# Blast Project

***Release***

**Nov 03, 2017**

# Contents

This documentation explain how base entities works.

BaseEntities adds doctrine behaviors to manage entity common data structure and behaviors.

The first idea was to use OOP inheritance, but it had too many disadvantage (particularly because PHP does not know multiple inheritance), and this method provides a powerful workaround.

CHAPTER 1

# Architecture

## 1.1 Traits

Traits have been introduced in PHP since version 5.4.0. [see official PHP Traits documentation](http://php.net/manual/fr/language.oop5.traits.php)

In a Traits, we will define the default implementation of methods and/or attributes.

```
namespace Blast\BaseEntitiesBundle\Entity\Traits;

trait BaseEntity
{
    public function __toString()
    {
        if (method_exists(get_class($this), 'getName'))
        {
            return (string)$this->getName();
        }
        if (method_exists(get_class($this), 'getId'))
            return (string)$this->getId();
        return '';
    }
}
```

This Traits act as a macro, allowing code factoring in single element.

See the previous Entity using this Trait :

```
namespace Librinfo\CRMBundle\Entity;

use Blast\BaseEntitiesBundle\Entity\Traits\BaseEntity;

/**
 * Category
 */
```

```
class Category
{
    // Here we « include » the Trait
    use BaseEntity;

    /**
     * @var string
     */
    protected $id;

    /**
     * @var string
     */
    private $name;

}
```

We don't have to write the « __toString() » method because the trait already holds it.

Note

Traits don't support standard OOP « inheritance ». But, if you want to « override » a trait's method, you just have to override it as if it where a parent class.

```
namespace Librinfo\CRMBundle\Entity;

use Blast\BaseEntitiesBundle\Entity\Traits\BaseEntity;

/**
 * Category
 */
class Category
{
    use BaseEntity;

    public function __toString()
    {
        return 'overrided_method';
    }
}
```

When executing this instruction : `echo new Category();` it outputs `overrided_method`.

## 1.2 EventSubscribers

**We're using standard Doctrine EventSubscriber to manage BaseEntities behaviors.**

- [see official Symfony documentation](http://symfony.com/doc/current/cookbook/doctrine/event_listeners_subscribers.html#creating-the-subscriber-class)
- [see official Doctrine documentation](http://doctrine-orm.readthedocs.org/projects/doctrine-orm/en/latest/reference/events.html#the-event-system)

Here's a simplified example of Timestampable EventSubscriber :

```
namespace Blast\BaseEntitiesBundle\EventListener;
```

```
use DateTime;
use Doctrine\Common\EventSubscriber;
use Doctrine\ORM\Event\LifecycleEventArgs;
use Doctrine\ORM\Event\LoadClassMetadataEventArgs;
use Doctrine\ORM\Mapping\ClassMetadata;

class TimestampableListener implements EventSubscriber
{
    /**
     * Returns an array of events this subscriber wants to listen to.
     *
     * @return array
     */
    public function getSubscribedEvents()
    {
        return [
            'loadClassMetadata', // event when doctrine build Entities mapping
            'prePersist', // event when doctrine creates new entity
            'preUpdate' // event when doctrine update existing entity
        ];
    }

    /**
     * define Timestampable mapping at runtime
     *
     * @param LoadClassMetadataEventArgs $eventArgs
     */
    public function loadClassMetadata(LoadClassMetadataEventArgs $eventArgs)
    {
        /** @var ClassMetadata $metadata */
        $metadata = $eventArgs->getClassMetadata();

        if (!$this->hasTrait($metadata->getReflectionClass(),
↪'Blast\BaseEntitiesBundle\Entity\Traits\Timestampable'))
            return; // return if current entity doesn't use Timestampable trait

        // [...]

        // setting default mapping configuration for Timestampable

        // createdDate
        $metadata->mapField([
            'fieldName' => 'createdDate',
            'type'      => 'datetime',
            'nullable'  => true
        ]);

        // [...]

        // createdBy
        $metadata->mapManyToOne([
            'targetEntity' => $this->userClass,
            'fieldName'    => 'createdBy',
            'joinColumn'   => [
                'name'                 => 'createdBy_id',
                'referencedColumnName' => 'id',
                'onDelete'             => 'SET NULL',
                'nullable'             => true
```

```
            ]
        ]);

        // [...]
    }
```

**This EventSubscriber declares which events it will manage with the method `getSubscribedEvents()`.**

- [see official Doctrine documentation](http://doctrine-orm.readthedocs.org/projects/doctrine-orm/en/latest/reference/events.html#lifecycle-events)

For each subscribed events, this class has to implement corresponding method :

php Event : `loadClassMetadata` => `Method : loadClassMetadata()`

**Let's take a usefull example :**

- The need : automaticaly inserting creation date of an entity and storing the User that created that entity.
- Expected : simplify entity lifecycle logging management.

```
class TimestampableListener implements EventSubscriber
{
    // [...]

    /**
     * sets Timestampable dateTime and user information when persisting entity
     *
     * @param LifecycleEventArgs $eventArgs
     */
    public function prePersist(LifecycleEventArgs $eventArgs)
    {
        $entity = $eventArgs->getObject();

        if (!$this->hasTrait($entity,
↪'Blast\BaseEntitiesBundle\Entity\Traits\Timestampable'))
            return;

        $user = $this->tokenStorage->getToken()->getUser(); // Using SF 2.6␣
↪TokenStorage service to retreive current user
        $now = new DateTime('NOW');

        $entity->setCreatedBy($user);
        $entity->setCreatedDate($now);
    }

    // [...]
}
```

This is quite trivial, this event listener appends data before persisting entities that use Timestampable trait.

CHAPTER 2

# Searchable trait

The searchable trait creates a database index (on a distinct table) for searching entities by keywords. The keywords are automatically updated each time an entity is created / updated / deleted.

**To enable this functionnality on an entity :**

- add the Searchable trait to the entity :

```
 namespace MyBundle\Entity;

 use Blast\BaseEntitiesBundle\Entity\Traits\Searchable;

 class Contact
 {
     use Searchable;
     [...]
 }


* Create a search index entity that extends SearchIndexEntity (the name must be the␣
↪entity name suffixed by "SearchIndex") and specify the fields that need to be␣
↪indexed :
```

```
namespace MyBundle\Entity;

use Blast\BaseEntitiesBundle\Entity\SearchIndexEntity;

class ContactSearchIndex extends SearchIndexEntity
{
    public static $fields = ['name', 'description', 'address', 'city', 'country',
↪'email', 'url'];
}
```

```
# MyBundle/Resources/doctrine/ContactSearchIndex.orm.yml
MyBundle\Entity\ContactSearchIndex:
  type: entity
```

This bundle comes with a `librinfo:update:search` console command to batch-update the search indexes :

```
php app/console librinfo:update:search MyBundle:MyEntity
```