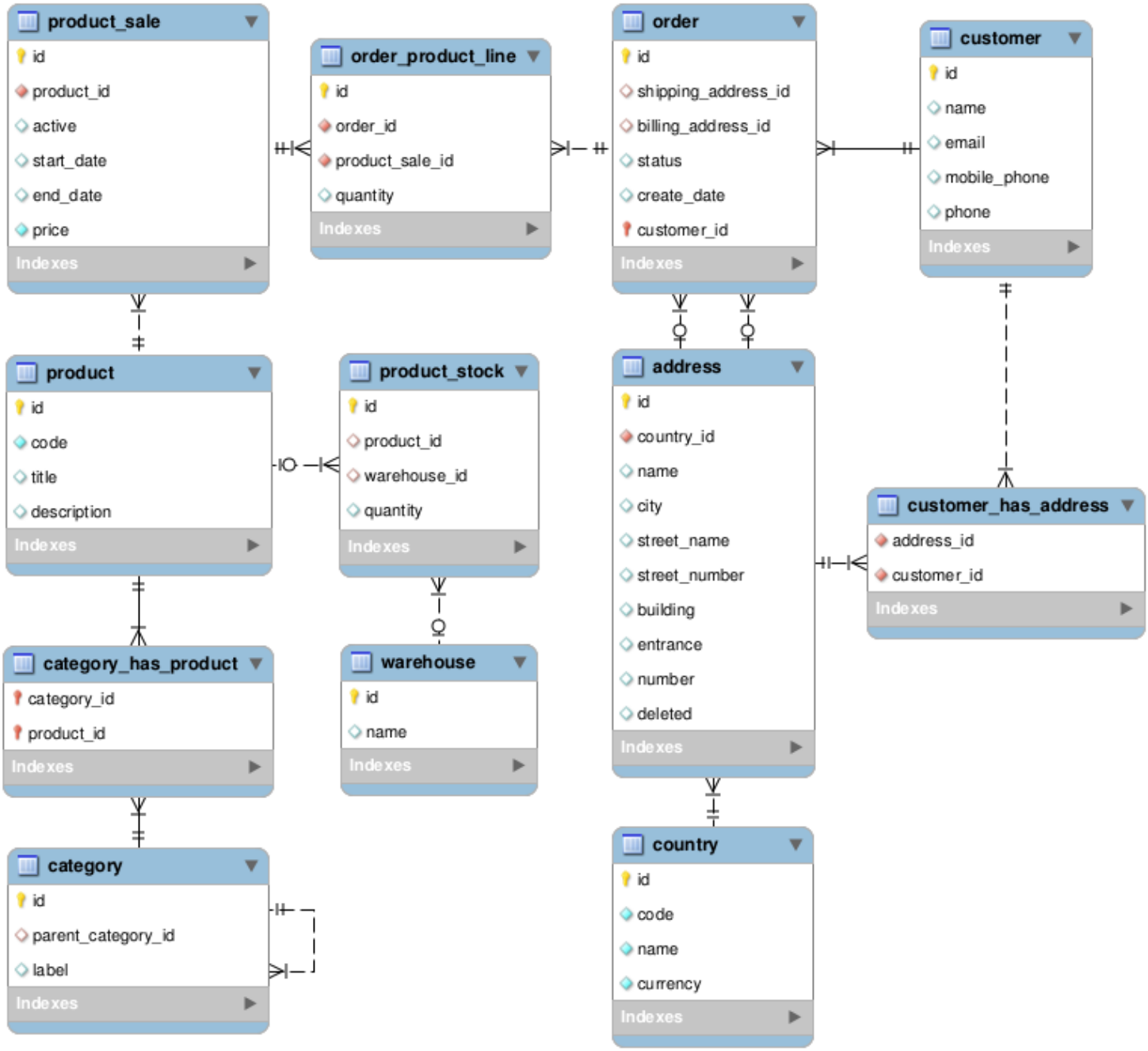
3. CRUD

3.1 Database schema

We will extend our database schema to add some of the concepts a typical e-commerce application may need. A global view of the database schema we are going to create:



We have already created **category**, **product** and **category_has_product** tables, now we proceed with the rest.

```
/* Warehouse */
CREATE TABLE IF NOT EXISTS `warehouse` (
  `id` SMALLINT(5) UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB;

/* Product Stock */
CREATE TABLE IF NOT EXISTS `product_stock` (
  `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `product_id` INT(11) UNSIGNED NOT NULL,
  `warehouse_id` SMALLINT(5) UNSIGNED NOT NULL,
  `quantity` INT(11) NOT NULL DEFAULT 0,
  PRIMARY KEY (`id`) ,
  INDEX `idx_product_id` (`product_id` ASC) ,
  INDEX `idx_warehouse_id` (`warehouse_id` ASC) ,
  CONSTRAINT `fk_product_stock_product_id`
    FOREIGN KEY (`product_id`)
    REFERENCES `product` (`id`),
  CONSTRAINT `fk_product_stock_warehouse_id`
    FOREIGN KEY (`warehouse_id`)
    REFERENCES `warehouse` (`id`))
ENGINE = InnoDB;

/* Country */
CREATE TABLE IF NOT EXISTS `country` (
  `id` SMALLINT(6) UNSIGNED NOT NULL AUTO_INCREMENT,
  `code` VARCHAR(5) NOT NULL,
  `name` VARCHAR(60) NOT NULL,
  `currency` VARCHAR(3) NOT NULL,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB;

/* Address */
CREATE TABLE IF NOT EXISTS `address` (
  `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `country_id` SMALLINT(6) UNSIGNED NOT NULL,
  `name` VARCHAR(45) NULL DEFAULT NULL,
  `city` VARCHAR(45) NULL DEFAULT NULL,
  `street_name` VARCHAR(45) NULL DEFAULT NULL,
  `street_number` VARCHAR(45) NULL DEFAULT NULL,
  `building` VARCHAR(45) NULL DEFAULT NULL,
  `entrance` VARCHAR(45) NULL DEFAULT NULL,
  `number` VARCHAR(45) NULL DEFAULT NULL,
  `deleted` TINYINT(1) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) ,
  INDEX `idx_country_id` (`country_id` ASC) ,
  CONSTRAINT `fk_address_country_id`
    FOREIGN KEY (`country_id`)
    REFERENCES `country` (`id`))
ENGINE = InnoDB;
```

```sql
/* Customer */
CREATE TABLE IF NOT EXISTS `customer` (
  `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL DEFAULT NULL,
  `email` VARCHAR(45) NULL DEFAULT NULL,
  `mobile_phone` VARCHAR(45) NULL DEFAULT NULL,
  `phone` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB;


/* Customer Has Address */
CREATE TABLE IF NOT EXISTS `customer_has_address` (
  `address_id` INT(11) UNSIGNED NOT NULL,
  `customer_id` INT(11) UNSIGNED NOT NULL,
  PRIMARY KEY (`address_id`, `customer_id`),
  INDEX `idx_address_id` (`address_id` ASC) ,
  INDEX `idx_customer_id` (`customer_id` ASC) ,
  CONSTRAINT `fk_customer_has_address_address_id`
    FOREIGN KEY (`address_id`)
    REFERENCES `address` (`id`),
  CONSTRAINT `fk_customer_has_address_customer_id`
    FOREIGN KEY (`customer_id`)
    REFERENCES `customer` (`id`))
ENGINE = InnoDB;


/* Order */
CREATE TABLE IF NOT EXISTS `order` (
  `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `shipping_address_id` INT(11) UNSIGNED NULL DEFAULT NULL,
  `billing_address_id` INT(11) UNSIGNED NULL DEFAULT NULL,
  `status` TINYINT(4) NULL DEFAULT NULL,
  `create_date` DATETIME NULL DEFAULT NULL,
  `customer_id` INT(11) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`, `customer_id`) ,
  INDEX `idx_shipping_address_id` (`shipping_address_id` ASC) ,
  INDEX `idx_billing_address_id` (`billing_address_id` ASC) ,
  INDEX `idx_status` (`status` ASC) ,
  INDEX `idx_customer_id` (`customer_id` ASC) ,
  CONSTRAINT `fk_order_shipping_address_id`
    FOREIGN KEY (`shipping_address_id`)
    REFERENCES `address` (`id`),
  CONSTRAINT `fk_order_billing_address_id`
    FOREIGN KEY (`billing_address_id`)
    REFERENCES `address` (`id`),
  CONSTRAINT `fk_order_customer_id`
    FOREIGN KEY (`customer_id`)
    REFERENCES `customer` (`id`))
ENGINE = InnoDB;
```

```
/* Product Sale */
CREATE TABLE IF NOT EXISTS `product_sale` (
  `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `product_id` INT(11) UNSIGNED NOT NULL,
  `active` TINYINT(1) NULL DEFAULT 0,
  `start_date` DATETIME NULL DEFAULT NULL,
  `end_date` DATETIME NULL DEFAULT NULL,
  `price` INT(11) NOT NULL,
  PRIMARY KEY (`id`) ,
  INDEX `idx_product_id` (`product_id` ASC) ,
  INDEX `idx_start_date` (`start_date` ASC) ,
  INDEX `idx_end_date` (`end_date` ASC) ,
  CONSTRAINT `fk_product_sale_product_id`
    FOREIGN KEY (`product_id`)
    REFERENCES `product` (`id`))
ENGINE = InnoDB;

/* Order Product Line*/
CREATE TABLE IF NOT EXISTS `order_product_line` (
  `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
  `order_id` INT(11) UNSIGNED NOT NULL,
  `product_sale_id` INT(11) UNSIGNED NOT NULL,
  `quantity` INT(11) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) ,
  INDEX `idx_order_id` (`order_id` ASC) ,
  INDEX `idx_product_sale_id` (`product_sale_id` ASC) ,
  CONSTRAINT `fk_order_product_line_order_id`
    FOREIGN KEY (`order_id`)
    REFERENCES `order` (`id`),
  CONSTRAINT `fk_order_product_line_product_sale_id`
    FOREIGN KEY (`product_sale_id`)
    REFERENCES `product_sale` (`id`))
ENGINE = InnoDB;
```

Now we need to create entities to map the newly created tables. Fortunately, Doctrine comes with a set of useful commands, you can take a look at them in the **doctrine** namespace when you run `app/console`. Today we are going to use some of them.

Import the database schema and create entities for each table.

```
$ app/console doctrine:mapping:import AppBundle annotation
Importing mapping information from "default" entity manager
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Address.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Category.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Country.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Customer.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Order.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/OrderProductLine.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Product.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/ProductSale.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/ProductStock.php
  > writing /home/ubuntu/web/symfony-tutorial/src/AppBundle/Entity/Warehouse.php
```

From the doctrine documentation:

> Reverse Engineering is a one-time process that can get you started with a project. Converting an existing database schema into mapping files only detects about 70-80% of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

This means that the `mapping:import` command (and most of automatic code generation tools) should be considered as a helper, not a replacement for you. So we will need some work to get our mapping informations match the database schema.

- Add `options={"unsigned"=true}` to all primary keys.
  > `unsigned` is not standard SQL and few database servers support it. Since we decided to use MySQL for this project, we are going to use, whenever appropriate, specific MySQL features.
- Add `options={"default"=0}` to Category::deleted and ProductSale::active and ProductStock::quantity
- Add `nullable=false` to non nullable foreign keys.

To make sure the mapping matches the database, you can run the command

```
$ app/console doctrine:schema:validate
[Mapping]  OK - The mapping files are correct.
[Database] OK - The database schema is in sync with the mapping files.
```

3.2 Entity accessors

The previous command generated the required entities, but without members accessors. We can also generate setters and getters for all the entities using the following command.

```
$ app/console doctrine:generate:entities AppBundle --no-backup
Generating entities for bundle "AppBundle"
  > generating AppBundle\Entity\Address
  > generating AppBundle\Entity\OrderProductLine
  > generating AppBundle\Entity\Category
  > generating AppBundle\Entity\Product
  > generating AppBundle\Entity\ProductSale
  > generating AppBundle\Entity\Customer
  > generating AppBundle\Entity\ProductStock
  > generating AppBundle\Entity\Warehouse
  > generating AppBundle\Entity\Order
  > generating AppBundle\Entity\Country
```

Again, we will inspect the generated code and adjust it as needed.

- Remove `addCustomer`, `removeCustomer` and `getCustomer` methods from Address entity.
- Remove `addProduct`, `removeProduct` and `getProduct` methods from Category entity.
- Rename the member `category` to `categories` and `getCategory` to `getCategories` in Product entity
- Add `setCategories` method to Product entity

  ```
  /**
  * Add category
  * @param \AppBundle\Entity\Category $categories
  * @return Product
  */
  public function setCategories(
  \Doctrine\Common\Collections\ArrayCollection $categories
  )
  {
    $this->categories = $categories;
    return $this;
  }
  ```

- Change `inversedBy="category"` to `inversedBy="categories"` in the ManyToMany mapping on Category::product
- Change `setParent` to `setParentCategory` in src/AppBundle/DataFixtures/ORM/CategoryFixtures.php L 56
- Change `parent` occurrences to `parentCategory` in src/AppBundle/Form/Type/CategoryType.php
- Add `const REPOSITORY = 'AppBundle:Entity'` to all entities (change `Entity` to the entity name)

We are done.

3.3 Basic CRUD for Country entity

SensioGeneratorBundle is part of Symfony standard edition and provides a command to automatically generate CRUD actions from an entity. Let's see it in action.

```
$ app/console doctrine:generate:crud --with-write --entity=AppBundle:Country \
--format=yml --no-interaction

  CRUD generation

Generating the CRUD code: OK
Generating the Form code: OK
Importing the CRUD routes: OK

  You can now start using the generated code!
```

The command generated the following commands, you are advised to inspect them all

- **src/AppBundle/Controller/CountryController.php** with index, show, new, create, edit, and update actions.
- **src/AppBundle/Form/CountryType.php** Form type
- **src/AppBundle/Resources/config/routing/country.yml** contains routes definition for country actions.
- **src/AppBundle/Resources/views/Country/index|show|edit|new.html.twig** are basic Twig templates to render the different actions.
- **src/AppBundle/Tests/Controller/CountryControllerTest.php** Contains a commented skeleton for a test suite.

- **src/AppBundle/Resources/config/routing.yml** was updated to include the newly created country.yml routes configuration file.

Let's try to see what tests were generated. Edit **src/AppBundle/Tests/Controller/CountryControllerTest.php** and uncomment the only method.
The generated test is just a skeleton and doesn't work out of the box. We need to tweak it a bit.

Update the form creation after `// Fill in the form and submit it` to

```
$form = $crawler->selectButton('Create')->form(array(
    'appbundle_country[code]' => 'Test',
    'appbundle_country[name]' => 'Test',
    'appbundle_country[currency]' => 'Test',
));
```

Update the line 37 to set a `code` field instead of `field_name` `'appbundle_country[code]' => 'Foo',`

We noticed an old test at **src/AppBundle/Tests/Controller/DefaultControllerTest.php**, delete that file. Also delete **src/AppBundle/Controller/DefaultController.php**

Let's run the test suite

```
$ phpunit -c app
PHPUnit 3.7.28 by Sebastian Bergmann.

Configuration read from /home/ubuntu/web/symfony-tutorial/app/phpunit.xml.dist
.

Time: 2.82 seconds, Memory: 18.25Mb

OK (1 test, 4 assertions)
```

To get a coverage report from phpunit, you can add a **--coverage-text** argument

```
$ phpunit -c app --coverage-text
PHPUnit 3.7.28 by Sebastian Bergmann.

Configuration read from /home/ubuntu/web/symfony-tutorial/app/phpunit.xml.dist
.

Time: 8.05 seconds, Memory: 19.25Mb

OK (1 test, 4 assertions)

Code Coverage Report
  2015-10-04 00:24:13

 Summary:
  Classes: 14.29% (3/21)
  Methods: 10.20% (15/147)
  Lines:   12.42% (97/781)

\AppBundle\Controller::CountryController
  Methods: 100.00% (10/10)   Lines:  85.06% ( 74/ 87)
\AppBundle\Entity::Country
  Methods: 100.00% ( 7/ 7)   Lines: 100.00% ( 10/ 10)
\AppBundle\Event\Listener::SoftDelete
  Methods: 100.00% ( 1/ 1)   Lines:  41.67% (  5/ 12)
\AppBundle\Form::CountryType
  Methods: 100.00% ( 3/ 3)   Lines: 100.00% (  8/  8)
```

> What's the deal with functional tests? If you are reading this book, most probably you are familiar with automated testing. A detailed study about testing benefits and techniques is out of the scope of this material. There are plenty of books and on-line resources about the subject. Here we will just comment on our example, and we will continue writing more functional and unit tests.

The CRUD generate command generated a functional tests suite for us, that basically does the following:

- Visited the countries index page
- Created a new country
- Updated the created country
- Deleted the created country

Alternatively, we could fire up a web browser and check manually that the code works as expected by submitting forms and clicking on links. Having automated tests helps us test the application faster, get reproducible results, be able to automatically test any code change and catch bugs earlier in development cycle.

3.4 Override default CRUD templates

The generate:crud command did pretty much a good job. Not perfect though, we can already spot few issues

- We don't want the generated tests to be commented.
- The `testCompleteScenario` is testing all the actions, would be better to split it in separate test methods for each action.

It is easy to update the generated code to meet those requirements. The changes we want to make doesn't seem to be related to the Country entity, and we will probably need them for most of the generated entities. Our approach will be to change the way the code is generated. This way we will have those changes propagate to all the generated actions.

- Create **app/Resources/SensioGeneratorBundle/skeleton/crud/tests/test.php.twig**

```
{% extends "skeleton/crud/tests/test.php.twig" %}

{% block class_body %}
    {%- if 'new' in actions %}
        {%- include 'crud/tests/others/full_scenario.php.twig' -%}
    {%- else %}
        {%- include 'crud/tests/others/short_scenario.php.twig' -%}
    {%- endif %}
{% endblock class_body %}
```

Let's take a quick look at the template we are extending from, is located at **vendor/sensio/generator-bundle/Sensio/Bundle/GeneratorBundle/Resources/skeleton/crud/tests/test.php.twig**

```
<?php

namespace {{ namespace }}\Tests\Controller{{ entity_namespace ? '\\'~entity_namespace:''}}

{% block use_statements %}
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
{% endblock use_statements %}

{% block class_definition %}
class {{ entity_class }}ControllerTest extends WebTestCase
{% endblock class_definition %}
{
{% block class_body %}
    /*

{%- if 'new' in actions %}
    {%- include 'crud/tests/others/full_scenario.php.twig' -%}
{%- else %}
    {%- include 'crud/tests/others/short_scenario.php.twig' -%}
{%- endif %}

    */
{% endblock class_body %}
}
```

We are just overriding the **class_body** block and removed the comment tags `/* */`. To see the result of this change, we need to run the generate command again. Now we need to add the **--overwrite** option so it won't complain about existing files.

```
$ app/console doctrine:generate:crud --with-write --entity=AppBundle:Country \
 --format=yml --no-interaction --overwrite


  CRUD generation


Generating the CRUD code: OK
Generating the Form code: Already exists, skipping
Importing the CRUD routes: FAILED


  The command was not able to configure everything automatically.
  You must do the following changes manually.


- Import the bundle's routing resource in the bundle routing file
  (/home/ubuntu/web/symfony-tutorial/src/AppBundle/Resources/config/routing.yml).

    AppBundle_country:
        resource: "@AppBundle/Resources/config/routing/country.yml"
        prefix:   /country
```

If you inspect **src/AppBundle/Tests/Controller/CountryControllerTest.php** you will notice that the test is not commented anymore.

We are not done yet. We need to generate several tests for each action. We will override the **full_scenario.php.twig** template.

- Create **app/Resources/SensioGeneratorBundle/skeleton/crud/tests/others/full_scenario.php.twig**

```
{%- set formType =  form_type_name|lower %}

public function testIndexAction()
{
    $client = static::createClient();
    $crawler = $client->request('GET', '/{{ route_prefix }}/');
    $unexpectedStatus = "Unexpected HTTP status code for GET /{{ route_prefix }}/";
    $this->assertEquals(200, $client->getResponse()->getStatusCode(), $unexpectedStatus);
}

public function testNewAction()
{
    $client = static::createClient();
    $crawler = $this->createNewEntity($client, '/{{route_prefix}}/', '{{formType}}');

    foreach ($this->getNewFormFields() as $value) {
        $missingElementError = sprintf('Missing element td:contains("%s")', $value);
        $valueCount = $crawler->filter(sprintf('td:contains("%s")', $value))->count();
        $this->assertGreaterThan(0, $valueCount, $missingElementError);
    }
}

public function testUpdateAction()
{
    $client = static::createClient();
    $crawler = $this->createNewEntity($client, '/{{route_prefix}}/', '{{formType}}');

    $crawler = $client->click($crawler->selectLink('Edit')->link());

    $formFields = $this->getUpdateFormFields();

    $form = $crawler->selectButton('Update')->form(
        $this->constructFormValues($formFields, '{{formType}}')
    );

    $client->submit($form);
    $crawler = $client->followRedirect();

    foreach ($this->getUpdateFormFields() as $value) {
        $missingElementError = sprintf('Missing element td:contains("%s")', $value);
        $valueCount = $crawler->filter(sprintf('[value="%s"]', $value))->count();
        $this->assertGreaterThan(0, $valueCount, $missingElementError);
    }
}
```

```
/**
 *
 * @return array 'field_name'=>'value' to be used for testNewAction
 */
protected function getNewFormFields()
{
    return array(
        'field_name' => 'value',
        //other fields
    );
}
/**
 *
 * @return array 'field_name'=>'value' to be used for testUpdateAction
 */
protected function getUpdateFormFields()
{
    return $this->getNewFormFields();
}

protected function constructFormValues($fields, $formTypeName)
{
    $values = array();
    foreach ($fields as $fieldName => $value) {
        $values[sprintf('%s[%s]', $formTypeName, $fieldName)] = $value;
    }
    return $values;
}

protected function createNewEntity(Client $client, $routePrefix, $formTypeName)
{
    $crawler = $client->request('GET', $routePrefix);
    $crawler = $client->click($crawler->selectLink('Create a new entry')->link());

    $form = $crawler->selectButton('Create')->form(
            $this->constructFormValues($this->getNewFormFields(), $formTypeName)
    );

    $client->submit($form);
    return $client->followRedirect();
}
```

> Although this is a template to generate code, we must think about the maintainability of the generated code as well. We declared some reusable methods as protected. We are preparing to move those methods to a parent class and override them whenever required.

The method `createNewEntity` expects an object of type Symfony\Component\HttpKernel\Client. We will add the required use statement to the **test.php.twig** template.

- Edit **app/Resources/SensioGeneratorBundle/skeleton/crud/tests/test.php.twig** add the following block right after the `extends` tag

```
{% block use_statements %}
{{ parent() }}
use Symfony\Component\HttpKernel\Client;
{% endblock use_statements %}
```

Execute the generate:crud command again and inspect the generated CountryControllerTest. It should look better than before. All we need to do now is to update the return value of `getNewFormFields` to return some valid values.

```
protected function getNewFormFields()
{
    return array(
        'code' => 'code',
        'name' => 'name',
        'currency' => 'eur',
    );
}
```

Let's run the tests again and see what happens.

```
$ phpunit -c app
PHPUnit 3.7.28 by Sebastian Bergmann.

Configuration read from /home/ubuntu/web/symfony-tutorial/app/phpunit.xml.dist
...

Time: 2.86 seconds, Memory: 18.25Mb

OK (3 tests, 7 assertions)
```

Perfect! Now we have 3 tests and 7 assertions instead of one test and 4 assertions. The more our tests are granular, the easier it will be to spot problems if a test fails. To know that **testUpdateAction** failed is definitely better than knowing that **testCompleteScenario** failed.

3.5 Pagination

Whenever displaying lists of data, you should limit the number of records to show. Ideally, you should limit the number of records you retrieve at once from the data storage. Failing to do so can have undesirable consequences like crashing the user's browser, unnecessary usage of the network bandwidth, and eventually putting your data storage to do more work than required.

Let's implement pagination for the Country index action.

- Edit **src/AppBundle/Controller/CountryController.php** and update `indexAction` as following

```php
<?php
/**
 * Lists Country entities per page.
 *
 */
public function indexAction($page = 1)
{
    $recordsPerPage = 10;
    $manager = $this->getDoctrine()->getManager();
    $dql = sprintf('SELECT e from %s e', Country::REPOSITORY);
    $query = $manager->createQuery($dql)
            ->setFirstResult(($page - 1) * $recordsPerPage)
            ->setMaxResults($recordsPerPage);

    $paginator = new \Doctrine\ORM\Tools\Pagination\Paginator($query, false);

    return $this->render('AppBundle:Country:index.html.twig', array(
                'entities' => $paginator->getIterator(),
                'currentPage' => $page,
                'totalPages' => ceil($paginator->count()/$recordsPerPage)
    ));
}
```

We passed `false` as second argument ($fetchJoinCollection) to the `Paginator` constructor. From the Doctrine documentation, we can read

> By default the pagination extension does the following steps to compute the correct result:
>
> - Perform a Count query using DISTINCT keyword.
> - Perform a Limit Subquery with DISTINCT to find all ids of the entity in from on the current page.
> - Perform a WHERE IN query to get all results for the current page. This behavior is only necessary if you actually fetch join a to-many collection. You can disable this behavior by setting the $fetchJoinCollection flag to false;

In simple words, there are two ways to retrieve the data from the database.

- Fetch all the columns from the database applying a limit and an offset

```
SELECT * FROM country ORDER BY id LIMIT 10 OFFSET 0;
```

- Fetch only the primary keys from the database applying a limit and an offset, then fetch all the columns corresponding to those primary keys

```
SELECT id FROM country ORDER BY id LIMIT 10 OFFSET 0
SELECT * FROM country WHERE id in (1,2,3,4,5,6,7,8,9,10)
```

  There is no general answer to the question "which method is better?". The performance of the query depends on many factors, like the query itself, the used storage engine, the schema structure, the dataset volume and distribution.. amnog others. You must benchmark and profile every query in order to decide which approach works better in your situation. Don't forget that every situation is a special case.

As an example, let's profile the possible queries to see the difference.

```
# query 1

mysql> explain SELECT * FROM country ORDER BY id LIMIT 10 OFFSET 0;
+--+-----------+---------+-------+---------------+---------+-------+----+------+-------+
|id|select_type| table   | type  | possible_keys | key     |key_len|ref | rows | Extra |
+--+-----------+---------+-------+---------------+---------+-------+----+------+-------+
| 1| SIMPLE    | country | index | NULL          | PRIMARY | 2     |NULL|   10 |       |
+--+-----------+---------+-------+---------------+---------+-------+----+------+-------+
```

```
# query 2

mysql> explain SELECT id FROM country ORDER BY id LIMIT 10 OFFSET 0;
+--+-----------+---------+-------+---------------+---------+-------+----+----+-----------+
|id|select_type| table   | type  | possible_keys | key     |key_len|ref |rows| Extra     |
+--+-----------+---------+-------+---------------+---------+-------+----+----+-----------+
| 1| SIMPLE    | country | index | NULL          | PRIMARY | 2     |NULL| 10 |Using index|
+--+-----------+---------+-------+---------------+---------+-------+----+----+-----------+

# query 3

mysql> explain SELECT * FROM country WHERE id in (1,2,3,4,5,6,7,8,9,10);
+--+-----------+---------+-------+---------------+---------+-------+----+----+-----------+
|id|select_type| table   | type  | possible_keys | key     |key_len|ref |rows| Extra     |
+--+-----------+---------+-------+---------------+---------+-------+----+----+-----------+
| 1| SIMPLE    | country | range | PRIMARY       | PRIMARY | 2     |NULL| 10 |Using where|
+--+-----------+---------+-------+---------------+---------+-------+----+----+-----------+
```

- query 1 uses a **type** `index` without **extra** which means MySQL is planning to perform a full table scan.
- query 2 uses a **type** `index` with an **extra** `Using index` which means MySQL is planning to scan only the index tree. When using InnoDB storage engine, this can result in less IO work and better performance since InnoDB uses separate files to store indexes.
- query 3 uses a **type** `range` which means that MySQL is going to read rows from specific ranges of the id key.

In our situation, the Country table is small enough and will fit in memory. I don't think even a third world war would reshape the world as we know into millions or even thousands of countries.

We are done with the controller, let's update the **country** route configuration to support a **page** parameter

- Edit **src/AppBundle/Resources/config/routing/country.yml** and update the **country** route as following

```
country:
  path:     /{page}
  defaults: { _controller: "AppBundle:Country:index", page:1 }
  requirements:
      page:  \d+
```

Now we need to update the view to be able to navigate to pages.

- Edit **src/AppBundle/Resources/views/Country/index.html.twig** and add the navigation table after the last `ul` (right before the closing block tag)

```
<table>
  <tr>
      <td>Navigation</td>
      {% for page in 1..totalPages %}
          <td>
              <a href="{{ path('country', { 'page': page }) }}">{{page}}</a>
          </td>
      {% endfor %}
  </tr>
</table>
```

We are done, before checking the result of our work, let's run our tests suite to make sure we didn't broke anything.

```
$ phpunit -c app
PHPUnit 3.7.28 by Sebastian Bergmann.

Configuration read from /home/ubuntu/web/symfony-tutorial/app/phpunit.xml.dist

FEE

Time: 3.07 seconds, Memory: 16.00Mb

There were 2 errors:

1) AppBundle\Tests\Controller\CountryControllerTest::testNewAction
InvalidArgumentException: The current node list is empty.

/.../vendor/symfony/symfony/src/Symfony/Component/DomCrawler/Crawler.php:750
/.../src/AppBundle/Tests/Controller/CountryControllerTest.php:88
/.../src/AppBundle/Tests/Controller/CountryControllerTest.php:23

2) AppBundle\Tests\Controller\CountryControllerTest::testUpdateAction
InvalidArgumentException: The current node list is empty.

/.../vendor/symfony/symfony/src/Symfony/Component/DomCrawler/Crawler.php:750
/.../src/AppBundle/Tests/Controller/CountryControllerTest.php:88
/.../src/AppBundle/Tests/Controller/CountryControllerTest.php:35

--

There was 1 failure:

1) AppBundle\Tests\Controller\CountryControllerTest::testIndexAction
Unexpected HTTP status code for GET /country/
Failed asserting that 404 matches expected 200.

/.../src/AppBundle/Tests/Controller/CountryControllerTest.php:17

FAILURES!
```

Well, seems that we broke something! Indeed, we changed the **country** route path from `/` to `/{page}` that means the url /**country**/ is not a matched route anymore. This is our first experience during this journey to catch a bug early. If we would have static links to **http://symfony.local/country/** we would need to update them. For the moment, we just need to update our test and change occurrences of `/country/` to `/country`.

3.6 Exercises

3.6.1 The pagination feature we added to the Country::indexAction was awesome. We can already imagine that we will need it for most of our entities. We will extend the crud generator to generate the indexAction with pagination already implemented. Exactly like we did with Country::indexAction. So basically we need to reproduce what we did with Country, but in a more generic way. You will need to override the following skeletons:

- vendor/sensio/generator-bundle/Sensio/Bundle/GeneratorBundle/Resources/skeleton/crud/actions/index.php.twig
- vendor/sensio/generator-bundle/Sensio/Bundle/GeneratorBundle/Resources/skeleton/crud/config/routing.yml.twig
- vendor/sensio/generator-bundle/Sensio/Bundle/GeneratorBundle/Resources/skeleton/crud/views/index.html.twig.twig And update app/Resources/SensioGeneratorBundle/skeleton/crud/tests/others/full_scenario.php.twig to fix the failing test (the issue with the path /country/)

3.6.2 The generated test suite is very helpful. It needs some annoying manual maintenance though. The method `getNewFormFields` looks like

```
protected function getNewFormFields()
{
    return array(
        'field_name' => 'value',
        //other fields
    );
}
```

We want to automate this part so the generated method would look like

```
protected function getNewFormFields()
{
    return array(
        'code' => 'txt_c',
        'name' => 'txt_name',
        'currency' => 'txt',
    );
}
```

> Note that the values are truncated to match the column length.

To build such an array, we definitely need the mapping metadata. We can't solve this issue just by overriding a skeleton. To understand how the mechanism works, take a look at `Sensio\Bundle\GeneratorBundle\Command\GenerateDoctrineCrudCommand::createGenerator()` is returning an instance of `Sensio\Bundle\GeneratorBundle\Generator::DoctrineCrudGenerator`. We are interested in a particular method from the later class, `generateTestClass` doesn't pass any metadata informations to the rendered template `crud/tests/test.php.twig`.

Besides overriding **crud/tests/test.php.twig** template, you will need to extend `Sensio\Bundle\GeneratorBundle\Command\GenerateDoctrineCrudCommand` (maybe with the name `app:generate:crud`) and alter it's behavior so it passes metadata to the rendered template.

Make sure the tests pass.