5828 – Foundations of Software Engineering
Spring 2012

SYMFONY2 WEB FRAMEWORK

By
Mazin Hakeem
Khaled Alanezi

Agenda

- Introduction
- What is a Framework?
- Why Use a Framework?
- What is Symfony2?
- Symfony2 from Scratch
- Symfony2 Overall Structure
- In Depth Look at the Controller
- Routing Component
- Templating Component
- The Model
- Testing Component
- Validation Component
- Forms Component
- Security Component
- Conclusion
- References
- Executive Summary

Introduction

- Symfony2 is an open source PHP-based web application development framework
- Based on the Model-View-Controller Design Pattern
- It enhances reusability, productivity and maintainability by providing solutions to common web application software problems
- These solutions are based on years of accumulated experience when dealing with web applications development
- The beauty of Symfony2 comes from the fact that it is totally customizable. Use what you need and throw out what you don't need!

What is a Framework?

- A collection of libraries (i.e. code or software) that are used to provide generic (i.e. common) functionalities and activities via well defined APIs
- In other words, it works as a tool to make the development process easier and more productive
- Caters various types of specific applications such as “web frameworks” for developing web applications and web services
- Usually based on Object Oriented paradigms
- Implements many kinds of design patterns like Model-View-Controller (MVC) pattern presented often in web frameworks

Why Use a Framework? (1)

- Promotes rapid development
  - Saves time
  - Reuse generic modules
- Code and design reuse; not reinventing the wheel by working from scratch
  - Less coding, more productivity
- Helps focusing on your application features development instead of working on basic mundane low-level details
  - Utilizing the framework to focus on the requirements needs
- Few bugs to worry about since the framework is well tested and used on many applications
  - Developers worry about the bugs form their codes only
  - Applying unit tests on the coded features

Why Use a Framework? (2)

- Includes various components and libraries in one package, like database connection, user interface forms, security, caching, and many others
  - Easy to use and well defined API calls in a unified structure and naming convention
- Full compliance with business rules and market needs (interoperability)
  - Structured code and environment
  - Easily maintainable and upgradable

What is Symfony2? (1)

- Definition according to *Fabien Potencier* the lead developer of the Symfony2 framework
- There are two main points when defining Symfony2:
  - Components: Symfony2 provides set of reusable PHP components. These components enables developers to solve web problems by providing solutions that are built according to known standards of high cohesion and low coupling
  - Being a full-stack web framework: Symfony2 provides an end-to-end web solution. However, developers still given the flexibility to make their solutions partially use Symfony2 components and implement other parts on their own

What is Symfony2? (2)

- Though Symfony2 basic structure is based on MVC, Fabien feels that it is better not view Symfony2 as an MVC because it offers more than that:

  "Have a look at the documentation, and you will see that the MVC pattern is only mentioned once or twice"
- By the introduction of Symfony2, the framework departed from being a monolithic solution. Now, components can be used as needed which provides a better chance for Symfony2 to spread by coexisting with other solutions
- Therefore, Symfony2 must be viewed as a provider for PHP low-level architecture
  - The solid components allow Symfony2 to play this role because they evolved over years of dealing with web development common problems

What is Symfony2? (3)

- However, if you want to look at Symfony2 as MVC:
  - It provides the tools for the control part
  - It provides the view part
  - It doesn’t provide the model. You can implement your own and use Doctrine to adopt it (more on Doctrine later)
- But remember, being an MVC is not the target. Separation of concerns is what matters by avoid mingling code that does different actions. This enables:
  - Reusability
  - Maintainability
  - Productivity

Symfony2 From Scratch (1)

- Let’s demonstrate how Symfony2 is organized by mapping a PHP page code that does “everything” to the Symfony2 structure
- Keep in mind that the overall goal of a web page is to simply receive a URL request from the user and return an HTML page as a response
- The PHP page we’ll use, receive a request from the user to display all blog posts, retrieve the blog post from the database and displays them using HTML (see code next page)

Symfony2 From Scratch (2)

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);
?>

<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = mysql_fetch_assoc($result)): ?>
            <li>
                <a href="/show.php?id=<?php echo $row['id'] ?>">
                    <?php echo $row['title'] ?>
                </a>
            </li>
            <?php endwhile; ?>
        </ul>
    </body>
</html>

<?php
mysql_close($link);
```

Connect to DB

Query DB

Loop thru records

Display as links

Problems:
- As code grows, forget about maintainability!
- No reusability
- Tied to specific implementation details (MySQL in this example)

Symfony2 From Scratch (3)

- Step 1: Put the display method in a separate file “templates/list.php”

```
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
            <li>
                <a href="/read?id=<?php echo $post['id'] ?>">
                    <?php echo $post['title'] ?>
                </a>
            </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

Input received in the posts variable

Output returned as links in an HTML page

Benefits:

- We can render the output in different formats (e.g. JSON) by only changing this part.
- Developer knows that output is handled inside the view!

Symfony2 From Scratch (4)

- Now, our index.php (the super page that does everything) looks like:

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);

$posts = array();
while ($row = mysql_fetch_assoc($result)) {
    $posts[] = $row;
}

mysql_close($link);

// include the HTML presentation code
require 'templates/list.php';
```

Connect to DB

Query DB

Store results in posts variable

We would like to:

- Separate application logic to be subject to reusability
- Write connect & close code only once and reuse them
- Write SQL statements only once and reuse them

Symfony2 From Scratch (5):

- Step (2) : Introduce the model and move all application logic & data access to it

```
<?php
// model.php

function open_database_connection()  ←(1)
{
    $link = mysql_connect('localhost', 'myuser', 'mypassword');
    mysql_select_db('blog_db', $link);

    return $link;
}

function close_database_connection($link)  ←(2)
{
    mysql_close($link);
}

function get_all_posts()  ←(3)
{
    $link = open_database_connection();

    $result = mysql_query('SELECT id, title FROM post', $link);
    $posts = array();
    while ($row = mysql_fetch_assoc($result)) {
        $posts[] = $row;
    }
    close_database_connection($link);

    return $posts;
}
```

1, 2 & 3 are functions that contain app logic.

Benefits:

- We can reuse written app logic
- Developer knows that app logic is handled inside the model!

Symfony2 From Scratch (6):

- Since we have developed the model and the view, we'll let index.php act as the controller
- The controller is responsible for receiving user input and returning a response
- This what the controller exactly does in the code that is left!

```
<?php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

← Get data from the model (require_once 'model.php';)

← Call view to display output (require 'templates/list.php';)

But:

- A web application usually consists of dozens of pages
- So, let's test our new structure flexibility towards adding new pages

Symfony2 From Scratch (7)

- The customer would like to add a new web page that takes an ID as input and output the corresponding blog entry
- With our new architecture, we know exactly where to put the new code:
  - Add a new method in the *model* that takes blog ID, query DB and return a variable containing the blog
  - Create a new *view* template to render a single blog entry
  - Create a new *controller* method for handling the URL of single blog show feature
- The code for the new controller show.php:

```
<?php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Symfony2 From Scratch (8)

- Great, we can see the benefit of our new organized structure.
  - No need to rewrite the code to open and close DB
  - More importantly, we have a convention of where to add our code (Maintainability and Extensibility)
- But, now we have two controllers handling two different possible URLs. There are two problems with our controllers structure:
  - No error handling: for example if the user supplies invalid user ID the page will crash. A more logical response is to reply with 404 (page not found)
  - No flexibility: for each controller we have to include the model.php, If we have dozen of controllers we will need to add the statement require_once 'model.php'; dozen times. The same goes for every global functionality to be added to the features.

Symfony2 From Scratch (9)

- Step(3): create front controller to handle all incoming URL requests. Our controllers now will only have the methods to be invoked for handling these requests.
- Now, controllers are simple and can be merged in a single file controllers.php:

```
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

Routing method is needed to divert incoming requests to corresponding method calls

Benefit:
- Any method call for handling new feature will be included in the controller

Symfony2 From Scratch (10)

- We can let the front controller perform the control logic as follows:

```
$uri = $_SERVER['REQUEST_URI'];
if ($uri == '/index.php') {
    list_action();
} elseif ($uri == '/index.php/show' && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

Problem: User concerned about the architecture and handling different requests

- Instead, Symfony2 routing uses a configuration file where user can define URL patterns and their corresponding method invocations

```
# app/config/routing.yml
blog_list:
    pattern:  /blog
    defaults: { _controller: AcmeBlogBundle:Blog:list }

blog_show:
    pattern:  /blog/show/{id}
    defaults: { _controller: AcmeBlogBundle:Blog:show }
```

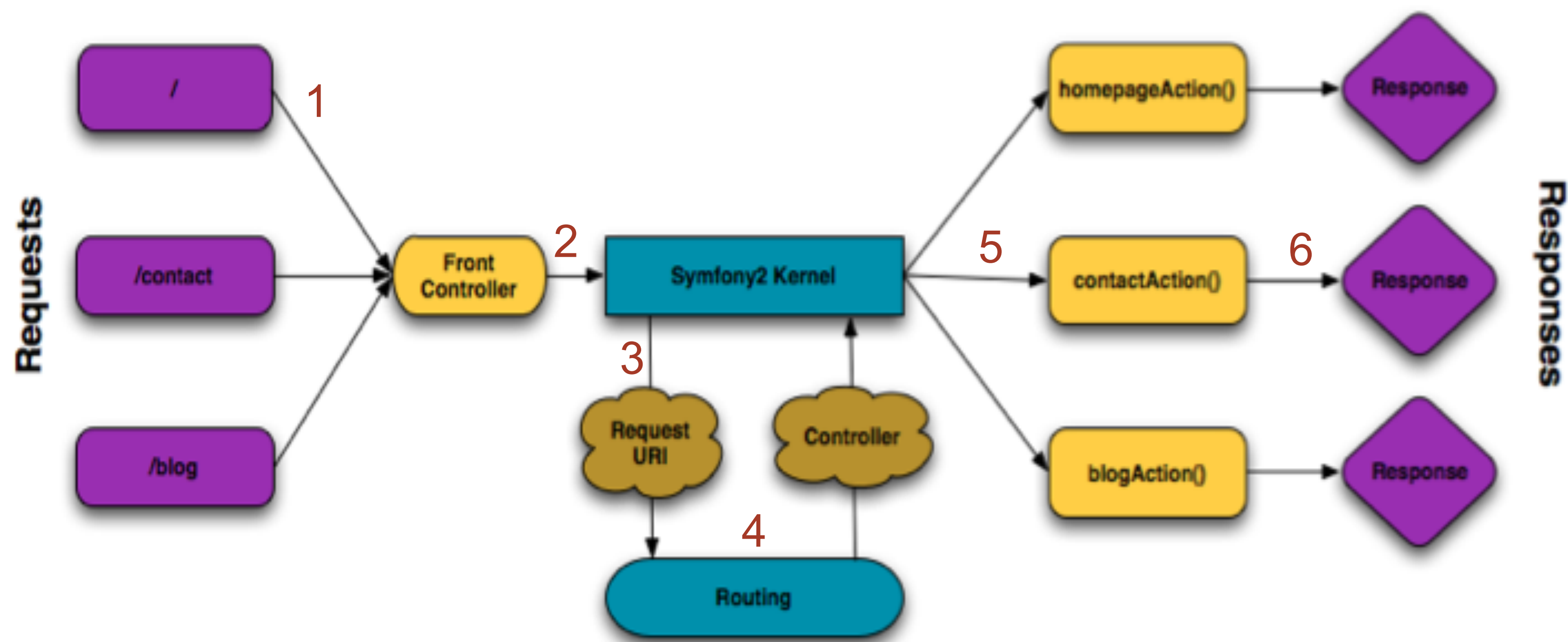
URL pattern (/blog)

URL pattern (/blog/show/{id})

Call methods list_action() show_action() in controllers.php

Symfony2 Overall Structure (1)

- To summarize let's look at the below diagram illustrating the flow control of a typical Symfony2 web app:



1) URL requests received by front controller
2) Then, sent to kernel
3) Symfony2 kernel forward request to routing
4) Routing uses config file to match url pattern to controller method
5) Kernel calls corresponding method
6) Method returns a response

Symfony2 Overall Structure (2)

- Symfony2 application consists of bundles where each bundle implements set of features. Symfony2 application's folder looks like:
  - app/
    - Contains the application configuration
    - Map a bundle to it's corresponding routing config file
  - src/
    - Contains the source code of your project
    - Controller code, view template and routing config file
  - vendor/
    - Hosts library files provided by third parties
  - web/
    - Contains the front controller of the application which calls the kernel to bootstrap the application
    - Also, contains your static files (images, style sheets…etc)

In Depth Look at the Controller

- So far we have seen that the controller has the method calls to be invoked upon the arrival of a URL request. These calls will return the required response
- The controller can extend the base controller class which provides basic controller tasks. Here are some examples with explanation:

```
return $this->render('AcmeHelloBundle:Hello:index.html.twig'
array('name' => $name));
```

The render method takes the template and output to be displayed as argument and return the rendered output. More on templates in the templates section

Redirect method

```
return $this->redirect($this->generateUrl('homepage'));
```

```
$this->get('session')->setFlash('notice', 'Your changes were saved!');
```

Display flash messages

Routing Component (1)

- As described in the introduction section, Sumfony2 uses routing to map a URL pattern to a method inside the controller
- Router provides two benefits:
  - Writing nice URLs rather than ugly ones:
    - Index.php?article_id=1 → Bad
    - /read/intro-to-symfony → Good
  - Flexibility when changing the URL name
    - No need to traverse all pages to search for links related to the updated page
- Routing is written in a separate routing file and can be specified in three different languages namely YAML, XML and PHP
- The application routing file is located at:
  - App/config/routing.yml

Routing Component (2)

- An example using YAML:

```
# app/config/routing.yml
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

URL pattern

Variable slug will be available inside the controller method

Method to call from the controller

- Pretty simple, however, web apps need more than this!
- For example, we would like to support pagination:
  - /blog/2 → will display the second page
- Requirements come into play. They are specified using regular expressions. The pagination requirement can be written as follows:

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
    requirements:
        page:  \d+
```

Regular expression specifying that the value of page variable should be one or more digits

Default value for {page}, If nothing supplied use page=1

Routing Component (3)

- The use of regular expressions in routes provides great deal of flexibility when defining requirements
- Here is another example where a website have two versions for two different languages

```
homepage:
    pattern:   /{culture}
    defaults:  { _controller: AcmeDemoBundle:Main:homepage, culture: en }
    requirements:
        culture:  en|fr
```

Use "en" if nothing specified in URL

{culture} is either "en" or "fr"

- Also, we can use the type of the coming Http request and do routing based on it as in the following example:

```
contact:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
    requirements:
        _method:  GET

contact_process:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contactProcess }
    requirements:
        _method:  POST
```

Use same URL for GET and POST and yet trigger different controller methods!

Routing Component (4)

- This last example, shows how advanced your routing requirements can be:

```
article_show:
  pattern:  /articles/{culture}/{year}/{title}.{_format}
  defaults: { _controller: AcmeDemoBundle:Article:show, _format: html }
  requirements:
      culture:  en|fr
      _format:  html|rss
      year:     \d+
```

- In this example:
  - {year} is mandatory and is a digit or more
  - {culture} is optional where if not provided the used value is “en”
  - {_format} is optional, it can be either html or rss. The method can use it to decide upon the format of the response it is going to return
- Following URLs will match:
  - /articles/en/2010/my-post
  - /articles/en/2010/my-post.rss

Templating Component (1)

- The Controller delegates the work to the templates engine when visual representation is needed like HTML and CSS files
- A template is a text based file that can provide the visual representation by generating text files like HTML, LaTeX, CSV, XML, and many more.
- Symfony2 uses “Twig” (PHP can be used which is up to the developer), a powerful and readable templating language
- Used to template and provide visual presentation.
- No code logic is written inside it
- With Twig, readable and concise code is easily written

Templating Component (2)

```
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

{% ... %}: "Does something": a tag that controls the logic of the template; it is used to execute statements such as for-loops for example

{{ ... }}: "Says something": prints a variable or the result of an expression to the template

Templating Component (3)

- With Twig, a base layout can be built that includes common aspects of a webpage like headers and footers, and then a child page can inherit this layout and override its own fields and bocks.

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
            {% endblock %}
        </div>

        <div id="content">
            {% block body %}{% endblock %}
        </div>
    </body>
</html>
```

Base layout

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

The child “extends” the base page

Templating Component (4)

- As mentioned, the controller delegates to the template to render the view page.
- We add the following in the controller class:

```
// creates a Response object whose content is the rendered template
$response = $this->render('AcmeArticleBundle:Article:index.html.twig');

// creates a Response object whose content is simple text
$response = new Response('response content');
```

The Model (1)

- Any dynamic website requires communicating with a database for fetching data
- Therefore, the Model layer makes that part easier
- It is an abstraction layer of communicating with databases, which is isolated from the rest of the web application for reusability and maintainability
- Symfony2 does not have its own Model layer
- Instead, a popular Object Relational Mapping (ORM) technology is used (or “reused”) called “Doctrine”
- Doctrine allows to “map objects to a relational database” like MySQL, MS SQL Server, and many others

The Model (2)

- To connect to the database, all database information is listed inside the “parameters.ini” file instead of copying those info inside the code files; hence, more secure web application.

```
;app/config/parameters.ini
[parameters]
    database_driver   = pdo_mysql
    database_host     = localhost
    database_name     = test_project
    database_user     = root
    database_password = password
```

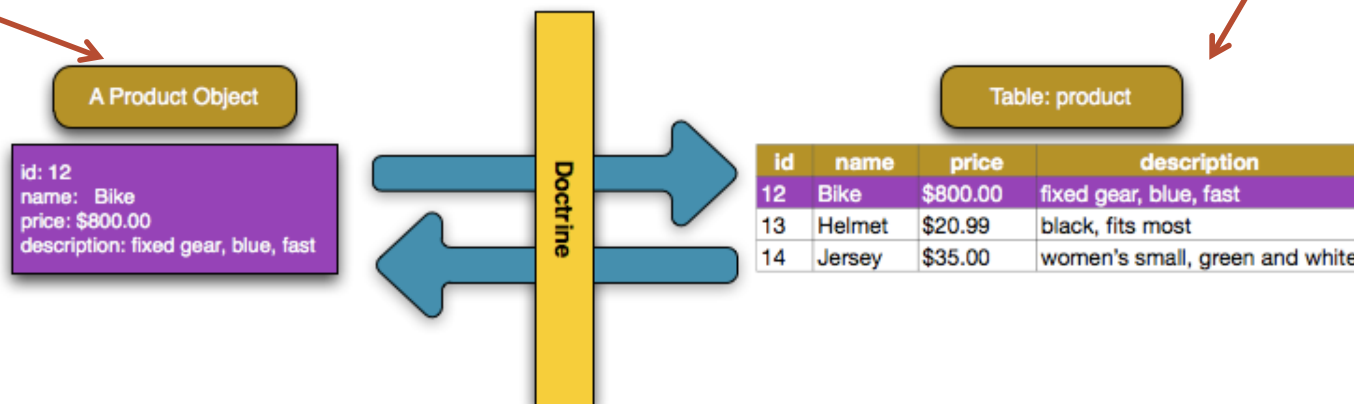
Here, MySQL driver is used

The Model (3)

- The Doctrine ORM fetches and saves entire objects to and from the database instead of dealing with each rows
- The idea is to map PHP class files to columns in the table

Instantiated object from a PHP class

Relational DB table



id	name	price	description
12	Bike	$800.00	fixed gear, blue, fast
13	Helmet	$20.99	black, fits most
14	Jersey	$35.00	women's small, green and white

- This is done by including a metadata to guide Doctrine how to map PHP class properties to the database fields using PHP Annotations, YAML, or XML (see next slide)

The Model (4)

```php
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

Managing the object

Writing the object data to DB after being managed

```php
// src/Acme/StoreBundle/Controller/DefaultController.php
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

public function createAction()
{
    $product = new Product();
    $product->setName('A Foo Bar');
    $product->setPrice('19.99');
    $product->setDescription('Lorem ipsum dolor');

    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($product);
    $em->flush();

    return new Response('Created product id '.$product->getId());
}
```

```php
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // do something, like pass the $product object into a template
}
```

A PHP class w/ annotations to guide Doctrine (getters and setters must be defined in this class)

Fetching data from DB

Testing Component (1)

- Symfony2 integrates with the PHPUnit testing framework which is a unit testing framework for PHP projects
- It provides the following benefits:
  - Makes writing tests easy
    - If learning to write the tests is difficult or the process of writing a test is difficult, the team will likely run away from testing
  - Makes running tests easy and quick
    - This will encourage the team to run test hundreds and sometimes thousands of times
  - Prevent dependency between tests:
    - Tests can run independently. Also, any order of running tests will generate similar results. This reduces the complexity of testing

Testing Component (2)

- All tests must be placed in the Tests/ directory of your project
- The test directory will replicate the structure of your bundle's directory
- Following is an example. The code of the tested controller method:

```
//src/Acme/DemoBundle/Utility/Calculator.php

namespace Acme\DemoBundle\Utility;
class Calculator {
    public function add($a, $b) {
        return $a + $b;
    }
}
```

Similar folder name Utility/ in bundle and Tests folders

- The test code:

```
// src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php
namespace Acme\DemoBundle\Tests\Utility;

use Acme\DemoBundle\Utility\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);
    // assert that our calculator added the numbers correctly!
    $this->assertEquals(42, $result);
    }
}
```

Extend to use PHPUnit methods

Simple PHPUnit assertion method

Testing Component (3)

- The last example was trivial as it tests only the addition of two numbers. The question is how to test web pages with complex structures (e.g. links, forms…etc.)?
  - No worries, PHPUnit provides the client and the crawler objects to help you out
  - The client simulates a browser by allowing the developer to make requests
  - The crawler is returned from a client request and allows traversal of the content of the reply
    - e.g. traverse HTML tags, select nodes, links and forms
  - Below is an example:

```
$client = createClient();
$crawler = $client->request('GET', '/demo/hello/Fabien');
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);
```

Creating client object

Submit HTTP GET request which returns the crawler object

Look for "Go elsewhere…" link and click on it!

Testing Component (4)

- The last example was for a link, how about submitting a form?
- Here is an example:

```
$buttonCrawlerNode = $crawler->selectButton('submit');

$form = $buttonCrawlerNode->form();

$form = $buttonCrawlerNode->form(array(
        'name'   => 'Fabien',
        'my_form[subject]' => 'Symfony rocks!',
));

$client->submit($form);
```

Validation Component (1)

- Being able to test the results of the form as provided by PHPUnit test is good. However, best practice is to ensure correctness of the input before submitting your request whether to a form, web service or a database.
- This is exactly what the validation component does for you!
- Validation rules of an object (called constraints) are first defined using YAML, XML or PHP. This is an example with YAML

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
properties:
firstName:
- NotBlank: ~
- MinLength: 3
```

Author.firstName shouldn't be blank and should be minimum three chars

- Then constraints used inside the code of the controller as follows:

```
public function indexAction(){
    $author = new Author();
    $validator = $this->get('validator');
    $errors = $validator->validate($author);
    if (count($errors) > 0)
        return new Response(print_r($errors, true));
    else
        return new Response('The author is valid! Yes!');
}
```

Call validate and pass the object name as a reference which will return number of errors by matching against the constraints

Validation Component (2)

- Since an object might violate several rules, you can loop thru the resultant error messages:

```
{# src/Acme/BlogBundle/Resources/views/Author/validate.html.twig #}

    <h3>The author has the following errors</h3>
    <ul>
    {% for error in errors %}
    <li>{{ error.message }}</li>
    {% endfor %}
    </ul>
```

- So far, we have seem minimum length and not blank constraints. Validation component provides others such as:
  - Max Length (for string)
  - Min and Max (for numbers)
  - Choice (e.g. male or female for gender attribute)
  - …and many others!

Forms Component (1)

- Forms are an essential part of any dynamic web pages in which a user can interact
- It is known that PHP does not offer its own HTML based forms like ASP .NET
- Hence, it uses the regular HTML forms
- PHP has powerful functionalities to interact with HTML forms
- However, the process can sometimes get difficult, cluttered, and mundane
- Symfony2 provides an integrated built-in forms component to make the task easier

Forms Component (2)

- The steps to create forms:
    1. Create a separate class to make it generic and to store data into it if needed to interact with databases
    2. Call that class inside the Controller class (or define another class for reusability and then call it inside the Controller class) to build the form and to render it to HTML
    3. Defining the object in a Twig to help rendering the forms

(see examples in next slides)

Forms Component (3)

```php
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

class Task
{
    protected $task;

    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }
    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

A generic class to handle the forms (To render a text box, data picker, and a “Submit” button, for example here)

```twig
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

<form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

A Twig template to specify the forms properties instead of defining inside the PHP file

Forms Component (4)

```php
// src/Acme/TaskBundle/Controller/DefaultController.php
namespace Acme\TaskBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TaskBundle\Entity\Task;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // create a task and give it some dummy data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->getForm();

        return $this->render('AcmeTaskBundle:Default:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

A Controller class where it calls the class that handles the forms and then renders the Twig template

This function creates the built-in forms easily

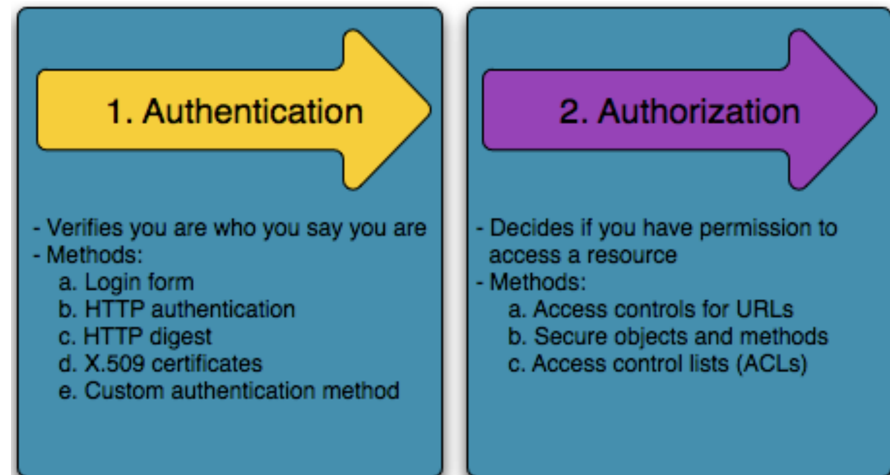
The end result

Task: Write a blog post
Duedate: Jul 24, 2011
Submit

Security Component (1)

- Almost any dynamic website has user security; not any unregistered user can access the website
- Providing user roles and privileges is usually a hassle and complex

- Therefore, Symfony2 addresses that issue and provides a security component that is based on 2 steps: Authentication and Authorization



Security Component (2)

- Security configuration is done inside the "security.yml" file (XML and PHP files can be used instead as well)
- Lets see a simple example:

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
        in_memory:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

A regular expression means "match every incoming request"

Default value

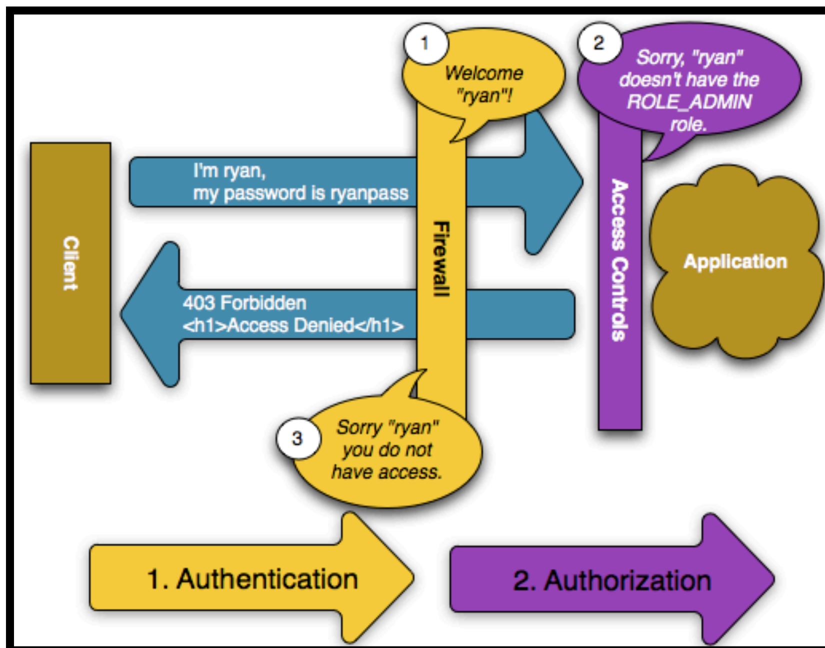
Allow only /admin/[any name] users with the Admin role

The authenticated users

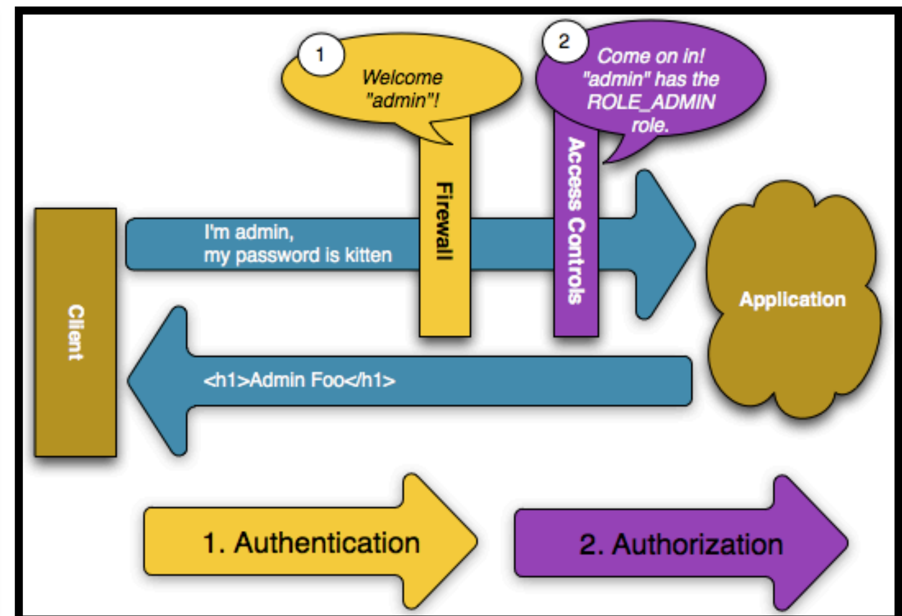
Password encoding type (text in this example)

Security Component (3)

- It is pretty strait forward; any user passes the firewall goes through an authorization process to access the website



A diagram reflecting the flow for a user without admin privileges gets denied



A diagram reflecting the flow for a user with admin privileges gets accepted to use the web app

Conclusion

- Though Symfony2 is an MVC based framework, it allows for a great deal of reusability by means of its components
- Symfony2 integrates with external components to provide common services. Examples include:
  - Twig for templating
  - PHPUnit for testing
  - Doctrine for database integrating
- Symfony2 is a full stack web framework, however, developers have the flexibility to use components and leave other as suitable for their projects

References

- http://symfony.com/why-use-a-framework
- http://en.wikipedia.org/wiki/Software_framework
- http://docforge.com/wiki/Framework
- http://symfony.com/doc/current/book/index.html
- http://fabien.potencier.org/article/49/what-is-symfony2
- http://www.phpunit.de
- http://en.wikipedia.org/wiki/Symfony

Executive Summary (1)

- Though Symfony2 is an MVC based framework, it provides much more than this via its large range of components
- Symfony2 can be best viewed as a provider for PHP low-level architecture
- This is achieved by reusing the components provided by Symfony2
- Examples of such components that were discussed in the presentation:
  - Routing Component:

    Provides mechanisms for routing a URL request to corresponding method inside the controller
  - Templating Component:

    Allows for defining templates for returning visual representations. These can be used by multiple features in the website

Executive Summary (2)

- Testing Component:

  Provides the developer with tools that makes the process of writing test and conducting them easy and quick
- Validation Component:

  Allows the user to define all the validations for objects in a single configuration file and call them as necessary inside the code
- Forms Component:

  Collaborates with the templating component in order to generate hassle free dynamic forms
- Security Component:

  Unifies the processes of authentication and authorization