Подробное введение в Backbone.Marionette

Backbone.js все быстрее становится самой популярной платформой для создания модульных JavaScript приложений на стороне клиента. В основном это просто благодаря отсутствию больших препятствий в начале работы; начать работать с ним супер-просто. Однако, в отличие от Ember.js, из-за такой минимальности Backbone разработчику приходится многое додумывать самому.

Итак, как только вы хотите расширить приложение, все становится не таким уж и простым. Backbone.Marionette был создан для того, чтобы облегчить все **трудности, связанные с Backbone** **разработкой.**  Согласно его автору, Backbone.Marionette «заставляет танцевать ваши Backbone.js приложения с помощью составной архитектуры приложения!».

Эта «композитная» архитектура в основном относится к многочисленным типам views которые призваны помочь в управлении subview. Сегодня мы не будем обсуждать views (хотя мы коснемся regions, которые являются малой частью управления subview, предоставляемые Marionette), но вы можете найти документацию по этому проекту на [GitHub repository](https://github.com/marionettejs/backbone.marionette). Он предлагает множество компонентов, которые расширяют Backbone и позволяют писать меньше шаблонов, но делать больше вещей без каких либо хлопот, особенно когда дело доходит до views.

**Объект Центрального приложения**

В основном, когда кто-то создает Backbone приложение, он делает центральный объект, к которому все привязано, что часто называют App или Application. Backbone не предлагает ничего, из чего можно создать этот объект, поэтому **большинство людей просто создают основной роутер** и называют его app объектом. Хотя и здорово, что люди присоединяют все к центральному объекту так, что глобальное пространство имен не запутано, роутер не был спроектирован для того, чтобы справиться с этим заданием.

У Дерика Бейли, создателя Marionette, была идея получше. Он создал «класс», из которого вы можете реализовать объект, специально разработанный для того чтобы справляться с обязанностями основного корневого объекта всего приложения. Вы начинаете создание нового приложения с var App = new Backbone.Marionette.Application(), а потом, когда все настроено, вы запускаете приложение с App.start(options). В ближайшее время мы будем обсуждать аргумент options. А сейчас просто помните, что это необязательно.

**инициализаторы**

Одной из классных вещей касательно Marionette  Application являются инициализаторы. Если у вас модульный код, то несколько его частей должны быть инициализированы при запуске приложения. Вместо заполнения main.js файла большим количеством кода для инициализации всех этих объектов, вы можете просто настроить модули таким образом, чтобы инициализация происходила внутри кода самого модуля. Вы можете сделать это при помощи addInitializer. Например:

var SomeModule = function(o){

// Constructor for SomeModule

};

App.addInitializer(function(options) {

App.someModule = new SomeModule(options);

});

Все инициализаторы добавленные таким образом, будет выполняться при вызове App.start. Обратите внимание, что аргумент options передается к инициализатору. Это тот же самый объект, передаваемый в тот момент, когда вы вызываете App.start(options).Это отличная возможность разрешить передать конфигурацию так, чтобы каждый модуль мог ее использовать.

Несколько событий также вызываются при работе через эти инициализаторы:

* initialize:before вызывается как раз перед тем, как запускаются инициализаторы.
* initialize:after вызывается как раз после того как инициализаторы закончили свою работу.
* start вызывается после  initialize:after.

Вы можете прослушивать эти события и иметь еще больший контроль. Прослушивайте эти события таким образом:

App.on('initialize:before', function(options) {

options.anotherThing = true; // Add more data to your options

});

App.on('initialize:after', function(options) {

console.log('Initialization Finished');

});

App.on('start', function(options) {

Backbone.history.start(); // Great time to do this

});

Все довольно просто и это также даст вам максимальную гибкость в начале создания ваших приложений.

**Агрегатор событий**

Объект Application дает еще больше возможностей для декомпозиции Backbone приложения путем использования агрегатора событий. Некоторое время назад я писал о [масштабируемых JavaScript приложениях](http://www.joezimjs.com/javascript/scalable-javascript-applications/) в которых я упомянул, что модули системы должны быть в полном неведении друг о друге, и что единственный способ, через который они должны иметь возможность общаться друг с другом это **события по всему приложению**. Таким образом, каждый беспокоящийся модуль может прослушивать изменения и события, в которых они нуждаются так, что они могут на них реагировать и ничего в системе даже не поймет, что они вообще существуют.

Marionette делает этот вид декомпозиции более возможным при помощи агрегатора событий, который автоматически прилагается к объекту приложения. Хотя это всего лишь один из механизмов, который я описывал в той статье, это только начало, и может быть очень полезным даже в меньших приложениях.

Агрегатор событий можно найти через свойство в приложении под названием vent. Соответственно вы можете подписаться и отписаться от событий просто через on и off методы, (или bind и unbind). Эти функции могут показаться знакомыми, и это потому, что агрегатор событий это просто расширение [Backbone](#Events) объекта событий. В действительности, единственное, что здесь есть нового и за что вам нужно беспокоиться так это то, что мы используем события на объекте, которые должны быть доступны везде в пределах вашего приложения так, чтобы каждая часть вашего приложения могла общаться через него. Агрегатор событий также доступен как отельный модуль, так что вы можете добавлять его в любой объект, который хотите, также как и Backbone Event.

**REGION**

Region это еще один модуль Marionette, который позволяет легко привязать views к различным region HTML документа. Я не буду вдаваться в подробности о том, как работает region — это не сегодняшняя тема — но я кратко расскажу и объясню, как использовать его с Application.

Region это объект — обычно создается вместе с new Backbone.Marionette.Region({ el: 'selector'}) — который управляет областью, где вы добавляете view. Вы добавите view и автоматически сгенерируете его с помощью show. Потом вы можете закрыть этот view (то есть он будет удален из DOM, и если вы используете один из Marionette view, то отмените любые привязки, сделанные view) и сгенерируйте другой view, просто вызвав show еще раз, или вы можете просто закрыть view, вызвав close. Region способен на большее, но тот факт, что он справляются с генерацией и закрытием view одним вызовом функции, делает их чрезвычайно полезными. Вот пример кода для тех, кто понимает в нем больше, чем в английском языке:

// Create a region. It will control what's in the #container element.

var region = new Backbone.Marionette.Region({

el: "#container"

});  
  
// Add a view to the region. It will automatically render immediately.

region.show(new MyView());  
  
// Close out the view that's currently there and render a different view.

region.show(new MyOtherView());  
  
// Close out the view and display nothing in #container.

region.close();

Если вы непосредственно хотите иметь Region в объекте вашего приложения (например, App.someRegion), есть простой способ как можно быстро его добавить: addRegions. Есть три способа использования addRegions. В любом случае, вы должны будете передать объект, чье имя свойства будет добавлено в приложение как region, но значение каждого из них могут быть различными в зависимости от того, как вы хотите реализовать это.

**Селектор**

Просто добавьте селектор и будет создан стандартный region, который использует этот селектор, как его  el  свойство.

App.addRegions({

container: "#container",

footer: "#footer"

});  
  
// This is equivalent to

App.container = new Backbone.Marionette.Region({el:"#container"});

App.footer = new Backbone.Marionette.Region({el:"#footer"});

**Кастомный тип region**

Вы можете расширить Region  для создания своих собственных regionтипов. Если вы хотите использовать свой собственный region тип, вы можете воспользоваться синтаксисом, приведенном ниже. Обратите внимание, этот  el  синтаксис уже должен быть определен в вашем region типе.

var ContainerRegion = Backbone.Marionette.Region.extend({

el: "#container", // Must be defined for this syntax

// Whatever other custom stuff you want

});

var FooterRegion = Backbone.Marionette.Region.extend({

el: "#footer", // Must be defined for this syntax

// Whatever other custom stuff you want

});

// Use these new Region types on App.

App.addRegions({

container: ContainerRegion,

footer: FooterRegion

});

// This is equivalent to:

App.container = new ContainerRegion();

App.footer = new FooterRegion();

**Кастомный тип region с селектором**

Если вы не определяете  el — или вы хотите перезаписать его — то вы можете использовать этот синтаксис в вашем кастомном region типе:

var ContainerRegion = Backbone.Marionette.Region.extend({});

var FooterRegion = Backbone.Marionette.Region.extend({});

// Use these new Region types on App.

App.addRegions({

container: {

regionType: ContainerRegion,

selector: "#container"

},

footer: {

regionType: FooterRegion,

selector: "#footer"

}

});

// This is equivalent to:

App.container = new ContainerRegion({el:"#container"});

App.footer = new FooterRegion({el:"#footer"});

Как вы видите, добавить region по всему приложению чрезвычайно просто (особенно, если вы используете обычный Region тип), и он добавляет много полезного функционала.

**Выводы**

Как вы уже убедились, Marionette добавляет множество отличных функций для упрощения Backbone разработки, и мы рассмотрели лишь один из большого количества модулей, которые он предоставляет (также, мы затронули несколько других модулей, которыми пользуется само Application, но есть еще много чего предстоит узнать). Я надеюсь, что это будет немного привлекать Backbone программистов и вызывать интерес на прочтение остальных статей этой серии, когда я расскажу больше о модулях.

* 1. A Thorough Introduction To Backbone.Marionette

Backbone.js is quickly becoming the most popular framework for building modular client-side JavaScript applications. This is largely due to its low barrier to entry; getting started with it is super-simple. However, unlike Ember.js, Backbone, being so minimal, also leaves a lot up to the developer to figure out.

So, once you start getting into more advanced applications, it’s no longer so simple. Backbone.Marionette was created to alleviate a lot of the growing pains of Backbone development. Backbone.Marionette “make[s] your Backbone.js apps dance with a composite application architecture!,” according to its author.

This “composite” architecture refers mainly to the numerous view types that have been provided to help with subview management. We won’t be discussing those views today (although we will touch on regions, which are a small part of the subview management that Marionette offers), but you can find documentation for this project in the [GitHub repository](https://github.com/marionettejs/backbone.marionette). It offers numerous components that extend Backbone and that enable you to write less boilerplate and do more stuff with little to no hassle, especially when it comes to views.

* + 1. The Central Application Object

Most of the time, when someone creates a Backbone application, they make a central object that everything is attached to, which is often referenced as App or Application. Backbone doesn’t offer anything to make this object from, so most people just create a main router and make that the app object. While it’s great that people are attaching things to a central object so that the global namespace isn’t so convoluted, the router was not designed to handle this task.

Derick Bailey, the creator of Marionette, had a better idea. He created a “class” that you could instantiate an object from that is specifically designed to handle the responsibilities of being the go-to root object of the entire application. You create a new application with var App = new Backbone.Marionette.Application(), and then, when everything is set, you start the application with App.start(options). I’ll discuss the optionsargument soon. For now, just remember that it’s optional.

* + 1. Initializers

One of the coolest things about Marionette’s Application is the initializers. When your code is modular, several pieces will need to be initialized when the application starts. Rather than filling a main.js file with a load of code to initialize all of these objects, you can just set the modules up for initialization within the code for the module. You do this using addInitializer. For example:

var SomeModule = function(o){

// Constructor for SomeModule

};  
  
App.addInitializer(function(options) {

App.someModule = new SomeModule(options);

});

All of the initializers added this way will be run when App.start is called. Notice the options argument being passed into the initializer. This is the very same object that is passed in when you call App.start(options). This is great for allowing a configuration to be passed in so that every module can use it.

A few events are also fired when running through these initializers:

* initialize:before Fires just before the initializers are run.
* initialize:after Fires just after the initializers have all finished.
* start Fires after initialize:after.

You can listen for these events and exert even more control. Listen for these events like this:

App.on('initialize:before', function(options) {

options.anotherThing = true; // Add more data to your options

});

App.on('initialize:after', function(options) {

console.log('Initialization Finished');

});

App.on('start', function(options) {

Backbone.history.start(); // Great time to do this

});

Pretty simple, and it gives you a ton of flexibility in how you start up your applications.

* + 1. Event Aggregator

The Application object brings even more possibilities for decoupling a Backbone application through the use of an event aggregator. A while back I wrote a post about [scalable JavaScript applications](http://www.joezimjs.com/javascript/scalable-javascript-applications/), in which I mentioned that modules of a system should be completely ignorant of one another, and that the only way they should be able to communicate with each other is through application-wide events. This way, every module that cares can listen for the changes and events they need to so that they can react to them without anything else in the system even realizing it exists.

Marionette makes this kind of decoupling largely possible via the event aggregator that is automatically attached to the application object. While this is only one of the mechanisms that I wrote about in that article, it is a start and can be very useful in even smaller applications.

The event aggregator is available through a property in the application called vent. You can subscribe and unsubscribe to events simply via the on and off methods, respectively (or bind and unbind, if you prefer). These functions might sound familiar, and that’s because the event aggregator is simply an extension of[Backbone’s](#Events) Event object. Really, the only thing new here that you need to worry about is that we’re using the events on an object that should be accessible everywhere within your app, so that every piece of your application can communicate through it. The event aggregator is available as its own module too, so you can add it to any object you want, just like Backbone’s Event.

* + 1. Regions

Region is another module for Marionette that enables you to easily attach views to different regions of an HTML document. I won’t go into detail about how regions work here — that’s a topic for another day — but I’ll cover it briefly and explain how to use them with Application.

A region is an object — usually created with new Backbone.Marionette.Region({ el: 'selector'}) — that manages an area where you attach a view. You would add a view and automatically render it by using show. You can then close out that view (meaning it will remove it from the DOM and, if you’re using one of the Marionette views, undo any bindings made by the view) and render a different view simply by calling showagain, or you can just close the view by calling close. Regions can do more than that, but the fact that they handle the rendering and closing for you with a single function call makes them extremely useful. Here’s a code sample for those who speak in code rather than English:

// Create a region. It will control what's in the #container element.

var region = new Backbone.Marionette.Region({

el: "#container"

});  
  
// Add a view to the region. It will automatically render immediately.

region.show(new MyView());  
  
// Close out the view that's currently there and render a different view.

region.show(new MyOtherView());  
  
// Close out the view and display nothing in #container.

region.close();

If you want a Region directly on your application object (e.g. App.someRegion), there’s a simple way to add one quickly: addRegions. There are three ways to use addRegions. In every case, you would pass in an object whose property names will be added to the application as regions, but the value of each of these may be different depending on which way you wish to accomplish this.

* + - 1. Selector

Simply supply a selector, and a standard region will be created that uses that selector as its el property.

App.addRegions({

container: "#container",

footer: "#footer"

});  
  
// This is equivalent to

App.container = new Backbone.Marionette.Region({el:"#container"});

App.footer = new Backbone.Marionette.Region({el:"#footer"});

* + - 1. Custom Region Type

You can extend Region to create your own types of regions. If you want to use your own type of region, you can use the syntax below. Note that, with this syntax, el must already be defined within your region type.

var ContainerRegion = Backbone.Marionette.Region.extend({

el: "#container", // Must be defined for this syntax

// Whatever other custom stuff you want

});  
  
var FooterRegion = Backbone.Marionette.Region.extend({

el: "#footer", // Must be defined for this syntax

// Whatever other custom stuff you want

});  
  
// Use these new Region types on App.

App.addRegions({

container: ContainerRegion,

footer: FooterRegion

});  
  
// This is equivalent to:

App.container = new ContainerRegion();

App.footer = new FooterRegion();

* + - 1. Custom Region Type with Selector

If you don’t define el — or you want to override it — in your custom region type, then you can use this syntax:

var ContainerRegion = Backbone.Marionette.Region.extend({});  
  
var FooterRegion = Backbone.Marionette.Region.extend({});  
  
// Use these new Region types on App.

App.addRegions({

container: {

regionType: ContainerRegion,

selector: "#container"

},

footer: {

regionType: FooterRegion,

selector: "#footer"

}

});  
  
// This is equivalent to:

App.container = new ContainerRegion({el:"#container"});

App.footer = new FooterRegion({el:"#footer"});

As you can see, adding application-wide regions is dead simple (especially if you’re using the normal Regiontype), and they add a lot of useful functionality.

* + 1. Conclusion

As you can already see, Marionette adds a ton of great features to make Backbone development simpler, and we’ve covered only one of many modules that it provides (plus, we’ve touched on a couple of other modules that Application itself uses, but there’s plenty more to learn about those). I hope this will entice Backbone programmers a bit and make you eager to read the rest of this series, when I’ll cover more of the modules.

Credits of image on start page:[*Dmitry Baranovskiy*](http://www.flickr.com/photos/dmitry-baranovskiy/2378867408/).

(al)

**Тщательное введение в Backbone.Marionette**

Backbone.js быстро становится самой популярной платформой для создания модульных приложений на стороне клиента JavaScript.Во многом это связано с его низким барьером для входа; Приступая к работе с это супер-простой.Однако, в отличие Ember.js, Магистральная, будучи настолько минимальны, также оставляет желать до разработчиков, чтобы понять out.

Итак, когда вы начнете получать в более сложных приложений, это уже не так просто.Backbone.Marionette была создана, чтобы облегчить многие **болезни роста магистральной development**.Backbone.Marionette "делать [с] Backbone.js ваших программах танец с композитной архитектуры приложения,!" В соответствии с его author.

Это «Композит» архитектуры относится в основном к многочисленным типам зрения, которые были предоставлены, чтобы помочь с руководством подвид.Мы не будем обсуждать эти взгляды сегодня (хотя мы будем касаться регионов, которые являются небольшой частью управления подвид, что марионеток предложения), но вы можете найти документацию по этому проекту [GitHub repository](https://github.com/marionettejs/backbone.marionette).Он предлагает многочисленные компоненты, расширяющие Магистральные и что позволит вам писать меньше шаблона и сделать больше вещей с практически никаких хлопот, особенно когда дело доходит до views.

**ЦЕНТРАЛЬНАЯ ПРИЛОЖЕНИЙ OBJECT**

Большую часть времени, когда кто-то создает Применение в магистральной сети, они делают центрального объекта, что все связаны, которая часто упоминается как App или Application.Магистральные не предлагает ничего, чтобы сделать этот объект, так **большинство людей просто создать основной router** и делают, что приложение объекта.Хотя это здорово, что люди придают вещи в центральный объект, так что глобальное пространство имен не так уж запутанным, маршрутизатор не была предназначена для обработки этого task.

Дерик Бейли, создателем марионеток, была идея получше.Он создал «класс», что можно создать экземпляр объекта с, которая специально предназначена для обработки обязанности бытия идти к корневой объект всего приложения.Вы создаете новое приложение с var App = new Backbone.Marionette.Application(), а потом, когда все настроено, вы запустите приложение с App.start(options).Я буду обсуждать options аргумент в ближайшее время.А сейчас просто помните, что это optional.

**INITIALIZERS**

Одна из лучших вещей о марионеток в Application является инициализаторы.Когда ваш код является модульной, несколько кусков должны быть инициализированы при запуске приложения.Вместо заполнения main.js файл с нагрузкой код для инициализации всех этих объектов, вы можете просто установить модули для инициализации в коде для модуля.Это делается с помощью addInitializer.Например:

var SomeModule = function(o){

// Constructor for SomeModule

};

App.addInitializer(function(options) {

App.someModule = new SomeModule(options);

});

Все инициализаторы добавлены таким образом, будет выполняться при App.start называется.Обратите внимание на options аргумент передается в инициализаторе.Это тот же самый объект, передаваемый в том, когда вы звоните App.start(options).Это отличная возможность дляпозволяет конфигурация будет принят в так что каждый модуль может использовать it.

Несколько событий также уволены при работе через эти инициализаторы:

* initialize:beforeПожары перед инициализаторы run.
* initialize:afterПожары сразу после инициализации все finished.
* startПожары после initialize:after.

Вы можете слушать эти события и оказывать еще больший контроль.Прислушайтесь к этим событиям так:

App.on('initialize:before', function(options) {

options.anotherThing = true; // Add more data to your options

});

App.on('initialize:after', function(options) {

console.log('Initialization Finished');

});

App.on('start', function(options) {

Backbone.history.start(); // Great time to do this

});

Довольно просто, и это дает вам тонну гибкость в том, как начать свой applications.

**СОБЫТИЯ AGGREGATOR**

Application объект приносит еще больше возможностей для развязки Применение в магистральной сети с использованием событий агрегатор.Некоторое время назад я написал пост о [масштабируемых JavaScript applications](http://www.joezimjs.com/javascript/scalable-javascript-applications/), в которых я упомянул, что модули системы должны быть в полном неведении друг от друга, и что единственный способ, которым они должны иметь возможность общаться друг с другом является **черезна уровне приложения events**.Таким образом, каждый модуль, который заботится можете слушать изменений и событий, они должны так, чтобы они могли реагировать на них без ничего в системе даже не осознавая этого exists.

Марионеток делает этот вид развязки в значительной степени возможно через агрегатор событие, которое автоматически прилагаются к заявлению объекта.Хотя это всего лишь один из механизмов, который я написал в этой статье, это только начало, и может быть очень полезна в еще меньшей applications.

События агрегатор доступны через свойство в приложение под названием vent.Вы можете подписаться и отписаться на события просто через on и off методы, соответственно (или bind и unbind, если вы предпочитаете).Эти функции может показаться знакомым, и это потому, что события агрегатор это просто расширение[Магистральные это](#Events)Event object.Действительно, единственное, что здесь нового, что вам нужно беспокоиться о том, что мы используем события на объекте, который должен быть доступен везде в пределах вашего приложения, так что каждая часть вашего приложения могут общаться через него.Мероприятие агрегатор доступны как свой собственный модуль, так что вы можете добавить его в любой объект, который, как Магистральная в Event.

**REGIONS**

Region другой модуль для марионеток, которая позволяет легко прикрепить просмотры в различные регионы HTML документа.Я не буду вдаваться в подробности о том, как регионы здесь работают — это тема для другой день — но я расскажу это кратко и объяснить, как использовать их с Application.

Регион является объектом — обычно создаются с new Backbone.Marionette.Region({ el: 'selector'}) — который управляет областью, где вы приложите зрения.Вы бы добавить вида и автоматически делают это с помощью show.Вы можете закрыть эту точку зрения (то есть он будет удалить его из DOM, и, если вы используете одну из марионеток взгляды, отменить любые привязки сделаны зрения) и оказываютиной точки зрения, просто вызвав show раз, или вы можете просто закрыть окно, вызвав close.Регионы могут сделать больше, чем это, но тот факт, что они обращаются с рендерингом и закрытия для вас с одного вызова функции делает их чрезвычайно полезными.Вот пример кода для тех, кто говорит в коде, а не на английском:

// Create a region. It will control what's in the #container element.

var region = new Backbone.Marionette.Region({

el: "#container"

});

// Add a view to the region. It will automatically render immediately.

region.show(new MyView());

// Close out the view that's currently there and render a different view.

region.show(new MyOtherView());

// Close out the view and display nothing in #container.

region.close();

Если вы хотите Region непосредственно на объект приложения (например, App.someRegion), есть простой способ добавить один быстро: addRegions.Есть три способа использования addRegions.В любом случае, вы должны перейти в собственность объекта, имена будут добавлены к применению в качестве регионов, но значение каждого из них могут быть различными в зависимости от того, как вы хотите достичь this.

**Selector**

Simply поставлять селектор, и стандартный регионе будет создан, который использует этот селектор, как ее el property.

App.addRegions({

container: "#container",

footer: "#footer"

});

// This is equivalent to

App.container = new Backbone.Marionette.Region({el:"#container"});

App.footer = new Backbone.Marionette.Region({el:"#footer"});

**Custom области Type**

Вы можете расширить Region создать свои собственные типы регионов.Если вы хотите использовать свой собственный тип региона, вы можете использовать синтаксис ниже.Отметим, что с этим синтаксисом, el должны быть уже определены в вашем регионе type.

var ContainerRegion = Backbone.Marionette.Region.extend({

el: "#container", // Must be defined for this syntax

// Whatever other custom stuff you want

});

var FooterRegion = Backbone.Marionette.Region.extend({

el: "#footer", // Must be defined for this syntax

// Whatever other custom stuff you want

});

// Use these new Region types on App.

App.addRegions({

container: ContainerRegion,

footer: FooterRegion

});

// This is equivalent to:

App.container = new ContainerRegion();

App.footer = new FooterRegion();

**Custom Type области с Selector**

Если вы не определите el — или вы хотите переопределить его — В пользовательского типа регионе, то вы можете использовать следующий синтаксис:

var ContainerRegion = Backbone.Marionette.Region.extend({});

var FooterRegion = Backbone.Marionette.Region.extend({});

// Use these new Region types on App.

App.addRegions({

container: {

regionType: ContainerRegion,

selector: "#container"

},

footer: {

regionType: FooterRegion,

selector: "#footer"

}

});

// This is equivalent to:

App.container = new ContainerRegion({el:"#container"});

App.footer = new FooterRegion({el:"#footer"});

Как вы можете видеть, добавив уровне приложения регионах мертвых простой (особенно, если вы используете нормальный Region типа), и они добавляют много полезной functionality.

**CONCLUSION**

Как вы уже можете видеть, Marionette добавляет тонну большие возможности, чтобы сделать развитие магистральной проще, и мы рассмотрели лишь один из многих модулей, которые он предоставляет (плюс, мы затронули несколько других модулей, которые Application использует сам, но есть еще много узнать о них).Я надеюсь, что это будут соблазнять Магистральные программисты немного и сделать вас хотят читать остальной части этой серии, когда я расскажу более modules.

*Кредиты изображение на стартовой странице.*[*Дмитрий Baranovskiy*](http://www.flickr.com/photos/dmitry-baranovskiy/2378867408/)

*(AL)*

Источник:[*http://coding.smashingmagazine.com/2013/02/11/introduction-backbone-marionette/*](http://coding.smashingmagazine.com/2013/02/11/introduction-backbone-marionette/)