3. Working with the database

We are going to be working on the repository that you have created up to this point. If you skipped ahead, don't worry. You can clone the repository from its Github repository.

```
$ git clone git@github.com:skafandri/symfony-tutorial.git --branch ch2
Cloning into 'symfony-tutorial'...
remote: Counting objects: 57, done.
remote: Total 57 (delta 0), reused 0 (delta 0), pack-reused 57
Receiving objects: 100% (57/57), 46.98 KiB | 0 bytes/s, done.
Checking connectivity... done.
```

Don't forget to `composer install`.

3.1 Database design

3.1.1 Introduction

A study about database design and data structures deserves many books on itself. Therefore, we will try to cover the most common good design practices that can serve as a starting point when designing your application's requirements data structures. Database desing is a very important step in software development lifecycle because any design decision can dramatically affect the project's performance and maintainance cost.
At this point we are going to use an RDBMS (Relational database management system). We will talk about other storages engines like noSql later.

When designing a database schema, there are two major questions you should ask yourself:

1. How I am going to query the data?
   Consider the follwing simple table

   ```
   `user_account` (
   `id` INT(10),
   `login` VARCHAR(45) NULL DEFAULT NULL,
   `password` VARCHAR(100) NULL DEFAULT NULL,
   `email` VARCHAR(45) NULL DEFAULT NULL,
   PRIMARY KEY (`id`),
   ENGINE = InnoDB;
   ```

   Knowing that I will execute queries like `SELECT * from user_account WHERE email like "search_pattern%"` I may think about adding an **index** on user_account.email column to speed up this select query. However, if I know that most of the queries will be like `SELECT * from user_account WHERE email like "%search_pattern%"`, adding an index will have no effect except increasing the table index size.

2. How the data will be distributed?
   How many records this table will contain? How many records will have a NULL value in this column? What cardinality this column is going to have? Accurately predicting answers to those questions can be a tricky task. I've seen tables designed to hold *no more than 10 records* that ended up with thousands of entries.

When comparing design variants, there are three metrics that interests the application developer: read speed, update speed, and data storage size. Usually the last one is the least of your concerns, although it can be very important in some situations. You often need to make a balance between read and update speed depending on your application's needs.

An example comparaison:
Benchmark with 5000000 records

Variant	Avg SELECT speed	Avg UPDATE speed	Index size
With index	50 ms	800 ms	250 MB
Without index	400 ms	200 ms	0

There is absolutely no generic pattern on making a decision in this case. It solely depends on your specific application's needs.

3.1.2 Data schema

We are still adopting the target bullet process. That means our first database design is not final and we will eventually update it as needed while going through implementing new features.

For this tutorial, we are going tu use MySQL database server. That said, the SQL syntax will be MySQL.

- Create the schema

```
CREATE SCHEMA IF NOT EXISTS `symfony` DEFAULT CHARACTER SET utf8;
```

- Category table

```
CREATE TABLE IF NOT EXISTS `symfony`.`category` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `label` VARCHAR(45) NULL,
  `parent_category_id` INT UNSIGNED NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_category_category_idx` (`parent_category_id` ASC),
  CONSTRAINT `fk_category_category`
    FOREIGN KEY (`parent_category_id`)
    REFERENCES `symfony`.`category` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

> Whenever you can, take advantage of the dabase server new ID generation strategy for your primary keys (AUTO_INCREMENT for MySQL) You may be tempted to develop your own generation strategy (example: pattern based IDs like "day_date_sequential_number"), don't. You can have such a column in your table but keep it separate from table **id** key.
>
> When using auto generated keys, make them **UNSIGNED**. The values will be all positive (by default) so there is no point of deviding your values range by 2. For example the INT type for MySQL has the follwing ranges:
>
> | | **Minumum value** | **Maximum value** |
> |---|---|---|
> | Signed | -2147483648 | 2147483647 |
> | Unsigned | 0 | 4294967295 |

- product table

```
CREATE TABLE IF NOT EXISTS `symfony`.`product` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `code` VARCHAR(45) NULL DEFAULT NULL,
  `title` VARCHAR(200) NULL DEFAULT NULL,
  `description` TEXT NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  INDEX `code_index` (`code` ASC),
  INDEX `title_index` (`title` ASC))
ENGINE = InnoDB;
```

- a product belongs to many categories

```
CREATE TABLE IF NOT EXISTS `symfony`.`category_has_product` (
  `category_id` INT(10) UNSIGNED NOT NULL,
  `product_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`category_id`, `product_id`),
  INDEX `fk_category_has_product_product_idx` (`product_id` ASC),
  INDEX `fk_category_has_product_category_idx` (`category_id` ASC),
  CONSTRAINT `fk_category_has_product_category`
    FOREIGN KEY (`category_id`)
    REFERENCES `symfony`.`category` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_category_has_product_product`
    FOREIGN KEY (`product_id`)
    REFERENCES `symfony`.`product` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci;
```

- warehouse table

```
CREATE TABLE IF NOT EXISTS `symfony`.`warehouse` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL DEFAULT NULL,
  `address` TEXT NULL DEFAULT NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;
```

- a product can be present in many warehouses but with different quantities

```
CREATE TABLE IF NOT EXISTS `symfony`.`warehouse_has_product` (
  `warehouse_id` INT(10) UNSIGNED NOT NULL,
  `product_id` INT(10) UNSIGNED NOT NULL,
  `quantity` INT(11) NULL DEFAULT NULL,
  PRIMARY KEY (`warehouse_id`, `product_id`),
  INDEX `fk_warehouse_has_product_product_idx` (`product_id` ASC),
  INDEX `fk_warehouse_has_product_warehouse_idx` (`warehouse_id` ASC),
  CONSTRAINT `fk_warehouse_has_product_warehouse`
    FOREIGN KEY (`warehouse_id`)
    REFERENCES `symfony`.`warehouse` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_warehouse_has_product_product1`
    FOREIGN KEY (`product_id`)
    REFERENCES `symfony`.`product` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION) ENGINE = InnoDB;
```

- a product can be put in sale for a specific period with a given sale price

```
CREATE TABLE IF NOT EXISTS `symfony`.`product_sale` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `active` TINYINT(4) NULL DEFAULT NULL,
  `start_date` DATETIME NULL DEFAULT NULL,
  `end_date` DATETIME NULL DEFAULT NULL,
  `price` INT(11) NULL DEFAULT NULL,
  `product_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_product_sale_product_idx` (`product_id` ASC),
  INDEX `start_date_index` (`start_date` ASC),
  INDEX `end_date_index` (`end_date` ASC),
  CONSTRAINT `fk_product_sale_product`
    FOREIGN KEY (`product_id`)
    REFERENCES `symfony`.`product` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION) ENGINE = InnoDB;
```

> When working with MySQL prefer DATETIME above TMESTAMP. TMESTAMP has nothing more than DATETIME, however it has some drowbacks like having a maximum value of 2038-01-09 03:14:07 UTC

- vendor table

```
CREATE TABLE IF NOT EXISTS `symfony`.`vendor` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci;
```

- product_acquisition table

```
CREATE TABLE IF NOT EXISTS `symfony`.`product_acquisition` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `date` DATETIME NULL DEFAULT NULL,
  `quantity` INT(11) NULL DEFAULT NULL,
  `price` INT(11) NULL DEFAULT NULL,
  `vendor_id` INT(10) UNSIGNED NOT NULL,
  `product_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_product_acquisition_vendor_idx` (`vendor_id` ASC),
  INDEX `fk_product_acquisition_product_idx` (`product_id` ASC),
  INDEX `date_index` (`date` ASC),
  CONSTRAINT `fk_product_acquisition_vendor`
    FOREIGN KEY (`vendor_id`)
    REFERENCES `symfony`.`vendor` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_product_acquisition_product`
    FOREIGN KEY (`product_id`)
    REFERENCES `symfony`.`product` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

> Use Integer types for money. You may be tempted to use database's money type or floating point numbers to store money values, don't. At best you are coupling yourself to a specific database and/or a specific platform. At worst you may have precision leacks that may be extremly hard to debug.

- contact table

```
CREATE TABLE IF NOT EXISTS `symfony`.`contact` (
  `id` INT(10) 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;
```

- account table

```sql
CREATE TABLE IF NOT EXISTS `symfony`.`account` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `login` VARCHAR(45) NULL DEFAULT NULL,
  `password` VARCHAR(100) NULL DEFAULT NULL,
  `email` VARCHAR(45) NULL DEFAULT NULL,
  `active` TINYINT(4) NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  INDEX `login_index` (`login` ASC),
  INDEX `password_index` (`password` ASC))
ENGINE = InnoDB;
```

- customer table

```sql
CREATE TABLE IF NOT EXISTS `symfony`.`customer` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `account_id` INT(10) UNSIGNED NOT NULL,
  `contact_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_customer_account_idx` (`account_id` ASC),
  INDEX `fk_customer_contact_idx` (`contact_id` ASC),
  CONSTRAINT `fk_customer_account`
    FOREIGN KEY (`account_id`)
    REFERENCES `symfony`.`account` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_customer_contact`
    FOREIGN KEY (`contact_id`)
    REFERENCES `symfony`.`contact` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci;
```

- country table

```sql
CREATE TABLE IF NOT EXISTS `symfony`.`country` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `code` VARCHAR(5) NULL DEFAULT NULL,
  `name` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;
```

- address table

```
CREATE TABLE IF NOT EXISTS `symfony`.`address` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `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(4) NULL DEFAULT 0,
  `country_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_address_country_idx` (`country_id` ASC),
  CONSTRAINT `fk_address_country`
    FOREIGN KEY (`country_id`)
    REFERENCES `symfony`.`country` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

> **deleted** column will be used to achive what is called a *soft delete*. That is to update the deleted column value instead of issuing a DELETE query.

- a customer can have many addresses

```
CREATE TABLE IF NOT EXISTS `symfony`.`customer_has_address` (
  `customer_id` INT(10) UNSIGNED NOT NULL,
  `address_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`customer_id`, `address_id`),
  INDEX `fk_customer_has_address_address_idx` (`address_id` ASC),
  INDEX `fk_customer_has_address_customer_idx` (`customer_id` ASC),
  CONSTRAINT `fk_customer_has_address_customer`
    FOREIGN KEY (`customer_id`)
    REFERENCES `symfony`.`customer` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_customer_has_address_address`
    FOREIGN KEY (`address_id`)
    REFERENCES `symfony`.`address` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

- order table

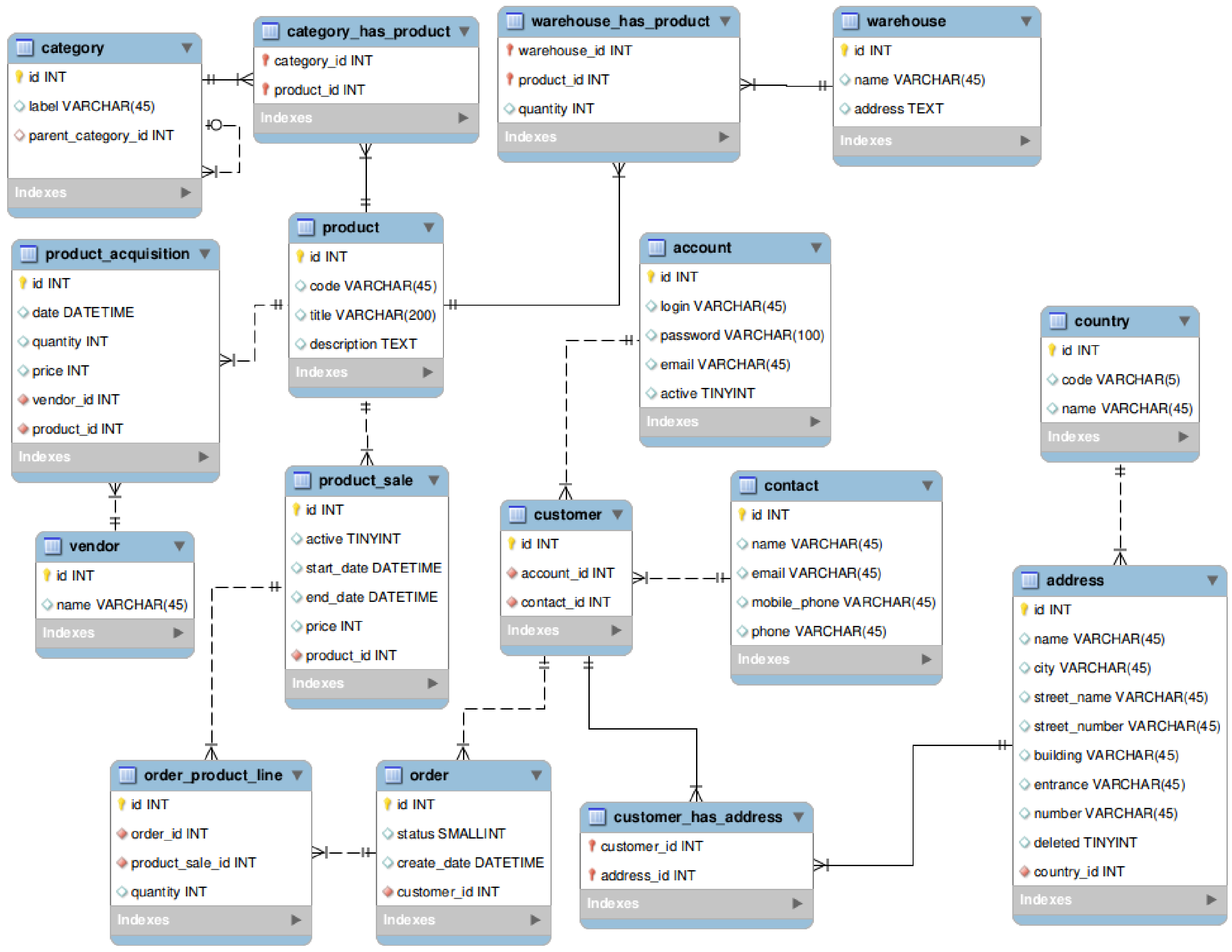
```
CREATE TABLE IF NOT EXISTS `symfony`.`order` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `status` SMALLINT(5) UNSIGNED NULL DEFAULT NULL,
  `create_date` DATETIME NULL DEFAULT NULL,
  `customer_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_order_customer_idx` (`customer_id` ASC),
  CONSTRAINT `fk_order_customer`
    FOREIGN KEY (`customer_id`)
    REFERENCES `symfony`.`customer` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci;
```

> Don't add indexes on low cardinality columns. For the moment, we can think of a handful *status* an order can have (new, delivered, cancelled..). We can also have a long sight and think about dozens of possible values. It is very unlinkly however, that an order can have thousands or even hundreds of possible status. Adding an index on this column will increase INSERT and UPDATE queries execution time without noticeably speeding up the SELECT queries.

- an order has many product_sales with different quantities

```
CREATE TABLE IF NOT EXISTS `symfony`.`order_product_line` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `order_id` INT(10) UNSIGNED NOT NULL,
  `product_sale_id` INT(10) UNSIGNED NOT NULL,
  `quantity` INT(11) NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_order_product_line_order_idx` (`order_id` ASC),
  INDEX `fk_order_product_line_product_sale_idx` (`product_sale_id` ASC),
  CONSTRAINT `fk_order_product_line_order`
    FOREIGN KEY (`order_id`)
    REFERENCES `symfony`.`order` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_order_product_line_product_sale`
    FOREIGN KEY (`product_sale_id`)
    REFERENCES `symfony`.`product_sale` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

A global view of the database schema



3.2 Doctrine

3.2.1 ORM

An ORM (Object Relational Mapping) abstracts the data table details and specific database implementations. It exposes objects that the programmer can use like any traditionnal object. For this course, we are going to use Doctrine ORM. Doctrine ORM is the default ORM that comes with the standard Symfony application.

Before proceeding make sure you created the database structure. Also make sure that your database details in app/config/parameters.yml are correct.

3.2.2 Command line

The doctrine bundle comes with a set of commands to help you increase your productivity. We will start by importing the database schema we just created into our application.

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

The mapping informations were generated as annotations because we invoked the command with format argument **annotation**. You can delete the generated entities and experiment importing mapping informations with different formats (yaml, xml, PHP) to see the differences. I recommand using annotations for mapping informations because it gives a greater code readibility.

> Use annotations for mapping informations.

The generated entities are simple classes with private members. We can also automatically generate members accessors for those entities.

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

> Be careful when using doctrine code generators tools, and any code generators in general. Those tools can be very helpful for better productivity. You should make sure however, to perfectly understand the generated code. When problems will appear (they usually do) you will prefer to debug a code you understand rather than one that you don't.

Based on our database schema, Doctrine generated 13 entities. Take some time to inspect the generated classes, there is no reason you would find difficulties understanding the code.

3.2.3 Working with data

Let's update our controller to fetch the data from the database.

- Edit **src**/**AppBundle**/**Controller**/**CatalogController.php** and update getCategories method

```
private function getCategories()
{
    $entityManager = $this->getDoctrine()->getManager();
    $categoryRepository = $entityManager->getRepository('AppBundle:Category');
    $categories = $categoryRepository->findAll();

    return $categories;
}
```

`$entityManager->getRepository('AppBundle:Category')` returns an entity repository. An entity repository provides easy access to basic data retrieval operations using simple filters. When we will need more complex operations, we will create our custom entity repositories.
'AppBundle:Category' is the repository name according to doctrine bundle naming convention ('BundleName:EntityName'). If a string is between quotes in source code, it is a constant. So we will define it as a constant and reference it whenever we need it.

- Edit **src**/**AppBundle**/**Entity**/**Category.php** and add the following line after the class definition

  ```
  const REPOSITORY = 'AppBundle:Category';
  ```

- Edit **src**/**AppBundle**/**Controller**/**CatalogController.php** and change in getCategories

  ```
  $categoryRepository = $entityManager->getRepository('AppBundle:Category');
  ```
  to

  ```
  $categoryRepository = $entityManager->getRepository(Category::REPOSITORY);
  ```
  Don't forget to add a use statement for the Category class

  ```
  use AppBundle\Entity\Category;
  ```

Let's insert some data in the database to have something visible.

```
INSERT INTO `symfony`.`category` (`id`, `label`) VALUES ('1', 'Computers');
INSERT INTO `symfony`.`category` (`id`, `label`) VALUES ('2', 'Phones');
INSERT INTO `symfony`.`category` (`id`, `label`) VALUES ('3', 'Tablets');
INSERT INTO `symfony`.`category` (`id`, `label`, `parent_category_id`)
                                  VALUES ('4', 'Desktop', '1');
INSERT INTO `symfony`.`category` (`id`, `label`, `parent_category_id`)
                                  VALUES ('5', 'Laptop', '1');
```

Now if you visit http://127.0.0.1:8000/catalog/category/list you should see the same list of categories but now they are fetched from the database.

If you visit http://127.0.0.1:8000/catalog/category/show/1 you should see an error message.

```
Method "parent" for object "AppBundle\Entity\Category"
does not exist in catalog/category/show.html.twig at line 14
```

That's because when we changed the data structure that the controller used to send to TWIG templates. We will just update our templates to deal with the new format.

- Edit **app/Resources/views/catalog/category/show.html.twig** and replace `category.parent` with `category.parentCategory`

Now if you visit http://127.0.0.1:8000/catalog/category/show/1 you should see it working.

We need to fix the edit view the same way

- Edit **app/Resources/views/catalog/category/edit-ajax.html.twig** and replace `category.parent` with `category.parentCategory`

Now if you visit http://127.0.0.1:8000/catalog/category/edit/1 you should see it working.

One more thing to fix, the tree view.

- Edit **src/AppBundle/Controller/CatalogController.php** and update the buildTree method

```
private function buildTree(array $categories, $parentId = null)
{
    $tree = array();
    foreach ($categories as $category) {
        $parentNode = !$parentId && !$category->getParentCategory();
        $childNode = $parentId && $category->getParentCategory() &&
                $category->getParentCategory()->getId() === $parentId;
        if ($parentNode || $childNode) {
            $children = $this->buildTree($categories, $category->getId());
            $tree[$category->getId()] = array(
              'category' => $category,
              'children' => $children
              );
        }
    }
    return $tree;
}
```

- Update **app/Resources/views/catalog/category/tree-node.html.twig**

```
{% set children = category.children %}
{% set category = category.category %}

{% set showUrl = path(
    'catalog_category_show', {'categoryId': category.id})
%}

{% set editUrl = path(
    'catalog_category_edit_ajax', {'categoryId': category.id})
%}
<li>
    <div style="float: left">
        <a href="{{ showUrl }}">{{ category.label }}</a>
    </div>
    <div style="float: right">
        <a href="{{ editUrl }}" class="ui-button button link-dialog">Edit</a>
    </div>
    {% if children is not empty %}
        <ul style="width: 200px">
            {% for child in children %}
                {% include 'catalog/category/tree-node.html.twig'
                  with {category:child} %}
            {% endfor %}
        </ul>
    {% endif %}
    <div style="clear: both"></div>
</li>
```

Now the tree view http://127.0.0.1:8000/catalog/category/tree should be working.

3.3 CRUD

3.3.1 Introduction

In the last chapter we created views, routes and controller actions to manipulate categories. In this chapter we even made it partially work with the database. CRUD (create, read, update and delete) operations implementation is usually straight forward. Is just kind of boring because is repetitive.

Next, we will work with CRUD generation command. But first, let's clean up the files we don't need anymore.

Before you proceed, commit any uncommited changes.

- Remove **app/Resources/views/catalog**
- Remove **src/AppBundle/Resources/config/routing/catalog.yml**
- Remove **src/AppBundle/Controller/CatalogController.php**
- Edit **app/config/routing.yml**

  ```
  app:
    resource: "@AppBundle/Resources/config/routing.yml"
  ```

- Edit **app/Resources/views/common/app.html.twig** remove the link in the *footer* block

What happens if I don't delete those files? If you don't remove those files nothing will happen, your code will execute just fine. However, leaving unused files is a very bad practice. In time, you can end up with hundreds or thousands of unused files, and it will become harder and harder to find them.

Once I started working on a project that consists in porting a piece of software from one platform to another. The code base was in c++, had about 50.000 lines of code spread through about 100 files. Before starting studying the software behaviour and internals, I did some *cleaning*, including unused files, commented code, and unreachable code.

I ended up with about 35.000 lines of code and 70 files. I just saved myself the pain of trying to understand about 15.000 lines of code that were never executed.

> Never leave unused files in your working tree

3.3.2 CRUD templates

The **doctrine:generate:crud** can generate: views, routes and actions. Every generated file is based on a template. There are default templates but they can be overridden by placing custom templates in one of the following locations, by order of priority:

```
BUNDLE_PATH/Resources/SensioGeneratorBundle/skeleton/crud
APP_PATH/Resources/SensioGeneratorBundle/skeleton/crud
```

To make our view fit in the design we adopted, we will define new views templates.

- **app/Resources/SensioGeneratorBundle/skeleton/crud/views/index.html.twig.twig**

```
{% block extends %}
    {{ "{% extends 'common/app.html.twig' %}" }}
{% endblock extends %}
{% block body %}
    {{ "{% block header%}" }}<h1>{{ entity }} list</h1>{{ "{% endblock %}" }}
    {{ "{% block content%}" }}
    <table class="records_list">
        <thead>
            <tr>
                {%- for field, metadata in fields %}
                <th>{{ field|capitalize }}</th>
                    {%- endfor %}
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            {{ '{% for entity in entities %}' }}
            <tr>
                {%- for field, metadata in fields %}
                {%- if loop.first and ('show' in actions) %}
                <td>
                    <a href="{{ "{{ path('" ~ route_name_prefix ~
                        "_show', { 'id': entity."~ identifier ~" }) }}" }}">
                        {{ '{{ entity.' ~ field|replace({'_': ''}) ~ ' }}' }}</a>
                </td>
                {%- elseif metadata.type in ['date', 'datetime'] %}
                <td>{{ '{% if entity.' ~ field|replace({'_': ''}) ~ ' %}
                            {{ entity.' ~ field|replace({'_': ''}) ~
                            '|date(\'Y-m-d H:i:s\') }}
                        {% endif %}' }}</td>
                {%- else %}
                <td>{{ '{{ entity.' ~ field|replace({'_': ''}) ~ ' }}' }}</td>
                {%- endif %}
                {%- if loop.last %}
                <td>
                    {%- include "crud/views/others/actions.html.twig.twig" %}
                </td>
                {%- endif %}
                {%- endfor %}
            </tr>
            {{ '{% endfor %}' }}
        </tbody>
    </table>
    {% if 'new' in actions %}
        <a class="button" href="{{ "{{ path('" ~ route_name_prefix ~ "_new') }}" }}">
            New {{ entity }}
        </a>
    {% endif %}
    {{ "{% endblock %}" }}
{% endblock body %}
```

- **app/Resources/SensioGeneratorBundle/skeleton/crud/views/new.html.twig.twig**

```
{% block extends %}
    {{ "{% extends 'common/app.html.twig' %}" }}
{% endblock extends %}

{% block body %}
    {{ "{% block header%}" }}
    <h1>New {{ entity }}</h1>
    {{ "{% endblock %}" }}

    {{ "{% block content %}" }}

    {{ '{{ form(form) }}' }}

    {% set hide_edit, hide_delete = true, true %}
    {% include 'crud/views/others/record_actions.html.twig.twig' %}
    {{ "{% endblock %}" }}
{% endblock body %}
```

- **app/Resources/SensioGeneratorBundle/skeleton/crud/views/edit.html.twig.twig**

```
{% block extends %}
    {{ "{% extends 'common/app.html.twig' %}" }}
{% endblock extends %}

{% block body %}
    {{ "{% block header %}" }}
    <h1>{{ entity }} edit</h1>
    {{ "{% endblock %}" }}

    {{ "{% block content %}" }}

    {{ '{{ form(edit_form) }}' }}

    {% set hide_edit, hide_delete = true, false %}
    {% include 'crud/views/others/record_actions.html.twig.twig' %}
    {{ "{% endblock %}" }}
{% endblock body %}
```

- **app/Resources/SensioGeneratorBundle/skeleton/crud/views/show.html.twig.twig**

```
{% block extends %}
    {{ "{% extends 'common/app.html.twig' %}" }}
{% endblock extends %}

{% block body %}
    {{ "{% block header%}" }}
    <h1>{{ entity }}</h1>
    {{ "{% endblock %}" }}

    {{ "{% block content%}" }}

    <table class="record_properties">
        <tbody>
            {%- for field, metadata in fields %}
            <tr>
                <th>{{ field|capitalize }}</th>

                {%- if metadata.type in ['date', 'datetime'] %}
                <td>{{ '{{ entity.' ~ field|replace({'_': ''}) ~
                    '|date(\'Y-m-d H:i:s\') }}' }}</td>
                    {%- else %}
                <td>
                    {{ '{{ entity.' ~ field|replace({'_': ''}) ~ ' }}' }}
                </td>
                {%- endif %}
            </tr>
            {%- endfor %}
        </tbody>
    </table>

    {% set hide_edit, hide_delete = false, false %}
    {% include 'crud/views/others/record_actions.html.twig.twig' %}
    {{ "{% endblock %}" }}
{% endblock body %}
```

- **app/Resources/SensioGeneratorBundle/skeleton/crud/views/others/actions.html.twig.twig**

```
<div class="actions">
    {%- for action in record_actions %}
    <a class="button" href="{{ "{{ path('" ~ route_name_prefix ~ "_" ~
                action ~ "', { 'id': entity."~ identifier ~" }) }}" }}">
        {{ action }}
    </a>
    {%- endfor %}
</div>
```

- **app/Resources/SensioGeneratorBundle/skeleton/crud/views/others/record_actions.html.twig.twig**

```
<div class="record-actions">
    <a class="button" href="{{ "{{ path('" ~ route_name_prefix ~ "') }}" }}">
        Back to the list
    </a>
    {% if ('edit' in actions) and (not hide_edit) %}
        <a class="button" href="{{ "{{ path('" ~ route_name_prefix ~ "_edit',
            { 'id': entity."~ identifier ~" }) }}" }}">
            Edit
        </a>
    {% endif %}
    {% if ('delete' in actions) and (not hide_delete) %}
        <div style="float:left">
        {{ '{{ form(delete_form) }}' }}
        </div>
        <div style="clear:both"></div>
    {% endif %}
</div>
```

> If you want to check the original templates that we just override, check vendor/sensio/generator-bundle/Sensio/Bundle/GeneratorBundle/Resources/skeleton

Done, we are now ready to start generating CRUD operations.

3.3.3 CRUD generate command

```
$ app/console doctrine:generate:crud --format=yml --with-write AppBundle:Category

  Welcome to the Doctrine2 CRUD generator

This command helps you generate CRUD controllers and templates.

First, you need to give the entity for which you want to generate a CRUD.
You can give an entity that does not exist yet and the wizard will help
you defining it.

You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name [AppBundle:Category]:

By default, the generator creates two actions: list and show.
You can also ask it to generate "write" actions: new, update, and delete.

Do you want to generate the "write" actions [yes]?

Determine the format to use for the generated CRUD.

Configuration format (yml, xml, php, or annotation) [yml]:

Determine the routes prefix (all the routes will be "mounted" under this
prefix: /prefix/, /prefix/new, ...).

Routes prefix [/category]:

  Summary before generation

You are going to generate a CRUD controller for "AppBundle:Category"
using the "yml" format.

Do you confirm generation [yes]?

  CRUD generation

Generating the CRUD code: OK
Generating the Form code: OK
Confirm automatic update of the Routing [yes]?
Importing the CRUD routes: OK

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

Done, let's see what we got.

The command generated:

- **src**/**AppBundle**/**Resources**/**config**/**routing**/**category.yml** with the different CRUD routes.
- **src**/**AppBundle**/**Resources**/**views**/**category**/**[index|show|new|edit].html.twig**.
- **src**/**AppBundle**/**Controller**/**CategoryController.php** with all CRUD actions.
- **src**/**AppBundle**/**Form**/**CategoryType.php** we will talk about form types later.

One more thing to make it work, our application doesn't know how to display a Category or a Product as simple text. We will add __toString methods to both entities.

- Edit **src**/**AppBundle**/**Entity**/**Category.php** and add this method to the end of the class

```
public function __toString()
{
    return $this->getLabel();
}
```

- Edit **src**/**AppBundle**/**Entity**/**Product.php** and add this method to the end of the class

```
public function __toString()
{
    return $this->getTitle();
}
```

You can navigate to http://127.0.0.1:8000/category/ to see the generated layout.

3.4 Validation

Data validation is as important as difficult to perform correctly. Symfony ships with a Validator component that makes this task easier.

If you go http://127.0.0.1:8000/category/new you can add a new category with an empty label, which is not OK.

Let's see how we can prevent this by a validation constraint.

- Edit **src**/**AppBundle**/**Entity**/**Category.php**

Add the follwing to the use section

```
use Symfony\Component\Validator\Constraints as Assert;
```

Update the `$label` annotation so it looks like:

```
/**
 * @var string
 *
 * @ORM\Column(name="label", type="string", length=45, nullable=true)
 * @Assert\NotBlank(message="The label cannot be empty")
 */
private $label;
```

Done, if you visit http://127.0.0.1:8000/category/new you will see even HTML5 validation added to the form.

We have another little problem, when you edit a category, you can set as parent the same category. Let's fix it.

- Edit **src/AppBundle/Entity/Category.php** again and add the following method to the end of the class.

```
/**
 * @Assert\True(message = "Parent category cannot be the same as child")
 */
public function isNotSameAsParent()
{
    if (!$this->getParentCategory()) {
        return true;
    }
    return $this->getId() !== $this->getParentCategory()->getId();
}
`
```

This was an overview of the Validator component. And we will see more features in the following chapters.

For a full list of the constraints list please check the[Symfony documentation]

3.4 Homework

1. Generate CRUD actions for the remaining entities.
2. Add a validation constraint on Account entity that both login and password should be unique.
3. In the product listing table, add a button that pop up a dialog containing the product's acquisition history.