Audit SPI Implementation

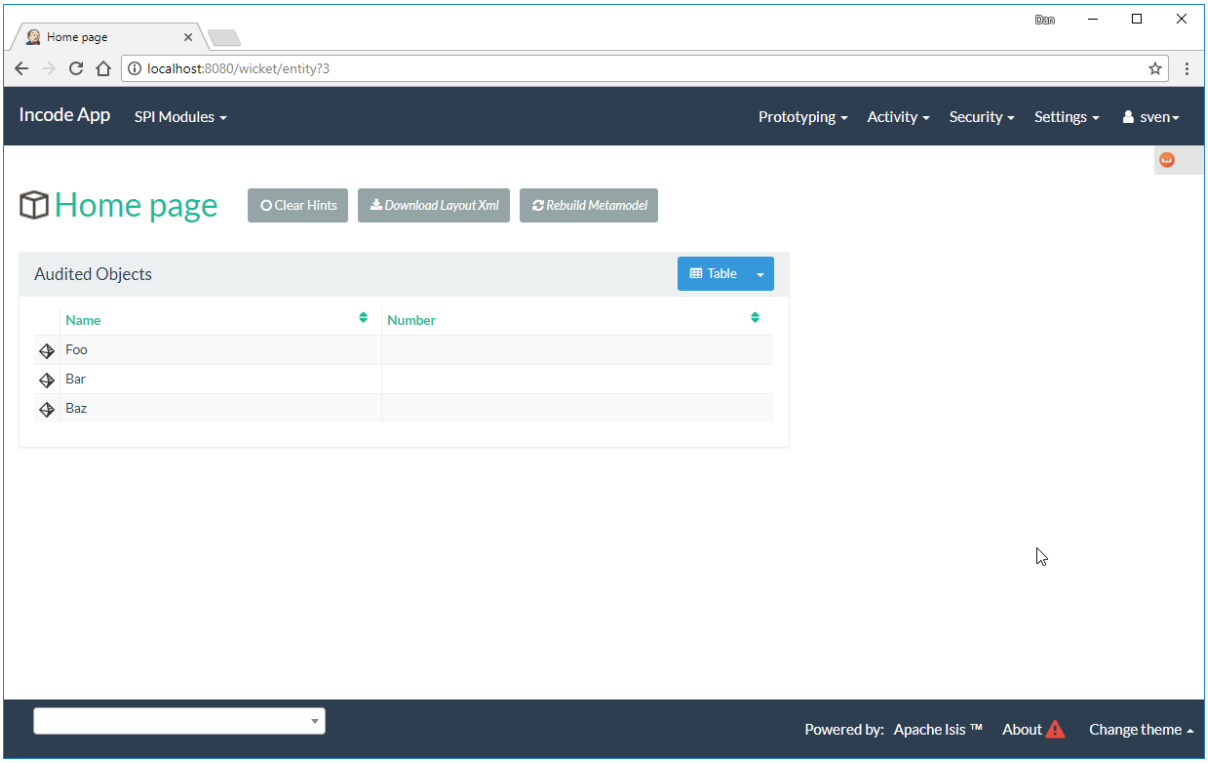
Table of Contents

This module (`isis-module-audit`) provides an implementation of Isis' `AuditerService` SPI (introduced in `1.13.0`, replacing the earlier `AuditingService3` SPI). The implementation persists audit entries using Isis' own (JDO) objectstore. Typically this will be to a relational database; the module's `AuditEntry` entity is mapped to the "isisaudit.AuditEntry" table.

Screenshots

The module's functionality can be explored by running the quickstart with example usage using the `org.incode.domainapp.example.app.modules.ExampleDomSpiAuditAppManifest`.

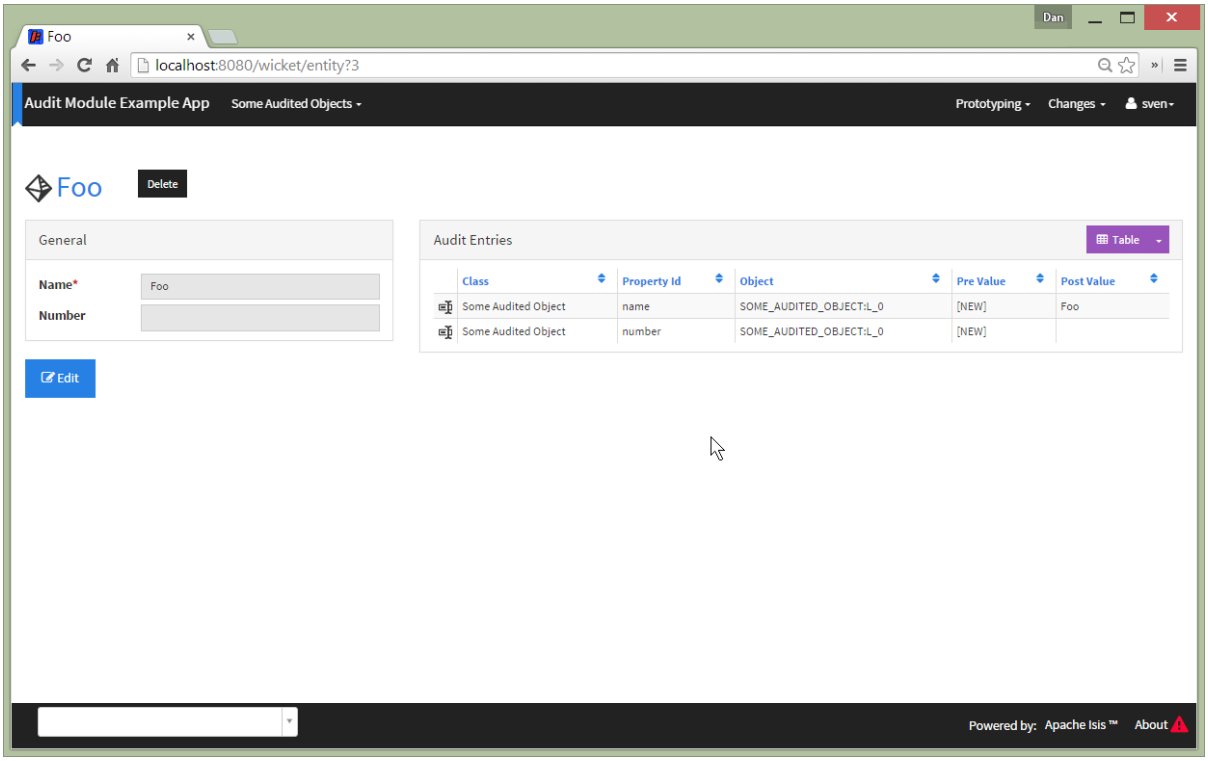
A home page is displayed when the app is run:



The remaining screenshots below **do** demonstrate the functionality of this module, but are out of date in that they are taken from the original isisaddons/incodehq module (prior to being amalgamated into the incode-platform).

Object with initial (creation) audit entries

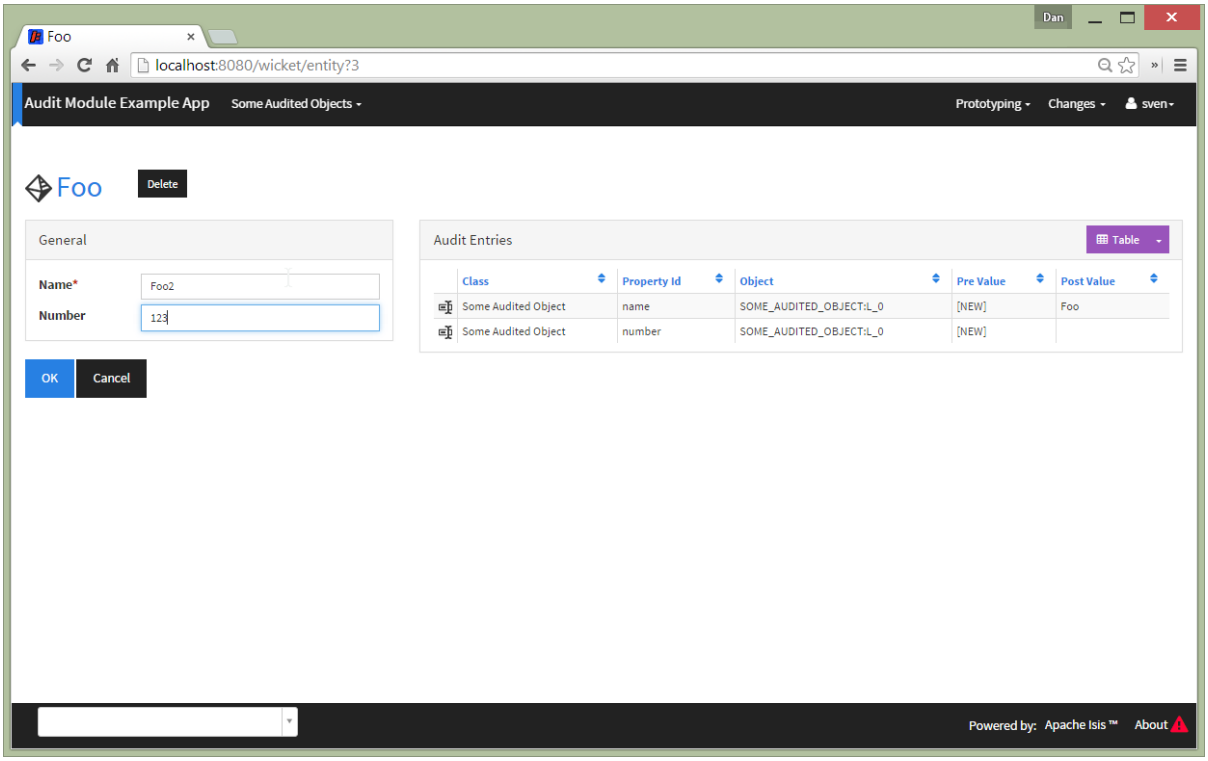
Because the example entity is annotated with `@DomainObject(auditing = Auditing.ENABLED)`, the initial creation of that object already results in some audit entries:



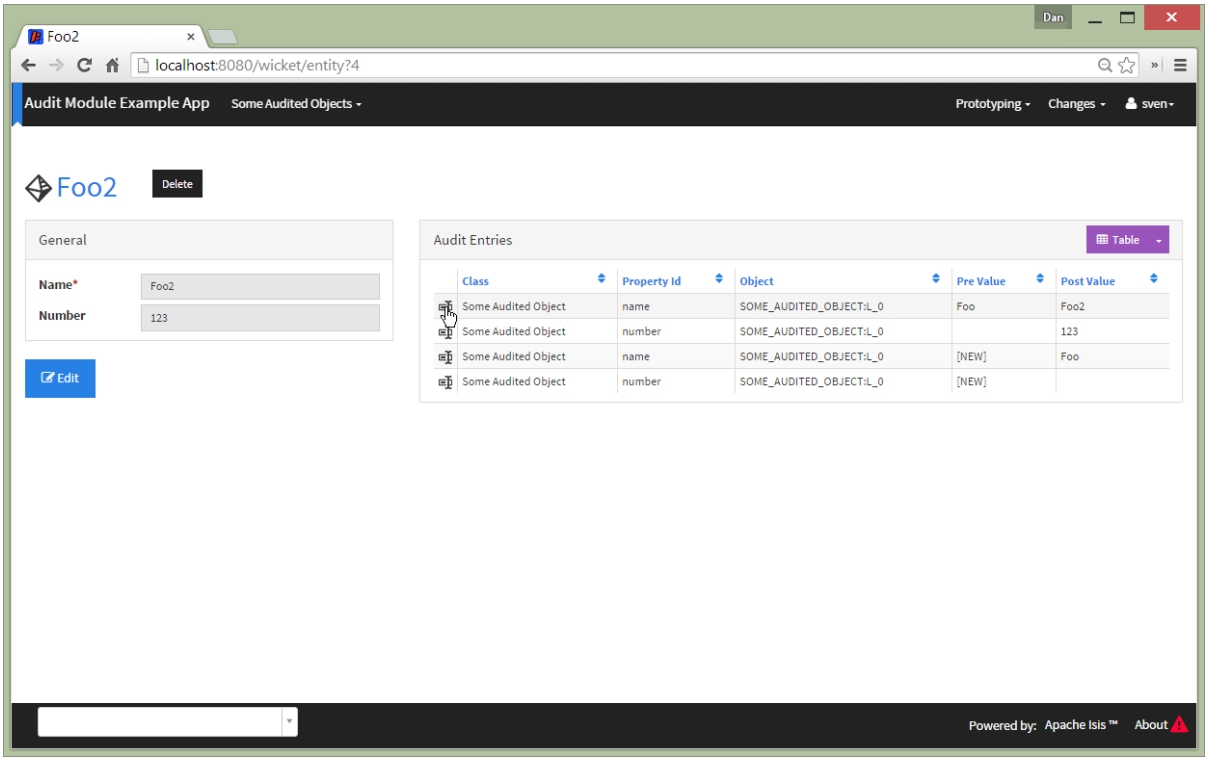
As the screenshot shows, the demo app lists the audit entries for the example entity (a polymorphic association utilizing the `BookmarkService`).

Audit Entry for each changed property

Changing two properties on an object:

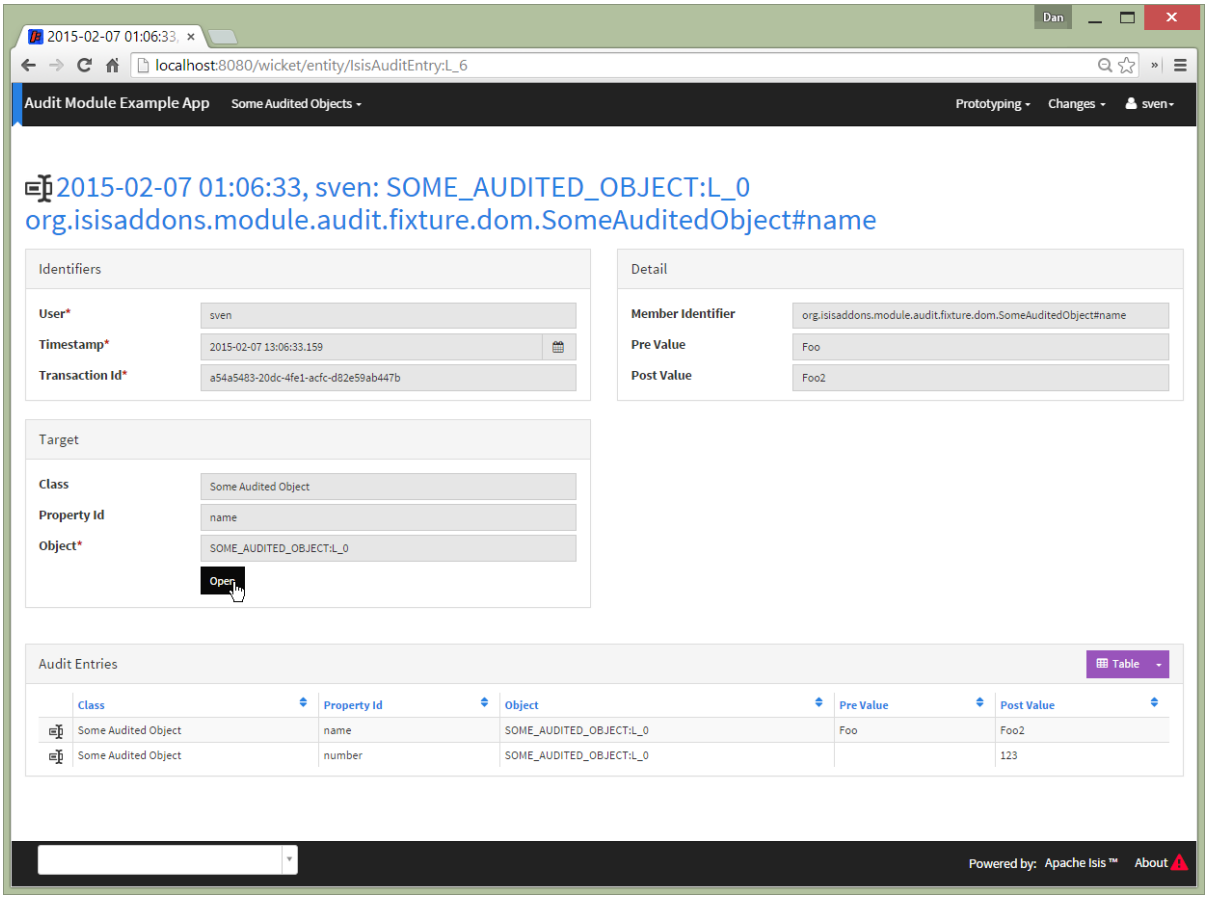


which results in *two* audit entries created, one for each property:

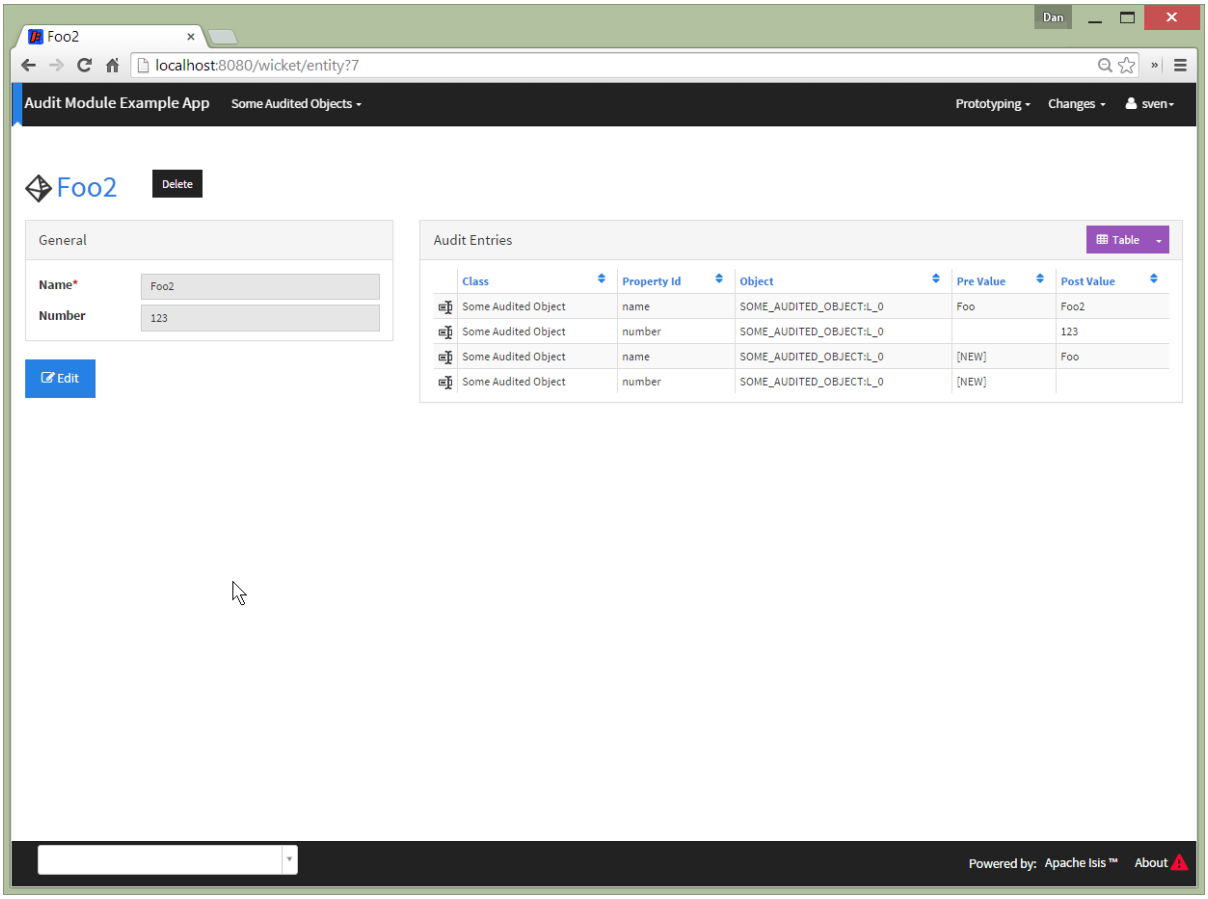


Audit entry

The audit entry is an immutable record, can also inspect other audit entries created in the same transaction:



It is of course also possible to navigate back to audited object:


Foo2
Dan
localhost:8080/wicket/entity?7
Audit Module Example App
Some Audited Objects
Prototyping
Changes
sven
Foo2
Delete
General
Name*
Foo2
Number
123
Edit
Audit Entries
Table
Class
Property Id
Object
Pre Value
Post Value
Some Audited Object
name
SOME_AUDITED_OBJECT:L_0
Foo
Foo2
Some Audited Object
number
SOME_AUDITED_OBJECT:L_0
123
Some Audited Object
name
SOME_AUDITED_OBJECT:L_0
[NEW]
Foo
Some Audited Object
number
SOME_AUDITED_OBJECT:L_0
[NEW]
Powered by: Apache Isis™
About

How to Configure/Use

Classpath

Update your classpath by adding this dependency in your project's `dom` module's `pom.xml`:

```
<dependency>
    <groupId>org.isisaddons.module.audit</groupId>
    <artifactId>isis-module-audit-dom</artifactId>
</dependency>
```

Check for later releases by searching Maven Central Repo.

For instructions on how to use the latest `-SNAPSHOT`, see the contributors guide.

Bootstrapping

In the `AppManifest`, update its `getModules()` method, eg:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
            ...
            org.isisaddons.module.audit.AuditModule.class,
    );
}
```

Configuration Properties

For audit entries to be created when an object is changed, some configuration is required. This can be either on a case-by-case basis, or globally:

- by default no object is treated as being audited unless it has explicitly annotated using `@Audited`. This is the option used in the example app described above.
- alternatively, auditing can be globally enabled by adding a key to `isis.properties`:

  ```
  isis.services.audit.objects=all
  ```

An individual entity can then be explicitly excluded from being audited using `@Audited(disabled=true)`.

API

The `AuditerService` defines the following API:

```
@Programmatic
public void audit(
        final UUID transactionId,
        final int sequence,
        final String targetClass,
        final Bookmark target,
        final String memberIdentifier,
        final String propertyId,
        final String preValue,
        final String postValue,
        final String user,
        final java.sql.Timestamp timestamp);
```

Isis will automatically call this method on the service implementation if configured. The method is called often, once for every individual property of a domain object that is changed.

Implementation

The `AuditerService` SPI is implemented in this module by the `org.isisaddons.module.audit.AuditerServiceUsingJdo` class. This implementation simply persists an audit entry (`AuditEntry`) each time it is called. This results in a fine-grained audit trail.

The `AuditEntry` properties directly correspond to parameters of the `AuditerService audit()` API:

```
public class AuditEntry
    ...
    private UUID transactionId;
    private int sequence;
    private String targetClass;
    private String targetStr;
    private String memberIdentifier;
    private String propertyId;
    private String preValue;
    private String postValue;
    private String user;
    private Timestamp timestamp;
    ...
}
```

where:

- `transactionId` is a unique identifier (a GUID) of the transaction in which this audit entry was persisted.
- `timestamp` is the timestamp for the transaction
- `targetClass` holds the class of the audited object, eg `com.mycompany.myapp.Customer`
- `targetStr` stores a serialized form of the `Bookmark`, in other words a provides a mechanism to look up the audited object, eg `CUS:1234` to identify customer with id 1234. ("CUS" corresponds to the `@DomainObject(objectType=⋯)` annotation attribute).
- `memberIdentifier` is the fully-qualified class and property Id, similar to the way that Javadoc words, eg `com.mycompany.myapp.Customer#firstName`
- `propertyId` is the property identifier, eg `firstName`
- `preValue` holds a string representation of the property's value prior to it being changed. If the object has been created then it holds the value "[NEW]". If the string is too long, it will be truncated with ellipses '...'.
- `postValue` holds a string representation of the property's value after it was changed. If the object has been deleted then it holds the value "[DELETED]". If the string is too long, it will be truncated with ellipses '...'.

The combination of `transactionId`, `targetStr` and `propertyId` make up an alternative key to uniquely identify an audit entry. However, there is (deliberately) no uniqueness constraint to enforce this rule.

The `AuditEntry` entity is designed such that it can be rendered on an Isis user interface if required.

Supporting Services and Mixins

As well as the `AuditingService` service (that implements the `AuditingService3` API), the module also provides two further domain services:

- The `AuditingServiceMenu` provides actions to search for `AuditEntry`s, underneath an 'Activity' menu on the secondary menu bar.
- `AuditingServiceRepository` provides the ability to search for persisted (`AuditEntry`) audit entries. None of its actions are visible in the user interface (they are all `@Programmatic`) and so this service is automatically registered.
- `HasTransactionId_auditEntries` mixin contributes the `auditEntries` collection to the `HasTransactionId` interface. This will therefore display all audit entries that occurred in a given transaction, in other words whenever a command, a published event or another audit entry is displayed.

(As of 1.8.x and later) these services are automatically registered, meaning that any UI functionality they provide will appear in the user interface. If this is not required, then either use security permissions or write a vetoing subscriber on the event bus to hide this functionality, eg:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class HideIsisAddonsAuditingFunctionality extends AbstractSubscriber {
    @Programmatic @Subscribe
    public void on(final AuditingModule.ActionDomainEvent<?> event) { event.hide(); }
}
```

Known issues

In `1.6.0` through `1.9.x` a call to `DomainObjectContainer#flush()` is required in order that any newly created objects are populated.

Note that Apache Isis automatically performs a flush prior to any repository call, so in many cases there may not be any need to call flush explicitly.

Related Modules/Services

As well as defining the `AuditingService3` API, Isis' applib also defines several other closely related services. Implementations of these services are referenced by the Isis Add-ons website.

The `CommandContext` defines the `Command` class which provides request-scoped information about an action invocation. Commands can be thought of as being the cause of an action; they are created "before the fact". Some of the parameters passed to `AuditingService3` - such as `target`, `user`, and `timestamp` - correspond exactly to the `Command` class.

The `CommandService` service is an optional service that acts as a `Command` factory and allows `Command`s to be persisted. `CommandService`'s API introduces the concept of a `transactionId`; once again this is the same value as is passed to the `AuditingService3`.

The `PublishingService` is another optional service that allows an event to be published when either an object has changed or an actions has been invoked. There are some similarities between publishing to auditing; they both occur "after the fact". However the publishing service's primary use case is to enable inter-system co-ordination (in DDD terminology, between bounded contexts). As such, publishing is much coarser-grained than auditing, and not every change need be published. Publishing also uses the `transactionId`.

The `CommandService` and `PublishingService` are optional; as with the `AuditingService3`, Apache Isis will automatically use call each if the service implementation if discovered on the classpath.

If all these services are configured - such that commands, audit entries and published events are all persisted, then the `transactionId` that is common to all enables seamless navigation between each. (This is implemented through contributed actions/properties/collections; `AuditEntry` implements the `HasTransactionId` interface in Isis' applib, and it is this interface that each module has services that contribute to).

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/spi/audit/impl -D excludeTransitive=true
```

which, excluding Apache Isis itself, returns no direct compile/runtime dependencies.