Orientation in Objects

Auditierung mit JPA und Hibernate

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Falk Sippach
Orientation in Objects GmbH

Veröffentlicht am: 20.11.2009

EINLEITUNG

Viele Unternehmen müssen aufgrund gesetzlicher Vorgaben oder für interne Überprüfungen und Auswertungen jederzeit alle Modifizierungen an ihrer Datenbasis nachvollziehen können. Diese Verfahren werden als Auditierung bzw. im Falle der Vollprotokollierung als Historisierung bezeichnet. Für jeden vergangenen Zeitpunkt ermöglichen sie die Erstellung von Änderungsprotokollen oder gar die Wiederherstellung früherer Stände. Sowohl der Java Persistenz Standard (JPA) als auch das Persistenz-Framework Hibernate bieten verschiedene Möglichkeiten auf interne Ereignisse zu reagieren, um zum Beispiel bei schreibenden Zugriffen den Status von Entitäten zu manipulieren bzw. zu protokollieren oder beim Laden die Daten zu filtern. An mehreren Beispielen sollen in diesem Tutorial die Interceptoren und das Eventsystem von Hibernate sowie die äquivalenten Möglichkeiten in JPA aufgezeigt werden.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

GRUNDBEGRIFFE - AUDITIERUNG UND HISTORISIERUNG

Bei der Auditierung handelt es sich um Untersuchungsverfahren, die Prozesse hinsichtlich der Erfüllung von Anforderungen und Richtlinien bewerten. Die Audits (von lat. "Anhörung") werden dabei in den verschiedensten Bereichen von Unternehmen durchgeführt, so zum Beispiel beim Qualitätsmanagement, bei Produktionsabläufen, beim Datenschutz, beim Kundenmanagement usw. Bei einem Audit wird der Ist-Zustand anaylisiert und die ursprüngliche Zielsetzung mit dem tatsächlich Erreichten verglichen. Ziel ist es, Probleme in den Prozeßabläufen festzustellen, um anschließend Verbesserungsvorschläge zu erarbeiten.

Bei der Implementierung von Unternehmensoftware, der Zugriffsschicht auf die firmenweite, meist zentrale Datenbasis, bietet es sich an, die Grundlagen für Auditierungsprozesse zu legen. Die Daten werden heute immer noch hauptsächlich in relationale Datenbank Management Systeme gespeichert. Für gewöhnlich erfolgt mittels OR-Mapping Technologien die Anbindung an die objektorientierte Welt der Applikationsentwicklung. Sehr häufig kommt dabei Hibernate oder seit einigen Jahren eine der JPA-Implementierungen (Java Persistence API) zum Einsatz. Zu den bekanntesten JPA Providern zählen EclipseLink, OpenJPA und Hibernate.

Bei der Speicherung von Objekten in relationalen Datenbanken mittels OR-Mapping Tools kommt es ständig zu wiederkehrenden Programmieraufgaben. Es handelt sich dabei um die sogenannten Querschnittsbelange, also Codeschnipsel, die mit der eigentlichen fachlichen Programmlogik nichts zu tun haben, sich aber anwendungsübergreifend als Duplikationen durch die gesamte Codebasis ziehen. Solche Programmteile versucht man mit Hilfe der aspektorientierten Programmierung (AOP) zu zentralisieren und in eigene Module auszulagern, um sie dann über diverse Mechanismen an ihre eigentlichen Aufruf-Orte einzuweben. Der Vorteil dieser Bestrebungen ist, daß der fachliche Code schlanker, lesbarer und damit wartbarer wird. Dem Prinzip der Kapselung folgend, muß bei Änderungen an den Querschnittsbelangen statt mehrerer Stellen nur noch eine angepaßt werden.

Typische Querschnittsbelange in der Software-Entwicklung sind das Steuern von Transaktionen, das Tracing bzw. Monitoring, die Behandlung von Fehlern und das Überprüfen von Berechtigungen. Bei der Persistierung von Daten kommen im speziellen noch das Schreiben von Auditierungs-Logs, die Vollprotokollierung für eine vollständige Historisierung, die Validierung und das Filtern von Daten hinzu.

Um die die Persistenz betreffenden Querschnittsbelange vom eigentlichen Code zu trennen, muß man nicht gleich schwere Geschütze wie AspectJ oder Spring AOP auffahren. Vielmehr bringen OR-Mapper auf die Persistierung von Objekten spezialisierte Lösungen mit. In diesem Artikel sollen einige Möglichkeiten anhand von Code-Beispielen vorgestellt werden.

AUDIT LOG UND AUDIT TRAIL

Der Audit Log ist die einfachste Form, um die den Datensatz betreffenden zeitlichen Informationen zu speichern. Die Idee ist, daß beim Zugriff auf die Daten (Schreiben aber theoretisch auch Lesen) zusätzliche Informationen über das Was, das Wer und das Wann in die Datenbank geschrieben werden. Natürlich gibt es verschiedene Ausprägungen. Im einfachsten Fall enthalten die Datensätze selbst zusätzliche Spalten, um die Versionsnummer, das Erzeugungs- und Änderungsdatum bzw. den initialen und den letzten Bearbeiter abzuspeichern. Der Nachteil ist, daß nur die initiale Erzeugung und die letzte Bearbeitung für die spätere Auswertung zur Verfügung stehen. Alternativ dazu können die Aktivität, der Zeitstempel und der Bearbeiter aber auch in einer separaten Tabelle gespeichert werden, so daß man im Nachhinein jederzeit ein Bearbeitungsprotokoll erstellen kann. Dadurch ist es einfach nachvollziehbar, wer wann welche Änderungen an den Daten vorgenommen hat.

Solche Informationen könnten wiederum die Grundlage für die Auswertung von Berechtigungen für weiterführende Aktionen sein. Ein solcher Fall wäre das 4- bzw. 6-Augen-Prinzip. Dies besagt, daß nach einem Bearbeitungsschritt nur ein anderer Benutzer (ein zweites bzw. drittes Paar Augen) den nächsten bzw. die nächsten Vorgänge ausführen darf. Dadurch können firmeninterne Prüfungen von Prozessen durch Vorgesetzte abgebildet werden.

Berechtigungsprüfungen können natürlich auch anhand des fachlichen Inhalts der Daten durchgeführt werden. Dazu zählt das Filtern beim Laden bzw. das Verbieten von schreibenden Änderungen. Ein Anwender muß die entsprechenden Berechtigungen besitzen, um beispielsweise Daten aus einem bestimmten Bereich (Land, Postleitzahlenbereich, ...) einsehen und ändern zu dürfen. Genauso könnten gewisse Teil-Informationen "geschwärzt" angezeigt werden.

Die einfachen Auditierungsdaten können desweiteren für die Auswertung von Protokollen herangezogen werden, um zum Beispiel eine Fehlbedienung und deren Bearbeiter ausfindig zu machen. Oder man ermittelt regelmäßig die Datensätze, die seit längerer Zeit nicht mehr bearbeitet wurden. Diese "Karteileichen" könnten dann aufgeräumt oder im Falle von Kundenkontakten durch neue Werbe-Aktionen wieder aufgefrischt werden.

Einen Schritt weiter als die einfachen Audit Logs gehen die sogenannten Audit Trails, die bei jedem Schreibvorgang eine Kopie des Datensatzes mit Zeitstempel und Bearbeiter anlegen. Damit kann man eine komplette Historie erzeugen und wäre so in der Lage zu jedem Objekt den Zustand zu einem bestimmten Zeitpunkt zu ermitteln und ggf. wiederherzustellen. Möchte man allerdings einzelne Änderungen rückgängig machen können (Undo/Redo), bietet sich als Alternative das sogenannte Change Log an. Hier wird nicht der komplette Zustand vor und nach der Änderung, sondern vielmehr die Änderung selbst in die Datenbank geschrieben (neuer und alter Wert). Das spart zum einen natürlich Speicherplatz, außerdem lassen sich dadurch sehr einfach statistische Auswertungen über den Datenbestand ermitteln, wie zum Beispiel Fragen: 'Wie oft ändern Kunden Ihre Rechnungsanschrift?'. Nichtsdestotrotz erhöhen Vollprotokollierungen die Last bei der Ausführung und auch mehr oder weniger den Speicherverbrauch. Man muß je nach Anwendungsfall entscheiden, ob für die spätere Auswertung der Historisierungsdaten die Audit Trails oder die Change Logs besser geeignet sind, beide bringen Vor- und Nachteile mit sich.

VORSCHLÄGE ZUR TECHNISCHEN UMSETZUNG

Auditierungsprotokolle können im einfachsten Fall direkt im RDBMS mit Datenbank-Triggern schnell und effizient umgesetzt werden. Etablierte Datenbank-Systeme bieten außerdem meist eigene, sehr mächtige aber teilweise auch komplexe Lösungen für das Schreiben von Auditierungs-Logs an. Möchte man nun allerdings datenbankunabhängig entwickeln, muß man die Protokollierung selbst organisieren und im Falle einer Historisierung alle zu erfassenden Operationen in eine separate Datenbank-Tabelle speichern. Da das Schreiben von Protokollen für gewöhnlich nach dem gleichen Schema abläuft, bietet sich hier die Zentralisierung der Audit-Schritte mittels der JPA Entity Listener, der Hibernate Interceptoren oder des Hibernate Event Systems an. Alternativ würde man diesen immer wiederkehrenden Code zur Erzeugung der Audit Logs bzw. Trails in der gesamten Applikation verstreuen, was spätestens während der Wartungsphase der Anwendung zum Alptraum wird.

JPA ENTITY LISTENERS

Die JPA Entity Listener erlauben über Callback-Methoden das Abfangen von Ereignissen (Laden, Speichern, ...). Diese Lebenszyklus-Methoden können entweder direkt in der Entität oder in einer eigenen Klasse implementiert werden, die dann per Annotationen bzw. per XML-Konfiguration mit den entsprechenden Entitäten verknüpft wird. Zu empfehlen ist dabei eher die zweite Variante, weil das Definieren von Callback-Methoden innerhalb der Entitäten das Domänenmodell nur unnötig mit den Querschnittsbelangen aufbläht. Und spätestens wenn das Verhalten für mehrere Entitäten benötigt wird, kann man eine separate Entity Listener Klasse leicht wiederverwenden. Die Callback-Methoden unterliegen dabei keinen Namenskonventionen, sie müssen allerdings `void` zurückgeben und dürfen keine Checked Exceptions werfen. Die Sichtbarkeit (`private`, `public`, ...) der Methoden ist egal. Die Entity Listener Klasse selbst ist ein POJO (Plain Old Java Object), welches außer einem argumentlosen Konstruktor keinerlei Bedingungen erfüllen, also weder von einer bestimmten Klasse ableiten, noch bestimmte Interfaces implementieren muß.

Damit eine Listener-Methode als Callback erkannt wird und auf bestimmte Ereignisse reagiert, muß sie mit vordefinierten Annotationen markiert werden. Folgende stehen zur Auswahl:

Annotation	Beschreibung
@PostLoad	Wird aufgerufen, nachdem eine Instanz mit `find()`, `getReference()` oder über eine Abfrage geladen wurde bzw. nach dem Aufruf von `refresh()`.
@PrePersist @PostPersist	Wird aufgerufen bevor der EntityManager die Entität speichert bzw. nachdem der Datenbank-Eintrag erfolgt ist.
@PreUpdate @PostUpdate	Wird vor bzw. nach dem Flush aufgerufen, also bevor bzw. nachdem der Persistenzkontext mit der Datenbank synchronisiert wurde. Wird aber nur aufgerufen, wenn sich das Objekt wirklich geändert hat (Dirty Checking).
@PreRemove @PostRemove	Wird aufgerufen, bevor der EntityManager das Objekt löscht bzw. nachdem es aus der Datenbank entfernt wurde.

Tabelle 1: JPA Callback Annotationen

Dabei kann eine Methode auf mehrere Callback-Ereignisse reagieren, es darf aber nicht mehrere Methoden mit der gleichen Callback-Annotation geben. Zu beachten ist auch, daß innerhalb einer Callback-Methode geworfene RuntimeExceptions eine laufende JTA-Transaktion automatisch zurückrollen.

```
// erlaubt
@PrePersist
@PreUpdate
private void logSave() {
  [..]
}
// NICHT erlaubt: @PreUpdate an zweiter Methode
@PreUpdate
private void logUpdateOnly() {
  [..]
}
```

Beispiel 1: Mehrere Annotationen pro Methode sind erlaubt, NICHT aber zwei Methoden mit der gleichen Annotation!

Im folgenden Beispiel soll eine Log-Nachricht ausgegeben werden, bevor eine Person gespeichert wird. In diesem Fall ist die Listener-Methode direkt in der persistenten Klasse implementiert und wird durch `@PrePersist` aktiviert.

```
@Entity
public class Person implements Auditable<Long> {
  [..]
  @PrePersist
  private void logSaving() {
    System.out.println(this + " will be persisted
    immediately!");
  }
}
```

Beispiel 2: Callback-Methode direkt in der Entität

Im nächsten Beispiel werden das Erzeugungs- bzw. Änderungsdatum und der initiale bzw. letzte Bearbeiter per Listener gespeichert. Diesmal wurde eine separate Klasse für die Listener-Methoden erzeugt, die dann per `@EntityListeners(AuditListener.class)` für bestimmte Entitäten aktiviert werden kann.

```
public class AuditListener {
  @PrePersist
  public void setCreationDate(Object entity) {
    if (entity instanceof AbstractEntity) {
      AbstractEntity<?> myEntity = (AbstractEntity<?>) entity;
      myEntity.setCreatedAt(new Date());
    }
  }
  @PreUpdate
  public void setChangeDate(Object entity) {
    if (entity instanceof AbstractEntity) {
      AbstractEntity<?> myEntity = (AbstractEntity<?>) entity;
      myEntity.setLastModifiedAt(new Date());
    }
  }
}
@Entity
@EntityListeners(AuditListener.class)
public class Person extends AbstractEntity<Long> {
  [..]
}
```

Beispiel 3: Separate Entity Listener Klasse

Der `AuditListener` kann bei anderen Entitäten wiederverwendet werden. Es ist auch möglich, ihn für eine komplette Vererbungshierarchie von Domain-Klassen zu aktvieren, es muß nur die Superklasse annotiert werden. Desweiteren kann man in der persistence.xml (globale JPA-Konfigurationsdatei) auch Default-Listener registrieren, die automatisch für alle Entitäten verwendet werden. Mit `@ExcludeSuperclassListeners` und `@ExcludeDefaultListeners` kann man diese "globaleren" Einstellungen für einzelne Domain-Klassen aber wiederum deaktivieren.

INTERCEPTOREN

Mit den Interceptoren gibt es auch in Hibernate die Möglichkeit, Querschnittsbelange mittels Callback-Methoden umzusetzen. Die Hibernate-Session ruft während ihrer Datenbank-Transaktionen bei bestimmten Ereignissen diese sogenannten Hook-Methoden auf, die man überschreiben und somit auf die Ereignisse reagieren kann. In den ersten Versionen ließen sich die Hibernate-Macher stark von anderen ORM-Lösungen beeinflußen und stellten die beiden Interfaces `Lifecycle` und `Validatable` zur Verfügung. Dadurch konnten persistente Objekte direkt auf Ereignisse reagieren, die ihren eigenen Persistenz-Lebenszyklus betrafen. Allerdings sind sie mittlerweile aus gutem Grund als veraltet markiert, da die Entitäten-Klassen von Hibernate abhängig werden und somit nicht mehr portabel sind.

Den besseren Ansatz stellen die Interceptoren dar. Laut API-Doc erlauben sie den Zustand von Entitäten zu untersuchen und zu ändern, und zwar bevor Änderungen an die Datenbank gesendet bzw. nach dem Daten aus der Datenbank gelesen wurden. Auf die folgenden Ereignisse kann reagiert werden:

- Laden - `onLoad(..)`
- Speichern - `onSave(..)`
- Updaten - `onFlushDirty(..)`
- Löschen - `onDelete(..)`

Desweiteren hat man die Möglichkeit in den Lebenszyklus der Session einzugreifen:

- Vor dem Session Flush - `preFlush(..)`
- Nach dem Session Flush - `postFlush(..)`
- Nach dem Transaktionsbeginn - `afterTransactionBegin(..)`
- Vor dem Abschließen der Transaktion - `beforeTransactionCompletion(..)`
- Nach dem Abschließen der Transaktion - `afterTransactionCompletion(..)`

Weitere Ereignisse, auf die ein Interceptor reagieren kann, sind der Lebenszyklus von Collections und das Ermitteln des Transient/Detached/Dirty-Status einer Entität. Außerdem kann man in die Instanziierung von persistenten Objekten eingreifen, sofern sie vom Standard-Verhalten abweichen soll. Und zuletzt bietet sich noch die Möglichkeit, ein erzeugtes Prepared Statement zu ändern, bevor die Abfrage an die Datenbank gesendet wird.

Um einen eigenen Interceptor zu schreiben, muß man das Interface `org.hibernate.Interceptor` implementieren oder von der Klasse `org.hibernate.EmptyInterceptor` ableiten. Letztere hat den Vorteil, daß alle Interceptor-Methoden bereits leer implementiert sind und man nur die Benötigten überschreiben muß. Grundsätzlich hat man sowohl bei lesenden als auch schreibenden Zugriffen die Möglichkeit, die bereits geladenen oder zu speichernden Entitäten zu ändern oder auch komplett neue Datensätze zu speichern bzw. zurückzugeben. Wichtig ist aber, daß man in einer Callback-Methode für zusätzliche Schreibvorgänge anderer Datensätze (z. B. das Speichern der Audit Logs) nie die gleiche Session wiederverwenden darf, die den Interceptor-Aufruf ausgelöst hat. Weiterhin muß man sich sehr genau die JavaDoc-Beschreibung der Interceptor-Methode anschauen. Dort ist unter anderem beschrieben, auf welche Art und Weise man den Zustand der auslösenden Entität verändern darf. Die Methode `onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)` erhält die zu speichernde Entität mehrfach als Parameter übergeben (in verschiedenen Formen). Man darf aber nur das Object-Array `state` verändern, da dort die Liste der zu schreibenden Attribute enthalten ist, die nach Verlassen der Interceptor-Methode direkt für das SQL INSERT Statement verwendet wird. Änderungen an der ebenfalls als Parameter übergebenen Entität werden dahingehend ignoriert. Im Gegensatz zu den JPA Entity Listenern wird das Ändern der Entität im Interceptor dadurch leider verkompliziert, wie auch das nächste Beispiel zeigt.

Um einen Interceptor zu aktivieren muß er Hibernate bekannt gemacht werden. Dafür gibt es zwei Möglichkeiten. Entweder man übergibt beim Öffnen einer neuen Session (`SessionFactory.openSession(Interceptor interceptor)`) eine neue Interceptor-Instanz. Der Vorteil dieser Variante ist, daß jede Session ihren eigenen Interceptor erhält, der in diesem Fall nicht threadsafe implementiert sein muß. In der zweiten Variante wird pro Anwendung nur eine Instanz des Interceptors an der SessionFactory registriert, die dann bei jedem Öffnen einer neuen Session automatisch zugewiesen wird. Hier muß bei der Implementieurng des Interceptors unbedingt auf Threadsicherheit geachtet werden, da mehrere Session-Instanzen konkurrierend auf die eine Instanz zugreifen können.

```
Session session = sf.openSession(new AuditInterceptor());
```

Beispiel 4: Pro Session eine eigene Interceptor-Instanz

```
new Configuration().setInterceptor(new AuditInterceptor());
```

Beispiel 5: Globaler Interceptor für alle Sessions

Alternativ könnte man auch seinen eigenen `CurrentSessionContext` implementieren und sich dort um das automatische Öffnen neuer Sessions mit jeweils einem neu instanziierten Interceptor kümmern. Dann würde sich der Aufruf von `SessionFactory.getCurrentSession()` genauso wie `SessionFactory.openSession(Interceptor interceptor)` verhalten.

Das folgende Beispiel zeigt einen einfachen TracingInterceptor, der sowohl bei lesenden als auch schreibenden DB-Zugriffen einen Log-Eintrag erzeugt. Da dieser Interceptor weder Änderungen an den Entitäten vornimmt, noch irgendwelche zusätzlichen Daten in die DB schreibt, muß an dieser Stelle keine der eben angesprochenen Problematiken beachtet werden. Der Rückgabewert der `on*()` -Methoden teilt der Hibernate Session mit, ob das Objekt im Interceptor geändert wurde. In diesem Fall wird jeweils `false` zurückgegeben. Die `onDelete()` -Methode hat gar keinen Rückgabewert, da das Ändern eines zu löschenden Objektes keinen Sinn machen würde.

```
public class TracingInterceptor extends EmptyInterceptor {
  private static final Logger LOG =
     LoggerFactory.getLogger(TracingInterceptor.class);
  private static final long serialVersionUID = 1L;
  @Override
  public boolean onSave(Object entity, Serializable id,
  Object[] state,
     String[] propertyNames, Type[] types) {
   LOG.debug(String.format("Objekt %s wird gespeichert!",
   entity));
   return false;
  }
  @Override
  public boolean onFlushDirty(Object entity, Serializable
  id,
     Object[] currentState, Object[] previousState,
     String[] propertyNames, Type[] types) {
   LOG.debug(String.format("Objekt %s wird geupdated!",
   entity))
   return false;
  }
  @Override
  public boolean onLoad(Object entity, Serializable id,
  Object[] state,
     String[] propertyNames, Type[] types) {
   LOG.debug(String.format("Objekt %s wird geladen!",
   entity));
   return false;
  }
  @Override
  public void onDelete(Object entity, Serializable id,
  Object[] state,
     String[] propertyNames, Type[] types) {
   LOG.debug(String.format("Objekt %s wird gelöscht!",
   entity));
  }
}
```

Beispiel 6: Beispiel für einen Tracing Interceptor

Das nächste Beispiel verändert dagegen die dem Interceptor übergebenen Objekte. Und zwar sollen beim Datenbank-Insert von Subklassen der Klasse `AbstractEntity` automatisch das Erzeugungsdatum und der initiale Bearbeiter erzeugt und gespeichert werden. Das Datum wird mit dem aktuellen Zeitstempel initialisiert, die ID des aktuellen Benutzers wird über eine Referenz auf einen global verfügbaren Authorisierungskontext (`authContext`) gelesen.

```
public class AuditInterceptor extends EmptyInterceptor {
  private static final long serialVersionUID = 1L;
  private AuthContext authContext = ...
  @Override
  public boolean onSave(Object entity, Serializable id,
  Object[] state,
     String[] propertyNames, Type[] types) {
   if (entity instanceof AbstractEntity) {
    for (int i = 0; i < propertyNames.length; i++) {
     if ("createdAt".equals(propertyNames[i])) {
      state[i] = new Date();
     }
     if ("createdBy".equals(propertyNames[i])) {
      state[i] = authContext.getUserId();
     }
     return true;
    }
   }
   return false;
  }
}
```

Beispiel 7: Beispiel für einen Interceptor

Man sieht in diesem Beispiel sehr schön, daß es relativ umständlich ist, Werte der betroffenen Entität zu ändern. Wie weiter oben beschrieben, darf nicht der Parameter `entity` geändert werden. Vielmehr muß man das Object-Array `state` anpassen. Dafür muß der Namen des Attributs bekannt sein, um darüber den Index des zu schreibenden Wertes zu ermitteln. Dieser unschöne, aber vom Hibernate Interceptor erzwungene Programmierstil kann insbesondere beim nachträglichen Umbenennen von Attributen in der Entität zu schwer nachvollziehbaren Fehlern führen. Möchte man das letzte Änderungsdatum und den letzten Bearbeiter speichern, müßte zusätzlich die `onFlushDirty(..)` -Methode ähnlich `onSave(..)` für die Attribute `changedAt` und `changedBy` überschrieben werden. Mit dem Rückgabewert `true` muß man Hibernate an dieser Stelle mitteilen, daß das übergebene Objekt im Interceptor verändert wurde.

Im nachfolgenden Beispiel soll für jede Entität ein eigener Auditierungs-Log-Eintrag in einer separaten Tabelle geschrieben werden. Für diese zusätzliche Tabelle muß eine weitere Hibernate-Entität angelegt werden. Die Klasse `AuditLog` gehört dabei nicht zum Domain-Modell, deswegen sollte sie eher in einem Util-Package (z. B. neben den Usertypen) abgelegt werden. Es gibt einige Besonderheiten, die diese Klasse von den gewöhnlichen persistenten Klassen abgrenzt. Zum einen wird ein Audit Log-Datensatz nie geändert, deswegen ist er hier im Mapping als `mutable="false"` gekennzeichnet. Desweiteren wird keine ID in der Klasse benötigt, da die Audit Logs nie per Primärschlüssel geladen werden sollen. Hibernate wird trotzdem intern einen künstlichen PK verwalten. In diesem noch einfachen Beispiel wird nur der Typ der Änderung (INSERT, UPDATE, DELETE, ...), die Benutzer-ID, der Klassenname und die ID des bearbeiteten Datensatzes gespeichert. Benötigt man eine Vollprotokollierung, müßte man zusätzlich die Differenz zwischen dem aktuellen und dem letzten Zustand ermitteln und mit persistieren.

```
public class AuditLog {
  private final String type;
  private final Long entityId;
  private final Class<?> clazz;
  private final Long userId;
  public AuditLog(String type, Long entityId, Class<?> clazz,
  Long userId) {
    this.type = type;
    this.entityId = entityId;
    this.clazz = clazz;
    this.userId = userId;
  }
  // getter ...
}
<hibernate-mapping default-access="field"
package="de.oio.entity">
  <class name="AuditLog" table="AUDIT_LOG" mutable="false">
    <id column="AUDIT_LOG_ID" type="long">
      <generator class="increment" />
    </id>
  <property name="type" column="TYPE" />
    <property name="entityId" column="ENTITY_ID" />
    <property name="clazz" column="ENTITY_CLASS" type="class"
    />
    <property name="userId" column="USER_ID" />
  </class>
</hibernate-mapping>
```

Beispiel 8: Beispiel für eine Audit Log Entität

Bei der Implementierung des Interceptors gilt es mehrere Klippen zu umschiffen. Zum einen muß man sich alle zu einer Transaktion gehörenden Daten merken, da Hibernate aus Performance-Gründen für gewöhnlich mehrere Operationen sammelt. Erst am Ende der Transaktion wird der Datenbank-Flush ausgeführt. Bestimmte Interceptor-Methoden wie `onSave(..)` werden aber sehr viel früher aufgerufen, so daß Informationen, wie zum Beispiel die Zuweisung einer von der Datenbank generierten ID, noch nicht gesetzt wurden. Erst nach dem Flush der Session-Daten garantiert Hibernate, daß alle Daten synchronisiert sind. Deshalb werden in diesem Beispiel alle zu auditierenden Datensätze in der `onSave()` Methode in einer Liste gesammelt. Diese Liste ist wird von einer ThreadLocal-Variablen gehalten, die garantiert, daß jeder Thread seine eigene Listen-Instanz zur Verfügung hat und sich mehrere Threads nicht gegenseitig stören. Zur Erinnerung, Threadsicherheit wird benötigt, wenn der Interceptor global an der SessionFactory angemeldet und dadurch für alle Sessions wiederverwendet wird. Erst in der `postFlush(..)` Methode, also direkt nachdem Hibernate alle angesammelten Änderungen zur Datenbank gesendet hat, werden dann die Audit Log-Einträge erzeugt.

Hier muß die zweite Klippe umschifft werden. Die aktuelle Session, aus welcher der Interceptor aufgerufen wurde, darf nicht für das Speichern der erzeugten Audit Log Datensätze wiederverwendet werden. Sie befindet sich während eines Interceptor-Aufrufs in einem fragilen Zustand. Man kann nicht ein neues Objekt speichern, während man sich noch im Speichervorgang eines anderen Objekts befindet. Deshalb muß an dieser Stelle entweder direkt mit JDBC gearbeitet oder eine neue Session geöffnet werden. Man kann aber die JDBC-Connection der anderen Session und damit eine bereits laufende Transaktion wiederverwenden. Das geschieht hier über die statische Utility Methode `HibernateUtil.getNewSession(...)`. Mit dem Interface `Auditable` werden die zu auditierenden Entitäten markiert. Es bietet in diesem Beispiel eine `getId()` Methode, über die der Primärschlüssel für den Audit Log Datensatz abgefragt wird. Genausogut sind hier weitere Methoden vorstellbar, um sich zum Beispiel die Unterschiede zur letzten Version direkt von der Entität ermitteln zu lassen.

```
public interface Auditable<T> {
  T getId();
}
public class AuditLogInterceptor extends EmptyInterceptor {
  private Session session;
  private Long userId;
  private static final long serialVersionUID = 1L;
  private static ThreadLocal<Set<Auditable<Long>>> insertsTL
  =
       new ThreadLocal<Set<Auditable<Long>>>() {
    @Override
    protected Set<Auditable<Long>> initialValue() {
      return new HashSet<Auditable<Long>>();
    }
  };
  @Override
  public boolean onSave(Object entity, Serializable id,
  Object[] state,
     String[] propertyNames, Type[] types) {
    if (entity instanceof Auditable) {
      insertsTL.get().add((Auditable<Long>) entity);
    }
    return false;
  }
  @Override
  public void postFlush(Iterator entities) {
    for (Auditable<Long> entity : insertsTL.get()) {
      Session tempSession =
        HibernateUtil.getNewSession(HibernateUtil.getSession().connection());
      try {
        AuditLog auditLog = new AuditLog("insert",
        entity.getId(),
          entity.getClass(), getUserId());
        tmpSession.save(auditLog);
        tmpSession.flush();
      } finally {
        tmpSession.close();
      }
    }
  }
}
```

Beispiel 9: Beispiel für einen Audit Log-Interceptor

Zum Abschluß folgt noch ein einfacher Interceptor, der Lösch-Berechtigungen überprüft und ggf. eine `AuthorizationException` wirft. Das ganze ist natürlich auch wieder auf Insert und Update ausdehnbar. Beim Laden von Entitäten hat man wiederum zwei Möglichkeiten, Daten zu filtern. Entweder wird ein Datensatz aufgrund fehlender Berechtigungen gar nicht zurückgegeben oder man könnte einzelne Attribute durch eigene Werte ersetzen, um zum Beispiel Paßwörter vor nicht authorisiertem Zugriff zu schützen.

```
public class AuthInterceptor extends EmptyInterceptor {
  private AuthContext authContext = ...
  @Override
  public void onDelete(Object entity, Serializable id,
  Object[] state,
     String[] propertyNames, Type[] types) {
    if (Role.ADMIN != authContext.getRole()) {
      throw new AuthorizationException("Benutzer " +
      authContext.getUser()
         + " darf nicht löschen!");
    }
  }
}
```

Beispiel 10: Beispiel für einen Security Interceptor

In Hibernate kann immer nur ein Interceptor pro Session aktiv sein. Will man in einer Anwendung aber verschiedene orthogonale Aspekte über Interceptoren abhandeln, muß man mehrere Anliegen in einer Klasse implementieren. Das widerspricht dem Prinzip Separation of Concerns. Dieser eine große Interceptor wäre schwer zu lesen und vor allem zu warten. Als Alternative könnte man alle Belange in jeweils eigenen Interceptor-Klassen implementieren und über einen sogenannten ChainedInterceptor in einer gewissen Reihenfolge verketten. Eine Beispiel-Implementierung findet sich unter [10]. Problematisch wird eine solche Implementierung allerdings, wenn die Callback-Methoden nicht void (wie bei `onDelete()` und `post/preFlush()`) oder boolean (wie bei `onSave` oder `onLoad()`) zurückgeben. Dann müssen die verschiedenen Rückgabewerte aggregiert werden. Die einfachste Lösung wäre, daß der erste Interceptor, der einen sinnvollen Wert (ungleich `null`) zurückliefert, gewinnt.

HIBERNATE'S EVENT SYSTEM

Beim Sprung von 2.x auf 3.x wurde ein komplettes Redesign des Hibernate-Kerns durchgeführt. Der neue Kern basiert auf einem Modell von Ereignissen und Listenern (Observer Pattern). Wenn Hibernate z. B. ein Objekt löschen möchte, dann wird ein Löschereignis angestossen. Wer auch immer auf dieses Ereignis horcht, kann es fangen und sich um das Löschen einer Datenbank-Entität kümmern. Standardmäßig verarbeitet der `DefaultDeleteEventListener`, eine Implementierung des Interface `DeleteEventListener` das DeleteEvent.

Es besteht aber die Möglichkeit, eigene Listener anzumelden, die entweder die DefaultListener ersetzen oder Funktionalität zum Standardverhalten hinzufügen. Wirklich ersetzen wird man die DefaultListener vermutlich eher selten, dann muß der eigene Listener nämlich auch die entsprechende Funktionalität abdecken, die sonst von Hibernate automatisch übernommen wird. Ein benutzerdefinierter Listener muß das entsprechende Interface implementieren oder eine der Convenience Basis-Klassen (zum Beispiel `DefaultDeleteEventListener`) überschreiben. Im folgenden Beispiel wird ein DeleteListener erzeugt, der vor dem Löschen sicherstellt, daß die betroffene Entität in der Datenbank überhaupt entfernt werden darf.

```
public class SecurityDeleteListener extends
DefaultDeleteEventListener {
  @Override
  public void onDelete(DeleteEvent event)
     throws HibernateException {
    if (authContext.isAuthorized(event.getObject())) {
      throw new AuthorizationException("must not delete entity
      "
  + event.getEntityName());
    }
 super.onDelete(event);
  }
}
```

Beispiel 11: SecurityDeleteListener

Der Listener prüft dazu in der `isAuthorized()` Methode, ob der Benutzer die nötigen Rechte für das Löschen dieser Entität besitzt. Wenn er nicht löschen darf, wird eine `AuthorizationException` geworfen, ansonsten wird die Ausführung an den DefaultDeleteEventListener weitergegeben.

Listener werden von mehreren Requests gemeinsam genutzt. Sie sollten daher als Singletons betrachtet werden und keine Zustände als Instanzvariablen halten, welche Transaktionen betreffen. Die Registrierung erfolgt entweder programmatisch am `Configuration` -Objekt oder in der Hibernate-XML-Konfiguration (die Registrierung in *.properties-Dateien wird nicht unterstützt). Wichtig ist, daß man mit der Registrierung eines benutzerdefinierten Listeners die Default-Listener automatisch deaktiviert. Falls man dies nicht möchte, kann man sie aber zusätzlich hinzufügen. Die Registrierung erfolgt dabei in der gleichen Reihenfolge wie in der Konfiguration angegeben. Deshalb sollte der Default-Listener immer an der ersten Stelle stehen, damit auf jeden Fall zuerst das Standardverhalten von Hibernate ausgeführt wird. In diesem Fall ruft der `SecurityDeleteListener` per `super()` den `DefaultDeleteEventListener` auf, deshalb reicht es, nur diesen einen zu registrieren.

```
<session-factory>
  <event type="delete">
    <listener
    class="de.oio.util.listener.SecurityDeleteListener"/>
  </event>
  [..]
</session-factory>
```

Beispiel 12: Listener-Registrierung in XML-Konfiguration

Achtung, wenn man Listener in der XML-Konfiguration mehrfach registriert, dann werden die Instanzen nicht wiederverwendet. Vielmehr resultiert jede `<listener />` Referenz in einem separaten Objekt der Listener-Klasse. Will man dies umgehen und die Listener-Instanzen teilen, dann muß man sie programmatisch registrieren.

```
Configuration cfg = new Configuration();
DeleteEventListener[] deleteListeners = {new
SecurityDeleteListener()};
cfg.getEventListeners().setDeleteEventListeners(deleteListeners);
```

Beispiel 13: Programmatische Listener-Registrierung

JPA definiert kein Listener-Konzept. Bei Verwendung von Hibernate als Implementierung hat man aber ebenfalls die Möglichkeit, dem EntityManager benutzerdefinierte Listener zuzuweisen. Im folgenden Beispiel wird wiederum der `SecurityDeleteListener` registriert, der auf das Werfen eines `DeleteEvent` reagiert.

```
<persistence-unit name="...">
  <properties>
    <property name="hibernate.ejb.event.delete"
      value="de.oio.util.listener.SecurityDeleteListener" />
  </properties>
</persistence-unit>
```

Beispiel 14: Listener-Registrierung beim EntityManager

Bei dem Hibernate Event System handelt es sich um eine sehr mächtige Eingriffsmöglichkeit. Theoretisch könnte durch dieses System jede Kernfunktionalität von Hibernate ausgetauscht werden. Man wird aber eher selten eigene `EventListener` schreiben, da in den meisten Fällen die Interceptoren flexibel genug sein sollten. Ein Beispiel für benutzerdefinierte `EventListener` findet sich in dem Versionierungs-Framework `Envers` (ENtity VERSioning).

Envers ist eine Zusatzbibliothek zu Hibernate und wird ab Version 3.5 Teil des Core sein. Mit Envers soll die Historisierung von Daten für den Anwendungsentwickler möglichst transparent erfolgen. Daten können wie bisher mit Hibernate geschrieben und gelesen werden. Das Datenbank-Schema wird nicht geändert sondern nur um zusätzliche Tabellen für die zu historisierenden Daten ergänzt. Für die Umsetzung der Auditierung setzt Envers auf das Hibernate Event System, dementsprechend muß man die folgenden `EventListener` registrieren.

```
<property name="hibernate.ejb.event.post-insert"
   value="org.hibernate.envers.event.AuditEventListener" />
<property name="hibernate.ejb.event.post-update"
   value="org.hibernate.envers.event.AuditEventListener" />
<property name="hibernate.ejb.event.post-delete"
   value="org.hibernate.envers.event.AuditEventListener" />
<property name="hibernate.ejb.event.pre-collection-update"
   value="org.hibernate.envers.event.AuditEventListener" />
<property name="hibernate.ejb.event.pre-collection-remove"
   value="org.hibernate.envers.event.AuditEventListener" />
<property name="hibernate.ejb.event.post-collection-recreate"
   value="org.hibernate.envers.event.AuditEventListener" />
```

Beispiel 15: Für Envers notwendige EventListener-Einträge in der JPA-Konfigurationsdatei.

Die Code-Änderungen an den Entitäten-Klassen ist minimal, lediglich die Annotation `@Audited` wird entweder an der Klasse oder an den zu historisierenden Attributen benötigt. Für jede dieser mit `@Audited` markierten Klassen wird eine eigene Tabelle erzeugt, welche alle Änderungen an der Entität aufnimmt. Historierte Daten können wiederum sehr einfach abgefragt werden.

```
@Audited
@Entity
public class Person {
  @Id
  @GeneratedValue
  private Long id;
  private String name;
  // ...
}
```

Beispiel 16: Markierung einer auditierbaren Entität.

Unter [12] finden sich weitere Informationen zu Envers, unter anderem wie man Abfragen zu früheren Versionen der Entitäten erstellen kann.

ZUSAMMENFASSUNG

Hibernate und zu Teilen auch JPA bieten diverse Möglichkeiten an, um Querschnittsbelange im Persistenz-Code besser zu strukturieren und somit Duplikationen zu vermeiden. Zu diesen orthogonalen Aspekten zählen in erster Linie das Überprüfen von Berechtigungen, das Schreiben von Log- und Auditierungseinträgen, die Historisierung von Daten und das Aussortieren von nicht relevanten Datensätzen. Für die meisten Anwendungsfälle sollten die Interceptoren bzw. die JPA-Entity-Listener genug Möglichkeiten bieten. Der Ansatz von JPA ist einfacher (keine Interfaces, dafür Annotationen und einfache Änderung der auditierten Daten). Dafür sind die Hibernate Interceptoren flexibler einsetzbar, auch wenn die Entwicklung etwas holpriger und umständlicher ist.

Die Entity Listener sind im Gegensatz zu den Hibernate Interceptoren zustandslose Klassen, d. h. man kann keine Zustände über mehrere Callback-Aufrufe puffern. Es ist dementsprechend nicht wie im Hibernate `AuditInterceptor` Beispiel möglich, alle angefallenen Änderungen vor dem Flush zu sammeln und dann auf einmal die angelaufenen Auditierungsinformationen zu erzeugen und zu speichern. Hinzu kommt, daß der Entity Listener keinen EntityManager verwenden darf. Mit Hibernate als JPA-Implementierung könnte man zwar einen Trick anwenden und einen temporären zweiten Persistenzkontext zur Verfügung stellen, die bessere Lösung wäre aber, die Protokollierung der Audit-Informationen dann in einer höheren Schicht der Anwendung abzuhandeln. Im Falle von EJB3 wären dies die Session Beans. In Hibernate ist es dagegen problemlos möglich, einen zweiten Session-Kontext zu erzeugen.

Benötigt man mächtigere Mittel, steht das Event System in Hibernate zu Verfügung. Diesem bedient sich auch das Versionierungs-Framework Hibernate Envers, mit dessen Hilfe auf relativ einfache und vor allem transparente Weise Entitäten historisiert werden können. Bevor man also eine eigene Vollprotokollierungslösung schreibt, sollte man zuerst einmal einen Blick auf http://jboss.org/envers werfen.

REFERENZEN

- [1] Hibernate Homepage
  (http://hibernate.org)
- [2] Java Persistence with Hibernate
  King, Gavin ; Bauer, Christian
  (http://www.manning.com/bauer2/)
- [3] Dokumentation Hibernate Interceptoren
  (http://docs.jboss.org/hibernate/stable/core/reference/en-US/html/events.html#objectstate-interceptors)
- [4] Dokumentation Hibernate Event System
  (http://docs.jboss.org/hibernate/stable/core/reference/en-US/html/events.html#objectstate-events)
- [5] Dokumentation Filter
  (http://docs.jboss.org/hibernate/stable/core/reference/en-US/html/filters.html)
- [6] Audit Trail
  (http://www.webopedia.com/TERM/A/audit_trail.html)
- [7] Versionierte vs Historisierte Objekte
  (http://blog.schauderhaft.de/2008/09/14/versionierte-vs-historisierte-objekte)
- [8] Unbounded Result Set
  (http://in.relation.to/Bloggers/HandlingFailureWithHibernate)
- [9] Hibernate Wiki: Audit Logging
  (https://www.hibernate.org/318.html)
- [10] Hibernate Wiki: Chained Interceptor
  (https://www.hibernate.org/92.html)
- [11] Martin Fowler über Audit Log
  (http://www.martinfowler.com/ap2/auditLog.html)
- [12] Automatische Historisierung mit Hibernate Envers
  (http://it-republik.de/jaxenter/artikel/Automatische-Historisierung-mit-Hibernate-Envers-2477.html)