Bundesverwaltungsamt

IsyFact-Standard

Detailkonzept Komponente Datenzugriff

Version 2.15
14.07.2016

IsyFact Standards

„Detailkonzept Komponente Datenzugriff“
des Bundesverwaltungsamts ist lizenziert unter einer
Creative Commons Namensnennung 4.0 International Lizenz.

Ansprechpartner:

Referat Z II 2
Bundesverwaltungsamt
E-Mail: isyfact@bva.bund.de
Internet: www.isyfact.de

Dokumentinformationen

Dokumenten-ID:	Detailkonzept_Komponente_Datenzugriff.docx

Java Bibliothek / IT-System

Name	Art	Version
plis-persistence	Bibliothek	1.4.x

Inhaltsverzeichnis

1. Einleitung

In den Anwendungen der IsyFact werden dauerhaft zu speichernde Daten über eine Zugriffsschicht in einer relationalen Datenbank abgelegt. Die Zugriffsschicht wird für alle Datenbankzugriffe verwendet.

Die Implementierung der Zugriffsschicht wird durch die Verwendung eines standardisierten Frameworks erleichtert. Als Framework wird hier JPA mit einer Implementierung durch Hibernate verwendet.

In diesem Dokument werden allgemeine Vorgaben für das Persistenzmodell (die dauerhaft zu speichernden Objekte), für die Transaktionssteuerung der Datenbank-Zugriffe sowie für die Verwendung von JPA und Hibernate festgelegt. Weiterhin wird das Thema Historisierung von Daten behandelt.

Die Vorgaben zum Thema JPA/Hibernate betreffen die folgenden Themenbereiche:

- Die Definition des Mappings zwischen Persistenzobjekten und der Datenbank.
- Die Konfiguration von JPA.
- Die Konfiguration von Hibernate.
- Die Verwendung von JPA in der Anwendung, vor allem das Zusammenspiel mit dem Spring Framework.

2. Überblick

Gemäß den Vorgaben der Referenzarchitektur der IsyFact (IsyFact – Referenzarchitektur) basiert die Architektur eines IT-Systems auf der bekannten Drei-Schichten-Architektur. Eine dieser Schichten ist die Persistenzschicht. In dieser Schicht ist alle Funktionalität enthalten, die zum Erzeugen, Suchen, Bearbeiten und Löschen von Datensätzen benötigt wird. Der Zugriff auf eine relationale Datenbank ist in dieser Schicht vollständig gekapselt. In Richtung Datenbank kommuniziert diese Schicht mittels JDBC und SQL, in Richtung des Anwendungskerns stellt die Schicht Geschäftsobjekte zur Verfügung.

In Abbildung 1 ist die Referenzarchitektur eines IT-Systems noch einmal dargestellt, wobei die Persistenzschicht mit der Komponente Datenhaltung hervorgehoben ist.

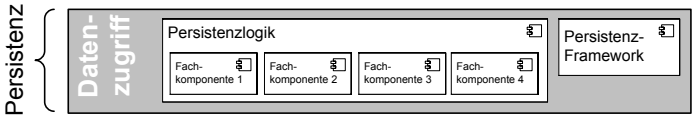

Abbildung 1: Referenzarchitektur eines IT-Systems

In diesem Dokument werden die Vorgaben zum Bau der Komponente Datenzugriff im Detail beschrieben. Die Implementierung der Zugriffsschicht wird durch die Verwendung des standardisierten Frameworks JPA erleichtert. JPA wird hier mit einer Implementierung durch Hibernate verwendet.

Dieses Dokument ist aber keine allgemeine Anleitung zur Verwendung von JPA. Für den Einstieg in JPA mit Hibernate ist das Buch „Java Persistence with Hibernate“ (Bauer & King, 2007) sehr zu empfehlen.

Für ein Verständnis der in diesem Dokument geforderten JPA-Nutzung ist vor allem das Codebeispiel Vorlage-Anwendung (Beispielimplementierung „Vorlage-Anwendung“) zu empfehlen:

- seine Konfiguration in den Dateien `jpa.properties`, `jpa.xml`, `persistence.xml` und `hibernate.cfg.xml`,
- seine Persistenzklassen `Cd` und `Interpret`,
- seine Data Access Objects `CdDatenDao` und `BestandDao`,
- seine JPQL Queries in der Datei `NamedQueries.hbm.xml`

entsprechen den in diesem Dokument gemachten Vorgaben und können bequem als Vorlage für eigene Implementierungen verwendet werden. Die in diesem Dokument gezeigten Code-Beispiele stammen in der Regel auch aus der Vorlage-Anwendung.

JPA kann (gerade in Verbindung mit dem Spring Framework) auf verschiedene Arten verwendet werden. Die in diesem Dokument aufgestellten Vorgaben für die Verwendung von JPA verfolgen vor allem zwei Ziele:

- Die Verwendung von JPA projektübergreifend zu vereinheitlichen.
- Die Verwendung von JPA so einfach, komfortabel und verständlich wie möglich zu gestalten.

Auf Grund dieser Anforderungen wird versucht, möglichst ausschließlich die Mechanismen von JPA zu verwenden. Nur wo dies nicht möglich ist, darf auf die Hibernate-Implementierung zugegriffen werden. Dabei wird in diesem Dokument die Verwendung bestimmter Hibernate-Features verboten bzw. wird von ihrer Verwendung abgeraten. Diese Ausnahmen sind (falls nur abgeraten wurde) mit dem Technischen Chefdesigner des Teilprojekts bzw. (falls sie verboten wurden) mit dem Chefarchitekten des Gesamtprojekts abzustimmen und vom Architekturboard genehmigen zu lassen. Einzig Ausnahmen, die in diesem Dokument beschrieben werden, sind von diesem Verfahren ausgenommen.

Dieses Dokument ist wie folgt aufgebaut:

- Zunächst werden in Kapitel 3 die Anforderungen an die Komponente Datenzugriff kurz motiviert.
- Anschließend werden in Kapitel 4 grundsätzliche Regeln vorgestellt, wie die Abbildung von Objekten auf Datenbanken stattfinden soll.
- Kapitel 5 vertieft anschließend das Thema Mapping, wobei konkrete Hinweise zur Abbildung von Attributen auf Datenbankfelder gegeben werden.
- Im Anschluss daran wird die konkrete Nutzung des Datenbankzugriffs in der Anwendung beschrieben. Zunächst wird gezeigt, wie die JPA aus der Anwendung heraus genutzt werden soll (Kapitel 5.9). Anschließend wird erläutert, wie Datenbank-Abfragen durchgeführt werden (Kapitel 6.2).
- Die generelle Konfiguration von JPA und Hibernate ist in Kapitel 7 beschreiben.
- Abschließend werden noch die weiterführenden Themen Transaktionssteuerung (Kapitel 8) und Historisierung (Kapitel 9) behandelt.

3. Anforderungen

Die in diesem Dokument aufgestellten Vorgaben setzen folgende Anforderungen um:

Einfachheit der Verwendung von JPA: Dies betrifft sowohl die Erstellung von JPA-Konfigurationen und des JPA verwendenden Codes als auch deren Verständlichkeit und Wartbarkeit. Konkret kann diese Anforderung in die folgenden Anforderungen aufgeteilt werden:

- Einfachheit und Verständlichkeit der Konfiguration und der Mapping-Definition von JPA.
- Einfachheit und Verständlichkeit des JPA verwendenden Codes.

Schmale, definierte JPA-Schnittstelle: Die Verwendung und Konfiguration von JPA soll isoliert und über eine schmale Schnittstelle erfolgen.

- Die Komponente Datenzugriff soll auf JPA über die schmale Schnittstelle zugreifen.
- Die Stellen, an welchen JPA verwendet wird, sollen keine Fachlogik enthalten.

Effizienz und Schnelligkeit der JPA Zugriffsschicht und der dahinter liegenden Hibernate-Implementierung: Die von JPA abgesetzten Datenbank-Aufrufe sollen effizient sein. Unnötige Zugriffe und doppelte Zugriffe sollen vermieden werden.

Einheitlichkeit der Verwendung von JPA: Die Konfiguration und Programmierung der JPA-Zugriffe soll zum einen über die gleichen Mechanismen und zum anderen an den gleichen Stellen vorgenommen werden.

4. Persistenz

Ein Persistenz-Klassenmodell ist das Modell der Entitäten, welche dauerhaft abgespeichert werden sollen. Bevor auf den Einsatz von JPA eingegangen wird, werden in diesem Kapitel gewünschte Eigenschaften des Persistenz-Klassenmodells beschrieben. Für das Modell werden Verwendungsregeln aufgestellt.

4.1. Persistenz-Klassenmodell und Datenbank-Schema sollen möglichst ähnlich sein

Im Idealfall wird jedes Persistenzobjekt auf eine Tabelle des Datenbankschemas abgebildet. Eine solche Abbildung ist intuitiv und erleichtert das Verständnis der Anwendung und des Datenbankschemas, was wiederum in der Wartung ein großer Vorteil ist.

Tatsächlich ist es aus Gründen der Datenbankperformance aber oft erforderlich, von diesem Idealfall abzuweichen. Hier gilt es, auf möglichst wenige Tabellen zuzugreifen, um an die benötigten Informationen zu gelangen.

So ist es zum Beispiel sinnvoll, für 1:1-Beziehungen im Persistenz-Klassenmodell den JPA-Mechanismus der Embeddables zu verwenden. Hier wird der Inhalt einer Datenbank Tabelle durch ein entsprechendes JPA-Mapping auf mindestens zwei Persistenzklassen verteilt. Solche Persistenzobjekte können dann über das Lesen einer einzigen Tabellenzeile aus der Datenbank gefüllt werden.

4.2. Verwendung generischer Datenstrukturen vermeiden

JPA ermöglicht die Verwendung generischer Datenstrukturen. Dabei können die Spalten einer Tabelle in Abhängigkeit eines Deskriptorwertes in einer speziell definierten Spalte auf unterschiedliche Attribute verschiedener Persistenzklassen abgebildet werden. Eine solche Vorgehensweise erschwert das Verständnis der Daten bei einem direkten Zugriff auf die Datenbank mit Datenbankwerkzeugen und damit auch die Fehleranalyse und Wartung.

Trotzdem kann es aus Gründen der Datenbankperformance erforderlich sein, generische Datenstrukturen zu verwenden. Werden z. B. Persistenzklassen mit einer gemeinsamen Oberklasse als generische Datenstruktur in einer Tabelle abgelegt und sollen die Attribute der Oberklasse gesucht werden, so kann die Suche auf der Datenbank in einer einzigen Tabelle durchgeführt werden. Die Persistenzobjekte können ohne zusätzliche Joins aus der Datenbank gelesen werden.

Werden generische Datenstrukturen verwendet, dann ist es obligatorisch, dass für jede Persistenzklasse, die in dieser Datenstruktur abgespeichert wird, ein eigener Datenbank-View definiert wird. Dieser Datenbank-View enthält alle Attribute der Persistenzklasse mit einem sprechenden Namen. Diese Datenbank-Views können dann beim direkten Zugriff mit Datenbankwerkzeugen und bei der Fehleranalyse verwendet werden.

4.3. Vererbung im Persistenz-Klassenmodell vermeiden

Klassenhierarchien (der Einsatz von Vererbung) sind bei Persistenzobjekten zu vermeiden. Abweichungen sind nur für eine Reduzierung der Komplexität des Persistenz-Klassenmodells erlaubt.

Falls Vererbung im Persistenz-Klassenmodell eingesetzt wird, ist sie im Datenbank-Schema auf folgende Weise umzusetzen:

- Über eine Tabelle, welche die Daten aller Klassen der Hierarchie („Table per class hierarchy“, siehe (Bauer & King, 2007) Kapitel 5.1.3) enthält. Diese Art der Abbildung ist vorzuziehen. Falls sie auf Grund vieler unterschiedlicher Felder oder Problemen mit Pflichtfeldern in den Klassen nicht verwendbar ist, können auch andere Strategien verwendet werden. Die Vor- und Nachteile der einzelnen Strategien sind in (Bauer & King, 2007) beschrieben.

Die Klassenhierarchie ist in jedem Fall möglichst flach zu halten: Soweit möglich sollen nur zwei Stufen verwendet werden.

4.4. Fachlogik in Persistenzklassen vermeiden

Die Implementierung von fachlicher Logik in den Persistenzklassen ist zu vermeiden. Idealerweise sollten die Persistenzklassen lediglich `get`- und `set`-Methoden für die persistierten Daten enthalten. Jegliche Logik sollte im Anwendungskern implementiert werden.

Zur Fachlogik gehören auch Validierungen.

4.5. Methoden equals und hashCode implementieren

Im Normalfall müssen für Entitätsklassen die Methoden `equals` und `hashCode` nicht überschrieben werden.

Nur für Embeddable-Klassen (siehe (Bauer & King, 2007) Kapitel 4.4.2) müssen die Methoden `equals` und `hashCode` implementiert werden. Die Methoden müssen dafür sämtliche Attribute mit einbeziehen Zusätzlich muss das Interface `Serializable` implementiert werden.

Für Beispiele zu den `equals`- und `hashCode`-Implementierungen siehe die Klasse `Name` der Vorlage-Anwendung (Beispielimplementierung „Vorlage-Anwendung“).

4.6. Initialisieren von String-Feldern

Für die Verarbeitung im Regelwerk (siehe (Konzept Regelwerk)) ist es hilfreich, dass String-Felder initialisiert werden, da ansonsten in nahezu allen Regeln zwischen `""` und `null` differenziert werden müsste. In Objekten, die in das Regelwerk eingegeben werden sollen, wird daher bei der Definition von String-Feldern initial ein Leer-String gesetzt:

```
public class Interpret {
    private String name = "";
...
```

5. Die Definition des Mappings zwischen Objekten und Datenbank

Im vorherigen Abschnitt wurden allgemeine Regeln für das Persistenz-Klassenmodell aufgestellt. In diesem Kapitel wird die Abbildung dieses Modells auf ein Datenbankschema in JPA beschrieben.

5.1. Definition des Mappings über Annotationen

Die Definition des Mappings wird über Annotationen in den Persistenzklassen (Entitätsklassen) durchgeführt. Pro Klasse wird über die Annotationen definiert, auf welche Tabelle sie abgebildet werden und wie ihre Variablen auf Datenbank-Felder abgebildet werden. Für Beispiele zu Annotationen siehe die Klassen `Cd`, `Interpret` und `Bestand` in der Vorlage-Anwendung (Beispielimplementierung „Vorlage-Anwendung“).

Über Annotationen können einige wenige Mappings nicht definiert werden, welche über eine XML-Konfigurationsdatei definierbar sind. Ein Beispiel dafür ist das Mapping einer Klasse auf zwei verschiedene Tabellen.

Falls eine XML-Mapping-Konfiguration für eine Klasse notwendig ist, ist die Konfiguration für diese Klasse in einer XML-Konfigurationsdatei abzulegen. Diese wird automatisch von JPA verwendet.

5.2. 1:n Assoziationen in der Regel als Set (ohne Reihenfolge) definieren

Beim Abbilden einer 1:n Assoziation („Collection Mapping“, siehe (Hiberate Documentation: Chapter 6. Collection Mapping)) ist in der Regel als Java-Typ `Set` zu definieren, da in einem Set keine Reihenfolge definiert ist.

```
@OneToMany(mappedBy = "isbn",
           targetEntity = Cd.class,
           cascade = { CascadeType.ALL })
public Set<Cd> getCds() {...
```

Wird von der Anwendung eine Sortierung benötigt und sind alle für die Sortierung benötigten Attribute in der Entität enthalten, dann kann auch der Java-Typ `List` verwendet werden, da die Datenbank effizienter sortieren kann als eine Java-Implementierung.

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "CD_ISBN")
@OrderBy("id ASC")
public List<Track> getTracks() {...
```

5.3. Identifizierende Attribute verwenden

Falls für eine Entität genau ein identifizierendes Attribut existiert, ist dieses sowohl in der Datenbank als auch im Hibernate Mapping als Primärschlüssel zu verwenden. Künstliche ID-Spalten sind nur dann als Schlüssel zu verwenden, wenn kein identifizierendes Attribut für die Entität vorliegt oder nur mehrere Attribute zusammen die Entität eindeutig

identifizieren. Zusammengesetzte Schlüssel dürfen nicht verwendet werden.

Das identifizierende Attribute darf beliebige Typen besitzen: Es dürfen Zeichenketten oder Datumsangaben sein.

5.4. Bidirektionale Assoziationen vermeiden

Bidirektional traversierbare Assoziationen (`get`-Methoden auf beiden Seiten) sind zu vermeiden. Für die Traversierung in Gegenrichtung sollte eine Query verwendet werden.

Grund für die Vorgabe ist, dass Änderungen am „inversen Ende“ der Assoziation nicht persistiert werden. Falls wirklich eine bidirektionale Assoziation benötigt wird, sind in der Entität am „inversen Ende“ der Assoziation `add`/`remove` Methoden zu definieren, welche die Assoziation korrekt manipulieren.

Explizit verboten sind bidirektional traversierbare n:m Assoziationen. Hierfür sind zwei 1:n (bzw. n:1) Mappings zu definieren.

5.5. Behandlung von Zeitangaben

Für die Speicherung von Zeitangaben wird in Java einheitlich `java.util.Date` verwendet. In der Datenbank erfolgt die Speicherung in einem Attribut vom Typ `TIMESTAMP`. In der Entitätsklasse ist das Mapping wie folgt anzugeben:

```
@Temporal(TemporalType.TIMESTAMP)
public java.util.Date getEingangsdatum() {...
```

Falls die Genauigkeit des Timestamp-Datentyps fachlich nicht gewünscht ist, kann der Technische Chefdesigner entscheiden, dass in der Datenbank der Typ `DATE` verwendet wird. Das Mapping muss dann folgendermaßen festgelegt werden:

```
@Temporal(TemporalType.DATE)
public java.util.Date getGeburtsdatum() {...
```

Hibernate erzeugt beim Laden der Daten aus der Datenbank implizit `java.sql.Timestamp`- bzw. `javal.sql.Date`-Objekte für diese Attribute. Beide Typen sind von `java.util.Date` abgeleitet und dieses Verhalten damit für den Entwickler transparent.

Vergleiche von Zeitangaben unterschiedlicher Genauigkeit sind jedoch problematisch:

- Grundsätzlich darf der Vergleich **nicht mit der `Equals`-Methode** durchgeführt werden, es muss immer `compareTo` verwendet werden.
- Ein Vergleich mit **`CompareTo` muss immer auf dem Attribut mit höherer Genauigkeit** (also auf dem `java.sql.Timestamp`) aufgerufen werden:

```
getEingangsdatum().compareTo(getGeburtsdatum());   //
OK
getGeburtsdatum().compareTo(getEingangsdatum());  //
Nicht OK
getGeburtsdatum().equals(getEingangsdatum());     //
Nicht OK
```

Analoges gilt für die Verwendung im Regelwerk (siehe (Konzept Regelwerk)). Hier wird automatisch `CompareTo` verwendet. Das Attribut mit der höheren Genauigkeit muss aber auf der linken Seite des Vergleichsoperators stehen:

```
rule "Datums-Operation"
  when
    Interpret(eingangsDatum < geburtsdatum)   // OK
    Interpret(eingangsDatum == geburtsdatum) // OK
    Interpret(geburtsdatum > eingangsDatum)  // Nicht
OK
  then
    ...
end
```

Für Berechnungen, z. B. das Hinzuaddieren von Tagen, oder das Setzen von Feldern, ist der Daten-Typ `java.util.Calendar` zu verwenden.[1] In diesem Fall wird im Anwendungskern temporär ein `Calendar`-Objekt für das entsprechende Datum erzeugt:

```
Calendar cal = Calendar.getInstance();
cal.add(Calendar.DAY_OF_MONTH, 1); // Einen Tag
addieren
cal.set(Calendar. MONTH, 11); // Monat auf Dezember
setzen
```

5.6. Boolesche Variablen

Für die Ablage von booleschen Werten in der Datenbank ist stets ein `NUMBER` Feld zu verwenden, kein Textfeld. Der Wert wird über das default Hibernate-Mapping auf 1 für wahr und 0 für falsch abgebildet.

5.7. Enum-Variablen

Für die Ablage von Enum-Feldern persistenter Entitäten in der Datenbank sind in JPA zwei Modi vorgesehen[2], die jedoch beide mit Nachteilen verbunden sind:

- `ORDINAL`: Die Enum-Ausprägungen werden durchnummeriert und als Integer abgelegt. Diese Ablage ist sehr ungünstig, weil sich beim Hinzufügen oder Entfernen einer Enum-Ausprägung, die nicht die letzte ist, die Nummern verschieben und dadurch eine Datenmigration erforderlich wird.
- `STRING`: Es wird der Java-Name der Enum-Ausprägung in der Datenbank abgelegt. Diese Ablage ist problematisch, weil sie eine

[1] Insbesondere dürfen die als Deprecated markierten Methoden von Date nicht verwendet werden.

[2] Siehe `javax.persistence.EnumType`

enge Kopplung des Java-Codes an die Datenbankinhalte erzeugt. Unter Umständen sollen im Java-Code lange, sprechende Namen genutzt werden, während für die Ablage in der Datenbank eine kurze, Speicherplatz sparende Darstellung gewünscht ist.

Aufgrund der genannten Schwächen werden in der Bibliothek plis-persistence zwei Hibernate User-Types zur Verfügung gestellt, um Enum-Werte auf eine VARCHAR-Spalte der Datenbank abzubilden:

- `EnumUserType` erlaubt es, in einem Enum per Annotation die gewünschte Datenbankdarstellung zu jeder Ausprägung anzugeben.
- `EnumWithIdUserType` erlaubt die Persistierung von Enums, die einen fachlichen Schlüssel als Attribut besitzen.

Beispiel für eine Enum-Klasse mit annotierten Persistenzwerten:

```
public enum Geschlecht {
    @PersistentValue("M")
    MAENNLICH,
    @PersistentValue("W")
    WEIBLICH
}
```

Beispiel für eine Enum-Klasse mit natürlichem Schlüssel:

```
public enum Geschlecht {
    MAENNLICH("M"),
    WEIBLICH("W");

    private final String id;

    private Geschlecht(String id) {
        this.id = id;
    }

    @EnumId
    public String getId() {
        return id;
    }
```

Beispiel für eine persistente Entität, die ein Enum-Feld enthält:

```
@Entity
public class Interpret {
    …

    @Column(nullable = false, length = 1)
    @Type(type =
"de.bund.bva.pliscommon.persistence.usertype.Enum(WithI
d)UserType", parameters = { @Parameter(
        name = "enumClass",
        value = "<Package>.Geschlecht") })
    public Geschlecht getGeschlecht() {
        return geschlecht;
    }
    …
}
```

5.8. Datenbankschema anfangs über hbm2ddl erzeugen

Für die Erstellung des Datenbank-Schemas wird empfohlen, es initial über Hibernate zu erzeugen. Dies ist einfach zu konfigurieren: In der Konfiguration der Session-Factory (siehe Datei `hibernate.cfg.xml` in der Vorlage-Anwendung) ist die Konfiguration von `hbm2ddl.auto` auf `create` zu setzen:

```
<property name="hibernate.hbm2ddl.auto">
    create
</property>
```

Grundsätzlich ist es möglich, sämtliche Tabellen-Eigenschaften (etwa auch die Feldlängen und Indizes) über Annotationen zu definieren und das Datenbank-Schema komplett durch hbm2ddl zu erzeugen. Hierzu wird keine Vorgabe erstellt: Ob die DDL während der Entwicklung stets generiert wird oder sie nach einer initialen Generierung verändert und parallel gepflegt wird, ist je nach Komplexität des Schemas zu entscheiden.

Befindet sich die Anwendung aber in Produktion, dann muss der Parameter `hbm2ddl.auto` auskommentiert werden, damit weder eine Generierung noch eine Validierung des Schemas stattfindet. (Anmerkung: Es gibt keinen gültigen Parameterwert „none“ oder ähnliches, der hbm2ddl bei gesetztem Parameter `hbm2ddl.auto` deaktivieren würde. Der Parameter muss daher aus der Konfiguration entfernt werden.) Eine Validierung durch Setzen des Parameters auf `validate` findet nicht statt. Stattdessen wird eine explizite Versionierung des Schemas verwendet: Bei jedem Start der Anwendung wird überprüft, ob in der Datenbank die Schemaversion vorliegt, die die Anwendung erwartet. Die Funktionalität hierzu ist in Abschnitt 10.5 beschrieben.

5.9. Vergabe von Indizes

Indizes sind ein wichtiges Element, um eine gute Performance des Datenbankzugriffs sicherzustellen. Indizes müssen dabei gezielt vergeben werden. Fehlende Indizes führen häufig zu einer schlechten Performance der Anwendung und belasten die Datenbank unter Umständen durch das Auftreten von Full-Table-Scans sehr stark. Zu viele Indizes verschlechtern die Performance beim Schreiben von Datensätzen und verbrauchen unnötigen Speicherplatz.

Die tatsächlich notwendigen Indizes können letztendlich häufig nur in Produktion festgestellt werden. In dem Sinne ist es sinnvoll während der Entwicklung zunächst nur die sicher notwendigen Indizes anzulegen und diese später durch Erkenntnisse aus Lasttests und Produktion zu ergänzen.

Initial sind folgende Indizes vorzusehen:

- Ein Index auf jeder Spalte, die als Fremdschlüssel verwendet wird
- Ein Index auf (fachliche) Schlüsselattribute die sehr häufig im Rahmen der Verarbeitung genutzt werden. Beispiel Nummer eines Registereintrags, Kennung einer Nachricht usw.

6. Verwendung von JPA in der Anwendung

Nachdem ein Persistenzmodell erstellt und das Mapping auf ein Datenbankschema definiert wurde (siehe Kapitel 4 und 5), können die Persistenzobjekte in der Anwendung verwendet werden. Die Verwendung der Persistenzobjekte sowie der benötigten JPA-Klassen wird in diesem Kapitel beschrieben.

6.1. Zugriff auf JPA nur über Data-Access-Objects (DAOs)

Die Persistenzfunktionen werden in Data-Access-Objects (DAOs) mithilfe des JPA Entity Managers implementiert.

Für DAO-Klassen wird in der plis-persistence die Basisschnittstelle `Dao` und deren Implementierung `AbstractDao` bereitgestellt. Diese stellen Methoden zum Anlegen, Löschen und Suchen über den Primärschlüssel bereit. Sie benötigen zwei Typparameter: Den Entitätstyp und den Typ des Primärschlüssels.

```
/**
 * Basisschnittstelle für Data Access Objects (DAOs).
 *
 * @param <T>
 *            die Entitätsklasse
 * @param <ID>
 *            die Primärschlüsselklasse
 */
public interface Dao<T, ID extends Serializable> {

    /**
     * Speichert die gegebene Entität.
     *
     * @param entitaet
     *            die Entität
     */
    public void speichere(T entitaet);

    /**
     * Löscht die gegebene Entität.
     *
     * @param entitaet
     *            die Entität
     */
    public void loesche(T entitaet);

    /**
     * Sucht eine Entität über ihren Primärschlüssel.
     *
     * @param id
     *            der Primärschlüssel
     *
     * @return die Entität, oder <code>null</code>
     *             wenn keine gefunden wurde
     */
    public T sucheMitId(ID id);

}
```

Für ein konkretes DAO ist eine eigene Schnittstelle von der Basisschnittstelle `Dao` abzuleiten. In dieser können weitere DAO-Operationen definiert werden, zum Beispiel zur Durchführung von Queries. Die Implementierungsklasse des konkreten DAOs ist von `AbstractDao` abzuleiten.

Die Basisklasse `AbstractDao` stellt den Zugriff auf den JPA-Entity Manager bereit. Dieser wird per Dependency-Injection gesetzt. Weiterhin ist sie mit der Annotation `@Repository` versehen, damit alle vom Entity Manager erzeugten Exceptions in die besser auszuwertenden Spring-DataAccessExceptions umgewandelt werden.

In der Spring-Konfiguration wird die Klasse AbstractDao wie folgt definiert:[3]

```
<!-- Factory-Bean, um den Shared-Entity-Manager für die
DAOs zu erzeugen -->
<bean id="entityManagerFactoryBean"
class="org.springframework.orm.jpa.support.SharedEntity
ManagerBean">
    <property name="entityManagerFactory"
              ref="entityManagerFactory" />
</bean>

<!-- Abstrakte Basisklasse für DAOs -->
<bean id="abstractDao"
class="de.bund.bva.pliscommon.persistence.dao.AbstractD
ao" abstract="true">
    <property name="entityManager"
              ref="entityManagerFactoryBean" />
</bean>

<!-- Diese Bean sorgt dafür, dass in mit @Repository
annotierten DAOs die JPA-Exception auf die besser
behandelbaren Spring-Persistence-Exceptions übersetzt
werden -->
<bean
class="org.springframework.dao.annotation.PersistenceEx
ceptionTranslationPostProcessor" />
```

Der Zugriff auf die Datenbank aus dem Anwendungskern heraus erfolgt immer über die DAOs. Die DAOs werden als Spring-Beans in den Anwendungskern injiziert. Zudem wird für jedes DAO ein Interface angelegt.

DAOs werden im Persistenzpaket der Komponente abgelegt, welche die Datenhoheit über die Tabelle(n) des DAOs besitzt (zum Thema

[3] Wir benutzen nicht die Annotation `@PersistenceContext` und den `PersistenceAnnotationBeanPostProcessor`, weil dieser Probleme beim Starten des Application Contexts verursacht (zum Ermitteln der EntityManagerFactory werden alle Beans zu einem (zu) frühen Zeitpunkt instanziiert, was bei zirkulären Abhängigkeiten und gewrappten Beans Probleme verursacht).

Datenhoheit siehe (IsyFact – Referenzarchitektur für IT-Systeme)). Falls die Datenhoheit keiner einzelnen Komponente zugewiesen werden kann, erhält die Komponente Basisdaten die Datenhoheit (siehe auch (Detailkonzept der Komponente Anwendungskern)). Die DAOs werden nur von Klassen der Datenhoheits-Komponente aufgerufen.

Während über DAOs Persistenzobjekte aus der Datenbank gelesen und in die Datenbank eingefügt werden, können sie auch außerhalb dieser Klassen verändert bzw. befüllt werden. Dies darf jedoch gemäß der Referenzarchitektur (IsyFact – Referenzarchitektur) nur von Klassen innerhalb der gleichen Teilanwendung erfolgen: Komponenten anderer Teilanwendungen dürfen sie nicht verändern oder befüllen. Sie erhalten daher lediglich Deep-Copies bzw. nicht änderbare Varianten der Entitäten.

Eine Ausnahme hierzu bildet die Komponente Basisdaten: Sie gibt die Entitäten an andere Komponenten weiter, welche diese verändern und befüllen dürfen.

Als Beispiel für DAOs siehe die Klassen `BestandDAO` und `CdDatenDAO` der Vorlage-Anwendung (Beispielimplementierung „Vorlage-Anwendung“).

6.2. JPQL für Datenbank-Abfragen nutzen

Für Datenbank-Abfragen stellt JPA die Java Persistence Query Language JPQL bereit. In dieser werden Queries über Objekte und Variablen, nicht über Tabellen und Felder definiert.

Wann immer möglich sollten JPQL Abfragen und keine „nativen“ SQL Abfragen verwendet werden. Der einzige Grund für die Verwendung von SQL ist die Verwendung von Oracle SQL Features, welche durch JPQL nicht angeboten werden.

Ein Beispiel für eine JPQL Anfrage findet man in Datei `NamedQueries.hbm.xml` in der Vorlage-Anwendung (Beispielimplementierung „Vorlage-Anwendung“).

6.3. Ablage von Query-Definitionen in Konfigurationsdatei

Queries (egal ob in JPQL oder SQL) sollen in einer Hibernate-Konfigurationsdatei abgelegt werden. In der Vorlage-Anwendung ist dies die Datei `NamedQueries.hbm.xml`. Von dort aus werden sie über die JPA-Sitzung als „NamedQueries“ zur Verfügung gestellt. So werden alle Query-Definitionen an einer Stelle zusammengeführt.

Falls der Query-String selbst erst zur Laufzeit zusammengesetzt wird, kann diese Query nicht in der Konfigurationsdatei abgelegt werden. Dies ist aber zu vermeiden. In diesem Fall soll zur Vermeidung von SQL-Injection Attacken die Criteria-API von JPA eingesetzt werden.

Die Konfiguration von Queries über Annotationen in Entitätsklassen ist verboten.

6.4. Verwendung von Oracle Hints bei Optimizer-Problemen

NamedQueries werden als JDBC `PreparedStatements` umgesetzt. Deshalb werden sie vom Oracle Optimizer bereits analysiert und ein Ausführungsplan erstellt, bevor ihre Parameter gebunden werden.

Dies führt in Ausnahmefällen dazu, dass ein benötigter Index für die Query-Bearbeitung nicht verwendet wird und „Full Tablescans“ durchgeführt werden.

Im Falle von Index-Problemen bei NamedQueries sind Oracle-Hints zu verwenden. Die Queries sind als native SQL-Queries in der XML Konfigurationsdatei abzulegen.

Ein Beispiel für einen Oracle-Hint in einer SQL Query:

```
select /*+ INDEX(aendno AENDERUNGS_NOTIFIKATION_STATUS)
*/ aendno from AENDERUNGS_NOTIFIKATION aendno where
aendno.status = ?1 and aendno.zeitpunktNotifikation >
:datumVon and aendno.zeitpunktNotifikation < :datumBis
```

Eine Kurzanleitung zur Verwendung von Oracle-Traces für die Ermittlung von Ausführungsplänen:

In SQL*Plus als sysdba:

```
sqlplus sys/sys@ DATA.LOCAL.VM AS SYSDBA
```

Trace für ganze DB-Instanz anschalten:

```
alter system set sql_trace=true;
```

Time-Informationen anschalten

```
alter system set timed_statistics=true;
```

Ort an dem das Trace-File liegt ermitteln:

```
select value from v$parameter
where name = 'user_dump_dest'
```

TKPROF drüberlaufen lassen, als oracle user,
damit tkprof schon gesetzt ist

```
tkprof ora_19952.trc auswertung.txt
```

Am Ende: Trace für ganze DB-Instanz abschalten:

```
alter system set sql_trace=false;
```

6.5. Verwendung von Hibernate Filtern

Parametrisierte Hibernate Filter bieten die Möglichkeit Daten zur Laufzeit mit Sichtbarkeitsregeln auszuwerten, ohne viele verschiedene Varianten von Abfragen schreiben zu müssen. Dabei können sie pro Session aktiviert oder deaktiviert werden, standardmäßig sind sie deaktiviert. Die Filter können auf Klassen- oder Collection-Ebene definiert werden und können bestehende „where“-Klauseln erweitern.

Wenn das fachliche Datenmodell variable Sichtbarkeitsregeln in größerem Umfang benötigt, sollten diese mit Hibernate Filtern umgesetzt werden. Das ersetzt eine Multiplizierung aller Abfragen.

Filter müssen als Annotationen mit @FilterDef, @Filters und @Filter umgesetzt werden.

6.6. Verbot von Bulk-Queries

JPA bietet über die Methode query.executeUpdate() die Möglichkeit in JPQL formulierte DELETE- und UPDATE-Statements, sog. Bulk-Queries, auszuführen. Die Nutzung solcher Bulk-Queries ist verboten. Wo aus Performancegründen massenhafte DELETEs oder UPDATEs direkt in der Datenbank benötigt werden, können native SQL-Anweisungen verwendet werden. Sofern bei solchen Bulk-Operationen kaskadierende Änderungen benötigt werden (z.B. weil Kind-Tabellen mitgelöscht werden sollen), müssen entsprechende Constraints in der Datenbank angelegt werden.

Begründung: Hibernate erzeugt bei der Ausführung von BULK-Queries unter bestimmten Umständen zur Laufzeit implizit Hilfstabellen (Temporäre Tabellen mit dem Präfix HT_).[4] Dies führt dazu, dass der Datenbank-User der Anwendung entsprechende CREATE TABLE-Rechte benötigt, was i.d.R. nicht zugelassen ist. Weiterhin führt die Nutzung der temporären Tabellen in vielen Fällen zu Performance-Problemen.

Um die Einhaltung dieser Anforderung sicherzustellen, sollten auch in der Entwicklung bzw. bei frühen Tests die Rechte auf die Testdatenbanken entsprechend beschränkt werden.

6.7. Sicherheitsaspekte von Anfragen

Bei der Formulierung von Anfragen sind einige Aspekte zu beachten, da ansonsten negative Auswirkungen auf die Stabilität, die Verfügbarkeit oder Sicherheit der Anwendung die Folge sind.

- Der %-Operator ist nach Möglichkeit zu vermeiden, da hiermit leicht inperformante Abfragen erzeugt werden können, die die Anwendung blockieren und die Datenbank unnötig belasten können.
- Für rein lesende Zugriffe und feste Auswertungen sind nach Möglichkeit Views zu verwenden und die Berechtigungen entsprechend zu setzen. Dadurch kann der Zugriff auf die tatsächlich benötigten Daten gesteuert und eingeschränkt werden.
- Bei der Formulierung von Anfragen sind die Eigenheiten des Optimizers des eingesetzten DMBS zu beachten.
- Es ist darauf zu achten, dass Datenbankabfragen in Anwendungen durch Indizes in der Datenbank unterstützt werden.
- Bei der Definition von Anfragen ist darauf zu achten, dass nicht zu viele Daten selektiert werden. Im Zweifel, insbesondere bei freien Anfragen, die aus Benutzereingaben erzeugt werden, sollte die Anzahl der selektierten Datensätze beschränkt werden.
- Um die SQL-Injection Attacken zu verhindern sollen Named-Queries oder Criteria-Queries verwendeten werden, bei denen der OR-Mapper für ein Escaping der Query-Parameter sorgt.

[4] siehe http://in.relation.to/Bloggers/MultitableBulkOperations

7. Konfiguration von JPA und Hibernate in der Anwendung

In den folgenden Abschnitten werden konkrete Vorgaben gemacht, welche Konfigurationen für die Umsetzung des Datenzugriffs verwendet werden sollen.

7.1. Konfiguration von JPA über Spring Beans durchführen

Spring bietet die Möglichkeit, die für die Verwendung von JPA notwendigen Klassen komplett deskriptiv über Beans zu konfigurieren und zu erzeugen. Es ist eine Vorgabe, JPA auf diese Weise zu konfigurieren.

Ein Beispiel für diese Konfiguration findet sich in Datei `jpa.xml` in der Vorlage-Anwendung. Für die Verwendung von JPA wurden in der Vorlage-Anwendung verschiedene Beans konfiguriert:

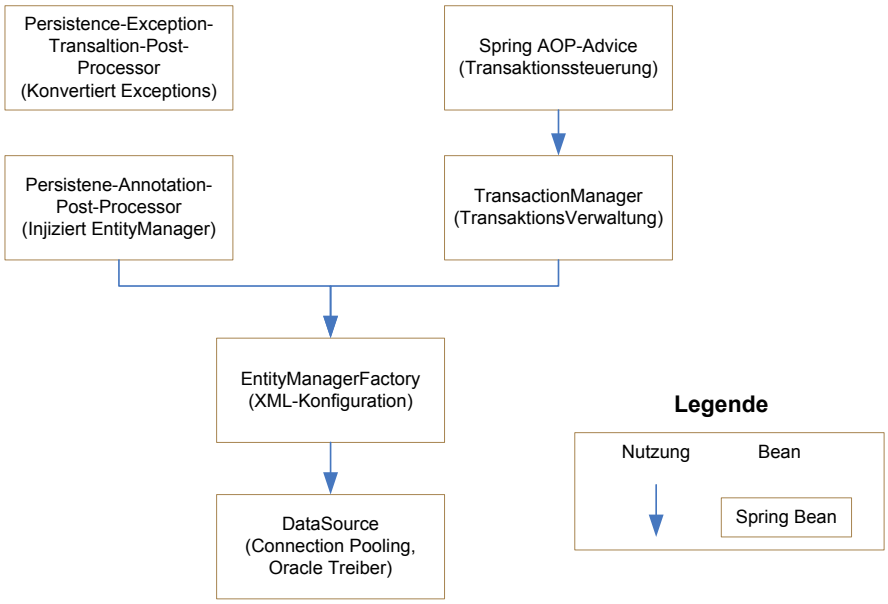

Abbildung 2: JPA-Konfiguration über Spring Beans

7.2. Konfiguration des EntityManagers

Der Zugriff auf JPA erfolgt über einen EntityManager, der wie folgt konfiguriert wird

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManage
rFactoryBean">
  <property name="persistenceProviderClass"
value="org.hibernate.ejb.HibernatePersistence" />
  <property name="persistenceUnitName"
value="hibernatePersistence" />
  <property name="dataSource">
    <ref bean="appDataSource" />
  </property>
  <property name="jpaDialect">
    <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaDialect
" />
```

```
    </property>
    <property name="jpaProperties">
      <props>
       <prop
key="hibernate.dialect">org.hibernate.dialect.Oracle10gDialec
t</prop>
       <prop key="hibernate.connection.isolation">4</prop>
       <!-- <prop key="hibernate.hbm2ddl.auto">create</prop>
        -->
       <prop
key="hibernate.connection.useUnicode">true</prop>
       <prop
key="hibernate.connection.characterEncoding">utf-8</prop>
       <prop key="hibernate.jdbc.batch_size">0</prop>
       <prop
key="hibernate.jdbc.use_streams_for_binary">true</prop>
       <prop key="hibernate.show_sql">false</prop>
       <prop key="hibernate.format_sql">false</prop>
       <prop
key="hibernate.default_schema">${database.schema.default}</pr
op>
       <!-- Hibernate-Mappings und Lazy-Loading per Default
              werden in hibernate.cfg.xml konfiguriert -->
       <prop
key="hibernate.ejb.cfgfile">/resources/persistence/hibernate.
cfg.xml</prop>
       <prop
key="hibernate.ejb.metamodel.generation">enabled</prop>
    </props>
  </property>
</bean>
```

Da Hibernate als Implementierung von JPA verwendet wird, muss in zwei Schritten JPA mit Hibernate verbunden und Hibernate selbst konfiguriert werden. Die Verknüpfung zwischen JPA und Hibernate erfolgt über die Datei `persistence.xml`. Bei Bedarf können Hibernate spezifische Parameter z.B. zusätzliche Mappings in der Datei `hibernate.cfg.xml` konfiguriert werden.

7.3. Konfiguration der Datasource

Als Datasource-Implementierung muss die Implementierung aus `de.bund.bva.pliscommon.persistence.datasource.PlisDataSource` genutzt werden:

```
<bean id="appDataSource"
class="de.bund.bva.pliscommon.persistence.datasource.PlisDataSource">
       <property name="schemaVersion" value="x.x.x" />
       <property name="invalidSchemaVersionAction" value="fail" />
       <property name="targetDataSource">
         <bean id="oracleDataSourceUcp"
class="oracle.ucp.jdbc.PoolDataSourceFactory" factory-
method="getPoolDataSource" lazy-init="true">
[…]
    </bean>
  </property>
</bean>
```

Diese prüft die Version des Datenbankschemas (siehe Abschnitt 10.5) und dient als Wrapper für die wirkliche Datasource des Connections-Pools, dessen Konfiguration im nächsten Abschnitt erläutert wird.

7.4. Oracle Universal Connection Pool (UCP) verwenden

Bei der Verwendung von JPA mit Spring **muss** zwingend ein Datenbank-Connection-Pooling verwendet werden: Die aktuelle Spring Implementierung der EntityManagerFactory fragt bei jedem Entity Manager (und somit bei jeder Transaktion) eine Datenbank-Verbindung an.

Für das Datenbank-Connection-Pooling ist der Oracle Universal Connection Pool (UCP) einzusetzen. Dieser kann auf der Oracle Website heruntergeladen werden.

Zur Laufzeit bietet der Pool Informationen per JMX an, die zur Überwachung der Poolaktivität nützlich sind. Dazu zählt unter anderem die Anzahl aktuell ausgeliehener Verbindungen.

Die Konfiguration des Pools beschränkt sich auf die Konfiguration der Klasse `oracle.ucp.jdbc.PoolDataSource` in der DataSource-Bean. Eine Instanz dieser Klasse kann über die Bean `oracle.ucp.jdbc.PoolDataSourceFactory` erzeugt werden.

Die zu setzenden Parameter können der folgenden Vorlage entnommen werden, wobei die genaue Bedeutung der Parameter der Oracle Dokumentation (Universal Connection Pool for JDBC Developer's Guide, n.d.) entnommen werden kann:

```
<bean class="oracle.ucp.jdbc.PoolDataSourceFactory" factory-
method="getPoolDataSource"
    lazy-init="true">
    <property name="connectionFactoryClassName"
value="oracle.jdbc.pool.OracleDataSource" />
    <property name="connectionPoolName" value="NAME_DES_POOLS" />
    <property name="user" value="${database.username}" />
    <property name="password" value="${database.password}" />
    <property name="URL" value="${database.url}" />
    <property name=" initialPoolSize"
value="${database.connections.initial.size}" />
    <property name="minPoolSize"
value="${database.connections.min.active}" />
    <property name="maxPoolSize"
value="${database.connections.max.active}" />
    <property name="connectionWaitTimeout"
value="${database.connections.wait.timeout}" />
    <property name="inactiveConnectionTimeout"
value="${database.connections.inactive.timeout}" />
    <property name="timeToLiveConnectionTimeout"
value="${database.connections.timetolive.timeout}" />
    <property name="abandonedConnectionTimeout"
value="${database.connections.abandoned.timeout}" />
    <property name="maxConnectionReuseTime"
value="${database.connections.max.reusetime}" />
    <property name="maxConnectionReuseCount"
value="${database.connections.max.reusecount}" />
    <property name="validateConnectionOnBorrow"
value="${database.connections.validate.onborrow}" />
```

```
    <property name="maxStatements"
value="${database.connections.statement.cache}" />
    <property name="connectionProperties">
        <props merge="default">
            <prop
key="oracle.net.disableOob">${database.jdbc.disable.oob}</prop>
            <prop
key="oracle.net.CONNECT_TIMEOUT">${database.jdbc.timeout.connect}</
prop>
            <prop
key="oracle.jdbc.ReadTimeout">${database.jdbc.timeout.read}</prop>
        </props>
    </property>
</bean>
```

Entsprechend werden in der betrieblichen Konfigurationsdatei jpa.properties folgende Properties konfigurierbar gemacht:

```
# Connection-String für die Datenbankverbindung
database.url=jdbc:oracle:thin:@database.local.vm:1521:i
syfact
# Name des Datenbankbenutzers
database.username=anwendungxyz
# Passwort für den Datenbankbenutzer
database.password=anwendungxyz
# Default-Schema für die Anwendung
database.schema.default=anwendungxyz

# Anzahl der minimal offenen Verbindungen im Connection
Cache
database.connections.min.active=5
# Anzahl der maximal moeglichen Verbindungen im Connection
Cache
database.connections.max.active=40
# Anzahl der initialen Connections im Connection Cache
database.connections.initial.size=10
# Aktiviert/deaktiviert die Pruefung von
Datenbankverbindungen vor ihrer Benutzung
(validateConnectionOnBorrow)
database.connections.validate.onborrow=true

# Zeit in Sekunden, nach der bei Nichtverfuegbarkeit einer
neue Verbindung ein Fehler geworfen wird
database.connections.wait.timeout=10
# Zeit in Sekunden, nach der eine bereitstehende und
untätige Verbindung geschlossen und aus dem Pool entfernt
wird
database.connections.inactive.timeout=120
# Zeit in Sekunden, nach der eine ausgeliehene Verbindung
wieder zwangsweise zurück in den Pool geholt wird.
# Offene Transaktionen werden zurückgerollt. Standard ist 0
(deaktiviert).
database.connections.timetolive.timeout=0
# Zeit in Sekunden, nach der eine ungenutzte aber verliehene
Verbindung wieder in den Pool geholt wird.
# Offene Transaktionen werden zurückgerollt. Standard ist 0
(deaktiviert).
database.connections.abandoned.timeout=0
```

```
# Zeit in Sekunden, nach der eine physikalische Verbindung
im Pool geordnet abgebaut wird. Sie wird erst abgebaut,
# wenn die Verbindung nicht mehr genutzt wird und zurück im
Pool ist. Kann genutzt werden, wenn bspw. Firewalls
# nach einer zeitlichen Beschränkung Verbindungen
schliessen. Standard ist 0, deaktiviert.
database.connections.max.reusetime=0
# Maximale Anzahl, die eine Verbindung ausgeliehen werden
kann, bevor sie endgueltig abgebaut wird. Standard 0
(deaktiviert)
database.connections.max.reusecount=0
# Anzahl der Statements, die pro Verbindung gecacht werden
sollen (Statement Cache). Standard ist 0 (deaktiviert).
database.connections.statement.cache=0


# --- Konfiguration des Oracle JDBC Datenbanktreibers ---
# Der Wert fuer oracle.net.CONNECT_TIMEOUT des Oracle JDBC
Treibers. Der Timeout bestimmt die maximale Zeit in ms,
# welche zum Aufbau einer Netzwerkverbindung zum
Datenbankserver gewartet wird.
database.jdbc.timeout.connect=10000
# Der Wert fuer oracle.jdbc.ReadTimeout des Oracle JDBC
Treibers. Der Timeout bestimmt die maximale Zeit in ms,
# welche auf Socketebene zum Lesen von Daten gewartet
wird.Dadurch koennen abgebrochene TCP Verbindungen erkannt
werden.
database.jdbc.timeout.read=300000
# Verbindungen können im regulären band (inband) oder
asynchron (out-of-band) beendet werden. Standardmässig
passiert das
# per OOB. Kann bei Problemen deaktiviert werden.
database.jdbc.disable.oob=true
```

Hierbei ist zu beachten, dass die hier angegebenen Werte der Konfigurationsparameter nur beispielhaft sind. Sie müssen je nach Anwendung und Lastprofil angepasst werden.

7.5. Zu verwendende Konfigurations-Properties

Zur Konfiguration der Hibernate `SessionFactory` können verschiedene Konfigurationsparameter angegeben werden. Standardmäßig sollen folgende Parameterwerte verwendet werden:

Konfiguration	Wert (Erklärung)
hibernate.dialect	org.hibernate.dialect.Oracle10gDialect
hibernate.hbm2ddl.auto	*auskommentieren bzw. nicht verwenden* (Wird nur zur initialen Generierung des Schemas verwendet)
hibernate.connection. isolation	4 (repeatable Read)
hibernate.jdbc.batch_size	0 (Es werden keine jdbc Batch-Updates verwendet)
hibernate.jdbc.	true

Konfiguration	Wert (Erklärung)
use_streams_for_binary	
hibernate.transaction. factory_class	org.hibernate.transaction. JDBCTransactionFactory

Tabelle 1: Konfigurationsproperties von Hibernate

7.6. Standardmäßig Lazy Loading verwenden

Standardmäßig verwendet Hibernate ein Lazy Loading über dynamische Proxies für alle 1:n und n:m Assoziationen. Für n:1 oder 1:1 Assoziationen wird Eager Loading eingesetzt. Dies muss überschrieben werden:

Standardmäßig soll für alle Assoziationen Lazy Loading verwendet werden. Für das Lazy Loading sollen dynamische Proxies und keine Bytecode-Manipulationen verwendet werden.

Überschrieben wird dies in einer Hibernate-Mapping-Definition (in der Vorlage-Anwendung in Datei NamedQueries.hbm.xml). Dort wird definiert:

```
<hibernate-mapping default-lazy="true">
```

Es ist erlaubt und erwünscht, dieses Verhalten für Assoziationen zu überschreiben, bei denen Eager Loading Sinn macht. Hierfür kann eine Annotation wie die folgende verwendet werden:

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

Die Verwendung der Annotationen `@LazyToOne` und `@LazyCollection` ist zu vermeiden, falls man nicht den `@LazyCollection` Wert „Extra“ für extra große Collections benötigt.

7.7. Standardmäßig optimistisches Locking verwenden

Standardmäßig ist für Hibernate ein optimistisches Locking zu verwenden: Objekte werden bei dieser Locking-Strategie nicht per „select for update“ gesperrt. Stattdessen wird am Ende der Transaktion geprüft, ob lokal veränderte Objekte parallel in der Datenbank geändert wurden. Ist dies der Fall, wird eine Ausnahme geworfen.

Dieser Vorgehensweise liegt die Annahme zugrunde, dass konkurrierende schreibende Zugriffe in einer Fachanwendung nicht oder höchstens in Ausnahmefällen vorkommen. Sollte dies nicht zutreffen, muss explizites Locking verwendet werden (vgl. Abschnitt 7.8). In der Anwendung ist keine explizite Fehlerbehandlung (etwa durch das Mergen der Daten) zu implementieren. Die geworfene Ausnahme ist (gewrappt) an den Aufrufer weiter zu geben.

Um zu erkennen, ob sich das Objekt in der Datenbank verändert hat, empfiehlt Hibernate die Verwendung eines numerischen Versions-Felds in jeder Datenbank-Tabelle. Dies ist umzusetzen. Die zugehörigen Hibernate-

Entitäten sind über folgenden Tag als optimistisch gelockt zu kennzeichnen:

```
@org.hibernate.annotations.Entity(optimisticLock =
OptimisticLockType.VERSION)
```

In den Entitäten ist die numerische Versions-Property für Hibernate zu kennzeichnen:

```
    @Version
    public int getVersion() {
        return version;
    }
```

Dieses Feld wird einzig von Hibernate verwaltet. Es ist weder zu lesen noch zu schreiben.

7.8. Bei Bedarf explizites Locking verwenden

Falls für einen Teil der Entitäten konkurrierende Zugriffe möglich sind, ist für genau diese Entitäten ein explizites (pessimistisches) Locking zu verwenden.

7.9. Transaktionsübergreifendes Caching vermeiden

Caching-Strategien sind kein Teil der JPA-Spezifikation. Für das definieren von Caching muss deswegen auf Hibernate-Spezifische Mechanismen zugegriffen werden.

Hibernate bietet für das Cachen von Objekten über Transaktionsgrenzen hinweg zwei Cache-Möglichkeiten:

- Den Cache im `Session`-Objekt. Da ein `Session`-Objekt Thread-gebunden ist, gilt dieser nur für den aktuellen Thread. In IsyFact-Anwendungen wird bei jedem Aufruf ein neues `Session`-Objekt verwendet, weshalb dieser Cache nicht verwendet werden kann.

  Bei Batches muss (wie in Kapitel 8.2 erwähnt) der Session-Cache regelmäßig geleert werden.

- Der VM-weite „2nd Level Cache“. Dieser Cache macht vor allem für unveränderliche, häufig verwendete Informationen wie Schlüsseldaten Sinn. In IsyFact-Anwendungen werden Schlüsseldaten jedoch durch einen separaten Service-Aufruf erhalten und können nicht im Hibernate 2nd Level Cache gespeichert werden. Deshalb ist eine Verwendung dieses Caches meist unnötig.

Die Verwendung von über eine Transaktion hinausgehenden Caches ist deshalb zu vermeiden. Falls auf Grund von Spezial-Anforderungen (etwa in einer Fachanwendung selbst abgelegt Sekundärdaten) ein 2nd Level Cache benötigt wird, ist auf folgende Punkte zu achten:

- Für den Cache ist eine gesonderte Cache-Region zu verwenden.
- Nur unveränderliche Daten dürfen in den Cache.

- Man kann nicht davon ausgehen, dass der Cache bei Änderungen der Objekte aktualisiert wird.

7.10. Nutzung und Anbindung einer zweiten Datenbank

Einige Anwendungsfälle machen es notwendig, eine zweite Datenbank zu nutzen. Das ist beispielsweise notwendig, wenn Daten aus einem Altsystem über die Datenbank für andere Systeme bereitgestellt werden und diese Daten in eine IsyFact-Anwendung über einen Batch importiert werden sollen. Der Batch muss dann sowohl auf die Datenbank der IsyFact-Anwendung, als auch auf die Datenbank des Altsystems zugreifen.

Die Anbindung einer zweiten Datenbank erfolgt analog zur Anbindung der primären Datenbank über Spring und die Nutzung über JPA, die in Kapitel 7.1 beschrieben ist. Dabei erfolgt der Zugriff auf die zweite Datenbank getrennt über einen weiteren Entity Manager und eine weitere Data Source.

In der Konfigurationsdatei `jpa.xml` werden die Spring-Beans für die Transaktionskontrolle, `EntityManager` und `DataSource` dupliziert und mit den Werten der zweiten Datenbank versehen. Um zu vermeiden, dass die zweite Datenbank an Stellen genutzt wird, wo es nicht geplant ist, muss allen Beans das Attribut `autowire-candidate="false"` hinzugefügt werden.

Zusätzlich muss in der Datei `persistence.xml` eine weitere Persistence Unit deklariert werden, auf die in der `EntityManagerFactory` der zweiten Datenbank verwiesen wird.

Die Datei `jpa.properties` wird um die neuen Konfigurationsparameter für die zweite Datenbankverbindung erweitert.

7.11. Konfiguration der ID und Sequenz

Primärschlüssel werden in JPA mittels der @Id und @GeneratedValue Annotation markiert. Der GenerationType der @GeneratedValue Annotation muss in jedem Fall AUTO sein. Als Generator kommt unter Oracle ein @SequenceGenerator zum Einsatz, der eine Datenbanksequenz benutzt.

Es muss unbedingt darauf geachtet werden, die Inkrementierung (INCREMENT BY) der zur ID-Generierung genutzt Datenbanksequenz auf denselben Wert einzustellen, der auch beim JPA SequenceGenerator mit allocationSize angegeben ist.

Ein Konfigurationsbeispiel kann folgendermaßen aussehen:

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO,
generator="my_seq")
```

```
@SequenceGenerator(name="my_seq",sequenceName="MY_SEQ",
allocationSize=50)
```

8. Transaktionssteuerung

Die Transaktionssteuerung definiert, wann im Kontext der Anfragebearbeitung eine Transaktion gestartet wird, wann sie beendet wird und wie auf Fehler reagiert wird. Das geforderte Verhalten hierzu wird im folgenden Abschnitt definiert. Danach wird die Umsetzung dieses Verhaltens mit JPA und Spring beschrieben.

8.1. Eine Transaktion pro Anfrage verwenden

Eine Anfrage wird innerhalb einer einzelnen Transaktion abgearbeitet: Die Transaktion wird beim Eintreffen des Aufrufs gestartet und mit der Rückgabe des Ergebnisses beendet. Da eine Fachanwendung zustandslose Dienste anbietet, ist eine langlebige Transaktion oder ein Caching über Transaktionsgrenzen hinaus nicht notwendig und zu vermeiden. Falls bei der Verarbeitung einer Anfrage ein nicht behebbarer Fehler auftritt, wird dieser an den Aufrufer zurück übermittelt. In diesem Fall wird die Transaktion nicht fortgeschrieben (committet), sondern zurückgerollt.

Geschachtelte Transaktionen, etwa um Tabellen nur kurz zu blockieren, sind zu vermeiden. Abweichungen sind mit dem Chefdesigner des Teilprojekts abzustimmen.

8.2. Bei Batch-Verarbeitung regelmäßig Session-Cache leeren

Im Batch wird die Transaktionssteuerung über den „Batch-Rahmen“ definiert. Da hier eine große Anzahl an Transaktionen innerhalb einer Session durchgeführt werden, muss der Session-Cache in regelmäßigen Abständen geflushed und geleert werden: Ansonsten würden die Objekte aller Transaktionen im Cache gehalten werden und könnten nicht durch die Garbage Collection entfernt werden.

Die Logik des Batch-Rahmens inklusive seiner Wiederaufsetzpunkte und Commit-Raten wird in einem separaten Konzept beschrieben (siehe (Batch-Verarbeitung mit Hibernate)). Im gleichen Konzept wird auch der Code genannt, der zum Löschen des Caches notwendig ist.

8.3. Transaktionssteuerung für JPA über Annotationen

Spring ermöglicht die Transaktionssteuerung mit Annotationen zu definieren. Hierbei kann auf Klassen oder Methoden-Ebene das Transaktionsverhalten vorgegeben werden.

Die Transaktionssteuerung wird im Normalfall pro Service-Klasse vorgegeben. Wichtig ist, dass die Fehlerbehandlung auf jeden Fall die Transaktion umschließt. Nur so ist gewährleistet, dass auch Fehler, die beim Commit entstehen von der Fehlerbehandlung erfasst werden.

Die Transaktionssteuerung wird per Annotation in der Service-Implementierung realisiert. Dabei wird im Normalfall der Propagation-Level auf „required“ gesetzt und festgelegt, dass bei jedem Fehler ein Rollback durchgeführt wird.

Im der Beispiel-Anwendung ist das wie folgt umgesetzt:

```
@Transactional(rollbackFor = Throwable.class,
propagation = Propagation.REQUIRED)
public class MeldungServiceImpl implements
MeldungService {
```

Damit Spring die Annotationen auswertet, muss folgende Konfiguration durchgeführt werden:

```
<!-- Transaktionssteuerung per Annotationen -->
<tx:annotation-driven transaction-
manager="transactionManager"/>
```

Durch diese Konfiguration erzeugt Spring für die Annotationen passende AOP-Proxies, welche die Transaktionssteuerung übernehmen.

8.4. Bei REST-Webservices Rollback explizit anfordern

Eine Sonderstellung nehmen REST-Webservices ein, da diese im Fehlerfall keine Exceptions werfen, sondern die Fehler in der Antwortnachricht übermitteln: Der AOP-Transaktionsmanager wird niemals ein Rollback durchführen, da alle Exceptions abgefangen werden, auf die er reagieren könnte. Um auch in diesem Fall ein Rollback der Transaktion zu erzwingen, ist ein expliziter Aufruf durchzuführen:

```
TransactionAspectSupport.currentTransactionStatus().set
RollbackOnly();
```

Im der Vorlage-Anwendung findet man hierfür kein Beispiel, da auf ein separates REST-Skeleton verzichtet wurde.

9. Historisierung

9.1. Grundlagen

Unter Historisierung (auch temporale Datenhaltung genannt (Demelt)) versteht man das Festhalten der zeitlichen Entwicklung von Daten durch Speichern in einer Datenbank. Bei den Datensätzen gibt es zwei relevante Aspekte: Den Gültigkeitszeitraum eines Datensatzes und den Bearbeitungszeitpunkt eines Datensatzes.

Der Gültigkeitszeitraum gibt an, wie lange ein Datensatz gültig ist. Während der Beginn des Gültigkeitszeitraumes meistens genau bekannt ist, so kann das Ende der Gültigkeit so lange unbekannt sein, bis der Datensatz ungültig wird. Beispiel: Der Preis einer Ware oder Dienstleistung ist so lange gültig, bis er neu festgelegt wird.

Der Bearbeitungszeitpunkt definiert den Zeitpunkt wann eine Entscheidung getroffen wurde und ist in vielen Fällen identisch mit dem Beginn des Gültigkeitszeitraumes , kann jedoch auch davon abweichen, wenn z. B. für eine Ware eine Preisänderung zu einem bestimmten Datum im Voraus festgelegt wird.

Eine Historisierung von Datensätzen wird durchgeführt, wenn Fragen über den Wert eines Datensatzes zu einem vergangenen Zeitpunkt beantwortet werden müssen (z. B. Was kostete X zum Zeitpunkt Y), oder wenn der Verlauf eines Wertes über die Zeit beobachtet werden muss (z. B. Wann und warum wurde welche Änderung durchgeführt?).

9.1.1 Abgrenzung Archivierung

Bei der Archivierung handelt es sich um die Aufbewahrung eines Datensatzes über eine längere Zeit. Dies ist meist aus rechtlichen Gründen notwendig z. B. wegen gesetzlicher Aufbewahrungsfristen. Bei der Archivierung sind dementsprechend Randbedingungen wie Integrität, Unveränderlichkeit und Vertraulichkeit einzuhalten (Informationstechnik).

9.1.2 Abgrenzung Datensicherung (Backup)

Bei der Datensicherung handelt es sich um das redundante Aufbewahren von Datensätzen. Das Ziel ist es, bei Verlust oder ungewünschter Manipulation von Datensätzen diese Datensätze auf den gespeicherten Stand zurücksetzen zu können.

9.1.3 Abgrenzung Protokollierung

Ziel der Protokollierung ist das Nachvollziehen von Änderungen und Auskünften. Dazu werden je nach Bedarf die Suchschlüssel und Nettodaten von Aufrufen gespeichert.

9.1.4 Abgrenzung Logging

Beim Logging werden Notizen zu technischen Aufrufen innerhalb eines Systems oder zwischen Anwendungen in Dateien abgelegt. Das Logging

hat einen technischen Fokus und dient in der Regel als Hilfsinstrument zur Fehlerbehebung.

9.2. Anforderungen

Die beabsichtigte Nutzung der Historisierung lässt sich mit Blick auf die Referenzarchitektur zu Anforderungen verallgemeinern, die in diesem Abschnitt dargestellt werden.

Für die Historisierung von Datensätzen in einer Anwendung gelten folgende Anforderungen und Grundsätze:

- Es dürfen nur solche Daten historisiert werden, die auch angezeigt werden.
- Die Speicherung von historischen Daten wird durch individuelle Löschfristen von Datensätzen begrenzt.
- Datensätze müssen beim Eintreten bestimmter Ereignisse komplett inklusive aller historisierten Datensätze gelöscht werden.
- Für die meisten Daten ist eine Historisierung weder notwendig noch erlaubt. Dies ist durch Vorgaben des Datenschutzes und der Geheimhaltung begründet.

Diese Anforderungen führen zu folgenden Festlegungen:

- Eine automatische Historisierung von Daten, bei der jeder Datensatz in mehreren Versionen vorgehalten ist, wird nicht realisiert.
- Sollte es fachlich gewünscht sein, so wird explizit für die betroffenen Datensätze ein Historienverwalter implementiert, dessen Aufgabe die Historisierung von Datensätzen ist.

Die Referenzarchitektur dieses Historienverwalters ist im folgenden Kapitel beschrieben.

9.3. Architektur für die Umsetzung von Historisierung

In diesem Kapitel wird beschrieben, wie die technische Umsetzung der Historisierung erfolgt. Dabei werden die beiden in Kapitel 9.1 eingeführten Aspekte der Historisierung „Gültigkeitszeitraum“ und „Verlauf der Bearbeitung“ getrennt beschrieben, wobei der zweite Aspekt aufwändiger umzusetzen ist und daher den Großteil des Kapitels einnimmt.

9.3.1 Abbildung eines Gültigkeitszeitraums

Manche Daten haben einen Zeitbezug, d. h. der Inhalt eines Datensatzes bezieht sich nur auf einen bestimmten Zeitraum. Man möchte z. B. beschreiben, dass für eine Ware in einem bestimmten Zeitraum ein Rabatt gewährt wird. Um einen solchen Gültigkeitszeitraum abzubilden, werden zu dem ursprünglichen Datensatz zwei zusätzliche Datumsattribute ergänzt. Falls diese Datumsattribute bereits fachlich etablierte Namen haben, werden diese genutzt. Sonst werden die Namen `gueltigVon` und `gueltigBis` benutzt. Diese Attribute werden durch die Anwendung genauso gepflegt wie alle anderen Attribute des Datensatzes auch.

9.3.2 Abbildung der Historie der Bearbeitung

In diesem Abschnitt wird beschrieben, wie die Historie der Bearbeitung gepflegt werden soll, z. B. wenn die letzten zehn Änderungen zu einem Datensatz abgespeichert werden sollen. Dazu wird zunächst beschrieben, wie die prinzipielle Herangehensweise dazu ist. Anschließend wird dies durch Angabe eines Entwurfsmusters präzisiert.

Die grundlegenden Prinzipien bei der technischen Abbildung sind die, dass Historisierung explizit durchgeführt wird, dass die Nutzungsvorgabe in Form eines Patterns erfolgt und dass die Historisierungslösung konsistent mit den bereits getroffenen Festlegungen zur Persistenz sein soll.

Explizite Historisierung: Die Historisierung der Bearbeitung erfolgt explizit, d. h. die zu historisierenden Daten werden durch die Anwendungslogik gepflegt und persistiert.

Theoretisch wäre es auch möglich, eine solche Historisierung auf der Ebene der Datenbankzugriffsschicht durchzuführen. Dazu würden dann in der Datenbankzugriffschicht die UPDATE-Statements durch INSERT-Statements ersetzt. Die Daten der INSERT-Statements würden dann durch einen Zeitstempel ergänzt. Beim SELECT würde immer der aktuellste Datensatz geliefert werden. Dieses Vorgehen lohnt sich aber nicht, da nur sehr wenige Datensätze historisiert werden sollen und ebenso widerspricht es der Anforderung, dass keine Daten gespeichert werden sollen, die nicht auch angezeigt werden. Sinnvoll wäre ein solches Vorgehen dann, wenn über die Historisierung eine Nachvollziehbarkeit der Änderungen erreicht werden soll. Dies ist im Rahmen der Referenzarchitektur aber explizit die Aufgabe der Protokollierung.

Historisierung durch Vorgabe eines Patterns: Die beschriebene Historisierungsfunktionalität lässt sich nur schwer in der Form von Bibliotheken mit abstrakten Oberklassen, Interfaces und ähnlichem abbilden. Die dadurch entstehenden Java-Konstrukte wären nur sehr sperrig zu nutzen und würden die Entwicklung eher behindern als beschleunigen. Deshalb wird in diesem Dokument ein Entwurfsmuster vorgegeben, nach dem die Historisierung zu erfolgen hat. Diese Entwurfsmuster sind für den Entwickler leichter zu handhaben.

9.4. Vorgehen zur Historisierung der Bearbeitung

9.4.1 Schritt 1: Ergänzen von Datumsattributen

Historisierte Versionen und die aktuelle Version eines Datensatzes werden in der gleichen Tabelle gepflegt. Dazu wird diese um zwei neue Datumsattribute erweitert: `aktuellVon` und `aktuellBis`. Der aktuell gültige Datensatz ist somit der mit dem neuesten `aktuellVon`-Datum. Das `aktuellBis`-Datum vereinfacht den Zugriff auf die Tabelle per SQL. Es wird dadurch einfacher, den Datensatz zu finden, der zu einem bestimmten Datum aktuell war. Das Attribut `aktuellBis` des aktuellsten Datensatzes wird per Konvention auf das Datum 31.12.9999 gesetzt. Damit

kann dieses Attribut zur Ermittlung des aktuellen Datensatzes genutzt werden. Der Chefdesigner eines Projekts kann festlegen, dass dieses Attribut Teil des Schlüssels ist. Dadurch ist es möglich, die Tabelle der Datenbank zu partitionieren, um die Verarbeitungsgeschwindigkeit zu erhöhen.

In Ausnahmefällen darf auch eine eigene Tabelle zur Speicherung der Historie angelegt werden. Dies muss der Chefdesigner eines Projekts entscheiden. Dabei ist zu beachten, dass dadurch der Datenzugriff verlangsamt wird, da in diesem Fall immer zwei Tabellen statt einer geschrieben werden.

Durch das Einführen der Datumsattribute erweitert sich der fachliche Schlüssel des Datensatzes. Der somit aus mehreren Attributen zusammengesetzte fachliche Schlüssel wird genauso behandelt, wie jeder andere zusammengesetzte fachliche Schlüssel auch.

9.4.2 Schritt 2: Erweiterung des DAOs

Alle Datenzugriffe auf zu persistierende Objekte werden über das zugehörige DAO (Data Access Object) vorgenommen. Insbesondere muss das DAO auch dafür sorgen, dass die Attribute `aktuellVon` und `aktuellBis` mit den korrekten Werten belegt sind.

Falls das zu persistierende Objekt den Namen `Xyz` trägt, heißt das zugehörige DAO `XyzDao`. Es hat die Funktion eines Datenverwalters. Dieses DAO wird wie folgt angepasst und erweitert:

Erstellen einer neuen Methode `Xyz leseXyz(Schlüssel, Calendar)`: Durch die Historisierung wird der bisherige Schlüssel des Objekts um einen Datumsbezug erweitert. Daher muss jetzt beim Lesen eines Objekts ein Datum angegeben werden, an dem das zu lesende Objekt aktuell sein soll. Diese Methode liefert das Objekt mit dem übergebenen Schlüssel, das zum übergebenen Datum aktuell war.

Ändern der Methode `Xyz lese Xyz(Schlüssel)`: Diese Methode ist im DAO bereits vorhanden. Sie wird so angepasst, dass sie das aktuell gültige Objekt zurückgibt. Dies ist das Objekt mit den übergebenen Schlüsselattributen, dessen `aktuellBis`-Eintrag der 31.12.9999 ist.

Erstellen einer neuen Methode `List<Xyz> leseXyzHistorie (Schlüssel)`: Diese Methode liefert die gesamte Historie eines Datensatzes.

Erstellen einer neuen Methode `Xyz erzeugeNeueVersion(Xyz)`: Bei einer Umsetzung ohne Historisierung konnten Objekte direkt über ihren Konstruktor erzeugt werden und mit Hilfe der Methode `speichereXyz(Xyz)` persistiert werden. Dies ist jetzt nicht mehr

möglich, da in diesem Fall die Attribute `aktuellVon` und `aktuellBis` nicht korrekt belegt werden würden. Daher bietet das DAO eine Methode an, um auf Basis eines bestehenden Objekts eine neue Version dieses Objekts zu erstellen. Die Idee dabei ist, dass das bisher aktuelle Objekt einen Nachfolger erhält. Beim bisher aktuellen Objekt wird vermerkt, dass es nicht mehr aktuell ist und das neu erzeugte Objekt wird als aktuelles Objekt gekennzeichnet. Im Detail werden dabei die folgenden Schritte durchgeführt:

- Ausgangslage: Das bisher aktuelle Objekt wird als Parameter übergeben.
- Schritt 1: Der Zeitstempel des übergebenen Objekts wird verändert und damit dieses Objekt als nicht mehr aktuell markiert. Das übergebene Objekt ist das bisher aktuelle Objekt, der Zeitstempel `aktuellBis` war bisher auf den 31.12.9999 gesetzt. Dieser Zeitstempel wird auf den aktuellen Zeitstempel gesetzt.
- Schritt 2: Es wird ein neues Objekt `Xyz` erzeugt.
- Schritt 3: Der Zeitstempel `aktuellVon` des neu erzeugten Objekts wird auf den aktuellen Zeitstempel gesetzt.
- Schritt 4: Die Daten des übergebenen Objekts werden in das aktuelle Objekt kopiert.
- Schritt 5: Der Zeitstempel `aktuellBis` wird auf den 31.12.9999 gesetzt. Damit ist es als das aktuelle Objekt gekennzeichnet.
- Schritt 6: Das neue Objekt wird in der Session des Persistenzmanagers registriert, damit es beim späteren `commit` persistiert wird.

Als Parameter der Methode darf auch `null` übergeben werden. In diesem Fall wird ein neuer, leerer Datensatz angelegt, dessen Zeitstempel aber korrekt befüllt sind. Dies ist nötig, um das erste Objekt einer Historie erzeugen zu können.

Nach konkretem Bedarf kann die Methode `Xyz erzeugeNeueVersion()` auch noch durch zusätzliche „convenience“-Methoden ergänzt werden, die andere Parameter erwarten, z. B. durch eine Methode, die als Parameter nur die Schlüsselwerte des Objekts und nicht das Objekt selbst erwartet oder durch eine Methode, die die aktuellste Version eines Datensatzes selber ermittelt.

Löschen der Methode `void speichereXyz(Xyz)`: Es ist nicht mehr möglich, ein neues Objekt zu erzeugen und direkt in der Datenbank zu speichern und damit die Historisierung zu umgehen.

Optionale Erweiterungen: Falls eine Obergrenze für die Anzahl der zu historisierenden Datensätze vorgegeben ist, wird die Einhaltung dieser Obergrenze ebenfalls durch das DAO sichergestellt. In diesem Fall wird bei der Erzeugung einer neuen Version geprüft, ob dadurch die Obergrenze überschritten wird und ggf. die älteste Version gelöscht. Der Wert dieser Obergrenze wird in einer Klassenkonstante des DAOs

gehalten. Diese Klassenkonstante ist `public`, damit deren Wert bei einer Veränderung der Historie außerhalb des DAOs berücksichtigt werden kann. Sie trägt den Namen `MAX_EINTRAEGE_HISTORIE`.

Es wurden in der Schnittstelle des DAOs bewusst keine Funktionen vorgesehen, um die Historie verändern zu können. Der Regelfall ist der, dass die Zeitstempel automatisch durch den Historienverwalter gesetzt werden und die Historie nicht mehr verändert wird.

Eine Veränderung der Historie ist technisch nicht ausgeschlossen, dies kann direkt durch die Bearbeitung der historisierten Datensätze geschehen. Dies ist allerdings ein fachlicher Ausnahmefall. Im Regelfall darf die Historie nicht verändert werden. Änderungen der Historie dürfen nur in Abstimmung mit den fachlichen Chefarchitekten vorgenommen werden.

9.4.3 Beispiel

Das fachliche Szenario für dieses Beispiel ist das Folgende: Der Bestand einer CD soll historisiert werden.

Schritt 1: Ergänzen von Datumsattributen

Der Bestand der CDs ist ohne Historisierung wie in Abbildung 3 modelliert.

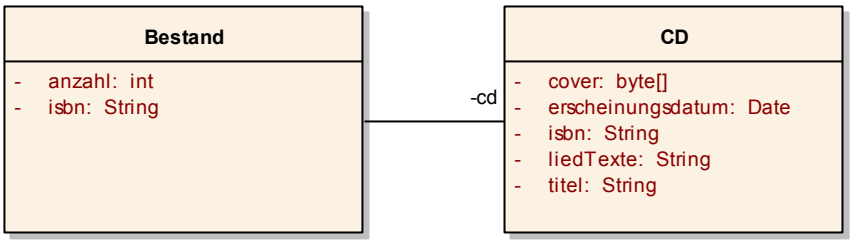

Abbildung 3: Modellierung des Bestands ohne Historisierung

Es gibt eine Entität `CD`, die eine konkrete CD repräsentiert. Der Schlüssel dieser CD ist die `isbn`. Der Bestand dieser CD wird in einer separaten Entität `Bestand` vorgehalten. Die Relation zwischen `Bestand` und `CD` ist eine 1:1-Relation. Eventuell könnte diese Relation in der Datenbank so modelliert werden, dass sowohl `Bestand` als auch `CD` in einer Tabelle zusammengefasst sind. Um den Bestand historisierbar zu machen, müsste diese Tabelle in zwei Tabellen zerlegt werden.

In die Entität Bestand werden die Attribute `aktuellVon` und `aktuellBis` eingefügt. Dies ist in Abbildung 4 dargestellt.

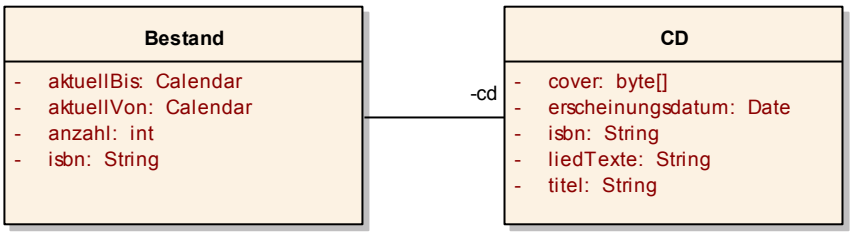

Abbildung 4: Modellierung des Bestands mit Historisierung

Schritt 2: Erweiterung des DAOs

Das DAO für die Entität Bestand ohne Historisierung ist in Abbildung 5 dargestellt.

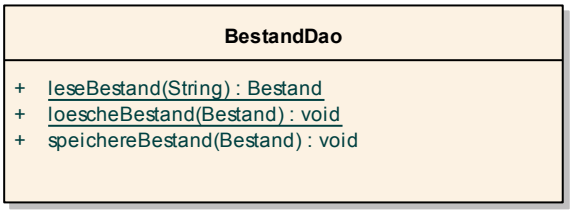

Abbildung 5: BestandDao ohne Funktionen zur Historisierung

Um ein neues Objekt Bestand zu persistieren, wird eine Instanz von `Bestand` erzeugt und anschließend `speichereBestand(Bestand)` aufgerufen. Die Methode `leseBestand(String)` liest den Bestand einer CD, die durch den übergebenen String (die `isbn`) identifiziert wird. Die Methode `loescheBestand(Bestand)` löscht den Datensatz aus der Datenbank. Um den Bestand historisierbar zu machen, werden die folgenden Erweiterungen vorgenommen, die in Abbildung 6 dargestellt sind.

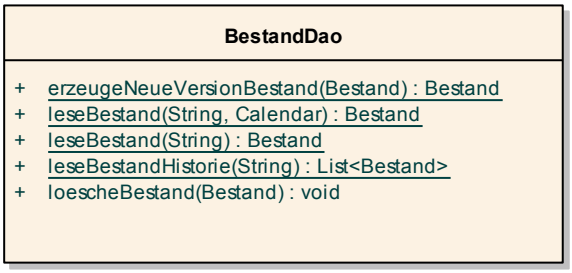

Abbildung 6: BestandDao mit Erweiterungen für Historisierung

Die Methode `erzeugeNeueVersionBestand(Bestand)` wurde eingefügt.

Die Methode `leseBestand(String, Calendar)` wurde eingefügt.

Die Methode `leseBestand(String)` wurde geändert, so dass der aktuelle Datensatz geliefert wird.

Die Methode `leseBestandHistorie(String)` wurde eingefügt.

Die Methode `speichereBestand(Bestand)` wurde entfernt.

10. Versionierung von Datenbankschemas

Die Struktur der Daten, die von einer Anwendung dauerhaft gespeichert werden, kann sich im Laufe des Lebenszyklus der Anwendung ändern. Das bedeutet, dass sich neben der Anwendung auch das Datenbankschema ändert. Die Anwendung und das Datenbank Schema müssen zueinander passen.

Die Verwaltung von Versionsinformationen für ein Datenbankschema innerhalb der Datenbank soll sicherstellen, dass die Anwendung und Datenmigrationsskripte erkennen können, ob ein Datenbankschema die erwartete Version hat. Zusätzlich sollen die Datenbankadministratoren nachvollziehen können, welche Änderungen am Datenbankschema bereits erfolgt sind.

Die Versionsnummer eines Datenbankschemas ist gleich der Versionsnummer der Anwendung, mit der das Schema angelegt bzw. zuletzt geändert wurde. Damit ist auf einen Blick zu erkennen, welche Versionsnummer eine Anwendung mindestens haben muss, um mit dem Schema zusammenarbeiten zu können.

Wird nur eine Anwendung geändert, das Datenbankschema aber nicht, so bleibt die Versionsnummer des Datenbankschemas sowohl in der Anwendung als auch in den Datenbank-Skripten unverändert. Nur die Versionsnummer der Anwendung selbst wird erhöht.

Zusätzlich wird ein Update-Zähler mitgeführt, der jedes Mal hochgezählt wird, wenn sich das Datenbankschema ändert, aber die Anwendung unverändert bleibt. Das ist z.B. dann der Fall, wenn zusätzliche Indexe angelegt werden oder Views, die die Anwendung selbst nicht benötigt.

Im Folgenden wird ein Verfahren festgelegt das diese Anforderungen umsetzt.

10.1. Struktur der Versionsmetadaten

Die Informationen über Versionen und durchgeführte Änderungen an einem Datenbankschema werden innerhalb des Schemas in eigenen Metadatentabellen gespeichert. Hierzu muss jedes Datenbankschema die folgenden Tabellen enthalten.

10.1.1 Tabelle M_SCHEMA_VERSION

Die Tabelle M_SCHEMA_VERSION enthält die Information über die aktuelle Version des Schemas. Die Tabelle hat die folgende Struktur:

Spalte	Typ	Beschreibung
version_nummer	varchar2(25 char)	Versionsnummer des Datenbankschemas. Diese Versionsnummer entspricht der Versionsnummer der Anwendung, mit der sich das

Spalte	Typ	Beschreibung
		Schema geändert hat.
update_nummer	varchar2(5 char)	Update-Zähler, der jedes Mal hochgezählt wird, wenn sich das Datenbankschema ändert, aber die Anwendung unverändert bleibt.
status	varchar2(25 char)	Status des Schemas: • gueltig: Das Schema wurde korrekt installiert bzw. aktualisiert und kann verwendet werden. • ungueltig: Das Schema befindet sich im Aufbau bzw. in der Änderung oder die Installation wurde nur teilweise durchgeführt und wurde mit Fehlern abgebrochen. Das Schema kann nicht verwendet werden und muss überprüft werden.

Tabelle 2: Tabelle M_SCHEMA_VERSION

10.1.2 Tabelle M_SCHEMA_LOG

Die Tabelle M_SCHEMA_LOG enthält Information über eingespielte Skripte zur Anpassung des Schemas. Die Tabelle hat die folgende Struktur:

Spalte	Typ	Beschreibung
schemaversion	varchar2(25 char)	Versionsnummer des Schemas, zu dessen Erstellung bzw. Anpassung das Skript genutzt wurde.
schemaupdate	varchar2(5 char)	Update-Zähler, der jedes Mal hochgezählt wird, wenn sich das Datenbankschema ändert, aber die Anwendung unverändert bleibt.
schritt	varchar2(10 char)	Nummer des Schrittes im Installationsablauf.
beschreibung	varchar2(100 char)	Kurzbeschreibung des Installationsschrittes.
skript	varchar2(100 char)	Name des ausgeführten Skripts.
skript_start	timestamp	Zeitpunkt, an dem das Skript gestartet wurde.

Spalte	Typ	Beschreibung
skript_ende	timestamp	Zeitpunkt, an dem das Skript beendet wurde.
status	varchar2(25 char)	Status der Skriptausführung: • wird ausgeführt: Skript wurde gestartet und läuft oder wurde abgebrochen • erfolgreich: Skript wurde erfolgreich abgearbeitet

Tabelle 3: Tabelle M_SCHEMA_LOG

10.2. Installationsablauf bei der Neuanlage

Die Neuinstallation eines Datenbankschemas erfolgt in mehreren Schritten, die jeweils aufeinander aufbauen. Für die automatisierte Installation werden diese Schritte von einem Datenbankskript nacheinander durchgeführt.

Schritt 1: Umgebungsvariablen laden

Für Testzwecke ist es erforderlich, Datenbankschemas in unterschiedlichen Umgebungen zu installieren. Umgebungsspezifische Konfigurationsparameter, wie z.B. der Schemaname oder die Angaben zur Datenbankverbindung werden in einem eigenen Datenbankskript abgelegt, das Umgebungsvariablen mit den entsprechenden Werten setzt. Die übrigen Installationsschritte verwenden dann diese Variablen.

Schritt 2: Tablespace erstellen

Erstellen aller Tablespaces, die für die Installation der Datenbankobjekte benötigt werden.

Schritt 3: Benutzer anlegen

Anlegen aller Datenbankbenutzer einschließlich ihrer Rollen und Berechtigungen. Mit diesen Benutzern werden die anwendungsspezifischen Datenbankobjekte angelegt. Es müssen daher alle hierfür benötigten Rechte für die Dauer der Installation gesetzt werden.

Schritt 4: Erzeugen der anwendungsspezifischen Datenbankobjekte

Es werden alle Tabellen, Indexe, Views, Prozeduren und Funktionen für die Anwendung angelegt. Weiterhin werden benötigte spezielle Datenbankobjekte, z.B. für das Oracle-Advanced-Queuing angelegt. Die anwendungsspezifischen Datenbankobjekte werden mit den in Schritt 3 angelegten Benutzern erstellt.

Schritt 5: Abschlussbearbeitung

In diesem Schritt können alle Operationen ausgeführt werden, die sich auf die bisher angelegten Datenbankobjekte beziehen.

Schritt 6: Rechte entziehen

Falls den Benutzern im Schritt 3 Rechte zugewiesen wurden, die nur

für die Installation benötigt wurden, werden sie in diesem Schritt wieder entzogen.

Die nachfolgende Abbildung zeigt noch einmal die einzelnen Schritte der Installation.

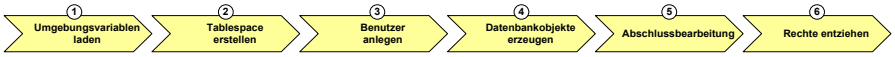

Abbildung 7: Installationsablauf bei der Neuanlage

10.2.1 Struktur der Installationsskripte für die Neuanlage

Für die automatisierte Installation wird eine Strukturierung der Installationsskripte festgelegt. Es existieren folgende Aufrufbeziehungen:

Shell-Skripte: Über die Shell-Skripte `install-db-schema.bat` (Windows) bzw. `install-db-schema.sh` (Linux) wird das SQL-Skript `00_install-main.sql` aufgerufen. Als Parameter werden das Skript für das Anlegen der Umgebungsvariablen und die Log-Datei mitgegeben.

00_install-main.sql: Das SQL-Skript ruft die eigentlichen Installationsskripte in der richtigen Reihenfolge über das Hilfsskript `99_starte-skript-mit-logging.sql` nacheinander auf. Dabei werden auch die Tabellen zur Versionierung angelegt und korrekt gefüllt.

99_starte-skript-mit-logging.sql: Das Hilfsskript führt ein SQL-Skript aus und befüllt die Versionstabelle korrekt. Als Parameter werden der Pfad des Skripts, die Schnittnummer inklusiv der Unterschrittnummer und die Beschreibung mit übergeben.

<Installationsskript>.sql: Die eigentlichen Installationsskripte haben das feste Namensschema:
`<Schrittnummer>-<Unterschrittnummer>_<Name>.sql`
Die Schrittnummer ist 2-stellig und entspricht der Schrittnummer aus Kapitel 10.2. Falls zu einem Schritt mehrere Skripte gehören, gibt die Unterschrittnummer die Reihenfolge an, in der diese ausgeführt werden. Der Name kann frei vergeben werden, sollte aber sprechend sein.

Die nachfolgende Abbildung zeigt noch einmal die Beziehung zwischen den einzelnen Skripten.

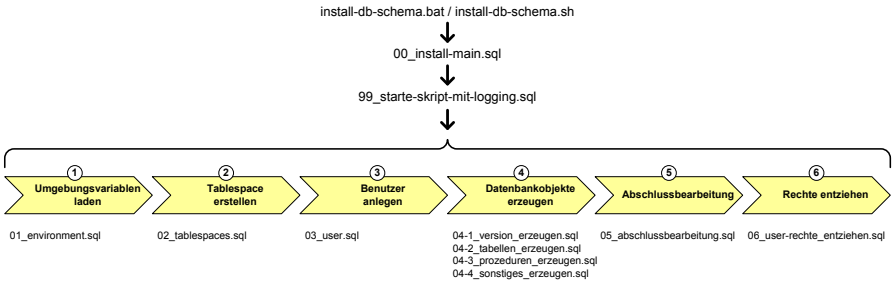

Abbildung 8: Beziehungen zwischen den Installationsskripten

Templates für die Skripte sind als Ressourcen in der Bibliothek plis-persistence abgelegt.

10.3. Installationsablauf bei der Schemaänderung

Die Änderung eines Datenbankschemas erfolgt analog zur Neuanlage ebenfalls in mehreren Schritten, die jeweils aufeinander aufbauen. Für die automatisierte Änderung werden diese Schritte von einem Datenbankskript nacheinander durchgeführt.

Schritt 1: Umgebungsvariablen laden

Dieser Schritt unterscheidet sich nicht von der Neuanlage. Je nach Art der durchzuführenden Änderung kann es aber erforderlich sein, hier weitere Variablen zu setzen.

Schritt 2: Rechte setzen

Falls erforderlich, werden für den Benutzer, mit dem die Änderungen durchgeführt werden sollen, alle für die Änderung des Datenbankschemas benötigten Berechtigungen gesetzt.

Schritt 3: Durchführen der Schemaänderungen

Es werden alle Änderungen am Datenbankschema vorgenommen. Das umfasst sowohl das Anlegen neuer Datenbankobjekte, wie z.B. Tabellen, Views und Indexe, als auch die Änderung bereits vorhandener Datenbankobjekte, wie z.B. das Löschen und Hinzufügen von Spalten in Tabellen. Die Änderungen werden mit dem Benutzer durchgeführt, für den in Schritt 2 die Berechtigungen entsprechend gesetzt wurden.

Schritt 4: Abschlussbearbeitung

In diesem Schritt können alle Operationen ausgeführt werden, die sich auf die bisher angelegten Datenbankobjekte beziehen.

Schritt 5: Rechte entziehen

Falls in Schritt 2 für Benutzer Rechte gesetzt wurden, die nur für die Durchführung der Änderungen benötigt wurden, werden sie in diesem Schritt wieder entzogen.

Die nachfolgende Abbildung zeigt noch einmal die einzelnen Schritte der Installation.

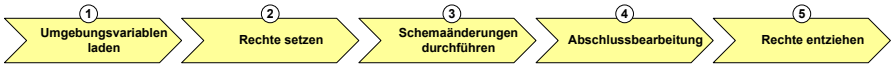

Abbildung 9: Ablauf bei der Schemaänderung

10.3.1 Struktur der Änderungsskripte

Für die automatisierte Änderung wird eine Strukturierung der Änderungsskripte festgelegt. Diese ist analog zur Struktur der Installationsskripte aus Abschnitt 10.2.1. Es existieren folgende Aufrufbeziehungen:

Shell-Skripte: Über die Shell-Skripte `update-db-schema.bat` (Windows) bzw. `update-db-schema.sh` (Linux) wird das SQL-Skript `00_update-main.sql` aufgerufen. Als Parameter werden das Skript für das Anlegen der Umgebungsvariablen und die Log-Datei mitgegeben.

00_update-main.sql: Das SQL-Skript ruft die eigentlichen Installationsskripte in der richtigen Reihenfolge über das Hilfsskript `99_starte-skript-mit-logging.sql` nacheinander auf. Dabei werden auch die Tabellen zur Versionierung angelegt und korrekt gefüllt.

99_starte-skript-mit-logging.sql: Das Hilfsskript führt ein SQL-Skript aus und befüllt die Versionstabelle korrekt. Als Parameter werden der Pfad des Skripts, die Schnittnummer inklusiv der Unterschrittnummer und die Beschreibung mit übergeben.

<Update-Skript>.sql: Die eigentlichen Änderungsskripte haben das feste Namensschema `<Schrittnummer>-<Unterschrittnummer>_<Name>.sql`. Die Schrittnummer ist 2-stellig und entspricht der Schrittnummer aus Kapitel 10.3. Falls zu einem Schritt mehrere Skripte gehören, gibt die Unterschrittnummer die Reihenfolge an, in der diese ausgeführt werden. Der Name kann frei vergeben werden, sollte aber sprechend sein.

Die nachfolgende Abbildung zeigt noch einmal die Beziehung zwischen den einzelnen Skripten.

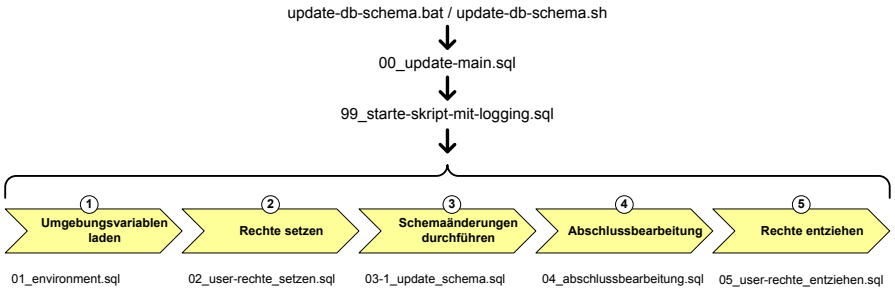

Abbildung 10: Beziehungen zwischen den Änderungsskripten

Templates für die Skripte sind als Ressourcen in der Bibliothek plis-persistence abgelegt.

10.4. Ablage der Skripte und Namenskonventionen

Die Skripte werden im Source-Projekt der Anwendung abgelegt, zu der sie gehören und zwar im Verzeichnis „src/main/skripte/sql/[dbschema-name]“. In diesem Verzeichnis werden das Unterverzeichnis „db-install-<Versionsnummer>“ für Installationsskripte und das Unterverzeichnis db-update-<Versionsnummer> für die Updateskripte angelegt.

<Versionsnummer> gibt dabei die Versionsnummer des Datenbankschemas einschließlich der Update-Nummer an, z.B. 1.2.3_01.

Für jede Datenbankversion muss es ein vollständiges Installationsskript geben. Wurden Änderungen am Schema vorgenommen, gibt es zusätzlich ein entsprechendes Update-Skript von der Vorversion.

Die einzelnen Skripte werden auf der Grundlage der Templates aus plis-persistence erstellt und behalten auch deren Namen.

Die eigentlichen Installations- und Update-Skripte haben das feste Namensschema:
`<Schrittnummer>-<Unterschrittnummer>_<Name>.sql`
Die Schrittnummer ist 2-stellig und entspricht der Schrittnummer aus Kapitel 10.2 bzw. 10.3. Falls zu einem Schritt mehrere Skripte gehören, gibt die Unterschrittnummer die Reihenfolge an, in der diese ausgeführt werden. Der Name kann frei vergeben werden, sollte aber sprechend sein.

10.5. Prüfen der Schema-Version

Jede Anwendung prüft beim Start, ob das DB-Schema die erwartete Version hat. Diese Prüfung ist in der Bibliothek plis-persistence ab Version 1.1.0 fest eingebaut. In der Spring Konfiguration müssen in der `appDataSource-Bean` (Klasse `de.bund.bva.pliscommon.persistence.datasource.PlisDataSource`) die folgenden Properties gesetzt werden:

```
<property name="schemaVersion" value="1.2.3" />
<property name="schema" value="dbschema" />
<property name="invalidSchemaVersionAction"
          value="fail" />
```

In der Property `schemaVersion` wird die Versionsnummer des Datenbankschemas angegeben, das die Anwendung erwartet. Wird die Property `schemaVersion` oder der Wert für die Version nicht angegeben, so findet keine Überprüfung der Schemaversion statt.

In der Property `schema` wird der Schemaname des Datenbankschemas angegeben, das die Anwendung erwartet. Wird die Property nicht angegeben, so werden die Überprüfungsstatements ohne Angabe eines Schemas ausgeführt.

In der Property `invalidSchemaVersionAction` wird festgelegt, wie die Data-Source auf eine falsche Schema-Version reagieren soll. Der Wert `fail` legt fest, dass eine Exception geworfen wird. Das hat zur Folge, dass die Anwendung nicht gestartet werden kann. Der Wert `warn` legt fest, dass lediglich eine Warnung in die Log-Datei geschrieben wird.

Der Wert für Property `invalidSchemaVersionAction` soll über die Properties gesetzt werden können, damit er zur Laufzeit vom Betrieb verändert werden kann. Hierzu sind in der Datei default.properties die folgenden Zeilen hinzuzufügen:

```
# ------------------------------------------------------------
# Persistenz
# ------------------------------------------------------------
# Reaktion der Data-Source auf eine falsche Schema-Version.
# fail: es wird eine Exception geworfen und die Anwendung
#       kann nicht gestartet werden.
# warn: es wird lediglich eine Warnung in die
#       Log-Datei geschrieben.
persistenz.invalidSchemaVersionAction = fail
```

Die zugehörige Zeile in der Spring-Konfiguration sieht damit wie folgt aus:

```
<property name="invalidSchemaVersionAction"
          value="${persistenz.invalidSchemaVersionAction}" />
```

Bei Datenmigrationen muss jedes Skript vor der Ausführung prüfen, ob die Datenbank die erwartete Version hat. Das kann mit dem folgenden SQL-Statement durchgeführt werden:

```
SELECT version
  FROM m_schema_version
 WHERE version_nummer = '<schemaVersion>'
   AND status = 'gueltig';
```

Liefert dieses SQL-Statement einen Wert zurück, dann ist die erwartete Schemaversion vorhanden. Liefert es keinen Wert zurück, dann ist die erwartete Schemaversion nicht vorhanden.

11. Quellenverzeichnis

Batch-Verarbeitung mit Hibernate. (kein Datum). Abgerufen am 10. 12 2014 von http://www.hibernate.org/hib_docs/v3/reference/en/html/batch.html

Bauer, C., & King, G. (2007). *Java Persistence with Hibernate.* Manning Publications.

Beispielimplementierung „Vorlage-Anwendung“. (kein Datum). Von Wird auf Anfrage bereitgestellt abgerufen

Demelt, A. (kein Datum). *Zeitmaschine – Temporale Datenhaltung.* Von http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2003/05/demelt_JS_05_03.pdf abgerufen

Detailkonzept der Komponente Anwendungskern. (kein Datum). Von 10_Blaupausen\technische_Architektur\Detailkonzept_Komponente_Anwendungskern.pdf abgerufen

Hiberate Documentation: Chapter 6. Collection Mapping. (kein Datum). Abgerufen am 10. 12 2014 von http://www.hibernate.org/hib_docs/v3/reference/en/html/collections.html

Informationstechnik, B. f. (kein Datum). *IT-Grundschutzkatalog, Baustein B 1.12, Stand 2006.* Von https://www.bsi.bund.de/DE/Themen/ITGrundschutz/itgrundschutz_node.html abgerufen

IsyFact – Referenzarchitektur. (kein Datum). Von 00_Allgemein\IsyFact-Referenzarchitektur.pdf abgerufen

IsyFact – Referenzarchitektur für IT-Systeme. (kein Datum). Von 00_Allgemein\IsyFact-Referenzarchitektur-IT-System.pdf abgerufen

Konzept Regelwerk. (kein Datum). Von 20_Bausteine\Regelwerk\Konzept_Regelwerk.pdf abgerufen

Konzept Spooling. (kein Datum). Von 20_Bausteine\Spooling\Konzept_Spooling.pdf abgerufen

Universal Connection Pool for JDBC Developer's Guide. (n.d.). Retrieved 08 13, 2015, from Optimizing Universal Connection Pool Behavior: https://docs.oracle.com/cd/E11882_01/java.112/e12265/optimize.htm

12. Abbildungsverzeichnis