# Anleitung BaseLink

Ulrich Hilger

Version 3.0, 26. Mai 2020

# Inhalt

BaseLink vereinfacht die Nutzung von Datenbanken mit Java.

# 1. Überblick

Die Java-Plattform enthält mit der Java Database Connectivity (JDBC) eine umfangreiche Sammlung von Werkzeugen zum Umgang mit Datenbanken. Allerdings erfordert die Programmierung immer wieder benötigter Routineaufgaben wie beispielsweise des CRUD-Musters (Create, Read, Update, Delete) vergleichsweise viel Boilerplate Code.

Hier kommt BaseLink ins Spiel und liefert eine Sammlung von Methoden, die viele wiederkehrende Aufgaben bei der Arbeit mit Datenbanken ausführen so dass Entwickler sie nicht immer wieder neu schreiben müssen. Ferner liefert BaseLink eine einfache Form der objektrelationalen Abbildung um Java-Objekte in Datenbank-Tabellen und umgekehrt Tabelleninhalte in Java-Objekte zu überführen.

# 2. Installation

BaseLink erfordert eine Java-Ablaufumgebung (engl. Java Runtime Environment, JRE). Das JRE ist von AdoptOpenJDK frei erhältlich. Nach Beschaffung des JRE sind zur Installation von BaseLink die folgenden Schritte nötig:

1. Herunterladen des Verteilpakets von BaseLink
2. Entpacken der Datei `baselink.zip`
3. Kopieren der Datei `BaseLink.jar` in den Klassenpfad der Anwendung

Der Klassenpfad bei Java-Webanwendungen ist beispielsweise der Ordner `WEB-INF/lib` des Webarchivs oder im Falle des Servers Tomcat der Ordner `$CATALINA_BASE/lib`.

# 3. Datenbanktreiber

BaseLink setzt auf die Package `javax.sql` auf, in der die Java Database Connectivity (JDBC) gemäß Spezifikation implementiert ist. Voraussetzung zur Nutzung einer Datenbank über JDBC ist die Verfügbarkeit eines JDBC-konformen Datenbanktreibers.

Der Datenbanktreiber liefert eine Implementierung aller Methoden der JDBC speziell für den betreffenden Datenbanktyp. Er wird als eindeutiger Name einer Klasse angegeben wie beispielsweise `org.apache.derby.jdbc.ClientDriver`. Der Hersteller einer Datenbank liefert die Angabe des Treibernamens jeweils in der Dokumentation zur Datenbank. Dort ist auch angegeben, wo der Treiber zu finden ist, bei der Datenbank Derby ist das zum Beispiel die Klassenbibliothek `derbyclient.jar`. Diese muss sich im Klassenpfad von Programmen befinden, die mit Derby-Datenbanken arbeiten möchten.

Ist der Datenbanktreiber wie erwähnt im Zugriff wird er mit BaseLink entweder über die Methoden `setDriverName` oder `setDatabase` des PersistenceManagers angegeben oder indirekt über die Angabe einer DataSource mit der Methode `setDataSourceName`.

*Beispiel für die Angabe einer DataSource*

```
  <Resource name="jdbc/NewUserDB"
      auth="Container"
      type="javax.sql.DataSource"
      username="dbadmin"
      password="changeit"
      driverClassName="org.apache.derby.jdbc.ClientDriver"
      url="jdbc:derby://localhost:1527/udb;create=true"
      maxActive="60"
      maxIdle="30"
      maxWait="5000"
  />
```

Das obige Beispiel einer DataSource enthält die Angabe des Datenbanktreibers im Attribut `driverClassName`.

Zur Laufzeit wird beim Aufruf der Methode `setDataSourceName` dem PersistenceManager der Ausdruck `jdbc/NewUserDB` übergeben. Über das Java Naming and Directory Interface (JNDI) wird daraufhin die Beschreibung der so benannten DataSource lokalisiert, gelesen, in ein Objekt der Klasse `DataSource` überführt und als Quellangabe für Datenbankzugriffe verwendet.

Hierbei wird dann der Datenbanktreiber jeweils der Angabe aus der DataSource entnommen.

# 4. Beispieltabelle

Um die Arbeit mit BaseLink zu veranschaulichen und den Teil auf der Seite der Datenbank zu repräsentieren wird in dieser Dokumentation die folgende Beispieltabelle verwendet:

*Die Beispieltabelle* `USERS`

```
CREATE TABLE APP.USERS
(
   USER_NAME     VARCHAR(250)    NOT NULL,
   USER_FIRST    VARCHAR(250),
   USER_LAST     VARCHAR(250),
   USER_EMAIL    VARCHAR(250),
   PRIMARY KEY (USER_NAME)
);
```

Die Beispieltabelle enthält der Einfachheit halber nur Felder des Datentyps VARCHAR. BaseLink unterstützt aber alle Datentypen, die in einer Datenbank vorkommen können, solange ein JDBC-konformer Datenbanktreiber für die Datenbank existiert.

# 5. Objektrelationale Abbildung

BaseLink verwendet Transferobjekte (engl. Data Transfer Objects, DTO) als Entwurfsmuster zur Abbildung von Objekten in Datenbanken. Orientiert an der Struktur einer Datenbanktabelle

werden DTO mit Annotationen versehen, um die Getter und Setter des Objekts zu kennzeichnen, die zum Lesen und Schreiben einzelener Felder eines Datensatzes dienen (siehe auch DTO erzeugen). Folgende Annotation werden von BaseLink hierzu verwendet:

- `DBTable` - Name der Tabelle, an die ein Objekt gebunden ist
- `DBColumn` - Name des Feldes, aus dem ein Getter des Objekts seinen Inhalt bezieht
- `DBPrimaryKey` - Name des Feldes, über das der primäre Index der Tabelle gebildet wurde. Dies kann auch eine Aufzählung von Feldern sein.

Dabei gelten folgende Vorgaben zwingend:

- Es muss stets der Getter einer Eigenschaft mit der entsprechenden Annotation zum Tabellenfeld versehen werden.
- Die Namen der Methoden von Gettern müssen mit `get`, von Settern mit `set` beginnen
- Die Namen der Objekteigenschaft sowie ihres Getters und Setters müssen gleich lauten.
  - Beispiel: Eigenschaft `id`, Getter `getId`, Setter `setId`
- Datenbanktabellen müssen einen Primärindex haben

Das folgende Beispiel des Objektes `User` zeigt den Einsatz der Annotationen für DTO.

*ein DTO mit Annotationen zur objektrelationalen Abbildung*

```
@DBTable(name="app.users")
@DBPrimaryKey({"user_name"})
public class User {

  private String id;
  private String firstName;
  private String lastName;
  private String email;

  @DBColumn(name = "user_name")
  public String getId() {
    return id;
  }

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

  // ..weiterer Code hier
}
```

**Anmerkung:** Beim relationalen Datenbanksystem Derby werden andere Elemente wie Tabellen oder Indizes einem Schema zugewiesen. Tabellen werden daher mit `[Schemaname].[Tabellenname]` bezeichnet. Wenn nichts anderes angegeben wird, unterscheidet Derby standardmäßig zwischen Systemtabellen (Schema `SYS`) und Tabellen einer Anwendung (Schema `APP`).

## 5.1. Wandler-Objekte

Ist eine Klasse wie im vorigen Abschnitt beschrieben mittels Annotationen an eine Datenbanktabelle gebunden, können Objekte dieser Klasse als DTO auf die Datenbank abgebildet werden. BaseLink kann für solche DTO mit Hilfe der Klasse `GenericRecord` das SQL für die Datenbankoperationen dynamisch erzeugen.

Stattdessen kann für die Klasse eines DTO auch zuvor ein Wandler-Objekt gebildet werden:

```
Record userWandler = new GenericRecord(User.class);
```

Mit dieser Vorgehensweise muss BaseLink nicht immer erst implizit einen Wandler für ein DTO bilden und später wieder freigeben. Ein so erzeugtes Wandler-Objekt ist für alle DTOs einer Klasse wiederverwendbar. Im Falle häufiger Datenbankoperationen kann damit effizienter gearbeitet werden.

# 6. Datenbankoperationen

Dieser Abschnitt beschreibt die von BaseLink direkt unterstützten Datenbankoperationen. Im Bereich Hilfmittel und Tipps werden darüber hinaus zusätzliche Vorgehensweisen im Zusammenhang mit der Nutzung von BaseLink beschrieben.

## 6.1. PersistenceManager

Der `PersistenceManager` ist die zentrale Klasse von BaseLink. Sie stellt Verbindungen zu einer Datenbank her und hält Methoden zur Arbeit mit der Datenbank bereit. Eine Instanz des PersistenceManagers repräsentiert eine Datenbank.

Instanzen des PersistenceManagers verbrauchen ansonsten keine Ressourcen, sie können bedenkenlos nach Bedarf instanziiert werden, die Garbage Collection der Java Virtual Machine gibt sie automatisch wieder restlos frei. Lediglich im Falle der Nutzung von Transaktionen sollten die Verbindungen zur Datenbank stets geschlossen werden bevor ein PersistenceManager der Garbage Collection überlassen wird.

## 6.2. Speichern, Ändern, Löschen

Für das Speichern, Ändern und Löschen von Daten liefert BaseLink die Standardmethoden `insert`, `update` und `delete`. Sie sind jeweils gleich aufgebaut und erfordern im einfachsten Fall als Parameter das Objekt, das gespeichert, geändert oder gelöscht werden soll.

*Beispiel für das Speichern eines neuen Benutzers*

```
User user = new User();
user.setId("testfred");
user.setFirstName("Fred");
user.setLastName("Test");
user.setEMail("fred.test@example.com");
PersistenceManager db = new PersistenceManager();
db.setDataSourceName("jdbc/UserDatabase");
db.insert(user);
```

Wenn viele Datenbankoperationen ausgeführt werden sollen kann es effizienter sein, einen Wandler für eine Klasse bereitzuhalten, dann muss er nicht bei jeder Operation gebildet und wieder freigegeben werden (vgl. Abschnitt Wandler-Objekte). BaseLink hält für diesen Zweck eine Variante bereit, die als zusätzlichen Parameter einen Wandler akzeptiert.

*Beispiel für das Speichern eines neuen Benutzers mit Wandler*

```
private PersistenceManager db;
private Record userWandler;

public void initDb() {
  db = new PersistenceManager();
  db.setDataSourceName("jdbc/UserDatabase");
  userWandler = new GenericRecord(User.class);
}

public User create(User user) {
  return db.insert(user, userWandler);
}
```

Diese Vorgehensweise funktioniert so auch für die Methoden `update` und `delete`. Der Rückgabewert ist stets entweder das übergebene Objekt oder `null`, falls die Operation nicht durchgeführt werden konnte (vgl. Fehlerbehandlung). Für eine Rückgabe erzeugter Schlüssel siehe Abschnitt Schlüsselrückgabe.

## 6.3. Daten lesen

Die Methode `select` des PersistenceManagers dient in verschiedenen Varianten zum Lesen aus der Datenbank. Das geschieht stets mit Hilfe von Ausdrücken der Structured Query Language (SQL). Hierbei unterstützt BaseLink hauptsächlich an zwei Stellen:

1. Datensätze aus der Datenbank werden mit Hilfe der Klasse `GenericRecord` in Objekte überführt (objektrelationale Abbildung)
2. Parameter in Aufrufen der Methode `select` werden dynamisch in veränderliche Teile eines SQL-Ausdrucks übertragen

Für das Gelingen von Punkt 1 oben muss die SQL-Abfrage die Felder ausgeben, die vom Zielobjekt

erwartet werden. Die Parameter an den Aufruf von `select` müssen in der Reihenfolge übergeben werden, in der sie im SQL-Ausdruck enthalten sind, damit Punkt 2 funktioniert.

Die obigen Funktionen werden in folgendem Beispiel veranschaulicht:

*die Methode `getUser` im folgenden Beispielcode zeigt das Lesen aus der Datenbank*

```
private PersistenceManager db;
private Record userWandler;

public void initDb() {
  db = new PersistenceManager();
  db.setDataSourceName("jdbc/UserDatabase");
  userWandler = new GenericRecord(User.class);
}

public List getUser(String name1, String name2) {
  String sql =
    "select user_name,user_first,user_last,user_email" +
    " from app.users" +
    " where user_first = ? " +
    " or user_first = ?";
  return db.select(sql, userWandler, Record.WITHOUT_BLOBS, name1, name2);
}
```

Wird die Methode `getUser` in obigem Beispielcode mit zwei Vornamen gerufen wie z.B. in

```
List nutzerliste = getUser("Maria", "Peter");
```

gibt sie ein Objekt der Klasse `List` zurück. Diese Liste enthält alle Benutzer, deren Vorname Maria oder Peter lautet, als Objekte der Klasse `User`.

Ein Beispiel, das freilich so in der Praxis vermutlich nicht vorkommt. Es zeigt aber, dass die Methode `select` mit einer unterschiedlichen Anzahl von Parametern gerufen werden kann. Die Parameter richten sich nach den veränderlichen Elementen des SQL-Ausdrucks. Im Beispielcode werden die mit Fragezeichen markierten Platzhalter der SQL-Abfrage `where user_first = ? or user_first = ?` mit den Inhalten der Parameter `name1` und `name2` befüllt.

Damit SQL-Ausdrücke in der Praxis nicht wie im Beispiel hart codiert im Quellcode enthalten sein müssen, empfiehlt sich eine Vorgehensweise, wie sie der Abschnitt SQL-Konfiguration beschreibt.

Der Parameter `Record.WITHOUT_BLOBS` regelt, ob eine Abfrage den Inhalt von Binary Large Objects (BLOBs) ausgibt oder nicht. Er kann verwendet werden, um die Ausgabe zu beschleunigen, wenn es absehbar ist, dass die Abfrage keine BLOBs betrifft oder deren Inhalt nicht benötigt wird.

### 6.3.1. Lesen ohne objektrelationale Abbildung

Varianten der Methode `select` des PersistenceManagers, die ohne `GenericRecord` auskommen, wandeln Datenbankinhalte nicht in Objekte sondern liefern eine statische Liste aus Listen mit Strings. Mit diesen Methoden lassen sich dynamisch alle Ergebnisse eines beliebigen SQL-

Ausdrucks in Objekte abbilden, ohne Code zu schreiben, der besonders auf die jeweilige SQL-Abfrage bezogen sein muss.

Jede Liste aus Strings repräsentiert dabei die Feldinhalte eines Datensatzes. Mit der zuvor gezeigten Beispieltabelle würde für die Abfrage

```
List nutzerliste = db.select("select * from app.users", Record.WITHOUT_BLOBS);
```

im Rückgabewert `nutzerliste` ein Objekt der Klasse `List` wie folgt entstehen:

```
List {
  List {
    "user_name",
    "user_first",
    "user_last",
    "user_email"
  },
  List {
    "testfred",
    "Fred",
    "Test",
    "fred.test@example.com"
  },
  List {
    "muellermaria",
    "Maria",
    "Mueller",
    "maria.mueller@example.com"
  },
  List {
    "panpeter",
    "Peter",
    "Pan",
    "peter.pan@example.com"
  }
}
```

Inhalte des Rückgabewerts `nutzerliste` können z.B. mit `List.get(Datensatznr).get(Feldnr)` entnommen werden. Der Ausdruck `System.out.println(nutzerliste.get(2).get(3));` würde also `maria.mueller@example.com` ausgeben.

### 6.3.2. Feldname anstelle von Feldnummer

In einer Abwandlung der Vorgehensweise aus dem vorigen Abschnitt liefert ein Aufruf von

```
List nutzerliste = db.select("select * from app.users");
```

eine Liste aus Objekten der Klasse `Map`, die jeweils einen Datensatz enthalten und Feldinhalte über die Namen der Felder zugänglich machen. Mit dem obigen Beispielaufruf würde etwa `System.out.println(nutzerliste.get(2).get("user_email"));` der Ausgabe aus dem vorigen

Abschnitt entsprechen, aber mit der Angabe des Feldnamens funktionieren.

### 6.3.3. Nutzung von ResultSets

Gelesene Inhalte werden mit BaseLink stets als statische Datenobjekte ausgegeben. Datenbankcursor, mit denen dynamisch durch Teilmengen von Datenbankinhalten navigiert werden kann, wie sie von JDBC-ResultSets geliefert werden, sind nicht Teil des Leistungsumfangs von BaseLink, sie können mit den Klassen der Package `javax.sql` direkt genutzt werden.

### 6.3.4. Zusammenfassung

Wird die Methode `select` des PersistenceManagers von BaseLink mit einem Wandler-Objekt verwendet, bildet BaseLink aus den Datenbankinhalten die gewünschten Objekte, vorausgesetzt, die SQL-Abfrage liefert die Felder, die das Zielobjekt erwartet.

Die Methode `select` kann ohne individuelle Gestaltung des Codes eine unterschiedliche Anzahl Parameter annehmen und auf diese Weise mit derselben Methode beliebige generische SQL-Abfragen dynamisch verarbeiten. SQL-Abfragen mit variablen Anteilen wie z.B. `select * from app.users where user_name = ?` oder `select * from app.users where user_first = ? or user_last like ?` können mit derselben Methode ohne individuelle Anpassungen des Code gleichermaßen verarbeitet werden.

Die Ausgabe von Datenbankinhalten in ihrer generischen Form ohne objektrelationale Abbildung lässt eine nahezu unbegrenzte Anzahl von Abfragemöglichkeiten mit ein und derselben Methode zu. Darüber hinaus können die Methoden `toList` von BaseLink für weitere individuelle Anforderungen verwendet werden.

## 6.4. Transaktionen

Mit Transaktionen lassen sich mehrere Datenbankoperationen zu Funktionsblöcken zusammenfassen die entweder ganz durchgeführt oder ganz rückgängig gemacht werden. Auf diese Weise wird sichergestellt, dass voneinander abhängende Operationen im Falle von Fehlern keine inkonsistenten Zustände in der Datenbank hinterlassen.

BaseLink stellt Transaktionen über die Methoden `getConnection`, `startTransaction`, `commit`, `rollback` sowie `closeConnectionFinally` des PersistenceManagers bereit. Deren Nutzung ist im folgenden Code-Beispiel dargestellt:

*Beispiel für die Verwendung von Transaktionen*

```
public void dbTransaction(User user) {
  PersistenceManager db = new PersistenceManager();
  db.setDataSourceName("jdbc/MyFancyDatabase");
  Connection c = db.getConnection();
  c.startTransaction();
  try {
    db.insert(c, user);
    // ..hier weitere Datenbankoperationen..
    c.commit();
  } catch(Exception ex) {
    c.rollback();
  } finally {
    db.closeConnectionFinally(c);
  }
}
```

Nach dem Aufruf von `startTransaction` können beliebige Datenbankoperationen stattfinden, die einstweilen in der Datenbank noch nicht wirksam werden. Passiert bei einer Operation ein Fehler, werden mit dem Aufruf von `rollback` im `catch`-Zweig alle Operationen seit `startTransaction` rückgängig gemacht. Nur, wenn alle Operationen ohne Fehler durchgeführt werden, sorgt der Aufruf von `commit` zum Schluß dafür, dass alle Operationen in der Datenbank dauerhaft wirksam werden.

Die relevanten Methoden von BaseLink wie z.B. `insert`, `update`, `select` usw. sehen zur Arbeit mit Transaktionen nach obigem Muster jeweils Varianten vor, die ein Connection-Objekt als Parameter akzeptieren.

**Wichtig:** Bei der Arbeit mit Transaktionen muss wie im obigen Beispiel sichergestellt werden, dass eine Datenbankverbindung stets geschlossen wird weil andernfalls Speicherlecks entstehen können (siehe Ressourcenverwaltung).

## 6.5. Schlüsselrückgabe

Viele Datenbanksysteme bieten die Möglichkeit, eindeutige Schlüssel für Tabellen zu verwalten. Wenn in solchen Tabellen ein neuer Datensatz angelegt wird, sorgt das Datenbanksystem dafür, dass ihm ein eindeutiger Schlüssel zugewiesen wird. Das folgende SQL-Beispiel aus der Derby -Dokumentation zeigt diesen Fall

*eine Tabelle mit generiertem Schlüssel*

```
create table greetings
    (i int generated always as identity, ch char(50));
insert into greetings values (DEFAULT, 'hello');
insert into greetings(ch) values ('bonjour');
```

Mit BaseLink lässt sich ein so erzeugter eindeutiger Schlüssel mit der Methode `executeWithKeys` ermitteln. Ein Aufruf wie beispielsweise

```
PersistenceManager db = new PersistenceManager();
db.setDataSourceName("jdbc/MyFancyDatabase");
List keys = db.executeWithKeys("insert into app.greetings(ch) values('Guten Tag')");
```

liefert den erzeugten Schlüssel in `keys` zurück. Hier noch beschreiben, wie `keys` aufgebaut ist und gelesen wird.

Eine Alternative zur Nutzung datenbankspezifischer Mechanismen zur Schlüsselerzeugung zeigt der Abschnitt Eigene Schlüssel.

## 6.6. SQL-Skripte

Mit `PersistenceManager.executeScript` liefert BaseLink eine Methode zur Ausführung von SQL-Skripten. Die Methode lautet wie folgt

*Ausführen eines SQL-Skripts mit BaseLink*

```
PersistenceManager db = new PersistenceManager();
db.setDataSourceName("jdbc/MyFancyDatabase");
int[] ergebnis = db.executeScript(getSqlSkript());
```

Die Methode `executeScript` erwartet als Parameter einen String mit dem Skriptinhalt. Im Beispiel oben wird angenommen, dass eine separate Methode `getSqlSkript` das Skript liefert. Einzelne SQL-Kommandos müssen darin zeilenweise enthalten sein und jeweils mit Semikolon enden.

Der Rückgabewert ist ein Array mit ganzzahligen Werten, die für jede Zeile des Skripts entweder Statement.SUCCESS_NO_INFO oder die Anzahl der von ihr betroffenen Datensätze angeben. Sind alle Rückgabewerte ungleich Statement.EXECUTE_FAILED, war die Ausführung des gesamten Skriptes erfolgreich.

Im Abschnitt Datenbankstruktur erzeugen wird gezeigt, wie mit `executeScript` beispielsweise Skripte zur Anlage einer ganzen Datenbankstruktur ausgeführt werden können.

## 6.7. Fehlerbehandlung

Alle Methoden von BaseLink fangen Fehler ab und vermerken die Fälle im Protokoll. Die Klasse `de.uhilger.baselink.PersistenceManager` legt dafür einen entsprechenden Logger an. Über die Logging-Konfiguration eigener Apps kann gesteuert werden, ob Fehler protokolliert werden und in welchem Umfang.

Es werden also beim Aufruf von BaseLink keine Exceptions geworfen. Ein Fehler lässt sich nur am Rückgabewert einzelner Methoden ablesen.

# 7. Hilfsmittel und Tipps

Neben den von BaseLink unterstützten Datenbankoperationen gibt es eine Reihe von

Anwendungsfällen, die im Zusammenspiel mit dem Einsatz von BaseLink näher betrachtet werden sollten. Diese sind in diesem Abschnitt beschrieben.

## 7.1. DTO erzeugen

Zur Nutzung von BaseLink werden Datenbankstrukturen mit Annotationen an Java-Klassen gebunden (vgl. objektrelationale Abbildung). BaseLink unterstützt diese Aufgabe mit der Methode `Util.generateDTO`. Mit Übergabe eines Tabellennamens wie in folgendem Beispiel erzeugt die Methode den Quellcode einer kompletten Klasse mit Annotationen passend zur Struktur der Datenbanktabelle.

*Erzeugung einer Klasse für ein Data Transfer Object aus der Struktur einer Datenbanktabelle*

```
PersistenceManager db = new PersistenceManager();
db.setDataSourceName("jdbc/MyFancyDatabase");
System.out.println(new Util.generateDTO(db, "APP", "USER"));
```

Die Ausgabe des Quellcodes auf der Konsole kann direkt als neue Klasse gespeichert und kompiliert oder zuvor nach Wunsch von Hand ergänzt und geändert werden. Die Methode `generateDAO` erspart dabei die Tipparbeit bei der Entwicklung erheblich.

## 7.2. SQL-Konfiguration

Bei der Arbeit mit relationalen Datenbanken ist die Structure Query Language (SQL) ein zentrales Hilfsmittel. SQL wird wie eine Programmiersprache verwendet, beinhaltet aber veränderliche Teile der Datenbankstruktur wie z.B. Tabellen- oder Feldnamen. Bei Änderungen der Datenbankstruktur müssen SQL-Ausdrücke angepasst werden. Um Eingriffe in den Quellcode von Anwendungen zu vermeiden sollte SQL stets außerhalb des Quellcodes angelegt sein.

Die Java-Plattform liefert mit der Klasse `java.util.Properties` eine Möglichkeit, veränderliche Programmbestandteile wie z.B. Konfigurationseinstellungen außerhalb des Quellcodes zu halten und eignet sich damit für die Verwendung von SQL.

*Auszug aus der SQL-Konfiguration der App Nutzerverwaltung*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="getUserNameList">
    select user_name from app.users
  </entry>
  <entry key="getUserRoles">
    select role_name from app.user_roles where user_name = ? order by role_name
  </entry>
  <entry key="getUser">
    select * from app.users where user_name = ?
  </entry>
</properties>
```

Werden SQL-Kommandos in eine XML-Struktur notiert wie im obigen Beispiel, können sie zur Laufzeit gelesen werden. Die Klasse `Properties` liefert eine bequeme Methode zum Lesen von Eigenschaften aus einer wie im Beispiel strukturierten XML-Datei.

*Beispielcode zum Lesen von SQL-Kommandos aus einer XML-Datei*

```
public Properties sqlLesen(String dateiname) {
  File sqlFile = new File(dateiname);
  Properties sql = new Properties();
  sql.loadFromXML(new FileInputStream(sqlFile));
  return sql;
}
```

Ein auf diese Weise gelesenes `Properties`-Objekt lässt sich wie folgt in Methoden von BaseLink einbauen.

*Beispielcode zum Lesen von SQL-Kommandos aus einer XML-Datei*

```
private PersistenceManager db;
private Record userWandler;
private Properties sql;

public static final String SQL_GET_USER = "getUser";

public void initDb() {
  db = new PersistenceManager();
  db.setDataSourceName("jdbc/UserDatabase");
  userWandler = new GenericRecord(User.class);
  sql = sqlLesen("/pfad/zu/sql.properties");
}

public User getUser(String userId) {
  String sqlCmd = sql.getProperty(SQL_GET_USER);
  return db.select(sqlCmd, userWandler, Record.WITHOUT_BLOBS, userId);
}
```

In der Methode `getUser` des obigen Beispiels wird ein SQL-Kommando aus einem zuvor in der Methode `initDb` gelesenen `Properties`-Objekt entnommen und der Methode `select` von BaseLink übergeben. Ändert sich später einmal ein Teil der Datenbankstruktur, kann ohne Eingriff in den Quellcode das SQL in der Konfigurationsdatei angepasst werden.

## 7.3. Prüfung Datenbankstruktur

Datenbanksysteme unterscheiden sich in der Art, ob und wie mit ihnen der Zustand der Struktur einer Datenbank geprüft werden kann. Bei Derby wird beispielsweise mit einem URL wie `jdbc:derby://localhost:1527/udb;create=true` eine Datenbank angelegt, falls sie noch nicht existiert. Sie besitzt dann aber noch keine Datenbankstruktur im Innern.

In Kombination mit einer SQL-Konfiguration aus dem vorigen Abschnitt kann mit BaseLink in nur

wenigen Zeilen Code festgestellt werden, ob eine Datenbank bereits die gewünschte Datenstruktur beinhaltet. In besagter SQL-Konfiguration wird dafür zunächst die folgende SQL-Abfrage hinterlegt.

*eine SQL-Abfrage zur Prüfung auf das Vorhandensein einer Tabelle in einer Derby-Datenbank*

```
select s.schemaname, t.tablename
from SYS.SYSSCHEMAS as s,
SYS.SYSTABLES as t
where s.schemaid = t.schemaid
and s.schemaname = ?
and t.tablename = ?
```

Damit kann für eine Tabelle der eigenen App geprüft werden, ob sie bereits angelegt wurde wie im folgenden Code gezeigt.

*eine Methode zur Prüfung, ob eine Tabelle in einer Datenbank vorhanden ist*

```
public boolean dbVorhanden(PersistenceManager db, String sql) {
  boolean istVorhanden = false;
  List<List<String>> list = db.select(sql, Record.WITHOUT_BLOBS, "APP", "USERS");
  if(list != null && list.size() > 1) {
    istVorhanden = true;
    logger.fine("Datenbank ist vorhanden");
  }
  return istVorhanden;
}
```

Nötigenfalls können die Angaben `APP` und `USERS` noch über eine Environment-Variable parametriert werden, um die Methode veränderlich zu halten.

## 7.4. Datenbankstruktur anlegen

Setzt eine App auf eine bestimmte Datenbankstruktur auf, ist es im Zuge der Installation dieser App mitunter nötig, neben den sonstigen Bestandteilen der App auch die entsprechende Datenbankstruktur initial anzulegen. Mit den Hilfsmitteln der SQL-Konfiguration und der Prüfung der Datenstruktur aus den vorangegangenen Abschnitten kann dies mit wenigen zusätzlichen Handgriffen erledigt werden.

Die Vorgehensweise dazu sieht die Hinterlegung eines Skripts zur Erzeugung der Datenbankstrukur vor. Viele Datenbanksysteme beinhalten Werkzeuge, mit denen die komplette Struktur einer Datenbank als SQL-Skript erzeugt werden kann. Bei Derby ist dies z.B. `dblook`. Oft entsteht aber auch das entsprechende Skript nach und nach aus den SQL-Kommandos, die ein Entwickler zur Erstellung einzelner Tabellen nach und nach herstellt.

Davon ausgehend, dass ein Skript zur Erstellung einer Datenbank in der Datei `create_database.sql` vorliegt, ist mit dem folgenden Beispielcode die Datenbankstruktur im Handumdrehen angelegt.

*Beispielcode zur Herstellung der Datenbankstruktur aus einem SQL-Skript*

```
public void initDb() {
  PersistenceManager db = new PersistenceManager();
  db.setDataSourceName("jdbc/UserDatabase");
  if(!dbVorhanden(db, sql.getProperty(SQL_DB_VORHANDEN))) {
    int[] ergebnis = db.executeScript(getSqlSkript());
  }
}

private String getSqlSkript() throws Exception {
  File skript = new File("/pfad/zu/create_database.sql");
  return fromStream(new FileInputStream(skript));
}

private String fromStream(InputStream in) throws IOException
{
  BufferedReader reader = new BufferedReader(new InputStreamReader(in));
  StringBuilder out = new StringBuilder();
  String line;
  while ((line = reader.readLine()) != null) {
    out.append(line);
  }
  return out.toString();
}
```

Im obigen Beispiel wird zunächt geprüft, ob die Datenbankstruktur bereits besteht, indem mit dem Aufruf von `dbVorhanden` nachgesehen wird, ob die Tabelle `app.user` vorhanden ist. Wenn die Tabelle fehlt, wird das Skript `create_database.sql` aus einer Datei gelesen und ausgeführt.

## 7.5. Methoden zum Schließen

BaseLink implementiert für eigene Zwecke Methoden zum Schließen von Verbindungen, ResultSets und Statements. In Fällen einer individuellen Nutzung der Package `javax.sql` können diese Methoden verwendet und so Aufwand für die wiederholte eigene Implementierung gespart werden. Die Methoden sind

- `PersistenceManager.closeResultSetFinally`
- `PersistenceManager.closeStatementFinally`
- `PersistenceManager.closeConnectionFinally`

## 7.6. toList

Eine weitere Methode zur direkten Verwendung von BaseLink-Elementen ist die Methode `toList`. Sie verwandelt ein beliebiges ResultSet in eine statische Liste wie es für das Lesen von Daten ohne objektrelationale Abbildung von BaseLink implementiert ist.

Wann immer also eine individuelle Nutzung der Package `javax.sql` mit der Rückgabe eines

ResultSets endet, kann die Methode `toList` von BaseLink angewendet werden. Hierbei bietet sich auch die Kombination der Verwendung von `toList` und den Methoden zum Schließen an.

## 7.7. Eigene Schlüssel

Im Abschnitt Schlüsselrückgabe ist beschrieben, wie Datenbankfunktionen zur Erzeugung eindeutiger Schlüssel mit BaseLink eingebunden werden können. Ein Nachteil solcher Funktionen ist deren in der Regel nicht standardkonforme SQL-Syntax. Wenn das Datenbanksystem gewechselt werden soll, muss eine App angepasst werden, wenn sie eine solche Funktion nutzt.

Eine Alternative ist es, einen datenbankunabhängigen Mechanismus zu verwenden wie er beispielsweise im Projekt radiozentrale zur Anwendung kommt. Die nachfolgenden Codebeispiele sind diesem Projekt entlehnt.

Sie könnten so auch Teil von BaseLink sein, Datenbanken müssten dann aber stets mit den angegebenen Tabellen ausgestattet sein. Aus diesem Grund soll es an dieser Stelle genügen, den Mechanismus als Kopiervorlage für andere Projekte vorzustellen.

### 7.7.1. Tabelle zur Verwaltung eindeutiger Schlüssel

Zur Verwaltung von Schlüsseln in einer Datenbank wird die folgende Tabelle erzeugt:

*die Tabelle `keytable` zur Vergabe eindeutiger Schlüssel*

```
create table app.keytable (
  key_name          varchar(80) not null,
  key_next          int,
  primary key (key_name)
);
create index key_name_next ON app.keytable (key_name,key_next);
```

Jeder Schlüssel wird in der Tabelle `keytable` mit einem eindeutigen Namen versehen.

### 7.7.2. SQL zur Schlüsselverwaltung

Mit der folgenden SQL-Abfrage kann geprüft werden, wie der nächste Schlüssel lautet.

*SQL-Abfrage zum Abruf des nächsten Schlüssels*

```
select key_next
from app.keytable
where key_name = ?
```

Ist auf diese Weise der nächste Schlüssel ermittelt, kann das folgende SQL-Kommando verwendet werden, um die Schlüsseltabelle mit den nächsten Schlussel zu aktualisieren (inkrement).

*SQL-Kommando für das Schlüsselinkrement*

```
update app.keytable
set key_next = ?
where key_name = ?
and key_next = ?
```

### 7.7.3. Anwendungsbeispiel mit eindeutigem Schlüssel

Wir stellen uns nun vor, dass in unserer App eine Tabelle erstellt wird, die einen eindeutigen Schlüssel enthält wie etwa die Tabelle `sender` für die Speicherung von Radiosendern aus der App radiozentrale.

*die Tabelle `sender` mit dem eindeutigen Schlüssel im Feld `sender_id`*

```
create table app.sender
(
   sender_id     int not null,
   sender_name   varchar(1024) not null,
   sender_url    varchar(1024),
   sender_logo   varchar(1024),
   primary key (sender_id)
);
create index sname on app.sender (sender_name);
insert into app.sender (sender_id, sender_name, sender_url, sender_logo) values (1,
'hr info', 'http://hr-hrinfo-live.cast.addradio.de/hr/hrinfo/live/mp3/128/stream.mp3',
'../bilder/hr-info.png');
```

Für die Verwaltung der `sender_id` wird in der Tabelle `keytable` ein Eintrag wie folgt angelegt:

*Eintrag in `keytable` zur Verwaltung der `sender_id`*

```
insert into app.keytable (key_name, key_next) values ('sender_id',2);
```

Nun kann mit folgender Methode ein neuer Sender angelegt werden.

*eine Methode zur Anlage eines neuen Senders*

```
private PersistenceManager db;
private Record senderWandler;

public void initDb() {
  db = new PersistenceManager();
  db.setDataSourceName("jdbc/RadioDatabase");
  senderWandler = new GenericRecord(Sender.class);
}

public Sender neuerSender(Sender sender) {
  Sender neuerSender = null;
  int nextKey = getNextId(db, "sender_id");
  if(nextKey > -1) {
    sender.setId(nextKey);
    Object o = db.insert(sender, senderWandler);
    if(o instanceof Sender) {
      neuerSender = (Sender) o;
      logger.fine("Sender erstellt: " + sender.getId() + " " + sender.getName());
    } else {
      logger.info("Sender konnte nicht erstellt werden: " + sender.getName());
    }
  } else {
    logger.info("Sender konnte nicht erstellt werden, nextKey ist -1");
  }
  return neuerSender;
}
```

In der obigen Methode wird mit `getNextId` eine neue Sender-ID ermittelt. Die neue Sender-ID wird im Objekt mit dem neuen Sender eingetragen. Dann wird das neue Sender-Objekt mitsamt der neuen ID gespeichert.

War diese Operation erfolgreich, wird das Sender-Objekt *mit* der neu erzeugten ID als Rückgabewert der aufrufenden Methode zurückgemeldet. Im Falle eines Fehlers bleibt das Rückgabeobjekt `null`.

### 7.7.4. Methode `getNextId`

Mit der Methode `getNextId` wird der Wert für einen Schlüssel ermittelt, der als nächstes für einen Tabelleneintrag mit eindeutigem Schlüsssel vergeben werden muss.

*eine Methode zur Ermittlung des nächsten eindeutigen Schlüssels*

```
public int getNextId(PersistenceManager db, String key) {
  int nextKey = -1;
  List list = db.select(getSql(SQL_GET_NEXT_KEY), Record.WITHOUT_BLOBS, key);
  if(list != null && list.size() > 1) {
    nextKey = Integer.parseInt(list.get(1).get(0)); // erster Datensatz ist
Ueberschrift
    if(nextKey > -1) {
      int numRows = db.execute(getSql(SQL_INCREMENT_KEY), nextKey+1, key, nextKey);
      if(numRows < 1) {
        nextKey = -1;
      }
    }
  }
  return nextKey;
}
```

Die Methode `getNextKey` verwendet den Mechanismus zur SQL-Konfiguration um mit `getSql` die SQL-Kommandos zur Ermittlung des nächsten Schlüssels sowie zu dessen Inkrement abzurufen. Mit diesen SQL-Kommandos wird der nächste Schlüssel ermittelt und anschließend hochgezählt.

### 7.7.5. Schlussbemerkungen

Anzumerken ist noch, dass die Methode `getNextId` fehlschlagen kann, wenn zwischen der Ermittlung des nächsten Schlüssels und dem Inkrement des Schlüssels ein konkurrierender Zugriff dem Inkrement zuvor kommt. In diesem Fall würde das verwendete SQL-Kommando keinen Eintrag zum Hochzählen finden, weil der Wert des nächsten Schlüssels dann schon vom konkurrierenden Prozess hochgezählt wurde. Das muss so sein, andernfalls könnte es zu Doppelungen des eindeutigen Schlüssels kommen. Die Methode zur Anlage eines neuen Senders berücksichtigt dies, indem der neue Sender nur angelegt wird, wenn der neue Schlüssel größer als -1 ist.

Ebenfalls nicht unerwähnt bleiben sollte der Fall, dass das Anlegen des Senders fehlschlagen kann. Im Beispiel würde in diesem Fall zwar nichts Schlimmes passieren, es könnte aber eine Lücke in der Reihenfolge der vergebenen IDs entstehen. Es gäbe dann IDs, die keinem Tabelleneintrag zugewiesen sind. Dies lässt sich vermeiden, wenn der gesamte Ablauf in einer Transaktion ausgeführt wird, wurde aber der Einfachheit halber in diesem Beispiel weggelassen.

Alle Elemente der obigen Darstellung können als generische Vorlage für eigene Zwecke dienen. Nur die Sendertabelle und das dafür benötigte Transferobjekt `Sender` müssen an eigene Belange angepasst werden.

Die Verwendung eines solchen Mechanismus lässt die betreffende App unabhängig von proprietären Mechanismen eines bestimmten Datenbanksystems bleiben. Allerdings muss die App damit auch selbst sicherstellen, dass zur Vergabe von eindeutigen Schlüsseln nur der eigene Mechanismus verwendet wird, da andernfalls Schlüssel in den Tabellen gespeichert werden könnten, die nicht der gewünschen Form entsprechen.

# 8. Verbindung zur Datenbank

Um Inhalte in einer Datenbank zu verwenden muss eine Verbindung zur Datenbank aufgebaut werden. Das Öffnen und Schließen von Datenbankverbindungen geht mit Latenzen einher und offene Verbindungen belegen Systemressourcen.

Viele Web- und Applikationsserver wie zum Beispiel Tomcat halten einen Connection Pool bereit, der selbsttätig dafür sorgt, Datenbankverbindungen zu öffnen und zu schließen. Der Connection Pool verwendet Verbindungen mehrfach und sorgt für einen Ausgleich zwischen Ressourcenverbrauch einerseits sowie Latenzen beim Verbindungsauf- und -abbau andererseits.

Der bei Tomcat eingesetzte Connection Pool DBCP ist auch separat erhältlich. Zudem gibt es eine Reihe von weiteren Klassenbibliotheken, die einen Connection Pool implementieren, beispielsweise C3P0, UCP, BoneCP, H2 und so weiter.

BaseLink stellt Verbindungen zur Datenbank nach Bedarf her und schließt sie selbsttätig (Ausnahme: Transaktionen, siehe dort). Abhängig davon, ob die betreffende Datenbank über eine DataSource eingebunden ist oder direkt über einen JDBC-URL wird die Verbindung dabei tatsächlich geöffnet und geschlossen (URL) oder das Öffnen und Schließen vom Connection Pool verwaltet (DataSource), wenn ein solcher in Benutzung ist.

# 9. Ressourcenverwaltung

Die Java Virtual Machine (JVM) beinhaltet eine automatische Speicherverwaltung, der Mechanismus der Garbage Collection (GC) sorgt automatisch dafür, dass nicht mehr benötigte Ressourcen wieder freigegeben werden. In der Regel funktioniert das auch, wenn Referenzen auf Objekte nicht explizit von der App aufgelöst werden, die sie instanziiert hat.

Im Falle der JDBC sind Objekte der Klasse Connection potentielle Kandidaten für Ressourcenlecks, wenn die Garbage Collection nicht erkennen kann, dass eine Datenbankverbindung nicht mehr benötigt wird.

BaseLink sorgt dafür, Datenbankverbindungen selbst nach Bedarf zu öffnen und zu schließen womit eine ordentliche Ressourcenverwaltung sichergestellt ist. Einzige Ausnahme sind Fälle, in denen zum Zwecke der Verwendung von Transaktionen eine Datenbankverbindung an die App herausgereicht wird. Hier sollte stets die App darauf achten, eine Verbindung zu schließen, wenn sie nicht mehr benötigt wird.

# 10. Lizenz

BaseLink ist freie Software und wird unter den Bedingungen der GNU General Public License zur Verfügung gestellt.