

3 Hibernate – ein einfaches Beispiel

In diesem Kapitel steigen wir anhand eines kleinen Beispiels in das Thema Hibernate ein. Als Beispiel verwenden wir einen elektronischen Terminkalender, mit dem persönliche und Gruppentermine verwaltet werden können. Wir konzentrieren uns (natürlich) auf die Fachlogik und Datenhaltung und betrachten die Benutzungsoberfläche allenfalls am Rande.

Als Datenbank verwenden wir in diesem Kapitel die H2. H2 lässt sich ganz einfach installieren und konfigurieren. Außerdem erlaubt sie den Betrieb der Datenbank im Hauptspeicher (In-Memory-DB) ohne Sicherung auf der Festplatte. Sie ist daher zum Testen und Ausprobieren sehr gut geeignet.

3.1 Installation von Hibernate und H2

Hibernate ist ein Open-Source-Produkt. Es kann also ohne Zahlungsverpflichtungen verwendet werden. Außerdem ist der Quellcode öffentlich. Das ist insbesondere dann sehr nützlich, wenn man der genauen Funktionsweise von Hibernate auf den Grund gehen will oder muss.

Hibernate kann von der Hibernate-Homepage <http://www.hibernate.org> heruntergeladen werden. Hier sind immer mehrere Versionen verfügbar – ältere, aktuelle und solche, die zurzeit noch entwickelt werden. Wir verwenden in diesem Buch Hibernate 3 und laden deshalb das aktuelle Hibernate-Release der Version 3 oder höher herunter.

Hibernate wird in einem gepackten Format angeboten (zip oder tar.gz sind verfügbar). Diese Archive enthalten neben den für die Entwicklung mit Hibernate benötigten Bibliotheken im Java-typischen JAR-Format auch den Quellcode und die Dokumentation zu Hibernate. Wir entpacken Hibernate in ein eigenes Verzeichnis – z.B. `c:\Programme\Hibernate` –, um es nicht mit unseren jeweiligen Projekten zu vermischen.

Hibernate

Verzeichnisstruktur der
Hibernate-Installation

Hiernach sind folgende Unterverzeichnisse im Zielverzeichnis zu finden

Verzeichnisse in der Hibernate-Installation	
Verzeichnis	Inhalt
bin	Enthält Startskripte für die Hibernate-Werkzeuge zum Exportieren und Aktualisieren von Datenbankschemata.
doc	Enthält die Dokumentation zu Hibernate. Hier befinden sich die API-Dokumentation im JavaDoc-HTML-Format und die offizielle Hibernate-Referenz im HTML- und PDF-Format.
eg	Enthält ein einfaches Beispiel zur Verwendung von Hibernate.
etc	Enthält Beispiele zur Konfiguration verschiedener Datenbanken in Hibernate, zum Deployment von Hibernate in JBoss und zum Konfigurieren der unterstützten Caches in Hibernate.
grammar	Enthält Grammatiken im ANTLR ^a -Format, welche die Hibernate Query Language (kurz HQL; siehe hierzu Kapitel 7), SQL und die Transformation von HQL zu SQL beschreiben.
lib	Enthält eine Reihe von Bibliotheken für den Einsatz von Hibernate in verschiedenen Kontexten. Wir werden jeweils einen Teil dieser Bibliotheken verwenden und immer explizit machen, welche dieser Bibliotheken für das Nachvollziehen der einzelnen Beispiele notwendig sind.
src	Enthält den Hibernate-Quellcode. Der Quellcode ist interessant bei der Fehlersuche und erleichtert die Arbeit mit dem Rahmenwerk in einer Entwicklungsumgebung, da diese dann zu den Signaturen der Methoden auch die Namen der Parameter anzeigen kann. Außerdem ist der Quellcode interessant, wenn wir herausfinden wollen, wie bestimmte Aspekte von Hibernate implementiert sind.
test	Enthält die JUnit ^b -Testklassen für Hibernate. Die Testklassen eines Rahmenwerkes können häufig aufzeigen, wie die Entwickler sich dessen Verwendung vorgestellt haben.

- a. ANTLR (kurz für Another Tool for Language Recognition) ist ein Werkzeug zum Erzeugen von Parsern. Für die Verwendung von Hibernate und somit auch für die Lektüre dieses Buches ist ein vertieftes Verständnis dieses Gebiets nicht notwendig.
- b. JUnit ist eine Bibliothek, welche die Entwicklung automatisch ablaufender Tests unterstützt. Für eine Einführung ins Testen mit JUnit empfehlen wir das Buch »Software-Tests mit JUnit« von Johannes Link (siehe [Link05]).

H2 Hibernate kann mit allen gängigen Datenbanken verwendet werden (MySQL, Postgres, Oracle und DB2 seien hier als Beispiele genannt). In den Beispielen für dieses Buch verwenden wir die H2¹, die von <http://www.h2database.com/> heruntergeladen werden kann. Die H2 ist eine einfache, in Java implementierte Datenbank. Sie eignet sich sehr gut zum Entwickeln von Prototypen, da ihre Konfiguration unkompliziert ist und sie schnell gestartet und gestoppt werden kann. Auch die

H2 sollte in einem eigenen Verzeichnis – z.B. `c:\Programme\H2` – installiert werden.

Das H2-Archiv enthält folgende Verzeichnisse:

*Verzeichnisstruktur der
H2-Installation*

Verzeichnisse in der H2-Installation	
Verzeichnis	Inhalt
bin	Enthält das <code>h2.jar</code> , das die Datenbank und den JDBC-Treiber enthält. Außerdem ist hier ein Startskript zum Ausführen der H2 unter Windows enthalten.
docs	Enthält die API-Dokumentation im JavaDoc-HTML-Format und das H2-Handbuch im PDF- und HTML-Format.
odbc	Enthält ODBC-Treiber für die H2. Diese sind für uns irrelevant.
src	Enthält den Quellcode der H2.

3.2 Vorbereitung der Entwicklungsumgebung

In diesem Abschnitt zeigen wir, wie die Entwicklungsumgebung für die Entwicklung einer Hibernate-Anwendung vorbereitet werden soll. Wir gehen davon aus, dass der Leser weiß, wie ein Projekt in seiner Entwicklungsumgebung angelegt wird.

3.2.1 Die Auswahl der richtigen Bibliotheken

Die Masse der Bibliotheken, die mit Hibernate geliefert werden, ist auf den ersten Blick überwältigend. Man könnte einfach alle mitgelieferten Bibliotheken in das Projekt einbinden. Das verlangsamt die Entwicklungsumgebung aber nur unnötig und vergrößert den Umfang des auszuliefernden Installationspakets. Letzteres stört spätestens, wenn die Anwendung in den Produktivbetrieb geht.

Wir werden im Verlauf dieses Buches verschiedene Varianten unseres Terminplaner-Beispiels entwickeln, die zum Teil unterschiedliche Bibliotheken benötigen. Wir werden dann jeweils klarmachen, welche Bibliotheken erforderlich sind. Für den Einstieg benötigen wir die folgenden Bibliotheken:

1. In der ersten Auflage dieses Buches verwendeten wir die HSQLDB. Die H2 ist zur HSQLDB ähnlich, bietet jedoch einige zusätzliche Features. Bei der Verwendung der HSQLDB mit Hibernate kann es vorkommen, dass Daten nach dem Beenden der Anwendung nicht gesichert werden. Dieses Problem tritt mit der H2 nicht auf.

Benötigte Bibliotheken für das erste Beispiel	
JAR-Datei ^a	Beschreibung
h2.jar	Die H2. Dieses JAR befindet sich in der H2-Installation.
hibernate3.jar	Diese Bibliothek enthält alle Hibernate-Klassen.
cglib-X.jar	Im Gegensatz zu anderen OR-Mapping-Lösungen wie zum Beispiel JDO verändert Hibernate nicht den Quelltext oder die kompilierten Klassen, sondern erzeugt zur Laufzeit dynamische Proxys. Diese Proxys werden mit der CGLib erzeugt.
asm.jar, asm-attrs.jar	Bibliothek zum Verändern des Bytecodes von Klassen. Diese Bibliothek wird von der CGLib verwendet.
ehcache-X.jar	Hibernate verwendet einen Caching-Mechanismus für bereits in einer Session bekannte persistente Objekte. Hibernate kann mit mehreren Caching-Bibliotheken zusammenarbeiten. EHCache ist eine davon.
commons-logging-X.jar	Ein Wrapper um Log4J und andere Logging-Lösungen unter Java, der von EHCache verwendet wird.
jta.jar	Die Standard-Java-Transaktions-API wird von Hibernate intern zum Verwalten von Transaktionen verwendet.
log4j-X.jar	Hibernate verwendet Log4J für alle Debug- und Fehlermeldungen.
commons-collections-X.jar	Eine Bibliothek mit zusätzlichen Collection-Klassen.
jdbc2_0-sdtext.jar	Erweiterungen zu JDBC2, die von Hibernate verwendet werden.
antlr-X.jar	ANTLR wird von Hibernate zum Parsen von HQL-Ausdrücken verwendet.
dom4j-X.jar, xml-apis.jar, xerces-X.jar	Einige XML-Bibliotheken, die von Hibernate zum Lesen der Konfigurations- und Mapping-Dateien verwendet werden. Wird Hibernate mit dem JDK 5.0 oder einer neueren Version verwendet, so ist nur das dom4j-X.jar notwendig. Die anderen beiden sind bereits in den neueren Versionen des JDKs enthalten. →

Benötigte Bibliotheken für das erste Beispiel	
JAR-Datei ^a	Beschreibung
junit.jar	Wir werden Unit-Tests schreiben. Dafür benötigen wir JUnit, das von der JUnit-Website (http://www.junit.org) heruntergeladen werden kann. Einige Entwicklungsumgebungen wie zum Beispiel Eclipse bieten an, JUnit in das Projekt einzubinden, sobald die erste Testklasse geschrieben wird. Hier muss JUnit dann nicht extra heruntergeladen werden.

- a. Das X in einigen JAR-Namen steht für die Versionsnummer der jeweiligen Bibliothek. Diese ändert sich für einige Bibliotheken von einer Hibernate-Version zur nächsten.

3.2.2 Installation der Plugins für den Zugriff auf die Datenbank

Bei der Arbeit mit einer Datenbank ist es manchmal notwendig, sich deren aktuellen Inhalt anzusehen. Bei der Verwendung einer OR-Mapping-Lösung ist das sogar noch wichtiger, als wenn die Datenbank direkt über JDBC verwendet wird. Häufig kann man nur aus den tatsächlichen Tabellen und ihren Inhalten erschließen, warum etwas nicht oder nicht so wie erwartet funktioniert.

Direkte Sicht auf die Datenbank

Es gibt viele alleinstehende Werkzeuge zum Arbeiten mit Datenbanken. Die H2 selbst wird mit einer H2-Console ausgeliefert. Komfortabler und schneller ist der Zugriff auf die Datenbank, wenn dieses Werkzeug in die Entwicklungsumgebung integriert ist.

Für die Entwicklungsumgebung IntelliJ IDEA ist das SQLQuery-Plugin ein solches integriertes Werkzeug, das sich mit der Datenbank verbindet und das Ausführen von SQL-Befehlen auf dieser Datenbank erlaubt. SQLQuery kann über den in IDEA integrierten Plugin-Manager installiert werden.

SQLQuery für IntelliJ IDEA

Nach der Installation muss dem SQLQuery-Plugin der H2-Treiber bekannt gemacht werden. Dazu öffnet man das Tool-Fenster »SQL« und klickt auf das Werkzeug-Icon. In dem erscheinenden Fenster muss nun unter dem Reiter »Miscellaneous« über den Knopf »Add Jar/Directory« das h2.jar in die Liste der JDBC-Treiber aufgenommen werden.

Für Eclipse gibt es mehrere dem SQLQuery entsprechende Plugins. Wir empfehlen den SQLExplorer, der den ausgereiftesten Eindruck macht. Dieses Plugin muss von der SQLExplorer-Website (<http://sourceforge.net/projects/eclipsesql>) heruntergeladen und manuell in das Eclipse-Hauptverzeichnis entpackt werden.

SQLExplorer für Eclipse

In der Eclipse Konfiguration (Preferences im Menü Window) steht nun ein neuer Knoten »SQL Explorer« im Baum der Konfigurationsbereiche zur Verfügung. Für uns ist jetzt der Subknoten »JDBC Drivers«

interessant. Hier ist eine Liste von vorkonfigurierten Treibern zu sehen. Da der SQLExplorer nur die Konfiguration der Treiber ohne die dafür benötigten Treiber enthält, ist jede dieser Konfigurationen als fehlerhaft markiert. Sie muss um den jeweiligen Treiber ergänzt werden.

Für die H2 ist hier noch keine Vorkonfigurierung zu sehen. Wir müssen sie also anlegen. Dazu muss auf den Knopf »Add« neben der Treiberliste geklickt werden. In dem nun erscheinenden Fenster (siehe Abb. 3–1) muss zunächst ein Name für die Konfiguration vergeben werden und eine Beispiel-URL für den Zugriff auf die Datenbank angegeben werden. Danach muss unter dem Reiter »Extra Class Path« durch einen Klick auf »Add« das JAR in den Classpath der Konfiguration eingefügt werden. Über einen Klick auf »List Drivers« wird SQLExplorer angewiesen, nach allen JDBC-Treibern in dem JAR zu suchen. Ein Klick auf »OK« schließt die Konfiguration ab.

Abb. 3–1
Konfiguration eines
Treibers für SQLExplorer

The screenshot shows a dialog box titled "Create New Driver" with a close button (X) in the top right corner. Below the title bar, the text "Create New Driver" and "Provide the details for the new driver" is displayed. The dialog is divided into two main sections. The top section, labeled "Driver", contains two text input fields: "Name" with the value "H2 File" and "Example URL" with the value "jdbc:h2:file:<file>". The bottom section is split into two tabs: "Java Class Path" and "Extra Class Path". The "Extra Class Path" tab is selected and contains a list box with one entry, "H:\h2\bin\h2.jar", which is highlighted in yellow. To the right of this list box are five buttons: "List Drivers", "Up", "Down", "Add", and "Delete". At the bottom of the dialog, there is a "Driver Class Name" field containing the text "org.h2.Driver" and a dropdown arrow. A help icon (?) is located in the bottom left corner, and "OK" and "Cancel" buttons are in the bottom right corner.

Soll eine der bereits vorbereiteten Konfigurationen verwendet werden – zum Beispiel »MySQL Driver« –, muss dieser in der Liste markiert und durch einen Klick auf den Knopf »Edit« für die Bearbeitung geöffnet werden. Hier muss dann wie bei der Konfiguration der H2 auf dem Reiter »Extra Class Path« das JAR mit dem entsprechenden Treiber angegeben werden.

3.3 Ein erstes Beispiel mit Hibernate

Nachdem die Entwicklungsumgebung eingerichtet ist, wollen wir unsere ersten Schritte mit Hibernate wagen. Wir beginnen mit einer zentralen Klasse für unseren Terminplaner – dem Termin.

```
package net.sf.hibernate.sample.einfach;

import java.util.Date;
public class Termin {
    private long _id;
    private String _titel;
    private String _beschreibung;
    private String _ort;
    private Date _zeitPunkt;

    public String getTitel () {
        return _titel;
    }

    public void setTitel (String titel) {
        _titel = titel;
    }
}
```

*Termin.java
(gekürzte Version)*

Die Klasse Termin folgt in ihrem Aufbau der JavaBeans-Spezifikation. Zu jedem Attribut² gibt es eine Setter genannte Methode, mit der das Attribut gesetzt werden kann, und eine andere, Getter genannte, mit der es gelesen werden kann. Setter beginnen immer mit dem Präfix »set« und Getter mit dem Präfix »get« oder für boolesche Attribute »is«. Dem Präfix folgt immer der Name des Attributs, womit der Name des Setters bzw. Getters auch vollständig ist.

JavaBeans-Attribute

In der oben gezeigten gekürzten Version der Termin-Klasse sind exemplarisch ein Getter und ein Setter für das Attribut `titel` zu sehen. Der JavaBeans-Attributname – hier »titel« – muss dabei nicht unbedingt mit dem Namen des zugehörigen Attributs in der Klasse übereinstimmen. Er leitet sich alleine aus der Benennung der Zugriffsmethoden ab.

Klassen, deren Exemplare mit Hibernate persistent gemacht werden sollen, müssen den beschriebenen Aspekt der JavaBeans-Spezifikation erfüllen. Die Zugriffsmethoden müssen jedoch nicht `public` sein. Hibernate wird sie auch dann verwenden können, wenn sie `private` sind.

Code-Konventionen

-
- Wir benennen die Attribute einer Klasse nicht nach den von Sun festgelegten Code-Konventionen, sondern lassen Attributnamen immer mit einem Unterstrich anfangen. Das erleichtert die Unterscheidung zwischen Attributen und lokalen Variablen beim Lesen und Bearbeiten des Codes.

Achtung

Auch wenn es Hibernate beim Setzen und Lesen der Attribute nicht stört, wenn die Methoden `private` sind, sollte auf `private` Methoden verzichtet werden. In einigen Situationen erzeugt Hibernate Proxys um die Geschäftsobjekte. Diese sind wichtig für Hibernates Nachlademechanismus. Bei der Erzeugung eines Proxys wird dynamisch eine Subklasse der Klasse des Geschäftsobjektes erzeugt und jede überschreibbare Methode überschrieben. Hibernate kann dann darauf reagieren, wenn zum ersten Mal auf ein Objekt zugegriffen wird, und erst dann das Objekt aus der Datenbank laden. `private` Methoden und Methoden, die `final` sind, können nicht überschrieben werden. Wird eine solche Methode von außerhalb des Geschäftsobjektes aufgerufen, bekommt es der Proxy nicht mit, und Hibernate lädt das Objekt nicht nach.

`private` Methoden werden zum Beispiel häufig verwendet, um einen Kopier-Konstruktor zu schreiben, der eine Kopie des übergebenen Objektes erzeugt. Da Objekte der gleichen Klasse auf `private` Methoden des jeweils anderen zugreifen können, ist also ein Zugriff auf `private` Methoden von außen möglich und sorgt in diesem Fall für schwer auffindbare Fehler.

3.3.1 Konfiguration

Bevor wir mit dem Programmieren des Hibernate-spezifischen Teils unseres Beispiels beginnen, müssen wir einige Konfigurationsdateien erstellen, die zur Laufzeit gelesen werden, und Hibernate zum Beispiel mitteilen, wie eine Klasse auf Datenbanktabellen abzubilden ist.

Die Mapping-Datei

Zu jeder persistierbaren Klasse muss es eine Hibernate-Mapping-Beschreibung geben. Es hat sich eingebürgert, für jede dieser Klassen eine eigene Mapping-Datei mit der Endung »`hbm.xml`« zu erstellen, die im gleichen Package platziert wird wie die Klasse selbst. Für die `Termin`-Klasse sieht eine mögliche Beschreibung wie folgt aus:

Termin.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="net.sf.hibernatesample.einfach">
  <class name="Termin">
    <id name="id">
      <generator class="native"/>
    </id>
```

```
<property name="titel"/>
<property name="beschreibung"/>
<property name="zeitPunkt"/>
<property name="ort"/>
</class>
</hibernate-mapping>
```

Im Vorspann der Datei wird definiert, in welcher DTD das Format der Datei beschrieben ist. Wenn ein XML-Editor verwendet wird, ist das Erstellen der Datei sehr einfach, weil der Editor über die DTD eine Autovervollständigungsfunktionalität anbieten kann.

Im Tag `hibernate-mapping` wird das OR-Mapping für das angegebene Package definiert. Es ist also möglich, in einer Mapping-Datei das Mapping für alle persistierbaren Klassen eines Packages festzulegen. Ebenso ist es möglich, das Mapping eines Packages in beliebig viele Dateien aufzuteilen.

Über das `class`-Tag wird das Mapping für eine Klasse definiert. Die `property`-Tags definieren, welche Attribute der Klasse persistent sein sollen. In der hier verwendeten einfachen Form gibt es für jedes Attribut ein `property`-Tag, das den JavaBeans-Attributnamen in seinem `name`-Attribut angibt. Hibernate ermittelt bei dieser Variante selbst den Typ des Attributs und wählt eigenständig den Namen der Datenbank-Tabellenspalte für das Attribut.

Mapping einer Klasse

Das `id`-Tag ist ein spezielles `property`-Tag. Es definiert, dass das Attribut `id` persistent sein soll und dass es außerdem der Primärschlüssel für die entsprechenden Datensätze in der Datenbank sein soll.

Primärschlüssel

In Hibernate gibt es mehrere verfügbare Strategien für das Erzeugen von IDs, und eigene können ebenfalls implementiert werden. Die Variante `native` ist die einfachste Variante. Hibernate wählt bei dieser Einstellung selbst eine ID-Erzeugungsstrategie, die den Fähigkeiten der Datenbank entspricht.

Konfiguration von Hibernate

Damit haben wir unsere erste persistierbare Klasse geschrieben und Hibernate verraten, wie ihre Exemplare persistiert werden sollen. Hibernate weiß jetzt aber noch nicht, welche Datenbank verwendet werden soll, und ebenso wenig ist Hibernate bekannt, wo die Mapping-Datei liegt. Die Mapping-Datei wird zwar für gewöhnlich in das Package der Klasse platziert, aber das ist kein Muss, und Hibernate macht hier keine Annahmen.

Hier kommt die Hibernate-Konfiguration ins Spiel:

```
<!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="dialect">
      org.hibernate.dialect.H2Dialect
    </property>
    <property name="connection.driver_class">
      org.h2.Driver
    </property>
    <property name="connection.username">
      sa
    </property>
    <property name="connection.password">
    </property>
    <property name="connection.url">
      jdbc:h2:file:db/termine
    </property>
    <mapping resource=
      "net/sf/hibernatesample/einfach/Termin.hbm.xml"
    />
  </session-factory>
</hibernate-configuration>
```

SQL-Dialekt Die hier gezeigte Konfiguration legt fest, dass der H2-SQL-Dialekt verwendet werden soll. Zwar ist SQL standardisiert, aber jeder Datenbankhersteller reichert das von seiner Datenbank verstandene SQL um eigene Erweiterungen an. An Stellen, wo die SQL-Spezifikation nicht eindeutig ist und Freiräume für eigene Interpretationen bietet, kann sich die Reaktion zweier Datenbanken auf denselben SQL-Befehl unterscheiden. Außerdem gibt es für viele Datenbanken bewährte Arten, etwas zu erledigen, die zu einer performanteren Ausführung führen. Hibernate lässt sich durch die Auswahl eines speziellen Dialekts auf eine Datenbank optimieren.

Ort der Mapping-Dateien

Mit dem Tag `mapping` wird angegeben, wo Hibernate eine Mapping-Datei findet. Für jede im System existierende Mapping-Datei muss es ein solches Tag geben. Das Attribut `resource` gibt den Ort der Mapping-Datei über einen zum Classpath relativen Pfad an. Es ist also praktisch auch möglich, die Mapping-Dateien außerhalb der Packages an einem anderen im Classpath eingebundenen Ort zu halten.

Der Rest der Konfiguration beschreibt den zu verwendenden Treiber, den Benutzer und die einzusetzende Datenbank.

Wie »scott« und »tiger« bei Oracle sind »sa« und »« bei der H2 der Default-Benutzer und sein Passwort.

Tip

Der für uns interessantere Teil der Datenbankkonfiguration ist »connection.url«. Die H2 kann auf drei Arten gestartet werden:

- **Server:** Der gängige Modus, der auch von anderen Datenbanken her bekannt ist. Der Datenbankserver wird explizit gestartet und horcht auf einem Port auf Anfragen.
- **Embedded:** Der H2-Treiber startet die Datenbank eigenständig beim ersten SQL-Befehl, der eingeht. Es handelt sich dann um eine In-Process-Datenbank, die in derselben VM läuft wie der Aufrufer und die von außen nicht ansprechbar ist.
- **In-Memory:** Eine andere In-Process-Variante, bei der die Datenbank nur im Hauptspeicher erzeugt und verändert wird. Sobald die VM beendet ist, sind auch alle Daten verloren, die in der Datenbank abgelegt wurden. Dieser Modus ist vor allem für Tests sinnvoll.

Drei Arten, die H2 zu starten

Wir werden die H2 zunächst nur in den beiden In-Process-Varianten verwenden, da das Starten und Stoppen praktisch von selbst geht und somit kein Verwaltungsaufwand anfällt.

Die in der Konfiguration angegebene `connection.url` gibt eine Embedded-Datenbank an. Bei einer Embedded-Datenbank muss nur angegeben werden, in welchem Verzeichnis die Datenbank liegen und welchen Namen sie haben soll. Wir verwenden das Verzeichnis `db`, das sich im Projektverzeichnis befindet, und die Datenbank nennen wir `termine`. Es ist sinnvoll, für die Datenbank ein eigenes Verzeichnis zu wählen, da diese aus mehreren Dateien besteht und somit die Vermischung mit anderen Projektdateien verhindert wird.

Konfiguration einer Embedded-H2

Wir platzieren die Konfigurationsdatei im Hauptverzeichnis des Quellcodes einer Anwendung. Beim Kompilieren der Anwendung wird die Konfigurationsdatei von der Entwicklungsumgebung in das Verzeichnis mit den `class`-Dateien kopiert und ist dann auf der obersten Ebene des `Classpaths` verfügbar, wo sie von Hibernate gesucht und gefunden wird.

Platzierung der Konfigurationsdatei

Log4J-Konfiguration

Hibernate verwendet für Debug- und für alle anderen Meldungen `log4j`. Damit diese Meldungen auf der Konsole zu sehen sind, muss `log4j` entsprechend konfiguriert werden.

log4j sucht beim Starten nach einer Datei `log4j.properties` auf der obersten Ebene des Classpaths. Wir erstellen die folgende Datei also auch im Hauptverzeichnis des Quellcodes.

```
log4j.rootLogger=DEBUG, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=\
%-4r [%t] %-5p %c %x - %m%n
```

Achtung

Wenn die Zeichenketten nach den Gleichheitszeichen in derselben Zeile stehen wie die Gleichheitszeichen, muss das »\«, das sich hinter diesen befindet, entfernt werden.

3.3.2 Das Beispiel – eine Testklasse für die Grundoperationen von Hibernate

Nun ist es Zeit, den ersten Code zu schreiben, der Hibernate aktiv benutzt. Wir werden eine JUnit-Testklasse schreiben, die einige Grundoperationen bei der Arbeit mit Datenbanken testet. Diese Operationen werden häufig unter dem Kürzel »CRUD« für »Create«, »Read«, »Update« und »Delete« zusammengefasst.

setUp – Vorbereitung von Hibernate und Datenbank für den Test

Am Anfang gibt es für jede Anwendung, die Hibernate verwendet, einige Schritte, die durchlaufen werden müssen. Dazu gehört zum Beispiel das Konfigurieren von Hibernate. Diese Schritte fassen wir in der `setUp`-Methode der Testklasse zusammen.

```
private static final String ORT = "Hamburg";
private static final String BESCHREIBUNG = "termin";
private static final String TITEL = "titel";
private static final Date ZEIT_PUNKT =
    new Date(System.currentTimeMillis() + 172800000);

protected void setUp () throws Exception {
    super.setUp();
    Configuration configuration = new
    Configuration().configure();
    SchemaExport export = new SchemaExport(configuration);
    export.create(false, true);
    _sessionFactory = configuration.buildSessionFactory();
    _id = erzeugeTermin(TITEL,BESCHREIBUNG, ORT, ZEIT_PUNKT);
}
```

Zuerst wird eine Hibernate-Konfiguration erzeugt und mit dem Inhalt der Konfigurationsdatei gefüllt. Hibernate sucht die Konfigurationsdatei auf der obersten Ebene des Classpaths.

Es genügt nicht, die Configuration zu erzeugen. Sie muss explizit mit `configure()` gefüllt werden. `configure()` darf nur genau einmal aufgerufen werden. Wird `configure()` mehr als einmal aufgerufen, liest Hibernate die Konfigurationsdatei erneut ein. Dabei werden auch alle Mappings neu eingelesen, was dazu führt, dass Hibernate sich über doppelte Mappings beklagt.

Achtung

Der `SchemaExport` erzeugt die Tabellenstruktur in der Datenbank. Ein praktischer Nebeneffekt für das Testen ist, dass davor die Datenbank gelöscht wird. Man muss hier also nicht selbst dafür sorgen, dass jeder Test eine saubere Datenbank vorfindet. In den `log4j-Debug`-Meldungen gibt Hibernate aus, wie es die Tabelle erzeugt.

*Erzeugung des
Datenbankschemas*

```
create table Termin (
  id bigint generated by default as identity (start with 1),
  titel varchar(255),
  beschreibung varchar(255),
  zeitPunkt timestamp,
  ort varchar(255),
  primary key (id)
)
```

Die `id` wird also von der Datenbank generiert und ist als `primary key` eingetragen, `zeitPunkt` wurde richtig auf `timestamp` und nicht `date` abgebildet, was den Uhrzeitanteil am `Date`-Objekt erhält. Die Strings wurden pauschal auf `varchar(255)` abgebildet. Hier wird man bei einer späteren Optimierung für `titel` und `ort` wahrscheinlich kleinere `varchars` nehmen, während 255 Zeichen für eine Beschreibung zu knapp bemessen sein könnten. Unser erstes Mapping der `Termin`-Klasse lässt vieles offen, und Hibernate füllt die Lücken mit brauchbaren Werten. Das nächste Kapitel wird sich näher damit befassen, wie Mappings genauer spezifiziert werden können.

Zum Abschluss lässt `setUp` die Configuration eine `SessionFactory` erzeugen. Die `SessionFactory` wird als Fabrik für die im Folgenden benötigten Sessions verwendet. Zuletzt wird ein `Termin` erzeugt.

Erzeugung eines persistenten Objektes

Die Erzeugung eines Termins ist in eine eigene Methode ausgelagert, die in den Tests verwendet werden kann, wenn ein neuer Termin benötigt wird.

```
private long erzeugeTermin (String titel, String beschreibung,
                            String ort, Date zeitPunkt) {
    Termin termin = new Termin();
    termin.setTitel(titel);
    termin.setBeschreibung(beschreibung);
    termin.setOrt(ort);
    termin.setZeitPunkt(zeitPunkt);
    Session session = null;
    Transaction transaction = null;
    try {
        session = _sessionFactory.openSession();
        transaction = session.beginTransaction();
        session.save(termin);
        transaction.commit();
    }
    catch (HibernateException e) {
        if (transaction != null) {
            transaction.rollback();
            throw e;
        }
    }
    finally {
        if (session != null) {
            session.close();
        }
    }
    return termin.getId();
}
```

Das Anlegen des Termins selbst ist unspektakulär und funktioniert so, wie man es von allen anderen Java-Objekten her auch kennt.

*Bezug von der Hibernate-
Session und -Transaktion
zu SQL*

Der spannendere Bereich fängt mit dem Öffnen der Hibernate-Session an. Hierzu wird die SessionFactory verwendet, die wir im setUp erzeugt haben. An der Session wird dann eine Hibernate-Transaktion begonnen. Eine Hibernate-Transaktion ist eng mit der dahinter liegenden Datenbanktransaktion gekoppelt und funktioniert prinzipiell genauso. Was bei einer Datenbanktransaktion und der Verwendung von SQL ein INSERT wäre, ist hier das session.save(). Eine Hibernate-Transaktion wird im Erfolgsfall wie eine Datenbanktransaktion mit commit() abgeschlossen beziehungsweise mit rollback() rückgängig gemacht, wenn Probleme aufgetreten sind.

Wir schließen die Session nach der Erzeugung eines jeden Termins. Das ist jedoch keine Notwendigkeit, und häufig werden in realen Anwendungen mehrere Transaktionen mit derselben Session durchgeführt.

Wird eine Session über mehrere Transaktionen verwendet, so kommt der Cache der Session zum Einsatz. Das ist aus Performance-Gründen sinnvoll. Bei unseren Tests ist der Cache jedoch teilweise nachteilig und kann dazu führen, dass Tests in einer Gruppe (Test-Suite) erfolgreich durchlaufen, die einzeln – also mit unterschiedlichen Sessions – nicht durchlaufen. Wir schließen die Session, um sicher zu sein, dass ein erzeugter Termin beim nächsten Laden wirklich aus der Datenbank und nicht aus dem Cache geladen wird.

*Caching in einer
Hibernate-Session*

Laden eines persistenten Objektes

Nun wollen wir überprüfen, ob es funktioniert, einen zuvor in der Datenbank gespeicherten Termin dort wieder herauszuladen.

```
public void testLoad () {
    Session session = null;
    try {
        session = _sessionFactory.openSession();
        Termin termin = (Termin) session.load(Termin.class, _id);

        assertEquals(ZEIT_PUNKT, termin.getZeitPunkt());
        assertEquals(TITEL, termin.getTitel());
        assertEquals(BESCHREIBUNG, termin.getBeschreibung());
        assertEquals(ORT, termin.getOrt());
    }
    finally {
        if (session != null && session.isConnected()) {
            session.close();
        }
    }
}
```

Der Termin wurde bereits im `setUp()`, das vor jeder Testmethode ausgeführt wird, erzeugt. Hier bleibt uns also nur übrig, den zuvor erzeugten Termin zu laden, was über den Aufruf der Methode `load()` bewerkstelligt wird. Diese Methode bekommt die Klasse des Objekts, das geladen werden soll, und seine ID übergeben. Das Autoboxing³-Feature des JDK 5.0 erlaubt uns, die ID hier einfach hinzuschreiben. Die ID wird von der VM automatisch in einen Long-Wrapper verpackt und an die Methode übergeben. Um diesen Code im JDK 1.4 oder einer älteren Version kompilieren zu können, muss man diese Umverpackung explizit vornehmen. Der Test überprüft hierauf, ob der geladene Termin die richtigen Attribute enthält.

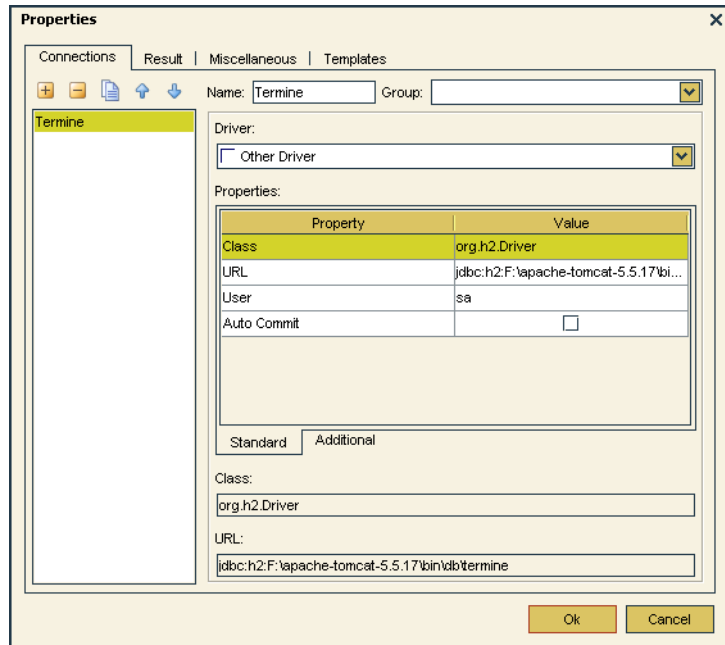
Überprüfung der Datenbank

Nach diesem Test bietet sich eine gute Gelegenheit an, mit einem der beiden Plugins auf die Datenbank zu sehen und zu überprüfen, dass die Datenbank wirklich den Termin enthält.

*Vorgehen beim
SQLQuery-Plugin*

Im SQLQuery-Plugin klickt man auf das Werkzeug-Symbol auf der rechten Seite und erzeugt unter dem Reiter »Connections« eine neue Verbindung. Als Treibertyp nehmen wir »Other Driver«, weil es keine spezielle Konfiguration für die H2 gibt. Für Class geben wir »org.h2.Driver« und für URL die URL unserer Datenbank ein, also zum Beispiel `jdbc:h2:file://g:/projekte/hibernate/db/termine` (Abb. 3–2 zeigt die Konfiguration des Plugins).

Abb. 3–2
Konfiguration von
SQL-Query



- In Java wird zwischen Werten primitiver Typen wie `int` und `float` und Objekten unterschieden. Diese Unterscheidung ist hinderlich, wenn man diese Werte an Stellen verwenden will, wo nur Objekte zulässig sind – zum Beispiel als Elemente einer `ArrayList`. Für diese Situationen enthält Java Klassen, welche die primitiven Typen kapseln (z.B. `Integer` für `int`, `Float` für `float`). Es hat sich jedoch als sehr lästig herausgestellt, die Konvertierung von einem Wert zum entsprechenden Objekt und wieder zurück immer selbst vornehmen zu müssen. Das Autoboxing genannte Feature des JDK 5 macht diese Konvertierung nun selbstständig, sobald ein Wert an einer Stelle verwendet wird, an der ein Objekt erwartet wird. Auch umgekehrt kann ein Objekt von zum Beispiel `Integer` einer `int`-Variablen zugewiesen werden. Autoboxing übernimmt automatisch die Konvertierung.

Im oberen Teil des Plugin-Bereichs können SQL-Befehle eingegeben und über einen Klick auf das grüne Icon ausgeführt werden. Der untere Bereich zeigt das Ergebnis der ausgeführten Befehle.

Abbildung 3–3 zeigt das Plugin nach der Ausführung einer SQL-Anfrage.

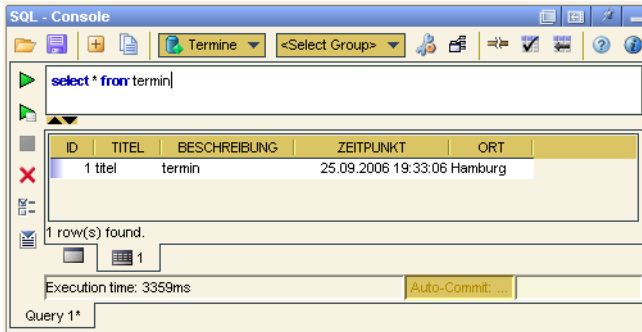


Abb. 3–3

SQL-Query in Aktion

In der SQLExplorer-Perspektive gibt es oben links eine View »Connections«, in der alle dem Plugin bekannten Datenbanken aufgelistet sind. Zurzeit kennt das Plugin keine Datenbanken. Um unsere H2-Termin-datenbank anzumelden, klicken wir mit der rechten Maustaste in dieses View und wählen »New Connection Profile« aus dem erscheinenden Menü aus. In dem nun sichtbaren Fenster wählen wir in der Driver-Combobox unseren zuvor konfigurierten H2-Treiber aus. Im URL-Feld geben wir wie schon für SQLQuery die URL der Datenbank ein (siehe Abb. 3–4). Hiernach muss durch einen Doppelklick auf den Datenbanknamen in der Liste eine Verbindung mit der Datenbank erzeugt werden. Nun können wir auf die Datenbank mit der rechten Maustaste klicken und in dem erscheinenden Kontextmenü »New

Vorgehen beim
SQLExplorer

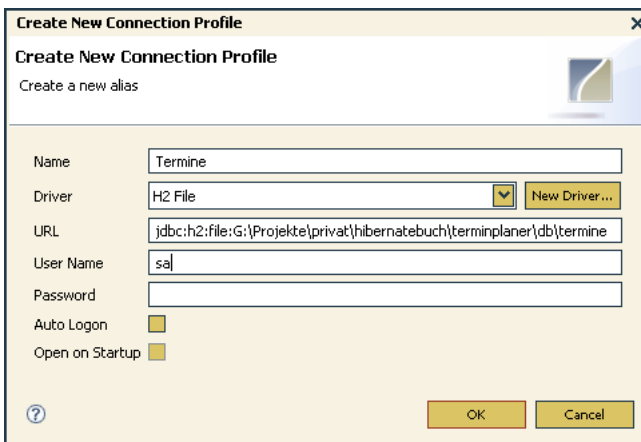


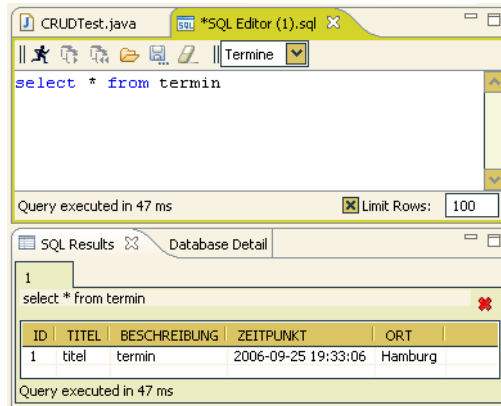
Abb. 3–4

Anmelden einer
Datenbank im
SQLExplorer

SQL Editor« auswählen. Das öffnet einen Editor, in dem SQL-Befehle eingegeben werden können. Ein Klick auf das schwarze Läufer-Icon führt diese Befehle aus (siehe Abb. 3–5).

Abb. 3–5

Ergebnis der Anfrage



Anzeigen aller Termine

Wir sehen uns zunächst an, welche Termine in der Datenbank vorhanden sind.

```
select * from termin
```

Wie erwartet, enthält die Datenbank einen Termin. Abbildung 3–5 zeigt das Ergebnis dieser Anfrage im SQLEditor.

Das Ausführen dieses Befehls führt zur Anzeige aller Daten in der Tabelle der Termine. Diese Tabelle enthält jetzt einen Datensatz.

Tipp

In der ersten Auflage dieses Buches haben wir an dieser Stelle erklärt, dass die HSQLDB mit dem shutdown-Befehl vor dem Starten eines neuen Tests geschlossen werden muss. Das ist mit der H2 nicht mehr notwendig, weil diese sich automatisch beendet, sobald keine aktive Verbindung mehr besteht.

Umschalten der H2 in den In-Memory-Modus

Für die automatisch ablaufenden Tests benötigen wir eigentlich keine Datenbank, die hinterher noch persistent sein muss. Die In-Memory-Variante der H2 eignet sich viel besser für Tests. Sie wird nicht auf die Festplatte gespeichert, und die Datenbankzugriffe werden viel schneller ausgeführt als auf einer persistenten Datenbank. Wir wollen aber nicht immer die Konfigurationsdatei anpassen, wenn wir Tests laufen lassen. Ein Feature, das uns hier gelegen kommt, ist die Möglichkeit, die Einstellungen in der Hibernate-Configuration im Programm zu

ergänzen oder zu überschreiben. In der `setUp()`-Methode können wir vor dem Erzeugen des `SchemaExports` Folgendes einfügen:

```
configuration.setProperty(  
    "hibernate.connection.url",  
    "jdbc:h2:mem:termine;DB_CLOSE_DELAY=-1");
```

Dies ersetzt in der `Configuration` die URL der Datenbank mit einer URL für eine In-Memory-H2, die erzeugt wird, sobald der H2-Treiber den ersten Befehl ausführt. Bei dieser Angabe der Property muss »hibernate.« als Präfix angegeben werden. Bei der Konfigurationsdatei kann dieses Präfix entfallen, weil vom Kontext her klar ist, dass es sich um eine Hibernate-Property handelt.

Der Zusatz `DB_CLOSE_DELAY=-1` ist notwendig, weil die H2 die Datenbank schließt, sobald keine aktive Verbindung mehr besteht. Das hat bei einer In-Memory-Datenbank den Effekt, dass es beim nächsten Starten der Datenbank keine Tabellen mehr gibt. Die Tabellen, die im `setUp` unseres Tests erzeugt werden, würden dann beim Ausführen der Tests nicht mehr zur Verfügung stehen. `-1` gibt hier an, dass die Datenbank bis zum Beenden der Java-VM offen bleiben soll. Wird hier eine positive Zahl angegeben, wartet die H2 noch die entsprechende Anzahl von Millisekunden, bis sie geschlossen wird. Das ist jedoch für Tests mit In-Memory-Datenbanken zu unsicher. Wenn zwischen zwei Datenbankverbindungen in einem Test zu viel Zeit vergeht, führt das zu schwer auffindbaren Fehlern. Diese Option kann zur Performance-Optimierung bei der Verwendung einer persistenten H2 benutzt werden.

Persistente Objekte suchen

Speichern und Laden eines Termins funktioniert also. Eine wichtige Eigenschaft von Datenbanken ist, dass in ihnen gut nach Datensätzen gesucht werden kann. In Hibernate sucht man mit `Query`s nach Objekten.

```
Query query = session.createQuery(  
    "from Termin where ort='" + ORT + "'");  
List result = query.list();  
assertEquals(1, result.size());  
Termin termin = (Termin) result.get(0);
```

Wir sparen uns hier den aus der vorigen Testmethode bekannten Vor- und Nachlauf und beschränken uns auf das Wesentliche. Eine Möglichkeit der Suche (andere Möglichkeiten werden wir in den folgenden Kapiteln zeigen) steht mit der `Hibernate-Query` bereit. Der Ausdruck, der in der `Query` angegeben ist, sieht einem SQL-Select ähnlich. Es fehlt

Hibernate-Query

nur das `select`, und hinter dem `from` steht nicht der Tabellename, sondern der Klassenname. `ort` bezieht sich nicht auf den Spaltennamen in der Datenbank, sondern auf das JavaBeans-Attribut der Klasse.

Die Query wird mit `query.list()` gestartet. Diese Methode gibt das Ergebnis der Query als Liste zurück. (`query.iterate()` ist eine alternative Art, die Query zu starten, und gibt einen Iterator auf das Ergebnis zurück.)

Wir stellen sicher, dass die Liste genau ein Element hat, und ziehen den einzigen in der Liste enthaltenen Termin heraus.

Persistente Objekte aktualisieren

Objekte, die einmal gespeichert wurden, will man auch wieder ändern können.

```
Termin termin = (Termin) session.load(Termin.class, _id);
Transaction transaction = session.beginTransaction();
termin.setTitel("neuer Termin");
transaction.commit();
```

Zuerst laden wir den Termin, den wir ändern wollen, öffnen eine Transaktion und ändern den Termin anschließend wie gewohnt. Hibernate erfordert also keine Änderung des normalen Umgangs mit Objekten. Nach der Änderung schließen wir die Transaktion ab, die wir zuvor aufgemacht haben.

Persistente Objekte löschen

Nun bleibt nur noch das Löschen von Objekten, bevor wir mit dem ersten Durchgang durch Hibernate fertig sind:

```
Termin termin = (Termin) session.load(Termin.class, _id);
Transaction transaction = session.beginTransaction();
session.delete(termin);
transaction.commit();
session.close();
session = _sessionFactory.openSession();
termin = (Termin) session.get(Termin.class, _id);
assertNull(termin);
```

delete Zum Löschen eines Objektes dient die Methode `delete` an der Hibernate-Session. Um diese Methode aufrufen zu können, müssen wir zuerst den Termin laden, der gelöscht werden soll.

Nach dem Löschen versuchen wir, das Objekt mit `session.get()` zu laden. `get()` ist `load()` ähnlich. Der Unterschied zwischen `get()` und `load()` ist, dass `get()` ein `null` zurückgibt, wenn es das angeforderte Objekt nicht gibt, während `load` in diesem Fall eine Exception wirft.

3.4 Zusammenfassung

Wir haben in diesem Kapitel an einem einfachen Beispiel gezeigt, wie man Hibernate verwendet. Dabei ist deutlich geworden, wie einfach und klar der Umgang mit Hibernate ist. Wir können unsere Geschäftsobjekte (im Beispiel Termin) ganz normal programmieren und später die Persistenz durch Definition der Mapping-Dateien hinzufügen. Die Mapping-Dateien lassen sich erst einmal in einer ganz einfachen Form definieren. Hibernate füllt die »Lücken« mit sinnvollen Werten. Danach kann man das Mapping schrittweise optimieren (z.B. die passenden Längen für die VARCHAR-Felder definieren).

Der Umgang mit Sessions und Transaktionen ist in Hibernate ebenfalls ganz einfach. Ein einfacher Aufruf von `save()` speichert das Objekt. Mit `load()` kann es wieder geladen werden, und mit Querys finden wir auch große Mengen von Objekten performant auf.

Damit haben wir in diesem Kapitel einen Hibernate-Querschnitt gezeigt. Die einzelnen Aspekte des Umgangs mit Hibernate werden in den folgenden Kapiteln vertieft. Wir werden die vielfältigen Möglichkeiten zeigen, die für die Definition der Mapping-Dateien existieren. Wir werden die Schnittstelle der Session ausführlich beschreiben und verschiedene Möglichkeiten des Umgangs mit Sessions und Transaktionen diskutieren. Und wir werden die Abfragemöglichkeiten mit Querys und Konsorten studieren.

3.5 Referenzen

[Eclipse] <http://eclipse.org/>

Eclipse ist eine beliebte Entwicklungsumgebung für Java, die unter einer Open-Source-Lizenz steht.

[Hibernate] <http://www.hibernate.org>

Auf der Hibernate-Website sind immer die aktuellen Hibernate-Versionen, aktuelle Neuigkeiten zu Hibernate sowie diverse andere Informationsquellen (Foren, Mailinglisten etc.) zu finden.

[H2] <http://h2database.com>

Diese Website hält die jeweils aktuelle Version der H2 und Informationen zur H2 bereit.

[IntelliJIDEA] <http://www.jetbrains.com/idea/>

IntelliJIDEA ist eine beliebte kommerzielle Java-Entwicklungsumgebung.

[JavaBeans] JavaBeans Specification 1.01 Final Release

Diese Spezifikation definiert unter anderem, welche Bedingungen eine Klasse erfüllen muss, um eine JavaBean zu sein. Die Spezifikation kann von <http://java.sun.com/products/javabeans/docs/spec.html> heruntergeladen werden.

[JUnit] <http://junit.org>

Die JUnit-Website hält die jeweils aktuelle JUnit-Version bereit und enthält Informationen zu anderen JUnit-nahen Technologien.

[Link05] Johannes Link: Software-Tests mit JUnit, dpunkt.verlag, 2005

Dieses Buch behandelt das Thema Testen im Allgemeinen und im Speziellen mit JUnit in ausführlicher Tiefe.

[log4j] <http://logging.apache.org/log4j>

log4j ist das Logging-Framework, das von Hibernate verwendet wird. Die Website enthält neben der aktuellen Version auch die log4j-Dokumentation, der zum Beispiel entnommen werden kann, wie die Ausgabe der Log-Meldungen formatiert werden kann.

[SQLExplorer] <http://eclipsesql.sourceforge.net/>

Das SQLExplorer-Plugin für Eclipse kann zum Zugriff auf Datenbanken aus Eclipse verwendet werden. Es erlaubt die Ausführung von SQL-Kommandos auf der Datenbank und bietet Zugang zu den Metadaten der Datenbank (Aufbau der Tabellen etc.).