Agile Web Development
www.liip.ch

< the php & javascript experts >

Control

Controller

Extending the default controller

```
namespace
  Demo\BlogBundle\Controller;

use Symfony\Bundle\Framework
Bundle\Controller\Controller;

class BlogController extends
Controller
{
  public function showAction(
    $id, $comments=false)
  {
    return $this->render(
      'DemoBlogBundle:Blog:in
dex.html.twig',
      array('id' => $id));
  }
}
```

Method names must end with *Action*. Method parameters are defined in the route and are referenced by name. (Order does not matter.) You can use `$this->container` to access the service container or get a service via `$this->get(..)`. The Parameters you pass to the *render* method are available in the twig template.

Redirection and flash message

```
use Symfony\Component\Http
Foundation\RedirectResponse;
...
$this->get('session')->setFlash
  ('notice', 'Comment saved.');
return new RedirectResponse(
  'http://symfony.com/', 302);
```

Flash message is available in session on next request. See Templating section for how to show flashes. Second parameter of RedirectResponse() is optional, tells the HTTP status code. Defaults to 302, moved temporarily.

Controller as a service

```
namespace
  Demo\BlogBundle\Controller;

use Symfony\Component\
HttpFoundation\Response;

class BlogController
{
  public function __construct(
    $templating)
  {
    $this->templating =
      $templating;
  }
  public function showAction(
    $id)
  {
    return $this->templating->
      renderResponse('DemoBlog
Bundle:Blog:index.html.twig',
      array('name' => $blog));
  }
}
```

Do *not* extend the Controller class. Inject everything you need explicitly, i.e in the constructor. Configure this class as service `demo_blog.blogController` (see Dependency Injection in Service Container section). See http://bit.ly/symfony-service for more.

Routing

Route with parameters

```
blog:
  pattern: /blog/{id}
  defaults: {_controller:
    DemoBlogBundle:Blog:show,
    id: 123}
  requirements:
    id: \d+
```

- To define **parameters**, put the name in curly brackets „{id}“.
- The **defaults** tell which controller to use and can be used to make URL parameters optional. This entry will match `/blog` or `/blog/33`.
- The **requirements** allow to define a regular expression for each parameter. This route will only match numerical ids, but not for example `/blog/abc`.

Generating a route

```
$router->generate('blog',
array('id' => 222), false);
```

- The default router is available in the container as „router“. Either inject it with Dependency Injection or when extending the base controller, use `$this->get('router');` to fetch the service.
- First parameter is the **route name.** (the top level token in the yml configuration)
- Second parameter is optional, specify values for a **route's parameters.**
- Third parameter tells wheter to generate an **absolute** URL. Defaults to false.

Routes and i18n

```
about:
  pattern: /{_locale}/about
  defaults: {_controller:
    DemoBlogBundle:Blog:about,
    _locale: de}
  requirements:
    _locale: en|de
```

- The **_locale** is used as the session locale. `/about` is the german page, `/en/about` the english one.

Routing to a controller service

```
blog:
  pattern:  /blog/{id}
  defaults: {_controller:demo_
      blog.blog_controller:
        showAction}
```

When referencing the controller as a service (see chapter „Service Container“), you need to write the full method name. You can also call something that does not end on Action).

View

Twig

```
{% .. %} DOES something
{{ .. }} PRINTS something
```

Variables

```
{{ date }}
{{ a.author ~ „ - “ ~ date }}

{% set var = „my test data“ %}
```

a.author will be resolved as:

- a is an array and `author` a valid index
- a is object and
  - there is a public `author` property
  - there is a public `author()` method
  - there is a public `getAuthor()` method
  - there is a public `isAuthor()` method
- otherwise will result in `null`

Built-in operators:

- Math: +, -, /, %, /, *, **
- Others: .., |, ~, ., [], ?:

Filters

Usage

```
{{ a.title|striptags|title }}
<div class=“blog_article_body“>
  {% filter striptags %}
    {{ article.body }}
  {% endfilter %}
</div>
<div class=“blog_categories“>
  {{ a.categories|join(', ') }}
</div>
```

Built-in filters:

- **formatting:** capitalize, lower, upper, date, format, number_format, title
- **string operations:** default, replace, striptags
- **string and arrays:** length, reverse, slice
- **arrays:** join, keys, merge, sort
- **encoding:** raw, convert_encoding, escape, json_encode, nl2br, url_encode

Functions

Usage

```
<div id=“footer“>
    {{ date() }}
</div>
```

Built-in functions: attribute, block, constant, cycle, date, dump, parent, random, range

If/Else

```
{% if booleanValue %}
  ...
{% endif %}
{% if anotherBooleanValue %}
  ...
{% elseif thirdBooleanValue %}
  ...
{% else %}
  ...
{% endif %}
```

Functionality for conditions:

- **Tests:** constant, defined, divisibleby, empty, even, in, is, null, odd, sameas
- **Logic:** and, or, not
- **Comparisons:** ==, !=, <, >, >=, <=, ===

Loops

In Twig, all loops are done using the **for** tag. If provides more flexibility than a regular for-loop in PHP or most other programming languages:

foreach-like loop iterating over all articles of a blog

```
{% for a in blog.arts %}
  <p>{{ a.body }}</p>
{% endfor %}
{% for key,a in blog.arts %}
  <h1>Article No {{key}}</h1>
  <p>{{ a.body }}</p>
{% endfor %}
{% for k in blog.arts|keys %}
  <a href=“/blog/article/{{
k }}“>Article No {{ k }}</a>
{% endfor %}
```

loop iterating over sequence

```
{% for l in 'x'..'z' %}
  {{ l }}
{% endfor %}
{# output: x y z #}
{% for i in 0..6 %}
  {{ i }}
{% endfor %}
{# output: 0 1 2 3 4 5 6 #}
```

loop with filter, loop and alternative if empty

```
{% for a in blog.arts if
  a.published %}
  <p>{{ a.body }}</p>
{% endfor %}
{% for a in blog.arts %}
  <p>{{ a.body }}</p>
{% else %}
  <p>No content.</p>
{% endfor %}
```

Comments

```
{# This is the comment tag.
   It can be used for multi-
   or single-line #}
```

Translation

With a configured translator, internationalization is very easy to use with Twig:

```
<head>
  <title>example.com //
  {% trans %}About{% endtrans %}
  </title>
</head>

<p>
{% transchoice numItems %}
  {0}No items|{1} One item|
  ]1,Inf] %numItems% items
{% endtranschoice %}
</p>
```

Internal Links

They are generated using the routes' names:

```
{# relative URL #}
<a href=“{{ path('about') }}“>
  About me</a>
{# absolute URL with params #}
<a href=“{{ url('article',
  {'id': 42}) }}“>My favorite
article</a>

{# relative URL #}
<a href=“{{ path('about') }}“>
  Über Mich</a>
{# absolute URL #}
<a href=“{{ url('article',
  {'id': 42}) }}“>Mein
  Lieblingsartikel</a>
```

Flash messages

If you have set a flash message, you can access it like this:

```
{% if app.session.hasFlash(
  'error') %}
  <p class=“auth-errors“>
    {{ app.session.flash(
      'error') }}
  </p>
{% endif %}
```

Compose templates

Includes

Templates can include other templates. This concept is sometimes referred to as *partials*. The bundle name is expanded to the folder `Resource/view` inside that bundle folder. If omitted, the current bundle is used. It is possible to pass parameters to the included template.

```
<div id=“header“>{% include
 'DemoBlogBundle:Blog:header.
 html' with {'title': article.
title} %}</div>
<div id=“main_wrapper“>
  <div id=“sidebar“>
    {% include '::sb.html' %}
  </div>
  <div id=“content“>...</div>
</div>
```

Block: Template Inheritance

Twig comes with a mechanism called *Template Inheritance.*

This enables you to build websites out of multiple components which are created by different software modules (e.g. the sidebar of a website is being generated by a Sidebar module and the main content is created by the controller responsible for the current request).

Blocks allow inheriting templates to decorate parts of the parent template:

- A template may contain **blocks.**
- **Blocks** can be overridden by inheriting templates.
- Inheritance is defined by using the ***extends*** tag.
- Inheriting templates can call the parent-block

Top-Level Template „base.html.twig“

```
<head>
{% block header %}
  <title>My Blog | {% block
    title %}{% endblock %}
  </title>
{% endblock %}
</head>
<body>
<div id=“sidebar“>...</div>
<div id=“content“>
  {% block content %}
    <div id=“statistics“>Page
      views: {{ page_views }}
    </div>
  {% endblock %}
</div>
</body>
```

Controller Template „scr/Demo/BlogBundle/Ressources/views/Blog/archive.html.twig“

```
{% extends '::base.html.twig' %}
{% block title %}Archive
{% endblock %}
{% block content %}
  {# print archived articles #}
  {{ parent() }}
{% endblock %}
```

Including Controllers/Actions

To render the output of another controller, you can call one from your templates! This is especially helpful when rendering a page with independent components managed by different controllers.

```
<div class=“blogStatistics“>
  {% render „DemoBlogBundle:
    Blog:statistics“ %}
</div>
```

Liip

Liip is Switzerland's leading developer of tailor-made, agile web applications. State-of-the-art technologies, tried and tested industry standards and open-source software are the foundations on which our services are built.

We regularly win awards for the know-how and experience that we invest in innovative projects for the Swiss market. A large number of our employees engage in open source software and communities. Such involvement and commitment is actively fostered by Liip.

Please contact us if you are interested in building your new application in Symfony 2 or if you plan other activities around Symfony 2.

www.liip.ch
twitter.com/liip

Model

Doctrine2

Object Relational Mapper
Table rows are represented by **entity objects.** Tables are mapped to **entity classes. Entity classes** are plain old PHP objects with attributes, and getter and setter methods. The central API of Doctrine2 is the **EntityManager.**

Entity Manager and Entity Objects

Central API of Doctrine2 for finding, deleting, persisting and accessing repositories:

```
// $em is the EntityManager
$article = $em->find(
'DemoBlogBundle:Article', 42);

// mark the article as deleted
$em->remove($article);
$a = new Article();
$a->setTitle(„My Article“);

// attach article so its saved
$em->persist($a);

// write all changes to DB
$em->flush();
```

Repositories

Repositories provide advanced mechanisms for finding entity objects:

```
$repo = $em->getRepository
('DemoBlogBundle:Article');

$article = $repo->find(42);
$a2 = $repo->findOneByTitle(
  „My new article“);
$moreArticles = $repo->findBy(
  array('category' => 'php'));
```

Custom repository classes help organizing special find operations for one entity class.

Metadata Description

To enable Doctrine2 to map database entries to entity objects correctly, you must provide a description using one of the following:

- XML description files
- YAML description files
- a scripted PHP description
- or Docblock annotations

All metadata description approaches are semantically almost identical.

xml mapping

```
<doctrine-mapping xmlns=“...“
  xmlns:xsi=“http://www.w3.org
    /2001/XMLSchema-instance“
  xsi:schemaLocation=“...“>
 <entity name=“Demo\BlogBundle
  \Entity\Article“
  table=“article“>
   <id name=“id“ type=“integer“
       column=“id“>
    <generator strategy=“AUTO“/>
   </id>
   <field name=“title“
     type=“string“ length=“50“
     nullable=“false“
     unique=“true“ />
   <field name=“text“
     type=“text“ />
  </entity>
</doctrine-mapping>
```

yml mapping

```
MyProject\Entity\User:
  type: entity
  table: article
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    title:
      type: string
      length: 50
    text:
      type: text
```

php mapping

```
$metadata->mapField(array(
'id'=>true, 'fieldName'=>'id',
'type'=>'integer'));
$metadata->mapField(array(
'fieldName'=>'title',
'type' => 'string'));
```

annotation mapping

```
/**
* @ORM\Entity
* @ORM\Table(name=“article“)
*/
class Article
{
  /**
  * @ORM\Id
  * @ORM\Column(type=“integer“)
  * @ORM\GeneratedValue
  */
  protected $id;
  /**
   * @ORM\Column(type=“string“)
   * /
  protected $title;
}
```

We recommend using annotations for the meta data, as it keeps all information about the entity in one file.

Associations

http://bit.ly/symfony-associations

Doctrine2 handles 1:1, 1:n and n:m associations and supports all kind of special cases. A very basic example of a bidirectional 1:n association is the one between blog articles and comments:

```
use Doctrine\Common\
Collections\ArrayCollection;
/**
 * @ORM\Entity
 * @ORM\Table(name=“article“)
 */
class Article
{
  /**
   * @ORM\Id
   * @ORM\Column(type=“integer“)
   */
  protected $id;
  /**
   * @ORM\OneToMany(
   *   targetEntity=“Comment“,
   *   mappedBy=“article“)
   */
  protected $comments;
  public function __construct()
  {
    $this->comments =
      new ArrayCollection();
  }
}
/**
 * @ORM\Entity
 * @ORM\Table(name=“comment“)
 */
class Comment
{
  /**
   * @ORM\Id
   * @ORM\Column(type=“integer“)
   */
  protected $id;
  /**
   * @ORM\ManyToOne(
   *   targetEntity=“Article“,
   *   inversedBy=“comments“)
   */
  protected $article;
}
```

Inheritance

http://bit.ly/symfony-inheritance

There are three ways of mapping inheritance into a relational database schema which are supported by Doctrine2:

- **Mapped Superclasses:**
  No sharing of fields in the database. Each type of object is stored in a separate table with no interference.
- **Single Table Inheritance:**
  All objects of a type hierarchy are stored in one table. Specific columns („discriminator columns“) determine the type of object.
- **Class Table Inheritance:**
  For every class there is a separate table in the database. However, common attributes are managed in „base tables“.

All three mechanisms can be described with annotations. In each case, the superclass gets the annotation describing the kind of inheritance used:

```
/** @ORM\MappedSuperclass **/
class User
{
  // common attributes of all
}

/**
 * @ORM\Entity
 * @ORM\Table(„administrator“)
 */
class Administrator extends User
{
  //admin specific attributes
}

/**
 * @ORM\Entity
 * @ORM\Table(„visitor“)
 */
class Visitor extends User
{
  //visitor specific attributes
}
```

In this example, Doctrine2 would create two tables: „administator“ and „visitor“ which both would have the specific attributes as well as those of the common base class. Using single table or class table inheritance you can control how much duplicate information you have to store in your database.

Alternatives

Doctrine also exists for non-relational databases: There is a version for the NoSQL database MongoDB and one for the tree-oriented, semi-structured PHP content repository PHPCR.

Forms & Validation

Forms

Creating a form builder

```
namespace Demo\BlogBundle\
Controller;

use Symfony\Bundle\Framework
Bundle\Controller\Controller;
use Symfony\Component\Http
Foundation\Request;
use Demo\BlogBundle\Entity\
BlogPost;

class BlogController
  extends Controller
{
  public function newAction(
      Request $request
  ) {
    $post = new BlogPost();
    $form = $this->
      createFormBuilder($post)
      ->add('title', 'text')
      ->add('date', 'date')
      ->getForm();
    return $this->render(
      'DemoBlogBundle:Blog:new
      .html.twig', array('form'
        => $form->createView(),
    ));
  }
}
```

Additional configuration

```
$form = $this->createFormBuilder
(new BlogPost())
  ->add('title', 'text',
     array('label' => 'Title'))
  ->add('date', 'date', array(
    'widget' => 'single_text',
    'label'  => 'Date',
  ))
  ->getForm();
```

“label“ is available on every field type. For some field types (e.g. „date“), it is possible to configure the rendering. Custom widgets are possible as well. If no field type is provided, Symfony2 guesses the file type based on your validation rules.

List of Built-in Field Types:

- **Text Fields:** text, textarea, email, integer, money, number, password, percent, search, url
- **Choice Fields:** choice, entity, country, language, locale, timezone
- **Date and Time Fields:** date, datetime, time, birthday
- **Other Fields:** checkbox, file, radio
- **Field Groups:** collection, repeated
- **Hidden Fields:** hidden, csrf
- **Base Fields:** field, form

Rendering a form

```
<form action=“{{ path(
  'blog_new') }}“ method=“post“
  {{ form_enctype(form) }}>
    {{ form_widget(form) }}
    <input type=“submit“ />
</form>
```

A hidden field with a token gets rendered into the form automatically to prevent cross-site request forgery (CSRF).

Adding the novalidate attribute to the form tag prevents HTML5 form validation (i.e. for testing).

To achieve a custom structure, the elements of the form can also be rendered manually (see following example in the next column).

```
<form action=“{{ path(
  'blog_new') }}“ method=“post“
  {{ form_enctype(form) }}>
  {{ form_errors(form) }}
  {{ form_row(form.title) }}
  <div class=“custom_class“>
   {{ form_label(form.date) }}
   {{ form_widget(form.date) }}
  </div>
  {{ form_rest(form) }}
  <input type=“submit“ />
</form>
```

`{{ form_rest(form) }}` should always be added to make sure the CSRF-token is still rendered.

Process form submission

```
public function newAction(
    Request $request
) {
  if ($request->getMethod() ==
    'POST') {
    // $f is FormBuilder
    $f->bindRequest($request);
    if ($f->isValid()) {
      // save post to DB...
      return $this->redirect(
        $this->generateUrl(
          'home'));
    }
    ...
  }
}
```

`bindRequest` fills the form object with the submitted data. The controller action that handles a form usually does three things:

1. When called with `GET`, it renders the form.
2. When called with `POST` and valid data, it saves the data and redirects the user.
3. When called with `POST` and invalid data, it binds the data to the form and renders the form again.

Validation

Form validation

```
namespace Demo\BlogBundle\
Entity;
use Symfony\Component\Validator
\Constraints as Assert;
class BlogPost
{
  /**
   * @Assert\NotBlank()
   */
  protected $title;
  /**
   * @Assert\NotBlank()
   * @Assert\Type(„\DateTime“)
   */
  protected $date;
  ...
}
```

Calling `$form->isValid()` on the formBuilder automatically checks the validation rules of the provided model class. It is possible to specify a callback method or custom validation constraints. Validation rules can also be specified in YML, XML or PHP.

List of Built-in Validation Constraints:

- **Basic:** NotBlank, Blank, NotNull, Null, True, False, Type
- **String:** Email, MinLength, MaxLength, Url, Regex, Ip
- **Number:** Max, Min
- **Date:** Date, DateTime, Time
- **Collection:** Choice, Collection, UniqueEntity, Language, Locale, Country
- **File:** File, Image
- **Other:** Callback, All, Valid

The validator service

Data objects do not only need validation when working with forms (imagine an importer) – thus, Symfony2 provides the `validator` service independent of forms.

```
$post = new BlogPost();
$post->setTitle('My first');
$v = $this->get('validator');
// array of ConstraintViolation
$errs = $v->validate($post);
```

Bundles & Service Container

Bundles

Create a bundle

There is a simple command to generate the bundle skeleton. Just keep the defaults unless you have other conventions in your project.

```
app/console generate:bundle
  --namespace=Acme/BlogBundle
```

If you confirm, the command will also make the necessary changes to `app/AppKernel.php` and the main routing configuration.

Most important folders in a bundle:

- `Controller/` contains classes that handle request
- `Resources/config/` contains all configuration files of the bundle
- `Resources/views/` contains the bundle templates, organised by controller name
- `Resources/public/` is copied or symlinked to the web/ directory with assets:install, for images or other static files
- `Tests/` contains tests for your bundle

Adding a bundle to your project

The following applies to Symfony 2.0.. Symfony 2.1 will introduce a new system to track dependencies.*

To add a third party bundle to your project, first add it to the *deps* file in the main project folder.

deps

```
[mybundle]
    git=http://github.com/acme
      /SharedBundle.git
    target=/bundles/Acme
      /SharedBundle
    version=v1.0
[mylibrary]
    git=http://github.com/acme
      /my-library.git
```

- `target` is used to specify a directory, relative to the directory `vendor/` . If omitted the remote is checked out as subdirectory of `vendor/`
- `version` is optional, if omitted the current master is used. This can be a branch, a tag or a commit hash. (everything you can use with `git checkout`)
- Once *deps* is created, run `bin/vendors install` to fetch the code.

If the code is not in `vendor/bundles` you first need to add an entry to `app/autoload.php`

app/autoload.php

```
$loader->registerNamespaces(array(
    ... (other namespaces)
    'My\\Namespace' => __DIR__
     .'/../vendor/my-library',
));
```

Bundles need to be registered with the application kernel.

app/AppKernel.php

```
public function registerBundles()
{
  $bundles = array(
    ... (other bundles)
    new Demo\BlogBundle
       \DemoBlogBundle(),
  );
```

If the bundle provides its own routes you need to load the routing file in the main routing configuration. `prefix` can be used to prepend a path to all routes coming from that bundle.

app/config/routing.yml

```
AcmeSharedBundle:
  resource: „@AcmeSharedBundle/
             Resources/config/
             routing.yml“
  prefix:   /
```

Service Container

A service definition is only executed when the service is actually requested.

src/Acme/BlogBundle/Resources/config/services.xml

```
<parameters>
  <parameter key=“demo_blog.
    helper_class“>Demo\Blog
    Bundle\Helper\Helper
  </parameter>
  <parameter key=“demo_blog.
    param“>param</parameter>
</parameters>
<services>
  <service id=“demo_blog.
    helper“ class=“%demo_blog.
    helper_class%“>
    <argument>
      %demo_blog.param%
    </argument>
  </service>
  <service id=“demo_blog.main_
      controller“ ...
    <argument type=“service“
      id=“demo_blog.helper“/>
  </service>
</services>
```

Naming conventions to avoid name clashes: Bundle name with namespaces separated by „_“, then a „.“ and a telling name. Arguments are normal parameters to the service class constructor. They can be values (resp. configuration options resolved with „%“) or services identified by id.

Special service bootstrap

In the service definition, you can also call a method on the service class, or use `<file>` to include php files incompatible with PSR-0.

```
<service id=“acme_blog.main_
    controller“ ...
  <file>%kernel.root_dir%/../
    vendor/my-library/file.php
  </file>
  <call method=“setOption“>
    <argument>foo</argument
    </argument>
  </call>
...
```

Load service configuration directly

For simple cases, you can directly load the configuration file from your main `app/config/config.yml`

app/config/config.yml

```
imports:
- { resource: @DemoBlogBundle
  /Resources/config/services.yml }
```

Container extension for configurable services

If you have a complex bundle, you can write a Dependency Injection Extension to load your services and allow for a user friendly configuration, validation and so on.

See the Symfony2 cookbook entry for more information: http://bit.ly/symfony-extensions

Event System

Events

The Event system is one way to decouple different parts of your code and get around limitations of single-inheritance. It implements the Observer pattern. There are three elements needed to use events in Symfony: the Event itself, Event Listeners or Subscribers that react to events being fired, and the Event Dispatcher that matches the events to the listeners.

Core Events

- **kernel.request:** Incoming request – create response or prepare for controller
- **kernel.controller:** Controller will be called – use to change controller when needed
- **kernel.view:** Controller returned something else than a Response – use to implement your own view system
- **kernel.response:** A response is returned to the client – use to alter the response before it is sent out
- **kernel.exception:** An exception was thrown – use to implement your own error handling

Listeners and Subscribers

Listeners and Subscribers are two options to wait for an event to be triggered. The difference is that listeners are registered for specific events, while subscribers have a function that returns the events they want to listen to.

Each type of event has a name. Several listeners can be registered for the same event, with optional priorities to control who comes first. Any listener can stop propagation of the event by calling `$event->stopPropagation();`

Registering event listeners using service tags

The easiest way is to register a service as event listener by tagging it with `kernel.event_listener:`

```
service:
demo.blogbundle.post_listener:
 class: Demo\BlogBundle\
    EventListener\PostListener
 tags:
  -{name: kernel.event_listener,
   event: demoblog.newPostAdded,
   method: onNewPost,
   priority: 5 }
```

- **name:** must be `kernel.event_listener`
- **event:** name of the event you want to listen to
- **method:** method on the service class to call with the event
- **priority:** set the priority

Registering an event listener

An event listener can also be registered programmatically (see below for how to get the event dispatcher object):

```
use Demo\BlogBundle\Event\PostEvent;
class Listener
{
  function onNewPost(
    NewPostEvent $event)
  {
    // add post to RSS feed

...
$eventDispatcher->addListener(
  'demoblog.newPostAdded',
  array($listener, 'onNewPost'),
  5);
```

- The **first argument** is the event name used when triggering the event.
- The **second argument** defines the method to be called when the event happens. The array notation is used to call `$listener->onNewPost()`.
- The **third argument** is the priority, used when there are several listeners for the same event.

Registering an event subscriber

Event subscribers are always registered programmatically (see below for how to get the event dispatcher object):

```
use Symfony\Component\
EventDispatcher\
EventSubscriberInterface;

class NewPostEventSubscriber
  extends EventSubscriber
  Interface
{
  public function
    getSubscribedEvents()
  {
    return array(
      // without priority
      'demoblog.newPostAdded'
        => 'onNewPost'
      // with priority
     ,'demoblog.newPostAdded'
       => array('myMethod', 5)
    );
  }
  public function onNewPost(
    NewPostEvent $event)
  {
    // add post to RSS feed
    ...
```

The `getSubscribedEvents()` function has to return an array of 'event name' => function name on this object (or an array of function name, priority).

The subscriber is added to the event dispatcher with `$eventDispatcher->addSubscriber(new NewPostEventSubscriber());`

The event

To build your own events, you need an Event class. This class is simply a container for the context data of the event and passed to the listeners. It has to extend the `Symfony\Component\EventDispatcher\Event` class.

```
namespace Demo\BlogBundle\Event;

use Symfony\Component\Event
Dispatcher\Event;

class NewPostEvent extends Event
{
  public $post;
  public function __construct(
    $post
  ) {
    $this->post = $post;
  }
}
```

The event dispatcher

Listeners are registered in the Event Dispatcher (see above). To trigger an event, you tell the dispatcher which will call all registered listeners.

Usually you want to use the global event dispatcher service called `event_dispatcher` which you can get with dependency injection or from the container.

```
$event =
  new NewPostEvent($post);
$eventDispatcher->dispatch(
  'demoblog.newPostAdded', $event);
```

Testing

Symfony 2 uses phpunit for testing. If you are not familiar with this tool, you should first read http://bit.ly/symfony-phpunit

By convention, tests are placed in the `Tests` subfolder of the bundle, replicating the namespace structure of your bundle.

You should place a phpunit.xml.dist file into your app/ folder. Then you can run your tests with `phpunit -c app/`. Otherwise phpunit will try to run the tests of the vendor bundles and components as well.

Functional Tests

Your functional tests will need a working symfony environment. The easiest way to get one is by extending from: `Symfony\Bundle\FrameworkBundle\Test\WebTestCase`

src/Demo/BlogBundle/Tests/Controller/BlogControllerTest.php

```
namespace Demo\BlogBundle\
Tests\Controller;

use Symfony\Bundle\Framework
Bundle\Test\WebTestCase;

class BlogControllerTest
  extends WebTestCase
{
  public function testIndex()
  {
    $client =
      self::createClient();
    $crawler = $client->
      request('GET', '/blog');
    $this->assertTrue($crawler
      ->filter('html:contains(
      „Dummy Post“)')->count()
      > 0);
  }
}
```

Client

`client` is used to simulate a web browser connecting to symfony.

Notable methods on `client`:

- **request() method:** see below
- **Navigation:** back(), forward(), reload(), getHistory()
- **Internal objects:** getCookieJar(), getRequest(), getResponse(), getCrawler()
- **Configuration:** followRedirects(true|false), followRedirect()

Request

The request simulates a web request to a given url. It accepts a couple of parameters:

- **method:** GET, POST, ...
- **url:** path in symfony to request, i.e. /blog
- **parameters:** key-value array of request parameters, i.e. array(„id“ => 1) to simulate /blog**?id=1**
- **files:** uploaded files, same format as $_FILES
- **server:** values as used in $_SERVER, i.e. HTTP_ACCEPT_LANGUAGE or HTTP_REFERRER
- **content:** raw content for POST or PUT request

Crawler

The request method returns the crawler. This is a jQuery-like wrapper to work with the HTML DOM tree of the response. The crawler provides a fluent interface, you can chain calls.

- **filter('div.myclass'):** match on CSS selector
- **filterXpath('h1'):** match on XPath expression
- **eq(3):** access by index
- **first():** first node in list
- **last():** last node in list
- **siblings():** all siblings of current node
- **nextAll():** all following siblings
- **previousAll():** all preceding siblings
- **parents():** parent nodes from direct parent to top node
- **children():** direct children nodes
- **reduce($lambda):** nodes for which the callback $lambda returns true
- **attr('href'):** the value of the specified attribute of first node
- **text():** text content of first node
- **extract():** array of attribute values of all nodes. use _text for text value.

To do tests on the response header, use the response from $client->getResponse().

Testing interaction and forms

```
$l = $crawler->selectLink(
  'Home')->link();
$crawler = $client->click($l);
$f = $crawler->selectButton
  ('submit')->form();
$f['name'] = 'Tester';
$f['form_name[comment]'] =
  'Test comment';
$f['choice']->select('entry');
$f['agb']->tick();
$f['myfilefield']->upload(
  '/path/to/file.png');
// submit the form f
$crawler = $client->submit($f);
```

This example uses the additional convenience methods provided by the crawler:

- **selectLink():** find a link with this text or with an image with this alt attribute. Call link() on this to get a link object usable with the client.
- **selectButton():** select the button in a form. Call form() on this to get the form this button is part of.

Mayflower

Do you plan a project using Symfony 2, do you need help migrating from V1 to V2 or can we support you regarding the concept and analysis of your project? If so, Mayflower is your perfect project partner. With more than 60 PHP and JavaScript developers, we are able to realize large-scale projects within teams working in an agile workflow.

Short time-to-market cycles and outstanding know how in the fields of Symfony, PHP and JavaScript are the basis to ensure a unique implemetation by Mayflower. Optional, we offer a 24/7 support for you.

Contact us:
sales@mayflower.de
T 0049 931 35965 1177

Resources

There is awesome stuff out there, use it! **Try to not re-invent the wheel** but reuse existing bundles. If they are not perfect, fork them and do pull requests to make them perfect. This is more sustainable and keeps the open source eco system alive.

A good place to look for bundles is: http://knpbundles.com

Notable Bundles

Integrate services

- **DoctrineMongoDBBundle:** abstraction layer for the NoSQL database MongoDB
- **DoctrinePHPCRBundle:** abstraction layer for the tree-oriented PHP content repository
- **SncRedisBundle**: Integrate with the redis key-value store
- **FOSFacebookBundle:** Facebook integration
- **FOQElasticaBundle:** Integrate elasticsearch
- **NelmioSolariumBundle:** Integrate the solr client solarium
- **FOSTwitterBundle:** Twitter integration helper
- **RabbitMqBundle:** Integrate rabbit mq message queue server
- **LiipCacheControlBundle:** Integrate varnish (and potentially other) caching proxies
- **GoogleBundle:** Google analytics integration

Added Functionality

- **AsseticBundle:** Manage css and javascript, integrates less, sass, yui compressor & many more
- **FOSRestBundle:** Handle REST the easy way
- **FOSJsRoutingBundle:** Provide symfony routes to javascript
- **FOSCommentBundle:** Handle comments on any page
- **FOSUserBundle:** Extended user management
- **SonataAdminBundle:** Framework for admin backend for your entities
- **SonataUserBundle:** Integration of FOS-UserBundle with SonataAdminBundle
- **SonataMediaBundle:** Media management with SonataAdminBundle
- **TranslationEditorBundle:** Web based UI to edit translation files
- **KnpMenuBundle:** Build and render menu systems
- **KnpPaginatorBundle:** Pagination and sorting of data
- **KnpMarkdownBundle:** Markdown rendering
- **LiipImagineBundle:** Image manipulation library with twig integration
- **LiipThemeBundle:** Theming for symfony2 bundles
- **ChainRoutingBundle:** Uses several routers and can do routing based on content
- **StofDoctrineExtensionBundle:** To use the doctrine extensions in Symfony
- **JMSSecurityExtraBundle:** Additional features for the security component
- **LiipFunctionalTestBundle:** Additional testing features like HTML5 validation and fixture loading
- **BehatBundle, MinkBundle:** Integrate behat/mink testing framework into symfony

Further links http://www.mayflower.de/ http://www.symfony.com/ http://www.liip.ch/