zf-doctrine-audit Documentation

Release stable

Nov 02, 2020

Table of Contents

This repository creates an auditing database to track your target database changes.

Auditing is a complex subject. I've spent years trying to find an easier answer and by leveraging Object Relational Mapping through Doctrine what we have here is a plugin for new or existing Doctrine Zend Framework projects to implement auditing across a selection of entities or the whole database.

CHAPTER 1

Quickstart

The easiest way to see this repository work is by creating an auditing database with the unit tests and exploring it. See *Unit Testing*

CHAPTER 2

Getting Started

2.1 Installation

Installation of this module uses composer. For composer documentation, please refer to getcomposer.org.

```
$ composer require api-skeletons/zf-doctrine-audit
```

Once installed, add `ZF\Doctrine\Audit` to your list of modules inside `config/application.config.php` or `config/modules.config.php`.

2.1.1 zf-component-installer

If you use zf-component-installer, that plugin will install zf-doctrine-audit as a module for you.

2.2 Configuration

Copy `config/zf-doctrine-audit.global.php.dist` to your `config/autoload` directory and rename it to `zf-doctrine-audit.global.php`.

There are several configuration variables to customize.

2.2.1 target_object_manager

This is the service manager alias for the object manager to audit entities. The default `doctrine.entitymanager.orm_default` is the same as `doctrine/doctrine-orm-module` default.

2.2.2 audit_object_manager

This is the service manager alias for the object manager for the audit database. You will need to add this object manager to your ORM project. See multiple object managers for help.

2.2.3 audit_table_name_prefix & audit_table_name_suffix

These configuration variables allow you to add a prefix or suffix to the generated tables in the audit database.

2.2.4 epoch_import_limit

When running the epoch tool data will be processed in batches to conserve memory. This is the number to process at a time.

2.2.5 entities

This associative array of entity names inside target_object_manager will be audited and an audit table will be created for each. This array takes the format:

```
'entities' => [
  'Db\Entity\User' => [],
]
```

The empty array for the entity name is reserved for future development. It may be used to store route information for canonical paths.

2.2.6 joinEntities

The joinEntities array is a list of pseudo entity names representing a many to many join across entities.

The format is the namespace of the owner entity followed by the table name (in most cases) to represent the entity. The ownerEntity is required as is the tableName. This information is used to find the join mapping in the metadata.

Example:

```
'joinEntities' => [
    'Db\Entity\ArtistToArtistGroup' => [
        'ownerEntity' => 'Db\Entity\ArtistGroup',
        'tableName' => 'ArtistToArtistGroup',
    ],
],
```

2.3 Audit Object Manager

You must setup a second object manager within your ORM application. See multiple object managers for help.

CHAPTER 3

Create Audit Database & Triggers

This walkthrough creates a new audit database. For changes to a target database see *Change Management*.

3.1 Create Database

With your application configured with a new entity manager for the audit database and your configuration containing the entities you want to audit, it's time to create your audit database:

```
php public/index.php orm:schema-tool:create --object-manager=doctrine.entitymanager.
↪orm_zf_doctrine_audit
```

This command will create the database for the given object manager. Be sure you specify the same object manager as the configuration `audit_object_manager`.

A note about migrations: Currently Doctrine doesn't have a way to run migrations for two databases. It makes sense to have another set of migrations for the audit database. This repository does not try to solve this problem and leaves the use of migrations up to you.

3.2 Run Fixtures

The audit database requires data, fixtures, to operate. To populate this data run this command:

```
php public/index.php data-fixture:import zf-doctrine-audit
```

The audit database has now been created.

3.3 Run Triggers

The next step is to run the generated triggers on your target database. This is not done directly on the database through code but instead the trigger code is output by the tool. We will be piping this directly to the target database:

```
php public/index.php audit:trigger-tool:create
```

Then pipe this output to the target database such as:

```
php public/index.php audit:trigger-tool:create | mysql -u user -p123 -h mysql target_
↪database
```

Any triggers with the same names will be removed. This allows you to re-run the trigger sql.

3.4 Drop Triggers

There is a tool for removing the triggers and functions created by the trigger-tool:

```
php public/index.php audit:trigger-tool:drop
```

Use this only if you need to adjust your audited entities then re-run the *:create* tool.

3.5 Auditing is Working

At this point if you connect to your target database and add a new record to a table which is audited through its entity the audit log will show up inside the audit database.

If your target database already has data in it you probably want to explore *Epoch*.

CHAPTER 4

Epoch

When this repository is first applied to an existing project there will probably already be data in the database which you want to start auditing. Creating a epoch record for each row will give auditing a reasonable starting point.

Like the trigger tool, creating an epoch should be piped to the database, but for epoch **you must pipe to the audit database**:

```
php public/index.php audit:epoch:import
```

Piping this output looks like:

```
php public/index.php audit:epoch:import | mysql -u user -p123 -h mysql audit_database
```

The epoch tool uses the configuration variable `epoch_import_limit`. This variable will paginate the epoch audit record creation. The default of 200 is acceptable.

Currently the epoch tool will create an epoch record for every row in the database in the entities configured to be audited.

**TODO: The epoch tool SHOULD only create an epoch record for rows which do not already have an epoch record. **

CHAPTER 5

How Auditing Works

Triggers are created for every database table for every entity and joinEntity configured for zf-doctrine-audit. Using triggers allows database modifications to be made anytime a change is made to the database whether through the ORM or through a console connection, etc. These triggers are created on the target database.

When a change happens on the target database to a record configured to be audited an audit record is created. Each target table has a RevisionEntity table in the audit database which is a copy of the target table with a new field added called `revisionEntity_id`. This is a foreign key to the RevisionEntity_Audit table. The RevisionEntity_Audit table works like a receipt for each set of changes to a row of audited data.

5.1 AuditEntity

This refers generically to any entity audited by this tool. The namespace for any entity audited is `ZF\Doctrine\Audit\AuditEntity`. When an audit occurs the data for the audited entity is stored in its corresponding AuditEntity.

5.2 RevisionEntity Entity

This table is a reciept for the change audited in the audit database. The RevisionEntity_Audit table is used by the `ZF\Doctrine\Audit\Entity\RevisionEntity` entity. This entity has a relationship to `ZF\Doctrine\Audit\Entity\RevisionType` which defines the type of audit which took place, whether insert, update, delete, or epoch.

Another relationship to `ZF\Doctrine\Audit\Entity\TargetEntity` defines which entity was acted upon. The `TargetEntity` contains dynamic data populated by data-fixture. This `TargetEntity` has a relationship to the `ZF\Doctrine\Audit\Entity\AuditEntity` which has information about the auditing entity.

Finally the RevisionAudit has a relationship to the `ZF\Doctrine\Audit\Entity\Revision` entity. This `Revision` entity groups `RevisionEntity` records together into ORM `flush();` operations. So if your object manager has three entities to persist or update when `flush();` is called there will be one `RevisionEntity`

record for each managed entity and one `Revision` entity. This design allows groups of related database changes to be saved via the audit record.

5.3 Revision Entity

Automatically populated when created, the `createdAt` field stores the time the `Revision` was created. This is the timestamp for a complete audit record.

There are several other fields which can be populated to help track who made a change and why the chane was made. `comment`, `userId`, `userName`, and `userEmail` can be set through the *Revision Comment* when working with the database through the ORM in PHP. These values will default to empty with a userName of 'not orm' when making changes to the database outside of PHP such as through a terminal connection.

CHAPTER 6

Audit Plugin

This module uses API-Skeletons/zf-doctrine-repository to provide a plugin in Doctrine repositories to run common audit queries.

This plugin will be extended in the future. Currently these functions are supported

6.1 getRevisionEntityCollection($entity)

This will return the complete audit history for the passed entity.

6.2 getOldestRevisionEntity($entity)

One of the most common fields in databases is createdAt (or date_created, etc). With auditing this field is unnecessary in the target database and can be derived by inspecting the oldest audit record for an entity.

By fetching the oldest revision entity you can get the createdAt with:

```
$oldestRevisionEntity = $this->plugin('audit')->getOldestRevisionEntity($entity);
$createdAt = $oldestRevisionEntity->getRevisionEntity()->getRevision()->
↪getCreatedAt();
```

6.3 getNewestRevisionEntity($entity)

To find the latest information about an entity, such as when it was last edited, fetch the newest revision entity.

CHAPTER 7

Revision Comment

In order to save which user made which change inside the ORM the `ZF\Doctrine\Audit\RevisionComment` class exists. This class is managed by the service manager so only one copy exists. Before you call a `flush();` to make changes to ORM data you may populate the `RevisionComment` to save additional information.

7.1 Fetching RevisionComment

When you wish to use the `RevisionComment` inject it via a factory:

```
use ZF\Doctrine\Audit\RevisionComment;

$instance->setRevisionComment($container->get(RevisionComment::class);
```

There is a trait and interface you may include located in the `persistence` directory for setting and getting the `RevisionComment`.

7.2 Using RevisionComment

Before you `flush();` your object manager set the values on the `RevisionComment`. The RevisionComment will be cleared after `flush();` The value you may set is `comment`.

7.3 Custom Identity for Revision Auditing

The `Revision` entity has space for `userId`, `userName`, and `userEmail`. These are populated based on the authenticated user. See `src/EventListener/PostFlush.php`. `ZF\OAuth2\Doctrine\Identity\AuthenticatedIdentity` is handled natively as is `ZF\MvcAuth\Identity\AuthenticatedIdentity` but if you're not using these identity strategies you'll have to write your own PostFlush handler to update the `Revision` and override it in the service manager using the key `ZF\Doctrine\Audit\EventListener\PostFlush`.

CHAPTER 8

Change Management

Database schemas change over time. Auditing has tools to keep your changes in-line with both databases.

8.1 Adding a new field or audited entity

Your first step is to make the change to the entity and consequently the column for the new field. These instructions do not describe change management on your target database as that is outside the scope of auditing.

8.1.1 Update the audit database with the new field

Run the orm schema tool:

```
php public/index.php orm:schema-tool:update --object-manager=doctrine.entitymanager.
↪orm_default
```

This command requires a –force or –dump-sql parameter too. This part of the process is no different than managing your target database with the schema tool.

8.1.2 Update the audit database fixtures

After running the schema tool and changing the database you must re-run the fixtures for the audit database. These fixtures are smart and will not produce duplicate data:

```
php public/index.php data-fixture:import zf-doctrine-audit
```

8.1.3 Update the target database triggers

The triggers generated by zf-doctrine-audit drop any existing triggers. This allows us to re-run the trigger tool to generate updated sql. The next step is to run the generated triggers on your target database. This is not done directly

on the database through code but instead the trigger code is output by the tool. We will be piping this directly to the target database:

```
php public/index.php audit:trigger-tool:create
```

Then pipe this output to the target database such as:

```
php public/index.php audit:trigger-tool:create | mysql -u user -p123 -h mysql target_
↪database
```

Your audit database is now up to date with your target database.

8.2 Removing a field from an entity

zf-doctrine-audit does not try to archive data for fields which have been removed from the target database. If you want to preserve data for a field or entity which you are removing from the target database you must dump that data and preserve it yourself.

After you have saved the old audit data you may follow the same steps above for Adding a new field or audited entity.

CHAPTER 9

Unit Testing

Because zf-doctrine-audit uses stored procedures sqlite isn't enough for unit testing. To setup unit testing follow these steps:

1. Clone the repository outside of a project and cd into the repository directory.
2. Run `docker-compose up -d` to start a container to work in. You may need to install Docker if you're not already using it.
3. Run `docker/connect` to connect to a shell inside the container.
4. Run `composer install` to install required libraries.
5. Run `vendor/bin/phpunit` to execute the unit tests.

After the unit tests have ran the database will still exist. Connect to mysql with `mysql -u root -h mysql test` to explore the database. The audit database has the audit trails for the test data. Manipulating data in the test database is immediatly audited in the audit database.

CHAPTER 10

Internals

This document describes how auditing works in the code, internal to the application.

There are four primary actions which zf-doctrine-audit implements. The first is building an audit database based on the configuration listing entities and joinEntities.

10.1 Audit Entity Autoloader

In order to map configured entities to a database we must have a class for each audited entity. The naming of these classes follows this pattern (found in `ZF\Doctrine\Audit\Repository\AuditEntityRepository`):

```
return "ZF\\Doctrine\\Audit\\RevisionEntity\\" . str_replace('\\', '_', $entityName);
```

So an entity in the target object manager named `Db\Entity\User` will be audited by an entity in the audit object manager named `ZF\\Doctrine\\Audit\\RevisionEntity\\Db_Entity_User`. Access to this entity through the audit object manager works as you would expect in a doctrine object manager:

```
$auditObjectManger->getRepository('ZF\\Doctrine\\Audit\\RevisionEntity\\Db_Entity_User
↪')
    ->findBy([
        'id' => 2,
    ]);
```

This code will fetch the complete audit history for the `Db\Entity\User` entity with id = 2.

This is possible because audit entity classes are dynamically created via an Autoloader.

10.2 Audit Object Manager Metadata

Mapping drivers for configured `entities` and `joinEntities` dynamically create metadata based on target entity metadata. By introspecting the existing target entity metadata a new metadata definition can be assigned to an Autoloader created class.

The dynamically created classes assigned to dynamically created metadata combined with static auditing entities creates a complete audit object manager in the application. This object manager can be used by the schema tool and that is how the audit database is created.

10.3 Trigger Tool

Currently zf-doctrine-audit only supports MySQL. However the code is written such that new databases may be supported. The SQL for the target database is generated by the Trigger Tool. This tool implements two functions:

```
get_revision_entity_audit
close_revision_audit
```

10.3.1 get_revision_entity_audit function

`get_revision_entity_audit` will create a new record in the RevisionEntity_Audit table and possibly the Revision_Audit table. When `get_revision_entity_audit` is called a check is done if a Revision_Audit record exists with the current `connectionId`. If found the `id` for that record is used. This allows multiple sql calls to be grouped into a single Revision.

Grouping of multiple sql calls into a single Revision only happens when running through the Object Manager. When changes are made to the target database through another means each Revision has only one RevisionEntity.

`get_revision_entity_audit` returns an id for a newly created RevisionEntity_Audit record. This id is assigned to the auditing table audit row.

10.3.2 close_revision_audit function

When running through an ORM this function is called `postFlush`. The *Revision Comment* comment is passed to the function along with the authenticated user. This changes the connectionId back to null. While the connectionId is not null it is used to group RevisionEntity together.

When running from a command line or attaching to the database is any other way than through the ORM this function is called at the end of each table create, update, and delete trigger.

10.3.3 create, update, delete triggers

Every table which is audited is assinged three triggers, one each for create, update, and delete. These triggers copy the audited row data into the audit database, assign a RevisionEntity, and either call `close_revision_audit` or not.

The create trigger populates the audit table with the new data. The update trigger populates the audit table with the new, updated, data. The delete trigger populates the audit table with the last data for the row. In other words, the last update audit data will match the delete audit data.

10.4 Epoch Tool

This tool creates temporary stored procedures, runs them, then deletes them. This method is the fastest way to copy data from the target database into the audit database. The stored procedures output status information while they work.

10.5 RevisionAudit Tool

This tool is responsible for closing the Revision_Audit table after changes have been made. This tool can be called on its own. This tool is called in *postFlush()*.