

Hibernate

Notes for Professionals

Chapter 1: Getting started with Hibernate

Version	Documentation Link	Release Date
4.2.0	http://hibernate.org/hibernate4/hibernate4.2.0/	2013-03-01
4.3.0	http://hibernate.org/hibernate4/hibernate4.3.0/	2013-12-01
5.0.0	http://hibernate.org/hibernate5/hibernate5.0.0/	2015-09-01

Section 1.1: Using XML Configuration to set up Hibernate

Create a file called `database-very-let-act` somewhere on the desktop. Initially your config file will look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<hibernate-configuration>
  <session-factory>
    <name>hibernate</name>
    <provider>org.hibernate.service.jdbc.internal.StandardJdbcConnectionPoolImpl$Builder</provider>
    <property name="url" value="jdbc:hsqldb:mem://hibernate" />
    <property name="dialect" value="org.hibernate.dialect.HSQLDialect" />
    <property name="connection.pool.provider" value="org.hibernate.service.jdbc.internal.StandardJdbcConnectionPoolImpl$Builder" />
    <property name="connection.pool.provider.configuration" value="org.hibernate.service.jdbc.internal.StandardJdbcConnectionPoolImpl$Builder" />
    <property name="connection.pool.provider.configuration" value="org.hibernate.service.jdbc.internal.StandardJdbcConnectionPoolImpl$Builder" />
  </session-factory>
</hibernate-configuration>
```

You'll notice I imported the `tx` and `hib` Spring namespaces. This is because we are using Spring's transaction management. First thing you want to do is enable annotation based transaction management in your configuration file:

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

We need to create a data source. The data source is basically the database that your objects. Generally one transaction manager will have one data source. If you have multiple transaction managers, you'll need to create multiple data sources.

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource" />
<property name="driverClassName" value="org.hsqldb.jdbcDriver" />
<property name="url" value="jdbc:hsqldb:mem://hibernate" />
<property name="username" value="sa" />
<property name="password" value="" />
```

The class of this bean can be anything that implements `javax.sql.DataSource`. The class provided by Spring, but doesn't have its own thread pool. A professional's example class is `org.springframework.jdbc.datasource.DriverManagerDataSource`. Commons.org, apache.commons.dbcp, BasicDataSource, but there are many others. The explanation of this is available on the internet.

- `driverClassName`: The path to your JDBC driver. This is a **database specific** JAR that should be available on your classpath.

Chapter 2: Fetching in Hibernate

Fetching is really important in JPA (Java Persistence API). In JPA, HQL (Hibernate Query Language) and JPQL (Java Persistence Query Language) are used to fetch the entities based on their relationships. Although it is way better than using so many joining queries and sub-queries to get what we want by using native SQL, the strategy how we fetch the associated entities in JPA are still essentially affecting the performance of our application.

Section 2.1: It is recommended to use FetchType.LAZY. Join fetch the columns when they are needed

Below is an `Employer` entity class which is mapped to the table `employer`. As you can see I used `fetch = FetchType.LAZY` instead of `fetch = FetchType.EAGER`. The reason I am using LAZY is because `Employer` may have a lot of properties (one on and every one) may not need to know all the fields of an `Employer`, so loading all of them will lead a bad performance than an `employer` is loaded.

```
@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;
    }
}
```

However, for LAZY fetched associations, uninitialized proxies are sometimes leads to `LazyInitializationException`. In this case, we can simply use `JOIN FETCH` in the HQL/JPQL to avoid `LazyInitializationException`.

Chapter 8: Custom Naming Strategy

Section 8.1: Creating and Using a Custom ImplicitNamingStrategy

Creating a custom `ImplicitNamingStrategy` allows you to tweak how Hibernate will assign names to non-explicitly named entity attributes, including Foreign Keys, Unique Keys, Identifier Columns, Basic Columns, and more. For example, by default, Hibernate will generate Foreign Keys which are hashed and look similar to:

```
FK_asset_tenant
```

While this is often not an issue, you may wish that the name was more descriptive, such as:

```
FK_asset_tenant
```

This can easily be done with a custom `ImplicitNamingStrategy`.

This example extends the `DefaultImplicitNamingStrategy` class, however you may choose to implement `ImplicitNamingStrategy` if you wish.

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.ImplicitForeignKeyNameSource;
import org.hibernate.boot.model.naming.ImplicitNamingStrategy;
import org.hibernate.boot.model.naming.ImplicitNamingStrategyJmsPackageIdentifier;

public class CustomNamingStrategy extends ImplicitNamingStrategyJmsPackageIdentifier {

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

To tell Hibernate which `ImplicitNamingStrategy` to use, define the `hibernate.implicit_naming_strategy` property in your persistence.xml or hibernate.cfg.xml file as below:

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

Or you can specify the property in `hibernate.properties` file as below:

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

In this example, all Foreign Keys which do not have an explicitly defined name will now get their name from the `CustomNamingStrategy`.

Section 8.2: Custom Physical Naming Strategy

When mapping our entities to database table names we rely on a `@Table` annotation. But if we have a naming convention for our database table names, we can implement a custom physical naming strategy in order to tell Hibernate to calculate table names based on the names of the entities, without explicitly stating those names with `@Table` annotation. Some goes for attributes and columns mapping.

30+ pages
of professional hints and tricks

Contents

About	1
Chapter 1: Getting started with Hibernate	2
Section 1.1: Using XML Configuration to set up Hibernate	2
Section 1.2: Simple Hibernate example using XML	4
Section 1.3: XML-less Hibernate configuration	6
Chapter 2: Fetching in Hibernate	8
Section 2.1: It is recommended to use FetchType.LAZY. Join fetch the columns when they are needed	8
Chapter 3: Hibernate Entity Relationships using Annotations	10
Section 3.1: Bi-Directional Many to Many using user managed join table object	10
Section 3.2: Bi-Directional Many to Many using Hibernate managed join table	11
Section 3.3: Bi-directional One to Many Relationship using foreign key mapping	12
Section 3.4: Bi-Directional One to One Relationship managed by Foo.class	12
Section 3.5: Uni-Directional One to Many Relationship using user managed join table	13
Section 3.6: Uni-directional One to One Relationship	14
Chapter 4: HQL	15
Section 4.1: Selecting a whole table	15
Section 4.2: Select specific columns	15
Section 4.3: Include a Where clause	15
Section 4.4: Join	15
Chapter 5: Native SQL Queries	16
Section 5.1: Simple Query	16
Section 5.2: Example to get a unique result	16
Chapter 6: Mapping associations	17
Section 6.1: One to One Hibernate Mapping	17
Chapter 7: Criterias and Projections	19
Section 7.1: Use Filters	19
Section 7.2: List using Restrictions	20
Section 7.3: Using Projections	20
Chapter 8: Custom Naming Strategy	21
Section 8.1: Creating and Using a Custom ImplicitNamingStrategy	21
Section 8.2: Custom Physical Naming Strategy	21
Chapter 9: Caching	24
Section 9.1: Enabling Hibernate Caching in WildFly	24
Chapter 10: Association Mappings between Entities	25
Section 10.1: One to many association using XML	25
Section 10.2: OneToMany association	27
Chapter 11: Lazy Loading vs Eager Loading	28
Section 11.1: Lazy Loading vs Eager Loading	28
Section 11.2: Scope	29
Chapter 12: Enable/Disable SQL log	31
Section 12.1: Using a logging config file	31
Section 12.2: Using Hibernate properties	31
Section 12.3: Enable/Disable SQL log in debug	31
Chapter 13: Hibernate and JPA	33

Section 13.1: Relationship between Hibernate and JPA	33
Chapter 14: Performance tuning	34
Section 14.1: Use composition instead of inheritance	34
Credits	35
You may also like	36

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:

<https://goalkicker.com/HibernateBook>

This *Hibernate Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Hibernate group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Hibernate

Version	Documentation Link	Release Date
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

Section 1.1: Using XML Configuration to set up Hibernate

I create a file called `database-servlet.xml` somewhere on the classpath.

Initially your config file will look like this:

```
<?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>
```

You'll notice I imported the `tx` and `jdbc` Spring namespaces. This is because we are going to use them quite heavily in this config file.

First thing you want to do is enable annotation based transaction management (`@Transactional`). The main reason that people use Hibernate in Spring is because Spring will manage all your transactions for you. Add the following line to your configuration file:

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

We need to create a data source. The data source is basically the database that Hibernate is going to use to persist your objects. Generally one transaction manager will have one data source. If you want Hibernate to talk to multiple data sources then you have multiple transaction managers.

```
<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>
```

The class of this bean can be anything that implements `javax.sql.DataSource` so you could write your own. This example class is provided by Spring, but doesn't have its own thread pool. A popular alternative is the Apache Commons `org.apache.commons.dbcp.BasicDataSource`, but there are many others. I'll explain each of the properties below:

- `driverClassName`: The path to your JDBC driver. This is a **database specific** JAR that should be available on

[Click here to download full PDF material](#)