Hibernate Envers - Easy Entity Auditing

1

# Hibernate Envers Reference Documentation

3.6.8.Final

## Preface

The Envers project aims to enable easy auditing of persistent classes. All that you have to do is annotate your persistent class or some of its properties, that you want to audit, with `@Audited`. For each audited entity, a table will be created, which will hold the history of changes made to the entity. You can then retrieve and query historical data without much effort.

Similarly to Subversion, the library has a concept of revisions. Basically, one transaction is one revision (unless the transaction didn't modify any audited entities). As the revisions are global, having a revision number, you can query for various entities at that revision, retrieving a (partial) view of the database at that revision. You can find a revision number having a date, and the other way round, you can get the date at which a revision was commited.

The library works with Hibernate and requires Hibernate Annotations or Entity Manager. For the auditing to work properly, the entities must have immutable unique identifiers (primary keys). You can use Envers wherever Hibernate works: standalone, inside JBoss AS, with JBoss Seam or Spring.

Some of the features:

1. auditing of all mappings defined by the JPA specification
2. auditing of Hibernate mappings, which extend JPA, like custom types and collections/maps of "simple" types (Strings, Integers, etc.) (see also *Chapter 10, Mapping exceptions*)
3. logging data for each revision using a "revision entity"
4. querying historical data

# Quickstart

If you're using JPA, when coniguring Hibernate (in `persistence.xml`), add the following event listeners: (this will allow Envers to check if any audited entities were modified)

```
<persistence-unit ...>
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<class>...</class>
<properties>
   <property name="hibernate.dialect" ... />
   <!-- other hibernate properties -->

   <property name="hibernate.ejb.event.post-insert"

value="org.hibernate.ejb.event.EJB3PostInsertEventListener,org.hibernate.envers.event.AuditEventListener"
>
   <property name="hibernate.ejb.event.post-update"

value="org.hibernate.ejb.event.EJB3PostUpdateEventListener,org.hibernate.envers.event.AuditEventListener"
>
   <property name="hibernate.ejb.event.post-delete"

value="org.hibernate.ejb.event.EJB3PostDeleteEventListener,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" />
</properties>
</persistence-unit>
```

If you're using Hibernate directly, add the following to `hibernate.cfg.xml`:

```
 <hibernate-configuration>
 <session-factory>

 <listener class="org.hibernate.envers.event.AuditEventListener" type="post-insert"/>
 <listener class="org.hibernate.envers.event.AuditEventListener" type="post-update"/>
 <listener class="org.hibernate.envers.event.AuditEventListener" type="post-delete"/>
 <listener class="org.hibernate.envers.event.AuditEventListener" type="pre-collection-update"/>
 <listener class="org.hibernate.envers.event.AuditEventListener" type="pre-collection-remove"/>
    <listener    class="org.hibernate.envers.event.AuditEventListener"    type="post-collection-
recreate"/>


 </session-factory>
 </hibernate-configuration>
```

The `EJB3Post...EvenListeners` are needed, so that ejb3 entity lifecycle callback methods work (`@PostPersist, @PostUpdate, @PostRemove`.

Then, annotate your persistent class with `@Audited` - this will make all properties audited. For example:

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited // that's the important part :)
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

And the referenced entity:

```
@Entity
@Audited
public class Address {
    @Id
    @GeneratedValue
    private int id;

    private String streetName;

    private Integer houseNumber;

    private Integer flatNumber;

    @OneToMany(mappedBy = "address")
    private Set<Person> persons;

    // add getters, setters, constructors, equals and hashCode here
}
```

And that's it! You create, modify and delete the entites as always. If you look at the generated schema, you will notice that it is unchanged by adding auditing for the Address and Person entities. Also, the data they hold is the same. There are, however, two new tables - `Address_AUD` and `Person_AUD`, which store the historical data, whenever you commit a transaction.

Instead of annotating the whole class and auditing all properties, you can annotate only some persistent properties with `@Audited`. This will cause only these properties to be audited.

You can access the audit (history) of an entity using the `AuditReader` interface, which you can obtain when having an open EntityManager.

```
AuditReader reader = AuditReaderFactory.get(entityManager);
Person oldPerson = reader.find(Person.class, personId, revision)
```

The `T find(Class<T> cls, Object primaryKey, Number revision)` method returns an entity with the given primary key, with the data it contained at the given revision. If the entity didn't exist at this revision, `null` is returned. Only the audited properties will be set on the returned entity. The rest will be `null`.

You can also get a list of revisions at which an entity was modified using the `getRevisions` method, as well as retrieve the date, at which a revision was created using the `getRevisionDate` method.

# Short example

For example, using the entities defined above, the following code will generate revision number 1, which will contain two new `Person` and two new `Address` entities:

```
entityManager.getTransaction().begin();

Address address1 = new Address("Privet Drive", 4);
Person person1 = new Person("Harry", "Potter", address1);

Address address2 = new Address("Grimmauld Place", 12);
Person person2 = new Person("Hermione", "Granger", address2);

entityManager.persist(address1);
entityManager.persist(address2);
entityManager.persist(person1);
entityManager.persist(person2);

entityManager.getTransaction().commit();
```

Now we change some entities. This will generate revision number 2, which will contain modifications of one person entity and two address entities (as the collection of persons living at `address2` and `address1` changes):

```
entityManager.getTransaction().begin();

Address address1 = entityManager.find(Address.class, address1.getId());
Person person2 = entityManager.find(Person.class, person2.getId());

// Changing the address's house number
address1.setHouseNumber(5)

// And moving Hermione to Harry
person2.setAddress(address1);

entityManager.getTransaction().commit();
```

We can retrieve the old versions (the audit) easily:

```
AuditReader reader = AuditReaderFactory.get(entityManager);

Person person2_rev1 = reader.find(Person.class, person2.getId(), 1);
assert person2_rev1.getAddress().equals(new Address("Grimmauld Place", 12));

Address address1_rev1 = reader.find(Address.class, address1.getId(), 1);
assert address1_rev1.getPersons().getSize() == 1;

// and so on
```

# Configuration

## 3.1. Basic configuration

To start working with Envers, all configuration that you must do is add the event listeners to persistence.xml, as described in the *Chapter 1, Quickstart*.

However, as Envers generates some entities, and maps them to tables, it is possible to set the prefix and suffix that is added to the entity name to create an audit table for an entity, as well as set the names of the fields that are generated.

## 3.2. Choosing an audit strategy

After the basic configuration it is important to choose the audit strategy that will be used to persist and retrieve audit information. There is a trade-off is between the performance of persisting and the performance of querying the audit information. Currently there two audit strategies:

1. The default audit strategy persists the audit data together with a start revision. For each row inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, together with the start revision of its validity. Rows in the audit tables are never updated after insertion. Queries of audit information use subqueries to select the applicable rows in the audit tables. These subqueries are notoriously slow and difficult to index.
2. The alternative is a validity audit strategy. This strategy stores the start-revision and the end-revision of audit information. For each row inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, together with the start revision of its validity. But at the same time the end-revision field of the previous audit rows (if available) are set to this revision. Queries on the audit information can then use 'between start and end revision' instead of subqueries as used by the default audit strategy. The consequence of this strategy is that persisting audit information will be a bit slower, because of the extra updates involved, but retrieving audit information will be a lot faster. This can be improved by adding extra indexes.

## 3.3. Reference

In more detail, here are the properties that you can set:

### Table 3.1. Envers Configuration Properties

| Property name | Default value | Description |
|---|---|---|
| org.hibernate.envers.audit_table_prefix | | String that will be prepended to the name of an audited entity to create the name of the entity, that will hold audit information. |

| Property name | Default value | Description |
| --- | --- | --- |
| org.hibernate.envers.audit_table_suffix | _AUD | String that will be appended to the name of an audited entity to create the name of the entity, that will hold audit information. If you audit an entity with a table name Person, in the default setting Envers will generate a `Person_AUD` table to store historical data. |
| org.hibernate.envers.revision_field_name | REV | Name of a field in the audit entity that will hold the revision number. |
| org.hibernate.envers.revision_type_field_name | REVTYPE | Name of a field in the audit entity that will hold the type of the revision (currently, this can be: add, mod, del). |
| org.hibernate.envers.revision_on_collection_change | true | Should a revision be generated when a not-owned relation field changes (this can be either a collection in a one-to-many relation, or the field using "mappedBy" attribute in a one-to-one relation). |
| org.hibernate.envers.do_not_audit_optimistic_locking_field | true | When true, properties to be used for optimistic locking, annotated with `@Version`, will be automatically not audited (their history won't be stored; it normally doesn't make sense to store it). |
| org.hibernate.envers.store_data_at_delete | false | Should the entity data be stored in the revision when the entity is deleted (instead of only storing the id and all other properties as null). This is not normally needed, as the data is present in the last-but-one revision. Sometimes, however, it is easier and more efficient to access it in the |

| Property name | Default value | Description |
| --- | --- | --- |
| | | last revision (then the data that the entity contained before deletion is stored twice). |
| org.hibernate.envers.default_schema | null (same as normal tables) | The default schema name that should be used for audit tables. Can be overriden using the `@AuditTable(schema="...")` annotation. If not present, the schema will be the same as the schema of the normal tables. |
| org.hibernate.envers.default_catalog | null (same as normal tables) | The default catalog name that should be used for audit tables. Can be overriden using the `@AuditTable(catalog="...")` annotation. If not present, the catalog will be the same as the catalog of the normal tables. |
| org.hibernate.envers.audit_strategy | org.hibernate.envers.strategy.DefaultAuditStrategy | The AuditStrategy that should be used when persisting audit data. The default stores only the revision, at which an entity was modified. An alternative, the `org.hibernate.envers.strategy.ValidityAudi` stores both the start revision and the end revision. Together these define when an audit row was valid, hence the name ValidityAuditStrategy. |
| org.hibernate.envers.audit_strategy_validity_end_rev_field_name | REVEND | The column name that will hold the end revision number in audit entities. This property is only valid if the validity audit strategy is used. |
| org.hibernate.envers.audit_strategy_validity_store_revend_timestamp | false | Should the timestamp of the end revision be stored, until which the data was valid, in addition to the end revision itself. This is useful to be able |

| Property name | Default value | Description |
|---|---|---|
| | | to purge old Audit records out of a relational database by using table partitioning. Partitioning requires a column that exists within the table. This property is only evaluated if the ValidityAuditStrategy is used. |
| org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name | REVEND_TSTMP | Column name of the timestamp of the end revision until which the data was valid. Only used if the ValidityAuditStrategy is used, and org.hibernate.envers.audit_strategy_validity_store_ evaluates to true |

**Important**

The following configuration options have been added recently and should be regarded as experimental:

1. org.hibernate.envers.audit_strategy
2. org.hibernate.envers.audit_strategy_validity_end_rev_field_name
3. org.hibernate.envers.audit_strategy_validity_store_revend_timestamp
4. org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name

To change the name of the revision table and its fields (the table, in which the numbers of revisions and their timestamps are stored), you can use the `@RevisionEntity` annotation. For more information, see *Chapter 4, Logging data for revisions*.

To set the value of any of the properties described above, simply add an entry to your `persistence.xml`. For example:

```
<persistence-unit ...>
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<class>...</class>
<properties>
   <property name="hibernate.dialect" ... />
   <!-- other hibernate properties -->
```

```
            <property name="hibernate.ejb.event.post-insert"

value="org.hibernate.ejb.event.EJB3PostInsertEventListener,org.hibernate.envers.event.AuditEventListener"
            >
               <property name="hibernate.ejb.event.post-update"

value="org.hibernate.ejb.event.EJB3PostUpdateEventListener,org.hibernate.envers.event.AuditEventListener"
            >
               <property name="hibernate.ejb.event.post-delete"

value="org.hibernate.ejb.event.EJB3PostDeleteEventListener,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" />

               <property name="org.hibernate.envers.versionsTableSuffix" value="_V" />
               <property name="org.hibernate.envers.revisionFieldName" value="ver_rev" />
               <!-- other envers properties -->
            </properties>
            </persistence-unit>
```

The `EJB3Post...EvenListeners` are needed, so that ejb3 entity lifecycle callback methods work (`@PostPersist, @PostUpdate, @PostRemove`.

You can also set the name of the audit table on a per-entity basis, using the `@AuditTable` annotation. It may be tedious to add this annotation to every audited entity, so if possible, it's better to use a prefix/suffix.

If you have a mapping with secondary tables, audit tables for them will be generated in the same way (by adding the prefix and suffix). If you wish to overwrite this behaviour, you can use the `@SecondaryAuditTable` and `@SecondaryAuditTables` annotations.

If you'd like to override auditing behaviour of some fields/properties in an embedded component, you can use the `@AuditOverride(s)` annotation on the place where you use the component.

If you want to audit a relation mapped with `@OneToMany+@JoinColumn`, please see *Chapter 10, Mapping exceptions* for a description of the additional `@AuditJoinTable` annotation that you'll probably want to use.

If you want to audit a relation, where the target entity is not audited (that is the case for example with dictionary-like entities, which don't change and don't have to be audited), just annotate it with `@Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)`. Then, when reading historic versions of your entity, the relation will always point to the "current" related entity.

If you'd like to audit properties encapsulated by any subset of your entity's mapped superclasses (which are not explicitly audited), list desired supertypes in `auditParents` attribute of `@Audited` annotation. If any `@MappedSuperclass` (or any of it's properties) is marked as `@Audited`, it's behavior is implicitly inherited by all audited subclasses.

# Logging data for revisions

Envers provides an easy way to log additional data for each revision. You simply need to annotate one entity with `@RevisionEntity`, and a new instance of this entity will be persisted when a new revision is created (that is, whenever an audited entity is modified). As revisions are global, you can have at most one revisions entity.

Please note that the revision entity must be a mapped Hibernate entity.

This entity must have at least two properties:

1. an integer- or long-valued property, annotated with `@RevisionNumber`. Most often, this will be an auto-generated primary key.
2. a long- or j.u.Date- valued property, annotated with `@RevisionTimestamp`. Value of this property will be automatically set by Envers.

You can either add these properties to your entity, or extend `org.hibernate.envers.DefaultRevisionEntity`, which already has those two properties.

When using a `Date`, instead of a `long/Long` for the revision timestamp, take care not to use a mapping of the property which will loose precision (for example, using `@Temporal(DATE)` is wrong, as it doesn't store the time information, so many of your revisions will appear to happen at exactly the same time). A good choice is a `@Temporal(TIMESTAMP)`.

To fill the entity with additional data, you'll need to implement the `org.jboss.envers.RevisionListener` interface. Its newRevision method will be called when a new revision is created, before persisting the revision entity. The implementation should be stateless and thread-safe. The listener then has to be attached to the revisions entity by specifying it as a parameter to the `@RevisionEntity` annotation.

Alternatively, you can use the `getCurrentRevision` method of the `AuditReader` interface to obtain the current revision, and fill it with desired information. The method has a `persist` parameter specifying, if the revision entity should be persisted before returning. If set to `true`, the revision number will be available in the returned revision entity (as it is normally generated by the database), but the revision entity will be persisted regardless of wheter there are any audited entities changed. If set to `false`, the revision number will be `null`, but the revision entity will be persisted only if some audited entities have changed.

A simplest example of a revisions entity, which with each revision associates the username of the user making the change is:

```
package org.jboss.envers.example;

import org.hibernate.envers.RevisionEntity;
import org.hibernate.envers.DefaultRevisionEntity;

import javax.persistence.Entity;
```

```
@Entity
@RevisionEntity(ExampleListener.class)
public class ExampleRevEntity extends DefaultRevisionEntity {
 private String username;

 public String getUsername() { return username; }
 public void setUsername(String username) { this.username = username; }
}
```

Or, if you don't want to extend any class:

```
package org.hibernate.envers.example;

import org.hibernate.envers.RevisionNumber;
import org.hibernate.envers.RevisionTimestamp;
import org.hibernate.envers.RevisionEntity;

import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Entity;

@Entity
@RevisionEntity(ExampleListener.class)
public class ExampleRevEntity {
    @Id
    @GeneratedValue
    @RevisionNumber
    private int id;

    @RevisionTimestamp
    private long timestamp;

    private String username;

    // Getters, setters, equals, hashCode ...
}
```

An example listener, which, if used in a JBoss Seam application, stores the currently logged in user username:

```
package org.hibernate.envers.example;

import org.hibernate.envers.RevisionListener;
import org.jboss.seam.security.Identity;
import org.jboss.seam.Component;

public class ExampleListener implements RevisionListener {
    public void newRevision(Object revisionEntity) {
        ExampleRevEntity exampleRevEntity = (ExampleRevEntity) revisionEntity;
       Identity identity = (Identity) Component.getInstance("org.jboss.seam.security.identity");

        exampleRevEntity.setUsername(identity.getUsername());
    }
```

```
}
```

Having an "empty" revision entity - that is, with no additional properties except the two mandatory ones - is also an easy way to change the names of the table and of the properties in the revisions table automatically generated by Envers.

In case there is no entity annotated with `@RevisionEntity`, a default table will be generated, with the name `REVINFO`.

## 4.1. Tracking entity names modified during revisions

By default entity types that have been changed in each revision are not being tracked. This implies the necessity to query all tables storing audited data in order to retrieve changes made during specified revision. Users are allowed to implement custom mechanism of tracking modified entity names. In this case, they shall pass their own implementation of `org.hibernate.envers.EntityTrackingRevisionListener` interface as the value of `@org.hibernate.envers.RevisionEntity` annotation. `EntityTrackingRevisionListener` interface exposes one method that notifies whenever audited entity instance has been added, modified or removed within current revision boundaries.

### Example 4.1. Custom implementation of tracking entity classes modified during revisions

```
CustomEntityTrackingRevisionListener.java

public class CustomEntityTrackingRevisionListener
             implements EntityTrackingRevisionListener {
    @Override
    public void entityChanged(Class entityClass, String entityName,
                              Serializable entityId, RevisionType revisionType,
                              Object revisionEntity) {
        String type = entityClass.getName();
        ((CustomTrackingRevisionEntity)revisionEntity).addModifiedEntityType(type);
    }

    @Override
    public void newRevision(Object revisionEntity) {
    }
}
```

```
CustomTrackingRevisionEntity.java

@Entity
@RevisionEntity(CustomEntityTrackingRevisionListener.class)
public class CustomTrackingRevisionEntity {
    @Id
```

```
    @GeneratedValue
    @RevisionNumber
    private int customId;

    @RevisionTimestamp
    private long customTimestamp;

    @OneToMany(mappedBy="revision", cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    private Set<ModifiedEntityTypeEntity> modifiedEntityTypes =
                                            new HashSet<ModifiedEntityTypeEntity>();

    public void addModifiedEntityType(String entityClassName) {
        modifiedEntityTypes.add(new ModifiedEntityTypeEntity(this, entityClassName));
    }

    ...
}
```

ModifiedEntityTypeEntity.java

```
@Entity
public class ModifiedEntityTypeEntity {
    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne
    private CustomTrackingRevisionEntity revision;

    private String entityClassName;

    ...
}
```

```
CustomTrackingRevisionEntity revEntity =
    getAuditReader().findRevision(CustomTrackingRevisionEntity.class, revisionNumber);
Set<ModifiedEntityTypeEntity> modifiedEntityTypes = revEntity.getModifiedEntityTypes()
```

# Queries

You can think of historic data as having two dimension. The first - horizontal - is the state of the database at a given revision. Thus, you can query for entities as they were at revision N. The second - vertical - are the revisions, at which entities changed. Hence, you can query for revisions, in which a given entity changed.

The queries in Envers are similar to *Hibernate Criteria* [http://www.hibernate.org/hib_docs/v3/reference/en/html/querycriteria.html], so if you are common with them, using Envers queries will be much easier.

The main limitation of the current queries implementation is that you cannot traverse relations. You can only specify constraints on the ids of the related entities, and only on the "owning" side of the relation. This however will be changed in future releases.

Please note, that queries on the audited data will be in many cases much slower than corresponding queries on "live" data, as they involve correlated subselects.

In the future, queries will be improved both in terms of speed and possibilities, when using the valid-time audit strategy, that is when storing both start and end revisions for entities. See *Chapter 3, Configuration*.

## 5.1. Querying for entities of a class at a given revision

The entry point for this type of queries is:

```
AuditQuery query = getAuditReader().createQuery().forEntitiesAtRevision(MyEntity.class,
 revisionNumber);
```

You can then specify constraints, which should be met by the entities returned, by adding restrictions, which can be obtained using the `AuditEntity` factory class. For example, to select only entities, where the "name" property is equal to "John":

```
query.add(AuditEntity.property("name").eq("John"));
```

And to select only entites that are related to a given entity:

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

You can limit the number of results, order them, and set aggregations and projections (except grouping) in the usual way. When your query is complete, you can obtain the results by calling the `getSingleResult()` or `getResultList()` methods.

A full query, can look for example like this:

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

## 5.2. Querying for revisions, at which entities of a given class changed

The entry point for this type of queries is:

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

You can add constraints to this query in the same way as to the previous one. There are some additional possibilities:

1. using `AuditEntity.revisionNumber()` you can specify constraints, projections and order on the revision number, in which the audited entity was modified
2. similarly, using `AuditEntity.revisionProperty(propertyName)` you can specify constraints, projections and order on a property of the revision entity, corresponding to the revision in which the audited entity was modified
3. `AuditEntity.revisionType()` gives you access as above to the type of the revision (ADD, MOD, DEL).

Using these methods, you can order the query results by revision number, set projection or constraint the revision number to be greater or less than a specified value, etc. For example, the following query will select the smallest revision number, at which entity of class `MyEntity` with id `entityId` has changed, after revision number 42:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
```

```
    .getSingleResult();
```

The second additional feature you can use in queries for revisions is the ability to maximalize/minimize a property. For example, if you want to select the revision, at which the value of the `actualDate` for a given entity was larger then a given value, but as small as possible:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.property("actualDate").minimize()
        .add(AuditEntity.property("actualDate").ge(givenDate))
        .add(AuditEntity.id().eq(givenEntityId)))
    .getSingleResult();
```

The `minimize()` and `maximize()` methods return a criteria, to which you can add constraints, which must be met by the entities with the maximized/minimized properties.

You probably also noticed that there are two boolean parameters, passed when creating the query. The first one, `selectEntitiesOnly`, is only valid when you don't set an explicit projection. If true, the result of the query will be a list of entities (which changed at revisions satisfying the specified constraints).

If false, the result will be a list of three element arrays. The first element will be the changed entity instance. The second will be an entity containing revision data (if no custom entity is used, this will be an instance of `DefaultRevisionEntity`). The third will be the type of the revision (one of the values of the `RevisionType` enumeration: ADD, MOD, DEL).

The second parameter, `selectDeletedEntities`, specifies if revisions, in which the entity was deleted should be included in the results. If yes, such entities will have the revision type DEL and all fields, except the id, `null`.

# Generating schema with Ant

If you'd like to generate the database schema file with the Hibernate Tools Ant task, you'll probably notice that the generated file doesn't contain definitions of audit tables. To generate also the audit tables, you simply need to use `org.hibernate.tool.ant.EnversHibernateToolTask` instead of the usual `org.hibernate.tool.ant.HibernateToolTask`. The former class extends the latter, and only adds generation of the version entities. So you can use the task just as you used to.

For example:

```
<target name="schemaexport" depends="build-demo"
  description="Exports a generated schema to DB and file">
  <taskdef name="hibernatetool"
    classname="org.hibernate.tool.ant.EnversHibernateToolTask"
    classpathref="build.demo.classpath"/>

  <hibernatetool destdir=".">
    <classpath>
      <fileset refid="lib.hibernate" />
      <path location="${build.demo.dir}" />
      <path location="${build.main.dir}" />
    </classpath>
    <jpaconfiguration persistenceunit="ConsolePU" />
    <hbm2ddl
      drop="false"
      create="true"
      export="false"
      outputfilename="versioning-ddl.sql"
      delimiter=";"
      format="true"/>
  </hibernatetool>
</target>
```

Will generate the following schema:

```


    create table Address (
        id integer generated by default as identity (start with 1),
        flatNumber integer,
        houseNumber integer,
        streetName varchar(255),
        primary key (id)
    );

    create table Address_AUD (
        id integer not null,
        REV integer not null,
        flatNumber integer,
        houseNumber integer,
        streetName varchar(255),
        REVTYPE tinyint,
        primary key (id, REV)
```

```
    );

    create table Person (
        id integer generated by default as identity (start with 1),
        name varchar(255),
        surname varchar(255),
        address_id integer,
        primary key (id)
    );

    create table Person_AUD (
        id integer not null,
        REV integer not null,
        name varchar(255),
        surname varchar(255),
        REVTYPE tinyint,
        address_id integer,
        primary key (id, REV)
    );

    create table REVINFO (
        REV integer generated by default as identity (start with 1),
        REVTSTMP bigint,
        primary key (REV)
    );

    alter table Person
        add constraint FK8E488775E4C3EA63
        foreign key (address_id)
        references Address;
```

# Generated tables and their content

For each audited entity (that is, for each entity containing at least one audited field), an audit table is created. By default, the audit table's name is created by adding a "_AUD" suffix to the original name, but this can be overriden by specifing a different suffix/prefix (see *Chapter 3, Configuration*) or on a per-entity basis using the `@AuditTable` annotation.

The audit table has the following fields:

1. id of the original entity (this can be more then one column, if using an embedded or multiple id)
2. revision number - an integer
3. revision type - a small integer
4. audited fields from the original entity

The primary key of the audit table is the combination of the original id of the entity and the revision number - there can be at most one historic entry for a given entity instance at a given revision.

The current entity data is stored in the original table and in the audit table. This is a duplication of data, however as this solution makes the query system much more powerful, and as memory is cheap, hopefully this won't be a major drawback for the users. A row in the audit table with entity id ID, revision N and data D means: entity with id ID has data D from revision N upwards. Hence, if we want to find an entity at revision M, we have to search for a row in the audit table, which has the revision number smaller or equal to M, but as large as possible. If no such row is found, or a row with a "deleted" marker is found, it means that the entity didn't exist at that revision.

The "revision type" field can currently have three values: 0, 1, 2, which means, respectively, ADD, MOD and DEL. A row with a revision of type DEL will only contain the id of the entity and no data (all fields NULL), as it only serves as a marker saying "this entity was deleted at that revision".

Additionaly, there is a "REVINFO" table generated, which contains only two fields: the revision id and revision timestamp. A row is inserted into this table on each new revision, that is, on each commit of a transaction, which changes audited data. The name of this table can be configured, as well as additional content stored, using the `@RevisionEntity` annotation, see *Chapter 4, Logging data for revisions*.

While global revisions are a good way to provide correct auditing of relations, some people have pointed out that this may be a bottleneck in systems, where data is very often modified. One viable solution is to introduce an option to have an entity "locally revisioned", that is revisions would be created for it independently. This wouldn't enable correct versioning of relations, but wouldn't also require the "REVINFO" table. Another possibility if to have "revisioning groups", that is groups of entities which share revision numbering. Each such group would have to consist of one or more strongly connected component of the graph induced by relations between entities. Your opinions on the subject are very welcome on the forum! :)

# Audit table partitioning

## 8.1. Benefits of audit table partitioning

Because audit tables tend to grow indefinitely they can quickly become really large. When the audit tables have grown to a certain limit (varying per RDBMS and/or operating system) it makes sense to start using table partitioning. SQL table partitioning offers a lot of advantages including, but certainly not limited to:

1. Improved query performance by selectively moving rows to various partitions (or even purging old rows)
2. Faster data loads, index creation, etc.

## 8.2. Suitable columns for audit table partitioning

Generally SQL tables must be partitioned on a column that exists within the table. As a rule it makes sense to use either the *end revision* or the *end revision timestamp* column for partioning of audit tables.

### Note

End revision information is not available for the default AuditStrategy.

Therefore the following Envers configuration options are required:

```
org.hibernate.envers.audit_strategy                                =
org.hibernate.envers.strategy.ValidityAuditStrategy
```

```
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp
= true
```

Optionally, you can also override the default values following properties:

```
org.hibernate.envers.audit_strategy_validity_end_rev_field_name
```

```
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name
```

For more information, see *Chapter 3, Configuration*.

The reason why the end revision information should be used for audit table partioning is based on the assumption that audit tables should be partionioned on an 'increasing level of interestingness', like so:

1. A couple of partitions with audit data that is not very (or no longer) interesting. This can be stored on slow media, and perhaps even be purged eventually.

2. Some partitions for audit data that is potentially interesting.

3. One partition for audit data that is most likely to be interesting. This should be stored on the fastest media, both for reading and writing.

## 8.3. Audit table partitioning example

In order to determine a suitable column for the 'increasing level of interestingness', consider a simplified example of a salary registration for an unnamed agency.

Currently, the salary table contains the following rows for a certain person X:

### Table 8.1. Salaries table

| Year | Salary (USD) |
|---|---|
| 2006 | 3300 |
| 2007 | 3500 |
| 2008 | 4000 |
| 2009 | 4500 |

The salary for the current fiscal year (2010) is unknown. The agency requires that all changes in registered salaries for a fiscal year are recorded (i.e. an audit trail). The rationale behind this is that decisions made at a certain date are based on the registered salary at that time. And at any time it must be possible reproduce the reason why a certain decision was made at a certain date.

The following audit information is available, sorted on in order of occurrence:

### Table 8.2. Salaries - audit table

| Year | Revision type | Revision timestamp | Salary (USD) | End revision timestamp |
|---|---|---|---|---|
| 2006 | ADD | 2007-04-01 | 3300 | null |
| 2007 | ADD | 2008-04-01 | 35 | 2008-04-02 |
| 2007 | MOD | 2008-04-02 | 3500 | null |
| 2008 | ADD | 2009-04-01 | 3700 | 2009-07-01 |
| 2008 | MOD | 2009-07-01 | 4100 | 2010-02-01 |
| 2008 | MOD | 2010-02-01 | 4000 | null |
| 2009 | ADD | 2010-04-01 | 4500 | null |

### 8.3.1. Determining a suitable partitioning column

To partition this data, the 'level of interestingness' must be defined. Consider the following:

1. For fiscal year 2006 there is only one revision. It has the oldest *revision timestamp* of all audit rows, but should still be regarded as interesting because it is the latest modification for this fiscal year in the salary table; its *end revision timestamp* is null.

   Also note that it would be very unfortunate if in 2011 there would be an update of the salary for fiscal year 2006 (which is possible in until at least 10 years after the fiscal year) and the audit information would have been moved to a slow disk (based on the age of the *revision timestamp*). Remember that in this case Envers will have to update the *end revision timestamp* of the most recent audit row.

2. There are two revisions in the salary of fiscal year 2007 which both have nearly the same *revision timestamp* and a different *end revision timestamp*. On first sight it is evident that the first revision was a mistake and probably uninteresting. The only interesting revision for 2007 is the one with *end revision timestamp* null.

Based on the above, it is evident that only the *end revision timestamp* is suitable for audit table partitioning. The *revision timestamp* is not suitable.

### 8.3.2. Determining a suitable partitioning scheme

A possible partitioning scheme for the salary table would be as follows:

1. *end revision timestamp* year = 2008

   This partition contains audit data that is not very (or no longer) interesting.

2. *end revision timestamp* year = 2009

   This partition contains audit data that is potentially interesting.

3. *end revision timestamp* year >= 2010 or null

   This partition contains the most interesting audit data.

This partitioning scheme also covers the potential problem of the update of the *end revision timestamp*, which occurs if a row in the audited table is modified. Even though Envers will update the *end revision timestamp* of the audit row to the system date at the instant of modification, the audit row will remain in the same partition (the 'extension bucket').

And sometime in 2011, the last partition (or 'extension bucket') is split into two new partitions:

1. *end revision timestamp* year = 2010

   This partition contains audit data that is potentially interesting (in 2011).

2. *end revision timestamp* year >= 2011 or null

   This partition contains the most interesting audit data and is the new 'extension bucket'.

# Building from source and testing

## 9.1. Building from source

Envers, as a module of Hibernate, uses the standard Hibernate build. So all the usual build targets (compile, test, install) will work.

The public Hibernate Git repository is hosted at GitHub and can be browsed using *GitHub* [https://github.com/hibernate/hibernate-core]. The source can be checked out using either

```
git clone https://github.com/hibernate/hibernate-core hibernate-core.git
git clone git://github.com/hibernate/hibernate-core.git
```

## 9.2. Contributing

If you want to contribute a fix or new feature, either:

- use the GitHub fork capability: clone, work on a branch, fork the repo on GitHub (fork button), push the work there and trigger a pull request (pull request button).
- use the pure Git approach: clone, work on a branch, push to a public fork repo hosted somewhere, trigger a pull request (`git pull-request`)
- provide a good old patch file: clone the repo, create a patch with git format-patch or diff and attach the patch file to JIRA

## 9.3. Envers integration tests

The tests use, by default, use a H2 in-memory database. The configuration file can be found in `src/test/resources/hibernate.test.cfg.xml`.

The tests use TestNG, and can be found in the `org.hibernate.envers.test.integration` package (or rather, in subpackages of this package). The tests aren't unit tests, as they don't test individual classes, but the behaviour and interaction of many classes, hence the name of package.

A test normally consists of an entity (or two entities) that will be audited and extends the `AbstractEntityTest` class, which has one abstract method: `configure(Ejb3Configuration)`. The role of this method is to add the entities that will be used in the test to the configuration.

The test data is in most cases created in the "initData" method (which is called once before the tests from this class are executed), which normally creates a couple of revisions, by persisting and updating entities. The tests first check if the revisions, in which entities where modified are

correct (the testRevisionCounts method), and if the historic data is correct (the testHistoryOfXxx methods).

# Mapping exceptions

## 10.1. What isn't and will not be supported

Bags (the corresponding Java type is List), as they can contain non-unique elements. The reason is that persisting, for example a bag of String-s, violates a principle of relational databases: that each table is a set of tuples. In case of bags, however (which require a join table), if there is a duplicate element, the two tuples corresponding to the elements will be the same. Hibernate allows this, however Envers (or more precisely: the database connector) will throw an exception when trying to persist two identical elements, because of a unique constraint violation.

There are at least two ways out if you need bag semantics:

1. use an indexed collection, with the `@IndexColumn` annotation, or
2. provide a unique id for your elements with the `@CollectionId` annotation.

## 10.2. What isn't and *will* be supported

1. collections of components

## 10.3. `@OneToMany`+`@JoinColumn`

When a collection is mapped using these two annotations, Hibernate doesn't generate a join table. Envers, however, has to do this, so that when you read the revisions in which the related entity has changed, you don't get false results.

To be able to name the additional join table, there is a special annotation: `@AuditJoinTable`, which has similar semantics to JPA's `@JoinTable`.

One special case are relations mapped with `@OneToMany+@JoinColumn` on the one side, and `@ManyToOne+@JoinColumn(insertable=false, updatable=false)` on the many side. Such relations are in fact bidirectional, but the owning side is the collection (see alse *here* [http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/#entity-hibspec-collection-extratype]).

To properly audit such relations with Envers, you can use the `@AuditMappedBy` annotation. It enables you to specify the reverse property (using the `mappedBy` element). In case of indexed collections, the index column must also be mapped in the referenced entity (using `@Column(insertable=false, updatable=false)`, and specified using `positionMappedBy`. This annotation will affect only the way Envers works. Please note that the annotation is experimental and may change in the future.

# Migration from Envers standalone

With the inclusion of Envers as a Hibernate module, some of the public API and configuration defaults changed. In general, "versioning" is renamed to "auditing" (to avoid confusion with the annotation used for indicating an optimistic locking field - `@Version`).

Because of changing some configuration defaults, there should be no more problems using Envers out-of-the-box with Oracle and other databases, which don't allow tables and field names to start with "_".

## 11.1. Changes to code

Public API changes involve changing "versioning" to "auditing". So, `@Versioned` became `@Audited`; `@VersionsTable` became `@AuditTable` and so on.

Also, the query interface has changed slightly, mainly in the part for specifying restrictions, projections and order. Please refer to the Javadoc for further details.

## 11.2. Changes to configuration

First of all, the name of the event listener changed. It is now named `org.hibernate.envers.event.AuditEventListener`, instead of `org.jboss.envers.event.VersionsEventListener`. So to make Envers work, you will have to change these settings in your `persistence.xml` or Hibernate configuration.

Secondly, the names of the audit (versions) tables and additional auditing (versioning) fields changed. The default suffix added to the table name is now `_AUD`, instead of `_versions`. The name of the field that holds the revision number, and which is added to each audit (versions) table, is now `REV`, instead of `_revision`. Finally, the name of the field that holds the type of the revision, is now `REVTYPE`, instead of `_rev_type`.

If you have a schema generated with the old version of Envers, you will have to set those properties, to use the new version of Envers without problems:

```
<persistence-unit ...>
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<class>...</class>
<properties>
   <property name="hibernate.dialect" ... />
   <!-- other hibernate properties -->

   <!-- Envers listeners -->

   <property name="org.hibernate.envers.auditTableSuffix" value="_versions" />
   <property name="org.hibernate.envers.revisionFieldName" value="_revision" />
   <property name="org.hibernate.envers.revisionTypeFieldName" value="_rev_type" />
   <!-- other envers properties -->
</properties>
```

```
</persistence-unit>
```

The `org.hibernate.envers.doNotAuditOptimisticLockingField` property is now by default `true`, instead of `false`. You probably never would want to audit the optimistic locking field. Also, the `org.hibernate.envers.warnOnUnsupportedTypes` configuraiton option was removed. In case you are using some unsupported types, use the `@NotAudited` annotation.

See *Chapter 3, Configuration* for details on the configuration and a description of the configuration options.

## 11.3. Changes to the revision entity

This section applies only if you don't have a custom revision entity. The name of the revision entity generated by default changed, so if you used the default one, you'll have to add a custom revision entity, and map it to the old table. Here's the class that you have to create:

```
package org.hibernate.envers.example;

import org.hibernate.envers.RevisionNumber;
import org.hibernate.envers.RevisionTimestamp;
import org.hibernate.envers.RevisionEntity;

import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Entity;
import javax.persistence.Column;
import javax.persistence.Table;

@Entity
@RevisionEntity
@Table(name="_revisions_info")
public class ExampleRevEntity {
    @Id
    @GeneratedValue
    @RevisionNumber
    @Column(name="revision_id")
    private int id;

    @RevisionTimestamp
    @Column(name="revision_timestamp")
    private long timestamp;

    // Getters, setters, equals, hashCode ...
}
```

# Links

Some useful links:

1. *Hibernate* [http://hibernate.org]
2. *Forum* [http://community.jboss.org/en/envers?view=discussions]
3. *Anonymous SVN* [http://anonsvn.jboss.org/repos/hibernate/core/trunk/envers/]
4. *JIRA issue tracker* [http://opensource.atlassian.com/projects/hibernate/browse/HHH] (when adding issues concerning Envers, be sure to select the "envers" component!)
5. *IRC channel* [irc://irc.freenode.net:6667/envers]
6. *Blog* [http://www.jboss.org/feeds/view/envers]
7. *FAQ* [https://community.jboss.org/wiki/EnversFAQ]