

Romer M.

Web Development with Zend Framework 2

Dzen Framework

Любительский перевод от Dolphin.

Время пришло.

Перевод создан только в целях ознакомления и рекламы. После ознакомления с содержанием данной книги Вам следует незамедлительно ее удалить. Сохраняя данный текст, Вы несете ответственность в соответствии с законодательством. Любое коммерческое и иное использование кроме предварительного ознакомления запрещено. Публикация данных материалов не преследует за собой никакой коммерческой выгоды. Этот перевод является всего лишь рекламой оригинального издания.

Оглавление

I. О книге	7
Предварительная редакция	7
Интернет-сообщество книги	7
II. Введение	8
Для кого эта книга?	9
Структура этой книги	9
Повторения	10
Как лучше всего работать с этой книгой	10
Нашли ошибку?	11
Условные обозначения, используемые в этой книге	11
Примеры кода	11
Командная строка	11
III. Zend Framework 2 – Обзор	12
Как в настоящее время разрабатывается фреймворк?	12
Стандарт кодирования PSR-2	13
Известные проблемы	13
Модульная система	14
Система событий	15
Реализация MVC	16
Дополнительные компоненты	17
Шаблоны проектирования: Interface, Factory, Manager, и другие	20
Интерфейс (Interface)	20
Слушатель (Listener)	20
Агрегат слушателей (ListenerAggregate)	21
Фабрика (Factory)	21
Сервис (Service)	21
Менеджер (Manager)	21
Стратегия (Strategy)	21
Модель-Представление-Контроллер (Model View Controller (MVC))	22
Действия (Actions)	22
Помощник представления (View Helper)	22

Плагины контроллера (Controller Plugins)	22
IV. Hello, Zend Framework 2!	23
Установка	23
ZendSkeletonApplication	24
Composer	24
Первые признаки жизни	26
Структура каталогов приложений на Zend Framework 2	28
Файл index.php	31
V. Подготовка собственного модуля	36
Подготовка модуля "Hello World"	36
Автозагрузка	42
Стандартный автозагрузчик (Standard autoloader)	43
Автозагрузчик карты классов (ClassMapAutoloader)	44
VI. За запросом и обратно	47
Сервис-менеджер (ServiceManager)	47
Сервисы и регистрация сервиса	47
Стандартные сервисы, часть 1	49
Вызываемые сервисы (Invokables)	49
Фабрики (Factories)	49
Конфигурация	50
Создание собственного сервиса	51
Предоставление доступа к сервису, как invocable	52
Предоставление доступа к контроллеру с помощью класса фабрики	53
Предоставление доступа к контроллеру с помощью обратного вызова фабрики	56
Предоставление доступа к сервису с помощью Zend\Di	56
Менеджер модулей (ModuleManager)	57
Формирование ModuleManager	57
Слушатели (Listeners), ориентированные на модули	57
Стандартные сервисы, часть 2	58
Вызываемые (Invokables)	58
Фабрики (Factories)	58
Загрузка модулей	61
Объекты событий модуля	61

Активация модуля	62
Методы класса модуля.....	62
Приложение (Application).....	65
Создание приложения и начальная загрузка	65
Выполнение приложения	66
Менеджер представления	68
Слушатели (Listeners), ориентированные на представление (View).....	68
Сервисы, ориентированные на представление (View)	69
Резюме	69
VII. Менеджер событий (EventManager)	72
Регистрация слушателей (listeners)	72
Регистрация нескольких слушателей одновременно	73
Удаление зарегистрированных слушателей	74
Инициирование события.....	74
SharedEventManager	76
Использование событий в своих собственных классах	78
Объект события.....	83
VIII. Модули	87
Модуль "Application"	87
Поведение, зависимое от модуля	89
Установка сторонних модулей	91
Источники сторонних модулей	91
Установка сторонних модулей	91
Настройка стороннего модуля.....	93
Изменение URL.....	94
Изменение представления	95
IX. Контроллер	97
Концепция и режим работы	97
Плагины контроллеров	98
Переадресация (Redirect).....	98
PostRedirectGet	100
Перенаправление (Forward)	100
Адрес (URL).....	101

Параметры (Params).....	101
Макет (Layout).....	102
Flash messenger	102
Создание собственного плагина контроллеров	103
X. Представление (View)	105
Концепция и режим работы	105
Макеты (Layouts).....	107
Макет и шаблоны – от переводчика	107
Помощники представления.....	109
Создание собственного помощника представления	110
XI. Модель (Model)	112
Persistence ignorance – от переводчика	113
Сущности, репозитории и объекты-значения	117
Бизнес-сервисы и фабрики	119
Бизнес-события	119
XII. Маршрутизация.....	120
Введение	120
Определение маршрутов.....	120
Проверка соответствия.....	121
Проверка пути	122
Проверка имени хоста, протокола и т.д.	124
Комбинация правил.....	125
Генерация URL.....	127
Стандартная маршрутизация	128
Креативная маршрутизация: A/B-тестирование	130
XIII. Внедрение зависимостей	132
Введение	132
Zend\Di для графов объектов	136
Zend\Di для управления конфигурацией.....	143
XIV. Персистентность с помощью Zend\Db	146
Соединение с базой данных	148
Создание и выполнение SQL-запросов	149
Работа с таблицами и записями	152

Организация запросов к базе данных	156
Создание адаптера базы данных с помощью ServiceManager	156
Капсулирование подобных запросов	158
Альтернатива Zend\DB Doctrine 2 ORM	164
XV. Валидаторы	165
Стандартные валидаторы	165
Создание собственных валидаторов	167
XVI. Формы	169
Подготовка формы	171
Отображение форм	174
Обработка элементов форм	175
Валидация элементов формы	176
Стандартные элементы формы	182
Набор полей формы (Fieldsets)	184
Связывание сущностей с формами	189
XVII. Дневники разработчика	200
Введение	200
Выработка концепции	200
Спринт 1 – Репозиторий и начальная база кода	200
Настройка репозитория git и первый коммит.	200
Спринт 2 – Пользовательский модуль, добавляющий функциональность продукта	202
Создание пользовательского модуля	202
Работа со статическими ресурсами	207
Форма для добавления продукции	207
Отображение формы	211
Юнит-тесты для веб-формы	213
Настройка сущности product	215
Персистентность сущности product	217
Обработка форм	219
Внедрение зависимостей	220
Юнит-тесты контроллера	221
Избегайте неоднозначных идентификаторов	224
Спринт 3 – Добавление акции, показ доступных акций	229

Стандарт кодирования	229
Сценарий инициализации базы данных	230
Сущность "акции"	231
Добавление новой акции	232
Показать текущие акции.....	239
Пользовательский тип контроллера для работы с формами	241
Юнит-тесты для AbstractFormController	246
Использование замыканий, как простых фабрик	248
Спринт 4 – Заказы	250
Спринт 5 – Сделать ZfDeals доступным в качестве модуля ZF2	254
Новый репозиторий для модуля ZfDeals	254
Упрощение конфигурации модуля	257
Лучший подход для отображения форм	261
Что дальше?	263
Zend\Di для внедрения зависимостей	263
Doctrine 2 для персистентности данных	263
Использование системы событий ZF2	264
Улучшение обработки статических ресурсов	264
Исходный код и послесловие – от переводчика	264

I. О книге

Предварительная редакция

Если вы читаете этот раздел, вы держите предварительную редакцию этой книги в ваших руках. "Предварительная редакция" означает, что вы уже можете приступить к чтению книги, в то время, как последующие главы все еще пишутся. Они будут доступны, как только будут окончены. Благодаря концепции лин-публикации (<http://leanpub.com>), вы намного более актуальны, чем когда-либо прежде. Содержимое этой книги относится к версии Framework 2.0, и, вероятнее всего, будет действительно для более поздних версий. Вполне возможно, что вы могли бы найти целый ряд орфографических ошибок или же ошибку - две в примерах кода в ранних изданиях. В настоящее время никто не поддержал меня с корректурой и редактированием, чтобы уменьшить количество ошибок в этой книге к минимуму. Естественно, я пытаюсь работать как можно тщательнее, но я не могу гарантировать абсолютное отсутствие ошибок на данный момент. Пожалуйста, потерпите. Если вы хотите помочь мне улучшить эту книгу, вы можете вступить в интернет-сообщество книги и дать мне знать о ваших идеях или о любых вопросах, которые возникнут при чтении книги. Я был бы вам очень благодарен за это.

Интернет-сообщество книги

Нашли ошибку в книге? Хотите поговорить о содержимом книги или Zend Framework 2 в целом? Пожалуйста, не стесняйтесь присоединиться к интернет-сообществу книги, Google-Group "Web Development with Zend Framework 2 Book" (<https://groups.google.com/forum/#!forum/web-development-with-zend-framework-2-book>). Вам понадобится аккаунт Google, который может быть создан бесплатно.

II. Введение

Zend Framework 2 является фреймворком с открытым исходным кодом для профессиональной разработки веб-приложений на PHP версии 5.3. и выше. Он работает на сервере и в основном используется для создания динамического веб-содержимого или проведения операций, таких, как приобретение товара в интернет-магазине. В этом контексте Zend Framework 2 поддерживает создание любого типа (Web) приложений, поскольку он обеспечивает универсальные решения, которые могут быть успешно использованы в электронной коммерции, управлении контентом, создании комьюнити, или SaaS приложений. Zend Framework, по сути, разрабатывается компанией Zend, которая также дала свое название фреймворку и предоставляет для него финансовую основу, не в последнюю очередь и потому, что сам Zend также участвует в разработке языка программирования PHP. Однако, кроме Zend, ряд престижных компаний также поддерживает Zend Framework и заинтересован в его долгосрочном успехе. Среди них также Microsoft и IBM. Все вышеперечисленное обеспечивает надежность выбора Zend Framework и делает этот выбор хорошим решением с экономической точки зрения в качестве основы для создания собственных приложений.

Zend Framework появился на сцене в стабильной версии 1.0 впервые 30.06.2007 и с тех пор производит неизгладимое впечатление на лице веб-разработки PHP. Можно действительно сказать, что программирование на PHP с Zend Framework впервые стало действительно приемлемо для создания приложений, критически важных для компаний. Если ранее в корпоративном контексте доверяли сложным J2E (Java) приложениям, то теперь с удовольствием выбирают PHP и Zend Framework с чистой совестью потому, что эта комбинация действительно обеспечивает сбалансированное соотношение легкости и профессионализма, чего не может достичь практически ни одна другая платформа.

Если принять во внимание первые предварительные релизы Zend Framework 1, то на сегодняшний день фреймворку уже более 6 лет. Даже если многое уже достигнуто в бесчисленных релизах от первого анонса, ряд остро необходимых улучшений и расширений не может быть реализован на старой базе кода; это оправдывает новый крупный релиз, который впервые больше не совместим с ранними версиями. Пришло время новых начинаний.

Zend Framework 2 знаменует новый этап в развитии PHP-фреймворков, но также и самого PHP, поскольку, как можно видеть на примере Java и Ruby, пользы от одного языка программирования недостаточно, чтобы быть по-настоящему успешным на широком фронте. Только фреймворки, такие, как Struts, Spring, Rails, или, на самом деле, Zend Framework, основанные на языке программирования и способные существенно снизить первоначальные усилия на разработку и долгосрочное сопровождение приложений, делают веб-платформу разработки действительно таковой.

Для кого эта книга?

Написать техническую книгу, которая находит баланс между теорией и практикой и позволяет как новичкам, так и профессионалам получить лучшее из книги – это вызов. Я с радостью взялся за эту задачу, но я оставил себе аварийный люк. Если у меня появляется ощущение, что мы заблудились в массе деталей, что ознакомление не может быть объяснено более подробно в соответствующем месте или даже полностью в этой книге, я отсылаю читателя к соответствующим отрывкам в книге или ко вторичным литературным источникам.

Еще одной проблемой является наличие или отсутствие у моих читателей каких-либо предварительных знаний о Zend Framework. Для разработчика, который только начал работать с версией 2 Zend Framework, требуется другая информация в некоторых местах, чем для "стариков", которые быстро находят свой путь вокруг многих уголков в версии 2, потому что он или она уже знакомы с идеями и концепциями версии 1. Для всех тех, кто знаком с версией 1, я буду часто ссылаться на предшественника версии 2 в соответствующих местах, всякий раз, когда я считаю нужным, не вдаваясь в излишние подробности. Это, возможно, поможет и новичкам, потому что таким образом они получат лучшее представление о том, зачем необходима 2 версия. Эта книга может быть полезна как новичкам, так и продвинутым пользователям Zend Framework.

Я предполагаю, что у вас есть базовые знания PHP. Вам не нужно быть экспертом PHP, в частности потому, что многие "родные" функции PHP даже устарели, в то время, как использующиеся фреймворком, например, управление сессиями, отображены в объектно-ориентированной манере, и, таким образом, к счастью, абстрагируются от некоторых низких уровней функциональности. Поэтому, если вы привыкли к синтаксису PHP, у вас есть базовое понимание принципов работы PHP-приложений, и вы знакомы с общими функциями ядра языка, вы хорошо подготовлены. При необходимости, вы также должны использовать руководства по PHP.

Структура этой книги

Эта книга не претендует на то, чтобы быть справочником, а, скорее, является прагматичным и практико-ориентированным введением в основные концепции и практическую работу с Zend Framework 2. С определенной точки в продвижении разработчика вперед официальная документация служит справочником для более опытных программистов, но реально она не подходит во время ознакомления с предметом; вместо этого она служит ссылкой для дальнейшего подробного изучения вопроса после того, как достигнуто необходимое общее представление о предмете. И именно это является целью данной книги.

Она построена таким образом, что вы можете прочитать ее от начала и до конца, и это то, что вы должны сделать. Мы начнем с обзора фреймворка и сначала рассмотрим основные понятия и идеи, которые составляют суть фреймворка, а также отличают его от своих

предшественников. По пути мы также неоднократно будем смотреть налево и направо, и, таким образом, ознакомимся с некоторыми условиями фреймворка, например как фреймворк был реально разработан. Тогда мы будем вдаваться в подробности и объяснять наиболее важные компоненты и отношения фреймворка; а затем рассмотрим, как обрабатывается HTTP запрос и напомним наш первый бит кода. Далее следует экскурс в среду фреймворка. Мы рассмотрим наиболее важные модули, которые были предоставлены третьими лицами, и, например, сделали реализацию пользовательского администрирования вышедшей из употребления. Большая часть магии новой версии фреймворка является результатом модульной концепции, и модули могут, как мы увидим позже, значительно ускорить и упростить разработку собственных приложений. Следует ожидать, что в течение короткого времени большое количество высококачественных модулей будут доступны для Zend Framework 2.

И последнее, что не менее важно, имеется дополнительная, интенсивная практика в разделе "дневники разработчиков". Мы вместе разработаем веб-приложение, которое предназначено для реального использования в дальнейшем для коммерческого предприятия, поэтому, будет лучше, если мы действительно будем стараться. И, наконец, мы начнем выполнять реальные практические работы.

Повторения

Вы скоро поймете, что я часто объясняю в этой книге контексты по несколько раз. Это может быть по двум причинам: 1) я потерял нить рассуждений, или же не имел ее (:-D) 2) я сознательно это сделал. Даже древние римляне знали: Repetitio mater studiorum est – повторение – мать учения. Но суть в том, что обсуждение в соответствующих контекстах, в разных условиях, в каждом конкретном случае, и, следовательно, с другой стороны, также способствует пониманию. Таким образом, если вам посчастливилось найти место в книге, где вы думаете, "я уже все это знаю!", просто радуйтесь тому, что вы так многому научились и продолжайте читать или просто пропустите соответствующий отрывок.

Как лучше всего работать с этой книгой

Программирование (или работа с программным фреймворком) изучается лучше всего тогда, когда вы сами проявляете активность. Действительно, в частности, первая глава, вплоть до практического раздела, уже полезна даже без открытой IDE, но наибольшего успеха вы достигнете, если воспроизведете некоторые строки кода или напишите некоторые самостоятельно. В идеале, у вас есть под рукой система и отладчик, с которыми вы работаете с фреймворком в пошаговом режиме. Некоторые знания Git и GitHub тоже не помешали бы, но не обязательно.

Нашли ошибку?

Вы нашли ошибку в тексте или коде? Пожалуйста, не стесняйтесь, сообщите об ошибке в bug tracker (<https://github.com/michael-romer/zf2book/issues>). Я буду благодарен за ваши отзывы и поддержку.

Условные обозначения, используемые в этой книге

Примеры кода

Листинги этой книги имеют подсветку синтаксиса, где это возможно, но не всегда соответствуют стандартам кодирования; это служит для того, чтобы сделать их более читаемыми. PHP код вводится, как `<?php`, а `//` [...] в соответствующих листингах обозначает исключенные фрагменты кода. Многие листинги (примеры кода) содержат ссылку на GitHub, откуда могут быть загружены в виде так называемых "gist", или просто получены методом "Copy & Paste". Таким образом, примеры кода могут быть легко воспроизведены на вашей собственной системе.

Командная строка

Если команды должны быть выполнены из командной строки, это символизирует предшествующий знак доллара (\$). Визуальный отклик команды обозначается знаком "предыдущее более" (>).

Пример:

```
1 $ phpunit --version
2 > PHPUnit 3.6.12 by Sebastian Bergmann.
```

Итак, достаточно предисловий, давайте приступим к работе!

III. Zend Framework 2 – Обзор

Прежде, чем начать погружаться по ходу книги в детали фреймворка, хотелось бы для начала получить обзор и выяснить некоторые основные аспекты и мысли, лежащие в основе фреймворка. Каковы основные характеристики фреймворка?

Как в настоящее время разрабатывается фреймворк?

Zend Framework – фреймворк с открытым исходным кодом. Прежде всего, это означает, что любой человек может изучить исходный код и использовать его в своих целях. В настоящее время фреймворк разрабатывается в рамках лицензии "New BSD License". В то время, как в соответствии с первоначальной "BSD License" все равно нужно было сослаться на использование библиотеки или фреймворка, в "New BSD License", для Zend Framework 2 это не обязательно; так называемый "маркетинговый пункт" не существует в "New BSD License". Не хотелось бы углубляться в законодательство о лицензировании программного обеспечения, но хочу сказать, что на данный момент, со ссылкой на закон, фреймворк облегчает нам жизнь, потому, что "New BSD License" действительно относится к так называемым бесплатным лицензиям на программное обеспечение, у которых практически нет ограничений или директив на использование кода.

В то время, как версия 1 до сих пор использует SVN для администрирования кода, команда проекта для нового выпуска решила администрировать код фреймворка с помощью GitHub (<https://github.com/zendframework/zf2>). GitHub очень хорошо поддерживает параллельную распределенную работу над кодом, тем более, что в разработке участвуют многие программисты и она распространилась по всему миру. Таким образом, Rob Allen проживает в Англии, в то время как Matthew Weier O'Phinney, руководитель проекта для Zend Framework 2, живет в США, а Ben Scholzen – в Германии. Вышеупомянутые программисты являются так называемыми "участниками разработки", то есть они уже сделали значительный вклад во фреймворк, и, следовательно, имеют особый статус в команде. Они решительно формируют структуру и дизайн фреймворка. Некоторые программисты работают непосредственно в Zend, например, Matthew Weier O'Phinney; другие являются фрилансерами или имеют другие условия труда, которые позволяют им совместно работать над Zend Framework – во время или после их работы. Таким образом, это яркая группа хороших людей. Компонентная ориентация Zend Framework позволяет использовать так называемых "сопровождающих компонента" (contributors) (<http://framework.zend.com/wiki/display/ZFDEV2/Component+Maintainers>), которые отвечают за конкретный компонент фреймворка, и они сами или совместно с другими программистами управляют судьбой определенного компонента.

Команда организует себя прежде всего через wiki (<http://framework.zend.com/wiki/dashboard.action>), почтовую рассылку и IRC чат. "Процесс предложений по Zend Framework" дает возможность каждому предоставлять предложения

по дальнейшему развитию фреймворка и регулярные "дни отловов ошибок" помогают целенаправленно объединять силы для исправления обнаруженных во фреймворке проблем. Становятся доступными и регулярно публикуются новые версии фреймворка с исправленными ошибками и новыми возможностями.

Стандарт кодирования PSR-2

Код фреймворка разрабатывается для всех компонентов с учетом стандарта кодирования PSR-2 (<https://github.com/php-fig/fig-standards/tree/master/accepted>). Идея "PHP Specification Request", для краткости PSR, родилась, будучи вдохновленной Java Specification Request (http://de.wikipedia.org/wiki/Java_Specification_Request). Используя эту методику, определены новые стандарты Java и расширения языка программирования Java или Java Runtime Environment, совместно разработаны и согласованы со всеми производителями. У этой методики много преимуществ для разработчиков приложений, поскольку это делает приложения должным образом более портативными и независимыми от производителей. Таким образом можно, например, более или менее просто сменить провайдера собственного сервера приложений.

"PHP Specification Requests" основаны на аналогичной идее — они должны, в частности, обеспечивать совместимость программных компонентов и фреймворков различных производителей друг с другом и возможность их использования в комбинации. В отличие от "JSR", методика "PSR" относительно новая. На сегодняшний день были согласованы только три характеристики:

- PSR-0: Определяет когерентности между пространствами имен PHP и организацией PHP файлов в файловой системе для того, чтобы сделать автозагрузку классов, которая также должна быть независимой от компонентов и производителей, как можно более простой.
- PSR-1 / PSR-1 basic: Новый общий стандарт кодирования.
- PSR-2: Расширение стандарта кодирования PSR-1.

Основной акцент в Zend Framework, как мы неоднократно видим в ходе этой книги, сделан на расширении функциональности приложений с помощью многократно используемых, либо просто интегрированных модулей. Соблюдение стандартов PSR является большим подспорьем в этом контексте.

Известные проблемы

Как и в случае для каждой большой части программного кода, фреймворк не может быть полностью свободным от ошибок.

Исходя из этого, GitHub issues (<https://github.com/zendframework/zf2/issues?state=open>) используется для трассировки... Таким образом, если вы обнаружили проблему, стоит сначала посмотреть там, чтобы увидеть, не является ли эта ошибка уже известной. Если нет, то там вы можете открыть вопрос.

Модульная система

Модульная система – это центральный узел 2 версии. Matthew Weier O’Phinney заявил об этом предельно ясно в списке рассылки:

"Модули являются, пожалуй, наиболее важной новой разработкой в ZF2."

Система модулей фреймворка была полностью переработана для версии 2. Даже если уже действительно можно было организовать собственный код в модули, эти модули никогда не были действительно полезными независимо и не могли быть переданы в другие приложения без необходимости подгонки модуля к соответствующему коду с многочисленными усилиями. Усилия эти были настолько велики, что проще было запрограммировать подобную функциональность самостоятельно. Модульная система версии 1 была, таким образом, ограничена преимуществами лучшей организации кода в автономных приложениях. Результатом является то, что сейчас, наверное, существуют тысячи реализаций для проверки подлинности пользователей, или аналогичных функций, которые, как правило, применимы, поскольку они специфичны для конкретного приложения.

Мы привыкли добавлять функциональность с помощью плагинов и расширений в такие приложения, как WordPress, Drupal или Magento. Даже у Symfony 2 (<http://symfony.com>) – популярного альтернативного веб-фреймворка с бандлами – есть модульная концепция, что делает возможным, например, интеграцию системы управления контентом (CMS) в собственные приложения без необходимости тратить время на самостоятельное программирование. А теперь Zend Framework также включил эту расширенную возможность в новой версии. Снова приведем конкретный пример: есть приложение, разработанное на основе Zend Framework, в дальнейшем требуется функциональность блога. Это обычное требование, поскольку блог является чрезвычайно практичным, например, для SEO мероприятий. Любой, кто работал с Wordpress & Co. знает, насколько широки стали существующие требования к современной блоговой системе. Быстро становится ясно, что развивать собственное программное обеспечение для блога – не самая лучшая идея.

Напротив, представляется целесообразным использовать одну из доступных бесплатных или коммерческих блогговых систем, которая, однако, в виду ее концепции может быть установлена лишь параллельно с собственным приложением. У этого есть ряд недостатков: например, нелегко будет использовать систему регистрации пользователей в вашем приложении без танцев с бубном, недоступны будут кэширующие слои. Данные блога находятся в другой базе данных, и если мы хотим отобразить тизеры последних 3 публикаций блога на главной странице приложения, то, чтобы реализовать эти врезки, нужно использовать API блога, возможно, через RSS канал, или что-нибудь в этом роде (или возиться с базами данных сторонних приложений...). Естественно, что учетные записи администраторов, которые позволяют нашим сотрудникам управлять основными пользовательскими данными, не существуют в системе блога, и о технологии единого входа (http://ru.wikipedia.org/wiki/Технология_единого_входа) не может быть и речи. Мы должны имитировать наш корпоративный дизайн в системе шаблонов блога, так как

компоновка, разметка и стили там не всегда доступны. В последствии, любую адаптацию дизайна нужно будет там реконструировать. Скрипты нашего приложения не могут быть использованы для блога, напротив, релизы должны быть адаптированы так, чтобы как-то учитывать стороннюю систему блога. Таким образом, при данном подходе мы буксуем из-за сложности интеграции корпоративных приложений, Enterprise Application Integration (EAI) (http://ru.wikipedia.org/wiki/Enterprise_Application_Integration) и отсутствия сервис-ориентированной архитектуры, Service Oriented Architecture (SOA) (http://ru.wikipedia.org/wiki/Сервис-ориентированная_архитектура) и, таким образом, создаем себе красочный букет новых задач и проблем.

Если бы вместо этого был бы доступен модуль блога для Zend Framework 2, который легко бы интегрировался в существующую аутентификацию, разработку & релиз, кэширование, логирование, и реализацию дизайна – все было бы намного проще, практически, тривиально. Именно эта мысль является ключевой идеей модульной системы. В модуле Zend Framework есть все, что необходимо для его работы. Это включает в себя не только соответствующий код PHP, но и HTML шаблоны, CSS, код JavaScript, изображения, и т.д., так что модуль является по-настоящему автономным пакетом. Хороший модуль может быть легко интегрирован в приложения Zend Framework 2 и ... просто работает. Модульная система, таким образом, делает Zend Framework 2 чем-то большим, нежели просто веб-фреймворк, более того, она далеко выходит за эти рамки и действительно является платформой для интегрированных приложений и функций. В следующей главе мы расскажем подробнее о технических деталях модульной системы, и рассмотрим, как она функционирует внутри, и как разрабатываются модули, поскольку даже собственное приложение представлено в коде в виде модуля. В новом Zend Framework модули повсюду! Если вы понимаете модули, это уже пол-дела, как для разработки собственных приложений, так и для встраивания уже существующих функций и систем. Далее в книге мы рассмотрим некоторые существующие модули потому, что кроме упомянутых функций по управлению пользователями, их много больше, например, модули, относящиеся к Doctrine 2 (<http://www.doctrine-project.org>), известной PHP-ORM системе (http://de.wikipedia.org/wiki/Objektrelationale_Abbildung). Эти модули используют так называемый "glue code", что позволяет легко использовать эту библиотеку в приложениях на Zend Framework 2.

Система событий

В основе Zend Framework 2 лежит концепция событийно-ориентированной архитектуры, Event-driven architecture (EDA) (http://ru.wikipedia.org/wiki/Событийно-ориентированная_архитектура). Этот подход основан на идее того, что определенные действия происходят в системе после того, как произошло определенное событие. Чтобы добиться этого, действие регистрируется для уведомления о том, что событие произошло. По аналогии с реальным миром: когда мы ждем на автобусной остановке (мы "зарегистрировались" на событие прибытия автобуса), и когда автобус наконец подъезжает (происходит событие), его дверь открывается (действие 1), мы покупаем билет (действие 2), ищем свободное место (действие 3), и автобус трогается (действие 4).

Вот еще один пример: продажа эксклюзивного автомобиля на eBay. Это событие вызывает ряд действий системы:

- Подтверждение покупки отправляется покупателю.
- Подтверждение продажи отправляется продавцу.
- Вычисляется сумма продажи и переводится на счет продавца.
- Авто, о котором идет речь, будет удалено из поискового индекса и уже не сможет быть найдено (оно уже продано).

Если eBay позже решит проинформировать проигравших участников о том, что аукцион уже закончился (на самом деле, это уже сделано), это действие может быть добавлено к другим действиям для этого события.

Тем не менее, EDA является "палкой о двух концах". С одной стороны, она достигает невероятной гибкости в структурировании процессов, использующих этот стиль архитектуры. Новые действия могут быть легко добавлены. Таким образом все методы могут быть впоследствии легко модифицированы. Гибкость является решающим аргументом. В противоположность этому, отсутствует четкая информация о том, что же действительно происходит в приложении при наступлении события. По существу, реакция на событие может быть зарегистрирована практически в любом месте приложения, и эта связь может не просматриваться в коде, где это событие действительно происходит впоследствии. Еще одной проблемой является последовательность действий. Если продавец eBay в примере, приведенном выше, также уведомляется о причитающемся гонораре (который вычисляется на основе конечной цены аукциона), это действие должно происходить после того, как сумма будет вычислена (т.е., должно произойти другое действие). Теперь вещи начинают усложняться. В двух словах, EDA может привести к процессам, которые трудно понять. Причины ошибок сложнее идентифицировать и отладка приложений становится более сложной.

Не взирая на это, преимущества перевешивают недостатки настолько сильно, в частности, и в связи с модульной системой фреймворка, что разработчики решили использовать EDA для Zend Framework 2. Если вы знаете, где лежат подводные камни, то вы можете и легко избежать их.

Реализация MVC

В Zend Framework 2 реализация MVC также играет не малую роль. Хотя Zend Framework 2 и предоставляет возможность свободного использования его компонентов, например, игнорируя MVC, на практике это следует делать только в исключительных случаях. В действительности, большинство случаев – это как раз реализация MVC, и, как результат, выгодная структура кода в собственном приложении, что часто является основой для выбора Zend Framework. Zend\Mvc структурирует приложение с помощью логического разделения кода компонентов на модели, виды и контроллеры, и таким образом обеспечивает определенный порядок, который полезен не только для работоспособности и

расширяемости приложения, но и способствует повторному использованию функциональности.

А так выглядит Zend\Mvc в действии: после того, как Zend\ModuleManager, центральное звено модульной системы, подготовит все доступные модули для использования, прочтет конфигурацию и инициализирует дополнительные компоненты, Zend\Mvc\Router обеспечивает создание подходящего класса контроллера (Controller), конкретизированного в модуле, и в нем вызывается правильное действие (метод). В данном контексте маршрутизация основана на предварительно сконфигурированных маршрутах, которые представляют собой соответствия между URL и контроллерами и действиями соответственно. Выбранный контроллер возвращает на новое рассмотрение объект запроса (Request Object) для последующей обработки. Это позволяет объектно-ориентированный доступ к параметрам запроса, доступного в объекте MvcEvent. Последний, в свою очередь, был создан в начале и будет предоставлен в соответствующих местах. Затем контроллер использует прикладные модели и получает доступ к постоянным данным, услугам и бизнес-логике приложения. Он, в конечном счете, генерирует "модель представления" (view model), и на ее основе и на основе соответствующих HTML шаблонов (templates), а также с использованием так называемых "помощников представления" (view helper), генерируется результат вызова. При необходимости вывод теперь также вставляется в макет (layout) и конечный результат возвращается в вызывающую программу. Но более подробно мы это рассмотрим позже.

Поскольку Zend\Mvc широко использует систему событий фреймворка, здесь предусмотрено множество параметров для влияния на стандартную линию поведения, упомянутую выше. Таким образом, например, управление доступом или входной фильтр могут быть выполнены до приведения в действие контроллера. Перед возвращением результатов можно убедиться, что сгенерированная HTML разметка не содержит ошибок, при необходимости, с помощью HTMLPurifier (<http://htmlpurifier.org>).

Код для реализации MVC является совершенно новой разработкой. Реализация MVC во фреймворке 1 версии по-прежнему довольно негибкая, в то время, как новая версия, безусловно, более гибкая и, в конечном счете, позволяет выполнить настройку любого произвольно адаптированного бизнес-процесса.

Дополнительные компоненты

В дополнение к Zend\Mvc, Zend\View, Zend\ServiceManager, Zend\EventManager и Zend\ModuleManager, которые вместе составляют "ядро каркаса", Zend Framework 2 имеет ряд дополнительных компонентов, которых мы вкратце рассмотрим далее. Большинство этих компонентов будут рассматриваться более подробно впоследствии в этой книге. Тогда мы будем иметь дело с каждым соответствующим компонентом более интенсивно. В отличие от многих других фреймворков, Zend Framework проектировался таким образом, что можно использовать только выбранные компоненты, в то время, как другие компоненты можно проигнорировать.

При рассмотрении списка компонентов те, кто использовал Zend Framework 1, могут заметить, что некоторых компонент больше не существует. В частности, многие компоненты для подключения к различным веб-сервисам больше не являются частью фреймворка версии 2.0, но сохраняются, как самостоятельные проекты или библиотеки. Так, например, Zend_Service_Twitter больше не является частью программы, но в настоящее время реализуется на Git аккаунте zendframework (<https://github.com/zendframework>) в собственном репозитории. Смысл этого в том, чтобы очертить профиль фреймворка и сфокусировать его прикладную область. Также в новой версии отсутствует Zend_Registry. Его задачу теперь выполняет ServiceManager, который, как мы увидим в дальнейшем, может сделать намного больше. Zend_Test также отсутствует. Хорошая новость заключается в том, что в результате слабой связанности отдельных объектов и сервисов, которая преобладает в Zend Framework 2, никаких дополнительных функций, кроме тех, что доступны в PHPUnit (<https://github.com/sebastianbergmann/phpunit>), для тестирования не требуется. Это очень хорошая новость, и, как будет видно в следующей главе и в практическом разделе этой книги, модульное тестирование стало намного проще в новой версии.

В стандартной комплектации Zend Framework 2 содержит следующие дополнительные компоненты:

- **Authentication:** служит для реализации функции "входа в систему", в которой компьютер проверяет, действительно ли пользователь является тем, за кого он себя выдает (например, на основании того, что он знает секретный пароль).
- **Barcode:** библиотека для создания штрих-кодов.
- **Cache & Memory:** универсальная реализация кэширующей системы на различных движках по усмотрению, например, Memcached или APC.
- **Captcha:** генерирование CAPTCHA, например, для использования в веб-формах.
- **Code:** инструменты для автоматической генерации кода.
- **Config:** инструмент, обрабатывающий чтение и запись конфигураций приложения в самые разнообразные форматы, например, YAML или XML.
- **Console:** библиотека для использования функциональности приложения, например, контроллера, из командной строки (вместо HTTP запросов браузера).
- **Crypt:** функции шифрования.
- **Db:** библиотека для работы с базами данных (но не ORM система).
- **Di:** реализация контейнера внедрения зависимостей, Dependency Injection (DI).
- **Dom:** библиотека для работы с DOM на стороне сервера.
- **Escaper:** инструмент экранирования выходных данных.
- **Feed:** Генерация RSS и Atom-лент.
- **File:** поддержка работы с файлами.
- **Filter:** функции для фильтрации данных, например, в рамках веб-форм.
- **Form:** библиотека для основанной на PHP объектно-ориентированной генерации веб-форм с учетом валидаторов и фильтров.

- **Http**: поддержка работы с HTTP.
- **I18n**: обширная библиотека для интернационализации приложений, например, для вывода переведенного содержимого.
- **InputFilter**: позволяет применять фильтры и валидаторы к полученным данным.
- **Json**: инструмент для сериализации и десериализации структур данных JSON.
- **Ldap**: библиотека для связывания LDAP систем, например, в сочетании с **Authentication**.
- **Loader**: автозагрузка функций для PHP классов, а также загрузка MVC модулей.
- **Log**: реализация универсальной функциональности логгера с поддержкой различных типов журналирования.
- **Mail & Mime**: библиотека для отправки (составных-) сообщений электронной почты.
- **Math**: поддержка различных математических операций.
- **Navigation**: генерация и вывод навигации сайта.
- **Paginator**: генерация и вывод постраничной навигации, например, в виде результирующих списков.
- **Permissions**: библиотека для управления правами доступа к системе.
- **ProgressBar**: генерация и воспроизведение индикаторов выполнения (в том числе и в командной строке).
- **Serializer**: инструменты сериализации и десериализации объектов, например, для длительного хранения.
- **Server, Soap & XmlRpc**: библиотека для генерации веб-сервисов, например, на основе SOAP или XML-RPC. Часть Soap также полезна для реализации клиента SOAP.
- **Session**: администрирование пользовательских сессий.
- **Stdlib**: разнообразные стандартные функции и объекты, например, реализация "Userland PriorityQueue", которую, кстати, использует EventManager.
- **Tag**: функции администрирования тегов, например, генерации облака тегов для сайта.
- **Text**: средство обработки строк и скриптов. Используется внутренне, например, для Captcha.
- **Uri**: функции для генерации и валидации URI.
- **Validator**: обширная библиотека для синтаксической проверки допустимости данных, например, записей в формах для множественных прикладных случаев, среди которых ISBN, IBAN, адреса электронной почты и многое другое.
- **Version**: содержит информацию об используемой версии фреймворка.

Шаблоны проектирования: Interface, Factory, Manager, и другие.

Если начать рассматривать код Zend Framework 2, через некоторое время можно заметить, что он переполнен реализациями так называемых "архитектурных шаблонов проектирования" (http://ru.wikipedia.org/wiki/Шаблон_проектирования). Если вы более знакомы с "типичным" PHP кодом – и это не в коем случае не осуждение – то это займет какое-то время, прежде, чем вы найдете свой путь к Zend Framework 2. Если же вы уже работали с Java – вы будете чувствовать себя намного комфортнее просто потому, что шаблоны проектирования нашли свое место в мире Java несколькими годами ранее или даже развились там, соответственно. Чтобы упростить для вас материал, давайте далее рассмотрим несколько простых конструкций Zend Framework 2. Строго говоря, не все из них являются шаблонами проектирования, но на данный момент мы должны игнорировать этот факт. Таким же образом вы должны отнестись к тому, что я использую приемлемые упрощения в своих объяснениях в некоторых местах книги. Это не всесторонняя книга по шаблонам разработки, и те знания, которые должны быть получены в первую очередь, предназначены помочь читателю развить понимание Zend Framework. В то же время, возможно, следует повторить следующий совет: разработчику приложений не нужно понимать всю механику Zend Framework 2 в деталях. Напротив, фреймворк предназначен для того, чтобы уменьшить затрачиваемые на работу усилия и позволить сконцентрироваться на программировании самой бизнес-логики. Тем не менее, это может очень помочь, если у вас есть фундаментальное понимание взаимосвязи отдельных компонентов фреймворка, и, что не менее важно, базовой концепции. Таким образом, не стоит пропускать эту главу.

Интерфейс (Interface)

Интерфейсы являются неотъемлемым элементом объектно-ориентированного программирования, PHP всесторонне поддерживает их, начиная с 5 версии. Интерфейсы позволяют отделить код вызова от конкретной реализации. Если вы разрабатываете приложение под определенный интерфейс, можно не сомневаться в том, что предусмотренные методы будут доступны во время выполнения программы. Действительно, можно также использовать альтернативную реализацию, не изменяя вызывающий код.

Слушатель (Listener)

Слушатель – короткая строка кода, который выполняется, как только определенное событие происходит в приложении. С технической точки зрения, чтобы добиться этого, слушатель регистрируется заранее, используя так называемый "EventManager" для какого-либо события.

Агрегат слушателей (ListenerAggregate)

ListenerAggregate объединяет ряд слушателей вместе, например, для того, чтобы зарегистрировать их в EventManager. Не намного более, но и не менее того.

Фабрика (Factory)

Фабрика используется в том случае, если создание экземпляра конкретного объекта является сложным, т.е., когда требуется целый ряд манипуляций, чтобы сделать объект готовым к использованию. Например, Zend Framework использует фабрику для создания экземпляров ModuleManager и, кроме того, чтобы зарегистрировать ряд соответствующих слушателей модуля.

Сервис (Service)

Сервис обеспечивает доступ к определенным файлам или функциям. Этот термин является весьма общим и может иметь совершенно иной смысл в каждом конкретном случае в зависимости от контекста. В рамках фреймворка под службой понимают объект, который сделан доступным из ServiceManager и предоставляет определенную услугу. Таким образом, слушателей, зарегистрированных в ServiceManager, также называют сервисами, как, например, и ModuleManager, но, к тому же, и controller, и view helper.

Менеджер (Manager)

Менеджер – это объект, который управляет администрированием определенного типа другого объекта в системе. Например, у Doctrine 2 (<http://www.doctrine-project.org>) есть так называемый EntityManager, который администрирует сущности (entity), т.е. определенные постоянные объекты (например, для магазина: предложения, категории, клиенты, заказы и т.д.) в течении всего их жизненного цикла и гарантирует, что объекты считаны из базы данных и изменения прозрачно возвращены.

Стратегия (Strategy)

За шаблоном проектирования Strategy стоит идея взаимозаменяемости алгоритмов, которые в противном случае были бы "монолитной конструкцией" в коде вашего класса, и, таким образом, возможность сделать эти алгоритмы сменными. Соответственно, алгоритмы сортировки, например, являются хорошими кандидатами для шаблона стратегии. Различные алгоритмы, согласно которым, например, могут быть отсортированы списки продуктов (увеличение/уменьшение цены, увеличение/уменьшение рейтинга, и т.д.), жестко не закодированы, но вместо этого реализованы в виде отдельных классов и все они реализуют единый интерфейс с обязательным методом sort(). Если вы когда-либо внедрили подобный механизм, любую другую произвольную процедуру сортировки впоследствии будет легко разработать и добавить, создав новый класс, реализующий этот интерфейс.

Модель-Представление-Контроллер (Model View Controller (MVC))

Шаблон MVC значительно влияет на структуру кода приложения потому, что логически отделяет друг от друга те компоненты, которые управляют представлением (View), обработкой взаимодействия с пользователем (Controller) и бизнес-логикой (Model) с ее объектами и сервисами.

Действия (Actions)

Действия – это подход для дальнейшего структурирования кода, используемого для обработки взаимодействия с пользователем в контроллерах. Поэтому они тесно связаны с шаблоном MVC и также используются в Zend Framework 2.

Помощник представления (View Helper)

С помощью помощников представления код для логики представления может быть инкапсулирован и повторно использован стандартным образом.

Плагины контроллера (Controller Plugins)

Посредством плагинов контроллера часто используемый код может быть организован для взаимосвязанной обработки и использоваться в нескольких контроллерах.

IV. Hello, Zend Framework 2!

Оставим серую теорию и посмотрим на фреймворк в действии

Как отмечалось в предыдущей главе, Zend\Mvc является независимым, необязательным, но важным компонентом фреймворка. Zend\Mvc также навязывает приложению определенную структуру кода и каталогов. Однако, фактически это довольно практично. Если вы уже знакомы с приложениями на Zend Framework 2, то можете легко сориентироваться в других приложениях на основе фреймворка. Хотя у вас действительно есть свобода в создании абсолютно отличающейся структуры каталогов и кода, это сделало бы жизнь слишком сложной, как мы увидим в дальнейшем. Если создавать приложение, имеет смысл использовать Zend\Mvc с самого начала. Однако, если вы хотите расширить существующее приложение функциями Zend Framework 2, то, пожалуй, имеет смысл отказаться от Zend\Mvc полностью, либо поначалу, а использовать впоследствии.

Zend Framework 1 и 2 параллельно

Можно также работать с Zend Framework 2 параллельно с 1 версией и вначале использовать Zend Framework 2 лишь время от времени.

Установка

В принципе, установка Zend Framework 2 не должна быть утомительной. Можно просто загрузить код (<https://packages.zendframework.com>), сделать его доступным на веб-сервере с установленным PHP и сразу начинать. Тем не менее, проблема состоит в том, что кода Zend Framework 2 самого по себе недостаточно, чтобы реально увидеть Zend\Mvc-приложение в действии. Как уже упоминалось, Zend\Mvc, т.е. компонент, который ответственен за обработку HTTP запросов, не является обязательным компонентом, и, соответственно своей сути, "вшитым" для использования. Таким образом, для начала вам необходимо убедиться, что Zend\Mvc оснащен конфигурацией и кодом инициализации – так называемым "шаблонным кодом", чтобы его можно было действительно использовать. В противном случае, сначала ... вы не увидите ничего.

Чтобы избежать этих усилий и сделать начало работы со 2 версией максимально простым, в ходе разработки фреймворка был создан так называемый "ZendSkeletonApplication", он служит шаблоном для собственного проекта и содержит необходимый "шаблонный код", который в противном случае пришлось бы готовить самостоятельно с большим трудом.

ZendSkeletonApplication

Установка ZendSkeletonApplication является простейшей с помощью Git. Чтобы воспользоваться ей, необходимо сначала установить Git на своем компьютере. На системах Mac и во многих дистрибутивах Linux Git уже предварительно установлен. Для установки на Windows-систему доступен Git для Windows (<http://msysgit.github.io>). Установка под Linux почти всегда работает под соответствующим диспетчером пакета. После установки и вызова наберите в командной строке

```
$ git --version
```

можно будет увидеть этот или подобный "признак жизни":

```
> git version 1.7.0.4
```

Далее все очень просто – перейдите в каталог, в котором должен быть создан подкаталог для приложения и который будет доступен для веб-сервера, как корень документа (document root), и скачайте ZendSkeletonApplication:

```
$ git clone git://github.com/zendframework/ZendSkeletonApplication.git
```

Правда, на жаргоне Git это называется "клонировать", а не "загружать". Но мы пока проигнорируем это обстоятельство. И, кстати, не бойтесь Git! Вам не нужно знать о Git что-либо еще, чтобы успешно работать с этой книгой и фреймворком. Конечно, в далеком будущем читатель также может управлять своим собственным кодом приложения даже постоянно с помощью Git, но это не является необходимостью. Следовательно, впоследствии вместо Git можно будет использовать для управления собственным кодом subversion, CVS или вообще не использовать какую-либо систему без всяких проблем.

Загрузка ZendSkeletonApplication происходит очень быстро даже для медленных интернет-соединений, но такое соединение должно быть в любом случае. Причина быстрой загрузки в том, что код самого фреймворка не скачивается, а только соответствующий шаблонный код для разработки собственных приложений, основанный на Zend Framework 2.

Composer

ZendSkeletonApplication использует Composer (<http://getcomposer.org>), еще один инструмент на базе PHP, хорошо зарекомендовавший себя некоторое время назад для управления зависимостями других библиотек кода. Идея composer столь же проста, сколь гениальна. Конфигурационный файл composer с расширением .json содержит определение библиотек кода, от которых зависит приложение, и адреса, откуда эти библиотеки могут быть получены. В нашем случае приложение зависит от Zend Framework 2, как можно увидеть в файле composer.json в корневом каталоге приложения.

```
1 {
2     "name": "zendframework/skeleton-application",
3     "description": "Skeleton Application for ZF2",
```

```

4  "license": "BSD-3-Clause",
5  "keywords": [
6      "framework",
7      "zf2"
8  ],
9  "homepage": "http://framework.zend.com/",
10 "require": {
11     "php": ">=5.3.3",
12     "zendframework/zendframework": "2.*"
13 }
14 }

```

Listing 4.1 (<https://gist.github.com/3820657>)

В строках 11 и 12 объявлены две зависимости приложения. Требуются PHP выше версии 5.3.3 и Zend Framework выше 2 версии. Следующие две команды обеспечивают загрузку Zend Framework 2 и дополнительно интегрируют его в приложение таким образом, чтобы он сразу был готов к использованию и соответствующие классы фреймворка были доступны для автозагрузки:

```

1 $ cd ZendSkeletonApplication
2 $ php composer.phar install
3 > Installing zendframework/zendframework (dev-master)

```

Теперь composer скачал Zend Framework 2 в каталог vendor и сделал его доступным к применению.

Phar-Архивы

Phar-архивы обеспечивают возможность создания PHP приложений, доступных в виде единственного файла. Если взглянуть на репозиторий composer на GitHub, становится ясно, что composer состоит не из одного файла, как сперва может показаться, а что его компоненты упакованы в Phar-архив для распространения приложения. (<https://github.com/composer/composer>)

Phar-архивы и Suhosin

Если в вашей системе используется Suhosin, то сначала использование Phar необходимо разрешить явным образом, добавив в suhosin.ini строку

```
suhosin.executor.include.whitelist=phar
```

В противном случае при выполнении команд composer могут возникнуть проблемы.

(<http://www.hardened-php.net/suhosin>)

Инсталляция без использования Git и Composer

При необходимости фреймворк и ZendSkeletonApplication можно загрузить обычным образом или с помощью Ptgus (установщик PEAR). Дополнительную информацию можно найти на официальном сайте загрузки (<http://framework.zend.com/downloads>).

Также необходимые файлы можно скачать обычным способом в виде zip архива с GitHub (<https://github.com/zendframework/ZendSkeletonApplication>), нажав на кнопку с надписью «Zip».

Первые признаки жизни

Мы уже завершили все необходимые приготовления. Теперь нам нужно добавить виртуальный хост для вашего веб-сервера так, чтобы набрав в браузере URL `http://hello.zend`, веб-сервер открыл для вас ваше первое приложение. В принципе, порядок добавления виртуального хоста зависит от используемых вами операционной системы и веб-сервера. В общем случае настройки для веб-сервера Apache описаны ниже.

Для установки виртуального хоста для Apache на базе ОС Debian (например, Ubuntu), вам необходимо сначала создать конфигурационный файл виртуального хоста `hello.zend.conf` в каталоге `/etc/apache2/sites-available`, а затем создать на него символическую ссылку в каталоге `/etc/apache2/sites-enabled`, причем от имени пользователя `root`:

```
# ln -L /etc/apache2/sites-available/hello.zend.conf /etc/apache2/sites-enabled/hello.zend.conf
```

Затем для пользователей этих ОС в файле `/etc/apache2/sites-available/hello.zend.conf` необходимо прописать конфигурацию виртуального хоста, приведенную далее по тексту.

Обычно же настройка виртуального хоста осуществляется в файле `httpd.conf` или `extra/httpd-vhosts.conf`. Если же Вы используете `httpd-vhosts.conf` убедитесь, что этот файл подключен в главный `httpd.conf`.

Убедитесь, что `NameVirtualHost` определен и установлен в `"*: 80"`, а затем определите виртуальный хост следующими строками:

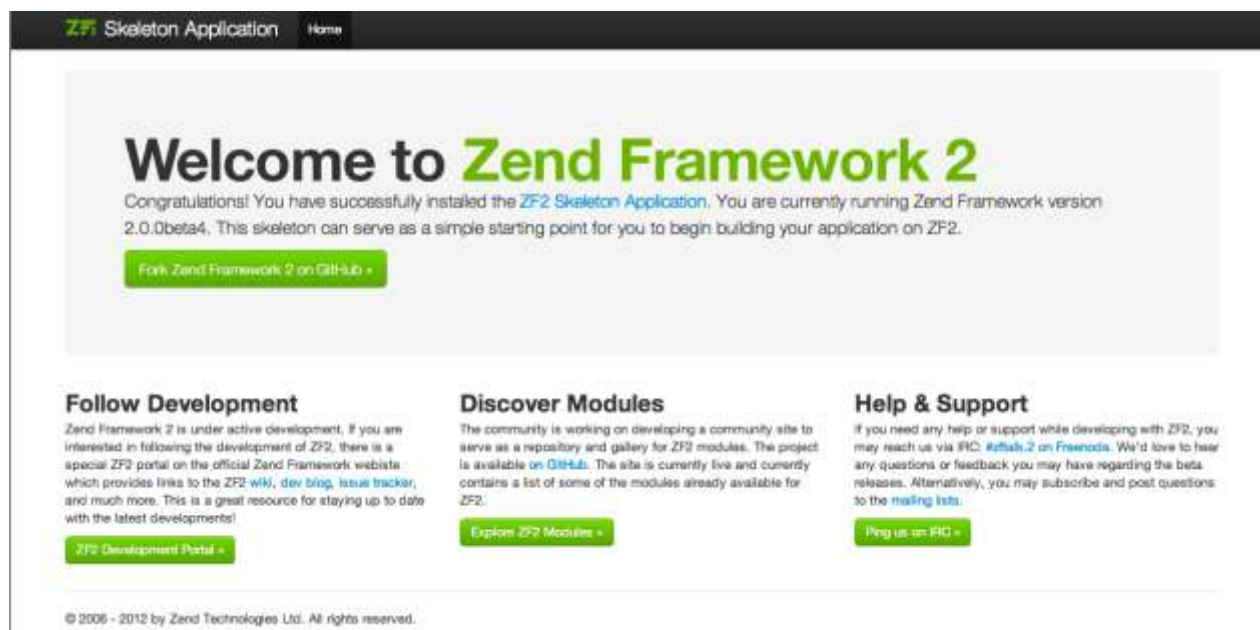
```
<VirtualHost *:80>
    ServerName hello.zend
    DocumentRoot /path/to/ZendSkeletonApplication/public
    <Directory /path/to/ZendSkeletonApplication/public>
        DirectoryIndex index.php
        AllowOverride All
        Order deny,allow
        Deny from all
        Allow from localhost
    </Directory>
</VirtualHost>
```

Проверьте, что Вы добавили запись в файл `/etc/hosts` (Unix) или `c:\windows\system32\drivers\etc\hosts` (Windows) так, чтобы при обращении к адресу `hello.zend` он ссылался на `127.0.0.1`

```
127.0.0.1    hello.zend
```

После всех манипуляций перезагрузите Apache.

Если вы все сделали правильно, то набрав в браузере `http://hello.zend`, вы увидите главную страницу `ZendSkeletonApplication`, оформленную, как приветствие `Zend Framework 2`:



Если же что-то не получилось, обратитесь к документации вашего веб-сервера, в раздел добавления виртуальных хостов, чтобы обнаружить допущенную ошибку. Вы можете также проверить логи веб-сервера, чтобы выявить характер ошибки.

Так, например, пользователи Debian/Ubuntu могут столкнуться с тем, что для Apache не был установлен по умолчанию какой-либо из модулей, например, `rewrite`, и в результате в логах появится:

```
/.htaccess: Invalid command 'RewriteEngine', perhaps misspelled or defined by a module not included in the server configuration
```

Подключите соответствующий модуль:

```
# a2enmod rewrite (или $ sudo a2enmod rewrite).
```

Структура каталогов приложений на Zend Framework 2

Теперь, наконец, мы можем взглянуть на приложение на базе Zend Framework 2 с характерным расположением каталогов, типовой конфигурацией и кодом инициализации:

```
1 ZendSkeletonApplication/
```

```
2     config/
3         application.config.php
4     autoload/
5         global.php
6         local.php
7         ...
8     module/
9     vendor/
10    public/
11        .htaccess
12        index.php
13    data/
```

При установке виртуального хоста мы указали каталог `public` в директории `ZendSkeletonApplication`, как корневой (`document root`) для веб-сервера. Это сделано для того, чтобы пользователь не мог случайно или преднамеренно вызвать какой-либо файл выше каталога `public`. Корнем же приложения (`application root`) является каталог `ZendSkeletonApplication`.

Конфигурация приложения находится в файле `application.config.php`, расположенном в каталоге `ZendSkeletonApplication/config` и содержит основную конфигурацию для `Zend\Mvc` и его компонентов в виде массива PHP. В частности, там содержатся настройки для `ModuleManager`; мы часто будем обсуждать его детали в ходе этой книги.

При необходимости, каталог `autoload` содержит дополнительные конфигурационные данные в виде дополнительных PHP файлов; поначалу это кажется несколько странным, но впоследствии вы привыкните к этому. Начнем с того, что такое обозначение каталога, как "автозагрузка" (`autoload`), немного раздражает. Здесь "автозагрузка" не имеет ничего общего с автозагрузкой классов PHP, но вместо этого указывает, что конфигурации, которые будут находиться в этом каталоге, будут автоматически учтены. И хронологически это происходит после конфигурирования с помощью `application.config.php`, а также после конфигурирования отдельных модулей, о чем мы расскажем позже. Эта последовательность считывания конфигураций крайне важна, поскольку она позволяет перезаписывать поверх значения конфигурации в зависимости от ситуации. Подобный же принцип применяется к `global.php` и `local.php`: конфигурация в `global.php` всегда действительна, но она может быть перезаписана конфигурацией `local.php`. С технической точки зрения, фреймворк сначала считывает `global.php`, а затем `local.php`, в результате чего определенные значения могут быть заменены при необходимости. Чем это хорошо? Таким образом конфигурация может быть определена независимо от среды выполнения. Предположим, что программисты, работающие с приложением, установили среду выполнения локально на своих компьютерах. Поскольку для приложения требуется база данных `MySQL`, все разработчики установили ее на своих компьютерах и в процессе настроили права доступа таким образом, что у каждого разработчика были свои учетные данные. Однако, поскольку у каждого разработчика потенциально есть индивидуальный

пароль к базе данных, эта конфигурация не может быть жестко зашита, а должна индивидуально настраиваться. Для этого разработчик вводит свои данные в файл `local.php`, который он сохраняет на локальном компьютере и не регистрирует их в общей системе управления кодом. Не смотря на то, что данные соединения для "живой системы" хранятся в файле `global.php`, каждый разработчик может работать со своими собственными параметрами подключения к данным, которые определены при помощи `local.php`. Так могут быть сохранены даже специальные конфигурации для установки и тестирования систем. Кстати, файлы конфигурации вида `xyz.local.php`, например, `db.local.php`, обрабатываются фреймворком таким же образом, как было описано выше и также перезаписывают `global.php`.

Отдельные модули приложения расположены в каталоге `module`. Каждый модуль имеет свое типичное дерево каталогов, подробнее мы рассмотрим это позже. Тем не менее, в настоящее время важно то, что у каждого модуля может быть собственная конфигурация. Теперь у нас есть три места, в которых может быть сконфигурировано что-либо: `application.config.php`, специфичная для модуля конфигурация, и файлы `global.php` и `local.php` (или их "специализация", как описано выше), которые система считывает именно в таком порядке и в конечном счете обеспечивает большой общий объект конфигурации, поскольку в ходе выполнения эти конфигурации объединяются. Если конфигурация из `application.config.php` представляет интерес только на предварительном этапе начальной загрузки, то конфигурация модулей и тех самых `global.php` и `local.php` важна в дальнейшем ходе технологической цепочки и щедро предоставляется сервис-менеджеру (`ServiceManager`). Об этом мы также узнаем впоследствии. Сейчас внимательный читатель уже понимает, что в результате такой "каскадной" конфигурации параметры модулей, которые входят в состав приложения, как модули сторонних производителей, можно расширить или даже заменить. Это очень практично.

Каталог поставщиков (`vendor`) концептуально содержит код, который вы не писали сами или который не написан специально для вашего приложения (на сегодняшний день исключение составляет код `ZendSkeletonApplication`, который вы можете написать самостоятельно в случае каких-либо сомнений). `Zend Framework 2`, таким образом, может быть расположен где-то в этом каталоге, но, при необходимости, также и в других библиотеках. Когда имеешь дело с дополнительными библиотеками, нужно всегда позаботиться о том, чтобы соответствующие классы были доступны для приложения. Однако, если можно установить соответствующие библиотеки с помощью `composer`, то разработчику не нужно также проделывать эту работу. Поэтому в идеальном случае установка дополнительных библиотек всегда должна выполняться с помощью `composer`. Интересен тот факт, что модули `ZF2`, которые на самом деле должны быть расположены в каталоге `module`, могут быть доступными из каталога `vendor`. (Чтобы быть предельно точным, нужно сказать, что это можно настроить с помощью `application.config.php`, и тогда модули смогут располагаться практически где угодно.) Это означает, что библиотеки сторонних производителей, которые поддерживают модульный стандарт `Zend Framework 2`, также могут быть добавлены подобным образом. Таким образом можно гарантировать, что в

каталоге `module` будут находиться только те модули, которые были разработаны в рамках соответствующего приложения. Все остальные модули можно сделать доступными из каталога `vendor`.

Все файлы, которые должны быть доступны извне через веб-сервер, расположены в каталоге `public` (за исключением определенных ограничений в конфигурации веб-сервера). Это также место для изображений, файлов CSS и JS, а также "единой точки входа", файла `index.php`. Идея заключается в том, что каждый HTTP запрос, который достигает веб-сервера и конкретного приложения, первоначально приводит к вызову `index.php`. Единственным исключением являются URL-адреса, ссылающиеся на фактически существующий файл в пределах или ниже каталога `public`. Только в этом случае `index.php` не выполняется, а вместо этого соответствующий файл считывается и возвращается. Эта механика достигается посредством типичного для Zend Framework файла `.htaccess` в каталоге `public`:

```
1 RewriteEngine On
2 RewriteCond %{REQUEST_FILENAME} -s [OR]
3 RewriteCond %{REQUEST_FILENAME} -l [OR]
4 RewriteCond %{REQUEST_FILENAME} -d
5 RewriteRule ^.*$ - [NC,L]
6 RewriteRule ^.*$ index.php [NC,L]
```

Чтобы это работало, должно быть выполнено несколько условий. С одной стороны, веб-сервер должен быть оснащен так называемым `RewriteEngine` (http://httpd.apache.org/docs/current/mod/mod_rewrite.html), который должен быть активирован. С другой стороны, веб-сервер должен позволять приложению устанавливать директивы с помощью собственного `.htaccess`. Для достижения этой цели следующая директива должна быть прописана в `httpd.conf` Apache:

```
1 AllowOverride All
```

Спецификация для каталога `data` относительно не конкретизирована. В основном, здесь могут храниться данные всех типов, которые не имеют ничего общего с приложением (документация, тестовые данные, и т.д.), или которые создаются во время работы приложения (кэшированные данные, сгенерированные файлы, и т.д.).

Файл `index.php`

Каждый запрос, который не сопоставлен с фактически существующим в каталоге `public` файлом, будет, таким образом, перенаправлен с помощью `.htaccess` на файл `index.php`. Поэтому он имеет особое значение для работы с Zend Framework 2. В то же время, опять-таки, важно понимать, что сам `index.php` не является частью фреймворка, но что он необходим для использования MVC компонентов фреймворка. Пожалуйста, запомните: `Zend\Mvc` — это компонент, который предоставляет "каркас обработки" для приложения.

Файл `index.php` поставляется вместе с `ZendSkeletonApplication`; таким образом, мы не программируем его самостоятельно. В виду важности `index.php` – как для фреймворка, так и для нашего понимания механики фреймворка – мы рискнем подробно рассмотреть этот очень понятный файл:

```
ZendSkeletonApplication/public/index.php
1 <?php
2 chdir(dirname(__DIR__));
3 require 'init_autoloader.php';
4 Zend\Mvc\Application::init(
5     include 'config/application.config.php'
6 )->run();
```

Listing 4.1

В самом начале (строка 2) устанавливается корневой каталог приложения в качестве текущего каталога, чтобы иметь возможность легко обращаться к другим ресурсам. Затем подключается файл `init_autoloader.php`, это первоначально активирует все, что было загружено с помощью `composer`. Этот невзрачный вызов гарантирует, что для классов всех библиотек, которые были установлены автоматически с помощью `composer`, будет доступен механизм автозагрузки:

```
ZendSkeletonApplication/init_autoloader.php
1 <?php
2 // [...]
3 if (file_exists('vendor/autoload.php')) {
4     $loader = include 'vendor/autoload.php';
5 }
6 // [...]
```

Listing 4.2

(Загружая дополнительные библиотеки кода, `composer` создает файл `vendor/autoload.php` и ряд других файлов, необходимых для механизма автозагрузки.)

Следовательно, нам не придется писать многочисленные `require()` повсюду в нашем приложении. Несколько строк, приведенные здесь без особых эмоций, на самом деле представляют собой огромное достижение для нас, как PHP-разработчиков: нет ничего проще для интеграции библиотек в свои приложения.

В `init_autoloader.php` обеспечена альтернативная загрузка классов ZF2 с помощью переменной окружения `ZF2_PATH` или с помощью подмодуля `Git` в случае, если вы не получали `Zend Framework` с помощью `composer`, так как в таком случае вышеуказанного механизма автозагрузки `composer` будет достаточно. С помощью переменной окружения `ZF2_PATH`, например, несколько различных приложений в системе могут использовать один и тот же экземпляр фреймворка. Это действительно удобно, в особенности для разработчика, так, как при разработке нескольких различных приложений на одном компьютере не нужно загружать несколько одинаковых экземпляров фреймворка для

каждого приложения. Сейчас же бегло проверим, загружен ли Zend Framework 2 – в противном случае просто ничего не произойдет – и приступим к работе:

```
ZendSkeletonApplication/public/index.php
1 <?php
2 // [...]
3 Zend\Mvc\Application::init(include 'config/application.config.php')->run();
```

Listing 4.3

Вызов метода `init()` класса `Application` прежде всего обеспечивает добавление сервис-менеджера (`ServiceManager`). `ServiceManager` – это центральный объект в Zend Framework 2. Он позволяет получить доступ к другим объектам различными способами, обычно является "основной точкой контакта" в цепи обработки и является также первой точкой входа вообще. Мы рассмотрим сервис-менеджер впоследствии более подробно. Для простоты можно представить себе для начала `ServiceManager`, как своего рода глобальный каталог, в котором некоторый объект может быть сохранен под определенным ключом (как в массиве). Для всех тех, кто уже работал с 1 версией фреймворка, `ServiceManager` изначально представляется, как своего рода `Zend_Registry`. На данный момент нам, возможно, следует сделать небольшой шаг вперед. Не только предварительно сгенерированные экземпляры объектов могут быть занесены в `ServiceManager` в качестве значений под оговоренным ключом, но и фабрики, генерирующие соответствующие объекты – в контексте сервис-менеджера обозначенные, по аналогии, как "сервисы". Основная идея этих сервисов состоит в том, что они могут быть сформированы только тогда, когда это действительно необходимо. Эта процедура называется "отложенной инициализацией", шаблон проектирования предназначен для того, чтобы как можно более отсрочить создание ресурсоемких объектов. Действительно, некоторое количество сервисов для некоторых типов запросов не понадобятся; зачем же всегда создавать их экземпляры заранее?

Но вернемся к коду: метод `init()` принимает конфигурацию приложения в качестве параметра и она учитывается при генерации `ServiceManager`:

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 // [...]
3     $serviceManager = new ServiceManager(
4         new ServiceManagerConfig(
5             $configuration['service_manager']
6         )
7     );
8 // [...]
```

Listing 4.4

Теперь в этой точке `ServiceManager` инициализирован и обеспечен теми сервисами, которые требуются в области обработки запросов `Zend\Mvc`. Однако, `ServiceManager` может также эффективно использоваться для совершенно различных целей и вне `Zend\Mvc`. Далее

конфигурация приложения сохраняется в `ServiceManager` для последующего использования.

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 // [...]
3     $serviceManager->setService('ApplicationConfig', $configuration);
4 // [...]
```

Listing 4.5

Затем у `ServiceManager` впервые запрашивают выполнить его сервисы.

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 // [...]
3     $serviceManager->get('ModuleManager')->loadModules();
4 // [...]
```

Listing 4.6

Метод `get()` запрашивает сервис. Кстати, это как раз тот случай, в котором `ServiceManager` не возвращает экземпляр объекта, а использует фабрику, чтобы создать запрашиваемый сервис. В этом случае используется фабрика `Zend\Mvc\Service\ModuleManagerFactory`, которая и генерирует запрошенный `ModuleManager`.

Но откуда `ServiceManager` в действительности знает, что всякий раз, когда запрашивается `ModuleManager`, вышеупомянутая фабрика должна быть вызвана для его генерации? Давайте вернемся к предыдущему коду:

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 // [...]
3 $serviceManager = new ServiceManager(
4     new ServiceManagerConfig($configuration['service_manager'])
5 );
6 // [...]
```

Listing 4.7

В результате передачи `ServiceManagerConfig`, `ServiceManager` подготовлен к использованию из MVC и, среди прочего, точно зарегистрировал фабрику для `ModuleManager`. В следующих главах мы еще раз рассмотрим все это более подробно, а также рассмотрим другие сервисы, которые входят в стандартную комплектацию.

Но давайте вернемся к последовательности кода: после того, как `ModuleManager` стал доступен теперь через `ServiceManager`, метод `loadModules()` инициализирует все модули, указанные в `application.config.php`. Если модули готовы, снова вступает в действие `ServiceManager` и вызывает сервис `Application`.

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
```

```

2 // [...]
3 return $serviceManager->get('Application')->bootstrap();
4 // [...]

```

Listing 4.8

Этот факт может показаться несколько странным, главным образом из-за того, что вся последовательность обработки действительно исходно начиналась с `Zend\Mvc\Application`. Но затем становится ясно, что его метод `init()` сначала только инициализирует `ServiceManager`, тогда как само приложение генерируется затем в качестве службы.

И вот теперь начинается очень сложная процедура, которая отвечает за обработку самого запроса. Приложение подготавливается – происходит начальная загрузка (bootstrapping). Возвращаясь назад в `index.php`, приложение запускается и результат возвращается в вызывающую программу.

```

ZendSkeletonApplication/public/index.php
1 <?php
2 // [...]
3     Zend\Mvc\Application::init(include 'config/application.config.php')
4         ->run();
5 // [...]

```

Listing 4.9

Я посвятил в этой книге отдельную главу точному рассмотрению процесса обработки запроса, в виду его важности и сложности. Но до тех пор, пока мы к этому подойдем, будем иметь ввиду: `index.php` – это центральная точка входа для всех запросов, обрабатываемых приложением. Эти самые запросы технически перенаправляются к `index.php` с помощью `.htaccess`. Фактический URL, вызванный пользователем, естественно, сохраняется и впоследствии читается фреймворком, чтобы найти подходящий контроллер и его действия. `ServiceManager` находится в центре обработки и дает доступ к сервисам приложения. Таким образом, на начальном этапе необходимо создать `ServiceManager`, прежде чем он, в свою очередь, даст нам доступ к `ModuleManager`, с помощью которого мы можем получить зарегистрированные модули, и к объекту приложения `Application`, который отвечает за обработку и выполнение запросов. Пока что неплохо.

Zend Framework 2 на альтернативном веб-сервере

Естественно, вместо Apache может быть использован альтернативный веб-сервер, например, nginx. В этом случае специфическая для Apache конфигурация, как та, что в .htaccess, должна быть аналогично преобразована, например, с помощью правил nginx (nginx rules).

(<http://nginx.org>)

V. Подготовка собственного модуля

Фактически, логика приложения, то есть отдельные страницы, шаблоны, формы и т.д. инкапсулируются в модулях. Теперь, когда в нашем распоряжении есть выполняемый ZendSkeletonApplication, самое время подготовить наш собственный первый модуль. Поскольку для начала мы должны сосредоточиться на отдельных шагах, которые необходимо выполнить, чтобы подготовить и инкапсулировать свой модуль, мы начнем с классического модуля, Hello, World!

Подготовка модуля "Hello World"

Модуль Zend Framework 2 характеризуется прежде всего определенной структурой каталогов и несколькими файлами, которые должны быть частью каждого модуля.

```
ZendSkeletonApplication/module/Helloworld
1 Module.php
2 config/
3     module.config.php
4 src/
5     Helloworld/
6         Controller/
7             IndexController.php
8 view/
9     helloworld/
10         index/
11             index.phtml
```

Эта структура должна быть создана в каталоге `Helloworld` в директории `module` вашего приложения. По соглашению, модуль – это свое собственное пространство имен, которое, благодаря PHP 5.3, мы можем определить, как таковое, изначально. В 1 версии фреймворка все еще должны быть использованы псевдо-пространства имен, это привело к очень длинному обозначению классов, например, `Zend_Form_Decorator_Captcha_Word`. К счастью, с PHP 5.3 и Zend Framework 2 эта проблема осталась в прошлом. Для начала мы вольем жизнь в файл `Module.php`.

```
ZendSkeletonApplication/module/Helloworld/Module.php
1 <?php
2 namespace Helloworld;
3
4 class Module
5 {
6     public function getConfig()
7     {
8         return include __DIR__ . '/config/module.config.php';
9     }
10 }
```

Listing 5.1

Классу модуля назначается пространство имен, предусмотренное нашим модулем, в данном случае `Helloworld`. Сам по себе класс является обычным классом PHP с рядом методов, которые могут быть вызваны различными менеджерами и компонентами фреймворка, например, в ходе инициализации. К ним относится и метод `getConfig()`. Как и в Zend Framework 1, подход "соглашение по конфигурации" широко используется и в новой версии. Это означает, что существует конвенция (соглашение), которая, если используется, как договаривались, делает дальнейшие настройки ненужными. В этом случае конвенцией предусмотрено следующее: если вы реализуете метод `getConfig()` в классе модуля, то он будет вызван в рамках инициализации `ModuleManager`. Сказано - сделано! Тем не менее, наш метод не возвращает напрямую конфигурацию, вместо этого он в каталоге модуля `config` считывает файл `module.config.php` следующего содержания:

```
ZendSkeletonApplication/module/Helloworld/config/module.config.php
1 <?php
2 return array(
3     'view_manager' => array(
4         'template_path_stack' => array(
5             __DIR__ . '/../view'
6         )
7     )
8 );
```

Listing 5.2

Для начала становится очевидным, что конфигурация для модуля отображается с помощью массива PHP. Существует масса вариантов того, каким образом можно поддерживать конфигурации, например, с помощью INI - файлов, с помощью YAML, или с помощью

XML - структуры. Все эти структуры требуют более или менее сложного синтаксического анализа. Однако наиболее эффективный и предпочтительный для фреймворка метод – это внести конфигурацию непосредственно в РНР код. Это делает любой синтаксический анализ ненужным, и простой `include()` уже обеспечивает желаемый эффект чтения конфигурации. Но и здесь снова применяется "соглашение по конфигурации". Если мы укажем раздел `view_manager` в нашей конфигурации, эти значения всегда будут рассмотрены впоследствии при поиске подходящего шаблона, как по волшебству. Таким образом, здесь мы указали каталог, в котором будут храниться файлы представления (View) нашего модуля (HTML - шаблоны). Соответственно, теперь здесь уже не "соглашение по конфигурации", а, скорее, точная информация.

Кроме того, мы должны определить в `Module.php`, как должна функционировать автозагрузка отдельных классов модуля. Для этого мы реализуем метод `getAutoloaderConfig()`, который будет обработан в ходе инициализации `ModuleManager` – еще раз, согласно соглашению.

```
ZendSkeletonApplication/module/Helloworld/Module.php
1 <?php
2 // [...]
3 public function getAutoloaderConfig()
4 {
5     return array(
6         'Zend\Loader\StandardAutoloader' => array(
7             'namespaces' => array(
8                 __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
9             )
10        )
11    );
12 }
13 // [...]
```

Listing 5.2

Тему "автозагрузка" впоследствии мы рассмотрим более подробно, а сейчас придется довольствоваться знанием того, что вышеописанные конструкции обеспечивают автоматическую загрузку классов этого модуля, в особенности, контроллера, и те будут учитываться фреймворком. Файл `Module.php` теперь выглядит так:

```
ZendSkeletonApplication/module/Helloworld/Module.php
1 <?php
2 namespace Helloworld;
3
4 class Module
5 {
6     public function getAutoloaderConfig()
7     {
8         return array(
9             'Zend\Loader\StandardAutoloader' => array(
```

```

10         'namespaces' => array(
11             __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
12         )
13     )
14 );
15 }
16
17 public function getConfig()
18 {
19     return include __DIR__ . '/config/module.config.php';
20 }
21 }

```

Listing 5.4

Теперь займемся контроллером — IndexController в каталоге /src/Helloworld/Controller:

```

ZendSkeletonApplication/module/Helloworld/src/Helloworld/Controller/IndexController.php
1 <?php
2 namespace Helloworld\Controller;
3
4 use Zend\Mvc\Controller\AbstractActionController;
5 use Zend\View\Model\ViewModel;
6
7 class IndexController extends AbstractActionController
8 {
9     public function indexAction()
10     {
11         return new ViewModel(array('greeting' => 'hello, world!'));
12     }
13 }

```

Listing 5.4

Для начала снова обратим внимание на пространство имен. Наш IndexController, таким образом, относится к модулю Helloworld и представляет собой контроллер (Controller). Пока все в порядке. Класс наследует от Zend\Mvc\Controller\AbstractActionController все, чем он сейчас является: класс может обработать запрос ("диспетчеризация") и в процессе использует свои действия. Термин "действия" означает публичные методы, которые снова соответствуют определенному соглашению о присвоении имен: метод класса тогда становится "действием", когда слово "действие" ("Action") добавлено к имени метода. У контроллера теперь есть действие, это indexAction.

Множество разных задач могут теперь решаться в indexAction, например, обработка параметров запроса, запись или чтение данных из баз данных, доступ к удаленным веб-сервисам, и т.д. Как правило, действие не выполняет это все самостоятельно (в противном случае, речь бы шла о так называемых "толстых контроллерах"), а делегирует отдельные

задачи другим компаньонам. Этот подход, как правило, увеличивает возможность повторного использования и удобство сопровождения кода.

Как правило, действие заканчивает свою работу путем предоставления результатов операций, выполняемых для отображения в вызывающей программе браузера. В Zend Framework 2 все необходимые данные возвращаются в виде так называемых "моделей представления" (ViewModel). Проще говоря, модели представления представляют данные, лежащие в основе пользовательского интерфейса (user interface, UI) и, кроме того, также контролируют состояние некоторых компонентов пользовательского интерфейса. Более подробно мы разберемся с этим позже.

Если мы теперь хотим отобразить приветствие "hello, world!" в браузере, то до сих пор отсутствует соответствующий тег `h1`. Поскольку в приложении на базе MVC за представление в HTML отвечает так называемое "представление" (View), то мы должны его создать (строго говоря, мы должны создать только "шаблон представления" (view template), который приведет к желаемому результату в области отрисовки представления (view rendering) и с помощью модели представления). Для каждого действия существует обычно ровно одно представление или шаблон представления соответственно. В каталоге `view/helloworld/index` создайте файл шаблона `index.phtml` со следующим содержимым:

```
ZendSkeletonApplication/module/Helloworld/view/helloworld/index/index.phtml
```

```
// приветствие (greeting)
1 <h1><?php echo $this->greeting; ?></h1>
```

Listing 5.5

Также в структурировании представления играет роль пространство имен модуля, название контроллера и действия. Наше представление расположено в каталоге `view`. Это непреложно и никогда не изменяется. Затем подкаталог с именем модуля, затем подкаталог с именем контроллера. Само представление носит имя, соответствующее действию, с расширением `.phtml`. Файл с расширением `.phtml`, согласно соглашения, может содержать PHP код и HTML разметку, где код PHP ограничивается презентацией и не должен содержать, например, бизнес-логику. Кстати, расширение "phtml" обозначает PHP + HTML. В нашем файле представления мы создаем HTML разметку, а затем получаем доступ к данным модели представления с помощью `$this`, эти данные мы создали в контроллере заранее. Для приветствия мы из контроллера в модель представления (view model) назначили ключ "greeting" со значением "hello, world!". Затем мы выводим его из шаблона с помощью `echo`.

Но это еще не все. Мы хотим увидеть на экране "hello, world!" при вызове URL `http://hello.zend/sayhello`. Для этого нам нужно добавить в конфигурацию модуля `module.config.php` соответствующий маршрут и детали нашего контроллера.

```
ZendSkeletonApplication/module/Helloworld/config/module.config.php
```

```
1 <?php
2     return array(
3         'view_manager' => array(
```

```

4         'template_path_stack' => array(
5             __DIR__ . '/../view',
6         ),
7     ),
8     'router' => array(
9         'routes' => array(
10            'sayhello' => array(
11                'type' => 'Zend\Mvc\Router\Http\Literal',
12                'options' => array(
13                    'route' => '/sayhello',
14                    'defaults' => array(
15                        'controller' => 'Helloworld\Controller\Index',
16                        'action' => 'index',
17                    )
18                )
19            )
20        )
21    ),
22    'controllers' => array(
23        'invokables' => array(
24            'Helloworld\Controller\Index'
25            => 'Helloworld\Controller\IndexController'
26        )
27    )
28 );

```

Listing 5.6

Аналогично ключу `view_manager`, ключ `router` обеспечивает — по соглашению — доступность конфигурирования маршрутов соответствующих компонентов. Так, как мы рассмотрим маршрутизацию ниже более подробно, то сейчас ограничимся следующим: на данный момент мы передаем массив с индивидуальными маршрутами, один из которых мы назвали "sayhello". Он должен вступать в силу всегда, когда строка "/sayhello" следует за информацией об узле (в нашем случае `http://hello.zend`) в URL. В этом случае фреймворк должен обеспечить выполнение `IndexController`, который мы только что подготовили, а в нем действие `index`. На этом, собственно, и все. Из-за нотации вложенных массивов конфигурация поначалу кажется немного неясной. Но пройдет немного времени, и вы быстро к ней привыкните.

Интересно отметить, что мы определили `Helloworld\Controller\Index`, как значение для контроллера, несмотря на то, что в действительности контроллер называется `IndexController`. Объяснение этому мы дадим немного позже.

```

ZendSkeletonApplication/module/Helloworld/config/module.config.php
1 <?php
2 // [...]
3 'controllers' => array(
4     'invokables' => array(
5         'Helloworld\Controller\Index'

```

```

6         => 'Helloworld\Controller\IndexController'
7     )
8 )
9 // [...]

```

Listing 5.7

Этим небольшим фрагментом кода мы определяем уникальное имя для нашего контроллера, которое будет действительно во всех модулях приложения. Чтобы гарантировать его однозначность, мы указали имя модуля и обозначение контроллера. `Helloworld\Controller\Index` теперь станет, таким образом, символическим именем для нашего контроллера, которое, соответственно, будет использовано в конфигурации маршрутов.

Наконец, что не менее важно, мы должны теперь расширить файл `application.config.php` таким образом, чтобы наш новый модуль вообще учитывался. Для этого мы внесем имя нашего модуля в раздел модулей.

```

ZendSkeletonApplication/config/appliacation.config.php
1 <?php
2 // [...]
3 'modules' => array(
4     'Application',
5     'Helloworld',
6 )
7 // [...]

```

Listing 5.8

Теперь URL `http://hello.zend/sayhello` должен обеспечить желаемый результат и вывод "hello, world!" на экран.

Автозагрузка

Как правило, сначала классы должны быть предоставлены посредством вызова `require()` (или подобным образом), прежде чем они смогут быть использованы впервые, это действительно так потому, что во время выполнения сценария доступен только тот код, который был предварительно предоставлен интерпретатору PHP. Таким образом, недостаточно запрограммировать класс PHP и сохранить его сначала где-нибудь в файловой системе, затем где-нибудь еще, чтобы обратиться к нему из выполняющегося сценария, не сделав прежде класс известным. И снова небольшое отступление: нужно безусловно отличать код, который является частью ядра PHP от так называемого "пользовательского" кода. В то время, как, например, класс ядра позволяет использовать `\DateTime` без предварительной регистрации, это не относится к классам, которые вы написали сами, т.е. пользовательскому коду. Такой код должен всегда быть доведен до сведения PHP интерпретатора заранее, и соответствующие PHP - файлы, которые содержат определения классов, должны быть загружены.

Zend Framework 2 интенсивно использует автозагрузку. Автозагрузка просто означает, что регистрация классов выполняется автоматически; соответствующие PHP - файлы с классами, которые в них определены, загружаются при необходимости. Чтобы это работало, должна быть выполнена определенная конфигурация – как мы уже видели в методе `getAutoloaderConfig()` файла `Module.php`.

В самом деле, мы избавляемся от многочисленного набора текста при наличии автозагрузки, поскольку в противном случае придется загружать каждый файл явно с помощью `require()`, но, с другой стороны, функция автозагрузки дополнительно нагружает систему. Таким образом, автозагрузка несет определенные накладные расходы, которые могут дать знать о себе во время выполнения приложения на Zend Framework 2. Хорошо, что существуют различные способы, в том числе и высокопроизводительные, с помощью которых может быть реализована автозагрузка, и которые поддерживаются фреймворком. У фреймворка есть два основных класса, которые служат для автозагрузки.

Стандартный автозагрузчик (Standard autoloader)

Стандартный автозагрузчик – реализация, которая, между тем, стала общепринятым способом понимания автозагрузки. В этом контексте имя класса транслируется непосредственно в имя файла. Следовательно, соответствующий загрузчик ожидает, например, что класс `Zend_Translate` (на Zend Framework 1), определен в файле `Translate.php` в каталоге `Zend`. Это также относится к классам, которые используют "реальные пространства имен": ожидается, что у класса `Translator`, который будет определен в пространстве имен `Zend`, будет такое же расположение в файловой системе. Это соглашение соответствует как стандартам PEAR (<http://pear.php.net>), так и PSR-0 (<https://github.com/php-fig/fig-standards>) от группы взаимодействия фреймворков на PHP (PHP Framework Interoperability Group, PHP-FIG). Важно то, где вы предполагаете расположить в файловой системе соответствующий каталог для соответствующего (псевдо-) пространства имен, как мы делали в файле `Module.php`.

```
ZendSkeletonApplication/module/Helloworld/Module.php
1 <?php
2 namespace Helloworld;
3
4 class Module
5 {
6     public function getAutoloaderConfig()
7     {
8         return array(
9             'Zend\Loader\StandardAutoloader' => array(
10                 'namespaces' => array(
11                     __NAMESPACE__
12                     => __DIR__ . '/src/' . __NAMESPACE__,
13                 ),
14             ),
15         );
16     }
```

```

17
18 // [...]
19 }

```

Listing 5.9

То есть, `include_path` больше не принимается во внимание, что должно ускорить загрузку классов в максимально возможной степени. На данный момент это все, что вам нужно знать. Если соответствовать этим соглашениям, классы будут автоматически загружаться без проблем.

Автозагрузчик карты классов (ClassMapAutoloader)

Тем не менее, реализацией самой производительной автозагрузки является `ClassMapAutoloader`; он действует на основе простого ассоциативного массива PHP, который содержит полные имена классов в качестве ключа для каждого случая и соответствующие имена файлов в качестве значений. Выглядит это примерно так:

```

ZendSkeletonApplication/module/Mymodule/autoload_classmap.php
1 <?php
2 return array(
3     'PhlyContact\Service>ContactControllerFactory'
4     => __DIR__ . '/src/PhlyContact/' .
5         'Service/ContactControllerFactory.php',
6 );

```

Listing 5.10

Если запрошен соответствующий класс, загрузчик ищет в массиве соответствующее значение и загружает файл. Вот и все. В этом случае недостаток очевиден: карта классов должна постоянно поддерживаться. Если некоторый класс там отсутствует, автозагрузка не удастся. К счастью, с одной стороны есть возможность сгенерировать карту классов автоматически (например, при сборке проекта), чтобы гарантировать, что вы не забудете какой-либо класс, а с другой стороны – можно объединить несколько методов автозагрузки.

```

ZendSkeletonApplication/module/Helloworld/Module.php
1 <?php
2 public function getAutoloaderConfig()
3 {
4     return array(
5         'Zend\Loader\ClassMapAutoloader' => array(
6             __DIR__ . '/autoload_classmap.php'
7         ),
8         'Zend\Loader\StandardAutoloader' => array(
9             'namespaces' => array(
10                __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
11            ),
12        ),
13    );

```

Listing 5.11

Таким образом, первоначально будет опрошена карта классов (ClassMap), а при необходимости будет использоваться механизм PSR-0, если никаких совпадений не найдено до этого момента.

Сама карта классов находится в собственном файле, вне `Module.php`, и обращаются к ней только оттуда. Основная идея состоит в том, чтобы обобщить процедуру для реализации автозагрузки до такой степени, чтобы даже за пределами Zend Framework библиотеки из различных источников могли быть интегрированы без проблем. В последние несколько лет некоторые библиотеки кода уже поняли, что это реальная дополнительная ценность для пользователя, если библиотека обеспечивает некоторые формы поддержки автоматической загрузки. Проблема заключается в том, что в каждом конкретном случае это снова вопрос изолированных решений. При конкретной реализации автозагрузки в случае сомнений каждая библиотека идет своим собственным путем, и даже, если способы похожи, они действительно отличаются в деталях. В рамках модулей приложения на ZF2 фреймворк ориентируется на дополнительные стандарты, из которых, к примеру, полезен вышеупомянутый `composer`, и, таким образом, позволяет выполнить автозагрузку "транслибиотек" самым простым возможным способом. Для этого каждый модуль поставляется с тремя дополнительными файлами, относящимися к автозагрузке: `autoload_classmap.php`, `autoload_function.php` и `autoload_RegisterNow.php`.

`autoload_classmap.php` возвращает массив PHP, отображающий имена классов к именам файлов, как описано выше, чтобы любой произвольный автозагрузчик, а не только тот, что в Zend Framework – но и, например, в `composer`, мог обрабатывать его и, при необходимости, комбинировать с картами классов других библиотек. В противоположность этому, `autoload_function.php` возвращает PHP - функцию:

ZendSkeletonApplication/module/Helloworld/autoload_function.php

```
1 <?php
2 return function ($class) {
3     static $classmap = null;
4     if ($classmap === null) {
5         $classmap = include __DIR__ . '/autoload_classmap.php';
6     }
7     if (!isset($classmap[$class])) {
8         return false;
9     }
10    return include_once $classmap[$class];
11 };
```

Listing 5.12

Возвращенная функция также может быть обработана автозагрузчиком, например, в виде дополнительного источника для автозагрузки классов. В конечном счете, эта функция

также снова получает доступ к карте классов. А `autoload_Register now.php` еще более тонкий:

```
1 <?php
2 spl_autoload_Register now(include __DIR__ . '/autoload_function.php');
```

Listing 5.13

В этом случае ранее определенная функция автозагрузки (в файле `autoload_function.php`), которая снова получает доступ к надлежащей карте классов (в файле `autoload_classmap.php`), непосредственно зарегистрирована для выполнения автозагрузки и ничего не возвращает в вызывающую программу. Тогда простое выполнение кода:

```
1 <?php
2 require_once 'autoload_Register now.php';
```

Listing 5.14

обеспечивает автозагрузку именно этих функциональных компонентов. Это, однако, без альтернативы осуществления дальнейших оптимизаций, например, в последовательности обработки всех зарегистрированных функций автозагрузки.

Три файла автозагрузки, которые мы рассмотрели, не обязательны для функционирования модуля, потому что все эти файлы относятся к автозагрузке карты классов, которая не является абсолютно необходимой, но может заметно улучшить производительность приложения.

В заключение, полная структура каталогов модуля "Hello world", в комплекте с файлами автозагрузки, будет выглядеть следующим образом:

```
ZendSkeletonApplication/Helloworld
1 Module.php
2 autoload_classmap.php
3 autoload_function.php
4 autoload_Register now.php
5 config/
6     module.config.php
7 src/
8     Helloworld/
9         Controller/
10             IndexController.php
11 views/
12     helloworld/
13         index/
14             index.phtml
```

VI. За запросом и обратно

Давайте подробно рассмотрим, что же происходит с запросом и как различные компоненты фреймворка формируют ответ с нашим "hello, world!" в браузере.

Все начинается с вызова URL `http://www.hello.zend/sayhello`. Запрос достигает веб-сервера, который, с учетом `.htaccess`, решает, что `index.php` должен быть обработан интерпретатором PHP. В рамках `index.php` происходит начальная конфигурация автозагрузки, и в дальнейшем вызывается метод инициализации приложения `Application::init()`.

```
ZendSkeletonApplication/public/index.php
1 <?php
2 Zend\Mvc\Application::init(
3     include 'config/application.config.php')
4     ->run();
```

Listing 6.1

Сервис-менеджер (ServiceManager)

Экземпляр сервис-менеджера создается здесь:

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 // [...]
3 $serviceManager = new ServiceManager(
4     new ServiceManagerConfig($configuration['service_manager'])
5 );
6 // [...]
```

Listing 6.2

В результате передачи `ServiceManagerConfig`, `ServiceManager` оснащается несколькими стандартными сервисами, которые требуются для нормального функционирования `Zend\Mvc`. Кроме того, соответствующая конфигурация передается в `ServiceManagerConfig` из ранее загруженного `application.config.php`.

`ServiceManager` – это своего рода `Zend_Registry` с расширенной функциональностью. В то время, как `Zend_Registry` 1 версии фреймворка мог просто зарегистрировать существующие объекты под определенным ключом и впоследствии загрузить их снова (храня значение ключа), `ServiceManager` идет на несколько шагов дальше.

Сервисы и регистрация сервиса

Помимо администрирования сервисов в виде объектов, "генератор" может быть зарегистрирован для ключа, который первоначально создает соответствующий сервис

только при необходимости. Для этого укажите либо классы (полностью, т.е., если необходимо, с декларацией пространства имен) в виде

```
1 <?php
2 $serviceManager->setInvokableClass(
3     'MyService',
4     'Helloworld\Service\MyService'
5 );
```

Listing 6.3

, либо фабрики.

```
1 <?php
2 $serviceManager->setFactory(
3     'MyServiceFactory',
4     'Helloworld\Service\MyServiceFactory'
5 );
```

Listing 6.4

Для того, чтобы `ServiceManager` мог управлять `MyServiceFactory`, последняя должна реализовать `FactoryInterface`, этого требует метод `createService`. Затем этот метод вызывается в `ServiceManager`. Как фактор, облегчающий реализацию, может быть непосредственно передана функция обратного вызова.

```
1 <?php
2 $serviceManager->setFactory(
3     'MyServiceFactory',
4     function($serviceManager) {
5         // [...]
6     }
7 );
```

Listing 6.5

В качестве альтернативы, можно аналогично зарегистрировать абстрактную фабрику, где она добавляется без идентификатора, вот пример с функцией обратного вызова:

```
1 <?php
2 $serviceManager->addAbstractFactory(
3     function($serviceManager) {
4         // [...]
5     }
6 );
```

Listing 6.6

Кроме того, могут быть зарегистрированы так называемые "инициализаторы"; они гарантируют, что сервис, когда он вызывается, будет обеспечен значениями или ссылками на другие объекты. "Инициализаторы" могут рассматриваться, как инъекция объектов в сервис, если соответствующий сервис реализует определенный интерфейс. Мы увидим это в действии позже. На данный момент мы только должны помнить, что они существуют.

Все сервисы, которые предоставляет ServiceManager, являются так называемыми "разделяемыми сервисами", это означает, что именно этот экземпляр сервиса – создаваемый при необходимости, или уже существующий – будет возвращен при повторном запросе этого сервиса. Экземпляр сервиса, таким образом, используется повторно. Это справедливо для всех стандартных сервисов, определенных фреймворком, единственным исключением в этом контексте является EventManager. При регистрации нового сервиса можно предотвратить повторное использование экземпляров (передав в качестве соответствующего параметра false).

```
1 <?php
2 $serviceManager->setInvokableClass(
3     'myService',
4     'Helloworld\Service\MyService',
5     false
6 );
```

Listing 6.7

Стандартные сервисы, часть 1

В самом начале обработки запроса ServiceManagerConfig обеспечивает доступность ряда стандартных сервисов. Важно понимать, что существуют сервисы в ServiceManager, которые могут частично использоваться только фреймворком (или, точнее, его реализацией MVC), тогда как некоторые другие сервисы также полезны для разработчиков приложений, например, в контексте контроллера (Controller). Это станет понятнее чуть позже.

Вызываемые сервисы (Invokables)

Данные сервисы доступны в качестве "invokable", т.е. определяют класс, экземпляр которого может быть создан при необходимости.

- `SharedEventManager` (`Zend\EventManager\SharedEventManager`) : позволяет регистрацию слушателей для определенных событий, даже если менеджер событий, необходимый для этого, еще не доступен. `SharedEventManager` автоматически становится доступным для нового `EventManager`, если он сгенерирован `ServiceManager`. Подробные объяснения `SharedEventManager` будут даны позже в этой книге.

Фабрики (Factories)

В контексте сервис-менеджера, фабрики делают доступными сервисы, которые не существуют, пока не произойдет реальный запрос, но, скорее, строятся фабриками "по требованию". Следующие сервисы косвенно доступны для фабрик в стандартной комплектации:

- `SharedEventManager` (`Zend\EventManager\SharedEventManager`) : `EventManager` может генерировать события и сообщать о них зарегистрированным слушателям. Он

может также быть запрошен с помощью Zend\EventManager\EventManagerInterfacealias.

- `ModuleManager` (`Zend\Mvc\Service\ModuleManagerFactory`): администрирует модули приложения на Zend Framework 2.

Конфигурация

Таким образом, `ServiceManager` окончательно управляется в процессе обработки запроса двумя конфигурациями: конфигурацией `ServiceManagerConfig`, которая задает ряд стандартных сервисов для обработки запроса, а также конфигурацией `application.config.php` или конфигурацией конкретного модуля, соответственно. В каждом случае для этого необходим ключ `service_manager`:

```
1 <?php
2 return array(
3     // [...]
4     'service_manager' => array(
5         // [...]
6     ),
7     // [...]
8 );
```

Listing 6.8

Для ключа `service_manager` возможны следующие вложенные ключи:

- `services` (службы): определение служб при помощи объектов, чьи экземпляры уже созданы.
- `invocables` (вызываемые): определение служб объявлением класса, экземпляр которого будет создан при необходимости.
- `factories` (фабрики): определение фабрик, которые строят экземпляр класса.
- `abstract_factories` (абстрактные фабрики): определение абстрактных фабрик.
- `aliases` (псевдонимы): определение псевдонимов.
- `shared` (разделяемые): позволяет точно указать, может ли определенный сервис использоваться несколько раз, или должен быть создан новый экземпляр, если он снова потребуется.

Как только становится доступным `ServiceManager`, конфигурация, включающая `application.config.php` в целом, также с другими соответствующими разделами, не относящимися к `ServiceManager`, сама становится доступной в качестве службы.

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 // [...]
3 $serviceManager->setService('ApplicationConfig', $configuration);
```

Listing 6.9

Это важно потому, что другие компоненты, такие, как `ModuleManager` или `ViewManager`, также должны получить доступ к этим сервисам.

На данный момент `ServiceManager`, уже оснащенный разнообразными стандартными сервисами, находится в состоянии готовности, и, образно говоря, только и ждет начала шоу. В самом деле, до сих пор ничего не произошло, за исключением нескольких основных приготовлений. Фактически, в этой точке почти все упомянутые службы все еще не существуют, потому, что они еще не были затребованы, и будут созданы при необходимости.

Создание собственного сервиса

Давайте создадим собственный сервис для нашего модуля "Hello world" в качестве образца. Для этого мы сначала добавим новый подкаталог `Service` в директории `src/Helloworld` нашего модуля.

```
1 Module.php
2 config/
3     module.config.php
4 src/
5     Helloworld/
6         Controller/
7             IndexController.php
8         Service/
9             GreetingService.php
10 view/
11     helloworld/
12         index/
13             index.phtml
```

Здесь мы создадим класс `GreetingService` (сервис приветствия). Этот класс не будет реализовывать специальные интерфейсы, или вызываться из какого-либо основного класса; это будет так называемый "Plain Old PHP Object", POPO (POPO – "простой PHP объект в старом стиле" - простой PHP - объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели). Важно лишь то, что мы не забудем сделать класс доступным в правильном пространстве имен.

```
ZendSkeletonApplication/module/Helloworld/src/Helloworld/Service/GreetingService.php
1 <?php
2 namespace Helloworld\Service;
3
4 class GreetingService
5 {
6     public function getGreeting() // получить приветствие
7     {
```

```

8         if(date("H") <= 11) {
9             return "Good morning, world!"; // доброе утро, мир
10        } else if (date("H") > 11 && date("H") < 17) {
11            return "Hello, world!";          // привет, мир
12        } else {
13            return "Good evening, world!"; // добрый вечер, мир
14        }
15    }
16 }

```

Listing 6.10

Предоставление доступа к сервису, как invocable

Чтобы использовать этот класс в качестве сервиса в нашем контроллере и отображать ориентированные на время приветствия, мы должны добавить класс в ServiceManager, как сервис. Мы можем сделать это в рамках нашего модуля двумя путями: в ходе конфигурации модуля (module.config.php) путем добавления соответствующего раздела

```

ZendSkeletonApplication/module/Helloworld/config/module.config.php
1 <?php
2 // [...]
3 'service_manager' => array(
4     'invokables' => array(
5         'greetingService' => 'Helloworld\Service\GreetingService'
6     )
7 )
8 // [...]

```

Listing 6.11

или же программным способом, добавив метод getServiceConfig() к классу Helloworld\Module в файле Module.php:

```

ZendSkeletonApplication/module/Helloworld/Module.php
1 <?php
2 // [...]
3 public function getServiceConfig()
4 {
5     return array(
6         'invokables' => array(
7             'greetingService'
8                 => 'Helloworld\Service\GreetingService'
9         )
10    );
11 }
12 // [...]

```

Listing 6.12

Оба способа действенны. Наш сервис доступен теперь, как "вызываемый" (invokable). Теперь в IndexController нашего "Hello World" мы можем запросить сервис:

ZendSkeletonApplication/module/Helloworld/src/Helloworld/Controller/IndexController.php

```
1 <?php
2 // [...]
3 public function indexAction()
4 {
5     $greetingSrv = $this->getServiceLocator()
6         ->get('greetingService');
7
8     return new ViewModel(
9         array('greeting' => $greetingSrv->getGreeting())
10    );
11 }
```

Listing 6.13

Предоставление доступа к контроллеру с помощью класса фабрики

Тем не менее, у нас теперь новая проблема: контроллер зависит от сервиса (и от `ServiceManager`), к которым он активно обращается. Правда, он хотя бы не создает экземпляры класса самостоятельно, что уже хорошо, поскольку дает возможность при необходимости предоставить альтернативную реализацию сервиса в `ServiceManager` и при этом активно гарантирует, что все зависимости будут решены. Проблемы у нас появятся позднее, когда мы захотим выполнить юнит-тестирование. Возможной альтернативой в данном случае могли бы стать "внедрение зависимостей" или "инверсия управления". В этом контексте необходимые компоненты становятся доступны автоматически и больше не должны активно требоваться. В рамках `ServiceManager` мы можем реализовать эту процедуру, например, с помощью фабрики.

Для этого мы создаем фабрику `IndexControllerFactory` в том же каталоге, в котором был сохранен `IndexController`:

ZendSkeletonApplication/module/Helloworld/src/Helloworld/Controller/IndexControllerFactory.php

```
1 <?php
2 namespace Helloworld\Controller;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class IndexControllerFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $ctr = new IndexController();
12
13        $ctr->setGreetingService( // внедряется зависимость, Setter Injection
14            $serviceLocator->getServiceLocator()
15                ->get('greetingService')
16        );
17    }
18 }
```

```

17
18     return $ctr;
19 }
20 }

```

Listing 6.14

Кроме того, нужно изменить `module.config.php` в разделе `controllers` следующим образом:

```

ZendSkeletonApplication/module/Helloworld/config/module.config.php
1 <?php
2 // [...]
3 'controllers' => array(
4     'factories' => array(
5         'Helloworld\Controller\Index'
6         => 'Helloworld\Controller\IndexControllerFactory'
7     )
8 )
9 // [...]

```

Listing 6.15

Внедрение зависимостей (DI) имеет следующие техники реализации:

- Через метод класса – Setter Injection
- Через конструктор – Constructor Injection
- Через интерфейс внедрения – Interface Injection

С этого момента наш `IndexController` больше не генерируется созданием экземпляра точно определенного класса, но вносится фабрикой, которая предварительно была сформирована сервисами контроллеров и в данном случае внедряет зависимость в контроллер через метод класса (Setter Injection). Мы все еще должны добавить соответствующий метод внедрения ("Setter") в `IndexController` и в ходе самого действия (Action) обеспечить доступ к соответствующей переменной-члену класса вместо обращения к `ServiceLocator`.

```

ZendSkeletonApplication/module/Helloworld/src/Helloworld/Controller/IndexController.php
1 <?php
2
3 namespace Helloworld\Controller;
4
5 use Zend\Mvc\Controller\AbstractActionController;
6 use Zend\View\Model\ViewModel;
7
8 class IndexController extends AbstractActionController
9 {
10     private $greetingService;
11
12     public function indexAction()
13     {

```

```

14         return new ViewModel(
15             array(
16                 'greeting' => $this->greetingService->getGreeting()
17             )
18         );
19     }
20     // метод ("Setter") для внедрения зависимостей
21     public function setGreetingService($service)
22     {
23         $this->greetingService = $service;
24     }
25 }

```

Listing 6.16

Таким образом, фабрика классов может быть использована как для создания сервиса, так и для создания контроллера. Действительно, все именно так: `Zend\ServiceManager` используется в ZF2 несколькими способами. Один выглядит так, как мы уже обсуждали: как центральный экземпляр, с помощью которого генерируется даже само приложение, т.е., можно сказать, "контейнер" для всего приложения. А кроме того, есть, как мы скоро расскажем более подробно, ряд специализированных "ServiceManagers", например, только для контроллеров приложения. В этом контексте следующие строки в `IndexControllerFactory` представляют особый интерес:

```

ZendSkeletonApplication/module/Helloworld/src/Helloworld/Controller/IndexControllerFactory.php
1 <?php
2 public function createService(ServiceLocatorInterface $serviceLocator)
3 {
4     // [...]
5     $ctr->setGreetingService(
6         $serviceLocator->getServiceLocator()
7         ->get('greetingService')
8     );
9     // [...]
10 }
11 // [...]

```

Listing 6.17

Очевидно, что `getServiceLocator()` первоначально вызывается из `$serviceLocator`. Причина в том, что в метод фабрик `createService()` всегда передается тот "ServiceManager", который отвечает за создание этого сервиса, соответственно, в данном случае это – `ControllerLoader` (вызываемый фреймворком автоматически), который зарезервирован для создания контроллеров. Однако, у него, в свою очередь, нет никакого доступа к `GreetingService`, подготовленному нами заранее, и доступному только из "центрального ServiceManager" (действительно, `GreetingService` – это, в конечном счете, не контроллер). Для того, чтобы, несмотря на это, сделать сервисы центрального `ServiceManager` доступными, `ControllerLoader` получает доступ к центральному

`ServiceManager` с помощью `getServiceLocator()`. Несколько позже в этой главе вы узнаете больше о деталях этого механизма.

Предоставление доступа к контроллеру с помощью обратного вызова фабрики

Однако, с `IndexControllerFactory` у нас теперь есть дополнительный класс в нашем исходном коде. В настоящее время это, в принципе, не проблема, но это может быть не очень хорошо в случае, подобном этому, где создание фабрики не является такой уж сложной задачей. Простой альтернативой, которая также дает нам возможность избежать прямых зависимостей, является использование функции обратного вызова для фабрики в `module.config.php`:

`ZendSkeletonApplication/module/Helloworld/config/module.config.php`

```
1 <?php
2 // [...]
3 'controllers' => array(
4     'factories' => array(
5         'Helloworld\Controller\Index' => function($serviceLocator) {
6             $ctr = new Helloworld\Controller\IndexController();
7
8             $ctr->setGreetingService(
9                 $serviceLocator->getServiceLocator()
10                    ->get('greetingService')
11            );
12
13            return $ctr;
14        }
15    )
16 )
17 // [...]
```

Listing 6.18

Код для создания `IndexController`, который ранее был расположен в `IndexControllerFactory`, в настоящее время перемещается непосредственно в `module.config.php`.

Предоставление доступа к сервису с помощью `Zend\Di`

Независимо от того, какая форма фабрики используется, во всех них создание соответствующего объекта происходит программно, это означает, что должен быть написан соответствующий PHP код. `Zend\Di` предоставляет альтернативный вариант, с помощью которого мы можем создать единые графы объектов с помощью конфигурационных файлов. Подробности мы рассмотрим позже.

Менеджер модулей (ModuleManager)

Но теперь давайте вернемся к обработке запроса: после того, как `ServiceManager` был соответствующе подготовлен, он был использован впервые для создания `ModuleManager` посредством зарегистрированных фабрик, перед тем, как инициализируется загрузка модулей.

```
Zend/Mvc/Application.php, Application::init($configuration = array())
1 <?php
2 $serviceManager->get('ModuleManager')->loadModules();
```

Listing 6.19

Формирование ModuleManager

`ModuleManagerFactory` служит для предоставления сервиса `ModuleManager`. Как мы помним, фабрика используется тогда, когда создание объекта становится более сложным. В рамках создания `ModuleManager`, первоначально у `ServiceManager` запрашивается новый `EventManager` и передается в `ModuleManager`. Следовательно, `ModuleManager` способен генерировать события и информировать заранее зарегистрированных "слушателей" (`Listeners`). Следующие события инициируются менеджером модулей (в более поздний момент времени):

- `loadModules`: инициируется при загрузке модулей.
- `loadModule.resolve`: инициируется для каждого модуля, который должен быть загружен, когда считываются необходимые данные.
- `loadModule`: инициируется при загрузке модуля для каждого модуля, когда уже прочитанные данные экспортируются.
- `loadModules.post`: инициируется после загрузки всех модулей.

Слушатели (Listeners), ориентированные на модули

Тем не менее, `ModuleManagerFactory` делает гораздо больше. Начнем с того, что она генерирует большое количество слушателей, зарегистрированных для вышеупомянутых событий.

- `ModuleAutoloader`: обеспечивает загрузку класса `Module` отдельных модулей.
- `ModuleResolverListener`: инициализирует `Module.php` соответствующего модуля.
- `AutoloaderListener`: вызывает метод `getAutoloaderConfig()` в модулях для того, чтобы получить информацию о том, как классы модулей могут быть загружены.
- `OnBootstrapListener`: проверяет, содержат ли модули метод `onBootstrap()` и регистрируют вызов этого метода для события начальной загрузки, которое будет инициировано приложением в более поздний момент времени.
- `InitTrigger`: проверяет, содержат ли модули метод `init()`. Если содержат, то он вызывается.

- **ConfigListener**: проверяет, содержат ли модули метод `getConfig()`, и если он присутствует, то вызывается, и возвращаемый массив конфигураций модуля объединяется с другими конфигурациями.
- **LocatorRegistrationListener**: обеспечивает, чтобы экземпляры всех классов модуля, которые реализуют интерфейс `ServiceLocatorRegister` `nowedInterface`, были введены в `ServiceManager`.
- **ServiceListener**: вызывает методы `getServiceConfig()`, `getControllerConfig()`, `getControllerPluginConfig()`, `getViewHelperConfig()` класса модуля, если они присутствуют (или считывает соответствующие конфигурации; подробнее об этом в дальнейшем), обрабатывает объединенную конфигурацию всех модулей, применяет ее к `ServiceManager`, и далее добавляет в последний стандартные сервисы.

Стандартные сервисы, часть 2

После того, как ряд стандартных сервисов предоставляются в ходе создания `ServiceManager` – среди них, помимо `EventManager`, даже сам `ModuleManager` – `ServiceListener` регистрирует (по запросу своей фабрики) ряд дополнительных сервисов, которые требуются в ходе обработки запроса. Сейчас уместен краткий комментарий: на данный момент режим работы каждого сервиса не может и не должен вами пониматься! Многие сервисы, которые зарегистрированы на данный момент менеджером модулей будут рассмотрены снова в ходе этой главы или книги. Следующий список, таким образом, должен в большей степени служить справочником и перспективой того, что все еще впереди. Следующие сервисы, таким образом, – перечисленные здесь согласно способу регистрации – регистрируются.

Вызываемые (Invokables)

- **RouteListener** (`Zend\Mvc\RouteListener`): слушатель ожидает результат MVC `onRoute` и затем обеспечивает маршрутизатор (`router`) разрешением на соответствующий контроллер.
- **DispatchListener** (`Zend\Mvc\DispatchListener`): слушатель ожидает результат MVC `onRoute` и затем обеспечивает, чтобы `ControllerLoader` загрузил ранее выбранный контроллер и запустил его.

Фабрики (Factories)

- **Application** (`Zend\Mvc\Service\ApplicationFactory`): приложение (созданное депонированной фабрикой) представляет собой, так сказать, полную технологическую цепочку и в целом все приложение.
- **Configuration** (`Zend\Mvc\Service\ConfigFactory`): созданный сервис `Config` возвращает объединенную конфигурацию приложения. Он также доступен через псевдоним `Config`.
- **ConsoleAdapter** (`Zend\Mvc\Service\ConsoleAdapterFactory`): сервис для доступа к командной строке.

- `DependencyInjector` (`Zend\Mvc\Service\DiFactory`): у Zend Framework есть своя собственная реализация так называемого "внедрения зависимостей", с помощью которой автоматически "объединяются" сложные графы объектов на основе комплексной конфигурации. Вместо ключей `DependencyInjector` можно использовать псевдонимы `Di` или `Zend\Di\LocatorInterface`. Мы рассмотрим более подробно `Zend\Di` позже.
- `Router`, `HttpRouter`, `ConsoleRouter` (`Zend\Mvc\Service\RouterFactory`): основанный на запрошенном URL, сгенерированный фабриками сервис `Router` определяет контроллер, который должен быть вызван – также, при необходимости, в режиме командной строки.
- `Request` (`Zend\Mvc\PhpEnvironment\Request`): обеспечивает доступ ко всей информации о запросе, например, к параметрам запроса.
- `Response` (`Zend\Http\PhpEnvironment\Response`): предоставляет ответ, сгенерированный в процессе обработки, клиенту.
- `ViewManager`: `ViewManager` выполняет функцию администрирования представлений (views) и их обработки, которая подобна выполняемой `ModuleManager` для модулей и `ServiceManager` для служб. Это гарантирует, что данные когда-нибудь станут, например, веб-страницами с разметкой HTML.
- `ViewJsonRenderer` (`Zend\Mvc\Service\ViewJsonRendererFactory`): позволяет реализовать RESTful (<http://ru.wikipedia.org/wiki/REST>) контроллеры и, следовательно, веб-сервисы, которые соответствуют стилю архитектуры REST. Эта тема обсуждается в отдельной главе книги.
- `ViewJsonStrategy` (`Zend\Mvc\Service\ViewJsonStrategyFactory`): обеспечивает вызов `ViewJsonRenderer` при необходимости. В рамках этой стратегии, например, система проверяет, принадлежит ли возвращаемая контроллером модель представления (`ViewModel`) типу `JsonModel`.
- `ViewFeedRenderer` (`Zend\Mvc\Service\ViewFeedRendererFactory`): позволяет реализациям RSS или Atom каналов возвращать из контроллера данные представления (view data).
- `ViewFeedStrategy` (`Zend\Mvc\Service\ViewJsonStrategyFactory`): гарантирует, что `ViewFeedRenderer` будет вызван при необходимости. Одной из задач этой стратегии является определение того, принадлежит ли возвращаемая контроллером `ViewModel` типу `FeedModel`.
- `ViewResolver` (`Zend\Mvc\Service\ViewResolverFactory`): позволяет найти шаблоны представлений (view templates).
- `ViewTemplateMapResolver` (`Zend\Mvc\Service\ViewResolverFactory`): позволяет `ViewResolver` найти шаблоны представлений на основе карт.
- `ViewTemplatePathStack` (`Zend\Mvc\Service\ViewTemplatePathStackFactory`): позволяет `ViewResolver` найти шаблоны представлений на основе списка путей.

И кроме того:

- `ControllerLoader` (`Zend\Mvc\Service\ControllerLoaderFactory`) : `ControllerLoader` может загрузить контроллер, который был предварительно локализован с помощью маршрутизации.
- `ControllerPluginManager` (`Zend\Mvc\Service\ControllerPluginManagerFactory`) : создает `ControllerPluginManager`, благодаря чему доступен ряд плагинов, которые могут быть использованы в контроллерах; среди них, например, плагин `redirect`, с помощью которого может быть реализована переадресация. Этот сервис может быть также запрошен с помощью ключей `ControllerPluginBroker`, `Zend\Mvc\Controller\PluginBroker` или `Zend\Mvc\Controller\PluginManager`.
- `ViewHelperManager` (`Zend\Mvc\Service\ViewHelperManagerFactory`) : создает `ViewHelperManager`, который отвечает за управление так называемыми "помощниками представления" (`view helpers`).

Последние три сервиса представляют особый интерес, поскольку они, в свою очередь, представляют собой новые "ServiceManager", называемые "сферой сервис-менеджеров" на жаргоне Zend Framework. Ох, теперь вещи становятся немного сложнее. Давайте посмотрим на вещи не спеша – шаг за шагом. Для начала вы должны помнить, что в системе есть один "центральный" `ServiceManager`. Все важные "сервисы приложения" генерируются, используя его. Это как `ServiceManager` в техническом смысле, так и "центральный сервис-менеджер" для нас. Тем не менее, есть определенные сервисы, которые `ServiceManager` не предоставляет, но вместо этого доступны специализированные "суб-сервис-менеджеры" или "сфера сервис-менеджеров", которые, соответственно, также могут предоставлять сервисы с помощью известных механизмов, т.е. "invocables", "factories", и др. В техническом смысле все они – также сервис-менеджеры.

В этом контексте давайте еще раз взглянем на последнюю главу, в которой мы написали наш собственный контроллер. Там мы находим следующее место из `module.config.php`:

`ZendSkeletonApplication/module/Helloworld/config/module.config.php`

```
1 <?php
2 // [...]
3 'controllers' => array(
4     'invokables' => array(
5         'Helloworld\Controller\Index'
6         => 'Helloworld\Controller\IndexController'
7     )
8 )
9 // [...]
```

Listing 6.20

Когда этот фрагмент конфигурации интерпретируется, это приводит к ссылке на соответствующий класс контроллера `Helloworld\Controller\Index` с ключом, который зарегистрирован, как "invokable" в `ControllerLoader`, одном из стандартных из "сферы сервис-менеджеров". Таким образом, если впоследствии контроллер будет определен, как подходящий в области маршрутизации, и необходимо будет создать экземпляр его класса,

система использует для этого `ControllerLoader`. В этом контексте можно охарактеризовать контроллер, как сервис.

Эта процедура специализированных суб-сервис-менеджеров имеет ряд определенных преимуществ для определенных видов сервисов. Например, таким образом центральный сервис-менеджер для сервисов приложений не перегружен бесчисленными сервисами, и легко определить все контроллеры, задача, которая в противном случае не была бы столь простой. Здесь снова коротко представлены различные сервис-менеджеры:

- `Application Services` (`Zend\ServiceManager\ServiceManager`) – сервисы приложения: конфигурация с помощью ключа `service_manager` или метода `getServiceConfig()` (определен в `ServiceProviderInterface`).
- `Controllers` (`Zend\Mvc\Controller\ControllerManager`) – контроллеры: конфигурация с помощью ключа `controllers` или метода `getControllerConfig()` (определен в `ControllerProviderInterface`).
- `Controller plugins` (`Zend\Mvc\Controller\PluginManager`) – плагины контроллеров: конфигурация с помощью ключа `controller_plugins` или метода `getControllerPluginConfig()` (определен в `ControllerPluginProviderInterface`). Они могут быть получены из центрального сервис-менеджера через обозначение сервиса `ControllerPluginManager`.
- `"View helpers"` (`Zend\View\HelperPluginManager`) – помощники представления: конфигурация с помощью ключа `view_helpers` или метода `getViewHelperConfig()` (определен в `ViewHelperProviderInterface`). Они могут быть получены из центрального сервис-менеджера через обозначение сервиса `ViewHelperManager`.

Загрузка модулей

После того, как `ModuleManager` был подготовлен и необходимые слушатели (`listeners`) были зарегистрированы, фактическая загрузка модулей инициализируется вызовом метода `loadModules()` менеджера модулей. На этот момент здесь действительно мало что происходит потому, что фактическая обработка, например, вызов вышеупомянутых методов из `Module.php`, на самом деле осуществляется множеством зарегистрированных слушателей. Первоначально иницируется событие `loadModules.pre`, затем – событие `loadModule.resolve` и `loadModule` для каждого активированного модуля. Наконец, иницируется событие `loadModules.postevent`. И это действительно все.

Объекты событий модуля

Концепция `EventManager` заключается в том, что в дополнение к существующим инициаторам для событий и слушателям, зарегистрированным для события (приемникам), само событие, представленное, как самостоятельный объект, по-прежнему существует. Оно доступно для всех слушателей. Этот объект служит для передачи дополнительной информации о событии, например, ссылки на место в коде, откуда событие было иницировано. Кроме того, могут быть переданы дополнительные данные, которые

полезны для обработки событий в слушателях. Таким образом модуль, который в настоящее время загружен, как правило, интересен для слушателя в процессе выполнения `loadModule`. Отдадим должное тому факту, что в зависимости от контекста события `EventManager` Zend Framework 2 позволяет сохранить другие данные, представляющие интерес, в специальных ситуативно-зависимых классах событий. Для событий принципиально в рамках `ModulManager`, существует специальный класс `ModuleEvent`, который, например, который содержит как сам модуль, так и дополнительно имя модуля.

Активация модуля

Чтобы модуль был принят во внимание вообще, требуется явная активация его в `application.config.php` в каталоге конфигурации:

```
1 <?php
2 return array(
3     'modules' => array(
4         'Application',
5         'Helloworld',
6     )
7 );
```

Listing 6.21

Методы класса модуля

Как мы уже видели, большое количество методов класса модуля вызывается, если мы их реализовали. У фреймворка существуют две соответствующие возможности выяснить, так ли это. Либо класс модуля реализует определенный интерфейс (это, на самом деле, может быть протестировано с использованием экземпляра), либо соответствующий метод просто реализован (это можно проверить вызовом `method_exists()`).

Давайте теперь еще раз тщательно рассмотрим методы класса модуля, которые автоматически вызываются фреймворком и могут быть использованы разработчиками приложений:

- `getAutoloadingConfig()` (определен в `AutoloaderProviderInterface`): мы уже создавали этот метод в нашем модуле `Helloworld`. Он предоставляет информацию о том, как классы модуля могут быть автоматически загружены. Если опустить этот метод, классы модуля (например, его контроллер) не могут быть загружены автоматически, и возникают серьезные проблемы, когда вызывается соответствующий URL. Следовательно, он всегда должен предоставлять информацию фреймворку о том, как классы могут быть загружены автоматически. Кстати, в чисто техническом контексте, фреймворк берет информацию для регистрации подходящей для этого модуля реализации автозагрузчика с помощью `spl_autoload_Register now()`.
- `init()` (определен в `InitProviderInterface`): этот метод позволяет разработчику приложений выполнить инициализацию своего собственного модуля, например, чтобы зарегистрировать своих собственных слушателей для определенных событий.

ModuleManager передается в этот метод, и последний может таким образом, при необходимости, получить доступ к соответствующим событиям (из ModuleManager) или получить доступ к модулям. Важно то, что этот "метод" вызывается всегда – это значит, что для каждого запроса и вообще для каждого модуля. Следует также понимать, что это хорошее место, чтобы нанести ущерб времени загрузки приложения. Таким образом, только небольшие и только легковесные операции должны выполняться в рамках метода `init()`. Если вы пытаетесь оптимизировать скорость ZF2 приложений, всегда следует сначала взглянуть на методы `init()` активированных модулей.

Вот пример использования метода `init()` :

```
1 <?php
2 namespace Helloworld;
3
4 use Zend\ModuleManager\ModuleManager;
5 use Zend\ModuleManager\ModuleEvent;
6
7 class Module
8 {
9     public function init(ModuleManager $moduleManager)
10    {
11        $moduleManager->getEventManager()
12            ->attach(
13                ModuleEvent::EVENT_LOAD_MODULES_POST,
14                array($this, 'onModulesPost')
15            );
16    }
17
18    public function onModulesPost()
19    {
20        die("Modules loaded!");
21    }
22
23 // [...]
24 }
```

Listing 6.21

- `onBootstrap()` (определен в `BootstrapListenerInterface`): дополнительная возможность для разработчика приложений для реализации начальной загрузки конкретного модуля. По сути, у этого метода то же назначение и полезные свойства, что и у `init()`, но вызывается `onBootstrap` позже в ходе обработки, а именно, когда `ModuleManager` уже завершил свою работу и вернул управление приложению (`Application`). Таким образом, при использовании `onBootstrap()` доступны методы, сервисы и данные, которые еще не были доступны в `init()`.

- `getConfig()` (определен в `ConfigProviderInterface`): Мы также уже знакомы с этим методом. Он обеспечивает возможность передачи файла конфигурации модуля, который в соответствии с соглашением называется `module.config.php` и хранится в этом модуле в подкаталоге `config`. Однако, это не обязательно. Строго говоря, этот метод абсолютно необходим для выполнения модуля, но на практике никто не может обойтись без конфигурационного файла этого модуля, который становится доступным для фреймворка с помощью этого модуля. Что касается конфигурации, фреймворк обеспечивает определенную степень гибкости. Таким образом, либо все конфигурации могут быть доступными в одном или более внешних файлах через `getConfig()`, либо специальные "конфигурационные методы" могут быть реализованы в классе модуля. Последние относятся к сервис-менеджерам, которые присутствуют в системе, то есть к `ServiceManager`, `ControllerLoader`, `ViewHelperManager` и `ControllerPluginManager`.
- `getServiceConfig()`: позволяет настраивать `ServiceManager` и эквивалентен ключу массива `config` в разделе `service_manager` в `module.config.php`.
- `getControllerConfig()`: позволяет настраивать `ControllerLoader` и эквивалентен ключу массива `config` в разделе `controllers` в `module.config.php`.
- `getControllerPluginConfig()`: позволяет настраивать `ControllerPluginManager` и эквивалентен ключу массива `config` в разделе `controller_plugins` в `module.config.php`.
- `getViewHelperConfig()`: позволяет настраивать `ViewHelperManager` и эквивалентен ключу массива `config` в разделе `view_helpers` в `module.config.php`.

В этой связи еще один пример: наш собственный помощник представления (`ViewHelper`) (подробнее о концепции "помощник представления" на следующих страницах) можно сделать известным в рамках `module.config.php` следующим образом

```
1 <?php
2 'view_helpers' => array(
3     'invokables' => array(
4         'displayCurrentDate'
5         => 'Helloworld\View\Helper\DisplayCurrentDate'
6     )
7 )
```

Listing 6.22

либо с помощью следующего метода в `Module.php`:

```
1 <?php
2 public function getViewHelperConfig()
3 {
4     return array(
5         'invokables' => array(
6             'displayCurrentDate'
7             => 'Helloworld\View\Helper\DisplayCurrentDate'
8         )
9     );
10 }
```

```
9    );  
10 }
```

Listing 6.23

Приложение (Application)

Теперь, когда `ServiceManager` оснащен необходимыми сервисами и `ModuleManager` загрузил модули приложения, само приложение может быть запущено и обработка запросов может быть инициирована.

Создание приложения и начальная загрузка

Объект приложения генерируется с помощью фабрики, которая зарегистрирована в `ServiceManager`. В рамках генерации, приложение вызывает ряд стандартных сервисов, в том числе запрос (`Request`) в качестве основы для дальнейшей обработки, и ответ (`Response`), который должен быть заполнен результатами обработки запроса. Кроме того, приложение получает ссылку на `ModuleManager` и собственный экземпляр `EventManager` (который, в действительности, хранится в `ServiceManager`, как неразделяемый, поэтому возвращается новый экземпляр этой службы). Последний дает возможность приложению – аналогично как для `ModuleManager` – инициировать события и информировать слушателей. Приложение устанавливает ряд событий в области обработки.

- `bootstrap` (начальная загрузка) : выполняется при запуске приложения.
- `route` (маршрут) : происходит, когда для URL - адреса определяются контроллер и действие.
- `dispatch` (диспетчеризация) : происходит, когда определенный контроллер выявлен и вызывается.
- `finish` (завершение) : выполняется, когда работа приложения была завершена.

В ходе начальной загрузки приложение регистрирует (кстати, в данном случае, само приложение, а не `ApplicationFactory`) ряд слушателей, которых оно получает через `ServiceManager`: `RouteListener` для события `route`, `DispatchListener` для события `dispatch`, и `ViewManager` для красочного букета дополнительных слушателей, которые выполняют обработку шаблонов и макетов. Подробнее об этом в следующем разделе.

Кроме того, приложение создает затем MVC - событие (`MvcEvent`), которое регистрируется с помощью `EventManager`, как объект события. `MvcEvent` затем позволяет слушателям получить доступ к `Request`, `Response`, `Application` и `Router`.

Наконец, событие начальной загрузки инициируется, и выполняются зарегистрированные слушатели. Они также могут быть, в частности, отдельными модулями приложения, зарегистрированными только для этого события.

Выполнение приложения

Метод `run()`, который выполняется после начальной загрузки, приводит в действие затем ряд "рычагов", которые уже были помещены в корректную позицию. Начнем с того, приложение инициирует событие `route`. `RouteListener`, который был зарегистрирован для этого события заранее, запускается, и `Router` из `MvcEvent` запрашивается на предоставление своего сервиса, т.е. на приведение соответствия URL к определенному маршруту. В этом случае маршрут является описанием URL - адреса на основе определенного шаблона. Детально мы рассмотрим механику маршрутизации позже. На данный момент мы должны только помнить, что теперь либо маршрутизатор найдет маршрут, соответствующий вызванному URL и, таким образом, определит соответствующий контроллер, а также соответствующее действие, либо маршрутизатор вернется с плохими новостями, и поиск не принес результатов вообще. Но давайте сначала остановимся на успешном пути в этом случае. Соответствующий маршрут сохраняется в виде объекта `RouteMatch` в `MvcEvent` посредством `RouteListener`. `MvcEvent` таким образом все больше оказывается центральным объектом в ряду других имеющихся важных объектов и данных. Затем инициируется событие `dispatch`, вызывается `ControllerLoader` и использует `MvcEvent`, чтобы найти выявленный контроллер, экземпляр которого должен быть создан. Для этого `DispatchListener` запрашивает `ControllerLoader` из `ServiceManager` (а `ControllerLoader`, с технической точки зрения, такой же "ServiceManager"), а затем из него (т.е., из `ControllerLoader`) текущий контроллер. В ходе этого контроллер также оснащается своим собственным `EventManager`. Теперь, таким образом, он может приводить в действие события и управлять слушателями. Затем вызывается метод контроллера `dispatch()`, и он выполняет дальнейшую обработку самостоятельно. Если в ходе этого предприятия возникает любая некритическая проблема, инициируется событие `dispatch.error`; в противном случае, результат вызова `dispatch()` сохраняется в `MvcEvent` и дополнительно возвращается, и на этом процесс диспетчеризации в контроллере завершается.

В Zend Framework контроллеры задуманы таким образом, чтобы у них был метод `dispatch()`. Он вызывается извне, в процессе передается объект запроса, и ожидается, что контроллер вернет объект типа `Zend\Stdlib\ResponseInterface` после завершения работы. Чтобы принцип контроллера и действий функционировали так, как мы привыкли и этого ожидаем, эта логика должна быть реализована в самом контроллере, в противном случае будет вызываться только метод `dispatch`, но не соответствующий метод "действия". Для того, чтобы это обеспечить, не нужно делать это самостоятельно, достаточно наследовать свой собственный контроллер от `Zend\Mvc\Controller\AbstractActionController`. Соответственно, любые произвольные действия могут быть размещены в контроллере, если придерживаться соглашения, что имя метода должно заканчиваться на "action":

```
1 <?php
2
3 namespace Helloworld\Controller;
4
5 use Zend\Mvc\Controller\AbstractActionController;
6 use Zend\View\Model\ViewModel;
```

```

7
8 class IndexController extends AbstractActionController
9 {
10     private $greetingService;
11
12     public function indexAction()
13     {
14         return new ViewModel(
15             array(
16                 'greeting' => $this->greetingService->getGreeting(),
17                 'date' => $this->currentDate()
18             )
19         );
20     }
21 }

```

Listing 6.25

Возвращаясь к приложению, два результата – в настоящее время иницируются `render` и `finish` и выполнение метода `run()` завершается. Кстати, следующий факт очень интересен и полезен: как правило, действие возвращает объект типа `ViewModel` в конце обработки. Это, таким образом, косвенный сигнал последующему шагу обработки о том, что результат все равно должен быть обработан прежде, чем он может быть возвращен.

```

1 <?php
2 public function indexAction()
3 {
4     return new ViewModel(
5         array(
6             'greeting' => $this->greetingService->getGreeting(),
7             'date' => $this->currentDate()
8         )
9     );
10 }

```

Listing 6.26

Тем не менее, очень практичной деталью реализации является то, что если вместо `ViewModel` возвращается объект `Response`, последующая деятельность по визуализации опускается.

```

1 <?php
2 public function indexAction()
3 {
4     $resp = new \Zend\Http\PhpEnvironment\Response;
5     $resp->setStatusCode(503);
6     return $resp;
7 }

```

Listing 6.27

Этот механизм полезен, если пользователь решит, например, кратко вернуть 503 код в случае, если приложение просто проходит плановое техническое обслуживание, или когда необходимо вернуть данные определенного MIME - типа, например, содержимое изображения, PDF документ или что-либо подобное.

Менеджер представления

Прежде, чем закончится процесс обработки, ViewManager снова вступает в действие. В предыдущем разделе мы уже видели, что метод `onBootstrap()` выполняется в рамках начальной загрузки приложения (поскольку он зарегистрирован для соответствующего события), и что в нем выполнен ряд дополнительных приготовлений. До сих пор мы смешивали это для простоты, но теперь мы должны также рассмотреть детали. После того, как ViewManager с его многочисленными сотрудниками завершил свою работу, мы фактически сразу проложили себе путь через всю цепочку обработки.

Начнем с того, что ViewManager получает Config из ServiceManager, который в это время уже представляет собой объединенную "общую конфигурацию" приложения и модулей. ViewManager ищет ключ `view_manager` и использует сохраненные под этим ключом конфигурации. Тогда старые игры в регистрацию разнообразных слушателей и в предоставление дополнительных услуг начинаются снова.

Слушатели (Listeners), ориентированные на представление (View)

Следующие слушатели формируются, и все они прикрепляются к событию `dispatch` класса `ActionController`:

- `CreateViewModelListener`: гарантирует, что после выполнения контроллера объект типа `ViewModel` будет доступен для визуализации, даже если контроллером был представлен только `NULL` или массив.
- `RouteNotFoundStrategy`: создает модель представления (`ViewModel`) для случая, когда никакой контроллер не был определен, и никакая модель представления не может быть сгенерирована (ошибка 404).
- `InjectTemplateListener`: добавляет соответствующий шаблон в модель представления для последующей визуализации.
- `InjectViewModelListener`: добавляет модель представления в `MvcEvent`. Этот слушатель зарегистрирован также для события приложения `dispatch.error` для того, чтобы можно было сделать модель представления доступной также в случае ошибки.

Событие `dispatch` в этой связи несколько неприятно: оно происходит в системе дважды. В первый раз оно инициируется приложением (`Application`), а затем снова контроллером действий (`ActionController`). Даже если название событий идентично (оно действительно может быть выбрано произвольно), то в связи с тем, что оно вызвано двумя различными менеджерами событий, мы имеем дело с двумя совершенно различными событиями.

Кстати, в это время `EventManager`, который соответствует конкретному олицетворению `ActionController`, еще не существует, поскольку последний еще не был сформирован вообще. В этом случае проблема решается с помощью `SharedEventManager`. С помощью `SharedEventManager` слушателей для события можно зарегистрировать в `EventManager`, даже не существующем на момент регистрации. В настоящее время проще оставить все, как есть. Мы более подробно рассмотрим, как реализуется этот механизм в следующей главе.

Сервисы, ориентированные на представление (View)

Кроме того, ряд сервисов, ориентированных на представление, доступны в `ServiceManager`:

- `View and View Model`: начнем с того, что существует само представление с его моделью представления, отображение "полезной нагрузки", сформированной из запроса для ответа.
- `DefaultRenderingStrategy`: может иметь доступ к представлению и зарегистрирована для события приложения `render`. Если это событие наступает, представление (`View`) передается в модель представления (`ViewModel`), которая берется из `MvcEvent`, а затем вызывает визуализацию (`render`) только этого представления (`View`).
- `ViewPhpRendererStrategy`: тем не менее, фактическая визуализация (`rendering`) не выполняется представлением (`View`) самостоятельно, а вместо этого делегирует к `ViewPhpRendererStrategy`, которая изначально определяет соответствующие визуализаторы и передает после обработки готовые результаты в `Response`.
- `RouteNotFoundStrategy`: определяет соответствующее поведение в случае 404 ситуации.
- `ExceptionStrategy`: определяет соответствующее поведение в случае ошибок диспетчеризации (`dispatch errors`).
- `ViewRenderer`: берет на себя работу фактической визуализации, то есть слияние данных из `ViewModel` и соответствующего шаблона.
- `ViewResolver`: для локализации соответствующего шаблона `ViewRenderer` обращается к `ViewResolver`.
- `ViewTemplatePathStack`: позволяет резольверу локализовать шаблон на основе сохраненных путей.
- `ViewTemplateMapResolver`: позволяет резольверу локализовать шаблон на основе назначения соответствий "ключ-значение".
- `ViewHelperManager`: делает возможным доступ к помощникам вида (`ViewHelpers`) в шаблонах, это упрощает генерацию динамической разметки.

Резюме

На первый взгляд отношения кажутся очень сложными; реализация шаблона MVC в `Zend Framework` кажется как-то чересчур усложненной. Основной причиной этого чувства является то, что, с одной стороны, вся процедура обработки разбивается на чрезвычайно малые отдельные шаги, которые представлены через отдельные классы в каждом

отдельном случае, и которые также должны быть в хронологическом и концептуальном порядке "организованы" с тем, чтобы они в конечном счете конструктивно взаимодействовали в той степени, которая требуется. С другой стороны, чрезмерное использование событий и слушателей делает его сложным для понимания процессов и отношений в рамках приложения. Также использование многочисленных шаблонов проектирования точно не способствует пониманию, особенно в начале, особенно, если вы еще не привыкли к ним.

Разве нельзя значительно упростить вещи? Должна ли действительно реализация MVC всегда быть настолько сложной? Быстрый, рефлекторный ответ на эти вопросы будет: Да. Нет. Существует большое количество так называемых "Микро MVC фреймворков", таких, как Silex или MicroMVC, которые, на первый взгляд, могут достичь подобных результатов при значительно меньшей сложности, чем реализация MVC в Zend Framework 2. Однако, это верно только на первый взгляд.

Начнем с того, что Zend\Mvc – это, фактически, намного больше, чем "MVC". Это на самом деле платформа для приложений, которая позволяет 1) выполнить простую и эффективную интеграцию дополнительной функциональности в виде собственных или сторонних модулей 2) изменить или полностью реструктурировать выполнение запросов практически произвольно 3) что, в результате этого, приводит к низкой связанности отдельных компонентов и сервисов, а также позволяет выполнить требования приложений компании в отношении удобства сопровождения и расширяемости, а также тестируемости. Zend\Mvc успешно выполняется в среде программных компонентов и способен поднять уровень абстракции, на котором ранее продвигался разработчик при создании веб-приложений на PHP, на новый, значительно более продуктивный. По крайней мере, это начало для достижения всего этого. Удастся ли это в действительности – должно быть доказано. Является ли Zend Framework 2 правильным программным обеспечением для моего собственного проекта? Я думаю, что ответить на этот вопрос для версии 2 сложнее, чем это было для версии 1. Преимущества версии 2, в частности, направлены на профессиональных разработчиков, которые не всегда имеются. Является ли Zend Framework 2 подходящим программным обеспечением для приложения моей компании в веб? Да, я бы категорически сказал, что в любом случае.

Давайте подведем итоги этой главы и кратко вновь рассмотрим процесс обработки запроса. Начнем с того, что запрос переадресуется на index.php посредством URL rewriting (например, с помощью "mod_rewrite" и соответствующего файла .htaccess). Здесь сконфигурирована автозагрузка, которая обеспечивает функционирование должным образом как самого Zend Framework, так и, при необходимости, дополнительно используемых библиотек. Затем запускается приложение (Application), которое изначально гарантирует, что ServiceManager, центральный "сервис доступа", создан и оснащен важными сервисами: ModuleManager и EventManager. Кроме того, доступны SharedEventManager. Как и в случае с ModuleManager и EventManager, "доступны" часто означает, что для начала известны только фабрики, которые используются для последующего создания фактических сервисов в каждом конкретном случае. Почему мы должны идти в обход с помощью фабрик? С одной стороны потому, что они позволяют нам

заменять конкретную реализацию соответствующего сервиса при необходимости. А с другой стороны потому, что в рамках создания сервиса, создается не только сам сервис, но также и некоторое количество слушателей (listeners), которые зарегистрированы для последующей обработки событий, как это имеет место для ModuleManager. Действительно, ModuleManagerFactory, ApplicationFactory и ViewManager выполняют подобным образом создание сервисов. Они изначально создают собственный сервис, а затем ряд дополнительных (суб-) сервисов, которые впоследствии будут доступны для сервиса, и, наконец, один или несколько слушателей для событий своего или других сервисов. Слушатели затем берут на себя конкретную задачу или обращаются обратно к сервисам.

Эта процедура гарантирует, что методы запуска, loadModules, bootstrap, ModuleManager & со., Application & со. очень скудны, и реально сами ничего не делают, кроме инициирования событий. Все остальное происходит с помощью слушателей сервисов. В то время, как это выполняется, ModuleManager иницирует следующие события: loadModules.pre, loadModule.resolve, loadModule и loadModules.post. Затем приложение устанавливает начальную загрузку, маршрут, и события диспетчеризации; затем – контроллер с собственной диспетчеризацией событий (не следует путать с диспетчеризацией приложения); наконец, снова приложение с визуализацией (render), перед тем, как представление (View) предоставит визуализацию и ответ (Response). Последнее, но не менее важное – приложение завершается инициированием события finish. В случае ошибки в области маршрутизации или в процессе диспетчеризации, приложение иницирует событие dispatch.error. Это сигнализирует о том, что произошла ошибка во время обработки.

VII. Менеджер событий (EventManager)

Мы увидели в предыдущей главе, в какой степени фреймворк в целом базируется на идее инициаторов событий (triggers), объектов событий и слушателей событий (listeners). В процессе соответствующие объекты или "менеджеры", каждый обращается к EventManager, который управляет событиями соответствующего объекта, а также позволяет добавлять и удалять слушателей. Поскольку ZendEventManager играет такую важную роль для функционирования фреймворка, а также потому, что это может оказаться очень полезным при разработке собственных приложений, мы детально рассмотрим его далее.

Регистрация слушателей (listeners)

В частности, EventManager содержит два метода, интересных для слушателей:

```
1 <?php
2 public function attach($event, $callback = null, $priority = 1);
3 public function detach($listener);
```

Listing 7.1

С помощью метода attach() слушатель может быть зарегистрирован для конкретного события, в то время, как detach() удаляет слушателя. Слушатели, которые зарегистрированы для события, информируются (о наступлении события) в порядке очередности. Причем в данном контексте "информируются" означает, что соответственно зарегистрированные обратные вызовы – знакомых нам родных функций PHP, в дополнение к функциям, методам классов и методам объектов, также в виде замыканий – будут вызваны.

В этом контексте должно быть первоначально определено обозначение события, для которого должен быть зарегистрирован слушатель. В качестве соглашения для обозначения события, часто прибегают к магической константе `__FUNCTION__`, чтобы имя метода, который инициировал событие, стало именем для самого события. Однако, это не обязательно, имя может быть выбрано произвольно:

```
1 <?php
2 $greetingService->getEventManager()->attach(
3     'event1',
4     function($e) {
5         // [...]
6     }
7 );
```

Listing 7.2

В этом случае для события `event1` зарегистрирована функция обратного вызова. В ходе регистрации вместо строки может быть передан массив с несколькими обозначениями события, если вы хотите зарегистрировать одного слушателя для многих событий.

```
1 <?php
2 $greetingService->getEventManager()->attach(
3     array('event1', 'event2'),
4     function($e) {
5         // [...]
6     }
7 );
```

Listing 7.3

Регистрация нескольких слушателей одновременно

Однако, можно не только зарегистрировать одного слушателя сразу для ряда событий, но также и зарегистрировать ряд слушателей для одного или даже одновременно для нескольких событий одним махом. Для этого используются так называемые "агрегаты слушателей" (`ListenerAggregates`). Они особенно полезны, когда мы хотим сгруппировать регистрацию отдельных слушателей логически. Для этого нужно сделать собственный класс, реализующий интерфейс `Zend\EventManager\ListenerAggregateInterface` и, таким образом, содержащий методы `attach()` и `detach()`. Конкретные слушатели регистрируются в теле этих методов:

```
1 <?php
2 namespace Helloworld\Event;
3
4 use Zend\EventManager\ListenerAggregateInterface;
5 use Zend\EventManager\EventManagerInterface;
6
7 class MyGetGreetingEventListenerAggregate
8 implements ListenerAggregateInterface
9 {
10     public function attach(EventManagerInterface $eventManager)
11     {
12         $eventManager->attach(
13             'getGreeting',
14             function($e) {
15                 // [...]
16             }
17         );
18
19         $eventManager->attach(
20             'refreshGreeting',
21             function($e) {
```

```

22             //[...]
23         }
24     );
25 }
26
27 public function detach(EventManagerInterface $events)
28 {
29     // [...]
30 }
31 }

```

Listing 7.4

Добавление слушателей можно выполнить одной строкой кода, так как метод `attach()` выполняет всю черную работу и вызывается автоматически:

```

1 <?php
2 $greetingService
3     ->getEventManager()
4     ->attach(
5         new \Helloworld\Event\MyGetGreetingEventListenerAggregate()
6     );

```

Listing 7.5

Удаление зарегистрированных слушателей

Уже зарегистрированный слушатель может быть удален с помощью метода `EventManager detach()`. Для удаления нескольких слушателей можно либо удалять слушателей индивидуально, либо использовать `ListenerAggregateInterface detach()`, аналогично `attach()`. Для этого нужно передать в `detach()` соответствующий `CallbackHandler`, который был, например, возвращен в качестве результата методом `attach()`:

```

1 <?php
2 $handler = $greetingService->getEventManager()
3     ->attach('getGreeting', function($e){ // [...] });
4
5 $greetingService->getEventManager()->detach($handler);

```

Listing 7.6

Инициирование события

Следующего вызова достаточно для инициирования события, если `EventManager` доступен в переменной-члене объекта:

```

1 <?php
2 $this->eventManager->trigger('event1');

```

Listing 7.7

Таким образом вызываются все слушатели, которые были зарегистрированы для данного события. Весь процесс, естественно, происходит последовательно. Каждый слушатель вызывается индивидуально, и только после полной обработки обрабатывается следующий слушатель. В процессе последовательность обработки обусловлена приоритетом слушателя, который может быть объявлен для слушателя в методе `attach()`, или, как бы то ни было, последовательностью, в которой слушатели были добавлены.

В метод `trigger()` могут быть переданы либо, как показано выше, строка, которая представляет собой имя соответствующего события, либо соответствующий объект события:

```
1 <?php
2 $event = new Zend\EventManager\Event();
3 $event->setName('getGreeting');
4 $this->eventManager->trigger($event);
```

Listing 7.7

Кстати, вызов `trigger()` возвращает результат типа `ResponseCollection`, а именно: все то, что вызванные слушатели ранее вернули индивидуально, возвращается коллективно. С помощью методов `first()` и `last()` можно получить доступ к наиболее важным возвращаемым значениям, и проверить наличие конкретного возвращаемого значения в коллекции – с помощью `contains($value)`. В этом контексте результаты выполнения слушателей представлены, как уже было описано выше, либо с приоритетом, явно указанным в рамках `attach()`, либо, в качестве альтернативы, просто в последовательности, с которой слушатели были добавлены. Здесь применяется принцип FIFO: первый вошел, первый вышел. Слушатель, который был добавлен первым, будет, таким образом, выполнен в первую очередь.

И еще один интересный момент: обработка отдельных слушателей может быть также прервана интерактивно. Чем это хорошо? Давайте рассмотрим на примере: представьте, что мы разрабатываем сайт, на котором пользователи могут оставлять отзывы о книгах. Для того, чтобы присваивать эти отзывы правильно, мы для начала находим ISBN книги и тем самым обеспечиваем, чтобы название книги, автор, и т.д. не вводились пользователем вручную. Для этого нам нужно одно из двух – либо получить данные, которые уже находятся в базе данных (однако, это только в том случае, если хотя бы одна рецензия на эту книгу уже была написана), либо получить подробности о книге с удаленного веб-сервиса, например, с Amazon. Как теперь загрузить наши данные наиболее эффективно? Мы могли бы подготовить два слушателя, которые мы хотели бы зарегистрировать для события `onBookDataLoad` нашего приложения, при этом первый слушатель ищет нужные данные в базе данных, а второй – на удаленном веб-сервисе. Тем не менее, второй слушатель выполняется только тогда, когда первый не был успешен. Для достижения этой цели выполнение события может быть продлено на обратный вызов, который будет проверяться на конкретное возвращаемое значение:

```
1 <?php
2 $results = $this->events()->trigger(
```

```

3  __FUNCTION__,
4  $this,
5  array(),
6  function ($returnValue) {
7      return ($returnValue instanceof MyModule\Model\Book);
8  }
9 );

```

Listing 7.8

Если первый слушатель вернет объект типа `MyModule\Model\Book`, значит подробности о книге были успешно загружены, и обработка в этот момент завершится.

В качестве альтернативы, обработка может быть также прекращена в случае, если слушатель активно вызовет метод объекта события `stopPropagation()`.

SharedEventManager

`SharedEventManager` является решением следующей проблемы: что делать, если слушатель хочет зарегистрировать себя для инициатора события, который еще не существует на момент, нужный для регистрации? На практике эта проблема возникает, когда модуль хочет зарегистрировать функцию обратного вызова для события контроллера "dispatch" в рамках своего метода `init()` или `onBootstrap()`. В этот момент контроллер со своим менеджером `EventManager` еще не воплощен в жизнь. Таким образом, это просто невозможно.

Если новый `EventManager` запрашивается через `ServiceManager`, последний обеспечивает, чтобы первый дополнительно получил ссылку на `SharedEventManager`, разделяемый для всех экземпляров `EventManager`, которые были получены таким образом. Слушатели для определенного события могут быть зарегистрированы в `SharedEventManager` при помощи декларации "идентификатора" слушателя. В этом контексте для него изначально нет никакой разницы, существует ли уже `EventManager`, который позже запустит событие, или нет. Важен лишь тот факт, что используются "идентификаторы" при регистрации слушателя, и что соответствующий `EventManager` впоследствии распознает и выполняет его. Давайте рассмотрим конкретный пример, который позволит уточнить принцип. Когда выполняется метод `init()` класса `Module` нашего `HelloWorld`, `Application` еще не существует. Первые признаки жизни появятся, когда модуль будет полностью загружен. Если мы, таким образом, хотим теперь зарегистрировать слушателя для события приложения `route`, мы должны это сделать с помощью `SharedEventManager`, у нас нет другого выбора:

```

1 <?php
2 // [...]
3 public function init(ModuleManager $moduleManager)
4 {
5     $moduleManager
6         ->getEventManager()

```

```

7         ->attach(
8             ModuleEvent::EVENT_LOAD_MODULES_POST,
9             array($this, 'onModulesPost')
10        );
11
12    $sharedEvents = $moduleManager
13        ->getEventManager()->getSharedManager();
14
15    $sharedEvents->attach(
16        'application',
17        'route',
18        function($e) {
19            die("Event '{$e->getName()}' wurde ausgeloeset!");
20        }
21    );
22 }

```

Listing 7.9

Сейчас мы находимся в файле `Module.php` модуля `HelloWorld`. С помощью `ModuleManager` мы получаем `EventManager`, а с помощью последнего, в свою очередь, `SharedEventManager`, который автоматически был сделан разделяемым (совместно используемым) для всех "EventManagers", которые будут сгенерированы сервис-менеджером. Здесь мы теперь регистрируем обратный вызов (характерно в виде замыкания) для события `route`, которое будет инициировано/запущено приложением или его `EventManager` –ом, соответственно, в какой-то момент в будущем. Ключ `application` в этом случае предусмотрен соглашением, является строкой, это нужно знать. На момент, когда приложение инициирует затем событие `route` с помощью собственного `EventManager` – который мы не могли использовать для регистрации слушателя до этого времени – все слушатели, которые регистрировались в `SharedEventManager` для ключа `application` и соответствующего события, также будут проинформированы. Это очень практично!

Следующие идентификаторы предварительно сконфигурированы для индивидуальных компонентов фреймворка:

- Для `ModuleManager`: `module_manager`, `Zend\ModuleManager\ModuleManager`.
- Для `Application`: `application`, `Zend\Mvc\Application`.
- `AbstractActionController` (базовый класс для собственного контроллера): `Zend\Stdlib\DispatchableInterface`, `Zend\Mvc\Controller\AbstractActionController`, первая часть пространства имен контроллера (например, для `HelloWorld\Controller\IndexController` это было бы `HelloWorld`)
- `View`: `Zend\View\View`

`SharedEventManager` поэтому особенно подходит для ситуаций, в которых `EventManager`, который фактически будет нести ответственность, и который мы хотели бы использовать для регистрации слушателей, еще не доступен.

Помимо упомянутых выше идентификаторов, которые автоматически создаются фреймворком, могут быть определены дополнительные идентификаторы для своих собственных целей.

Использование событий в своих собственных классах

Zend\EventManager дает возможность администрировать слушателей и позволяет классу стать инициатором события. Его функциональность не ограничивается классами и другими компонентами фреймворка, как раз наоборот, Zend\EventManager вполне подходит для того, чтобы служить независимой реализацией обработки управления событиями.

Для этого собственный класс, который должен инициировать событие, должен просто содержать экземпляр Zend\EventManager в переменной-члене. Давайте снова воспользуемся GreetingService, которым мы пользовались ранее. Предположим, что мы хотели бы делать запись в лог-файл всякий раз, когда генерируется дата, чтобы всегда знать, как часто эта услуга вызывается в определенный промежуток времени. Как это можно реализовать? Давайте рассмотрим одну возможность.

Начнем с того, что мы создадим дополнительный сервис в нашем модуле. Для этого мы подготовим файл src/Helloworld/Service/LoggingService.php со следующим содержанием:

```
1 <?php
2 namespace Helloworld\Service;
3
4 class LoggingService
5 {
6     public function onGetGreeting()
7     {
8         // Logging-Implementierung
9     }
10 }
```

Listing 7.10

Мы проигнорируем определенную реализацию журналирования (предусмотренную фреймворком) на данный момент, потому что мы прежде всего рассматриваем взаимодействие различных служб через систему событий. Мы хотим вызвать onGetGreeting(), как только произойдет событие getGreeting сервиса GreetingService.

Кроме того, мы должны обеспечить, чтобы служба была известна системе. Для этого мы добавляем соответствующий класс, как invocable в Module.php нашего модуля, в метод getServiceConfig().

```
1 <?php
2 // [...]
```

```

3 'invokables' => array(
4     'loggingService' => 'Helloworld\Service\LoggingService'
5 )
6 // [...]

```

Listing 7.11

С этого момента `LoggingService` может быть запрошен из сервис-менеджера с помощью ключа `loggingService`. Пока все хорошо. Теперь мы должны дополнительно обеспечить, чтобы `GreetingService` мог инициировать событие, на основе которого может быть запущен метод `onGetGreeting` сервиса `LoggingService`. Чтобы предоставить для `GreetingService` `EventManager` и одновременно зарегистрировать выполнение метода `onGetGreeting` сервиса `LoggingService`, мы поместим фабрику для `GreetingService`:

```

1 <?php
2 namespace Helloworld\Service;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class GreetingServiceFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $greetingService = new GreetingService();
12
13         $greetingService->setEventManager (
14             $serviceLocator->get('eventManager')
15         );
16
17         $loggingService = $serviceLocator->get('loggingService');
18
19         $greetingService->getEventManager()
20             ->attach(
21                 'getGreeting',
22                 array($loggingService, 'onGetGreeting')
23             );
24
25         return $greetingService;
26     }
27 }

```

Listing 7.12

Мы формируем файл фабрики в `src/Helloworld/Service/GreetingServiceFactory.php`; он, таким образом, находится в том же каталоге, что и сама служба. Первоначально, фабрика генерирует `GreetingService`, который она, в конечном итоге, возвращает. Предварительно фабрика размещает `EventManager` в `GreetingService`, так что `GreetingService` может теперь также инициировать события и управлять слушателями. А

затем `LoggingService` и его метод `onGetGreeting()` регистрируются для события `getGreeting`. При использовании замыкания мы можем даже обеспечить, чтобы `$loggingService`, который запрашивается в приведенном выше примере непосредственно, впервые запрашивался в фактический момент события с помощью "отложенной инициализации":

```
1 <?php
2 namespace HelloWorld\Service;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class GreetingServiceFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $greetingService = new GreetingService();
12
13        $greetingService->setEventManager(
14            $serviceLocator->get('eventManager')
15        );
16
17        $greetingService->getEventManager()
18            ->attach(
19                'getGreeting',
20                function($e) use($serviceLocator) {
21                    $serviceLocator
22                        ->get('loggingService')
23                        ->onGetGreeting($e);
24                }
25            );
26
27        return $greetingService;
28    }
29 }
```

Listing 7.13

Преимущество очевидно: `LoggingService` генерируется фактически только тогда, когда он должен быть использован. Действительно, событие `getGreeting` может просто никогда не состояться.

Мы все еще должны адаптировать метод `getServiceConfig()` класса `Module` нашего модуля таким образом, чтобы `ServiceManager` использовал теперь новую фабрику для генерации `GreetingService`. Теперь `getServiceConfig()` выглядит следующим образом:

```
1 <?php
2 // [...]
3 public function getServiceConfig()
```

```

4 {
5     return array(
6         'factories' => array(
7             'greetingService'
8                 => 'Helloworld\Service\GreetingServiceFactory'
9         ),
10        'invokables' => array(
11            'loggingService'
12                => 'Helloworld\Service\LoggingService'
13        ),
14    );
15 }

```

Listing 7.14

Теперь мы научим GreetingService запускать соответствующее событие:

```

1 <?php
2 namespace Helloworld\Service;
3
4 use Zend\EventManager\EventManagerInterface;
5
6 class GreetingService
7 {
8     private $eventManager;
9
10    public function getGreeting()
11    {
12        $this->eventManager->trigger('getGreeting');
13
14        if(date("H") <= 11) {
15            return "Good morning, world!";
16        } else if (date("H") > 11 && date("H") < 17) {
17            return "Hello, world!";
18        } else {
19            return "Good evening, world!";
20        }
21    }
22
23    public function getEventManager()
24    {
25        return $this->eventManager;
26    }
27
28    public function setEventManager(EventManagerInterface $em)
29    {
30        $this->eventManager = $em;
31    }
32 }

```

Listing 7.15

И это все. Если вызывается метод `getGreeting()`, соответствующее событие, для которого мы регистрируем наш логгер, будет инициировано.

Если мы используем `SharedServiceManager`, можно еще больше упростить `GreetingServiceFactory`:

```
1 <?php
2 namespace Helloworld\Service;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class GreetingServiceFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $serviceLocator
12            ->get('sharedEventManager')
13            ->attach(
14                'GreetingService',
15                'getGreeting',
16                function($e) use($serviceLocator) {
17                    $serviceLocator
18                        ->get('loggingService')
19                        ->onGetGreeting($e);
20                }
21            );
22
23        $greetingService = new GreetingService();
24        return $greetingService;
25    }
26 }
```

Listing 7.16

Тем не менее, нужно обеспечить, чтобы в этом показательном примере, непосредственно в сервисе перед запуском события, соответствующий `EventManager` почувствовал ответственность за этот "идентификатор". Это можно сделать, используя метод `addIdentifiers()`:

```
1 <?php
2 namespace Helloworld\Service;
3
4 use Zend\EventManager\EventManagerAwareInterface;
5 use Zend\EventManager\EventManagerInterface;
6 use Zend\EventManager\Event;
7
8 class GreetingService implements EventManagerAwareInterface
```

```

9 {
10     private $eventManager;
11
12     public function getGreeting()
13     {
14         $this->eventManager->addIdentifiers('GreetingService');
15         $this->eventManager->trigger('getGreeting');
16
17         if(date("H") <= 11) {
18             return "Good morning, world!";
19         } else if (date("H") > 11 && date("H") < 17) {
20             return "Hello, world!";
21         } else {
22             return "Good evening, world!";
23         }
24     }
25
26     public function getEventManager()
27     {
28         return $this->eventManager;
29     }
30
31     public function setEventManager(EventManagerInterface $em)
32     {
33         $this->eventManager = $em;
34     }
35 }

```

Listing 7.17

Объект события

Инициатор события и слушатель события взаимодействуют через объект события, который создается инициатором события и становится доступным для слушателя при вызове. Когда событие инициируется при помощи его имени

```

1 <?php
2 $this->eventManager->trigger('event1');

```

Listing 7.18

слушателю передается объект типа `Zend\EventManager\Event`, который генерируется автоматически и не предоставляет собой ничего более, чем внутреннюю структуру данных для общей передачи параметров, а также информацию о так называемой "цели", т.е. о месте, где само событие было инициировано и об обозначении самого события. Если вы хотите сделать определенные данные доступными для слушателей, это может быть сделано при помощи параметров:

```

1 <?php
2 $this->eventManager
3     ->trigger('event1', $this, array("key" => "value"));

```

Listing 7.19

В слушателе можно получить доступ к структуре данных с помощью метода `getParams()`. Таким образом, фактически все, что может понадобиться, может быть достигнуто с помощью `Zend\EventManager\Event`. Однако, если вы хотите работать со специфическими обозначениями и переменными-членами, доступ к которым можно получить с помощью геттеров и сеттеров, вместо обобщенной структуры данных, у вас есть возможность, как у разработчика приложения, декларировать свой собственный класс событий, экземпляр которого может быть создан и заполнен "на лету" при необходимости. Zend Framework 2 сам активно использует этот механизм, действительно, существуют `ModuleEvent`, `ViewEvent` и, наконец, `MvcEvent`, который, например, обеспечивает прямой доступ к следующим объектам:

```
1 <?php
2 protected $application;
3 protected $request;
4 protected $response;
5 protected $result;
6 protected $router;
7 protected $routeMatch;
8
9 // [...]
```

Listing 7.20

Лучше всего наследовать собственный класс событий от `Zend\EventManager\Event`:

```
1 <?php
2 namespace HelloWorld\Event;
3
4 use Zend\EventManager\Event;
5
6 class MyEvent extends Event
7 {
8     private $myObject;
9
10    public function setMyObject($myObject)
11    {
12        $this->myObject = $myObject;
13    }
14
15    public function getMyObject()
16    {
17        return $this->myObject;
18    }
19 }
```

Listing 7.21

Для регистрации класса события можно либо предварительно сделать эту информацию известной, и затем вызвать событие с помощью имени события

```
1 <?php
2 $this->eventManager->setEventClass('Helloworld\Event\MyEvent');
3 $this->eventManager->trigger('getGreeting');
```

Listing 7.22

или передать соответствующий объект в `trigger()`:

```
1 <?php
2 $event = new \Helloworld\Event\MyEvent();
3 $event->setName('getGreeting');
4 $this->eventManager->trigger($event);
```

Listing 7.23

Таким образом можно создавать совершенно разные классы событий с предопределенными именами:

```
1 <?php
2 namespace Helloworld\Event;
3
4 use Zend\EventManager\Event;
5
6 class MyGetGreetingEvent extends Event
7 {
8     private $myObject;
9
10    public function __construct()
11    {
12        parent::__construct();
13        $this->setName('getGreeting'); // предопределенное имя события
14    }
15
16    public function setMyObject($myObject)
17    {
18        $this->myObject = $myObject;
19    }
20
21    public function getMyObject()
22    {
23        return $this->myObject;
24    }
25 }
```

Listing 7.24

Запускающий механизм этого события будет менее подвержен ошибкам, а код более компактным:

```
1 <?php
2 $event = new \Helloworld\Event\MyGetGreetingEvent();
3 $this->eventManager->trigger($event);
```

Listing 7.25

VIII. Модули

Наряду с EventManager, концепция модулей играет решающую роль в Zend Framework 2. В обработке запросов этим фреймворком ModuleManager важен потому, что он действительно обеспечивает, чтобы активированный модуль всегда учитывался и загружался, т.е. функциональность приложения.

Важно то, что модули приложения не образуют "закрытые сущности". Напротив, функциональность отдельных модулей объединяется в рамках загрузки модуля, чтобы сформировать "общую функциональность" приложения. Мы видели, что ModuleManager объединяет конфигурации всех модулей в один объект конфигурации, общий для приложения. Из этого факта вытекает несколько следствий, которые мы, как разработчики приложений, должны понимать. Все сервисы, которые делают модуль доступным, делают его доступным для всего приложения, т.е., например, "плагины контроллеров" или "помощники представления", а также, например, контроллеры одного модуля – всегда доступны для других модулей, нет отдельного ControllerLoader для каждого модуля.

Модуль "Application"

Если мы начинаем разрабатывать собственное приложение на основе ZendSkeletonApplication, то мы сталкиваемся с модулем Application практически сразу же. Если мы знаем о вышеупомянутом факте, что функциональность всех модулей формирует общую функциональность приложения, и если мы рискнем разобраться, что доступно в этом модуле, мы быстро узнаем цель этого "стандартного модуля". Он выполняет конфигурацию целого ряда сервисов и генерирует несколько основных функций, которые должно иметь каждое приложение и собственная реализация которых не должна выполняться. Помимо того, что ZendSkeletonApplication только обеспечивает характерный контроллер для "главной страницы" приложения с его конфигурацией маршрутов, ряд настроек для "слоя представления" содержатся в module.config.php модуля Application. Их считывают затем ViewManagerFactory или ViewManager, и последние либо используют их сами, либо передают другим объектам:

```
1 <?php
2 // [...]
3 'view_manager' => array(
4     'display_not_found_reason' => true,
5     'display_exceptions' => true,
6     'doctype' => 'HTML5',
7     'not_found_template' => 'error/404',
8     'exception_template' => 'error/index',
9     'template_map' => array(
10         'layout/layout'
11             => __DIR__ . '/../view/layout/layout.phtml',
12         'application/index/index'
13             => __DIR__ . '/../view/application/index/index.phtml',
```



```

14     'error/404'
15     => __DIR__ . '/../view/error/404.phtml',
16     'error/index'
17     => __DIR__ . '/../view/error/index.phtml',
18 )
19 )

```

Listing 8.1

- С помощью `display_not_found_reason`, стратегии `RouteNotFoundStrategy` (стандартный способ обработки ошибки 404) указывается сделать причину (`reason`) того, что вызов URL привел к 404 ошибке, доступной для дальнейшего представления. Как мы видели в предыдущих главах, каждый результат обработки запроса основывается на "модели представления", которая содержит как пользовательские данные, так и (HTML) шаблон, который должен использоваться для этой цели. Однако когда возникает ошибка 404, нет вообще никакой модели представления, сформированной контроллером, просто потому, что не было никакого ответственного контроллера. `RouteNotFoundStrategy` обеспечивает, чтобы модель представления создавалась таким образом, чтобы процесс, описанный выше, смог в любое время осуществиться. `ZendSkeletonApplication` также предоставляет соответствующий шаблон в `view/error/404.phtml`. Доступ к причине проблемы может быть получен через `$this->reason`.
- `not_found_template` явно определяет шаблон, который будет использоваться в ситуациях с 404 ошибкой. Обычно фреймворк ищет шаблон `404.phtml`. Поэтому будет достаточно создать соответствующий шаблон `404.phtml` в каталоге `view` одного из модулей. Кстати, если мы создадим файл `404.phtml` более чем в одном модуле, будет использован самый последний активированный модуль. Как вы помните, конфигурация отдельных модулей объединяется при загрузке `ModuleManager`. Шаблон `error/404` для ситуаций с ошибкой 404 определяется в `ZendSkeletonApplication`, который, в свою очередь, указывает на физический файл с помощью следующей конфигурации `template_map`:

```

1 'error/404' => __DIR__ . '/../view/error/404.phtml'

```

Listing 8.2

- Функциональность `display_exceptions` аналогична `display_not_found_reason`. В этом случае стратегии `ExceptionStrategy` аналогично указывается сделать доступным `$this->display_exceptions` в соответствующей модели представления и, следовательно, в определенном шаблоне. Таким образом, здесь может быть принято решение о том, должны ли быть представлены дальнейшие детали исключения, или нет.
- С помощью `exception_template` явно указывается шаблон, который реализуется в исключительной ситуации, т.е. всегда, когда где-то в процессе возникает исключение, которое не замечено и не обработано разработчиком приложения. Обычно фреймворк ищет шаблон `error.phtml`. Поэтому будет достаточно создать

соответствующий шаблон `error.phtml` в каталоге `view` одного из модулей. Шаблон `error/index` для исключительных ситуаций определяется в `ZendSkeletonApplication`, который, в свою очередь, указывает на физический файл с помощью следующей конфигурации `template_map`:

```
1 'error/index' => __DIR__ . '/../view/error/index.phtml'
```

Listing 8.3

- Помощник представления `"doctype"`, который мы рассмотрим более подробно позже, конфигурируется с помощью `doctype`. Декларация типа документа может быть выведена в шаблоне (`template`) или в макете (`layout`) (подробнее об этом также позже) с помощью помощника представления `"doctype"` без необходимости добавлять его вручную. Сохраненные значения используются фабрикой `ViewHelperManagerFactory`, чтобы обеспечить помощника представления `"doctype"` надлежащей конфигурацией.
- Шаблон (`template`), который функционирует, как `"визуальный каркас"`, определяется с помощью ключа `layout`. Обычно фреймворк ожидает шаблон макета под ключом `layout/layout` в `template_map` или в соответствующем файле в файловой системе. При необходимости, другой шаблон может быть объявлен под ключом `layout`.

```
1 'layout' => 'myLayout'
```

Listing 8.4

В модуле `Application`, таким образом, нет ничего особенного, но он действительно снижает первоначальные усилия по конфигурации для разработчиков приложений. Я рекомендую сознательное дальнейшее программирование модуля `Application` и сохранение здесь всех определений и конфигураций, которые нужны всем или большинству модулей. Теоретически, можно разработать приложение таким образом, что оно не будет содержать вообще никаких модулей, кроме `Application`, и вся функциональность которого будет реализована непосредственно в нем. В некоторых случаях это, конечно, эффективная процедура.

Поведение, зависимое от модуля

Факт в том, что после того, как модули будут загружены, в работающем приложении нет больше модулей, но вместо этого есть само приложение со всеми его функциями и конфигурациями, которые были созданы путем объединения отдельных модулей. Следовательно, в основном невозможно настроить поведение, зависимое от модуля, в более позднее время, поскольку нет никакой доступной информации о текущем активном модуле. Тем не менее, если мы хотим выполнить определенные действия, например, когда выполняется контроллер определенного модуля, приходится прибегать к `SharedEventManager` и трюку:

```
1 <?php
2 namespace HelloWorld;
```

```

3
4 use Zend\ModuleManager\ModuleManager;
5
6 class Module
7 {
8     public function init(ModuleManager $moduleManager)
9     {
10         $sharedEvents = $moduleManager->getEventManager()
11             ->getSharedManager();
12
13         $sharedEvents->attach(
14             __NAMESPACE__,
15             'dispatch',
16             function($e) {
17                 $controller = $e->getTarget();
18                 $controller->layout('layout/helloWorldLayout');
19             },
20             100
21         );
22     }
23 }

```

Listing 8.5

В этом примере мы настроим другой макет для всех страниц, которые создаются с помощью контроллера действий нашего модуля `HelloWorld`. Для этого мы присоединяем функцию обратного вызова для обработки события, но только для тех контроллеров, которые ответственны за "идентификатор" `HelloWorld` (в данном случае это соответствует значению магической константы `__NAMESPACE__`), т.е. для всех контроллеров модуля `HelloWorld`. Как именно это работает? В этом случае у нас есть частная ситуация, в которой мы хотим зарегистрировать слушателя (в данном случае в виде функции обратного вызова) с помощью менеджера `EventManager`, который еще не существует на момент регистрации. Действительно, контроллеру назначается только свой собственный менеджер событий во время его генерации.

Это значит, что мы должны использовать `SharedEventManager` (см. также предыдущую главу). А вот в чем трюк: контроллер в `Zend Framework 2` всегда сконфигурирован таким образом, что когда он является дочерним от `AbstractActionController`, он обращается к `SharedEventManager`, когда его слушатель уведомлен о событии; среди прочего он объявляет пространство имен контроллера (т.е. в данном случае `HelloWorld`). И мы реализовали обратный вызов в `init()` описанным выше способом только для этого, первый теперь вызывается, получает определенный контроллер с помощью объекта `MvcEvent`, а затем устанавливает другой макет шаблона с помощью плагина контроллера. Сам шаблон, естественно, должен существовать, или он должен быть дополнительно сделан доступным с помощью карты шаблонов (`template_map`). В противном случае это приведет к ошибке.

Скажем по-другому, если мы регистрируем слушателей для события с идентификатором, который соответствует пространству имен модуля, они принимаются во внимание контроллером соответствующего модуля.

Установка сторонних модулей

Одним из главных достижений 2 версии является тот факт, что собственное приложение можно расширить простым способом, без необходимости программировать его самостоятельно. Для установки сторонних модулей обязательны несколько фундаментальных оперативных шагов, и, в зависимости от модуля, могут потребоваться несколько манипуляций.

Источники сторонних модулей

Двумя хорошими источниками для получения модулей являются официальная страница ZF модулей (<http://modules.zendframework.com>) и GitHub (<https://github.com>). В этом контексте следует особо отметить ZF-Commons-Repository (<https://github.com/ZF-Commons>). Дополнительные репозитории можно найти с помощью функции поиска на GitHub. Краткая заметка на данный момент: не у всех доступных модулей качество, которое вы устанавливаете, как минимальный стандарт для собственного кода. Многие модули, особенно с очень маленькими номерами версий, по-прежнему содержат многочисленные ошибки и бреши в защите. Поскольку функции и конфигурации всех модулей объединяются с помощью ModuleManager в рамках загрузки модулей, ранее зарегистрированные сервисы и т.д. могут внезапно стать недоступными, например, потому что они были неумышленно перезаписаны другими реализациями. Поэтому важно использовать сторонние модули с осторожностью и принимать здравые решения за или против использования определенного модуля.

Установка сторонних модулей

Существует масса вариантов для предоставления дополнительных модулей для собственного приложения. Самый простой способ – это вручную загрузить коды модуля и скопировать исходные файлы в свое приложение. Таким образом, например, можно загрузить модуль ZfcTwig (<https://github.com/ZF-Commons/ZfcTwig/tarball/master>) и скопировать содержимое архива в папку модуля своего приложения (например, в каталог ZfcTwig). Если вы помните, модуль активируется в application.config.php (добавлением значения "ZfcTwig" в разделе modules), и модуль в основном готов к работе. Однако, ZfcTwig это модуль, для работы которого, кроме того, требуется библиотека PHP "Twig" (<http://twig.sensiolabs.org>). Twig является шаблонизатором от создателей фреймворка Symfony, одного из конкурентов Zend Framework. В то же время Twig слегка опередил менее перспективный Smarty и уже стал популярен. Кстати, можно задаться вопросом, зачем нужен еще один язык шаблонов, вроде Twig, вообще для программирования шаблонов на PHP, тем более что PHP сам по себе является языком шаблонов. В конце концов, типичные Zend Framework "phtml" файлы также состоят из HTML, PHP кода и нескольких помощников представления при необходимости, и Zend Framework 2, как и

версия 1, не содержит других шаблонизаторов. Все эти вопросы совершенно правильны и весьма справедливы. Краткий ответ на эти вопросы: "Нет, нам действительно не нужны никакие дополнительные системы шаблонов, синтаксис которых нужно дополнительно изучать и ограничения которого нам известны". Все и без этого хорошо. Однако, есть прикладные ситуации, в которых дополнительный механизм шаблонов может быть очень полезен, если мы хотим предотвратить "недобросовестную работу", выполняемую в шаблонах. Шаблон на чистом PHP содержит все возможности PHP, например, доступ ко всем функциям ядра языка. Если мы хотим предотвратить это и сознательно ограничить использование в шаблонах определенной функциональности, шаблонный движок может быть полезен. И, кроме того, "синтаксический сахар" (http://ru.wikipedia.org/wiki/Синтаксический_сахар) – другая основная причина его использования.

Но давайте вернемся к актуальной проблеме зависимости модуля ZfcTwig от библиотеки Twig. Модуль ZfcTwig таким образом, действует, как "связующий код" и гарантирует, что функциональность Twig может использоваться в контексте приложения на Zend Framework. Таким образом, нам нужно скачать Twig, затем разместить его в каталоге vendor и сконфигурировать автозагрузку Twig в надлежащих местах. Тогда это будет работать.

Поскольку это трудоемкий и подверженный ошибкам процесс, целесообразно использовать composer для установки сторонних модулей по мере возможности, как мы это уже делали при установке ZendSkeletonApplication. Если мы поступим таким образом, composer не только выполнит загрузку модуля ZfcTwig и сделает его доступным, но и займется его зависимостями, т.е. библиотекой Twig. Для этого мы расширяем composer.json нашего приложения в разделе require:

```
1 "require": {  
2     "zf-commons/zfc-twig": "dev-master"  
3 }
```

Listing 8.6

и запускаем composer снова в каталоге нашего проекта

```
1 $ php composer.phar update
```

Теперь composer загрузит необходимые библиотеки, а также настроит автозагрузку. Однако, если в настоящее время мы посмотрим в каталог module нашего приложения, мы обнаружим, что модуль туда не был добавлен. Вместо этого composer стандартно сохраняет все загружаемые им библиотеки в каталог vendor. В application.config.php следующие разделы контролируют маршруты, по которым приложение ожидает установленные модули:

```
1 <?php  
2 // [...]  
3 'module_paths' => array(  
4     'zf-commons/zfc-twig' => 'vendor/zf-commons/zfc-twig'5 )
```

```
4     './module',  
5     './vendor',  
6 ),
```

Listing 8.7

Таким образом, `vendor` – это совершенно нормально, и в этом месте даже могут быть настроены совершенно разные пути.

Если мы теперь откроем `/sayhello`, мы увидим... мешанину символов. То есть, шаблоны, которые мы все еще используем в настоящее время в нашем приложении `Helloworld`, основаны на PHP и потому несовместимы с Twig. Twig действительно требует наличия Twig-совместимой разметки шаблона. К счастью, `ZfcTwig` предоставляет альтернативные Twig-совместимые шаблоны для типовых страниц `ZendSkeletonApplication`, которые можно использовать вместо "нормальных" шаблонов модуля приложения. После установки `ZfcTwig` с помощью `composer` их можно найти в `vendor/zf-commons/zfc-twig/examples`. Мы перезапишем текущие шаблоны в `module/application/view` этими файлами. Поступив таким образом, мы уже сделали макет и шаблоны для страниц ошибок Twig-совместимыми. Теперь то же самое нужно сделать с шаблонами, которыми мы написали сами. Тогда можно будет вызвать `/sayhello` как обычно. Шаблоны теперь обрабатываются с помощью Twig.

Настройка стороннего модуля

Структура других модулей более похожа на структуру MVC - модулей; она предоставляет контроллеры, представления, маршруты и т.д. Хорошим примером такого модуля является `ZfcUser`, который отображает функции регистрации пользователей, включая "Log-in/Log-out". Мы снова рассмотрим этот модуль более детально позже.

Для простоты давайте представим себе на минуту, что модуль `Helloworld` в последней главе был сторонним модулем, который мы установили с помощью `composer`. С помощью модуля `Helloworld`, URL `/sayhello` был бы сейчас, таким образом, зарегистрирован в системе. Но что нам делать, если в действительности мы хотим предоставить доступ к соответствующей странице под URL `/welcome?`

Естественно, мы могли бы сейчас открыть `module.config.php` модуля и внести непосредственно туда наши изменения. Но это будет означать, что мы "разветвляем" базу кода `Helloworld`, и улучшения, которые были сделаны в `Helloworld` оригинальным автором (мы притворяемся, что это сторонний модуль), не смогут быть импортированы в нашу систему так легко. Мы бы перезаписывали наши изменения каждый раз при обновлении, и пришлось бы вручную после каждого обновления изменять данные. Это не очень оптимально.

Вместо этого мы будем вносить изменения в наш собственный модуль, и таким образом отделим его от кода `Helloworld`. Таким образом, `Helloworld` сохраняет свою

первоначальную структуру и может быть обновлен без риска того, что наши изменения будут потеряны.

Изменение URL

Для того, чтобы изменить URL `/sayhello` на `/welcome`, мы создадим новый подкаталог и, тем самым, новый модуль в каталоге `module`, и назовем его `HelloWorldMod`. Так мы делаем понятным, что это – наша модификация другого модуля. Здесь нам нужен только `Module.php`, в методе `getConfig()` которого мы берем определение маршрута для `sayhello` и перезаписываем его на `/welcome`.

```
1 <?php
2 namespace HelloWorldMod;
3
4 class Module
5 {
6     public function getConfig()
7     {
8         return array (
9             'router' => array(
10                 'routes' => array(
11                     'sayhello' => array(
12                         'type' => 'Zend\Mvc\Router\Http\Literal',
13                         'options' => array(
14                             'route' => '/welcome'
15                         )
16                     )
17                 )
18             )
19         );
20     }
21
22 // [...]
23 }
```

Listing 8.8

Теперь мы должны убедиться, что модуль будет активирован, и для этого мы расширяем `application.config.php` в разделе `modules` соответствующим образом.

```
1 <?php
2 return array(
3     'modules' => array(
4         'Application',
5         'HelloWorld',
6         'HelloWorldMod',
7     ),
8     'module_listener_options' => array(
9         'config_glob_paths' => array(
```

```

10         'config/autoload/{,*.}{global,local}.php',
11     ),
12     'module_paths' => array(
13         './module',
14         './vendor',
15     ),
16 ),
17 );

```

Listing 8.9

Когда мы теперь ссылаемся на `/sayhello`, мы получаем, как и ожидалось, ошибку 404. В противоположность этому, `/welcome` выдает теперь привычную страницу. В этом контексте мы используем два механизма. Во-первых, конфигурация всех модулей объединена в рамках их процесса загрузки таким образом, что мы можем сделать как бы конфигурацию "для других модулей" в одном из модулей. Во-вторых, конфигурация модулей индивидуальна и считывается последовательно. Так, в нашем случае, они считываются следующим образом: сначала `Application`, затем `Helloworld`, и, наконец, конфигурация `HelloworldMod`. Таким образом, конфигурация предыдущего модуля может быть перезаписана нами конфигурацией следующего, как мы это сделали для маршрута с обозначением `sayhello`, который первоначально был определен в `Helloworld`, а затем его перенастроил в своих параметрах маршрута `HelloworldMod` на `/welcome`.

Изменение представления

Презентация сторонних модулей также может быть адаптирована. Как правило, это важнее и требуется чаще, чем адаптация URL. Чтобы добиться этого, класс `Module` модуля `HelloworldMod` расширяется в разделе `view_manager`, и там шаблон для действия `index` контроллера `index` модуля `Helloworld` указывает на соответствующим образом измененный шаблон в модуле `HelloworldMod`.

```

1 <?php
2 namespace HelloworldMod;
3
4 class Module
5 {
6     public function getConfig()
7     {
8         return array (
9             'router' => array(
10                 'routes' => array(
11                     'sayhello' => array(
12                         'type' => 'Zend\Mvc\Router\Http\Literal',
13                         'options' => array(
14                             'route' => '/welcome'
15                         )
16                     )
17                 )
18             )
19         );
20     }
21 }

```



```

18         ),
19         'view_manager' => array(
20             'template_map' => array(
21                 'helloworld/index/index'
22                 => __DIR__ . '/view/helloworld-mod/index/index.phtml'
23             )
24         )
25     );
26 }
27 }

```

Listing 8.10

Если мы теперь снова вызовем `/sayhello`, мы увидим страницу, отображаемую с использованием нового шаблона.

IX. Контроллер

Концепция и режим работы

Задача контроллера состоит в обработке взаимодействия с пользовательским интерфейсом, в нашем случае, таким образом, с веб-сайтом или в более широком смысле, с самим браузером.

Шаблон MVC ("C" здесь означает "Controller", контроллер), кстати, уже довольно старый. Впервые он был использован в конце 1970-х годов для реализации пользовательских интерфейсов, когда язык программирования Smalltalk все еще был популярен. Тем не менее, в то время интерфейсы пользователя, также как и обработка взаимодействия, обычно были ограничены замкнутой системой. В web все не так. Здесь пользовательский интерфейс проявляется в браузере, то есть на стороне клиента, в то время, как обработка происходит на сервере. Браузер передает информацию о том, что происходит в нем или на веб-сайте, который он отображает, и как пользователь взаимодействует с ним. Если пользователь нажимает ссылку, браузер передает тот факт, что пользователь только что запросил другую веб-страницу на сервере. На сервере теперь фреймворк определяет, какой контроллер в системе предусмотрен для обработки этого взаимодействия (маршрутизация), и возлагает на него дальнейшую ответственность за генерацию ответа. Потом он обеспечивает, чтобы представление изменилось в браузере пользователя. Либо загружается совершенно новая страница, либо, если используется AJAX, будет обновлена только определенная часть уже отображаемой страницы.

Как правило, мы создаем для каждого "типа страницы" в системе свой контроллер. Например, в интернет-магазине был бы `IndexController` для главной страницы, `CategoryController` для представления списка предлагаемых товаров определенной категории и `ItemController` для представления страницы с детальной информацией о конкретном предлагаемом товаре. В Zend Framework контроллер дополнительно подразделяется на так называемые "действия". Фактическая обработка таким образом выполняется не самим контроллером, а его действиями. В этом контексте действия являются публичными методами класса контроллера. Помимо вышеупомянутых контроллеров, интернет-магазин обычно предоставляет корзину, поэтому в системе, скорее всего, будет `CartController` (контроллер корзины). Чтобы предоставить нормальное взаимодействие с корзиной, "`CartController`" скорее всего будет оснащаться рядом действий, среди них, например, "`show action`" (показать содержимое корзины), "`add action`" (добавить в корзину), "`remove action`" (удалить из корзины), "`remove all action`" (удалить всё), и т.д.

Соответствующая процедура, выполняемая в коде самих контроллеров или действий, должна быть как можно меньшей ("тонкой"). В идеальном случае, контроллер должен только зарегистрировать и оценить действия пользователя и затем решить, какой рычаг тянуть, чтобы получить желаемый результат. Как правило, для этого он использует

сервисы, которые позволяют получить доступ к базам данных, сессиям или другой информации и функциональности.

Плагины контроллеров

Вместе с тем, контроллеры часто используют так называемые "плагины контроллеров". Они инкапсулируют поведенческо-зависимый код, который часто необходим, и могут использоваться различными контроллерами.

Фреймворк поставляется с большим количеством плагинов, которые могут быть непосредственно использованы в ваших собственных контроллерах. Все плагины наследуют один и тот же основной класс, который обеспечивает, чтобы соответствующий плагин всегда имел доступ к контроллеру, в котором он сейчас используется. Это практично во многих ситуациях, а в некоторых из них это просто необходимо, как мы увидим далее.

Переадресация (Redirect)

Плагин `redirect` позволяет пересылать клиента на другой адрес: в этом контексте данное перенаправление выполняется не "внутренне фреймворком", а, скорее, с помощью обратной связи с клиентом, который затем активно вызывает новый URL. Для этого фреймворк посылает ответ со статусом 302 HTTP кода, и клиент, в большинстве случаев – браузер пользователя, отправляет новый запрос на указанный URL (http://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP).

В этой ситуации URL может быть введен непосредственно.

```
1 <?php
2 $this->redirect()->toUrl('http://www.meinedomain.de/zielseite');
```

Listing 9.1

Или URL могут быть созданы на основе маршрутов, доступных в системе. Таким образом, если мы хотим выполнить перенаправление на URL `http://localhost:8080/sayhello`, мы можем в качестве альтернативы позволить создать URL с помощью конфигурации маршрутов:

```
1 <?php
2 $this->redirect()->toRoute('sayhello');
```

Listing 9.2

В данном случае `sayhello` – это имя маршрута, который мы определили в нашем конфигурационном файле модуля, `module.config.php`:

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'sayhello' => array(
```

```

6         'type' => 'Zend\Mvc\Router\Http\Literal',
7         'options' => array(
8             'route' => '/sayhello',
9             'defaults' => array(
10                'controller' => 'Helloworld\Controller\Index',
11                'action' => 'index',
12            )
13        )
14    )
15 )
16 )

```

Listing 9.3

Но что произойдет, если URL является не статической строкой, как в нашем примере, а динамически компилируется, как, например, `http://www.zalando.de/nike-performance-free-run-3-laufschuh-black-relfecting-silver-platinum`. В этом случае также существует решение: в дополнение к названию маршрута динамические компоненты, которые затем объединяются в виде URL, могут быть переданы в качестве дополнительных параметров. В следующей главе мы снова будем исследовать механизм маршрутизации в деталях и, к тому же, снова повторно рассмотрим содержимое плагина `redirect`. Тогда мы увидим, как именно целевой URL создается из динамических компонентов.

Плагин `redirect` использует преимущества того факта, что контроллер возвращает объект типа `Response`, в противном случае обычная дальнейшая обработка пропускается. Соответственно, дальнейшая визуализация представления в этом случае не происходит.

Для того, чтобы плагин `redirect` функционировал должным образом, необходимо, чтобы контроллер, в котором он используется, содержал объект `MvcEvent`, поскольку маршрутизатор (`Router`), который требуется для генерации URL на основании имени маршрута, получается из него. Обычно, мы всегда получаем наш собственный контроллер, наследуя базовый класс фреймворка `AbstractActionController`, в котором большое количество контроллерно-зависимых интерфейсов уже реализовано и, соответственно, гарантирует, например, что `MvcEvent` будет доступен:

```

1 <?php
2 namespace Zend\Mvc\Controller;
3
4 // use [...];
5
6 abstract class AbstractActionController implements
7     Dispatchable,
8     EventManagerAwareInterface,
9     InjectApplicationEventInterface,
10    ServiceLocatorAwareInterface
11 {
12     // [...]
13 }

```

Listing 9.4

AbstractActionController фреймворка реализует интерфейс InjectApplicationEventInterface и таким образом позволяет сервису ControllerLoader, который отвечает за создание экземпляра и вызов контроллера, соответствующего маршруту после маршрутизации, инжектировать объект MvcEvent.

Кстати, реализация интерфейса EventManagerAwareInterface гарантирует, что EventManager будет доступен контроллеру, а реализация ServiceLocatorAwareInterface гарантирует, что контроллер сможет получить доступ к ServiceManager и, следовательно, также к сервисам приложения.

PostRedirectGet

Этот плагин обеспечивает решение проблемы специальных перенаправлений для форм, отправленных методом POST. Если пользователь инициирует перезагрузку страницы после того, как предварительно отправил форму на страницу подтверждения, данные POST снова передаются на сервер в большинстве случаев, и в нежелательных случаях происходят двойные покупки, заказы и т.д. Чтобы избежать этой проблемы, страница подтверждения, которая отображается после POST транзакции, не должна создаваться на той же стадии. Вместо этого в ответ на успешный запрос POST, сервер сначала посылает 301/302 код состояния HTTP, а затем пересылает на страницу подтверждения. Последняя запрашивается затем браузером методом GET и таким образом может быть вызвана несколько раз без риска. Плагин PostRedirectGet служит для того, чтобы не нужно было программировать этот механизм самостоятельно.

<http://ru.wikipedia.org/wiki/Post/Redirect/Get>

Перенаправление (Forward)

Плагин redirect реализует переадресацию на стороне клиента с помощью соответствующего HTTP заголовка, в то время, как плагин forward позволяет внутри контроллера выполнить вызов другого контроллера. На самом деле, название "перенаправление" не совсем правильное. Ведь плагин никуда никого не перенаправляет, вместо этого он "выполняет" другой контроллер. Если снова внимательно рассмотреть контроллеры Zend Framework 2, становится ясно следующее: контроллер в действительности не более, чем класс, в распоряжении которого есть метод dispatch(), который ожидает объект типа Request и объект типа Response и таким образом выполняет контракт DispatchableInterface. Что делает соответствующий контроллер с объектом Request, чтобы вернуть надлежащий ответ (Response), остается за самим контроллером.

Соответственно, с плагином forward выполняется только метод контроллера dispatch(), и он возвращает, как это обычно бывает, когда контроллер выполнен, объект типа ViewModel в качестве результата. Действительно, то, что сейчас происходит с результатом, остается на усмотрение контроллера. Если мы хотим имитировать "перенаправление контроллера", следующая строка показывает, как это сделать:

```

1 <?php
2 return $this->forward()->dispatch('Helloworld\Controller\Other');

```

Listing 9.5

В этом случае "Other" аллегорически представляет собой еще один контроллер, который к тому же был ранее известен системе в рамках конфигурации модуля. В этом случае ViewModel, которую сгенерировал OtherController, возвращается один к одному первоначально вызвавшему контроллеру, который больше не генерирует ViewModel самостоятельно.

Однако плагин forward может также использоваться, чтобы агрегировать результаты работы нескольких других контроллеров. Но мы обсудим это позже в разделе "концепция и режим работы" главы "представление".

Кстати, если мы хотим вызвать определенное действие другого контроллера, это можно сделать следующим образом:

```

1 <?php
2 return $this->forward()
3     ->dispatch(
4         'Helloworld\Controller\Other',
5         array('action' => 'test')
6     );

```

Listing 9.6

Адрес (URL)

С помощью плагина URL может быть сгенерирован адрес URL на базе ранее определенного маршрута и в соответствии с декларацией маршрута назначения, а также, при необходимости, с дополнительными параметрами:

```

1 <?php
2 $url = $this->url()->fromRoute('routedesignation', $params);

```

Listing 9.7

Параметры (Params)

Обеспечивает простой доступ к параметрам запроса, к которым иначе было бы трудно получить доступ. На самом деле к параметрам POST в контроллере можно получить доступ следующим образом:

```

1 <?php
2 $this->getRequest()->getPost($param, $default);

```

Listing 9.8

С помощью плагина доступ может быть выполнен более элегантно, даже если процедура не намного короче:

```

1 <?php
2 $this->params()->fromPost('param', $default);

```

Listing 9.9

Если опустить обозначение `param`, все параметры возвращаются в виде ассоциативного массива:

```
1 <?php
2 $this->params()->fromPost();
```

Listing 9.10

Главное – указать правильный источник. Следующие вызовы возможны:

```
1 <?php
2 $this->params()->fromPost ($param, $default); // получить параметр из POST
3 $this->params()->fromQuery ($param, $default); // получить параметр из GET
4 $this->params()->fromRoute ($param, $default); // получить параметр из Route
5 $this->params()->fromFiles ($name, $default); // получить файл по имени
6 $this->params()->fromHeader($header, $default); // получить параметр из заголовка
```

Listing 9.11

Макет (Layout)

С помощью плагина `layout` макет для использования может быть сконфигурирован в любое время:

```
1 <?php
2 $this->layout('layout/my-layout');
```

Listing 9.12

Плагин `layout` полезен, например, если мы хотим использовать альтернативный макет в рамках некоторого действия (action).

Flash messenger

Сообщения для пользователя могут переноситься при переходе между страницами с помощью `Flash messenger`. С технической точки зрения для этого на стороне сервера генерируется сессия и в ней кратковременно сохраняется соответствующее сообщение. Сообщения для пользователя могут быть добавлены во `Flash messenger` следующим образом:

```
1 <?php
2 $this->flashMessenger()->addMessage('Запись была удалена.');
```

Контроллер целевой страницы может получать и отображать сообщение (или даже несколько) следующим образом:

```
1 <?php
2 $result = array('success' => true);
3 $flashMessenger = $this->flashMessenger();
4 if ($flashMessenger->hasMessages()) {
```

```

5     $result['messages'] = $flashMessenger->getMessages();
6 }
7 return $result;

```

Listing 9.13

Создание собственного плагина контроллеров

В плагинах контроллеров Zend Framework 2 на самом деле нет ничего особенного. Действительно, это всего лишь более или менее "обычный" класс, который фактически предполагает только методы `getController()` и `setController()`; и в соответствующий плагин контроллера, который в настоящее время применяется, инжецируется экземпляр контроллера. Однако реализации этих методов можно избежать, если мы наследуем наш собственный плагин от класса `Zend\Mvc\Controller\Plugin\AbstractPlugin`. Затем нужно просто вдохнуть жизнь в метод `__invoke()`, чтобы можно было легко использовать плагин в контроллере.

```

1 <?php
2
3 namespace Helloworld\Controller\Plugin;
4
5 use Zend\Mvc\Controller\Plugin\AbstractPlugin;
6
7 class CurrentDate extends AbstractPlugin
8 {
9     public function __invoke()
10    {
11        return date('d.m.Y');
12    }
13 }

```

Listing 9.14

Этот плагин контроллера обеспечивает генерацию текущей даты (правда, в реальности нам не нужен плагин контроллера, чтобы это сделать). Определение класса находится в модуле **Helloworld** в `src/Helloworld/Controller/Plugin/CurrentDate.php`. Действие `index` контроллера `IndexController` адаптировано таким образом, что оно использует `CurrentDate` и обеспечивает просмотр результатов его работы:

```

1 <?php
2 // [...]
3 public function indexAction()
4 {
5     return new ViewModel(
6         array(
7             'greeting' => $this->greetingService->getGreeting(),
8             'date' => $this->currentDate()

```



```

9         )
10    );
11 }

```

Listing 9.15

Плагин `CurrentDate` может быть легко вызван, как если бы это был метод текущего контроллера (через `$this`). Однако для вызова этой функции мы должны сделать плагин `CurrentDate` известным менеджеру `ControllerPluginManager` заранее. Мы можем сделать это либо в `module.config.php`, либо в классе `Module`:

```

1 <?php
2 // [...]
3 public function getControllerPluginConfig()
4 {
5     return array(
6         'invokables' => array(
7             'currentDate'
8                 => 'Helloworld\Controller\Plugin\CurrentDate'
9         )
10    );
11 }

```

Listing 9.16

После регистрации этот плагин может быть вызван во всех контроллерах. Кстати, `CurrentDate` не ограничивается модулем `Helloworld`, но также может быть использован в контроллерах других модулей. Конечно, будет ли это на самом деле уместно, зависит от конкретного случая.

Х. Представление (View)

Концепция и режим работы

Представление, которое часто называют также в рамках шаблона MVC "слоем представления", отвечает за отображение результатов обработки. С момента успешного выполнения контроллером (или действием) до представления окончательного результата в браузер пользователя, результат представления обретает ряд различных форм и подлежит большому количеству трансформационных процессов, некоторым из них еще на сервере, а некоторым из них – на стороне клиента. Первоначальное представление результата обработки, так называемая "модель представления" (view model), генерируется контроллером приложения. Модель представления изначально не содержит никакой презентационной информации (например, HTML), но вместо этого просто содержит "базовую структуру данных" в виде пар "ключ-значение":

```
1 <?php
2 public function indexAction()
3 {
4     return new ViewModel(
5         array(
6             'event' => 'Beatsteaks',
7             'place' => 'Berlin',
8             'date' => $this->currentDate()
9         )
10    );
11 }
```

Listing 10.1

В качестве альтернативы также может быть возвращен просто массив PHP:

```
1 <?php
2 // [...]
3 public function indexAction()
4 {
5     return array(
6         'event' => 'Beatsteaks',
7         'place' => 'Berlin',
8         'date' => $this->currentDate()
9     );
10 }
```

Listing 10.2

С помощью слушателя фреймворка `CreateViewModelListener`, который стандартно реагирует на результат диспетчеризации контроллера `ActionControllers`, сгенерированный массив впоследствии автоматически преобразуется в модель представления.

Тем не менее, модель представления – это больше, чем просто контейнер для полезных данных. Он также содержит информацию о шаблоне, с которым эти данные впоследствии объединяются. Эта информация ретроспективно добавляется во фреймворк и `InjectViewModelListener`; это происходит в рамках события `dispatch` контроллера `ActionController`.

Модели представления могут быть, к тому же, вложенными. Этот факт очень полезен для практического применения и подходит для распределения создания одной страницы между несколькими контроллерами. Плагин `Forward`, с которым мы уже знакомы, может быть использован для этого, т.е. для того, чтобы работал не один только контроллер или действие, а целая последовательность контроллеров или действий, соответственно, при необходимости в рамках обработки запроса. Таким образом, множество моделей представления может быть сгенерировано и обработано одновременно.

```
1 <?php
2 // [...]
3 public function indexAction()
4 {
5     $widget = $this->forward()
6         ->dispatch('Helloworld\Controller\Widget');
7
8     $page = new ViewModel(
9         array(
10             'greeting' => $this->greetingService->getGreeting(),
11             'date' => $this->currentDate()
12         )
13     );
14
15     $page->addChild($widget, 'widgetContent');
16     return $page;
17 }
```

Listing 10.3

В этом действии `indexAction` мы сначала обеспечили запуск `WidgetController` (или его `indexAction`, соответственно). Возвращаемая им модель представления кэшируется в переменной `$widget`. Сохраняемая в `$page` ссылка – это текущая кэшированная модель представления `indexAction`. Однако прежде, чем она будет возвращена, `addChild` присоединяет сгенерированную контроллером `WidgetController` модель представления к этой, назовем ее для простоты, "первичной модели представления". Таким образом, можно получить доступ к результатам визуализации модели представления контроллера `WidgetController` с помощью ключа `widgetContent`:

```
1 // [...]
2 <sidebar>
3     <?php echo $this->widgetContent ?>
4 </sidebar>
5 // [...]
```

Макеты (Layouts)

Если знать, как вкладывать модели представления, можно быстро понять методику работы с макетами. Но, возможно, сначала мы должны сделать шаг назад: идея макетов заключается в получении HTML кода, который запрашивается многими или даже всеми контроллерами и действиями (пока что мы представим ее так для простоты, но, конечно же, с помощью Zend Framework 2 могут быть также получены структуры данных и вне HTML). Макеты включают в себя мета-теги, основной каркас HTML, ссылки на файлы CSS и тому подобное.

В техническом смысле, макет – это не более, чем модель представления, которая ссылается на другую модель представления, сгенерированную другим действием контроллера или другим контроллером в качестве "дочерней". Чтобы получить доступ к модели представления в шаблоне макета, фреймворк автоматически регистрирует ключ `content`, точно так, как мы создали вручную ключ `widgetContent` в предыдущем примере. Таким образом, в шаблоне макета результат действия контроллера можно получить следующим образом:

```
1 <html>
2 <head>
3 <title>My website</title>
4 </head>
5 <body>
6 <?php echo $this->content; ?>
7 </body>
8 </html>
```

Listing 10.5

Шаблоном макета, который автоматически учитывается фреймворком, можно управлять из контроллера с помощью плагина контроллера, который обсуждался в предыдущей главе, или с помощью помощника представления, который мы также рассмотрим более детально впоследствии.

Макет и шаблоны – от переводчика

По-видимому, внятно эту главу перевести мне не удалось, поскольку вопросы у читателей все равно возникают. Поэтому попробую объяснить своими словами.

Во-первых, что касается макета (layout). Как можно понять из вышеизложенного, основные строки в макете – это "`<?php echo $this->content; ?>`". В этом месте подключается шаблон (template), соответствующий конкретному действию контроллера. Для действия `indexAction` контроллера `IndexController` это будет шаблон `/index/index.phtml`. В остальном задача макета (layout) проста – вывести `doctype`, `css` и `js` файлы, общие для всех макетов. Другими словами, макет служит своеобразной "оберткой" для подключения

шаблона. Для упрощения понимания, можно назвать макет (layout) "файлом темы". Кстати, Zend Framework 2 в стандартной комплектации в качестве темы (css и js) содержит "twitter bootstrap" (<http://twitter.github.io/bootstrap>).

Во-вторых, на этом задача макета (layout) и заканчивается, и в дело вступает шаблон (template) конкретного действия. И вот здесь начинается самое интересное. Поскольку дизайн можно разбить на отдельные блоки: блок рекламы, блок анонсов, блок погоды, блок содержимого и т.д., то наш `indexAction` контроллера `indexController` может запрашивать эти блоки (иногда их называют "виджеты") у других контроллеров и вставлять в свой основной шаблон. Рассмотрим еще раз листинг 10.3:

```
indexController
1 <?php
2 // [...]
3 public function indexAction()
4 {
5     $widget = $this->forward()
6         ->dispatch('Helloworld\Controller\Widget');
7
8     $page = new ViewModel(
9         array(
10             'greeting' => $this->greetingService->getGreeting(),
11             'date' => $this->currentDate()
12         )
13     );
14
15     $page->addChild($widget, 'widgetContent');
16     return $page;
17 }
```

Listing 10.3.U - 1

Контроллер `WidgetController` вернет модель представления, содержащую не только специфические данные, но и путь к собственному шаблону:

```
WidgetController
1 <?php
2 // [...]
3 public function indexAction()
4 {
5     $widgetView = new ViewModel(array('test' => 'test');
6     // $widgetView->setTemplate('content/main-widget'); возможность указать альтернативный шаблон
7     return $widgetView;
8 }
```

Listing 10.3.U – 2

Если для `ViewModel` не задать специфический шаблон с помощью метода `setTemplate`, для модели представления будет установлен шаблон в соответствии со стандартной схемой именования шаблонов, при котором сам шаблон именуется, как имя действия и находится в

каталоге, соответствующем имени контроллера, т.е. в данном случае – 'helloworld/widget/index.phtml'.

Затем в нашем шаблоне helloworld/index/index.phtml, соответствующем действию `indexAction` контроллера `IndexController` мы можем вывести этот виджет, указав то имя, под которым мы добавили его, как дочерний в листинге 10.3.U – 1 с помощью метода `addChild`.

```
module/Helloworld/view/helloworld/index/index.phtml
1 <?php echo $this->widgetContent; ?>
```

Listing 10.3.U – 3

И в этом месте в файле `index.phtml` будет отображен наш виджет. В нашем примере 10.3.U – 2 в модель представления виджета мы передали переменную `test`, но мы можем, конечно же, установить для шаблона виджета любые переменные, в том числе и из базы данных.

Подобный подход вполне вписывается в рамки модульной концепции, поскольку при развитии проекта виджеты могут быть выполнены в виде отдельных модулей. В этой связи особенно интересной мне лично, как переводчику ☺ кажется возможность указать альтернативное размещение для модулей, представляющих из себя виджеты:

```
ZendSkeletonApplication/config/application.config.php
1 'module_listener_options' => array(
2     'module_paths' => array(
3         './module',
4         './vendor',
5         './widget',
6     )
7 )
```

Listing 10.3.U – 4

и таким образом логически отделить модули виджетов от остальных модулей.

Более подробную информацию о слое представления можно найти в переводах официальной документации (<http://zf2.com.ua/doc/68>). В других разделах этого ресурса можно найти документацию по помощникам представления.

Помощники представления

Часто бывает необходима дополнительная обработка данных модели представления в процессе визуализации или для создания дополнительных данных. Например, в контексте таких задач, как "навигация", "мета-теги" и т.д. есть много повторяющихся задач, связанных с представлением, которые можно решить один раз, а затем сделать их многократно используемыми в качестве помощников представления. В стандартной комплектации Zend Framework 2 предоставляет большое количество помощников представления, готовых к использованию. Кроме того, можно создать собственные помощники представления.

Создание собственного помощника представления

В Zend Framework 2 помощники представления не представляют собой ничего особенного, это просто более или менее "обычный класс", который фактически предполагает только методы `getView()` и `setView()`; и в соответствующий помощник представления, который в настоящее время применяется, инжецируется экземпляр представления. Однако реализации этих методов можно избежать, если наследовать наш собственный плагин от класса `Zend\View\Helper\AbstractHelper`. Затем нужно просто вдохнуть жизнь в метод `__invoke()`, чтобы можно было легко использовать плагин в контроллере.

```
1 <?php
2
3 namespace Helloworld\View\Helper;
4
5 use Zend\View\Helper\AbstractHelper;
6
7 class DisplayCurrentDate extends AbstractHelper
8 {
9     public function __invoke()
10    {
11        return date('d.m.Y');
12    }
13 }
```

Listing 10.6

Этот помощник представления обеспечивает вывод текущей даты в представлении (правда, в реальности нам не нужен помощник представления, чтобы это сделать). Определение класса находится в модуле Helloworld в `src/Helloworld/View/Helper/DisplayCurrentDate.php`. Файл `index.phtml`, т.е. представление для действия `index` контроллера `Index`, адаптировано как это показано ниже, и в настоящее время использует этот помощник:

```
1 <h1><?php echo $this->greeting; ?></h1>
2 <h2><?php echo $this->displayCurrentDate(); ?></h2>
```

Listing 10.7

`DisplayCurrentDate` может быть очень легко вызван, как если бы это был метод в текущем представлении (через `$this`). Тем не менее, для того, чтобы это работало, мы должны сделать помощник `DisplayCurrentDate` известным для менеджера `ViewHelperManager` заранее. Мы можем сделать это либо в `module.config.php`, либо в классе `Module` нашего модуля:

```
1 <?php
2 public function getViewHelperConfig()
3 {
4     return array(
```

```

5         'invokables' => array(
6             'displayCurrentDate'
7             => 'Helloworld\View\Helper\DisplayCurrentDate'
8         )
9     );
10 }

```

Listing 10.8

или

```

1 <?php
2 'view_helpers' => array(
3     'invokables' => array(
4         'displayCurrentDate'
5         => 'Helloworld\View\Helper\DisplayCurrentDate'
6     )
7 )
8 // [...]

```

Listing 10.9

После регистрации, этот помощник может быть вызван во всех представлениях. Кстати, `DisplayCurrentDate` не ограничивается модулем `Helloworld`, но также может быть использован в представлениях других модулей. Конечно, будет ли это на самом деле уместно, зависит от конкретного случая.

XI. Модель (Model)

или немного теории

Что же такое "модель" на самом деле? Представление и контроллер относительно четко определены с точки зрения их содержимого и хорошо сформулированы в своих очертаниях и функциях во фреймворке, в то время, как для модели это не так очевидно. В первую очередь это обусловлено тем, что модель может иметь очень разную структуру в зависимости от типа приложения и ситуации. Например, мы уже знакомы с моделью представления. Это та модель, которая инкапсулирована в MVC? Нет, но у нее есть некоторое сходство с тем типом модели, которая должна быть рассмотрена.

Давайте исходить изначально из определения. Википедия [немецкая версия википедии] характеризует модель следующим образом:

1. Иллюстрация – модель всегда является моделью чего-либо, а именно образа, представления натурального или искусственного оригинала, который сам по себе может быть моделью.
2. Сокращение – модели, как правило, охватывает не все атрибуты оригинала, а только те, которые проявляются в отношении создателя модели или пользователя модели.
3. Прагматизм – модели четко не связаны с их оригиналами. Они выполняют свои функции замены а) для некоторых субъектов (для кого?) б) в определенных интервалах времени (когда?) в) при ограничении определенных концептуальных или физических операций (зачем?)

Модель, таким образом, это всегда упрощенное представление реальности, которое в соответствующее время является достаточным для соответствующих целей приложения. Таким образом, модель представления является упрощенным представлением о веб-странице, ограничена данными для отображения, информацией о состоянии некоторых элементов управления (активная или неактивная кнопка, открытый или закрытый бокс, и т.д.), а также ссылкой на шаблон, с которым эти данные должны быть впоследствии связаны. В рамках визуализации модель представления становится дальнейшим, если хотите "более высоко разработанным" представлением реальности, которая ближе к действительности: полностью генерируемая разметка используется для последующего представления на сайте. Она возвращается в вызывающую программу, и браузер на ее основе разворачивает другую модель, DOM. На основе объектной модели документа (Document Object Model, DOM), которая уже несет соответствующий термин "модель" в своем названии, цветные пиксели появляются волшебным образом на экране на заключительном этапе. И даже этот результат фактически тоже является всего лишь моделью. И даже само это объяснение является лишь моделью, потому что, как видим, я опустил сейчас ненужную информацию и детали.

Но прежде, чем это обсуждение станет чрезмерно философским, давайте вернемся к модели MVC. В более строгом определении, эта модель – отражение действительности, затронутой приложением, т.е. специализированным доменом, от которого все зависит. Например, отражение процессов электронной коммерции в онлайн-магазинах или торговых площадках, управление клиентами, контактами и инцидентами в CRM системах или документами, авторами и рубриками в CMS системах. И это лишь некоторые из возможных примеров! Эта специализированная модель, таким образом, по существу является полностью специализированной для конкретного приложения и в конечном итоге не может быть определена обобщенно с любой более высокой точностью. Если взглянуть на стандарты в области корпоративных приложений или типичных веб-приложений, для которых Zend Framework 2 в первую очередь и предназначен, налицо имеется определение "предметно-ориентированного проектирования", Domain-Driven Design (http://ru.wikipedia.org/wiki/Проблемно-ориентированное_проектирование). Здесь к специализированной модели применяются следующие единицы приложения:

- Сущности
- Репозитории
- Объекты-значения
- Агрегаты
- Ассоциации
- Бизнес-сервисы
- Бизнес-события
- Фабрики

Это не введение в мир предметно-ориентированного проектирования, в частности потому, что оно представляет только один из способов реализации модели собственных приложений и в этом плане не обладает какой-либо универсальной ценностью. Тем не менее, некоторые конструкции предметно-ориентированного проектирования соотносятся с таковыми в Zend Framework, и, таким образом, к ним стоит присмотреться.

Persistence ignorance – от переводчика

Далее автор раскрывает для нас терминологию DDD и ни слова не говорит о том, что это такое и зачем оно нужно. А если вы попытаетесь сделать это самостоятельно и в ограниченные сроки, у вас просто вынесет мозг. А на самом деле все очень просто.

Все началось с термина "persistence ignorance", т.е. с утверждения о том, что хорошая объектно-ориентированная модель не должна зависеть от того, как и где сохраняются данные. Более того, мы должны разрабатывать саму модель без оглядки на то, где и как будут храниться ее объекты.

Для примера возьмем компанию, занимающуюся грузоперевозками. Мы мало что знаем о грузоперевозках, но поговорив с заказчиком, который хочет получить программу учета, и внимательно его выслушав, мы можем выделить несколько ключевых терминов, как "клиент" (Customer), "груз" (Cargo) и "пункт назначения" (Location). Как будет

доставляться груз? Сначала крупные партии различных грузов отправятся в "транзитные пункты", скажем, в подразделения компании грузоперевозок в областных центрах. Оттуда – в более мелкие подразделения в райцентрах, и т.д., пока не достигнут адресата (это больше похоже на работу почты, но пример-то условный). Поэтому мы создадим объект "история доставки" (DeliveryHistory), содержащий что-то вроде списка населенных пунктов, через которые пройдет груз. С помощью последнего элемента этого списка мы всегда сможем узнать в любой момент времени, на каком этапе наша доставка и в каком населенном пункте сейчас находится груз. Нужно не забывать, что основная задача на данный момент – это создание *модели предметной области* (domain model), т.е. вычленение в ходе общения со специалистом в своей области ключевых объектов и их взаимоотношений. А это, между прочим, искусство.

Далее по ходу общения всплывает еще один термин, "заявка на доставку груза", другими словами "заказ" (Order). В компанию по грузоперевозкам приходит их клиент и программа должна прежде всего оформить заявку на доставку. Технически говоря, у нас есть контроллер OrderController и действие createAction. В этом действии мы создаем экземпляр модели OrderModel и вызываем ее метод create(), чтобы оформить новую заявку. Далее этой модели будут необходимы некоторые значения других объектов. Что необходимо сделать, чтобы оформить заявку? Принять у клиента груз (физически положить его на склад), зарегистрировав его в системе. Для этого нужно создать новый объект груза Cargo с *уникальным идентификатором* и начальной точкой маршрута. С точки зрения DDD объекты модели можно поделить на два типа: "*сущности*", вроде груза Cargo, обладающие уникальными идентификаторами, и "*объекты-значения*", уникальными свойствами не обладающие. Например, заявка на доставку груза Order уникальными свойствами обладать не будет. Действительно, у груза есть идентификатор, есть он и у клиента (паспортные данные), но объект Order будет представлять собой временный объект, который просто предоставит программе некоторые свойства груза (его Id) и варианты пункта назначения (например, выпадающий список возможных населенных пунктов), но эта информация не принадлежит самому объекту Order, он берет эти значения из других источников. Объект Order не нужно сохранять в базе данных, т.е. он не обладает свойством *persistence* (персистентность, живучесть, сохраняемость). Подобные объекты с точки зрения DDD называются "*объектами-значениями*". Разница между сущностью и объектом-значением в том, что объект-значение не уникален, поэтому его не нужно сохранять в БД. После того, как мы создадим груз с уникальным идентификатором, заявка Order нам не нужна. Мы всегда можем найти груз по его идентификатору или клиента по его данным. Итак, мы уже знаем, что означают термины "*domain model*", "*сущность*" (entity), "*объект-значение*" (value object) и "*persistence*".

Для создания новых сущностей мы используем знакомые нам уже фабрики. Хотя в простейшем случае можно обойтись просто конструктором класса Cargo, но простейшие случаи бывают очень редко. Например, сущность Cargo помимо идентификационного номера должна содержать информацию о населенном пункте (и отделении грузоперевозок), в котором был принят груз. Данные параметры в конструктор сущности может передать фабрика, получив их из конфигурации программы. Итак, за создание новой сущности (а в

случае необходимости – и объекта-значения) отвечают фабрики. Но как нам найти уже существующую сущность? Допустим, мы хотим посмотреть характеристики определенного груза, скажем, вес. У нас есть контроллер груза `CargoController` и его `indexAction` должен вернуть нам модель `ViewModel`, содержащую характеристики конкретного груза с заданным идентификатором (подобно тому, как это происходит для обычной веб-страницы). Что должно сделать действие нашего контроллера, чтобы получить нужный объект `Cargo`? Вся работа с БД в DDD ведется с помощью хранилищ (`Repositories`). Отличие DDD как раз состоит в том, что у самого объекта `Cargo` нет каких-либо методов для сохранения своих свойств/параметров в БД или поиска в ней. Сущность `Cargo` реализует только бизнес-логику и ничего более. Всю работу по сохранению и поиску выполняют хранилища. Если контроллеру `CargoController` понадобится сущность `Cargo` с заданным идентификатором, он обратится к хранилищу `CargoRepository`. Хранилища содержат различные методы для поиска по заданным параметрам, например `getId()` или `getByWeight()` и т.д. Хранилище, найдя в БД объект с заданными свойствами, передает найденные свойства в конструктор сущности или в фабрику (в зависимости от того, как именно в системе создается эта сущность), и те возвращают новый экземпляр объекта. Итак, фабрики создают новые объекты, а хранилища сохраняют их и восстанавливают с помощью баз данных, причем при восстановлении выполняют только поиск, а само создание объекта делегируют фабрикам. Хранилища также реализуют методы поиска в БД сущностей по различным критериям, т.е. в рамках DDD вся работа с БД происходит через хранилища.

Разработка модели в DDD происходит "сверху вниз", т.е. сначала мы изучаем предметную область (`domain`), вычленив из нее ключевые объекты и моделируем их свойства и поведение в классах, т.е. создаем *модель предметной области* (*domain model*). Затем определяем, какие из созданных нами классов будут обладать столь уникальными свойствами, что их нужно будет сохранять в базе данных. Такие объекты в рамках DDD получают определение "*сущностей*". Сущностям требуются *хранилища* для их сохранения и извлечения. Самое интересное заключается в том, что на этапе проектирования верхнего слоя бизнес-модели мы можем создать хранилище с его методами, но не реализовывать эти методы, а сделать так называемые "заглушки". Заглушки будут возвращать некоторые массивы или переменные с "условными" данными, позволяя нам в первую очередь протестировать работу основного слоя бизнес-логики. Т.е., "*persistence ignorance*".

По сути, хранилища реализуют логику работы с базой данных. Поскольку между реляционными базами данных и объектными моделями существует принципиальное различие, обмен данными между ними становится слишком сложен. Если объект модели "знает" о реляционной структуре, то изменение в одной из структур приведет к проблемам в другой.

Считается очень плохим тоном, если хранилища используют в своих методах `sql`-запросы. Обычно для этих целей в приложении используется какой-либо дополнительный инструментальный, дополнительный слой абстракции к базам данных. Этот дополнительный слой абстракции должен позволять легко сменить тип используемой базы данных и быть достаточно гибким, чтобы позволить реализовать логику работы хранилища. Подобный

инструментарий должен выбираться еще на этапе проектирования в зависимости от размеров проекта. Для небольших и средних проектов подойдет Zend\Db. Он реализует так называемую "функциональность CRUD" (Create, Read, Update, Delete), т.е. четыре базовые функции при работе с персистентными хранилищами данных – создание, чтение, редактирование и удаление. Как мы видим в примере "быстрый старт с ZendFramework2" (<http://zf2.com.ua/doc/35>) в качестве хранилища используется объект класса AlbumTable. В качестве инструментария для работы с БД этот класс использует Zend\Db, а если быть совсем точным, то Zend\Db\Gateway\TableGateway. Пример Album плохо демонстрирует возможности DDD, поскольку последний используется для построения систем со сложной бизнес-логикой, а в данном случае логика сводится всего лишь к операциям CRUD. Тем не менее, этот пример демонстрирует работу популярного шаблона Table Data Gateway, хотя авторы "быстрого старта" и предупреждают, что для сложных систем возможностей этого шаблона может оказаться недостаточно.

Нужно отметить, что в рамках DDD существует возможность не использовать хранилища вообще и они не являются обязательными элементами модели. Например, всю работу, которую должно выполнять хранилище, может взять на себя ORM – система (<http://ru.wikipedia.org/wiki/ORM>), такая, как Doctrine или Propel.

Кроме того, шаблоны Table Data Gateway и Row Data Gateway, реализация которых в виде соответствующих классов входит в стандартную поставку Zend Framework 2, не являются единственными в своем роде. "Классическим" в этом смысле решением является шаблон Data Mapper, существуют и другие, как Identity Map и Unit of Work, последний приобрел огромную популярность в среде разработчиков на С#. Я очень рекомендовал бы начинающим пока что не начинать изучения этих архитектурных решений.

Но вернемся к теории DDD. Нам уже знакомо значение терминов "domain model", "сущность", "объект-значение" и "persistence". Вернемся к нашему примеру с грузоперевозками. Как мы помним, у каждого груза (Cargo) имеется собственная "история доставки" (DeliveryHistory). Какой из этих объектов главный? Очевидно, что груз, ведь если мы захотим удалить конкретный груз из базы данных, зачем оставлять там историю доставки? А если мы удалим только историю доставки, то мы просто "заметаем следы" ☺. Как мы реализовали бы эти взаимоотношения в рамках ООП? Объект Cargo содержал бы в переменной-члене конкретный экземпляр DeliveryHistory. Подобное взаимоотношение объектов, когда один объект содержит в переменной-члене другой объект, называется с точки зрения ООП агрегированием. С точки зрения DDD оба этих объекта называются агрегатом, причем объект Cargo является корневым объектом агрегата. Вы спросите, зачем это нужно? Все очень просто. На практике очень часто приходится иметь дело с подобными составными объектами. Чуть выше мы уже рассмотрели один пример – удаление объекта. В базе данных свойства Cargo и DeliveryHistory будут храниться в разных таблицах, причем у DeliveryHistory будет внешний ключ, указывающий на конкретный груз (Cargo). Система агрегатов позволяет определить рамки ответственности, необходимые для взаимодействия нескольких сложных составных объектов. Корневой объект агрегата – это один конкретный объект-сущность (entity), содержащийся в агрегате. Кроме того, корневой объект – единственный член агрегата, на который могут ссылаться

внешние объекты, в то время, как объекты внутри агрегата (или, как говорят по-другому, внутри границы агрегата), могут ссылаться друг на друга как угодно. Существует ряд правил, позволяющих реализовать концепцию агрегата в виде программной конструкции, но это тема для самостоятельного изучения. Пока же мы узнали, что такое агрегаты и для чего они нужны. Кроме того, при рассмотрении агрегатов мы затронули тему ассоциаций, т.е. взаимоотношений объектов модели между собой.

Все это, и множество других занятных хобби на ближайшие пару лет ожидают вас по адресу:

- Эрик Эванс. "Предметно-ориентированное проектирование".
- Мартин Фаулер. "Архитектура корпоративных программных приложений".

Начинающим я бы не советовал на данном этапе изучать эту литературу. А основные необходимые сведения автор раскроет более подробно (и в частности, опишет различия между шаблонами) далее по тексту и в главе "Персистентность с помощью Zend\Db".

Итак, предоставляем слово автору...

Сущности, репозитории и объекты-значения

"Сущность" (entity) представляет собой определенный объект в системе (например, клиента или заказ в магазине системы), обладающий уникальным идентификатором. Как правило, идентификация осуществляется путем использования IDs (идентификаторов), например, определенного номера клиента. Иногда также можно объединить характеристики объекта таким образом, что существует однозначность. Однако этого, например, очень трудно достичь для физических лиц. Так, например, в городах вроде Москвы, Киева или Минска будут несколько человек с одинаковыми именами. У некоторых из них дополнительно будут даже дни рождения в тот же день. Однако, это принципиально разные "сущности" (лица), которых абсолютно необходимо различать.

Обозначение "сущность" проявляется к тому же особенно в связи с так называемым "сохранением состояния" (persistence), т.е. сохранением таких объектов в базах данных, в результате чего они выживают между запросами. Так, например, ORM система Doctrine 2 (<http://www.doctrine-project.org>) использует обозначение "сущность" (entity) для всех объектов, которые хранятся и администрируются в базе данных.

"Хранилища" (repository) позволяют получить доступ к сущностям, сохраненным в базе данных. В зависимости от конкретной реализации, хранилище действует, как контейнер SQL запроса для конкретного типа сущности. Тем не менее, иногда это нечто большее, как мы скоро увидим.

У "объекта-значения" (value object), в отличие от "сущности", нет собственной идентичности. Например, два экземпляра класса DateTime, стандартного объекта PHP, предназначенного для представления даты и времени, которые были созданы с одинаковой временной отметкой (time stamp) и, следовательно, имеют одинаковые, просто идентичные

характеристики. Установление различий концептуально не нужно. Как правило, объекту-значению не нужно "сохранение состояния", хотя это принципиально возможно с технической точки зрения.

Сущность, или даже объект-значение могут быть представлены с помощью простого класса:

```
1 <?php
2 class User
3 {
4     private $name;
5     private $email;
6     private $password;
7
8     public function setEmail($email)
9     {
10         $this->email = $email;
11     }
12
13     public function getEmail()
14     {
15         return $this->email;
16     }
17
18     public function setName($name)
19     {
20         $this->name = $name;
21     }
22
23     public function getName()
24     {
25         return $this->name;
26     }
27
28     public function setPassword($password)
29     {
30         $this->password = $password;
31     }
32
33     public function getPassword()
34     {
35         return $this->password;
36     }
37 }
```

Listing 11.1

Как мы увидим позже в рамках `Zend\Db` и `Zend\Form`, сущности и объекты-значения, представленные таким образом, могут быть чрезвычайно полезны в разных местах приложения.

Бизнес-сервисы и фабрики

Мы уже обсуждали сервисы и фабрики. Сервис может быть создан с помощью фабрики – или с помощью `Zend\Di`, как мы увидим впоследствии. Операции в рамках предметной области, которые по своей сути не принадлежат ни одному конкретному объекту, относятся к "бизнес-сервисам". Они по своей сути являются не предметами, а видами деятельности. В системе магазина можно, например, определить сервис обмена валют `"CurrencyConversion"`, которому передается объект-значение со свойствами `"сумма"` и `"валюта"`, вместе с информацией о валюте, в которую должно быть выполнено преобразование. Результат преобразования возвращается в виде нового объекта-значения. По сути, преобразование в данном контексте можно было бы реализовать так же, как часть объекта-значения, а также реализовать функцию в соответствующем классе – назовем его `"прайс"` (который определяется на основе `"суммы"` и `"валюты"`). Однако соответствующий объект-значение в этом случае должен знать валюты, поддерживаемые системой, которые по-видимому, сохраняются в базе данных в виде сущностей. Сомнительно, что мы на самом деле хотели бы таких связей.

Данная функциональность затрагивает, таким образом, не только один тип объектов в системе, но вместо этого несколько одновременно. `"CurrencyConversionService"` поэтому был бы хорошим кандидатом для представления в виде "бизнес-сервиса" и, следовательно, также классифицировался бы, как модель приложения.

Так называемые "технические сервисы" или "сервисы инфраструктуры", которые обеспечивают неспециализированные услуги, например, физическую отправку сообщений электронной почты или SMS-сообщений, функции регистрации и т.д., меньше зависят от специализированных объектов приложения и в рамках моделируемой предметной области не предоставляют самостоятельной функциональности. `Zend Framework 2` уже содержит большое количество "технических сервисов", которые могут использоваться независимо от соответствующего опыта. Таким образом, в отличие от "бизнес-сервисов", "технические сервисы" не рассматриваются в качестве модели приложения.

Бизнес-события

В предыдущих главах мы уже сталкивались с системой событий и многими событиями фреймворка в рамках обработки запроса, например, с событиями `route`, `dispatch` и `render`. В `getGreeting` мы уже конструировали наше собственное событие, которое не носило технически-функциональный характер, а вместо этого главным образом помогало сохранять гибкость бизнес-процессов в приложении. Таким образом, мы можем адаптировать рабочие процессы, добавлять или удалять действия, а также изменять последовательность выполнения в любое время. Кроме того, эти специализированные

события также являются частью модели приложения. В магазине системы, например, типичным событием было бы, скажем, помещение товара в корзину или регистрация полученной прибыли в соответствующей административной консоли сотрудников магазина.

XII. Маршрутизация

Введение

За техническим решением маршрутизации стоит фундаментальная идея, заключающаяся в том, что больше нет необходимости в привязке URL к существующему (PHP-) файлу. Вместо этого URL-адрес может быть выбран произвольно, и связанная с ним логика – обычно это контроллер и действие – соответствующим образом обработана. Теперь существующая дисциплина поисковой оптимизации, SEO, (https://ru.wikipedia.org/wiki/Поисковая_оптимизация) и интернет-маркетинг могут обеспечить, чтобы URL-адреса приложения были сформированы "в стиле Google" и содержали любые корректные слова благодаря приобретенной гибкости в формулировании URL. В частности, у локализованных приложений, URL-адреса которых должны быть созданы на разных языках, быстро достигаются пределы возможностей при отсутствии развитой системы маршрутизации. Zend Framework 2, как и его предшественник, обеспечивает очень высокопроизводительное и гибкое решение маршрутизации, которое было полностью перепрограммировано, выполняется лучше, чем раньше, а также разработано более когерентно.

Кстати, сопоставление URL-адреса контроллеру – это только частный случай, поскольку обязательно должен быть URL, который берется в качестве начального значения для сопоставления. Как мы увидим позже в рамках Zend\Console, контроллерам также могут быть назначены произвольно заданные команды. Это, например, очень практично для постановки заданий для cron. Подробнее мы рассмотрим это впоследствии. Для простоты я буду использовать URL в качестве исходной точки для дальнейших разъяснений субъекта маршрутизации, даже если у этой спецификации общее значение и принципиально не ограничивается URL.

Определение маршрутов

Конкретное отображение URL на контроллер обозначается, как "route", маршрут. Если затребован URL "X", то выполняется контроллер "Y" и его действие "Z". Простое определение маршрута в рамках `module.config.php` выглядит следующим образом:

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
```

```

5         'sayhello' => array(
6             'type' => 'Zend\Mvc\Router\Http\Literal',
7             'options' => array(
8                 'route' => '/sayhello',
9                 'defaults' => array(
10                    'controller' => 'helloworld-index-controller',
11                    'action' => 'index',
12                )
13            )
14        )
15    )
16 )
17 // [...]

```

Listing 12.1

Когда вызывается путь URL `/sayhello`, выполняются контроллер `index` модуля `Helloworld` и его действие `index`. Не намного больше, но и не меньше. Довольно сложное приложение будет содержать большое количество таких определений.

В данном случае ключ `helloworld-index-controller` был определен в качестве псевдонима для фактического контроллера в рамках конфигурации DI (больше информации по `Zend\Di` будет далее в этой книге):

```

1 <?php
2 // [...]
3 'alias' => array(
4     'helloworld-index-controller'
5     => 'Helloworld\Controller\IndexController'
6 )

```

Listing 12.2

Во фреймворке происходит следующее: маршрутизатор принимает запрос и "читает" по списку все сохраненные маршруты. Это происходит либо в виде "стека" (т.е. маршрут, который был добавлен последним, будет, соответственно, проверен первым, а маршрут, добавленный первым – будет проверен последним), либо маршруты сохраняются в виде дерева. В любом случае совпадающий результат приводит к созданию и возвращению объекта типа `RouteMatch`, и работа маршрутизатора заканчивается. Значения передаются в объект `RouteMatch` роутера, среди прочего контроллеру, сконфигурированному для данного маршрута, и затем оцениваются в дальнейшей обработке запроса. Таким образом становится известно, экземпляр какого контроллера должен быть создан для диспетчеризации.

Проверка соответствия

Фреймворк предоставляет ряд параметров для определения простых и/или сложных, например, вложенных правил соответствия.

Проверка пути

Мы уже рассмотрели выше простейший вариант, так называемый маршрут Literal (буквальное соответствие). Он определяет строку символов, которая должна точно соответствовать пути для запрошенного адреса URL. В этом случае маршрут функционирует, и необходимые механизмы работают на основе его конфигурации.

Более гибким является маршрут Regex, в котором путь должен соответствовать URL-адресу на основании регулярного выражения. Давайте еще раз рассмотрим конкретный пример: интернет-магазин обуви мог бы использовать, например, следующий маршрут URL для страниц с подробностями о товаре, где передняя часть заключает в себе "стержень" (slug) определения продукта, в то время, как следующее за ней число представляет собой номер элемента: /converse-as-ox-can-sneaker-black/373682726. Маршрут Regex, который будет работать для этого типа URL, выглядит следующим образом:

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Regex',
7             'options' => array(
8                 'regex' => '/(<slug>[a-zA-Z0-9_-]+)/(<id>[0-9]+)',
9                 'spec' => '/%slug%/%id%',
10                'defaults' => array(
11                    'controller' => 'helloworld-index-controller',
12                    'action' => 'index',
13                )
14            )
15        )
16    )
17 )
18 // [...]
```

Listing 12.3

Контроллер может затем получить доступ к значению slug следующим образом:

```
1 <?php
2 $this->getEvent()->getRouteMatch()->getParam('slug');
```

Listing 12.4

Давайте еще раз точно рассмотрим регулярное выражение '/(<slug>[a-zA-Z0-9_-]+)/(<id>[0-9]+)': путь должен начинаться с символа "/". Далее может следовать любое количество цифр, букв, подчеркивание и дефис, но по меньшей мере один символ должен присутствовать в этом месте (что обозначено символом "+"). Все, что находится между передней частью "/" и задней частью "/", обозначено, как slug и, соответственно, будет доступно через объект RouteMatch. После обязательной второй черты ("/") могут

последовать одна или несколько цифр в диапазоне от 0 до 9. Это число обозначено, как `id`, и также может быть доступно в объекте `RouteMatch`.

Мы еще раз рассмотрим объект `"spec"` впоследствии. Он используется для "обратного пути", т.е. построения URL на базе этого маршрута.

В качестве альтернативы, можно было также использовать маршрут типа `Segment`:

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Segment',
7             'options' => array(
8                 'route' => '/:slug/:id',
9                 'defaults' => array(
10                    'controller' => 'helloworld-index-controller',
11                    'action' => 'index',
12                )
13            )
14        )
15    )
16 )
17 // [...]
```

Listing 12.5

Мы можем обойтись и без параметра `"spec"` в этом случае, потому что это зависит от маршрута. Кроме того, могут быть определены необязательные сегменты, для этого их при определении помещают в квадратные скобки, например, `[/:id]`. Контроллер затем также может получить доступ к `slug`, например, следующим образом:

```
1 <?php
2 $this->getEvent()->getRouteMatch()->getParam('slug');
```

Listing 12.6

Маршрут типа `Segment` также может быть реализован с помощью проверки на базе регулярных выражений, если их включить в `constraints`, но в этом случае для индивидуальных сегментов:

```
1 <?php
2 // [...]
3 'constraints' => array(
4     'slug' => '[a-zA-Z0-9_-]+',
5     'id' => '[0-9]+'
6 )
7 // [...]
```

Listing 12.7

Проверка имени хоста, протокола и т.д.

С помощью маршрута типа `Hostname` можно также проверить имя хоста.

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Hostname',
7             'options' => array(
8                 'route' => 'blog.mydomain.ru',
9                 'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => 'index',
12                 )
13             )
14         )
15     )
16 )
17 // [...]
```

Listing 12.8

Если составить настоящее правило, спецификация пути в URL уже не будет играть вообще никакой роли. Каждый URL, который содержит имя хоста `blog.mydomain.ru`, будет сопоставлен с указанным контроллером.

С помощью маршрута типа `Scheme` можно проверить, используется ли "https", это очень практично, если нужно сделать некоторое содержимое, например "счет клиента", доступным только посредством https.

С помощью маршрута типа `Method` можно проверить метод, которым был послан HTTP-запрос (например, "POST", "GET" или "PUT"). С помощью этого типа маршрута можно, например, использовать REST (<http://ru.wikipedia.org/wiki/REST>) для структуризации веб-сервисов (тем не менее, как мы увидим позже, существует лучшее решение с использованием `AbstractRestController`), или элегантно разделять отображение и обработку формы:

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'blog' => array(
6             'type' => 'Zend\Mvc\Router\Http\Literal',
7             'options' => array(
8                 'route' => '/contactform',
9             ),
10             'child_routes' => array(
11                 'formShow' => array(
```

```

12         'type' => 'method',
13         'options' => array(
14             'verb' => 'get',
15             'defaults' => array(
16                 'controller' => 'form-controller',
17                 'action' => 'show',
18             )
19         )
20     ),
21     'formProcess' => array(
22         'type' => 'method',
23         'options' => array(
24             'verb' => 'post',
25             'defaults' => array(
26                 'controller' => 'form-controller',
27                 'action' => 'process',
28             )
29         )
30     )
31 )
32 )
33 )
34 )
35 // [...]

```

Listing 12.9

В этом случае если поступит GET-запрос на URL `/contactform`, будет выполнено действие `show` контроллера формы, а если поступит POST-запрос, будет выполнено действие `process`. В противном случае можно было бы реализовать неуклюжую логику с использованием `isPost()` в самом действии, чтобы можно было различать первоначальное представление формы (GET) и ее обработку (POST) (в разделе "практика" мы разработаем отличное решение для обработки форм на базе специальных типов контроллеров).

Комбинация правил

Некоторые правила лишены смысла вне определенного контекста. Возможность проверки имени узла, не принимая во внимание путь, в любом случае может быть очень полезна. По этой причине правила могут быть объединены друг с другом в виде деревьев, как можно было видеть в предыдущем примере. Давайте рассмотрим еще один пример: допустим, по адресу `www.mydomain.ru` у нас работает интернет-магазин, и по адресу `blog.mydomain.ru` — соответствующий блог. Самая последняя запись блога должна отображаться по адресу URL `blog.mydomain.ru/recent`, в то время, как URL `www.mydomain.ru/recent` вовсе не существует и должен генерировать сообщение об ошибке 404. Для этого маршруты `Hostname` и `Literal` могут быть скомбинированы:

```

1 <?php
2 // [...]

```

```

3 'router' => array(
4     'routes' => array(
5         'blog' => array(
6             'type' => 'Zend\Mvc\Router\Http\Hostname',
7             'options' => array(
8                 'route' => 'blog.mydomain.ru',
9                 'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => 'index',
12                 )
13             ),
14             'child_routes' => array(
15                 'recent' => array(
16                     'type' => 'literal',
17                     'options' => array(
18                         'route' => '/recent',
19                         'defaults' => array(
20                             'controller' => 'helloworld-index-controller',
21                             'action' => 'recent',
22                         )
23                     )
24                 )
25             )
26         )
27     )
28 )
29 // [...]

```

Listing 12.10

Это маршрут вступит в действие только в том случае, если будет запрошен путь `/recent` и в то же время имя хоста `blog.mydomain.ru`. Это определено в разделе `child_routes`. Любой дочерний маршрут может в свою очередь снова содержать `child_routes`, и таким образом создается древовидная структура. Если существуют `child_routes` на одном и том же "уровне", они будут рассматриваться в качестве альтернативы:

```

1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'blog' => array(
6             'type' => 'Zend\Mvc\Router\Http\Hostname',
7             'options' => array(
8                 'route' => 'blog.mydomain.ru',
9                 'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => 'index',
12                 )
13             ),

```

```

14         'child_routes' => array(
15             'recent' => array(
16                 'type' => 'literal',
17                 'options' => array(
18                     'route' => '/recent',
19                     'defaults' => array(
20                         'controller' => 'helloworld-recent-controller',
21                         'action' => 'show',
22                     )
23                 )
24             ),
25             'recentImage' => array(
26                 'type' => 'literal',
27                 'options' => array(
28                     'route' => '/recent/image',
29                     'defaults' => array(
30                         'controller' => 'helloworld-recent-controller ',
31                         'action' => 'image',
32                     )
33                 )
34             )
35         )
36     )
37 )
38 )
39 // [...]

```

Listing 12.11

В этом случае маршруты для URL `/recent` и `/recent/image` определены для разных действий одного контроллера, но в обоих случаях предполагается, что имя хоста `blog.mydomain.ru`.

Генерация URL

Если необходимо создать ссылку в приложении, которая ссылается на внутреннюю страницу и которую можно получить с помощью определенного маршрута, создать атрибут `href`, необходимый для URL, довольно просто. Преимущество по сравнению с ручным созданием URL-адреса состоит в том, что впоследствии можно адаптировать URL централизованно, без необходимости искать по всему приложению и индивидуально адаптировать каждую ссылку. Давайте снова рассмотрим маршрут типа `Regex`:

```

1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Regex',

```



```

7         'options' => array(
8             'regex' => '/(<slug>[a-zA-Z0-9_-]+)/(<id>[0-9]+)/',
9             'spec' => '/%slug%/%id%',
10            'defaults' => array(
11                'controller' => 'helloworld-index-controller',
12                'action' => 'index',
13            )
14        )
15    )
16 )
17 )
18 // [...]

```

Listing 12.12

Интересную роль в создании URL на основе определения маршрута играет `spec`. Здесь определена схема построения URL для этой страницы. После того, как она будет определена, URL-адреса могут быть созданы с помощью плагина контроллера `url`:

```

1 <?php
2 // [...]
3 $href = $this->url(
4     ->fromRoute(
5         'detailPage',
6         array("slug" => "adidas-samba-sneaker", "id" => 34578347)
7     );

```

Listing 12.13

В этом контексте важно использовать правильное имя маршрута и передать все необходимые компоненты URL-адреса.

Стандартная маршрутизация

Стандартная маршрутизация, которая обеспечивает известный комфорт за счет гибкости, известна и оценена еще с 1 версии Zend Framework. Стандартная маршрутизация гарантирует, что путь в URL стандартно отображает модуль, контроллер и действие. URL вида `/blog/entry/add` инициирует, таким образом, действие `Add` контроллера `Entry` модуля `Blog`.

Стандартная маршрутизация существовала в виде "выпечки" (жестко закодированной) в механике MVC 1 версии фреймворка, но уже не во 2 версии: стандартной маршрутизации в таком виде фактически не существует, но если вы внимательно прочли предыдущие разделы, ее можно легко эмулировать. Мы можем задать стандартную маршрутизацию для модуля `Helloworld` с помощью следующего определения:

```

1 <?php
2 // [...]
3 'router' => array(

```

```

4     'routes' => array(
5         'helloworld' => array(
6             'type' => 'Literal',
7             'options' => array(
8                 'route' => '/helloworld',
9                 'defaults' => array(
10                    '__NAMESPACE__' => 'Helloworld\Controller',
11                    'controller' => 'Index',
12                    'action' => 'index',
13                ),
14            ),
15            'may_terminate' => true,
16            'child_routes' => array(
17                'default' => array(
18                    'type' => 'Segment',
19                    'options' => array(
20                        'route' => '[:controller][:action]]',
21                        'constraints' => array(
22                            'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
23                            'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
24                        ),
25                        'defaults' => array(
26                        )
27                    )
28                )
29            )
30        )
31    )
32 )
33 // [...]

```

Listing 12.14

Тем не менее, соответствующие контроллеры все равно должны быть изначально известны системе (что отличается от 1 версии фреймворка):

```

1 <?php
2 // [...]
3 'controllers' => array(
4     'invokables' => array(
5         'Helloworld\Controller\Widget'
6         => 'Helloworld\Controller\WidgetController',
7         'Helloworld\Controller\Index'
8         => 'Helloworld\Controller\IndexController'
9     )
10 )

```

Listing 12.15

Таким образом могут быть теперь вызваны, например, URL-адреса `/helloworld` и `/helloworld/widget/index`.

Вот еще две дополнительные заметки по определению маршрута: конфигурация `may_terminate` сообщает роутеру, который в действительности оценивает правила, что даже `/helloworld` в одиночку, рассматриваемый индивидуально, представляет собой допустимый маршрут, и что нет абсолютной необходимости учитывать определения `child_routes`. Однако если опустить `may_terminate` (и таким образом неявно установить его в `false`), `/helloworld` не будет работать, а только `/helloworld/widget/index` (если придерживаться примера выше в данном случае). И с помощью параметра `__NAMESPACE__` для контроллера, определенного из URL, используется полное имя класса таким образом, что оно соответствует топологии хранения контроллера. Например, в противном случае поиск для контроллера с именем "Widget" для URL `/helloworld/widget/index` окончится неудачей. Однако его правильное имя, которое автоматически корректно формируется таким образом, будет просто `Helloworld\Controller\Widget`.

Креативная маршрутизация: A/B-тестирование

Немного творчества, и маршрутизация может быть использована в целях, которые изначально нельзя было бы и предположить. Таким образом, например, можно маршрутизировать часть URL, в данном случае (теоретически) его половину, в виде альтернативной реализации.

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Literal',
7             'options' => array(
8                 'route' => '/examplewebsite',
9                 'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => rand(0,1) ? 'original' : 'variation'
12                 )
13             ),
14         )
15     )
16 )
17 // [...]
```

Listing 12.16

В этом случае в контроллере `helloworld-index-controller` должно быть действие `original` и действие `variation`, первое из которых содержит исходную версию страницы, а второе – гипотетически усовершенствованный вариант этой страницы. Последняя должна доказать свое превосходство в рамках A/B-тестирования (<http://ru.wikipedia.org/wiki/A/B->

[тестирование](#)), например, по коэффициенту конверсии ([http://ru.wikipedia.org/wiki/Конверсия \(в интернет-маркетинге\)](http://ru.wikipedia.org/wiki/Конверсия_(в_интернет-маркетинге))). Совместно с системой интернет-статистики (<http://ru.wikipedia.org/wiki/Веб-аналитика>), такой, как бесплатный Google Analytic (<http://www.google.com/intl/ru/analytics>), могут быть произведены и оценены даже сложные сценарии А/В-тестирования.

XIII. Внедрение зависимостей

Введение

Идея внедрения зависимостей (dependency injection, DI) состоит в следующем: объект не создает дополнительные объекты по мере необходимости, а вместо этого они предоставляются ему из внешнего источника, образно говоря "внедряются". Существенное преимущество этой процедуры в том, что сам зависимый объект больше не должен знать, от чего именно он зависит. Информация о зависимости извлекается и управляется отдельно. Это позволяет разработчику приложений предоставлять зависимому объекту в определенных ситуациях, например при проведении юнит-тестов, альтернативные реализации.

[http://ru.wikipedia.org/wiki/Внедрение_зависимости]

Если вновь рассмотреть наши GreetingService и LoggingService, но при этом игнорировать инициирование и обработку событий, код можно представить следующим образом: фабрика GreetingServiceFactory обеспечивает, чтобы зависимость GreetingService от LoggingService была решена:

```
1 <?php
2 namespace Helloworld\Service;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class GreetingServiceFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $greetingService = new GreetingService();
12
13         $greetingService->setLoggingService(
14             $serviceLocator->get('loggingService')
15         );
16
17         return $greetingService;
18     }
19 }
```

Listing 13.1

GreetingService просто обращается к внедренному LoggingService, не проверяя, находится ли тот в его распоряжении. В этом контексте он полагается на то, что этот сервис стал доступен прежде, чем он будет использоваться, независимо от того, кто его предоставил.

```

1 <?php
2 namespace Helloworld\Service;
3
4 class GreetingService
5 {
6     private $loggingService;
7
8     public function getGreeting()
9     {
10         $this->loggingService->log("getGreeting ausgefuehrt!");
11
12         if(date("H") <= 11) {
13             return "Good morning, world!";
14         } else if (date("H") > 11 && date("H") < 17) {
15             return "Hello, world!";
16         } else {
17             return "Good evening, world!";
18         }
19     }
20
21     public function setLoggingService($loggingService)
22     {
23         return $this->loggingService = $loggingService;
24     }
25
26     public function getLoggingService()
27     {
28         return $this->loggingService;
29     }
30 }

```

Listing 13.2

Затем `LoggingService`, соответственно, выполняет свою работу:

```

1 <?php
2 namespace Helloworld\Service;
3
4 class LoggingService
5 {
6     public function log($str)
7     {
8         // code for logging
9     }
10 }

```

Listing 13.3

Если мы хотим теперь избежать того, чтобы при проведении юнит-тестов наш журнал засорился многочисленным "фиктивным" доступом, мы можем внедрить

`FakeLoggingService` в `GreetingService` на время выполнения теста, что с одной стороны позволит `GreetingService` выполнить свою работу, а с другой стороны просто отменит бесполезные записи в журнале.

```
1 <?php
2 namespace Helloworld\Service;
3
4 class FakeLoggingService
5 {
6     public function log($str)
7     {
8         return;
9     }
10 }
```

Listing 13.4

Мы можем внедрить `FakeLoggingService` во время выполнения теста:

```
1 <?php
2 namespace Helloworld\Service;
3
4 class GreetingService extends \PHPUnit_Framework_TestCase
5 {
6     public function testGetGreeting()
7     {
8         $greetingService = new GreetingService();
9         $fakeLoggingService = new FakeLoggingService();
10        $greetingService->setLoggingService($fakeLoggingService);
11        $result = $greetingService->getGreeting();
12        $greetingSrv = $serviceLocator$this->assertEquals(/* [...] */);
13    }
14 }
```

Listing 13.5

Как бы для проформы и чтобы убедиться, что `GreetingService` действительно всегда обеспечен "logging-подобным сервисом", можно использовать интерфейс:

```
1 <?php
2 namespace Helloworld\Service;
3
4 interface LoggingServiceInterface
5 {
6     public function log($str);
7 }
```

Listing 13.6

`LoggingServiceInterface` будет затем реализован как "реальным" сервисом `LoggingService`,

```
1 <?php
```

```

2 namespace Helloworld\Service;
3
4 class LoggingService implements LoggingServiceInterface
5 {
6     public function log($str)
7     {
8         // code for logging
9     }
10 }

```

Listing 13.7

так и "фальшивой реализацией":

```

1 <?php
2 namespace Helloworld\Service;
3
4 class FakeLoggingService implements LoggingServiceInterface
5 {
6     public function log($str)
7     {
8         return;
9     }
10 }

```

Listing 13.8

Если используется соответствующий "сеттер", в GreetingService может быть указан тип, чтобы задать интерпретатору PHP правильный тип объекта:

```

1 <?php
2 // [...]
3 public function setLoggingService(LoggingServiceInterface $loggingService)
4 {
5     return $this->loggingService = $loggingService;
6 }
7 // [...]

```

Listing 13.9

Кстати, использование нативной функции date в GreetingService очень усложняет юнит-тестирование. Впоследствии при рассмотрении юнит-тестирования мы подробно рассмотрим возможное решение для данного конкретного примера.

Zend\Log

Кстати, как мы скоро увидим, нам, как разработчикам, не придется тратить усилия на самостоятельное программирование ведения логов вообще: Zend Framework уже предоставляет очень гибкую реализацию журналирования в виде Zend\Log. Подробности мы рассмотрим чуть позже.

Zend\Di для графов объектов

Для начала можно представить себе Zend\Di, как общую фабрику, которая служит "контейнером для внедрения зависимостей". В противоположность этому наша фабрика GreetingServiceFactory была крайне специфической, поскольку мы решили все зависимости от LoggingService явно и вручную. Кроме того, мы можем возложить решение этой зависимости на Zend\Di:

```
1 <?php
2 namespace Helloworld\Service;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class GreetingServiceFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $di = new \Zend\Di\Di();
12
13        $di->configure(new \Zend\Di\Config(array(
14            'definition' => array(
15                'class' => array(
16                    'Helloworld\Service\GreetingService' => array(
17                        'setLoggingService' => array(
18                            'required' => true
19                        )
20                    )
21                )
22            ),
23            'instance' => array(
24                'preferences' => array(
25                    'Helloworld\Service\LoggingServiceInterface'
26                    => 'Helloworld\Service\LoggingService'
27                )
28            )
29        )));
30
31        $greetingService = $di->get('Helloworld\Service\GreetingService');
32        return $greetingService;
33    }
34 }
35 }
```

Listing 13.10

Прежде, чем мы рассмотрим фактическую конфигурацию, давайте сначала сделаем несколько дополнительных улучшений. GreetingServiceFactory нам в действительности больше не нужен, поскольку Zend\Di выполняет за нас всю работу, как обобщенная

реализация фабрики. Вместо этого мы перенесем код конфигурации непосредственно в класс `Module` модуля `Helloworld`. При этом мы воспользуемся тем, что фреймворк уже автоматически сделал экземпляр `Zend\Di` доступным для нас в виде сервиса в `ServiceManager`. Мы можем получить этот экземпляр двумя различными способами. Либо мы вручную запросим `DependencyInjector` из `ServiceManager`:

```
1 <?php
2 // [...]
3 $di = $serviceManager->get('DependencyInjector');
4 // [...]
```

Listing 13.11

и с его помощью получим `GreetingService`:

```
1 <?php
2 // [...]
3 $di = $serviceManager->get('DependencyInjector');
4 $greetingService = $di->get('Helloworld\Service\GreetingService');
5 // [...]
```

Listing 13.12

Или же в качестве альтернативы мы можем сделать проще и использовать "резервный механизм" `ServiceManager`: в случае, если `ServiceManager` не может предоставить запрошенный сервис, поскольку он попросту ничего не знает о сервисе, он вновь еще раз запрашивает `DependencyInjector`, может ли тот оказать помощь и предоставить соответствующий сервис. А в нашем случае – может. Следовательно, мы обойдемся без `GreetingServiceFactory` и полностью перенесем код конфигурации `Zend\Di` непосредственно в `module.config.php` модуля `Helloworld`:

```
1 <?php
2 return array(
3     'di' => array(
4         'definition' => array(
5             'class' => array(
6                 'Helloworld\Service\GreetingService' => array(
7                     'setLoggingService' => array(
8                         'required' => true
9                     )
10                )
11            )
12        ),
13        'instance' => array(
14            'preferences' => array(
15                'Helloworld\Service\LoggingServiceInterface'
16                => 'Helloworld\Service\LoggingService'
17            )
18        )
19    ),
```

```

20     'view_manager' => array(
21         'template_path_stack' => array(
22             __DIR__ . '/../view'
23         )
24     ),
25     'router' => array(
26         'routes' => array(
27             'sayhello' => array(
28                 'type' => 'Zend\Mvc\Router\Http\Literal',
29                 'options' => array(
30                     'route' => '/sayhello',
31                     'defaults' => array(
32                         'controller' => 'Helloworld\Controller\Index',
33                         'action' => 'index',
34                     )
35                 )
36             )
37         )
38     ),
39     'controllers' => array(
40         'factories' => array(
41             'Helloworld\Controller\Index'
42             => 'Helloworld\Controller\IndexControllerFactory'
43         ),
44         'invokables' => array(
45             'Helloworld\Controller\Widget'
46             => 'Helloworld\Controller\WidgetController'
47         )
48     ),
49     'view_helpers' => array(
50         'invokables' => array(
51             'displayCurrentDate'
52             => 'Helloworld\View\Helper\DisplayCurrentDate'
53         )
54     )
55 );

```

Listing 13.13

Верхняя секция с ключом `di` является новой и соответствует коду, который у нас первоначально был в `GreetingServiceFactory`. Фреймворк ищет записи для `di` и при наличии передает конфигурацию в `DependencyInjector`, который на самом деле становится доступным автоматически. Теперь мы можем удалить метод `getServiceConfig()` из класса `Module` модуля `Helloworld`. Он нам больше не нужен, если мы сделаем небольшую коррекцию в `IndexControllerFactory` и убедимся, что используем полное имя `Helloworld\Service\GreetingService`, когда запрашиваем сервис (до сих пор у нас используются обозначения, которые не соответствуют классу в данном случае. Тем не

менее, чтобы избежать "конфликта имен" между сервисами различных модулей, желательно всегда использовать полное имя класса для обозначения сервисов):

```
1 <?php
2 namespace Helloworld\Controller;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class IndexControllerFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $ctr = new IndexController();
12        $serviceLocator = $serviceLocator->getServiceLocator();
13
14        $greetingSrv = $serviceLocator->get(
15            'Helloworld\Service\GreetingService'
16        );
17
18        $ctr->setGreetingService($greetingSrv);
19        return $ctr;
20    }
21 }
```

Listing 13.14

Что здесь происходит? ServiceManager получает запрос на GreetingService и, поскольку не находит подходящего ответа (ведь мы удалили конфигурацию сервиса и перенесли все в DependencyInjector), запрашивает DependencyInjector. Последний помогает первому и в процессе обеспечивает, чтобы LoggingService находился в режиме ожидания и GreetingService был доступен. ServiceManager теперь получает рабочий сервис и возвращает его вызывающей программе.

Вот еще немного информации: мы иногда используем термин ServiceManager, а иногда – ServiceLocator. Это может немного запутать. ServiceLocator – это интерфейс, тогда как ServiceManager является его конкретной реализацией, которую предоставляет непосредственно фреймворк. Таким образом, ServiceManager – это тот, кто выполняет всю фактическую работу. Как и во многих местах фреймворка, идея, лежащая в основе, рассматривает возможность замены ServiceManager другой реализацией. Для того, чтобы остальная часть фреймворка и приложения на его основе по-прежнему функционировали, эта альтернативная реализация должна соответствовать шаблону интерфейса ServiceLocator. Таким образом, когда используют термин ServiceLocator, на самом деле говорят о конкретной стандартной реализации, о ServiceManager.

Вернемся к упомянутым выше деталям конфигурации DI. Вначале Zend\Di сообщают, что существует класс Helloworld\Service\GreetingService, у которого есть метод setLoggingService() и что в любом случае зависимость должна быть доступна:

```

1 <?php
2 // [...]
3 'definition' => array(
4     'class' => array(
5         'Helloworld\Service\GreetingService' => array(
6             'setLoggingService' => array(
7                 'required' => true
8             )
9         )
10     )
11 )
12 // [...]

```

Listing 13.15

Но какая из них конкретно подразумевается? В этом случае вступает в игру `RuntimeDefinition`: `Zend\Di` активно ищет ее в декларации соответствующего метода `GreetingService`

```

1 <?php
2 // [...]
3 public function setLoggingService(LoggingServiceInterface $loggingService)
4 {
5     return $this->loggingService = $loggingService;
6 }
7 // [...]

```

Listing 13.16

и самостоятельно находит информацию о типе объекта, который должен быть внедрен в этом случае. Если бы мы объявили определенный класс, как своего рода указание вместо `LoggingServiceInterface`, мы бы уже закончили: `Zend\Di` создал бы экземпляр соответствующего класса и передал бы его в `GreetingService` с помощью `setLoggingService()`, как только последний был бы запрошен.

Вот как! Однако в нашем случае `Zend\Di` найдет не определенный класс, как ссылку на тип, а указатель на интерфейс. Следовательно, теперь мы должны предоставить дополнительную информацию, которая фактически укажет `Zend\Di` класс, который должен использоваться в этом случае:

```

1 <?php
2 // [...]
3 'instance' => array(
4     'preferences' => array(
5         'Helloworld\Service\LoggingServiceInterface'
6         => 'Helloworld\Service\LoggingService'
7     )
8 )
9 // [...]

```

Listing 13.17

Эта конфигурация утверждает следующее: "если вам понадобится Helloworld\Service\LoggingServiceInterface (и вы не знаете, как его получить), используйте Helloworld\Service\LoggingService". Что может быть проще! Теперь Helloworld\Service\GreetingService можно косвенно запрашивать с помощью ServiceManager:

```
1 <?php
2 // [...]
3 $greetingSrv = $serviceLocator->get('Helloworld\Service\GreetingService');
4 // [...]
```

Listing 13.18

В предыдущих главах мы создали фабрику для IndexController модуля Helloworld таким образом, чтобы решить его зависимость от GreetingService. Таким же образом мы можем при необходимости использовать Zend\Di, но только тогда, когда мы изначально включили ("показали") соответствующий контроллер для загрузки с помощью Zend\Di:

```
1 <?php
2 return array(
3     'di' => array(
4         'allowed_controllers' => array(
5             'helloworld-index-controller'
6         ),
7         'definition' => array(
8             'class' => array(
9                 'Helloworld\Service\GreetingService' => array(
10                     'setLoggingService' => array(
11                         'required' => true
12                     )
13                 ),
14                 'Helloworld\Controller\IndexController' => array(
15                     'setGreetingService' => array(
16                         'required' => true
17                     )
18                 )
19             )
20         ),
21         'instance' => array(
22             'preferences' => array(
23                 'Helloworld\Service\LoggingServiceInterface'
24                 => 'Helloworld\Service\LoggingService'
25             ),
26             'Helloworld\Service\LoggingService' => array(
27                 'parameters' => array(
28                     'logfile' => __DIR__ . '/../../../data/log.txt'
29                 )
30             ),
31             'alias' => array(
```

```

32         'helloworld-index-controller'
33         => 'Helloworld\Controller\IndexController',
34     ),
35 )
36 ),
37 // [...]
38 );

```

Listing 13.19

Для того, чтобы контроллер загружался с помощью Zend\Di, необходимы следующие настройки в `module.config.php`:

- Фабрика `IndexControllerFactory` удаляется из раздела `controller`. Действительно, мы не хотим ее больше использовать, а вместо этого будем создавать `IndexController` с помощью `Zend\Di`.
- Добавляется подраздел `alias` в разделе `instance` в `di`. Поскольку обратные слешы не допускаются для псевдонимов (`aliases`) в `Zend\Di`, в этом случае мы используем `helloworld-index-controller`. До этого мы все еще могли использовать `Helloworld\Controller\Index` и, следовательно, могли обратиться к фабрике. К сожалению, теперь это уже невозможно, но, фактически, это не имеет никакого значения.
- Маршрут `sayhello` также адаптируется таким образом, чтобы использовался псевдоним `helloworld-index-controller`.
- Добавляется раздел `allowed_controllers` со значением `helloworld-index-controller`, т.е. псевдонимом контроллера, который мы хотим загрузить с помощью `Zend\Di`. Если мы забудем указать таким образом контроллер в "белом списке", он не сможет быть загружен с помощью `Zend\Di`, даже если все другие настройки верны – это средство для обеспечения безопасности.
- Последнее, но не менее важное, мы обеспечиваем, чтобы зависимость от `GreetingService` была решена. Фабрика, которая обеспечивала решение зависимости от `GreetingService`, уже отключена. Чтобы позволить `Zend\Di` взять на себя эту задачу, добавляется запись `Helloworld\Controller\IndexController` в раздел `di > definition > class`, с помощью которого мы указываем `Zend\Di`, что метод `setGreetingService()` в `IndexController` должен быть вызван в обязательном порядке. Если теперь обеспечить в этом методе контроллера `IndexController` модуля `Helloworld` указанием типа, `Zend\Di` сможет определиться, какой класс должен быть внедрен в этом случае:

```

1 <?php
2 // [...]
3 public function setGreetingService(\Helloworld\Service\GreetingService $service)
4 {
5     $this->greetingService = $service;
6 }

```

Listing 13.20

Кстати, у `ControllerManager` есть собственный экземпляр `Zend\Di`, с помощью которого при необходимости может быть получен соответствующий контроллер.

Теперь мы можем таким образом устранить все наши фабрики с помощью `Zend\Di`. Если и дальше двигаться в этом направлении, можно было бы, в принципе, обойтись вообще без фабрик и полностью настроить все графы объектов со всеми их зависимостями описательно, не записывая необходимого в противном случае кода инициализации.

`Zend\Di` действительно должен стать центральным стержневым элементом `Zend\Mvc` и сделать его отдельные фабрики излишними. Тем не менее, если внимательно рассмотреть обработку запроса фреймворком, то становится очевидным, что `Zend\Di` практически никогда не используется, и почти все сервисы создаются собственными фабриками. Сервис-менеджер и соответствующие фабрики возложили на себя задачи, предназначенные для `Zend\Di`, во многих местах. Причины для этого не столько технического, сколько стратегического характера: обеспечить, чтобы `Zend Framework 2` оставался легким для обучения начинающих. А `Zend\Di` неизбежно приносит в игру много скрытой "магии". В принципе, ситуация сходна с собственным выбором, за или против `Zend\Di` и фабрик соответственно. Оба подхода являются проверенными методами для достижения слабой связанности компонентов и, таким образом, для разработки системы, остающейся устойчиво тестируемой, легкой в сопровождении и расширении. Разработчик может выбирать самостоятельно. Посредством вышеописанного "резервного" механизма обе процедуры могут быть без проблем объединены друг с другом и тем самым позволяют выбрать лучший образ действий в данной ситуации.

Все вдруг стало слишком сложно? Не волнуйтесь – это выглядит гораздо хуже, чем оно есть на самом деле. Все, что нужно – это немного практики.

Альтернативные подходы ко "внедрению"

Дополнительно к рассмотренному здесь `RuntimeDefinition`, который использует механизм отражения для понимания структуры кода для разрешения зависимостей, при необходимости могут быть использованы и другие варианты, такие как `CompilerDefinition`, который может обеспечить преимущество в скорости за счет удобства.

Zend\Di для управления конфигурацией

"Внедрение зависимостей" в целом и `Zend\Di` в частности чрезвычайно полезны в двух контекстах: с одной стороны сложные графы объектов (как изображено в примерах выше) могут быть объединены во время работы без необходимости активного получения зависимых объектов или жестких зависимостей которые, например, будут помехой для

какого-либо юнит-тестирования. С другой стороны, отдельные объекты также могут быть настроены с помощью внедрения зависимостей. Таким образом, могут потребоваться данные для доступа к базе данных или внешней веб-службе в любом произвольном месте, информация о хосте, а также любые имеющиеся данные доступа. Правда, соответствующий сервис нашего приложения может сейчас аналогичным образом активно получить доступ к конфигурации приложения с помощью `ServiceManager` и взять там соответствующие значения.

```
1 <?php
2 // [...]
3 $config = $serviceManager->get('Config');
4 $host = $config['dbConfig']['host'];
5 $user = $config['dbConfig']['user'];
6 $pwd = $config['dbConfig']['pwd'];
7 // [...]
```

Listing 13.21

Однако тогда и зависимость от `ServiceManager` и информация о внутренней конфигурационной структуре была бы жестко зашита в коде, что чрезвычайно невыгодно. Вот другой пример: как сообщить нашему `LoggingService` о том, какой файл будет лучшим вариантом для записи логов? Лучше было бы расширить конструктор сервиса так, чтобы мы могли внедрять конфигурацию из внешнего источника:

```
1 <?php
2 namespace Helloworld\Service;
3
4 class LoggingService implements LoggingServiceInterface
5 {
6     private $logfile = null;
7
8     public function __construct($logfile)
9     {
10         $this->logfile = $logfile;
11     }
12
13     public function log($str)
14     {
15         file_put_contents($this->logfile, $str, FILE_APPEND);
16     }
17 }
```

Listing 13.22

Дополнительно расширим конфигурацию DI в `module.config.php`:

```
1 <?php
2 // [...]
3 'di' => array(
4     'definition' => array(
```

```

5      'class' => array(
6          'Helloworld\Service\GreetingService' => array(
7              'setLoggingService' => array(
8                  'required' => true
9              )
10         )
11     ),
12 ),
13 'instance' => array(
14     'preferences' => array(
15         'Helloworld\Service\LoggingServiceInterface'
16         => 'Helloworld\Service\LoggingService'
17     ),
18     'Helloworld\Service\LoggingService' => array(
19         'parameters' => array(
20             'logfile' => __DIR__ . '/../../data/log.txt'
21         )
22     )
23 )
24 )
25 // [...]

```

Listing 13.23

Лучшее в этом то, что мы можем выполнять конфигурацию файла журнала "на месте", т.е. задать конфигурацию сервиса, но без необходимости зашивать [и запрашивать] его непосредственно в самом классе сервиса. В отличие от бесконечных конфигурационных файлов, в которых нанизаны бесчисленные разрозненные значения конфигурации – как, например, по-прежнему происходит в 1 версии фреймворка в файле application.ini – теперь точно известно, где искать, если что-то должно быть изменено.

XIV. Персистентность с помощью Zend\Db

[Персистентность – в программировании означает способность состояния существовать дольше, чем процесс, создавший его.

Источник: http://life-prog.ru/view_zam2.php?id=209&cat=5&page=14

]

Персистентность, т.е. долговременное хранение данных, например, в базах данных, поддерживается PHP долгое время. Для расширения PDO, например, существуют драйверы баз данных для наиболее распространенных производителей и систем таким образом, что почти всегда легко удастся подключиться к базе данных и сохранить данные. Zend Framework 2 обеспечивает с помощью Zend\Db дополнительную поддержку для работы с базами данных и таким образом несколько облегчает персистентность.

Кстати, Zend\Db не является ORM-системой и, следовательно, не будет конкурировать, например, с Doctrine или Propel, но вместо этого ее следует воспринимать, как "легкую" альтернативу, как дополнительную абстракцию, и использовать соответственно. Любой, кто использует "Doctrine" или другую ORM-систему, таким образом, весьма вероятно, обойдется без Zend\Db. Тема "персистентность", к сожалению, полностью перегружена чрезвычайно разными, иногда противоречивыми обозначениями и определениями так, что действительно сложно понять, что скрывается за реализацией. Чтобы было больше возможностей классифицировать Zend\Db – даже если существует риск оказаться под угрозой серьезной критики за то, что упрощаю вещи слишком сильно – давайте кратко рассмотрим наиболее важные подходы к "персистентности":

- Object Relational Mapper (ORM) – объектно-реляционное отображение: отправной точкой для ORM-системы является образ мышления, что мы действительно программируем в объектно-ориентированном стиле, но затем сохраняем данные, как реляционные. Между двумя этими парадигмами действительно есть некоторые параллели, но есть и существенные различия. Можно сказать, что ORM-система таким образом пытается сделать "разницу между мирами", так сказать, как можно более незаметной для объектно-ориентированного мышления и действий разработчика приложения, т.е. мы создаем и манипулируем объектами, а ORM-система берет на себя все остальное, что относится к базам данных. ORM-системы часто предоставляют альтернативы SQL: так, у Doctrine 2, например, есть DQL (Doctrine Query Language), который очень похож на SQL (что, безусловно, является преимуществом для опытных пользователей SQL), но также содержит ряд дополнительных свойств для решения объектно-ориентированных задач настолько эффективно, насколько это возможно, и таким образом, для упрощения операций с базами данных.

- Data Mapper – преобразователь данных: часто считают, что Data Mapper и ORM-система это одно и то же, но может также поддерживать "бэкэнды", отличные от реляционных баз данных. Так, Doctrine 2 (ORM-система, ядром которой является реализация шаблона Data Mapper), например, обеспечивает возможность реализации персистентности в MongoDB или CouchDB, которые являются документо-ориентированными системами. Если упоминают ORM и Doctrine 2, то следует говорить о "Doctrine 2 ORM", потому что есть еще и "Doctrine 2 ODM" (Object Document Mapper). Основной характеристикой преобразователя данных является то, что он полностью связывает один мир (например, объектно-ориентированный) с другим миром (например, реляционным), выступая в роли "преобразователя" между ними. Еще одно важное свойство Data Mapper состоит в том, что он может сохранять и загружать объекты из нескольких различных источников (например, из нескольких таблиц базы данных).
- Table Data Gateway – шлюз к данным таблицы: в то время, как Data Mapper или ORM-система концептуально пытаются скрыть подробности информации о структуре данных "другого мира" в максимально возможной степени, то "Table Data Gateway" использует совершенно иной подход. В этом случае "программа выкладывает карты на стол" (игра слов, стол – англ. table) и создает для каждой таблицы в базе данных объект, который управляет операциями этой таблицы. Важно то, что в отличие от "data mappers", акцент делается на таблице реляционной базы данных.
- Row Data Gateway – шлюз к данным записи: в то время, как Table Data Gateway представляет собой таблицу с помощью объекта, "row data gateway" выступает в качестве объекта для строки, то есть, "элемента", или "записи", если угодно, таблицы базы данных. Важной характеристикой Row Data Gateway является то, что соответствующий объект, в отличие от Table Data Gateway, выполняет код, необходимый для загрузки или сохранения самого себя. Таким образом, они автономны. В отличие от данных, которые загружаются из базы данных в "data mapper", часто также называемых "сущностями" в этом контексте и не способными на такое. Вместо этого центральный "entity manager" предоставляет доступ для обеспечения персистентности объектов. Преимущества "entity manager" в том, что он оперирует так называемыми "единицами работы" (unit of work) и, таким образом, может объединить несколько подобных запросов (statements) в единый запрос.
- Active Record – активная запись: "active record" можно сравнить с "row data gateway". Единственное различие, которое также является скорее определяющим различием, состоит в том, что "active record" – в дополнение к требованиям к персистентности – позволяет работать с областью специализации, т.е. он также содержит код, который не только имеет дело с представлением объекта в строке таблицы, но и касается бизнес-логики приложения.
- Data Access Object (DAO) объект доступа к данным: для "data access object" сложно дать определение (http://ru.wikipedia.org/wiki/Data_Access_Object). Если учесть определение от Core J2EE Pattern Catalogue (<http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>), он позволяет

получить доступ к источнику данных, т.е. он создает соединение с базой данных и работает на "очень низком уровне"; он может служить в качестве основы для упомянутых выше архитектурных решений, которые, в свою очередь, даже не заботятся об определенных моментах. Действительно, у J2E (ранее: J2EE) всегда были несколько широкие концептуальные основы. Таким образом, речь идет о том, что конкретные типы объектов (User, Product, Category, и т.д.) могут быть также получены из различных источников, например, из систем баз данных, но и из плоских файлов, реализаций LDAP или аналогичных источников. В зависимости от источника данных возможны соответствующие подходящие объекты доступа к данным. Однако, поскольку все перманентные объекты часто располагаются в идентичных хранилищах, объекты доступа к данным не играют главную роль, или, так сказать, предпочитают оставаться в тени. За исключением тех случаев, когда говорят "data access object", но реально подразумевают "table data gateway". Эти два термина, к сожалению, действительно часто используются, как синонимы.

Соединение с базой данных

Zend\Db\Adapter отвечает за подключение к базам данных и за объединение характеристик реализации отдельных SQL систем, а также различных типов соединения с базой данных, реализованных PHP. С одной стороны, не все поставщики поддерживают все стандарты SQL в своих системах, либо часто дополнительно предоставляют платные расширения, и, с другой стороны, в распоряжении PHP есть большое количество различных вариантов для взаимодействия с базами данных, в дополнение к PDO, например, также MySQLi и старая версия MySQL (расширение PHP, не СУБД). Таким образом, при использовании Zend\Db\Adapter вместо родных функций и объектов PHP увеличивается переносимость. Если мы снова вернемся к определениям, приведенным выше, мы можем обозначить Zend\Db\Adapter, как наш объект доступа к данным, "data access object".

Для того, чтобы Zend\Db функционировал должным образом, сначала должно быть установлено подключение к базе данных с помощью адаптера:

```
1 <?php
2 $adapter = new \Zend\Db\Adapter\Adapter(
3     array(
4         'driver' => 'Pdo_Mysql',
5         'hostname' => 'localhost'
6         'database' => 'app',
7         'username' => 'root',
8         'password' => ''
9     )
10 );
```

Listing 14.1

Использование пароля

В production-системе мы, естественно, должны гарантировать, что используются надежные пароли; и даже лучше не работать под пользователем root вообще, а под особым пользователем, чьи возможности будут ограничены наиболее необходимыми из них.

MySQL установлен заранее на localhost, и пользователь root доступен без пароля (это нормально при программировании системы). Кроме того, создана таблица log с помощью следующего SQL-запроса:

```
1 CREATE TABLE log (  
2     id int(10) NOT NULL auto_increment,  
3     ip varchar(16) NOT NULL,  
4     timestamp varchar(10) NOT NULL,  
5     PRIMARY KEY (id)  
6 );
```

Создание и выполнение SQL-запросов

Если мы вручную добавим в таблицу несколько наборов данных, то их можно будет получить с помощью сгенерированного ранее адаптера.

```
1 <?php  
2 $stmt = $adapter->createStatement('SELECT * FROM log');  
3 $results = $stmt->execute();  
4  
5 foreach($results as $result){  
6     var_dump($result);  
7 }
```

Listing 14.2

Более элегантной, объектно-ориентированной процедурой для генерации SQL-запросов является Zend\Db\Sql:

```
1 <?php  
2 $sql = new \Zend\Db\Sql\Sql($adapter);  
3 $select = $sql->select();  
4 $select->from('log');  
5 $statement = $sql->prepareStatementForSqlObject($select);  
6 $results = $statement->execute();
```

Listing 14.3

Кроме того, можно также легко добавить условие WHERE:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $select = $sql->select();
4 $select->from('log');
5 $select->where(array('ip' => '127.0.0.1'));
6 $statement = $sql->prepareStatementForSqlObject($select);
7 $results = $statement->execute();
```

Listing 14.4

То же самое касается других обычных конструктивов, таких, как LIMIT, OFFSET, ORDER, и т.д., использование которых интуитивно понятно. Добавить JOIN, по сути, так же просто:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $select = $sql->select();
4 $select->from('log');
5 $select->join('host', 'host.ip = log.ip');
6 $select->where(array('log.ip' => '127.0.0.1'));
7 $statement = $sql->prepareStatementForSqlObject($select);
8 $results = $statement->execute();
```

Listing 14.5

В этом случае мы дополнительно ссылаемся на таблицу host, ее можно создать с помощью следующего запроса:

```
1 CREATE TABLE host (
2     id int(10) NOT NULL auto_increment,
3     ip varchar(16) NOT NULL,
4     hostname varchar(100) NOT NULL,
5     PRIMARY KEY (id)
6 );
```

Этот " join" является внутренним объединением (inner join). Другие типы объединений поддерживаются подобным образом:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $select = $sql->select();
4 $select->from('log');
5
6 $select->join('host',
7     'host.ip = log.ip',
8     array('*'),
9     \Zend\Db\Sql\Select::JOIN_LEFT
10 );
11
12 $select->where(array('log.ip' => '127.0.0.1'));
13 $statement = $sql->prepareStatementForSqlObject($select);
```

```
14 $results = $statement->execute();
```

Listing 14.6

В данном случае параметр `array('*')` указывает столбцы, которые должны быть переданы в результате из таблицы `host` в рамках объединения `"join"`. Если указан `array('*')`, то используются все столбцы; в соответствии со спецификацией определенных столбцов результат можно ограничить и использовать ассоциативный массив для того, чтобы были приняты во внимание только псевдонимы определенных столбцов.

Подобным же образом, данные можно элегантно записать в базу данных:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $insert = $sql->insert('host');
4 $insert->columns(array('ip', 'hostname'));
5 $insert->values(array('192.168.1.15', 'michaels-ipad'));
6 $statement = $sql->prepareStatementForSqlObject($insert);
7 $results = $statement->execute();
```

Listing 14.7

а также обновить:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $update = $sql->update('host');
4 $update->set(array('ip' => '192.168.1.20'));
5 $update->where('hostname = "michaels-ipad"');
6 $statement = $sql->prepareStatementForSqlObject($update);
7 $results = $statement->execute();
```

Listing 14.8

или удалить:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $delete = $sql->delete('');
4 $delete->from('host');
5 $delete->where('hostname = "michaels-ipad"');
6 $statement = $sql->prepareStatementForSqlObject($delete);
7 $results = $statement->execute();
```

Listing 14.9

Разнообразные варианты

Дополнительную информацию по Zend\Db\Sql\Sql можно найти при необходимости в переводах официальной документации <http://zf2.com.ua/doc/105> или в официальной документации:

<https://packages.zendframework.com/docs/latest/manual/en/modules/zend.db.sql.html>

Работа с таблицами и записями

Использование Zend\Db\TableGateway очень облегчает жизнь. Как уже говорилось во введении этой главы, объект TableGateway отображает таблицу в базе данных. Запрос может быть реализован следующим образом:

```
1 <?php
2 $hostTable = new \Zend\Db\TableGateway\TableGateway('host', $adapter);
3 $results = $hostTable->select(array('hostname' => 'michaels-mac'));
4
5 foreach ($results as $result){
6     var_dump($result);
7 }
```

Listing 14.10

Кроме того, в этом случае соответствующий адаптер был создан заблаговременно:

```
1 <?php
2 $adapter = new \Zend\Db\Adapter\Adapter(
3     array(
4         'driver' => 'Pdo_Mysql',
5         'database' => 'app',
6         'username' => 'root',
7         'password' => ''
8     )
9 );
```

Listing 14.11

Если необходимо дополнительно обработать полученные данные, например, изменить или удалить наборы данных, можно запросить объекты "row data gateway", они отображают набор данных в соответствующей таблице и предоставляют функции для манипулирования ними. Для этого, запрос, приведенный выше нужно изменить следующим образом:

```
1 <?php
2 $hostTable = new \Zend\Db\TableGateway\TableGateway(
3     'host',
4     $adapter,
5     new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
```

```

6 );
7
8 $results = $hostTable->select(array('hostname' => 'michaels-mac'));
9
10 foreach ($results as $result) {
11     var_dump($result);
12 }

```

Listing 14.12

При создании объекта TableGateway мы передаем RowGatewayFeature в качестве дополнительного параметра. В результате этого, TableGateway больше не создает отдельные результаты запроса, доступные, как массив или ArrayObjects, а вместо этого создает коллекции объектов RowGateway. Последние предоставляют методы save() и delete(), которые позволяют вернуть изменения в наборе данных в базу данных или даже удалить прежние:

```

1 <?php
2 $hostTable = new \Zend\Db\TableGateway\TableGateway(
3     'host',
4     $adapter,
5     new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6 );
7
8 $results = $hostTable->select(array('ip' => '127.0.0.1'));
9 $result = $results->current();
10 $result->hostname = 'michaels-macbook';
11 $result->save(); // или: $result->delete();

```

Listing 14.13

Все объекты Row Data Gateway, возвращаемые в результате запроса, относятся к типу Zend\Db\RowGateway\RowGateway. Таким образом, они дают дополнительные преимущества, которые могут обеспечить персистентность, но с другой стороны, они не несут никакой специальной информации. В этом случае в игру вступает так называемый "гидратор", в который данные загружаются из базы данных и прозрачно передаются в специальный объект. [В Zend Framework 2 под гидрацией понимают заполнение объекта набором данных. Zend\Stdlib\Hydrator – это простой компонент, предоставляющий механизм как для гидратации объекта (заполнения данными), так и для извлечения из него набора данных. Перевод документации по гидратору <http://zf2.com.ua/doc/65>] В процессе объект, несомненно, теряет свои функции персистентности, но в операциях чтения можно легко первоначально обойтись и без них.

Мы начнем с того, что создадим специальный объект, так называемую "сущность" (entity), в модуле Helloworld – /src/Helloworld/Entity/Host.php:

```

1 <?php
2 namespace Helloworld\Entity;
3

```

```

4 class Host
5 {
6     protected $ip;
7     protected $hostname;
8
9     public function getHostname()
10    {
11        return $this->hostname;
12    }
13
14    public function getIp()
15    {
16        return $this->ip;
17    }
18 }

```

Listing 14.14

Изначально у Host есть только два protected - свойства и "геттера". Следующий запрос помещает результаты в объект типа host, которые затем сохраняются в переменной host:

```

1 <?php
2 $hostTable = new \Zend\Db\TableGateway\TableGateway(
3     'host',
4     $adapter,
5     new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6 );
7
8 $results = $hostTable->select(array('ip' => '127.0.0.1'));
9
10 $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
11     new \Zend\Stdlib\Hydrator\Reflection(),
12     new \Helloworld\Entity\Host() // объект (сущность), который будет служить прототипом
13 );
14
15 $hosts->initialize($results->toArray());

```

Listing 14.15

Результат запроса передается в HydratingResultSet [Перевод документации по HydratingResultSet <http://zf2.com.ua/doc/104>]. Для этого необходимо урегулировать, каким образом данные должны быть назначены (с помощью Reflection) и куда их поместить в конечном итоге (в экземпляр Host). Чтобы достичь этого, используется так называемый "шаблон prototype", в котором исходный объект (в данном случае созданный новый экземпляр класса Host) дублируется (клонировается) для каждого набора данных в ResultSet и снабжает их данными. Таким образом задают "объект-прототип" и получают столько клонов этого объекта (каждый инициализирован правильными данными), сколько потребуется.

Но почему новые объекты не создаются просто с помощью оператора `new`? Идея заключается в том, что у объекта (а также у полей данных, которые должны быть заполнены) могут быть дополнительные зависимости от дополнительных объектов, которые не могут быть решены автоматически и без проблем. Вместо этого просто клонируется уже настроенный объект, и проблемы, таким образом, исключаются.

В этом случае предполагается, что имена столбцов таблицы согласуются со свойствами объекта. Если это не (всегда) так, мы получаем таким образом только частично заполненные объекты.

```
1 object(Helloworld\Entity\Host)#236 (2) {
2     ["ip":protected]=> string(9) "127.0.0.1"
3     ["hostname":protected]=> NULL
4 }
```

Значение `hostname` будет пустым, если соответствующее поле в базе данных будет называться, скажем, `workstation`. Теперь можно либо переименовать поле базы данных в `hostname`, либо свойство объекта в `workstation`. Однако так как это не всегда подходит или это невозможно, мы можем вместо этого создать дочерний "гидратор", который выполнит все необходимые "отображения" (`mapping`).

```
1 <?php
2 namespace Helloworld\Mapper;
3
4 use Zend\Stdlib\Hydrator\Reflection;
5 use Helloworld\Entity\Host;
6
7 class HostHydrator extends Reflection
8 {
9     public function hydrate(array $data, $object)
10    {
11        if (!$object instanceof Host) {
12            throw new \InvalidArgumentException(
13                '$object must be an instance of Helloworld\Entity\Host'
14            );
15        }
16
17        $data = $this->mapField('workstation', 'hostname', $data);
18        return parent::hydrate($data, $object);
19    }
20
21    protected function mapField($keyFrom, $keyTo, array $array)
22    {
23        $array[$keyTo] = $array[$keyFrom];
24        unset($array[$keyFrom]);
25        return $array;
26    }
27 }
```

Listing 14.16

Этот класс находится в каталоге `src/Helloworld/Mapper/HostHydrator.php` и, соответственно, отображает (maps) поле `workstation` на поле `hostname`. Теперь нужно изменить только вызов, чтобы применить соответствующий "гидратор":

```
1 <?php
2 $hostTable = new \Zend\Db\TableGateway\TableGateway(
3     'host',
4     $adapter,
5     new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6 );
7
8 $results = $hostTable->select(array('ip' => '127.0.0.1'));
9
10 $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
11     new \Helloworld\Mapper\HostHydrator(),
12     new \Helloworld\Entity\Host()
13 );
14
15 $hosts->initialize($results->toArray());
16 var_dump($hosts->current());
```

Listing 14.17

Теперь загрузка данных снова действует, как требуется:

```
1 object(Helloworld\Entity\Host)#236 (2) {
2     ["ip":protected]=> string(9) "127.0.0.1"
3     ["hostname":protected]=> string(12) "michaels-mac"
4 }
```

Организация запросов к базе данных

До сих пор мы выполняли соединение с базой данных и запросы к базе данных непосредственно в контроллере. Для демонстрационных целей это, конечно, было приемлемо. Однако, фактическое применение требует лучшего образа действий с запросами к базе данных. Рекомендуется следующая последовательность действий: создание адаптера базы данных, который позволяет получить доступ к базе данных, сместить в `ServiceManager`, параметры соединения сохранить в файле конфигурации, и запросы данных инкапсулировать в отдельные таблицы или сущности, соответственно, в специальные объекты.

Создание адаптера базы данных с помощью `ServiceManager`

[перевод документации по адаптеру <http://zf2.com.ua/doc/102>]

Для того, чтобы подготовить `ServiceManager` для создания адаптера базы данных, мы сначала модифицируем `module.config.php` модуля `Helloworld` следующим образом:

```
1 <?php
2 // [...]
3 'service_manager' => array(
4     'factories' => array(
5         'Zend\Db\Adapter\Adapter' => function ($sm) {
6             $config = $sm->get('Config');
7             $dbParams = $config['dbParams'];
8
9             return new Zend\Db\Adapter\Adapter(array(
10                 'driver' => 'Pdo_Mysql',
11                 'dsn' =>
12                     'mysql:dbname=' . $dbParams['database']
13                     . ';host=' . $dbParams['hostname'],
14                 'database' => $dbParams['database'],
15                 'username' => $dbParams['username'],
16                 'password' => $dbParams['password'],
17                 'hostname' => $dbParams['hostname'],
18             ));
19         },
20     ),
21 )
22 // [...]
```

Listing 14.18

Любые другие уже существующие определения сервисов, естественно, должны быть сохранены при необходимости. Кстати, модуль, в котором должен быть определен этот сервис, выбирается абсолютно произвольно. Так же, как было показано выше для модуля `Helloworld`, мы можем разместить его в некоторых случаях в модуле `Application`. Если необходимо использовать адаптер базы данных в целом ряде "функциональных модулей", то целесообразно переместить определение в модуль `Application`, потому что известно "по соглашению", где нужно смотреть, когда ищешь определение сервиса, который используется в нескольких модулях.

Вышеупомянутая функция обратного вызова создает адаптер базы данных и для этого запрашивает параметры подключения, которые хранятся в файле конфигурации. Для этой цели используется файл `dev1.local.php`:

```
1 <?php
2 return array(
3     'dbParams' => array(
4         'database' => 'app',
5         'username' => 'root',
6         'password' => '',
7         'hostname' => 'localhost',
8     )
9 )
```

```
9 );
```

Listing 14.19

Таким образом, можно возложить ответственность на ServiceManager за создание адаптера базы данных по требованию, вместо

```
1 <?php
2 $adapter = new \Zend\Db\Adapter\Adapter(
3     array(
4         'driver' => 'Pdo_Mysql',
5         'database' => 'app',
6         'username' => 'root',
7         'password' => ''
8     )
9 );
```

Listing 14.20

Расположенный, например, в контроллере, вызов выглядит следующим образом:

```
1 <?php
2 $adapter = $this->getServiceLocator()->get('Zend\Db\Adapter\Adapter');
```

Listing 14.21

Капсулирование подобных запросов

Чтобы избежать рассеивания запросов к базе данных по всему приложению и для предотвращения изменения схем данных или даже во избежание того, чтобы обслуживание самого запроса стало кошмаром, желательно самого начала консолидировать все запросы, которые относятся к одной и той же сущности или таблице базы данных, в одном месте. Для этого эффективно использование TableGateway, с которым мы уже знакомы и который обеспечивает простой доступ к таблицам базы данных. Мы создаем специальный TableGateway для каждой сущности. В этом контексте мы ориентируемся на соответствующие сущности:

```
1 <?php
2 namespace Helloworld\Mapper;
3
4 use Helloworld\Entity\Host as HostEntity;
5 use Zend\Stdlib\Hydrator\HydratorInterface;
6 use Zend\Db\TableGateway\TableGateway;
7 use Zend\Db\TableGateway\Feature\RowGatewayFeature;
8
9 class Host extends TableGateway
10 {
11     protected $tableName = 'host';
12     protected $idCol = 'id';
13     protected $entityPrototype = null;
14     protected $hydrator = null;
15 }
```

```

16     public function __construct($adapter)
17     {
18         parent::__construct($this->tableName,
19             $adapter,
20             new RowGatewayFeature($this->idCol)
21         );
22
23         $this->entityPrototype = new HostEntity();
24         $this->hydrator = new HostHydrator();
25     }
26
27     public function findByIp($ip)
28     {
29         return $this->hydrate(
30             $this->select(array('ip' => $ip))
31         );
32     }
33
34     public function hydrate($results)
35     {
36         $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
37             $this->hydrator,
38             $this->entityPrototype
39         );
40
41         return $hosts->initialize($results->toArray());
42     }
43 }

```

Listing 14.22

Мы сохраняем этот код для отображения (mapper) в `src/Helloworld/Mapper/Host.php` и, следовательно, в том же каталоге, где уже также находится наш гидратор, и который мы также снова используем в этом случае.

Естественно, этот код может быть оптимизирован, в этом нет никаких сомнений. Например, было бы более целесообразно выполнять инъекцию всех конфигурационных значений и зависимостей, которые в нем жестко закодированы. Но для данного примера достаточно и просто этого. Можно сделать это еще проще самостоятельно, опираясь на готовые к использованию `AbstractDbMapper` (<https://github.com/ZF-Commons/ZfcBase/blob/master/src/ZfcBase/Mapper/AbstractDbMapper.php>) из ZF-Commons (<https://github.com/ZF-Commons>) Git-репозитория. Эта работа уже сделана, частично в "главном офисе" (репозиторий поддерживается разработчиками, которые также непосредственно вовлечены в работу над фреймворком). В любом случае будет очень полезно взглянуть на него.

Вызов, по-прежнему запрашиваемый в контроллере, уже очень компактен:

```
1 <?php
```



```

2 $adapter = $this->getServiceLocator()->get('Zend\Db\Adapter\Adapter');
3 $mapper = new \Helloworld\Mapper\Host($adapter);
4 $hosts = $mapper->findByIp('127.0.0.1');

```

Listing 14.23

Мы могли бы сократить код еще больше, если бы могли выполнить инъекцию адаптера автоматически при создании отображения (mapper):

```

1 <?php
2 // [...]
3 'service_manager' => array(
4     'factories' => array(
5         'Zend\Db\Adapter\Adapter' => function ($sm) {
6             $config = $sm->get('Config');
7             $dbParams = $config['dbParams'];
8
9             return new Zend\Db\Adapter\Adapter(array(
10                 'driver' => 'pdo',
11                 'dsn' =>
12                     'mysql:dbname=' . $dbParams['database']
13                     . ';host=' . $dbParams['hostname'],
14                 'database' => $dbParams['database'],
15                 'username' => $dbParams['username'],
16                 'password' => $dbParams['password'],
17                 'hostname' => $dbParams['hostname'],
18             ));
19         },
20         'Helloworld\Mapper\Host' => function ($sm) {
21             return new \Helloworld\Mapper\Host(
22                 $sm->get('Zend\Db\Adapter\Adapter')
23             );
24         }
25     ),
26 ),
27 // [...]

```

Listing 14.24

Теперь вызов, по-прежнему запрашиваемый в контроллере, можно выполнить одной строкой:

```

1 <?php
2 $hosts = $this->getServiceLocator()
3     ->get('Helloworld\Mapper\Host')->findByIp('127.0.0.1');

```

Listing 14.25

За счет использования TableGateway мы теперь можем элегантно решить проблему с сущностями, теряющими свою способность к персистентность вследствие гидратации. Однако, изменение сущности уже не может быть непосредственно записано в базу данных с использованием save(). Теперь мы поручаем это сделать отображению Host mapper:

```

1  <?php
2
3  namespace Helloworld\Mapper;
4
5  use Helloworld\Entity\Host as HostEntity;
6  use Zend\Stdlib\Hydrator\HydratorInterface;
7  use Zend\Db\TableGateway\TableGateway;
8  use Zend\Db\TableGateway\Feature\RowGatewayFeature;
9  use Zend\Db\Sql\Sql;
10 use Zend\Db\Sql\Insert;
11
12 class Host extends TableGateway
13 {
14     protected $tableName = 'host';
15     protected $idCol = 'id';
16     protected $entityPrototype = null;
17     protected $hydrator = null;
18
19     public function __construct($adapter)
20     {
21         parent::__construct($this->tableName,
22             $adapter,
23             new RowGatewayFeature($this->idCol)
24         );
25
26         $this->entityPrototype = new HostEntity();
27         $this->hydrator = new HostHydrator();
28     }
29
30     public function findByIp($ip)
31     {
32         return $this->hydrate(
33             $this->select(array('ip' => $ip))
34         );
35     }
36
37     public function hydrate($results)
38     {
39         $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
40             $this->hydrator,
41             $this->entityPrototype
42         );
43
44         return $hosts->initialize($results->toArray());
45     }
46
47     public function insert($entity)
48     {

```

```

49         return parent::insert($this->hydrator->extract($entity));
50     }
51
52     public function updateEntity($entity)
53     {
54         return parent::update(
55             $this->hydrator->extract($entity),
56             $this->idCol . "=" . $entity->getId()
57         );
58     }
59 }

```

Listing 14.26

Новые методы здесь - `insert()` и `updateEntity()`. Однако мы должны также расширить `HostHydrator`, если имена столбцов в таблице базы данных не будет соответствовать свойствам объекта:

```

1  <?php
2  namespace Helloworld\Mapper;
3
4  use Zend\Stdlib\Hydrator\Reflection;
5  use Helloworld\Entity\Host as HostEntity;
6
7  class HostHydrator extends Reflection
8  {
9      public function hydrate(array $data, $object)
10     {
11         if (!$object instanceof HostEntity) {
12             throw new \InvalidArgumentException(
13                 '$object must be an instance of Helloworld\Entity\Host'
14             );
15         }
16
17         $data = $this->mapField('workstation', 'hostname', $data);
18         return parent::hydrate($data, $object);
19     }
20
21     public function extract($object)
22     {
23         if (!$object instanceof HostEntity) {
24             throw new \InvalidArgumentException(
25                 '$object must be an instance of Helloworld\Entity\Host'
26             );
27         }
28
29         $data = parent::extract($object);
30         $data = $this->mapField('hostname', 'workstation', $data);
31         return $data;

```

```

32     }
33
34     protected function mapField($keyFrom, $keyTo, array $array)
35     {
36         $array[$keyTo] = $array[$keyFrom];
37         unset($array[$keyFrom]);
38         return $array;
39     }
40 }

```

Listing 14.27

Теперь мы можем изменить ранее загруженный набор данных в контроллере:

```

1 <?php
2 $hosts = $this->getServiceLocator()
3     ->get('Helloworld\Mapper\Host')->findByIp('127.0.0.1');
4
5 $host = $hosts->current();
6 $host->setHostname('my-mac');
7
8 $this->getServiceLocator()
9     ->get('Helloworld\Mapper\Host')->updateEntity($host);

```

Listing 14.28

Вставка нового набора данных с помощью Host-Mapper функционирует теперь аналогично:

```

1 <?php
2 $newEntity = new \Helloworld\Entity\Host();
3 $newEntity->setHostname('michaels-iphone');
4 $newEntity->setIp('192.168.1.56');
5
6 $this->getServiceLocator()
7     ->get('Helloworld\Mapper\Host')->insert($newEntity);

```

Listing 14.29

Ради полноты изложения, вот теперь Host-Entity в своем окончательном виде:

```

1 <?php
2 namespace Helloworld\Entity;
3
4 class Host
5 {
6     protected $id;
7     protected $ip;
8     protected $hostname;
9
10    public function getHostname()
11    {
12        return $this->hostname;
13    }

```

```

14
15     public function getIp()
16     {
17         return $this->ip;
18     }
19
20     public function setIp($ip)
21     {
22         $this->ip = $ip;
23     }
24
25     public function setHostname($hostname)
26     {
27         $this->hostname = $hostname;
28     }
29
30     public function setId($id)
31     {
32         $this->id = $id;
33     }
34
35     public function getId()
36     {
37         return $this->id;
38     }
39 }

```

Listing 14.30

В частности, параметр `id` также по-прежнему важен, поскольку набор данных, который обновляется, определяется с его помощью.

Альтернатива Zend\Db Doctrine 2 ORM

Если говорить абсолютно честно, Zend\Db действительно полезен, но быстро достигает своего предела. Задача "персистентности" не может быть удовлетворительно решена в сложных системах с его помощью. Zend\Db не предоставляет реализацию "active record" [от переводчика: что, возможно, и к лучшему ☺] или функциональности ORM [☹], который в любом случае обеспечивает управляемость приложения. Кроме того, необходимо писать самостоятельно большой объем кода для обеспечения персистентности. Поэтому я категорически рекомендую вам взглянуть на Doctrine 2. Zend Framework и Doctrine 2 великолепно дополняют друг друга и действительно полностью выявляют свои сильные стороны при совместном использовании.

[Здесь я хочу обратить внимание читателя на то, что недавно свет увидела новая книга автора: "PHP Data Persistence with Doctrine 2 ORM" (Concepts, Techniques and Practical Solutions). Остается только сожалеть, что до русскоязычного читателя подобная литература доходит такими слабыми темпами.]

XV. Валидаторы

Стандартные валидаторы

С помощью Zend\Validator фреймворк предоставляет простой, но очень полезный механизм для проверки значения в отношении определенных требований, например, содержимого, отправленного клиентом в рамках POST-запроса. Для наиболее распространенных потребностей он также предоставляет конкретные реализации. Например, с помощью нескольких строк кода значение может быть проверено на соответствие стандарту ISBN:

```
1 <?php
2 $validator = new \Zend\Validator\Isbn();
3
4 if($validator->isValid('315000017')) {
5     echo "В порядке!";
6 }
```

Listing 15.1

Важно то, что это синтаксическая, а не семантическая проверка. Таким образом "в порядке" также отобразится, если указанный ISBN действительно корректен, но еще не был присвоен какой-либо книге.

Валидаторы, как правило, предоставляют сообщения об ошибках для очень разных ошибочных ситуаций, которые могут быть получены после процедуры проверки посредством `getMessages()`:

```
1 <?php
2 $validator = new \Zend\Validator\Isbn();
3
4 if($validator->isValid('315090017')) {
5     echo "В порядке!";
6 } else {
7     foreach ($validator->getMessages() as $messageId => $message) {
8         echo $message;
9     }
10 }
```

Listing 15.2

В данном случае мы увидим на экране:

```
1 The input is not a valid ISBN number
```

Однако если вызвать:

```
1 <?php
2 $validator = new \Zend\Validator\Isbn();
3
4 if($validator->isValid('12.23')) {
```

```

5     echo "В порядке!";
6 } else {
7     foreach ($validator->getMessages() as $messageId => $message) {
8         echo $message;
9     }
10 }

```

Listing 15.3

это даст следующее:

```

1 Invalid type given. String or integer expected

```

Таким образом, валидатор Isbn охватывает два различных случая ошибки. Как и можно было ожидать, отдельные сообщения об ошибке можно установить с помощью метода `setMessage()`:

```

1 <?php
2 $validator = new \Zend\Validator\Isbn();
3
4 $validator->setMessage(
5     'Для проверки это должно быть строкой или целым значением!',
6     \Zend\Validator\Isbn::INVALID
7 );
8
9 if($validator->isValid(12.23)) {
10     echo "В порядке!";
11 } else {
12     foreach ($validator->getMessages() as $messageId => $message) {
13         echo $message;
14     }
15 }

```

Listing 15.4

Для этого необходимо уточнить ключ сообщения, который используется валидатором. У фреймворка есть следующие интуитивно-понятные валидаторы:

[Логично, что автор не пытается дублировать официальную документацию. Более подробно об этих валидаторах можно узнать в переводах официальной документации <http://zf2.com.ua/doc/172>]

- Alnum (буквенно-цифровой)
- Alpha (буквенный)
- Barcode (штрихкод)
- Between (диапазон)
- Callback (обратный вызов)
- CreditCard (кредитная карта)
- Date (дата)

- Db\RecordExists (наличие записи в БД) и Db\NoRecordExists (отсутствие записи в БД)
- Digits (цифры)
- EmailAddress (электронный адрес)
- GreaterThan (больше, чем)
- Hex (шестнадцатеричные символы)
- Hostname (имя хоста)
- Iban (международный номер банковского счета)
- Identical (идентичность)
- InArray (в массиве)
- Ip
- ISBN (универсальный номер книжного издания)
- LessThan (меньше, чем)
- NotEmpty (не пусто)
- PostCode (почтовый индекс)
- Regex (регулярное выражение)
- Sitemap Validators (валидатор карты сайта)
- Step (шаг)

Выполнение нескольких проверок одновременно

С помощью класса `Zend\Validator\ValidatorChain` несколько классов могут быть при необходимости объединены для последовательной проверки.

(<http://zf2.com.ua/doc/174>)

Создание собственных валидаторов

Если требуются дополнительные функции проверки, можно легко написать собственные валидаторы, если они будут наследовать `AbstractValidator`:

```
1 <?php
2 class HelloWorld\Validator\Float extends Zend\Validator\AbstractValidator
3 {
4     const FLOAT = 'float';
5
6     protected $messageTemplates = array(
7         self::FLOAT => "'%value%' не является числом с плавающей запятой."
8     );
9
10    public function isValid($value)
```



```

11  {
12      $this->setValue($value);
13
14      if (!is_float($value)) {
15          $this->error(self::FLOAT);
16          return false;
17      }
18
19      return true;
20  }
21 }

```

Listing 15.5

[На самом деле, у фреймворка есть стандартный валидатор для чисел с плавающей запятой. Однако поскольку он зависит от заданной локали и принадлежит пространству имен `Zend\I18n\Validator`, документация для него (так же, как и, скажем, для валидаторов файлов) была вынесена в отдельный раздел:

(<http://framework.zend.com/manual/2.0/en/modules/zend.i18n.validators.html>)

Дополнительную информацию по созданию собственных валидаторов можно найти на <http://zf2.com.ua/doc/175>

]

XVI. Формы

Формы являются неотъемлемой частью web-приложений: они являются основной возможностью пользователя для передачи данных на сервер, т.е. приложению. В основном с формами у разработчика приложений связаны разнообразные сложные задачи. Всесторонняя поддержка форм является неотъемлемой частью хорошего веб-фреймворка, что в некоторой степени изгоняет наш страх перед ними. С помощью Zend\Form Zend Framework 2 обеспечивает высокопроизводительное решение, которое может сделать почти все, что пожелаешь. Даже если это не очень легко понять.

Прежде всего нам понадобится немного теории: каждая форма принципиально основана на одном или нескольких объектах типа Zend\Form\Element. Они формируют основные элементы форм, основанных на Zend\Form. В этом контексте один элемент изначально соответствует, по существу, обычному элементу знакомой нам уже HTML-формы, т.е. полю ввода (text), радио кнопке (radio), списку выбора (select) и т.д. Фильтр, который следует использовать для соответствующего элемента входных данных, и правила, которые должны использоваться для валидации этого элемента входных данных, можно задать с помощью так называемого "Input" как только данные будут получены [для каждого Input задается имя, соответствующее имени (name) элемента формы]. Типичными фильтрами являются, например, StripTags, который удаляет все HTML-теги из введенной строки символов, или StringToLower, который (как и можно ожидать) преобразует все символы в строчные буквы. Валидация подразумевает проверку исходных данных на соответствие ранее определенным условиям. Так, например, валидатор ISBN может использоваться для поля ввода, чтобы проверить данные на соответствие номеру ISBN, тогда как NotEmpty проверяет, было ли хоть что-либо введено в поле. Отдельные "Inputs" объединены в так называемый "InputFilter" и он назначается соответствующей веб-форме. Отдельные элементы формы могут, в свою очередь, быть семантически сгруппированы в так называемые "fieldsets". Кстати, с технической точки зрения, Zend\Form\Fieldset также является "элементом" в этом контексте, наследующим классу Zend\Form\Element. "Fieldsets" или отдельные элементы формы (или все вместе) затем, в свою очередь, объединяются для формирования Zend\Form\Form, который снова технически является только "элементом" – он также наследует классу Zend\Form\Element.

[Два раза прочитал свой перевод и ничего не понял. Хотел было уже выложить в оригинале на английском, но раз уж я начал комментировать, то продолжу, чтобы как-то компенсировать недостатки перевода. Суть состоит в том, чтобы в сценарии представления, в файле *.phtml, не писать никакого HTML-кода для формы или ее элементов. Т.е. в сценарии представления мы просто пишем

```
<?php echo $this->formRow($this->form->get('username')) ?>
```

и выводится `<input type="text" id="username" class="my-input-text" />`

Для этого должен существовать отдельный от сценария представления класс, в котором элементы формы настраиваются программно с помощью PHP, а не HTML. Эта такая

своеобразная "модель формы", к которой больше привыкли программисты десктопных приложений, чем веб-дизайнеры. Эта "модель формы" (пусть специалисты не плюются, я просто упрощаю) может называться как угодно, скажем, MyForm, но наследоваться должна от Zend\Form\Form. Практически, нас в этой "модели формы" поначалу интересует только конструктор. В конструкторе мы для каждого элемента, тех же "username" или "password", добавляем в данную форму параметры элементов в виде массивов методом `$this->add(array(...))`, и таким образом создаем элементы формы с заданными параметрами. Т.е. те же инпуты, селекты и чекбоксы. Каждый вызов `add()` добавляет один элемент. По большому счету, для отображения формы – это все, что нужно знать. После того, как мы это сделаем, мы можем спокойно выводить с помощью `echo` отдельные элементы формы из сценария представления, нужно только указать в сценарии представления то имя элемента, которое мы задали ему в "модели формы". Автор подчеркивает, что все элементы формы, `fieldset`, и сама наша форма наследуются от `Zend\Form\Element`. Да, это верно, но... прозрачно для пользователя, т.е. мы добавляем методом `add()` массив, а то, что он потом становится конкретным элементом посредством метода `add()` – это уже детали.

Таким образом, наша форма – это модель со вложенными элементами, коллекция. Но эти элементы нужно как-то проверять. И для этого существует другая коллекция – `InputFilter`. Сейчас специалисты начнут плевать еще больше, но я сравню `InputFilter` с коллекцией контроллеров. Точно так же, как и для формы с ее элементами, в конструкторе `MyInputFilter` мы добавляем фильтры и валидаторы. Точно так же `MyInputFilter` должен наследовать стандартному классу, в данном случае - классу `InputFilter`. При этом, добавляя фильтры, мы должны в массиве указать имя – к какому элементу формы эти фильтры должны применяться, например, "username". Таким образом, одному элементу формы мы можем назначить много фильтров и валидаторов, указав каждому, скажем, наш "username". Да, и пусть слово "Filter" в `InputFilter` вас не обманывает, с его помощью мы добавляем и валидаторы тоже. И то, что наши массивы, добавляющиеся с помощью метода `add()` нашего `MyInputFilter`, превращаются в объекты `Input`, это тоже только технические подробности. Ничего не мешает вам вместо простого массива запросить фабрику `InputFactory` на создание конкретного элемента `Input` и потом передать его вместо массива в `add()` (чтобы все видели, как у нас все здесь сурово ☺).

И в заключении мы в конструкторе нашей `MyForm` назначаем нашей форме наши же фильтры:

```
$this->setInputFilter(new HelloWorld\Form\MyInputFilter());
```

Вот и все. Все остальное делает фреймворк.

С одной стороны все это довольно удобно. Все элементы настраиваются программно. Все стандартизировано. Если вы ожидали в текстовом поле цифры, а придут буквы – валидатор этого так не оставит, в контроллере вашего модуля вы проверите `$form->isValid()`, кроме того, возле конкретного элемента формы автоматически будет напечатана причина, по которой валидатору что-то не понравилось. Но есть несколько "но". Во-первых, элементов формы (как и валидаторов и фильтров) достаточно много, автор не дублирует

официальную документацию, так что придется поискать их параметры. Их много и параметры у них несколько отличаются. (Правда, все они есть на сайте <http://zf2.com.ua>) Во-вторых, мы имеем, по сути, три файла – сценарий представления, класс MyForm и класс MyInputFilter. Я, как человек ленивый, считаю, что три файла – это очень много. Правда, есть выход – в примере быстрого старта "Album" нет отдельного класса для MyInputFilter, там у модели AlbumModel\Album есть метод getInputFilter(), который просто возвращает настроенный экземпляр InputFilter (<http://zf2.com.ua/doc/37>) (и там все сурово ☺). С другой стороны, наверное, так и должно быть – каждый класс делает только то, что должен делать, не больше, но и не меньше; а поскольку у нас для каждого класса свой файл...

Можно добавить, что каждому конкретному типу элемента формы, будь то инпут или чекбокс, соответствует свой помощник представления, который и позволяет отобразить данный тип элемента. В противном случае мы писали бы не

```
<?php echo $this->formRow($this->form->get('username')) ?>
```

а

```
<?php echo this->form->get('username') ?>
```

что на практике будет не верно, поскольку мы должны обратиться к помощнику представления, в данном случае – formRow, а не к объекту типа Zend\Form\Element. Кстати, в большинстве случаев мы и будем обращаться именно к этому помощнику для большинства элементов формы, а не к соответствующему помощнику конкретного элемента. Во-первых, это упростит нашу задачу, поскольку formRow сам разберется, какой там помощник ему нужен в данном конкретном случае. Во-вторых, formRow выведет не только элемент формы, но и присвоенную ему label (<label for="username">). И в-третьих, при наличии ошибка валидации, formRow выведет описание этой ошибки.

Вот, в принципе, основная логика форм. Т.е., хлопотно, но уж никак не сложно. Так что предоставляю слово автору...

]

Подготовка формы

Форму можно быстро создать с помощью HTML:

```
1 <form action="#" method="post">
2 <fieldset>
3 <label for="username">Full Name:</label>
4 <input type="text" id="username" />
5 <label for="email">Email Address:</label>
6 <input type="email" id="email" />
7 <input type="submit" value="Register now" />
8 </fieldset>
9 </form>
```

Listing 16.1

Однако, опытный разработчик приложений знает, что это не означает, что достигнуто все, что нужно – отнюдь нет: полученные данные должны быть синтаксически и семантически проверены для дальнейшей обработки – например, для сохранения в базе данных – и ошибочные ситуации должны быть обработаны определенным образом, не в последнюю очередь потому, что проверка входных данных, отправленных клиентом, даже просто из соображений безопасности никогда не бывает адекватной. Весь этот код нужно программировать вручную и соответствующий код реализовывать, например, в контроллере. И фреймворк может помочь в этом случае. Основная идея состоит в реализации формы в виде отдельного структурного компонента, который содержит всю информацию о содержимом полей и их синтаксических и семантических условиях, а также необходимую информацию о том, как ввести данные и впоследствии выдать наилучшим образом. Во фреймворке для работы с формами используется ряд различных классов и компонентов.

- `Zend\Form`: основные компоненты, с помощью которых разработчики взаимодействуют с формами.
- `Zend\InputFilter`: расширяет формы возможностью фильтрации и валидации.
- `Zend\View\Helper`: обеспечивает визуальное представление формы.
- `Zend\Stdlib\Hydrator`: обеспечивает для форм возможность автоматической передачи данных из форм в другие объекты или импортирование их оттуда.

Фреймворк предоставляет разработчикам ряд решений относительно того, как форма должна быть структурирована. В этом контексте маятник колеблется, как это часто бывает в `Zend Framework 2`, между написанием кода и записью конфигурации. Кроме этого, существует широкий набор возможностей для работы с формами. Достоинством этого подхода является сопоставление форм посредством собственных классов форм. Если бы мы хотели отобразить форму, приведенную выше, для регистрации на рассылку новостей посредством объекта `Zend\Form`, мы должны были бы определить форму в файле `src/Helloworld/Form/SignUp.php`.

```
1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5
6 class SignUp extends Form
7 {
8     public function __construct()
9     {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
```

```

15         'name' => 'name',
16         'attributes' => array(
17             'type' => 'text',
18             'id' => 'name'
19         ),
20         'options' => array(
21             'label' => 'Full Name:'
22         ),
23     ));
24
25     $this->add(array(
26         'name' => 'email',
27         'attributes' => array(
28             'type' => 'email',
29             'id' => 'email'
30         ),
31         'options' => array(
32             'label' => 'Email Address:'
33         ),
34     ));
35
36     $this->add(array(
37         'name' => 'submit',
38         'attributes' => array(
39             'type' => 'submit',
40             'value' => 'Register now'
41         ),
42     ));
43 }
44 }

```

Listing 16.2

Новые элементы или `fieldsets` добавляются в форму методом `add()`. Для этого метод `add()` вызывает в свою очередь `Zend\Form\Factory`; последний, в свою очередь, знает, как интерпретировать спецификацию (называемую "Spec"), которая передается в виде массива, и как сгенерировать соответствующий элемент. Наиболее важными компонентами спецификации являются `name`, `type`, `attributes` и `options`. Имя (`name`) может быть выбрано произвольно, а тип (`type`) представляет собой тип элемента, который действительно существует во фреймворке (в данном случае генерируется элемент типа `Zend\Form\Element\Text` потому, что ничего другого не было указано, это значение по умолчанию). Параметр `attributes` устанавливает все атрибуты сгенерированного в конечном счете элемента (`<input name="name" type="text" id="name">`). В разделе `options` можно определить `label` (т.е. то, что дополнительно отображается до или после основного поля ввода) посредством параметра `label` или же с помощью `label_attributes` соответственно.

Помимо отдельных элементов, ряд атрибутов разработчики могут задать с помощью `setAttribute()`, в том числе `action` и `method`, которые абсолютно необходимы для того, чтобы форму можно было впоследствии отправить.

Отображение форм

Хорошая новость для всех, кто использовал 1 версию фреймворка: больше не существует декораторов в любом виде. Декораторы форм были довольно совершенны, но в то же время подход, который позволял отобразить форму посредством вложенности объектов представления, каждый из которых затем генерировал небольшую часть окончательной HTML разметки, был довольно сложен. Я считаю, что декораторы форм были лишь наиболее трудным аспектом Zend Framework 1, и в остальном создавали большое количество проблем, которые нельзя было предусмотреть заранее. Но хватит вспоминать, они больше не существуют во 2 версии. Преимущество очевидно: простое использование форм, но менее компактный код, если просто использовать средства, доступные в программе. Для отображения формы, определенной выше, нужно несколько пересмотреть код `<?php echo $this->form; ?>`, которого было достаточно в прежние времена. Тем не менее, это также может быть реализовано с небольшими усилиями, как мы увидим в практической части книги.

```
1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRow($this->form->get('name'));
5 echo $this->formRow($this->form->get('email'));
6 echo $this->formSubmit($this->form->get('submit'));
7 echo $this->form()->closeTag();
```

Listing 16.3

Этот код передает в файл представления (View) из действия (Action) созданный в нем экземпляр класса Form, который будет возвращен в рамках ViewModel:

```
1 <?php
2 return new ViewModel(
3     array(
4         'form' => new \Helloworld\Form\SignUp()
5     )
6 );
```

Listing 16.4

Ряд помощников представления элементов и самой формы используются для отображения. Грубо говоря, есть соответствующие ViewHelper, которые могут быть использованы для отображения, для каждого типа элемента формы, который определяет HTML. Более того, существует ряд дополнительных вспомогательных конструкций, таких как FormRow, который обеспечивает отображение поля label, самого поля формы и, при необходимости, существующие сообщения об ошибке в соответствующей "последовательности". С

помощниками представления почти все ясно. Главное – вызвать `prepare()` до того, как обращаться к любым другим элементам; в противном случае это приводит к ошибке.

Обработка элементов форм

В принципе, есть две возможности реализации обработки форм: либо в том же действии (Action), в котором пустая форма была создана, либо в отдельном действии. Если обработка будет происходить в том же действии, можно использовать "проверку `isPost()`", чтобы определить, нужно ли отображать форму или обработать ее элементы:

```
1 <?php
2 if ($this->getRequest()->isPost()) {
3     // Обработка формы
4 } else {
5     return new ViewModel(
6         array(
7             'form' => new \Helloworld\Form\SignUp()
8         )
9     );
10 }
```

Listing 16.5

Чтобы получить доступ к отправленным данным, теперь можно или напрямую получить данные из POST:

```
1 <?php
2 $form = new \Helloworld\Form\SignUp();
3
4 if ($this->getRequest()->isPost()) {
5     $data = $this->getRequest()->getPost();
6     var_dump($data); exit;
7 } else {
8     return new ViewModel(
9         array(
10             'form' => $form
11         )
12     );
13 }
```

Listing 16.6

в результате выводится следующее:

```
1 class Zend\Stdlib\Parameters#74 (3) {
2     public $name => string(13) "Michael Romer"
3     public $email => string(24) "zf2buch@michael-romer.de"
4     public $submit => string(9) "Register now"
5 }
```


или получить отправленные данные из объекта Form. Однако это возможно только в том случае, если данные предварительно пройдут валидацию с помощью формы.

Валидация элементов формы

До сих пор преимущество Zend\Form было относительно незатейливым, но теперь мы собираемся взлетать. Если мы получим отправленные данные не из массива POST PHP, а наоборот, из самой формы, элементы могут быть автоматически проверены простым способом и данные отфильтрованы.

Для этого для начала должен быть определен так называемый "InputFilter". Термин "InputFilter" немного вводит в заблуждение, потому что не только фильтры, но и валидаторы могут быть заданы таким образом. Фильтры изменяют входные данные по мере необходимости, в то время, как валидаторы проверяют данные на соответствие определенным условиям, например, максимальной длине строки.

Существует несколько способов определения "InputFilter". Например, определения могут быть помещены в собственном классе или же они стали доступны посредством метода getInputFilter() класса формы. В этом случае мы реализуем класс "InputFilter" в собственном классе в файле src/Helloworld/Form/SignUpFilter.php:

```
1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5 use Zend\InputFilter\InputFilter;
6
7 class SignUpFilter extends InputFilter
8 {
9     public function __construct()
10     {
11         $this->add(array(
12             'name' => 'email',
13             'required' => true,
14             'validators' => array(
15                 array(
16                     'name' => 'EmailAddress'
17                 )
18             ),
19         ));
20
21         $this->add(array(
22             'name' => 'name',
23             'required' => true,
24             'filters' => array(
25                 array(
26                     'name' => 'StringTrim'
```

```

27         )
28     )
29     ));
30 }
31 }

```

Listing 16.7

В этом случае отдельные Inputs добавляются в InputFilter методом `add()`. В этом контексте Input соответствует элементу формы, т.е. является неким элементом Input, который ссылается на конкретное имя [т.е., элемент формы с таким именем] и впоследствии учитывается [при фильтрации элемента с таким именем]. Так или иначе, обязательное поле обозначается с помощью `required`. Аналогично, валидаторы и фильтры можно определить с помощью `validators` и `filters`, где `name` должно соответствовать фильтру или валидатору, поставляемому фреймворком, или самостоятельно добавленному фильтру или валидатору соответственно. Просмотрев исходный код `Zend\Validator\ValidatorPluginManager`, можно увидеть стандартные валидаторы и их символические имена, с помощью которых они могут быть получены / вызваны. `Zend\Filter\FilterPluginManager` предоставляет информацию о фильтрах фреймворка.

Теперь мы должны расширить определение формы с помощью `SignUpFilter`.

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5
6 class SignUp extends Form
7 {
8     public function __construct()
9     {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13         $this->setInputFilter(new \Helloworld\Form\SignUpFilter());
14
15         $this->add(array(
16             'name' => 'name',
17             'attributes' => array(
18                 'type' => 'text',
19             ),
20             'options' => array(
21                 'id' => 'name',
22                 'label' => 'Full Name:'
23             ),
24         ));
25
26         $this->add(array(
27             'name' => 'email',

```

```

28         'attributes' => array(
29             'type' => 'email',
30         ),
31         'options' => array(
32             'id' => 'email',
33             'label' => 'Email Address:'
34         ),
35     ));
36
37     $this->add(array(
38         'name' => 'submit',
39         'attributes' => array(
40             'type' => 'submit',
41             'value' => 'Register now'
42         ),
43     ));
44 }
45 }

```

Listing 16.8

В этом случае метод `setInputFilter()` связывает форму и `"InputFilter"`. Но и в этом случае общая рекомендация в том, что желательно не кодировать жестко применение `SignUpFilter`; вместо этого он должен быть введен с помощью соответствующей фабрики и `ServiceManager` или, в качестве альтернативы, с помощью `Zend\Di`.

Теперь обработка форм может производиться в контроллере следующим образом:

```

1 <?php
2 public function indexAction()
3 {
4     $form = new \Helloworld\Form\SignUp();
5
6     if ($this->getRequest()->isPost()) {
7         $form->setData($this->getRequest()->getPost());
8
9         if ($form->isValid()) {
10             var_dump($form->getData());
11         } else {
12             return new ViewModel(
13                 array(
14                     'form' => $form
15                 )
16             );
17         }
18     } else {
19         return new ViewModel(
20             array(
21                 'form' => $form
22             )

```

```

23     );
24 }
25 }

```

Listing 16.9

В этом случае успешный ввод даст радостный вывод:

```

1 array(2) {
2     ["email"]=> string(24) "zf2buch@michael-romer.de"
3     ["name"]=> string(13) "Michael Romer"
4 }

```

Таким образом, данные существуют теперь в форме в виде массива, готового к дальнейшей обработке. Любые фильтры, которые были зарегистрированы для соответствующих полей, уже были применены к этому времени.

Кстати, таким же образом, сообщения об ошибках теперь также уже отображены, и если это тот случай [т.е., случай ошибки], то форма не будет ратифицирована (validated), и мы возвращаем ее для повторного отображения.

Если отправить форму без записей (что будет выявлено), мы увидим следующее для обоих полей:

```

1 "Value is required and can't be empty"

```

Для того, чтобы изменить отображаемое сообщение об ошибке, используется массив `options` при определении валидатора. Тем не менее, сообщение об ошибке, показанное выше, имеет одну особенность – ведь `SignUpFilter` еще даже не использует валидатор `NotEmpty`, который мы могли бы предоставить с помощью необходимых "options" (параметров). Вместо этого требование того, что поле обязательно должно быть заполнено, выражается с помощью `'required' => true`. Основываясь на этом, фреймворк автоматически создаст соответствующий валидатор. Таким образом, выражение `'required' => true` является "ярлыком быстрого доступа". Когда мы следующим образом адаптируем фильтр, мы можем внести индивидуальные сообщения об ошибках:

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5 use Zend\InputFilter\InputFilter;
6
7 class SignUpFilter extends InputFilter
8 {
9     public function __construct()
10     {
11         $this->add(array(
12             'name' => 'email',
13             'validators' => array(
14                 array(
15                     'name' => 'NotEmpty',

```

```

16         'options' => array(
17             'messages' => array(
18                 \Zend\Validator\NotEmpty::IS_EMPTY =>
19                 'Please input something.'
20             )
21         ),
22     ),
23     array(
24         'name' => 'EmailAddress',
25         'options' => array(
26             'messages' => array(
27                 \Zend\Validator\EmailAddress::INVALID_FORMAT =>
28                 'Please enter the correct email address.'
29             )
30         )
31     ),
32 ),
33 );
34
35 $this->add(array(
36     'name' => 'name',
37     'filters' => array(
38         array(
39             'name' => 'StringTrim'
40         )
41     ),
42     'validators' => array(
43         array(
44             'name' => 'NotEmpty',
45             'options' => array(
46                 'messages' => array(
47                     \Zend\Validator\NotEmpty::IS_EMPTY =>
48                     'Please input something.'
49                 )
50             )
51         )
52     )
53 ));
54 }
55 }

```

Listing 16.10

Если теперь отправить форму, не введя адрес электронной почты, мы получим следующее сообщение об ошибке:

```

1 Please input something. // Пожалуйста, введите что-нибудь.
2 Please enter the correct email address. // Пожалуйста, введите корректный e-mail.

```

Если быть более точным: мы видим 2 сообщения об ошибке сразу. Следует знать, что в процессе валидаторы обрабатываются один за другим, и любые сообщения о появляющихся ошибках накапливаются. Чтобы избежать этого, можно установить параметр 'break_chain_on_failure' => true. Это гарантирует, что после неудачной валидации последующие не будут реализованы вообще и никакие дополнительные сообщения об ошибках не будут сгенерированы:

```
1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5 use Zend\InputFilter\InputFilter;
6
7 class SignUpFilter extends InputFilter
8 {
9     public function __construct()
10     {
11         $this->add(array(
12             'name' => 'email',
13             'validators' => array(
14                 array(
15                     'name' => 'NotEmpty',
16                     'break_chain_on_failure' => true,
17                     'options' => array(
18                         'messages' => array(
19                             \Zend\Validator\NotEmpty::IS_EMPTY =>
20                                 'Please input something.'
21                         )
22                     )
23                 ),
24                 array(
25                     'name' => 'EmailAddress',
26                     'options' => array(
27                         'messages' => array(
28                             \Zend\Validator\EmailAddress::INVALID_FORMAT =>
29                                 'Please enter the correct email address.'
30                         )
31                     )
32                 ),
33             ),
34         ));
35
36         $this->add(array(
37             'name' => 'name',
38             'filters' => array(
39                 array(
40                     'name' => 'StringTrim'
41                 )
42             )
43         ));
44     }
45 }
```

```

42         ),
43         'validators' => array(
44             array(
45                 'name' => 'NotEmpty',
46                 'options' => array(
47                     'messages' => array(
48                         \Zend\Validator\NotEmpty::IS_EMPTY =>
49                             'Please input something.'
50                     )
51                 )
52             )
53         )
54     ));
55 }
56 }

```

Listing 16.11

Поскольку отдельный валидатор может генерировать различные сообщения об ошибках в зависимости от ситуации, сообщения должны указываться с соответствующим ключом, который доступен через константу соответствующего класса валидатора. Если есть неопределенность, посмотрите исходный код соответствующего валидатора, это поможет.

Стандартные элементы формы

Как показано в предыдущих примерах, можно генерировать любой произвольный HTML элемент и выбрать соответствующие фильтры и валидаторы. Тем не менее, в зависимости от элемента, эта процедура может потребовать много усилий, в частности потому, что вся конфигурация должна быть выполнена вручную. В этой ситуации у фреймворка есть ряд доступных вариантов: большое количество "предварительно сконфигурированных" элементов, которые немедленно предоставляют необходимую конфигурацию.

- Button
- Captcha
- Checkbox
- Collection
- Color
- Csrf
- Date
- DateTime
- DateTimeLocal
- Email
- File
- Hidden
- Image
- Month

- MultiCheckbox
- Number
- Password
- Radio
- Range
- Select
- Submit
- Text
- Textarea
- Time
- URL
- Week

Возьмем, к примеру, элемент Number. Мы хотели бы создать поле ввода, в котором могут быть указаны числовые значения определенного диапазона. Если бы мы создавали эти правила сами, мы должны были бы настроить большое количество валидаторов, среди них:

- NumberValidator: могут быть введены только цифры.
- GreaterThanValidator: введенное значение должно быть больше или равно указанному минимуму.
- LessThanValidator: введенное значение должно быть меньше или равно указанному максимуму.
- StepValidator: допускаются только целочисленные значения.

Мы можем обойтись без этой работы, если сразу используем `Zend\Form\Element\Number`:

```
1 <?php
2 // [...]
3 $this->add(array(
4     'name' => 'age',
5     'type' => 'Zend\Form\Element\Number',
6     'attributes' => array(
7         'id' => 'age',
8         'min' => 18,
9         'max' => 99,
10        'step' => 1
11    ),
12    'options' => array(
13        'label' => 'How old are you?'
14    ),
15 ));
16 // [...]
```

Listing 16.12

В зависимости от браузера и от поддержки им HTML5, сразу становится очевидно, что при использовании правил проверка хорошо работает не только на сервере, но и

непосредственно в клиенте запрашиваются несоответствующие записи. Конечно же, конфигурация в массиве `attributes` не только рассматривается впоследствии соответствующим помощником представления, но также используется валидаторами.

Кстати, элемент `Number` также одновременно добавляет `Zend\Filter\StringTrim`, так что это не нужно программировать самостоятельно.

Набор полей формы (Fieldsets)

Более внимательные из нас возможно уже заметили, что мы не полностью воссоздали оригинальную форму, которую создавали вручную, в этой версии объекта формы. Отсутствует `fieldset`. Сейчас вы уже должны понимать, что когда я использую термин "fieldsets", я имею в виду не упомянутый ранее отсутствующий HTML `fieldset`, а, скорее, `fieldsets` в том смысле, в каком его определяет Zend Framework, в котором принципиально нет ничего общего с HTML `fieldsets`, однако может использоваться последними при необходимости. Это различие является критически важным для нашего понимания.

"Fieldset" служит для группировки отдельных полей. Это особенно уместно, когда форма состоит из элементов, имеющих различные требования. Например, если бы мы расширили форму, запрограммированную выше, адресом пользователя, эти данные, вероятно, управлялись бы своей собственной сущностью, хранящейся в собственной таблице базы данных, и ссылающейся на сущность "user" или соответствующую таблицу. Техническое преимущество `fieldsets` в том, что они могут быть повторно использованы в различных формах. Если мы будем придерживаться нашего примера, это означает, что пользователь должен указать свой адрес в рамках регистрации на почтовую рассылку (давайте сделаем вид, что это было бы уместно), но впоследствии может адаптировать пользовательский аккаунт с помощью формы изменения адреса. В обоих случаях будет использоваться `fieldset`, который был определен. Давайте рассмотрим примерный способ определения `fieldset` для двух полей формы:

```
1 <?php
2 namespace HelloWorld\Form;
3
4 use Zend\Form\Fieldset;
5
6 class UserFieldset extends Fieldset
7 {
8     public function __construct()
9     {
10         parent::__construct('user');
11
12         $this->add(array(
13             'name' => 'name',
14             'attributes' => array(
15                 'type' => 'text',
16                 'id' => 'name'
```

```

17         ),
18         'options' => array(
19             'id' => 'name',
20             'label' => 'Full Name:',
21         )
22     ));
23
24     $this->add(array(
25         'name' => 'email',
26         'attributes' => array(
27             'type' => 'email',
28         ),
29         'options' => array(
30             'id' => 'email',
31             'label' => 'Email Address:',
32         ),
33     ));
34 }
35 }

```

Listing 16.13

Определение fieldset очень похоже на форму, разработанную ранее, за исключением того, что fieldset сам не предназначен наподобие формы для решения внешних задач, но вместо этого должен быть включен в ZendForm для отображения:

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5
6 class SignUp extends Form
7 {
8     public function __construct()
9     {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13         $this->setInputFilter(new \Helloworld\Form\SignUpFilter());
14
15         $this->add(new \Helloworld\Form\UserFieldset());
16
17         $this->add(array(
18             'name' => 'submit',
19             'attributes' => array(
20                 'type' => 'submit',
21                 'value' => 'Register now'
22             ),
23         ));

```

```
24 }
25 }
```

Listing 16.14

Таким образом, вот снова форма theSignUp, но на этот раз с UserFieldset, вместо отдельных полей, которые сгруппированы теперь в UserFieldset. Для того, чтобы форма по-прежнему отображалась правильно, мы должны еще адаптировать код представления:

```
1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRow($this->form->get('user')->get('name'));
5 echo $this->formRow($this->form->get('user')->get('email'));
6 echo $this->formSubmit($this->form->get('submit'));
7 echo $this->form()->closeTag();
```

Listing 16.15

Теперь мы первоначально получаем доступ к fieldset с помощью get('user'), а оттуда доступ к элементам fieldset. Если мы не сделаем эту корректировку сейчас, это будет ошибкой. Пока все хорошо. Однако, мы еще не полностью закончили, поскольку валидация нашей формы больше не выполняется корректно, т.к. SignUpFilter все еще назначен веб-форме, но эта конфигурация больше не подходит.

```
1 <?php
2 // [...]
3 $this->setInputFilter(new \Helloworld\Form\SignUpFilter());
4 // [...]
```

Listing 16.16

Теперь мы должны обеспечить, чтобы у самого UserFieldset была необходимая конфигурация. Мы осуществляем это, предоставляя нашему UserFieldset метод getInputFilterSpecification() и в нем выполняем всю необходимую конфигурацию:

```
1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Fieldset;
5
6 class UserFieldset extends Fieldset
7 {
8     public function __construct()
9     {
10         parent::__construct('user');
11
12         $this->add(array(
13             'name' => 'name',
14             'attributes' => array(
15                 'type' => 'text',
16                 'id' => 'name'
```

```

17         ),
18         'options' => array(
19             'id' => 'name',
20             'label' => 'Full Name:',
21         )
22     ));
23
24     $this->add(array(
25         'name' => 'email',
26         'attributes' => array(
27             'type' => 'email',
28         ),
29         'options' => array(
30             'id' => 'email',
31             'label' => 'Email Address:'
32         ),
33     ));
34 }
35
36 public function getInputFilterSpecification()
37 {
38     return array(
39         'email' => array(
40             'validators' => array(
41                 array(
42                     'name' => 'NotEmpty',
43                     'break_chain_on_failure' => true,
44                     'options' => array(
45                         'messages' => array(
46                             \Zend\Validator\NotEmpty::IS_EMPTY =>
47                                 'Please input something.'
48                         )
49                     )
50                 ),
51                 array(
52                     'name' => 'EmailAddress',
53                     'options' => array(
54                         'messages' => array(
55                             \Zend\Validator\EmailAddress::INVALID_FORMAT
56                             => 'Please enter the correct email address.'
57                         )
58                     )
59                 ),
60             ),
61         ),
62         'name' => array(
63             'filters' => array(
64                 array(

```

```

65         'name' => 'StringTrim'
66     )
67 ),
68     'validators' => array(
69         array(
70             'name' => 'NotEmpty',
71             'options' => array(
72                 'messages' => array(
73                     \Zend\Validator\NotEmpty::IS_EMPTY =>
74                         'Please input something.'
75                 )
76             )
77         )
78     )
79 )
80 );
81 }
82 }

```

Listing 16.17

Итак, чего же мы теперь здесь достигнули? Мы взяли определения из `SignUpFilter` и перенесли их непосредственно в метод `getInputFilterSpecification()` нашего `fieldset`.

[А вот согласно официальной документации, чтобы это работало в `UserFieldset`, он должен реализовывать `InputFilterProviderInterface`, причем это справедливо только для элементов `fieldsets`, а обычные пользовательские элементы должны реализовывать интерфейс `InputProviderInterface`. Т.е.

```

use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;

class UserFieldset extends Fieldset implements InputFilterProviderInterface
{
    ...
}

```

или

```

use Zend\Form\Element;
use Zend\InputFilter\InputProviderInterface;

class MyColor extends Element implements InputProviderInterface
{
    ...
}

```

Поэтому делаем сразу правильно ☺, иначе валидация не будет производиться, а значит будет пропускать любые значения. Более подробно это описано здесь <http://zf2.com.ua/doc/161#Hinting>, но я бы перевел "hinting" не как "намеки", а как "неявные указания", поскольку они указывают фабрике, что класс содержит спецификацию для `InputFilter`.]

Это необходимо, поскольку использующая `fieldsets` форма получит все свои спецификации для `"InputFilter"` из метода `getInputFilterSpecification()` соответствующего `fieldset`. Теперь мы получим следующее, если форма была отправлена с валидными данными:

```
1 array(2) {
2     'submit' => string(9) "Register now"
3     'user' => array(2) {
4         'name' => string(13) "Michael Romer"
5         'email' => string(24) "zf2buch@michael-romer.de"
6     }
7 }
```

Связывание сущностей с формами

Как правило, веб-формы приложения связаны с его сущностями. Продукт вместе с информацией о количестве помещается в корзину покупок и таким образом создается заказ, пользователь регистрирует логин, адрес доставки регистрируется в сфере покупок и т.д. По этой причине часто осуществляют передачу проверенных данных из формы в соответствующую сущность, которая затем сохраняется, например, в базе данных. Или наоборот, данные из базы данных передаются из сущности в веб-форму, например, чтобы сделать ее характеристики редактируемыми в ней.

Для упрощения этой процедуры используются так называемые "гидраторы", с которыми мы уже познакомились в рамках нашей работы с `Zend\Db`. Таким образом, можно сказать, что мы прошли полный круг. Для начала нам нужна соответствующая сущность `User` для формы:

```
1 <?php
2 namespace Helloworld\Entity;
3
4 class User
5 {
6     protected $id;
7     protected $email;
8     protected $name;
9
10    public function setEmail($email)
11    {
12        $this->email = $email;
13    }
14
15    public function getEmail()
16    {
17        return $this->email;
18    }
19 }
```

```

20     public function setId($id)
21     {
22         $this->id = $id;
23     }
24
25     public function getId()
26     {
27         return $this->id;
28     }
29
30     public function setName($name)
31     {
32         $this->name = $name;
33     }
34
35     public function getName()
36     {
37         return $this->name;
38     }
39 }

```

Listing 16.18

Для того, чтобы эта сущность использовалась, как "контейнер данных" в форме SignUp, необходимо выполнить необходимую для этого конфигурацию в соответствующем месте. В этом примере мы снова сначала будем работать без fieldsets. Таким образом, мы имеем дело с "обычной" формой, в которой фактически находятся отдельные элементы.

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5
6 class SignUp extends Form
7 {
8     public function __construct()
9     {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'name' => 'name',
16             'attributes' => array(
17                 'type' => 'text',
18                 'id' => 'name'
19             ),
20             'options' => array(
21                 'id' => 'name',

```

```

22         'label' => 'Full Name:',
23     )
24 );
25
26 $this->add(array(
27     'name' => 'email',
28     'attributes' => array(
29         'type' => 'email',
30     ),
31     'options' => array(
32         'id' => 'email',
33         'label' => 'Email Address:',
34     )
35 ));
36
37 $this->add(array(
38     'name' => 'submit',
39     'attributes' => array(
40         'type' => 'submit',
41         'value' => 'Register now',
42     ),
43 ));
44 }
45 }

```

Listing 16.19

Для простоты мы пока игнорируем определения валидаторов и фильтров на данный момент. Теперь мы скомпонуем сущность в контроллере и настроим ее для используемого гидратора:

```

1 <?php
2
3 namespace Helloworld\Controller;
4
5 use Zend\Mvc\Controller\AbstractActionController;
6 use Zend\View\Model\ViewModel;
7
8 class IndexController extends AbstractActionController
9 {
10     public function indexAction()
11     {
12         $form = new \Helloworld\Form\SignUp();
13         $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
14         $form->bind(new \Helloworld\Entity\User());
15
16         if ($this->getRequest()->isPost()) {
17             $form->setData($this->getRequest()->getPost());
18         }
19     }
20 }

```



```

19         if ($form->isValid()) {
20             var_dump($form->getData());
21         } else {
22             return new ViewModel(
23                 array(
24                     'form' => $form
25                 )
26             );
27         }
28     } else {
29         return new ViewModel(
30             array(
31                 'form' => $form
32             )
33         );
34     }
35 }
36 }

```

Listing 16.20

Если теперь отправить форму с валидными данными, `$form->getData()` возвращает полностью заполненную сущность (вместо массива, как это было ранее):

```

1 class HelloWorld\Entity\User#186 (3) {
2     protected $id => NULL
3     protected $email => string(24) "zf2buch@michael-romer.de"
4     protected $name => string(13) "Michael Romer"
5 }

```

Основная настройка конфигурации заключается в вызове `bind()`, с помощью которого мы должны передать объект сущности (или ее класс, соответственно) в форму, в которую должны быть переданы данные с помощью предварительно определенного гидратора `Reflection`. Следующие гидраторы включены во фреймворк в качестве стандарта:

- **ArraySerializable**: это стандартный гидратор, который `Zend\Form` использует, если ничто другое не было определено. Он предполагает, что соответствующий объект реализует методы `getArrayCopy()` и `exchangeArray()` или `populate()`, соответственно, и таким образом предоставляет необходимую информацию.
- **ClassMethods**: использует `Getter/Setter` методы объекта (или класса, соответственно) для вставки (`hydrate()`) или чтения (`extract()`), соответственно, необходимых данных.
- **ObjectProperty**: использует открытые свойства объекта.
- **Reflection**: использует `ReflectionClass` PHP для определения свойств объекта и для установки или чтения соответствующих значений. Поскольку этот гидратор использует `$property->setAccessible(true)`, с его помощью можно управлять приватными свойствами объекта.

А теперь давайте рассмотрим все, снова используя fieldsets, поскольку в этом контексте существует полезная функциональность, если работать с несколькими объектами, ссылающимися друг на друга. Для начала создадим другую сущность, которую назовем "UserAddress" и к которой мы применим адрес пользователя, который мы хотели бы запросить непосредственно в рамках входа в систему. Данные должны управляться приложением, но также могут отдельно рассматриваться, как сущность и, к тому же, впоследствии будут храниться в базе данных в собственной таблице.

```
1 <?php
2 namespace Helloworld\Entity;
3
4 class UserAddress
5 {
6     private $street;
7     private $streetNumber;
8     private $zipcode;
9     private $city;
10
11     public function setStreet($street)
12     {
13         $this->street = $street;
14     }
15
16     public function getStreet()
17     {
18         return $this->street;
19     }
20
21     public function setCity($city)
22     {
23         $this->city = $city;
24     }
25
26     public function getCity()
27     {
28         return $this->city;
29     }
30
31     public function setStreetNumber($streetNumber)
32     {
33         $this->streetNumber = $streetNumber;
34     }
35
36     public function getStreetNumber()
37     {
38         return $this->streetNumber;
39     }
40
```

```

41 public function setZipcode($zipcode)
42 {
43     $this->zipcode = $zipcode;
44 }
45
46 public function getZipcode()
47 {
48     return $this->zipcode;
49 }
50 }

```

Listing 16.21

Расширим сущность `User` свойством `$userAddress`, которая символизирует ссылку на соответствующую сущность `UserAddress`.

```

1 <?php
2 namespace Helloworld\Entity;
3
4 class User
5 {
6     protected $id;
7     protected $email;
8     protected $name;
9     protected $userAddress;
10
11 public function setEmail($email)
12 {
13     $this->email = $email;
14 }
15
16 public function getEmail()
17 {
18     return $this->email;
19 }
20
21 public function setId($id)
22 {
23     $this->id = $id;
24 }
25
26 public function getId()
27 {
28     return $this->id;
29 }
30
31 public function setName($name)
32 {
33     $this->name = $name;
34 }

```

```

35
36 public function getName()
37 {
38     return $this->name;
39 }
40
41 public function setUserAddress($userAddress)
42 {
43     $this->userAddress = $userAddress;
44 }
45
46 public function getUserAddress()
47 {
48     return $this->userAddress;
49 }
50 }

```

Listing 16.22

Кроме того, мы создаем новый `UserAddressFieldset` вместе с необходимой спецификацией фильтрации, с которой мы уже знакомы, также, как и со ссылкой на гидратор, который должен использоваться, и соответствующей сущностью, в которую данные этого fieldset будут переданы.

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Fieldset;
5
6 class UserAddressFieldset extends Fieldset
7 {
8     public function __construct()
9     {
10         parent::__construct('userAddress');
11         $this->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
12         $this->setObject(new \Helloworld\Entity\UserAddress());
13
14         $this->add(array(
15             'name' => 'street',
16             'attributes' => array(
17                 'type' => 'text',
18             ),
19             'options' => array(
20                 'label' => 'Street:',
21             )
22         ));
23
24         $this->add(array(
25             'name' => 'streetNumber',

```

```

26         'attributes' => array(
27             'type' => 'text',
28         ),
29         'options' => array(
30             'label' => 'House Number:',
31         )
32     ));
33
34     $this->add(array(
35         'name' => 'zipcode',
36         'attributes' => array(
37             'type' => 'text',
38         ),
39         'options' => array(
40             'label' => 'ZIP Code:',
41         )
42     ));
43
44     $this->add(array(
45         'name' => 'city',
46         'attributes' => array(
47             'type' => 'text',
48         ),
49         'options' => array(
50             'label' => 'City:',
51         )
52     ));
53 }
54 }

```

Listing 16.23

Мы надлежащим образом включаем `UserAddressFieldset`, но не непосредственно в форму регистрации `SignUp`, но во "вложенный" `UserFieldSet`:

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Fieldset;
5
6 class UserFieldset extends Fieldset
7 {
8     public function __construct()
9     {
10         parent::__construct('user');
11
12         $this->add(array(
13             'name' => 'name',
14             'attributes' => array(
15                 'type' => 'text',

```

```

16         'id' => 'name'
17     ),
18     'options' => array(
19         'id' => 'name',
20         'label' => 'Full Name:',
21     )
22 );
23
24 $this->add(array(
25     'name' => 'email',
26     'attributes' => array(
27         'type' => 'email',
28     ),
29     'options' => array(
30         'id' => 'email',
31         'label' => 'Email Address:'
32     ),
33 );
34
35 $this->add(array(
36     'type' => 'Helloworld\Form\UserAddressFieldset',
37 );
38 }
39 }

```

Listing 16.24

Фактическая форма выглядит теперь так:

```

1 <?php
2 namespace Helloworld\Form;
3
4 use Zend\Form\Form;
5
6 class SignUp extends Form
7 {
8     public function __construct()
9     {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'type' => 'Helloworld\Form\UserFieldset',
16             'options' => array(
17                 'use_as_base_fieldset' => true
18             )
19         ));
20
21         $this->add(array(

```

```

22         'name' => 'submit',
23         'attributes' => array(
24             'type' => 'submit',
25             'value' => 'Register now'
26         ),
27     ));
28 }
29 }

```

Listing 16.25

В этом случае самое главное – это `'use_as_base_fieldset' => true`, чтобы корректно функционировало присвоение значений и свойств объекта, и порядок осуществления исходил из `UserFieldset`, который распределяет данные по соответствующим сущностям.

И, наконец, не забудьте настроить файл представления соответствующих `fieldset's`, так, чтобы все поля корректно отображались и не было никаких ошибок:

```

1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4
5 echo $this->formRow($this->form->get('user')
6     ->get('name'));
7
8 echo $this->formRow($this->form->get('user')
9     ->get('email'));
10
11 echo $this->formRow($this->form->get('user')
12     ->get('userAddress')->get('street'));
13
14 echo $this->formRow($this->form->get('user')
15     ->get('userAddress')->get('streetNumber'));
16
17 echo $this->formRow($this->form->get('user')
18     ->get('userAddress')->get('zipcode'));
19
20 echo $this->formRow($this->form->get('user')
21     ->get('userAddress')->get('city'));
22
23 echo $this->formSubmit($this->form->get('submit'));
24 echo $this->form()->closeTag();

```

Listing 16.26

Если теперь отправить форму с валидными данными, мы теперь получим не только объект `User`, но и ссылку на заполненный данными объект `UserAddress`:

```

1 class Helloworld\Entity\User#201 (4) {
2     protected $id => NULL
3     protected $email => string(24) "zf2buch@michael-romer.de"

```

```
4     protected $name => string(13) "Michael Romer"
5     protected $userAddress =>
6         class Helloworld\Entity\UserAddress#190 (4) {
7             private $street => string(14) "Grevingstrasse"
8             private $streetNumber => string(2) "35"
9             private $zipcode => string(5) "48151"
10            private $city => string(8) "Münster"
11        }
12 }
```

Дальнейшее упрощение обработки с помощью аннотации

Наброски способов обработки форм уже сделают жизнь разработчика намного проще. Тем не менее, для выполнения конфигурации по-прежнему требуется немало кода. Альтернативным способом является использование аннотации в классах сущностей, при которой конфигурация динамически генерируется фреймворком, в противном случае большая часть создавалась бы вручную. Дополнительную информацию по этой теме можно найти при необходимости в официальной документации фреймворка.

XVII. Дневники разработчика

Введение

До этого момента мы, в основном, говорили о концепции ядра фреймворка и его отдельных компонентов. Теперь мы перейдем к примеру приложения и будем решать проблемы повседневного бизнеса с Zend Framework.

Для разработки примера-приложения мы будем использовать метод "скрам" (<http://ru.wikipedia.org/wiki/Scrum>) всякий раз, когда это возможно. Скрам – это основа итеративной (http://ru.wikipedia.org/wiki/Итеративная_разработка) и пошаговой гибкой разработки программного обеспечения, предназначенная для управления проектами программного обеспечения и разработки приложений. Я принял гибкие методологии разработки несколько лет назад и, в частности, скрам – примерно в 2008 году. Я считаю, скрам действительно помогает овладению искусством разработки профессионального программного обеспечения, и он очень помог мне в моих профессиональных проектах. Вот почему я рекомендую рассмотреть скрам, когда это будет возможно. Просто не достаточно писать хороший код, например, с помощью Zend Framework 2, чтобы преуспеть в сложных проектах программного обеспечения (http://ru.wikipedia.org/wiki/Управление_проектами). Вам также необходимо организовать себя.

Начнем с так называемой "выработки концепции" (Envisioning), которая поможет сформировать концепцию конечного продукта, который мы будем разрабатывать затем с каждым так называемым "спринтом" – фиксированным окном времени разработки.

Выработка концепции

В дальнейшем мы хотим реализовать функциональные возможности "ZfDeals" ("ZfКоммерция"). ZfDeals представляет собой приложение для продажи продукции онлайн по специальным сниженным ценам. Идея заключается в том, чтобы создать не автономное приложение, а ZF2 модуль, который может использоваться другими в своих приложениях. В целях демонстрации и разработки, мы придумаем также образец "хост-приложения" для модуля.

Спринт 1 – Репозиторий и начальная база кода

Первый спринт предназначен для самостоятельной подготовки к разработке.

Настройка репозитория git и первый коммит.

Во-первых мне нужно хранилище кода для поддержания моего кода. Я просто хочу переключаться между версиями (versions), выполнять ветвление (branch) и слияние (merge) и просто знать, что мой код управляется хорошо. Я выбираю git локально, а также GitHub,

как мой внешний репозиторий, в который я могу регулярно передавать сделанные изменения (push). У меня уже есть аккаунт GitHub, так что мне не нужно регистрироваться заново. Тем не менее регистрация займет всего несколько минут и, кстати, она бесплатна, если используется для проектов с открытым исходным кодом. Отлично!

[Википедия, <http://ru.wikipedia.org/wiki/Git>, раздел "ссылки". Здесь можно выбрать документацию по git на любой вкус и уровень. Мне лично понравилось это <http://githowto.com/ru> пошаговое руководство.]

Git уже установлен на моей локальной машине, так что я могу сделать

```
1 pwd
2 cd path/to/webserver-project
3 $ git clone git://github.com/zendframework/ZendSkeletonApplication.git ZfDealsApp
```

Первой строкой мы проверяем, в каком каталоге сейчас находимся. Второй строкой перемещаемся в рабочий каталог на локальном веб-сервере, в котором должен будет находиться наш проект ZfDealsApp. И третьей строкой клонируем ZendSkeletonApplication, который будет выступать в качестве отправной точки для нашего индивидуального проекта.

Composer загрузит Zend Framework 2 в каталог vendor просто выполнив:

```
1 $ php composer.phar install
```

В каталоге ZfDealsApp выполните следующие команды:

```
1 $ rm -Rf .git
2 $ git init
3 $ git add .
4 $ git commit -m "SkeletonApplication"
```

Таким образом мы удалили из каталога .git ссылки на все внешние репозитории, список разработчиков zend, историю изменений ZendSkeletonApplication и т.д. Предупреждение: на некоторых Unix системах первая строка может удалить не только каталог .git с содержимым, но и по цепочке ссылки ".." на вышележащие директории – если вы не уверены, вы можете выполнить удаление каталога .git обычным для вашей Unix системы способом. Мы создали также начальную точку для разработки и назвали ее "SkeletonApplication" и начинаем с чистого листа. Поскольку ZF2 распространяется по новой лицензии BSD, не обязательно, чтобы на вашем репозитории на GitHub присутствовала лишняя информация (например, о контрибьюторах). Мало того, так вы выполните требование лицензии:

Ни имя Zend Technologies USA, Inc., ни имена его контрибьюторов не могут быть использованы для поддержки или продвижения продуктов, основанных на этом ПО, без предварительного письменного разрешения.

Теперь мы должны создать новый репозиторий на GitHub. Он выступит в качестве средства резервирования кода для нашего локального репозитория, а также это сделает удобным обмен кодом с другими. Укажем для проекта ZfDealsApp наш репозиторий, например:

```
1 $ git remote add origin https://github.com/michael-romer/ZfDealsApp.git
```

Конечно же, вместо `https://github.com/michael-romer/ZfDealsApp.git` вы должны указать свой собственный репозиторий. Затем мы выгрузим наш проект на GitHub:

```
1 $ git push -u origin master
```

Спринт 2 – Пользовательский модуль, добавляющий функциональность продукта

В то время, как спринт 1 был сосредоточен на создании начального кода, спринт 2 выдвигает первое функциональное требование: добавление новых продуктов в систему, которые могут быть позднее проданы со специальной скидкой. В скрам требования обычно задаются с помощью техники, называемой "история пользователей":

Пользовательские истории (англ. User Story) — способ описания требований к разрабатываемой системе, сформулированных как одно или более предложений на ежедневном или деловом языке пользователя.

(http://ru.wikipedia.org/wiki/Пользовательские_истории)

Как правило, предложения соответствуют схеме:

"Чтобы [получить выгоду] я, как [роль], хочу [цели/пожелания]"

Таким образом, первое требование гласит:

"Чтобы предложить продукт со специальной скидкой я, как предприниматель, хочу добавлять в систему детальное описание продукции."

Это логично. Как правило, дополнительно к "требованиям пользовательской истории" заявляют так называемые критерии приемки, которые более детально описывают требования:

- Нужно добавить уникальный ID, описание и информацию о наличии продукта.
- Нужно добавить все данные о продукте с помощью веб-формы.

Создание пользовательского модуля

Итак, начали! Во-первых я добавил новый специализированный ZF2-модуль, который содержит все функциональные возможности ZfDeals. Я задал следующую структуру каталогов и файлов в модуле:

```
1 ZfDeals/
```

```
2  Module.php
3  config/
4      module.config.php
5  src/
6      ZfDeals/
7          Controller/
8              AdminController.php
9  view/
10     zf-deals/
11         admin/
12             index.phtml
13             layout/
14                 admin.phtml
```

[Если вы будете работать с Git Bash под Windows и использовать готовый модуль ZfDeals, git начнет ругаться на то, что перенос строк в модуле в Unix-формате:

```
fatal: LF would be replaced by CRLF in module/ZfDeals/Module.php
```

Самое простое решение в данном случае – отключить проверку формата:

```
1 $ git config --global core.autocrlf false
2 $ git config --global core.safecrlf false
]
```

ZfTool

Вместо создания каталогов и файлов вручную, вы можете воспользоваться ZfTool, и пусть он автоматически создает большинство каталогов и файлов. Однако, ZfTool не является частью библиотеки и должен быть загружен отдельно <https://packages.zendframework.com/zftool.phar>

[Вполне вменяемая информация по ZfTool на русском <http://habrahabr.ru/post/166385>

Единственное замечание, не нужно качать инструментарий в виде модуля, используйте phar-архив по ссылке выше. И, следовательно, не нужно прописывать его, как модуль в составе вашего приложения. Зачем? Он и так прекрасно справляется со своими задачами:

```
1 $ zftool.phar create project <path>
2 $ zftool.phar create module Test <path>
]
```

В Module.php я добавляю только основной код для автозагрузки классов этого модуля и указываю менеджеру ModuleManager фреймворка на конфигурационный файл модуля:

```

1 <?php
2 namespace ZfDeals;
3
4 class Module
5 {
6     public function getConfig()
7     {
8         return include __DIR__ . '/config/module.config.php';
9     }
10
11     public function getAutoloaderConfig()
12     {
13         return array(
14             'Zend\Loader\StandardAutoloader' => array(
15                 'namespaces' => array(
16                     __NAMESPACE__
17                         => __DIR__ . '/src/' . __NAMESPACE__,
18                 ),
19             ),
20         );
21     }
22 }

```

Listing 26.1

Мой новый пустой AdminController наследует классу AbstractActionController, позволяющему работать с пользовательскими "действиями":

```

1 <?php
2 namespace ZfDeals\Controller;
3
4 use Zend\Mvc\Controller\AbstractActionController;
5 use Zend\View\Model\ViewModel;
6
7 class AdminController extends AbstractActionController
8 {
9     public function indexAction()
10    {
11        return new ViewModel();
12    }
13 }

```

Listing 26.2

Файл module.config.php незатейлив. Он содержит маршруты типа Literal для домашней страницы раздела администратора (admin) ZfDeals

```

1 <?php
2 return array(
3     'router' => array(
4         'routes' => array(

```

```

5         'zf-deals\admin\home' => array(
6             'type' => 'Zend\Mvc\Router\Http\Literal',
7             'options' => array(
8                 'route' => '/deals/admin',
9                 'defaults' => array(
10                     'controller'
11                         => 'ZfDeals\Controller\Admin',
12                     'action'
13                         => 'index',
14                 ),
15             ),
16         ),
17     ),
18 ),
19 // [...]
20 )

```

Listing 26.3

а также декларации AdminController

```

1 <?php
2 // [...]
3 'controllers' => array(
4     'invokables' => array(
5         'ZfDeals\Controller\Admin'
6         => 'ZfDeals\Controller\AdminController'
7     ),
8 ),
9 // [...]

```

Listing 26.4

и декларацию admin layout (макета администратора). В классе Module я обеспечиваю, чтобы этот макет использовался всякий раз, когда выполняется диспетчеризация AdminController:

```

1 <?php
2 // [...]
3 public function init(\Zend\ModuleManager\ModuleManager $moduleManager)
4 {
5     $sharedEvents = $moduleManager
6         ->getEventManager()->getSharedManager();
7     $sharedEvents->attach(
8         'ZfDeals\Controller\AdminController',
9         'dispatch',
10        function($e) {
11            $controller = $e->getTarget();
12            $controller->layout('zf-deals/layout/admin');
13        },
14        100

```

```

15 );
16 }
17 // [...]

```

Listing 26.5

Я регистрирую функцию обратного вызова для события `dispatch` в методе `init`. Если событие `dispatch` выполняется для контроллера `ZfDeals\Controller\AdminController`, выполняется функция обратного вызова. Причина, по которой здесь используется `SharedEventManager`, проста: на момент, когда добавляется мой обработчик события, экземпляр `AdminController` вместе с его собственным `EventManager` еще не создан. Так что мне нужно использовать здесь `SharedEventManager`, чтобы добавить мой обработчик.

Функция обратного вызова использует плагин контроллера `layout` для установки макета, заданного в `module.config.php`:

```

1 <?php
2 // [...]
3 'view_manager' => array(
4     'template_map' => array(
5         'zf-deals/layout/admin' => __DIR__ . '/../view/layout/admin.phtml',
6     ),
7     'template_path_stack' => array(
8         __DIR__ . '/../view',
9     ),
10 ),
11 // [...]

```

Listing 26.6

Не забудьте активировать новый модуль, добавив его в список модулей `application.config.php`:

```

1 <?php
2 return array(
3     'modules' => array(
4         'Application',
5         'ZfDeals'
6     ),
7 // [...]
8 );

```

Listing 26.7

После добавления пустого метода действия `index` в новом контроллере, а также создания файла представления, я могу открыть `/deals/admin` в браузере и посмотреть дизайн макета на экране.

Работа со статическими ресурсами

На этом этапе я спрашиваю себя, что делать со статическими ресурсами, которые будет содержать мой модуль, такими, как изображения, логотип ZfDeals, файлы css и js. Если нужно предоставить клиенту ресурсы, в основном они должны быть общедоступны, например, помещены в каталог `public` приложения. Это означает, что если я предоставляю свои ресурсы в комплекте с модулем ZfDeals, они не будут доступны по умолчанию. Итак, что делать? К сожалению, ZF2 не предоставляет никакой поддержки "из коробки", и нужен творческий подход, чтобы решить эту проблему. Вот варианты:

- Я разрабатываю контроллер, который предоставляет статические ресурсы с помощью PHP, используя `file_get_contents()` или что-то подобное. Это, вероятно, будет работать нормально, однако это не очень умно — добавлять подобный код "инфраструктуры" в ZfDeals, и если я не внедрю механизм кэширования, это, несомненно, станет узким местом в производительности на каком-то этапе.
- Я использую библиотеку менеджера ресурсов для ZF2, такую, как `AssetManager` (например, <https://github.com/RWOverdijk/AssetManager>), и добавляю этот модуль в список зависимостей ZfDeals. Однако для этого мне потребуется устанавливать другой модуль, помимо ZfDeals, чтобы все заработало должным образом.
- Я просто скопирую все ресурсы в каталог `public` приложения. Однако это означает, что при распространении модуля мне нужно будет инструктировать разработчика о том, как интегрировать ZfDeals в свое приложение, скопировав файлы вручную.
- Я просто скопирую ресурсы в каталог `public` приложения. Но вместо того, чтобы поместить их в каталог `public`, я добавляю другой каталог, предпочтительно, названный по имени модуля. Однако это означает, что при распространении модуля мне нужно будет инструктировать разработчика о том, как интегрировать ZfDeals в свое приложение, скопировав файлы вручную.
- Я оставляю ресурсы внутри модуля, но добавляю символическую ссылку в каталог `public`. И снова это означает, что при распространении модуля мне нужно будет инструктировать разработчика о том, как интегрировать ZfDeal в свое собственное приложение, создав символическую ссылку.

Я думаю, использование надлежащего менеджера ресурсов (`asset manager`) будет моим предпочтительным выбором в долгосрочной перспективе, однако на данный момент я обойдусь копированием файлов в каталог `public` приложения.

Форма для добавления продукции

Теперь я добавляю первую форму в ZfDeals. Форма используется для добавления новых продуктов в систему. Вместо того, чтобы помещать поля, имеющие отношение к продукции, непосредственно в форму, я инкапсулирую их в `fieldset`. `Fieldset` добавляется к форме следующим образом:

```
1 <?php
2 namespace ZfDeals\Form;
3
```



```

4 use Zend\Form\Form;
5
6 class ProductAdd extends Form
7 {
8     public function __construct()
9     {
10         parent::__construct('login');
11         $this->setAttribute('action', '/deals/admin/product/add');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'type' => 'ZfDeals\Form\ProductFieldset',
16             'options' => array(
17                 'use_as_base_fieldset' => true
18             )
19         ));
20
21         $this->add(array(
22             'name' => 'submit',
23             'attributes' => array(
24                 'type' => 'submit',
25                 'value' => 'Добавить'
26             ),
27         ));
28     }
29 }

```

Listing 26.8

Форма ProductAdd состоит из кнопки "Add Product", а также ProductFieldset, который также содержит определения фильтров и валидаторов:

```

1 <?php
2 namespace ZfDeals\Form;
3
4 use Zend\Form\Fieldset;
5 use Zend\InputFilter\InputFilterInterface;
6 use Zend\InputFilter\InputFilterProviderInterface;
7
8 class ProductFieldset extends Fieldset
9 implements InputFilterProviderInterface
10 {
11     public function __construct()
12     {
13         parent::__construct('product');
14
15         $this->add(array(
16             'name' => 'id',
17             'attributes' => array(
18                 'type' => 'text',

```

```

19         ),
20         'options' => array(
21             'label' => 'ID продукта:',
22         )
23     ));
24
25
26     $this->add(array(
27         'name' => 'name',
28         'attributes' => array(
29             'type' => 'text',
30         ),
31         'options' => array(
32             'label' => 'Название продукта:',
33         )
34     ));
35
36     $this->add(array(
37         'name' => 'stock',
38         'attributes' => array(
39             'type' => 'number',
40         ),
41         'options' => array(
42             'label' => '# Количество:',
43         ),
44     ));
45 }
46
47 public function getInputFilterSpecification()
48 {
49     return array(
50         'id' => array (
51             'required' => true,
52             'filters' => array(
53                 array(
54                     'name' => 'StringTrim'
55                 )
56             ),
57             'validators' => array(
58                 array(
59                     'name' => 'NotEmpty',
60                     'options' => array(
61                         'message' =>
62                             "Пожалуйста, введите ID продукта."
63                     )
64                 )
65             )
66         ),

```

```

67     'name' => array (
68         'required' => true,
69         'filters' => array(
70             array(
71                 'name' => 'StringTrim'
72             )
73         ),
74         'validators' => array(
75             array(
76                 'name' => 'NotEmpty',
77                 'options' => array(
78                     'message' =>
79                         "Пожалуйста, введите название продукта."
80                 ),
81             )
82         )
83     ),
84     'stock' => array (
85         'required' => true,
86         'filters' => array(
87             array(
88                 'name' => 'StringTrim'
89             )
90         ),
91         'validators' => array(
92             array(
93                 'name' => 'NotEmpty',
94                 'options' => array(
95                     'message' =>
96                         "Пожалуйста, укажите количество."
97                 )
98             ),
99             array(
100                 'name' => 'Digits',
101                 'options' => array(
102                     'message' =>
103                         "Пожалуйста, введите целое число."
104                 )
105             ),
106             array(
107                 'name' => 'GreaterThan',
108                 'options' => array(
109                     'min' => 0,
110                     'message' =>
111                         "Пожалуйста, введите значение >= 0."
112                 )
113             )
114         )

```

```

115         )
116     );
117 }
118 }

```

Listing 26.9

Этот подход позволяет повторно использовать определения полей продукта и в других формах, например, форме, которая предназначена для редактирования существующего продукта.

Отображение формы

Дополнительные маршруты и действия заботятся об отображении формы:

```

1 <?php
2 return array(
3     'router' => array(
4         'routes' => array(
5             'zf-deals\admin\home' => array(
6                 'type' => 'Zend\Mvc\Router\Http\Literal',
7                 'options' => array(
8                     'route' => '/deals/admin',
9                     'defaults' => array(
10                        'controller'
11                            => 'ZfDeals\Controller\Admin',
12                        'action'
13                            => 'index',
14                    ),
15                ),
16            ),
17            'zf-deals\admin\product\add' => array(
18                'type' => 'Zend\Mvc\Router\Http\Literal',
19                'options' => array(
20                    'route' => '/deals/admin/product/add',
21                    'defaults' => array(
22                        'controller'
23                            => 'ZfDeals\Controller\Admin',
24                        'action'
25                            => 'add-product',
26                    )
27                )
28            )
29        )
30    )
31    // [...]
32 )

```

Listing 26.10

Само действие выглядит следующим образом:

```

1 <?php
2 // [...]
3 public function addProductAction()
4 {
5     $form = new \ZfDeals\Form\ProductAdd();
6
7     if ($this->getRequest()->isPost()) {
8         $form->setData($this->getRequest()->getPost());
9
10        if ($form->isValid()) {
11            // todo
12        } else {
13            return new ViewModel(
14                array(
15                    'form' => $form
16                )
17            );
18        }
19    } else {
20        return new ViewModel(
21            array(
22                'form' => $form
23            )
24        );
25    }
26 }
27 // [...]

```

Listing 26.11

В соответствующем представлении я отображаю форму:

```

1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRow($this->form->get('product')->get('id'));
5 echo $this->formRow($this->form->get('product')->get('name'));
6 echo $this->formRow($this->form->get('product')->get('stock'));
7 echo $this->formSubmit($this->form->get('submit'));
8 echo $this->form()->closeTag();

```

Listing 26.12

Если я снова открою `/deals/admin/product/add`, я уже вижу, что форма отображается. Тем не менее, выглядит она не так хорошо. Я могу сделать ее более привлекательной, добавив "Twitter Bootstrap" (<http://twitter.github.com/bootstrap>), и вместо того, чтобы добавлять всю необходимую разметку вручную, я добавляю другой ZF2-модуль: "DluTwBootstrap" (<https://bitbucket.org/dlu/dlutwbootstrap/overview>). Как всегда, я использую composer для установки и настройки модуля, добавив зависимость:

```

1 "dlu/dlutwbootstrap": "dev-master"

```

[Собственно, это счастье так просто с помощью composer устанавливаться не хочет, поэтому, чтобы спринт нормально работал, я уже скачал этот модуль и добавил его в vendor, поэтому ничего с composer.json не делайте.]

Затем я запускаю:

```
1 $ php composer.phar update
```

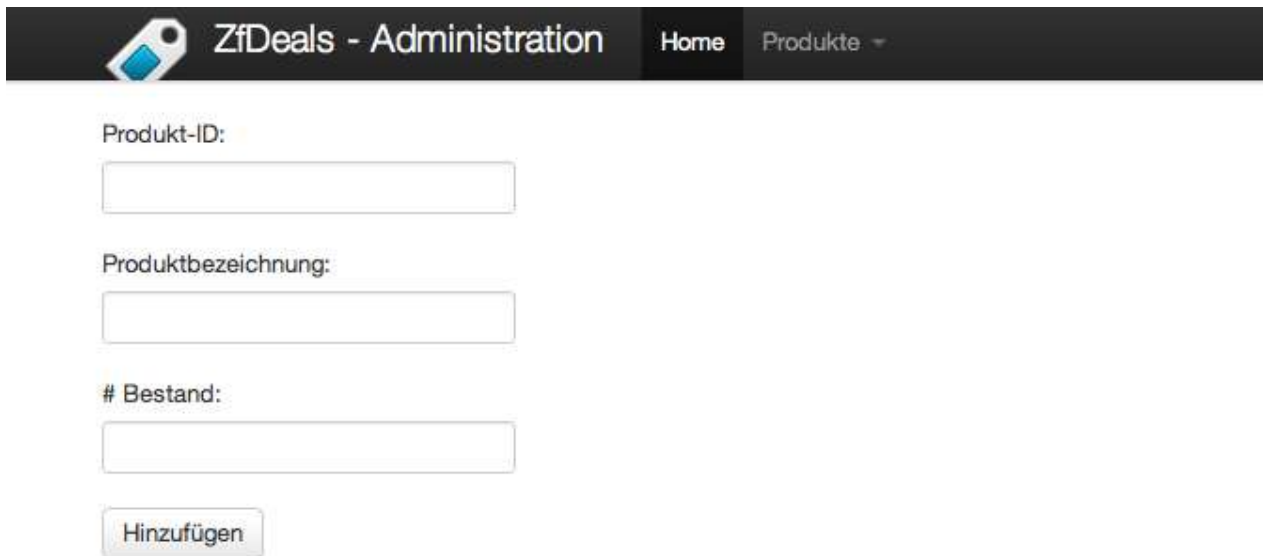
и модуль будет установлен (не забудьте включить его в application.config.php !). Модуль в основном регистрирует группу дополнительных помощников представления для отображения формы с использованием Twitter Bootstrap:

```
1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRowTwb($this->form->get('product')->get('id'));
5 echo $this->formRowTwb($this->form->get('product')->get('name'));
6 echo $this->formRowTwb($this->form->get('product')->get('stock'));
7 echo $this->formSubmitTwb($this->form->get('submit'));
8 echo $this->form()->closeTag();
```

Listing 26.13

И это то, как она выглядит сейчас:

ю

The screenshot shows the 'ZfDeals - Administration' web interface. At the top, there is a dark navigation bar with a logo on the left, the text 'ZfDeals - Administration' in the center, and 'Home' and 'Produkte' (with a dropdown arrow) on the right. Below the navigation bar, the main content area is white. It contains a form with three input fields: 'Produkt-ID:', 'Produktbezeichnung:', and '# Bestand:'. Each field has a corresponding text input box. Below these fields is a button labeled 'Hinzufügen'.

Юнит-тесты для веб-формы

Прежде, чем я начну работу над получением данных из формы и в базу данных, я сначала добавлю несколько модульных тестов для формы ProductAdd, просто чтобы убедиться, что она настроена правильно. Я создаю новый каталог test в корне приложения и добавляю файл phpunit.xml для настройки каталогов, содержащих тесты:

```

1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./ZfDealsTest/FormTest</directory>
5         </testsuite>
6     </testsuites>
7 </phpunit>

```

Поскольку мне нужна начальная загрузка приложения со всеми его сервисами, чтобы фактически запустить тесты, я добавляю также файл bootstrap.php в каталог test:

```

1 <?php
2 use Zend\Loader\StandardAutoloader;
3
4 chdir(dirname(__DIR__));
5
6 include 'init_autoloader.php';
7
8 $loader = new StandardAutoloader();
9 $loader->registerNamespace('ZfDealsTest', __DIR__ . '/ZfDealsTest');
10 $loader->register();
11
12 Zend\Mvc\Application::init(include 'config/application.config.php');

```

Listing 26.14

Это автоматически выполняется для PHPUnit при выполнении тестов. В bootstrap.php я настраиваю автозагрузку классов теста, хранящихся в каталоге ZfDealsTest. Тестовый файл ProductAddTest расположен в подкаталоге FormTest:

```

1 <?php
2 namespace ZfDealsTest\FormTest;
3
4 use ZfDeals\Form\ProductAdd;
5
6 class ProductAddTest extends \PHPUnit_Framework_TestCase
7 {
8     private $form;
9     private $data;
10
11     public function setUp()
12     {
13         $this->form = new ProductAdd();
14         $this->data = array(
15             'product' => array(
16                 'id' => '',
17                 'name' => '',
18                 'stock' => ''
19             )
20         );

```

```

21     }
22
23     public function testEmptyValues()
24     {
25         $form = $this->form;
26         $data = $this->data;
27
28         $this->assertFalse($form->setData($data)->isValid());
29
30         $data['product']['id'] = 1;
31         $this->assertFalse($form->setData($data)->isValid());
32
33         $data['product']['name'] = 1;
34         $this->assertFalse($form->setData($data)->isValid());
35
36         $data['product']['stock'] = 1;
37         $this->assertTrue($form->setData($data)->isValid());
38     }
39
40     public function testStockElement()
41     {
42         $form = $this->form;
43         $data = $this->data;
44         $data['product']['id'] = 1;
45         $data['product']['name'] = 1;
46
47         $data['product']['stock'] = -1;
48         $this->assertFalse($form->setData($data)->isValid());
49
50         $data['product']['stock'] = "test";
51         $this->assertFalse($form->setData($data)->isValid());
52
53         $data['product']['stock'] = 12.3;
54         $this->assertFalse($form->setData($data)->isValid());
55
56         $data['product']['stock'] = 12;
57         $this->assertTrue($form->setData($data)->isValid());
58     }
59 }

```

Listing 26.15

Чтобы проверить форму, я передаю в различных комбинациях тестовые данные и проверяю поведение формы.

Настройка сущности product

Сейчас я начинаю моделирование так называемого "бизнес-домена", добавляя класс сущности product, представляющий продукт в системе:


```

1 <?php
2 namespace ZfDeals\Entity;
3
4 class Product
5 {
6     protected $id;
7     protected $name;
8     protected $stock;
9
10    public function setName($name)
11    {
12        $this->name = $name;
13    }
14
15    public function getName()
16    {
17        return $this->name;
18    }
19
20    public function setId($id)
21    {
22        $this->id = $id;
23    }
24
25    public function getId()
26    {
27        return $this->id;
28    }
29
30    public function setStock($stock)
31    {
32        $this->stock = $stock;
33    }
34
35    public function getStock()
36    {
37        return $this->stock;
38    }
39 }

```

Listing 26.16

У сущности `product` есть собственный ID [как вы помните, первый признак сущности], имя и информация о количестве товара (stock information). Файл будет добавлен в каталог с именем `Entity` в `scr/ZfDeals` в моем модуле.

Персистентность сущности product

database mapper, отображающий поля сущности на столбцы таблицы в базе данных выглядит следующим образом:

```
1 <?php
2
3 namespace ZfDeals\Mapper;
4
5 use ZfDeals\Entity\Product as ProductEntity;
6 use Zend\Stdlib\Hydrator\HydratorInterface;
7 use Zend\Db\TableGateway\TableGateway;
8 use Zend\Db\TableGateway\Feature\RowGatewayFeature;
9 use Zend\Db\Sql\Sql;
10 use Zend\Db\Sql\Insert;
11
12 class Product extends TableGateway
13 {
14     protected $tableName = 'product';
15     protected $idCol = 'id';
16     protected $entityPrototype = null;
17     protected $hydrator = null;
18
19     public function __construct($adapter)
20     {
21         parent::__construct($this->tableName,
22             $adapter,
23             new RowGatewayFeature($this->idCol)
24         );
25
26         $this->entityPrototype = new ProductEntity();
27         $this->hydrator = new \Zend\Stdlib\Hydrator\Reflection;
28     }
29
30     public function insert($entity)
31     {
32         return parent::insert($this->hydrator->extract($entity));
33     }
34 }
```

Listing 26.17

"Соглашение по конфигурации" делает mapper простым, если поля сущности соответствуют именам колонок в таблице базы данных. Благодаря `\Zend\Stdlib\Hydrator\Reflection` вся "магия отображения" в основном происходит автоматически при вызове метода `insert`.

Теперь давайте добавим недостающий адаптер базы данных в `module.config.php`. Это нужно, чтобы осуществить подключение к базе данных:

```

1 <?php
2 // [...]
3 'service_manager' => array(
4     'factories' => array(
5         'Zend\Db\Adapter\Adapter' => function ($sm) {
6             $config = $sm->get('Config');
7             $dbParams = $config['dbParams'];
8
9             return new Zend\Db\Adapter\Adapter(array(
10                 'driver' => 'pdo',
11                 'dsn' =>
12                     'mysql:dbname=' . $dbParams['database']
13                     . ';host=' . $dbParams['hostname'],
14                 'database' => $dbParams['database'],
15                 'username' => $dbParams['username'],
16                 'password' => $dbParams['password'],
17                 'hostname' => $dbParams['hostname'],
18             ));
19         }
20     )
21 )
22 // [...]

```

Listing 26.18

Я поместил параметры доступа для соединения с базой данных в `db.local.php` в каталог `/config/autoload`. Я не добавлял этот файл в репозиторий кода, поскольку он содержит конфиденциальные данные. Однако я добавил еще один файл `db.local.php.dist`, который содержится в репозитории вместо этого. Он выступает в качестве шаблона для файла конфигурации, который должен присутствовать в комплекте работающего ZfDeals:

```

1 <?php
2 return array(
3     'dbParams' => array(
4         'database' => '',
5         'username' => '',
6         'password' => '',
7         'hostname' => '',
8     )
9 );

```

Listing 26.19

Таким образом я убедился в том, что разработчики могут понять структуру конфигурационного файла базы данных для интеграции ZfDeals в собственные приложения.

В `ServiceManager` я делаю доступным `ZfDeals\Mapper\Product` и внедряю зависимость в `Zend\Db\Adapter\Adapter`:

```

1 <?php
2 // [...]
3 'service_manager' => array(
4     'factories' => array(
5         'ZfDeals\Mapper\Product' => function ($sm) {
6             return new \ZfDeals\Mapper\Product(
7                 $sm->get('Zend\Db\Adapter\Adapter')
8             );
9         },
10     )
11 )
12 // [...]

```

Listing 26.20

Обработка форм

Теперь я могу добавить код для обеспечения персистентности в `addProductAction()`. Я считываю данные из формы и, в первую очередь, путем связывания (binding) нового объекта сущности продукта, вызов `getData()` возвращает объект, автоматически заполненный данными. `ZfDeals\Mapper\Product` используется для сохранения сущности в базе данных:

```

1 <?php
2 // [...]
3 public function addProductAction()
4 {
5     $form = new \ZfDeals\Form\ProductAdd();
6
7     if ($this->getRequest()->isPost()) {
8         $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
9         $form->bind(new \ZfDeals\Entity\Product());
10        $form->setData($this->getRequest()->getPost());
11
12        if ($form->isValid()) {
13            $newEntity = $form->getData();
14
15            $mapper = $this->getServiceLocator()
16                ->get('ZfDeals\Mapper\Product');
17
18            $mapper->insert($newEntity);
19            $form = new \ZfDeals\Form\ProductAdd();
20
21            return new ViewModel(
22                array(
23                    'form' => $form,
24                    'success' => true
25                )
26            );
27        } else {

```

```

28         return new ViewModel(
29             array(
30                 'form' => $form
31             )
32         );
33     }
34 } else {
35     return new ViewModel(
36         array(
37             'form' => $form
38         )
39     );
40 }
41 }
42 // [...]

```

Listing 26.21

Но все это будет работать только после того, как будет создана таблица базы данных:

```

1 CREATE TABLE product (
2     id varchar(255) NOT NULL,
3     name varchar(255) NOT NULL,
4     stock int(10) NOT NULL, PRIMARY KEY (id)
5 );

```

Теперь можно отправить форму, и новая запись будет добавлена в базу данных. Если все удалось, отображается сообщение об успешном выполнении:

```

1 <?php if ($this->success) { ?>
2 <div class="alert alert-success">Produkt hinzugefügt!</div>
3 <?php } ?>
4
5 <?php
6 $this->form->prepare();
7 echo $this->form()->openTag($this->form);
8 echo $this->formRowTwb($this->form->get('product')->get('id'));
9 echo $this->formRowTwb($this->form->get('product')->get('name'));
10 echo $this->formRowTwb($this->form->get('product')->get('stock'));
11 echo $this->formSubmitTwb($this->form->get('submit'));
12 echo $this->form()->closeTag();

```

Listing 26.22

Внедрение зависимостей

Пока все хорошо. Однако есть много кода, который может быть улучшен. Слишком часто я использую оператор `new` в моем коде, делая его зависимым от других классов. Это плохая практика делает мой код трудно тестируемым, и я хочу по возможности избежать ее, применяя внедрение зависимостей. Это позволит облегчить тестирование, а также сделает код более универсальным. Давайте сделаем рефакторинг сразу.

Новая фабрика `AdminControllerFactory` добавляется для создания `AdminController`. Фабрика заботится о внедрении зависимости `ZfDeals\Mapper\Product`:

```
1 <?php
2 namespace ZfDeals\Controller;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class AdminControllerFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $ctr = new AdminController();
12        $form = new \ZfDeals\Form\ProductAdd();
13        $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
14        $form->bind(new \ZfDeals\Entity\Product());
15        $ctr->setProductAddForm($form);
16
17        $mapper = $serviceLocator->getServiceLocator()
18            ->get('ZfDeals\Mapper\Product');
19
20        $ctr->setProductMapper($mapper);
21        return $ctr;
22    }
23 }
```

Listing 26.23

В `module.config.php` я указываю фабрику:

```
1 <?php
2 // [...]
3 'controllers' => array(
4     'factories' => array(
5         'ZfDeals\Controller\Admin'
6         => 'ZfDeals\Controller\AdminControllerFactory'
7     )
8 )
9 // [...]
```

Listing 26.24

Юнит-тесты контроллера

Не идеально, но намного лучше. По крайней мере, теперь я могу добавить несколько модульных тестов для моего контроллера. До рефакторинга я вообще не мог выполнить необходимое юнит-тестирование моего контроллера.

Но что именно тестировать в контроллере? В лучшем случае, на долю контроллера не приходится много магии. В основном, контроллер должен управлять другими объектами

для выполнения сложных задач. Что я должен здесь проверить – это то, что контроллер управляет правильными объектами в нужное время в нужном месте. Главным образом, у нас есть следующие случаи, когда наш контроллер должен иметь дело с:

1. Форма запрошена посредством GET: форма должна быть показана.
2. Форма отправлена посредством POST, но не прошла валидацию: показать форму снова с сообщением об ошибке.
3. Форма отправлена посредством POST и успешно прошла проверку: должна быть создана новая сущность на основе предоставленных данных и сохранена в базе данных с помощью mapper.

Тогда тесты могли бы выглядеть подобно:

```
1 <?php
2 namespace ZfDeals\ControllerTest;
3
4 use ZfDeals\Controller\AdminController;
5 use Zend\Http\Request;
6 use Zend\Http\Response;
7 use Zend\Mvc\MvcEvent;
8 use Zend\Mvc\Router\RouteMatch;
9
10 class AdminControllerTest extends \PHPUnit_Framework_TestCase
11 {
12     private $controller;
13     private $request;
14     private $response;
15     private $routeMatch;
16     private $event;
17
18     public function setUp()
19     {
20         $this->controller = new AdminController();
21         $this->request = new Request();
22         $this->response = new Response();
23         $this->routeMatch = new RouteMatch(array('controller' => 'admin'));
24         $this->routeMatch->setParam('action', 'add-product');
25         $this->event = new MvcEvent();
26         $this->event->setRouteMatch($this->routeMatch);
27         $this->controller->setEvent($this->event);
28     }
29
30     public function testShowFormOnGetRequest()
31     {
32         $fakeForm = new \Zend\Form\Form('fakeForm');
33         $this->controller->setProductAddForm($fakeForm);
34         $this->request->setMethod('get');
35         $response = $this->controller->dispatch($this->request);
```

```

36     $viewModelValues = $response->getVariables();
37     $formReturned = $viewModelValues['form'];
38     $this->assertEquals($formReturned->getName(), $fakeForm->getName());
39 }
40
41 public function testShowFormOnValidationError()
42 {
43     $fakeForm = $this->getMock('Zend\Form\Form', array('isValid'));
44
45     $fakeForm->expects($this->once())
46         ->method('isValid')
47         ->will($this->returnValue(false));
48
49     $this->controller->setProductAddForm($fakeForm);
50     $this->request->setMethod('post');
51     $response = $this->controller->dispatch($this->request);
52     $viewModelValues = $response->getVariables();
53     $formReturned = $viewModelValues['form'];
54     $this->assertEquals($formReturned->getName(), $fakeForm->getName());
55 }
56
57 public function testCallMapperOnFormValidationSuccess()
58 {
59     $fakeForm = $this->getMock(
60         'Zend\Form\Form', array('isValid', 'getData')
61     );
62
63     $fakeForm->expects($this->once())
64         ->method('isValid')
65         ->will($this->returnValue(true));
66
67     $fakeForm->expects($this->once())
68         ->method('getData')
69         ->will($this->returnValue(new \stdClass()));
70
71     $fakeMapper = $this->getMock('ZfDeals\Mapper\Product',
72         array('insert'),
73         array(),
74         '',
75         false
76     );
77
78     $fakeMapper->expects($this->once())
79         ->method('insert')
80         ->will($this->returnValue(true));
81
82     $this->controller->setProductAddForm($fakeForm);
83     $this->controller->setProductMapper($fakeMapper);

```



```

84         $this->request->setMethod('post');
85         $response = $this->controller->dispatch($this->request);
86     }
87 }

```

Listing 26.25

Хорошая отправная точка, я думаю. Чтобы запустить все тесты с помощью одной команды, я добавил это в файл `phpunit.xml`:

```

1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./ZfDealsTest/FormTest</directory>
5             <directory>./ZfDealsTest/ControllerTest</directory>
6         </testsuite>
7     </testsuites>
8 </phpunit>

```

Избегайте неоднозначных идентификаторов

Есть одна вещь, которая приходит мне в голову: что, собственно, случится, если я добавлю во второй раз продукт с таким же ID? Я разрушу систему, поскольку я сконфигурировал столбец, как уникальный в базе данных. Это первичный ключ. Мы можем обработать ситуацию, добавив соответствующий оператор `try/catch`:

```

1 <?php
2 public function addProductAction()
3 {
4     $form = $this->productAddForm;
5
6     if ($this->getRequest()->isPost()) {
7         $form->setData($this->getRequest()->getPost());
8
9         if ($form->isValid()) {
10             $model = new ViewModel(
11                 array(
12                     'form' => $form
13                 )
14             );
15
16             try {
17                 $this->productMapper->insert($form->getData());
18                 $model->setVariable('success', true);
19             } catch (\Exception $e) {
20                 $model->setVariable('insertError', true);
21             }
22
23             return $model;

```

```

24         } else {
25             return new ViewModel(
26                 array(
27                     'form' => $form
28                 )
29             );
30         }
31     } else {
32         return new ViewModel(
33             array(
34                 'form' => $form
35             )
36         );
37     }
38 }

```

Listing 26.26

Теперь файл представления выглядит следующим образом:

```

1 <?php if ($this->success) { ?>
2 <div class="alert alert-success">Товар добавлен в корзину!</div>
3 <?php } ?>
4
5 <?php if ($this->insertError) { ?>
6 <div class="alert alert-error">
7     Товар не может быть добавлен.
8 </div>
9 <?php } ?>
10
11 <?php
12 $this->form->prepare();
13 echo $this->form()->openTag($this->form);
14 echo $this->formRowTwb($this->form->get('product')->get('id'));
15 echo $this->formRowTwb($this->form->get('product')->get('name'));
16 echo $this->formRowTwb($this->form->get('product')->get('stock'));
17 echo $this->formSubmitTwb($this->form->get('submit'));
18 echo $this->form()->closeTag();

```

Listing 26.27

Давайте добавим еще один тест для проверки того, что try/catch работает, как необходимо. По ходу добавления теста я выполнил некоторую очистку тестового кода и извлек код для создания поддельных (fake) объектов:

```

1 <?php
2 namespace ZfDeals\ControllerTest;
3
4 use ZfDeals\Controller\AdminController;
5 use Zend\Http\Request;
6 use Zend\Http\Response;

```

```

7 use Zend\Mvc\MvcEvent;
8 use Zend\Mvc\Router\RouteMatch;
9
10 class AdminControllerTest extends \PHPUnit_Framework_TestCase
11 {
12     private $controller;
13     private $request;
14     private $response;
15     private $routeMatch;
16     private $event;
17
18     public function setUp()
19     {
20         $this->controller = new AdminController();
21         $this->request = new Request();
22         $this->response = new Response();
23         $this->routeMatch = new RouteMatch(array('controller' => 'admin'));
24         $this->routeMatch->setParam('action', 'add-product');
25         $this->event = new MvcEvent();
26         $this->event->setRouteMatch($this->routeMatch);
27         $this->controller->setEvent($this->event);
28     }
29
30     public function testShowFormOnGetRequest()
31     {
32         $fakeForm = new \Zend\Form\Form('fakeForm');
33         $this->controller->setProductAddForm($fakeForm);
34         $this->request->setMethod('get');
35         $response = $this->controller->dispatch($this->request);
36         $viewModelValues = $response->getVariables();
37         $formReturned = $viewModelValues['form'];
38         $this->assertEquals($formReturned->getName(), $fakeForm->getName());
39     }
40
41     public function testShowFormOnValidationError()
42     {
43         $fakeForm = $this->getFakeForm(false);
44         $this->controller->setProductAddForm($fakeForm);
45         $this->request->setMethod('post');
46         $response = $this->controller->dispatch($this->request);
47         $viewModelValues = $response->getVariables();
48         $formReturned = $viewModelValues['form'];
49         $this->assertEquals($formReturned->getName(), $fakeForm->getName());
50     }
51
52     public function testCallMapperOnFormValidationSuccessPersistenceSuccess()
53     {
54         $fakeForm = $this->getFakeForm();

```

```

55
56     $fakeForm->expects($this->once())
57         ->method('getData')
58         ->will($this->returnValue(new \stdClass()));
59
60     $fakeMapper = $this->getFakeMapper();
61
62     $fakeMapper->expects($this->once())
63         ->method('insert')
64         ->will($this->returnValue(true));
65
66     $this->controller->setProductAddForm($fakeForm);
67     $this->controller->setProductMapper($fakeMapper);
68     $this->request->setMethod('post');
69     $response = $this->controller->dispatch($this->request);
70     $viewModelValues = $response->getVariables();
71     $this->assertTrue(isset($viewModelValues['success']));
72 }
73
74 public function testCallMapperOnFormValidationSuccessPersistenceError()
75 {
76     $fakeForm = $this->getFakeForm();
77
78     $fakeForm->expects($this->once())
79         ->method('getData')
80         ->will($this->returnValue(new \stdClass()));
81
82     $fakeMapper = $this->getFakeMapper();
83
84     $fakeMapper->expects($this->once())
85         ->method('insert')
86         ->will($this->throwException(new \Exception));
87
88     $this->controller->setProductAddForm($fakeForm);
89     $this->controller->setProductMapper($fakeMapper);
90     $this->request->setMethod('post');
91     $response = $this->controller->dispatch($this->request);
92     $viewModelValues = $response->getVariables();
93     $this->assertTrue(isset($viewModelValues['form']));
94     $this->assertTrue(isset($viewModelValues['insertError']));
95 }
96
97 public function getFakeForm($isValid = true)
98 {
99     $fakeForm = $this->getMock(
100         'Zend\Form\Form', array('isValid', 'getData')
101     );
102

```

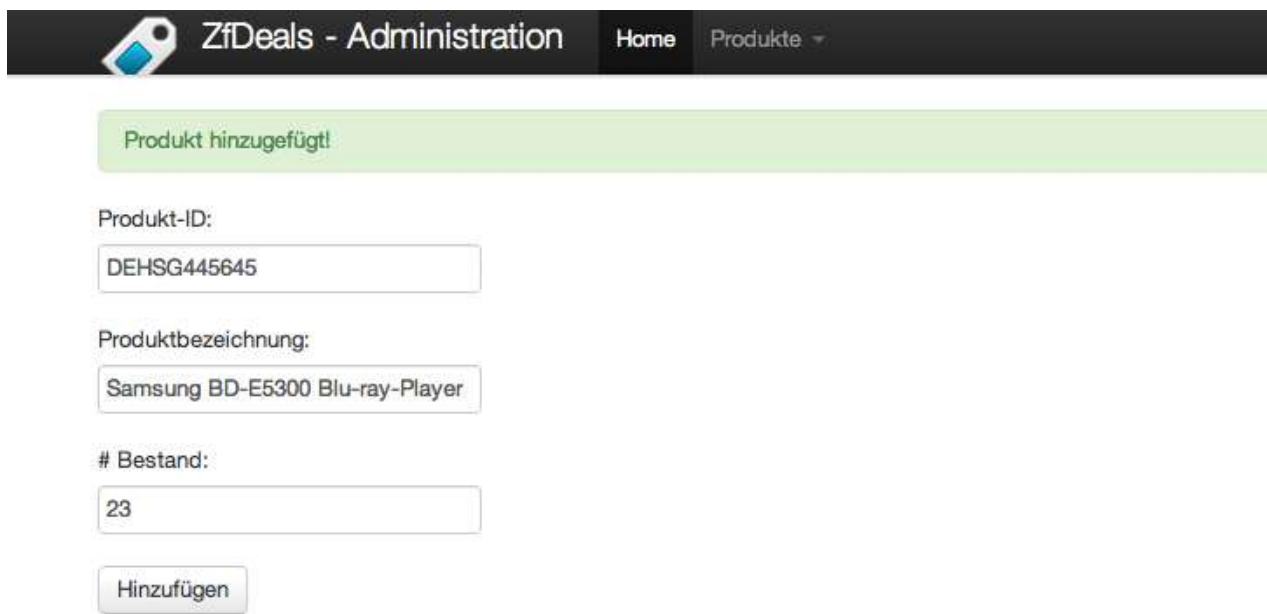
```

103     $fakeForm->expects($this->once())
104         ->method('isValid')
105         ->will($this->returnValue($isValid));
106
107     return $fakeForm;
108 }
109
110 public function getFakeMapper()
111 {
112     $fakeMapper = $this->getMock('ZfDeals\Mapper\Product',
113         array('insert'),
114         array(),
115         '',
116         false
117     );
118
119     return $fakeMapper;
120 }
121 }

```

Listing 26.28

И вот как теперь это выглядит:



The screenshot shows the 'ZfDeals - Administration' web interface. At the top, there is a navigation bar with 'Home' and 'Produkte' (with a dropdown arrow). Below the navigation bar, a green message box states 'Produkt hinzugefügt!'. The main content area contains a form with three input fields: 'Produkt-ID:' with the value 'DEHSG445645', 'Produktbezeichnung:' with the value 'Samsung BD-E5300 Blu-ray-Player', and '# Bestand:' with the value '23'. At the bottom of the form is a button labeled 'Hinzufügen'.

Поздравляю, это был спринт 2!

Спринт 3 – Добавление акции, показ доступных акций

Первая история пользователей для 3 спринта гласит:

"Для того, чтобы объявить акцию, мне нужно добавить ее в систему."

Критерии приемки:

- С помощью формы может быть добавлена новая акция на основании существующего в системе продукта с информацией о цене, датой начала, датой окончания и информацией о количестве.

Вторая пользовательская история для этого спринта гласит:

"Для того чтобы приобрести продукт, как клиент, я хочу увидеть все доступные акции с первого взгляда".

Критерии приемки:

- Отображаются все акции, которые существуют в настоящее время.
- Отображаются только те акции, для которых в наличии есть товар.

Но прежде, чем приступить к работе над новыми требованиями, я поработаю над некоторыми техническими усовершенствованиями.

Стандарт кодирования

ZF2 соблюдает стандарт кодирования PSR-2. (<https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md>) С самого начала я хочу следовать правилам PSR-2, чтобы быть уверенным, что мой код будет выглядеть знакомым для других, тех, кому необходимо понять, изменить или улучшить его. Для поддержки внедрения PSR-2 можно установить PHP_CodeSniffer. (http://pear.php.net/package/PHP_CodeSniffer/download) Он предустановлен на моем локальном компьютере, однако может быть быстро установлен с помощью PEAR. (<http://pear.php.net>)

```
1 $ pear install PHP_CodeSniffer-1.3.6
```

Теперь при запуске

```
1 $ phpcs -v --standard=psr2 ZfDeals
```

в каталоге модулей, PHP_CodeSniffer автоматически тестирует мой код модуля ZfDeals на соответствие PSR-2. К сожалению, при текущем коде у PHP_CodeSniffer есть много поводов для критики:

```
1 FILE: /module/ZfDeals/Module.php
```

```
2 -----
```

```
3 FOUND 7 ERROR(S) AFFECTING 3 LINE(S)
```

```

4 -----
5 11 | ERROR | Opening parenthesis of a multi-line
6 function call must be the last content on the line
7
8 11 | ERROR | Only one argument is allowed per line
9 in a multi-line function call
10
11 11 | ERROR | Only one argument is allowed per line
12 in a multi-line function call
13
14 11 | ERROR | Expected 1 space after FUNCTION keyword; 0 found
15
16 14 | ERROR | Only one argument is allowed per line
17 in a multi-line function call
18
19 14 | ERROR | Closing parenthesis of a multi-line function
20 call must be on a line by itself
21
22 32 | ERROR | Expected 1 blank line at end of file; 0 found

```

После того, как все вопросы по СК (стилю кодирования) устранены, PHP_Codesniffer утверждает мой код молчанием и ничего не выводит в командную строку.

Автоматическое исправление вопросов СК

Fabien Potencier недавно выпустил очень полезный инструмент под названием "PHP-CS-Fixer" для автоматического устранения типовых вопросов CS (СК, стилю кодирования). Стоит принять его во внимание.

Чтобы сделать проверку моего стиля кодирования как можно проще, я создаю небольшой вспомогательный сценарий в тестах:

```

1 <?php
2 echo shell_exec('phpcs --standard=psr2 ../module/ZfDeals') . PHP_EOL;

```

Listing 26.29

Теперь я легко могу проверить мой код на соответствие PSR-2, выполнив:

```

1 $ php checkstyle.php

```

Сценарий инициализации базы данных

Для работы ZfDeals необходима специальная схема базы данных. Так называемый "сценарий DDL" в /module/ZfDeals/data позволяет легко создать необходимую схему:

```

1 CREATE TABLE product (
2     id varchar(255) NOT NULL,
3     name varchar(255) NOT NULL,
4     stock int(10) NOT NULL, PRIMARY KEY (id)
5 );

```

Это отправная точка. Дополнительные определения данных будут, безусловно, добавлены в ближайшее время.

Сущность "акции"

Вернемся к пользовательской истории этого спринта. Во-первых, я написал код класса сущности акции:

```

1 <?php
2 namespace ZfDeals\Entity;
3
4 class Deal
5 {
6     protected $id;
7     protected $price;
8     protected $startDate;
9     protected $endDate;
10    protected $product;
11
12    public function setEndDate($endDate)
13    {
14        $this->endDate = $endDate;
15    }
16
17    public function getEndDate()
18    {
19        return $this->endDate;
20    }
21
22    public function setStartDate($startDate)
23    {
24        $this->startDate = $startDate;
25    }
26
27    public function getStartDate()
28    {
29        return $this->startDate;
30    }
31
32    public function setId($id)
33    {
34        $this->id = $id;
35    }
36

```



```

37     public function getId()
38     {
39         return $this->id;
40     }
41
42     public function setPrice($price)
43     {
44         $this->price = $price;
45     }
46
47     public function getPrice()
48     {
49         return $this->price;
50     }
51
52     public function setProduct($product)
53     {
54         $this->product = $product;
55     }
56
57     public function getProduct()
58     {
59         return $this->product;
60     }
61 }

```

Listing 26.30

Акция содержит свою цену (на товар), дату начала, дату окончания и ссылку на реализуемый товар. Следующая структура данных необходима для обеспечения персистентности акции, поэтому я добавляю ее в `structure.sql`:

```

1 CREATE TABLE deal(
2     id int(10) NOT NULL AUTO_INCREMENT,
3     price float NOT NULL,
4     startDate date NOT NULL,
5     endDate date NOT NULL,
6     product varchar(255) NOT NULL,
7     PRIMARY KEY (id)
8 );

```

Добавление новой акции

Я добавил раздел "Акции" и элемент "Добавить акцию" к панели навигации администратора. Он указывает на `/deals/admin/deal/add`. Как обычно, чтобы все работало, я создал маршрут в `module.config.php`. Форма `DealAdd` используется для добавления новой акции в систему:

```

1 <?php

```

```

2 namespace ZfDeals\Form;
3
4 use Zend\Form\Form;
5 use Zend\ServiceManager\ServiceManager;
6 use Zend\ServiceManager\ServiceManagerAwareInterface;
7
8 class DealAdd extends Form
9 {
10     public function __construct()
11     {
12         parent::__construct('dealAdd');
13         $this->setAttribute('action', '/deals/admin/deal/add');
14         $this->setAttribute('method', 'post');
15
16         $this->add(
17             array(
18                 'type' => 'ZfDeals\Form\DealFieldset',
19                 'options' => array(
20                     'use_as_base_fieldset' => true
21                 )
22             )
23         );
24
25         $this->add(
26             array(
27                 'name' => 'submit',
28                 'attributes' => array(
29                     'type' => 'submit',
30                     'value' => 'Добавить'
31                 ),
32             )
33         );
34     }
35 }

```

Listing 26.31

Определение DealFieldset:

```

1 <?php
2 namespace ZfDeals\Form;
3
4 use Zend\Form\Fieldset;
5 use Zend\InputFilter\InputFilterInterface;
6
7 class DealFieldset extends Fieldset
8 {
9     public function __construct()
10     {
11         parent::__construct('deal');

```

```

12
13     $this->add(
14         array(
15             'name' => 'product',
16             'type' => 'ZfDeals\Form\ProductSelectorFieldset',
17         )
18     );
19
20     $this->add(
21         array(
22             'name' => 'price',
23             'type' => 'Zend\Form\Element\Number',
24             'attributes' => array(
25                 'step' => 'any'
26             ),
27             'options' => array(
28                 'label' => 'Preis:',
29             )
30         )
31     );
32
33     $this->add(
34         array(
35             'name' => 'startDate',
36             'type' => 'Zend\Form\Element\Date',
37             'options' => array(
38                 'label' => 'Startdatum:'
39             ),
40         )
41     );
42
43     $this->add(
44         array(
45             'name' => 'endDate',
46             'type' => 'Zend\Form\Element\Date',
47             'options' => array(
48                 'label' => 'Enddatum:'
49             ),
50         )
51     );
52 }
53 }

```

Listing 26.32

DealFieldset содержит ProductSelectorFieldset, который служит для выбора продукта при добавлении новой акции:

```

1 <?php
2 namespace ZfDeals\Form;

```

```

3
4 use Zend\Form\Fieldset;
5 use Zend\InputFilter\InputFilterInterface;
6
7 class ProductSelectorFieldset extends Fieldset
8 {
9     public function __construct()
10    {
11        parent::__construct('productSelector');
12        $this->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());
13        $this->setObject(new \ZfDeals\Entity\Product());
14
15        $this->add(
16            array(
17                'name' => 'id',
18                'type' => 'Zend\Form\Element\Select',
19                'options' => array(
20                    'label' => 'Продукт-ID:',
21                    'value_options' => array(
22                        '1' => 'Label 1',
23                        '2' => 'Label 2',
24                    ),
25                )
26            )
27        );
28    }
29 }

```

Listing 26.33

Список инициализирован фиктивными данными. Настоящие товары добавляются контроллером до фактической визуализации формы.

Расширение INTL

Открытие нового URL приводит у меня к результату: "PHP Fatal error: Class 'NumberFormatter' not found". Причина в том, что расширение PHP INTL может отсутствовать в системе, но оно необходимо фреймворку, даже если вы непосредственно не будете иметь дело с такой функциональностью ZF2, как I18N. На Linux-системах можно легко установить расширение INTL, выполнив `apt-get install php5-intl`. Не забудьте перезапустить Apache после этого.

Обработка формы выполняется в `AdminController`. Я расширяю `AdminControllerFactory` таким образом, что форма `DealAdd` также внедряется:

```
1 <?php
```

```

2 namespace ZfDeals\Controller;
3
4 use Zend\ServiceManager\FactoryInterface;
5 use Zend\ServiceManager\ServiceLocatorInterface;
6
7 class AdminControllerFactory implements FactoryInterface
8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $ctr = new AdminController();
12        $productAddForm = new \ZfDeals\Form\ProductAdd();
13        $productAddForm->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
14        $productAddForm->bind(new \ZfDeals\Entity\Product());
15        $ctr->setProductAddForm($productAddForm);
16
17        $mapper = $serviceLocator->getServiceLocator()
18            ->get('ZfDeals\Mapper\Product');
19
20        $ctr->setProductMapper($mapper);
21        $dealAddForm = new \ZfDeals\Form\DealAdd();
22        $ctr->setDealAddForm($dealAddForm);
23
24        $dealAddForm
25            ->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
26
27        $dealAddForm->bind(new \ZfDeals\Entity\Deal());
28
29        $dealMapper = $serviceLocator->getServiceLocator()
30            ->get('ZfDeals\Mapper\Deal');
31
32        $ctr->setDealMapper($dealMapper);
33        return $ctr;
34    }
35 }

```

Listing 26.34

Код обработки формы находится в `addDealAction` контроллера `AdminController`. Во-первых, я хочу, чтобы `ProductSelectorFieldset` инициализировался реальными данными товара, извлекаемыми из базы данных. `DealMapper` отвечает за добавление акций в базу данных, а также за поиск активных акций:

```

1 <?php
2
3 namespace ZfDeals\Mapper;
4
5 use ZfDeals\Entity\Deal as DealEntity;
6 use Zend\Stdlib\Hydrator\HydratorInterface;
7 use Zend\Db\TableGateway\TableGateway;

```

```

8 use Zend\Db\TableGateway\Feature\RowGatewayFeature;
9 use Zend\Db\Sql\Sql;
10 use Zend\Db\Sql\Insert;
11
12 class Deal extends TableGateway
13 {
14     protected $tableName = 'deal';
15     protected $idCol = 'id';
16     protected $entityPrototype = null;
17     protected $hydrator = null;
18
19     public function __construct($adapter)
20     {
21         parent::__construct(
22             $this->tableName,
23             $adapter,
24             new RowGatewayFeature($this->idCol)
25         );
26
27         $this->entityPrototype = new DealEntity();
28         $this->hydrator = new \Zend\Stdlib\Hydrator\Reflection;
29     }
30
31     public function insert($entity)
32     {
33         return parent::insert($this->hydrator->extract($entity));
34     }
35
36     public function findActiveDeals()
37     {
38         $sql = new \Zend\Db\Sql\Sql($this->getAdapter());
39         $select = $sql->select()
40             ->from($this->tableName)
41             ->join('product', 'deal.product=product.id')
42             ->where('DATE(startDate) <= DATE(NOW())')
43             ->where('DATE(endDate) >= DATE(NOW())')
44             ->where('stock > 0');
45
46         $stmt = $sql->prepareStatementForSqlObject($select);
47         $results = $stmt->execute();
48
49         return $this->hydrate($results);
50     }
51
52     public function hydrate($results)
53     {
54         $deals = new \Zend\Db\ResultSet\HydratingResultSet(
55             $this->hydrator,

```

```

56         $this->entityPrototype
57     );
58
59     return $deals->initialize($results);
60 }
61 }

```

Listing 26.35

Мое действие addDealAction еще незатейливо, однако многое уже становится запутанным:

```

1 <?php
2 // [...]
3 public function addDealAction()
4 {
5     $form = $this->dealAddForm;
6
7     $products = $this->productMapper->select();
8     $fieldElements = array();
9
10    foreach ($products as $product) {
11        $fieldElements[$product['id']] = $product['name'];
12    }
13
14    $form->get('deal')->get('product')
15        ->get('id')->setValueOptions($fieldElements);
16
17    if ($this->getRequest()->isPost()) {
18        $form->setData($this->getRequest()->getPost());
19
20        if ($form->isValid()) {
21            $model = new ViewModel(
22                array(
23                    'form' => $form
24                )
25            );
26
27            $newDeal = $form->getData();
28            $newDeal->setProduct($newDeal->getProduct()->getId());
29
30            try {
31                $this->dealMapper->insert($newDeal);
32                $model->setVariable('success', true);
33            } catch (\Exception $e) {
34                $model->setVariable('insertError', true);
35            }
36
37            return $model;
38        } else {
39            return new ViewModel(

```

```

40         array(
41             'form' => $form
42         )
43     );
44 }
45 } else {
46     return new ViewModel(
47         array(
48             'form' => $form
49         )
50     );
51 }
52 }
53 // [...]

```

Listing 26.36

Показать текущие акции

Я создаю контроллер `IndexController` и его фабрику `IndexControllerFactory` для отображения текущих акций для клиентов. `IndexControllerFactory` внедряет mapper `Deal` для получения активных акций из базы данных:

```

1 <?php
2 namespace ZfDeals\Controller;
3
4 use Zend\Mvc\Controller\AbstractActionController;
5 use Zend\View\Model\ViewModel;
6 use Zend\Form\Annotation\AnnotationBuilder;
7
8 class IndexController extends AbstractActionController
9 {
10     private $dealMapper;
11     private $productMapper;
12
13     public function indexAction()
14     {
15         $deals = $this->dealMapper->findActiveDeals();
16         $dealsView = array();
17
18         foreach ($deals as $deal) {
19             $deal->setProduct(
20                 $this->productMapper->findOneById($deal->getProduct())
21             );
22
23             $dealsView[] = $deal;
24         }
25
26         return new ViewModel(
27             array(

```



```

28         'deals' => $dealsView
29     )
30 );
31 }
32
33 public function setDealMapper($dealMapper)
34 {
35     $this->dealMapper = $dealMapper;
36 }
37
38 public function getDealMapper()
39 {
40     return $this->dealMapper;
41 }
42
43 public function setProductMapper($productMapper)
44 {
45     $this->productMapper = $productMapper;
46 }
47
48 public function getProductMapper()
49 {
50     return $this->productMapper;
51 }
52 }

```

Listing 26.37

Цикл `foreach` здесь не очень элегантен, но поскольку я не исключаю на данный момент использования больших объемов, я считаю, что пока что с это можно оставить и улучшить это позже. Тем не менее, я знаю, что это узкое место.

Я буду использовать другой макет для показа акций клиентам. Это настраивается в классе `Module` в методе инициализации `init`:

```

1 <?php
2 [...]
3 public function init(ModuleManager $moduleManager)
4 {
5     $sharedEvents = $moduleManager->getEventManager()->getSharedManager();
6
7     $sharedEvents->attach(
8         'ZfDeals\Controller\AdminController',
9         'dispatch',
10        function ($e) {
11            $controller = $e->getTarget();
12            $controller->layout('zf-deals/layout/admin');
13        },
14        100
15    );

```

```

16
17     $sharedEvents->attach(
18         'zfDeals\Controller\IndexController',
19         'dispatch',
20         function ($e) {
21             $controller = $e->getTarget();
22             $controller->layout('zf-deals/layout/site');
23         },
24         100
25     );
26 }
27 [...]

```

Listing 26.38

Теперь /deals выводит список всех текущих акций в системе.

Пользовательский тип контроллера для работы с формами

Мне еще не очень нравится код. Во-первых, давайте рассмотрим AdminController: у него есть два действия add. Одно для продуктов, одно для акций. Оба метода выглядят, подобно "скопированы и вставлены", поскольку по большей части делают одно и то же: обработку формы. И оба выглядят громоздко и слишком вложено. Причина в том, что одно и то же действие отображает форму, выполняет валидацию данных, обработку ошибок, а также сохранение данных. Может быть, это слишком большая ответственность для одного действия, и это должно быть реорганизовано. То же самое верно для его файла шаблона. Другой вопрос заключается в том, что IndexControllerFactory по умолчанию создает и внедряет в оба действия формы каждый раз, даже если только одна форма будет использоваться на данный момент. Это не выглядит целесообразным.

Чтобы исправить все упомянутые вопросы одним росчерком, я решил воспользоваться тем, что контроллер в ZF2 просто требует наличия метода onDispatch() для нормального функционирования, создающего ответ (Response) на основе заданного запроса (Request). Это позволяет мне выиграть за счет использования специального типа контроллера, созданного для работы с формами своим собственным образом, вышеуказанного абстрактного класса контроллера, называющегося AbstractFormController:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractController;
5  use Zend\Mvc\MvcEvent;
6  use Zend\Form\Form as Form;
7  use Zend\View\Model\ViewModel;
8
9  abstract class AbstractFormController extends AbstractController
10 {
11     protected $form;
12

```

```

13 public function __construct(Form $form)
14 {
15     $this->form = $form;
16 }
17
18 public function onDispatch(MvcEvent $e)
19 {
20     if (method_exists($this, 'prepare')) {
21         $this->prepare();
22     }
23
24     $routeMatch = $e->getRouteMatch();
25
26     if ($this->getRequest()->isPost()) {
27         $this->form->setData($this->getRequest()->getPost());
28
29         if ($this->form->isValid()) {
30             $routeMatch->setParam('action', 'process');
31             $return = $this->process();
32         } else {
33             $routeMatch->setParam('action', 'error');
34             $return = $this->error();
35         }
36
37     } else {
38         $routeMatch->setParam('action', 'show');
39         $return = $this->show();
40     }
41
42     $e->setResult($return);
43     return $return;
44 }
45
46 abstract protected function process();
47
48 protected function show()
49 {
50     return new ViewModel(
51         array(
52             'form' => $this->form
53         )
54     );
55 }
56
57 protected function error()
58 {
59     return new ViewModel(
60         array(

```

```

61         'form' => $this->form
62     )
63 );
64 }
65
66 public function setForm($form)
67 {
68     $this->form = $form;
69 }
70
71 public function getForm()
72 {
73     return $this->form;
74 }
75 }

```

Listing 26.39

И вот как это работает: при маршрутизации запрашивается подходящий URL-адресу контроллер, наследующий `AbstractFormController`. Вызывается его метод `dispatch()`, а затем `onDispatch()`. Вместо вызова действия, что обычно происходит, когда используется обычный известный фреймворку `AbstractActionController`, вызывается другой метод, основанный на режиме обработки формы. Позвольте мне объяснить, что дальше.

Если url-адрес запрошен с помощью HTTP-метода GET, очевидно, что мы просто должны первоначально отобразить форму для пользователя. Следовательно, вызывается `show()`. Это уже дает `AbstractFormController`. Конкретный контроллер на основе `AbstractFormController` может перезаписать этот метод, если у него есть особые потребности отображения формы. `AbstractFormController` разработан таким образом, что при создании экземпляра данная форма должна быть введена. Таким образом, при запросе POST контроллер может просто вызвать метод формы `isValid()`. Если валидация формы была успешной, вызывается `process()`. Если нет, вместо этого вызывается метод `error()`. Поскольку обработка (processing) успешно проверенной формы зависит от запрошенной формы, `AbstractFormController` предоставляет только абстрактный метод `process()`. Он должен быть реализован индивидуально разработчиком приложения.

Короче говоря, `AbstractFormController` помогает предотвратить "спагетти-код" (<http://ru.wikipedia.org/wiki/Спагетти-код>) при работе контроллера с формами. Вы можете удивиться, почему все еще используется `setParam()` для установки значения ключа `action`. Это просто для того, чтобы обеспечить, чтобы подстановка шаблона позднее все еще работала. Поэтому для правильной работы `AbstractFormController` необходимы шаблоны `show.phtml`, `process.phtml` и `error.phtml`.

Теперь с использованием `AbstractFormController` каждая форма получает свой собственный контроллер. `ProductAddFormController` выглядит теперь так:

```

1 <?php
2 namespace ZfDeals\Controller;

```

```

3
4 use Zend\Mvc\Controller\AbstractActionController;
5 use Zend\View\Model\ViewModel;
6 use Zend\Stdlib\Hydrator\Reflection;
7 use ZfDeals\Entity\Product as ProductEntity;
8 use ZfDeals\Form\ProductAdd as ProductAddForm;
9
10 class ProductAddFormController extends AbstractFormController
11 {
12     private $productMapper;
13
14     public function __construct(ProductAddForm $form)
15     {
16         parent::__construct($form);
17     }
18
19     public function prepare()
20     {
21         $this->form->setHydrator(new Reflection());
22         $this->form->bind(new ProductEntity());
23     }
24
25     public function process()
26     {
27         $model = new ViewModel(
28             array(
29                 'form' => $this->form
30             )
31         );
32
33         try {
34             $this->productMapper->insert($this->form->getData());
35             $model->setVariable('success', true);
36         } catch (\Exception $e) {
37             $model->setVariable('insertError', true);
38         }
39
40         return $model;
41     }
42
43     public function setProductMapper($productMapper)
44     {
45         $this->productMapper = $productMapper;
46     }
47
48     public function getProductMapper()
49     {
50         return $this->productMapper;

```

```
51     }
52 }
```

Listing 26.40

DealAddFormController выглядит так:

```
1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6  use Zend\Stdlib\Hydrator\Reflection;
7  use ZfDeals\Entity\Deal as DealEntity;
8
9  class DealAddFormController extends AbstractFormController
10 {
11     private $productMapper;
12     private $dealMapper;
13
14     public function prepare()
15     {
16         $this->form->setHydrator(new Reflection());
17         $this->form->bind(new DealEntity());
18
19         $products = $this->productMapper->select();
20         $fieldElements = array();
21
22         foreach ($products as $product) {
23             $fieldElements[$product['id']] = $product['name'];
24         }
25
26         $this->form->get('deal')
27             ->get('product')
28             ->get('id')->setValueOptions($fieldElements);
29     }
30
31     public function process()
32     {
33         $model = new ViewModel(
34             array(
35                 'form' => $this->form
36             )
37         );
38
39         $newDeal = $this->form->getData();
40         $newDeal->setProduct($newDeal->getProduct()->getId());
41
42         try {
43             $this->dealMapper->insert($newDeal);
```

```

44         $model->setVariable('success', true);
45     } catch (\Exception $e) {
46         $model->setVariable('insertError', true);
47     }
48
49     return $model;
50 }
51
52 public function setProductMapper($productMapper)
53 {
54     $this->productMapper = $productMapper;
55 }
56
57 public function getProductMapper()
58 {
59     return $this->productMapper;
60 }
61
62 public function setDealMapper($dealMapper)
63 {
64     $this->dealMapper = $dealMapper;
65 }
66
67 public function getDealMapper()
68 {
69     return $this->dealMapper;
70 }
71 }

```

Listing 26.41

Метод `prepare()` позволяет запустить код инициализации до обработки формы, если он реализован контроллером.

Юнит-тесты для **AbstractFormController**

Теперь я добавляю юнит-тесты для `AbstractFormController`. Так я протестирую основную логику обработки форм в одном месте, что сэкономит мне много времени, поскольку я могу теперь отбросить большинство индивидуальных тестов форм:

```

1  <?php
2  namespace ZfDeals\ControllerTest;
3
4  use ZfDeals\Controller\AbstractFormController;
5  use Zend\Http\Request;
6  use Zend\Http\Response;
7  use Zend\Mvc\MvcEvent;
8  use Zend\Mvc\Router\RouteMatch;
9  use ZfDeals\Form\ProductAdd as ProductAddForm;
10

```

```

11 class AbstractFormControllerTest extends \PHPUnit_Framework_TestCase
12 {
13     private $controller;
14     private $request;
15     private $response;
16     private $routeMatch;
17     private $event;
18
19     public function setUp()
20     {
21         $fakeController = $this->getMockForAbstractClass(
22             'ZfDeals\Controller\AbstractFormController',
23             array(),
24             '',
25             false
26         );
27
28         $this->controller = $fakeController;
29         $this->request = new Request();
30         $this->response = new Response();
31
32         $this->routeMatch = new RouteMatch(
33             array('controller' => 'abstract-form')
34         );
35
36         $this->event = new MvcEvent();
37         $this->event->setRouteMatch($this->routeMatch);
38         $this->controller->setEvent($this->event);
39     }
40
41     public function testShowOnGetRequest()
42     {
43         $this->form = new \Zend\Form\Form('fakeForm');
44         $this->controller->setForm($this->form);
45         $this->request->setMethod('get');
46         $response = $this->controller->dispatch($this->request);
47         $viewModelValues = $response->getVariables();
48         $formReturned = $viewModelValues['form'];
49
50         $this->assertEquals(
51             $formReturned->getName(), $this->form->getName()
52         );
53     }
54
55     public function testErrorOnValidationError()
56     {
57         $fakeForm = $this->getMock(
58             'Zend\Form\Form', array('isValid')

```



```

59         );
60
61         $fakeForm->expects($this->once())
62             ->method('isValid')
63             ->will($this->returnValue(false));
64
65         $this->controller->setForm($fakeForm);
66         $this->request->setMethod('post');
67         $response = $this->controller->dispatch($this->request);
68         $viewModelValues = $response->getVariables();
69         $formReturned = $viewModelValues['form'];
70         $this->assertEquals($formReturned, $fakeForm);
71     }
72
73     public function testProcessOnValidationSuccess()
74     {
75         $fakeForm = $this->getMock(
76             'Zend\Form\Form', array('isValid')
77         );
78
79         $fakeForm->expects($this->once())
80             ->method('isValid')
81             ->will($this->returnValue(true));
82
83         $this->controller->setForm($fakeForm);
84         $this->request->setMethod('post');
85
86         $this->controller->expects($this->once())
87             ->method('process')
88             ->will($this->returnValue(true));
89
90         $response = $this->controller->dispatch($this->request);
91     }
92 }

```

Listing 26.42

Использование замыканий, как простых фабрик

В моем коде у меня есть несколько полномасштабных фабрик, которые делают лишь крошечную работу. Например, `DealAddFormControllerFactory` только извлекает некоторые сервисы из `ServiceManager` их при создании контроллера:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class DealAddFormControllerFactory implements FactoryInterface

```

```

8 {
9     public function createService(ServiceLocatorInterface $serviceLocator)
10    {
11        $form = new \ZfDeals\Form\DealAdd();
12        $ctr = new DealAddFormController($form);
13
14        $dealMapper = $serviceLocator
15            ->getServiceLocator()->get('ZfDeals\Mapper\Deal');
16
17        $ctr->setDealMapper($dealMapper);
18
19        $productMapper = $serviceLocator->getServiceLocator()
20            ->get('ZfDeals\Mapper\Product');
21
22        $ctr->setProductMapper($productMapper);
23        return $ctr;
24    }
25 }

```

Listing 26.43

Для сохранения компактности кода (и даже снижая риск ошибок в моем приложении) я переношу фабрики в простые замыкания в конфигурационный файл модуля:

```

1 <?php
2 // [...]
3 'controllers' => array(
4     'invokables' => array(
5         'ZfDeals\Controller\Admin'
6         => 'ZfDeals\Controller\AdminController',
7     ),
8     'factories' => array(
9         'ZfDeals\Controller\DealAddForm' => function ($serviceLocator) {
10             $form = new ZfDeals\Form\DealAdd();
11             $ctr = new ZfDeals\Controller\DealAddFormController($form);
12
13             $dealMapper = $serviceLocator
14                 ->getServiceLocator()->get('ZfDeals\Mapper\Deal');
15
16             $ctr->setDealMapper($dealMapper);
17
18             $productMapper = $serviceLocator->getServiceLocator()
19                 ->get('ZfDeals\Mapper\Product');
20
21             $ctr->setProductMapper($productMapper);
22             return $ctr;
23         },
24         'ZfDeals\Controller\ProductAddForm' => function ($serviceLocator) {
25             $form = new \ZfDeals\Form\ProductAdd();
26             $ctr = new ZfDeals\Controller\ProductAddFormController($form);

```

```

27
28     $productMapper = $serviceLocator->getServiceLocator()
29         ->get('ZfDeals\Mapper\Product');
30
31     $ctr->setProductMapper($productMapper);
32     return $ctr;
33 },
34 'ZfDeals\Controller\Index' => function ($serviceLocator) {
35     $ctr = new ZfDeals\Controller\IndexController();
36
37     $productMapper = $serviceLocator->getServiceLocator()
38         ->get('ZfDeals\Mapper\Product');
39
40     $dealMapper = $serviceLocator->getServiceLocator()
41         ->get('ZfDeals\Mapper\Deal');
42
43     $ctr->setDealMapper($dealMapper);
44     $ctr->setProductMapper($productMapper);
45     return $ctr;
46 }
47 ),
48 ),

```

Listing 26.44

И это все, что касается спринта 3!

Спринт 4 – Заказы

И снова, перед началом работы с требованиями этого спринта я использую перерыв между последним и текущим спринтом, так называемый "период затишья", для дальнейшего приведения моего кода к лучшему виду. На сегодняшний день я в первую очередь хочу знать, каково мое "покрытие кода" (http://ru.wikipedia.org/wiki/Покрытие_кода). Оно описывает степень, в которой был протестирован исходный код программы. Благодаря PHPUnit это простая задача. Мне просто нужно добавить раздел logging в phpunit.xml:

```

1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./ZfDealsTest/FormTest</directory>
5             <directory>./ZfDealsTest/ControllerTest</directory>
6         </testsuite>
7     </testsuites>
8     <logging>
9         <log type="coverage-html"
10             target="./reports/coverage"
11             charset="UTF-8"
12             yui="true"
13             highlight="false"

```

```

14         lowUpperBound="35"
15         highLowerBound="70" />
16     </logging>
17 </phpunit>

```

Теперь при выполнении

```
1 $ phpunit
```

выполняются не только все тесты, но теперь также и "отчеты покрытия кода" записываются на диск. Они показывают общую степень покрытия с первого взгляда, а также подробную информацию построчно:

```

1 ZfDeals: 55.16%
2     config: 0%
3     src: 67.51%
4         Controller: 90.41%
5         Entity: 25%
6         Form: 79.31%
7         Mapper: 0%
8     Module.php: 0%

```

То, что я вижу – явно не идеально. В общем, желаемое покрытие кода 70-80%, а я вижу, что в некоторых областях у меня нет никаких тестов вообще. Над этим наверняка нужно будет работать.

Есть еще кое-что, прежде, чем я доберусь до пользовательской истории спринта 4: я с самого начала хочу убедиться, что пользовательский интерфейс ZfDeals поддерживает несколько языков и предоставляет по меньшей мере переводы для английского и немецкого языков. Во-первых, я могу удалить все языковые файлы, которые поставляются с ZendSkeletonApplication в модуле Application [внимание, не перепутайте! Автор говорит о файлах только модуля Application, т.е. расположенных в /module/Application/language/. Там действительно нет ничего ценного, разве только радость по поводу того, что вам удалось-таки установить ZendSkeletonApplication]. Мне они больше не нужны. Я также удалю мои пользовательские сообщения об ошибках для валидаторов форм и буду использовать сообщения об ошибках для ZF2 по умолчанию, которые уже переведены для наиболее распространенных языков. Поэтому я копирую Zend_Validate.php из vendor/zendframework/zendframework/resources/languages в каталог language модуля ZfDeals и настраиваю переводчик в module.config.php для выбора этого языкового файла:

```

1 <?php
2 // [...]
3 'translator' => array(
4     'locale' => 'de_DE',
5     'translation_file_patterns' => array(
6         array(
7             'type' => 'PhpArray',
8             'base_dir' => __DIR__ . '/../language',
9             'pattern' => '%s.php',
10        ),

```

```

11 )
12 )
13 // [...]

```

Listing 26.45

[Здесь нужно сказать, что для простоты автор слегка шукавил. Дело в том, что четвертым параметром при указании параметров может идти "text_domain". Это такой своеобразное пространство имен для наших языковых файлов. Обычно его задают таким образом, чтобы оно совпадало с пространством имен модуля. Если текстовый домен явно не задан, он принимает значение "default". Пример можно посмотреть здесь <http://samiminds.com/2012/09/zend-framework-2-translate-i18n-locale>. Важно в этом случае то, что как особо подчеркивает официальная документация, для каждой комбинации текстового домена и локали может быть загружен только один файл. Последующее добавление файла для той же локали и текстового домена аннулирует предыдущий файл. То есть автор здесь использует следующий подход – он берет стандартный файл перевода, который используется фреймворком, дописывает в него необходимый для модуля перевод – и получается один файл. С одной стороны это не совсем правильное решение, я бы даже сказал – совсем не правильное, поскольку делать это для каждого модуля не резонно. А с другой стороны – пример обучающий, а использование текстового домена

```

'translator' => array(
    'translation_file_patterns' => array(
        array(
            'type'      => 'gettext',
            'base_dir' => __DIR__ . '/../language',
            'pattern'   => '%s.mo',
            'text_domain' => __NAMESPACE__,
        ),
    ),
),

```

усложнит выполнение перевода, поскольку при вызове

```
$translator->translate($message, $textDomain, $locale);
```

придется явно указывать не только \$message, но и \$textDomain для нашего модуля, обычно так:

```
$translator->translate($message, __NAMESPACE__);
```

Правда, есть несколько способов, как этого можно избежать и сократить запись, но здесь я не буду их описывать.

]

Кроме того, я настраиваю мои валидаторы на применение переводчика для их сообщений об ошибках по умолчанию:

```

1 <?php
2 // [...]
3 public function onBootstrap($e)

```

```

4 {
5     \Zend\Validator\AbstractValidator::setDefaultTranslator(
6         $e->getApplication()->getServiceManager()->get('translator')
7     );
8
9     $eventManager = $e->getApplication()->getEventManager();
10    $moduleRouteListener = new ModuleRouteListener();
11    $moduleRouteListener->attach($eventManager);
12 }
13 // [...]

```

Listing 26.46

Теперь я один раз прохожу через все шаблоны представлений и формы модуля, чтобы убедиться, что переводчик используется всякий раз, когда отображается текст.

Пользовательская история этого спринта гласит:

"Для того, чтобы купить продукт со скидкой я, как клиент, хочу заполнить форму заказа."

Критерии приемки:

- У всех активных акций, показанных на сайте, есть кнопка "Купить".
- Нажатие на кнопку "купить" приводит клиента на форму запроса имени клиента и адреса доставки. Все данные обязательны.
- Список всех полученных заказов [кнопка просмотра заказов] добавляется в раздел администрирования ZfDeals.

Это более или менее теперь уже выглядит, как обычный бизнес. Для формы заказа я создал маршрут, формы, контроллер и сущность заказа. Заказ хранит ссылку на товар и сохраняет все дополнительные данные заказа. Новый маркер для заказа заботится о персистентности заказа. Сервис "Checkout" является первым "бизнес-сервисом" ZfDeals. Checkout заботится о добавлении нового заказа в систему, а также о снижении запасов заказанного продукта на единицу. Поэтому я поместил этот код в отдельный сервис – поскольку он касается двух отдельных сущностей, то, естественно, не вписывается ни в одну из них. Кроме того, я не хочу вводить логику проверки в CheckoutFormController, поскольку я, возможно, захочу использовать повторно эту логику в другом месте, в другом контроллере или веб-сервисе. Преимущественно, CheckoutService состоит из так называемого метода process():

```

1 <?php
2 public function process($ordering)
3 {
4     try {
5         $this->orderMapper->insert($ordering);
6         $deal = $this->dealMapper->findOneById($ordering->getDeal());
7         $product = $this->productMapper->findOneById($deal->getProduct());
8
9         $this->productMapper->update(

```

```

10         array('stock' => $product->getStock() - 1),
11         array('productId' => $product->getProductId())
12     );
13
14     } catch (\Exception $e) {
15         throw new \DomainException('Order could not be processed.');
```

Listing 26.47

Код прямолинеен. Он, конечно, еще не обрабатывает любой тип исключения, который может возникнуть, но на данный момент он удовлетворителен. По крайней мере, он делает то, что должен.

И вот оно!

Спринт 5 – Сделать ZfDeals доступным в качестве модуля ZF2

Пользовательская история для спринта 5:

"Чтобы располагать функциональностью ZfDeals в моем приложении я, как разработчик, хочу добавить ZfDeals в мой собственный код с помощью Composer."

Новый репозиторий для модуля ZfDeals

Прежде всего, я создал новый репозиторий, предназначенный для кода модуля [заметьте, отдельный для ZfDealsApp, отдельный для ZfDeals]. Я перемещаю весь код модуля из локального приложения в новый репозиторий.

```

1 $ cd ZfDealsApp/module/ZfDeals
2 $ git init
3 $ git add .
4 $ git commit -m "ZfDeals"
5 $ git remote add origin https://github.com/michael-romer/ZfDealsApp.git
6 $ git push -u origin master
```

[Вы должны указать свой собственный репозиторий, а не <https://github.com/michael-romer/ZfDealsApp.git>]

После того, как код добавлен в репозиторий на GitHub, я удаляю каталог ZfDeals из локального приложения. Изменения на диске нужно зафиксировать (занести в коммит):

```

1 $ cd ZfDealsApp
2 $ git add .
3 $ git commit -m "Application"
```

Затем я добавляю новый репозиторий (код модуля), как подмодуль `git` для существующего (кода локального приложения):

```
1 $ cd ZfDealsApp
1 $ git submodule add https://github.com/michael-romer/ZfDeals module/ZfDeals
2 $ git submodule init
3 $ git submodule update
```

[Здесь мы имитируем, что нам нужно в локальное приложение установить сторонний модуль. Для этого используется `git submodule add`, после чего регистрируем подмодуль для приложения с помощью `git submodule init`.]

Теперь у меня есть удобная среда разработки: в моей рабочей области у меня есть и основное приложение для моего модуля, и сам модуль. Поскольку оба управляются с помощью собственных хранилищ, я могу выполнить коммит для них индивидуально, в зависимости от того, где я запускаю команду `git` в командном интерпретаторе. Итак, у меня исходный код модуля отделен от всего остального, чтобы я мог его легко распространять. В целом, это хороший способ программировать модули ZF2 всякий раз, когда вы хотите сохранить код модуля отдельно.

При перемещении файлов из одного репозитория в другой, я включаю статические ресурсы модуля в `public`, а также языковой файл и юнит-тесты. Все они принадлежат коду модуля и распространяются вместе с ним. Кроме того, я перемещаю адаптер базы данных из `module.config.php` `ZfDeals` и взамен добавляю его в конфигурацию приложения. Это означает, что если `ZfDeals` используется в приложении, он требует от приложения предоставления адаптера базы данных для использования `ZfDeals`. Это общий шаблон для ZF2 модулей: самые основные, общесистемные сервисы, такие как адаптер базы данных, должны быть предоставлены основным приложением, а затем распределяться между всеми установленными модулями.

Чтобы сделать `ZfDeals` доступным с помощью `composer`, я добавляю файл `composer.json` в модуль:

```
1 {
2     "name": "zf2book/zf-deals",
3     "description": "This is the companion to the
4         book 'Webentwicklung mit Zend Framework 2'",
5     "type": "library",
6     "keywords": [
7         "zfdeals"
8     ],
9     "homepage": "http://zendframework2.de",
10    "authors": [
11        {
12            "name": "Michael Romer",
13            "email": "zf2buch@michael-romer.de",
14            "homepage": "http://zendframework2.de"
15        }
16    ]
17 }
```



```

16  ],
17  "require": {
18      "php": ">=5.3.3",
19      "zendframework/zendframework": "2.*"
20  },
21  "autoload": {
22      "psr-0": {
23          "ZfDeals": "src/"
24      },
25      "classmap": [
26          "./Module.php"
27      ]
28  }
29 }

```

В разделе `autoload` сконфигурированы все классы из каталога `src` для автозагрузки, а также основной класс модуля `Module`.

Теперь я могу добавить модуль в виде одного пакета Packagist в официальный репозиторий Composer-пакетов packagist.org [официальным, конечно, считается репозиторий packagist.org, хотя `composer` успешно работает и с GitHub, и с другими. Подробнее об этом здесь <http://habrahabr.ru/post/145946>] с использованием идентификатора `zf2book/zf-deals` (<https://packagist.org/packages/zf2book/zf-deals>). После этого `ZfDeal` теперь можно легко установить, пользуясь `Composer`. Не в последнюю очередь я добавляю `README.md`, содержащий инструкции по установке:

```

1  Install
2  =====
3
4  Main Install
5  -----
6
7  1. Add the following statement to the requirements-block
8  of your composer.json: "zf2book/zf-deals": "dev-master",
9  "dlu/dlutwbootstrap": "dev-master"
10
11 2. Run a composer update to download the libraries needed.
12
13 3. Add "ZfDeals" and "DluTwBootstrap" to the list of active
14 modules in `application.config.php`
15
16 4. Import the SQL schema located in
17 `/vendor/zf2book/zf-deals/data/structure.sql`
18
19 5. Copy `/vendor/zf2book/zf-deals/data/public/zf-deals`
20 to the public folder of your application.
21
22 Post Install
23 -----

```

```

24
25 1. If you do not already have a valid
26 Zend\Db\Adapter\Adapter in your service manager configuration,
27 put the following in `/config/autoload/db.local.php`:
28
29 <?php
30
31 $dbParams = array(
32     'database' => 'changeme',
33     'username' => 'changeme',
34     'password' => 'changeme',
35     'hostname' => 'changeme',
36 );
37
38 return array(
39     'service_manager' => array(
40         'factories' => array(
41             'Zend\Db\Adapter\Adapter' => function
42                 ($sm) use ($dbParams) {
43                 return new Zend\Db\Adapter\Adapter(array(
44                     'driver' => 'pdo',
45                     'dsn' =>
46                         'mysql:dbname='.$dbParams['database'].';host='.$dbParams['hostname'],
47                     'database' => $dbParams['database'],
48                     'username' => $dbParams['username'],
49                     'password' => $dbParams['password'],
50                     'hostname' => $dbParams['hostname'],
51                 ));
52             },
53         ),
54     ),
55 );
56
57 2. Navigate to http://yourproject/deals or http://yourproject/deals/admin

```

Как можно видеть, помимо установки модуля с помощью composer, есть еще несколько вещей, которые должен сделать разработчик, использующий ZfDeals:

- Помимо модуля ZfDeals в application.config.php должен быть добавлен модуль DluTwBootstrap. Это еще один ZF2-модуль, используемый нами для отображения форм.
- Должна быть создана структура базы данных, заданная с помощью structure.sql.
- Статические ресурсы, используемые ZfDeals, должны быть скопированы в каталог приложения public.

Упрощение конфигурации модуля

Файл module.config.php из ZfDeals выглядит уже несколько хаотично. Он содержит все определения маршрутов, а также все, что касается сервисов и контроллеров. Чтобы сделать

его более читаемым, я перемещаю некоторые определения из файла. Во-первых, я перемещаю все определения сервисов в `services.config.php`:

```
1 <?php
2 return array(
3     'factories' => array(
4         'ZfDeals\Mapper\Product' => function ($sm) {
5             return new \ZfDeals\Mapper\Product(
6                 $sm->get('Zend\Db\Adapter\Adapter')
7             );
8         },
9         'ZfDeals\Mapper\Deal' => function ($sm) {
10            return new \ZfDeals\Mapper\Deal(
11                $sm->get('Zend\Db\Adapter\Adapter')
12            );
13        },
14        'ZfDeals\Mapper\Order' => function ($sm) {
15            return new \ZfDeals\Mapper\Order(
16                $sm->get('Zend\Db\Adapter\Adapter')
17            );
18        },
19        'ZfDeals\Validator\DealAvailable' => function ($sm) {
20            $validator = new \ZfDeals\Validator\DealActive();
21            $validator->setDealMapper($sm->get('ZfDeals\Mapper\Deal'));
22
23            $validator->setProductMapper(
24                $sm->get('ZfDeals\Mapper\Product')
25            );
26
27            return $validator;
28        },
29        'ZfDeals\Service\Checkout' => function ($sm) {
30            $srv = new \ZfDeals\Service\Checkout();
31
32            $srv->setDealAvailable(
33                $sm->get('ZfDeals\Validator\DealAvailable')
34            );
35
36            $srv->setProductMapper($sm->get('ZfDeals\Mapper\Product'));
37            $srv->setOrderMapper($sm->get('ZfDeals\Mapper\Order'));
38            $srv->setDealMapper($sm->get('ZfDeals\Mapper\Deal'));
39            return $srv;
40        },
41    ),
42 );
```

Listing 26.48

Все определения контроллеров переходят в `controllers.config.php`:

```

1 <?php
2 return array(
3     'invokables' => array(
4         'ZfDeals\Controller\Admin' => 'ZfDeals\Controller\AdminController',
5     ),
6     'factories' => array(
7         'ZfDeals\Controller\CheckoutForm' => function ($serviceLocator) {
8             $form = new \ZfDeals\Form\Checkout();
9             $ctr = new ZfDeals\Controller\CheckoutFormController($form);
10
11             $productMapper = $serviceLocator
12                 ->getServiceLocator()->get('ZfDeals\Mapper\Product');
13
14             $ctr->setProductMapper($productMapper);
15
16             $validator = $serviceLocator->getServiceLocator()
17                 ->get('ZfDeals\Validator\DealAvailable');
18
19             $ctr->setdealActiveValidator($validator);
20
21             $checkoutService = $serviceLocator
22                 ->getServiceLocator()->get('ZfDeals\Service\Checkout');
23
24             $ctr->setCheckoutService($checkoutService);
25             return $ctr;
26         },
27         'ZfDeals\Controller\DealAddForm' => function ($serviceLocator) {
28             $form = new ZfDeals\Form\DealAdd();
29             $ctr = new ZfDeals\Controller\DealAddFormController($form);
30
31             $dealMapper = $serviceLocator
32                 ->getServiceLocator()->get('ZfDeals\Mapper\Deal');
33
34             $ctr->setDealMapper($dealMapper);
35
36             $productMapper = $serviceLocator
37                 ->getServiceLocator()->get('ZfDeals\Mapper\Product');
38
39             $ctr->setProductMapper($productMapper);
40             return $ctr;
41         },
42         'ZfDeals\Controller\ProductAddForm' => function ($serviceLocator) {
43             $form = new \ZfDeals\Form\ProductAdd();
44             $ctr = new ZfDeals\Controller\ProductAddFormController($form);
45
46             $productMapper = $serviceLocator
47                 ->getServiceLocator()->get('ZfDeals\Mapper\Product');
48

```

```

49         $ctr->setProductMapper($productMapper);
50         return $ctr;
51     },
52     'ZfDeals\Controller\Index' => function ($serviceLocator) {
53         $ctr = new ZfDeals\Controller\IndexController();
54
55         $productMapper = $serviceLocator
56             ->getServiceLocator()
57             ->get('ZfDeals\Mapper\Product');
58
59         $dealMapper = $serviceLocator->
60             getServiceLocator()->
61             get('ZfDeals\Mapper\Deal');
62
63         $ctr->setDealMapper($dealMapper);
64         $ctr->setProductMapper($productMapper);
65         return $ctr;
66     },
67     'ZfDeals\Controller\Order' => function ($serviceLocator) {
68         $ctr = new ZfDeals\Controller\OrderController();
69
70         $ctr->setOrderMapper($serviceLocator
71             ->getServiceLocator()->get('ZfDeals\Mapper\Order'));
72
73         return $ctr;
74     },
75 ),
76 );

```

Listing 26.49

Я добавил следующие методы в класс модуля, чтобы включить дополнительные файлы конфигурации:

```

1 <?php
2 // [...]
3 public function getServiceConfig()
4 {
5     return include __DIR__ . '/config/services.config.php';
6 }
7
8 public function getControllerConfig()
9 {
10     return include __DIR__ . '/config/controllers.config.php';
11 }
12 // [...]

```

Listing 26.50

Сам файл `module.config.php` сейчас только содержит определения маршрутов и конфигурацию представления (view).

Лучший подход для отображения форм

Я также могу настроить отображение форм. В настоящее время у меня во всех трех шаблонах `show.phtml`, в основном, следующий код:

```
1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRowTwb($this->form->get('product')->get('id'));
5 echo $this->formRowTwb($this->form->get('product')->get('name'));
6 echo $this->formRowTwb($this->form->get('product')->get('stock'));
7 echo $this->formSubmitTwb($this->form->get('submit'));
8 echo $this->form()->closeTag();
```

Listing 26.51

К тому же, если я добавлю еще одно поле в моей форме, мне также нужно будет изменять шаблон, чтобы появилось новое поле. Здесь может помочь пользовательский "помощник представления". Следующие строки кода выполняют динамическую визуализацию кода на основе определения формы. Это позволяет выполнить визуализацию формы в одну строчку:

```
1 <?php
2 echo $this->renderForm($form);
```

Listing 26.52

Сам помощник представления `RenderForm` выглядит так:

```
1 <?php
2 namespace ZfDeals\View\Helper;
3
4 use Zend\View\Helper\AbstractHelper;
5
6 class RenderForm extends AbstractHelper
7 {
8     public function __invoke($form)
9     {
10         $form->prepare();
11         $html = $this->view->form()->openTag($form) . PHP_EOL;
12         $html .= $this->renderFieldsets($form->getFieldsets());
13         $html .= $this->renderElements($form->getElements());
14         $html .= $this->view->form()->closeTag($form) . PHP_EOL;
15         return $html;
16     }
17
18     private function renderFieldsets($fieldsets)
19     {
```

```

20     $html = '';
21
22     foreach($fieldsets as $fieldset)
23     {
24         if(count($fieldset->getFieldsets()) > 0) {
25             $html .= $this->renderFieldsets(
26                 $fieldset->getFieldsets()
27             );
28         }
29
30         $html .= $this->renderElements(
31             $fieldset->getElements()
32         );
33     }
34
35     return $html;
36 }
37
38 private function renderElements($elements)
39 {
40     $html = '';
41
42     foreach($elements as $element) {
43         $html .= $this->renderElement($element);
44     }
45
46     return $html;
47 }
48
49 private function renderElement($element)
50 {
51     if($element->getAttribute('type') == 'submit') {
52         return $this->view->formSubmitTwo($element) . PHP_EOL;
53     } else {
54         return $this->view->formRow($element) . PHP_EOL;
55     }
56 }
57 }

```

Listing 26.53

Важным аспектом `RenderForm` является то, что он обрабатывает имеющиеся `fieldsets` с помощью рекурсии. `RenderForm` должен быть объявлен до того, как его можно будет использовать в представлениях (views). Опять же, чтобы конфигурация модулей оставалась легковесной, его объявление помещается в отдельный файл, называемый `viewhelper.config.php`:

```

1 <?php
2 return array(

```

```

3     'invokables' => array(
4         'renderForm' => 'ZfDeals\View\Helper\RenderForm'
5     )
6 );

```

Listing 26.54

Метод `getViewHelperConfig()` в классе `Module` предоставляет его модулю:

```

1 <?php
2 // [...]
3 public function getViewHelperConfig()
4 {
5     return include __DIR__ . '/config/viewhelper.config.php';
6 }
7 // [...]

```

Listing 26.55

Теперь `ZfDeals` готов к использованию в сторонних приложениях!

Что дальше?

Конечно, `ZfDeals` еще не особо богат возможностями, но уже полностью функционален и немного полезен. Теперь мы можем двигаться дальше и добавлять функциональность за функциональностью. Кроме того, есть еще некоторый рефакторинг, который я мог бы выполнить однажды. Посмотрим!

Zend\Di для внедрения зависимостей

Я написал много кода в моем приложении по применению внедрения зависимостей. Большинство созданных фабрик выглядят сложнее, чем следовало бы. В основном, достаточно просто создать сервис или контроллер только для получения других сервисов и внедрения их с помощью "методов-сеттеров". Кажется, что я нарушаю принцип DRY (http://ru.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself). Действительно ли мне нужно повторяться и создавать отдельные фабрики только для внедрения зависимостей? Я подумаю об использовании `Zend\Di`. Это более или менее устраняет все мои ручную запрограммированные фабрики.

Doctrine 2 для персистентности данных

Далее, огромная часть моего кода занимается чтением и записью данных в базу данных. Системы, как ORM `Doctrine 2` могут почти полностью исключить весь этот пользовательский код, сделав персистентность данных полностью прозрачной для пользователя. Просто работайте с вашими PHP-объектами, и пусть `Doctrine` заботится о всей работе над персистентностью.

Использование системы событий ZF2

В настоящее время я предполагаю, что ZfDeals используется "как есть". У нее почти нет параметров для настройки. Но что, если, к примеру, кто-то хочет выслать уведомление о заказе на обслуживание клиентов? Чтобы сделать систему событий более расширяемой, я могу внедрить систему событий ZF2 и инициировать надлежащее событие, если оно произойдет. Это позволит разработчикам добавить событие на основе пользовательского кода, чтобы расширить или изменить логику работы ZfDeals.

Улучшение обработки статических ресурсов

Копирование статических ресурсов модуля в каталог `public` приложения подвержено ошибкам и неудобно. Я могу внедрить подходящий менеджер ресурсов (Asset Manager), подобный `zf2-module-assets` (<https://github.com/albulescu/zf2-module-assets>) или `Assetic` и его связующий с ZF2-модулями код (<https://github.com/widmogrod/zf2-assetic-module>), позволяющие статические ресурсы напрямую внутри модулей.

Исходный код и послесловие – от переводчика

К данному материалу приложен исходный код для "Дневников разработчика". Это сделано потому, что книга развивается и исходный код, предоставленный автором – также. Таким образом, вы будете располагать примерами кода, актуальными на момент ознакомления с этой версии книги.

Оригинал книги можно приобрести на Amazon или leanpub. По мере развития книги, автор обеспечивает почтовую рассылку (по крайней мере, для тех, кто приобрел книгу на Amazon), и эта рассылка позволяет вам оставаться "в курсе событий".