Один год с Symfony

Пишем правильный, переиспользуемый код для Symfony 2

Matthias Noback
перевод: Дмитрий Быкадоров

Один год с Symfony

Перевод книги “A year with Symfony” от Matthias Noback

Dmitry Bykadorov и Matthias Noback

Эта книга предназначена для продажи на http://leanpub.com/a-year-with-symfony-ru

Эта версия была опубликована на 2016-11-24

Оглавление

От переводчика

Книгу “Один год с Symfony” написал разработчик из Голландии - Matthias Noback. Книга в английском варианте доступна на сайте leanpub.

Маттиас, по завершению работы над книгой, сделал её доступной бесплатно, так что вы можете обратиться к первоисточнику. Сам же я об этой книге я узнал случайно, из какой-то рассылки, как раз когда она стала бесплатной. Да, она про Symfony2, но она описывает и более общиее принципы разработки нежели просто версию одного фреймворка, так что я полагаю, что эта книга еще долго будет актуальна. Во всяком случае для тех, кто использует Symfony3 она также “must read”.

Касательно перевода: некоторые фразы и конструкции я перевел, как мне кажется, в более литературном виде. Некоторые, привычные разрабочикам термины, я не стал переводить и просто транслитерировал, например: фреймворк, бандл. Имена же, наоборот, оставил без перевода на случай, если вы захотите загуглить - кто же это такой. Небольшие соглашения по наименованиям:

- framework - фреймворк;
- bundle - бандл;
- controller - контроллер;
- router (route) - маршрутизатор (маршрут)

Мои примечания будут расположены в тексте в таком виде: (@dbykadorov: текст примечания). Если же примечание будет большое, это будет отдельный абзац, начинающийся с @dbykadorov. Также я намерен проверять все примеры на Symfony 3.2, так что, если будут какие-то расхождения в работе ныне актуальной версии Symfony с той, которую использовал Маттиас (2.3) - я укажу на это и предложу также вариант для 3.2.

Если у вас есть предложение как улучшить перевод - пишите мне или делайте pull-request.

Я надеюсь, перевод и книга вам поравятся.

Happy coding!

Dmitry Bykadorov

Предисловие

От Luis Cordova

Большинство open source проектов имеют свою историю и серьёзные основания для их появления и развития. PHP фреймворк Symfony2 (а теперь и 3) активно развивается последние несколько лет. Многие разработчики, попробовав использовать этот фреймфорк, испытывали и испытывают сложности, разбираясь в нюансах его функционирования. И, хотя большинство из них так или иначе преодолевают все трудности, в конце концов у них остаётся много сомнений, касательно того как же всё-таки правильно разрабатывать в стиле Symfony.

У Symfony1 основной документацией была книга, котороя освещала основые особенности и практики в использовании этого фреймворка. У Symfony2 также есть книга, которая является основной документацией для него (@dbykadorov: на начало августа 2016 года это уже не так - был произведен крупный рефакторинг структуры документации, с целью сделать её более дружелюбной для новичков. Подробнее читайте тут. И да, я надеюсь что кто-то читает предисловия). Приложением к книге идёт “Книга Рецептов” - Cookbook, которая более детально раскрывает некоторые аспекты практического использования фреймворка. Тем не менее, за прошедшие годы, далеко не все аспекты использования и инженерные практики были отражены в этих книгах, однако это не значит, что они менее важны и востребованы разработчиками, которые хотят знать ответы не только на вопрос “как”, но и “почему?”. Разработчики также испытывают необходимость в изучении “лучших практик”, которые постигаются только в ежедневном использовании Symfony на реальных проектах. В сети есть блоги о разработке на Symfony, но они разрознены и требуют от читателей знания множества вещей, в том числе и экспериментальных фич и особенностей фреймворка. Чего всем этим страждущим знаний не хватало до сих пор - это авторитетного технического заключения, представленного в виде книги и указывающего на путь в стиле Symfony.

Matthias работает с Symfony уже много лет, отлично понимает его изнутри и отвечает на наши вопросы “почему”. Эту миссию я не доверил бы никому кроме Сертифицированного Разрабочика Symfony. Изредка я чувствую, что понимаю Symfony настолько хорошо, что могу использовать его в своём арсенале. Во время чтения этой книги - был как раз такой момент. Другой раз такое ощущение у меня было, когда я читал Kris Wallsmith - разъяснения о том как работает Symfony Security Component. Я считаю, эта книга “Один год с Symfony” должна быть в арсенале кажого разработка, который стремится к глубокому пониманию фреймворка.

Matthias в этой книге раскрыл мне много секретов, так что я думаю вы тоже будете частенько подглядывать в неё, чтобы посмотреть ту или иную рекомендацию, и узнать, что надо делать в том или ином случае. Matthias также написал о таких вещах, о которых я даже не думал, что здесь их найду и он сделал это в очень доходчивой манере, с примерами кода.

Я очень надеюсь, что вам понравится эта книга.

Ваш друг из сообщества symfony,

Luis Cordova (@cordoval)

Введение

Один год с Symfony. На самом деле, для меня это был даже не год, а почти 6 лет. Начинал я с symfony 1 (именно так, с маленькой буквы и отдельно стоящая единичка), потом продолжил с Symfony2. Symfony2 - это то, что можно охарактеризовать как “взрослый” фреймворк, с его помощью вы можете делать весьма продвинутые вещи. И когда вы захотите сделать что-нибудь продвинутое, вам даже не обязательно устанавливать весь фреймворк, вы можете воспользоваться одим или несколькими из его компонентов.

Начало работы с Symfony2 означало для меня следующее: изучение множества вещей о программировании в общем и применение многих вещей, о которых я узнал из книг к любому коду, который я написал с тех пор. Symfony2 сделал это: воодушевил меня делать вещи правильно.

В то же время я много писал о Symfony2, внося вклад в его документацию (конкретно - некоторые статьи из Cookbook и документацию на Security и Config компоненты), я запустил свой блог со статьями о PHP в общем, Symfony2, его компонентах и сопутствующих фреймворках и библиотеках, таких как Silex и Twig. И я стал Сертифицированным разработчиком Symfony2 во время самой первой экзаменационной сессии в Париже, на Symfony Live Conference в 2012 году.

Всё это нашло отражение в книге, которую вы сейчас читаете - “Один год с Symfony”. Она содержит многие из лучших практик, которые я и мои уважаемые коллеги из IPPZ разработали, трудясь над крупными приложениями на Symfony2. Она также наделит вас более глобокими познаниями, которые вам потребуются, когда вы копнёте чуть глубже, чем просто написание контроллеров или шаблонов.

Благодарности

Прежде чем я продолжу, позвольте мне поблагодарить несколько человек, которые помогли мне закончить эту книгу. В первую очередь, Luis Cordova, который следовал по моим стопам, с тех пор как я впервые начал писать о Symfony2 в 2011. Он провел исчерпывающий анализ первых черновиков. Мои коллеги из IPPZ также предоставили мне очень ценные замечания,воодушевляя меня делать некоторые вещи более понятными, а другие - более интересными: Dennis Coorn, Matthijs van Rietschoten и Maurits Henneke. Работая с ними два года, разделяя с ними опасения по поводу поддерживаемости, читабельности, повторного использования и прочих насущных вопросов, таких как смехотворность (@dbykadorov: здесь речь шла о всяких “*bilities”, например “laughability”, не уверен в корректности перевода) я получил массу положительных эмоций. Также хочу поблагодарить Lambert Beekhuis, организатора митапов датской юзергруппы Symfony2, за то что дал мне очень ценные советы касательно моего английского.

Кому предназначена эта книга

Я написал эту книгу для разработчков, которые хорошо знают PHP, но с Symfony2 знакомы несколько недель, может быть месяцев. Я предполагаю, что вы прочли официальную документацию Symfony2 и уже знакомы с основами создания приложения на Symfony2. Я также полагаю, что вы уже знаете базовую структуру приложения (стандартную структуру директорий, как создать или подключить бандл), как создать контроллер и сконфигурировать маршрутизатор для него, как создавать формы и Twig шаблоны.

Я также полагаю, что вы успели поработать с какой-либо библиотекой для взаимодействия с базами данных, например Doctrine ORM, Doctrine MongoDB ODM, Propel, и так далее. Тем не менее, в этой книге для упрощения я буду использовать только Doctrine. Если вы используете другую библиотеку для сохранения объектов, вы вероятно сможете разобраться, как применить идеи изложенные в этой книге к коду, написанному под вашу библиотеку.

Соглашения

Так как эта книга только про Symfony2, с данного момента я буду писать просто “Symfony” - это выглядит более элегантно. Всё что я скажу о Symfony, будет относиться к версии 2. Я написал и протестировал примеры кода для этой книги на Symfony 2.3. Тем не менее, они могут быть вполне применимы к Symfony 2.1.* и 2.2.* и, возможно, к Symfony 2.0.*. (@dbykadorov: версия 3 конечно имеет отличия от версии 2, но большинство сказанного, особенно базовые принципы разработки типа “low coupling” - будут также справедливы и для неё. Я постараюсь отметить эти отличия в ходе перевода и протестировать все примеры на Symfony 3.2)

В этой книге я покажу примеры кода самого фреймворка Symfony. Для удобства отображения на странице и большей читабельности, время от времени я модифицировал его.

Обзор содержания книги

Первая часть этой книги называется “путешествие от запроса до ответа”. Она проведёт вас от точки входа в приложение Symfony во фронт-контроллере, до последнего вздоха, который фреймворк делает перед тем, как отправить ответ клиенту. Я покажу как внедриться в этот процесс и модифицировать его или же изменить результаты промежуточных шагов.

Следующая часть называется “Шаблоны внедрения зависимостей”. Она содержит коллекцию шаблонов, которые являются решениями периодически возникающих проблем, возникающих приложение создании или модификации сервисов, основанных на конфигурации бандла. Я покажу вам много практических примеров, которые вы сможете использовать для создания расширений, конфигурационных классов и проходов компилятора (compiler passes) для ваших бандлов.

Третья часть будет посвящена структуре проекта. Я предложу различные способы как сделать ваши контроллеры более понятными, путём делегирования действий обработчикам форм (form handlers), доменным менеджерам (domain managers) и слушателям событий (event listeners). Мы также посмотрим на состояния и как избежать их на сервисном уровне вашего приложения.

Далее последует небольшое интермеццо о соглашениях по конфигурированию. Эта часть должна будет помочь вам наладить конфигурирование вашего приложения. Эта глава воодушевит вас принять и использовать некоторые полезные соглашения по конфигурированию.

Пятая часть очень важна, так как она касается любого более-менее серьёзного приложения, использующего пользовательские сессии и уязвимые данные, например пароли пользователей. Эта часть будет о безопасности. В идеале здесь должны были бы быть затронуты все компоненты Symfony (в конце концов и сам фреймворк прошел аудит безопасности) и Twig, но, к сожалению, это не возможно. Вы всегда должны быть начеку и заботиться о безопасности вашего приложения. Эта часть книги содержит различные советы о том, как быть с безопасностью приложения, на что обратить внимание, когда вы можете положиться на фреймворк и когда вам нужно контролировать безопасность самостоятельно.

Шестая целиком будет посвящшена аннотациям. Когда Symfony2 впервые вышел в релиз в 2011 году, он представил всем аннотации как революционный способ конфигурирования приложения через док-блоки классов, методов и свойств. Первая глава этой части разъясняет как аннотации работают. После этого вы узнаете как создавать свои собственные аннотации и как можно использовать аннотации для того, чтобы воздействовать на ответ, который генерируется для запроса.

Заключительная часть будет о том, как быть Symfony-разработчиком, хотя, в сущности, эта часть будет вдохновлять вас не быть только разработчиком Symfony, но писать код как можно менее зависимый от Symfony или от любого другого фреймворка. Это означает разделение кода на повторно используемый и специфичный для конкретного проекта, а затем выделение повторно используемого кода в библиотеки и бандлы. Я буду обсуждать и прочие идеи, который сделают ваш код красивым, чистым и дружелюбным к другим проектам.

Наслаждайтесь!

I От запроса до ответа

1 HttpKernelInterface

Symfony знаменит благодаря своему HttpKernelInterface:

```
namespace Symfony\Component\HttpKernel;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

interface HttpKernelInterface
{
    const MASTER_REQUEST = 1;
    const SUB_REQUEST = 2;

    /**
     * @return Response
     */
    public function handle(
        Request $request,
        $type = self::MASTER_REQUEST,
        $catch = true
    );
}
```

Реализация этого интерфейса должна содержать один метод и с его помощью иметь возможность каким либо образом превратить полученный запрос в ответ. Если вы взглянете на любой из фронт-контроллеров в директории /web вашего Symfony проекта, вы можете увидеть, что этот метод `handle()` играет главную роль в обработке веб-запроса - чего и стоило ожидать:

```
// /web/app.php
$kernel = new AppKernel('prod', false);
// ...
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
// ...
```

Сначала создаётся экземпляр ядра AppKernel. Это класс - специфичный для вашего приложения и вы можете найти его в диретории app: /app/AppKernel.php. Он позволяет регистрировать ваши бандлы и изменять некоторые основные настройки, такие как распложение директории с кэшем или указание какой конфигурационный файл нужно загрузить. Аргументы его конструктора - это наименование окружения и флаг активации режима отладки в ядре (debug mode).

Окружение

Окружением (или именем окружения), может быть любая строка. В основном, это способ определить, какой конфигурационный файл должен быть загружен (например, config_dev.yml или же config_prod.yml). Эта проверка производится в классе `AppKernel`:

```php
public function registerContainerConfiguration(LoaderInterface $loader)
{
    $loader
        ->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
}
```

Режим отладки

В режиме отладки у вас будут следующие возможности:

- Удобная и информативная страница ошибки, отображающая информацию о запросе для дальнейшей отладки;
- Подробные сообщения об ошибках если страница ошибки из предыдущего пункта не может быть отображена;
- Исчерпывающая информация о времени выполнения отдельных частей приложения (начальная загрузка, обращения к базе данных, рендеринг шаблонов и так далее).
- Расширенная информация о запросах (с использованием веб-профайлера и сопутствущей панели).
- Автоматическая инвалидация кэша: эта функция позволяет не беспокоиться о том что изменения в config.yml, routing.yml и прочих конфигурационных файлах не будут учтены без пересборки всего сервисного контейнера или сопоставителя маршрутов (routing matcher) для каждого запроса (однако, это занимает больше времени).

Продолжаем разбирать процесс обработки запроса: далее создается объект Request, базирующийся на существующих суперглобальных массивах ($_GET, $_POST, $_COOKIE, $_FILES и $_SERVER). Класс Request вместе с прочими классами компонента HttpFoundation предоставляет объектно-ориентированный интерфейс к этим суперглобальным массивам. Все эти классы также покрыват разичные проблемные ситуации, которые могут возникать при использовании разных версий PHP или же разных платформ. Всегда использовать объект Request для получения различных данных о запросе, вместо того чтобы использовать суперголобальные переменные, это разумный выбор (в контексте Symfony).

Итак, далее вызывается метод `handle()` экземпляра AppKernel. Его единственным аргументом является объект Request для текущего запроса. Аргументы для типа запроса (“master”) и нужно ли перехватывать и обрабатывать исключения (да, перехватывать) берутся по умолчанию. Результат метода `handle()` гарантированно будет экземпляром класса Response (также являющегося частью компонента HttpFoundation). И, наконец, ответ будет отправлен обратно клиенту, который сделал запрос, например браузеру.

1.1 Загрузка ядра

Как вы уже могли догадаться, вся магия происходит внутри метода `handle()` в ядре. Вы можете найти реализацию этого метода в классе Kernel, который является родителем класса AppKernel:

```php
// Symfony\Component\HttpKernel\Kernel

public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    if (false === $this->booted) {
        $this->boot();
    }

    return $this->getHttpKernel()->handle($request, $type, $catch);
}
```

Во-первых, проверяется что ядро загружено, прежде чем произойдёт обращение к HttpKernel. Процесс загрузки включает в себя:

- Инициализация всех зарегистрированных бандлов;
- Инициализация сервисного контейнера;

Бандлы и расширения контейнера

Бандлы широко известны среди разработчиков на Symfony как место, где размещается разрабатываемый вами код. Каждый бандл должен иметь имя, котрое отражает его назначение. Например, у вас могут быть такие бандлы: BlogBundle, CommunityBundle, CommentBundle, и так далее. Вы регистрируете ваши бандлы в AppKernel.php, добавляя их к существующему списку:

```php
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            // ... тут прочие бандлы
            new Matthias\BlogBundle()
        );

        return $bundles;
    }
}
```

И это определённо хорошая идея - можно легко расширить функциональность вашего проекта, добавив лишь одну строчку кода. Тем не менее, когда мы смотрим на ядро Kernel и на то, как оно работает с бандлами, включая ваши, становится ясно, что бандлы прежде всего понимаются как способ расширения сервисного контейнера, а не как библиотеки кода. Вот почему вы найдёте директорию DependencyInjection внутри любого бандла, а внутри неё класс {наименованиеБандла}Extension. В процессе инициализации сервисного контейнера, каждый бандл имеет возможность зарегистрировать собственные сервисы в сервисном контейнере, также можно добавить несколько параметров, и даже при необходимости модифицировать определения некоторых сервисов, прежде чем контейнер будет скомпилирован и выгружен в директорию с кэшем:

```php
namespace Matthias\BlogBundle\DependencyInjection;

use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;

class MatthiasBlogExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader($container,
        new FileLocator(__DIR__.'/../Resources/config'));

        // добавляем определения сервисов в контейнер
        $loader->load('services.xml');

        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );

        // добавляем параметр
        $container->setParameter(
            'matthias_blog.comments_enabled',
            $processedConfig['enable_comments']
        );
    }

    public function getAlias()
    {
        return 'matthias_blog';
    }
}
```

Имя, возвращаемое методом getAlias() - это фактически ключ, по которому вы можете устанавливать значения параметров (например в config.yml):

```yaml
matthias_blog:
    enable_comments: true
```

Подробнее о конфигурировании бандлов вы узнаете в следующем разделе - “Шаблоны внедрения зависимостей”. Каждый корневой ключ в конфигурации соответствует определенному бандлу. В примере выше вы могли заметить, что `matthias_blog` это ключ к настройкам, относящимся к `MatthiasBlogBundle`. Так что для вас не будет большим сюрпризом, что это также справедливо для всех корневых ключей, которые вы можете встретить в файле `config.yml` и прочих ему подобных: настройки, доступные по ключу `framework` относятся к `FrameworkBundle`, ключ `security` (даже если он определен в другом файле - `security.yml`) относится к `SecurityBundle`. Проще простого!

Создание сервисного контейнера

После того, как все бандлы подключили свои сервисы и параметры, создание контейнера завершается процессом, которые имеет наименование “компиляция”. В ходе этого процесса всё ещё остаётся возможность внести последние изменения в определения сервисов или изменить параметры. Также, это хороший момент для того, чтобы проверить правильность

и оптимизировать определения сервисов. После этого контейнер принимает свой окончательный вид и он дампится на диск в двух различных форматах: в XML файл с определениями всех обнаруженных сервисов и параметров и в PHP файл, готовый для использования в качестве единого и единственного сервисного контейнера для вашего приложения. Оба эти файла вы можете найти в директории с кэшем, соответствующей окружению с которым было загружено ядро, например, /app/cache/dev/appDevDebugProjectContainer.xml. XML файл выглядит как любой другой файл с определениями сервисов, только намного больше:

```
<service id="event_dispatcher" class="...\ContainerAwareEventDispatcher">
    <argument type="service" id="service_container"/>
    <call method="addListenerService">
        <argument>kernel.controller</argument>
        <!-- ... прочие аргументы, если нужно -->
    </call>
    <!-- ... прочие параметры сервиса, если нужно -->
</service>
```

PHP файл содержит метод для каждого сервиса, который может быть запрошен. Вся логика создания сервисов, такая как аргументы контроллера, вызовы методов после инициализации, можно найти в этом файле, и это пожалуй замечательное место для отладки определений ваших сервисов, если вдруг с ними что-то идёт не так:

```
class appDevDebugProjectContainer extends Container
{
    // ...

    protected function getEventDispatcherService()
    {
        $this->services['event_dispatcher'] =
            $instance = new ContainerAwareEventDispatcher($this);

        $instance->addListenerService('kernel.controller', ...);

        // ...

        return $instance;
    }

    //...
}
```

1.2 От Kernel до HttpKernel

Теперь, когда ядро загружено (т.е. все бандлы инициализированы, их расширения зарегистрированы и сервисный контейнер инициализирован), раельная обработка запроса ложится на плечи экземпляра класса HttpKernel:

```php
// Symfony\Component\HttpKernel\Kernel

public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    if (false === $this->booted) {
    $this->boot();
}

    return $this->getHttpKernel()->handle($request, $type, $catch);
}
```

Класс HttpKernel реализует интерфейс HttpKernelInterface и точно знает как конвертировать запрос в ответ. Метод handle() выглядит так:

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

        return $this->handleException($e, $request, $type);
    }
}
```

Как вы можете видеть, основная часть работы выполняется приватным методом handleRaw(), и блок try/catch здесь нужен для перехвата любых исключений. Когда аргумент $catch имеет значение true (что является значением по умолчанию для “master”-запросов), каждое исключение будет перезвачено. HttpKernel постарается найти кого-то, кто сможет создать объект Response для (см. также главу “Обработка исключений”).

2 События, приводящие к ответу

Метод `handleRaw()` класса `HttpKernel` это замечательный пример кода, анализируя который становится ясно, что алгоритм обработки запроса сам по себе не является дерерминированным (@dbykadorov: т.е. допускает отклонения и изменения в процессе). Это означает, что у вас есть несколько различных путей для внедрения в этот процесс, путём которого вы можете либо полностью заменить или модифицировать ответ на промежуточных шагах его формирования.

2.1 Ранний ответ

Как вы думаете, когда вы можете взять контроль над обработкой запроса? Ответ - сразу же после начала его обработки. Как правило, `HttpKernel` пытается сгенерировать ответ, выполняя контроллер. Однако, любой слушатель (listener), который ожидает событие `KernelEvents::REQUEST (kernel.request)` может сгенерировать полностью уникальный ответ:

```
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

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
        );
    }

    // ...
}
```

Как вы можете видеть, объект события тут - это экземпляр `GetResponseEvent` и он позволяет слушателям заменить объект `Response` на свой, используя метод события `setResponse()`, например:

```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class MaintenanceModeListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $response = new Response(
            'This site is temporarily unavailable',
            503
        );

        $event->setResponse($response);
    }
}
```

Регистрация слушетелей событий (event listeners)

Диспетчер событий, используемый классом `HttpKernel` также доступен как сервис `event_dispatcher`. Когда вы захотите автоматически зарегистрировать какой-нибудь класс, как слушатель, Вам нужно будет создать сервис для него и добавить ему тэг `kernel.event_listener` или `kernel.event_subscriber` (в случае, если вы хотите реализовать интерфейс `EventSubscriberInterface`).

```
<service id="..." class="...">
<tag name="kernel.event_listener"
     event="kernel.request"
     method="onKernelRequest" />
</service>
```

Или:

```
<service id="..." class="...">
<tag name="kernel.event_subscriber" />
</service>
```

Вы также можете указать приоритет вашего слушателя, что может дать ему преимущество перед другими слушателями:

```
<service id="..." class="...">
<tag name="kernel.event_listener"
     event="kernel.request"
     method="onKernelRequest"
     priority="100" />
</service>
```

Чем выше приоритет, тем тем раньше слушатель события будет уведомлен.

Слушатели kernel.request, о которых вам нужно знать

Фреймворк содержит много слушателей события `kernel.request`. В основном, это слушатели, делающие некоторые приготовления, прежде чем дать ядру возможность вызвать какой-либо контроллер. Например, один слушатель даёт возможность приложению использовать локали (например, локаль по умолчанию или же _locale из URI), другой обрабатывает запросы фрагментов страниц.

Тем не менее, имеется два основных игрока, на стадии ранней обработке запроса: `RouterListener` и `Firewall`. Слушатель `RouterListener` получает инвормаю о запрошенном пути из запроса `Request` и пытается сопоставить его с одним из известных маршрутов. Он хранит результат процесса сопоставления в объекте запроса в качестве атрибута, например, в виде имени контроллера, который соответствует найденному маршруту:

```php
namespace Symfony\Component\HttpKernel\EventListener;

class RouterListener implements EventSubscriberInterface
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $request = $event->getRequest();

        $parameters = $this->matcher->match($request->getPathInfo());

        // ...

        $request->attributes->add($parameters);
    }
}
```

Например, когда сопоставителя запросов (matcher) просят найти контроллер для пути `/demo/hello/World`, а конфигурация маршрутов выглядит таким образом:

```yaml
_demo_hello:
    path: /demo/hello/{name}
    defaults:
        _controller: AcmeDemoBundle:Demo:hello
```

то параметры, возвращаемые методом match() будут являться комбинацией значений, определённых в секции `defaults:`, а также значений переменных (типа `{name}`), которые будут заменены на их значения из запроса:

```php
array(
    '_route' => '_demo_hello',
    '_controller' => 'AcmeDemoBundle:Demo:hello',
    'name' => 'World'
);
```

Эти данные сохраняются в объекте `Request`, в структуре типа `parameter bag`, имеющей наименование `attributes`. Несложно догадаться, что в дальнейшем, `HttpKernel` проверит эти атрибуты и выполнит запрошенный контроллер.

Другой, не менее важный слушатель - это `Firewall`. Как уже было отмечено ранее, `RouterListener` не предоставляет объект `Response` ядру HttpKernel, он лишь выполняет некоторые действия в начале процесса обработки запроса. `Firewall` же, напротив, иногда даже принудительно возвращает некоторые предопределённые экземпляры ответов, например, когда пользователь не аутентифицирован, когда должен был бы, так как запрашивает защищенную страницу. `Firewall` (посредством сложного процесса) форсирует редирект на страницу логина (например), или устанавливает некоторые заголовки, которые обязуют пользователя ввести его логин и пароль и аутентифицироваться при помощи HTTP-аутентивикации.

2.2 Определение контроллера для запуска

Выше мы уже видели, что `RouterListener` устанавливает атрибут запроса, именуемый `_controller` и содержащий некоторую ссылку на контроллер, который необходимо выполнить. Эта информация не известна `HttpKernel`. Вместо этого имеется специальный объект - `ControllerResolver`, который ядро запрашивает, чтобы получить контроллер для обработки текущего запроса:

```
private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // событие "kernel.request"
    ...

    if (false === $controller = $this->resolver->getController($request)) {
        throw new NotFoundHttpException();
    }
}
```

Резолвер является экземпляром класса, реализующего интерфейс `ControllerResolverInterface`:

```
namespace Symfony\Component\HttpKernel\Controller;

use Symfony\Component\HttpFoundation\Request;

interface ControllerResolverInterface
{
    public function getController(Request $request);

    public function getArguments(Request $request, $controller);
}
```

Позднее он будет использоваться для определения аргументов для контроллера, но его первичной задачей является определение контроллера. Стандартный резолвер получает контроллер из атрибута `_controller` обрабатываемого запроса:

```
public function getController(Request $request)
{
    if (!$controller = $request->attributes->get('_controller')) {
        return false;
    }

    ...

    $callable = $this->createController($controller);

    ...

    return $callable;
}
```

Так как в большинстве случаев контроллер будет представлен в виде строки, указывающей так или иначе на класс, объект контроллера необходимо создать, перед тем как вернуть его.

Всё что может быть контроллером

`ControllerResolver` из компонента HttpKernel Component поддерживает: - Массив вызываемых объектов (callable) ([объект, метод] или [класс, статический метод]) - Вызываемые (invokable) объекты (объекты с магическим методом `__invoke()`, такие как анонимные функции, которые являются экземплярами класса `\Closure`) - Классы вызываемых объектов - Обычные функции

Все прочие определения контроллеров, которые преставлены в виде строки, должны следовать шаблону `class::method`. Также ControllerResolver из FrameworkBundle добавляет дополнительные шаблоны для имён контроллеров:

- BundleName:ControllerName:actionName
- service_id:methodName

После создания экземпляра контроллера, `ControllerResolver` также проверяет, реализует ли данный контроллер интерфейс `ContainerAwareInterface`, и если да, то вызывает его метод `setContainer()`, чтобы передать ему контейнер. Вот почему контейнер по умолчанию доступен в стандартном контроллере.

2.3 Возможность замены контроллера

Давайте же вернёмся в `HttpKernel`: контроллер теперь полностью доступен и почти готов к выполнению. Но даже если мы предположим, что controller resolver выполнил всё что в его силах, для того чтобы подготовить контроллер к вызову перед его выполнением, сейчас имеется последний шанс заменить его каким-либо другим контроллером (которым может быть любой callable элемент). Этот шанс нам предоставляет событие `KernelEvents::CONTROLLER` (`kernel.controller`):

```
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // событие "kernel.request"
    // определяем контроллер при помощи controller resolver
    ...

    $event = new FilterControllerEvent($this, $controller, $request, $type);
    $this->dispatcher->dispatch(KernelEvents::CONTROLLER, $event);
    $controller = $event->getController();
}
```

Вызов метода `setController()` объекта класса `FilterControllerEvent` делает возможным замену контролера, который был подготовлен к исполнению:

```
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

class ControllerListener
{
    public function onKernelController(FilterControllerEvent $event)
    {
        $event->setController(...);
    }
}
```

Распространение событий (event propagation)

Когда вы переопределяете промежуточный результат, например, когда вы полностью заменяете контроллер после того как наступило событие `kernel.filter_controller`, вы можете захотеть, чтобы прочие слушатели этого события, которые будут вызваны после вашего - смогли бы провернуть тот же трюк. Вы можете это сделать, вызвав метод события:

```
$event->stopPropagation();
```

Также удостоверьтесь, что ваш слушатель имеет более высокий приоритет и будет вызван первым. См. также `Регистрация слушетелей событий`.

Некоторые примечательные слушатели события kernel.controller

Фреймворк сам по себе не имеет слушателей события `kernel.controller`. То есть только сторонние бандлы, которые слушают это событие для того чтобы определить тот факт, что контроллер был определён и что он будет выполнен.

Слушатель `ControllerListener` из бандла `SensioFrameworkExtraBundle`, к примеру, выполняет кое-какую весьма важную работу прямо перед выполнением контроллера, а ииименно: он собирает аннотации типа `@Template` и `@Cache` и сохраняет их в виде атрибутов запроса с тем же именем, но с префиксом - подчёркиванием: `_template` и `_cache`. Позднее, в процессе обработки запроса, эти аннотации (или конфигурации, как они названы в коде этого бандла) будут использованы для рендеринга шаблона или же для того, чтобы установить заголовки, относящиеся к кэшированию.

`ParamConverterListener` из того же банла умеет конвертировать аргументы контролера, например загружать сущность (entity) по парметру `id`, определённому в маршруте:

```
/**
 * @Route("/post/{id}")
 */
public function showAction(Post $post)
{
    ...
}
```

Преобразователи параметров (Param converters)

Бандл `SensioFrameworkExtraBundle` укомплектован конвертером `DoctrineParamConverter`, который помогает конвертировать пары имя/значение (например id), в сущности (ORM) или документы (ODM). Но вы также можете создать свои преобразователи параметров. Вам всего лишь нужно создать класс, реализующий интерфейс `ParamConverterInterface`, создать определние сервиса для него и присвоить ему таг `request.param_converter`. См. также документацю к @ParamConverter.

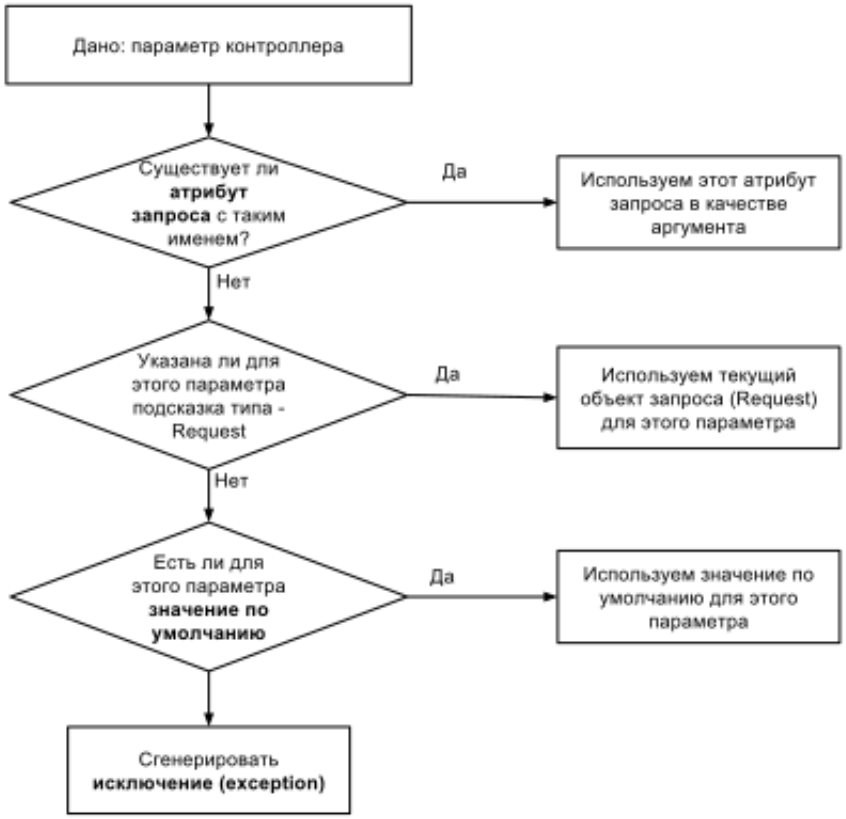
2.4 Сбор аргументов для выполнения контроллера

После того как отработали листенеры, которые могли бы заменить контроллер, мы можем быть уверены, что результирующий контроллер - тот что нам нужен. Следующий шаг - сбор аргументов, которые будут использоваться для выполнения результирующего контроллера:

```php
private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // событие "kernel.request"
    // определяем контроллер при помощи controller resolver
    // событие "kernel.controller"
    ...

    $arguments = $this->resolver->getArguments($request, $controller);
}
```

Экземпляр controller resolver запрашивается для получения аргументов контроллера. Стандартный `ControllerResolver` компонента `HttpKernel` использует рефлексию (reflection) и атрибуты из объекта Request, для того чтобы определить аргументы контроллера. Он перебирает все параметры метода контроллера. Для определения каждого из аргументов используется такая логика:



Логика controller resolver'а

2.5 Выполнение контроллера

Наконец, пришло время выполнить контроллер. Получаем ответ и двигаемся дальше:

```
private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // событие "kernel.request"
    // определяем контроллер при помощи controller resolver
    // событие "kernel.controller"
    // используем controller resolver для того, чтобы получить аргументы контроллера
    ...

    $response = call_user_func_array($controller, $arguments);

    if (!$response instanceof Response) {
        ...
    }
}
```

Как вы можете помнить из документации Symfony, контроллер должен возвращать объект `Response`. Если же контроллер этого не сделал, какая-то другая часть приложения должна иметь возможность конвертировать возвращаемое значение в объект `Response` тем или иным образом.

2.6 Вход в слой представления (view)

Если вы решили вернуть из вашего контроллера объект `Response`, вы можете таким образом срезать угол и обойти шаблонизатор, например вернув уже готовую HTML разметку:

```
class SomeController
{
    public function simpleAction()
    {
        return new Response(
            '<html><body><p>Pure old-fashioned HTML</p></body></html>'
        );
    }
}
```

Тем не менее, когда вы вернёте что-нибудь другое (как правило - массив с переменными рендеринга для шаблона), возвращаемое значение нужно конвертировать в объект `Response`, прежде чем он будет использован в качестве результата, которые будет отправлен на клиент (в браузер пользователя). Ядро `HttpKernel` не привязано ни к какому конкретному шаблонизатору, типа `Twig`. Вместо этого оно использует диспетчер событий (event dispatcher), чтобы позволить любому слушателю события `KernelEvents::VIEW` (kernel.view) создать правильный ответ, основанный на значении, которое вернул контроллер (даже если он полностью проигнорирует это значение):

```php
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // событие "kernel.request"
    // определяем контроллер при помощи controller resolver
    // событие "kernel.controller"
    // используем controller resolver для того, чтобы получить аргументы контроллера
    // исполняем контроллер
    ...

    $event = new GetResponseForControllerResultEvent(
        $this,
        $request,
        $type,
        $response
    );
    $this->dispatcher->dispatch(KernelEvents::VIEW, $event);

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

    if (!$response instanceof Response) {
        // тут ядру ОООООЧЧЧЕНЬ нужен ответ...

        throw new \LogicException(...);
    }
}
```

Слушатели этого события могут использовать метод `setResponse()` объекта события `GetResponseForControllerResultEvent`:

```php
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;

class ViewListener
{
    public function onKernelView(GetResponseForControllerResultEvent $event)
    {
        $response = new Response(...);

        $event->setResponse($response);
    }
}
```

Примечательные слушатели события kernel.view

Слушатель `TemplateListener` из арсенала `SensioFrameworkExtraBundle` получает значение, которое вернул контроллер и использует его в частве переменных для рендеринга шаблона, который должен быть указан при помощи аннотации `@Template` (храниться это значение будет в атрибуте запроса `_template`):

```php
public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $parameters = $event->getControllerResult();

    // получаем движок шаблонизатора
    $templating = ...;

    $event->setResponse(
        $templating->renderResponse($template, $parameters)
    );
}
```

2.7 Фильтрация ответа

Ну и в самом конце, прямо перед тем, как вернуть объект `Response` в качестве финального результата обработки текущего объекта запроса `Request`, будет уведомлен любой слушатель события `KernelEvents::RESPONSE` (kernel.response):

```php
private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    // событие "kernel.request"
    // определяем контроллер при помощи controller resolver
    // событие "kernel.controller"
    // используем controller resolver для того, чтобы получить аргументы контроллера
    // конвертируем рещультат, который вернул запрос в объект Response

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

private function filterResponse(Response $response, Request $request, $type)
{
    $event = new FilterResponseEvent($this, $request, $type, $response);

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

    return $event->getResponse();
}
```

Слушатели события могут модифицировать объет оответа `Response` и даже полностью его заменить:

```php
class ResponseListener
{
    public function onKernelResponse(FilterResponseEvent $event)
    {
        $response = $event->getResponse();

        $response->headers->set('X-Framework', 'Symfony2');

        // or

        $event->setResponse(new Response(...));
    }
}
```

Примечательные слушатели события kernel.response

Слушатель `WebDebugToolbarListener` из комплекта инструментов бандла `WebProfilerBundle` внедряет HTML и JavaScript код в конце ответа, для того, чтобы панель профайлера отобразилась (как правило, в конце страницы).

Слушатель `ContextListener` из компонента `Symfony Security Component` сохраняет сериализованную версию токена безопасонсти в сессии. Это позволяет ускорить процесс аутентификации при следующем запросе. В компонент `Security Component` также входит слушатель `ResponseListener`, который устанавливает cookie, который содержит информацию о `remember-me`. Содержимое этого cookie может быть использовано для авто-логина пользователя, даже если оригинальная сессия уже была завершена.

3 Обработка исключений

Не исключено, что в процессе долгого путешествия от запроса до ответа, возникнет та или иная ошибка. По умолчанию, ядро проинструктировано перехватывать любое исключение и даже после этого оно пытается подобрать подходящий для него ответ `Response`. Как мы уже видели, обработка каждого запроса обёрнута в блок `try/catch`:

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

        return $this->handleException($e, $request, $type);
    }
}
```

Когда переменная `$catch` имеет значение `истина` (true), вызывается метод `handleException()` и ожидается, что он вернёт объект ответа. Этот метод отправляет событие `KernelEvents::EXCEPTION` (kernel.exception) с экземпляром события `GetResponseForExceptionEvent`.

```
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;

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()) {
        throw $e;
    }

    $response = $event->getResponse();

    ...
}
```

Слушатели события `kernel.exception` могут:

- Установить объект ответа `Response`, соответствующий пойманному исключению.
- Заменить исходный объект исключения.

Если ни один из слушателей не вызвал метод события `setResponse()`, исключение будет вызвано еще раз, но в этот раз оно не будет перехвачено автоматически. Таким образом,

если в настройка вашего интерпритатора PHP параметр `display_errors` имеет значение `истина` (true), PHP просто отобразит ошибку как есть, без изменений (если отображение ошибок у вас отключено - не будет выведено ничего). В случае же, если один из слушателей установил объект ответа `Response`, класс `HttpKernel` проверяет этот объект на предмет корректной установки http статус-кода:

```
// проверяем наличие специфического статус-кода
if ($response->headers->has('X-Status-Code')) {
    $response->setStatusCode($response->headers->get('X-Status-Code'));

    $response->headers->remove('X-Status-Code');
} elseif (
    !$response->isClientError()
    && !$response->isServerError()
    && !$response->isRedirect()
) {
    // убеждаемся, что у нас действительно есть ответ
    if ($e instanceof HttpExceptionInterface) {
        // сохраняем HTTP статус-код и заголовки
        $response->setStatusCode($e->getStatusCode());
        $response->headers->add($e->getHeaders());
    } else {
        $response->setStatusCode(500);
    }
}
```

Это очень удобно, так как мы можем форсировать любой статус-код, добавляя заголовок `X-Status-Code` к объекту ответа `Response` (важно! это будет работать лишь для исключений, которые были перехвачены в `HttpKernel`), или же создав исключение, которое реализует интерфейс `HttpExceptionInterface`. В противном случае статус-код будет иметь значение по-умолчанию - 500, что означает `Internal server error`. Это намного лучше, чем то, что предлагает нам “чистый” PHP, который в случае ошибки вернул бы ответ со статусом 200, что означает `OK`. Когда слушатель установил объект ответа, этот ответ не будет обрабатываться каким-то иным образом, нежели обычный ответ, поэтому последний шаг в этом процессе - фильтрация ответа. Если в ходе фильтрации ответа будет сгенерировано другое исключение, это исключение будет проигнорировано и клиенту будет отправлен неотфильтрованный ответ:

```
try {
    return $this->filterResponse($response, $request, $type);
} catch (\Exception $e) {
    return $response;
}
```

3.1 Примечательные слушатели события kernel.exception

Слушатель `ExceptionListener` компонента `HttpKernel` пытается самостоятельно обработать исключение, логгируя его (если доступен логгер) и выполняя контроллер, который может отобразить страницу с информацией об ошибке. Как правило, это контроллер, который указан в файле конфигурации `config.yml`:

```
twig:
    # points to Symfony\Bundle\TwigBundle\Controller\ExceptionController
    exception_controller: twig.controller.exception:showAction
```

Другой важный слушатель - это `ExceptionListener` компонента `Security`. Этот слушатель проверяет, является ли исходное исключение экземпляром `AuthenticationException` или же `AccessDeniedException`. В первом случае он начинает процесс аутентификации, если это возможно. Во втором случае он пытается перелогинить пользователя (например если тот использовал `remember me` опцию при логине), если же это не выходит, предоставляет возможность `access denied handler` разобраться с возникшей ситуацией.

4 Подзапросы

Вероятно вы знаете о том, что при вызове метода `HttpKernel::handle()` вторым параметром идёт агумент типа запроса - `$type`:

```
public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    ...
}
```

В интерфейсе `HttpKernelInterface` определены две константы, позволяющие определить тип запроса:

1. `HttpKernelInterface::MASTER_REQUEST` - главный запрос (мастер)
2. `HttpKernelInterface::SUB_REQUEST` - поздапрос

Для каждого запроса в вашем PHP-приложении, первый запрос, который обрабатывается ядром - имеет тип мастер - `HttpKernelInterface::MASTER_REQUEST`. Это определено неявно, так как аргумент `$type` не указывается во фронт-контроллерах (app.php и app_dev.php):

```
$response = $kernel->handle($request);
```

Многие слушатели ожидают события ядра, как это было описано ранее, но включаются в работу лишь тогда, когда запрос имеет тип `HttpKernelInterface::MASTER_REQUEST`. Например, `Firewall` не будит ничего делать в случае, если запрос имеет тип подзапроса:

```
public function onKernelRequest(GetResponseEvent $event)
{
    if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
        return;
    }

    ...
}
```

4.1 Когда используются подзапросы?

Подзапросы используются для изоляции создания объекта ответа. Например, когда исключение перехвачено ядром, стандартный обработчик исключений пытается выполнить назначенный для этого исключения контроллер (см. предыдущую часть, Обработка исключений). Для этого создаётся подзапрос:

```php
public function onKernelException(GetResponseForExceptionEvent $event)
{
    $request = $event->getRequest();

    ...

    $request = $request->duplicate(null, null, $attributes);
    $request->setMethod('GET');

    $response = $event
        ->getKernel()
        ->handle($request, HttpKernelInterface::SUB_REQUEST, true);

    $event->setResponse($response);
}
```

Также, каждый раз, когда вы рендерите контроллер через Twig-шаблон, создаётся и обрабатывается подзапрос:

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

Когда вы пишите собственный слушатель событий ядра...

Спросите себя, должен ли ваш слушатель реагировать на мастер-запрос, на подзапрос, или же на оба типа. И используйте условие для проверки, которое приведено выше.

II Приёмы внедрения зависимостей

5 Что такое бандл (bundle)

Как мы уже могли отметить в предыдущей главе, запуск Symfony-приложения означает загрузку ядра и обработку запроса или выполнение команд. В свою очередь, загрузка ядра означает: загрузку всех бандлов и регистрацию их расширений сервисного контейнера (которые в любом банде расположены в директории `DependencyInjection`).

Обычно, расширение контейнера загружает файл services.xml (однако, формат может быть и другим) и конфигурацию бандла, которая представлена двумя классами, как правило в одном и том же пространстве имён - `Configuration`. Всё это вместе (бандл, расширение контейнера и конфигурация) может быть использовано для того, чтобы связать бандл с прочими частями приложения: вы можете определять параметры и сервисы, чтобы функции из вашего бандла были бы доступны также и в других частях приложения. Вы можете двинуться ещё дальше и регистрировать дополнительные “этапы компилятора” (`compiler passes`) для того, чтобы модифицировать сервисный контейнер, до того как он примет окончательный вид.

После создания множества бандлов, я понял, что бОльшая часть моей работы, как разработчика Symfony-приложений состоит в написании кода в основном для трёх вещей: бандлов, расширений и классов конфигурации (а также их `compiler passes`). Если вы уже знаете, как писать хороший код, вам всё ещё нужно узнать, как создавать хорошие бандлы, и это в основном означает, что вам нужно знать как создавать хорошие определения ваших сервисов. Имеется много способов определния сервисов и далее в этой главе я расскажу о большинстве из них. Зная все возможности, вы можете сделать правильный выбор, применительно к конкретно взятой зитуации.

Не используйте команды генерации

Когда вы начинаете использовать Symfony, вероятно вы захотите использовать команды, которые предоставляет `SensioGeneratorBundle` для создания бандлов, контроллеров, сущностей и форм. Я не рекомендую пользоваться ими. Классы, которые генерируют эти команды хорошо иметь перед глазами, как образец для создания ваших классов, но не автоматически, а вручную, так как они содержат слишком много ненужного вам кода, или же кода, не нужного вам на начальных этапах. Так что воспользуйтесь этими командами по одному разу, посмотрите, как нужно действовать, а затем поймите сами, как добиться похожих целей без использования команд генерации. Понимание этих вещей быстро сделает вас разработчиком, который хорошо понимает выбранный фреймворк.

6 Приёмы создания сервисов

Сервис - это объект, зарегистрированный в сервисном контейнере с некоторым идентификатором (id), который может быть создан в любой момент через посредством этого контейнера.

6.1 Обязательные зависимости

Многие объекты для выполнения своих функций нуждаются в других объектах, скалярных значениях (например ключ API) и может быть даже в массивах значений (скаляров или объектов). Эти объекты, скаляры и массивы называются зависимостями. Сначала мы рассмотрим как вы можете определить обязательные зависимости для ваших сервисов.

Обязательные параметры конструктора

Самый простой способ передать сервису его зависимости - указать их в качестве аргументов конструктора:

```
class TokenProvider
{
    private $storage;

    public function __construct(TokenStorageInterface $storage)
    {
        $this->storage = $storage;
    }
}
```

Определение сервиса для класа `TokenProvider` должно выглядеть таким образом:

```
<service id="token_provider" class="TokenProvider">
    <argument type="service" id="token_storage" />
</service>

<service id="token_storage" class="...">
</service>
```

Первый аргумент сервиса `token_provider` это ссылка на сервис `token_storage`. Класс хранилища токнов, таким образом, должен реализовывать интерфейс `TokenStorageInterface`, иначе он не будет корректным аргументом и вы получите фатальную ошибку.

Абстрактные определения для дополнительных аргументов

Предположим у вас есть другой провайдер токенов, `ExpiringTokenProvider`, который наследуется от `TokenProvider`, но имеет дополнительно свой аргумент конструктора - `$lifetime`:

```php
class ExpiringTokenProvider extends TokenProvider
{
    private $lifetime;

    public function __construct(TokenStorageInterface $storage, $lifetime)
    {
        $this->lifetime = $lifetime;

        parent::__construct($storage);
    }
}
```

Создавая определение сервиса для второго провайдера, вы можете просто скопировать аргумент из определения сервиса `token_provider`:

```xml
<service id="expiring_token_provider" class="ExpiringTokenProvider">
    <argument type="service" id="token_storage" />
    <argument>3600</argument><!-- lifetime -->
</service>
```

Однако, лучшим выходом из данной ситуации будет создание определения родительского сервиса, которое вы сможете использовать для определения дочерних сервисов, предоставляя им необходимые базовые аргументы:

```xml
<service id="abstract_token_provider" abstract="true">
    <argument type="service" id="token_storage" />
</service>

<service id="token_provider" class="TokenProvider"
    parent="abstract_token_provider">
</service>

<service id="expiring_token_provider" class="ExpiringTokenProvider"
    parent="abstract_token_provider">
    <argument>3600</argument><!-- lifetime -->
</service>
```

Абстрактный сервис из примера выше имеет один аргумент и отмечен как абстрактный. Сервисы провайдеров используют `abstract_token_provider` в качестве родителя. Сервис `token_provider` не имеет дополнительных аргументов, таким образом он лишь наследует первый аргумент конструктора от `abstract_token_provider`. Сервис `expiring_token_provider` также наследует первый аргумент `token_storage`, но также добавляет свой дополнительный аргумент `$lifetime`.

Наследование свойств

Вне зависимости от того, является ли родительский сервис абстрактным или нет, дочерний сервис наследует нижеперечисленные свойства родительского сервиса:

- Класс
- Аргументы конструктора (в порядке их появления)

- Вызовы методов после создания экземпляра класса (@dbykadorov: судя по всему имеются в виду call вызовы при использовании setter injection)
- Свойства, используемые для `property injection` (недостаки этого способа внедрения параметров описаны тут)
- Фабричный класс или сервис, фабричный метод
- Конфигуратор (штука весьма экзотичная, не рассматривается в данной книге)
- Файл (необходимый для создания сервиса)
- Признак, является ли создаваемый сервис публичным

Вызов обязательных set-методов (setters)

Иногда вы можете попасть в ситуациию, когда вы не захотите (или не сможете) переопределить конструктор сервиса и, следовательно, не сможете добавить дополнительные параметры в него, или же, возможно, некоторые зависимости еще не определены в момент создания класса сервиса. В таких случаях вы можете добавить в ваш класс set-метод, что позволит внедрить зависимость сразу после создания сервиса (или, фактически, в любой момент после создания):

```
class SomeController
{
    private $container;

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

    public function indexAction()
    {
        $service = $this->container->get('...');
    }
}
```

Так как контроллеру из примера выше для выполнения метода `indexAction()` необходим экземпляр `ContainerInterface`, вызов `setContainer` является обязательным. Поэтому в определении сервиса для него вы должны позаботиться о внедрении сервисного котейнера:

```
<service id="some_controller" class="SomeController">
    <call method="setContainer">
        <argument type="service" id="service_container" />
    </call>
</service>
```

Преимущества при использовании set-методов для внедрения зависимостей в том, что вам не нужно указывать все зависимости в виде аргументов конструктора. Иногда это означает, что вам не конструктор и вовсе не понадобится, или же вы можете оставить конструктор в неизменном виде. Недостатком данного метода является возможность ситуации, когда вы забудете вызвать set-метод, что приведёт к отсутствию необходимых зависимостей. В

примере выше это привеёт к вызову метода `get()` от `null`, что, в свою очередь, вызовет фатальную ошибку PHP. По моему мненю, этот недостаток перекрывает все достоинства данного подхода, так как код получается “с душком”, который называется `временнАя связность` (@dbykadorov: от слова “время”, имеется в виду, что работоспособность вашего кода зависит от того был ли перед использованием сервиса вызван нужный set-метод или вы забыли добавить его описание сервиса) или `temporal coupling`. Таким образом этот тип внедрения делает ваш класс менее надёжным (@dbykadorov: о `temporal coupling` в PHP можно дополнительно почитать, например, тут).

Для того, чтобы предотвратить серьёзные сбои (и чтобы помочь разработчику, который будет исправлять проблему, если она всё-таки возникнет) вы можете ограничить доступ к зависимому свойству через его геттер (метод `getXxx()`) и в нём проверять валидность соответствующего значения, например вот так:

```
class SomeController
{
    public function indexAction()
    {
        $service = $this->getContainer()->get('...');
    }

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

        return $this->container;
    }
}
```

ContainerAware

Компонент Symfony DependencyInjection содержит интерфейс `ContainerAwareInterface` и абстрактный класс `ContainerAware`, которые вы можете использовать для того, чтобы указать, что класс “осведомлён” (“aware” в оригинале) о сервисном контейнере. Использование этих классов предоставит вам set-метод по имени `setContainer()`, с помощью которого вы сможете передать сервисный контейнер в ваш класс. Контроллеры, которые реализуют интерфейс `ContainerAwareInterface`, автоматически получают доступ к контейнеру при помощи этого set-метода. Стандартный класс `Controller` из состава Symfony FrameworkBundle является “осведомлённым-о-контейнере” - `container-aware`. См. также главу 1, раздел 2.2 - `Определение контроллера для запуска`.

Вызов методов в абстрактных сервисах

Когда вы создаёте `container-aware` сервис, у вас будет много дублирующего кода в его определении. Разумным будет добавить вызов метода `setContainer()` в определение абстрактного сервиса:

```
<service id="abstract_container_aware" abstract="true">
    <call method="setContainer">
        <argument type="service" id="service_container" />
    </call>
</service>

<service id="some_controller" class="SomeController"
    parent="abstract_container_aware">
</service>
```

Соглашение об именовании родительских сервисов

Определения родительских сервисов не обязательно должны быть абстрактными. Однако, когда вы опускаете атрибут `abstract="true"`, определение родительского сервиса будет трактоваться как определение обычного сервиса (и соответствующим образом валидироваться).

Если же вы хотите создать определение абстрактного сервиса - пометьте его таковым, присвоив трибуту `abstract` значение `true` и добавьте префикс `abstract_` к его id (по аналогии с абстрактныи классами, имена которых традиционно начинаются с `Abstract`).

Если же у вас есть определение родительского сервиса, который также должен сам быть самостоятельным сервисом, добавлять атрибут `abstract` не нужно, но, возможно, будет полезным добавить префикс `base_` к его id.

6.2 Необязательные (опциональные) зависимости

Иногда зависимости являются не обязатальными. Термин “не обязательные зависимости” может вам показаться противоречивым, так как если вы реально от чего-то не зависите, странно называть это “зависиомстями”. Однако, встречаются ситуации, когда один сервис знает как использовать другой сервис, но в общем-то этот другой сервис не необходим для выполнения его функций. Например, сервис может знать как использовать сервис журналирования (логгер) для того, чтобы писать в лог какие-либо отладочные данные.

Необязательные аргументы конструктора

В случае, когда класс вашего сервиса знает как работать с логгером, он может иметь необязательный аргумент конструктора для него:

```
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
use Psr\Log\LoggerInterface;

class AuthenticationListener
{
    private $eventDispatcher;
    private $logger;

    public function __construct(
        EventDispatcherInterface $eventDispatcher,
        LoggerInterface $logger = null
    ) {
        $this->eventDispatcher = $eventDispatcher;
```

```
14          $this->logger = $logger;
15      }
16  }
```

Для аргументов конструктора, которые должны быть либо экземплярами указанных классов, либо могут отсутствовать, вы можете использовать значение по умолчанию `null`. В этом случае, в определении вашего сервиса в можете выбрать стратегию поведения в случае их отсутствия:

```
<service id="authentication_listener" class="AuthenticationListener">
    <argument type="service" id="logger" on-invalid="ignore" />
</service>
```

Стратегия `ignore` на данный момент эквивалентна null-стратегии, в том смысле, что конструктор будет вызван со значением null, вместо запрошенного сервиса, если тот недоступен. Есть еще стратегия `exception`, которая применяется по умолчанию. При её использовании будет вызвано исключение, если внедряемый сервис не будет найден.

> **Проверка необязательных зависимостей**
>
> Если вы хотите проверить, внедрена или нет необязательная зависимость, вы должны написать примерно такой код:
>
> ```
> if ($this->logger instanceof LoggerInterface) {
>     ...
> }
> ```
>
> Это более надёжный способ проверки, чем сравнение с NULL:
>
> ```
> if ($this->logger !== null) {
>     ...
> }
> ```
>
> Думайте об этом в таком ключе: если что-то не является NULL, можем ли мы быть уверены, что это logger?

Не обязательные вызовы set-методов

Также, как и в случае с обязательными зависимостями, иногда бывает удобно/необходимо внедрить необязательные зависимости при помощи set-методов. Как правило, эта потребность возникает, если вы не хотите захлямлять сигнатуру конструктора:

```php
class AuthenticationListener
{
    private $eventDispatcher;
    private $logger;

    public function __construct(EventDispatcherInterface $eventDispatcher)
    {
        $this->eventDispatcher = $eventDispatcher;
    }

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

В определении сервиса вы можете добавить вызов метода `setLogger()` с сервисом журналирования в качестве аргумента. Вы также можете указать, что этот аргумент должен быть проигнорирован, в случае, если соответствующий сервис не будет найден (что делает эту зависимость по-настоящему необязательной):

```xml
<service id="authentication_listener" class="AuthenticationListener">
    <call method="setLogger">
        <argument type="service" id="logger" on-invalid="ignore" />
    </call>
</service>
```

Аргумент вызова метода `setLogger()` может иметь значение NULL, (когда сервис не определён), но метод будет вызван в любом случае, таким образом вы должны иметь в виду, что NULL является одним из валидных аргументов вызова метода `setLogger()`.

> **Определение закрытых (non-public) зависимостей**
>
> Когда вы пишете код в тру-ООП стиле, у вас всегда всегда будет куча небольших сервисов, каждый из которых выполняет только одну фиксированную функцию. Сервисы более высокого уровня будут внедрять их в качестве зависимостей. Сервисы более низких уровней как правило не должны пересекаться между собой; они лишь выполняют роль помошников для сервисов более высокого уровня. Чтобы не допустить возможности использования низкоуровневых сервисов в других частях приложения, например так:
>
> ```php
> $container->get('low_level_service_id');
> ```
>
> вы должны пометить их как закрытые (non-public), для этого в опредении низкоуровневого сервиса нужно присвоить атрибуту public значение “false”:
>
> ```xml
> <service id="low_level_service_id" class="..." public="false">
> </service>
> ```

6.3 Коллекции сервисов

В большинстве случаев вы будете внедрять зависимости через конструктор или аргументы set-методов. Но иногда возникает необходимость внедрить в качестве зависиомсти целую коллекцию сервисов, например, если вы хотите предоставить несколько альтернатив (стратегий) для достижения некоторых целей:

```php
class ObjectRenderer
{
    private $renderers;

    public function __construct(array $renderers)
    {
        $this->renderers = $renderers;
    }

    public function render($object)
    {
        foreach ($this->renderers as $renderer) {
            if ($renderer->supports($object) {
                return $renderer->render($object);
            }
        }
    }
}
```

Определение такого сервиса может выглядеть следующим образом:

```xml
<service id="object_renderer" class="ObjectRenderer">
    <argument type="collection">
        <argument type="service" id="domain_object_renderer" />
        <argument type="service" id="user_renderer" />
    </argument>
</service>
```

Аргумент типа `collection` будет преобразован в массив, содержащий сервисы, id которых перечислены в этой коллекции:

```php
array(
    0 => ...
    1 => ...
)
```

Вы также можете для каждого элемента коллекции указать ключ при помощи атрибута `key`:

```xml
<service id="object_renderer" class="ObjectRenderer">
    <argument type="collection">
        <argument
            key="domain_object" type="service"
            id="domain_object_renderer" />
        <argument
            key="user" type="service"
            id="user_renderer" />
    </argument>
</service>
```

Значение атрибута `key` будет использовано в качестве ключа для соответствующих значений коллекции:

```
array(
    'domain_object' => ...
    'user' => ...
)
```

Вызов нескольких методов

Если вы включите “строгий” режим и перечитаете код класса `ObjectRenderer`, вы вероятно заметите, что нельзя доверять массиву `$renderers` в том, что он содержит только валидные рендереры (которые, к примеру, должны реализовывать интерфейс `RendererInterface`). Следовательно, вы вероятно захотите выделить отдельный метод для добавления рендерера:

```
class ObjectRenderer
{
    private $renderers;

    public function __construct()
    {
        $this->renderers = array();
    }

    public function addRenderer($name, RendererInterface $renderer)
    {
        $this->renderers[$name] = $renderer;
    }
}
```

Конечно же, если имя рендерера не имеет значения, можно убрать параметр `$name`. Важно тут другое: когда кто-либо вызовет метод `addRenderer` и передаст ему в качестве аргумента объект, который не является реализацией интерфейса `RendererInterface`, этот вызов не будет успешным, так как не будет соблюдено ограничение по типу аргумента. Определение сервиса также необходимо изменить, чтобы для каждого рендерера вызывался бы метод `addRenderer()`:

```
<service id="object_renderer" class="ObjectRenderer">
    <call method="addRenderer">
        <argument>domain_object</argument>
        <argument type="service" id="domain_object_renderer" />
    </call>
    <call method="addRenderer">
        <argument>user</argument>
        <argument type="service" id="user_renderer" />
    </call>
</service>
```

Лучшее из двух миров

Возможно вас также заинтересует идея о том, как объединить походы для работы с коллекциями сервисов, описанные выше, что позволило бы разработчикам передавать как набор рендереров по-умолчанию через аргумент конструктора и/или добавлять рендереры один за другим, используя метод `addRenderer()`:

```php
class ObjectRenderer
{
    private $renderers;

    public function __construct(array $renderers)
    {
        foreach ($renderers as $name => $renderer) {
            $this->addRenderer($name, $renderer);
        }
    }

    public function addRenderer($name, RendererInterface $renderer)
    {
        $this->renderers[$name] = $renderer;
    }
}
```

Метки сервисов (tags)

Мы уже умеем добавлять рендереры вручную, но что, если другие части вашего приложения (например другие бандлы) должны иметь возможность регистрировать свои рендереры? Наилучшим способом достичь этого будет использование меток (тагов) для сервисов:

```xml
<!-- in some other bundle -->
<service id="date_time_renderer" class="DateTimeRenderer">
    <tag name="specific_renderer" alias="date_time" />
</service>
```

У каждого тага есть имя, которое вы можете выбрать самостоятельно. Каждый таг также может иметь дополнительные атрибуты (например `alias` в примере выше). Эти атрибуты в дальнейшем позволят вам определять дальнейшее повение сервиса.

Для того, чтобы получить список всех сервисов с нужным тагом, вам нужно создать т.н. `compiler pass`, например такой:

```php
namespace Matthias\RendererBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;

class RenderersPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        // получаем все сервисы, отмеченные определённым тагом из всего проекта
        $taggedServiceIds
            = $container ->findTaggedServiceIds('specific_renderer');

        $objectRendererDefinition
            = $container->getDefinition('object_renderer');

        foreach ($taggedServiceIds as $serviceId => $tags) {

            // сервисы могут иметь более одного тага с одни и тем же именем
            foreach ($tags as $tagAttributes) {

                // вызываем метод addRenderer() для того, чтобы зарегистрировать рендерер
```

```
                $objectRendererDefinition
                    ->addMethodCall('addRenderer', array(
                        $tagAttributes['alias'],
                        new Reference($serviceId),
                    ));
            }
        }
    }
}
```

Созданный выше класс `compiler pass` нужно зарегистрировать в классе вашего бандла:

```
use Matthias\RendererBundle\DependencyInjection\Compiler\RenderersPass;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class RendererBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        $container->addCompilerPass(new RenderersPass());
    }
}
```

Метод `process()` класса `RenderersPass` в первую очередь получает все сервисы с тегами, которые имеют имя `specific_renderer`. В результате будет получен массив, ключами котророго будут id сервисов, а значениями - массив их атрибутов. Так сделано потому, что определние любого сервиса может иметь более одного тега с одним и тем же именем (но, возможно, с другими атрибутами).

Потом, запрашивается определение сервиса `object_renderer` для класса `ObjectRenderer`, после чего выполняется цикл по всем найденным тагам. В каждой итерации создаётся экземпляр класса `Reference`, который ссылается на текущий (в данной итерации) сервис рендерера (который, в свою очередь, помечен тагом `specific_renderer`) и вместе с значением атрибута `alias` они используются в качестве аргументов для вызова метода `addRenderer()`.

Всё это вместе означает что когда сервис `object_renderer` будет запрошен, сначала будет создан экземпляр класса `ObjectRenderer`. Но затем будет выполнено несколько вызовов метода `addRenderer()`, которые добавят рендереры, отмеченные тагом `specific_renderer`.

Вызов одного метода

Есть много возможностей обработки сервисов в `compiler pass`. Например, вы можете собрать ссылки (references) на сервисы в массив и обработать их все разом, указав их в качестве аргумента метода `setRenderers()`:

```
class RenderersPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        $taggedServiceIds = ...;

        $objectRendererDefinition = ...;

        $renderers = array();

        foreach ($taggedServiceIds as $serviceId => $tags) {
            foreach ($tags as $tagAttributes) {
                $name = $tagAttributes['alias'];
                $renderer = new Reference($serviceId);
                $renderers[$name] = $renderer;
            }
        }

        $objectRendererDefinition
            ->addMethodCall('setRenderers', array($renderers));
    }
}
```

Замена аргумента конструктора

Если имеется возможность внедрения коллекции сервисов в виде конструктора - например рендереры из примера выше - есть также другой способ сделать это: установить аргумент конструктора напрямую:

```
class RenderersPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        $taggedServiceIds = ...;

        $objectRendererDefinition = ...;

        $renderers = array();

        // получаем референсы на сервисы
        ...

        $objectRendererDefinition->replaceArgument(0, $renderers);
    }
}
```

Замена аргумента может быть выполнена только в случае, если этот аргумент определён первым (например как пустой аргумент):

@dbykadorov: TODO - перепроверить, либо неправильно понял, либо это не всегда так

```
<service id="object_renderer" class="ObjectRenderer">
    <argument /><!-- наши рендереры -->
</service>
```

Передаём ID сервисов вместо референсов

Когда вы запрашиваете сервис `object_renderer`, все рендереры, переданные ему в качестве аргументов также будут созданы. В зависимости от цены (@dbykadorov: в плане

производительности) создания этих рендереров возможо было бы неплохо предусмотреть возможность их “ленивой” загрузки - `lazy-loading`. Этого можно добиться сделав `ObjectRenderer` “осведомлённым о контейнере” - `container-aware` и внедряя id сервисов, вместо них самих:

```
class LazyLoadingObjectRenderer
{
    private $container;
    private $renderers;

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

    public function addRenderer($name, $renderer)
    {
        $this->renderers[$name] = $renderer;
    }

    public function render($object)
    {
        foreach ($this->renderers as $name => $renderer) {
            if (is_string($renderer)) {
                // здесь $renderer - это id сервиса, строка
                $renderer = $this->container->get($renderer);
            }

            // проверяем, является ли renderer экземпляром RendererInterface
            ...
        }
    }
}
```

Также `compiler pass` нужно модифицировать таким образом, чтобы он не передавал ссылки на services, а только лишь их id:

```
class RenderersPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        $taggedServiceIds = ...;

        $objectRendererDefinition = ...;

        foreach ($taggedServiceIds as $serviceId => $tags) {
            foreach ($tags as $tagAttributes) {
                $objectRendererDefinition
                    ->addMethodCall('addRenderer', array(
                        $tagAttributes['alias'],
                        $serviceId,
                    )
                );
            }
        }
    }
}
```

Также не забудьте указать сервисный контейнер в качестве аргумента конструктора:

```
<service id="object_renderer" class="LazyLoadingObjectRenderer">
    <argument type="service" id="service_container" />
</service>
```

И конечно же любая из стратегий, описанных выше, может быть использована с этим классом, реализующим “ленивую” загрузку - вызов одного метода, вызов нескольких методов, замена аргумента.

Перед тем, как вы решите изменить свой класс для использования сервисного контейнера напрямую, пожалуйста ознакомьтесь с разедами TODO: Уменьшаем связность кода с фреймворком, особенно TODO: заметку быстродействии.

6.4 Делегирование создания

Вместо того, чтобы создавать сервисы заранее при помощи их определений с указанием класса, аргументов конструктора и вызовов методов, вы можете не указывать все эти детали, делегровав их заполнение фабричному методу (`factory method`) во время выполнения. Фабричные методы могут быть как статичными методами, так и методами объектов. В первом случае вы можете указать имя класса и имя метода в качестве атрибутов при определении сервиса:

```
<service id="some_service" class="ClassOfResultingObject"
    factory-class="Some\Factory" factory-method="create">
    <argument>...</argument>
</service>
```

Когда сервис `some_service` будет запрошен в первый раз, он будет получен как результат вызова статического метода `Some\Factory::create()` с использованием указанных аргументов. Результат будет сохранён в памяти, поэтому фабричный метод будет вызван лишь раз (@dbykadorov: один раз в рамках текущего запроса, кэшироваться на диск такой вызов не будет). В наши дни большинство фабричных методов не являются статическими, что онзначает вызов фабричного метода у экземпляра фабричного класса. Следовательно, этот экземпляр фабрики должен быть предварительно сам определён как сервис:

```
<!-- сервис, создаваемый фабрикой some_factory_service -->
<service id="some_service" class="ClassOfResultingObject"
    factory-service="some_factory_service" factory-method="create">
    <argument>...</argument>
</service>

<!-- собственно сервис самой фабрики -->
<service id="some_factory_service" class="Some\Factory">
</service>
```

Не очень полезно...

Хотя возможность делегирования создания сервисов другим сервисам выглядит великолепно, я не использовал её слишком часто. Этав возможность полезна по большей части

только для случаев, когда сервис создаётся для PHP-классов из (не очень далёкого) прошлого, логика создания экземпляров которых часто спрятана внутри статических фабричных классов(помните `Doctrine_Core::getTable()`?).

Мои возражения против фабричных классов со статическими фабричными методами основаны на том, что статический код - это глобальный код и выполнение этого кода может иметь побочные эффекты, которые нельзя изолировать (например в тестовом сценарии). Кроме того, любая зависимость такого статического фабричного метода по определнию также должна быть статической, что также очень плохо для изоляции и не даст вам возможности подменить (часть) логики создания своим кодом.

Фабрики-объекты (или фабричные сервисы) лишь немногим лучше. Тем не менее, необходимость их использования скорее всего указывает не проблемы в дизайне (архитектуре) приложения. Сервис не должен нуждаться в фабрике так как он создаётся один раз заранее определённым (и детерминированным (@dbykadorov - т.е. при одинаковых входных данных получается одинаковый же результат)) способом и с момента создания он полностью готов к повторному использованию любым другим объектом. Переменными по отношению к сервису должны быть лишь аргументы методов, которые являются частью его публичного интерфейса (см. также TODO: Состояние и Контекст).

Иногда всё-таки полезно...

Одним из частных случаев, когда использование фабричных сервиса и метода для получения сервиса оправдано - является случай с репозиториями Doctrine. Когда вам нужен один из них, вы как правило можете внедрить `entity manager` в качестве аргумента конструктора и позднее получить нужный репозиторий:

```
use Doctrine\ORM\EntityManager;

class SomeClass
{
    public function __construct(EntityManager $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function doSomething()
    {
        $repository = $this->entityManager->getRepository('User');

        ...
    }
}
```

Но, используя фабричный сервис вы можете внедрить нужный репозиторий напрямую:

```
class SomeClass
{
    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }
}
```

Вот соответствующие определения сервисов для этого случая:

```
<service id="some_service" class="SomeClass">
    <argument type="user_repository" />
</service>

<service id="user_repository" class="UserRepository"
    factory-service="entity_manager" factory-method="getRepository">
    <argument>User</argument>
</service>
```

Если взглянуть на аргументы конструктора `SomeClass` сразу становится ясно, что на ходе ожидается репозиторий `User`, что намного более читабельно, нежели предыдущий пример, где `SomeClass` ожидает `EntityManager`. Кроме того, класс сам по себе стал чище, а также стало проще создать замену для репозитория, например, когда вы будете писать модульный тест для этого класса. Вместо того, чтобы создать mock-объекты для менеджера сущности и репозитория, теперь можно создать только один - для репозитория.

6.5 Создание сервисов вручную

Обычно вы создаёте сервисы, загружая их определения из файла:

```
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;

class SomeBundleExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $locator = new FileLocator(__DIR__.'/../Resources/config');
        $loader = new XmlFileLoader($container, $locator);
        $loader->load('services.xml');
    }
}
```

Но некоторые сервисы не могут быть определены в конфигурационном файле. Они должны создаваться динамически, потому что их имена, классы, аргументы, таги и прочие атрибуты не фиксированы.

Определение

Создание сервиса вручную означает создание экземпляра класса “определения” `Definition`, и (опционально) передача ему имени класса. Определение получит идентификатор при его добавлении в `ContainerBuilder`:

```php
use Symfony\Component\DependencyInjection\Definition;

$class = ...; // присваиваем значение имени класса для определения

$definition = new Definition($class);

$container->setDefinition('the_service_id', $definition);
```

Эквивалентом этого определения был бы такой XML:

```xml
<service id="the_service_id" class="...">
</service>
```

Вы можете сделать определение закрытым (non-public), если оно существует лишь как зависимость для других сервисов:

```php
$definition->setPublic(false);
```

Аргументы

Когда для создания сервиса нужно передать в конструктор аргументы, вы можете задать их все разом:

```php
use Symfony\Component\DependencyInjection\Reference;

$definition->setArguments(array(
    new Reference('logger') // ссылка на другой сервис
    true // логическое (булево) значение,
    array(
        'table_name' => 'users'
    ) // массив
    ...
));
```

Аргументы должны быть ссылками на другие сервисы, массивами или скалярами (или их сочетаниями). Это требование - следствие того, что все определения сервисов в конечном итоге будут сохранены в простом PHP файле. Ссылка на другой сервис будет создана с использованиме объекта `Reference` с указанием ID сервиса, который должен быть внедрён.

Вы также можете добавлять аргументы по одному, в том порядке, в котором они перечислены в конструкторе:

```php
$definition->addArgument(new Reference('logger'));
$definition->addArgument(true);
...
```

И, наконец, если вы модифицируете определение уже существующего сервиса с некоторым списком аргументов, вы можете заменить их, используя численные индексы:

```
$definition->setArguments(array(null, null));

...

$definition->replaceArgument(0, new Reference('logger'));
$definition->replaceArgument(1, true);
```

Соответствующий XML выглядел бы так:

```
<service id="..." class="...">
    <argument type="service" id="logger" />
    <argument>true</argument>
</service>
```

Таги

Есть еще кое-что, что вы можете сделать, работая с объектом Definition: добавить таги (метки). Таг состоит из его имени и массива атрибутов. Определение может иметь более одного тага с одним именем:

```
$definition->addTag('kernel.event_listener', array(
    'event' => 'kernel.request'
));
$definition->addTag('kernel.event_listener', array(
    'event' => 'kernel.response'
));
```

XML определение в этом случае было бы таким:

```
<service id="..." class="...">
    <tag name="kernel.event_listener" event="kernel.request">
    <tag name="kernel.event_listener" event="kernel.response">
</service>
```

Алиасы (псевдонимы)

Перед тем, как поговорить о том, что вам делать с вашими новыми знаниями, есть еще один момент, который вам надо знать: как создавать алиасы для сервисов:

```
$container->setAlias('some_alias', 'some_service_id');
```

Теперь, когда бы вы ни запросили сервис `some_alias`, фактически вы получите сервис `some_- service_id`.

6.6 Класс Configuration

Прежде чем продолжить, нужно дать несколько пояснений качательно класса `Configuration`. Вы могли обратить на него внимание ранее, а также, возможно, даже создавали его его самостоятельно.

Большую часть времени вы будете использовать класс `Configuration` для того, чтобы определять все возможные конфигурационные опции для вашего бандла (обратите внимание, компонент Config имеет слабые связи и вы можете также использовать всё сказанное ниже в совершенно другом контексте). Имя класса и его пространство имён не имеют особого значения, пока класс реализует интерфейс `ConfigurationInterface`:

```
use Symfony\Component\Config\Definition\ConfigurationInterface;
use Symfony\Component\Config\Definition\Builder\TreeBuilder;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('name_of_bundle');

        $rootNode
            ->children()
            // определяем узлы конфигурации
            ...
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Тут у нас один публичный метод - `getConfigTreeBuilder()`. Этот метод должен возвращать экземпляр `TreeBuilder`, который вы должны использовать для того, чтобы описать все возможные настройки вашего приложения, а также правила для их проверки на корректность (правила валидации). Создание конфигурационного дерева начинается к определения корневого узла:

```
1 $rootNode = $treeBuilder->root('name_of_bundle');
```

Имя корневого узла должно быть именем бандла без суффикса “bundle”, в нижнем регистре и с разделителем в виде подчерка. Например, имя корневого узла для `MatthiasAccountBundle` будет `matthias_account`. Корневой узел - это всегда узел типа “массив”. Он может содержать любой дочерний узел, какой вы захотите:

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

Учимся составлять замечательные деревья конфигураций

Если вы хотите стать профессионалом в создании бандлов, усердно практикуйтесь в создании конфигурационных деревьев. Конфигурация вашего бандла в

> этом случае будет значительно лучше и гибче. Подробнее об узлах конфигурации вы можете узнать в документации компонента Config. Также обратите внимание на классы конфигурации сторонних бандров (@dbykadorov: например - Friends Of Symfony, FOS) и попробуйте последовать их примеру.

Как правило, вы будете использовать экземпляр класса Configuration в классе расширения вашего бандла, для того, чтобы обработать заданный набор конфигурационных массивов. Эти конфигурационные массивы собираются ядром, загружая все подходящие конфигурационные файлы (например, config_dev.yml, config.yml, parameters.yml, и т.д.).

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

Метод `processConfiguration()` класса расширения создаёт экземпляр класса `Processor` и финализирует конфигурационное дерево, загруженное из объекта Configuration. Затем он просит processor обработать (выполнить валидацию и объединение) “сырых” конфигурационных массивов:

```
final protected function processConfiguration(
    ConfigurationInterface $configuration,
    array $configs
) {
    $processor = new Processor();

    return $processor->processConfiguration($configuration, $configs);
}
```

Если ошибок валидации не выявлено, вы можете использовать конфигурационные значения любым необходимым вам способом. Вы можете определить или модифицировать параметры контейнера или изменить определения сервисов, основывающихся на конфигурационных значениях. В следующих главах мы обсудим много различных способов сделать это.

6.7 Динамическое добавление тагов

Допустим, вы хотите создать универсальный слушатель (event listener), который слушает настраиваемый список событий, например `kernel.request`, `kernel.response`, и т.д. Вот как мог бы выглядеть соответствующий класс Configuration:

```php
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('generic_listener');

        $rootNode
            ->children()
                ->arrayNode('events')
                    ->prototype('scalar')
                    ->end()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Он позволяет сконфигурировать список имён событий следующим образом:

```yaml
generic_listener:
    events: [kernel.request, kernel.response, ...]
```

Стандартный способ зарегистрировать слушатель заключается в добавлении тагов к сервису слушателя в файле services.xml:

```xml
<service id="generic_event_listener" class="...">
    <tag name="kernel.event_listener" event="..." method="onEvent" />
    <tag name="kernel.event_listener" event="..." method="onEvent" />
</service>
```

Но в нашем случае мы не знаем, какие события слушатель должен слушать, так что мы не можем указать их явно в файле конфигурации. К счастью, как мы уже знаем, мы можем добавлять таги к определению сервиса прямо на лету. Это можно провернуть в классе расширения контейнера:

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

        // загружаем services.xml
        $loader = ...;
        $loader->load('services.xml');

        $eventListener = $container
            ->getDefinition('generic_event_listener');

        foreach ($processedConfig['events'] as $eventName) {
```

```
            // добавляем таги kernel.event_listener для каждого из указанных событий
            $eventListener
                ->addTag('kernel.event_listener', array(
                    'event' => $eventName,
                    'method' => 'onEvent'
                ));
        }
    }
}
```

Есть ещё один шаг, который нужно выполнить, чтобы предотвратить “подвисание” сервиса слушателя если ни одного события для него не сконфигурировано:

```
if (empty($processedConfig['events'])) {
    $container->removeDefinition('generic_event_listener');
}
```

6.8 Используем паттерн Стратегия для загрузки сервисов

Чсто бандлы предлагают разные способы выполнить то иную функцию. Например, бандл предоставляющий функцию почтового ящика в каком-то виде, может иметь разные реализации хранилища, например, один менеджер хранения для Doctrine ORM, а другой для MongoDB. Чтобы предоставить возможность выбора конкретного менеджер хранения, давайте создадим класс конфигурации:

```
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('browser');

        $rootNode
            ->children()
                ->scalarNode('storage_manager')
                    ->validate()
                        ->ifNotInArray(array('doctrine_orm', 'mongo_db')
                        ->thenInvalid('Invalid storage manager')
                    ->end()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Затем нужно создать файлы с определениями сервисов для каждого их менеджеров хранения, один - doctrine_orm.xml:

```xml
<services>
    <service id="mailbox.doctrine_orm.storage_manager" class="...">
    </service>
</services>
```

И другой - mongo_db.xml:

```xml
<services>
    <service id="mailbox.mongo_db.storage_manager" class="...">
    </service>
</services>
```

Затем вы должны загрузить один из этих файлов при помощи вот такого кода в классе расширения:

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

        // создаём XmlLoader
        $loader = ...;

        // загружаем только сервисы для выбранного менджера хранения
        $storageManager = $processedConfig['storage_manager'];
        $loader->load($storageManager.'.xml');

        // делаем выбранный менеджер дооступным по умолчанию
        $container->setAlias(
            'mailbox.storage_manager',
            'mailbox.'.$storageManager.'.storage_manager'
        );
    }
}
```

В конце мы создаём удобный алиас, для того чтобы другие части приложения могли обращаться к сервису `mailbox.storage_manager`, не заботясь о том, какая именно схема хранения на самом деле используется. Тем не мене, этот способ не очень гибок: id каждого менеджера хранения должен соответствовать шаблону`mailbox.{storageManagerName}.storage_manager`. Будет лучше определить алиас внутри файла с определением сервисов:

```
<services>
    <service id="mailbox.doctrine_orm.storage_manager" class="...">
    </service>

    <service id="mailbox.storage_manager"
        alias="mailbox.doctrine_orm.storage_manager">
    </service>
</services>
```

Используя паттерн “стратегия” для загрузки сервисов, мы получаем следующие преимущества:

- Загружаются только те сервисы, которые реально будут использованы в данном конкретном приложении. Если у вас нет сервера MongoDB, то у вас не будет и сервисов, которые от него зависят.
- Такая конфигурация открыта для расширения, так как вы можете добавить имя другого менеджера хранения в список в классе Configuration и затем добавить определения сервисов и алиасов - всё готово.

6.9 Загрузка и конфигурирование дополнительных сервисов

Положим у вас есть бандл, который должен заниматься фильтрацией входных данных. Вероятно вы предоставляете несколько различных сервисов, например сервисы для фильтрации данных html-форм, а также сервисы для фильтрации данных, сохранённых при помощи Doctrine ORM. Соответственно должна быть возможность в любое время активировать или деактивировать любой из этих сервисов или целую коллекцию сервисов, так как могут быть не применимы к вашей конкретной ситуации. Имеется удобный способ для того чтобы при конфигурировании добиться такой возможности:

```
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('input_filter');

        $rootNode
            ->children()
                ->arrayNode('form_integration')
                    // будет активно по умолчанию
                    ->canBeDisabled()
                ->end()
                ->arrayNode('doctrine_orm_integration')
                    // будет отключено по умолчанию
                    ->canBeEnabled()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Имея такое дерево конфигурации, вы можете активировать или деактивировать отдельные части бандла в config.yml:

```
input_filter:
    form_integration:
        enabled: false
    doctrine_orm_integration:
        enabled: true
```

В классе расширения вам остаётся только загрузить соответствующие сервисы:

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

        if ($processedConfig['doctrine_orm_integration']['enabled']) {
            $this->loadDoctrineORMIntegration(
                $container,
                $processedConfig['doctrine_orm_integration']
            );
        }

        if ($processedConfig['form_integration']['enabled']) {
            $this->loadFormIntegration(
                $container,
                $processedConfig['form_integration']
            );
        }

        ...
    }

    private function loadDoctrineORMIntegration(
        ContainerBuilder $container,
        array $configuration
    ) {
        // загружаем сервисы и т.д.
        ...
    }

    private function loadFormIntegration(
        ContainerBuilder $container,
        array $configuration
    ) {
        ...
    }
}
```

Каждая из отдельных частей бандла теперь может быть загружена независимо от других.

Подчищаем класс конфигурации

Одна или две части бандла можно легко поддерживать таким образом, как это описано выше, но если развивать ваш бандл таким образом, вскоре класс Configuration будет содержать слишком много строк кода для одного метода. Можно слегка подчистить этот код, задействовав метод `append()` в комбинации с несколькими приватными методами:

```php
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();

        $rootNode = $treeBuilder->root('input_filter');

        $rootNode
            ->append($this->createFormIntegrationNode())
            ->append($this->createDoctrineORMIntegrationNode())
        ;

        return $treeBuilder;
    }

    private function createDoctrineORMIntegrationNode()
    {
        $builder = new TreeBuilder();

        $node = $builder->root('doctrine_orm_integration');

        $node
            ->canBeEnabled()
            ->children()
                // тут можно добавить дополнительные опции конфигурации
                ...
            ->end();

        return $node;
    }

    private function createFormIntegrationNode()
    {
        ...
    }
}
```

6.10 Настраиваем какой сервис использовать

Вместо того, чтобы использовать паттерн “стратегия” для загрузки сервисов, вы также можете разрешить разработчикам конфигурировать вручную сервисы, которые они хотят использовать. Например, если вашему бандлу нужен какой-либо сервис-шифратор (encrypter) и бандл не содержит таковой, вы можете попросить разработчика указать ID сервиса-шифратора:

```
matthias_security:
    encrypter_service: my_encrypter_service_id
```

Класс конфигурации в таком случае будет выглядеть таким образом:

```php
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('matthias_security');

        $rootNode
            ->children()
                ->scalarNode('encrypter_service')
                    ->isRequired()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

В классе расширения бандла, вам нужно создать алиас для сконфигурированного сервиса.

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

        $container->setAlias(
            'matthias_security.encrypter',
            $processedConfig['encrypter_service']
        );
    }
}
```

Таким образом, даже учитывая что id сервиса-шифратора может быть любым, теперь вы всегда будете точно знать, как обратиться к нему из любого сервиса по известному вам и вашему бадлу псевдониму:

```xml
<service id="matthias_security.encrypted_data_manager" class="...">
    <argument type="service" id="matthias_security.encrypter" />
</service>
```

Конечно же вы не можете быть уверены в том, что сконфигурированный вручную сервис - это действительно валидный экземпляр шифратора. Вы не можете проверить это на этапе конфигурации, поэтому придётся проверять это во время выполнения. Типичный способ выполнить такую проверку - добавить подсказку типа (type-hint) в классы сервисов, которые используют этот сервис-шифратор:

```php
class EncryptedDataManager
{
    public function __construct(EncrypterInterface $encrypter)
    {
        // здесь мы можем быть уверены, что $encrypter валидный
    }
}
```

6.11 Полностью динамическое определение сервисов

Также встречаются ситуации, когда заранее вы практически ничего не знаете о сервисе, который вам нужен, до того момента, когда у вас есть обработанная конфигурация. Положим, вы хотите, чтобы пользователи вашего бандла могли бы определять сервисы в виде набора ресурсов. Эти ресурсы могут иметь тип `файл` или `директория`. Вы хотите создавать эти сервисы на лету, так как они могут отличаться от приложения к приложению и вам необходимо собирать их, используя особый таг - `resource`. Ваш класс Configuration для данного случая может выглядеть примерно так:

```php
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('resource_management');

        $rootNode
            ->children()
                ->arrayNode('resources')
                    ->prototype('array')
                        ->children()
                            ->scalarNode('type')
                                ->validate()
                                    ->ifNotInArray(
                                        array('directory', 'file')
                                    )
                                    ->thenInvalid('Invalid type')
                                ->end()
                            ->end()
                            ->scalarNode('path')
                            ->end()
                        ->end()
                    ->end()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Пример конфигурации ресурсов:

```
resource_management:
    resources:
        global_templates:
            type: directory
            path: Resources/views
        app_kernel:
            type: file
            path: AppKernel.php
```

Когда ресурсы определены таким образом, вы можете создавать определения сервисов для них в расширении контейнера:

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

        $resources = $processedConfig['resources'];

        foreach ($resources as $name => $resource) {
            $this->addResourceDefinition($container, $name, $resource);
        }
    }

    private function addResourceDefinition(
        ContainerBuilder $container,
        $name,
        array $resource
    ) {
        // определяем класс
        $class = $this->getResourceClass($resource['type']);

        $definition = new Definition($class);

        // добавляем таг
        $definition->addTag('resource');

        $serviceId = 'resource.'.$name;

        $container->setDefinition($serviceId, $definition);
    }

    private function getResourceClass($type)
    {
        if ($type === 'directory') {
            return 'Resource\Directory';
        } elseif ($type === 'file') {
            return 'Resource\File';
        }

        throw new \InvalidArgumentException('Type not supported');
    }
}
```

Если эти, созданные вручную, определения сервисов требуют некоторых аргументов, вызовов методов и т.д. - используйте техники, описанные выше, для того чтобы добавить динамически и их.

7 Приёмы создания параметров

Сервисный контейнер помимо сервисов содержит также и параметры. Параметры - это простые значения, которые могуть быть представлены в виду констант-скаляров или массивов в любых сочетаниях. Таким образом, валидными параметрами будут: строка ‘matthias’, число 23, а также массив `array(23 => 'matthias')`, `array(23 => array('matthias'))`, и т.д.

Вы можете определять параметры с любым ключом, какой пожелаете. Тем не менее, желательно придерживаться следующего формата: `name_of_your_bundle_without_bundle.parameter_name`. Параметры можно определить в разных местах приложения.

7.1 Файл parameters.yml

Некоторые базовые параметры вашего приложения (у которых, вероятно нет значения по-умолчанию) можно обнаружить в файле `/app/config/parameters.yml`. Параметры загружаются вместе с определениями сервисов и конфигурациями расширений контейнера. По этой причине стандартный файл конфигурации `config.yml` начинается с таких строк:

```
1  imports:
2      - { resource: parameters.yml }
3      - { resource: security.yml }
4
5  framework:
6      secret: %secret%
7      ...
```

Сначала импортируются файлы `parameters.yml` и `security.yml`. Файл `parameters.yml` как правило начинается так:

```
1  parameters:
2      database_driver: pdo_mysql
3      ...
```

А `security.yml`, в свою очередь, как правило начинается так:

```
1  security:
2      encoders:
3          Symfony\Component\Security\Core\User\User: plaintext
4      ...
```

Эти файлы будут импортированы в том порядке, в котором они указаны. Таким образом, `config.yml` мог бы выглядеть так:

```
parameters:
    ...
security:
    ...
framework:
    ...
```

Так как все конфигурационные массивы будут в конце концов объединены (merged), то в `config.yml` можно переопределить любой из параметров, указанных в `parameters.yml`:

```
parameters:
    ... # загружено из parameters.yml
    database_driver: pdo_sqlite
```

В файле `config.yml` можно даже создавать определения сервисов (их также можно создавать в любом конфигурационном файле, отвечающем за конфигурацию сервисного контейнера):

```
parameters:
    ...
services:
    some_service_id:
        class: SomeClass
```

7.2 Определение и загрузка параметров

Значения, определённые в файлах `config.yml` и `parameters.yml`, а также в определениях сервисов и их вргументоа могут содержать т.н. заполнители (или места подстановки aka placeholders) для значений, которые могу быть определны в виде параметров. Когда сервисный контейнер компилируется, значения, содержащие заполнители также обрабатываются и принимают свои окончательные значения путём замены заполнителей на соответствующие параметры. Например, в прмере выше мы определили параметр `database_driver` в `parameters.yml`. В файле config.yml мы можем сослаться на этот параметр используя заполнитель %database_driver%:

```
doctrine:
    dbal:
        driver: %database_driver%
```

При создании опредений сервисов, бандлы Symfony обычно применяют этот приём для указания имени класса сервиса:

```
<parameters>
    <parameter key="form.factory.class">
        Symfony\Component\Form\FormFactory
    </parameter>
</parameters>

<service id="form.factory" class="%form.factory.class%">
</service>
```

Параметры для имени класса

Использование параметров для имён классов даёт возможность другим частям приложения переопределять эти параметры, вместо того, чтобы напрямую манипулировать определениями сервисов. Если представить, что у всех сервисов классы будут определены через параметры, то эти параметры в конце концов попали бы в контейнер и вы налету могли бы выполнять вызов типа `$container->getParameter('form.factory.class')`, чтобы получить имя класса фабрики форм (но, вероятно, этого никогда не случится). Я считаю этот подход избыточным и не рекомендую его использовать при создании ваших сервисов.

Когда вы захотите изменить определение сервиса - вы должны это делатьв соответствующем `compiler pass`:

```
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;

class ReplaceClassCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        $myCustomFormFactoryClass = ...;

        $container
            ->getDefinition('form.factory')
            ->setClass($myCustomFormFactoryClass);
    }
}
```

Сборка значений параметров вручную

Когда вы используете параметры в ваших расширениях (или в компиляторе - compiler pass), значения этих параметров ещё не определены. Например, ваш бандл может определять параметр `my_cache_dir`, ссылающийся на параметр `%kernel.cache_dir%`, который в свою очередь содержит расположение директории для кэша, используемой ядром:

```
parameters:
    my_cache_dir: %kernel.cache_dir%/my_cache
```

Метод `load()` вашего расширения должен создать эту директорию, если она не существует:

```
class MyExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $myCacheDir = $container->getParameter('my_cache_dir');

        ...
    }
}
```

Когда метод `load()` будет вызван, значение параметра `my_cache_dir` всё еще будет иметь вид `%kernel.cache_dir%/my_cache`. К счастью, вы можете использовать метод `ParameterBag::resolveValue()` для того, чтобы заменить все "заполнители" их реальными значениями:

```
$myCacheDir = $container->getParameter('my_cache_dir');

$myCacheDir = $container->getParameterBag()->resolveValue($myCacheDir);

// теперь вы можете создать директорию для кэширования
mkdir($myCacheDir);
```

Параметры ядра

Ядро добавляет следующие параметры к контейнеру, перед тем как загрузить бандлы (пути указаны от корня проекта):

- `kernel.root_dir` - место расположения класса ядра (как правило `/app`)
- `kernel.environment` - наименование окружения (например, `dev`, `prod`, и т.д.)
- `kernel.debug` - активирован или нет режим отладки (`true` или `false`)
- `kernel.name` - имя директории, где расположено ядро (как правило `app`)
- `kernel.cache_dir` - место расположения директории с кэшем (по умолчанию `/app/cache` в Symfony 2.x и `/var/cache` в 3.x)
- `kernel.logs_dir` - место расположения директории с логами (по умолчанию `/app/logs` в Symfony 2.x и `/var/logs` в 3.x)
- `kernel.bundles` - список активированных бандлов (например, `array('FrameworkBundle' => 'Symfony\\Bundle\\FrameworkBundle\\FrameworkBundle', ...)`),
- `kernel.charset` - кодировка, используемая для ответа (например, UTF-8)
- `kernel.container_class` - имя класса сервисного контейрера (например в dev-окружении по умолчанию будет `appDevDebugProjectContainer`), который располагается в `kernel.cache_dir`.

Ядро также добавляет в контейнер значения переменных окружения, которые начинаются с `SYMFONY__` (если таковые обнаружатся). Перед добавлением таких параметров удаляется префикс, двойной подчерк `__` заменяется на точку `.`, имя приводится к нижнему регистру, таким образом, например, переменная окружения `SYMFONY__DATABASE__USER` станет параметром контейнера `database.user`.

7.3 Определяем параметры в расширениях контейнера

В будущем в множество раз окажетесь в таких ситуациях:

- Вы хотите, чтобы разработчик предоставил некоторое значение вашему бандлу через конфигурацию в `config.yml`.
- Затем вы вероятно захотите использовать это значение в качестве аргумента для одного из сервисов вашего бандла.

Положим у вас есть бандл `BrowserBundle` и вы хотите, чтобы разработчик указал значение таймаута для сервиса `browser`:

```
browser:
    timeout: 30
```

Для этого класс конфигурации должен иметь следующий вид:

```
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('browser');

        $rootNode
            ->children()
                ->scalarNode('timeout')
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Затем в расширении контейнера вашего бандла нужно обработать конфигурационные значения из `config.yml` таким образом:

```
class BrowserExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        // загружаем определения сервисов
        $fileLocator = new FileLocator(__DIR__.'/../Resources/config');
        $loader = new XmlFileLoader($container, $fileLocator);
        $loader->load('services.xml');

        // обрабатываем конфигурацию
        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );
    }
}
```

Сервис `browser` определён в `services.xml`:

```
<service id="browser" class="...">
    <argument>%browser.timeout%</argument>
</service>
```

Значение таймаута переданное через config.yml (30) будет доступно в методе расширения `load()` в виде `$processedConfig['timeout']`, таким образом, вам остаётся лишь создать параметр `browser.timeout` в контейнере и присвоить ему переданное значение:

```
$container->setParameter('browser.timeout', $processedConfig['timeout']);
```

7.4 Переопределение параметров при помощи компилятора (compiler pass)

Иногда вам может потребоваться проанализировать и заменить параметр, определённый в другом бандле. Например, вам может потребоваться модифицировать иерархию ролей пользователя, которая определена в `security.yml`, и доступна в виде параметра `security.role_hierarchy.roles`. Вот как выглядит стандартная иерархия:

```
array (
    'ROLE_ADMIN' => array (
        0 => 'ROLE_USER',
    ),
    'ROLE_SUPER_ADMIN' => array (
        0 => 'ROLE_USER',
        1 => 'ROLE_ADMIN',
        2 => 'ROLE_ALLOWED_TO_SWITCH',
    ),
)
```

Положим, у вас есть другой механизм для определения иерархии ролей (к примеру, вы можете загружать её из другого файла настроек), в этом случае вы можете модифицировать или заменить иерархию ролей через отдельный compiler pass:

```
class EnhanceRoleHierarchyPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        $parameterName = 'security.role_hierarchy.roles';

        $roleHierarchy = $container->getParameter($parameterName);

        // модифицируем иерархию ролей
        ...

        $container->setParameter($parameterName, $roleHierarchy);
    }
}
```

Не забывайте регистрировать ваши проходы компилятора в классе вашего бандла:

```php
class YourBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        $container->addCompilerPass(new EnhanceRoleHierarchyPass());
    }
}
```

Валидация определений сервисов

Когда что-то не в порядке с определением сервиса, в большинстве случаев вы об этом узнаете лишь запустив приложение.

Для того, чтобы получать предупреждения о некорректных определениях сервисов пораньше, установите бандл SymfonyServiceDefinitionValidator и активируйте его compiler pass. После этого, во время компиляции контейнера, вы автоматически получите сообщения об ошибках, возникших на этапе компиляции, например определение сервиса с несуществующим классом или вызов несуществующего метода. Валидатор также умеет распознавать ошибки в аргументах конструктора.

III Структура проекта

В предыдущих частях мы познакомились с тем, что происходит внутри ядра, когда оно создаёт ответ на каждый переданный ему запрос. Мы также основательно познакомились со всеми способами, с помощью которых вы можете создавать бандлы с гибкой конфигурацией. С этим багажом знаний, вы можете помещать ваш код в сервисы и делать его доступным для других частей приложения. Когда же речь заходит о структуре всего приложения, то тут всё еще есть вопросы. Как не допустить того, что весь код приложения сосредотачивался в контроллерах? Как писать код, пригодный для повторного использования? И как, наконец, писать код, который может работать как в web так и в командной строке?

Эти вопросы мы и рассмотрим в этой главе.

8 Организация слоёв приложения

8.1 Тонкие контроллеры

Во многих Symfony приложениях, очень часто контроллеры выглядят вот так:

```
1  namespace Matthias\AccountBundle\Controller;
2
3  use Symfony\Bundle\FrameworkBundle\Controller;
4  use Symfony\Component\HttpFoundation\Request;
5  use Matthias\AccountBundle\Entity\Account;
6
7  class AccountController extends Controller
8  {
9      public function newAction(Request $request)
10     {
11         $account = new Account();
12
13         $form = $this->createForm(new AccountType(), $account);
14
15         if ($request->isMethod('POST')) {
16             $form->bind($request);
17
18             if ($form->isValid()) {
19                 $confirmationCode = $this
20                     ->get('security.secure_random')
21                     ->nextBytes(4);
22                 $account
23                     ->setConfirmationCode(md5($confirmationCode));
24
25                 $entityManager = $this->getDoctrine()->getManager();
26                 $entityManager->persist($account);
27                 $entityManager->flush();
28
29                 $this->sendAccountConfirmationMessage($account);
30
31                 return $this->redirect($this->generateUrl('mailbox_index'));
32             }
33         }
34
35         return array(
36             'form' => $form->createView(),
37         );
38     }
39
40     private function sendAccountConfirmationMessage(Account $account)
41     {
42         $message = \Swift_Message::newInstance()
43             ->setSubject('Confirm account')
44             ->setFrom('noreply@matthias.com')
45             ->setTo($account->getEmailAddress())
46             ->setBody('Welcome! ...');
47
48         $this->get('mailer')->send($message);
49     }
50 }
```

Если взглянуть на контроллер `newAction`, вы можете увидеть там форму `AccountType`, связанную с классом `Matthias\AccountBundle\Entity\Account`. После привязки дан-

ных и валидации формы генерируется код подтверждения и объект аккаунта сохраняется в базу. После этого создаётся и высылается подтверждение по электронной почте.

В этом контроллере происходит много всего, и в результате мы имеем следующее:

1. Нет возможности разделить код, который можно повторно использовать от кода, специфичного для данного конкретного проекта. Положим вы хотите повторно использовать часть логики создания аккаунта в будущих ваших проектах. В данном случае это может быть достигнуто лишь копипастингом кода из этого контроллера в новый проект. Это называется иммобильность (неподвижность, недвижимость) - когда код не может быть легко перенесён в другое приложение.
2. Также у нас нет возможности повторно использовать логику содания аккаунта в какой-либо другой части этого приложения, так как весь код размещён внутри одного контроллера. Представьте, что вам нужно создать консольную команду, для импорта CSV файла, который содержит данные пользовательских аккаунтов из предыдущей версии приложения. В этом случае у вас нет возможности проделать эту процедуру без копипасты (опять?!) части кода из контроллера в другой класс. Я называю это контроллероцентризмом - когда код формируется преимущественно вокруг контроллеров.
3. Код очень тесно связан с двумя другими библиотеками: `SwiftMailer` и `Doctrine ORM`. Нет возможности запустить этот код без любой из этих библиотек, несмотря на то, что есть много альтернатив для них обеих. Это состояние называется “тесными связями” и это, как правило, такой код не является хорошим.

Для того, чтобы иметь возможность повторного использования кода в других приложениях, или же для того, чтобы иметь возможность спользовать код в других частях того же приложения, или же для того, чтобы можно было легко сменить реализацию менеджера хранения, вам нужно разделить код на несколько классов, каждый из которых занимется только одной задачей.

8.2 Обработчики форм

Первым нашим шагом будет следущий: делегируем обработку формы специальному обработчику. Этот обработчик будет простым классом, который будет обрабатывать нашу форму и выполнять все действия связанные с этим. Результатом первого рефакторинга будет `CreateAccountFormHandler`:

```
1  namespace Matthias\AccountBundle\Form\Handler;
2
3  use Symfony\Component\HttpFoundation\Request;
4  use Symfony\Component\Form\FormInterface;
5  use Doctrine\ORM\EntityManager;
6  use Matthias\AccountBundle\Entity\Account;
7  use Symfony\Component\Security\Core\Util\SecureRandomInterface;
8
9  class CreateAccountFormHandler
10 {
11     private $entityManager;
12     private $secureRandom;
13
```

```
14      public function __construct(
15          EntityManager $entityManager,
16          SecureRandomInterface $secureRandom
17      ) {
18          $this->entityManager = $entityManager;
19          $this->secureRandom = $secureRandom;
20      }
21
22      public function handle(FormInterface $form, Request $request)
23      {
24          if (!$request->isMethod('POST')) {
25              return false;
26          }
27
28          $form->bind($request);
29
30          if (!$form->isValid()) {
31              return false;
32          }
33
34          $validAccount = $form->getData();
35
36          $this->createAccount($validAccount);
37
38          return true;
39      }
40
41      private function createAccount(Account $account)
42      {
43          $confirmationCode = $this
44              ->secureRandom
45              ->nextBytes(4);
46
47          $account
48              ->setConfirmationCode(md5($confirmationCode));
49
50          $this->entityManager->persist($account);
51          $this->entityManager->flush();
52      }
53  }
```

Определение сервиса для этого обработчика будет следующим:

```
1  <service id="matthias_account.create_account_form_handler"
2      class="Matthias\AccountBundle\Form\Handler\CreateAccountFormHandler">
3      <argument type="service" id="entity_manager" />
4      <argument type="service" id="security.secure_random" />
5  </service>
```

Как вы можете видеть, метод `handle()` возвращает значение `true`, если он смог выполнить всё, что должен был, и `false`, если что-то пошло не так при обработке формы и форма должна быть отображена опять. Используя этот простой механизм, мы слегка “похудеем” наш контроллер:

```php
class AccountController extends Controller
{
    public function newAction(Request $request)
    {
        $account = new Account();

        $form = $this->createForm(new AccountType(), $account);

        $formHandler = $this
            ->get('matthias_account.create_account_form_handler');

        if ($formHandler->handle($form, $request)) {
            $this->sendAccountConfirmationMessage($account);

            return $this->redirect($this->generateUrl('mailbox_index'));
        }

        return array(
            'form' => $form->createView(),
        );
    }
}
```

Обработчики форм должны быть как можно более простыми и не должны создавать исключений, которые так или иначе были предназначены для предоставления пользователю информации о возникших проблемах. Любую обратную связь с пользователем в обработчике формы вы должны осуществлять путём добавления ошибок к обрабатываемой форме и возвращая `false`, что будет означать наличие проблемы при обработке формы:

```php
use Symfony\Component\Form\FormError;

public function handle(FormInterface $form, Request $request)
{
    if (...) {
        $form->addError(new FormError('У нас возникла проблемка...'));

        return false;
    }
}
```

Тем не менее, всегда держите в уме, что в идеале любая ошибка, относящаяся к форме - это ошибка валидации. Это означает, что обработчик формы не должен выполнять валидацию любым другим способом, кроме как вызовом метода формы `isValid()`. Просто создайте нужный вам класс проверки ограничений (validation constraint) и валидатор для него, чтобы быть уверенным, что все проверки на валидность доступны централизованно и, следовательно, могут быть повторно использованы.

8.3 Доменные менеджеры

Обработчик формы (и, возможно, класс формы) - это замечательные кандидаты на повторное использование. Тем не менее, в нашем лучае в нём всё ещё происходит много разных действий. Полагая, что обязанностью обработчика формы является “просто обработать форму”, окажется, что создание кода подтверждения тут лишнее. Также, обращение к слою

хранения данных (в нашем случае это Doctrine ORM) - это тоже лишнее для простого обработчика форм.

Решением этой проблемы является делегирование задач относящихся к доменной модели специализированным доменным менеджерам. Эти менеджеры могут работать напрямую со слоем хранения данных. Давайте создадим класс менеджера для задач, связанных с аккаунтом и назовём его `AccountManager`. Он может выглядеть таким образом:

```
namespace Matthias\AccountBundle\DomainManager;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Form\FormInterface;
use Doctrine\ORM\EntityManager;
use Matthias\AccountBundle\Entity\Account;
use Symfony\Component\Security\Core\Util\SecureRandomInterface;

class AccountManager
{
    private $entityManager;
    private $secureRandom;

    public function __construct(
        EntityManger $entityManager,
        SecureRandomInterface $secureRandom
    ) {
        $this->entityManager = $entityManager;
        $this->secureRandom = $secureRandom;
    }

    public function createAccount(Account $account)
    {
        $confirmationCode = $this
            ->secureRandom
            ->nextBytes(4);

        $account
            ->setConfirmationCode(md5($confirmationCode));

        $this->entityManager->persist($account);
        $this->entityManager->flush();
    }
}
```

Теперь обработчик формы будет просто использовать `AccountManager` для того чтобы собственно создать аккаунт:

```
class CreateAccountFormHandler
{
    private $accountManager;

    public function __construct(AccountManager $accountManager)
    {
        $this->accountManager = $accountManager;
    }

    public function handle(FormInterface $form, Request $request)
    {
        ...

        $validAccount = $form->getData();
```

```
15
16          $this->accountManager->createAccount($validAccount);
17      }
18  }
```

Ниже представлены определния соответствующих сервисов для обработчика формы и доменного менеджера:

```
1   <service id="matthias_account.create_account_form_handler"
2       class="Matthias\AccountBundle\Form\Handler\CreateAccountFormHandler">
3       <argument type="service" id="matthias_account.account_manager" />
4   </service>
5
6   <service id="matthias_account.account_manager"
7       class="Matthias\AccountBundle\DomainManager\AccountManager">
8       <argument type="service" id="entity_manager" />
9       <argument type="service" id="security.secure_random" />
10  </service>
```

Доменные менеджеры могут выполнять с объектом домена всё, что этот объект не может выполнить самостоятельно. Вы можете использовать их для инкапсуляции следующей логики:

- Создание и сохранение объектов
- Создание связей между объектами (например, между двумя пользователями)
- Дублирования объектов
- Удаления объектов
- ...

8.4 События

Как вы могли отметить выше, внутри контроллера, после содания нового аккаунта, отправлялось письмо с подтверждением. Подобные действия лучше выносить из контроллеров. Отправка писем - это далеко не единственное действие, которое может потребоваться после создания нового аккаунта. Возможно потребуется заполнить для нового пользователя некоторые настройки по-умолчанию, или же отправить уведомление владельцу сайта, что в его продукте зарегистрирован новый пользователь.

Это прекрасный случай воспользоваться событийно-ориентированным (`event-driven`) подходом: в нашей ситуации внутри `AccountManager` происходит одно из базовых событий, (а именно, “создан новый аккаунт”). Другие части приложения должны иметь возможность отреагировать на это событие. В этом случае должен существовать как минимум слушатель события, который отправил бы письмо с подтверждением аккаунта новому пользователю.

Для обработки такого специализированного события, использующего некоторые данные, нужно создать новый класс события, который должен наследоваться от базового класса `Event`:

```php
namespace Matthias\AccountBundle\Event;

use Symfony\Component\EventDispatcher\Event;

class AccountEvent extends Event
{
    private $account;

    public function __construct(Account $account)
    {
        $this->account = $account;
    }

    public function getAccount()
    {
        return $this->account;
    }
}
```

Затем нужно придумать имя для нового события - назовём его `matthias_account.new_account_created`. Как правило, хорошей практикой является хранение этого имени в виде константы в отдельном классе где-то в вашем бандле:

```php
namespace Matthias\AccountBundle\Event;

class AccountEvents
{
    const NEW_ACCOUNT_CREATED = 'matthias_account.new_account_created';
}
```

Теперь нам необходимо модифицировать `AccountManager`, чтобы отправить наше новое событие `matthias_account.new_account_created`:

```php
namespace Matthias\AccountBundle\DomainManager;

use Symfony\Component\EventDispatcher\EventDispatcherInterface;
use Matthias\AccountBundle\Event\AccountEvents;
use Matthias\AccountBundle\Event\AccountEvent;

class AccountManager
{
    ...

    private $eventDispatcher;

    public function __construct(
        ...
        EventDispatcherInterface $eventDispatcher
    ) {
        ...

        $this->eventDispatcher = $eventDispatcher;
    }

    public function createAccount(Account $account)
    {
        ...

        $this->eventDispatcher->dispatch(
```

```
            AccountEvents::NEW_ACCOUNT_CREATED,
            new AccountEvent($account)
        );
    }
}
```

Не забудьте добавить сервис `event_dispatcher` в качестве аргумента к определению сервиса `AccountManager`:

```
<service id="matthias_account.account_manager"
    class="Matthias\AccountBundle\DomainManager\AccountManager">
    <argument type="service" id="entity_manager" />
    <argument type="service" id="security.secure_random" />
    <argument type="service" id="event_dispatcher" />
</service>
```

Слушатель события `matthias_account.new_account_created` будет выглядеть следующим образом:

```
namespace Matthias\AccountBundle\EventListener;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Matthias\AccountBundle\Event\AccountEvents;
use Matthias\AccountBundle\Event\AccountEvent;

class SendConfirmationMailListener implements EventSubscriberInterface
{
    private $mailer;

    public static function getSubscribedEvents()
    {
        return array(
            AccountEvents::NEW_ACCOUNT_CREATED => 'onNewAccount'
        );
    }

    public function __construct(\SwiftMailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function onNewAccount(AccountEvent $event)
    {
        $this->sendConfirmationMessage($event->getAccount());
    }

    private function sendConfirmationMessage(Account $account)
    {
        $message = \Swift_Message::newInstance();

        ...

        $this->mailer->send($message);
    }
}
```

Так как этому слушателю для отправки сообщения нужен mailer, нам нужно внедрить его, добавив в качестве аргумента в определение сервиса слушателя. Также необходимо добавить метку `kernel.event_subscriber`, которая определит `SendConfirmationMailListener` в качестве подписчика на события:

```
<service id="life_online_account.send_confirmation_mail_listener"
    class="Matthias\AccountBundle\EventListener\SendConfirmationMailListener">
    <argument type="service" id="mailer" />
    <tag name="kernel.event_subscriber" />
</service>
```

Лучшие практики использования слушателей событий

Слушатель события должен именоваться от выполняемого им действия, а не от события, которое он слушает. Таким образом, вместо того, чтобы назвать слушатель `NewAccountEventListener`, вы должны назвать его `SendConfirmationMailListener`. Это также поможет другим разработчикам, если они захотят найти место в коде, где отправляется сообщение с подтверждением регистрации.

Также, когда должно выполниться другое действие при возникновении события, как, например, отправка еще одного сообщения, но уже владельцу сайта, вам нужно создать другой слушатель для этого события, вместо того, чтобы добавлять код в существующий слушатель. Включение или отключение конкретного слушателя - это простая операция, что уменьшает сложность поддержки кода, потому что вы не сможете изменить поведение системы случайно.

События уровня хранения (persistence)

Как вы можете помнить, `AccountManager` (доменный менеджер) генерирует код подтверждения для учётной записи прямо перед тем, как сохранить его:

```
class AccountManager
{
    private $entityManager;
    private $secureRandom;

    public function __construct(
        EntityManger $entityManager,
        SecureRandomInterface $secureRandom
    ) {
        $this->entityManager = $entityManager;
        $this->secureRandom = $secureRandom;
    }

    public function createAccount(Account $account)
    {
        $confirmationCode = $this
            ->secureRandom
            ->nextBytes(4);

        $account
            ->setConfirmationCode(md5($confirmationCode));

        $this->entityManager->persist($account);
        $this->entityManager->flush();
    }
}
```

Это не очень хороший подход. Повторю ещё раз: учётная запись может быть создана где-то ещё и она может не иметь кода подтверждения. С точки зрения “ответственности

(responsibility)", если мы посмотрим на зависимости `AccountManager`, станет непонятно, почему там должен быть объект, реализующий интерфейс `SecureRandomInterface` и возникнет вопрос: зачем ему это, если в его обязанности входит лишь создание учётной записи?

Эту логику нужно вынести куда-то в другое место, ближе к реальному сохранению новой учётной записи. Большинство реализаций слоя хранения данных поддерживает что-то типа событий или поведений (behaviors), при помощи которых вы можете внелриться в процесс сохранения, обновления или удаления объектов.

Doctrine ORM это реализуется черезк подписчики события:

```
use Doctrine\Common\EventSubscriber;
use Doctrine\ORM\Event\LifecycleEventArgs;

class CreateConfirmationCodeEventSubscriber implements EventSubscriber
{
    private $secureRandom;

    public function __construct(SecureRandomInterface $secureRandom)
    {
        $this->secureRandom = $secureRandom;
    }

    public function getSubscribedEvents()
    {
        return array(
            'prePersist'
        );
    }

    public function prePersist(LifecycleEventArgs $event)
    {
        // this will be called for *each* new entity

        $entity = $event->getEntity();
        if (!($entity instanceof Account)) {
            return;
        }

        $this->createConfirmationCodeFor($entity);
    }

    private function createConfirmationCodeFor(Account $account)
    {
        $confirmationCode = $this
            ->secureRandom
            ->nextBytes(4);

        $account
            ->setConfirmationCode(md5($confirmationCode));
    }
}
```

Вы можете зарегистрировать этот подписчик, использовав метку `doctrine.event_subscriber`:

```
<service id="create_confirmation_code_listener" class="...">
    <tag name="doctrine.event_subscriber" />
</service>
```

Вообще говоря событий там имеется больше, например, `postPersist`, `preUpdate`, `preFlush`, и т.д. Эти события позволяют вам внедряться в любую стадию жизненного цикла ваших сущностей. В частности, событие `preUpdate` может быть очень удобным, для того, чтобы определять что кто-то изменил значение какого-либо поля:

```
use Doctrine\ORM\Event\PreUpdateEventArgs;

class CreateConfirmationCodeEventSubscriber implements EventSubscriber
{
    public function getSubscribedEvents()
    {
        return array(
            'preUpdate'
        );
    }

    public function preUpdate(PreUpdateEventArgs $event)
    {
        $entity = $event->getEntity();
        if (!($entity instanceof Account)) {
            return;
        }

        if ($event->hasChangedField('emailAddress')) {
            // create a new confirmation code
            $confirmationCode = ...;
            $event->setNewValue('confirmationCode', $confirmationCode);
        }
}
```

Как вы можете видеть, слушатели события `preUpdate` получают особый объект события. Вы можете использовать его для проверки полей, которые были изменены и для того, чтобы изменить что-то ещё.

> **Подводные камни событий Doctrine**
>
> Ниже приведено несколько неочевидных вещей, связанных с использованием событий Doctrine: - Событие `preUpdate` отправляется только тогда, когда значение какого-либо поля было изменено, то есть не обязательно каждый раз когда вы выполняете метод `flush()` у вашего менеджера сущностей. - Событие `prePersist` отправляется только в том случае, если сущность ранее не была сохранена (persisted) ранее. - В некоторых ситуациях вы можете выполнить изменения в объекте сущности слишком поздно и вам нужно будет вручную запросить у `UnitOfWork` пересчитать изменения:
>
> ```
> $entity = $event->getEntity();
> $className = get_class($entity);
> $entityManager = $event->getEntityManager();
> $classMetadata = $entityManager->getClassMetadata($className);
> $unitOfWork = $entityManager->getUnitOfWork();
> $unitOfWork->recomputeSingleEntityChangeSet($classMetadata, $entity);
> ```

IV Соглашения по конфигурированию

Настройка конфигурации приложения

Соглашения по конфигурированию

V Безопасность

Введение

Аутентификация и сессии

Проектирование контроллеров

Проверка ввода

Экранирование вывода

Будучи скрытным...

VI Используем аннотации

Введение

Аннотация - это лишь Value Object

Приемлемые случаи для использования аннотаций

Используем аннотации в вашем Symfony приложении

Проектирование для повторного использования

Заключение

VII Быть Symfony разработчиком

Код для повторного использования имеет слабые связи

Код для повторного использования должен быть переносимым

Код для повторного использования должен быть расширяемым

Код для повторного использования должен быть прост в использовании

Код для повторного использования должен быть надёжен

Заключение