

Один год с Symfony

*Пишем красивый, пригодный для повторного использования
код на Symfony (версии 2.x, 3.x)*

Matthias Noback
перевод: Дмитрий Быкадоров

Один год с Symfony

Перевод книги “A year with Symfony” от Matthias Noback [В ПРОЦЕССЕ]

Dmitry Bykadorov и Matthias Noback

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

Эта версия была опубликована на 2017-04-22



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#)

Оглавление

| | |
|---|----|
| От переводчика | 1 |
| Предисловие | 2 |
| Введение | 4 |
| Благодарности | 5 |
| Кому предназначена эта книга | 6 |
| Соглашения | 7 |
| Обзор содержания книги | 8 |
| I От запроса до ответа | 9 |
| HttpKernelInterface | 10 |
| Загрузка ядра | 11 |
| Бандлы и расширения контейнера | 12 |
| Создание сервисного контейнера | 13 |
| От Kernel до HttpKernel | 14 |
| События, приводящие к ответу | 16 |
| Ранний ответ | 16 |
| Слушатели kernel.request, о которых вам нужно знать | 17 |
| Определение контроллера для запуска | 18 |
| Возможность замены контроллера | 20 |
| Примечательные слушатели события kernel.controller | 21 |
| Сбор аргументов для выполнения контроллера | 22 |
| Выполнение контроллера | 23 |
| Вход в слой представления (view) | 24 |
| Примечательные слушатели события kernel.view | 25 |
| Фильтрация ответа | 25 |
| Примечательные слушатели события kernel.response | 26 |
| Обработка исключений | 27 |
| Примечательные слушатели события kernel.exception | 28 |
| Подзапросы | 30 |
| Когда используются подзапросы? | 30 |
| II Приёмы внедрения зависимостей | 32 |
| Что такое бандл (bundle) | 33 |
| Приёмы создания сервисов | 34 |
| Обязательные зависимости | 34 |

ОГЛАВЛЕНИЕ

| | |
|---|-----------|
| Обязательные параметры конструктора | 34 |
| Абстрактные определения для дополнительных аргументов | 34 |
| Вызов обязательных set-методов (setters) | 36 |
| Вызов методов в абстрактных сервисах | 37 |
| Необязательные (опциональные) зависимости | 38 |
| Необязательные аргументы конструктора | 38 |
| Необязательные вызовы set-методов | 39 |
| Коллекции сервисов | 40 |
| Вызов нескольких методов | 42 |
| Лучшее из двух миров | 42 |
| Метки сервисов (tags) | 43 |
| Вызов одного метода | 44 |
| Замена аргумента конструктора | 45 |
| Передаём ID сервисов вместо референсов | 45 |
| Делегирование создания | 47 |
| Не очень полезно... | 47 |
| Иногда всё-таки полезно... | 48 |
| Создание сервисов вручную | 49 |
| Определение | 49 |
| Аргументы | 50 |
| Таги | 51 |
| Алиасы (псевдонимы) | 51 |
| Класс Configuration | 51 |
| Динамическое добавление тегов | 53 |
| Используем паттерн Стратегия для загрузки сервисов | 55 |
| Загрузка и конфигурирование дополнительных сервисов | 57 |
| Подчищаем класс конфигурации | 58 |
| Конфигурируем сервис, который будем использовать | 59 |
| Полностью динамическое определение сервисов | 61 |
| Приёмы создания параметров | 63 |
| Файл parameters.yml | 63 |
| Определение и загрузка параметров | 64 |
| Параметры для имени класса | 65 |
| Сборка значений параметров вручную | 65 |
| Определяем параметры в расширениях контейнера | 67 |
| Переопределение параметров при помощи компилятора (compiler pass) | 68 |
| III Структура проекта | 70 |
| Организация слоёв приложения | 71 |
| Тонкие контроллеры | 71 |
| Обработчики форм | 72 |
| Доменные менеджеры | 74 |
| События | 76 |
| События уровня хранения (persistence) | 79 |
| Состояния и контекст | 82 |

ОГЛАВЛЕНИЕ

| | |
|---|-----------|
| Контекст безопасности | 82 |
| Запрос | 85 |
| Избегаем зависимостей от текущего запроса | 87 |
| Используем слушатель (event listener) | 88 |
| Представлять объект запроса во время выполнения | 88 |
| Использование только нужных значений | 89 |
| IV Соглашения по конфигурированию | 90 |
| Настройка конфигурации приложения | 91 |
| Локальные конфигурационные файлы | 92 |
| Храните parameters.yml | 93 |
| Добавьте default_parameters.yml | 93 |
| Соглашения по конфигурированию | 95 |
| Маршрутизатор | 95 |
| Правила именования маршрутов | 96 |
| Сервисы | 96 |
| Метаданные Doctrine | 97 |
| V Безопасность | 98 |
| Введение | 99 |
| Symfony и безопасность | 99 |
| Цели: предотвращение и ограничение | 100 |
| Минимизация урона | 101 |
| Оценка мер по обеспечению безопасности | 101 |
| Перед тем как мы начнём... | 102 |
| Аутентификация и сессии | 103 |
| Инвалидация сессии | 103 |
| Угон сессии | 103 |
| Долгоиграющие сессии | 104 |
| Дизайн контроллеров | 107 |
| Защита действий в контроллерах | 108 |
| Размещение контроллеров за файрволлом | 109 |
| Валидация входных данных | 110 |
| Безопасные формы | 110 |
| HTML5-валидация | 110 |
| Ограничения валидатора | 111 |
| Формы без сущности | 111 |
| Валидация значений из запроса | 114 |
| Атрибуты запроса | 115 |
| Параметры маршрута | 115 |
| Query (GET) и request (POST) параметры | 115 |
| Используем ParamFetcher | 117 |
| Очистка HTML | 118 |
| Автоматизация очистки | 119 |
| Экранирование вывода | 120 |
| Twig | 120 |

ОГЛАВЛЕНИЕ

| | |
|--|------------|
| Контекст экранирования | 120 |
| Экранирование вывода функций | 120 |
| Экранирование аргументов функций | 121 |
| Опасности raw фильтра | 122 |
| Быть скрытым | 124 |
| Маскируйте ошибки аутентификации | 124 |
| Предотвращайте отображение исключений | 125 |
| Настройте страницы ошибок | 125 |
| Не сообщайте ничего определённого о конфиденциальных данных | 126 |
| VI Используем аннотации | 128 |
| Введение | 129 |
| Аннотация - это простой Объект-Значение (Value Object) | 131 |
| Добавляем атрибуты к вашим аннотациям | 133 |
| Передача атрибутов через конструктор | 134 |
| Заполнение публичных свойств указанными атрибутами | 134 |
| Валидация при помощи @Attributes | 135 |
| Валидация при помощи аннотаций @var и @Required | 135 |
| Ограничения на использование аннотации | 136 |
| Когда стоит использовать аннотации | 138 |
| Загрузка конфигураций | 138 |
| Контроль процесса выполнения приложения | 139 |
| Используем аннотации в вашем Symfony-приложении | 141 |
| Реагируем на атрибуты запроса (Request): аннотация @Referrer | 141 |
| Предотвращаем выполнение контроллера: аннотация @RequiresCredits | 144 |
| Модифицируем ответ: аннотация @DownloadAs | 148 |
| Проектирование для повторного использования | 152 |
| Заключение | 154 |
| VII Быть Symfony разработчиком | 155 |
| Повторно используемый код должен иметь слабую связность | 156 |
| Разделяйте код копании от кода продукта | 156 |
| Разделяйте код на “библиотечный” и “банковский” | 157 |
| Уменьшайте связность с фреймворком | 158 |
| Слушатели событий (listeners) вместо подписчиков (subscribers) | 158 |
| Аргументы конструктора вместо получения параметров из контейнера | 159 |
| Аргументы конструктора вместо получения сервисов из контейнера | 159 |
| О производительности | 161 |
| Контроллеры, не зависящие от фреймворка | 161 |
| Тонкие команды | 162 |
| Окружение | 163 |
| Повторно используемый код должен быть легко переносимым | 165 |
| Управление зависимостями и контроль версий | 165 |
| Package repositories | 166 |
| Hard-coded storage layer | 166 |
| Auto-mapped entities | 166 |

ОГЛАВЛЕНИЕ

| | |
|---|------------|
| Storage-agnostic models | 167 |
| Object managers | 168 |
| 25.3 Hard-coded filesystem references | 168 |
| Using the filesystem | 169 |
| Повторно используемый код должен быть открыт для расширения | 170 |
| Configurable behavior | 170 |
| Everything should be replaceable | 170 |
| Use lots of interfaces | 170 |
| Use the bundle configuration to replace services | 172 |
| Add extension points | 172 |
| Service tags | 172 |
| Events | 173 |
| Повторно используемый код должен быть легко используемым | 174 |
| Add documentation | 174 |
| Throw helpful exceptions | 174 |
| Use specific exception classes | 174 |
| Set detailed and friendly messages | 175 |
| Повторно используемый код должен быть надёжным | 176 |
| Add enough tests | 176 |
| Test your bundle extension and configuration | 176 |
| Заключение | 179 |

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 об этой книге я узнал случайно, из какой-то рассылки, как раз тогда, когда она стала бесплатной. Да, она про Symfony2, но описывает и более общие принципы разработки, нежели просто версию одного фреймворка, так что, я полагаю, эта книга ещё 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 разработчикам термины, я не стал переводить и просто транслитерировал, например: фреймворк, бандл, файрволл. 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)
- firewall - файрволл или брандмауэр

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 сверять написанное в книге с текущей версией Symfony (на декабрь 2016 - это 3.2 + 2.8LTS), 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 при необходимости 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 понравятся.

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, которая освещала основные 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 Разработчика 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. В другой раз такое ощущение у меня было, когда я читал Kris Wallsmith - разъяснения о том, как работает Symfony Security Component. Я считаю, что эта книга - “Один год с Symfony” - должна быть в арсенале каждого разработчика, 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 найду их здесь 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 одним 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 экзаменацационной 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 глубокими 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 Symfony 2 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 разработчиков, 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 Symfony 2, 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, иногда я немного 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 его промежуточных шагов.

Следующая часть называется “Шаблоны внедрения зависимостей”. Она содержит коллекцию шаблонов, которые являются решениями проблем, периодически возникающих в приложении при 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. Также, я надеюсь, этот раздел воодушевит вас на использование некоторых полезных 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 невозможно. Вы всегда должны быть начеку и заботиться о безопасности вашего приложения. Эта часть книги содержит различные советы о том, как обеспечить безопасность приложения, на что обратить внимание, когда вы можете положиться на фреймворк и когда вам нужно контролировать безопасность самостоятельно.

Шестая целиком будет посвящена 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, на самом деле, эта часть будет вдохновлять вас 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 другие идеи, которые могут сделать ваш код красивым, чистым и дружелюбным к другим проектам.

Наслаждайтесь!

I От запроса до ответа

HttpKernelInterface

Symfony знаменит благодаря своему HttpKernelInterface:

```

1  namespace Symfony\Component\HttpKernel;
2
3  use Symfony\Component\HttpFoundation\Request;
4  use Symfony\Component\HttpFoundation\Response;
5
6  interface HttpKernelInterface
7  {
8      const MASTER_REQUEST = 1;
9      const SUB_REQUEST = 2;
10
11     /**
12      * @return Response
13     */
14    public function handle(
15        Request $request,
16        $type = self::MASTER_REQUEST,
17        $catch = true
18    );
19 }
```

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:

```

1 // /web/app.php
2 $kernel = new AppKernel('prod', false);
3 // ...
4 $request = Request::createFromGlobals();
5 $response = $kernel->handle($request);
6 $response->send();
7 // ...
```

Сначала создаётся экземпляр ядра `AppKernel`. Это класс - специфичный для вашего приложения и вы можете найти его в директории `app/AppKernel.php`. Он позволяет регистрировать ваши бандлы и изменять некоторые основные настройки, такие как расположение директории с кэшем, или указать, 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`:

```
1 public function registerContainerConfiguration(LoaderInterface $loader)
2 {
3     $loader
4         ->load(__DIR__.'/config/config_'. $this->getEnvironment(). '.yml');
5 }
```

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 сопутствующей 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 представляет объектно-ориентированный интерфейс к этим суперглобальным массивам. Все эти классы также обрабатывают различные проблемные ситуации, которые могут возникать при использовании разных версий PHP или же разных платформ. В контексте Symfony всегда разумнее вместо суперглобальных переменных использовать объект Request для получения различных данных о запросе.

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.

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:

```

1 // Symfony\Component\HttpKernel\Kernel
2
3 public function handle(
4     Request $request,
5     $type = HttpKernelInterface::MASTER_REQUEST,
6     $catch = true
7 ) {
8     if (false === $this->booted) {
9         $this->boot();
10    }
11
12    return $this->getHttpKernel()->handle($request, $type, $catch);
13 }

```

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

Среди Symfony-разработчиков бандлы прежде всего известны как место, где размещается разрабатываемый вами код. Каждый бандл должен иметь имя, которое 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:

```

1 class AppKernel extends Kernel
2 {
3     public function registerBundles()
4     {
5         $bundles = array(
6             new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
7             // ... тут прочие бандлы
8             new Matthias\BlogBundle()
9         );
10
11         return $bundles;
12     }
13 }

```

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 `{наименование бандла}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:

```

1 namespace Matthias\BlogBundle\DependencyInjection;
2
3 use Symfony\Component\HttpKernel\DependencyInjection\Extension;
4 use Symfony\Component\Config\FileLocator;
5 use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
6
7 class MatthiasBlogExtension extends Extension
8 {
9     public function load(array $configs, ContainerBuilder $container)
10    {
11        $loader = new XmlFileLoader($container,
12            new FileLocator(__DIR__ . '/../Resources/config'));
13
14        // добавляем определения сервисов в контейнер
15        $loader->load('services.xml');
16
17        $processedConfig = $this->processConfiguration(
18            new Configuration(),
19            $configs
20        );
21
22        // добавляем параметр
23        $container->setParameter(
24            'matthias_blog.comments_enabled',
25            $processedConfig['enable_comments']
26        );
27    }
28
29    public function getAlias()
30    {
31        return 'matthias_blog';
32    }
33 }

```

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

```

1 matthias_blog:
2     enable_comments: true

```

Подробнее о конфигурировании бандлов вы узнаете в следующем разделе - [Приёмы 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, который называется “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:

```

1 <service id="event_dispatcher" class="...\\ContainerAwareEventDispatcher">
2   <argument type="service" id="service_container"/>
3     <call method="addListenerService">
4       <argument>kernel.controller</argument>
5         <!-- ... прочие аргументы, если нужно -->
6     </call>
7     <!-- ... прочие параметры сервиса, если нужно -->
8   </service>
```

PHP файл содержит метод для каждого сервиса, который может быть запрошен. Всю логику создания сервисов, такую как аргументы контроллеров, 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:

```

1 class appDevDebugProjectContainer extends Container
2 {
3   // ...
4
5   protected function getEventDispatcherService()
6   {
7     $this->services['event_dispatcher'] =
8       $instance = new ContainerAwareEventDispatcher($this);
9
10    $instance->addListenerService('kernel.controller', ...);
11
12    // ...
13
14    return $instance;
15  }
16
17  //...
18}
```

От Kernel до HttpKernel

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

```
1 // Symfony\Component\HttpKernel\Kernel
2
3 public function handle(
4     Request $request,
5     $type = HttpKernelInterface::MASTER_REQUEST,
6     $catch = true
7 ) {
8     if (false === $this->booted) {
9         $this->boot();
10    }
11
12    return $this->getHttpKernel()->handle($request, $type, $catch);
13 }
```

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

```
1 public function handle(
2     Request $request,
3     $type = HttpKernelInterface::MASTER_REQUEST,
4     $catch = true
5 ) {
6     try {
7         return $this->handleRaw($request, $type);
8     } catch (\Exception $e) {
9         if (false === $catch) {
10             throw $e;
11         }
12
13         return $this->handleException($e, $request, $type);
14     }
15 }
```

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 перехвачено. HttpKernel постараётся найти кого-то, кто сможет создать объект Response для (см. также главу [Обработка исключений](#)).

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 детерминированным (@dbykadorov: т.е. допускает отклонения и изменения в процессе). Это означает, что у вас есть несколько различных способов для внедрения в этот процесс, путём которого вы можете полностью заменить или частично модифицировать ответ на промежуточных шагах его формирования.

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:

```

1 use Symfony\Component\HttpKernel\Event\GetResponseEvent;
2
3 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
4 {
5     $event = new GetResponseEvent($this, $request, $type);
6     $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);
7
8     if ($event->hasResponse()) {
9         return $this->filterResponse(
10             $event->getResponse(),
11             $request,
12             $type
13         );
14     }
15
16     // ...
17 }
```

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

```

1 use Symfony\Component\HttpFoundation\Response;
2 use Symfony\Component\HttpKernel\Event\GetResponseEvent;
3
4 class MaintenanceModeListener
5 {
6     public function onKernelRequest(GetResponseEvent $event)
7     {
8         $response = new Response(
9             'This site is temporarily unavailable',
10            503
11        );
12
13        $event->setResponse($response);
14    }
15 }
```

Регистрация слушателей 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 для него сервис и добавить ему тэг `kernel.event_listener` или `kernel.event_subscriber` (в случае, если вы хотите реализовать интерфейс `EventSubscriberInterface`).

```
1 <service id="..." class="...">
2   <tag name="kernel.event_listener"
3     event="kernel.request"
4     method="onKernelRequest" />
5 </service>
```

Или:

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

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

```
1 <service id="..." class="...">
2   <tag name="kernel.event_listener"
3     event="kernel.request"
4     method="onKernelRequest"
5     priority="100" />
6 </service>
```

Чем выше приоритет, 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, выполняющие 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 обработки запроса: `RouterListener` и `Firewall`. Слушатель `RouterListener` получает информацию о запрошенном пути из запроса `Request` и пытается сопоставить этот путь 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:

```

1 namespace Symfony\Component\HttpKernel\EventListener;
2
3 class RouterListener implements EventSubscriberInterface
4 {
5     public function onKernelRequest(GetResponseEvent $event)
6     {
7         $request = $event->getRequest();
8
9         $parameters = $this->matcher->match($request->getPathInfo());
10
11        // ...
12
13        $request->attributes->add($parameters);
14    }
15 }

```

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

```

1 _demo_hello:
2     path: /demo/hello/{name}
3     defaults:
4         _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:

```

1 array(
2     '_route' => '_demo_hello',
3     '_controller' => 'AcmeDemoBundle:Demo:hello',
4     'name' => 'World'
5 );

```

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, хотя должен был быть, так как запрашивается защищённая страница. `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-аутентификации.

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:

```

1 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
2 {
3     // событие "kernel.request"
4     ...
5
6     if (false === $controller = $this->resolver->getController($request)) {
7         throw new NotFoundHttpException();
8     }
9 }
```

Резолвер является экземпляром класса, реализующего интерфейс `ControllerResolverInterface`:

```

1 namespace Symfony\Component\HttpKernel\Controller;
2
3 use Symfony\Component\HttpFoundation\Request;
4
5 interface ControllerResolverInterface
6 {
7     public function getController(Request $request);
8
9     public function getArguments(Request $request, $controller);
10 }
```

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:

```

1 public function getController(Request $request)
2 {
3     if (!$controller = $request->attributes->get('_controller')) {
4         return false;
5     }
6
7     ...
8
9     $callable = $this->createController($controller);
10 ...
11 ...
12
13     return $callable;
14 }
```

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.

... ВОТ ЭТО 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 представлены 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.

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`):

```
1 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
2
3 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
4 {
5     // событие "kernel.request"
6     // определяем контроллер при помощи controller resolver
7     ...
8
9     $event = new FilterControllerEvent($this, $controller, $request, $type);
10    $this->dispatcher->dispatch(KernelEvents::CONTROLLER, $event);
11    $controller = $event->getController();
12 }
```

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

```

1 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
2
3 class ControllerListener
4 {
5     public function onKernelController(FilterControllerEvent $event)
6     {
7         $event->setController(...);
8     }
9 }

```

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 не захотеть, чтобы прочие слушатели этого события, которые будут вызваны после вашего смогли бы провернуть тот же трюк. Вы можете это сделать, вызвав метод события:

```
1 $event->stopPropagation();
```

Также удостоверьтесь, что ваш слушатель имеет более высокий приоритет и будет вызван первым. См. также [Регистрация слушателей событий](#).

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 именно: 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 бандла умеет конвертировать аргументы контроллера, например, загружать сущность (entity) по параметру `id`, определённому в маршруте:

```

1 /**
2 * @Route("/post/{id}")
3 */
4 public function showAction(Post $post)
5 {
6     ...
7 }

```

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

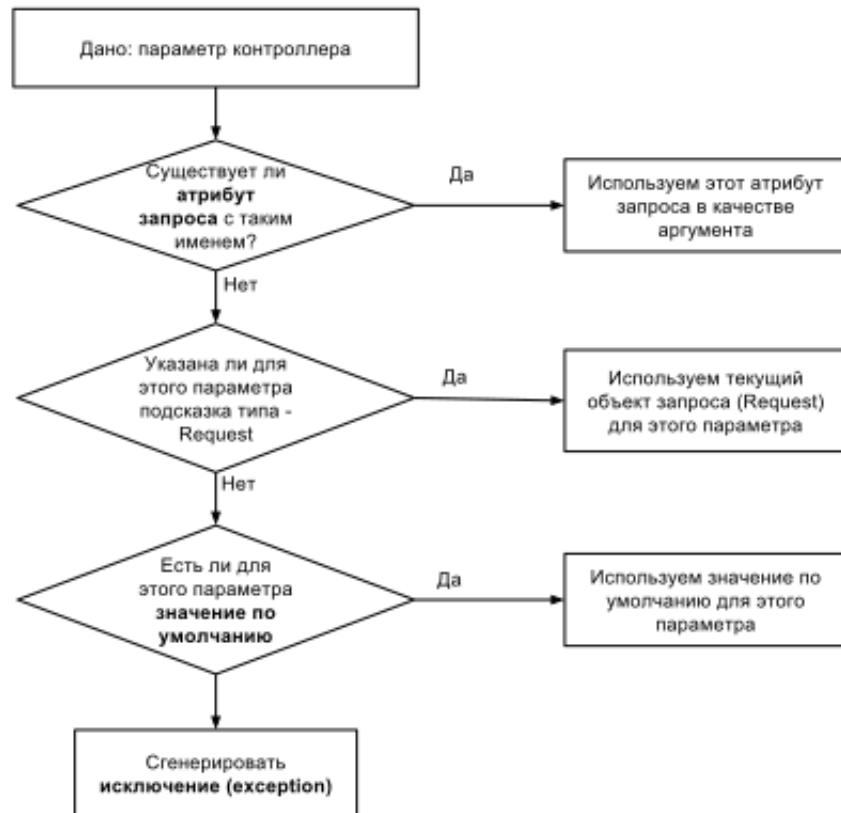
интерфейс `ParamConverterInterface`, создать определение сервиса для него и присвоить ему таг `request.param_converter`. См. также документацию к [@ParamConverter](#).

Сбор аргументов для выполнения контроллера

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

```
1 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
2 {
3     // событие "kernel.request"
4     // определяем контроллер при помощи controller resolver
5     // событие "kernel.controller"
6     ...
7
8     $arguments = $this->resolver->getArguments($request, $controller);
9 }
```

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'a

Выполнение контроллера

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

```

1 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
2 {
3     // событие "kernel.request"
4     // определяем контроллер при помощи controller resolver
5     // событие "kernel.controller"
6     // используем controller resolver для того, чтобы получить аргументы контроллера
7     ...
8
9     $response = call_user_func_array($controller, $arguments);
10
11    if (!$response instanceof Response) {
12        ...
13    }
14 }
  
```

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.

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:

```

1 class SomeController
2 {
3     public function simpleAction()
4     {
5         return new Response(
6             '<html><body><p>Старый добрый HTML</p></body></html>';
7         );
8     }
9 }
```

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, который 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):

```

1 use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
2
3 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
4 {
5     // событие "kernel.request"
6     // определяем контроллер при помощи controller resolver
7     // событие "kernel.controller"
8     // используем controller resolver для того, чтобы получить аргументы контроллера
9     // исполняем контроллер
10    ...
11
12    $event = new GetResponseForControllerResultEvent(
13        $this,
14        $request,
15        $type,
16        $response
17    );
18    $this->dispatcher->dispatch(KernelEvents::VIEW, $event);
19
20    if ($event->hasResponse()) {
21        $response = $event->getResponse();
22    }
23
24    if (!$response instanceof Response) {
25        // тут яду 00000ЧЧЕНЬ нужен ответ...
26
27        throw new \LogicException(...);
28    }
29}
```

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

```

1 use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
2
3 class ViewListener
4 {
5     public function onKernelView(GetResponseForControllerResultEvent $event)
6     {
7         $response = new Response(...);
8
9         $event->setResponse($response);
10    }
11 }

```

Примечательные слушатели события kernel.view

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

```

1 public function onKernelView(GetResponseForControllerResultEvent $event)
2 {
3     $parameters = $event->getControllerResult();
4
5     // получаем движок шаблонизатора
6     $templating = ...;
7
8     $event->setResponse(
9         $templating->renderResponse($template, $parameters)
10    );
11 }

```

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`):

```

1 private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
2 {
3     // событие "kernel.request"
4     // определяем контроллер при помощи controller resolver
5     // событие "kernel.controller"
6     // используем controller resolver для того, чтобы получить аргументы контроллера
7     // конвертируем результат, который вернул запрос в объект Response
8
9     return $this->filterResponse($response, $request, $type);
10 }
11
12 private function filterResponse(Response $response, Request $request, $type)
13 {
14     $event = new FilterResponseEvent($this, $request, $type, $response);
15
16     $this->dispatcher->dispatch(KernelEvents::RESPONSE, $event);
17
18     return $event->getResponse();
19 }

```

Слушатели события могут модифицировать объект ответа `Response` и даже полностью его заменить:

```
1 class ResponseListener
2 {
3     public function onKernelResponse(FilterResponseEvent $event)
4     {
5         $response = $event->getResponse();
6
7         $response->headers->set('X-Framework', 'Symfony2');
8
9         // or
10
11         $event->setResponse(new Response(...));
12     }
13 }
```

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 безопасности в сессии. Это позволяет ускорить процесс аутентификации при следующем запросе. В компонент `Security Component` также входит слушатель `ResponseListener`, который устанавливает cookie, содержащий информацию о `remember-me`. Содержимое этого cookie может быть использовано для авто-логина пользователя, даже если оригинальная сессия уже была завершена.

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:

```

1 public function handle(
2     Request $request,
3     $type = HttpKernelInterface::MASTER_REQUEST,
4     $catch = true
5 ) {
6     try {
7         return $this->handleRaw($request, $type);
8     } catch (\Exception $e) {
9         if ($false === $catch) {
10             throw $e;
11         }
12
13         return $this->handleException($e, $request, $type);
14     }
15 }
```

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`.

```

1 use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
2
3 private function handleException(\Exception $e, $request, $type)
4 {
5     $event = new GetResponseForExceptionEvent($this, $request, $type, $e);
6     $this->dispatcher->dispatch(KernelEvents::EXCEPTION, $event);
7
8     // a listener might have replaced the exception
9     $e = $event->getException();
10
11    if (!$event->hasResponse()) {
12        throw $e;
13    }
14
15    $response = $event->getResponse();
16
17    ...
18 }
```

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 настройках 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:

```

1 // проверяем наличие специфического статус-кода
2 if ($response->headers->has('X-Status-Code')) {
3     $response->setStatusCode($response->headers->get('X-Status-Code'));
4
5     $response->headers->remove('X-Status-Code');
6 } elseif (
7     !$response->isClientError()
8     && !$response->isServerError()
9     && !$response->isRedirect()
10 ) {
11     // убеждаемся, что у нас действительно есть ответ
12     if ($e instanceof HttpExceptionInterface) {
13         // сохраняем HTTP статус-код и заголовки
14         $response->setStatusCode($e->getStatusCode());
15         $response->headers->add($e->getHeaders());
16     } else {
17         $response->setStatusCode(500);
18     }
19 }
```

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 генерировано другое исключение, это исключение будет проигнорировано и клиенту будет отправлен неотфильтрованный ответ:

```

1 try {
2     return $this->filterResponse($response, $request, $type);
3 } catch (\Exception $e) {
4     return $response;
5 }
```

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`:

```
1 twig:  
2     # points to Symfony\Bundle\TwigBundle\Controller\ExceptionController  
3     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 перelogинить пользователя (например, если тот использовал `remember me` опцию при логине), если же это не выходит, предоставляет возможность `access denied handler` разобраться с возникшей ситуацией.

Подзапросы

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

```

1 public function handle(
2     Request $request,
3     $type = HttpKernelInterface::MASTER_REQUEST,
4     $catch = true
5 ) {
6     ...
7 }
```

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

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

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`):

```
1 $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 будет ничего делать в случае, если запрос имеет тип подзапроса:

```

1 public function onKernelRequest(GetResponseEvent $event)
2 {
3     if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
4         return;
5     }
6     ...
7 }
8 }
```

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:

```
1 public function onKernelException(GetResponseForExceptionEvent $event)
2 {
3     $request = $event->getRequest();
4
5     ...
6
7     $request = $request->duplicate(null, null, $attributes);
8     $request->setMethod('GET');
9
10    $response = $event
11        ->getKernel()
12        ->handle($request, HttpKernelInterface::SUB_REQUEST, true);
13
14    $event->setResponse($response);
15 }
```

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

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

Когда вы пишете 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

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 бандле 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 определения сервисов и далее в этой главе я расскажу о большинстве из них. Зная все возможности, вы можете сделать правильный выбор, применительно к конкретно взятой ситуации.

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.

Приёмы создания сервисов

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

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:

```

1 class TokenProvider
2 {
3     private $storage;
4
5     public function __construct(TokenStorageInterface $storage)
6     {
7         $this->storage = $storage;
8     }
9 }
```

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

```

1 <service id="token_provider" class="TokenProvider">
2     <argument type="service" id="token_storage" />
3 </service>
4
5 <service id="token_storage" class="...">
6 </service>
```

Первый аргумент сервиса `token_provider` - это ссылка на сервис `token_storage`. Класс хранилища токенов, 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`:

```

1 class ExpiringTokenProvider extends TokenProvider
2 {
3     private $lifetime;
4
5     public function __construct(TokenStorageInterface $storage, $lifetime)
6     {
7         $this->lifetime = $lifetime;
8
9         parent::__construct($storage);
10    }
11 }

```

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

```

1 <service id="expiring_token_provider" class="ExpiringTokenProvider">
2     <argument type="service" id="token_storage" />
3     <argument>3600</argument><!-- lifetime -->
4 </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:

```

1 <service id="abstract_token_provider" abstract="true">
2     <argument type="service" id="token_storage" />
3 </service>
4
5 <service id="token_provider" class="TokenProvider"
6     parent="abstract_token_provider">
7 </service>
8
9 <service id="expiring_token_provider" class="ExpiringTokenProvider"
10    parent="abstract_token_provider">
11    <argument>3600</argument><!-- lifetime -->
12 </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`, а 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](#) (недостатки 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 ситуацию, когда вы не захотите (или не сможете) переопределить конструктор сервиса и, следовательно, не сможете добавить дополнительные параметры в него, или же, возможно, некоторые зависимости ещё 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):

```

1 class SomeController
2 {
3     private $container;
4
5     public function setContainer(ContainerInterface $container)
6     {
7         $this->container = $container;
8     }
9
10    public function indexAction()
11    {
12        $service = $this->container->get('...');
13    }
14 }
```

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:

```

1 <service id="some_controller" class="SomeController">
2     <call method="setContainer">
3         <argument type="service" id="service_container" />
4     </call>
5 </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 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 приведёт к вызову метода `get()` от `null`, что, в свою очередь, вызовет фатальную ошибку PHP. По моему мнению, 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 в 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:

```

1 class SomeController
2 {
3     public function indexAction()
4     {
5         $service = $this->getContainer()->get('...');
6     }
7
8     private function getContainer()
9     {
10        if (!($this->container instanceof ContainerInterface)) {
11            throw new \RuntimeException('Service container is missing');
12        }
13
14        return $this->container;
15    }
16 }
```

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 [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:

```
1 <service id="abstract_container_aware" abstract="true">
2   <call method="setContainer">
3     <argument type="service" id="service_container" />
4   </call>
5 </service>
6
7 <service id="some_controller" class="SomeController"
8   parent="abstract_container_aware">
9 </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 таким, присвоив признаку `abstract` значение `true` и добавьте префикс `abstract_` к его `id` (по аналогии с абстрактными классами, имена которых традиционно начинаются с `Abstract`).

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

Необязательные (опциональные) зависимости

Иногда зависимости являются не обязательными. Термин “не обязательные зависимости” может вам показаться противоречивым, так как если вы реально от чего-то не зависите, странно называть это “зависимостями”. Однако, встречаются ситуации, когда один сервис знает как использовать другой сервис, но в общем-то этот другой сервис не является необходимым, 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:

```

1 use Symfony\Component\EventDispatcher\EventDispatcherInterface;
2 use Psr\Log\LoggerInterface;
3
4 class AuthenticationListener
{
5     private $eventDispatcher;
6     private $logger;
7
8     public function __construct(
9         EventDispatcherInterface $eventDispatcher,
10        LoggerInterface $logger = null
11    ) {
12        $this->eventDispatcher = $eventDispatcher;
13        $this->logger = $logger;
14    }
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 вы можете выбрать стратегию поведения в случае их отсутствия:

```

1 <service id="authentication_listener" class="AuthenticationListener">
2     <argument type="service" id="logger" on-invalid="ignore" />
3 </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:

```

1 if ($this->logger instanceof LoggerInterface) {
2     ...
3 }

```

Это более надёжный способ проверки, чем сравнение с `NULL`:

```

1 if ($this->logger !== null) {
2     ...
3 }

```

Думайте об этом в таком ключе: если что-то не является `NULL`, можем ли мы быть уверены, что это `logger`?

Необязательные 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 захламлять сигнатуру конструктора:

```

1 class AuthenticationListener
2 {
3     private $eventDispatcher;
4     private $logger;
5
6     public function __construct(EventDispatcherInterface $eventDispatcher)
7     {
8         $this->eventDispatcher = $eventDispatcher;
9     }
10
11    public function setLogger(LoggerInterface $logger = null)
12    {
13        $this->logger = $logger;
14    }
15 }

```

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):

```

1 <service id="authentication_listener" class="AuthenticationListener">
2     <call method="setLogger">
3         <argument type="service" id="logger" on-invalid="ignore" />
4     </call>
5 </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 true-ООП стиле, у вас 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 помощников для сервисов более высокого уровня. Чтобы не допустить возможности использования низкоуровневых сервисов в других частях приложения, например, так:

```
1 $container->get('low_level_service_id');
```

вы должны пометить их как закрытые (non-public), для этого в определении низкоуровневого сервиса нужно присвоить атрибуту `public` значение “`false`”:

```
1 <service id="low_level_service_id" class="..." public="false">
2 </service>
```

Коллекции сервисов

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

```

1  class ObjectRenderer
2  {
3      private $renderers;
4
5      public function __construct(array $renderers)
6      {
7          $this->renderers = $renderers;
8      }
9
10     public function render($object)
11     {
12         foreach ($this->renderers as $renderer) {
13             if ($renderer->supports($object)) {
14                 return $renderer->render($object);
15             }
16         }
17     }
18 }

```

Определение такого сервиса может выглядеть следующим образом:

```

1 <service id="object_renderer" class="ObjectRenderer">
2     <argument type="collection">
3         <argument type="service" id="domain_object_renderer" />
4         <argument type="service" id="user_renderer" />
5     </argument>
6 </service>

```

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

```

1 array(
2     0 => ...
3     1 => ...
4 )

```

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

```

1 <service id="object_renderer" class="ObjectRenderer">
2     <argument type="collection">
3         <argument
4             key="domain_object" type="service"
5             id="domain_object_renderer" />
6         <argument
7             key="user" type="service"
8             id="user_renderer" />
9     </argument>
10 </service>

```

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

```

1 array(
2     'domain_object' => ...
3     'user' => ...
4 )

```

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:

```

1 class ObjectRenderer
2 {
3     private $renderers;
4
5     public function __construct()
6     {
7         $this->renderers = array();
8     }
9
10    public function addRenderer($name, RendererInterface $renderer)
11    {
12        $this->renderers[$name] = $renderer;
13    }
14 }

```

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()`:

```

1 <service id="object_renderer" class="ObjectRenderer">
2     <call method="addRenderer">
3         <argument>domain_object</argument>
4         <argument type="service" id="domain_object_renderer" />
5     </call>
6     <call method="addRenderer">
7         <argument>user</argument>
8         <argument type="service" id="user_renderer" />
9     </call>
10    </service>

```

Лучшее из двух миров

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

```

1  class ObjectRenderer
2  {
3      private $renderers;
4
5      public function __construct(array $renderers)
6      {
7          foreach ($renderers as $name => $renderer) {
8              $this->addRenderer($name, $renderer);
9          }
10     }
11
12     public function addRenderer($name, RendererInterface $renderer)
13     {
14         $this->renderers[$name] = $renderer;
15     }
16 }

```

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:

```

1 <!-- in some other bundle -->
2 <service id="date_time_renderer" class="DateTimeRenderer">
3     <tag name="specific_renderer" alias="date_time" />
4 </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:

```

1 namespace Matthias\RendererBundle\DependencyInjection\Compiler;
2
3 use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
4 use Symfony\Component\DependencyInjection\ContainerBuilder;
5 use Symfony\Component\DependencyInjection\Reference;
6
7 class RenderersPass implements CompilerPassInterface
8 {
9     public function process(ContainerBuilder $container)
10    {
11        // получаем все сервисы, отмеченные определённым тагом из всего проекта
12        $taggedServiceIds
13            = $container ->findTaggedServiceIds('specific_renderer');
14
15        $objectRendererDefinition
16            = $container->getDefinition('object_renderer');
17
18        foreach ($taggedServiceIds as $serviceId => $tags) {
19
20            // сервисы могут иметь более одного тага с одним и тем же именем
21            foreach ($tags as $tagAttributes) {
22
23                // вызываем метод addRenderer() для того, чтобы зарегистрировать рендерер

```

```

24     $objectRendererDefinition
25         ->addMethodCall('addRenderer', array(
26             $tagAttributes['alias'],
27             new Reference($serviceId),
28         ));
29     }
30 }
31 }
32 }
```

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

```

1 use Matthias\RendererBundle\DependencyInjection\Compiler\RenderersPass;
2 use Symfony\Component\DependencyInjection\ContainerBuilder;
3
4 class RendererBundle extends Bundle
5 {
6     public function build(ContainerBuilder $container)
7     {
8         $container->addCompilerPass(new RenderersPass());
9     }
10 }
```

Метод `process()` класса `RenderersPass` в первую очередь получает все сервисы с тагами, которые имеют имя `specific_renderer`. В результате будет получен массив, ключами которого будут id сервисов, а значениями - массив их атрибутов. Так сделано потому, что определение любого сервиса может иметь более одного тага 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()`.

Вместе это означает, что когда сервис `object_renderer` будет запрошен, сначала будет создан экземпляр класса `ObjectRenderer`. 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()`:

```

1 class RenderersPass implements CompilerPassInterface
2 {
3     public function process(ContainerBuilder $container)
4     {
5         $taggedServiceIds = ...;
6
7         $objectRendererDefinition = ...;
8
9         $renderers = array();
10
11        foreach ($taggedServiceIds as $serviceId => $tags) {
12            foreach ($tags as $tagAttributes) {
13                $name = $tagAttributes['alias'];
14                $renderer = new Reference($serviceId);
15                $renderers[$name] = $renderer;
16            }
17        }
18
19        $objectRendererDefinition
20            ->addMethodCall('setRenderers', array($renderers));
21    }
22 }

```

Замена аргумента конструктора

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

```

1 class RenderersPass implements CompilerPassInterface
2 {
3     public function process(ContainerBuilder $container)
4     {
5         $taggedServiceIds = ...;
6
7         $objectRendererDefinition = ...;
8
9         $renderers = array();
10
11        // получаем референсы на сервисы
12        ...
13
14        $objectRendererDefinition->replaceArgument(0, $renderers);
15    }
16 }

```

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

```

1 <service id="object_renderer" class="ObjectRenderer">
2     <argument /><!-- наши рендереры -->
3 </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, возможно, 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:

```

1  class LazyLoadingObjectRenderer
2  {
3      private $container;
4      private $renderers;
5
6      public function __construct(ContainerInterface $container)
7      {
8          $this->container = $container;
9      }
10
11     public function addRenderer($name, $renderer)
12     {
13         $this->renderers[$name] = $renderer;
14     }
15
16     public function render($object)
17     {
18         foreach ($this->renderers as $name => $renderer) {
19             if (is_string($renderer)) {
20                 // здесь $renderer - это id сервиса, строка
21                 $renderer = $this->container->get($renderer);
22             }
23
24             // проверяем, является ли renderer экземпляром RendererInterface
25             ...
26         }
27     }
28 }
```

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

```

1  class RenderersPass implements CompilerPassInterface
2  {
3      public function process(ContainerBuilder $container)
4      {
5          $taggedServiceIds = ...;
6
7          $objectRendererDefinition = ...;
8
9          foreach ($taggedServiceIds as $serviceId => $tags) {
10              foreach ($tags as $tagAttributes) {
11                  $objectRendererDefinition
12                      ->addMethodCall('addRenderer', array(
13                          $tagAttributes['alias'],
14                          $serviceId,
15                      )
16                  );
17              }
18          }
19      }
20 }
```

Также не забудьте указать сервисный контейнер в качестве аргумента конструктора:

```

1 <service id="object_renderer" class="LazyLoadingObjectRenderer">
2   <argument type="service" id="service_container" />
3 </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 разделами [Уменьшаем связность кода с фреймворком](#), особенно [с заметкой о быстродействии](#).

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, делегировав 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:

```

1 <service id="some_service" class="ClassOfResultingObject"
2   factory-class="Some\Factory" factory-method="create">
3   <argument>...</argument>
4 </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 означает вызов фабричного метода у экземпляра фабричного класса. Следовательно, этот экземпляр фабрики должен быть предварительно сам определён как сервис:

```

1 <!-- сервис, создаваемый фабрикой some_factory_service -->
2 <service id="some_service" class="ClassOfResultingObject"
3   factory-service="some_factory_service" factory-method="create">
4   <argument>...</argument>
5 </service>
6
7 <!-- собственно сервис самой фабрики -->
8 <service id="some_factory_service" class="Some\Factory">
9 </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 определению также должна быть статической, что 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 на 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 [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:

```
1 use Doctrine\ORM\EntityManager;
2
3 class SomeClass
4 {
5     public function __construct(EntityManager $entityManager)
6     {
7         $this->entityManager = $entityManager;
8     }
9
10    public function doSomething()
11    {
12        $repository = $this->entityManager->getRepository('User');
13
14        ...
15    }
16 }
```

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

```

1 class SomeClass
2 {
3     public function __construct(UserRepository $userRepository)
4     {
5         $this->userRepository = $userRepository;
6     }
7 }

```

Вот соответствующие определения сервисов для этого случая:

```

1 <service id="some_service" class="SomeClass">
2     <argument type="user_repository" />
3 </service>
4
5 <service id="user_repository" class="UserRepository"
6     factory-service="entity_manager" factory-method="getRepository">
7     <argument>User</argument>
8 </service>

```

Если взглянуть на аргументы конструктора `SomeClass`, сразу становится ясно, что на входе 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 создавать мок-объекты для менеджера сущности и репозитория, теперь можно создать только один - для репозитория.

Создание сервисов вручную

Обычно вы создаёте сервисы, загружая их определения из файла:

```

1 use Symfony\Component\HttpKernel\DependencyInjection\Extension;
2 use Symfony\Component\Config\FileLocator;
3 use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
4
5 class SomeBundleExtension extends Extension
6 {
7     public function load(array $configs, ContainerBuilder $container)
8     {
9         $locator = new FileLocator(__DIR__ . '/../Resources/config');
10        $loader = new XmlFileLoader($container, $locator);
11        $loader->load('services.xml');
12    }
13 }

```

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) передачу ему имени класса. Определение получит идентификатор при его добавлении в `ContainerBuilder`:

```

1 use Symfony\Component\DependencyInjection\Definition;
2
3 $class = ...; // присваиваем значение имени класса для определения
4
5 $definition = new Definition($class);
6
7 $container->setDefinition('the_service_id', $definition);

```

Эквивалентом этого определения был бы такой XML:

```

1 <service id="the_service_id" class="...">
2 </service>

```

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

```

1 $definition->setPublic(false);

```

Аргументы

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

```

1 use Symfony\Component\DependencyInjection\Reference;
2
3 $definition->setArguments(array(
4     new Reference('logger') // ссылка на другой сервис
5     true // логическое (булево) значение,
6     array(
7         'table_name' => 'users'
8     ) // массив
9     ...
10));

```

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 использованием 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:

```

1 $definition->addArgument(new Reference('logger'));
2 $definition->addArgument(true);
3 ...

```

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

```

1 $definition->setArguments(array(null, null));
2 ...
3 ...
4 ...
5 $definition->replaceArgument(0, new Reference('logger'));
6 $definition->replaceArgument(1, true);

```

Соответствующий XML выглядел бы так:

```

1 <service id="..." class="...">
2   <argument type="service" id="logger" />
3   <argument>true</argument>
4 </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:

```

1 $definition->addTag('kernel.event_listener', array(
2   'event' => 'kernel.request'
3 );
4 $definition->addTag('kernel.event_listener', array(
5   'event' => 'kernel.response'
6 );

```

XML определение в этом случае было бы таким:

```

1 <service id="..." class="...">
2   <tag name="kernel.event_listener" event="kernel.request">
3   <tag name="kernel.event_listener" event="kernel.response">
4 </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:

```

1 $container->setAlias('some_alias', 'some_service_id');

```

Теперь, когда бы вы ни запросили сервис `some_alias`, фактически вы получите сервис `some_service_id`.

Класс Configuration

Прежде чем продолжить, нужно дать несколько пояснений касательно класса `Configuration`. Вы могли обратить на него внимание ранее, а также, возможно, даже создавали 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 использовать всё сказанное ниже в совершенно другом контексте). Имя класса и его пространство имён не имеют особого значения, пока класс реализует интерфейс ConfigurationInterface:

```

1 use Symfony\Component\Config\Definition\ConfigurationInterface;
2 use Symfony\Component\Config\Definition\Builder\TreeBuilder;
3
4 class Configuration implements ConfigurationInterface
5 {
6     public function getConfigTreeBuilder()
7     {
8         $treeBuilder = new TreeBuilder();
9         $rootNode = $treeBuilder->root('name_of_bundle');
10
11         $rootNode
12             ->children()
13             // определяем узлы конфигурации
14             ...
15             ->end()
16     ;
17
18     return $treeBuilder;
19 }
20 }
```

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 с 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 подчёрка. 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:

```

1 $rootNode
2     ->children()
3         ->booleanNode('auto_connect')
4             ->defaultTrue()
5             ->end()
6         ->scalarNode('default_connection')
7             ->defaultValue('default')
8             ->end()
9     ->end()
10 ;
```

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.).

```

1 class MatthiasAccountExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         $processedConfig = $this->processConfiguration(
6             new Configuration(),
7             $configs
8         );
9     }
10 }
```

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:

```

1 final protected function processConfiguration(
2     ConfigurationInterface $configuration,
3     array $configs
4 ) {
5     $processor = new Processor();
6
7     return $processor->processConfiguration($configuration, $configs);
8 }
```

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.

Динамическое добавление тегов

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:

```

1 use Symfony\Component\Config\Definition\ConfigurationInterface;
2
3 class Configuration implements ConfigurationInterface
4 {
5     public function getConfigTreeBuilder()
6     {
7         $treeBuilder = new TreeBuilder();
8         $rootNode = $treeBuilder->root('generic_listener');
9
10        $rootNode
11            ->children()
12                ->arrayNode('events')
13                    ->prototype('scalar')
14                    ->end()
15                ->end()
16            ->end()
17        ;
18
19        return $treeBuilder;
20    }
21 }

```

Он позволяет сконфигурировать список имён событий следующим образом:

```

1 generic_listener:
2     events: [kernel.request, kernel.response, ...]

```

Стандартный способ зарегистрировать “слушатель” заключается в добавлении тегов к сервису слушателя в файле services.xml:

```

1 <service id="generic_event_listener" class="...">
2     <tag name="kernel.event_listener" event="..." method="onEvent" />
3     <tag name="kernel.event_listener" event="..." method="onEvent" />
4 </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:

```

1 class GenericListenerExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         $processedConfig = $this->processConfiguration(
6             new Configuration(),
7             $configs
8         );
9
10        // загружаем services.xml
11        $loader = ...;
12        $loader->load('services.xml');
13
14        $eventListener = $container
15            ->getDefinition('generic_event_listener');
16
17        foreach ($processedConfig['events'] as $eventName) {

```

```

18     // добавляем таги kernel.event_listener для каждого из указанных событий
19     $eventListener
20         ->addTag('kernel.event_listener', array(
21             'event' => $eventName,
22             'method' => 'onEvent'
23         ));
24     }
25 }
26 }
```

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

```

1 if ($processedConfig['events']) {
2     $container->removeDefinition('generic_event_listener');
3 }
```

Используем паттерн Стратегия для загрузки сервисов

Часто бандлы предлагают разные способы выполнить ту или 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 менеджера хранения, давайте создадим класс конфигурации:

```

1 use Symfony\Component\Config\Definition\ConfigurationInterface;
2
3 class Configuration implements ConfigurationInterface
4 {
5     public function getConfigTreeBuilder()
6     {
7         $treeBuilder = new TreeBuilder();
8         $rootNode = $treeBuilder->root('browser');
9
10        $rootNode
11            ->children()
12                ->scalarNode('storage_manager')
13                    ->validate()
14                        ->ifNotInArray(array('doctrine_orm', 'mongo_db'))
15                            ->thenInvalid('Invalid storage manager')
16                        ->end()
17                    ->end()
18                ->end()
19        ;
20
21        return $treeBuilder;
22    }
23 }
```

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

```

1 <services>
2     <service id="mailboxdoctrine_ormstorage_manager" class="...">>
3     </service>
4 </services>

```

И другой - mongo_db.xml:

```

1 <services>
2     <service id="mailboxmongodbsstorage_manager" class="...">>
3     </service>
4 </services>

```

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

```

1 class MailboxExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         $processedConfig = $this->processConfiguration(
6             new Configuration(),
7             $configs
8         );
9
10        // создаём XmlLoader
11        $loader = ...;
12
13        // загружаем только сервисы для выбранного менеджера хранения
14        $storageManager = $processedConfig['storage_manager'];
15        $loader->load($storageManager.'.xml');
16
17        // делаем выбранный менеджер дооступным по умолчанию
18        $container->setAlias(
19            'mailbox.storage_manager',
20            'mailbox.'.$storageManager.'.storage_manager'
21        );
22    }
23 }

```

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 менее, 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:

```

1 <services>
2   <service id="mailboxdoctrine_ormstorage_manager" class="...">
3     </service>
4
5   <service id="mailboxstorage_manager"
6     alias="mailboxdoctrine_ormstorage_manager">
7     </service>
8 </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.

Загрузка и конфигурирование дополнительных сервисов

Предположим, 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 они могут быть не применимы к вашей конкретной ситуации. Есть удобный способ добиться этого:

```

1 class Configuration implements ConfigurationInterface
2 {
3   public function getConfigTreeBuilder()
4   {
5     $treeBuilder = new TreeBuilder();
6     $rootNode = $treeBuilder->root('input_filter');
7
8     $rootNode
9       ->children()
10      ->arrayNode('form_integration')
11        // будет активно по умолчанию
12        ->canBeDisabled()
13      ->end()
14      ->arrayNode('doctrine_orm_integration')
15        // будет отключено по умолчанию
16        ->canBeEnabled()
17      ->end()
18      ->end()
19    ;
20
21    return $treeBuilder;
22  }
23 }

```

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

```

1  input_filter:
2      form_integration:
3          enabled: false
4      doctrine_orm_integration:
5          enabled: true

```

В классе расширения вам остаётся только загрузить соответствующие сервисы:

```

1  class InputFilterExtension extends Extension
2  {
3      public function load(array $configs, ContainerBuilder $container)
4      {
5          $processedConfig = $this->processConfiguration(
6              new Configuration(),
7              $configs
8          );
9
10         if ($processedConfig['doctrine_orm_integration']['enabled']) {
11             $this->loadDoctrineORMIntegration(
12                 $container,
13                 $processedConfig['doctrine_orm_integration']
14             );
15         }
16
17         if ($processedConfig['form_integration']['enabled']) {
18             $this->loadFormIntegration(
19                 $container,
20                 $processedConfig['form_integration']
21             );
22         }
23
24         ...
25     }
26
27     private function loadDoctrineORMIntegration(
28         ContainerBuilder $container,
29         array $configuration
30     ) {
31         // загружаем сервисы и т.д.
32         ...
33     }
34
35     private function loadFormIntegration(
36         ContainerBuilder $container,
37         array $configuration
38     ) {
39         ...
40     }
41 }

```

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:

```

1  class Configuration implements ConfigurationInterface
2  {
3      public function getConfigTreeBuilder()
4      {
5          $treeBuilder = new TreeBuilder();
6
7          $rootNode = $treeBuilder->root('input_filter');
8
9          $rootNode
10             ->append($this->createFormIntegrationNode())
11             ->append($this->createDoctrineORMIntegrationNode())
12         ;
13
14         return $treeBuilder;
15     }
16
17     private function createDoctrineORMIntegrationNode()
18     {
19         $builder = new TreeBuilder();
20
21         $node = $builder->root('doctrine_orm_integration');
22
23         $node
24             ->canBeEnabled()
25             ->children()
26                 // тут можно добавить дополнительные опции конфигурации
27                 ...
28             ->end();
29
30         return $node;
31     }
32
33     private function createFormIntegrationNode()
34     {
35         ...
36     }
37 }
```

Конфигурируем сервис, который будем 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 такой, вы можете попросить разработчика указать ID сервиса-шифратора:

```

1 matthias_security:
2     encrypter_service: my_encrypter_service_id
```

Класс конфигурации в этом случае будет выглядеть таким образом:

```

1 class Configuration implements ConfigurationInterface
2 {
3     public function getConfigTreeBuilder()
4     {
5         $treeBuilder = new TreeBuilder();
6         $rootNode = $treeBuilder->root('matthias_security');
7
8         $rootNode
9             ->children()
10            ->scalarNode('encrypter_service')
11            ->isRequired()
12            ->end()
13        ->end()
14    ;
15
16    return $treeBuilder;
17 }
18 }
```

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

```

1 class MatthiasSecurityExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         $processedConfig = $this->processConfiguration(
6             new Configuration(),
7             $configs
8         );
9
10        $container->setAlias(
11            'matthias_security.encrypter',
12            $processedConfig['encrypter_service']
13        );
14    }
15 }
```

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 бандлу псевдониму:

```

1 <service id="matthias_security.encrypted_data_manager" class="...">
2   <argument type="service" id="matthias_security.encrypter" />
3 </service>
```

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

```

1 class EncryptedDataManager
2 {
3     public function __construct(EncrypterInterface $encrypter)
4     {
5         // здесь мы можем быть уверены, что $encrypter валидный
6     }
7 }

```

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. Предположим, вы хотите, чтобы пользователи вашего бандла могли 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 тег - `resource`. Ваш класс Configuration для данного случая может выглядеть примерно так:

```

1 class Configuration implements ConfigurationInterface
2 {
3     public function getConfigTreeBuilder()
4     {
5         $treeBuilder = new TreeBuilder();
6         $rootNode = $treeBuilder->root('resource_management');
7
8         $rootNode
9             ->children()
10            ->arrayNode('resources')
11            ->prototype('array')
12                ->children()
13                    ->scalarNode('type')
14                    ->validate()
15                    ->ifNotInArray(
16                        array('directory', 'file')
17                    )
18                    ->thenInvalid('Invalid type')
19                    ->end()
20                ->end()
21                ->scalarNode('path')
22                ->end()
23                ->end()
24            ->end()
25        ->end()
26    ->end()
27    ;
28
29     return $treeBuilder;
30 }
31 }

```

Пример конфигурации ресурсов:

```

1 resource_management:
2     resources:
3         global_templates:
4             type: directory
5             path: Resources/views
6         app_kernel:
7             type: file
8             path: AppKernel.php

```

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

```

1 class ResourceManagementExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         $processedConfig = $this->processConfiguration(
6             new Configuration(),
7             $configs
8         );
9
10        $resources = $processedConfig['resources'];
11
12        foreach ($resources as $name => $resource) {
13            $this->addResourceDefinition($container, $name, $resource);
14        }
15    }
16
17    private function addResourceDefinition(
18        ContainerBuilder $container,
19        $name,
20        array $resource
21    ) {
22        // определяем класс
23        $class = $this->getResourceClass($resource['type']);
24
25        $definition = new Definition($class);
26
27        // добавляем тег
28        $definition->addTag('resource');
29
30        $serviceId = 'resource.' . $name;
31
32        $container->setDefinition($serviceId, $definition);
33    }
34
35    private function getResourceClass($type)
36    {
37        if ($type === 'directory') {
38            return 'Resource\Directory';
39        } elseif ($type === 'file') {
40            return 'Resource\File';
41        }
42
43        throw new \InvalidArgumentException('Type not supported');
44    }
45 }

```

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

Приёмы создания параметров

Сервисный контейнер помимо сервисов содержит также и параметры. Параметры - это простые значения, которые могут быть представлены в виде 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.

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:

```

1 parameters:
2     ...
3 security:
4     ...
5 framework:
6     ...

```

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

```

1 parameters:
2     ... # загружено из parameters.yml
3     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):

```

1 parameters:
2     ...
3 services:
4     some_service_id:
5         class: SomeClass

```

Определение и загрузка параметров

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

```

1 doctrine:
2     dbal:
3         driver: %database_driver%

```

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

```

1 <parameters>
2   <parameter key="form.factory.class">
3     Symfony\Component\Form\FormFactory
4   </parameter>
5 </parameters>
6
7 <service id="form.factory" class="%form.factory.class%">
8 </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 на лету 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 делать в соответствующем `compiler pass`:

```

1 use Symfony\Component\DependencyInjection\ContainerBuilder;
2 use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
3
4 class ReplaceClassCompilerPass implements CompilerPassInterface
5 {
6   public function process(ContainerBuilder $container)
7   {
8     $myCustomFormFactoryClass = ...;
9
10    $container
11      ->getDefinition('form.factory')
12      ->setClass($myCustomFormFactoryClass);
13  }
14 }

```

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:

```

1 parameters:
2   my_cache_dir: %kernel.cache_dir%/my_cache

```

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

```

1 class MyExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         $myCacheDir = $container->getParameter('my_cache_dir');
6
7         ...
8     }
9 }
```

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:

```

1 $myCacheDir = $container->getParameter('my_cache_dir');
2
3 $myCacheDir = $container->getParameterBag()->resolveValue($myCacheDir);
4
5 // теперь вы можете создать директорию для кэширования
6 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 контейнера (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 такие 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`.

Определяем параметры в расширениях контейнера

В будущем вы 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.

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

```
1 browser:
2   timeout: 30
```

Для этого класс конфигурации должен иметь следующий вид:

```
1 class Configuration implements ConfigurationInterface
2 {
3     public function getConfigTreeBuilder()
4     {
5         $treeBuilder = new TreeBuilder();
6         $rootNode = $treeBuilder->root('browser');
7
8         $rootNode
9             ->children()
10                ->scalarNode('timeout')
11                ->end()
12            ->end()
13        ;
14
15        return $treeBuilder;
16    }
17 }
```

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

```
1 class BrowserExtension extends Extension
2 {
3     public function load(array $configs, ContainerBuilder $container)
4     {
5         // загружаем определения сервисов
6         $fileLocator = new FileLocator(__DIR__.'../../../Resources/config');
7         $loader = new XmlFileLoader($container, $fileLocator);
8         $loader->load('services.xml');
9
10        // обрабатываем конфигурацию
11        $processedConfig = $this->processConfiguration(
12            new Configuration(),
13            $configs
14        );
15    }
16 }
```

Сервис browser определён в services.xml:

```

1 <service id="browser" class="...">>
2   <argument>%browser.timeout%</argument>
3 </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:

```
1 $container->setParameter('browser.timeout', $processedConfig['timeout']);
```

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:

```

1 array (
2   'ROLE_ADMIN' => array (
3     0 => 'ROLE_USER',
4   ),
5   'ROLE_SUPER_ADMIN' => array (
6     0 => 'ROLE_USER',
7     1 => 'ROLE_ADMIN',
8     2 => 'ROLE_ALLOWED_TO_SWITCH',
9   ),
10 )
```

Предположим теперь, что 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:

```

1 class EnhanceRoleHierarchyPass implements CompilerPassInterface
2 {
3   public function process(ContainerBuilder $container)
4   {
5     $parameterName = 'security.role_hierarchy.roles';
6
7     $roleHierarchy = $container->getParameter($parameterName);
8
9     // модифицируем иерархию ролей
10    ...
11
12    $container->setParameter($parameterName, $roleHierarchy);
13  }
14}
```

Не забывайте регистрировать ваши проходы компилятора в классе вашего бандла:

```
1 class YourBundle extends Bundle
2 {
3     public function build(ContainerBuilder $container)
4     {
5         $container->addCompilerPass(new EnhanceRoleHierarchyPass());
6     }
7 }
```

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.

Организация слоёв приложения

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->getEmail())
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 высыпается 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 создания 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 контроллеро-центризмом - 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 занимается только одной задачей.

Обработчики форм

Первым нашим шагом будет следующий: 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:

```

1 class AccountController extends Controller
2 {
3     public function newAction(Request $request)
4     {
5         $account = new Account();
6
7         $form = $this->createForm(new AccountType(), $account);
8
9         $formHandler = $this
10            ->get('matthias_account.create_account_form_handler');
11
12         if ($formHandler->handle($form, $request)) {
13             $this->sendAccountConfirmationMessage($account);
14
15             return $this->redirect($this->generateUrl('mailbox_index'));
16         }
17
18         return array(
19             'form' => $form->createView(),
20         );
21     }
22 }

```

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:

```

1 use Symfony\Component\Form\FormError;
2
3 public function handle(FormInterface $form, Request $request)
4 {
5     if (...) {
6         $form->addError(new FormError('У нас возникла проблемка...'));
7
8         return false;
9     }
10 }

```

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.

Доменные менеджеры

Обработчик формы (и, возможно, класс формы) - это замечательные кандидаты на повторное использование. Тем не менее, в нашем случае 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 слову

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:

```
1 namespace Matthias\AccountBundle\DomainManager;
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 AccountManager
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 createAccount(Account $account)
23     {
24         $confirmationCode = $this
25             ->secureRandom
26             ->nextBytes(4);
27
28         $account
29             ->setConfirmationCode(md5($confirmationCode));
30
31         $this->entityManager->persist($account);
32         $this->entityManager->flush();
33     }
34 }
```

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

```
1 class CreateAccountFormHandler
2 {
3     private $accountManager;
4
5     public function __construct(AccountManager $accountManager)
6     {
7         $this->accountManager = $accountManager;
8     }
9
10    public function handle(FormInterface $form, Request $request)
11    {
12        ...
13
14        $validAccount = $form->getData();
```

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

Ниже представлены определения 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
- ...

События

Как вы могли отметить выше, внутри контроллера, после создания 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:

```
1 namespace Matthias\AccountBundle\Event;
2
3 use Symfony\Component\EventDispatcher\Event;
4
5 class AccountEvent extends Event
6 {
7     private $account;
8
9     public function __construct(Account $account)
10    {
11         $this->account = $account;
12    }
13
14     public function getAccount()
15    {
16         return $this->account;
17    }
18 }
```

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:

```
1 namespace Matthias\AccountBundle\Event;
2
3 class AccountEvents
4 {
5     const NEW_ACCOUNT_CREATED = 'matthias_account.new_account_created';
6 }
```

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

```
1 namespace Matthias\AccountBundle\DomainManager;
2
3 use Symfony\Component\EventDispatcher\EventDispatcherInterface;
4 use Matthias\AccountBundle\Event\AccountEvents;
5 use Matthias\AccountBundle\Event\AccountEvent;
6
7 class AccountManager
8 {
9     ...
10
11     private $eventDispatcher;
12
13     public function __construct(
14         ...
15         EventDispatcherInterface $eventDispatcher
16     ) {
17         ...
18
19         $this->eventDispatcher = $eventDispatcher;
20     }
21
22     public function createAccount(Account $account)
23     {
24         ...
25
26         $this->eventDispatcher->dispatch(
```

```

27     AccountEvents::NEW_ACCOUNT_CREATED,
28     new AccountEvent($account)
29 );
30 }
31 }
```

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

```

1 <service id="matthias_account.account_manager"
2   class="Matthias\AccountBundle\DomainManager\AccountManager">
3   <argument type="service" id="entity_manager" />
4   <argument type="service" id="security.secure_random" />
5   <argument type="service" id="event_dispatcher" />
6 </service>
```

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

```

1 namespace Matthias\AccountBundle\EventListener;
2
3 use Symfony\Component\EventDispatcher\EventSubscriberInterface;
4 use Matthias\AccountBundle\Event\AccountEvents;
5 use Matthias\AccountBundle\Event\AccountEvent;
6
7 class SendConfirmationMailListener implements EventSubscriberInterface
8 {
9     private $mailer;
10
11    public static function getSubscribedEvents()
12    {
13        return array(
14            AccountEvents::NEW_ACCOUNT_CREATED => 'onNewAccount',
15        );
16    }
17
18    public function __construct(\SwiftMailer $mailer)
19    {
20        $this->mailer = $mailer;
21    }
22
23    public function onNewAccount(AccountEvent $event)
24    {
25        $this->sendConfirmationMessage($event->getAccount());
26    }
27
28    private function sendConfirmationMessage(Account $account)
29    {
30        $message = \Swift_Message::newInstance();
31
32        ...
33
34        $this->mailer->send($message);
35    }
36 }
```

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:

```

1 <service id="life_online_account.send_confirmation_mail_listener"
2   class="Matthias\AccountBundle\EventListener\SendConfirmationMailListener">
3   <argument type="service" id="mailer" />
4   <tag name="kernel.event_subscriber" />
5 </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:

```

1 class AccountManager
2 {
3     private $entityManager;
4     private $secureRandom;
5
6     public function __construct(
7         EntityManger $entityManager,
8         SecureRandomInterface $secureRandom
9     ) {
10        $this->entityManager = $entityManager;
11        $this->secureRandom = $secureRandom;
12    }
13
14    public function createAccount(Account $account)
15    {
16        $confirmationCode = $this
17            ->secureRandom
18            ->nextBytes(4);
19
20        $account
21            ->setConfirmationCode(md5($confirmationCode));
22
23        $this->entityManager->persist($account);
24        $this->entityManager->flush();
25    }
26}
```

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 внедриться в процесс сохранения, обновления или удаления объектов.

Doctrine ORM это реализуется через подписчики события:

```
1 use Doctrine\Common\EventSubscriber;
2 use Doctrine\ORM\Event\LifecycleEventArgs;
3
4 class CreateConfirmationCodeEventSubscriber implements EventSubscriber
5 {
6     private $secureRandom;
7
8     public function __construct(SecureRandomInterface $secureRandom)
9     {
10         $this->secureRandom = $secureRandom;
11     }
12
13     public function getSubscribedEvents()
14     {
15         return array(
16             'prePersist'
17         );
18     }
19
20     public function prePersist(LifecycleEventArgs $event)
21     {
22         // this will be called for *each* new entity
23
24         $entity = $event->getEntity();
25         if (!$entity instanceof Account) {
26             return;
27         }
28
29         $this->createConfirmationCodeFor($entity);
30     }
31
32     private function createConfirmationCodeFor(Account $account)
33     {
34         $confirmationCode = $this
35             ->secureRandom
36             ->nextBytes(4);
37
38         $account
39             ->setConfirmationCode(md5($confirmationCode));
40     }
41 }
```

Вы можете зарегистрировать этот подписчик, используя метку doctrine.event_subscriber:

```

1 <service id="create_confirmation_code_listener" class="...">
2   <tag name="doctrine.event_subscriber" />
3 </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:

```

1 use Doctrine\ORM\Event\PreUpdateEventArgs;
2
3 class CreateConfirmationCodeEventSubscriber implements EventSubscriber
4 {
5     public function getSubscribedEvents()
6     {
7         return array(
8             'preUpdate'
9         );
10    }
11
12    public function preUpdate(PreUpdateEventArgs $event)
13    {
14        $entity = $event->getEntity();
15        if (!$entity instanceof Account) {
16            return;
17        }
18
19        if ($event->hasChangedField('emailAddress')) {
20            // create a new confirmation code
21            $confirmationCode = ...;
22            $event->setNewValue('confirmationCode', $confirmationCode);
23        }
24    }
}

```

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:

```

1 $entity = $event->getEntity();
2 $className = get_class($entity);
3 $entityManager = $event->getEntityManager();
4 $classMetadata = $entityManager->getClassMetadata($className);
5 $unitOfWork = $entityManager->getUnitOfWork();
6 $unitOfWork->recomputeSingleEntityChangeSet($classMetadata, $entity);

```

Состояния и контекст

Сервисы можно разделить на две группы:

1. Статические сервисы
2. Динамические сервисы

Большинство сервисов, определённых в сервисном контейнере принадлежат к первой категории. Статический сервис выполняет одну и ту же работу каждый раз будучи вызванным. Получая на входе одни и те же значения такой сервис должен выдать одинаковый результат раз за разом. Примером такого сервиса могут быть сервис отправки email пользователю или сохранение объекта при помощи менеджера сущностей.

Во вторую категорию попадают сервисы, которые могут изменяться: когда вы используете динамический сервис, вы не можете заранее знать, работает ли он так, как вы ожидаете, так как результат его работы может зависеть от текущего запроса или, в случае Symfony-приложения: от факта, что ядро обрабатывает запрос.

Контекст безопасности

@dbykadorov: ВАЖНО! Отличия в версиях 2.6 и выше

Вообще говоря данный раздел уже изрядно устарел. Начиная с версии Symfony 2.6 `security.context` [объявлен устаревшим](#) и в 3.0 был удалён. Обязанности контекста безопасности разделили другие сервисы, прежде всего `security.token_storage` и `security.authorization_checker`

```
1 // Symfony 2.5
2 $user = $this->get('security.context')->getToken()->getUser();
3 // Symfony 2.6
4 $user = $this->get('security.token_storage')->getToken()->getUser();
5
6 // Symfony 2.5
7 if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) { ... }
8 // Symfony 2.6
9 if (false === $this->get('security.authorization_checker')->isGranted('ROLE_ADMIN')) { ... }
```

Используются они схожим образом, так что кардинальных отличий не будет, но имейте это в виду.

Одним из очевидных примеров динамического сервиса является контекст безопасности (`security context`). Вернёт ли он вам объект пользователя, когда вы выполните вызов методов `->getToken()`-`>getUser()`, зависит от многих факторов, прежде всего от запроса и от того, смог ли файрвол определить - залогинен пользователь или нет. Кроме того, многие сервисы зависят от `security context'a`, например, вот этот:

```

1  use Symfony\Component\Security\Core\SecurityContextInterface;
2
3  class UserMailer
4  {
5      private $securityContext;
6      private $mailer;
7
8      public function __construct(
9          SecurityContextInterface $securityContext,
10         \SwiftMailer $mailer
11     ) {
12         $this->securityContext = $securityContext;
13         $this->mailer = $mailer;
14     }
15
16     public function sendMailToCurrentUser($subject, $body)
17     {
18         $token = $this->securityContext->getToken();
19         if (!$token instanceof TokenInterface) {
20             // we are not behind a firewall
21             return;
22         }
23
24         $user = $token->getUser();
25         if (!$user instanceof User) {
26             // no logged in user
27             return;
28         }
29
30         $message = \Swift_Message::newInstance()
31             ->setTo($user->getEmail())
32             ->setSubject($subject)
33             ->setBody($messageBody);
34
35         $this->get('mailer')->send($message);
36     }
37 }
```

Определение этого сервиса выглядит вот так:

```

1 <service id="user_mailer" class="UserMailer">
2     <argument type="service" id="security.context" />
3     <argument type="service" id="mailer" />
4 </service>
```

Внутри контроллера вы можете использовать этот сервис для того, чтобы отправить письмо текущему пользователю (обратите внимание на аннотацию `@Secure` - она позволяет быть уверенным, что тут пользователь будет залогинен и у него будет как минимум роль `ROLE_USER`):

```

1 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
2 use JMS\SecurityExtraBundle\Annotation\Secure;
3
4 class SomeController extends Controller
{
5
6     /**
7      * @Secure("ROLE_USER")
8      */
9     public function sendMailAction()
10    {
11        $this->get('user_mailer')->sendMailToCurrentUser('Hi', 'there!');
12    }
13}

```

Хотите знать, почему этот код является плохим? Ну... вот вам как минимум 2 причины:

1. Сервис `user_mailer` может быть использован только для отправки писем текущему юзеру. В то же время, ничего специфичного относительно текущего пользователя (по сравнению с прочими) тут не выполняется. Метод `sendMailToCurrentUser($subject, $body)` может быть легко заменен на `sendMailTo(User $user, $subject, $body)`. Это делает ваш класс более общим и пригодным для повторного использования.
2. Сервис `user_mailer` может быть использован только лишь в запросах, которые обслуживаются внутри какого-либо файрволла (TODO: найти когда файрвол перестал быть необходим для вызова функций безопасности (ранее вне файрвола кидалось исключение)). Если запросить сервис `user_mailer` из консольной команды, он будет совершенно бесполезен, так как в этом случае файрвол не будет определён. Решение первой проблемы решит и эту без дополнительных усилий с вашей стороны.

Итак, давайте изменим класс `UserMailer`, чтобы решить указанные проблемы:

```

1 class UserMailer
2 {
3     private $mailer;
4
5     public function __construct(\SwiftMailer $mailer)
6     {
7         $this->mailer = $mailer;
8     }
9
10    public function sendMailToUser(User $user, $subject, $body)
11    {
12        $message = \Swift_Message::newInstance()
13            ->setTo($user->getEmail())
14            ->setSubject($subject)
15            ->setBody($body);
16
17        $this->mailer->send($message);
18    }
19}

```

В результате у нас класс стал намного меньше, а код - чище. Теперь сервис `user_mailer` ничего не знает о пользователе, которому должен отправить письмо (что также хорошо): поэтому вам каждый раз нужно будет передавать объект класса `User`, который может быть, а может и не быть текущим пользователем.

Контроллер является отличным местом для определения текущего пользователя, так как мы точно знаем, что контроллер выполняется лишь во время обработки запроса, и мы можем быть уверены, что он защищён файрволлом, если мы заранее явно сконфигурировали это. В этом случае нам будет доступен “текущий пользователь” (предполагается что файрволл корректно настроен) и мы можем переписать наш контроллер следующим образом:

```

1 class SomeController extends Controller
2 {
3     /**
4      * @Secure("ROLE_USER")
5      */
6     public function sendMailAction()
7     {
8         $user = $this->getUser();
9
10        $this->get('user_mailer')->sendMailTo($user, 'Hi', 'there!');
11    }
12 }
```

Текущий пользователь

Когда бы вы ни захотели выполнить какие-либо действия с текущим пользователем в сервисе, всегда желательно передавать его откуда-то извне, как аргумент или даже используя метод-сеттер, чем получать пользователя, используя `SecurityContext` в самом сервисе.

Запрос

Токен пользователя добавляется в контекст безопасности один раз для каждого запроса, но только для мастер-запроса. Но другие сервисы, которые вы создаёте, могут зависеть от текущего объекта запроса, который отличается от мастер-запроса или под-запроса. Этот объект запроса доступен в виде сервиса `request`, который является динамическим. В какой-то момент времени (фактически, перед тем как ядро начнёт обработку запроса) этот сервис запроса будет недоступен и не сможет быть использован любым другим объектом в качестве зависимости. В то же время, может быть ситуация, когда существует более одного сервиса запроса, а именно – когда ядро обрабатывает подзапрос.

В сервисном контейнере эта проблема решается при помощи т.н. сфер действия (scopes). Контейнер может входить в сферы действия и покидать их; в момент перехода между сферами действия либо восстанавливается предыдущий сервис, либо для новой сферы действия предоставляется новый. В случае с запросом, есть несколько состояний, в которых может быть контейнер:

1. Контейнер находится в сфере действия `container`, сервис запроса не определён.
2. Ядро обрабатывает главный запрос, оно переводит контейнер в сферу действия `request` и создаёт сервис запроса.
3. Ядро обрабатывает подзапрос, оно переводит контейнер в сферу действия `request`, но уже для другого запроса и устанавливает подзапрос в качестве сервиса запроса.
4. Ядро завершает обработку подзапроса и выводит контейнер из сферы действия `request` для подзапроса. Восстанавливается предыдущий сервис запроса.

- Ядро завершает обработку главного запроса и выводит контейнер из сферы действия запроса. Сервис запроса опять будет неопределён.

Таким образом, когда бы вы ни выполнили вызов `$container->get('request')` вы “всегда” получите текущий объект `Request`. Когда вам нужно предоставить сервис запроса одному из ваших объектов, для этого имеется [несколько способов](#), которые я, тем не менее, категорически не рекомендую использовать.

@dbykadorov: ВАЖНО! Отличия в версиях 2.8 и страшे

В Symfony, начиная с версии 2.8 принцип сфер действия - `scopes` [помечен как устаревший](#). В `current` ветке документации нет более такой статьи, поэтому ссылка ведёт на документацию версии 2.8.

Также изменения коснулись и сервиса запроса, который выше предлагается получать, например так: `$container->get('request')`. Эти изменения - часть процесса рефакторинга работы с запросом, который начался с версии 2.4 введением сервиса `request_stack` (см. [тут](#)), который позволил разрешить неопределенность с запросами, которую ранее пытались разрешить при помощи `scopes`.

Таким образом, начиная с версии 2.8 рекомендуется (а с версии 3.0 другого варианта и нет) получать текущий запрос через `request_stack`:

```
1 // Где-то в контроллере
2 ...
3 public function foo () {
4     $request = $this
5         ->get('request_stack')
6         ->getCurrentRequest();
7     // ... делаем с запросом что нам требуется
8 }
```

В принципе, Маттиас как раз далее и предлагает альтернативы для этой ситуации, актуальные на момент написания им книги, для Symfony 2.3 и настоятельно не рекомендует писать ваш код, зависимый от сервиса запроса.

Сама по себе необходимость иметь доступ к запросу для выполнения каких-либо действий не является чем-то экстремально плохим. Но вы не должны зависеть от сервиса запроса. Положим вам нужно сделать что-то типа счётчика посещений страницы:

```
1 use Symfony\Component\HttpFoundation\Request;
2
3 class PageCounter
4 {
5     private $counter;
6
7     public function __construct(Request $request)
8     {
9         $this->request = $request;
10    }
11
12    public function incrementPageCounter()
13    {
```

```

14     $uri = $this->request->getPathInfo();
15
16     // каким-либо образом инкрементим счётчик посещений для этого URI
17     ...
18 }
19 }
```

Определение этого сервиса будет таким:

```

1 <service id="page_counter" class="PageCounter" scope="request">
2   <argument type="service" id="request" />
3 </service>
```

Так как вы зависите от сервиса запроса, весь ваш сервис должен иметь сферу действия `request`.

В контроллере вызов счётчика в этом случае будет выглядеть так:

```

1 class InterestingController extends Controller
2 {
3     public function indexAction()
4     {
5         $this->get('page_counter')->incrementPageCounter();
6     }
7 }
```

Несомненно, сервис `page_counter` имеет возможность получить URI текущей страницы, так как у него есть весь объект `Request`. Но:

1. Статистика может быть получена лишь при выполнении текущего запроса. Когда ядро завершит обработку запроса, сервис запроса будет установлен в `null` и наш счётчик посещений будет бесполезен.
2. Класс `PageCounter` тесно связан с объектом класса `Request`. Счётчику же, на самом деле, требуется лишь URI. И этот URI не обязательно может быть получен из объекта `Request`. Будет лучше, если мы передадим его в качестве аргумента метода `incrementPageCounter():php` `public function incrementPageCounter($uri)`
`{ ... }`

Эти две проблемы могут казаться надуманными, но в один прекрасный день вам может потребоваться импортировать посещения, собранные каким-либо другим способом и придется вызывать метод `incrementPageCounter()` "вручную". В этом случае вы только порадуетесь, что ваш счётчик не привязан к классу `Request`.

Избегаем зависимостей от текущего запроса

Имеется две главных стратегии, как можно избежать зависимости от текущего запроса:

Используем слушатель (event listener)

Как вы вероятно можете помнить из первой главы - ядро всегда отправляет событие `kernel.request` для каждого запроса, который оно обрабатывает. В этот момент вы можете использовать слушатель для выполнения действий, которые вы бы хотели выполнять для каждого запроса, например, инкрементить счётчик посещений страницы, или может быть остановить дальнейшее выполнение, вызвав исключение.

```

1 class PageCounterListener
2 {
3     private $counter;
4
5     public function __construct(PageCounter $counter)
6     {
7         $this->counter = $counter;
8     }
9
10    public function onKernelRequest(GetResponseEvent $event)
11    {
12        $request = $event->getRequest();
13
14        $this->counter->incrementPageCounter($request->getPathInfo());
15    }
16 }
```

Предоставлять объект запроса во время выполнения

Когда вам необходим весь объект запроса целиком, всегда лучше начать его использование там, где вы точно уверены, что этот объект существует (и вы можете его получить) и в этом месте передать запрос в нужный сервис:

```

1 class SomeController extends Controller
2 {
3     public function indexAction(Request $request)
4     {
5         $this->get('page_counter')->handle($request);
6     }
7 }
```

Поиск совпадений для запросов (request matcher)

Во многих ситуациях, в которых вам будет нужен объект запроса для вашего сервиса, вы, вероятно, хотите его использовать для выполнения каких-либо сравнений. В этом случае вы можете абстрагироваться от логики сравнения при помощи обнаружителя совпадений запроса (request matcher):

```

1 use Symfony\Component\HttpFoundation\RequestMatcherInterface;
2 use Symfony\Component\HttpFoundation\Request;
3
4 class MyRequestMatcher implements RequestMatcherInterface
5 {
6     public function matches(Request $request)
7     {
8         return $request->getClientIp() === '127.0.0.1';
9     }
10 }
```

Или используя стандартный RequestMatcher:

```
1 use Symfony\Component\HttpFoundation\RequestMatcher;
2
3 $matcher = new RequestMatcher();
4 $matcher->matchIp('127.0.0.1');
5 // $matcher->matchPath('/^secure');
6 ...
7
8 if ($matcher->matches($request)) {
9     ...
10 }
```

Использование только нужных значений

Перед тем, как передать в сервис весь объект запроса, всегда спрашивайте себя: а нужна ли мне вся эта информация целиком? Или же мне нужна только её конкретная часть? Если это так, отвязывайте зависимость от запроса и передавайте в качестве аргументов лишь нужные вам данные:

Изменим это:

```
1 class AccessLogger
2 {
3     public function logAccess(Request $request)
4     {
5         $ipAddress = $request->getClientIp();
6
7         // log the IP address
8         ...
9     }
10 }
```

На это:

```
1 class AccessLogger
2 {
3     public function logAccess($ipAddress)
4     {
5         ...
6     }
7 }
```

Это сделает ваш код пригодным для повторного использования в других проектах, даже таких, которые не используют компонент Symfony HttpFoundation. См. также раздел книги [уменьшаем связность с фреймворком](#).

IV Соглашения по конфигурированию

Настройка конфигурации приложения

Symfony Standard Edition рекомендует использовать файл parameters.yml, для значений, которые специфичны для конкретного экземпляра приложения (т.е. это может быть production сервер, staging или же локальная инсталляция у разработчика на рабочей станции). Конфигурационные параметры, определяемые в файле config.yml содержат места для подстановки, ссылающиеся на значения из parameters.yml, который, в свою очередь, импортируется в секции imports:

```
1 imports:
2     - { resource: parameters.yml }
3 doctrine:
4     dbal:
5         user:      %database_user%
6         password: %database_password%
```

В файле parameters.yml вы должны определить значения:

```
1 parameters:
2     database_user:    matthias
3     database_password: cookies
```

Иначе вы получите исключение при запуске приложения:

```
1 You have requested a non-existent parameter "database_user".
```

Однако, так настраивать приложение не очень практично. Чем больше зависимостей будет у вашего проекта, тем больше параметров будут различаться у вас и ваших коллег (не говоря уже о различиях с продуктовым сервером). Таким образом перечень настроек в parameters.yml будет становиться всё больше и больше, и каждый раз, когда другой участник вашей команды будет добавлять новый параметр, вы будете получать ошибку ParameterNotFoundException после того, как запустите его изменения из репозитория.

Этот метод не очень гибок, когда речь заходит об временном изменении поведения некоторых бандлов. Вероятно время от времени вы будете по ошибке коммитить что-то типа такого:

```
1 # in config.yml
2 swiftmailer:
3     # deliver all emails sent by the application to this address:
4     delivery_address: matthiasnoback@gmail.com
```

Конечно, вы можете определить новый параметр для этого случая:

```

1 #in config.yml
2   swiftmailer:
3     delivery_address: %developer_email_address%
4
5 #in parameters.yml
6   parameters:
7     developer_email_address: matthiasnoback@gmail.com

```

Но вашим коллегам изменять поведение по умолчанию возможно никогда и не потребуется. Они лишь хотят чтобы всё работало после обновления проекта.

Локальные конфигурационные файлы

Наилучшим решением в данном случае будет использование таких конфигурационных файлов, которые существуют лишь локально у вас на рабочей станции. Например, для каждого окружения вы можете создать соответствующий файл - `local_{env}.yml`. Затем, вы можете загрузить его после загрузки основного файла `config_{env}.yml`. Для этого нужно модифицировать `app/AppKernel.php`:

```

1 public function registerContainerConfiguration(LoaderInterface $loader)
2 {
3     $loader->load(__DIR__ . '/config/config_'. $this->getEnvironment() . '.yml');
4
5     $localFile = __DIR__ . '/config/local_'. $this->getEnvironment() . '.yml';
6
7     if (is_file($localFile)) {
8         $loader->load($localFile);
9     }
10 }

```

Так как локальный файл настроек загружается после основного, каждый разработчик может переопределить любую часть конфигурации проекта. Если локальный конфигурационный файл не был создан, приложение не будет его загружать. Это означает, что при деплое вашего приложения не потребуется никаких дополнительных шагов.

Не забывайте добавлять локальные конфигурационные файлы `local_{env}.yml` в файл `.gitignore`. Также будет неплохо добавить `.dist` версии для `local_{env}.yml` файлов, содержащие некоторые настройки по умолчанию, которые удобно использовать, разрабатывая локально. Также проследите, чтобы все настройки по умолчанию были бы снабжены исчерпывающими комментариями:

```

1 #local_dev.yml.dist
2 imports:
3   - { resource: config_dev.yml }
4
5 #Swiftmailer:
6 #   delivery_address: your-mail-address@host

```

Также удостоверьтесь, что все `.dist` файлы локальных настроек `local_{env}.yml.dist` **были закоммичены** в репозиторий.

Храните parameters.yml

Использовать параметры для некоторых значений всё ещё является хорошей идеей. Например, без указания логина и пароля к базе данных, всё приложение не будет работать, так что имеет смысл запрашивать эти значения при инсталляции и/или обновлении проекта.

Если ваш Symfony проект основывается на версии меньше чем 2.3, вам нужно добавить в ваш проект бандл [ParameterHandler](#). После того как это будет выполнено, при использовании команд `composer install` или `composer update` ParameterHandler будет сравнивать содержимое файла `parameters.yml.dist` с содержимым `parameters.yml` и будет запрашивать недостающие параметры.

@dbykadorov:

в версиях Symfony 2.3 и старше Incenteev ParameterHandler [доступен “из коробки”](#)

Добавьте default_parameters.yml

Хотя теперь конфигурация приложения уже стала очень гибкой, имеются ситуации, когда вам придётся использовать ещё один дополнительный способ конфигурирования. Давайте рассмотрим следующую конфигурацию MongoDB:

```
1 doctrine_mongodb:  
2     connections:  
3         default:  
4             server: mongodb://[%mongo_host%]:%mongo_port%
```

Вам вероятно не захочется копировать всю эту иерархию в ваш файл `local_dev.yml` для того чтобы проект заработал на вашей рабочей станции. Вам нужны лишь параметры `%mongo_host%` и `%mongo_port%`. Тем не менее, в настройках по умолчанию эти параметры могут не сильно различаться. На большинстве рабочих станций разработчиков хост и порт MongoDB будут одинаковыми. Для такого случая вы можете добавить файл `default_parameters.yml`, который будет содержать эти параметры по умолчанию, после этого после “свежей” инсталляции разработчики не должны будут указывать эти значения. Импортируйте этот файл перед `parameters.yml`:

```
1 # in config.yml  
2 imports:  
3     - { resource: default_parameters.yml }  
4     - { resource: parameters.yml }  
5  
6 #...
```

Теперь в `default_parameters.yml` вы можете добавить параметры:

```
1 # in default_parameters.yml
2 parameters:
3     mongo_host: localhost
4     mongo_port: 27017
```

И в случае если порт MongoDB у вас будет отличаться, вы будете иметь возможность переопределить его в файле parameters.yml:

```
1 # in parameters.yml
2 parameters:
3     mongo_port: 71072
```

Если вы используете ParameterHandler, упомянутый выше, в комбинации default_parameters.yml и parameters.yml, убедитесь, что у вас в composer.json есть следующие строки:

```
1 "extra": {
2     "incenteev-parameters": {
3         "file": "app/config/parameters.yml",
4         "keep-outdated": true
5     }
6 }
```

В этом случае параметры в parameters.yml, которые будут переопределять значения из default_parameters.yml но не будут указаны в parameters.yml.dist - не будут автоматически удаляться из parameters.yml.

Заключение

Параметры в parameters.yml являются необходимыми для функционирования приложения.

Параметры в default_parameters.yml являются параметрами по умолчанию, вы не должны определять их в parameters.yml, но можете при необходимости их там переопределять.

В local_{env}.yml вы можете переопределять любые участки конфигурации в соответствии с вашими текущими девелоперскими потребностями.

Соглашения по конфигурированию

Наибольшая часть конфигурации приложения как правило загружается из файлов в различных форматах. Symfony поддерживает для файлов конфигураций форматы: чистый PHP, Yaml, INI и XML. Эти же форматы (за исключением INI) могут быть использованы для конфигурации маршрутизатора и валидатора. Но, с другой стороны маршруты и правила валидации (а также маппинг сущностей и документов Doctrine!) можно настраивать при помощи аннотаций.

Если вы прочитали предыдущий раздел этой главы, то конфигурация приложения у вас уже в полном порядке. Теперь вам и вашей команде нужно принять решение о форматах конфигурационных файлов для оставшихся частей приложения.

Сделал выбор, не меняйте его

Помните, не так важно что именно вы выберите (некоторые форматы более читабельны, другие более строгие и т.д.), более важным для вас будет придерживаться сделанного выбора во всём приложении.

Маршрутизатор

Маршруты, как и шаблоны для действий, наиболее удобно определять прямо в контроллерах с использованием аннотаций.

```
1 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
2 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
3 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
4
5 /**
6 * @Route("/account")
7 */
8 class AccountController
9 {
10    /**
11     * @Route("/new")
12     * @Method({"GET", "POST"})
13     * @Template
14     */
15    public function newAction() {
16        return array();
17    }
}
```

Каждый бандл должен иметь файл Resources/config/routing.yml, который загружает каждый контроллер в качестве ресурса:

```

1  MatthiasAccountBundle_AccountController:
2      resource: "@MatthiasAccountBundle/Controller/AccountController.php"
3      type: annotation
4
5  MatthiasAccountBundle_CredentialsController:
6      resource: "@MatthiasAccountBundle/Controller/CredentialsController.php"
7      type: annotation
8
9  #...

```

Кроме того, хотя это может быть менее читабельно, вы также можете загрузить всю директорию Controller вашего бандла:

```

1  MatthiasAccountBundleControllers:
2      resource: "@MatthiasAccountBundle/Controller/"
3      type: annotation

```

Конфигурация маршрутов приложения /app/config/routing.yml должна иметь ссылки на файлы routing.yml из всех активных бандлов:

```

1  MatthiasAccountBundle:
2      resource: "@MatthiasAccountBundle/Resources/config/routing.yml"

```

Правила именования маршрутов

Имена маршрутов часто находятся в полном беспорядке. Есть маршрут account_new, account_index, accounts, account_list и т.д. При изменении имени метода действия, старое имя маршрута продолжает проявляться то тут то там. И когда другой контроллер также по стечению обстоятельств получит маршрут accounts, он переопределит существующий маршрут или же сам будет переопределён...

Решение этой проблемы простое: следуйте вот такому шаблону именования (для вас он может даже покажется знакомым):

```
1  {имя бандла без "Bundle"}.{имя контроллера без "Controller"}.{имя действия без "Action"}
```

Например, AccountController из примера выше содержит метод действия newAction. Этот контроллер - часть бандла MatthiasAccountBundle, так что имя маршрута будет таким - matthias_account.account.new:

```

1 /**
2  * @Route("/new", name="matthias_account.account.new")
3 */

```

Запомните: когда вы меняете имя маршрута, также замените его во всём проекте при помощи search-and-replace.

Сервисы

Определяйте ваши сервисы при помощи XML файлов. Это позволит вам пользоваться валидацией XML, автоподстановкой и красивую подсветку синтаксиса (если конечно ваша IDE поддерживает это).

```

1  <?xmlversion="1.0"?>
2  <container xmlns="http://symfony.com/schema/dic/services"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://symfony.com/schema/dic/services
5          http://symfony.com/schema/dic/services/services-1.0.xsd">
6      <services>
7          <service id="matthias_account.account_controller"
8              class="Matthias\AccountBundle\Controller\AccountController">
9              </service>
10         </services>
11     </container>

```

Для id сервисов используйте подчёркивания и символы в нижнем регистре. Опционально добавляйте пространство имён через точку (примерно как выше было предложено делать для маршрутов). Id сервиса, который использует класс MailManager в бандле MatthiasAccountBundle может быть matthias_account.mail_manager.

Если ваши сервисы распределены по нескольким файлам, которые группируются по смыслу (например metadata, form, и т.д.), вы можете добавлять дополнительный префикс к id сервиса. Например, id для сервиса для класса формы CreateAccountType из бандла MatthiasAccountBundle и определённом в файле сервисов form.xml может быть matthias_account.form.create_account_type.

Метаданные Doctrine

Предпочтительным способом конфигурирования сущностей или документов являются аннотации:

```

1 /**
2  * @ORM\Entity
3  * @ORM\Table(name="accounts")
4  */
5 class Account
6 {
7     /**
8      * @ORM\Id
9      * @ORM\Column(type="secure_random_id")
10     */
11     private $id;
12 }

```

Если же речь идёт о (потенциально) переиспользуемом коде, аннотации использовать **не следует**. Почитайте об этом в разделе [модели, не зависимые от типа хранилища](#)

Рекомендуемые соглашения

Конфигурируйте:

- Приложение в целом с использованием Yaml файлов.
- Маршрутизатор с использованием аннотаций и Yaml файлов.
- Определения сервисов с использованием XML файлов.
- Документы и сущности с использованием аннотаций.

V Безопасность

Введение

Symfony и безопасность

Дистрибутив Symfony Standard Edition поставляется вместе с компонентом Security, бандлом SecurityBundle и (до версии 2.3) бандлом SecurityExtraBundle (@dbykadorov: фактически это был `jms/security-extra-bundle`, который, начиная с версии 2.3, не входит в стандартный дистрибутив и должен устанавливаться отдельно). Настраивая конфигурацию вашего приложения (используя `security.yml`), вы совершенно бесплатно (-: получаете следующие возможности:

- Одна или несколько областей, защищённых формой логина или HTTP-аутентификацией.
- Полную свободу в вопросе получения пользовательских данных (@dbykadorov: вероятно имеется в виду свобода выбора - откуда загружать пользовательские данные).
- Несколько способов хэширования пользовательских паролей.
- Способ для организации выхода.
- События, связанные с обеспечением безопасности, которые доставляются через диспетчер событий приложения.
- Авторизация пользователей, на основании их ролей.
- Настраиваемые обработчики хранения сессий (например, вы можете определить, где должны храниться данные сессий).
- Механизм списков контрола доступа (ACL), который вы можете использовать для выделения особых прав (таких как “редактирование”, “просмотр”, “удаление” и т.д.) для конкретных пользователей в отношении конкретных объектов.

Все функции, перечисленные выше, отлично реализованы, но объединение в них множества различных концепций (таких как : firewall, authentication listener, authentication provider, exception handler, entry point, ACL, ACE, security identity, object identity, etc.) может вызвать определённые затруднения при их использовании. Вот почему, если вы создаёте какое-либо приложение, в котором аутентифицированные пользователи отличаются от гостей или имеются ограничения по доступу на уровне объектов, вы должны взять себя в руки и прочитать об обеспечении безопасности в Symfony.

Вот несколько отличных ресурсов, с которых вы могли бы начать:

- Раздел **Безопасность** на сайте symfony.com, который даст вам основные представления по управлению аутентификацией и авторизацией в Symfony-приложении.
- Документация о **компоненте Security** (@dbykadorov: основа которой была также написана Маттиасом, автором этой книги), более детально описывающая все части компонента, которые так или иначе играют роль в обеспечении безопасности вашего приложения.

Хорошая новость для вас - там есть что почитать! После прочтения документации вам нужно будет сравнить список возможностей Symfony с вашим собственным (или вашего

приложения) списком потребностей, потому что плохая новость заключается в том, что многие вопросы там не рассматриваются.

Вот примеры функций и/или ситуаций, которые вам нужно будет создать и/или проверить самостоятельно:

- Очистка входных данных (для этого практически ничего не делается автоматически, так что этому вопросу будет выделен отдельный подраздел ниже).
- Автоматическая (интерактивная/по времени) инвалидация сессии.
- Мониторинг/предотвращение угона сессий.
- Предотвращение брутфорс-атак (простой подбор логинов-паролей) на формы логина.
- Хранение/управление типами пользователей, их ролями и группами ролей динамически.
- Отображение красивых страниц “у вас недостаточно прав для...”.
- Создание паролей, устойчивых к взлому.
- Предотвращение кэширования секретных данных браузерами.

Для того, чтобы реально понимать, что необходимо предпринять и какие решения будут лучшими, чтобы обломать зубы “плохим парням”, вам просто необходимо узнать о базовых принципах обеспечения безопасности PHP-приложений - конфигурировании веб-сервера в определённых случаях и собственно об обеспечении безопасности веб-приложения. По этим темам мнения людей широко расходятся, так что, после того как вы определите основные проблемы обеспечения безопасности, вам необходимо будет выбрать наилучшие решения, основываясь на мнениях разных людей. Я хочу дать вам наводку на очень хороший ресурс, фактически это книга, находящаяся в процессе написания - [Survive The Deep End: PHP Security](#), за авторством Pádraic Brady.

Когда же дело дойдёт до обсуждения дополнительных мер, связанных с безопасностью, которые не доступны по умолчанию в Symfony, я буду отправлять вас на мой собственный сайт, где я опубликовал несколько статей о том как [улучшить безопасность вашего приложения](#). Вы должны хорошо ориентироваться в вопросах, связанных с PHP, безопасностью в веб-приложениях и в Symfony. Не доверяйтесь слепо вашему фреймворку в таком важном вопросе. Вы должны не только уметь настроить вашу систему безопасности, но и понимать работает ли она и как она работает.

Цели: предотвращение и ограничение

Имеется 2 пути, при помощи которых вы можете усилить безопасность: во-первых вы можете попытаться предотвратить возникновение внештатных ситуаций, во-вторых вы можете принять такие меры, чтобы при возникновении внештатной ситуации, она так или иначе оставалась бы под контролем.

Краткое руководство [OWASP по практикам безопасного кодинга](#) описывает ситуацию следующим образом:

- 1 Злоумышленник взаимодействует с системой, которая может иметь уязвимость, которая в свою очередь может быть использована для нанесения ущерба.

Ваше Symfony приложение - это система, которая, как правило имеет уязвимости (за исключением случая, когда оно лишь отображает “Hello, World!”) и эти уязвимости могут быть использованы для преодоления защиты, которую вы воздвигли вокруг ваших (важных) данных. Хотя вы должны постараться сделать всё возможное, чтобы предотвратить взлом вашей системы, вы должны также думать и о том, что может произойти если это всё-таки случится. В конце концов, взлом может быть следствием похищения пароля, или даже брутфорса (@dbykadorov: от англ. “brute force” - дословно “грубая сила”, как правило подразумается подбор пароля путём простого перебора по словарю) учётной записи пользователя для работы с базой данных.

Что сможет получить хакер и как много времени потребуется ему, чтобы получить ещё больше? Какие данные могли быть скомпроментированы? Должны ли вы уведомить пользователей об атаке? Можете ли вы отследить атаку? И в случае взлома базы данных учётных записей возможно ли хэшированные пароли восстановить, используя брутфорс, за разумное время?

Например, если в вашем приложении есть система комментирования, которая позволяет пользователям оставлять комментарии прямо на странице, злоумышленник может вставить JavaScript код в комментарий. Если у вас не включено экранирование для текста комментариев, этот код будет размещён на странице так, как если бы это был код приложения. Такое поведение может стать серьёзной уязвимостью вашего приложения. Злоумышленник может внедрить в страницу код, который проверяет `document.cookie` на предмет наличия ID пользовательских сессий, так как они как раз хранятся в куках браузера. Это даже может быть причиной угона пользовательской сессии из его же собственного браузера.

Минимизация урона

Внедрение (или инъекция) JavaScript эксплоитов может нанести огромный урон, так как возможность получения сессии (любого) пользователя сама по себе является очень опасной уязвимостью, и становится вообще катастрофической, если пользователь является администратором. Для минимизации урона, вам требуется предпринять несколько мер по усилению безопасности приложения. Во-первых, вы должны сконфигурировать PHP таким образом, чтобы сессионная кука имела свойство HTTP-only. Это сделает невозможным чтение и запись этого кукиса при помощи JavaScript. Затем вы должны найти способ обнаруживать “угнанные” сессии и принудительно закрывать их. Есть также много других способов уменьшить возможность влома приложения, но они также помогают снизить возможный ущерб от взлома, если он всё-таки произошёл.

Оценка мер по обеспечению безопасности

Безусловно, вы должны всегда стараться находить и устранять уязвимости и стараться минимизировать ущерб от возможных (и даже на первый взгляд невозможных) атак. Но, что так же важно, вы должны иметь метрики безопасности, чтобы понимать, насколько

вы добились поставленных целей. Эти метрики помогут вам выбирать правильное решение при обнаружении проблем безопасности. Всегда критически оценивайте меры по обеспечению безопасности, которые вы предпринимаете. Иногда может случиться так, что вы принимаете меры, которые не помогают вообще, так как у вас уже используются более сильные меры по обеспечению безопасности. И также вы сможете увидеть, что одна незащищённая часть приложения может косвенным образом дать доступ злоумышленнику к другой, защищенной его части.

Перед тем как мы начнём...

Для обеспечения безопасности (веб) приложения можно выполнить множество разных вещей и, вероятно, вы не сможете реализовать их все. По большому счёту и не нужно стремиться к этому. Обсудите с вашей командой необходимые меры по обеспечению безопасности, а также такие меры, которые позволяют получить быстрый эффект с минимумом затрат. Также обсудите с вашим менеджером бюджет времени и важность безопасности для конкретного проекта. Обеспечение безопасности должно достигаться усилиями всей команды, оно также требует максимального понимания процесса и дисциплины. Так что убедитесь, что вся ваша команда думает одинаково, когда речь заходит о безопасности.

В следующих главах я покажу вам приёмы, которые помогут предотвратить возникновение внештатных ситуаций в ваших Symfony приложениях, а также расскажу как вы можете отслеживать и контролировать проблемные места в различных частях приложения.

Аутентификация и сессии

Как упоминалось ранее, фреймворк Symfony уже позаботился о многих вещах касательно входа (login) и выхода (logout) пользователей, а также в config.yml и security.yml имеются опции для настройки их параметров. Например, вы можете изменить некоторые настройки PHP, касающиеся безопасности, прямо из файла настроек config.yml:

```
1 framework:
2     session:
3         # имя сессионной куки
4         name: matthias_session
5         # сессионная кука не должна быть доступна для JavaScript
6         cookie_httponly: true
7         # сессионные данные должны "протухать" через n секунд (если не используются)
8         gc_maxlifetime: 3600
9         # "протухшие" сессионные данные будут очищены сборщиком мусора с вероятностью 1:10
10        gc_probability: 1
11        gc_divisor: 10
```

Symfony берёт на себя заботу о миграции сессии (фактически это смена ID сессии), когда вы логинитесь в систему, для предупреждения ситуации, в которой старая сессия (без аутентификации) получала бы дополнительные права (так как эта сессия может быть скомпрометирована в любое время). Symfony также берёт на себя инвалидацию (завершение) аутентифицированной сессии после выхода из приложения (для предотвращения её угона). Эти механизмы работают “из коробки”, но также могут быть настроены в файле конфигурации security.yml.

```
1 security:
2     # после аутентификации сессия будет мигрирована
3     session_fixation_strategy: migrate
4
5     firewalls:
6         secured_area:
7             logout:
8                 # аутентифицированная сессия будет недоступна после выхода
9                 invalidate_session: true
```

Инвалидация сессии

Угон сессии

Всё сказанное выше звучит классно, но кое-какие моменты мы ещё не рассмотрели. Представьте, что некто смог отследить ID сессии аутентифицированного пользователя и использовал его в своём сессионном кукисе для получения контроля над его сессией - никто об этом не узнает и злоумышленнику никто не сможет помешать. Это означает, что вам также необходимо проверять любые изменения в “сигнатуре” пользователя. Например, вы можете проверять IP адрес пользователя или его User Agent. Любое изменение в наблюдаемых характеристиках как минимум должно журнализироваться, чтобы была возможность мониторинга подозрительных ситуаций. Также вы можете потребовать у пользователя пройти аутентификацию заново и таким образом убедиться, что изменение сигнатуры не представляет угрозы вашему приложению.

```

1 namespace Matthias\SecurityBundle\EventListener;
2
3 use Matthias\SecurityBundle\Exception\UserSignatureChangedException;
4 use Symfony\Component\HttpKernel\Event\GetResponseEvent;
5 use Symfony\Component\HttpKernel\HttpKernelInterface;
6
7 class UserSignatureListener
8 {
9     public function onKernelRequest(GetResponseEvent $event)
10    {
11        if ($event->getRequestType() !== HttpKernelInterface::MASTER_REQUEST) {
12            return;
13        }
14
15        $request = $event->getRequest();
16
17        $clientIp = $request->getClientIp();
18
19        $userAgent = $request->headers->get('user-agent');
20
21        if (...) {
22            throw new UserSignatureChangedException();
23        }
24    }
25 }

```

Зарегистрировать этот слушатель можно, определив сервис следующего вида:

```

1 <service id="matthias_security.user_signature_listener"
2   class="Matthias\SecurityBundle\EventListener\UserSignatureListener">
3   <tag
4     name="kernel.event_listener"
5     event="kernel.request"
6     priority="100"
7     method="onKernelRequest" />
8 </service>

```

Иключение `UserSignatureChangedException`, которое будет сгенерировано в слушателе должно быть обработано в слушателе исключений (событие `KernelEvents::EXCEPTION`), который установит соответствующие объект ответа, например, `RedirectResponse` с перенаправлением на страницу аутентификации (логина).

Долгоиграющие сессии

Представьте, аутентифицированный пользователь вашего приложения долгое время не выполнял никаких действий, но при этом держал браузер открытым. Сессионные данные не запрашивались некоторое время и их время жизни практически истекло. И тут, пользователь обновляет страницу. Сессионный кукис, содержащий ID сессии настроен таким образом, что его срок действия истекает при закрытии браузера (`cookie_lifetime = 0`). Таким образом он будет всё-ещё валиден, так же как и сессионные данные (время жизни которых *почти* истекло), таким образом пользователь может продолжить работу со старой сессией как ни в чём не бывало. Таким образом, в данной ситуации ничто не мешает пользователю иметь вечную сессию. Вам, вероятно, хотелось бы инвалидировать такие долгоиграющие сессии, основываясь на дате, когда они были в последний раз использованы. У Symfony

нет встроенных средств для выполнения такой операции, однако же не составляет труда реализовать ваш собственный инвалидатор. Сессия имеет так называемый MetadataBag, который содержит Unix таймштампы с датами, когда сессия была создана и когда её данные были изменены в последний раз.

```

1 use Symfony\Component\HttpKernel\Event\GetResponseEvent;
2 use Symfony\Component\HttpKernel\HttpKernelInterface;
3
4 class SessionAgeListener
5 {
6     public function onKernelRequest(GetResponseEvent $event)
7     {
8         if ($event->getRequestType() !== HttpKernelInterface::MASTER_REQUEST) {
9             return;
10        }
11
12        $session = $event->getRequest()->getSession();
13        $metadataBag = $session->getMetadataBag();
14
15        $lastUsed = $metadataBag->getLastUsed();
16        if ($lastUsed === null) {
17            // сессия только что была создана
18            return;
19        }
20
21        $createdAt = $metadataBag->getCreated();
22
23        $now = time();
24
25        // $now, $lastUsed и $createdAt - это Unix таймштампы
26
27        // если сессия пытается "пробудиться" после того как прошло слишком много времени:
28        $session->invalidate();
29
30        // создаём ответ, информирующий пользователя о том что произошло:
31        $event->setResponse(...);
32    }
33 }
```

Сервис для активации SessionAgeListener выглядит следующим образом:

```

1 <service id="matthias_security.verify_session_listener"
2   class="Matthias\SecurityBundle\Event\Listener\SessionAgeListener">
3   <tag
4     name="kernel.event_listener"
5     event="kernel.request"
6     priority="100"
7     method="onKernelRequest" />
8 </service>
```

Журналирование информации, относящейся к безопасности

Всякий раз, когда происходит что-то подозрительное (пользователь внезапно сменил IP адрес или попытался залогиниться с неправильными учётными данными 100 раз за последнюю минуту и т.д.) вы должны записать информацию об этом в лог приложения. Кроме того, должно быть сразу видно, что эта запись относится к безопасности. Для этого, чтобы создать соответствующий сервис журналирования - присвойте ему таг monolog.logger

```
1 <service id="matthias_security.verify_session_listener"
2   class="...">
3   <argument type="service" id="logger" on-invalid="null" />
4   <tag name="monolog.logger" channel="security" />
5 </service>
6 {lang="xml"}
```

Всякий раз, когда вы используете этот сервис:

```
1 $this->logger->warning('Old session reused')
```

В лог-файле будет создана такая запись:

```
1 [2013-07-06 16:35:45] security.WARNING: Old session reused
```

Дизайн контроллеров

Когда речь заходит о дизайне контроллеров, я рекомендую создавать небольшие классы контроллеров. Прежде всего это означает группировку логически связанных действий в одном классе контроллера, а также создание небольших методов для действий, которые не содержат в себе тяжеловесной логики и максимум два возможных пути исполнения. Это, в свою очередь означает, что в конце концов у вас будет много разных контроллеров, однако же этот подход имеет много преимуществ, которые также связаны и с безопасностью:

1. Когда вам нужно изменить какое-то, связанное с пользователем, поведение, вы можете очень быстро найти путь к контроллеру по URL и имени маршрута.
2. Когда некоторые контроллеры будут требовать повышенного внимания с точки зрения безопасности, вы легко можете определить их и концентрировать защитную логику в одном месте.

Для иллюстрации второго утверждения приведу пример. Положим, у вас есть контроллер, имеющий отношение к персональным настройкам пользователя, но он зависит как от строго конфиденциальных данных, таких как информация о счёте, платёжной информации (номера кредитных карт), так и от публично доступных, таких как никнейм или аватар. Я рекомендую разделять страницы, относящиеся к просмотру и редактированию этих данных по разным контроллерам, наделяя их простыми и понятными именами. Внутри контроллеров каждое действие также должно иметь понятное имя, которое может быть длиннее, чем те, к которым вы привыкли, например AccountController::editAccountInformation (вместо короткого "edit") и PaymentController::listCreditCards. Это позволит вам уравнять ваш уровень бдительности с уровнем, необходимым для конкретного кода, над которым вы работаете (@dbykadorov: мне кажется, Маттиас имеет в виду, что Foo:edit и Bar:list не отражают реальной важности выполняемого действия, а в предлагаемой им нотации взгляд сразу зацепится за editAccountInformation или listCreditCards).

Более того, при взгляде на код сразу должно быть понятно, какой контроллер используется для модификации состояния приложения (например, сохранения или удаления данных). Вы должны явно указывать HTTP методы, разрешённые для действий. В этом смысле, страницы, которые могут быть запрошены с помощью метода GET - безобидные, а страницы, на которых пользователи могут выполнять POST запросы, таковыми не являются.

```
1 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
2
3 /**
4 * @Method("GET")
5 */
6 public function viewPublicProfileAction()
7 {
8 }
9
10 /**
11 * @Method("GET")
12 */
13 public function editPublicProfileAction()
14 {
```

```
15 }
16 /**
17 * @Method("POST")
18 */
19
20 public function updatePublicProfileAction()
21 {
22 }
```

И, конечно же, заведомо нехорошо, когда кто-то может видеть форму редактирования некоего объекта, в том случае, когда к этому объекту он не имеет необходимых прав доступа. И ещё хуже, когда он может реально изменить его.

Защита действий в контроллерах

Имеется немало способов, при помощи которых вы можете защитить действия. Код действия сам по себе иногда может генерировать исключение AccessDeniedException, и, конечно же, вы должны проверять, что конкретный объект принадлежит кому-то конкретному или может быть им модифицирован (либо на основании того, что владелец хранится как свойство объекта, либо на основании того, что права доступа регистрируются с помощью ACL). Но вы также должны каким-либо образом реализовать управление ролями. Использовать роли в Symfony очень легко. Только подумайте о новой роли - и вот, она уже существует. Когда существующие роли должны включать в себя новую роль, добавьте её в т.н. “иерархию ролей” в security.yml.

Очень важно начать составлять список ролей для любого контроллера, который требует, чтобы пользователь вошёл в приложение. И для этого несколько (хороших) причин. Во-первых, когда речь идет о безопасности, вы должны придерживаться принципа “наименьших привилегий”. Это означает, что по умолчанию аутентифицированный пользователь не может делать ровным счётом ничего (ну может быть может изменить свой пароль...). Вы, система или администратор могут предоставить ему некоторые дополнительные права. Вы можете использовать все виды выражений, касающиеся ролей в действиях, используя аннотацию @PreAuthorize, но в большинстве ситуаций, с которыми я столкнулся, достаточно просто воспользоваться аннотацией @Secure:

```
1 use JMS\SecurityExtraBundle\Annotation\Secure;
2
3 /**
4 * @Secure("ROLE_PAGE_EDITOR")
5 */
6 public function editPageAction()
7 {
8 }
```

Часто случается так, что разные типы пользователей должны иметь доступ к странице editPageAction, например, пользователь с ролью ROLE_ADMINISTRATOR, а также кто-то с ролью ROLE_CONTENT_MANAGER. Решением в данном случае будет не добавление дополнительных ролей в аннотацию @Secure (таким образом - @Secure({ "ROLE_ADMINISTRATOR", "ROLE_CONTENT_MANAGER"})), а доработка иерархии ролей следующим образом:

```
1 security:
2     role_hierarchy:
3         ROLE_ADMINISTRATOR: [ROLE_PAGE_EDITOR]
4         ROLE_CONTENT_MANAGER: [ROLE_PAGE_EDITOR]
```

Когда речи идёт о ролях, запомните:

- Роли не константны: вы вольны добавлять новые и (почти) вольны удалять их (по сути это просто строки)
- Роли работают лучше, если роль описывает пользователя, которому будет назначена эта роль (имен ролей в идеале должны оканчиваться на “MANAGER”, “EDITOR”, “MODERATOR, и т.д. и т.п.).

Размещение контроллеров за файрволлом

Теперь представьте, что у вас есть много контроллеров в каком-то бандле, например, SettingsBundle. Теперь вам нужно импортировать их все в файл routing.yml в этом бандле:

```
1 SettingsBundleControllers:
2     resource: "@SettingsBundle/Controller/"
3     type: annotation
```

Этот файл может быть импортирован разом в файле routing.yml, принадлежащему приложению (app/config/routing.yml). Это позволит вам определить префикс для всех маршрутов из SettingsBundle:

```
1 SettingsBundle:
2     resource: "@SettingsBundle/Resources/config/routing.yml"
3     type: yaml
4     prefix: /settings
```

Используя этот префикс, вы можете с лёгкостью определять множество ролей, необходимых для любого URI, который начинается с /settings:

```
1 security:
2     access_control:
3         - { path: ^/settings, roles: [IS_AUTHENTICATED_FULLY] }
```

Конфигурация выше означает, что вам необходимо быть полностью аутентифицированными, для того, чтобы выполнить любое действие, относящееся к натройкам (если вы вошли при помощи куки “remember me”, это также не даст вам необходимых прав доступа). Это хороший приём и у вас теперь есть единая точка для контроля безопасности.

Валидация входных данных

Любые входные данные, поступающие в ваше приложение извне, должны быть проверены и обработаны, прежде чем они будут реально использоваться или же будут сохранены тем или иным образом.

Безопасные формы

Symfony включает в себя два важных инструмента, которые вы можете использовать в борьбе за безопасные входные данные: это компоненты Form и Validator. Большинство Symfony разработчиков уже доверили свои “жизни” компоненту форм:

```
1 if ($request->isMethod('POST')) {  
2     $form->bind($request);  
3     if ($form->isValid()) {  
4         // persist!  
5     }  
6 }
```

И это доверие вполне обосновано, так как компоненты Form и Validator сами по себе классные и им можно доверить безопасность ваших форм. Но, вы не обязаны им слепо доверяться - в вопросах безопасности доверять “на слово” ничему нельзя. Ниже мы разберём как обрести уверенность, работая с этими двумя компонентами.

HTML5-валидация

Мысленно возвращаясь к временам, когда мы писали наши PHP приложения вручную от начала до конца, уже тогда нам нужно было заниматься валидацией данных форм. Например, если мы не хотели, чтобы пользователи оставляли пустым поле “name”, нужно было после отправки формы проверить присланное значение и вывести сообщение об ошибке, если поле не было заполнено. Сейчас же, когда мы собираем Symfony-форму из полей (которые, в терминах компонента, формально тоже являются формами) нам достаточно установить для поля свойство “required” в “true”. Когда мы откроем нашу форму в браузере и попробуем отправить её не заполняя, вы увидите красивое сообщение об ошибке (которое является особенностью HTML5) и форма не будет отправлена, пока обязательные значения не будут заполнены. Тем не менее, если вы отключите клиентскую HTML5-валидацию, добавив атрибут “novalidate” в тег формы, вы увидите, что по умолчанию серверной валидации у нашей формы нет. Таким образом, первым делом при создании любой формы вам необходимо (возможно только для DEV окружения) отключить HTML5 валидацию:

```
1 <form{% if app.debug %} novalidate="true"{% endif %}>
```

Теперь вы можете тестировать серверную валидацию ваших форм.

@dbykadorov: пользуясь правом переводчика, хочу отметить, что я не люблю использовать HTML5-валидацию и рекомендую ВСЕГДА отключать её для ВСЕХ

форм сразу при их создании (а не только в DEV окружении). Как отметил выше Маттиас, по умолчанию в формах имеется минимум валидации и нужно всегда опираться на те правила, которые вы принудительно определите для вашей формы.

Ограничения валидатора

Как правило, у вас имеется некий доменний объект, назовём его сущностью, который вам нужно создать или модифицировать при помощи формы. В этом случае вам необходимо указать имя класса этой сущности в опции “`data_class`” соответствующей формы. В методе класса формы `buildForm` при помощи экземпляра класса `FormBuilder` вы можете добавить поля, соответствующие атрибутам сущности. Теперь, для того, чтобы выполнить валидацию данных формы, вам необходимо добавить ограничения к этим атрибутам (для этого удобно использовать аннотации). Этот процесс [очень хорошо описан](#) в официальной документации.

```
1 namespace Matthias\AccountBundle\Entity;
2
3 use Symfony\Component\Validator\Constraints as Assert;
4
5 class User
6 {
7     /**
8      * @Assert\Email()
9      * @Assert\NotBlank()
10     */
11     private $emailAddress;
12 }
```

Пользовательские ограничения валидации

Порой стандартных ограничений валидатора, включенных в компонент `Symfony Validator Component`, бывает недостаточно для проверки ваших данных. Но не отчаивайтесь, просто создайте [ваше собственное ограничение валидации](#) и соответствующий валидатор.

Формы без сущности

Если у вас есть данные, которые не соотносятся один к одному с какой-либо сущностью, в этом случае документация предлагает не использовать класс данных и, фактически, создавать форму сразу в контроллере:

```

1 public function contactAction(Request $request)
2 {
3     $defaultData = array('message' => 'Type your message here');
4     $form = $this->createFormBuilder($defaultData)
5         ->add('name', 'text')
6         ->add('email', 'email')
7         ->add('message', 'textarea')
8         ->getForm();
9
10    $form->handleRequest($request);
11
12    if ($form->isValid()) {
13        // $data тут будет представлен в виде массива с ключами "name", "email", и "message"
14        $data = $form->getData();
15    }
16 }

```

Для меня этот звучит как плохая идея по двум причинам: во-первых, вы всегда должны определять формы при помощи соответствующих классов, наследуемых от AbstractType, например так:

```

1 use Symfony\Component\Form\AbstractType;
2 use Symfony\Component\Form\FormBuilderInterface;
3
4 class ContactFormType extends AbstractType
5 {
6     public function buildForm(
7         FormBuilderInterface $builder,
8         array $options
9     ) {
10        $builder
11            ->add('name', 'text')
12            ->add('email', 'email')
13            ->add('message', 'textarea')
14        ;
15    }
16
17    public function getName()
18    {
19        return 'contact_form';
20    }
21 }

```

Такой подход позволит поместить логику, относящуюся к форме, в одном месте, что сделает возможным её повторное использование и упростит обслуживание.

Во-вторых, вы всегда должны указывать в классе формы её data_class. Контактная форма выше может и не сохраняться в базу данных, но всё же она имеет определенную монолитную структуру и вполне определённое назначение. В этом случае нужно выбрать подходящее наименование для объекта, который будет содержать данные этой формы, например, ContactDetails, создать класс с требуемыми полями и добавить ограничения для каждого поля, чтобы можно было убедиться в консистентности данных:

```

1 use Symfony\Component\Validator\Constraints as Assert;
2
3 class ContactDetails
4 {
5     /**
6      * @Assert\NotBlank
7      */
8     private $name;
9
10    /**
11     * @Assert\NotBlank
12     * @Assert\Email
13     */
14    private $email;
15
16    /**
17     * @Assert\NotBlank
18     */
19    private $message;
20
21    // также добавьте get- и set-методы для каждого поля
22 }

```

Теперь можно указать этот класс в опции `data_class` формы `ContactFormType`:

```

1 use Symfony\Component\OptionsResolver\OptionsResolverInterface;
2
3 class ContactFormType extends AbstractType
4 {
5     public function setDefaultOptions(OptionsResolverInterface $resolver)
6     {
7         $resolver->setDefaults(array(
8             'data_class' => 'LifeOnline\ContactBundle\Model>ContactDetails'
9         ));
10    }
11 }

```

Теперь ваши данные в намного лучшей форме, так как они инкапсулированы, валидированы и вообще находятся под вашим контролем. Вы можете:

- предотвращать появление вредоносных данных, фильтруя данные в `set`-методах:

```

1 class ContactDetails
2 {
3     public function setMessage($message)
4     {
5         $this->message = strip_tags($message);
6     }
7 }

```

- легко задавать значения по умолчанию (без необходимости знать структуру данных):

```

1 public function contactAction(Request $request)
2 {
3     $contactDetails = ContactDetails::createForUser($this->getUser());
4
5     $form = $this->createForm(new ContactFormType(), $contactDetails);
6
7     // ...
8 }
```

- работать с объектом известного типа после привязки и валидации формы:

```

1 public function contactAction(Request $request)
2 {
3     $form = $this->createForm(new ContactFormType());
4
5     if ($request->isMethod('POST')) {
6         $form->bind($request);
7         if ($form->isValid()) {
8             $contactDetails = $form->getData();
9
10            // переменная $contactDetails является экземпляром ContactDetails
11        }
12    }
13 }
```

Резюмируем: все формы в вашем приложении:

- Должны иметь опцию `data_class`, указывающую на определённый класс.
- Должны быть определены в своих собственных классах, наследуемых от `AbstractType`.
- Предпочтительно должны быть определены как сервисы, для повторного использования.

Валидация значений из запроса

Объект запроса в основе своей является лишь обёрткой для суперглобальных массивов PHP. В файле /web/app.php вы можете увидеть, как именно создаётся объект запроса:

```
1 $request = Request::createFromGlobals();
```

Внутри этого метода вы можете найти такую строку:

```
1 $request = new static($_GET, $_POST, array(), $_COOKIE, $_FILES, $_SERVER);
```

`@dbykadorov`: в более старших версиях Symfony, она будет иметь немного другой вид:

```
1 $request = self::createRequestFromFactory($_GET, $_POST, array(), $_COOKIE, $_FILES, $server);
```

Таким образом, хотя вы можете чувствовать себя более защищёнными, получая данные из объекта запроса `$request->query->get('page')`, по сути этот метод не более защищён, чем простой вызов `$_GET['page']`. Хотя в контексте Symfony следует всегда использовать объект запроса для получения его данных, но вы всегда должны относиться с подозрением к ним, валидировать их и принудительно переводить в нужный формат. Строгая типизация и проверка допустимых значений и диапазонов строго обязательна.

Атрибуты запроса

Параметры маршрута

Прежде всего, если шаблон URI содержит подстановки, например id в /comment/{id}, и некоторый запрос соответствует этому шаблону, RouterListener должен убедиться, что все параметры скопированы в объект запроса в качестве атрибутов. Эти атрибуты будут использованы классом ControllerResolver для сбора аргументов контроллера, основываясь на именах параметров и подсказками типов. Вы можете ознакомиться с разделом книги [События, приводящие к ответу](#), если хотите узнать детали этого процесса.

Так как большинство аргументов контроллера просто копируются напрямую (более или менее) из URI из запроса, вы должны очень осторожны, используя их. Во-первых, вы должны позаботиться о требованиях к параметрам маршрута (типа id). Требования определяются в виде регулярных выражений. По умолчанию требованием для параметра маршрута является [^ /]+: они могут содержать любые символы, кроме слэша. Поэтому вы должны определить более жёсткие требования, например \w+, что будет означать как минимум “слово” (которое может содержать латинские буквы a-z, A-Z, цифры 0-9 и подчерк _) или \d+, что будет означать как минимум одну цифру (0-9). Если вы ещё не знакомы с регулярными выражениями, вы срочно должны познакомиться с ними (как вы могли заметить ранее, они также используются при работе с путями, определяемыми в security.yml). Ниже вы можете найти несколько примеров маршрутов с ограничениями:

```

1 /**
2  * id может быть только лишь числом:
3  * @Route("/comment/{id}", requirements={"id"="\d+"})
4 */
5
6 /**
7  * alias может содержать как минимум одну букву в нижнем регистре, цифру или тире:
8  * @Route("/user/{alias}", requirements={"alias"="[a-z0-9\-\-]+"})
9 */
10
11 /**
12  * type может иметь лишь значения 'commercial' или 'non-commercial':
13  * @Route("/create-account/{type}", requirements={
14  *     "type"="commercial|non-commercial"
15  * })
16 */

```

Важно проверять корректность этих значений на ранних стадиях, иначе некорректные значения могут быть использованы в качестве аргументов при вызове методов сервисов, где они могут вызвать InvalidArgumentExceptions или, того хуже, разного рода необъяснимые ошибки.

Query (GET) и request (POST) параметры

В жизни случается много различных ситуаций, когда вам потребуется некоторая гибкость в отношении параметров запроса в контроллере: вы можете использовать параметры query в вашем URI (например, ?page=1, т.е. GET параметры), или же параметры запроса, отправленные в его теле (т.е. POST параметры). Однако, ни те ни другие значения не заслуживают

доверия. GET и POST данные легко могут быть подменены на такие значения, которые вы не ожидаете получить. Следовательно, когда вы получаете эти данные из объекта запроса:

- Выполняйте валидацию при помощи компонентов Form и Validator, используя Form::bind() (@dbykadorov: в более поздних версиях Symfony - Form::handleRequest())
- Или:
 - Приводите значения к типу, который вы ожидаете (чаще всего это будет строка или целое число)
 - Выполните валидацию этих значений самостоятельно при помощи метода validateValue() из сервиса validator. Вам может потребоваться проверить диапазон значений (“тут должно быть как минимум значение 1”) или соответствие одной из опций (“тут должно быть либо значение ‘commercial’, либо ‘non-commercial’”).

Например:

```

1 use Symfony\Component\Validator\Constraints\Choice;
2
3 public function createAccountAction()
4 {
5     $userTypeConstraint = new Choice(array(
6         'choices' => array('commercial', 'non-commercial')
7     );
8
9     $errorList = $this->get('validator')->validateValue(
10        $request->query->get('userType'),
11        $userTypeConstraint
12    );
13
14    if (count($errorList) == 0) {
15        // тип пользователя верный
16    }
17 }
```

Не используйте \$request->get()!

Не используйте \$request->get(), потому что внутри он выглядит вот так (в версии Symfony 2.3):

```

1 public function get($key, $default = null, $deep = false)
2 {
3     return $this->query->get($key,
4         $this->attributes->get($key,
5             $this->request->get($key, $default, $deep),
6             $deep),
7             $deep);
8 }
```

@dbykadorov: в Symfony 2.8+ этот метод выглядит по-опрятнее.

```

1 public function get($key, $default = null)
2 {
3     if ($this !== $result = $this->attributes->get($key, $this)) {
4         return $result;
5     }
6 }
```

```

7     if ($this !== $result = $this->query->get($key, $this)) {
8         return $result;
9     }
10    if ($this !== $result = $this->request->get($key, $this)) {
11        return $result;
12    }
13    return $default;
14 }
15 }
16 }
```

Тем не менее, я поддерживаю Маттиаса в его предостережении. Безусловно удобно получать параметры запроса, не задумываясь, GET они или POST. Но, с точки зрения безопасности, вы всегда должны контролировать контекст выполнения вашего запроса. Было бы странно ожидать данные, отправленные через POST, но получать их фактически через GET.

Всегда извлекайте данные специфическим для них способом, будь то query string, post data или прочие атрибуты запроса. Если ранее вы использовали суперглобальные переменные, то теперь просто обращайтесь к соответствующему хранилищу параметров (aka “parameter bag”) класса запроса:

- `$_GET['key']` теперь будет `$request->query->get('key')`
- `$_POST['key']` теперь будет `$request->request->get('key')`

Используем ParamFetcher

Имеется также ещё один инструмент, который я бы хотел упомянуть. Он является частью бандла

[FOSRestBundle](#), который предоставляет мощный инструментарий для создания REST-подобных вебсервисов. Он также предоставляет очень удобный способ валидации параметров запроса путём добавления дополнительных настроек в форму или же аннотаций для каждого параметра запроса, который вам будет нужен. Например, вот как вы можете сконфигурировать параметр “page”, который должен содержать только цифры, и по умолчанию равняется 1:

```

1 use FOS\RestBundle\Request\ParamFetcher;
2 use FOS\RestBundle\Controller\Annotations\QueryParam;
3
4 /**
5  * @QueryParam(
6  *     name="page",
7  *     requirements="\d+",
8  *     default="1",
9  *     description="Page number"
10 )
11 */
12 public function listAction(ParamFetcher $paramFetcher)
13 {
14     $page = $paramFetcher->get('page');
15 }
```

Этот способ работает похожим образом и для POST данных, которые не используются в какой-либо форме:

```
1 use FOS\\RestBundle\\Request\\ParamFetcher;
2 use FOS\\RestBundle\\Controller\\Annotations\\RequestParam;
3
4 /**
5 * @RequestParam(name="username", requirements="\w+")
6 */
7 public function deleteAccountAction(ParamFetcher $paramFetcher)
8 {
9     ...
10 }
```

Даже если ваше приложение не имеет никакого REST API, вы всё-равно можете установить FOSRestBundle без каких-либо негативных последствий. Просто убедитесь, что вы отключили все его слушатели, кроме ParamFetcherListener:

```
1 # in /app/config/config.yml
2 fos_rest:
3     param_fetcher_listener: true
```

Очистка HTML

У вас в приложении, вероятно, есть формы, которые позволяют пользователям форматировать текст с использованием редакторов rich text (или wysiwyg), делать его жирным, курсивом, подчёркивать, и может даже позволяет пользователям втасливать их собственные ссылки в текст. Плохая новость для вас состоит в том, что это конечно же очень небезопасная функция. Вы должны очень сильно озабочиться об ограничениях на доступные пользователям HTML-таги и тем более атрибуты, которое могут быть даже более опасны, так как могут содержать JavaScript. И не думайте, что `alert('Hi!')`; это худшая вещь, которую пользователь может провернуть через ваш редактор! Даже незакрытый таг может неделать бед - представьте, если в вашей HTML разметке случайно появится лишний закрывающий таг `</div>` (или же поигратесь с любым сайтом через Dev tools браузера).

Во-первых, не полагайтесь на языки разметки типа Markdown для решения проблемы с инъекцией HTML. Спецификация Markdown чётко и однозначно утверждает, что любая HTML разметка в исходном тексте должна оставаться нетронутой и добавляться “как есть” в сгенерированный вывод. Во-вторых, не полагайтесь на второй аргумент функции `strip_tags`, который позволяет оставлять лишь выбранные таги, так как эта функция будет также пропускать и все атрибуты внутри этих тагов. В-третьих: даже не мечтайте написать свои собственные регулярные выражения для того, чтобы разрешить использование некоторых тагов и атрибутов. Всегда будет оставаться возможность хакнуть ваши самописные правила. Вместо этого используйте [HTMLPurifier](#) и настройте его под вашу конкретную ситуацию.

Имеется также замечательный бандл, которые интегрирует HTMLPurifier в Symfony приложение: [ExerciseHTMLPurifierBundle](#) (@dbykadorov: бандл до сих пор существует и поддерживает как Symfony 2., так и Symfony 3.). Он определяет отдельный сервис для каждой конфигурации, которую вы определите в config.yml. Он также поддерживает автоматическую фильтрацию значений форм.

Автоматизация очистки

Проблема такова: в каждом Symfony-приложении вы должны вручную заботится об очистке входных данных. Было бы здорово, если бы все атрибуты запроса фильтровались бы автоматически на основании некоторых правил или же когда формы или сущности должны были бы содержать только стопроцентно-чистые значения. К сожалению, прямо сейчас в мире open source я не знаком ни с чем, что бы могло выполнить эту задачу =(Прототипом такого функционала может быть бандл [DMSFilterBundle](#) (@dbykadorov: тоже жив и поддерживает как Symfony 2., так и Symfony 3.), но его текущее состояние не позволяет обрабатывать все нужные ситуации. Он фильтрует только входные данные форм для корневого объекта. Он также не фильтрует данные, которые устанавливаются у сущности вручную.

Экранирование вывода

Twig

Symfony разработчики говорят так:

- 1 Если вы используете Twig, экранирование вывода по-умолчанию активировано и вы защищены.

Каждый, кто прочёт такое заявление будет нескованно рад, за исключением тех, кто реально озабочен безопасностью своих приложений и кто скептически относится к подобным заявлениям. Нет такого автоматического экранирования вывода, которое работало бы всегда и в любой ситуации. По умолчанию экранирование вывода в Symfony подразумевает, что вы выполняете рендеринг в HTML на уровне элементов. Также Twig следует многим специальным (хотя и безопасным) правилам автоэкранирования, о которых вам необходимо знать. Прочтите об этом в разделе [Документация Twig для разработчиков](#).

Контекст экранирования

Наиболее выжным тут является предположение о том, что контекстом по умолчанию является `html`. Но это не единственный контекст, который может быть использован в вашем приложении. Вам нужно понять, какие переменные в каком контексте выводятся (или должны выводиться). Twig “из коробки” поддерживает следующие контексты:

- `html` - когда переменная рендерится на уровне HTML элементов,
- `html_attr` - когда переменная рендерится внутри HTML атрибутов,
- `js` - когда переменная рендерится внутри JavaScript кода,
- `css` - когда переменная рендерится внутри CSS стилей,
- `url` - когда рендерится часть URL (например, параметр `query_string`)

Когда вы определитесь с контекстом и если он отличен от HTML, вам необходимо будет воспользоваться экранирующим фильтром Twig (`escape`):

- 1 `{{ someVariable|escape('js') }}`

Экранирование вывода функций

Вы должны всегда быть на страже, когда речь идёт про авто-экранирование. Особенно, если вы создаёте ваши собственные Twig-функции или фильтры. Вывод ваших функций будет также автоматически экранироваться в текущем контексте (по умолчанию - `html`):

```

1 class MyTwigExtension extends \Twig_Extension
2 {
3     public function getFunctions()
4     {
5         return array(
6             \Twig_SimpleFunction('important', function($thing) {
7                 return sprintf(
8                     '<strong>%s</strong> is important to me',
9                     $thing
10                );
11            });
12        );
13    }
14 }
```

Результат работы этой функции будет автоэкранирован, вызов функции:

```
1 {{ important('My family') }}
```

Выведет следующее:

```
1 <strong>My family</strong> is important to me
```

Если же вы добавите опцию `is_safe` при определении функции в вашем расширении Twig, экранирование выполнено не будет:

```

1 class MyTwigExtension extends \Twig_Extension
2 {
3     public function getFunctions()
4     {
5         return array(
6             \Twig_SimpleFunction('important', function($thing) {
7                 ...
8             }, array(
9                 'is_safe' => array('html')
10            )
11        );
12    }
13 }
```

Теперь вывод экранироваться не будет.

Экранирование аргументов функций

Когда мы отмечаем вывод функции как безопасный, в то же время мы включаем входные аргументы в этот же вывод и опять получаем небезопасную функцию:

```
1 {{ important('<script>alert("Security")</script>') }}
```

На выходе мы получим:

```
1 <strong><script>alert("Security")</script></strong> is important to me
```

Конечно же вы обычно не помещаете такой код в ваши шаблоны, но большую часть времени вы будете использовать в качестве аргументов Twig функций данные, предоставленные пользователями (и помните, что и вашей базе данных большинство данных скорее всего будут результатом сохранения данных, предоставленных пользователями). И такие данные вообще говоря могут быть всем чем угодно, если они предварительно не были должным образом очищены.

Итак, для нашего случая есть простое решение - добавить опцию `pre_escape`:

```
1 class MyTwigExtension extends \Twig_Extension
2 {
3     public function getFunctions()
4     {
5         return array(
6             \Twig_SimpleFunction('important', function($thing) {
7                 ...
8                 }, array(
9                     'is_safe' => array('html'),
10                    'pre_escape' => 'html'
11                )
12            );
13        }
14    }
```

Теперь все аргументы функции будут предварительно экранированы для указанного контекста.

Опасности `raw` фильтра

Вы можете полностью отменить автоэкранирование, поместив в конце выражения фильтр `raw`:

```
1 {{ message|raw }}
```

В этом случае ничего не будет экранировано, и если переменная `message` содержит какой-либо HTML код, он будет отображён “как есть”. Это означает, что вы должны быть на 100 процентов уверены, что сделали всё необходимое для предотвращения инцидентов. В большинстве случаев это означает следующее: вы должны быть уверены, что эта переменная прошла через какой-либо процесс очистки (например через очистку с использованием HTMLPurifier, о нём было рассказано в конце предыдущей части). Или же это означает что источнику данных можно доверять, например если это импортированный код, написанный другим разработчиком (и даже лучше, если это выполняется в дополнение к очистке). В то же время значение “`textarea`”, сохраняемое администратором не может считаться доверенным, потому что, если система безопасности вашего сайта так или иначе будет скомпроментирована, злоумышленник может получить админ-доступ к системе, что позволит ему внедрить вредоносный код, которые затронет множество пользователей вашего приложения. И поверьте, я не запугиваю вас, а лишь хочу чтобы вы были осторожны.

Проверка безопасности Twig

Регулярно проверяйте ваши Twig-функции и фильтры на предмет использования `is_safe` и одновременном отсутствии опции `pre_escape`. Также проверяйте, что все переменные, который употребляются с фильтром `raw` получаются из проверенных источников.

Быть скрытным

В многих случаях взлом системы подразумевает получение максимально возможного понимания того как эта система работает, понимать её уязвимые, а следовательно пригодные для взлома части. Таким образом вам нужно быть весьма осторожными в плане того, что вы вообще показываете во вне.

Маскируйте ошибки аутентификации

Документация Symfony содержит очень плохой с точки зрения безопасности пример, когда переменная с текстом исключения показывается на форме аутентификации (тут переменная error это экземпляр объекта Exception):

```
1  {% if error %}  
2    <div>{{ error.message }}</div>  
3  {% endif %}
```

@dbykadorov

Если мне не изменяет память, с версии 2.6 документация скорректирована в соответствии с изменениями компонента Security и не является таким открытым файлом. Маттиас вероятно имеет в виду эту часть документации: [Как использовать традиционную форму логина](#)

Сразу хочется настроить вас на “мудрое использование сообщений об исключениях”, так как они могут содержать очень важную с точки зрения безопасности информацию. И так скорее всего и будет, как я неоднократно убеждался. Иногда, когда вы захотите отобразить сообщение исключения напрямую, в шаблон может быть помещена информация об ошибке работы с базой данных, включая описание схемы данных, что совершенно небезопасно, так как даёт потенциальному злоумышленнику информацию о внутреннем устройстве вашей схемы данных.

К счастью, все исключения, унаследованные от AuthenticationException имеют метод getMessageKey. При вызове этого метода вы получите простое сообщение об ошибке, без каких-то критичных деталей о самом исключении. Это будет простое сообщение, типа “Неверные учетные данные.” или “Аутентификация не может быть выполнена вследствие системных проблем.”.

Таким образом, правильный шаблон формы логина, который отображает ошибки будет таким:

```
1  {% if error %}  
2    <div>{{ error.messageKey }}</div>  
3  {% endif %}
```

Предотвращайте отображение исключений

Если в вашем приложении имеются такие места, где вы хотите использовать исключения для отображения сообщений об ошибках пользователям, я могу вам предложить для этого следующие стратегии. Вы можете использовать приём описанный выше (использовать метод `getMessageKey`, который будет возвращать сообщение об ошибке, которое можно без опаски показывать пользователям). Вы также можете обернуть низкоуровневые исключения более высокоуровневыми и устанавливать новое сообщение об ошибке, которое можно показывать пользователям. Вам также надо побеспокоиться о базовых низкоуровневых исключениях, которые могут возникать на глубоких уровнях абстракции и которые вы по ошибке можете принять за одно из собственных исключений. Также вы можете помечать некоторые исключения как безопасные для отображения их пользователем, фильтруя их:

```
1 $error = null;
2
3 try {
4     // делаем что-то очень опасное
5     ...
6 } catch (SomeSpecificException $exception) {
7     $error = $exception;
8 } catch (\Exception $previous) {
9     // любое другое неожиданное исключение
10    $error = new GenericException('An error occurred', null, $previous);
11 }
12
13 // $error теперь можно использовать без опаски
```

При таком подходе есть одна проблема. Дело в том, что стандартный обработчик исключений Symfony (который также логгирует их, при включенном журналировании), не будет уведомлён о реально неожиданном исключении. Так как это как правило симптом проблем где-то на глубоких уровнях приложения, такое поведение весьма нежелательно. Следовательно, в большинстве ситуаций нужно отлавливать только несколько известных приложению исключений, и позволять перехватывать остальные встроенным в Symfony процедурам.

Настройте страницы ошибок

Когда “нормальное” исключение, указывающее на системную ошибку, перехватывается стандартным обработчиком исключений, вы должны быть уверены, что ничего лишнего не будет показано пользователю. Любая информация о системе, которая попадёт к пользователю, может быть использована для определения стратегии её взлома. Вам необходимо [создать ваши собственные страницы ошибок](#), которые будут выполнены в стиле всего остального приложения. Никогда не показывайте никакой генерированной информации о проблеме, никаких stack trace (низкоуровневый отчёт о ходе выполнения приложения до момента возникновения ошибки)! Проблемы должны автоматически логгироваться, вы также можете настроить отправку ошибок на email, например [настройив Monolog](#) или воспользовавшись любым другим инструментом журналирования, которые могут отправлять вам уведомления, когда что-то идёт не так. Последний вариант может быть даже более предпочтительным, так как приложение может прийти в такое состояние, когда оно само не сможет отправлять уведомления.

Не сообщайте ничего определённого о конфиденциальных данных

Проблемы аутентификации по большей части покрываются Security компонентом Symfony. Тем не менее имеется ряд узких мест, которые при стечении обстоятельств могут привести к раскрытию конфиденциальной информации о ваших пользователях или системе. Когда ваша система тем или иным образом даёт доступ к информации о том, какие пользователи в ней зарегистрированы, то она уязвима к т.н. “харвестингу” (т.е. к автоматизированному сбору данных пользователей, возможно конфиденциальных). Ниже приведены несколько примеров дизайна, которые допускают утечку информации о пользователях.

1. Пользователь может сбросить свой пароль на странице “Забыл пароль”, которая содержит форму с полем Email. Когда пользователь указывает свой email и отправляет форму, система ищет учётную запись пользователя и генерирует ссылку на восстановление пароля, которая отправляется на email, указанный пользователем. После отправки формы страница сообщает что-то похожее на “Мы выслали письмо с инструкциями по восстановлению пароля на указанный адрес”. Проблема в данном случае заключается в следующем: если кто-то пытается собирать email-адреса ваших пользователей, он теперь знает, зарегистрирован ли у вас в системе конкретный адрес или нет.
2. Та же ситуация, что и в примере выше, но теперь сообщение такое: “Если email принадлежит одному из наших пользователей, то мы выслали письмо с инструкциями по восстановлению пароля на него.”. Это сообщение выглядит более безопасно, но злоумышленник может делать определённые выводы на основании скорости выполнения запросов (если email отправляется - запрос будет выполняться дольше). Это называется “атакой по времени выполнения”
3. Когда пользователь вошёл в систему, он имеет возможность изменения email адреса, с использованием формы с одним полем - Email. После отправки формы система изменяет адрес или отображает сообщение “Такой адрес уже используется”. Опять - мы можем определять зарегистрирован ли в системе пользователь с конкретным адресом, даже если это конфиденциальная информация.

Наилучшим решением по предотвращению харвестинга будет неопределённость в сообщениях касательно пользовательских данных: никогда не говорите что есть точное совпадение, что вы отправили email и т.д. Вы также должны быть уверены, что это нельзя определить по длительности выполнения запросов.

В случае с проблемой номер 3 можно предпринять следующее:

1. Если вы разрешаете пользователю смену email адреса - не сохраняйте его сразу.
2. Отправьте на новый адрес email с ссылкой, по клику на которую пользователь сможет подтвердить запрос на смену email адреса.
3. После подтверждения смены адреса, обновите данные о новом email адресе пользователя.

@dbykadorov: будьте пааноиком

Эту часть надо было бы назвать “быть параноиком” =) Так как в вопросах обеспечения безопасности никогда не бывает “черезчур”. В решении проблемы 3 я тоже вижу определённые недостатки. Проблему харвестинга она допустим решает. Но что если пользователь, email которого подвергается атаке, получив непонятное письмо со ссылкой просто кликнет на неё, не особо задумываясь (или случайно). В этом случае он может лишиться своего аккаунта. И тут бы наверное можно было бы сказать “сам виноват”, но если вы дорожите своими пользователями, я бы посоветовал использовать какой-нибудь альтернативный способ смены адреса.

Например, вместо ссылки, которая сразу активирует действие, можно присыпать код, введя который в личном кабинете (лишь у того пользователя, который запросил смену адреса!) можно подтвердить смену email адреса. Более того, код можно ограничить по времени действия. И это, вероятно, ещё не самый безопасный вариант.

Так что, если вы действительно заботитесь о безопасности, не доверяйте никому, всё подвергайте сомнению, будьте на шаг впереди!

VI Используем аннотации

Введение

Первый релиз Symfony2 поставлялся вместе с [Doctrine Common library](#). Эта библиотека содержит некоторые инструменты, используемые в проектах Doctrine второго поколения, например в Doctrine ORM, Doctrine MongoDB ODM и т.д. Эта библиотека содержит функции и полезные классы, такие как менеджер событий, интерфейсы, касающиеся персистентности, некоторые полезные базовые классы и “считыватель” аннотаций (annotation reader).

Этот считыватель изначально использовался только в Doctrine ORM для парсинга аннотаций в классах сущностей. Doctrine ORM (Doctrine2) использует аннотации в качестве способа определения соответствия между PHP-классами и таблицами в реляционной СУБД:

```
1 use Doctrine\ORM\Mapping as ORM;
2
3 /**
4 * @ORM\Entity
5 */
6 class User
7 {
8     /**
9      * @ORM\Column(type="string")
10     */
11     private $name;
12 }
```

Этот “синтаксический сахар” вскоре был признан очень удобной альтернативой существующим способам определения маппинга, таким как описание конфигураций в Yaml или XML файлах. Создатели Symfony2 вдохновились использованием аннотаций и начали добавлять их в свои компоненты в качестве альтернативного способа определения конфигурации. Например, компонент Validator поддерживает аннотации для настройки правил валидации классов. Компонент Routing также поддерживает аннотации для связывания маршрутов с классами и их методами.

Аннотации: Предметно-ориентированные языки

Аннотации имеют весьма интересные характеристики. Они пишутся не на PHP: `@Route("/admin/", name="admin_index")` - этот текст не может быть распарсен интерпритатором PHP. И всё-же, аннотации имеют отношение к рантайму, так как они парсятся при помощи “считывателя” аннотаций во время выполнения PHP-приложения. Само их присутствие имеет вполне реальные эффекты на процесс выполнения приложения.

Более того, писать аннотации просто. Узнав все атрибуты, которые поддерживают аннотации, вы будете знать всё необходимое, чтобы использовать их в своём приложении. И самое интересное тут то, что вам не обязательно быть PHP-программистом, чтобы использовать аннотации, так как в конечном итоге, используя их, PHP код вы не пишете.

И, наконец, аннотации всегда предметно-ориентированы (domain-specific). Они изначально были введены, чтобы передать некоторую высокоуровневую концеп-

цию для определённой предметной области, например маршрутизатора, шаблонизатора, хранилища и т.д. Аннотации помогают абстрагироваться от деталей и мыслить более высокоуровневыми категориями.

Эти характеристики приводят нас к следующему выводу: аннотации следует рассматривать в качестве одной из форм предметно-ориентированных языков ([DSL](#)). DSL это небольшой язык, используемый для перевода высокоуровневых концепций в низкоуровневые детали. Этого можно добиться при помощи языков общего назначения вроде PHP, но также можно использовать отдельный синтаксис (в данном случае - синтаксис аннотаций).

Каждый набор предметно-ориентированных аннотаций образует язык конкретной предметной области, отражающий детали, из которых состоят высокоуровневые концепции. Например, аннотации, предоставляемые Doctrine ORM - это небольшой язык, который описывает в классе сущности (на высоком уровне) что должно произойти на низком уровне, чтобы этот уровень был готов для хранения объектов. Это означает, что когда вы используете аннотации Doctrine для конфигурирования маппинга сущности - вы не беспокоитесь по поводу реализации деталей вроде таких - как лучше определить колонку с булевым значением или какой синтаксис использовать для "обязательных" значений.

Когда я в первые начал работать с Symfony2, я увидел, что все спользуют аннотации, сам же я не решался их использовать. Мне казалось довольно опасным использовать комментарии для управления приложением. Затем, спустя несколько месяцев, я привык к работе с аннотациями и ни разу не столкнулся с проблемами при их использовании (@dbykadorov: вообще говоря я сталкивался - был один момент, когда популярный тогда ещё eaccelerator вырезал док-блоки и делал аннотации полностью нефункциональными. Но, строго говоря, это не проблема аннотаций как таковых). Вместо этого я изучил аннотации: что они из себя представляют и как они используются в Symfony и Doctrine. Вскоре я начал создавать свои собственные приложения и использовал аннотации в различных ситуациях.

В этой главе вы узнаете о том, как устроены аннотации и как вы можете создавать их самостоятельно. В последней главе этой части я покажу вам специфичные для Symfony случаи, когда вы можете использовать аннотации для того, чтобы влиять на процесс выполнения приложения.

Аннотация - это простой Объект-Значение (Value Object)

(@dbykarodov: если вы ещё не знаете что такое Value Object и чем он отличается от Entity - ознакомьтесь с неплохой статьёй на Хабре: [Entity vs Value Object: полный список отличий](#), если коротко - сущности идентифицируются по их identity, объекты-значения по из значению)

Аннотации пишутся в так называемых док-блоках, которые, в свою очередь, являются особым видом многострочного комментария, который начинается с `/**` (обычный комментарий начинается с `/*`). Обычные аннотации начинаются с символа “собачка” (коммерческое at) - `@` и трактуются как комментарии к коду, который следует сразу после, например:

```

1 /**
2  * @param int $id The ID of the user
3  * @return array Template variables
4  */
5 public function editAction($id)
6 {
7     ...
8 }
```

@dbykadorov: В данном примере первая аннотация описывает входной параметр функции, вторая же описывает возвращаемое значение. Подробнее о док-блоках читайте на сайте [PHP Documentor](#)

Когда происходит парсинг докблока, считыватель аннотаций Doctrine пропускает обычные аннотации в док-блок стиле, такие как `@param` и `@return`, опираясь на заранее определённый список. Когда же встречается аннотация не из списка, считыватель полагает, что эта аннотация является именем класса. Например, аннотация `@Route` метода `UserController::editAction()` в примере ниже - это фактически имя класса `SensioBundleFrameworkExtraBundleConfigurationRoute`, который должен быть импортирован при помощи выражения `use`:

```

1 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
2
3 class UserController
4 {
5     /**
6      * @Route("/users/{id}/edit", name="user.edit")
7      */
8     public function editAction($id)
9     {
10         ...
11     }
12 }
```

Считыватель аннотаций попытается создать экземпляр этого класса с помощью данных, расположенных между скобок (“`/users/{id}/edit`”, `name=”user.edit”`). Этотовый экземпляр объекта `Route` может быть использован в дальнейшем загрузчиком маршрутов для добавления дополнительного маршрута в `RouteCollection`.

В каждом Symfony приложении экземпляр считывателя аннотаций уже доступен “из коробки” в виде сервиса `annotation`:

```

1 // получаем контейнер
2 $container = ...
3
4 $reader = $container->get('annotation_reader');

```

Для парсинга док-блока UserController::editAction() нам нужно сначала создать **reflection object** для этого метода:

```

1 $method = new \ReflectionMethod('UserController', 'editAction');

```

Затем нужно запросить считыватель аннотаций распарсить аннотации для рефлексированного метода:

```

1 $annotations = $reader->getMethodAnnotations($method);

```

В результате парсинга аннотаций метода будет получен массив с одним объектом Route. Свойства объекта будут содержать атрибуты, которые были указаны в аннотации @Route:

```

1 print_r($annotations);
2
3 /*
4 Array
5 (
6     [0] => Sensio\Bundle\FrameworkExtraBundle\Configuration\Route Object
7         (
8             ...
9                 [path]
10                    => /users/{id}/edit
11                 [name]
12                    => user.edit
13             ...
14         )
15     )
16 */

```

Каждый класс аннотаций (типа класса Route) в свою очередь должен иметь аннотацию @Annotation для того, чтобы он был определён считывателем аннотаций в качестве класса аннотаций:

```

1 /**
2 * @Annotation
3 */
4 class MyAnnotation
5 {
6 }

```

Таким образом, если класс имеет аннотацию @Annotation, он может быть использован в качестве аннотации для класса, метода или свойства:

```

1  /**
2  * @MyAnnotation
3  */
4 class SomeClass
{
5     /**
6      * @MyAnnotation
7      */
8     private $someProperty;
9
10    /**
11     * @MyAnnotation
12     */
13    public function someFunction()
14    {
15    }
16
17    /**
18     * @MyAnnotation(@MyAnnotation)
19     */
20    public function otherFunction()
21    {
22    }
23}
24

```

Да, вы не ошиблись, можно даже использовать аннотацию `@MyAnnotation` внутри другой `@MyAnnotation`, как это было указано для док-блока `SomeClass::otherFunction()`. Когда мы попросим считыватель аннотаций распарсить аннотации для класса `SomeClass`, он вернёт массив с одним объектом - экземпляром класса `MyAnnotation`:

```

1 // получаем считыватель аннотаций
2 $reader = ...
3
4 $class = new \ReflectionClass('SomeClass');
5 $annotations = $reader->getClassAnnotations($class);
6
7 print_r($annotations);
8
9 /*
10  Array(
11      [0] => MyAnnotation object
12  )
13 */

```

Добавляем атрибуты к вашим аннотациям

Для того, чтобы ваши собственные аннотации были более полезными, нужно предусмотреть возможность сохранения в них данных. Эти данные могут быть предоставлены программистом, когда он добавляет вашу аннотацию к одному из своих классов, методов и т.д.

Парсер аннотаций поддерживает разные типы синтаксиса для заполнения атрибутов аннотации. Он поддерживает строки, числа, булевые значения, массивы объектов (которые также являются аннотациями). Например:

```

1 /**
2  * @MyAnnotation(
3  *     "some string",
4  *     "hashMap" = {
5  *         "key" = "value"
6  *     },
7  *     "booleanValue" = true,
8  *     "nestedAnnotation" = @MyAnnotation
9  * )
10 */

```

Любая логическая комбинация типов допустима. Например, имеется возможность расместить скаляр или объект в хэш-таблице.

Когда программист определяет данные для аннотации, как в примере выше, считыватель аннотаций должен передать эти данные в объект аннотации, который будет создан. Имеется две различные стратегии, которые считыватель аннотаций может применить.

Передача атрибутов через конструктор

Во-первых, парсер аннотаций будет искать конструктор класса MyAnnotation. Если таковой будет найден, он передаст все атрибуты в качестве первого аргумента конструктора, когда будет создавать экземпляр класса MyAnnotation:

```

1 /**
2  * @Annotation
3  */
4 class MyAnnotation
5 {
6     public function __construct(array $attributes)
7     {
8         // "какая-то строка"
9         $value = $attributes['value'];
10
11        // массив вида - array('key' => 'value', ...)
12        $hashMap = $attributes['hashMap'];
13
14        // булево значение - true
15        $booleanValue = $attributes['booleanValue'];
16
17        // экземпляр класса MyAnnotation
18        $nestedAnnotation = $attributes['nestedAnnotation'];
19    }
20}

```

С этого момента вы можете делать что угодно с этими значениями. Вероятно, вам нужно будет выполнять их валидацию и сохранить их в приватных свойствах.

Заполнение публичных свойств указанными атрибутами

В случае, если конструктор не обнаружен (или у конструктора нет аргументов), парсер будет пытаться присвоить значения предоставленных атрибутов публичным свойствам класса MyAnnotation:

```

1  /**
2  * @Annotation
3  */
4 class MyAnnotation
5 {
6     public $value;
7     public $hashMap;
8     public $booleanValue;
9     public $nestedAnnotation;
10}

```

Используя данную стратегию, вы не можете выполнить предварительную валидацию атрибутов перед тем, как они будут скопированы в публичные свойства. К счастью, имеется возможность добавить базовые правила валидации прямо в класс аннотации.

Валидация при помощи @Attributes

Мы можем использовать аннотацию @Attributes в классе MyAnnotation. Она принимает массив аннотаций @Attribute, которые могут быть использованы для описания каждого из поддерживаемых атрибутов: тип ожидаемого значения, а также признак того, является ли атрибут обязательным или нет.

```

1 /**
2 * @Annotation
3 * @Attributes({
4 *     @Attribute("value", type="string", required=true),
5 *     @Attribute("hashMap", type="array<string>", required=false),
6 *     @Attribute("booleanValue", type="boolean"),
7 *     @Attribute("nestedAnnotation", type="MyAnnotation")
8 * })
9 */
10 class MyAnnotation
11 {
12     public $value;
13     public $hashMap;
14     public $booleanValue;
15     public $nestedAnnotation;
16 }

```

По умолчанию атрибуты не являются обязательными. Когда необязательный атрибут не был указан, его значение будет установлено в null. Если атрибут имеет тип “массив” - указанное значение будет конвертировано в массив автоматически таким образом “some string value” станет массивом array(“some string value”).

Валидация при помощи аннотаций @var и @Required

Опции валидации, которые имеются при использовании аннотации @Attributes, очень полезны. Но, если вам не нравится факт, что правила для каждого свойства не указываются непосредственно над определением данного свойства, вместо аннотации @Attributes вы можете указать тип для каждого свойства:

```

1  /**
2  * @Annotation
3  */
4  class MyAnnotation
5  {
6      /**
7      * @var string
8      * @Required
9      */
10     public $value;
11
12    /**
13    * @var array<string>
14    */
15    public $hashMap;
16
17    /**
18    * @var boolean
19    */
20    public $booleanValue;
21
22    /**
23    * @var MyAnnotation
24    */
25    public $nestedAnnotation;
26 }

```

Для того, чтобы отметить атрибут аннотации как обязательный, добавьте аннотацию `@Required` в док-блок над соответствующим свойством.

Есть еще одна полезная опция, которая добавляет дополнительные возможности к процессу валидации: это аннотация `@Enum`. Вы можете использовать её для определения списка значений, которые допустимы для конкретного атрибута. Эта аннотация должна быть указана в док-блоке соответствующего свойства. Она работает как в комбинации с аннотацией `@Attribute`, так и с `@var`:

```

1 /**
2 * @Annotation
3 */
4 class MyAnnotation
5 {
6     /**
7     * @Enum({"yes", "no"})
8     */
9     $answer;
10 }

```

Теперь аннотация `@MyAnnotation(answer="yes")` будет валидной, а `MyAnnotation(answer="unsure")` будет вызывать ошибку.

Ограничения на использование аннотации

Различные типы аннотаций имеют различные условия применимости. Например, аннотацию `@Entity` из пакета Doctrine ORM имеет смысл использовать только для классов, не для методов. С другой

стороны, аннотацию `@Template` из пакета `SensioFrameworkExtraBundle` нужно использовать только для методов, но никак не для классов.

Более того, некоторые аннотации могут использоваться лишь для свойств, например аннотация `@Type` из пакета `JMS Serializer`. Также, некоторые аннотации могут быть использованы лишь внутри других аннотаций в качестве их атрибутов, например аннотация `@Attribute`, которую можно использовать лишь внутри аннотации `@Attributes`. Эти варианты использования также зовутся “целями” (`targets`) и мы можем настраивать их самостоятельно для наших классов аннотаций, добавляя к ним аннотацию `@Target`:

```
1 /**
2  * @Annotation
3  * @Target("CLASS")
4  */
5 class MyAnnotation
6 {
7 }
```

Допустимыми целями являются `CLASS`, `METHOD`, `PROPERTY`, `ANNOTATION` и `ALL` (`ALL` используется по умолчанию, если вы не указали цель). Если аннотация имеет несколько возможных целей, вы можете указать массив строк:

```
1 /**
2  * @Annotation
3  * @Target({"CLASS", "METHOD"})
4  */
5 class MyAnnotation
6 {
7 }
```

Когда стоит использовать аннотации

При помощи аннотаций вы можете делать всё что угодно, но в моей практике встречалось лишь несколько типов ситуаций, когда стоит использовать аннотации.

Загрузка конфигураций

Часто аннотации используются для загрузки статической конфигурации для классов. Вот несколько хороших примеров библиотек, которые используют аннотации таким образом:

- [Doctrine ORM](#) - маппинг сущностей на таблицы в базе данных.
- [Symfony Routing Component](#) - конфигурирование маршрутов в классах контроллеров.
- [Symfony Validator Component](#) - добавление правил валидации для классов.
- [JMS Serializer](#) - настройка параметров сериализации для любых классов.

Во всех этих библиотеках аннотации используются лишь как источник настроек. Все они предлагают также другие эквивалентные способы выполнить те же настройки, например загрузить некоторые XML, Yaml или PHP файлы конфигурации. Результат загрузки конфигурации при помощи любого из этих ресурсов будет использован для выполнения одних и тех же функций (создания схемы БД, валидации или сериализации объекта, генерации маршрута и т.д.). В данном случае аннотации, используемые для загрузки конфигурации не являются чем-то особенным и незаменимым - они лишь предоставляют одну из возможных альтернатив.

Важной характеристикой всех этих библиотек, которые используют аннотации для описания конфигурации, является то, что они парсят соответствующие аннотации единожды и комбинируют результаты с данными, полученными из других источников. Полученные в итоге данные кэшируются с целью сделать их повторное использование более быстрым. После первого запуска уже не имеет значения как были получены эти значения - из XML файлов, аннотаций или ещё откуда. Данные были скомбинированы и объединены в единый формат.

Аннотации и связность кода

Некоторые разработчики жалуются, что аннотации увеличивают связность классов. На практике существуют как аргументы за, так и против этой позиции: аннотации можно рассматривать как “всего-лишь комментарии” и с этой точки зрения они абсолютно не увеличивают связность классов, так как этот показатель учитывает как классы (а не их комментарии) связаны между собой. Тем не менее, после того как докблок распарсен, аннотации ставятся в соответствие классам. И в этом смысле ваш класс становится связанным с классом аннотации, которую он использует.

Этот особый тип связности становится весьма заметным, когда у вас есть класс, который использует несколько типов аннотаций, например Doctrine ORM и JMS Serializer:

```
1 use JMS\Serializer\Annotation as Serialize;
2 use Doctrine\ORM\Mapping as ORM;
```

```
3  
4 class Something  
5 {  
6     /**  
7      * @ORM\Column(type="string")  
8      * @Serialize\Type("string")  
9      */  
10    private $property;  
11}
```

Положим, проект, в котором вы хотите использовать этот класс, не имеет JMS Serializer'а среди зависимостей, но имеет Doctrine ORM и вы хотите использовать метаданные маппинга для того, чтобы иметь возможность хранить экземпляры класса *Something* в вашей реляционной базе данных. И вот, как только Doctrine начнёт считывать аннотации - она тут же выдаст ошибку:

```
1 [Semantical Error] The annotation "@Serialize\Type" in [...] does not  
2 exist, or could not be auto-loaded.
```

Эта ошибка является явным указанием на то, что класс *Something* на самом деле тесно связан со своими аннотациями. Ошибки типа этой являются аргументом против использования аннотаций в классах, если вы хотите их повторно использовать в дальнейшем, так как вне вашего проекта, шансы на то, что вы будете получать ошибки, используя классы с аннотациями, из за отсутствующих зависимостей - весьма велики.

Использование аннотаций в классах, которые используются только внутри вашего проекта, лично для меня является более чем приемлемым (это уже обсуждалось в другой главе этой книги - [Соглашения по конфигурированию](#)). Связность классов с аннотациям в данном случае не будет иметь никакого значения, так как классы приложения не будут использоваться в другом контексте.

Контроль процесса выполнения приложения

Вместо того, чтобы быть распарсеными лишь однажды, в виде части конфигурации, аннотации могут также быть использованы для влияния на процесс выполнения приложений. Наилучшим примером такого использования аннотаций является [SensioFrameworkExtraBundle](#), который включает в себя аннотации, контролирующие процесс генерации ядром HttpKernel ответа на конкретный запрос. Как было рассмотрено в первой главе этой книги, в ходе процесса генерации ответа имеется много мест, где слушатели событий имеют возможность изменить результат. Бандл SensioFrameworkExtraBundle содержит много различных слушателей, которые модифицируют объект Response, создают его, если он не был создан в контроллере, а также модифицируют полученный объект ответа, на основании аннотаций, которые были указаны в докблоке контроллера.

Например вы можете использовать аннотацию [@Template annotation](#) над методом действия в классе контроллера, для того, чтобы указать, что результат выполнения этого действия должен быть использован в качестве переменных шаблона:

```
1 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
2
3 class UserController
4 {
5     /**
6      * @Template("MatthiasAdminBundle:User:edit.html.twig")
7      */
8     public function editAction(User $user)
9     {
10         return array(
11             'user' => $user,
12         );
13     }
14 }
```

Для того чтобы эта аннотация работала, `SensioFrameworkExtraBundle` регистрирует слушатель, который перехватывает событие `kernel.controller` и получает аннотацию `@Template` для текущего контроллера. Затем он (слушатель) пытается определить, какой шаблон должен быть отрендерён и сохраняет имя файла шаблона в атрибут запроса `_template`. Когда контроллер был выполнен и вернул что-то отличное от объекта `Response`, другой слушатель перехватывает событие `kernel.view` и рендерит шаблон, указанный в атрибуте запроса `_template`, используя значение, которое вернул контроллер в качестве переменных шаблона (как правило, на выходе ожидается массив с переменными шаблона).

Используем аннотации в вашем Symfony-приложении

Будучи PHP разработчиком вы, вероятно, пишите соответствующую случаю библиотеку каждый раз, когда необходимо загрузить конфигурацию из различных источников, одним из которых могут быть аннотации. Если это так, то я рекомендую вам попробовать библиотеку [jms/metadata package](#). Она предоставит вам удобные инструменты, которые помогут объединить конфигурацию (т.е. “метаданные”) из различных источников и сохранить её в виде объекта в файловом кэше.

Я написал несколько статей в моём блоге об особенностях использования этой библиотеки начиная со [сбора метаданных с использованием аннотаций](#), далее о [добавлении альтернативного драйвера для сбора метаданных](#), и, наконец, о [правильном кэшировании метаданных](#).

В этой главе я решил не касаться подробно этого вопроса, так как использование аннотаций для сбора метаданных не является спецификой именно Symfony-приложений. Фактически любое приложение или библиотека может использовать аннотации таким образом.

Вместо этого, я покажу вам несколько способов, которыми вы можете контролировать процесс выполнения приложения используя аннотации (второй случай из предыдущей главы). Этот вопрос также частично освещался в моей статье о [предотвращении выполнения контроллера при помощи аннотаций](#). В следующих подразделах я опишу разнообразные опции, которые у вас имеются при использовании аннотаций совместно с событиями ядра.

Реагируем на атрибуты запроса (Request): аннотация @Referrer

Браузеры имеют одну полезную особенность - добавляют заголовок Referer к запросу (если это не прямой запрос, как например когда вы сами набираете URL вручную или выбираете одну из ваших закладок). Наименование заголовка на самом деле содержит опечатку (пропущена одна буква “r”: Referrer), но нам не стоит беспокоиться об этом, разве что самим стараться не делать опечаток.

Заголовок Referer содержит полный URL страницы, которая была посещена клиентом перед тем, как был запрошен текущий URL. При обработке запроса вы можете использовать этот заголовок для перенаправления клиента обратно (если это необходимо). Вы также можете сохранить предыдущий URL для того, чтобы позднее проанализировать откуда приходят ваши пользователи и как они перемещаются по сайту.

В данном конкретном случае я хочу использовать заголовок Referer для применения некоторых правил. Например, некоторые действия внутри контроллеров моего приложения должны быть выполнены лишь тогда, когда пользователь приходит с конкретных страниц, другие же должны быть доступны когда предыдущий URL имеет тот же домен, как и текущий URL. Хотя это и нельзя считать хорошей мерой по обеспечению безопасности, но иногда такая функция может быть полезной. Что более важно - данный пример является хорошей демонстрацией того, как вы можете воздействовать на процесс выполнения приложения, основываясь на атрибутах текущего объекта Request.

Не забывайте о проблемах безопасности

Когда вы создаёте логику вашего приложения, основываясь на атрибутах запроса, обращайте внимание на вопросы безопасности, которые могут возникнуть в процессе. Некоторые возможные проблемы с атрибутами запроса были рассмотрены в главе о [Безопасности]{#request-attributes}.

Основываясь на сценарии, который я описал выше, я должен иметь возможность написать такую конструкцию:

```

1 class SomeController
2 {
3     /**
4      * @Referrer(pattern="^/demo", sameDomain=true)
5      */
6     public function specialAction()
7     {
8         ...
9     }
10 }
```

Эта аннотация должна запускать некоторый механизм валидации, который получает заголовок Referer и проверяет, соответствует ли его путь указанному шаблону и имеет ли предыдущий URL тот же домен, что и текущий.

Соответствующий класс аннотации может выглядеть примерно так:

```

1 namespace Matthias\ReferrerBundle\Annotation;
2
3 /**
4  * @Annotation
5  * @Attributes({
6  *     @Attribute("pattern", type="string"),
7  *     @Attribute("sameDomain", type="boolean")
8  * })
9 */
10 class Referrer
11 {
12     public $pattern = '.*';
13     public $sameDomain = false;
14 }
```

Ни один из этих атрибутов не является обязательным. Вместо этого я добавил им значения по умолчанию.

Нам нужно, чтобы процесс валидации рефerrer'a активировался бы при помощи аннотации, которая относится к методу действия в классе контроллера. И когда же это лучше всего сделать? Как вы можете помнить из первой главы этой книги - после того как контроллер был полностью определён, ядро отправляет событие kernel.controller. Слушатели этого события после этого могут делать с этим контроллером всё что нужно. Кажется что этот момент подходит наилучшим образом. Итак, мы должны создать слушатель, который будет слушать событие kernel.controller. Когда слушатель уведомляется об этом событии, он инспектирует текущий контроллер и ищет в нём аннотацию @Referrer.

Слушатель ReferrerListener, описанный ниже, выполняет эту функцию. Теперь давайте убедимся, что метод этого слушателя onKernelController() вызывается, когда kernel.controller отправляется ядром. Я также полагаю, что вы знаете как [зарегистрировать этот слушатель](#).

```
1 namespace Matthias\ReferrerBundle\EventListener;
2
3 use Doctrine\Common\Annotations\Reader;
4 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
5 use Matthias\ReferrerBundle\Annotation\Referrer;
6 use Symfony\Component\HttpFoundation\Request;
7
8 class ReferrerListener
9 {
10     private $annotationReader;
11
12     public function __construct(Reader $annotationReader)
13     {
14         $this->annotationReader = $annotationReader;
15     }
16
17     public function onKernelController(FilterControllerEvent $event)
18     {
19         // текущий контроллер
20         $controller = $event->getController();
21
22         if (!is_array($controller)) {
23             // we only know how to handle a callable that looks like
24             // array($controllerObject, 'nameOfActionMethod')
25             return;
26         }
27
28         // считывателю аннотаций нужен reflection object такого вида:
29         $action = new \ReflectionMethod($controller[0], $controller[1]);
30
31         $referrerAnnotation = $this
32             ->annotationReader
33             ->getMethodAnnotation(
34                 $action,
35                 'Matthias\ReferrerBundle\Annotation\Referrer'
36             );
37
38         // $referrerAnnotation это либо экземпляр класса Referrer или null
39         if (!$referrerAnnotation instanceof Referrer) {
40             return;
41         }
42
43         $this->validateReferrer($event->getRequest(), $referrerAnnotation);
44     }
45
46     private function validateReferrer(
47         Request $request,
48         Referrer $configuration
49     ) {
50         $actualReferrer = $request->headers->get('referer');
51
52         $pattern = $configuration->pattern;
53         $sameDomain = $configuration->sameDomain;
54
55         // делайте всё что вам нужно
56         // киньте исключение, если что-то вам не понравится
57     }
58 }
```

Убедитесь, что сервис, который вы создали для этого слушателя будет получать сервис annotation_reader в качестве первого аргумента конструктора.

Аннотации должны быть закешированы

Когда вы зависите от считывателя аннотаций Doctrine, всегда указывайте для него подсказку типа DoctrineCommonAnnotationsReader. Symfony использует класс DoctrineCommonAnnotationsReaderCachedReader, который реализует этот интерфейс. Это по сути прокси к классу AnnotationReader. Кэширующий считыватель будет кэшировать прочитанные аннотации в виде сериализованных объектов. В Symfony приложении эти аннотации хранятся в директории app/cache/{env}/annotations. Например, кэшированный массив аннотаций для метода specialAction() (см. выше) будет выглядеть следующим образом:

```
1 <?php return unserialize('a:1:{i:0;O:43:"Matthias\\ReferrerBundle\\Annotation\\Referrer":2:{s:7:"pattern";s:6:"^/demo";s:10:"sameDomain";b:1;}}');
```

Предотвращаем выполнение контроллера: аннотация @RequiresCredits

В следующем примере я хочу либо предотвратить либо разрешить выполнение контроллера, основываясь на наличии неких “кредитов” у пользователя. Представьте, что ваше приложение имеет некоторую внутреннюю валюту под названием “кредит”. Пользователь может покупать кредиты, после чего он может посещать некоторые страницы с оплатой за каждый просмотр. Например, страница А стоит 100 кредитов. Когда у пользователя на счёте есть 150 кредитов, после посещения страницы А у него останется 50 кредитов. Также пользователь не должен иметь возможности посещать эту страницу пока у него не будет достаточно кредитов. Я представляю себе эту систему следующим образом:

```
1 class PayPerViewController
2 {
3     /**
4      * @RequiresCredits(100)
5      */
6     public function expensiveAction()
7     {
8         ...
9     }
10
11    /**
12     * @RequiresCredits(50)
13     */
14    public function cheapAction()
15    {
16        ...
17    }
18 }
```

Для начала давайте реализуем соответствующий класс аннотации:

```
1 namespace Matthias\CreditsBundle\Annotation;
2
3 /**
4 * @Annotation
5 * @Attributes({
6 *     @Attribute("credits", type="integer", required=true)
7 * })
8 */
9 class RequiresCredits
{
    public $credits;
}
```

Затем, примерно как мы это делали в примере выше, мы будем слушать событие kernel.controller и анализировать на предмет наличия аннотации @RequiresCredits, для того, чтобы узнать, сколько кредитов требуется для выполнения этого действия:

```
1 namespace Matthias\CreditsBundle\EventListener;
2
3 use Matthias\CreditsBundle\Annotation\RequiresCredits;
4 use Matthias\CreditsBundle\Exception\InsufficientCreditsException;
5 use Doctrine\Common\Annotations\Reader;
6 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
7
8 class CreditsListener
{
    private $annotationReader;
11
12     public function __construct(Reader $annotationReader)
13     {
14         $this->annotationReader = $annotationReader;
15     }
16
17     public function onKernelController(FilterControllerEvent $event)
18     {
19         $controller = $event->getController();
20
21         if (!is_array($controller)) {
22             return;
23         }
24
25         $action = new \ReflectionMethod($controller[0], $controller[1]);
26
27         $annotation = $this
28             ->annotationReader
29             ->getMethodAnnotation(
30                 $action,
31                 'Matthias\CreditsBundle\Annotation\RequiresCredits'
32             );
33
34         if (!($annotation instanceof RequiresCredits)) {
35             return;
36         }
37
38         $amountOfCreditsRequired = $annotation->credits;
39
40         // каким-то образом определяем, может ли пользователь позволить себе выполнить это действие
41         $userCanAffordThis = ...;
42
43         if (!$userCanAffordThis) {
44             // и что же будет теперь? вам решать
45             ...
46     }
47 }
```

```

46         }
47     }
48 }
```

Конечно же, вычисления, которые необходимо выполнить для определения того, может ли пользователь посетить страницу, должны выполняться при помощи специализированного сервиса, который можно внедрить в качестве аргумента конструктора. Тут же должен возникнуть вопрос: что нам нужно сделать, чтобы HttpKernel не выполнил контроллер, если пользователь не имеет достаточного количества кредитов. Для этого у нас есть несколько возможностей:

1. Мы можем заменить текущий контроллер другим, например контроллером, который рендерит страницу, где пользователь может докупить дополнительные кредиты.
2. Мы можем генерировать исключение, например InsufficientCreditsException.

Если вы выберите первый вариант, вы должны подменить текущий контроллер любым валидным “callable”. В идеале вы должны создать новый контроллер, но лучше будет, если вы внедрите его в качестве аргумента конструктора:

```

1 class CreditsListener
2 {
3     ...
4     private $creditsController;
5
6     public function __construct(
7         ...
8         CreditsController $creditsController
9     ) {
10    ...
11    $this->creditsController = $creditsController;
12 }
13
14 public function onKernelController(FilterControllerEvent $event)
15 {
16     ...
17
18     // заменяем текущий контроллер новым
19     $event->setController(
20         array(
21             $this->creditsController,
22             'buyAction'
23         )
24     );
25 }
26 }
```

Если вы выберите второй вариант, вы должны вызвать исключение, например, вот так:

```

1 namespace Matthias\CreditsBundle\Exception;
2
3 class InsufficientCreditsException extends \RuntimeException
4 {
5     public function __construct($creditsRequired)
6     {
7         parent::__construct(
8             sprintf(
9                 'User can not afford to pay %d credits',
10                $creditsRequired
11            )
12        );
13    }
14}
15
16 class CreditsListener
17 {
18     ...
19
20     public function onKernelController(FilterControllerEvent $event)
21     {
22         ...
23
24         throw new InsufficientCreditsException($annotation->credits);
25     }
26 }

```

Если просто вызвать исключение, будет показана стандартная страница ошибки. Вместо этого вы можете настроить ответ для конкретного исключения. Этого можно добиться, зарегистрировав слушатель события kernel.exception. Слушатель будет получать объект GetResponseForExceptionEvent. Затем можно проверить, является ли пойманное исключение экземпляром InsufficientCreditsException и если это так, сгенерировать более подходящий для этого случая ответ, например, страницу, где можно купить дополнительные кредиты. Вот как это может выглядеть:

```

1 use Symfony\Bundle\FrameworkBundle\Templating\EngineInterface;
2
3 class CreditsListener
4 {
5     ...
6     private $templating;
7
8     public function __construct(
9         ...
10        EngineInterface $templating // внедряем сервис "templating"
11    ) {
12        ...
13        $this->templating = $templating;
14    }
15
16     public function onKernelException(GetResponseForExceptionEvent $event)
17     {
18         $exception = $event->getException();
19         if (!$exception instanceof InsufficientCreditsException) {
20             return;
21         }
22
23         $response = $this
24             ->templating
25             ->renderResponse(

```

```

26         'MatthiasCreditsBundle::insufficientCredits.html.twig',
27         array(
28             'requiredCredits' => $exception->getRequiredCredits()
29         )
30     );
31
32     $event->setResponse($response);
33 }
34 }
```

В результате вместо стандартной страницы ошибки HttpKernel будет возвращать объект Response, который вернул CreditsListener.

Модифицируем ответ: аннотация @DownloadAs

Последним примером в этой главе будет модификация объекта Response на основании аннотации.

Обычно, когда вы хотите предложить клиенту результат выполнения действия контроллера в виде скачиваемого файла, вы должны выполнить что-то подобное:

```

1 use Symfony\Component\HttpFoundation\Response;
2 use Symfony\Component\HttpFoundation\ResponseHeaderBag;
3
4 class SomeController
5 {
6     public function downloadAction()
7     {
8         $response = new Response('body of the response');
9
10        $dispositionHeader = $response->headers->makeDisposition(
11            ResponseHeaderBag::ATTACHMENT,
12            'filename.txt'
13        );
14
15        $response
16            ->headers
17            ->set('Content-Disposition', $dispositionHeader);
18
19        return $response;
20    }
21 }
```

Этот вариант содержит некоторое количество довольно трудно читаемого кода, и, разумеется, не очень хорошо копировать этот код каждый раз, когда вам нужно предложить пользователю скачать результат действия в виде файла.

Избыточный код и копипаст в действиях контроллеров часто свидетельствует о том, что повторяющиеся функции нужно выносить в отдельные сервисы. Но, иногда, это также означает, что повторяющуюся функциональность можно разместить в некотором слушателе событий, который возьмёт на себя выполнение нужной задачи в соответствующий момент обработки запроса.

В данном случае логично предположить, что создание слушателя событий будет правильным выбором. Нужно будет слушать событие kernel.response. Это событие отправляется как

раз перед тем, как объект Response будет передан во фронт-контроллер, и это наиболее подходящий момент для того, чтобы превратить ответ в скачиваемый файл, добавив ему заголовок Content-Disposition:

```

1 namespace Matthias\DownloadBundle\EventListener;
2
3 use Symfony\Component\HttpFoundation\ResponseHeaderBag;
4 use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
5
6 class DownloadListener
7 {
8     public function onKernelResponse(FilterResponseEvent $event)
9     {
10         // нам нужно определить имя файла
11         $downloadAsFilename = ...;
12
13         $response = $event->getResponse();
14
15         $dispositionHeader = $response
16             ->headers
17             ->makeDisposition(
18                 ResponseHeaderBag::DISPOSITION_ATTACHMENT,
19                 $downloadAsFilename
20             );
21
22         $response
23             ->headers
24             ->set('Content-Disposition', $dispositionHeader);
25     }
26 }
```

Если мы сейчас зарегистрируем этот слушатель, каждый ответ будет превращаться в скачиваемый файл. Это не совсем то, чего мы хотели добиться. Ответ должен “превращаться” в файл лишь в случаях, когда соответствующее действие контроллера помечено как “downloadable”. Давайте сделаем это возможным при помощи аннотации @DownloadAs:

```

1 class SomeController
2 {
3     /**
4      * @DownloadAs("users.csv")
5      */
6     public function downloadAction()
7     {
8         ...
9     }
10 }
```

Реализация аннотации @DownloadAs может быть такой:

```

1 namespace Matthias\DownloadBundle\Annotation;
2
3 /**
4 * @Annotation
5 * @Attributes({
6 * @Attribute("filename", type="string", required=true)
7 * })
8 */
9 class DownloadAs
{
    public $filename;
}

```

Теперь нам нужно как-то проверить, имеет ли выполняемое действие аннотацию @DownloadAs. К сожалению, в методе onKernelResponse() мы ничего не знаем о контроллере, который был использован для создания ответа. Это означает, что мы должны внедриться в процесс несколько раньше, в тот момент, когда мы знаем какой контроллер выполняется. И снова, наилучшим способом будет слушать событие kernel.controller и использовать считыватель аннотаций для проверки наличия аннотации @DownloadAs.

Мы можем добавить еще один метод в класс DownloadListener: onKernelController(). Он будет вызван при возникновении события kernel.controller. Мы используем считыватель аннотаций для того, чтобы определить наличие аннотации @DownloadAs. Если она найдена, мы сохраним имя файла в виде атрибута текущего объекта Request:

```

1 use Matthias\DownloadBundle\Annotation\DownloadAs;
2 use Doctrine\Common\Annotations\Reader;
3 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
4
5 class DownloadListener
{
    private $annotationReader;
7
9     public function __construct(Reader $annotationReader)
10    {
11        $this->annotationReader = $annotationReader;
12    }
13
14     public function onKernelController(FilterControllerEvent $event)
15    {
16        // эта часть более-менее похожа на то что мы уже делали ранее
17        $controller = $event->getController();
18        if (!is_array($controller)) {
19            return;
20        }
21        $action = new \ReflectionMethod($controller[0], $controller[1]);
22
23        $annotation = $this
24            ->annotationReader
25            ->getMethodAnnotation(
26                $action,
27                'Matthias\DownloadBundle\Annotation\DownloadAs'
28            );
29
30        if (!$annotation instanceof DownloadAs) {
31            return;
32        }
33
34        // сохраняем имя файла в виде атрибута запроса

```

```
35     $event->getRequest()->attributes->set(
36         '_download_as_filename',
37         $annotation->filename
38     );
39 }
40 ...
41 }
42 }
```

Не забудьте зарегистрировать этот класс в качестве сервиса и объявить его слушателем событий kernel.controller и kernel.response

Теперь, когда контроллер для запроса определен, метод DownloadListener::onKernelController() будет искать аннотацию @DownloadAs, копировать имя файла для скачивания и сохранять его в атрибуте запроса _download_as_filename.

Нам осталось доделать лишь один нюанс - добавить проверку в метод onFilterResponse(), которая будет определять присутствует ли этот атрибут в запросе или нет. Если присутствует, тогда ответ будет доступен для скачивания (т.е. будет иметь формат размещения "attachment"), если искомый атрибут не будет найден, ответ не будет изменяться:

```
1 class DownloadListener
2 {
3     ...
4
5     public function onKernelResponse(FilterResponseEvent $event)
6     {
7         $downloadAsFilename = $event
8             ->getRequest()
9                 ->attributes
10                ->get('_download_as_filename');
11
12        if ($downloadAsFilename === null) {
13            // ответ не является скачиваемым файлом
14            return;
15        }
16
17        $response = $event->getResponse();
18
19        // устанавливаем тип размещения контента
20        ...
21    }
22 }
```

Проектирование для повторого использования

Вероятно вы уже заметили некоторую схожесть между разными слушателями событий, которые вы видели в предыдущих секциях:

- Большинство слушателей ожидают одно и то же событие - kernel.controller. Только некоторые из них ожидают другие события, например, kernel.response.
- Все слушатели используют считыватель аннотаций Doctrine для того, чтобы получить аннотации, относящиеся к конкретному действию в контроллере.
- Все слушатели работают лишь тогда, когда контроллер - это callable в виде массива из объекта и имени метода.

Эту обобщенную логику легко можно абстрагировать от контекста и это то, что мы просто обязаны сделать, так как это сделает наши слушатели более простыми. Позвольте представить вам моё решение этой проблемы (да, как правило есть несколько разных способов достигнуть того же результата - вы можете посмотреть мою реализацию и использовать её в качестве основы для ваших решений):

```
1 namespace Matthias\ControllerAnnotationsBundle\EventListener;
2
3 use Doctrine\Common\Annotations\Reader;
4 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
5
6 abstract class AbstractControllerAnnotationListener
7 {
8     private $annotationReader;
9
10    /**
11     * Возвращает класс аннотации, который должен присутствовать для текущего контроллера, для того
12     * чтобы можно было вызвать метод processAnnotation()
13     *
14     * @return string
15     */
16    abstract protected function getAnnotationClass();
17
18    /**
19     * Этот метод будет вызван лишь тогда, когда найдена аннотация для класса,
20     * который вернул метод getAnnotationClass()
21     */
22    abstract protected function processAnnotation(
23         $annotation,
24         FilterControllerEvent $event
25     );
26
27    public function __construct(Reader $annotationReader)
28    {
29        $this->annotationReader = $annotationReader;
30    }
31
32    public function onKernelController(FilterControllerEvent $event)
33    {
34        $controller = $event->getController();
35
36        if (!is_array($controller)) {
37            return;
```

```

38     }
39
40     $action = new \ReflectionMethod($controller[0], $controller[1]);
41
42     $annotationClass = $this->getAnnotationClass();
43
44     $annotation = $this
45         ->annotationReader
46         ->getMethodAnnotation(
47             $action,
48             $annotationClass
49         );
50
51     if (!$annotation instanceof $annotationClass) {
52         return;
53     }
54
55     $this->processAnnotation($annotation, $event);
56 }
57 }
```

Вы можете использовать этот абстрактный класс как родительский для каждого слушателя, который разрабатывается для работы с аннотациями контроллеров. Например, ReferrerListener можно сделать намного более простым и лаконичным, если унаследовать его от AbstractControllerAnnotation и реализовать абстрактные методы:

```

1 namespace Matthias\ReferrerBundle\EventListener;
2
3 use Matthias\ControllerAnnotationsBundle\EventListener\
4     AbstractControllerAnnotationListener;
5
6 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
7
8 class ReferrerListener extends AbstractControllerAnnotationListener
9 {
10     protected function getAnnotationClass()
11     {
12         return 'Matthias\ReferrerBundle\Annotation\Referrer';
13     }
14
15     protected function processAnnotation(
16         $annotation,
17         FilterControllerEvent $event
18     ) {
19         $actualReferrer = $event->getRequest()->headers->get('referer');
20
21         $pattern = $annotation->pattern;
22         $sameDomain = $annotation->sameDomain;
23     }
24 }
```

Вам не обязательно копипастить этот код в ваш проект.

Согласитесь, если каждый раз копировать базовый класс слушателя событий из проекта в проект - это будет выглядеть как-то странно. Я создал небольшой пакет для него [matthiasnoback/symfony-controller-annotation](#). На момент написания этой книги этот пакет содержал лишь один абстрактный класс базового слушателя событий. Если у вас есть потребности, которые этот класс не покрывает - пожалуйста откройте тикет или создайте pull request [на GitHub](#).

Заключение

Оглядываясь назад, на те примеры, что мы разобрали ранее в этой главе, я полагаю вы все согласитесь с тем, что использование аннотаций в контроллерах - это отличный способ повлиять на процесс выполнения приложения.

Аннотации контроллеров позволяют вам писать доменно-ориентированный код на языке высокого уровня (на языке аннотаций). Пример аннотации `@RequiresCredits` показывает это особенно хорошо. Эта аннотация скрывает большой объём кода от глаз разработчика, позволяет быстрее и проще разбираться с тем что происходит и реализовывать такую же логику в другом месте приложения, не копипастя сложный PHP-код. Такой подход также позволяет предотвратить дублирование кода и повторно использовать код.

Таким образом, в следующий раз, когда вам потребуется повторно использовать часть кода контроллера - спросите себя: является ли данный случай подходящим для использования аннотаций? У вас на займёт много времени создать прототип аннотации, если вы воспользуетесь пакетом

[matthiasnoback/symfony-controller-annotation package](#).

VII Быть Symfony разработчиком

Многие из разработчиков, которых я знаю, называют себя “PHP-разработчиками”. Некоторые из них возможно даже называют себя “Symfony-разработчиками”. Другие говорят просто - я разработчик, не указывая их любимый язык программирования и/или фреймворк.

Когда я начал работать с Symfony, я почти сразу проникся тёплыми чувствами к этому великолепному фреймворку. Его первая версия уже была лучше всего того, что я использовал до этого. Но его вторая версия была ровно тем что было нужно для моего мозга - я начал усиленно изучать фреймворк, погружаясь все глубже в его код, создавая документацию для его частей, которые еще не были документированы, создавая статьи о том, как достичь тех или иных целей с Symfony, а также выступая с докладами о Symfony и сопутствующих библиотеках.

После всего этого, я, наконец, могу называть себя Symfony-разработчиком. А теперь самое интересное: всё, что я узнал, работая с Symfony, также применимо к приложениям, написанным на любом PHP-фреймворке. Даже если я работаю с не-Symfony или с “унаследованным” PHP приложением (где совсем нет переиспользуемого кода), имеет смысл подумать об использовании кода из “экосистемы Symfony” или любой другой библиотеки, доступной через Composer для улучшения этого приложения - и это окупится.

В этой части я продемонстрирую, что быть хорошим Symfony разработчиком - это значит знать хорошо фреймворк, но при этом писать код так, чтобы он был применим к любому проекту на PHP и просто дополнять его в конце тонкой прослойкой между вашим кодом и Symfony, делая тем самым большую часть вашего кода пригодным для повторного использования, даже если он будет перенесен в не-Symfony проект.

Повторно используемый код должен иметь слабую связность

Разделяйте код компании от кода продукта

В качестве Symfony разработчика вы можете:

- Работать на компанию.
- Работать на заказчиков (внутренних и внешних) для которых вы создаете web-приложения.

Когда вы начинаете работу над новым приложением, вы должны разместить код вашего проекта в директорию /src. Но перед тем как начать, добавьте туда еще 2 директории: /src/NameOfYourCompany и /src/NameOfTheProduct. Конечно же имена этих директорий будут также отражены в пространствах имен в классах, которые вы создадите для вашего проекта.

Когда бы вы ни начали работать над новой функцией вашего приложения - думайте о том, какую часть вы в теории могли бы повторно использовать, а какая часть уникальна для этого конкретного приложения. Даже если переиспользуемая часть никогда и не будет использована, качество такого кода будет существенно лучше. После начинайте создавать классы в пространстве имен вашей компании. Когда же вы реально почувствуете, что нужно сделать что-то специфичное именно для этого проекта, переходите работать в пространство имен продукта или же используйте принцип расширения (наследование, конфигурирование, слушатели событий и т.д.).

Написание переиспользуемого кода для вашей компании не означает, что этот код должен быть открытым (open source). Это всего лишь означает, что вы должны писать его так, как если бы он был таковым. Вы не обязаны создавать побочные проекты на основе кода компании (по крайней мере сразу). Вы можете разрабатывать его внутри проекта, над которым сейчас работаете, и возможно в дальнейшем подготовите его к повторному использованию в других ваших проектах. Подробнее о практических аспектах такого подхода почитайте в разделе [Управление зависимостями и контроль версий](#)

Связность между кодом компании и приложения

Когда вы строго следуете принципу разделения кода продукта и кода компании, то следует придерживаться нескольких правил, которые помогут вам сделать это разделение реально полезным (если же вы не следите этим правилам - вам на самом деле нет смысла разделять пространства имен):

1. Код компании может знать о или зависеть от другого кода компании.
2. Код компании не должен знать или зависеть от кода продукта.
3. Код продукта может знать или зависеть от другого кода этого продукта.

Первые два правила должны применяться строго: если вы будете следовать им - только тогда ваш код будет переиспользуемым или готовым для open source. Третье правило напротив, дает свободу: так как вы по определению не будете переиспользовать код, специфичный для продукта, вы можете делать его сколь угодно связанным

Разделяйте код на “библиотечный” и “бандловый”

Когда вы пишите код не зависящий от Symfony или отдельных его компонентов, в этом случае вы должны разделять ваш код на “библиотечный” и “бандловый”. Библиотечный код – это часть кода, которая более-менее самостоятельна (хотя могут иметься и внешние зависимости). Библиотечный код может быть использован разработчиком, который работает с Zend framework, или с микрофреймворком Silex (и это лишь малая часть примеров, которые можно привести).

Бандловый код заменяет или расширяет некоторые классы из библиотечного кода, добавляя некоторые функции, специфичные для использования с Symfony. Он может определять конфигурацию бандла и его сервисы, для того чтобы экземпляры библиотечных классов были доступны в приложении. Фактически бандл просто делает библиотечный код доступным с минимумом усилий со стороны разработчиков, которые захотели бы использовать эту библиотеку в своих Symfony-приложениях.

Ниже перечислены элементы, которые должны быть внутри бандла:

- Контроллеры
- Расширения контейнера
- Определения сервисов
- Компиляторы (Compiler passes)
- Подписчики на события
- Классы, осведомленные о контейнере (Container-aware), которые расширяют базовые классы
- Классы типов для форм
- Конфигурация маршрутизатора
- Прочие метаданные
- ...

И этот список можно продолжать дальше, но его также можно и сократить. Если пораскинуть мозгами, то только первые несколько строк из списка выше реально специфичны для бандлов (т.е. используются только в контексте Symfony-приложений). Все же прочие элементы могут быть использованы в проектах, которые используют лишь некоторые из компонентов Symfony. Например, классы типов форм могут быть использованы в любом PHP проекте, который использует компонент Form.

Примеры библиотечного и бандлового кода

Имеется много хороших примеров такого разделения бандлов и библиотечного кода. Если вы посмотрите на код самого Symfony, вы можете увидеть применение подобного подхода: директория Component содержит все компоненты Symfony (это в наших терминах – “библиотечный код”), а директория Bundle содержит бандлы, которые связывают все классы вместе и предоставляют возможность конфигурирования их через аргументы конструкторов. Много отличных примеров такой стратегии можно найти в бандле FrameworkBundle (основополагающий бандл, когда он присутствует в проекте, мы можем говорить, что это Symfony-проект).

Уменьшайте связность с фреймворком

Вы можете пойти дальше и уменьшить связность с самим фреймворком или же с одним из его компонентов, что позволит сделать ваш код еще более пригодным для повторного использования.

Слушатели событий (listeners) вместо подписчиков (subscribers)

Например, предпочтительно использование слушателей событий вместо подписчиков на события. Подписчики - это особые слушатели, которые реализуют интерфейс EventSubscriberInterface:

```

1 use Symfony\Component\EventDispatcher\EventSubscriberInterface;
2
3 class SendConfirmationMailListener implements EventSubscriberInterface
4 {
5     public static function getSubscribedEvents()
6     {
7         return array(
8             AccountEvents::NEW_ACCOUNT_CREATED => 'onNewAccount',
9         );
10    }
11
12    public function onNewAccount(AccountEvent $event)
13    {
14        ...
15    }
16 }
```

Вы можете зарегистрировать подписчики типа рассмотренного выше, используя таг kernel.event_subscriber:

```

1 <service id="send_confirmation_mail_listener"
2     class="SendConfirmationMailListener">
3     <tag name="kernel.event_subscriber" />
4 </service>
```

При использовании такого подхода возникает несколько проблем:

- Подписчик становится совершенно бесполезным, если в проекте не используется компонент Symfony EventDispatcher, хотя в самом подписчике может и не быть никакой Symfony-специфики.
- Метод onNewAccount() получает объект класса AccountEvent, но нигде не определено, что такой объект может возникнуть только из события с именем AccountEvents::NEW_ACCOUNT_CREATED.

Таким образом, реализация слушателей в виде таких подписчиков не является приемлемой для переиспользуемого кода. Они связывают код с компонентом EventDispatcher. В данном случае лучше удалить использование интерфейса, удалить метод, реализующий интерфейс и зарегистрировать методы слушателя вручную, используя таг kernel.event_listener:

```

1 <service id="send_confirmation_mail_listener"
2   class="SendConfirmationMailListener">
3   <tag name="kernel.event_listener"
4     event="new_account_created" method="onNewAccount" />
5 </service>

```

Аргументы конструктора вместо получения параметров из контейнера

Когда вы знаете, что необходимые вам параметры имеются в контейнере, вы можете внедрить контейнер целиком лишь для того чтобы получить эти параметры (это уже звучит как расточительство =):

```

1 use Symfony\Component\DependencyInjection\ContainerInterface;
2
3 class SendConfirmationMailListener
4 {
5     private $container;
6
7     public function __construct(ContainerInterface $container)
8     {
9         $this->container = $container;
10    }
11
12     private function sendConfirmationMail()
13     {
14         $mailFrom = $this->container->getParameter('send_mail_from');
15
16         ...
17     }
18 }

```

И определение соответствующего сервиса будет таким:

```

1 <service id="send_confirmation_mail_listener"
2   class="SendConfirmationMailListener">
3   <argument type="service" id="service_container" />
4 </service>

```

Очевидно, что это очень плохо для переносимости сервиса: он может функционировать лишь в случае, если в приложении доступен контейнер Symfony. И ситуация даже хуже: даже если в проекте используется контейнер, никто не даст гарантий, что параметр `send_mail_from` будет в нем определен.

Таким образом, всегда внедряйте параметры контейнера в качестве аргументов конструктора, например так:

```

1 <service id="send_confirmation_mail_listener"
2   class="SendConfirmationMailListener">
3   <argument>%send_mail_from%</argument>
4 </service>

```

Аргументы конструктора вместо получения сервисов из контейнера

Также как и получение параметров из контейнера напрямую, использование контейнера для получения сервисов - это также плохая идея.

```

1 class SendConfirmationMailListener
2 {
3     private $container;
4
5     public function __construct(ContainerInterface $container)
6     {
7         $this->container = $container;
8     }
9
10    private function sendConfirmationMail()
11    {
12        $mailer = $this->container->get('mailer');
13
14        ...
15    }
16 }
```

Целью внедрения всего контейнера обычно является быстродействие

`@dbykadorov`: на мой взгляд спорное утверждение - так как внедрять немаленький контейнер в каждый сервис (как и я делал, когда только начинал писать на Symfony) - уже как минимум затратно по памяти.

Положим мы внедряем майлер напрямую в виде аргумента конструктора:

```

1 class SendConfirmationMailListener
2 {
3     private $mailer;
4
5     public function __construct(\Swift_Mailer $mailer)
6     {
7         $this->mailer = $mailer;
8     }
9 }
```

Теперь этот класс - слушатель события `new_account_created`. Когда бы слушатель не был создан, сервис майлера будет также инициализирован, даже если письмо будет отправлено лишь в некоторых случаях. Это займет некоторое время и некоторое количество системных ресурсов. И, хотя это действительно так, внедрение контейнера целиком для получения сервисов - это плохо, потому что:

1. Это связывает ваш код с особым типом контейнера (Symfony service container).
2. Предположение (потенциально неверное) что а) сервис майлера существует и б) он будет экземпляром `Swift_Mailer`
3. Внутри сервиса вы сможете получить из контейнера все что угодно, даже больше чем следовало бы.

Собственно разработчики вдохновляются поступать таким образом, используя стандартный класс контроллера Symfony, который реализует `ContainerAwareInterface`. Имеются ситуации, когда целесообразно внедрить контейнер целиком, но в подавляющем большинстве случаев целый контейнер вам не будет нужен, нужен будет лишь конкретный сервис из него. И ваш класс должен быть связан лишь с этим сервисом, а еще лучше с его интерфейсом, но не с DI контейнером.

@dbykadorov: я считаю что внедрение контейнера в сервисы - это один из самых частых и самых неприятных антипаттернов в Symfony приложениях. Происходит это от того, что при создании сервиса, не имеется четкого понимания, что он будет делать, и часто потом такие сервисы превращаются в GodObject'ы, которые и почту отправят и файл загрузят и для фото миниатюру создадут.

Также применение этого антипаттерна обусловлено засилием “толстых” контроллеров, которые любят создавать не очень ответственные разработчики. Такие контроллеры, как правило, наследуются от стандартного Symfony контроллера, который, как было сказано выше, осведомлен о контейнере (ContainerAware). Что означает доступность контейнера в любой момент, к чему разработчики привыкают и тянут эту привычку в создаваемые ими сервисы, что приводит к созданию “толстых” сервисов. О создании “тонких” контроллеров написано ранее в главе III.

Таким образом, если у вас появляется желание внедрить контейнер в ваш сервис - это значит, что у вас есть проблемы с дизайном приложения. Остановитесь, задумайтесь о той функции, которую должен выполнять этот сервис, и внедрите лишь то, что реально нужно.

О производительности

Но что же будет с производительностью? Выше я написал, что тот путь, который я рекомендую (внедрение конкретного сервиса, а не контейнера), может вызвать оверхед, особенно в случаях, когда внедренный сервис может и не быть использован. Ну что ж, у меня хорошие новости. Начиная с Symfony 2.3 вы можете не беспокоиться об этом. Вы можете добавить атрибут `lazy="true"` к определению сервиса, который может использоваться, или же не использоваться в вашем сервисе. В случае если атрибут `lazy` будет иметь значение “истина”, будет создан [proxy-класс](#). В результате вы можете внедрять зависимости в сервисы, но они будут полностью инициализироваться лишь тогда, когда вы вызовете один из их методов в первый раз. Я создал небольшое расширение - [LazyServicesBundle](#), которое позволяет вам назначать сервисы “ленивыми”, отмечая те аргументы конструктора сервиса.

Контроллеры, не зависящие от фреймворка

В Symfony приложениях контроллеры, как правило, это те сущности, которые наиболее тесно связаны с фреймворком. Если вы всегда стремитесь к тому, чтобы минимизировать зависимость вашего кода от чего бы то ни было (как это делаю я), то контроллеры - это хорошие кандидаты для рефакторинга. Но сначала мы должны рассмотреть функции, которые выполняют большинство контроллеров:

- получение каких-то данных из запроса (параметр маршрута или атрибут запроса).
- получение каких-то сервисов для выполнения запрошенных действий.
- рендеринг шаблона с использованием переменных, полученных из сервиса...
- или добавление флеш-сообщений в сессию, для уведомления пользователя о результатах действия...

- или генерация URL на другую страницу, для перенаправления пользователя.

Все это должно быть выполнено, так как фреймворк ожидает от вас этого. Как вы вероятно уже поняли, все это делает ваши контроллеры очень тесно связанными с фреймворком. И это должно быть так!

Symfony контроллер не сможет нормально функционировать в другом приложении, которое не использует такие Symfony компоненты как Router, HttpKernel и HttpFoundation. И такие приложения я называю “Symfony-приложениями”.

Поэтому в случае классов контроллеров вы можете пропустить все те шаги, что вы обычно делаете для уменьшения связности. Поэтому не стоит отказываться от наследования ваших контроллеров от стандартного Symfony-контроллера. Он может быть очень полезен, так как имеет много полезных методов для выполнения регулярно возникающих задач (например, `createForm()`, `generateUrl()`, `createNotFoundException()`, `redirectToRoute()` и т.д.).

Также вы не должны определять ваши контроллеры в виде сервисов. Они не будут повторно использованы в виде сервисов. Тем не менее, временами я начинаю думать, что это было бы неплохой идеей - иметь полный контроль над процессом создания контроллера, который в противном случае создается “магически” при помощи ControllerResolver, после чего - еще более магическим способом - в контроллер внедряется контейнер, если этот контроллер реализует ContainerAwareInterface (подробнее об этом читайте в разделе [всё, что может быть контроллером](#)).

Последнее замечание, перед тем, как вы радостно начнете привязывать ваши контроллеры к фреймворку: вы все еще должны стремиться делать ваши контроллеры маленькими, насколько это возможно. Когда вы входите в контроллер, вы должны быстро перевести фокус из контроллера в некоторый сервис, который выполнит всю необходимую работу. Смотрите также раздел [Тонкие контроллеры](#).

Тонкие команды

Создавать консольных команд в Symfony очень просто и, как правило, рекомендуется для выполнения некоторых специфичных задач для вашего бандла, с использованием командной строки. А то же время, команды должны рассматриваться также как и [контроллеры](#): они должны быть тонким слоем между входными параметрами, предоставленными пользователем и сервисом, который используется для выполнения задачи и получения результата, который должен увидеть пользователь.

Таким образом, в идеальном случае, метод `execute()` должен содержать лишь проверки входных параметров, получение необходимого сервиса из контейнера и вызов некоторого его метода.

```

1 namespace Matthias\BatchProcessBundle\Command;
2
3 use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
4 use Symfony\Component\Console\Input\InputInterface;
5 use Symfony\Component\Console\Output\OutputInterface;
6
7 class BatchProcessCommand extends ContainerAwareCommand
8 {
9     protected function configure()
10    {
11        $this->setName('matthias:batch-process');
12    }
13
14    protected function execute(
15        InputInterface $input,
16        OutputInterface $output
17    ) {
18        $processor = $this->getContainer()->get('batch_processor');
19
20        foreach (array('main', 'sub', 'special') as $collection) {
21            $processor->process($collection);
22        }
23    }
24 }

```

Если вам нужно сделать что-то большее внутри команды, перенесите как можно больше функций в специфичные сервисы.

Такой подход упростит повторное использование кода команды, даже при использовании другой библиотеки для выполнения команд, а также поможет тестировать ваш код, который “реально делает дело”, так как тестировать код внутри команды намного сложнее (с учетом меньшей изоляции), чем тестирование кода в обычном PHP классе.

Вынесение кода из класса команды сделает генерацию вывода команды в консоль более сложной задачей. Но имеется и весьма интересное решение для этой задачи: **AOP - аспектно ориентированное программирование**.

Окружение

Symfony имеет весьма удобный способ разделения и последовательного включения конфигурации бандла для различных окружений. Эти окружения, по принятому соглашению, называются `dev`, `test` и `prod`. Однако, эти значения в общем случае могут быть произвольными. Это даже не константы класса, но это первый аргумент, используемый для создания экземпляра ядра фронт-контроллера в `/web`:

```

1 // in /web/app_dev.php
2 $kernel = new AppKernel('dev', true);

```

Таким образом, вы никогда не должны зависеть от имени окружения, или же хардкодить его внутри любой части вашего кода. Вот такой код вы никогда не должны писать:

```
1 class SendConfirmationMailListener
2 {
3     private $environment;
4
5     public function __construct($environment)
6     {
7         $this->environment = $environment;
8     }
9
10    private function sendConfirmationMail()
11    {
12        if ($this->environment === 'dev') {
13            // always send the mail to matthiasnoback@gmail.com :)
14        }
15    }
16 }
```

Любая модификация поведения, зависимая от окружения приложения должна иметь место на уровне конфигурации вашего бандла, но даже в этом случае она должна быть неявной, так как другой разработчик может использовать совсем другие имена окружений. Фактически, когда вы копируете все из config_dev.yml в config_prod.yml, ваше продуктовое окружение будет выглядеть и работать как окружение для разработки.

Повторно используемый код должен быть легко переносимым

The main characteristic of mobile code is that you should be able to take the code and easily transfer it to another application. After installing the code in another project, you know that it is mobile when:

- You can easily fix bugs in the code, even though the code has been duplicated many times.
- It does not fail because of another library, component, bundle or PHP extension that is not present in the other project.
- You can easily configure it to work with another database and maybe even another type of database.
- It does not fail because certain database tables or collections don't exist.
- It does not fail because it is placed in an unusual directory inside the project.

Управление зависимостями и контроль версий

The first two characteristics of mobile code are related to infrastructure: when reused, your code should not merely be duplicated - it should still be under version control so that each bug-fix can be easily distributed over the different copies of the code.

You can do this using a mechanism like externals, or sub-modules (depending on the type of version control software that you use), but the better way is to use [Composer](#). Each piece of code that you want to share between projects should be in its own repository. In case you want to keep the code for yourself (or for your company): host the repository privately, on [GitHub](#), [Bitbucket](#), etc. When you want to make your code open source, GitHub would be the currently fashionable place.

In both cases you need a composer.json file. This file will basically make your code known as something coherent: a package. In composer.json you can define the requirements (PHP version, PHP extensions and other libraries), you can instruct the autoloader on where to find classes and functions, and you can add some information about yourself, the package, and the tags it should have.

As an example: this is a somewhat modified composer.json from the knplabs/gaufrette library:

```
1  {
2      "name": "knplabs/gaufrette",
3      "type": "library",
4      "description": "Library that provides a filesystem abstraction layer",
5      "keywords": ["file", "filesystem", "media", "abstraction"],
6      "minimum-stability": "dev",
7      "homepage": "http://knplabs.com",
8      "license": "MIT",
9      "authors": [
10         {
11             "name": "KnpLabs Team",
12             "homepage": "http://knplabs.com"
13         }
14     ],
15     "require": {
```

```

16     "php": ">=5.3.2"
17 },
18 "require-dev": {
19     "amazonwebservices/aws-sdk-for-php": "1.5.*",
20     "phpspec/phpspec2": "dev-master",
21     "rackspace/php-cloudfiles": "*",
22     "doctrine/dbal": ">=2.3",
23     "dropbox-php/dropbox-php": "*",
24     "herzult/php-ssh": "*",
25     "phpunit/phpunit": "3.7.*"
26 },
27 "suggest": {
28     "knplabs/knp-gaufrette-bundle": "*",
29     "dropbox-php/dropbox-php": "to use the Dropbox adapter",
30     "amazonwebservices/aws-sdk-for-php": "to use the Amazon S3 adapter",
31     "doctrine/dbal": "to use the Doctrine DBAL adapter",
32     "ext-zip": "to use the Zip adapter"
33 },
34 "autoload": {
35     "psr-0": { "Gaufrette": "src/" }
36 }
37 }
```

The list of requirements should be exhaustive for all the core functionality of your package. When the package contains classes that can be used optionally, which require extra dependencies, mention them under suggest. When you have written tests for these classes (like you should) make sure to list these extra dependencies together with your testing framework under require-dev so that all tests can be executed by anyone and no test will fail or get skipped because of a missing dependency.

Package repositories

Whenever a repository has a composer.json in its root directory, you can submit it as a package on [Packagist](#). This will effectively make your code open source. When you don't want this to happen, you can also privately host a package repository using [Satis](#). This works great for sharing code over many projects in the same company.

After creating the composer.json based on what your code really needs, and registering it as a package, either using Packagist or Satis, you can install the package in any of your projects by running:

```
1 composer.phar require [name-of-package] 0.1.*
```

Now, your code also complies with the second characteristic of mobile code: after installing it, it will not fail because of missing or incorrect dependencies.

Hard-coded storage layer

Auto-mapped entities

One of the biggest problems with bundles out there is that some of them define entity classes (using for instance annotations) and then put them in the Entity directory inside the bundle. This

automatically makes these specific entities available in your project. This may seem nice, but in most situations it is not: you are then not able to choose your own schema, add extra fields, or maybe leave out the entities entirely and redefine the model using, for instance, MongoDB.

When a bundle enforces a specific storage layer like described above, you will notice this as soon as you run a command like doctrine:schema:update: suddenly all kinds of tables are being created, beyond your control basically, since by enabling the bundle in which there are defined, these will automatically be registered. The same problem arises when you run doctrine:migrations:diff. Then a migration script will be generated for creating these tables that you never wanted in the first place.

Storage-agnostic models

The right way to define your bundle's model is to provide base classes. These base classes contain all the usual getters, setters and other methods for modifying or inspecting the state of an object. They have protected properties (instead of the usual private properties) with no mapping-related annotations at all. These storage-agnostic model classes should reside in the Model directory of your bundle, and they should not contain any code which is specific for their manager (like an EntityManager or a DocumentManager).

```
1 namespace Matthias\MessageBundle\Model;
2
3 class Message
4 {
5     protected $body;
6
7     public function.getBody()
8     {
9         return $this->body;
10    }
11 }
```

This allows developers who want to use your bundle to extend from the base class and define any metadata required for mapping the data to some kind of database in their own way:

```
1 namespace Matthias\ProjectBundle\Entity;
2
3 use Matthias\MessageBundle\Model\Message as BaseMessage;
4 use Doctrine\ORM\Mapping as ORM;
5
6 /**
7 * @ORM\Entity
8 */
9 class Message extends BaseMessage
10 {
11     /**
12      * @ORM\Column(type="text")
13      */
14     protected $body;
15
16     // Application specific data:
17
18     /**
19      * @ORM\Column(type="integer")
20     }
```

```
20      */
21  protected $upVotes;
22
23 /**
24 * @ORM\Column(type="integer")
25 */
26 protected $downVotes;
27 }
```

As you can see, this will also allow someone to add extra fields which are not defined in the reusable, generic model. Make sure your bundle has a configuration option for setting the userdefined model class:

```
1 matthias_message:
2     message_class: Matthias\ProjectBundle\Entity\Message
```

Model classes are library code (actually)

As suggested earlier you should move any code that is not specific for Symfony applications to their own libraries. The same applies to model classes. They should not be part of a bundle, but part of a library.

Object managers

Even when you have defined a storage-agnostic model, it does not mean that your entire bundle needs to be storage-agnostic. The only thing you need to take care of is that the users of your bundle should be able to implement their own storage handling and disable the standard way that you implemented. You can do this by applying the [strategy pattern for loading exclusive services](#), and by making use of aliases.

Also, you should not forget to make the names of entity managers, document managers, etc. configurable. For instance Doctrine ORM allows users to define different entity managers for different entity classes. So even though the default entity manager is the right entity manager in most cases, you should add a configuration key for it in your bundle's configuration.

25.3 Hard-coded filesystem references

Make sure your bundle does not contain any hard-coded filesystem references outside* the bundle itself. You may assume that anything starting from the root of your bundle is in your control, but the location of the vendor directory, or the web directory, or any other directory should be considered unknown. In fact, they are: when it comes to the directory structure, nothing is fixed when using Symfony. There are only “sensible defaults”.

Any reference you want to make to any file above ./ should be configurable in config.yml as part of your bundle's configuration. You may of course offer sensible defaults, so your bundle works out-of-the-box when installed in a standard Symfony project. The standard way to accomplish this is to define a scalar node in your configuration tree, with a default value. Then process the configuration in the load() method of your bundle's extension class and define a parameter for it. This parameter can later be injected, for instance as a constructor argument. See also [Define parameters in a container extension](#).

Using the filesystem

When your bundle requires a location other than the database to store data, like uploaded files, consider making the bundle filesystem-independent. Since not every application has write-access to the hard disk of the server it is hosted on, you may choose to use a filesystem abstraction layer, like [Gaufrette](#). It has adapters for many kinds of filesystems (local or remote), and your services won't notice anything when you switch between them. There is also a [GaufretteBundle](#) which further eases integration of the library with your Symfony bundles.

Повторно используемый код должен быть открыт для расширения

Code that is reusable has to be flexible. A user of your bundle should be able to configure services in a different way than they were configured in the original project you developed it for. He should also be able to replace any service of the bundle with a custom implementation, without the entire system falling apart. In other words, reusable code should adhere to the principle “open for extension, closed for modification”. The user should not be forced to change the original code, but only to replace or extend specific parts in order to better support his needs.

Configurable behavior

To make your bundle flexible, it should have a rich set of configurable options, with sensible defaults and information about what a certain option means. Only the services that are requested based on these configuration values provided by the developer in for instance config.yml should be fully loaded. See also the part of this book about [Patterns of dependency injection](#). It contains many suggestions on how to make a bundle configurable.

Everything should be replaceable

By using a compiler pass (see also [Patterns of dependency injection](#)) it is possible to replace any previously created service definition with your own service definition, or to change the class or any argument of existing service definitions. This means that when a developer wants to replace part of the functionality of your bundle with his own, there will in theory always be a way to accomplish this. There are still some things that you have to do, to help developers with replacing just what they need to replace, while keeping other parts of the bundle unchanged.

Many small classes, many single responsibilities

Most likely you share this experience with me: you have tried to use an open source bundle but you did not like part of the functionality offered by it. You tried to replace part of it, but this part got bigger and bigger until you almost developed your own bundle (and then you better should in my opinion!). The problem usually is that there is too little “separation of concerns” in such a bundle: some big classes try to do too many things. The lesson for you being:

- Write many small classes, with just a few methods and a single (conceptual) responsibility.
- Create many small services for these classes.

Use lots of interfaces

Define lots of interfaces for the classes that you use, both in your bundle and in the corresponding library code. Interfaces are literally the contracts that are signed between one object and another. A class that implements an interface says: don’t worry about how I do things exactly, this is how you can talk to me.

A traditional example, using only classes:

```

1  class Translator
2  {
3      public function trans($id, array $parameters = array())
4      {
5          ...
6      }
7
8      ... lots of other methods
9  }
10
11 class TranslationExtension
12 {
13     private $translator;
14
15     public function __construct(Translator $translator)
16     {
17         $this->translator = $translator;
18     }
19
20     public function trans($id, array $parameters = array())
21     {
22         return $this->translator->trans($id, $parameters);
23     }
24 }
```

It has this service definition:

```

1 <service id="translator" class="...">
2 </service>
3
4 <service id="twig.extension.trans" class="...">
5     <argument type="service" id="translator" />
6 </service>
```

When a developer wants to replace the translator, he needs to extend from the existing Translator class to satisfy the constructor argument type-hinted Translator:

```

1 class MyTranslator extends Translator
2 {
3     ...
4 }
5
6 <service id="my_translator" class="MyTranslator">
7 </service>
```

But now MyTranslator inherits everything that is already in the Translator class, even though it is going to do things very differently.

A much better solution would be to define an interface for translators:

```

1 interface TranslatorInterface
2 {
3     public function trans($id, array $parameters = array());
4 }
```

The only thing a replacement translator has to do, is implement this interface:

```

1 class MyTranslator implements TranslatorInterface
2 {
3     public function trans($id, array $parameters = array())
4     {
5         // do things very differently
6         ...
7     }
8 }
```

Finally, any existing type-hints should be changed from Translator to TranslatorInterface:

```

1 class TranslationExtension
2 {
3     public function __construct(TranslatorInterface $translator)
4     {
5         ...
6     }
7 }
```

And now nothing stands in the way of replacing the existing translator service with the my_translator service, either removing the existing definition, adding a new definition, by changing the class of the existing definition or by defining an alias from translator to my_translator.

Cohesive interfaces

When trying to find out what methods really belong to an interface, strive for a coherent set of methods, that would definitely belong to any (future) class that implements the interface.

Use the bundle configuration to replace services

As I mentioned before, you could in theory replace everything there is in the service container by something you created yourself. There are many ways to do this. But they are all not so clean and they won't be good for maintainability. The best way I've seen so far is to allow other developers to replace specific services, by providing service ids through the bundle configuration:

```

1 # in /app/config/config.yml
2 matthias_message:
3     # point to a specific service that should be used
4     message_domain_manager: matthias_project.message_domain_manager
```

Add extension points

You can make behavior configurable, or allow others to replace parts of your bundle. But you can also add ways to extend the behavior of your bundle.

Service tags

One way to allow other bundles in an application to extend the behavior of your bundle, is to make use of service tags. You could think of the way you can register form types that can be used in the entire application:

```
1 <service id="address_type" class="...">>
2   <tag name="form.type" alias="address" />
3 </service>
```

Using a compiler pass you can find all services with a given tag and do whatever you want with them. See also Service tags for implementation details and more examples.

Events

Events are great for allowing other parts of an application to hook into the execution of your bundle's code. You could use events to simply notify the system that something has happened:

```
1 $eventDispatcher->dispatch(
2   'matthias_account.account_created',
3   new AccountEvent($account)
4 );
```

Or you could use an event object to let other parts of the application modify (filter) some value, like the Symfony kernel itself does, when it dispatches its kernel events (see also Early response).

Повторно используемый код должен быть легко используемым

Add documentation

The Symfony Cookbook article about [best practices for creating bundles](#) recommends writing documentation for bundles. The proposed format is [Re-Structured Text](#) and the main entry point should be Resources/doc/index.rst. However, many PHP projects I know of also have a README.md file (which is in the [Markdown](#) format), describing in a few words what the project is about and how to install it.

Documentation should cover at least:

- The main concepts in the design of your code.
- Use cases, illustrated with code and configuration examples.
- Service tags that can be used to extend functionality.
- Events that are being dispatched.
- Any [flow-controlling exceptions](#) that can be thrown.

Documentation for internal projects

When your bundle is going to be used only by your fellow developers, you should not expect them to read your documentation. They will just try to start using it, and you are in the same room with them, so they will just ask you if anything doesn't work as expected. However, you still need to provide some documentation for internal projects, to prevent future problems in case you would accidentally or purposefully leave the team.

Throw helpful exceptions

Chances are that someone trying to use your bundle will encounter some problems. Maybe you forgot to make some requirements explicit, maybe you made some other assumptions about how and in what situation the bundle's services would be used. In order to help your "user" to overcome any problem in this area, you should throw helpful exceptions, which point him in the direction of the solution.

Use specific exception classes

Choose your exception classes well, and preferably add some exception classes yourself:

```
1 namespace Matthias\ImportBundle\Exception;  
2  
3 class WrongFormatException extends \RuntimeException  
4 {  
5 }
```

This allows someone to catch exceptions from which the application may still be able to recover:

```
1 try {
2     $this->loadFile($file);
3 } catch (WrongFormatException $exception) {
4     // ...
5 } catch (FileNotFoundException $exception) {
6     // ...
7 }
```

Set detailed and friendly messages

There is a common misconception that you should not put all the relevant details in the message of an exception. This would be bad for “security”. Well, since you should prepare your production environment to display no error messages at all, there’s no problem in giving full disclosure in an exception message. On the contrary: you should provide all relevant details. So, use sprintf() to construct a nice and friendly message:

```
1 throw new \InvalidArgumentException(sprintf(
2     'Tag "%s" for service "%s" should have an "alias" attribute',
3     $tagName,
4     $serviceId
5 ));
```

In contrast with things like:

```
1 throw new \OutOfBoundsException('You are a hacker');
```

Which would communicate very clearly your opinion about the user, but not help a developer in finding out what he can do to prevent the error.

Повторно используемый код должен быть надёжным

Add enough tests

When your code should be stable and maintainable at any rate, it needs enough tests. What is enough? Maybe you don't need a 100 percent coverage, but at least the use cases that you say your code supports should be verified to work correctly using unit tests.

Writing the tests will not be such a difficult task, since when you followed my advice in the previous section, your bundle consists of small classes, and many of them have interfaces, so they should already be very test-friendly.

After you have tested the true units of your bundle, all the little classes in there, it is time to test them all together. In other words: write integration tests. In my bundles, this means most of the times manually instantiating some classes, prepare some constructor arguments and create some more objects, and finally run a single method on one of these objects (while keeping track of which is which). This step is important, as you want to find and fix any problems without going back to the browser after many hours of fast and furious coding in your favorite editor.

Use a simple service container for integration tests

When you start writing integration tests, and you have indeed created many small classes that work together, the setup code for your tests may become quite large, with lots of new operators. To be able to reuse the setup code and to enhance its maintainability, you should consider using a simple service container like Pimple for managing the object graph that is required for your integration test. See also [PHPUnit & Pimple: Integration Tests with a Simple DI Container](#).

Test your bundle extension and configuration

Something that is missing in the test suites of many bundles: unit tests for framework-specific classes like the Bundle class and, when applicable its Extension and Configuration class. You can usually skip the Bundle class, but the Extension and Configuration classes have logic that should be tested. After all, you want to make sure that all things inside your bundle are tied together correctly. To test the bundle's extension class, you only need a ContainerBuilder instance. Then run the load() method of the extension and provide it with an array of config arrays:

```
1 namespace Matthias\AccountBundle\Tests\DependencyInjection;
2
3 use Symfony\Component\DependencyInjection\ContainerBuilder;
4 use Matthias\AccountBundle\DependencyInjection\MatthiasAccountExtension;
5
6 class MatthiasAccountExtensionTest extends \PHPUnit_Framework_TestCase
7 {
8     public function testLoadsDoctrineORMServicesWhenEnabled()
9     {
10         $container = new ContainerBuilder();
11         $extension = new MatthiasAccountExtension();
12
13         $configs = array(
```

```

14     array(
15         'storage_engine' => 'doctrine_orm',
16     )
17 );
18 $extension->load($configs, $container);
19
20 $this->assertTrue(
21     $container->has('matthias_account.doctrine_orm.storage')
22 );
23 }
24 }
```

Configuration arrays

Please note that you have to provide an array of arrays as configuration values. This is the way application configuration works: the values can originate for instance from both config_dev.yml and config.yml. The configuration Processor merges these separate arrays into one.

Use the `SymfonyDependencyInjectionTest` library

Testing a bundle extension class (and compiler passes for that matter) will be much easier when you use the [SymfonyDependencyInjectionTest](#) library. It contains base classes for your own PHPUnit test cases and custom PHPUnit assertions.

Testing the bundle configuration is mainly an exercise in testing the Config Component itself. But since the code in the Configuration class is itself configuration (think one second longer about that...), you need to make sure that this configuration is sound. This is how you can do it:

```

1 namespace Matthias\AccountBundle\Tests\DependencyInjection;
2
3 use Matthias\AccountBundle\DependencyInjection\Configuration;
4 use Symfony\Component\Config\Definition\Processor;
5
6 class ConfigurationTest extends \PHPUnit_Framework_TestCase
7 {
8     public function testHasSensibleDefaults()
9     {
10         $configurations = array();
11         $processedConfig = $this->processConfiguration($configurations);
12
13         $expectedConfiguration = array(
14             'user_types' => array('user', 'administrator'),
15             'send_confirmation_mail' => true
16         );
17
18         $this->assertSame($expectedConfiguration, $processedConfig);
19     }
20
21     private function processConfiguration(array $configValues)
22     {
23         $configuration = new Configuration();
24
25         $processor = new Processor();
26
27         return $processor->processConfiguration(
28             $configuration,
```

```
29         $configValues
30     );
31 }
32 }
```

Use the `SymfonyConfigTest` library

Testing your Configuration class will be easier when you use the [SymfonyConfigTest](#) library. It contains a base class for your own PHPUnit test case and custom PHPUnit assertions.

Заключение

Though not from request to response, this has been quite a journey! I wrote the first pages of this book in April of 2013. In June I said it would take just 6 more weeks to finish it. Well, it took three months, but now it is exactly as I had imagined it to be: a book for developers familiar with Symfony, wanting to take an extra step. Now that you've finished the book, I hope that you found it interesting. And maybe you can use some of the practices described here in your work as a developer.

¹ Since this is the last page of the book, it's time for some meta-comments!

I think that the PHP community has taken some huge steps in the last couple of years. There are some great tools and platforms now, like GitHub and Composer, Vagrant and Puppet. PHP developers are becoming aware of the commonality of their individual ideas (for instance as expressed by the PSR-* standards). They have also slowly started to take care of handling version numbers well (e.g. semantic versioning). And they are generally very much concerned about bugs in their open-sourced software (which other people use in production environments). Above all, many PHP developers contribute to open-source projects for free, which generates a good vibe and a great tradition of giving and sharing.

But PHP developers are not used to all of the activities related to working with or even delivering open-source software: defining a manageable list of package dependencies, distributing packages, releasing software to the world, announcing changes to an API, handling bugs reported by strangers, working with continuous integration, etc.

More importantly: developing software for reusability has proven to be quite a difficult task in itself. Luckily, there are many principles you can adhere to. Some of them are described in the last part of this book. And some of them are well known to people writing in other languages than PHP. See for instance the list of [package design principles](#) as described by Robert Martin. Ideas like these are not known to everyone in the PHP community, while they should be. If I were to ever write a book again, it would be about that: PHP package design.