Основы Symfony 3 и не только

Albert Matevosov

Содержание

Основы Symfony 3 и не только 0

1. Основы Symfony 1

    1.1 Структура директорий приложения 1.1

    1.2 Структура директорий бандла 1.2

    1.3 Отдача ответа 1.3

    1.4 Другие задачи контроллера 1.4

    1.5 Управление сессией 1.5

    1.6 Маршрутизация 1.6

    1.7 Генерация URL 1.7

    1.8 Применение Assetic для CSS и JS 1.8

2. Основы TWIG 2

    2.1 Три вида тегов в TWIG: 2.1

    2.2 Экранирование переменных и текста: 2.2

    2.3 Шаблоны, блоки, фильтры и функции 2.3

    2.4 Макросы TWIG 2.4

    2.5 Глобальные переменные TWIG 2.5

    2.6 Добавление своих глобальных перменных 2.6

    2.7 Вставка CSS и JS 2.7

3. Основные команды Symfony 3

4. Doctrine в Symfony 4

4.1 EntityManager (Unit Of Work) 4.1
4.2 Основные операции (SELECT, INSERT, DELETE) 4.2
4.3 Doctrine's Query Builder 4.3
4.4 Doctrine Query Language (DQL) 4.4
4.5 Изменение местоположения классов репозитория 4.5
4.6 Отношения метаданных (Relations) 4.6
4.7 Lifecycle Callbacks 4.7
4.8 Наследование сущностей в Doctrine 4.8
5. Service Container (Dependency Injection) 5
5.1 Что такое Service Conteiner 5.1
5.2 Определение сервиса 5.2
5.3 Параметры (зависимости) сервиса 5.3
5.4 Способы внедрения зависимостей (Dependency Injection) 5.4
5.5 Необязательные зависимости 5.5
5.6 Закрытые непубличные сервисы 5.6
5.7 Коллекция сервисов 5.7
5.8 Делегирование создания сервиса 5.8
5.9 Создание сервиса вручную 5.9
5.10 Класс Configuration 5.10
5.11 Динамическое добавление тегов 5.11
5.12 Паттерн “Стратегия” для загрузки exclusive сервиса 5.12
5.13 Загрузка и настройка дополнительных сервисов 5.13
5.14 Ручной выбор какой сервис будет использоваться

5.15 Полностью динамическое определение сервиса 5.14
5.16 Псевдонимы сервисов 5.16 5.15
5.17 Подключение файлов 5.17
5.18 Теги сервисов 5.18
6. Event Dispatcher 6
6.1 Что такое Event Dispatcher 6.1
6.2 Именование событий 6.2
6.3 Объекты событий 6.3
6.4 Подключения пользовательских слушателей 6.4
6.5 Таблица событий Symfony 6.5
7. Принцип работы компонента HttpKernel 7
7.1 Вызов фронт-контроллера 7.1
7.2 Метод handle() класса Kernel 7.2
7.3 Метод handle() класса HttpKernel 7.3
7.4 Метод handleRaw() класса HttpKernel 7.4
7.4.1 Начальная подготовка или возврат Response 7.4.1
7.4.2 Определение контроллера (Resolve Controller)
7.4.3 Переопределение контроллера 7.4.3 7.4.2
7.4.4 Получение аргументов контроллера 7.4.4
7.4.5 Вызов контроллера 7.4.5
7.4.6 Шаблонизатор 7.4.6
7.4.7 Финальная обработка ответа 7.4.7
7.5 Событие kernel.terminate 7.5

7.6 Схема работы компонента HttpKernel 7.6
8. Обработка исключений (Exceptions) 8
9. Подзапросы (sub-requests) 9
9.1 Что такое подзапросы 9.1
9.2 Где используются подзапросы 9.2
9.3 Выполнение подзапросов 9.3
10. Тестирование 10
11. Формы 11

Основы Symfony 3 и не только

Здравствуйте, меня зовут Матевосов Альберт. Я работаю техническим директором в региональной веб-студии "ДонИнтернет". Занимаюсь PHP-программированием уже более 5 лет - сперва на Drupal, а затем на Symfony. Для собственных нужд, а также для внутренних нужд компании было принято решение написать книгу по фреймворку Symfony.

Данная книга предназначена как для тех кто только начинает изучать Symfony, так и для тех кто хочет углубить свои знания. Помимо образовательной задачи, я постарался решить и задачу аккумулирования важной информации по фреймворку Symfony. Собрав и структурировав в одном месте основные команды, конструкции кода и т.п.

Хочу отметить, что данная книга не претендует на эталон книгописания. Это моя первая книга и, наверное, я нарушил в ней много правил русского языка, грамматики, орфографии и т.п. Однако это не мешает данной книге справляться с её основными задачами.

1. Основы Symfony

1.1 Структура директорий приложения

- **app/** - конфигурация приложения
- **src/** - сам PHP код проекта + templates + config
- **vendor/** - внешние библиотеки
- **web/** - публично доступные файлы

1.2 Структура директорий бандла

- **Controller/** - все контрллеры бандла
- **DependencyInjections/** - определение зависимостей, который могут быть внедрены как сервисы (не обязательная директория)
- **Resources/config/** - конфигурационные файлы, включая маршрутизацию
- **Resources/views/** - шаблоны, сгруппированные по имени контроллера
- **Resources/public/** - web-содержимое (css, js, картинки)
- **Tests/** -все тесты бандла

1.1 Отдача ответа

- Простейшая отдача ответа в контроллере:

```
use Symfony\Component\HttpFoundation\Response;

class HelloController
{
    public function indexAction($name)
    {
        return new Response('<html><body>Hello '.$name.'!
</body></html>');
    }
}
```

- Отдача ответа с использованием шаблона:

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

Метод **render()** создаёт объект **Response** с полученными параметрами, подключая шаблон.

Название шаблона: **BundleName:ControllerName:TemplateName**.

Для того, чтобы использовать метод **render()** контроллер должен наследовать базовый класс **Symfony\Bundle\FrameworkBundle\Controller\Controller**.

Объект Response

Класс **Response** - это абстракций вокруг ответа HTTP:

```
use Symfony\Component\HttpFoundation\Response;

// create a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, Response::HTTP_OK);

// create a JSON-response with a 200 status code
$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

Имеется несколько специальных классов для простого создания ответов определенных типов:

- JsonResponse
- BinaryFileResponse
- StreamedResponse

Подробнее - http://symfony.com/doc/current/components/http_foundation/introduction.html#component-http-foundation-response

1.4 Другие задачи контроллера

- Redirect - перенаправление на другую страницу

```
return $this->redirect('http://symfony.com/doc'); return $this->redirectToRoute('homepage', array(), 301);
```

- Forward - внутреннее перенаправление без смены URL

```
return $this->forward('AcmeHelloBundle:Hello:index',array('name'=>$name));
```

- Render Templates - подстановка шаблонов

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

- Accessing other services - доступ к сервисам

```
$router = $this->get('router');
```

- 404error response - отдача 404 ошибки

```
throw $this->createNotFoundException('Not exist');
```

- Получение POST/GET параметров из класса Request

```
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $page = $request->query->get('page', 1);
    // ...
}
```

- Выдача исключений:

```
throw new \Exception('Something went wrong!');
```

1.5 Управление сессией

```
$session = $request->getSession();
```

- Установка и получение атрибутов сессий

```
$session->set('name','Albert');
$session->get('name');
```

- Flash-сообщения - хранятся в сессии только на один доп. запрос

```
$session->getFlashBaf()->add('notice','Profile updated!');

foreach ($session->getFlashBag()->get('notice',array()) as
$msg) {
    echo '<div>'.$msg.'</div>';
}
```

или в шаблоне:

```
{% for flashMessage in app.session.flashbag.get('notice') %}
<div>{{ flashMessage }}</div>
{% endfor %}
```

1.6 Маршрутизация

Указание маршрутов в YAML

Настройки маршрутизации указываются в файле **/app/config/routing.yml**. Пример роутинга для блога:

```
article_show:
    path: /articles/{lang}/{year}/{title}.{_format}
    defaults: { _controller: AcmeBundle:Article:show, _format:
html }
    methods:  [GET]
    requirements:
        lang: en|fr|ru
        _format: html|rss
        _locale: ru
        year: \d+
```

Подключение внешнего файла роутинга с добавлением префикса:

```
acme_hello:
    prefix: /admin
    resource: "@AcmeBundle/Resources/config/routing.yml"
```

Указание маршрута в PHP аннотациях

Также имеется возможность указывать маршруты в аннотациях к контроллерам:

```
/**
 * @Route("/hello/{name}", name="hello")
 */
public function indexAction($name)
{
    return new Response('<html><body>Hello '.$name.'!</body>
</html>');
}
`
```

Просмотр и отладки маршрутов в консоли

- Просмотр всех маршрутов приложения:

```
php bin/console debug:router
```

- Просмотр информации о маршруте по его названию:

```
php bin/console debug:router homepage
```

- Определение маршрута по URL:

```
php bin/console router:match /blog/my-latest-post
```

1.7 Генерация URL

- в контроллере

```
$url = $this->generateUrl( 'blog_show', array('slug'=>'my-bkig-post') );
```

- с помощью сервиса 'router'

```
$url = $thi->get('router')->match('/blog/my-blog-post');
$url = $thi->get('router')->generate('blog_show',['name'=>$name]);
```

- в js файлах

```
var url = Routing.generate( 'blog_show',{"slug":'my-blog-post'} );
```

Для этого необходим бандл - **FOSJsRoutingBundle**.

- в шаблонах TWIG

```
<a href="{{ path( 'blog_show',{'slug':'my-blog-post'} ) }}">Read post</a>
```

или абсолютный путь:

```
<a href="{{ url('blog_show', {'slug': 'my-blog-post'}) }}">Read post.</a>
```

1.8 Применение Assetic для CSS и JS

Для подключения Assetic к бандлу, необходимо изменить файл App/config/config.yml:

```
assetic:
    bundles: [BloggerBlogBundle]
```

Можно удалить строку bundles и подключить Assetic ко всем бандлам.

```
{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/style.css'
    '@BloggerBlogBundle/Resources/public/css/blog.css'
    debug=false
    output='css/blogger.css'
    filter='?yui_css'
%}
    <link href="{{ asset_url }}" rel="stylesheet"
media="screen" />
{% endstylesheets %}
```

Параметры:

- **debug** - установка режима отладки
- **output** - указания выводимого имени файла
- **filter** - подключение фильтров (знак вопроса перед yui_css принуждает работать компрессию в DEV).

Для работы фильтра минимизации CSS, используя YUI компрессор надо обновить app/config/config.yml:

```
assetic:
    filters:
    yui_css:
    jar: %kernel.root_dir%/Resources/java/yuicompressor-
2.4.7.jar
```

Фильтры в библиотеке ядра:

- **CssMinFilter** - минимизирование CSS
- **JpegoptimFilter** - оптимизация JPEG файлов
- **Yui\CssCompressorFilter** - сжатие CSS с использованием YUI компрессора
- **Yui\JsCompressorFilter** - сжатие JavaScript с использованием YUI компрессора
- **CoffeeScriptFilter** - компиляция CoffeeScript для JavaScript

Полный список фильтров Assetic - https://github.com/kriswallsmith/assetic/blob/master/README.md.

```
{% javascripts
    'https://code.jquery.com/jquery-1.11.1.min.js'
    '@AcmeBlogBundle/Resources/public/js/scripts.js'
    output='js/scripts.js'
    filter='yui_js'
%}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

2. Основы TWIG

2.1 Три вида тегов в TWIG:

- `{{ … }}` — тег для вывода чего-либо;
- `{% … %}` — тег для вычисления какого-либо выражения, например, цикл или условие;
- `{# … #}` — тег для размещения комментариев в шаблоне.

2.2 Экранирование переменных и текста:

Переменная экранируется дважды: `{{ article.content | e }}`

Переменная не экранируется: `{{ article.content | raw }}`

Экранируется весь кусок HTML:

```
{% autoescape true %}
<ul><li>Экранировать этот HTML</li></ul>
{% endautoescape %}
```

Оставить HTML как есть:

```
{% raw %}
<ul><li>Оставить этот список как есть</li></ul>
{% endraw %}
```

2.3 Шаблоны,блоки, фильтры и функции

Наследование шаблона: `{% extends "base.html.twig" %}`

Дочерний шаблон может содержать только переопределяемые блоки, а использование html вне блоков вызовет ошибку.

Простая вставка шаблона: `{% include 'education.html.twig' %}`

Создание блока: `{% block title %} Добро пожаловать на мой сайт {% endblock %}`

Вставка контроллера:

```
<div>
    {{ render(controller('AppBundle:Article:recentArticles',{
'max': 3 })) }}
</div>
```

Вставка контроллера в асинхронном режиме с использованием hinclude.js:

```
{{ render_hinclude(controller('...')) }}
{{ render_hinclude(url('...')) }}
```

Применение фильтров: `{{ title|upper }}`

Встроенные фильтры: **date, format, replace, url_encode, json_encode, title, capitalize, upper, lower, striptags, join, reverse, length, sort, merge, default, keys, escape, e** (полный список)

Применение функций: `{{ cycle(['odd', 'even'], i) }}`

Встроенные функции: **range, cycle, constant, random, attribute, block, parent, dump, date.** (полный список)

2.4 Макросы TWIG

Макрос — это аналог функции в PHP. Они применяются для многократного повторного использования HTML тегов. Например создадим макрос в отдельном шаблоне **forms.html.twig**:

```
{% macro input(name, value, type, size) %}
    <input
        type="{{ type | default('text') }}"
        name="{{ name }}"
        value="{{ value | e }}"
        size="{{ size | default(20) }}" />
{% endmacro %}
```

В нужном шаблоне необходимо выполнить импорт данного файла и обратиться к макросу:

```
{% import "forms.html" as forms %}
<p>{{ forms.input('username') }}</p>
<p>{{ forms.input('password', null, 'password') }}</p>
```

2.5 Глобальные переменные TWIG

- **app.user** - объект пользователя
- **app.request** - объект запроса
- **app.session**** - объект сессии
- **app.environment** - текущее окружение
- **app.debug** - отдаёт "true" если включен "debug mode"

Чтобы узнать, с какого скрипта вызвали рендер шаблона необязательно передавать его имя. Достаточно обратиться до глобального значения (глобальной переменной, или метода) (в данном случае мы получим роут, с которого был вызван шаблон):

```
{% if app.request.attributes.get('_route') == "route_name" %}
    ...
{% endif %}
```

Также можем получить параметры:

`{{ app.request.attributes.get('_route_params') }}`

2.6 Добавление своих глобальных перменных

Кроме стандартных глобальных переменных есть возможность добавить свои. Для этого нужно модифицировать файл конфигурации (**config.yml**):

```
twig:
    # ...
    globals:
        test: "Some text"
```

Теперь эта переменная будет доступна во всех шаблонов:

```
<p>My variable - {{ test }}</p>
```

2.7 Вставка CSS и JS

```
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('css/main.css') }}"
rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('js/main.js') }}"></script>
        {% endblock %}
    </body>
</html>
```

Если требуется добавить свой стиль - наследуем блог и добавляем:

```
{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('css/contact.css') }}"
rel="stylesheet" />
{% endblock %}
```

3. Основные команды Symfony

- Генерирование нового бандла:

```
php app/console generate:bundle --namespace=Acme/TestBundle --format=[yml,xml,php,annotation]
```

- Генерирование новой сущности:php app/console doctrine:generate:entity
- Генерирование сеттеров и геттеров на основе yml, xml, php и аннотаций для сущности:

```
php app/console doctrine:generate:entities Acme/StoreBundle
```

- Создание базы данных:php app/console doctrine:database:create
- Обновление таблиц в базе данных (только DEV !!!):

```
php app/console doctrine:schema:update --force
```

- Создание миграций и их внедрение:

```
php app/console doctrine:migrtations:diff
```

```
php app/console doctrine:migrations:migrate
```

- Генерирование файла контроллера:

```
php app/console generate:controller
```

- Генерирование класса форма для сущности:

```
php app/console generate:doctrine:form AcmeStoreBundle:Comment
```

- Очистка кеша:

`php app/console cache:clear --env=[prod|dev|...]`

`php app/console cache:warmup --env=[prod|dev|...] --no-debug`

- Crud генерация:

  ```
  php app/console doctrine:generate:crud
  --entity=EnsJobeetBundle:Job
  --route-prefix=ens_job
  --with-write
  --format=yml
  ```

- Загрузка данных - fixtures:

`php app/console doctrine:fixtures:load`

- Копирование CSS и JS (при использовании стандартного Asset):

`php app/console assets:install web --symlink`

- Копирование CSS и JS (при использовании Assetic):

`php app/console assetic:dump --env=[prod|dev|...]`

- Генерация админа для SonataAdmin:

`app/console fos:user:create admintest admin@test.com pass --super-admin`

- Соната админ:

`sonata:admin:list sonata registred services`

`sonata:admin:explain sonata explain some services`

```
sonata:admin:generate CoreDeliveryBundle:Service
```

- Обновление библиотек в папке vendors согласно composer.json:

```
php composer.phar update
```

4. Doctrine в Symfony

4.1 EntityManager (Unit Of Work)

Unit Of Work - класс, который обслуживает набор объектов и управляет записью изменений в БД.

В момент `$em->persist(...);` сообщается обо всех изменениях в **Unit Of Work**. А в момент `$em->flush();` идёт запись в базу данных.

Реализация паттерна **Unit Of Work** следит за всеми действиями приложения, которые могут изменить БД в рамках одного бизнес-действия. Когда бизнес-действие завершается, **Unit of Work** выявляет все изменения и вносит их в БД.

4.2 DataBase (Doctrine)

- **SELECT** - получение данных

```
$em = $this->getDoctrine();
$product = $em->getRepository('AcmeStoreBundle:Product')-
>find($id);
```

- **INSERT** - добавление записей

```
$product = new Product();
$product->setName('MyName');

$em = $this->getDoctrine()->getManager();

$em->persist($product);
$em->flush();
```

- **DELETE** - удаление записей

```
$em->remove($product);
$em->flush();
```

4.3 Doctrine's Query Builder

```
$repository = $this->getDoctrine()->getRepository('...');

$query = $repository->createQueryBuilder('p')
->where('p.price>:price')
->setParameter('price','19.99')
->orderBy('p.ptice','ASC')
->getQuery();

$products = $query->getResult(); // getSingleResult();
getOneOrNullResult();
```

4.4 Doctrine Query Language (DQL)

```
$query = $em->createQuery(
'SELECT p
FROM AcmeStoreBundle:Product p
WHERE p.price > :price
ORDER BY p.price ASC'
)->setParameter('price', '19.99');

$products = $query->getResult();
```

4.5 Изменение местоположения классов репозитория

Чтобы файл с классами репозиториев не лежал в одной директории со стандартными классами сеттеров и геттеров сущностей, необходимо указать перед объявлением соответствующего класса сущности в аннотациях следующее:

```
/**
*
@ORM\Entity(repositoryClass="Acme\StoreBundle\Entity\Repository\Pr

*/
```

Сам файл репозитория выглядит следующим образом:

```
// src/Acme/StoreBundle/Entity/Repository/ProductRepository.php

namespace Acme\StoreBundle\Entity\Repository;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository {

    public function findOneProductById($id) {
        $query = $this->getEntityManager()->createQuery(
            'SELECT p FROM AcmeStoreBundle:Product p WHERE p.id
= :id'
        )->setParameter('id',$id);

        try {
            return $query->getSingleResult();
        } catch(\Doctrine\ORM\NoResultExcetion $e) {
            return null;
        }
    }
}
```

Теперь данный метод доступен в контроллерах:

```
$products = $em->getRepository('AcmeStoreBundle:Product')-
>findAllOrderedByName();
```

Также в классах репозитория имеется связка с UnitOfWork.

```
$this->matching(Criteria ::create()->where());
```

4.6 Отношения метаданных (Relations)

- One (category) to many (products)

```
use Doctrine\Common\Collections\ArrayCollection;

class Category {

    /**

    * @ORM\OneToMany(targetEntity="Product",
mappedBy="category")

    */

    protected $products;

    public function __construct() {

     $this->products = new ArrayCollection();

    }
}
```

- Many (products) to one (category)

```
class Product

{

     /**

     * @ORM\ManyToOne(targetEntity="Category",
inversedBy="products")

     * @ORM\JoinColumn(name="category_id",
referencedColumnName="id")

     */

     protected $category;

}
```

- Many (Products) to many (categories)

```
class Product {

 /**

 * @ORM\ManyToMany(targetEntity="Category",
inversedBy="products")

 * @ORM\JoinTable(name="product_categories")

 */

     private $categories;

}

class Category {

 /**

 * @ORM\ManyToMany(targetEntity="Producr",
mappedBy="categories")

 */

     private $products;

}
```

После указания зависимостей необходимо, чтобы доктрина сгенерировала новый аксессоры:

```
php app/console doctrine:generate:entities Acme
```

А также, чтобы добавила необходимые поля и ключи в таблицу БД:

```
php app/console doctrine:schema:update --force
```

4.7 Lifecycle Callbacks

Методы, которые необходимо выполнить во время различных стадий жизни сущности.

- **preRemove (postRemove)** - перед/после удаления
- **prePersist (postPersist)**
- **preUpdate (postUpdate)** - перед/после обновления
- **postLoad** - загрузка или обновление
- **loadClassMetadata** - после загрузки аннотаций

Методы выполняются только через **EntityManager**.

4.8 Наследование сущностей в Doctrine

Типы наследований:

- MappedSuperclass
- SingleTable
- JoinedTable

Class A { prop 1; prop 2; }

Class B extends A { prop 3; prop 4; }

Class C extends A { prop 5; prop 6; }

- MappedSuperclass

Table B (1 | 2 | 3 | 4)

Table C (1 | 2 | 5 | 6)

- SingleTable

Table A (1 | 2 | 3 | 4 | 5 | 6 | Type [B/C])

- JoinedTable

Table A (id | 1 | 2 | Type [B/C])

Table B (id | 3 | 4)

Table C (id | 5 | 6)

5. Service Container (Dependency Injection)

5.1 Что такое Service Conteiner

Сервис (service, служба) - это PHP объект, зарегистрированный в Service Container под уникальным ID. Сервис выполняет какую-либо "глобальную" задачу и может использоваться в любой части приложения, где необходим его функционал.

Service Container (контейнер внедрения зависимости) - это простой PHP объект, который управляет инициализацией сервисов.

5.2 Определение сервиса

Чтобы заставить контейнер создавать некоторый объект в качестве сервиса - нужно прописать соответствующую конфигурацию в YAML, XML или PHP:

```
# app/config/config.yml
services:
    my_mailer:
        class: Acme\HelloBundle\Mailer
        arguments: [sendmail]
```

Экземпляр объекта Acme\HelloBundle\Mailer теперь можно получить через контейнер служб. А сам контейнер доступен в любом контроллере Symfony2 при помощи вспомогательного метода get():

```
$mailer = $this->get('my_mailer');
$mailer->send('indo@doninter.ru', ... );
```

Когда запрашивается сервис my_mailer, контейнер создаёт его объект и возвращает его. Таким образом сервис не создаётся вплоть до того момента, когда он будет нужен вам. А сервисы, которые не используются - не будут созданы.

Контейнер сервисов создаётся с использованием одного конфигурационного ресурса (app/config/config.yml). Все прочие ресурсы для сервисов должны импортироваться. Существует два способа импорта конфигураций контейнера:

- Директива "imports" внутри config.yml

  ```
  # app/config/config.yml
  imports:
  hello_bundle:
      resource:
  @AcmeHelloBundle/Resources/config/services.yml
  ```

- Расширения контейнера (Container Extensions)

  Конфигурация контейнера для сторонних пакетов, включая сервисы ядра Symfony, как правило, загружается при помощи другого метода, более гибкого и просто для настройки в вашем приложении.

  Расширение контейнера сервисов - это PHP класс, созданный для выполнения следующих функций:

  - Импорт всех ресурсов контейнера сервисов, необходимых для конфигурации всех служб бандла;
  - Предоставление простой и понятной конфигурации, при помощи которой пакет можно настроить, не взаимодействуя напрямую с конфигурацией контейнера сервисов пакета.

  http://blog.doninter.ru/konfiguraciya-bandla-cherez-extension-klass

5.3 Параметры (зависимости) сервиса

Параметры или зависимости сервиса могут быть строковые, массивы или др. сервисы:

```
# app/config/config.yml
parameters:
    my_mailer.class: Acme\HelloBundle\Mailer
    my_mailer.transport: sendmail
    my_mailer.gateways:
        - mail1
        - mail2
        - mail3

services:
    my_mailer:
        class: %my_mailer.class%
        arguments: [%my_mailer.transport%]
        public: false
   newsletter:
        class: Acme\HelloBundle\Newsletter
        arguments: [@my_mailer]
```

5.4 Способы внедрения зависимостей (Dependency Injection)

Есть 2 способа внедрения зависимостей:

1. Типичный способ удостовериться в том, что сервис получил свои зависимости - это прописать зависимости в качестве аргументов конструктора.

   ```
   class SomeService
   {
    private $container;
    public function __construct(ContainerInterface
   $container)
    {
        $this->container = $container;
    }
   }
   ```

2. В некоторых случаях требуется не переопределять конструктор или не добавлять к нему дополнительные обязательные аргументы. Или какие-либо зависимости еще не определены в момент создания сервиса. В этих случаях можно добавить в класс сервиса метод-сеттер, позволив таким образом внедрять зависимость сразу после создания сервиса. Но в таком случае необходимо позаботиться, чтобы сеттер был обязательно вызван.

```
class SomeService
{
 private $container;

 public function setContainer(ContainerInterface $storage)
 {
     $this->container = $container;
 }

 public function getContainer()
 {
     return $this->container;
 }
}
```

- Преимущество использования сеттеров в том, что не требуется иметь аргумент в конструкторе для каждой зависимости. Иногда это означает что нет необходимости создавать конструктор вообще, или что можно оставить текущий конструктор без изменений.
- Единственный недостаток данного подхода в том, что можно забыть вызвать сеттер и получить ошибку из-за отсутствия необходимой зависимости. Для избежания данной ошибки можно обернуть вызовы зависимостей проверкой на их существование.

```
...
public function getContainer()
    {
           if (!($this->container instanceof
ContainerInterface)) {
                throw new \RuntimeException('Service container
is missing');
           }
        return $this->container;
    }
...
```

Компонент Symfony Dependency Injection включает в себя интерфейс ContainerAwareInterface и трейт ContainerAwareTrait (\Symfony\Component\DependencyInjection\ContainerAwareTrait), который уже содержит приватное свойство $container и его сеттер. Классы, в которые передаётся трейт ContainerAwareTrait автоматически получают контейнер через сеттер setContainer(). Стандартный класс Controller из бандла FrameworkBundle использует как раз ContainerAwareTrait.

Однако такой подход вызовет большое количество дублирующего кода в конфигурациях сервиса. Дабы избежать это, можно создать абстракный сервис, в который добавить вызов setContainer(). И уже этот сервис наследовать всеми остальными.

```
abstract_container_aware:
    abstract:  true    calls:        - [setContainer,
['@service_container]]
        some_service:
            class: AcmeBundle\Service\SomeService
            parent: abstract_container_aware
```

Родительский сервис не обязательно делать абстрактным. Если он будет использоваться напрямую - можно опустить параметр abstract.

5.5 Необязательные зависимости

Бывают ситуации, когда сервис знает как использовать другой сервис, но на самом деле он не является необходимым для его работы. Например сервис может знать как работать с логгером, чтобы залогировать что-нибудь с целью отладки.

Тут также имеется 2 способа внедрения зависимостей:

1. Можно использовать необязательный аргумент конструктора, для которого задать значение по-умолчанию null:

   ```
   ...
   public function __construct(
    LoggerInterface $logger = null
   ) {
    $this->logger = $logger;
   }
   ...
   ```

   Тогда при определении сервиса необходимо пометить необязательные зависимости знаком вопроса:

   ```
   services:
   some_service:
       class:  AcmeBundle\Service\SomeService
       arguments: ['@?logger’]
   ```

Для проверки внедрена зависимость или нет можно использовать конструкцию:

```
if ($this->logger instanceof LoggerInterface) {
...
}
```

2. Второй способ - использовать необязательные вызовы сеттеров.

```
...
public function setLogger(LoggerInterface $logger = null)
 {
     $this->logger = $logger;
 }
...
```

А при определении сервиса необходимо добавить вызов setLogger(). Когда сервис не доступен, можно указать, что он должен быть игнорирован, то есть зависимость необязательная:

```
services:
    some_service:
        class:  AcmeBundle\Service\SomeService
        calls:
            - [setLogger, ['@?logger’]]
```

5.6 Закрытые непубличные сервисы

При разработки приложений часто будет создаваться большое количество небольших сервисов, каждый с одной единственной зависимостью. Более сложные высокоуровневые сервисы будут использовать их в качестве внедряемых зависимостей. А нльбольшие низкоуровневые сервисы не предназначены для того, чтобы их использовать самостоятельно. В таких случаях правильно будет закрыть прямой доступ к низкоуровневым сервисам и пометить их как “непубличные”:

```
services:
    some_service:
        class: AppBundle\Service\SomeService
        public: false
```

Если сервис публичный (по-умолчанию), то он будет создан лишь однажды и будет возвращаться один и тот же его экземпляр. Если сервис приватный (параметр "public" установлен в "false"), то он может иметь несколько экземпляров одновременно.

5.7 Коллекция сервисов - TODO

Иногда необходимо внедрить коллекцию сервисов, которые используются одинаковым образом, например когда нужна обеспечить несколько альтернативных способов (стратегий) для получения чего-либо

// TODO

5.8 Делегирование создания сервиса - TODO

Вместо полного предварительного определения сервисов вместо с классом, аргументами и вызываемыми методами, можно опустить детали, которые в свою очередь будут получены во время выполнения, путём делегирования создания сервисв методу «Фабрика». Методы «Фабрика» могут быть либо статическими методами, либо методами объекта.

// TODO

5.9 Создание сервиса вручную - TODO

Обычно сервис создаётся путём загрузки определения сервиса из конфигурационного файла **services.yml**. Но порой сервисы мог быть не предопределены в нём. В таком случае они должны быть определены динамически, потомучто их название, класс, аргументы и т.п. не фиксированные.

// TODO

5.10 Класс Configuration

Для определения всех возможных конфигураций бандла используется класс **Configuration**, находящийся в файле **/AcmeBundle/DependencyInjection/Configuration.php**. Название класса на самом деле может быть любым, главное чтобы класс реализовывал интерфейс **ConfigurationInterface**.

```
class Configuration implements ConfigurationInterface
{
   public function getConfigTreeBuilder()
   {
       $treeBuilder = new TreeBuilder();
       $rootNode = $treeBuilder->root('name_of_bundle');
       $rootNode
        ->children()
            // Определение узлов конфигураций
        ->end()
       ;
       return $treeBuilder;
   }
}
```

В нём имеется один публичный метод: **getConfigTreeBuilder()**. Этот метод должен возвращать экземпляр класса **TreeBuilder**, который помогает вам в построении описания всех опций конфигурации, включая правила их валидаций.

Создание дерева настроек начинается с определения корневого узла:

```
$rootNode = $treeBuilder->root('name_of_bundle');
```

Название корневого узла должно быть названием бандла без “bundle” и в нижнем индексе с нижними подчеркиваниями. Корневой узел - это массив узлов. Узлов может быть сколько угодно:

```
$rootNode
    ->children()
        ->booleanNode('auto_connect')
            ->defaultTrue()
        ->end()
        ->scalarNode('default_connection')
            ->defaultValue('default')
        ->end()
->end()
```

Подробноее - http://symfony.com/doc/current/components/config/definition.html

Обычно объект класс Configuration используется внутри класса **BundlenameExtension**, которыйрасширяет базовый класс **Extension**, и располагается в файле **/src/AcmeBundle/DependencyInjection/AcmeExtension.php**. Его назначение - получение массивов конфигураций, которые также получает ядро Symfony из соответствующих конфигурационных YAML файлов.

```
class MatthiasAccountExtension extends Extension
{
    public function load(array $configs, ContainerBuilder
$container)
    {
        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );
    }
}
```

Метод **processConfiguration()** получает дерево конфигураций из объекта **Configuration** и обрабатывает полученные массивы.

5.11 Динамическое добавление тегов

5.12 Паттерн “Стратегия” для загрузки exclusive сервиса

5.13 Загрузка и настройка дополнительных сервисов

5.14 Ручной выбор какой сервис будет использоваться

5.15 Полностью динамическое определение сервиса

5.16 Псевдонимы сервисов

Можно создавать псевдонимы сервисом, в том числе и приватных:

```
services:
    foo:
       class: Acme\HelloBundle\Foo
       public: false

    bar:
       alias: foo
```

Теперь можно получить доступ к сервису "foo" запрашивая сервис "bar".

5.17 Подключение файлов

Для подключения некоторого файла прямо перед загрузкой сервиса имеется директива "file":

```
services:
    foo:
        class: Acme\HelloBundle\Foo\Bar
        file: %kernel.root_dir%/src/path/to/file/foo.php
```

5.18 Теги сервисов

Тег означает, что служба используется для некоторых специфических функций. Он сообщает сторонним пакетам, что ваша служба должна быть зарегистрирована или использована некоторым особым способом внутри целевого пакета. Список тегов, доступных в ядре Symfony:

- assetic.filter
- assetic.templating.php
- data_collector
- form.field_factory.guesser
- kernel.cache_warmer
- kernel.event_listener - использование сервиса в качестве слушателя.
    - { name: kernel.event_listener, event: xxx, method: onXxx }
- monolog.logger
- routing.loader
- security.listener.factory
- security.voter
- templating.helper
- twig.extension
- translation.loader
- validator.constraint_validator

Подробнее - http://symfony.com/doc/current/reference/dic_tags.html

6. Event Dispatcher

6.1 Что такое Event Dispatcher

Event Dispatcher — это PHP библиотека, представляющая собой легковесную реализацию шаблона проектирования Наблюдатель (Observer).

Диспетчер - это центральный объект системы обработки событий. Как правило, создаётся единственный диспетчер, который обслуживает реестр слушателей. Когда событие поступает к диспетчеру - он уведомляет всех слушателей, подписанных на это событие.

Когда сообщение отправлено, оно идентифицируется по уникальному имени (например, kernel.response), которое могут ожидать некоторое число слушателей. Также создаётся экземпляр класса **Symfony\Component\EventDispatcher\Event**, который затем передаётся всем слушателям. Как вы увидите чуть позже, объект Event часто содержит данные о направляемом событии.

6.2 Именование событий

При выборе имени события желательно следование нескольким простым правилам:

- Допустимые символы: буквы в нижнем регистре, цифры, точка (.), подчерк (_);
- Добавляйте префикс пространства имён с точкой на конце (например, kernel.);
- Оканчивайте имя глаголом, который обозначает действие (например, .request).

6.3 Объекты событий

Когда диспетчер уведомляет слушателей, он передаёт им объект **Event**. Базовый класс Event очень прост: он содержит метод для прекращения воспроизведения (event propagation) и ничего более.

Зачастую, необходимо передавать в объекте Event также данные о событии, чтобы слушатели могли их обработать тем или иным образом. В случае события kernel.response, объект Event, передаваемый каждому слушателю, фактически имеет тип **Symfony\Component\HttpKernel\Event\FilterResponseEvent**, дочерний по отношению к Event класс. Этот класс содержит методы, такие как getResponse и setResponse, позволяющие слушателям получать и даже заменять объект Response.

6.4 Подключения пользовательских слушателей

Для того чтобы подключить слушателя (listener), нужно его добавить в виде сервиса в один из конфигурационных файлов и пометить тегом "kernel.event_listener", с указанием в параметрах - наименования события и метода, который будет вызван:

```
services:
    kernel.listener.your_listener_name:
        class: Fully\Qualified\Listener\Class\Name
        tags:
            - { name: kernel.event_listener, event: xxx,
method: onXxx }
```

Во многих случаях, слушателю передаётся специализированный дочерний класс Event. Это даёт слушателю доступ к информации о событии. Необходимо сверяться с документацией или реализацией каждого конкретного события для определения какой именно экземпляр **Symfony\Component\EventDispatcher\Event** будет передан. Так событие kernel.event передаёт экземпляр класса **Symfony\Component\HttpKernel\Event\FilterResponseEvent**:

```
use Symfony\Component\HttpKernel\Event\FilterResponseEvent
public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    $request = $event->getRequest();
}
```

6.5 Таблица событий Symfony

Name	KernelEvents Constant	Argume
kernel.request	KernelEvents::REQUEST	GetRespon
kernel.controller	KernelEvents::CONTROLLER	FilterContr
kernel.view	KernelEvents::VIEW	GetRespon
kernel.response	KernelEvents::RESPONSE	FilterRespo
kernel.finish_request	KernelEvents::FINISH_REQUEST	FinishRequ
kernel.terminate	KernelEvents::TERMINATE	PostRespor
kernel.exception	KernelEvents::EXCEPTION	GetRespon

7. Принцип работы компонента HttpKernel

Компонент HttpKernel - это тонкая обёртка поверх классов Request и Response, которая приводит способы обработки запросов к единому стандарту. Компонент также предоставляет полную настраиваемость и расширяемость благодаря внедрению зависимостей (Dependency Injection) и мощной системе пакетов (Bundles).

7.1 Вызов фронт-контроллера

Сперва вызывается фронт-контроллер /web/app.php

```
...
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
...
```

Вся основная работа производится в методе **handle()**, который принимает на входе запрос (**$request**) и отдаёт ответ (**$response**);

7.2 Метод handle() класса Kernel

```
//  \Symfony\Component\HttpKernel\Kernel.php
public function handle(Request $request, $type =
HttpKernelInterface::MASTER_REQUEST, $catch = true)
{
   if (false === $this->booted) {
       $this->boot();
   }
   return $this->getHttpKernel()->handle($request, $type,
$catch);
}
```

В данном методе проверяется, загружено ли ядро и если нет - оно загружается: подключается все бандлы, Service Container и сервисы внутри бандлов (**$this->boot()**). Далее запрос передаётся в метод **handle()**, но уже экземпляра класса **HttpKernel**.

7.3 Метод handle() класса HttpKernel

```
//  \Symfony\Component\HttpKernel\HttpKernel.php
public function handle(Request $request, $type =
HttpKernelInterface::MASTER_REQUEST, $catch = true)
{
   try {
       return $this->handleRaw($request, $type);
   } catch (\Exception $e) {
       if (false === $catch) {
           $this->finishRequest($request, $type);
           throw $e;
       }
       return $this->handleException($e, $request, $type);
   }
}
```

Основную работу здесь производит приватный метод **handleRaw()**, находящийся в этом же классе. Если ловится Exception то вызывается соответствующий метод **handleException()**;

7.4 Метод handleRaw() класса HttpKernel

7.4.1 Начальная подготовка или возврат Response

Обычно HttpKernel будет пытаться сгенерировать ответ вызывая контроллер. Но какой-либо listener, который слушает событие KernelEvents::REQUEST(kernel.request) может самостоятельно сгенерировать полностью настраиваемый ответ с помощью метода setResponse().

```
...
//  \Symfony\Component\HttpKernel\HttpKernel.php
private function handleRaw(Request $request, $type =
self::MASTER_REQUEST)
{
   $event = new GetResponseEvent($this, $request, $type);
   $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);
   if ($event->hasResponse()) {
       return $this->filterResponse($event->getResponse(),
$request, $type);
   }
   ...
}
```

Существует стандартные слушатели данного события:

- **RouteListener** - берёт путь из объекта Response и пытается соонтести его с каким-либо маршрутом. И сохраняет результат в атрибутах объекта (_controller).

- **Firewall** - проверяет права на посещение защищных страниц и если пользователь не авторизован - то может самостоятельно сгенерировать Response с 403 ошибкой.

Таким образом, цель события **kernel.request** - это либо создание и возврат ответа Response напрямую, либо добавление какой-либо информации в атрибуты объекта Request.

7.4.2 Определение контроллера (Resolve Controller)

Далее идёт попытка определения необходимого контроллера для текущего Request объекта с помощью объекта \Symfony\Component\HttpKernel\Controller\ControllerResolver (экземпляр интерфейса ControllerResolverInterface).

```
// private function handleRaw
...
if (false === $controller = $this->resolver-
>getController($request)) {
   throw new NotFoundHttpException(sprintf('Unable to find the
controller for path "%s". The route is wrongly configured.',
$request->getPathInfo()));
}
...
```

В методе getController() в специальный атрибут (_controller) объекта Request помещается название контроллера (строка, которая затем преобразуется в PHP-вызов).

7.4.3 Переопределение контроллера

Теперь всё готово для вызова контроллера. Однако у нас есть еще последняя возможность переопределить контроллер с помощью события KernelEvents::CONTROLLER(kernel.controler)

```
// private function handleRaw
```

…

$event = new FilterControllerEvent($this, $controller, $request, $type);

$this->dispatcher->dispatch(KernelEvents::CONTROLLER, $event);

$controller = $event->getController();

…

Для этого необходимо вызвать метод setController() объекта FilterControllerEvent. Сам фреймворк не имеет своих слушателей события kernel.controller. Есть только сторонние бандлы, которые слушают событие.

```
Например, ControllerListener из бандла
SensioFrameworkExtraBundle делает очень важную работу перед
вызовом контроллера: собирает аннотации (@Template, @Cache и
пр.) и сохраняет их в атрибутах запроса _template, _cache и
пр.). Эти аннотации будут использоваться далее в процессе
обработки запроса.

ParamConverterListener из этого же бандла преобразует
дополнительные аргументы контроллера для создания экземпляров
сущностей, которые должны быть получены по id из маршрута.
```

/**

- @Route("/post/{id}")

  */

  public function showAction(Post $post)

  {

  ...

  }

7.4.4 Получение аргументов контроллера

На этом шаге уже точно определен контроллер, который будет вызван. Далее необходимо получить все аргументы, которые будут использованы:

```
// private function handleRaw
...
$arguments = $this->resolver->getArguments($request,
$controller);
…
```

Controller Resolver определяет какие аргументы ему передавать, используя реверс-инжиниринг PHP (Reflection). Затем он проходит в цикле все атрибуты контроллера и использует следующие правила, чтобы понять какие значения должны быть отправлены для каждого аргумента:

1. Если существует параметр запроса Request, совпадающий по названию с аргументом контроллера, тогда используется его значение. Например, если первый аргумент в контроллере **$year** и имеется параметр запроса - **year.public** `function indexAction($year,$month,$day) { ... }`
2. Если у аргумента в контроллере указан класс Request, тогда в качестве значения передаётся объект **Request.public** `function indexAction(Request $request) { ... }`

Также можно совмещать оба варианта в одном контроллере: `public function indexAction(Request $request, $year) { ... }`

7.4.5 Вызов контроллера

На этом шаге наконец вызывается сам контроллер.

```
...
$response = call_user_func_array($controller, $arguments);
 if (!$response instanceof Response) {
   // Если контроллер не отдал Response напрямую (см. далее)
}
…
```

Работа контроллера заключается в отдаче овтета. Это должна быть HTML страница, JSON, XML или что-то еще. В отличие от других частей процесса - этот шаг реализуется непосредственно программистами компании для каждой страницы приложения.

7.4.6 Шаблонизатор

Контроллер на предыдущем шаге должен был::1) Либо сразу отдать напрямую объект Response2) Либо отдать что-то еще, например массив шаблонизатора, который уже будет преобразован в объект. Для того, чтобы шаблонизаторы могли “вклиниться” на данном этапе имеется событие KernelEvents::VIEW (**kernel.view**).

```
...
if (!$response instanceof Response) {
    $event = new GetResponseForControllerResultEvent($this,
$request, $type, $response);
    $this->dispatcher->dispatch(KernelEvents::VIEW, $event);

    if ($event->hasResponse()) {
        $response = $event->getResponse();
    }

    if (!$response instanceof Response) {
        // Ошибка, так как не получен Response и отдача
Exception
    }

 }
...
```

Слушатели данного события могут вызывать метод setResponse() для установки своего ответа.

7.4.7 Финальная обработка ответа

Независимо от способа создания объекта Response на предыдущем шаге, в конце перед отдачей самого ответа вызывается событие **KernelEvents::RESPONSE (kernel.response)**

```
...
return $this->filterResponse($response, $request, $type);
...
```

Слушатели данного события могу изменить объект Response и даже заменить его на свой. Например WebDebugToolbarListener добавляет некоторый JavaScript в конец страницы в окружении DEV для отображения панели отладки.

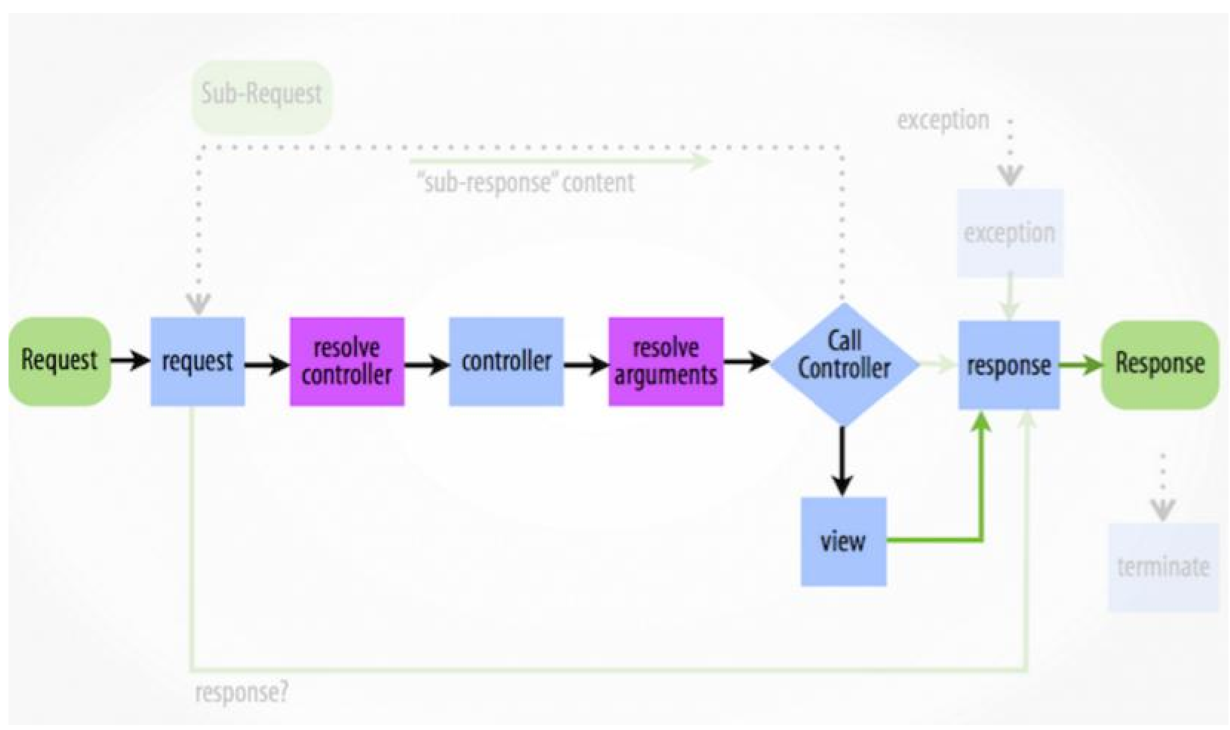
После данного события, конечный объект Response возвращается из метода handle(). В большенстве случаев вы можете вызвать метод send(), который отправить заголовки и выведет объект Response.

7.5 Событие kernel.terminate

Последним событием процесса HttpKernel является KernelEvents::TERMINATE (kernel.terminate), которе возникает уже после HttpKernel метода handle() и после того как ответ был отправлен пользователю. Основной целью является выполнение “тяжелых” операций, которые не будут задерживать отдачу ответа клиенту. Вызов terminate() происходит в /web/app.php:

```
$response = $kernel->handle($request);
$response->send();$kernel->terminate($request, $response);
…
```

7.6 Схема работы компонента HttpKernel



8. Обработка исключений (Exceptions)

Немаловероятно, что в процессе прохождения длинного пути от запроса до ответа будут появляться какие-либо ошибки. По-умолчанию Symfony ловитя любые исключения и пытается найти соответствующий ответ для них. В методе handle() класса HttpKernel вся обработка запроса обёрнут в блок try/catch:

```
//  \Symfony\Component\HttpKernel\HttpKernel.php
...
try {
   return $this->handleRaw($request, $type);
} catch (\Exception $e) {
   if (false === $catch) {
       $this->finishRequest($request, $type);
       throw $e;
   }
   return $this->handleException($e, $request, $type);
}
…
```

Если $catch установлена в true, то в случае выявления ошибки вызывается метод handleException(). Этот метод вызывает событие KarnelEvents::EXCEPTION (kernel.exception), которому передаётся объект GetResponseForExceptionEvent.

```
//  \Symfony\Component\HttpKernel\HttpKernel.php
private function handleException(\Exception $e, $request,
$type)
{
   $event = new GetResponseForExceptionEvent($this, $request,
$type, $e);
   $this->dispatcher->dispatch(KernelEvents::EXCEPTION,
$event);

   // a listener might have replaced the exception
   $e = $event->getException();
   if (!$event->hasResponse()) {
       $this->finishRequest($request, $type);
       throw $e;
   }
   $response = $event->getResponse();
   ...
}
```

Возможно 2 варианта:

1. Один из слушателей данного события установил надлежащий объект Response для специфичного исключения и заменил оригинальный объект Exception ($e).

   В данном случае HttpKernel проверяет новый объект Response для установки правильного статус-кода ответа. Если мы не установили свой код, то по-умолчанию будет отдаваться “500 - Internal server error”.

2. Если ни один слушатель не вызвал метод setResponse(), исключение будет выброшено повторно, но уже без автометической обработкой Symfony. В том случае если в настройках PHP включена директива display_errors - PHP выведет ошибку “как есть”.

9. Подзапросы (sub-requests)

9.1 Что такое подзапросы

Подзапросы выглядят и работают подобно остальным запросам, но обычно служат для обработки небольшой части страницы, вместо всей страницы целиком.

Методу **HttpKernel::handle()** передаётся специальный аргумент **$type**:

```
//  \Symfony\Component\HttpKernel\HttpKernel.php
public function handle(        Request $request,         $type
= HttpKernelInterface::MASTER_REQUEST,         $catch = true)
{
…
}
```

Существуют две константы в интерфейсе HttpKernelInterface:

1. **MASTER_REQUEST** - основной запрос
2. **SUB_REQUEST** - подзапрос

Для каждого запроса вашего приложение, первый запрос который обрабатывается ядром Symfony - это MASTER_REQUEST. Это задаётся неявно, так как аргумент $type не передаётся методу handle() во фронт-контроллерах.

Многие слушатели событий выполняются только для запросов типа MASTER_REQUEST. Для этог используется специальный метод Kernel:Event:isMasterRequest() Например компонент Firewall ничего не

делает, если это подзапрос:

```
public function onKernelRequest(GetResponseEvent $event)
{
    if (!$event->isMasterRequest()) {
       return;
    }
    ...
}
```

9.2 Где используются подзапросы

Подзапросы используются для изолирования создания объекта Response. Например, когда исключение перехватывается ядром Sf, стандартный обработчик исключения пытается выполнить исключение назначенное контроллером. Для этого создаётся подзапрос.

Чаще всего подзапросы выполняются из контроллера или внутри шаблона. Так, всякий раз, когда вы “рендерите” другой контроллер внутри шаблона twig, создаётся и обрабатывается отдельный подзапрос.

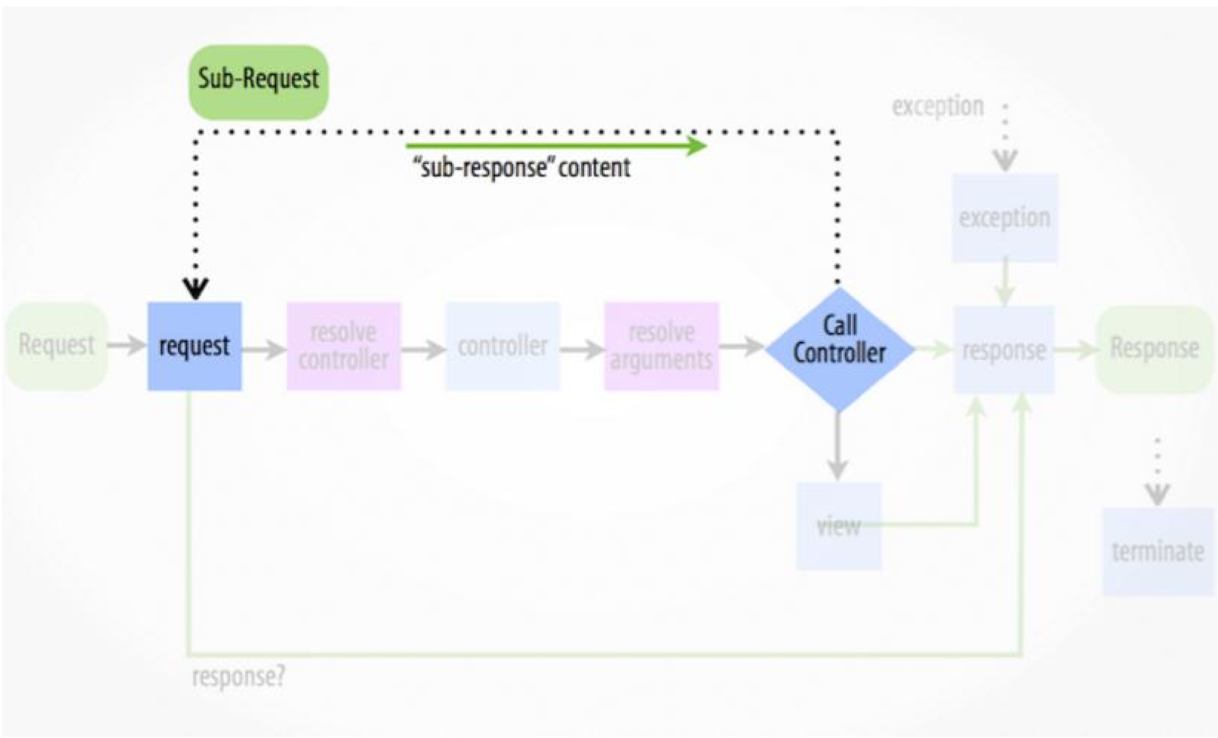
```
{{ render(controller('BlogBundle:Post:list')) }}
```

9.3 Выполнение подзапросов

Для выполнения подзапроса можно использовать метод HttpKernel::handle, изменив в нём второй аргумент:

'$response = $kernel->handle($request, HttpKernelInterface::SUB_REQUEST);'

В результате создаётся другой полный цикл запрос-ответ, в котором новый запрос Request трасформируется в ответ Response. Единственная разница в том, что некоторые слушатели реагируют только на MASTER_REQUEST.



10. Тестирование

11. Формы