ООП*, ПРОГРАММИРОВАНИЕ*, РАЗРАБОТКА ВЕБ-САЙТОВ*

MVC и Модель 2. Знания и обязанности компонентов

xskif 26 февраля 2015 в 13:55

Долгое время я изучал паттерн MVC. Больше полутора лет прошло с тех пор, как я впервые с ним познакомился и в течение всего этого времени я никак не мог упорядочить в своей голове зоны ответственности трех составляющих паттерн компонентов.

MVC — это сложное, но потрясающе изящное архитектурное решение. Я не представляю, во что бы превратились современные приложения без данного паттерна.

В интернете вся информация разбросана какими-то кусками, и теперь, через полтора года знакомства и бесконечных исследований, я наконец могу сказать: да, я знаю этот паттерн вдоль и поперек.

Я решил собрать всю недостающую информацию в одном месте. Это и стало причиной для написания статьи.

tl;dr: читаем итог. Остальных прошу устроиться поудобнее.

Согласно «банде четырех», MVC есть ни что иное, как стратегия + компоновщик + наблюдатель. Контроллер и представление

регистрируются у модели как наблюдатели. Представление регистрирует у себя контроллер как объект стратегии, реализуя свое поведение через композицию. Модель занимается оповещением наблюдателей при изменении состояния. Представление вкладывает внутрь себя другие представления, которые обычно зависят от своих моделей и представляют пользователю иное поведение через другие контроллеры. Это не HMVC. Это классика жанра.

В веб-разработке MVC является самым популярным паттерном. Сегодня понятие MVC-фреймворк достаточно обыденно, и мы привыкли считать, что окружающие знают об этих фреймворках достаточно, чтобы правильно разделять код на три и более составляющих. К сожалению, из-за повсеместной распространенности MVC-фреймворков и их попыток предоставить своим пользователям простоту в освоении, разработке и поддержке приложений, азы основополагающего паттерна потерялись.

Начнем с того, что MVC, как он есть, невозможно реализовать в клиент-серверном приложении из-за отсутствия постоянного рантайма во время смены состояния моделей. Я говорю сейчас именно о наблюдателе. Нет никакого смысла в обновлении представления после изменения модели, потому что представление мы получаем на клиенте, а модель лежит далеко на сервере. В свое время для решения данной проблемы в Java Web Applications был разработан паттерн Модель 2, основанный на старом добром MVC.

Суть паттерна — подстроить MVC под http протокол и реалии Интернета. Данный паттерн рассматривал контроллер как объект, принимающий запрос от пользователя. Контроллер должен разобрать запрос, на его основе инстанцировать определенную модель и вывести определенное представление, осуществив проброс модели и других данных в представление. Само по себе представление является обычным JSP. Модель использовалась для общения с базой данных и отображения строк таблицы в объекте (ORM). Выглядит чертовски знакомо, верно? И никаких тебе наблюдателей и компоновщиков.

Если присмотреться, разница в этих двух паттернах огромная. Проброс данных из контроллера в представление уже является грубым нарушением паттерна MVC. В классической реализации представление само забирало данные из модели напрямую или через контроллер, но чаще контроллер использовался для обработки действий пользователя над моделью по средством пользовательского интерфейса. То есть контроллер, повторюсь, это поведение представления. Подчиненный. Не хозяин.

К сожалению, такой подход не подходит для http протокола. Соединения с сервером обрываются, приложение прекращает работу, а в случае PHP и вовсе умирает до следующего запроса.

Модели 2, как и MVC, присущи все три паттерна, только их реализация несколько отличается от классической объектной реализации, описанной «бандой четырех».

Компоновщик остался, однако он реализуется через создание

комплексных файлов шаблонов, будь то JSP, PHP, ERB и так далее. А вот методов для работы с ним почти нет. Маловероятно, что в нынешних приложениях мы сможем работать с **V** как со связью *часть-целое*.

Стратегия тоже никуда не делась, но про нее все забыли, дав контроллерам новую роль — обработчики запросов. В большинстве нынешних приложений контроллер — это бог который решает, что, где и как делает. Контроллер вызывает модель, контроллер сохраняет модель, контроллер пишет в модель, контроллер генерирует представление, контроллер… ну вы поняли. Контроллер делает все. Из обычного поведения контроллер превратился в GodObject.

Наблюдатель. Это самое сложное. Все знают, как работает наблюдатель. Все знают, как сложно и иногда глупо реализовывать классического наблюдателя в контексте одного http запроса. Этот паттерн разрабатывался для актуализации данных на экране, и без веб-сокетов, лонг пулинга и прочей магии такое реализовать крайне сложно. В модели 2 наблюдателями являются и контроллер, и представление, только теперь считывание изменений состояния происходит после http запроса, а не все время.

Весьма сумбурно. Однако если именно так рассуждать об MVC и Модели 2, многие вещи сами становятся на свое место.

Далее я постараюсь описать обязанности каждого компонента. Я не прикован к конкретному языку, поэтому в тексте могут

встречаться сочетания типа «класс / структура», которые актуальны для разных в нынешнее время языков. Я хочу, чтобы статья была полезна не только ruby (railsl) / php разработчикам. Все написанное также актуально и для Golang с его структурами, python, C#, Java, и, конечно, JavaScript.

Статья писалась в несколько подходов, прошу не судить о дублирующейся информации.

Модели

Что можно сказать нового о моделях? О них все прекрасно сказано и не раз, например в этой замечательной статье отлично раскрыта сама суть модели. Модель — это состояние, это бизнес-логика, это единая точка входа в таблицу/монго документ/файл. Если нужно сохранить состояние между запросами пользователя — за это отвечает модель. Именно модель обязана иметь поля, если говорить о ней в контексте объекта. Модель не может быть stateless. Модель отвечает за валидацию и актуальность данных.

Если и существуют сущности, которыми можно описать любое приложение, без UI, естественно, то — это модели.

Представления и контроллеры

В классическом MVC представление является отдельным объектом, который занимается выводом информации на экран. Если говорить о CLI приложениях, то это какой-нибудь stdout.println(). В веб-приложениях используется модель 2 вместо

классического MVC, из чего следует, что представление — это просто файл с разметкой и встроенным кодом шаблонизатора. На первый взгляд кажется, что простой шаблон не имеет большого веса в системе. На самом же деле, именно представление определяет, какие действия доступны пользователю.

Представление — это часть двух паттернов: стратегии и компоновщика. Поведение (behavior) представления реализуется в контроллере. Получается, что контроллер является лишь частью некоего представления. И может быть заменен на другой контроллер, если нам понадобится другое поведение для конкретного представления.

На самом деле, тут очень много магии, которая проскальзывает сквозь пальцы «невнимательных путешественников».

Жадные и ленивые контроллеры

Из-за концепции «толстые модели, тонкие контроллеры» я заметил тенденцию упрощения действий контроллеров до одной строчки — return render('view'). В следствии этого большинство действий и выборка данных осуществляются прямо в представлении. Аргументируется такой подход тем, что чем меньше данных передается в представление, тем более независимыми становятся представления и контроллеры друг от друга. Упрощения доходят до абсурда, когда моделями начинают манипулировать прямо в файлах шаблона. Я не понимаю, почему все так боятся писать код в контроллерах. Если я передам в представление с десяток данных, это не означает, что мое

представление должно их все использовать. Представьте, что вы собрали какие-то вещи, засунули их в ящик и отправили по почте конкретному человеку. Вам эти вещи уже не нужны, и то, как с ними поступит получатель, абсолютно не важно. Он может выбрать себе пару-другую, а остальное выбросить. Я даже придумал для себя термин: «жадные и ленивые контроллеры».

В этой статье сказано, что action контроллера в терминологии фреймворка — это контроллер в терминологии MVC. Не смотря на то, что сама по себе статья потрясающая, я категорически не согласен с данным утверждением. Action — это действие пользователя над представлением. Это определяющая часть поведения. Если мы перепишем action, то поведение изменится, но мы внедримся в код готового класса, а я не люблю изменять код классов по чем зря. Если мы заменим контроллер, это так же приведет к изменению поведения представления, но уже без изменения существующих объектов стратегии. Возьмем, например, Angular.js. В нем явно прослеживается, где начинается контроллер, а где его методы. Любой метод контроллера так или иначе взаимодействует с представлением через директивы ng-click, ng-submit и так далее. Из чего можно сделать вывод: контроллер — это полноценный класс, а его методы (действия) — это действия представления, через которые пользователь взаимодействует с нашим приложением. Если мы заменим контроллер в директиве ng-controller, то поведение нашего представления поменяется в зависимости от реализации методов (действий) конкретного контроллера.

Второе утверждение, которое меня действительно пугает —

контроллеры это самые переиспользуемые части приложения. Что? Для меня контроллер — это исключительно одноразовый код. Действия (action's) могут быть переиспользованы, согласен. Особенно CRUD actions, в которых можно просто указать класс/структуру модели, с которой должно быть связано данное действие. Дублирование кода в контроллерах это нормальная практика, потому что они тонкие и лишь вызывают методы моделей, в которых и зашита вся логика. Если мне надо будет два раза подряд написать одно и то же (или почти одно и то же) действие для разных контроллеров, я не задумываясь сделаю это. Если мне придется вызвать одинаковые методы модели в разных контроллерах — я сделаю это. Контроллер можно выбросить и переписать заново, как, в принципе, и представление.

Модель выбросить сложно да и не нужно. Интерфейс модели, как правило, не изменяется во время разработки приложения или во время переноса модели из приложения в приложение, а только лишь дополняется. Контроллеры и представления почти в каждом приложении свои собственные. Попытайтесь перенести контроллер из одного фреймворка в другой, и вы поймете, о чем я говорю. А модель можно перенести из веб-приложения в мобильное или десктопное без (или почти без) изменения в коде прямо вместе с тестами преферансом и куртизанками.

Контроллер должен поддавать легкому рефакторингу. Возможно, вынос идентичных действий в отдельные объекты — это как раз то самое переиспользование контроллера, но оно не столь критично, чтобы уделять ему много внимания. Уж точно не 50 action классов на все случаи жизни.

Контроллер как поведение

Давайте возьмем наш любимый mp3 плеер (я имею ввиду приложение в смартфоне). Представление здесь — это кнопки перехода на соседние композиции, стоп, проигрывание и список воспроизведения. С кнопкой стоп и проигрыш все понятно, сомневаюсь, что их поведение будет меняться в зависимости от смены контроллера, а вот кнопки перехода на композиции могут так же являться кнопками перемотки вперед/назад текущей композиции. Итого, имея одно и то же представление, мы можем изменить его поведение просто заменой контроллера.

Можно представить, что на экране блокировки телефона во время проигрывания музыки кнопки перемотки меняют композицию, так как это удобно, а в самом приложении — перематывают композицию вперед/назад. Также, мы можем изменять контроллеры в рантайме через события. Событие удерживания пальца на кнопке перемотки активирует контроллер с перемоткой текущей композиции, быстрое касание — активирует контроллер перехода на различные композиции. Напоминает маршрутизатор, верно? В обоих контроллерах мы имеем дублирующуюся имплементацию действий play и stop, но это вполне приемлемо для паттерна стратегия. И оба контроллера используют **одно и то же представление**.

В MVC мы можем заменять не только контроллеры у представления, но и представления у одного контроллера. Хороший пример — это отображение файлов в любимом

файловом менеджере: плитка, список, таблица и т.д. Напоминает витрины товаров в интернет-магазинах, верно? На секунду представьте ваш файловый менеджер как Data Provider, View и Behavior(контроллер). И вы увидите, как MVC применяется вокруг нас в реальном мире.

Те же принципы работают и на медиа-центр с единым интерфейсом, но разными коллекциями данных (фото, музыка, видео). У каждой коллекции могут изменятся как представление, так и поведение, более того, мы можем смешать несколько behavior (контроллеров) для одного и того же представления. Например, в случае с видео подойдут поведения и для изображений (действие показать), и для музыки (действия стоп, воспроизведение, перемотка).

Виджеты

Виджеты это **V** со всеми вытекающими.

Чаще, чем хотелось бы, виджеты принимают за отдельный компонент системы. Мол, объект с подключаемым видом, почти как контроллер, значит и ответственность у него **почти как у контроллера**. И да, и нет. Часто **почти** воспринимают весьма буквально и в бедный виджет пихают всю работу с моделью, то есть всю работу **организатора** кроме фильтрации запросов. **Почти контроллер**, верно?

Виджет — бог

О вопросе ответственности и знаний виджета я задумался когда прочесывал GitHub в поисках удобного user-модуля для Yii. До этого мне казалось, что виджет — это настраиваемый шаблон, возможно, с подключаемым JavaScript, типа раскрывающегося меню, Pjax контейнера или GridView из того же Yii. Основываясь на этом мы получаем класс, который принимает в себя данные в качестве модели/коллекции моделей и отображает ее/их по своим внутренним правилам (convention) и настройкам (configuration). Или совсем не работает с моделью, а лишь добавляет оформление/поведение в пользовательский интерфейс.

Я был шокирован, когда увидел, что в виджетах люди создают, пишут, производят другие манипуляции с моделью напрямую. С одной стороны, когда мы имеем виджет LoginForm и модель LoginForm, то, казалось бы, логично использовать класс для валидации и логина пользователя в систему посредством манипуляций с моделью. Это же всего пара строк. Может где-то в другой Вселенной или в маленьком проекте так и есть, но в средних проектах или в целых модулях это создает многократное дублирование кода и смешивание ответственности разных по своей природе компонентов.

Один из примеров такого дизайна — это битрикс, где вся логика и представления заложены в виджетах, которые по неизвестной причине называются компонентами.

После подобных примеров в сети для различных фреймворков, я задумался — «вдруг это the right way». И тут же в голову закралось подозрение. Если бы мы засовывали всю логику в виджеты, то

контроллеры занимались бы только отображением конкретного представления. Но ведь это грубое нарушение паттерна. Если виджет является элементом **V**, то мы ни в коем случае не должны производить в нем манипуляции с моделью. Даже инстанцирование моделей рекомендуется делать в контроллерах, что уж говорить о сохранении.

С другой стороны, концепция drop in and it's work никак не стыкуется с обязательной конфигурацией виджета. К тому же, я очень люблю convention over configuration, которая, на мой взгляд, пока лучше всего представлена «из коробки» в Rails. В следствии чего я сделал следующие выводы: модель может инстанцироваться в виджете только в том случае, если инстанс не передается в виджет, и виджет является одноименным для конкретной модели, например, LoginForm или NewsLine (в ленте новостей правильно будет создать DataProvider на основе выборки). Но виджет ни в коем случае не должен уметь писать в модель или сохранять модель: это задача контроллера.

Все вышеописанное кажется весьма логичным, пока дело не доходит до обработки пользовательского ввода. Если нам необходимо после ввода данных в LoginForm авторизовать пользователя и остаться на странице, то как мы это сделаем без обработки прямо в виджете? Очевидно, что для этого нужен отдельный контроллер, но контроллер уводит пользователя со страницы, если в нем не указан явный редирект. В интернете, по большей части, используются три следующих сценария:

1. Пользователя перебрасывает на страницу логина, где показываются ошибки (не верный пароль) или происходит авторизация и отображение личного кабинета.
2. Пользователя перебрасывает на страницу логина, где показываются ошибки (не верный пароль) или происходит авторизация и редирект на предыдущую / главную страницу с (или без) отображением промежуточного представления (вспомним форумы на phpBB и им подобные).
3. Пользователь авторизуется посредством AJAX запроса, остается на странице. Все блоки, зависящие от авторизации (типа корзины) обновляются через какой нибудь Pjax.reload или простой перезагрузкой страницы.

В двух первых случаях виджет используется только как фасад над настоящим представлением и обработкой. Такой вариант имеет место быть. В последнем примере виджет неявно связан с контроллером, который можно заменить через конфигурацию.

В таком случае виджет представляет из себя минимальную логику — какой шаблон показать. Показать форму, если пользователь не авторизован; показать приветствие, если авторизован; никаких больше if else в шаблонах и никаких манипуляций в **V**.

Все три сценария легко реализуемы, позволяют избежать дублирования кода, но каждый из них сложнее манипуляций над моделью прямо в виджете, главное **не соблазняться простотой неверного подхода**.

> это практически не отличается от древней методики использования отдельных php-файлов для каждой «страницы» приложения.

Нужно больше виджетов!

Из-за большого количества готовых мелких решений фреймворки вызывают деградацию разработчиков. Я замечал за собой, как начинаю использовать какие-нибудь предустановленные виджеты и решать проблемы этих виджетов, чтобы они хоть как-то вписались в дизайн моего приложения, в то время как написать несколько своих строк кода на двух-трех языках было бы намного проще.

Переоцененный PJAX суется везде, даже там, где хватило бы обычного AJAX запроса. Свои JavaScript компоненты почти никто не пишет в попытках вложиться в рамки фреймворка. Я много раз видел на GitHub, как разработчики обсуждают JavaScript, необходимый для перезагрузки Pjax или другой работы связанной с ним, и что «надо бы засунуть по больше настроек в стандартный виджет фреймворка, дабы не отвлекаться на JS, это же неудобно!». А ведь создать свой виджет типа PjaxReloader занимает каких-то 10 минут вместе с тестированием. Честное слово, я больше времени потрачу на дискуссию.

```
$js = "jQuery(document).on('pjax:success', '#$id', function() {
jQuery.pjax.reload('$reloadSelector'); });";
$view->registerJs($js);
```

Виджет — это объект одной обязанности. По хорошему, все объекты в ООП должны отвечать за одну обязанность, но в реальном мире это не всегда так. Список комментариев к статье и форма добавления комментария — это разные виджеты. Один может включать в себя другой (опционально), но не смешиваться с ним. В конце концов, не забывайте, что **V** — это компоновщик. Вы можете объединять маленькие виджеты в один большой. Несколько больших в один огромный. И это не выйдет за рамки паттерна.

Итого

Модель хранит состояние и бизнес-логику. Знает все о своей предметной области, ей наплевать, какой контроллер ее вызывает и какое представление ее отображает. Модель можно инстанцировать вне приложения как отдельный и независимый класс. Модель легко поддается TDD, поэтому вся логика приложения должна храниться в моделях (напомню, что компонеты, сервисы, репозитории, и т.д. тоже являются моделями).

Представление знает **абсолютно все** о модели, кроме того, **как именно** она сохраняет свои состояния (база данных, файл, REST и т.д.). Представление знает, какие поля хранит модель, в каких полях хранятся отформатированные для представления свойства. Единственное, что представление не должно знать о модели — это методы записи, изменения и сохранения. Представление имеет право инстанцировать необходимые объекты и модели, а так же

собирать Data Provider's, но для избежания дублирования кода лучше выносить это в фабрики, хелперы и виджеты.

Контроллер ничего не знает ни о модели, ни о представлении. Контроллер знает, что за метод нужно вызвать у модели, но он не представляет что на самом деле происходит с моделью в данный момент. Контроллер просит сделать некие действия над моделью, но он не должен знать об ошибках (я имею ввиду рантайм ошибки и исключения), которые порождает модель, и ни в коем случае не должен их перехватывать и обрабатывать. Контроллер **может предложить (но это не обязательно)** модели отобразиться через определенное представление (ведь по сути, контроллер и есть часть одного или нескольких представлений) и передать в представление все, что он сгенерировал за время своего действия, но он не должен знать, **что именно** нужно представлению для отображения. Если он что-то не послал, представление должно адекватно на это отреагировать и просто не показать кусок себя или кинуть исключение, если речь идет о разрабатываемом приложении. Единственное исключение, которым может оперировать контроллер — NotFound 404! Остальные ошибки типа 403 — генерируются в фильтрах (request interceptors) или middleware.

Контроллер должен **по минимуму использоваться (или не использоваться вообще) для чтения данных**. Чтение могут организовать виджеты и им подобные объекты.

Виджеты не имеют никакого права манипулировать моделью и писать в базу данных, однако, они вполне могут прочитать данные

из базы/файла посредством абстракций, хелперов, фабрик, классов моделей и отдать их шаблону. Виджеты могут инстанцировать модель внутри себя, если инстанс модели не был передан в него из верхнего представления (паттерн компоновщик) или из контроллера. Виджеты могут собирать Data Provider's, так как это операция чтения.

В заключении хочу лишь привести цитату из моей любимой статьи на хабре. Я давал линк в самом начале статьи, в разделе про модели.

> Некоторые люди просто посмеются над всеми этими дураками, рассуждающими о необходимости хороших и независимых моделей предметной области (good independent domain models), и продолжат писать запутанный код. Пусть смеются. Ведь именно им придется поддерживать и тестировать свой бардак.

И да прибудет с вами композиция! Всем добра!

Комментарии (22)

Похожие публикации

.NET и паттерны проектирования

6

timyrik20 • 3 сентября 2013 в 00:00

Использование паттернов проектирования в javaScript: Порождающие паттерны

30

ИЗ ПЕСОЧНИЦЫ

yarkeev • 4 апреля 2013 в 16:15

Паттерны проектирования для iOS разработчиков. Observer, часть I

18

ИЗ ПЕСОЧНИЦЫ

avfonarev • 24 августа 2011 в 12:12

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов

20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить 85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16