Marionette.View

В состав Marionette входит `Marionette.View` - базовое представление с общей функциональностью, которое расширяют другие представления.

Замечание: Базовое представление `Marionette.View` не предназначено для использования напрямую. Оно присутствует в фреймворке как основа для дальнейшего расширения, в качестве исходного прототипа для будущих представлений вашего приложения.

Привязка обработчиков к событиям представления

Marionette.View расширяет `Marionette.BindTo`. (На самом деле, Marionette.View расширяет Backbone.View, с BindTo в Marionette периодически случаются недоразумения. Искать bindTo следует в документации к backbone.eventbinder - но данный объект является depricated начиная с 0.9.9 версии Marionette. - прим.пер.)

Рекомендуется использовать метод `listenTo` для привязки обработчиков к событиям моделей, коллекций, или для прослушивания любых других событий, поступающих из объектов Backbone и Marionette.

```
MyView = Backbone.Marionette.ItemView.extend({
 initialize: function(){
    this.listenTo(this.model, "change:foo", this.modelChanged);
    this.listenTo(this.collection, "add", this.modelAdded);
 },
 modelChanged: function(model, value){
 },
 modelAdded: function(model){
 }
});
```

Контекст (на него указывает ключевое слово `this`) будет автоматически установлен для данного представления. Есть возможность опционально менять контекст, передавая контекстный объект в качестве 4-го параметра в функцию `listenTo`.

Метод close

Представление реализует метод `close`, который автоматически вызывается регионами при их закрытии.
Детали реализации предполагают выполнение следующих действий:

- прекращается прослушивание всех событий, которые были переданы в `listenTo`
- прекращается прослушивание всех пользовательских событий представления
- прекращается прослушивание всех событий, поступающих из DOM
- элемент `this.el` удаляется из объектной модели документа
- вызывается метод `onBeforeClose`, если он существует в данном представлении
- вызывается метод `onClose`, если он существует в данном представлении

Таким образом, вы можете создавать код, который будет выполнен, когда представление завершит работу и будут очищены связанные с ним ресурсы. Для этого не нужно переопределять метод `close`, достаточно определить `onClose` в своем представлении.

```
Backbone.Marionette.ItemView.extend({
  onClose: function(){
    // ваш собственный код, который сработает на выходе
  }
});
```

View onBeforeClose

Точно так же и метод `onBeforeClose` будет вызван, если он задан в представлении. Если этот метод вернет `false`, представление не будет закрыто. Любые другие значения (включая null и undefined) не помешают представлению закрыться.

```
MyView = Marionette.View.extend({
  onBeforeClose: function(){
   // если мы не хотим, чтобы представление было выключено
   return false;
 }
});
var v = new MyView();
v.close(); // представление останется
```

View "dom:refresh" / onDomRefresh событие

Данное событие, или данный колбэк могут оказаться полезными при использовании сторонних библиотек, зависящих от DOM, таких как jQueryUI или KendoUI.

```
Backbone.Marionette.ItemView.extend({
  onDomRefresh: function(){
    // здесь можно производить всевозможные действия над элементом представления `el`,
    // поскольку он уже внедрен в страницу, и заполнен всем необходимым данному представлению
    // HTML-содержимым, готовым к работе.
  }
});
```

Для более детальной информации по поводу интеграции Marionette и KendoUI (что также применимо к jQueryUI и другим UI библиотекам), смотрите пост про интеграцию KendoUI и Backbone.

View.triggers

Представления могут определять ряд триггеров событий в качестве хеша, где что-то происходящее с DOM-элементами конвертируется в вызов соответствующих `view.triggerMethod`.

Левая часть хеша - стандартная запись, принятая в Backbone.View для событий в DOM, правая - события представления, которые могут быть порождены и обработаны в представлении.

```
MyView = Backbone.Marionette.ItemView.extend({
  // ...
  triggers: {
    "click .do-something": "something:do:it"
  }
});
view = new MyView();
view.render();
view.on("something:do:it", function(args){
  alert("событие конвертировано!");
});

// кликнем элемент с классом 'do-something'
// для демонстрации конвертации DOM-события
view.$(".do-something").trigger("click");
```

Вы можете также задать триггеры как функцию, возвращающую сконфигурированный хеш:

```
Backbone.Marionette.CompositeView.extend({
  triggers: function(){
    return {
      "click .that-thing": "that:i:sent:you"
    };
   }
});
```

Триггеры работают со всеми представлениями, произведенными от базового Marionette.View.

Аргументы, передаваемые в trigger-обработчики

Обработчики событий, сконфигурированных в `trigger`, получают единственный аргумент, который включает в себя поля:

- view
- model
- collection

Эти свойства соответствуют свойствам `view`, `model`, и `collection` представления, породившего событие.

```
MyView = Backbone.Marionette.ItemView.extend({
  // ...
  triggers: {
    "click .do-something": "some:event"
  }
});

view = new MyView();

view.on("some:event", function(args){
   args.view; // => экземпляр представления, который генерирует событие
   args.model; // => модель представления, если таковая существует
   args.collection; // => коллекция представления, если таковое существует
});
```

То обстоятельство, что мы имеем доступ к данным полям представления, обеспечивает большую гибкость в обработке событий. Например, компонент с возможностью переключения между вкладками, или разворачивающийся виджет (аккордеон), могут порождать определенное событие из различных вложенных в компонент представлений, которое будет обрабатываться одной-единственной функцией.

View.modelEvents и View.collectionEvents

Подобно хешу событий (`events`), представления могут задавать конфигурационный хеш
для событий коллекции и модели, где в качестве ключа будет выступать наименование события,
совершающегося над моделью или коллекцией, а в качестве значения - название метода представления.

```
Backbone.Marionette.CompositeView.extend({
  modelEvents: {
    "change:name": "nameChanged" // эквивалентно view.listenTo(view.model, "change:name", view.nameChanged,
view)
  },
  collectionEvents: {
    "add": "itemAdded" // эквивалентно view.listenTo(view.collection, "add", collection.itemAdded, view)
  },
  // ... обработчики событий
  nameChanged: function(){ /* ... */ },
  itemAdded: function(){ /* ... */ },
})
```

Таким образом, modelEvents и collectionEvents используют в определении зависимости метода от события безопасный с точки зрения расхода памяти метод `listenTo`, и устанавливают в качестве контекста обработчика (т.е. значения `this`) само данное представление. События привязываются к обработчикам в момент создания экземпляра представления, и если соответствующего обработчика не существует, будет брошено исключение.

И в modelEvents, и в collectionEvents события в конечном счете привязываются с помощью вызовов методов

Backbone.View `delegateEvents` и `undelegateEvents`. Это позволяет представлениям проходить через этапы многократного использования и переназначения событий.

(см. переопределенные в Marionette.View методы `delegateEvents` и `undelegateEvents` - прим.пер.)

Multiple Callbacks

Множество функций, обрабатывающих одно событие, может быть определено, если записать их названия в строку через пробел.

```
Backbone.Marionette.CompositeView.extend({
  modelEvents: {
    "change:name": "nameChanged thatThing"
  },
  nameChanged: function(){ },
  thatThing: function(){ },
});
```

Это справедливо как для `modelEvents`, так и для `collectionEvents`.

Конфигурация колбэков в качестве функции

В качестве обработчика может быть также определена анонимная функция, тогда она будет выступать единственным обработчиком события:

```
Backbone.Marionette.CompositeView.extend({
   modelEvents: {
     "change:name": function(){
        // здесь обрабатываем событие "change:name"
     }
   }
});
```

Это справедливо как для `modelEvents`, так и для `collectionEvents`.

Конфигурация событий в качестве функции

Стоит добавить, что modelEvents и collectionEvents могут быть представлены не в виде объектного литерала,
а в виде функции, которая возвращает хеш, соответствующий выше описанной конфигурации, т.е. ключ - это событие, а значение - функция, это событие обрабатывающая:

```
Backbone.Marionette.CompositeView.extend({
    modelEvents: function(){
      return { "change:name": "someFunc" };
    }
});
```

Метод View.serializeData

Метод `serializeData` сериализует данные модели/коллекции представления - отдавая предпочтение коллекции. То-есть, если в вашем представлении одновременно присутствуют и коллекция, и модель, метод `serializeData` вернет сериализованную коллекцию.

(На самом деле, метод serializeData присутствует только в производных от Marionette.View - Marionette.itemView и Marionette.Composite View - прим.пер.)

Метод View.bindUIElements

В некоторых случаях требуется иметь доступ к элементам пользовательского интерфейса внутри представления, чтобы получать от них какие-то данные, или манипулировать самими элементами.
Например, если у вас имеется какой-то элемент, который в зависимости от его состояния требуется показать или скрыть, или стоит задача добавить некоторый класс к определенному элементу.
Вместо того, чтобы жонглировать jQuery-селекторами по всему коду представления, вы можете просто задать `ui` хеш, который определит четкое соответствие между именами элементов
пользовательского интерфейса и их jQuery-селекторами. Впоследствии вы можете обращаться к ним с помощью конструкции `this.ui.elementName`. Примеры находятся в разделе, посвященном Marionette.ItemView.

Важно заметить, что эта функциональность обеспечена методом `bindUIElements`. Поскольку само базовое представление (Marionette.View) не реализует метод `render`, наследуя ваше представление напрямую от Marionette.View, вы столкнетесь с необходимостью
вызывать метод `bindUIElements` из метода `render` вашего представления. В Marionette.ItemView и Marionette.CompositeView о соответствующем методе уже позаботились.

Атрибут View.templateHelpers

Бывают случаи, когда нужно добавить в шаблон представления (view's template) определенную логику, а шаблонизатор не предоставляет такой возможности.
Например, мини-шаблонизатор Underscore не обеспечивает поддержку хелперов, в отличие от Handlebars.

Аттрибут `templateHelpers` может быть применен к любому объекту Marionette.View, который внедряет шаблон для отбражения данных внутри элементов. `templateHelpers` добавляет свое содержимое на правах миксина к объекту, возвращаемому методом `serializeData`. Это дает вам возможность создавать свои собственные вспомогательные методы (helpers), которые впоследствии будут применены внутри шаблонов.

Пример

```
<script id="my-template" type="text/html">
  I think that <%= showMessage() %>
</script>
```

```
MyView = Backbone.Marionette.ItemView.extend({
  template: "#my-template",
  templateHelpers: {
    showMessage: function(){
      return this.name + " is the coolest!"
    }
  }
});

model = new Backbone.Model({name: "Backbone.Marionette"});

view = new MyView({
  model: model
});

view.render(); //=> "I think that Backbone.Marionette is the coolest!";
```

Доступ к данным внутри хелперов

Доступ к данным внутри хелпера производится с помощью добавления `this`.
в качестве префикса. Так вы можете получить любые поля и методы сериализованного объекта, в том числе обратиться и к другим определенным для шаблона хелперам.

```
templateHelpers: {
  something: function(){
    return "Do stuff with " + this.name + " because it's awesome.";
  }
}
```

Объект или функция в качестве `templateHelpers`

В качестве `templateHelpers` вы можете определить объектный литерал (как показано выше), ссылку на объектный литерал, или функцию.

Если вы определяете в качестве `templateHelpers` функцию, она будет вызвана с текущим экземпляром представления в качестве контекста. Функция должна возвращать объект, который может быть добавлен к данным вашего представления на правах миксина.

```
Backbone.Marionette.ItemView.extend({
  templateHelpers: function(){
    return {
      foo: function(){ /* ... */ }
    }
  }
});
```

Замена шаблона представления

Бывают случаи, когда нужно заменить шаблон, используемый в представлении, на какой-то другой, например, на основании специфического аттрибута модели. Это возможно, если вы определите функцию `getTemplate` в вашем представлении, которая реализует логику выбора необходимого шаблона, и вернет его идентификатор.

```
MyView = Backbone.Marionette.ItemView.extend({
  getTemplate: function(){
    if (this.model.get("foo")){
      return "#some-template";
    } else {
      return "#a-different-template";
    }
  }
});
```

Это справедливо для всех типов представлений.