derickbailey / backbone.modelbinding

awesome model binding for Backbone.js — Read more

dev

formatter

master

- Files
- Commits
- Branches 3
- Tags 23
- Downloads 0

Latest commit to the **master** branch

bump version to v0.4.1, release notes, and minified

commit a6fc14e7bb

 derickbailey authored November 10, 2011

backbone.modelbinding / readme.md

- Fork and edit this file

100644 685 lines (493 sloc) 22.258 kb

- raw
- blame
- history

About Backbone.ModelBinding

Convention-based, awesome model binding for Backbone.js, inspired by Brad Phelan, Knockout.js ' data-binding capabilities, and Brandon Satrom's work with Knockout.

This plugin provides a simple, convention based mechanism to create bi-directional binding between your HTML form input elements and your Backbone models.

Instead of writing the same boiler plate code to read from your form inputs and populate the model attributes, for every input on your form, you can make a single call to `Backbone.ModelBinding.bind(myView)` and have all of your inputs automatically wired up. Any

change you make to a forn input will populate a corresponding model attribute for you. The binding is bi-directional, as well. This means that changes to your underlying model will be propagated to your form inputs without having to manually bind to your model's `change` events.

If you're looking for Knockout.js-style `data-bind` attributes, for Backbone, then this is the plugin for you. Backbone.ModelBinding provides some very basic support for `data-bind` attributes, allowing your Backbone model `change` events to modify nearly any HTML element on your page. Whether it's updating the text of a `<div>`, or changing the css class of an `<img>` tag, the `data-bind` support provides a very powerful and flexible means of creating a very rich user experience.

Getting Started

It's easy to get up and running. You only need to have Backbone (including underscore.js - a requirement for Backbone) and jQuery in your page before including the backbone.modelbining plugin.

Prerequisites

- Backbone.js v0.5.1 or higher
- jQuery v1.6.2 or higher

This is a plugin for Backbone.js and is built and tested against Backbone v0.5.1. It also uses jQuery to perform most of the bindng and manipulations, and is built and tested against v1.6.1. However, I am currently using this plugin in a production application with Backbone v0.3.3 and jQuery v1.5.1.

At this point, I make no guarantees of it working with any version of Backbone or jQuery, other than what it has been built and tested against. It works for me, so it may work for you with versions other than what is stated

Get The ModelBinding Plugin

Download the `backbone.modelbinding.js` file from this github repository and copy it into your javascripts folder. Add the needed `<script>` tag to bring the plugin into any page that wishes to use it. Be sure to include the modelbinding file *after* the backbone.js file.

Model Binding

The model binding code is executed with a call to `Backbone.ModelBinding.bind(view)`. There are several places that it can be called from, depending on your circumstances.

All of the element binding happens within the context of the view's `el`, therefore you must call the model binding code after your view's `el` has been populated with the elements that will be bound to.

Binding After Rendering

If your view modifies the html contents of the view's `el` in the `render` method, you should call the model binding after the modifications are made:

```
SomeView = Backbone.View.extend({
  render: function(){
    // ... render your form here
    $(this.el).html("... some html and content goes here ... ");
```

```
    // execute the model bindings
    Backbone.ModelBinding.bind(this);
  }
});
```

Binding A View That Does Not Render

If, however, your view has an `el` that represents an existing element in your html, and the contents of the `el` are not modified during a call to `render`, then you can make the call to the model binding code in the initializer or anywhere else.

```
<form id="some-form">
  Name: <input id="name">
</form>

FormView = Backbone.View.extend({
  el: "#some-form",

  initialize: function(){
    Backbone.ModelBinding.bind(this);
  }
});
```

Binding From Outside A View

There is no requirement for the model binding code to be called from within a view directly. You can bind the view from external code, like this:

```
FormView = Backbone.View.extend({
  el: "#some-form",
});

formView = new FormView();
Backbone.ModelBinding.bind(formView);
```

Model Unbinding

When your view has completed its work and is ready to be removed from the DOM, you not only need to unbnd your view's events (handled through the view's `remove` method, typically), you also need to unbind the model events that are bound in the view.

Backbone.ModelBinding can unbind its own events through a simple call to `Backbone.ModelBinding.unbind(view)`. If you do not call this method when your view is being closed / removed / cleaned up, then you may end up with memory leaks and zombie views that are still responding to model change events.

```
FormView = Backbone.View.extend({
  el: "#some-form",

  initialize: function(){
    Backbone.ModelBinding.bind(this);
  },

  close: function(){
    this.remove();
    this.unbind();
    Backbone.ModelBinding.unbind(this);
  }
```

```
});
```

Convention Bindings

Automatic bi-directional binding between your form input and your model.

The convention based binding requires no additional configuration or code in your view, other than calling the `Backbone.ModelBinding.bind(this);` as noted above. With the conventions binding, your `<input>` fields will be bound to the views model by the id of the input.

For example:

```
// something.html

<input id='name'>

// something.js

SomeModel = Backbone.Model.extend();

SomeView = Backbone.View.extend({
  render: function(){
    // ... render your form here

    // execute the defined bindings
    Backbone.ModelBinding.bind(this);
  }
});

model = new SomeModel();
view = new SomeView({model: model});

model.set({name: "some name"});
```

In this example, when `model.set` is called to set the name, "some name" will appear in the `#name` input field. Similarly, when the `#name` input field is changed, the value entered into that field will be sent to the model's `name` property.

Data-Bind Attributes

Backbone.ModelBinding supports Knockout-style data-bind attributes on any arbitrary HTML element. These bindings will populate any attribute, the text, or HTML contents of an HTML element based on your configurations. This is particularly useful when a model that is being edited is also being displayed elsewhere on the screen.

To bind an element to a model's properties, add a `data-bind` attribute to the element and specify what should be updated with which model property using a `elementAttr modelAttr` format. For example `<span data-bind="text name">` will update the span's text with the model's name property, when the model's name changes.

```
<form>
  <input type="text" id="name">
</form>
Name: <span data-bind="text name">

SomeView = Backbone.View.extend({
```

```
  // ...

  render: function(){
    // ...
    Backbone.ModelBinding.bind(this);
  }
});

someModel = new SomeModel();
someView = new SomeView({model: someModel});
```

In this example, the model's `name` will be updated when you type into the text box and then tab or click away from it (to fire the change event). When the model's `name` property is updated, the `data-bind` convention will pick up the change and set the text of the `span` to the model's name.

Data-Bind Multiple Attributes

Multiple attributes can be specified for a single element's `data-bind` by separating each with a `;` (semi-colon). For example:

```
<form>
  <input type="text" id="name">
</form>
Name: <span data-bind="text name; class name">

SomeView = Backbone.View.extend({
  // ...

  render: function(){
    // ...
    Backbone.ModelBinding.bind(this);
  }
});

someModel = new SomeModel();
someView = new SomeView({model: someModel});
```

In this example, both the text and the css class will be updated when you change the name input. You can data-bind as many attributes as you need, in this manner.

Special Cases For data-bind

There are several special cases for the data-bind attribute. These allow a little more functionality than just setting an attribute on an element.

- (default) - if you only specify the model property, defaults to the text of the html element
- text - replace the text contents of the element
- html - replace the html contents of the element
- enabled - enable or disable the html element

(default)

If you only specify the model's property in the data-bind attribute, then the data-bind will bind the value of that model property to the `text` of the html element.

```
<div data-bind="name"/>
```

See the document for data-bind text, below.

text

If you set the data-bind attribute to use `text`, it will replace the text contents of the html element instead of just setting an element attribute.

```
<div id="someDiv" data-bind="text someProperty"></div>


someModel.set({someProperty: "some value"});
```

html

If you set the data-bind attribute to use `html`, it will replace the entire inner html of the html element, instead of just setting an element attribute.

```
<div id="someDiv" data-bind="html someProperty"></div>


someModel.set({someProperty: "some value"});
```

enabled

This special case breaks the html element standard of using a `disabled` attribute, specifically to inver the logic used for enabling / disabling an element, to keep the data-bind attribute clean and easy to read.

If you have a model with a property that indicates a negative state, such as `invalid`, then you can use a data-bind attribute of `disabled`:

```
<button id="someButton" data-bind="disabled invalid"></div>


someModel.set({invalid: true});
```

However, some developers prefer to use positive state, such as `isValid`. In this case, setting the disabled attribute to the model's isValid property would result in the button being disabled when the model is valid and enabled when the model is not valid. To correct this, a special case has been added to enable and disable an element with `enabled`.

```
<button id="someButton" data-bind="enabled isValid"></div>


someModel.set({isValid: false});
```

This will disable the button when the model is invalid and enable the button when the model is valid.

displayed

This allows you to specify that an element should be shown or hidden by setting the css of the element according to the value of the model properties specified.

```
<div data-bind="displayed isValid" />

someModel.set({isValid: false});
```

When the model's property is set to false, the HTML element's `style` css will be set to `none`. When

When the model's property is set to false, the HTML element's `display` css will be set to `none`. When the model's property is set to true, the HTML element's `display` css will be set to `block`.

hidden

This is the inverse of `displayed`.

```
<div data-bind="hidden isValid" />

someModel.set({isValid: true});
```

When the model's property is set to false, the HTML element's `display` css will be set to `block`. When the model's property is set to true, the HTML element's `display` css will be set to `none`.

Data-Bind Substitutions

If a model's property is `unset`, the data-bind may not update correctly when using `text` or `html` as the bound attribute of the element.

```
<div data-bind="text something"></div>

model.set({something: "whatever"});
model.unset("something");
```

The result will be a div with it's text set to "". this is handled through the data-bind's substitutions for undefined values. The default substitution is to replace an undefined value with an empty string. However, this can be per attribute:

```
<div data-bind="text something"></div>
<div data-bind="html something"></div>

Backbone.ModelBinding.Configuration.dataBindSubst({
  text: "undefined. setting text to this",
  html: " "
});
model.set({something: "whatever"});
model.unset("something");
```

The result of this example will be a div that displays "undefined. setting the text to this" and a div whose contents is a single space, instead of being empty.

Form Binding Conventions

The following form input types are supported by the form convention binder:

- text
- textarea
- password
- checkbox
- select
- radio button groups

HTML5 * number * range * email * url * tel * search

Radio buttons are group are assumed to be grouped by the `name` attribute of the radio button items.

Select boxes will populate 2 separate fields into the model that they are bound to. The standard "$bindable

Select boxes will populate 2 separate fields into the model that they are bound to. The standard `#fieldid` will be populated with the selected value. An additional `{#fieldid}_text` will be populated with the text from the selected item. For example, a selected option of

```
<select id='company'>
  <option value="foo_bar">Foo Bar Widgets, Inc.</option>
  ...
</select>
```

will populate the `company` property of the model with "foo_bar", and will populate the `company_text` property of the model with "Foo Bar Widgets, Inc."

There is no support for hidden fields at the moment, because there is no 'change' event that jQuery can listen to on a hidden field.

Configuring The Bound Attributes

The convention binding system allows you to specify the attribute to use for the convention, by the input type. The default configuration is:

```
{
  text: "id",
  textarea: "id",
  password: "id",
  radio: "name",
  checkbox: "id",
  select: "id"
}
```

You can override this configuration and use any attribute you wish, by specifying any or all of these input types when you call the model binding. This is useful when you have field ids that do not match directly to the model properties.

Override All Element Binding Attributes

The following will use use the `class` attribute's value as the binding for all input field:

```
SomeView = Backbone.View.extend({
  render: function(){
    // ... some rendering here
    Backbone.ModelBinding.bind(this, { all: "class" });
  }
});
```

```
<input type="text" id="the_model_name" class="name">
```

If the same convention needs to be used throughout an application, and not just withing a single view, the configuration can be set at a global level:

```
Backbone.ModelBinding.Configuration.configureAllBindingAttributes("class");
```

Override Individual Element Binding Attributes

The following will use a `modelAttr` attribute value as the convention for text boxes, only.

```
SomeView = Backbone.View.extend({

  render: function(){
```

```
  render: function(){
    // ... some rendering here
    Backbone.ModelBinding.bind(this, { text: "modelAttr" });
  }
});

<input type="text" id="the_model_name" modelAttr="name">
```

When this text box has it's value changed, the model's `name` property will be populated with the value instead of `the_model_name`.

If the same convention needs to be used throughout an application, and not just withing a single view, the configuration can be set at a global level:

```
Backbone.ModelBinding.Configuration.configureBindingAttributes({text: "modelAttr"});
```

Now all text boxes will update the model property specified in the text box's `modelAttr`.

Pluggable Conventions

The convention based bindings are pluggable. Each of the existing form input types can have it's convention replaced and you can add your own conventions for input types not currently handled, etc.

To replace a convention entirely, you need to supply a json document that has two pieces of information: a jQuery selector string and an object with a `bind` method. Place the convention in the `Backbone.ModelBinding.Conventions` and it will be picked up and executed. The `bind` method receives three parameters: the jQuery selector you specified, the Backbone view, and the model being bound.

You can replace the handler of an existing convention. For example, this will set the value of a textbox called `#name` to some text, instead of doing any real binding.

```
var nameSettingsHandler = {
  bind: function(selector, view, model){
    view.$("#name").val("a custom convention supplied this name");
  }
};

Backbone.ModelBinding.Conventions.text.handler = nameSettingsHandler;
```

You can also create your own conventions that do just about anything you want. Here's an example that modifies the contents of <p> tags:

```
var PConvention = {
  selector: "p",
  handler: {
    bind: function(selector, view, model){
      view.$(selector).each(function(index){
        var name = model.get("name");
        $(this).html(name);
      });
    }
  }
};

Backbone.ModelBinding.Conventions.paragraphs = PConvention;
```

This example will find all <p> tags in the view and render the `name` property from the model into that

This example will find all <p> tags in the view and render the name property from the model into that paragraph, replacing all other text. Note that the name of the convention is set to `paragraphs` when added to the conventions. This name did not exist prior to this assignment, so the convention was added. If you assign a convention to an existing name, you will replace that convention.

The list of existing conventions includes:

- text
- password
- radio
- checkbox
- select
- textarea
- number
- range
- tel
- search
- url
- email
- databind

For fully functional, bi-directional binding convention examples, check out the source code to Backbone.ModelBinding in the `backbone.modelbinding.js` file.

Release Notes

v0.4.1

- Bind the existing value from an input element to the model on render, if no model value exists
- Added HTML5 input types to the form binding conventions: number, range, tel, search, url, email

v0.4.0

- Major internal rewrite to facilitate maintenance, new features, etc
- No public API changes (at least, I hope not!)
- Correctly unbind HTML element / jQuery events, when calling `unbind`
- Corrected the `data-bind` method for showing / hiding an element when using the `displayed` and `hidden` settings
- Corrected the global configuration so that it does not get reset after you call `bind` the first time
- Now uses an internal object call `ModelBinder`, which is attached to the view that binding occurs on
- The `ModelBinder` instance stores all of the binding configuration and callbacks for that view, allowing much faster / easier / better unbinding

v0.3.10

- When binding to a select box and the model has a value not present in the box, reset the model's value to the box's
- Fix for using Backbone.noConflict(), Backbone was either wrong version or undefined when it was used inside ModelBinding.
- Added some missing ;
- Fix some documentation issues

- Fix some documentation issues

v0.3.9

- Fixed an issue with jQuery 1.6.4 determining if check boxes are checked or not
- Minor internal clean up

v0.3.8

- Fix for Internet Explorer not having a `trim` method on strings

v0.3.7

- Data-bind multiple attributes for a single element
- Default data-bind substitutaion to '' for all attributes

v0.3.6

- Fixed a bug that prevented `<input>` elements with no `type` attribute from being bound

v0.3.5

- Fixed a bug in configuring the binding attribute for textarea elements

v0.3.4

- Data-bind defaults to the html element's `text` if you only specify the model property: `data-bind="name"`
- Fixed issue with binding 1 / 0 to checkboxes (truthy / falsy values)

v0.3.3

- Added data-bind attribute for setting an HTML element's `display` css
- Added inserve of data-bind `displayed` as data-bind `hidden`
- Corrected issue with binding a model's property to a checkbox, when the property is false

v0.3.2

- Data-bind substitutions - lets you replace "undefined" with another, set value, when using data-bind

v0.3.1

- Corrected issue with `unset`ing a model property, in the data-bind convention

v0.3.0

- **Breaking Change:** Changed the `Backbone.ModelBinding.call(view)` method signature to `Backbone.ModelBinding.bind(view)`
- Added ability to unbind model binding with `unbind` method, to prevent memory leaks and zombie forms

- Added backbone.modelbinding.min.js to the repository, compiled with Google's Closure Compiler Service
- Updated the selectors used for the conventions. Text inputs are now found with "input:text", which should select all text inputs, even without a `type='text'` attribute (though this seems to be buggy in jQuery v1.6.2)
- Significant internal restructuring of code

v0.2.4

- Data-bind will bind the model's value immediately instead of waiting for the model's value to change
- Support `enabled` functionality for data-bind: `data-bind="enabled isValid"`
- Documented existing support for data-bind `disabled`: `data-bind="disabled invalid"`

v0.2.3

- Fixes for 'falsey' value bindings
- Update the docs to include when and where to call the model bindings

v0.2.2

- Making some global vars not global

v0.2.1

- Configuration to easily set all binding attributes for all elements
- Fix for IE
- Making some global vars not global

v0.1.0 - v0.2.0

- Added data-bind convention
- Added configuration options
- Conventions for all form input types
- Removed formBinding code
- Removed htmlBinding code
- Significant internal code cleanup and restructuring

Legal Mumbo Jumbo (MIT License)

portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Markdown Cheat Sheet

Format Text

Headers

```
# This is an <h1> tag
## This is an <h2> tag
###### This is an <h6> tag
```

Text styles

```
*This text will be italic*
_This will also be italic_
**This text will be bold**
__This will also be bold__

*You **can** combine them*
```

Lists

Unordered

```
* Item 1
* Item 2
  * Item 2a
  * Item 2b
```

Ordered

```
1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b
```

Miscellaneous

Images

```
![GitHub Logo](/images/logo.png)
Format: ![Alt Text](url)
```

Links

```
http://github.com - automatic!
[GitHub](http://github.com)
```

Blockquotes

```
As Kanye West said:
```

```
> We're living the future so
> the present is our past.
```

Code Examples in Markdown

Syntax highlighting with GFM

````
```javascript
function fancyAlert(arg) {
  if(arg) {
    $.facebox({div:'#foo'})
  }
}
```
````

Or, indent your code 4 spaces

```
Here is a Python code example
without syntax highlighting:

    def foo:
      if not bar:
        return true
```

Inline code for comments

```
I think you should use an
`<addr>` element here instead.
```

Something went wrong with that request. Please try again. Dismiss