ZanuiFixturesBundle

Release 2.0.1

October 15, 2014

Contents

This bundle defines abstract fixture classes to load test data into a database. It complements the DoctrineFixturesBundle by extending and implementing classes from the Doctrine2 Data Fixtures library to ease the pain of loading data fixtures programmatically into the Doctrine ORM. Please read the documentation for that bundle first to get familiar with the basic concepts that we will build upon.

CHAPTER 1

Guide

1.1 Installation

Note: ZanuiFixturesBundle follows the semver specification

To install, add the following to your `composer.json` file:

```
{
    "require": {
        "zanui/zanui-fixtures-bundle": "~2.0"
    }
}
```

Update the vendor libraries:

```
$ php composer.phar update zanui/zanui-fixtures-bundle
```

Finally, register the `DoctrineFixturesBundle` and the `ZanuiFixturesBundle` in `app/AppKernel.php`.

```
// ...
public function registerBundles()
{
    $bundles = array(
        // ...
        new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
        new Zanui\FixturesBundle\ZanuiFixturesBundle(),
        // ...
    );
    // ...
}
```

1.2 Configuration

The **ZanuiFixturesBundle** can be used without any configuration, but there are two convenience fallback parameters that you can define in your `config.yaml` to facilitate the creation of fixtures:

`entity_namespace_fallback` (string) Defines a namespace to load entities from when a `namespace` property is not explicitly declared in the fixture class. If all (or most) of your entities belong to a common namespace, adding that namespace here will save you from having to add it in every fixture class.

base_order_fallback (integer, defaults to 1) Defines a base order for loading fixtures when an `order` property is not explicitly declared in the fixture class.

Here is a typical configuration:

```
# app/config/config.yml

zanui_fixtures:
  entity_namespace_fallback: 'Acme\HelloBundle\Entity'
  base_order_fallback: 100
```

1.3 Creating a simple fixture class

Let's walk through it with the same example as in the DoctrineFixtureBundle documentation. Imagine that you have a `User` class, and you would like to load a couple of `User` entries.

First, we would create a YAML file with the necessary information:

```
# src/Acme/HelloBundle/DataFixtures/ORM/Data/User.yml

options:
  add_reference: true
data:
  admin:
    username: admin
    password: admin
  test:
    username: test
    password: test
```

Note: we will explain what `options` *are available and what they do later on.*

Then, we will need a fixture class to load the information:

```
<?php
// src/Acme/HelloBundle/DataFixtures/ORM/LoadUserData.php

namespace Acme\HelloBundle\DataFixtures\ORM;

class LoadUserData extends AcmeHelloOrmFixture
{
    protected $name = 'User';
}
```

That's it! Well, not so fast. Notice `LoadUserData` is extending `AcmeHelloOrmFixture`, which we have not written yet. Luckily, it is also quite simple and we only need one like this per bundle. Here it is:

```
<?php
// src/Acme/HelloBundle/DataFixtures/ORM/AcmeHelloOrmFixture.php

namespace Acme\HelloBundle\DataFixtures\ORM;

use Zanui\FixturesBundle\DataFixtures\ZanuiOrmFixture;

abstract class AcmeHelloOrmFixture extends ZanuiOrmFixture
{
    protected $baseDir = __DIR__;
}
```

And now that really is it!

Of course, you could choose to add the `baseDir` property on the loading classes and extend them directly from `ZanuiOrmFixture`, but if you have a lot of classes this is the preferred way to go. Anyway, this is how the `LoadUserData` would look like in that case:

```
<?php
// src/Acme/HelloBundle/DataFixtures/ORM/LoadUserData.php

namespace Acme\HelloBundle\DataFixtures\ORM;

use Zanui\FixturesBundle\DataFixtures\ZanuiOrmFixture;

class LoadUserData extends ZanuiOrmFixture
{
    protected $baseDir = __DIR__;
    protected $name = 'User';
}
```

Notice that without `AcmeHelloOrmFixture`, we would need to add the `use` statement and the `baseDir` property to all fixture classes.

You might feel like there is still something missing: how is `ZanuiOrmFixture` calling the setter methods for my `User` entity, or even creating the entity in the first place? You are right to feel that way, but everything works because we followed a specific directory structure and stuck to a few naming conventions.

1.4 Directory structure and naming conventions

The `AcmeHelloOrmFixture` sets the `baseDir` to `__DIR__`, which will make `ZanuiOrmFixture` look for YAML files inside `__DIR__/Data/` with the filename matching that of the fixture. The name of the class `LoadUserData` can be anything, as long as its `name` property matches an existing file inside the `Data` directory. The `name` also needs to match that of the entity class.

```
Acme/
`-- HelloBundle/
    `-- DataFixtures/
        `-- ORM/
            |-- Data/
            |   |-- User.yml
            |   `-- (other data files)
            |-- AcmeHelloOrmFixture.php
            |-- LoadUserData.php
            `-- (other fixture classes)
```

The names of the entity fields inside the YAML file also need to follow a convention, as the the bundle uses it to infer the setter method to call in order to set their value:

- To have a setter method called `setUsername` invoked, the field in the YAML file needs to be called `username` or `Username`
- If the setter method was called `setUserName`, then the field would need to be called `user_name` or `UserName`.

You get the idea.

The `ZanuiFixture` class has a property `namespace` that falls back to the `entity_namespace_fallback` parameter. If the `User` entity class did not belong to that namespace, or `entity_namespace_fallback` was not declared in the bundle's configuration, we would need to add the correct namespace for the `LoadUserData` class:

```php
<?php
// src/Acme/HelloBundle/DataFixtures/ORM/LoadUserData.php

namespace Acme\HelloBundle\DataFixtures\ORM;

class LoadUserData extends AcmeHelloOrmFixture
{
    protected $namespace = 'Acme\OtherBundle\Entity';
    protected $name = 'User';
}
```

1.5 Controlling the loading order

When we want to create a new fixture that depends on other fixtures, we will need to make sure that it is loaded after all its dependencies. To do that, we simply need to add an `order` property to the class and make its value higher than that of all its dependencies.

We will also need to link the entity to its dependencies (foreign keys) in the YAML file. We do that by setting the value of the foreign key to be the key of the entity it depends on. Take the following example, in which we add a `Group` entity...:

```yaml
# src/Acme/HelloBundle/DataFixtures/ORM/Data/Group.yml

options:
  add_reference: true
data:
  admin:
    group_name: admin
```

```php
<?php
// src/Acme/HelloBundle/DataFixtures/ORM/LoadGroup.php

namespace Acme\HelloBundle\DataFixtures\ORM;

class LoadGroup extends AcmeHelloOrmFixture
{
    protected $name = 'Group';
}
```

... and a `UserGroup` entity to assign a `User` to a `Group`:

```yaml
# src/Acme/HelloBundle/DataFixtures/ORM/Data/UserGroup.yml

options:
  foreign_keys:
    - user
    - group
data:
  -
    user: User-admin
    group: Group-admin
```

```php
<?php
// src/Acme/HelloBundle/DataFixtures/ORM/LoadUserGroup.php

namespace Acme\HelloBundle\DataFixtures\ORM;
```

```
class LoadUserGroup extends AcmeHelloOrmFixture
{
    protected $name = 'UserGroup';
    protected $order = 200;
}
```

Notice that we referred to the admin user by making `user` have the value `User-admin`, in which the first part is the entity it refers to and the second part was the key for the admin user as defined in its YAML file (same applies for `group`). Also notice that the order is set to 200 to make sure `User` and `Group` are already loaded when we process `UserGroup`.

The loader knows that those values are foreign keys because we explicitly listed them using the `foreign_keys` option. Any fields that start with `fk_` (case insensitive) are automatically inferred to be foreign keys, so they don't need to be included in the list. The references exist because we added the option `add_reference: true` to our `User.yml` and `Group.yml` files.

We have just gone through examples that required the use of options, so let's jump straight into that topic and describe all available options.

1.6 Available data options and fixture properties

1.6.1 Data options

`flush_preserving_ids` (boolean, defaults to `false`) Indicates whether the entities should be saved overriding the default ID generation strategy to preserve the given IDs. This is only necessary if in some parts of your applications you have assumed that some entities have a certain ID (*eg.* to simplify queries).

`flush_on_every_row` (boolean, defaults to `false`) Indicates whether the entity should be flushed on every row instead of only at the end, *eg.* you depend on the ID of a previous row, like in a parent-child relationship.

`add_reference` (boolean, defaults to `false`) Indicates whether to set a reference for the current entity. Only necessary if the entity will act as a foreign key for other entities.

`foreign_keys` (array) Defines a list of fields that should be treated as foreign keys, *ie.* their values point to a previously saved reference. Fields that start with `fk_` (case insensitive) are assumed to be foreign keys, so they do not need to be listed.

`date_time_fields` (array) Defines a list of fields which values should be transformed to `DateTime`, *eg.* a value of `2000-01-01` would be passed to the setter as `\DateTime('2000-01-01')`.

`local_references` (array, only for fixtures using a"ZanuiCustomLoader") Similar to `foreign_keys`, but in this case the references point to entities saved within the same loader. They are especially useful when writing custom loaders.

1.6.2 Fixture properties

We have mentioned all of the following properties in previous sections, but here is a definition for relevant properties for fixture classes extending the `ZanuiOrmFixture` or `ZanuiCustomLoader` class:

`baseDir` Defines the base directory where data will be loaded from. Typically it will simply be `__DIR__`. As shown above, it is usually a good idea to set in your own base class and extend the rest of the fixture classes from it.

name Defines the name of the fixture. In the case of fixtures extending `ZanuiOrmFixture`, it must match the name of the YAML file where the data is stored to work out of the box. In the case of `ZanuiCustomLoader`, it must match the directory name in which the YAML files are stored.

namespace (only relevant for `ZanuiOrmFixture`) Defines the namespace to use in order to load the entity being loaded. It falls back to the `entity_namespace_fallback` parameter described above.

order Defines the order in which the fixture should be loaded. Fixtures with higher `order` will be loaded after fixtures with lower `order`. It falls back to the `base_order_fallback` parameter described above.

1.7 Advanced usage

Following the directory structure and naming conventions is recommended but not required. You may decide to extend any of the classes included in this bundle to change the default behaviour.

For example, you may want to override the `load(...)` and `loadInfo(...)` methods of the `ZanuiOrmFixture` class to follow your own conventions. You may even use the conventions in this bundle for some fixtures and extend directly from `AbstractFixture` of the Doctrine2 Data Fixtures library for others.

CHAPTER 2

Cookbooks

2.1 Writing custom loaders

This bundle also provides a `ZanuiCustomLoader` class to help create classes that load data into several (usually related) entities. Imagine we want to load data about a `team` and its `members`. With a custom loader, we can define a YAML file per team and define all relevant data within that same file, instead of having teams and members split into several YAML files. This is how the YAML file would look like:

```
# src/Acme/HelloBundle/DataFixtures/Teams/a-team.yml

data:
  team:
    options:
      add_reference: true
    data:
      -
        name: A-team
        motto: If you can find them... maybe you can hire... The A-Team.

  member:
    options:
      local_references:
        - team
    data:
      -
        team: team-0
        name: Hannibal
      -
        team: team-0
        name: Murdock
```

And here is your custom loader, which extends `ZanuiCustomLoader`:

```
<?php
// src/Acme/HelloBundle/DataFixtures/TeamLoader.php

namespace Acme\HelloBundle\DataFixtures;

use Doctrine\Common\Persistence\ObjectManager;
use Zanui\FixturesBundle\DataFixtures\ZanuiCustomLoader;

class TeamLoader extends ZanuiCustomLoader
{
```

```
    protected $name = 'Teams';
    protected $order = 1000;
    protected $baseDir = __DIR__;

    public function load(ObjectManager $manager)
    {
        $this->manager = $manager;
        $this->info = $this->loadInfo();

        foreach ($this->info as $current) {
            $this->current = $current;
            $this->referenceUniqueSuffix = $this->generateUniqueSuffix();

            $this->loadCustomEntity('Acme\HelloBundle\Entity\Team', 'team');
            $this->loadCustomEntity('Acme\HelloBundle\Entity\Member', 'member');
        }

        $manager->flush();
    }
}
```

The custom loader will load all files under `src/Acme/HelloBundle/DataFixtures/Teams/` (the `name` property of the loader needs to match that of the directory), so next to `a-team.yml` you could add other team files and they would be processed automatically.

References in custom loaders are saved with a unique ID to avoid collisions, so they cannot be used outside the custom loader. Local references have to be explicitly declared using the `local_references` option. Notice how for each team member we refer to their team as `team-0`, as their team is the first one defined in the file. Although several teams could be defined within the same file, it is recommended to divide them into separate files.