NHibernate Envers Reference Documentation

Version: 1.0.0

Table of Contents

Preface

The Envers project aims to enable easy auditing of persistent classes. All that you have to do is to configure Envers to audit some of your persistent classes. 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 source control systems, 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 and 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 NHibernate version 3.1 or later. For the auditing to work properly, the entities must have immutable unique identifiers (primary keys).

Some of the features:

1. Auditing of all NHibernate mappings
2. Auditing custom types and collections/maps of "simple" types (strings, integers, etc.) (see also Chapter 9, *Mapping exceptions*)
3. Logging data for each revision using a "revision entity"
4. Querying historical data

Chapter 1. Quickstart

Out of the box there are two ways to configure Envers, by attributes or by code (fluent).

Simple attribute configuration

```
nhConf.IntegrateWithEnvers(new AttributeConfiguration());
[...]

[Audited]
public class Person
[...]
```

Simple fluent configuration

```
var enversConf = new FluentConfiguration();
enversConf.Audit<Person>();
nhConf.IntegrateWithEnvers(enversConf);
```

And that's it! You create, modify and delete the entites as always. If you look at the generated schema, you will notice that current schema is unchanged. Also, the data they hold is the same. There are, however, new tables 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]` (attributes) or use "Exclude" method to exclude non audited properties (fluent).

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

```
var reader = AuditReaderFactory.Get(session);
var oldPerson = reader.Find(typeof(Person), personId, revision);
```

The `T Find<T>(object primaryKey, long 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 that 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.

Chapter 2. 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:

```
using(var tx = session.BeginTransaction())
{
        var address1 = new Address("Privet Drive", 4);
        var person1 = new Person("Harry", "Potter", address1);

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

        session.Save(address1);
        session.Save(address2);
        session.Save(person1);
        session.Save(person2);

        tx.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):

```
using(var tx = session.BeginTransaction())
{
        var address1 = session.Get<Address>(address1.Id);
        var person2 = session.Get<Person>(person2.Id);

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

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

        tx.Commit();
}
```

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

```
var reader = AuditReaderFactory.Get(session);

var person2_rev1 = reader.Find<Person>(person2.Id, 1);
Assert.AreEqual(person2_rev1.Address, new Address("Grimmauld Place", 12));

var address1_rev1 = reader.Find<Address>(address1.Id, 1);
Assert.AreEqual(address1_rev1.Persons.Count, 1);

// and so on
```

Chapter 3. Configuration

3.1. Basic configuration

To start working with Envers, all configuration needed is to call the extension method IntegrateWithEnvers() on the NH Configuration object, 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 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
nhibernate.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.
nhibernate.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

Property name	**Default value**	**Description**
		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.
nhibernate.envers.revision_field_name	REV	Name of a field in the audit entity that will hold the revision number.
nhibernate.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).
nhibernate.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).
nhibernate.envers.do_not_audit_optimistic_locking_field	true	When true, properties to be used for optimistic locking will be automatically not audited (their history won't be stored; it normally doesn't make sense to store it).
nhibernate.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 normally not 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 last revision (then the data that the entity contained before deletion is stored twice).
nhibernate.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="...")]` attribute. If not present, the schema will be the same as the schema of the normal tables.
nhibernate.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="...")]` attribute. If not present, the catalog will be the same as the catalog of the normal tables.

Property name	Default value	Description
nhibernate.envers.audit_strategy	NHibernate.Envers.Strategy.DefaultAuditStrategy	The audit strategy that should be used when persisting audit data. The default stores only the revision, at which an entity was modified. An alternative, the `NHibernate.Envers.Strategy.ValidityAuditStrategy` stores both the start revision and the end revision. Together these define when an audit row was valid, hence the name ValidityAuditStrategy.
nhibernate.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.
nhibernate.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 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.
nhibernate.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 nhibernate.envers.audit_strategy_validity_store_revend_timestamp evaluates to true
nhibernate.envers.collection_proxy_mapper_factory	NHibernate.Envers.Configuration.Metadata.DefaultCollectionProxyMapperFactory	Responsible to create collection proxies for audited entities. May be used if NHibernate Core isn't using its normal types for its mapped collections, eg if a user defined CollectionTypeFactory is used.

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]` attribute. 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 `hibernate.cfg.xml` (or to NH's `Configuration` object).

You can also set the name of the audit table on a per-entity basis, using the `[AuditTable]` attribute. 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 join 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 `[JoinAuditTable]` attribute.

If you'd like to override auditing behaviour of some fields/properties in an embedded component, you can use the `[AuditOverride]` atttribute on the place where you use the component.

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]` (or use fluent configuration: `ExcludeRelationData()`). Then, when reading historic versions of your entity, the relation will always point to the "current" related entity.

Chapter 4. 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]` (attribute) or `SetRevisionEntity` (fluent), 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 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 `DefaultRevisionEntity`, which already has those two properties.

To fill the entity with additional data, you'll need to implement the `IRevisionListener` 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]` attribute.

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:

```
[Entity]
[RevisionEntity(typeof(ExampleListener))]
public class ExampleRevEntity : DefaultRevisionEntity
{
        public virtual string UserName {get;set;}
}
```

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

```
[Entity]
[RevisionEntity(typeof(ExampleListener))]
public class ExampleRevEntity
{
    [RevisionNumber]
    private int id;

    [RevisionTimestamp]
    private long timestamp;

    private string username;

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

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`.

Chapter 5. 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 NHibernate Criteria [http://www.nhforge.org/doc/nh/en/index.html#querycriteria], 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 are usually a lot 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:

```
var query = AuditReader().CreateQuery()
        .ForEntitiesAtRevision(typeof(MyEntity, 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 `SingleResult()` or `ResultList()` methods.

A full query, can look for example like this:

```
var personsAtAddress = AuditReader().CreateQuery()
    .ForEntitiesAtRevision(typeof(Person), 12)
    .AddOrder(AuditEntity.property("surname").desc())
    .Add(AuditEntity.relatedId("address").eq(addressId))
    .SetFirstResult(4)
    .SetMaxResults(2)
    .ResultList();
```

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

The entry point for this type of queries is:

```
var query = AuditReader().CreateQuery()
    .ForRevisionsOfEntity(typeof(MyEntity), 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:

```
var revision = AuditReader().CreateQuery()
    .ForRevisionsOfEntity(typeof(MyEntity), false, true)
    .SetProjection(AuditEntity.RevisionNumber.Min())
    .Add(AuditEntity.Id.Eq(entityId))
    .Add(AuditEntity.RevisionNumber().Gt(42))
    .SingleResult();
```

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:

```
var revision = AuditReader().CreateQuery()
    .ForRevisionsOfEntity(typeof(MyEntity), 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)))
    .SingleResult();
```

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`.

5.3. `Auditer()`, an extension method on `ISession`

The extension method `Auditer()` offers an alternative API to query Envers.

```
IEnumerable<long> example = session.Auditer().GetRevisions<MyEntity>(id);
```

Chapter 6. Database schema

If following entities are audited,

```
        public class Address
        {
                public virtual int Id {get;set;}
                public virtual int FlatNumber {get;set;}
                public virtual int HouseNumber {get;set;}
                public virtual string StreetName {get;set;}
        }

        public class Person
        {
                public virtual int Id {get;set;}
                public virtual string Name {get;set;}
                public virtual string Surname {get;set;}
                public virtual Address Address {get;set;}
        }
```

we will need 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 not null,
        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 not null,
        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;
```

Chapter 7. 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]` attribute.

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 storage 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]` attribute, 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! :)

Chapter 8. Building from source and testing

You can clone the source code from BitBucket [https://RogerKratz@bitbucket.org/RogerKratz/nhibernate.envers].

A configuration template for tests, hibernate.cfg.xml.template, can be found in the root folder of the tests. Copy this file and rename it to hibernate.cfg.xml and point to a database of your choice.

The test data is in most cases created in the "initialize" 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).

Chapter 9. Mapping exceptions

9.1. What is not and will not be supported

Bags (the corresponding CLR type is IList<T>), 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. NHibernate 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, or
2. Provide a unique id for your elements.

9.2. What is not and *will* be supported

1. Collections of components

Chapter 10. Links

Some useful links:

1. NHibernate home page [http://nhforge.org]
2. User mailing list (ask questions here) [http://groups.google.com/group/nhusers]
3. Source code [https://bitbucket.org/RogerKratz/nhibernate.envers]
4. JIRA issue tracker [jira.nhforge.org] (when adding issues concerning Envers, be sure to select the "envers" component!)