VoltelExtraFoundryBundle Documentation

Volodymyr Telnov

Dec 08, 2021

CONTENTS

CHAPTER

ONE

A WRAPPER FOR `ZENSTRUCK/FOUNDRY` WITH EXTRA SPEED PERSISTING THOUSANDS OF ENTITIES

1.1 Contents

1.1.1 Seeding the development database with fixture entities

The recommended way to seed the database with sample data while developing is to use DoctrineFixturesBundle. To install Doctrine Fixture bundle, follow the instructions given on DoctrineFixturesBundle page of Symfony docs.

Steps to follow to seed the database

In general, the steps required to seed the development database with entities aren't much different with *VoltelExtraFoundryBundle* from when you use vanilla `zenstruck/foundry` bundle.

The difference is mostly with the implementation of stories (classes extending from `Zenstruck\Foundry\Story`). *VoltelExtraFoundryBundle* offers a `EntityProxyPersistService` (described *in this chapter*) and `SetUpFixtureEntityService` (described in "*How to set up custom entities in a test*" section) which make populating a database easy to code, clear to read and fast to execute.

1. First, create your Doctrine entities.
2. Then, for each entity, create a corresponding custom factory class extending `Zenstruck\Foundry\ModelFactory` as described in Model Factories.

   For reasons *described elsewhere*, it is recommended to extend `Voltel\ExtraFoundryBundle\Foundry\Factory\AbstractFactory` class which, in its turn, extends `Zenstruck\Foundry\ModelFactory` class. But it's not a big deal if you don't.
3. Use your custom factories inside your custom story classes as described in Stories.

   For example, inside the `CustomerStory` class, you can create a batch of customers (`Customer` entities) using `CustomerFactory` and, for each customer, create one or several addresses (`Address` entities) using `AddressFactory`.

   To get automatic access to the bundle's `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService` service, extend your story class from `Voltel\ExtraFoundryBundle\Foundry\Story\AbstractStory` class.

   Otherwise, use usual Symfony's dependency injection tricks to get access to the persist service inside your story class extending the `Zenstruck\Foundry\Story` class (e.g. type-hint `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService` in the `__construct` method of your story class).
4. Use your custom stories inside Doctrine Fixture classes.

So, your custom Fixture class extending from `Doctrine\Bundle\FixturesBundle\Fixture` may look like this:

```
class DataFixture extends Fixture
{
    public function load(ObjectManager $manager)
    {
          ProductStory::load();
          // see example of CustomerStory class below
          CustomerStory::load();
          OrderStory::load();
    }
}
```

For other examples of Using with DoctrineFixtureBundle read `zenstruck/foundry` docs.

Create entities with delayed persist/flush

Create separate entities with delayed persist/flush

Use `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService::createOne()` method to create a single entity.

Pass `Zenstruck\Foundry\Factory` object as the first parameter. In the second optional parameter, you can pass either an array of attributes for the factory, or a callback function to return an array of attributes as described in Attributes section of `zenstruck/foundry` docs.

The last (third) parameter is optional as well. It may accept an array with names of the factory states (i.e. names of the factory class methods, as described in Reusable Model Factory "States").

Note: States in the final parameter can be presented as just state names for methods that do not require arguments, or an associative array elements where an array key is the state/method name and an array value is the method argument(s).

If a state method has several arguments, they should be listed in an array (see "state three" in the example below). If a state method has only one argument, it can be presented w/o a wrapping array (see "state_four" in the example below):

```
// array representing states for "createOne()" and "createMany()" methods
[
    'state_one',
    'state_two',
    'state_three' => ['argument_1', 'argument_2'],
    'state_four' => 10,
]
```

```
// in CustomerStory.php

use Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService;
use Zenstruck\Foundry\Story;

class CustomerStory extends Story
{
    private $persistService;
```

(continues on next page)

(continued from previous page)

```
    /**
     * If your story class extends
     * "Voltel\ExtraFoundryBundle\Foundry\Story\AbstractStory"
     * the "EntityProxyPersistService" will be injected automatically
     * with standard Symfony configuration
     */
    public function __construct(EntityProxyPersistService $persistService)
    {
        $this->persistService = $persistService;
    }


    public function build(): void
    {
        $this->createCustomer();
    }


    private function createCustomer()
    {
        // Create a factory that won't immediately persist
        $customer_factory = CustomerFactory::new()->withoutPersisting();
        //
        // Although not explicitly used here, "withoutPersisting()"
        // will be automatically invoked for this factory
        // in persistService before creating entities
        $address_factory = AddressFactory::new();

        for ($i = 0; $i < 20; $i++) {
            $n_address_count = rand(1, 3);

            $this->createCustomerWithAddresses($customer_factory, $address_factory, $n_
↪address_count);
        }

        // Persist and flush new entities as a batch operation.
        // This takes less time than persisting/flushing each entity
        $this->persistService->persistAndFlushAll();
    }


    private function createCustomerWithAddresses(
        CustomerFactory $customer_factory,
        AddressFactory $address_factory,
        int $n_address_count = 1
    )
    {
        /** @var Customer $customer_entity */
        $customer_entity = $this->persistService->createOne($customer_factory);

        for ($i = 0; $i < $n_address_count; $i++) {
```

(continues on next page)

(continued from previous page)

```
            /** @var Address $address_entity */
            $address_entity = $this->persistService->createOne($address_factory);

            $address_entity->setCustomer($customer_entity);
            //
            // or use a specialized collection manipulation method
            // $customer_entity->addElementToAddressCollection($address_entity);

        }
    }

}
```

The `EntityProxyPersistService:createOne()` method is responsible for several things:

- Applying some custom factory states to the factory stub, if provided. An array of state names is optional and can be passed as the third (last) argument.
- Factory will be cloned to avoid immediate persistence, i.e. `Zenstruck\Foundry\Factory::withoutPersisting()` method is going to be invoked.
- An array of optional arguments for new entities, if provided, will be directly passed as an argument to `Zenstruck\Foundry\Factory::create()` method.
- A new `Zenstruck\Foundry\Proxy` object returned by `Factory:create()` will be internally put in a "proxy jar" to be later persisted by corresponding entity manager. Persisting entities in batches speeds up the whole process.

Create batches of entities with delayed persist/flush

Use `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService::createMany()` method to create several entities at once. Similar to `EntityProxyPersistService::createOne()`, pass the factory stub as the first argument and the number of entities to create as the second argument.

This method may conveniently be modified with the third parameter, that can accept either an array of attributes or, more importantly, a callback that returns an array of attributes. This enables random values for every of the created entities.

You can find examples of using a callback with `createMany()` in several sections of the `zenstruck/foundry` docs, e.g. in Using with DoctrineFixtureBundle and Many-To-One sections. Here is an example from test suite of this bundle:

```
// in OrderStory.php

private function createOrderItemsForOrder(Order $order_entity)
{
    $factory_order_item = OrderItemFactory::new()
        //->withoutPersisting()
        ->forOrder($order_entity);

    // This will create a batch of OrderItem entities,
    // each with its unique "unitCount" value
    //
    $this->persistService->createMany($factory_order_item, rand(1, 4), function() {
        return [
```

(continues on next page)

(continued from previous page)

```
            'unitCount' => rand(1, 20)
        ];
    });
}
```

I refer to the first argument as a "factory stub" because that the factory can be further modified by passing the fourth (last) argument – an array of state names (states) or a callback returning such an array.

States are just method names in the model factory class that will be invoked on the factory with the result that the factory will be cloned with new attributes as described in Reusable Model Factory "States".

Setting relationships between entities

The `zenstruck/foundry` library makes it easy to create related entities of `@ORM\ManyToOne` associations right from inside the factory class (i.e. by providing a factory for a particular entity property, as described under *TIP 2* in Many-To-One section).

```
// in OrderItemFactory.php

protected function getDefaults(): array
{
    $faker = self::faker();

    return [
        'notes' => $faker->realText(),

        // To randomly assign a product for this order item.
        // Products must be seeded in the database
        // before orders and order items.
        'product' => ProductFactory::repository()->random(),
    ];
}
```

The same is possible for Many-To-Many relationship.

I prefer to have this logic outlined inside a story, where related entities are created and referenced explicitly:

```
// in ProductStory.php

private function createProduct(ProductFactory $factory, int $n_entity_count = 20)
{
    $repo_category = CategoryFactory::repository();

    for ($i = 0; $i < $n_entity_count; $i++) {
        /** @var Product $product */
        $product = $this->persistService->createOne($factory);

        // Add from 1 to 3 categories to each Product
        // "Product" and "Category" have a Many-To-Many relationship
        $a_category_proxies = $repo_category->randomRange(1, 3);

        foreach ($a_category_proxies as $oThisCategoryProxy) {
            /** @var Category $oThisCategoryEntity */
```

(continues on next page)

(continued from previous page)

```
            $oThisCategoryEntity = $oThisCategoryProxy->object();

            $product->addElementToCategoryCollection($oThisCategoryEntity);
        }
    }
}
```

The example above could be rewritten to be more succinct, as described in Many-To-Many section:

```
// in ProductStory.php

private function createProduct(ProductFactory $factory, int $n_entity_count = 20)
{
    $repo_category = CategoryFactory::repository();

    $this->persistService->createMany($factory, $n_entity_count, function() use ($repo_
↪category) {
        return [
            'categoryCollection' => $repo_category->randomRange(1, 3),
        ];
    });

}
```

Note: In the code example above, if there is no “setter” for “categoryCollection” property, the factory should use a custom instantiator to “force-set” it.

This can only be a solution for **unidirectional** associations like in this example, where `Product` holds a unidirectional @ORM\Many-To-Many association with `Category` entities.

For **bidirectional** associations, you will most likely need a more sophisticated setter that will establish the opposite side of the relationship. For example, one `Customer` can be related to many `Address` entities, and each `Address` entity is related to one `Customer` (bidirectional @ORM\One-To-Many associations). In this case, the setter for `Customer::addressCollection` property could look like this:

```
// in Customer.php

/**
 * @param Address[]|null $addresses
 * @return Customer
 */
public function setAddressCollection(?array $addresses): Customer
{
    $this->addressCollection = new ArrayCollection($addresses);
    foreach ($addresses as $address) {
        $address->setCustomer($this);
    }
    return $this;
}
```

Alternatively, you could use a specialized collection manipulation method similar to the `addElementToCategoryCollection()` method (usage is shown in the *example above*):

```
// in Customer.php

public function addElementToAddressCollection(Address $address)
{
    if ($this->addressCollection->contains($address)) return;
    $this->addressCollection->add($address);
    $address->setCustomer($this);
}
```

Using factory states to populate arrays and establish relationships between entities

While Reusable Model Factory "States" is a great way to set model attributes in a more explicit way in terms of readability, with `zenstruck/foundry` it is not yet possible to manipulate array values, particularly, to add individual values to arrays using states.

If you extended your factory class from `Voltel\ExtraFoundryBundle\Foundry\Factory\AbstractFactory` class, you will have a `AbstractFactory::addValuesTo()` method at your disposal. This method can be used to do exactly what it says: add values to an array stored in a custom model attribute.

Let's see an example (it can be found in a `Voltel\ExtraFoundryBundle\Tests\Setup\Factory\ProductFactory` class):

```
// in ProductFactory.php

public function car(): self
{
    return $this->addState([
        // Note: this will add two values to existing values of "categories" attribute
        'categories' => $this->addValuesTo('categories', ['car', 'vehicle']),
    ]);
}
```

As `ModelFactory::addState()` method will create a clone of current factory, normally, the state that modifies some attribute will override all previous values, a behavior that is not sometimes desirable. So, `AbstractFactory::addValuesTo()` method will take the previous value of the attribute with the name passed in the first argument, and modify it to be an array holding all previous values and the new values, passed in the array in the second argument.

Imagine, you modified your `ProductFactory` with two states: `car` and `luxury`:

The `luxury` state is similar to the `car` state and might look like this:

```
// in ProductFactory.php

public function luxury(): self
{
    return $this->addState([
        // Note: this will add a "luxury" value to existing values of "categories"␣
↪attribute
        'categories' => $this->addValuesTo('categories', ['luxury']),
    ]);
}
```

With this setup, by the time you instantiate your `Product` entity, the attributes will look like this:

```
// in ProductFactory.php

protected function initialize()
{
    return $this
        ->instantiateWith((new Instantiator())
            ->allowExtraAttributes(['categories'])
        )
        ->beforeInstantiate(function($attributes) {
            // $attributes['categories'] => ['luxury', 'car', 'vehicle']
            return $attributes;
        })
```

Then, in the `afterInstantiate` callback, you can find those specific `Category` entities in the database and assign them to the `Product`:

```
// in ProductFactory.php

protected function initialize()
{
    // ...

    $this->afterInstantiate(function(Product $product, $attributes) {
        // If explicit category names were assigned by factory states,
        // find related categories and assign to the product
        if (!empty($attributes['categories'])) {
            foreach ((array) $attributes['categories'] as $c_this_category) {
                $category_proxy = CategoryFactory::findOrCreate([
                    'categoryName' => $c_this_category
                ]);

                /** @var Category $category_entity */
                $category_entity = $category_proxy->object();

                $product->addElementToCategoryCollection($category_entity);
            }
        }
    });

    // ...

    return $this;
}
```

When your setup will do just fine with random categories assigned to `Product` entities, there are obviously simpler ways to fetch random `Category` entities and set them on `Product::categoryCollection`. But when you need some specific product categories, moving this logic from stories into a model factory itself feels like a better alternative, and using states for this task makes it even more elegant. With just one line of code, you can create a batch of entities and establish some of the relationships “in one go”.

```
// in story or test class

public function createLuxuryCars()
```

(continues on next page)

(continued from previous page)

```
{
    $factory_product_stub = ProductFactory::new();

    // create 20 Product entities in categories "luxury", "car" and "vehicle" with␣
↪random "productName"
    $this->persistService->createMany($factory_product_stub, 20, function(Generator
↪$faker) {
        return [
            'productName' => $faker->randomElement([
                '2021 Porsche Boxster', '2021 Genesis G80', '2021 Volvo S90', '2021 BMW␣
↪7 Series',
                '2021 Chevrolet Corvette', '2021 Audi TT', '2021 Audi A5', '2020␣
↪Mercedes-Benz SL',
                '2021 Genesis G90', '2020 Kia K900', '2020 Mercedes-Benz E-Class',
                '2020 Audi R8', '2020 Mercedes-Benz S-Class',
            ]),
        ];
    }, ['luxury', 'car', 'recent', 'promoted']);

    $this->persistService->persistAndFlushAll();
}
```

Note: In the example above, states `recent` and `promoted` will modify model attributes (`registeredAt` and `inPromotion`, respectively), and states `luxury` and `car` will add values to a custom attribute `categories` which is used in `ProductFactory::afterInstantiate()` callback to find related `Category` entities in the database and assign them to the products.

1.1.2 Creating entities for tests using VoltelExtraFoundryBundle services

1. One of the ways to set up your testing environment with `zenstruck/foundry` bundle is to use Global State. If you want some initial database state to be used for all tests in the test suite, follow instructions in *How to use Global State in your app tests* section below.
2. You can also create only those entities that are needed to run a specific test. Read in *How to set up custom entities in a test* section on how you can do it a little easier with *VoltelExtraFoundryBundle*.

Those two approaches listed above are not mutually exclusive. You can have some Global State database entities created for all tests, and for tests that need some additional entities, you can create those at the start of the test function in the "arrange" phase. Learn about how you can use `zenstruck/foundry` for "Arrange", "Act", "Assert" testing patterns.

VoltelExtraFoundryBundle can help to do either of the two things with just a couple lines of code.

How to use Global State in your app tests

The Global State approach saves time by eliminating the need to seed your database with the same initial data before every test. You can load `zenstruck/foundry` stories that will create the initial database state, using factories and even other stories. This reduces time needed to run tests in your test suite.

The initial setup of the Global State in `tests/bootstrap.php` of your application can look like this:

```
// in tests/bootstrap.php

//...

Zenstruck\Foundry\Test\TestState::addGlobalState(function () {
    // place all initial state loading logic in one specialized class
    \App\DataFake\Foundry\Story\GlobalStory::load();

    // or just load several stories one by one, similar to fixtures
    \App\DataFake\Foundry\Story\UserStory::load();
    \App\DataFake\Foundry\Story\ProductStory::load();
    \App\DataFake\Foundry\Story\OrderStory::load();
});
```

But you can save even more time while running a test suite by loading the initial state from a MySQL dump file (produced in advance) instead of creating and persisting entities with factories and stories, even if it's only done once for all tests in a test suite.

Instead of creating entities "on the fly", we run a set of MySQL `INSERT` commands from the dump file.

To do this, follows these steps:

1. Using fixtures, *seed your development database* (in "dev" environment) with sample data you'd like to use as a Global State for your tests.

   Use `zenstruck/foundry` stories as described in *Steps to follow to seed the database*.
2. *Export your data into a MySQL dump file*.
3. *Place your dump file* inside your project (e.g. in `var/mysql_dumps` directory).
4. *Configure* your *VoltelExtraFoundryBundle* to locate the dump file with SQL queries to re-create the initial database state in test environment.
5. Create the `GlobalStory` class with `GlobalStory::load()` method to *load/import MySQL dump file* into a test database before the test suite is run.

Preparation

Set up a new MySQL test database. In MySQL, configure the test user (probably the same as for your `dev` environment database) with appropriate schema privileges. Then, create the schema.

1. Configure `DATABASE_URL` e.g. in `.env.test` or `.env.test.local`:

```
# in ".env.test.local"

DATABASE_URL=mysql://my_username:my_password@127.0.0.1:3306/my_database_test?
↪serverVersion=5.7
```

2. Create the schema/database for `test` environment. For example, you can do it from your application, with the console command:

   ```
   > php bin/console doctrine:schema:create --env=test
   ```

Step 1: Seed the development database

```php
// in UserFixture.php

use App\DataFake\Foundry\Story\UserStory;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class UserFixture extends Fixture
{
    public function load(ObjectManager $manager)
    {
        UserStory::load();
    }
}
```

In this example, `UserStory::build()` method will contain all the logic to create `User` entity and other related entities (e.g. `UserGroup` or `Address`, etc.).

Conveniently, `UserStory` might optionally use `Voltel\ExtraFoundryBundle\Service\FixtureEntity\SetUpFixtureEntityService` to create separate entities or `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService` to facilitate their persistence into the database, or use `zenstruck/foundry` Model Factories straightforward to create entities and persist/flush them into the database.

Then, when fixture classes are ready, load fixtures as usual (see Loading Fixtures section of Symfony docs). Run in the terminal:

```
> php bin/console doctrine:fixtures:load
```

Step 2: Dump your MySQL dev database into a file

I find it easy to export MySQL data with MySQL Workbench graphic user interface, but there are definitely other ways to do it (e.g. with `mysqldump` terminal command or `phpmyadmin` export). See examples of `mysqldump` usage at mysqldump program examples.

```
> mysqldump --user=my_username --password  my_database_dev > my_test_dump.sql
```

Important: Make sure that MySQL dump should contain `DROP TABLE` queries along with `CREATE TABLE` queries: existing data and indexes might prevent inserting new records and therefore need to be taken out of the way.

Step 3: Place the dump file inside your project

You can place your exported dump file anywhere in your project, since the location of the dump file should be configured (see next step). I place it in `var/mysql_dumps` folder.

```
your_project/
└─ var/
   ├─ mysql_dumps/
   ├─ cache/
   └─ log/
```

Step 4: Configure *VoltelExtraFoundryBundle*

In your project create a new configuration file, e.g. `voltel_extra_foundry.yaml` in `config/packages/test` directory.

```
your_project/
└─ config/
   └─ packages/
      └─ test/
```

An example of configuration is provided here:

```
voltel_extra_foundry:
    # Database (persistence layer) type: "mysql" is currently the only supported option.
    database_type:        mysql

    # Filesystem path of the directory where database dump files are located.
    dump_directory_path:  '%kernel.project_dir%/var/mysql_dumps'

    # File name (w/o file path) of database dump file that will be loaded in the current␣
↪database.
    dump_file_name:       my_test_dump.sql

    # Doctrine connection name to use for data loading.
    connection_name:      default
```

Step 5. Create the `GlobalStory` class

```
// in GlobalStory.php

namespace App\DataFake\Foundry\Story;


use Voltel\ExtraFoundryBundle\Service\FixtureLoad\SqlDumpLoaderService;
use Zenstruck\Foundry\Story;

class GlobalStory extends Story
{
    private $sqlDumpLoaderService;
```

(continues on next page)

(continued from previous page)

```
    public function __construct(
        SqlDumpLoaderService $sqlDumpLoaderService
    )
    {
        $this->sqlDumpLoaderService = $sqlDumpLoaderService;
    }


    public function build(): void
    {
        $this->sqlDumpLoaderService->loadSqlDump();
    }

}
```

`Voltel\ExtraFoundryBundle\Service\FixtureLoad\SqlDumpLoaderService` service will do two things:

1. Using the `database_type` bundle configuration option, it will locate an appropriate service implementing `Voltel\ExtraFoundryBundle\Service\FixtureLoad\LoadDumpFromDatabaseInterface`.

   Currently, there is only one service implementing this interface responsible for loading MySQL dumps: `Voltel\ExtraFoundryBundle\Service\FixtureLoad\MySqlDumpFileLoadingService`.

2. The `LoadDupmFromDatabaseInterface::loadSqlDump()` method will do the following:

   - Check the presence of file configured in `dump_directory_path` and `dump_file_name` options;
   - Execute every SQL query in the dump file skipping only empty strings and strings starting with `--` (comments).

   Note: Name of the Doctrine provided PDO connection can be configured in `connection_name` bundle configuration option and has a value of `default`.

   Note: The SQL queries being executed are one of the following:

     – `DROP TABLE IF EXISTS`;
     – `CREATE TABLE`;
     – `LOCK TABLES`;
     – `INSERT INTO`;
     – `UNLOCK TABLES`;
     – Numerous auxiliary queries that look e.g. like:

       `/*!40101 SET NAMES utf8 */;`

As a result, whenever you run your test suite, a Global State will load configured SQL dump in your test database.

How to set up custom entities in a test

In the “arrange” phase of a functional test, you will sometimes rely on some specific entities existing in the database. Moreover, these entities might be different for the same test with each new dataset returned by the data provider.

VoltelExtraFoundryBundle has a convenience service, `Voltel\ExtraFoundryBundle\Service\FixtureEntity\SetUpFixtureEntityService`, with the `createEntities()` method to create batches of entities “on the fly” using an array with instructions as a *blueprint*.

The method will take a model factory, an array with instructions for entity “spawning”, and an optional flag whether to immediately persist newly created entities or leave this task to the caller.

“Explicit” syntax of spawning instructions

The “spawning” instructions are provided as an array of arrays (a chunk), where keys of the outer array can either be omitted or used as descriptive labels (e.g. for documentation purposes), and values are nested arrays with three optional keys:

```
// How many entities to create in this spawn.
// An integer. If "0", the instruction entrance will be skipped.
'count' => 5


// What states the factory should be modified with.
// The states are method names on the entity factory class.
// If states take no arguments, just list their names.
'states' => ['stateOne', 'stateTwo']

// If a state takes arguments, pass the state name as a key
// and an array of parameters as a value.
'states' => ['stateOne' => ['param_1', 'param_2'], 'stateTwo' => ['param_1']]

// If a state takes exactly one argument, the value can be passed w/o an array.
// The following two instructions are equivalent:
'states' => ['stateOne' => ['param_1'], 'stateTwo' => [5]]
'states' => ['stateOne' => 'param_1', 'stateTwo' => 5]


// Attributes with which the entities should be created.
// These attributes will override the "defaults" provided by the entity factory.
// The attribute values can be of any type, but most often they are scalar.
'attributes' => ['attributeOne' => 15, 'attributeTwo' => 'some string']
```

As usual, it is easier to see the usage with an example:

```
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
use Zenstruck\Foundry\Test\Factories;
use Zenstruck\Foundry\Test\ResetDatabase;
use Voltel\ExtraFoundryBundle\Service\FixtureEntity\SetUpFixtureEntityService;

class MyTest extends KernelTestCase
{
    use ResetDatabase, Factories;

```

(continues on next page)

(continued from previous page)

```
    private const SETUP_CUSTOMERS = [
        'customer_1' =>['states' => ['american', 'human']],
        'customer_2' =>['states' => ['ukrainian', 'human']],
        'customer_3' =>['states' => ['human'], 'attributes' => ['firstName' => 'John',
↪'lastName' => 'Doe']],
        'customer_4' =>['states' => ['human'], 'attributes' => ['firstName' => '',
↪'lastName' => '']],
    ];

    public function __construct(SetUpFixtureEntityService $entityService)
    {
        $this->setUpFixtureEntityService = $entityService;
    }

    protected function setUp(): void
    {
        $kernel = self::bootKernel();
        // ...
    }

    public function testCreateEntities()
    {
        $factory_customer = CustomerFactory::new();

        $this->setUpFixtureEntityService-> createEntities($factory_customer, self::SETUP_
↪CUSTOMERS, true);
    }
}
```

The factory that is provided in the first argument of the `createEntities()` method will be used as a stub which will be modified if "spawning" instructions have any `states` listed.

In the second argument you should provide an array of arrays (a chunk) with spawning instructions. Bare this in mind if you use data providers, since a chunk can include only one item (one nested array), which may look a bit confusing.

The last argument to `createEntities()`, if set to `true`, will signal to persist and flush all the entities that `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService` has in its jars. This is important if your tests use repository assertions that expect certain entities to exist at some point in time.

Simplified ("implicit") syntax of spawning instructions

If you create entities in "one go" (i.e. all instructions are provided at once), as in the example above where `createEntities()` is passed a chunk of spawning instructions at once in the second parameter, you might take advantage of a simplified instructions syntax.

The "implicit" (i.e. simplified) syntax, as opposed to "explicit" syntax described above, doesn't rely on reserved array keys (i.e. `"count"`, `"states"` or `"attributes"`) but is resolved based on the following logic:

- if the first array item has key "0" and its value is numeric, the numeric value will be interpreted as equivalent to the "count" key with the "explicit" syntax;
- if a method exists on an entity factory with the name from an array item **value** (when the key in numeric), it will be interpreted as the name of the factory state to apply;

- if a method exists on an entity factory with the name from an array item **key** (when the key in not numeric), it will be interpreted as name of the factory state to apply. In this case, this array item's value will be interpreted as value(s) for this factory state.
- all other key-value pairs will be interpreted as attributes for the factory `create()` method.

Important: With simplified syntax, do not use any custom array keys/values that cannot be interpreted as valid factory states or entity attributes. All unknown states or non-existent attributes will cause an error.

Note: If you want to have some custom data on the dataset array, e.g. for a test data provider, use "explicit" syntax where you can add custom keys.

With "explicit" syntax, you can add any custom keys that are not reserved (i.e. "count", "states" or "attributes" keys) to an array with spawning instructions in order to pass additional information to data providers.

See the example below: the first set of instructions defines a custom key "expect_error" and the last set defines a custom key "exception_class". This can be used in tests along with data providers and methods like `expectException()`.

```
[
    'instructions set 1' => [
        'count' => 5,
        'states' => ['withStaffCount' => 3],
        'expect_error' => false
    ],
    'instructions set 2' => [
        'count' => 5,
        'states' => ['withStaffCount'],
        'expect_error' => true
    ],
    'instructions set 3' => [
        'count' => 5,
        'states' => ['withStaffCount' => 'many'],
        'exception_class' => BadMethodCallException::class
    ],
]
```

Using `SetUpFixtureEntityService` with data providers

The same entity setup instructions that are used to create batches of entities can be used by data providers as data sets for test methods.

With PhpUnit Data Providers specification in mind, remember that `createEntities()` method takes a chunk (an array of arrays) of instructions as the second parameter, whether it has one set of instructions or more:

```
class MyOtherTest extends KernelTestCase
{
    public const SETUP_PRODUCTS = [
        'product_1' => ['count' => 10, 'states' => ['luxury', 'car'] ] ,
        'product_2' => ['count' => 10, 'states' => ['ordinary', 'car'] ] ,
        'product_3' => ['count' => 20, 'states' => ['jewelry'] ],
        'product_4' => ['count' => 20, 'states' => ['furniture'] ],
```

(continues on next page)

(continued from previous page)

```
        'product_5' => ['count' => 10, 'states' => ['house'] ],
        'product_6' => ['count' => 10, 'states' => ['luxury', 'apartment'] ],
        'product_7' => ['count' => 10, 'states' => ['ordinary', 'apartment'] ],
    ];

    //...

    /**
     * Asserts that count of created entities is as expected.
     *
     * @dataProvider productDataProviderOneSpawnInAChunk
     */
    public function testCreateEntitiesWithDataProvider(
        array $a_spawn_instructions,
        int $n_expected_entity_count
    )
    {
        // ...
        $factory_product = ProductFactory::new();
        $setUpFixtureEntityService->createEntities($factory_product, $a_spawn_
↪instructions, true);

        $repo = ProductFactory::repository();
        $repo->assert()->count($n_expected_entity_count);
    }


    public function productDataProviderOneSpawnInAChunk()
    {
        foreach (self::SETUP_PRODUCTS as $c_label => $a_instructions_for_one_spawn) {

            $n_expected_entity_count = (int) $a_instructions_for_one_spawn['count'] ?? 1;

            // PhpUnit expects an array of arguments, so yield an array item
            yield $c_label => [
                // argument one - an array of arrays (a chunk) that holds instructions
                // for one "spawn" in this case
                [$a_instructions_for_one_spawn],

                // argument two - an integer with expected entity count
                $n_expected_entity_count
            ];
        }
    }
```

With "explicit" style of "spawning" instructions, you can configure the expected outcome of the test (i.e. whether to expect error/exception or not) and even specify the class name of the expected exception. For this, you need to add some logic both to the data provider and the test method itself:

```
// in MyTest class

private const STATES = SetUpFixtureEntityService::KEY_STATES;
```

(continues on next page)

(continued from previous page)

```
private const ATTRS = SetUpFixtureEntityService::KEY_ATTRIBUTES;
private const COUNT = SetUpFixtureEntityService::KEY_COUNT;

// custom keys
private const ERROR = 'expect_error';
private const EXCEPTION = 'exception_fqcn';

private const SETUP_CUSTOMERS_WITH_ERRORS = [
    '1. state w/o an obligatory parameter' => [self::ERROR => true, self::STATES => [
↪'company', 'withStaffCount'], self::ATTRS => ['firstName' => ' ']],
    '2. state w/o an obligatory parameter, exception FQCN specified' => [
        self::EXCEPTION => \ArgumentCountError::class,
        self::STATES => ['company', 'withStaffCount'], self::ATTRS => ['firstName' => '
↪']
    ],
    '3. unknown state' => [self::ERROR => true, self::STATES => ['company',
↪'unknownStateName'], self::ATTRS => ['firstName' => ' ']],
    '4. unknown state, exception FQCN specified' => [
        self::EXCEPTION => \BadMethodCallException::class,
        self::STATES => ['company', 'unknownStateName'], self::ATTRS => ['firstName' =>
↪' ']
    ],
    '5. state w/ bad parameter' => [self::ERROR => true, self::STATES => ['company',
↪'withStaffCount' => 'this must be an integer'], self::ATTRS => ['firstName' => 'Texas␣
↪Roadhouse']],
    '6. state w/ bad parameter, exception FQCN specified' => [
        self::EXCEPTION => \TypeError::class,
        self::STATES => ['company', 'withStaffCount' => 'this must be an integer'],
    ],
    '7. state w/ unknown attribute' => [self::ERROR => true, self::STATES => ['company',
↪'withStaffCount' => 5], self::ATTRS => ['unknownAttribute' => 'Texas Roadhouse']],
    '8. state w/ unknown attribute, exception FQCN specified' => [
        self::EXCEPTION => \InvalidArgumentException::class,
        self::STATES => ['company'], self::ATTRS => ['unknownAttribute' => 'Texas␣
↪Roadhouse'],
    ],
];


/**
 * @dataProvider customerDataProviderForExplicitSetUpDefinition
 */
public function testCreateEntitiesWithParameterizedState(
    array $a_spawn_instructions,
    int $n_expected_entity_count,
    bool $l_expect_error = false,
    string $c_expect_exception_fqcn = null
)
{
    // "Arrange" phase
    /** @var SetUpFixtureEntityService $setUpFixtureEntityService */
    $setUpFixtureEntityService = self::$container->get('test.voltel_extra_foundry.entity_
↪setup');
```

(continues on next page)

(continued from previous page)

```
    $factory_customer = CustomerFactory::new(); // ->withoutPersisting();

    if ($l_expect_error) {
        $this->expectException($c_expect_exception_fqcn ?? \Throwable::class);
    }//endif

    // Try and create/persist a spawn of Customer entities with provided instructions
    $setUpFixtureEntityService->createEntities($factory_customer, [$a_spawn_
↪instructions], true);

    $repo = CustomerFactory::repository();
    $repo->assert()->count($n_expected_entity_count);

    // ...
}


public function customerDataProviderForExplicitSetUpDefinition()
{
    foreach (self::SETUP_CUSTOMERS_WITH_ERRORS as $c_label => $a_one_spawn_instructions)
↪{
        // This dataset should generate an exception
        // 1) if there is a custom key "expect_error" set to "true", or
        // 2) if there is a custom key "exception_fqcn" with FQCN of the expected␣
↪exception,
        $c_expect_exception_fqcn = $a_one_spawn_instructions[self::EXCEPTION] ?? null;
        $l_expect_error = $a_one_spawn_instructions[self::ERROR] ?? is_null($c_expect_
↪exception_fqcn);

        // Count of entities to create can be found in a special key "count"
        $n_expected_entity_count = $a_one_spawn_instructions[self::COUNT] ?? 1;

        yield $c_label => [
            $a_one_spawn_instructions, $n_expected_entity_count, $l_expect_error, $c_
↪expect_exception_fqcn
        ];
    }
}
```

To see more detailed examples of the `SetUpFixtureEntityService` use in the testing environment, look in `Voltel\ExtraFoundryBundle\Tests\Service\FixtureEntity\SetUpFixtureEntityServiceTest` class source code.

On the whole, the `arrange` phase of your tests may look neat with only a couple of lines of code, when all the instructions for creation (spawning) of tested entities are given elsewhere (e.g., in class constants or data providers).

1.1.3 Testing VoltelExtraFoundryBundle

If you want to see examples of using *VoltelExtraFoundryBundle* and/or test it, you should look into the `tests` directory.

Two files in the root of the bundle are also important for testing set-up:

- `phpunit.xml.dist`
- `cli-config.php`

Bundle's `tests` directory structure

```
your_project/
└─ tests/
   ├─ Service/
   │  ├─ FixtureEntity/
   │  └─ FixtureLoad/
   ├─ Setup/
   │  ├─ Entity/
   │  ├─ Factory/
   │  ├─ Kernel/
   │  ├─ MySqlDump/
   │  ├─ Service/
   │  └─ Story/
   └─ bootstrap.php
```

- All tests are located in `tests/Service` directory.
- In `tests/Setup/Entity` directory, you will find entities that are going to be created during the tests.
- In `tests/Setup/Factory` directory, you will find `zendstruck/foundry` model factories describing the related entities (one factory per entity).
- In `tests/Setup/Story` directory, you shall definitely look at the way entities are created and persisted using services from *VoltelExtraFoundryBundle*.

> **Note:** Inspect classes in `tests/Setup/Story` directory to see examples of suggested `Voltel\ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService` usage.

- In `tests/Setup/Service` directory, you will find a faux service that is used in `afterPersist` callback in `ProductFactory` class. It is needed in order to to change the `slug` property on `Product` entity and assert during the test that the `afterPersist` callback was indeed invoked.
- The `tests/Setup/Kernel` directory contains the only file with `Voltel\ExtraFoundryBundle\Tests\Setup\Kernel\VoltelExtraFoundryTestingKernel` class where all services used during the tests are configured, including the services provided by *VoltelExtraFoundryBundle* and Doctrine.
- In `tests/Setup/MySqlDump` directory, `mysql_dump_for_tests.sql` file contains a MySQL dump that is asserted during the tests to be properly loaded/imported into the database.
- File `bootstrap.php` was modified to retrieve the value of `DATABASE_URL` from `phpunit.xml.dist` and set it in the super global `$_ENV` array to be later used when running Doctrine CLI commands (read below).

How to set up bundle tests

Overview

To set up testing with PhpUnit using a MySQL test database, several steps need to be done:

1. *Configure the kernel class* that is used by the testing suite;
2. *Create a test MySQL database* (e.g. “voltel_extra_foundry_test”);
3. *Set up MySQL schema*.

Step 1: Configure the kernel class

The testing kernel is configured in `Voltel\ExtraFoundryBundle\Tests\Setup\Kernel\VoltelExtraFoundryTestingKernel` class.

```
// in VoltelExtraFoundryTestingKernel.php

class VoltelExtraFoundryTestingKernel extends Kernel
{
    // ...

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(function (ContainerBuilder $container) use ($loader) {
            // Services that are used in tests
            // ...

            // Configure Doctrine
            $container->loadFromExtension('doctrine', [
                'dbal' => [
                    'default_connection' => 'default',
                    'connections' => [
                        'default' => [
                            'url' => $_ENV['DATABASE_URL'],
                            'logging' => false,
                            'override_url' => true,
                        ]
                    ],
                ],
                'orm' => [/* ... */]
            ]);
        });
    }

}//end of class
```

The connection URL in the code snippet above depends on the environmental variable *DATABASE_URL* which must be configured in `phpunit.xml.dist`:

```
<!-- in phpunit.xml.dist -->

<php>
```

(continues on next page)

(continued from previous page)

```
    <!-- ... -->
     <env name="DATABASE_URL" value="mysql://testuser:password@127.0.0.1:3306/
↪voltel_extra_foundry_test?serverVersion=5.7" />

     <env name="KERNEL_CLASS" value="Voltel\ExtraFoundryBundle\Tests\Setup\
↪Kernel\VoltelExtraFoundryTestingKernel" />

</php>
```

Important: Change DATABASE_URL definition in `phpunit.xml.dist` to reflect your MySQL test user's credentials and test database/schema name.

Note: The configuration in `phpunit.xml.dist` also contains a definition for another environmental variable, KERNEL_CLASS, which is internally used by `Symfony\Bundle\FrameworkBundle\Test\KernelTestCase`.

Step 2: Create a test MySQL database

In MySQL, create a new database, e.g. "voltel_extra_foundry_test". Configure your test user to have appropriate privileges for this database.

```
> mysql --user=root --password

mysql> CREATE USER IF NOT EXISTS 'testuser'@'localhost' IDENTIFIED BY 'password';
mysql> CREATE DATABASE IF NOT EXISTS voltel_extra_foundry_test;
mysql> GRANT ALL ON voltel_extra_foundry_test.* TO 'testuser'@'localhost';
mysql> quit;
```

Note: Database name, host URL, username, user password must match those configured in `phpunit.xml.dist` for DATABASE_URL environmental variable (see above).

Step 3: Set up MySQL schema

The `cli-config.php` in the root of the project is required for Doctrine bundle CLI tool. The file is quite short; it has only a few things to do:

- boot our custom kernel (`Voltel\ExtraFoundryBundle\Tests\Setup\Kernel\VoltelExtraFoundryTestingKernel`);
- retrieve entity manager from the kernel;
- return an instance of `Symfony\Component\Console\Helper\HelperSet` for the provided entity manager.

With Doctrine CLI configured, run in the terminal:

```
$ vendor/bin/doctrine orm:schema-tool:create
```

or, in Windows command prompt:

```
> "vendor/bin/doctrine" orm:schema-tool:create
```

Run bundle tests

Run the tests with this command:

```
> "vendor/bin/simple-phpunit"
```

Note: The following exact command was run under Windows to obtain the MySQL test database state and produce the dump that is located in `tests/Setup/MySqlDump/mysql_dump_for_tests.sql`.

```
> "vendor/bin/simple-phpunit" tests/Service/FixtureEntity/EntityProxyPersistServiceTest.
↪php --filter=testStories
```

1.1.4 VoltelExtraFoundryBundle

delayed persistence

delay persistence The practice of creating new entities with Doctrine when one or many entities are not immediately registered with Doctrine for persistence. When batches of entities under the same entity manager are finally persisted, the entity manager is then flushed to create entities in a database.

1.1.5 Bundle quirks

This bundle came into existence as an attempt to solve the problem of `zenstruck/foundry` taking way too much time persisting its newly created entities when the numbers of entities reached a thousand and above. The reason for that slow speed was the fact that every entity was immediately persisted and flushed, which is done with Doctrine and is a very resource engaging process.

So, the the first approach to solve this problem was to create factories that wouldn't immediately persist (i.e. to invoke a `Zenstruck\Foundry\Factory::withoutPersisting()` method), then collect the products of their labor (i.e. entities wrapped in objects of `\Zenstruck\Foundry\Proxy` class) and persist/flush them in batches.

There is a certain balance between the workload (count of entities to save), time that a persist operation takes and time that it takes to flush the batch. When the batch is too big, it takes a lot of computing resource to perform a persist operation with Doctrine calculating a large unit of work. But when the batches are small and the flush operation is frequent, we end up with what we started from – low performance due to the overall overhead of connecting to the database and executing many small queries.

I found out in my experiments, that by dealing in batches that do not exceed approximately 10,000 entities, there is a good chance that time that it takes to perform both persist and flush operations is going to be acceptably balanced.

So, I started doing just this until I realized that merely to *delay persistence* might not be enough – I still needed to invoke a related `afterPersist()` callback that could have been configured for a model factory in its `initialize()` method.

And while the *delayed persistence* approach didn't interfere with `instantiateWith()`, `beforeInstantiate()` or `afterInstantiate()` callbacks, it did prevent the invocation of the `afterPersist()` callback for factories modified with `withoutPersisting()`. Was this a quirk or a feature of a `zenstruck/foundry` I do not know.

Here is a permalink to source code of `Zenstruck\Foundry\Factory::create()` method where you can see that if the `isPersisting()` returns `false`, then the code responsible for executing the `afterPersist` callbacks is not run. And there is no available interface (i.e. `public` method) to execute those callbacks afterwards from your code.

```php
// in Zenstruck\Foundry\Factory

$proxy = new Proxy($object);

if (!$this->isPersisting()) {
    return $proxy;
}

return $proxy->save()->withoutAutoRefresh(function(Proxy $proxy) use ($attributes) {
    foreach ($this->afterPersist as $callback) {
        $proxy->executeCallback($callback, $attributes);
    }
});
```

With this new challenge, the first approach was to try and identify the factory class that produces entities of the class that has just been persisted and flushed, then get the callback from this class and execute it passing a newly persisted/flushed entity.

See the bundle's `EntityProxyPersistService::getFactoryForEntityOfClass()` method for implementation. You will see that to identify this factory, `Symfony\Component\DependencyInjection\ServiceLocator` service is injected, and it is configured in the bundle's `services.xml` to collect services tagged with `foundry.factory` tag and store them with the key provided by `Voltel\ExtraFoundryBundle\Foundry\Factory\AbstractFactory::getClassName()` method, which is public. Had `Zenstruck\Foundry\ModelFactory::getClass()` been declared `public`, it could have been used instead in the `ServiceLocator` declaration. But it is declared `protected` and cannot be used for the task.

```xml
    <!-- in bundle's "config/services.xml" -->

    <services>
        <!-- ... -->

        <service id="voltel_extra_foundry.persist_service" class="Voltel\
↪ExtraFoundryBundle\Service\FixtureEntity\EntityProxyPersistService">
            <argument type="service" id="doctrine" />
            <argument type="tagged_locator" tag="foundry.factory" default-index-method=
↪"getClassName" />
        </service>

        <!-- ... -->
    </services>
```

If you extend your factory classes from `Voltel\ExtraFoundryBundle\Foundry\Factory\AbstractFactory`, you can *"help"* service locator to identify the proper factory a little bit faster. Otherwise, the algorithm will iterate every factory in scope (i.e. services tagged with `foundry.factory` tag), try to "force-invoke" the protected `getClass()` method to see if it matches the class name of the entity in question. Not a *big deal* really, as it's all done blazingly fast.

But to get access to the `afterPersist()` callbacks, it invokes the `initialize` method (which is expected to set an array of callbacks), and then invoke each of the callbacks in the loop. It is not an elegant solution at all, especially considering the fact that with this approach there is no way we can identify attributes that were used during the `Factory::create()` invocation. If any of the `afterPersist()` callbacks depended on the attributes array, the results were going to be unpredictable.

So, this is when the need to get away from `addProxy` and `addProxyBatch` became obvious in favor of `createOne` and `createMany` to not only save in the internals entities for *delayed persistence*, but to save a factory that produced those entities as well, and invoke `afterPersist` callbacks that could exist on the factory after the flush operation.

Right now, the bundle has in its guts the `addProxy` and `addProxyBatch` methods, which once were `public`, now are declared `protected` but may be removed in future versions.

On the whole, the trickery described above wouldn't even be needed if there were a legitimate interface to execute the `afterPersist()` callbacks from inside the created `\Zenstruck\Foundry\Proxy` object, or at least from inside the factory that "spawned" the entity in question.

1.2 Extra

- search

1.3 Installation

Make sure Composer is installed globally, as explained in the Installation chapter of the Composer documentation.

Open a command console, enter your project directory and execute:

```
$ composer require voltel/extra-foundry-bundle
```

1.3.1 Applications that don't use Symfony Flex

Step 1: Download the Bundle

Open a command console, enter your project directory and execute the following command to download the latest stable version of this bundle:

```
$ composer require voltel/extra-foundry-bundle --dev
```

Step 2: Enable the Bundle

Then, enable the bundle by adding it to the list of registered bundles in the `config/bundles.php` file of your project:

```
// config/bundles.php
return [
    // ...
    Voltel\ExtraFoundryBundle\VoltelExtraFoundryBundle::class => ['dev' => true, 'test'␣
↪=> true],
];
```

1.4 Next step

Start by reading how to *seed your development database* using *VoltelExtraFoundryBundle* and its services.

INDEX