2. Symfony bundle

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 ch1
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`.

It's time to start working on our application. Along this course, we will build a simple, yet working online store. We will put emphasis on programming best practices in general and Symfony best practices in particular. Whenever applicable, you will see quotations from the Official Symfony Best Practices guide.

First, let's start by quoting the definition of *best practice* from the official Symfony best practices guide:

> **Best practice** is a noun that means "a well defined procedure that is known to produce near-optimum results". And that's exactly what this guide aims to provide. Even if you don't agree with every recommendation, we believe these will help you build great applications with less complexity.

Notes about best practices:

- Consider best practices as guides rather than restrictions.
- Each application is unique and may have its own implementation quirks.
- Best practices compliance is not a quality metric.
- Enforcing best practices is a tool to reach your goals, not a goal on itself.

> **Symfony best practice:** Create only one bundle called AppBundle for your application logic

Many application developers using the Symfony framework, tend to split their business logic into separate bundles: CatalogueBundle, OrderBundle, InvoiceBundle, etc. Doing so, increases your application complexity and encourages code duplication.

How big is a bundle? When deciding when a logical sub module of your application should be in a seperate bundle or not, ask yourself a very simple question:
Is this bundle useable in another application without code change?
If the answer is **yes**, then that's a perfect candidate for a seperate bundle.

That said, we will implement most of our aplication logic in the AppBundle.

2.1 Catalog

No online store can see the light without a product catalog. We will start implementing a simple product catalog for our application. A minimum setup to see something visible in a browser is a *view*, a *controller* and a *route*.

Initially, we will assume the following features: list categories, show a category, edit a category, create a category.

2.1.1 Views

Symfony has a flexible templating component that supports multiple templating engines. In this course we will use Twig.

> **Symfony best practice:** Use Twig templating format for your templates.

We will store the Twig templates files in **app**/**Resources**/**views**

> **Symfony best practice:** Store all your application's templates in app/Resources/views/ directory.

Let's start by creating the following templates:

- **app**/**Resources**/**views**/**catalog**/**index.html.twig**

```
{% extends 'base.html.twig' %}
{% block body %}
    <p>Catalog</p>
    <a href="{{ path('catalog_category_list') }}">Categories</a>
{% endblock %}
```

The first line means that our template **extends** from **app**/**Resources**/**views**/**base.html.twig** which contains a very basic HTML page. And defines three **blocks** (title, body, javascript) that any child template can *redefine*.

`path('catalog_category_list')` uses the Twig **path** function that generates a relative URL from a given route name.

> If you want to generate absolute URLs use**url** instead of **path**. ie:
> `url('catalog_category_list')`

- **app/Resources/views/catalog/category/list.html.twig**

```
{% extends 'base.html.twig' %}
{% block body %}
    <p>Categories</p>
    <table border="1">
        <tr>
            <th>#</th><th>Category</th><th colspan="2">Actions</th>
        </tr>
        {% for category in categories %}
            {% set showUrl = path(
                'catalog_category_show',
                {'categoryId': category.id})
            %}
            {% set editUrl = path(
                'catalog_category_edit',
                {'categoryId': category.id})
            %}
            <tr>
                <td>{{ category.id }}</td>
                <td>{{ category.label }}</td>
                <td><a href="{{ showUrl }}">Show</a></td>
                <td><a href="{{ editUrl }}">Edit</a></td>
            </tr>
        {% endfor %}
    </table>
    <a href="{{ path('catalog_index') }}">Back to catalog</a>
{% endblock %}
```

`set showUrl = path('catalog_category_show',{'categoryId': category.id})` is how you assign a value to a variable in Twig. This time we used the **path** function with a second argument `{'categoryId': category.id}`. This argument will be passed to the Controller that will render that route.

- **app/Resources/views/catalog/category/show.html.twig**

```
{% extends 'base.html.twig' %}
{% block body %}
    <table border="1">
        <tr>
            <td>ID</td>
            <td>{{ category.id }}</td>
        </tr>
        <tr>
            <td>Label</td>
            <td>{{ category.label }}</td>
        </tr>
        <tr>
            <td>Parent</td>
            {% if(category.parent is null) %}
                <td>No parent</td>
            {% else %}
                <td>
                    {% set parentUrl = path(
                        'catalog_category_show',
                        {'categoryId': category.parent.id})
                    %}
                    <a href="{{ parentUrl }}">
                        {{ category.parent.label }}
                    </a>
                </td>
            {% endif %}
        </tr>
    </table>
    <a href="{{ path('catalog_category_list') }}">Back to list</a>
{% endblock %}
```

- **app/Resources/views/catalog/category/edit.html.twig**

```
{% extends 'base.html.twig' %}
{% block body %}
<form method="post">
    <input type="hidden" value="{{ category.id }}" />
    <table border="1">
        <tr>
            <td>ID</td>
            <td>{{ category.id }}</td>
        </tr>
        <tr>
            <td>Label</td>
            <td>
            <input type="text" name="label" value="{{ category.label }}" />
            </td>
        </tr>
        <tr>
            <td>Parent</td>
            <td>
                <select name="parent">
                    <option value="0">No parent</option>
                    {% for parentCategory in parentCategories %}
                        {% set selected = '' %}
                        {% set parentId = 0 %}
                        {% if category.parent is not null %}
                            {% set parentId = category.parent.id %}
                        {% endif %}
                        {% if parentId == parentCategory.id  %}
                            {% set selected = 'selected' %}
                        {% endif %}
                        <option {{ selected }} value="{{ parentCategory.id }}">
                            {{ parentCategory.label }}
                        </option>
                    {% endfor %}
                </select>
            </td>
        </tr>
    </table>
    <input type="submit" value="save"/>
</form>
<a href="{{ path('catalog_category_list') }}">Back to list</a>
{% endblock %}
```

2.1.2 Controllers

In order to view the templates we just created, we need a controller to render them. We will create CatalogController to render those templates.

- **src/AppBundle/Controller/CatalogController.php**

```php
<?php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class CatalogController extends Controller
{
    public function indexAction()
    {
        return $this->render('catalog/index.html.twig');
    }

    public function listCategoriesAction()
    {
        $arguments = array('categories' => $this->getCategories());
        return $this->render('catalog/category/list.html.twig', $arguments);
    }

    public function showCategoryAction($categoryId)
    {
        $arguments = array('category' => $this->getCategory($categoryId));
        return $this->render('catalog/category/show.html.twig', $arguments);
    }

    public function editCategoryAction($categoryId)
    {
        $arguments = array(
            'category' => $this->getCategory($categoryId),
            'parentCategories' => $this->getCategories(),
        );
        return $this->render('catalog/category/edit.html.twig', $arguments);
    }

    public function saveCategoryAction($categoryId)
    {
        $arguments = array('categoryId' => $categoryId);
        return $this->redirectToRoute('catalog_category_show', $arguments);
    }
    private function getCategories()
    {
        return array(
            1 => array('id' => 1, 'label' => 'Phones', 'parent' => null),
            2 => array('id' => 2, 'label' => 'Computers', 'parent' => null),
            3 => array('id' => 3, 'label' => 'Tablets', 'parent' => null),
            4 => array('id' => 4, 'label' => 'Desktop', 'parent' => array(
                    'id' => 2,
                    'label' => 'Computers')
            ),
            5 => array('id' => 5, 'label' => 'Laptop', 'parent' => array(
                    'id' => 2,
                    'label' => 'Computers')
            ),
        );
    }

    private function getCategory($categoryId)
    {
        $categories = $this->getCategories();

        if (empty($categories[$categoryId])) {
          throw new \Exception("CategoryId $categoryId does not exist");
        }

        return $categories[$categoryId];
    }
}
```

- `indexAction` just renders the template `'catalog/index.html.twig'`.
- `listCategoriesAction` renders the template `catalog/category/list.html.twig` and passes to it an argument named **categories** having as value the return value of `CatalogController::getCategories()`.
- `editCategoryAction` renders a template, but with two arguments.
- `saveCategoryAction` **redirects** (HTTP code 302) to the URL matching the route **catalog_category_show**

`saveCategoryAction` and `getCategories` methods doesn't look very helpful!
Is fair to expect `getCategories` to pull data from a data storage and `saveCategoryAction` to perform an update on some existing data. However, we are adopting the **Tracer Bullet** development process.

> Tracer Bullet Development process aims to implement just what is necessary for an application to deliver minimum functionality. The goal is to have something working as soon as possible. This technique helps developers to design a simple architecture and avoiding to deal with third party APIs integrations, which database system to use.. It also helps the customers by providing them with an early view on the system. They can adjust their ideas on future features because they have concreate material to rely on.

> Recommended reading
>
> - Ship It! A Practical Guide to Successful Software Projects [ISBN 0-9745140-4-7]
>
> The authors dedicated a full chapter for Tracer Bullet Development. The rest of the book is not less valuable, is a helpful guide through good software development practices.

2.1.3 Routing

Our controllers are ready to render the templates. All we need now is to tell Symfony which controller action to invoke when we a request is made. This is exactly the job of the router component.

Edit **app**/**config**/**routing.yml** discard the existing content that we made in the last chapter and type in

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

`prefix: /catalog` tells the router that every **path** defined in the imported file will be prefixed by /**catalog**. So the real path of `/some_path` will be `/catalog/some_path`.

- Create **src**/**AppBundle**/**Resources**/**config**/**routing**/**catalog.yml**

```
catalog_index:
    path: /
    defaults: { _controller: AppBundle:Catalog:index}
catalog_category_list:
    path: /category/list
    defaults: { _controller: AppBundle:Catalog:listCategories}
catalog_category_show:
    path: /category/show/{categoryId}
    defaults: { _controller: AppBundle:Catalog:showCategory}
catalog_category_edit:
    path: /category/edit/{categoryId}
    defaults: { _controller: AppBundle:Catalog:editCategory, categoryId:0}
    methods:  [GET]
catalog_category_save:
    path: /category/edit/{categoryId}
    defaults: { _controller: AppBundle:Catalog:saveCategory, categoryId:0}
    methods:  [POST]
```

We bound all the actions we created previously in CatalogController to different routes.

> `catalog_category_edit` and `catalog_category_save` has the same path, right? Yes, but not the same HTTP method (First is GET, second is POST). The router matcher uses many constraints to match a route. Besides the URL, you can specify HTTP methods, host, domain, etc.. In this example we used the `methods` filter which has a default value**all methods**. We will more routing filters later in this tutorial.

At this point we are done, let's check if everything is working as expected. ``

- http://127.0.0.1:8000/catalog/ should show "Catalog" and a link to Categories list page.
- http://127.0.0.1:8000/catalog/category/list should show the list of 5 categories with (show/edit) links, and a link to the catalog index.
- http://127.0.0.1:8000/catalog/category/show/1 should show the category Phones with No parent and a link to Categories list page.
- http://127.0.0.1:8000/catalog/category/show/4 should show the category Desktop with the parent a link to Computers category page.
- http://127.0.0.1:8000/catalog/category/edit/1 should show a form to edit the label and the parent of the category Phones. Upon submit, you should get to the Phones Category page.
- http://127.0.0.1:8000/catalog/category/show/6 should show an error page with the message "CategoryId 6 does not exist "

2.1.4 Enhance the user interface

All the actions we created works as expected. However the view doesn't look very nice. We will not start a shiny design project, we will use JQueryUI css framework and JQuery framework to make the layout a little bit better.

Let's start by adding the required resources to our project.

- Download JQueryUI from http://jqueryui.com/resources/download/jquery-ui-1.11.4.zip
- Extract **jquery-ui-1.11.4/jquery-ui.min.css** to **src/AppBundle/public/css/jquery-ui/**
- Extract **jquery-ui-1.11.4/jquery-ui.theme.min.css** to **src/AppBundle/public/css/jquery-ui/**
- Extract **jquery-ui-1.11.4/images** to **src/AppBundle/public/css/jquery-ui/**
- Extract **jquery-ui-1.11.4/jquery-ui.min.js** to **src/AppBundle/public/js**
- Download JQuery from https://code.jquery.com/jquery-2.1.4.min.js
- Save it to **src/AppBundle/public/js/jquery-2.1.4.min.js**

Done, now we will import those resouces in the base template.

- Edit **app/Resources/views/base.html.twig**

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}
            <link rel="stylesheet"
              href="{{asset('bundles/app/css/jquery-ui/jquery-ui.min.css')}}" >
            <link rel="stylesheet"
              href="{{asset('bundles/app/css/jquery-ui/jquery-ui.theme.min.css')}}" >
        {% endblock %}
        {% block javascripts %}
            <script type="text/javascript"
                src="{{asset('bundles/app/js/jquery-2.1.4.min.js')}}" >
            </script>
            <script type="text/javascript"
                src="{{asset('bundles/app/js/jquery-ui.min.js')}}" >
            </script>
        {% endblock %}
        <link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />
    </head>
    <body>
    {% block body %}{% endblock %}
    </body>
</html>
```

The changes we are going to make seems to not to be closely related to the Catalog module. Also is very important to keep a consistant layout accross the application's sections. For this reason, we will create a common **app** template that eventually most of the pages templates will extend from.

- Create **app/Resources/views/common/app.html.twig**

```
{% extends 'base.html.twig' %}
{% block stylesheets %}
    {{ parent()}}
    <link rel="stylesheet" href="{{asset( 'bundles/app/css/app.css' )}}" >
{% endblock %}
{% block javascripts %}
    {{ parent()}}
    <script type="text/javascript"
    src="{{asset( 'bundles/app/js/app.js' )}}" ></script>
{% endblock %}
{% block body %}
    <div id="container" class="ui-widget">
        <div id="header" class="ui-widget-header">
            {% block header %}
            {% endblock %}
        </div>
        <div id="content" class="ui-widget-content">
            {% block content %}
            {% endblock %}
        </div>
        <div id="footer" class="ui-widget-content">
            {% block footer %}
            {% endblock %}
        </div>
    </div>
{% endblock %}
```

This new template defines a header/content/footer structure using blocks with the same names. This is a generic enough structure to ensure the aaplication pages share a common layout while letting every page define it's own structure internally. You can see that we added two references to files that we didn't created yet, **app.css** and **app.js** which will contain common CSS and JavaScript.

- Create **web/public/css/app.css**

```
html {
    font-size: 12px;
}
body{
    margin: 0px;
    padding: 0px;
}
```

- Create **web/public/js/app.js**

```
$(function () {
    setupUi();
});

function setupUi() {
    $(':input').addClass('ui-widget-content');
    $('input[type="submit"], .button, #footer a').button();
    $('th').addClass('ui-widget-header');
}
```

The app template is ready, all we need to do is to update our templates to use is.

Update the templates

- Edit **app/Resources/views/catalog/index.html.twig**

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

{% block header %}Catalog{% endblock %}
{% block content %}
    <a href="{{ path('catalog_category_list') }}">Category list</a>
    <a href="{{ path('catalog_category_tree') }}">Category tree</a>
{% endblock %}
```

- Edit **app/Resources/views/catalog/category/show.html.twig**

```
{% extends 'common/app.html.twig' %}
{% block content %}
    <table border="1">
        <tr>
            <td>ID</td>
            <td>{{ category.id }}</td>
        </tr>
        <tr>
            <td>Label</td>
            <td>{{ category.label }}</td>
        </tr>
        <tr>
            <td>Parent</td>
            {% if(category.parent is null) %}
                <td>No parent</td>
            {% else %}
                <td>
                    <a href="{{ path('catalog_category_show',
                                {'categoryId': category.parent.id}) }}">
                        {{ category.parent.label }}
                    </a>
                </td>
            {% endif %}
        </tr>
    </table>
{% endblock %}
{% block footer %}
    <a href="{{ path('catalog_category_list') }}">Back to list</a>
{% endblock %}
```

- Edit **app/Resources/views/catalog/category/list.html.twig**

```
{% extends 'common/app.html.twig' %}
{% block header %} Categories {% endblock %}
{% block content %}
    <table border="1">
        <tr>
            <th>#</th><th>Category</th><th colspan="2">Actions</th>
        </tr>
        {% for category in categories %}
            {% set showUrl = path(
                'catalog_category_show',
                {'categoryId': category.id})
            %}
            {% set editUrl = path(
                'catalog_category_edit',
                {'categoryId': category.id})
            %}
            <tr>
                <td>{{ category.id }}</td>
                <td>{{ category.label }}</td>
                <td><a href="{{ showUrl }}">Show</a></td>
                <td><a href="{{ editUrl }}">Edit</a></td>
            </tr>
        {% endfor %}
    </table>
{% endblock %}
{% block footer %}
    <a href="{{ path('catalog_index') }}">Back to catalog</a>
{% endblock %}
```

- Edit **app/Resources/views/catalog/category/edit.html.twig**

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

{% block header %} Edit category {% endblock %}

{% block content %}
<form method="post">
    <input type="hidden" value="{{ category.id }}" />
    <table border="1">
        <tr>
            <td>ID</td>
            <td>{{ category.id }}</td>
        </tr>
        <tr>
            <td>Label</td>
            <td>
            <input type="text" name="label" value="{{ category.label }}" />
            </td>
        </tr>
        <tr>
            <td>Parent</td>
            <td>
                <select name="parent">
                    <option value="0">No parent</option>
                    {% for parentCategory in parentCategories %}
                        {% set selected = '' %}
                        {% set parentId = 0 %}
                        {% if category.parent is not null %}
                            {% set parentId = category.parent.id %}
                        {% endif %}
                        {% if parentId == parentCategory.id  %}
                            {% set selected = 'selected' %}
                        {% endif %}
                        <option {{ selected }} value="{{ parentCategory.id }}">
                            {{ parentCategory.label }}
                        </option>
                    {% endfor %}
                </select>
            </td>
        </tr>
    </table>
    <input type="submit" value="save"/>
</form>
{% endblock %}

{% block footer %}
    <a href="{{ path('catalog_category_list') }}">Back to list</a>
{% endblock %}
```

Change `{% extends 'base.html.twig' %}` to `{% extends 'common/app.html.twig' %}`

Browse the different views again, they should look a bit better.

2.1.4 Tree view

We know that the Categories will have a tree structure. It would be nice to display them in a more tree-like layout. We will use JQueryUI menu widget to achieve it.

We will not replace the categories list page, we will just add another template and another action.

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

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

{% block header %} Categories {% endblock %}
{% block content %}
    {% for category in categories %}
        <ul class="tree" style="width: 200px; ">
            {% include 'catalog/category/tree-node.html.twig'
              with {category:category} %}
        </ul>
    {% endfor %}
{% endblock %}
{% block footer %}
    <a href="{{ path('catalog_index') }}">Back to catalog</a>
{% endblock %}
```

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

```
{% set showUrl = path(
    'catalog_category_show',
    {'categoryId': category.id})
%}
{% set editUrl = path(
    'catalog_category_edit',
    {'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">Edit</a>
    </div>

    {% if category.children is not empty %}
        <ul style="width: 200px">
            {% for child in category.children %}
                {% include 'catalog/category/tree-node.html.twig'
                  with {category:child} %}
            {% endfor %}
        </ul>
    {% endif %}
    <div style="clear: both"></div>
</li>
```

- Edit **web/public/js/app.js** and add `$('.tree').menu();` at the end of setupUi function.
- Edit **src/AppBundle/Resources/config/routing/catalog.yml** and add the follwing route after **catalog_category_list**

```
catalog_category_tree:
    path: /category/tree
    defaults: { _controller: AppBundle:Catalog:treeCategories}
```

- Edit **src/AppBundle/Controller/CatalogController.php**

Add the follwing action after `listCategoriesAction()`

```
public function treeCategoriesAction()
{
    $arguments = array('categories' => $this->buildTree($this->getCategories()));
    return $this->render('catalog/category/tree.html.twig', $arguments);
}
```

Add the follwing method after `getCategory()`

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

    return $tree;
}
```

Done, you can visit http://127.0.0.1:8000/catalog/category/tree and you should see the categories are displayed as a menu.

2.3 Homework

1. Update `CatalogController::treeCategoriesAction` to return a json response if Content-Type is application/json, and work as it is otherwise.
2. Create a route called **app_dev_phpinfo** with the path /**_dev**/**phpinfo** this route should display the default `phpinfo()` output. Make sure this route is accessible only when the application is running in **dev mode**
3. Make the **Back to catalog** link visible in all the pages (avoid code duplication)
4. In the category tree page, update the Edit buttons so they show a dialog message with the edit form instead of redirecting to the edit page.