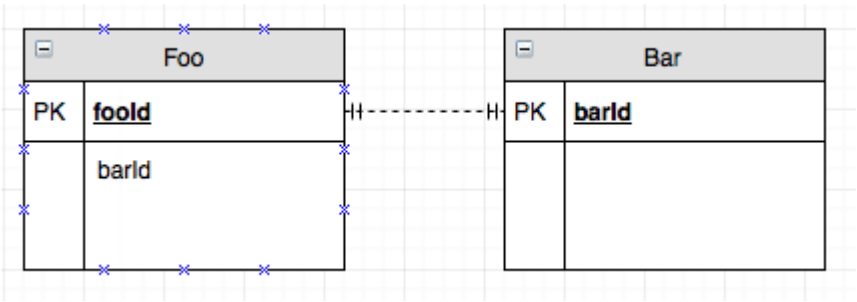
Unidirektionale Eins-zu-Eins-Beziehung

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;
    //No corresponding mapping to Foo.class
}
```

Gibt eine unidirektionale Beziehung zwischen einem `Foo` Objekt und einem `Bar` Objekt an.

Die `Foo` Objekte werden als Zeilen in einer Tabelle namens `FOO` gespeichert. Die `Bar` Objekte werden als Zeilen in einer Tabelle namens `BAR` gespeichert.

Beachten Sie, dass es keine Zuordnung von `Bar` Objekten zu `Foo` Objekten gibt. `Bar` können frei bearbeitet werden, ohne die `Foo` Objekte zu beeinflussen.



Hibernate Entity Relationships mit Anmerkungen online lesen:
https://riptutorial.com/de/hibernate/topic/5742/hibernate-entity-relationships-mit-anmerkungen

Kapitel 7: HQL

Einführung

HQL ist Hibernate-Abfragesprache, basiert auf SQL und wird hinter den Kulissen in SQL geändert, die Syntax unterscheidet sich jedoch. Sie verwenden Entitäts- / Klassennamen, nicht Tabellennamen und Feldnamen, nicht Spaltennamen. Es erlaubt auch viele Abkürzungen.

Bemerkungen

Bei der Verwendung von hql ist zu beachten, dass der Klassenname und die Feldnamen anstelle der in SQL üblichen Tabellen- und Spaltennamen verwendet werden.

Examples

Eine ganze Tabelle auswählen

```
hql = "From EntityName";
```

Wählen Sie bestimmte Spalten aus

```
hql = "Select id, name From Employee";
```

Fügen Sie eine Where-Klausel ein

```
hql = "From Employee where id = 22";
```

Beitreten

```
hql = "From Author a, Book b Where a.id = book.author";
```

HQL online lesen: https://riptutorial.com/de/hibernate/topic/9388/hql

Kapitel 8: Im Winterschlaf abholen

Einführung

Das Abrufen ist in JPA (Java Persistence API) sehr wichtig. In JPA werden HQL (Hibernate Query Language) und JPQL (Java Persistence Query Language) verwendet, um die Entitäten basierend auf ihren Beziehungen abzurufen. Obwohl es viel besser ist, als so viele Verknüpfungsabfragen und Unterabfragen zu verwenden, um mithilfe von nativem SQL zu erhalten, was wir wollen, wirkt sich die Strategie, wie wir die zugehörigen Entitäten in JPA abrufen, immer noch im Wesentlichen auf die Leistung unserer Anwendung aus.

Examples

Es wird empfohlen, FetchType.LAZY zu verwenden. Join holt die Spalten, wenn sie benötigt werden.

Nachfolgend finden Sie eine Employer Entity-Klasse, die dem Tabellen-Arbeitgeber zugeordnet ist. Wie Sie sehen, habe ich **fetch = FetchType.LAZY** anstelle von fetch = FetchType.EAGER verwendet. Der Grund, warum ich LAZY verwende, ist, dass der Arbeitgeber später viele Eigenschaften hat und jedes Mal, wenn ich nicht alle Felder eines Arbeitgebers kennen muss. Daher führt das Laden aller von ihnen zu einer schlechten Leistung, wenn ein Arbeitgeber geladen wird.

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

        @Column(name = "name")
        private String Name;

        @OneToMany(mappedBy = "employer", fetch = FetchType.LAZY,
               cascade = { CascadeType.ALL }, orphanRemoval = true)
        private List<Employee> employees;

        public Long getId() {
            return id;
        }

        public void setId(Long id) {
            this.id = id;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
```

```
            this.name = name;
        }

        public List<Employee> getEmployees() {
            return employees;
        }

        public void setEmployees(List<Employee> employees) {
            this.employees = employees;
        }
    }
```

Für von LAZY abgerufene Assoziationen führen jedoch nicht initialisierte Proxys manchmal zu LazyInitializationException. In diesem Fall können Sie JOIN FETCH einfach in HQL / JPQL verwenden, um LazyInitializationException zu vermeiden.

```
SELECT Employer employer FROM Employer
       LEFT JOIN FETCH employer.name
       LEFT JOIN FETCH employer.employee employee
       LEFT JOIN FETCH employee.name
       LEFT JOIN FETCH employer.address
```

Im Winterschlaf abholen online lesen: https://riptutorial.com/de/hibernate/topic/9475/im-winterschlaf-abholen

Kapitel 9: Kriterien und Projektionen

Examples

Liste mit Einschränkungen

Angenommen, wir haben eine TravelReview-Tabelle mit Städtenamen als Spalte "Titel"

```
 Criteria criteria =
    session.createCriteria(TravelReview.class);
  List review =
    criteria.add(Restrictions.eq("title", "Mumbai")).list();
  System.out.println("Using equals: " + review);
```

Wir können den Kriterien Einschränkungen hinzufügen, indem wir sie wie folgt verketten:

```
List reviews = session.createCriteria(TravelReview.class)
   .add(Restrictions.eq("author", "John Jones"))
   .add(Restrictions.between("date",fromDate,toDate))
   .add(Restrictions.ne("title","New York")).list();
```

Projektionen verwenden

Wenn wir nur einige Spalten abrufen möchten, können Sie dazu die Klasse Projections verwenden. Mit dem folgenden Code wird beispielsweise die Titelspalte abgerufen

```
 // Selecting all title columns
  List review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.property("title"))
    .list();
  // Getting row count
  review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.rowCount())
    .list();
  // Fetching number of titles
  review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.count("title"))
    .list();
```

Verwenden Sie Filter

`@Filter` wird als `WHERE` Camp verwendet, hier einige Beispiele

Studenteneinheit

```
@Entity
@Table(name = "Student")
public class Student
{
    /*...*/
```

```
    @OneToMany
    @Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and
s.id = study_id)")
    Set<StudentStudy> studies;

    /* getters and setters methods */
}
```

Lerneinheit

```
@Entity
@Table(name = "Study")
@FilterDef(name = "active")
@Filter(name = "active", condition="state = true")
public class Study
{
    /*...*/

    @OneToMany
    Set<StudentStudy> students;

    @Field
    boolean state;

    /* getters and setters methods */
}
```

StudentStudy-Entität

```
@Entity
@Table(name = "StudentStudy")
@Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and s.id
= study_id)")
public class StudentStudy
{
    /*...*/

    @ManytoOne
    Student student;

    @ManytoOne
    Study study;

    /* getters and setters methods */
}
```

Auf diese Weise wird jedes Mal, wenn der "aktive" Filter aktiviert ist,

-Jede Abfrage, die wir an der Studenteneinheit durchführen, wird **ALLE** Studenten mit **NUR** ihrem `state = true` Studien zurückgeben

-Jede Abfrage, die wir für die Study-Entität durchführen, gibt **ALL** `state = true` Studien zurück

-Jede Abfrage, die wir an der StudentStudy-Entity durchführen, gibt **NUR** die mit einer `state = true` Studienbeziehung zurück

Bitte beachten Sie, dass study_id der Name des Felds in der SQL StudentStudy-Tabelle ist

Kriterien und Projektionen online lesen: https://riptutorial.com/de/hibernate/topic/3939/kriterien-und-projektionen

Kapitel 10: Leistungsoptimierung

Examples

Verwenden Sie keinen EAGER-Abruftyp

Der Ruhezustand kann zwei Arten des Abrufs verwenden, wenn Sie die Beziehung zwischen zwei Entitäten `EAGER` : `EAGER` und `LAZY` .

Im Allgemeinen ist der `EAGER` keine gute Idee, da er JPA `EAGER` , die Daten *immer* abzurufen, selbst wenn diese Daten nicht erforderlich sind.

Beispiel: Wenn Sie eine `Person` und die Beziehung zu `Address` wie `Address` lautet:

```
@Entity
public class Person {

  @OneToMany(mappedBy="address", fetch=FetchType.EAGER)
  private List<Address> addresses;

}
```

Jedes Mal, wenn Sie eine `Person` abfragen, wird auch die Liste der `Address` dieser `Person` zurückgegeben.

Anstatt also Ihre Entität zuzuordnen mit:

```
@ManyToMany(mappedBy="address", fetch=FetchType.EAGER)
```

Benutzen:

```
@ManyToMany(mappedBy="address", fetch=FetchType.LAZY)
```

Eine andere Sache, die zu beachten ist, ist die Beziehung `@OneToOne` und `@ManyToOne` . Beide sind *standardmäßig* EAGER. Wenn Sie sich also Gedanken über die Leistung Ihrer Anwendung machen, müssen Sie den Abruf für diese Art von Beziehung festlegen:

```
@ManyToOne(fetch=FetchType.LAZY)
```

Und:

```
@OneToOne(fetch=FetchType.LAZY)
```

Verwenden Sie Komposition statt Vererbung

Der Winterschlaf hat einige Vererbungsstrategien. Der `JOINED` Vererbungstyp führt einen JOIN

zwischen der `JOINED` Entität und der übergeordneten Entität durch.

Das Problem bei diesem Ansatz besteht darin, dass der Ruhezustand **immer** die Daten aller beteiligten Tabellen in die Vererbung einbringt.

Wenn Sie beispielsweise die Entitäten `Bicycle` und `MountainBike` mit dem Vererbungstyp `JOINED` verwenden:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Bicycle {

}
```

Und:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class MountainBike extends Bicycle {

}
```

Jede JPQL-Abfrage, die auf `MountainBike` trifft, liefert die `Bicycle` und erstellt eine SQL-Abfrage wie:

```
select mb.*, b.* from MountainBike mb JOIN Bicycle b ON b.id = mb.id WHERE ...
```

Wenn Sie ein anderes übergeordnetes `Bicycle` für das `Bicycle` (z. B. " `Transport`), werden bei dieser obigen Abfrage auch die Daten dieses übergeordneten Elements abgerufen und ein zusätzlicher JOIN durchgeführt.

Wie Sie sehen, ist dies auch eine Art `EAGER` Mapping. Sie haben nicht die Wahl, nur die Daten der `MountainBike` Tabelle mit dieser Vererbungsstrategie mitzubringen.

Das Beste für die Leistung ist Komposition statt Vererbung.

Um dies zu erreichen, können Sie die `MountainBike` Entität einem Feldfahrrad `bicycle` :

```
@Entity
public class MountainBike {

    @OneToOne(fetchType = FetchType.LAZY)
    private Bicycle bicycle;

}
```

Und `Bicycle` :

```
@Entity
public class Bicycle {

}
```

Jede Abfrage bringt jetzt standardmäßig nur die `MountainBike` Daten mit.

Leistungsoptimierung online lesen:
https://riptutorial.com/de/hibernate/topic/2326/leistungsoptimierung

Kapitel 11: Native SQL-Abfragen

Examples

Einfache Abfrage

Angenommen, Sie haben einen Handle für das Hibernate `Session` Objekt, in diesem Fall mit dem Namen `session` :

```
List<Object[]> result = session.createNativeQuery("SELECT * FROM some_table").list();
for (Object[] row : result) {
    for (Object col : row) {
        System.out.print(col);
    }
}
```

Dadurch werden alle Zeilen in `some_table` abgerufen, in die `result some_table` und jeder Wert gedruckt.

Beispiel, um ein eindeutiges Ergebnis zu erhalten

```
Object pollAnswered = getCurrentSession().createSQLQuery(
        "select * from TJ_ANSWERED_ASW where pol_id = "+pollId+" and prf_log =
'"+logid+"'").uniqueResult();
```

Mit dieser Abfrage erhalten Sie ein eindeutiges Ergebnis, wenn Sie wissen, dass das Ergebnis der Abfrage immer eindeutig ist.

Wenn die Abfrage mehr als einen Wert zurückgibt, wird eine Ausnahme angezeigt

    org.hibernate.NonUniqueResultException

Sie können auch die Details in diesem Link hier mit mehr Beschreibung überprüfen

Stellen Sie daher sicher, dass Sie wissen, dass die Abfrage ein eindeutiges Ergebnis liefert

Native SQL-Abfragen online lesen: https://riptutorial.com/de/hibernate/topic/6978/native-sql-abfragen

Kapitel 12: SQL-Protokoll aktivieren / deaktivieren

Bemerkungen

Das Protokollieren dieser Abfragen ist **langsam** , sogar langsamer als normalerweise der Ruhezustand. Es beansprucht außerdem viel Speicherplatz. Verwenden Sie die Protokollierung nicht in Szenarien, in denen Leistung erforderlich ist. Verwenden Sie dies nur zum Testen der Abfragen, die Hibernate tatsächlich generiert.

Examples

Verwenden einer Protokollierungskonfigurationsdatei

Stellen Sie in der Protokollierungskonfigurationsdatei Ihrer Wahl die Protokollierung der folgenden Pakete auf die angezeigten Stufen ein:

```
# log the sql statement
org.hibernate.SQL=DEBUG
# log the parameters
org.hibernate.type=TRACE
```

Es werden wahrscheinlich einige für die Protokollierung spezifische Präfixe benötigt.

Log4j config:

```
log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE
```

Spring Boot- `application.properties` :

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
```

Logback `logback.xml` :

```
<logger name="org.hibernate.SQL" level="DEBUG"/>
<logger name="org.hibernate.type" level="TRACE"/>
```

Verwenden der Eigenschaften für den Ruhezustand

Dadurch wird das generierte SQL angezeigt, jedoch nicht die in den Abfragen enthaltenen Werte.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
```

```
    <property name="hibernateProperties">
        <props>
            <!-- show the sql without the parameters -->
            <prop key="hibernate.show_sql">true</prop>
            <!-- format the sql nice -->
            <prop key="hibernate.format_sql">true</prop>
            <!-- show the hql as comment -->
            <prop key="use_sql_comments">true</prop>
        </props>
    </property>
</bean>
```

Aktivieren / Deaktivieren des SQL-Protokolls im Debugging

Einige Anwendungen, die Hibernate verwenden, erzeugen beim Starten der Anwendung eine große Menge an SQL. Manchmal ist es besser, das SQL-Protokoll an bestimmten Stellen beim Debuggen zu aktivieren / deaktivieren.

Um dies zu aktivieren, führen Sie einfach diesen Code in Ihrer IDE aus, wenn Sie die Anwendung debuggen:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.DEBUG)
```

Etwas deaktivieren:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.OFF)
```

SQL-Protokoll aktivieren / deaktivieren online lesen:
https://riptutorial.com/de/hibernate/topic/3548/sql-protokoll-aktivieren---deaktivieren

Kapitel 13: Winterschlaf und JPA

Examples

Beziehung zwischen Hibernate und JPA

Hibernate ist eine Implementierung des JPA- Standards. Alles, was dort gesagt wird, gilt auch für Hibernate.

Der Ruhezustand hat einige Erweiterungen von JPA. Die Einrichtung eines JPA-Anbieters ist auch anbieterspezifisch. Dieser Dokumentationsabschnitt sollte nur die Besonderheiten des Ruhezustands enthalten.

Winterschlaf und JPA online lesen: https://riptutorial.com/de/hibernate/topic/6313/winterschlaf-und-jpa

Kapitel 14: Zuordnung von Assoziationen

Examples

Eins zu Eins Hibernate-Mapping

Jedes Land hat ein Kapital. Jede Hauptstadt hat ein Land.

Country.java

```
package com.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "countries")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "national_language")
    private String nationalLanguage;

    @OneToOne(mappedBy = "country")
    private Capital capital;

    //Constructor

    //getters and setters

   }
```

Capital.java

```
package com.entity;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
```

```
@Table(name = "capitals")
public class Capital {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    private long population;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "country_id")
    private Country country;

    //Constructor

    //getters and setters

}
```

HibernateDemo.java

```
package com.entity;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateDemo {

public static void main(String ar[]) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Country india = new Country();
    Capital delhi = new Capital();
    delhi.setName("Delhi");
    delhi.setPopulation(357828394);
    india.setName("India");
    india.setNationalLanguage("Hindi");
    delhi.setCountry(india);
    session.save(delhi);
    session.close();
  }

}
```

Zuordnung von Assoziationen online lesen:
https://riptutorial.com/de/hibernate/topic/6478/zuordnung-von-assoziationen

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit dem Winterschlaf	Community, JamesENL, Michael Piefel, Naresh Kumar, Reborn, user7491506
2	Assoziationszuordnungen zwischen Entitäten	StanislavL, user7491506
3	Benutzerdefinierte Benennungsstrategie	Mitch Talmadge, Naresh Kumar, veljkost
4	Caching	Mitch Talmadge
5	Faules Laden gegen eifriges Laden	BELLIL, Pramod, Pritam Banerjee, vicky
6	Hibernate Entity Relationships mit Anmerkungen	Aleksei Loginov, JamesENL
7	HQL	Daniel Käfer, user7491506
8	Im Winterschlaf abholen	rObOtAndChalie
9	Kriterien und Projektionen	Saifer, Sameer Srivastava
10	Leistungsoptimierung	Dherik, Michael Piefel
11	Native SQL-Abfragen	Daniel Käfer, Nathaniel Ford, Sandeep Kamath
12	SQL-Protokoll aktivieren / deaktivieren	Daniel Käfer, Dherik, JamesENL, Michael Piefel
13	Winterschlaf und JPA	Michael Piefel
14	Zuordnung von Assoziationen	Dherik, omkar sirra