Архитектура MVC

MVC это распространенная архитектура проектирования, и является аббревиатурой для 3-х слов: model, view, controller. И хотя MVC не является исключительно Rails особенностью, это, пожалуй, основной принцип (вместе с «convention over configuration» и «don’t repeat yourself», о которых поговорим в следующий раз), на котором все строится.  Если быть точным, то в Rails не MVC вовсе, а Model2 архитектура, разница между которыми хорошо описана [вот тут](http://andrzejonsoftware.blogspot.com/2011/09/rails-is-not-mvc.html). Но я, для простоты, все же буду опираться на MVC.

View — это графическое отображение данных пользователю. Пользователь не видит ничего кроме «вьюх». Все формы для ввода, всевозможные картинки, вывод статистики и так далее, все это является V в MVC. Views содержатся в части rails называемой ActionView, которая, в свою очередь, является частью ActionPack.

Model — это хранилище для данных, а также все методы для управления этими данными. В Rails, обычно, модели представляют собой таблицу в базе данных. А объекты модели — строку из соответствующей таблицы. В основном, вся логика приложения находится в моделях. За модели отвечает часть Rails, назваемая ActiveRecord.

Controller — посредник между view и model. Когда пользователь открывает какую-либо страницу или посылает форму на сервер, именно контроллер решает какую модель найти и какие данные с ней произвести. Также, получив результат от модели, выбирает какой отклик должен получить пользователь (или приложение, если это AJAX, например). Таким образом, контроллер управляет моделями в зависимости от данных от пользователя, и управляет отображением в зависимости от результата выполнения каких-либо действий модели. В Rails за контроллеры отвечает ActionController, который как и ActionView находится в ActionPack.

Теперь как же должно все это выглядеть? Не помню где прочитал, что «контроллер должен быть худым, а модели толстымии». Все верно, но не все так просто.

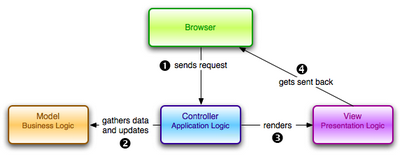
Что должны содержать views. Там седержится информация о графическом представлении информации. Логика приложения ограничивается максимум if-statment’ом или методом each на коллекции объектов, которые переданы из контроллера. Чуть более сложная логика (которая, опять же, непосредственно относится к отображению) выносится в так называемые view helpers. В моей практике обычно это какие-нибудь хитрые ссылки, названные link\_to\_active\_filters или нестандартное отображение валюты format\_currency. Никаких поисков объектов моделей, вызова каких-то классов напрямую, любой другой связи с базой данных здесь быть не должно. Тривиальная манипуляция объектами переданными из контроллера, обычно ограничивающаяся вызовом конкретно-необходимого метода объекта. Вызова таких методов может быть много, но никаких «цепочек», об этом должна позаботиться модель.

Контроллер должен получать параметры запроса пользователя, исходя из этой информации искать модель и вызывать на ней какие-то методы. После этого, в зависимости от результата, решать что возвращать пользователю. Хороший контроллер содержит в себе 2-3 действия в каждом action-method’e, в которых он ищет объект\объекты и может вызывать какой-то метод. И хотя поиск ведет контроллер, вся логика поиска ДОЛЖНА быть в модели. И модель предоставляет один интерфейсный метод для контроллера, который принимает необходимые параметры.

Модели содержат всю бизнесс-логику, поддерживаемую приложением. Здесь и запросы к базе данных, манипуляция с данными, всевозможные проверки и вспомогательные методы. Модель также должна предоставлять удобный интерфейс для views и controllers.

MVC – это паттерн архитектуры приложения, четко разделяющий три его компонента:

* **M**odel (далее **Модель**) является «сутью» приложения и отвечает за непосредственные алгоритмы, расчёты и тому подобное внутреннее устройство приложения. Также предоставляет линк к хранилищу данных.
* **V**iew (Представление, дальше **Вид**) предназначен для вывода данных, предоставленных Моделью. Это единственная часть MVC, которая непосредственно контактирует с пользователем.
* **C**ontroller (Поведение, далее **Контроллер**) получает данные от пользователя и передаёт их в Модель. Кроме того, он получает сообщения от Модели и передаёт их в Вид.



HTTP

**HTTP** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *HyperText Transfer Protocol* — «протокол передачи [гипертекста](http://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%BF%D0%B5%D1%80%D1%82%D0%B5%D0%BA%D1%81%D1%82)») — [протокол](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB) [прикладного уровня](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB%D1%8B_%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%BE%D0%B3%D0%BE_%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D1%8F) передачи данных (изначально — в виде гипертекстовых документов). Основой HTTP является [технология «клиент-сервер»](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%B5%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), то есть предполагается существование потребителей ([клиентов](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%B5%D0%BD%D1%82_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9%29)), которые инициируют соединение и посылают [запрос](http://ru.wikipedia.org/w/index.php?title=HTTP-%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81&action=edit&redlink=1), и поставщиков ([серверов](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_%28%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5%29)), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.

**Методы**

Метод HTTP ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *HTTP Method*) — последовательность из любых символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. Обратите внимание, что название метода чувствительно к регистру.

Каждый сервер обязан поддерживать как минимум методы GET и HEAD. Если сервер не распознал указанный клиентом метод, то он должен вернуть статус 501 (Not Implemented). Если серверу метод известен, но он неприменим к конкретному ресурсу, то возвращается сообщение с кодом 405 (Method Not Allowed). В обоих случаях серверу следует включить в сообщение ответа заголовок Allow со списком поддерживаемых методов.

Кроме методов GET и HEAD, часто применяется метод POST.

**OPTIONS**

Используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса. В ответ серверу следует включить заголовок [Allow](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%B7%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BA%D0%BE%D0%B2_HTTP#Allow) со списком поддерживаемых методов. Также в заголовке ответа может включаться информация о поддерживаемых расширениях.

Предполагается, что запрос клиента может содержать тело сообщения для указания интересующих его сведений. Формат тела и порядок работы с ним в настоящий момент не определён. Сервер пока должен его игнорировать. Аналогичная ситуация и с телом в ответе сервера.

Для того, чтобы узнать возможности всего сервера, клиент должен указать в URI звёздочку — «\*». Запросы «OPTIONS \* HTTP/1.1» могут также применяться для проверки работоспособности сервера (аналогично «пингованию») и тестирования на предмет поддержки сервером протокола HTTP версии 1.1.

Результат выполнения этого метода не кэшируется.

**GET**

Используется для запроса содержимого указанного ресурса. С помощью метода GET можно также начать какой-либо процесс. В этом случае в тело ответного сообщения следует включить информацию о ходе выполнения процесса.

Клиент может передавать параметры выполнения запроса в URI целевого ресурса после символа «?»:  
GET /path/resource?param1=value1&param2=value2 HTTP/1.1

Согласно стандарту HTTP, запросы типа GET считаются [идемпотентными](http://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BC%D0%BF%D0%BE%D1%82%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C)[[4]](http://ru.wikipedia.org/wiki/HTTP#cite_note-http-spec-methods-4)

Кроме обычного метода GET, различают ещё [условный GET](http://ru.wikipedia.org/wiki/HTTP#.D0.A3.D1.81.D0.BB.D0.BE.D0.B2.D0.BD.D1.8B.D0.B5_GET) и [частичный GET](http://ru.wikipedia.org/wiki/HTTP#.D0.A7.D0.B0.D1.81.D1.82.D0.B8.D1.87.D0.BD.D1.8B.D0.B5_GET). Условные запросы GET содержат заголовки If-Modified-Since, If-Match, If-Range и подобные. Частичные GET содержат в запросе Range. Порядок выполнения подобных запросов определён стандартами отдельно.

**HEAD**

Аналогичен методу GET, за исключением того, что в ответе сервера отсутствует тело. Запрос HEAD обычно применяется для извлечения [метаданных](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%B0%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5), проверки наличия ресурса (валидация URL) и чтобы узнать, не изменился ли он с момента последнего обращения.

Заголовки ответа могут кэшироваться. При несовпадении метаданных ресурса с соответствующей информацией в кэше копия ресурса помечается как устаревшая.

**POST**

Применяется для передачи пользовательских данных заданному ресурсу. Например, в [блогах](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%B3) посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего они передаются серверу методом POST и он помещает их на страницу. При этом передаваемые данные (в примере с блогами — текст комментария) включаются в тело запроса. Аналогично с помощью метода POST обычно загружаются файлы на сервер.

В отличие от метода GET, метод POST не считается идемпотентным[[4]](http://ru.wikipedia.org/wiki/HTTP#cite_note-http-spec-methods-4), то есть многократное повторение одних и тех же запросов POST может возвращать разные результаты (например, после каждой отправки комментария будет появляться одна копия этого комментария).

При результате выполнения 200 (Ok) в тело ответа следует включить сообщение об итоге выполнения запроса. Если был создан ресурс, то серверу следует вернуть ответ 201 (Created) с указанием URI нового ресурса в заголовке Location.

Сообщение ответа сервера на выполнение метода POST не кэшируется.

**PUT**

Применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существовало ресурса, то сервер создаёт его и возвращает статус 201 (Created). Если же был изменён ресурс, то сервер возвращает 200 (Ok) или 204 (No Content). Сервер не должен игнорировать некорректные заголовки Content-\*, передаваемые клиентом вместе с сообщением. Если какой-то из этих заголовков не может быть распознан или не допустим при текущих условиях, то необходимо вернуть код ошибки 501 (Not Implemented).

Фундаментальное различие методов POST и PUT заключается в понимании предназначений URI ресурсов. Метод POST предполагает, что по указанному URI будет производиться обработка передаваемого клиентом содержимого. Используя PUT, клиент предполагает, что загружаемое содержимое соответствует находящемуся по данному URI ресурсу.

Сообщения ответов сервера на метод PUT не кэшируются.

**PATCH**

Аналогично PUT, но применяется только к фрагменту ресурса.

**DELETE**

Удаляет указанный ресурс.

**TRACE**

Возвращает полученный запрос так, что клиент может увидеть, какую информацию промежуточные серверы добавляют или изменяют в запросе.

**LINK**

Устанавливает связь указанного ресурса с другими.

**UNLINK**

Убирает связь указанного ресурса с другими.

**CONNECT**

Преобразует соединение запроса в прозрачный TCP/IP туннель, обычно чтобы содействовать установлению защищенного SSL соединения через нешифрованный прокси.

### Ресурсный роутинг

Ресурсный роутинг позволяет быстро объявлять все обычные маршруты для заданного ресурсного контроллера. Вместо объявления отдельных маршрутов для экшнов index, show, new, edit, create, update и destroy, ресурсный маршрут объявляет их в единственной строке кода.

#### Ресурсы в вебе

Браузеры запрашивают страницы от Rails, выполняя запрос по URL, используя определенный метод HTTP, такой как GET, POST, PATCH, PUT и DELETE. Каждый метод - это запрос на выполнение операции с ресурсом. Ресурсный маршрут соединяет несколько родственных запросов с экшнами в одном контроллере.

Когда приложение на Rails получает входящий запрос для:

DELETE /photos/17

оно просит роутер соединить его с экшном контроллера. Если первый соответствующий маршрут такой:

resources :photos

Rails переведет этот запрос в метод destroy контроллера photos с { id: '17' } в params.

#### CRUD, методы и экшны

В Rails ресурсный маршрут предоставляет соединение между методами HTTP и URL к экшнам контроллера. По соглашению, каждый экшн также соединяется с определенной операцией CRUD в базе данных. Одна запись в файле роутинга, такая как:

resources :photos

создает семь различных маршрутов в вашем приложении, все соединенные с контроллером Photos:

|  |  |  |  |
| --- | --- | --- | --- |
| **Метод HTTP** | **Путь** | **Экшн** | **Использование** |
| GET | /photos | index | отображает список всех фото |
| GET | /photos/new | new | возвращает форму HTML для создания нового фото |
| POST | /photos | create | создает новое фото |
| GET | /photos/:id | show | отображает определенное фото |
| GET | /photos/:id/edit | edit | возвращает форму HTML для редактирования фото |
| PATCH/PUT | /photos/:id | update | обновляет определенное фото |
| DELETE | /photos/:id | destroy | удаляет определенное фото |

Поскольку роутер использует как метод HTTP, так и URL, для сопоставления с входящими запросами, четыре URL соединяют с семью различными экшнами.

Маршруты Rails сравниваются в том порядке, в котором они определены, поэтому, если имеется resources :photos до get 'photos/poll' маршрут для экшна show в строке resources совпадет до строки get. Чтобы это исправить, переместите строку get **над** строкой resources, чтобы она сравнивалась первой.

#### Путь и хелперы URL

Создание ресурсного маршрута также сделает доступными множество хелперов в контроллере вашего приложения. В случае с resources :photos:

* photos\_path возвращает /photos
* new\_photo\_path возвращает /photos/new
* edit\_photo\_path(:id) возвращает /photos/:id/edit (например, edit\_photo\_path(10) возвращает /photos/10/edit)
* photo\_path(:id) возвращает /photos/:id (например, photo\_path(10) возвращает /photos/10)

Каждый из этих хелперов имеет соответствующий хелпер \_url (такой как photos\_url), который возвращает тот же путь с добавленными текущими хостом, портом и префиксом пути.

#### Определение нескольких ресурсов одновременно

Если необходимо создать маршруты для более чем одного ресурса, можете сократить ввод, определив их в одном вызове resources:

resources :photos, :books, :videos

Это приведет к такому же результату, как и:

resources :photos

resources :books

resources :videos

Rails Generate. Scaffolding

##### Action Pack

Action Pack это отдельный гем, содержащий Action Controller, Action View и Action Dispatch. Буквы “VC” в аббревиатуре “MVC”.

**Action Controller**

В веб-приложении Контроллер регулирует поток логики приложения. Он находится на границе программы, перехватывая все запросы, на основе которых он изменяет какой-то объект Модели и вызывает Вид, чтобы отобразить обновленные данные. В RoR Action Controller является Контроллером, вот его основные функции:

* **Поддержка сессий**. Сессия – это период времени, проведенный пользователем на сайте. Его можно отследить с помощью **cookie** или **объекта сессии**. Cookie – небольшой файл, он не может содержать объекты, в отличие от объекта сессии.
* **Фильтрация**. Бывают ситуации, когда необходимо вызвать определенный код, перед тем как исполнять логику Контроллера или после него, например, аутентификация пользователей, логирование событий, предоставление персонального ответа. Помогают в таких случаях фильтры, предоставляемые Action Controller. Существуют три основных фильтра: before, after и around. О них – позже.
* **Кэширование**. Кэширование – это процесс, при котором наиболее запрашиваемый контент сохраняется в **кэше**, чтобы не было необходимости запрашивать его вновь и вновь.

##### Action Dispatch

Action Dispatch управляет маршрутизацией веб запросов и рассылкой их так, как Вы желаете, или к Вашему приложению, или к любому другому приложению Rack.

**Action View**

Вид включает в себя логику, необходимую для вывода данных Модели. Роль Вида в RoR играет Action View. Наилее часто используемые функции Action View:

* **Шаблоны** (Templates). Шаблоны – это файлы, содержащие заполнители (placeholders), которые буду заменены на контент. Шаблоны могут содержать HTML-код и код Ruby, встраиваемый в HTML с использованием синтакса встроенного (embedded) Ruby (**ERb**).
* **Помощники** (helper, далее хелпер) **форм и форматирования**. Хелперы форм позволяют создавать такие элементы страниц, как чекбоксы, списки, используя **готовые методы**. В свою очередь хелперы форматирования позволяют форматировать данные необходимым нам способом, методы существуют для дат, валют и строк.
* **Макет**. Макеты (layouts) определяют, как контент будет расположен на странице. Динамически создаваемая страница может содержать вложение из нескольких страниц, даже без использования таблиц и фрэймов, используя **API** Макета.

##### Action Mailer

Action Mailer это фреймворк для встроенных служб e-mail. Action Mailer можно использовать, чтобы получать и обрабатывать входящую электронную почту, или чтобы рассылать электронные письма, основанные на гибких шаблонах.

##### Active Model

Active Model предоставляет определенный интерфейс между службами гема Action Pack и гемами Object Relationship Mapping, такими как Active Record. Active Model позволяет Rails использовать другие фреймворки ORM вместо Active Record, если так нужно Вашему приложению.

**Active Record**

Active Record – это Модель в RoR. Модель хранит данные и предоставляет базу для работы с данными. Кроме этого Active Record также является ORM фрэймворком. ORM значит Object-relational mapping (Объектно-реляционная проекция). Собственно Active Record делает следующие вещи:

* **Проекция таблицы на класс**. Каждая таблица проецируется на один или несколько классов по принципу **convention over configuration** (соглашение выше конфигурации). Одно из таких соглашений – имя таблицы должно быть во множественном числе, а название класса – в единственном. **Атрибуты таблицы** налету проецируются в **атрибуты экземпляра Руби**. После того, как все проекции сделаны, каждый **объект** ORM класса представляет определенную **строку** таблицы, с которой класс был спроецирован.
* **Соединение с БД**. Вы можете подключиться к базе данных, используя **API**, предоставляемый Active Record, который создает необходимый вам запрос непосредственно в движок БД при помощи адаптеров. У Active Record есть адаптеры для MySQL, Postgres, MS SQLServer, DB2, и SQLite. Необходимо лишь записать параметры доступа к БД в файле database.yml.
* **Операции CRUD**. Это операции **c**reate (создание), **r**etrieve (получение), **u**pdate (обновление) и **d**elete (удаление) над таблицей. Так как Active Record – это ORM фрэймворк, вы всегда работаете с **объектами**. Чтобы создать новую строку таблицы, вы создаете новый объект класса и заполняете его переменные экземпляра значениями. Стоит заметить, что все это Active Record делает за вас.
* **Проверка данных**. Проверка данных перед помещением их в таблицу – это первый шаг в безопасности вашего проекта. Active Record предоставляет проверку **Модели**. Данные могут быть проверены автоматически с помощью множества готовых **методов**, которые, в случае необходимости, можно переписать под собственные нужды.

##### Active Resource

Active Resource представляет фреймворк для управления соединением между бизнес-объектами и веб-сервисами на основе RESTful. Он реализует способ привязки веб-ресурсов к локальным объектам с семантикой CRUD.

##### Active Support

Active Support это большая коллекция полезных классов и расширений стандартных библиотек Ruby, которые могут быть использованы в Rails, как в ядре, так и в Вашем приложении.