Backbone.js

Backbone supplies structure to JavaScript-heavy applications by providing **models** with key-value binding and custom events, **collections** with a rich API of enumerable functions, **views** with declarative event handling, and connects it all to your existing application over a RESTful JSON interface.

The project is hosted on GitHub, and the annotated source code is available, as well as an online test suite, and example application.

You can report bugs and discuss features on the issues page, on Freenode in the `#documentcloud` channel, or send tweets to @documentcloud.

Backbone is an open-source component of DocumentCloud.

Downloads & Dependencies (Right-click, and use "Save As")

Development Version (0.3.3)	*35kb, Uncompressed with Comments*
Production Version (0.3.3)	*3.9kb, Packed and Gzipped*

Backbone's only hard dependency is Underscore.js. For RESTful persistence, and DOM manipulation with Backbone.View, it's highly recommended to include json2.js, and either jQuery or Zepto.

Introduction

When working on a web application that involves a lot of JavaScript, one of the first things you learn is to stop tying your data to the DOM. It's all too easy to create JavaScript applications that end up as tangled piles of jQuery selectors and callbacks, all trying frantically to keep data in sync between the HTML UI, your JavaScript logic, and the database on your server. For rich client-side applications, a more structured approach is helpful.

With Backbone, you represent your data as Models, which can be created, validated, destroyed, and saved to the server. Whenever a UI action causes an attribute of a model to change, the model triggers a *"change"* event; all the Views that display the model's data are notified of the event, causing them to re-render. You don't have to write the glue code that looks into the DOM to find an element with a specific *id*, and update the HTML manually — when the model changes, the views simply update themselves.

Many of the examples that follow are runnable. Click the *play* button to execute them.

Backbone.Events

Events is a module that can be mixed in to any object, giving the object the ability to bind and trigger custom named events. Events do not have to be declared before they are bound, and may take passed arguments. For example:

```
var object = {};

_.extend(object, Backbone.Events);

object.bind("alert", function(msg) {
  alert("Triggered " + msg);
});

object.trigger("alert", "an event");
```

bind `object.bind(event, callback)`

Bind a **callback** function to an object. The callback will be invoked whenever the **event** (specified by an arbitrary string identifier) is fired. If you have a large number of different events on a page, the convention is to use colons to namespace them: `"poll:start"`, or `"change:selection"`

Callbacks bound to the special `"all"` event will be triggered when any event occurs, and are passed the name of the event as the first argument. For example, to proxy all events from one object to another:

```
proxy.bind("all", function(eventName) {
  object.trigger(eventName);
});
```

unbind `object.unbind([event], [callback])`

Remove a previously-bound **callback** function from an object. If no callback is specified, all callbacks for the **event** will be removed. If no event is specified, *all* event callbacks on the object will be removed.

```
object.unbind("change", onChange);  // Removes just the onChange callback.

object.unbind("change");            // Removes all "change" callbacks.

object.unbind();                    // Removes all callbacks on object.
```

trigger `object.trigger(event, [*args])`

Trigger callbacks for the given **event**. Subsequent arguments to **trigger** will be passed along to the event callbacks.

Backbone.Model

Models are the heart of any JavaScript application, containing the interactive data as well as a large part of the logic surrounding it: conversions, validations, computed properties, and access control. You extend **Backbone.Model** with your domainspecific methods, and **Model** provides a basic set of functionality for managing changes.

The following is a contrived example, but it demonstrates defining a model with a custom method, setting an attribute, and firing an event keyed to changes in that specific attribute. After running this code once, `sidebar` will be available in your browser's console, so you can play around with it.

```
var Sidebar = Backbone.Model.extend({
  promptColor: function() {
    var cssColor = prompt("Please enter a CSS color:");
    this.set({color: cssColor});
  }
});

window.sidebar = new Sidebar;

sidebar.bind('change:color', function(model, color) {
  $('#sidebar').css({background: color});
});

sidebar.set({color: 'white'});

sidebar.promptColor();
```

extend `Backbone.Model.extend(properties, [classProperties])`

To create a **Model** class of your own, you extend **Backbone.Model** and provide instance **properties**, as well as optional **classProperties** to be attached directly to the constructor function.

extend correctly sets up the prototype chain, so subclasses created with **extend** can be further extended and subclassed as far as you like.

```
var Note = Backbone.Model.extend({

  initialize: function() { ... },

  author: function() { ... },

  coordinates: function() { ... },

  allowedToEdit: function(account) {
    return true;
  }

});

var PrivateNote = Note.extend({

  allowedToEdit: function(account) {
    return account.owns(this);
  }

});
```

Brief aside on `super`*: JavaScript does not provide a simple way to call super — the function of the* *same name defined higher on the prototype chain. If you override a core function like `set`, or `save`, and you want to invoke the parent object's implementation, you'll have to explicitly call it, along these lines:*

```
var Note = Backbone.Model.extend({
  set: function(attributes, options) {
    Backbone.Model.prototype.set.call(this, attributes, options);
    ...
  }
});
```

constructor / initialize `new Model([attributes])`

When creating an instance of a model, you can pass in the initial values of the **attributes**, which will be set on the model. If you define an **initialize** function, it will be invoked when the model is created.

```
new Book({
  title: "One Thousand and One Nights",
  author: "Scheherazade"
});
```

get `model.get(attribute)`

Get the current value of an attribute from the model. For example: `note.get("title")`

escape `model.escape(attribute)`

Similar to get, but returns the HTML-escaped version of a model's attribute. If you're interpolating data from the model into HTML, using **escape** to retrieve attributes will prevent XSS attacks.

```
var hacker = new Backbone.Model({
  name: "<script>alert('xss')</script>"
});

alert(hacker.escape('name'));
```

set `model.set(attributes, [options])`

Set a hash of attributes (one or many) on the model. If any of the attributes change the models state, a `"change"` event will be triggered, unless `{silent: true}` is passed as an option. Change events for specific attributes are also triggered, and you can bind to those as well, for example: `change:title`, and `change:content`.

```
note.set({title: "October 12", content: "Lorem Ipsum Dolor Sit Amet..."});
```

If the model has a validate method, it will be validated before the attributes are set, no changes will occur if the validation fails, and **set** will return `false`. You may also pass an `error` callback in the options, which will be invoked instead of triggering an `"error"` event, should validation fail.

unset `model.unset(attribute, [options])`

Remove an attribute by deleting it from the internal attributes hash. Fires a `"change"` event unless `silent` is passed as an option.

clear `model.clear([options])`

Removes all attributes from the model. Fires a `"change"` event unless `silent` is passed as an option.

id `model.id`

A special property of models, the **id** is an arbitrary string (integer id or UUID). If you set the **id** in the attributes hash, it will be copied onto the model as a direct property. Models can be retrieved by id from collections, and the id is used to generate model URLs by default.

cid `model.cid`

A special property of models, the **cid** or client id is a unique identifier automatically assigned to all models when they're first created. Client ids are handy when the model has not yet been saved to the server, and does not yet have its eventual true **id**, but already needs to be visible in the UI. Client ids take the form: `c1, c2, c3 ...`

attributes `model.attributes`

The **attributes** property is the internal hash containing the model's state. Please use set to update the attributes instead of modifying them directly. If you'd like to retrieve and munge a copy of the model's attributes, use toJSON instead.

defaults `model.defaults`

The **defaults** hash can be used to specify the default attributes for your model. When creating an instance of the model, any unspecified attributes will be set to their default value.

```
var Meal = Backbone.Model.extend({
  defaults: {
    "appetizer":  "caesar salad",
    "entree":     "ravioli",
    "dessert":    "cheesecake"
  }
});

alert("Dessert will be " + (new Meal).get('dessert'));
```

toJSON `model.toJSON()`

Return a copy of the model's attributes for JSON stringification. This can be used for persistence, serialization, or for augmentation before being handed off to a view. The name of this method is a bit confusing, as it doesn't actually return a JSON string — but I'm afraid that it's the way that the JavaScript API for **JSON.stringify** works.

```
var artist = new Backbone.Model({
  firstName: "Wassily",
  lastName: "Kandinsky"
});

artist.set({birthday: "December 16, 1866"});
```

```
alert(JSON.stringify(artist));
```

fetch `model.fetch([options])`

Refreshes the model's state from the server. Useful if the model has never been populated with data, or if you'd like to ensure that you have the latest server state. A `"change"` event will be triggered if the server's state differs from the current attributes. Accepts `success` and `error` callbacks in the options hash, which are passed `(model, response)` as arguments.

```
// Poll every 10 seconds to keep the channel model up-to-date.
setInterval(function() {
  channel.fetch();
}, 10000);
```

***Cautionary Note:** When fetching or saving a model, make sure that the model is part of a collection with a url property specified, or that the model itself has a complete url function of its own, so that the request knows where to go.*

save `model.save(attributes, [options])`

Save a model to your database (or alternative persistence layer), by delegating to Backbone.sync. If the model has a validate method, and validation fails, the model will not be saved. If the model isNew, the save will be a `"create"` (HTTP `POST`), if the model already exists on the server, the save will be an `"update"` (HTTP `PUT`). Accepts `success` and `error` callbacks in the options hash, which are passed `(model, response)` as arguments. The `error` callback will also be invoked if the model has a `validate` method, and validation fails.

In the following example, notice how because the model has never been saved previously, our overridden version of `Backbone.sync` receives a `"create"` request.

```
Backbone.sync = function(method, model) {
  alert(method + ": " + JSON.stringify(model));
};

var book = new Backbone.Model({
  title: "The Rough Riders",
  author: "Theodore Roosevelt"
});

book.save();
```

destroy `model.destroy([options])`

Destroys the model on the server by delegating an HTTP `DELETE` request to Backbone.sync. Accepts `success` and `error` callbacks in the options hash.

```
book.destroy({success: function(model, response) {
  ...
}});
```

validate `model.validate(attributes)`

This method is left undefined, and you're encouraged to override it with your custom validation logic, if you have any that can be performed in JavaScript. **validate** is called before `set` and `save`, and is passed the attributes that are about to be updated. If the model and attributes are valid, don't return anything from **validate**; if the attributes are invalid, return an error of your choosing. It can be as simple as a string error message to be displayed, or a complete error object that describes the error programmatically. `set` and `save` will not continue if **validate** returns an error. Failed validations trigger an `"error"` event.

```
var Chapter = Backbone.Model.extend({
  validate: function(attrs) {
    if (attrs.end < attrs.start) {
      return "can't end before it starts";
    }
  }
});

var one = new Chapter({
  title : "Chapter One: The Beginning"
});

one.bind("error", function(model, error) {
  alert(model.get("title") + " " + error);
});

one.set({
  start: 15,
  end:   10
});
```

`"error"` events are useful for providing coarse-grained error messages at the model or collection level, but if you have a specific view that can better handle the error, you may override and suppress the event by passing an `error` callback directly:

```
account.set({access: "unlimited"}, {
  error: function(model, error) {
    alert(error);
  }
});
```

url `model.url()`

Returns the relative URL where the model's resource would be located on the server. If your models are located somewhere else, override this method with the correct logic. Generates URLs of the form: `"/[collection]/[id]"`.

Delegates to Collection#url to generate the URL, so make sure that you have it defined. A model with an id of `101`, stored in a Backbone.Collection with a `url` of `"/notes"`, would have this URL: `"/notes/101"`

parse `model.parse(response)`

parse is called whenever a model's data is returned by the server, in fetch, and save. The function is passed the raw `response` object, and should return the attributes hash to be set on the model. The default implementation is a no-op, simply passing through the JSON response. Override this if you need to work with a preexisting API, or better namespace your responses.

If you're working with a Rails backend, you'll notice that Rails' default `to_json` implementation includes a model's attributes under a namespace. To disable this behavior for seamless Backbone integration, set:

```
ActiveRecord::Base.include_root_in_json = false
```

clone `model.clone()`

Returns a new instance of the model with identical attributes.

isNew `model.isNew()`

Has this model been saved to the server yet? If the model does not yet have an `id`, it is considered to be new.

change `model.change()`

Manually trigger the `"change"` event. If you've been passing `{silent: true}` to the set function in order to aggregate rapid changes to a model, you'll want to call `model.change()` when you're all finished.

hasChanged `model.hasChanged([attribute])`

Has the model changed since the last `"change"` event? If an **attribute** is passed, returns `true` if that specific attribute has changed.

```
book.bind("change", function() {
  if (book.hasChanged("title")) {
    ...
  }
});
```

changedAttributes `model.changedAttributes([attributes])`

Retrieve a hash of only the model's attributes that have changed. Optionally, an external **attributes** hash can be passed in, returning the attributes in that hash which differ from the model. This can be used to figure out which portions of a view should be updated, or what calls need to be made to sync the changes to the server.

previous `model.previous(attribute)`

During a `"change"` event, this method can be used to get the previous value of a changed attribute.

```
var bill = new Backbone.Model({
  name: "Bill Smith"
});

bill.bind("change:name", function(model, name) {
  alert("Changed name from " + bill.previous("name") + " to " + name);
});

bill.set({name : "Bill Jones"});
```

previousAttributes `model.previousAttributes()`

Return a copy of the model's previous attributes. Useful for getting a diff between versions of a model, or getting back to a valid state after an error occurs.

Backbone.Collection

Collections are ordered sets of models. You can to bind `"change"` events to be notified when any model in the collection has been modified, listen for `"add"` and `"remove"` events, `fetch` the collection from the server, and use a full suite of Underscore.js methods.

Collections may also listen for changes to specific attributes in their models, for example: `Documents.bind("change:selected", ...)`

extend `Backbone.Collection.extend(properties, [classProperties])`

To create a **Collection** class of your own, extend **Backbone.Collection**, providing instance **properties**, as well as optional **classProperties** to be attached directly to the collection's constructor function.

model `collection.model`

Override this property to specify the model class that the collection contains. If defined, you can pass raw attributes objects (and arrays) to add, create, and refresh, and the attributes will be converted into a model of the proper type.

```
var Library = Backbone.Collection.extend({
  model: Book
});
```

constructor / initialize `new Collection([models], [options])`

When creating a Collection, you may choose to pass in the initial array of **models**. The collection's comparator function may be included as an option. If you define an **initialize** function, it will be invoked when the collection is created.

```
var tabs = new TabSet([tab1, tab2, tab3]);
```

models `collection.models`

Raw access to the JavaScript array of models inside of the collection. Usually you'll want to use `get`, `at`, or the **Underscore methods** to access model objects, but occasionally a direct reference to the array is desired.

toJSON `collection.toJSON()`

Return an array containing the attributes hash of each model in the collection. This can be used to serialize and persist the collection as a whole. The name of this method is a bit confusing, because it conforms to JavaScript's JSON API.

```
var collection = new Backbone.Collection([
  {name: "Tim", age: 5},
  {name: "Ida", age: 26},
  {name: "Rob", age: 55}
]);

alert(JSON.stringify(collection));
```

Underscore Methods (25)

Backbone proxies to **Underscore.js** to provide 25 iteration functions on **Backbone.Collection**. They aren't all documented here, but you can take a look at the Underscore documentation for the full details…

- forEach (each)
- map
- reduce (foldl, inject)
- reduceRight (foldr)
- find (detect)
- filter (select)
- reject
- every (all)
- some (any)
- include
- invoke
- max
- min
- sortBy
- sortedIndex
- toArray
- size
- first
- rest
- last
- without
- indexOf
- lastIndexOf
- isEmpty
- chain

```
Books.each(function(book) {
  book.publish();
});

var titles = Books.map(function(book) {
  return book.get("title");
});

var publishedBooks = Books.filter(function(book) {
  return book.get("published") === true;
});

var alphabetical = Books.sortBy(function(book) {
  return book.author.get("name").toLowerCase();
});
```

add `collection.add(models, [options])`

Add a model (or an array of models) to the collection. Fires an `"add"` event, which you can pass `{silent: true}` to suppress. If a model property is defined, you may also pass raw attributes objects, and have them be vivified as instances of the model.

```
var ships = new Backbone.Collection;

ships.bind("add", function(ship) {
  alert("Ahoy " + ship.get("name") + "!");
});

ships.add([
  {name: "Flying Dutchman"},
  {name: "Black Pearl"}
]);
```

remove `collection.remove(models, [options])`

Remove a model (or an array of models) from the collection. Fires a `"remove"` event, which you can use `silent` to suppress.

get `collection.get(id)`

Get a model from a collection, specified by **id**.

```
var book = Library.get(110);
```

getByCid `collection.getByCid(cid)`

Get a model from a collection, specified by client id. The client id is the `.cid` property of the model, automatically assigned whenever a model is created. Useful for models which have not yet been saved to the server, and do not yet have true ids.

at `collection.at(index)`

Get a model from a collection, specified by index. Useful if your collection is sorted, and if your collection isn't sorted, **at** will still retrieve models in insertion order.

length `collection.length`

Like an array, a Collection maintains a `length` property, counting the number of models it contains.

comparator `collection.comparator`

By default there is no **comparator** function on a collection. If you define a comparator, it will be used to maintain the collection in sorted order. This means that as models are added, they are inserted at the correct index in `collection.models`. Comparator functions take a model and return a numeric or string value by which the model should be ordered relative to others.

Note how even though all of the chapters in this example are added backwards, they come out in the proper order:

```
var Chapter  = Backbone.Model;
var chapters = new Backbone.Collection;

chapters.comparator = function(chapter) {
  return chapter.get("page");
};

chapters.add(new Chapter({page: 9, title: "The End"}));
chapters.add(new Chapter({page: 5, title: "The Middle"}));
chapters.add(new Chapter({page: 1, title: "The Beginning"}));

alert(chapters.pluck('title'));
```

Brief aside: This comparator function is different than JavaScript's regular "sort", which must return `0`, `1`, or `-1`, and is more similar to a `sortBy` — a much nicer API.

sort `collection.sort([options])`

Force a collection to re-sort itself. You don't need to call this under normal circumstances, as a collection with a comparator function will maintain itself in proper sort order at all times. Calling **sort** triggers the collection's `"refresh"` event, unless silenced by passing `{silent: true}`

pluck `collection.pluck(attribute)`

Pluck an attribute from each model in the collection. Equivalent to calling `map`, and returning a single attribute from the iterator.

```
var stooges = new Backbone.Collection([
  new Backbone.Model({name: "Curly"}),
  new Backbone.Model({name: "Larry"}),
  new Backbone.Model({name: "Moe"})
]);

var names = stooges.pluck("name");

alert(JSON.stringify(names));
```

url `collection.url or collection.url()`

Set the **url** property (or function) on a collection to reference its location on the server. Models within the collection will use **url** to construct URLs of their own.

```
var Notes = Backbone.Collection.extend({
  url: '/notes'
});

// Or, something more sophisticated:

var Notes = Backbone.Collection.extend({
  url: function() {
    return this.document.url() + '/notes';
  }
});
```

parse `collection.parse(response)`

parse is called by Backbone whenever a collection's models are returned by the server, in fetch. The function is passed the raw `response` object, and should return the array of model attributes to be added to the collection. The default implementation is a no-op, simply passing through the JSON response. Override this if you need to work with a preexisting API, or better namespace your responses.

```
var Tweets = Backbone.Collection.extend({
  // The Twitter Search API returns tweets under "results".
  parse: function(response) {
    return response.results;
  }
});
```

fetch `collection.fetch([options])`

Fetch the default set of models for this collection from the server, refreshing the collection when they arrive. The **options** hash takes `success` and `error` callbacks which will be passed `(collection, response)` as arguments. When the model data returns from the server, the collection will refresh. Delegates to Backbone.sync under the covers, for custom persistence strategies. The server handler for **fetch** requests should return a JSON array of models.

```
Backbone.sync = function(method, model) {
  alert(method + ": " + model.url);
};

var Accounts = new Backbone.Collection;
Accounts.url = '/accounts';

Accounts.fetch();
```

Note that **fetch** should not be used to populate collections on page load — all models needed at load time should already be bootstrapped in to place. **fetch** is intended for lazily-loading models for interfaces that are not needed immediately: for example, documents with collections of notes that may be toggled open and closed.

refresh `collection.refresh(models, [options])`

Adding and removing models one at a time is all well and good, but sometimes you have so many models to change that you'd rather just update the collection in bulk. Use **refresh** to replace a collection with a new list of models (or attribute hashes), triggering a single `"refresh"` event at the end. Pass `{silent: true}` to suppress the `"refresh"` event.

Here's an example using **refresh** to bootstrap a collection during initial page load, in a Rails application.

```
<script>
  Accounts.refresh(<%= @accounts.to_json %>);
</script>
```

create `collection.create(attributes, [options])`

Convenience to create a new instance of a model within a collection. Equivalent to instantiating a model with a hash of attributes, saving the model to the server, and adding the model to the set after being successfully created. Returns the model, or `false` if a validation error prevented the model from being created. In order for this to work, your should set the model property of the collection.

```
var Library = Backbone.Collection.extend({
  model: Book
});

var NYPL = new Library;

var othello = NYPL.create({
  title: "Othello",
  author: "William Shakespeare"
});
```

Backbone.Controller

Web applications often choose to change their URL fragment (`#fragment`) in order to provide shareable, bookmarkable URLs for an Ajax-heavy application. **Backbone.Controller** provides methods for routing client-side URL fragments, and connecting them to actions and events.

Backbone controllers do not yet make use of HTML5 ***pushState*** *and* ***replaceState****. Currently,* ***pushState*** *and* ***replaceState*** *need special handling on the server-side, cause you to mint duplicate URLs, and have an incomplete API. We may start supporting them in the future when these issues have been resolved.*

During page load, after your application has finished creating all of its controllers, be sure to call `Backbone.history.start()` to route the initial URL.

extend `Backbone.Controller.extend(properties, [classProperties])`

Get started by creating a custom controller class. You'll want to define actions that are triggered when certain URL fragments are matched, and provide a routes hash that pairs routes to actions.

```
var Workspace = Backbone.Controller.extend({

  routes: {
    "help":                 "help",    // #help
    "search/:query":        "search",  // #search/kiwis
    "search/:query/p:page": "search"   // #search/kiwis/p7
  },

  help: function() {
    ...
  },

  search: function(query, page) {
    ...
  }

});
```

routes `controller.routes`

The routes hash maps URLs with parameters to functions on your controller, similar to the View's events hash. Routes can contain parameter parts, `:param`, which match a single URL component between slashes; and splat parts `*splat`, which can match any number of URL components.

For example, a route of `"search/:query/p:page"` will match a fragment of `#search/obama/p2`, passing `"obama"` and `"2"` to the action. A route of `"file/*path"` will match `#file/nested/folder/file.txt`, passing `"nested/folder/file.txt"` to the action.

When the visitor presses the back button, or enters a URL, and a particular route is matched, the name of the action will be fired as an event, so that other objects can listen to the controller, and be notified. In the following example, visiting `#help/uploading` will fire a `route:help` event from the controller.

```
routes: {
  "help/:page":         "help",
  "download/*path":     "download",
  "folder/:name":       "openFolder",
  "folder/:name-:mode": "openFolder"
}
```

```
controller.bind("route:help", function(page) {
  ...
});
```

constructor / initialize `new Controller([options])`

When creating a new controller, you may pass its routes hash directly as an option, if you choose. All `options` will also be passed to your `initialize` function, if defined.

route `controller.route(route, name, callback)`

Manually create a route for the controller, The `route` argument may be a routing string or regular expression. Each matching capture from the route or regular expression will be passed as an argument to the callback. The `name` argument will be triggered as a `"route:name"` event whenever the route is matched.

```
initialize: function(options) {

  // Matches #page/10, passing "10"
  this.route("page/:number", "page", function(number){ ... });

  // Matches /117-a/b/c/open, passing "117-a/b/c"
  this.route(/^(.*?)\/open$/, "open", function(id){ ... });

}
```

saveLocation `controller.saveLocation(fragment)`

Whenever you reach a point in your application that you'd like to save as a URL, call **saveLocation** in order to update the URL fragment without triggering a `hashchange` event. (If you would prefer to trigger the event and routing, you can just set the hash directly.)

```
openPage: function(pageNumber) {
  this.document.pages.at(pageNumber).open();
  this.saveLocation("page/" + pageNumber);
}
```

Backbone.history

History serves as a global router (per frame) to handle `hashchange` events, match the appropriate route, and trigger callbacks. You shouldn't ever have to create one of these yourself — you should use the reference to `Backbone.history` that will be created for you automatically if you make use of Controllers with routes.

start `Backbone.history.start()`

When all of your Controllers have been created, and all of the routes are set up properly, call `Backbone.history.start()` to begin monitoring `hashchange` events, and dispatching routes.

```
$(function(){
  new WorkspaceController();
  new HelpPaneController();
  Backbone.history.start();
});
```

Backbone.sync

Backbone.sync is the function the Backbone calls every time it attempts to read or save a model to the server. By default, it uses `(jQuery/Zepto).ajax` to make a RESTful JSON request. You can override it in order to use a different persistence strategy, such as WebSockets, XML transport, or Local Storage.

The method signature of **Backbone.sync** is `sync(method, model, success, error)`

- **method** – the CRUD method (`"create"`, `"read"`, `"update"`, or `"delete"`)
- **model** – the model to be saved (or collection to be read)
- **success({model: ...})** – a callback that should be fired if the request works
- **error({model: ...})** – a callback that should be fired if the request fails

With the default implementation, when **Backbone.sync** sends up a request to save a model, its attributes will be passed, serialized as JSON, and sent in the HTTP body with content-type `application/json`. When returning a JSON response, send down the attributes of the model that have been changed by the server, and need to be updated on the client. When responding to a `"read"` request from a collection (Collection#fetch), send down an array of model attribute objects.

The default **sync** handler maps CRUD to REST like so:

- **create → POST** `/collection`
- **read → GET** `/collection[/id]`
- **update → PUT** `/collection/id`
- **delete → DELETE** `/collection/id`

As an example, a Rails handler responding to an `"update"` call from `Backbone` might look like this: *(In real code, never use* `update_attributes` *blindly, and always whitelist the attributes you allow to be changed.)*

```
def update
  account = Account.find params[:id]
  account.update_attributes params
  render :json => account
end
```

One more tip for Rails integration is to disable the default namespacing for `to_json` calls on models by setting `ActiveRecord::Base.include_root_in_json = false`

emulateHTTP `Backbone.emulateHTTP = true`

If you want to work with a legacy web server that doesn't support Backbones's default REST/HTTP approach, you may choose to turn on `Backbone.emulateHTTP`. Setting this option will fake `PUT` and `DELETE` requests with a HTTP `POST`, and pass them under the `_method` parameter. Setting this option will also set an `X-HTTP-Method-Override` header with the true method.

```
Backbone.emulateHTTP = true;

model.save();  // POST to "/collection/id", with "_method=PUT" + header.
```

emulateJSON `Backbone.emulateJSON = true`

If you're working with a legacy web server that can't handle requests encoded as `application/json`, setting `Backbone.emulateJSON = true;` will cause the JSON to be serialized under a `model` parameter, and the request to be made with a `application/x-www-form-urlencoded` mime type, as if from an HTML form.

Backbone.View

Backbone views are almost more convention than they are code — they don't determine anything about your HTML or CSS for you, and can be used with any JavaScript templating library. The general idea is to organize your interface into logical views, backed by models, each of which can be updated independently when the model changes, without having to redraw the page. Instead of digging into a JSON object, looking up an element in the DOM, and updating the HTML by hand, you can bind your view's `render` function to the model's `"change"` event — and now everywhere that model data is displayed in the UI, it is always immediately up to date.

extend `Backbone.View.extend(properties, [classProperties])`

Get started with views by creating a custom view class. You'll want to override the render function, specify your declarative events, and perhaps the `tagName`, `className`, or `id` of the View's root element.

```
var DocumentRow = Backbone.View.extend({

  tagName: "li",

  className: "document-row",

  events: {
    "click .icon":          "open",
    "click .button.edit":   "openEditDialog",
    "click .button.delete": "destroy"
  },

  initialize: function() {
    _.bindAll(this, "render");
  },

  render: function() {
    ...
  }

});
```

constructor / initialize `new View([options])`

When creating a new View, the options you pass are attached to the view as `this.options`, for future reference. There are several special options that, if passed, will be attached directly to the view: `model`, `collection`, `el`, `id`, `className`, and `tagName`. If the view defines an **initialize** function, it will be called when the view is first created. If you'd like to create a view that references an element *already* in the DOM, pass in the element as an option: `new View({el: existingElement})`

```
var doc = Documents.first();

new DocumentRow({
  model: doc,
  id: "document-row-" + doc.id
});
```

el `view.el`

All views have a DOM element at all times (the **el** property), whether they've already been inserted into the page or not. In this fashion, views can be rendered at any time, and inserted into the DOM all at once, in order to get high-performance UI rendering with as few reflows and repaints as possible.

`this.el` is created from the view's `tagName`, `className`, and `id` properties, if specified. If not, **el** is an empty `div`.

$ (jQuery or Zepto) `view.$(selector)`

If jQuery or Zepto is included on the page, each view has a **$** function that runs queries scoped within the view's element. If you use this scoped jQuery function, you don't have to use model ids as part of your query to pull out specific elements in a list, and can rely much more on HTML class attributes. It's equivalent to running: `$(selector,` `this.el)`

```
ui.Chapter = Backbone.View.extend({
  serialize : function() {
    return {
      title: this.$(".title").text(),
      start: this.$(".start-page").text(),
      end:   this.$(".end-page").text()
    };
  }
});
```

render `view.render()`

The default implementation of **render** is a no-op. Override this function with your code that renders the view template from model data, and updates `this.el` with the new HTML. A good convention is to `return this` at the end of **render** to enable chained calls.

```
var Bookmark = Backbone.View.extend({
  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  }
});
```

Backbone is agnostic with respect to your preferred method of HTML templating. Your **render** function could even munge together an HTML string, or use `document.createElement` to generate a DOM tree. However, we suggest choosing a nice JavaScript templating library. Mustache.js, Haml-js, and Eco are all fine alternatives. Because Underscore.js is already on the page, _.template is available, and is an excellent choice if you've already XSS-sanitized your interpolated data.

Whatever templating strategy you end up with, it's nice if you *never* have to put strings of HTML in your JavaScript. At DocumentCloud, we use Jammit in order to package up JavaScript templates stored in `/app/views` as part of our main `core.js` asset package.

remove `view.remove()`

Convenience function for removing the view from the DOM. Equivalent to calling `$(view.el).remove();`

make `view.make(tagName, [attributes], [content])`

Convenience function for creating a DOM element of the given type (**tagName**), with optional attributes and HTML content. Used internally to create the initial `view.el`.

```
var view = new Backbone.View;

var el = view.make("b", {className: "bold"}, "Bold! ");

$("#make-demo").append(el);
```

delegateEvents `delegateEvents([events])`

Uses jQuery's `delegate` function to provide declarative callbacks for DOM events within a view. If an **events** hash is not passed directly, uses `this.events` as the source. Events are written in the format `{"event selector": "callback"}`. Omitting the `selector` causes the event to be bound to the view's root element (`this.el`). By default, `delegateEvents` is called within the View's constructor for you, so if you have a simple `events` hash, all of your DOM events will always already be connected, and you will never have to call this function yourself.

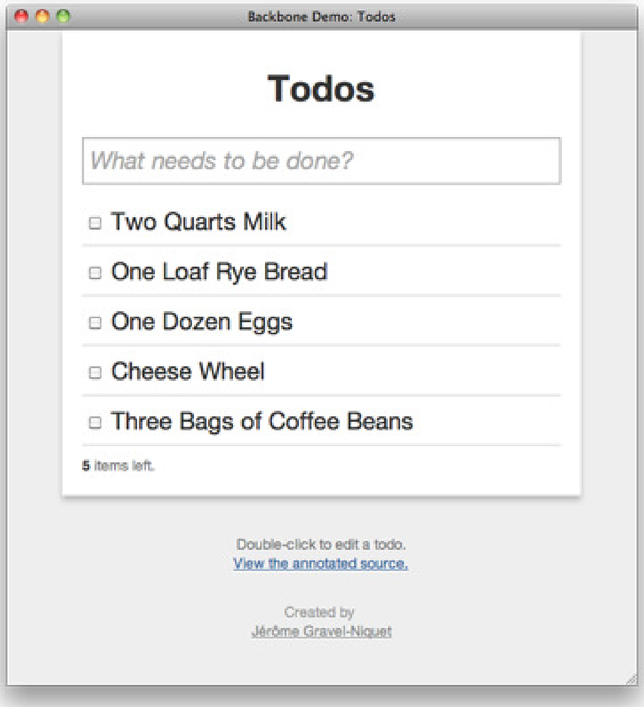
Using **delegateEvents** provides a number of advantages over manually using jQuery to bind events to child elements during render. All attached callbacks are bound to the view before being handed off to jQuery, so when the callbacks are invoked, `this` continues to refer to the view object. When **delegateEvents** is run again, perhaps with a different `events` hash, all callbacks are removed and delegated afresh — useful for views which need to behave differently when in different modes.

A view that displays a document in a search result might look something like this:

```
var DocumentView = Backbone.View.extend({

  events: {
    "dblclick"                : "open",
    "click .icon.doc"         : "select",
    "contextmenu .icon.doc"   : "showMenu",
    "click .show_notes"       : "toggleNotes",
    "click .title .lock"      : "editAccessLevel",
    "mouseover .title .date"  : "showTooltip"
  },

  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  },

  open: function() {
    window.open(this.model.get("viewer_url"));
  },

  select: function() {
    this.model.set({selected: true});
  },

  ...

});
```

Examples

Jérôme Gravel-Niquet has contributed a Todo List application that is bundled in the repository as Backbone example. If you're wondering where to get started with Backbone in general, take a moment to read through the annotated source. The app uses a LocalStorage adapter to transparently save all of your todos within your browser, instead of sending them to a server. Jérôme also has a version hosted at localtodos.com that uses a MooTools-backed version of Backbone instead of jQuery.



Todos

What needs to be done?

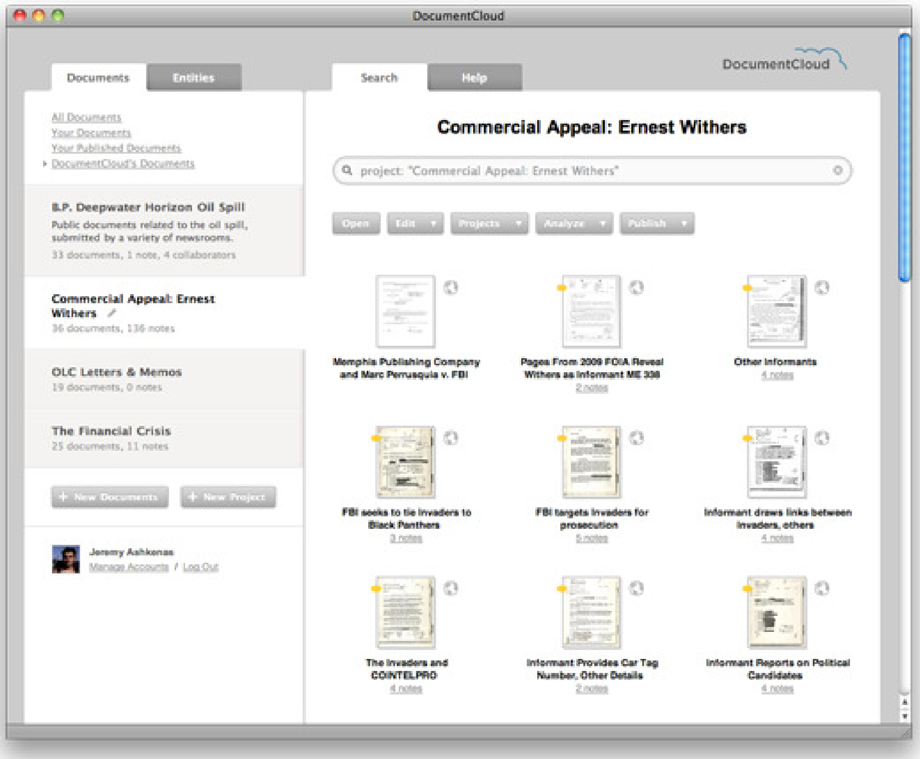
- Two Quarts Milk
- One Loaf Rye Bread
- One Dozen Eggs
- Cheese Wheel
- Three Bags of Coffee Beans

5 items left.

Double-click to edit a todo.
View the annotated source.

Created by
Jérôme Gravel-Niquet

The DocumentCloud workspace is built on Backbone.js, with *Documents*, *Projects*, *Notes*, and *Accounts* all as Backbone models and collections.



All Documents
Your Documents
Your Published Documents
DocumentCloud's Documents

B.P. Deepwater Horizon Oil Spill
Public documents related to the oil spill, submitted by a variety of newsrooms.
33 documents, 1 note, 4 collaborators

Commercial Appeal: Ernest Withers
36 documents, 136 notes

OLC Letters & Memos
19 documents, 0 notes

The Financial Crisis
25 documents, 11 notes

Jeremy Ashkenas
Manage Accounts / Log Out

Commercial Appeal: Ernest Withers

project: "Commercial Appeal: Ernest Withers"

Memphis Publishing Company and Marc Perrusquia v. FBI

Pages From 2009 FOIA Reveal Withers as Informant ME 338
2 notes

Other Informants
4 notes

FBI seeks to tie Invaders to Black Panthers
3 notes

FBI targets Invaders for prosecution
5 notes

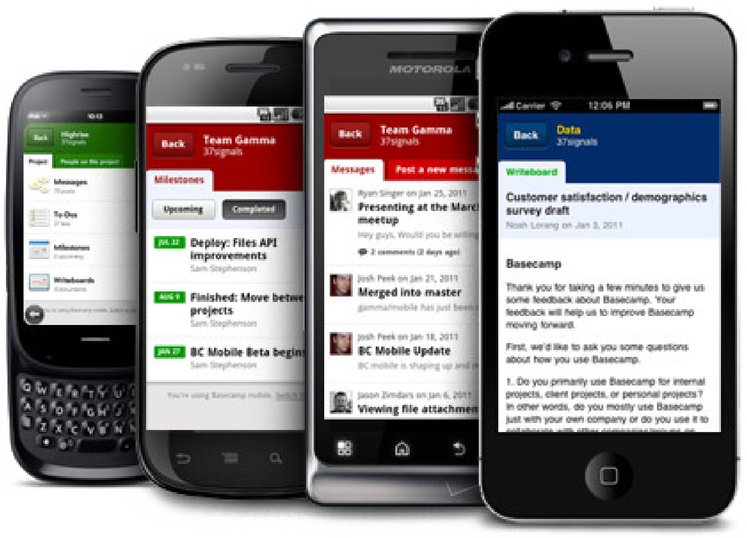
Informant draws links between Invaders, others
4 notes

The Invaders and COINTELPRO
4 notes

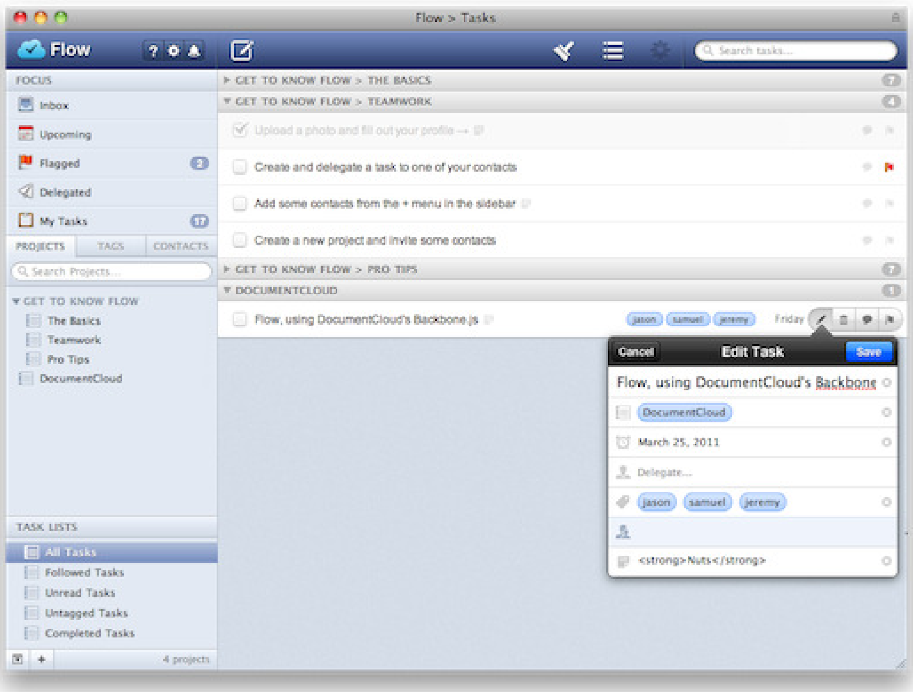
Informant Provides Car Tag Number, Other Details
2 notes

Informant Reports on Political Candidates
4 notes

37Signals used Backbone.js to create Basecamp Mobile, the mobile version of their popular project management software. You can access all your Basecamp projects, post new messages, and comment on milestones (all represented internally as Backbone.js models).



MetaLab used Backbone.js to create Flow, a task management app for teams. The workspace relies on Backbone.js to construct task views, activities, accounts, folders, projects, and tags. You can see the internals under `window.Flow`.



Upload a photo and fill out your profile →

Create and delegate a task to one of your contacts

Add some contacts from the + menu in the sidebar

Create a new project and invite some contacts

Flow, using DocumentCloud's Backbone.js jason samuel jeremy Friday

Flow, using DocumentCloud's Backbone

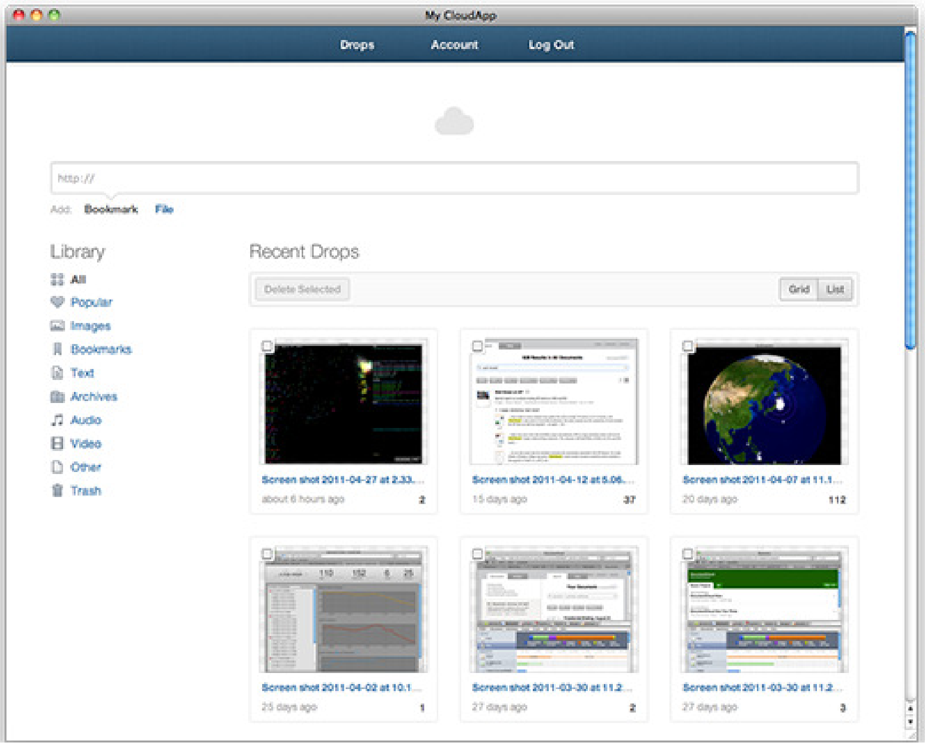
DocumentCloud

March 25, 2011

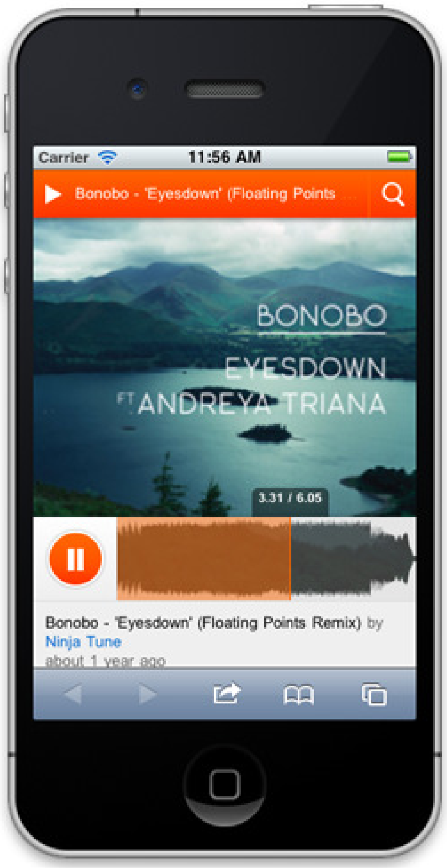
jason samuel jeremy

<strong>Nuts</strong>

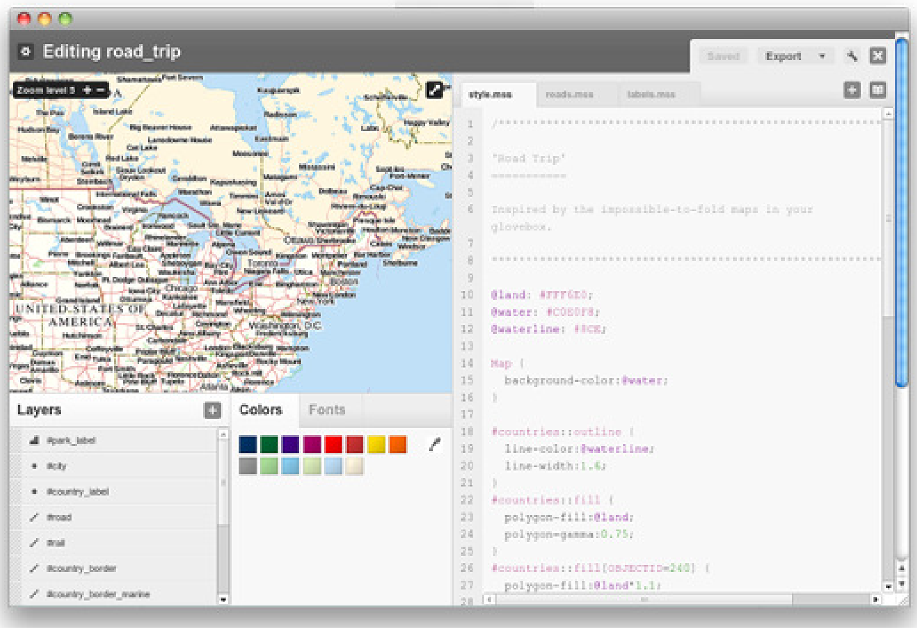
CloudApp is simple file and link sharing for the Mac. Backbone.js powers the web tools which consume the documented API to manage Drops. Data is either pulled manually or pushed by Pusher and fed to Mustache templates for rendering. Check out the annotated source code to see the magic.



SoundCloud is the best sound sharing platform on the internet, and Backbone.js provides the foundation for Mobile SoundCloud. The project uses the public SoundCloud API as a data source (channeled through a nginx proxy), jQuery templates for the rendering, Qunit and PhantomJS for the testing suite. The JS code, templates and CSS are built for the production deployment with various Node.js tools like ready.js, Jake, jsdom. The **Backbone.History** was modified to support the HTML5 `history.pushState`. **Backbone.sync** was extended with an additional SessionStorage based cache layer.

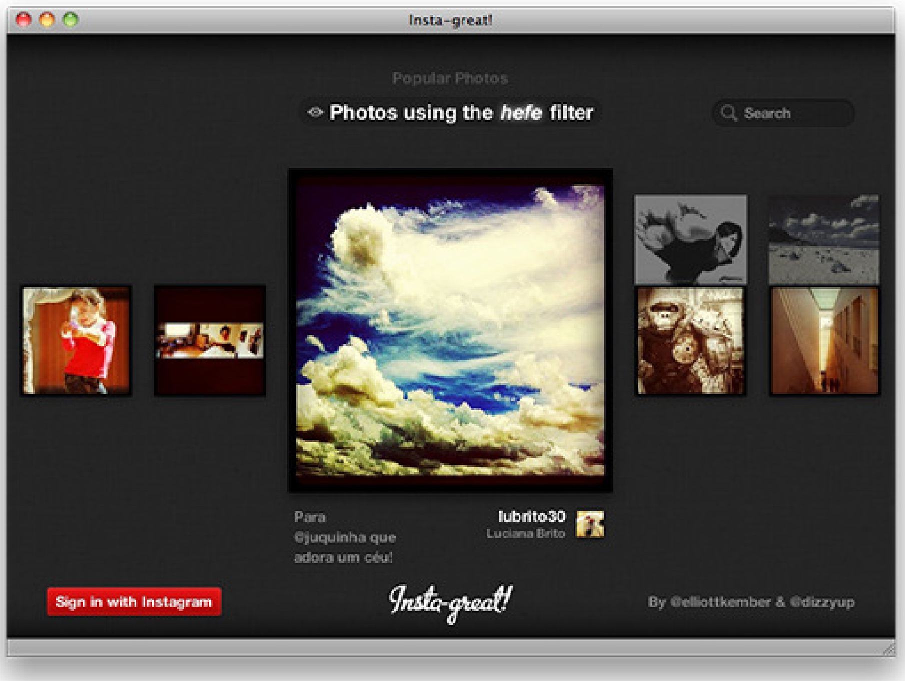

Carrier
11:56 AM
Bonobo - 'Eyesdown' (Floating Points ...
BONOBO
EYESDOWN
FT ANDREYA TRIANA
3.31 / 6.05
Bonobo - 'Eyesdown' (Floating Points Remix) by
Ninja Tune
about 1 year ago

Our fellow Knight Foundation News Challenge winners, MapBox, created an open-source map design studio with Backbone.js: TileMill. TileMill lets you manage map layers based on shapefiles and rasters, and edit their appearance directly in the browser with the Carto styling language.

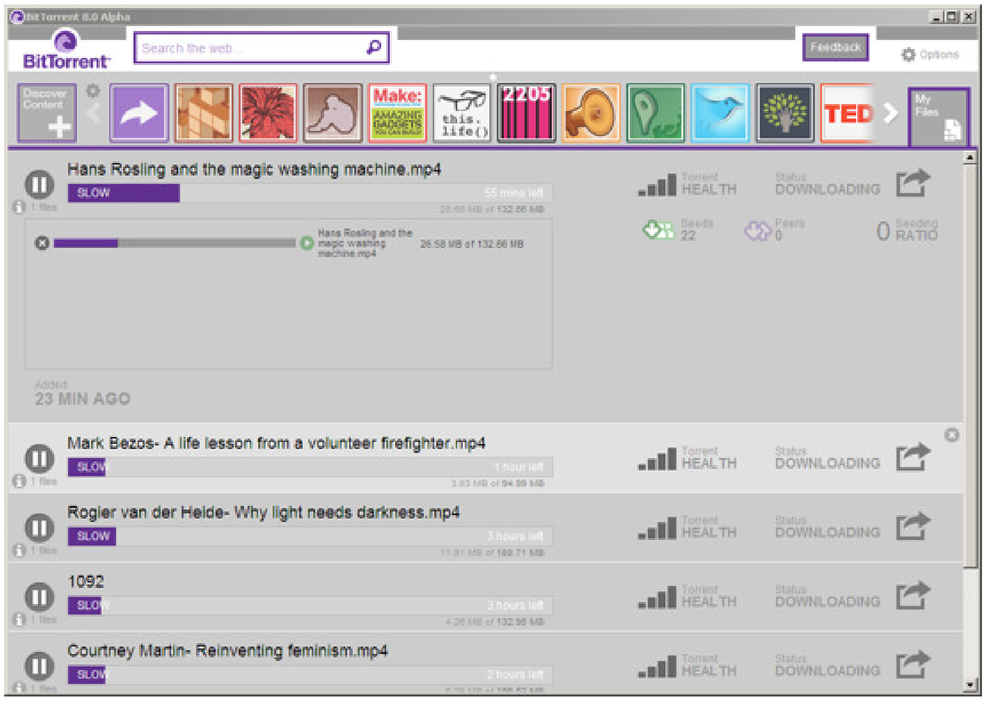


```
/*****************************************************

'Road Trip'
===========

Inspired by the impossible-to-fold maps in your
glovebox.

*****************************************************

@land: #FFF6E0;
@water: #C0E0F8;
@waterline: #8CE;

Map {
  background-color:@water;
}

#countries::outline {
  line-color:@waterline;
  line-width:1.6;
}
#countries::fill {
  polygon-fill:@land;
  polygon-gamma:0.75;
}
#countries::fill[OBJECTID=240] {
  polygon-fill:@land*1.1;
```

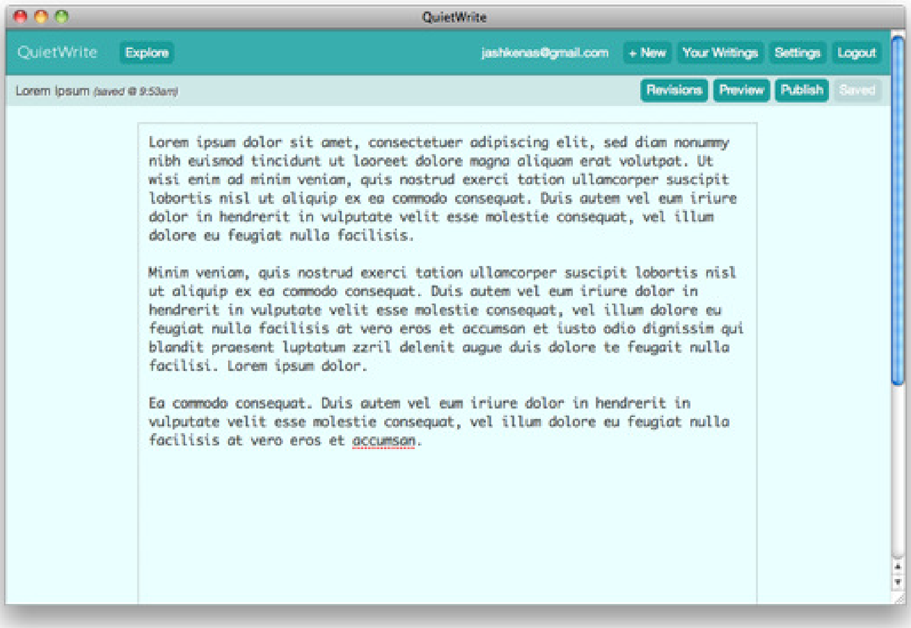
Elliott Kember and Hector Simpson built Insta-great! - a fun way to explore popular photos and interact with Instagram on the web. Elliott says, "Backbone.js and Coffeescript were insanely useful for writing clean, consistent UI code and keeping everything modular and readable, even through several code refactors. I'm in love."


Insta-great!
Popular Photos
Photos using the *hefe* filter
Search
Para @juquinha que adora um céu!
lubrito30
Luciana Brito
Sign in with Instagram
Insta-great!
By @elliottkember & @dizzyup

BitTorrent used Backbone to completely rework an existing Win32 UI. Models normalize access to the client's data and views rely heavily on the `change` events to keep the UI state current. Using Backbone and SCSS, our new design and UX prototypes are considerably easier to iterate, test and work with than the original Win32 UI.


BitTorrent 8.0 Alpha
BitTorrent
Search the web...
Feedback
Options
Hans Rosling and the magic washing machine.mp4
SLOW
55 mins left
23 MIN AGO
Mark Bezos- A life lesson from a volunteer firefighter.mp4
Rogier van der Heide- Why light needs darkness.mp4
1092
Courtney Martin- Reinventing feminism.mp4
Torrent HEALTH
Status DOWNLOADING
Seeds 22
Peers 0
0 Seeding RATIO

[James Yu](#) used Backbone.js to create [QuietWrite](#), an app that gives writers a clean and quiet interface to concentrate on the text itself. The editor relies on Backbone to persist document data to the server. He followed up with a Backbone.js + Rails tutorial that describes how to implement [CloudEdit, a simple document editing app](#).

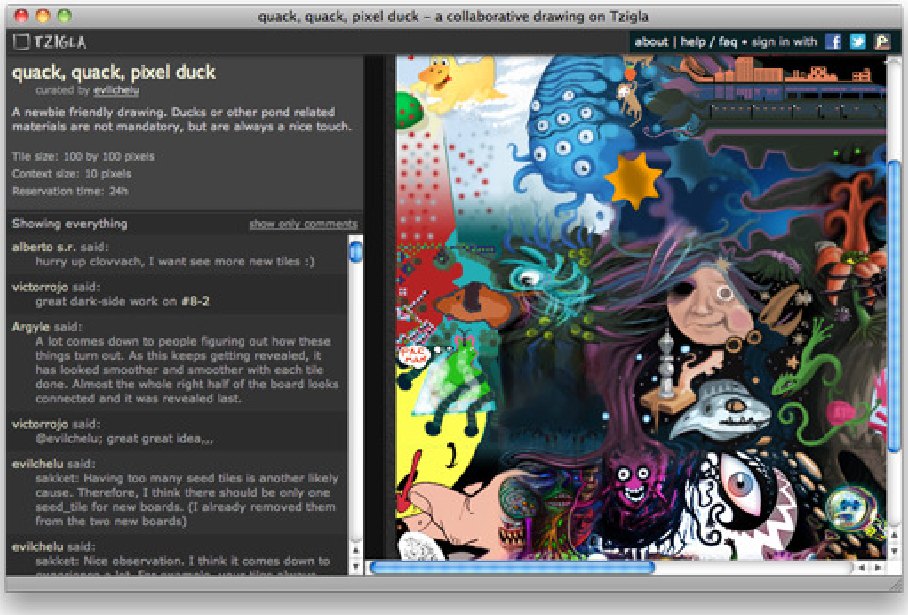


Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

Minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor.

Ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan.

[Cristi Balan](#) and [Irina Dumitrascu](#) created [Tzigla](#), a collaborative drawing application where artists make tiles that connect to each other to create [surreal drawings](#). Backbone models help organize the code, controllers provide [bookmarkable deep links](#), and the views are rendered with [haml.js](#) and [Zepto](#). Tzigla is written in Ruby (Rails) on the backend, and [CoffeeScript](#) on the frontend, with [Jammit](#) prepackaging the static assets.


quack, quack, pixel duck – a collaborative drawing on Tzigla
TZIGLA
about | help / faq • sign in with

quack, quack, pixel duck
curated by evilchelu

A newbie friendly drawing. Ducks or other pond related materials are not mandatory, but are always a nice touch.

Tile size: 100 by 100 pixels
Context size: 10 pixels
Reservation time: 24h

Showing everything show only comments

alberto s.r. said:
hurry up clovvach, I want see more new tiles :)

victorrojo said:
great dark-side work on #8-2

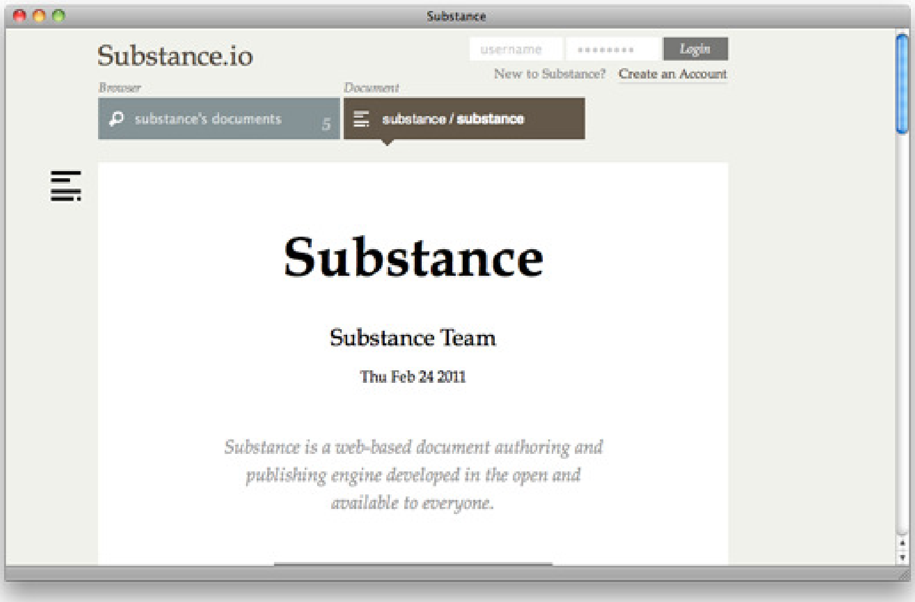
Argyle said:
A lot comes down to people figuring out how these things turn out. As this keeps getting revealed, it has looked smoother and smoother with each tile done. Almost the whole right half of the board looks connected and it was revealed last.

victorrojo said:
@evilchelu; great great idea,,,

evilchelu said:
sakket: Having too many seed tiles is another likely cause. Therefore, I think there should be only one seed_tile for new boards. (I already removed them from the two new boards)

evilchelu said:
sakket: Nice observation. I think it comes down to

Michael Aufreiter is building an open source document authoring and publishing engine: Substance. Substance makes use of Backbone.View and Backbone.Controller, while Backbone plays well together with Data.js, which is used for data persistence.



Substance

Substance Team

Thu Feb 24 2011

Substance is a web-based document authoring and publishing engine developed in the open and available to everyone.

F.A.Q.

Catalog of Events

Here's a list of all of the built-in events that Backbone.js can fire. You're also free to trigger your own events on Models and Views as you see fit.

- **"add"** (model, collection) — when a model is added to a collection.
- **"remove"** (model, collection) — when a model is removed from a collection.
- **"refresh"** (collection) — when the collection's entire contents have been replaced.
- **"change"** (model, collection) — when a model's attributes have changed.
- **"change:[attribute]"** (model, collection) — when a specific attribute has been updated.
- **"error"** (model, collection) — when a model's validation fails, or a save call fails on the server.
- **"route:[name]"** (controller) — when one of a controller's routes has matched.
- **"all"** — this special event fires for *any* triggered event, passing the event name as the first argument.

Nested Models & Collections

It's common to nest collections inside of models with Backbone. For example, consider a `Mailbox` model that contains many `Message` models. One nice pattern for handling this is have a `this.messages` collection for each mailbox, enabling the lazy-loading of messages, when the mailbox is first opened ... perhaps with `MessageList` views listening for `"add"` and `"remove"` events.

```
var Mailbox = Backbone.Model.extend({

  initialize: function() {
    this.messages = new Messages;
    this.messages.url = '/mailbox/' + this.id + '/messages';
    this.messages.bind("refresh", this.updateCounts);
  },

  ...

});

var Inbox = new Mailbox;

// And then, when the Inbox is opened:

Inbox.messages.fetch();
```

Loading Bootstrapped Models

When your app first loads, it's common to have a set of initial models that you know you're going to need, in order to render the page. Instead of firing an extra AJAX request to fetch them, a nicer pattern is to have their data already bootstrapped into the page. You can then use refresh to populate your collections with the initial data. At DocumentCloud, in the ERB template for the workspace, we do something along these lines:

```
<script>
  Accounts.refresh(<%= @accounts.to_json %>);
  Projects.refresh(<%= @projects.to_json(:collaborators => true) %>);
</script>
```

How does Backbone relate to "traditional" MVC?

Different implementations of the Model-View-Controller pattern tend to disagree about the definition of a controller. If it helps any, in Backbone, the View class can also be thought of as a kind of controller, dispatching events that originate from the UI, with the HTML template serving as the true view. We call it a View because it represents a logical chunk of UI, responsible for the contents of a single DOM element.

Comparing the overall structure of Backbone to a server-side MVC framework like **Rails**, the pieces line up like so:

- **Backbone.Model** – Like a Rails model minus the class methods. Wraps a row of data in business logic.
- **Backbone.Collection** – A group of models on the client-side, with sorting/filtering/aggregation logic.
- **Backbone.Controller** – Rails `routes.rb` + Rails controller actions. Maps URLs to functions.
- **Backbone.View** – A logical, re-usable piece of UI. Often, but not always, associated with a model.
- **Client-side Templates** – Rails `.html.erb` views, rendering a chunk of HTML.

Binding "this"

Perhaps the single most common JavaScript "gotcha" is the fact that when you pass a function as a callback, it's value for `this` is lost. With Backbone, when dealing with events and callbacks, you'll often find it useful to rely on _.bind and _.bindAll from Underscore.js. `_.bind` takes a function and an object to be used as `this`, any time the function is called in the future. `_.bindAll` takes an object and a list of method names: each method in the list will be bound to the object, so that it's `this` may not change. For example, in a View that listens for changes to a collection...

```
var MessageList = Backbone.View.extend({

  initialize: function() {
    _.bindAll(this, "addMessage", "removeMessage", "render");

    var messages = this.collection;
    messages.bind("refresh", this.render);
    messages.bind("add", this.addMessage);
    messages.bind("remove", this.removeMessage);
  }

});

// Later, in the app...

Inbox.messages.add(newMessage);
```

How is Backbone different than SproutCore or Cappuccino?

This question is frequently asked, and all three projects apply general Model-View-Controller principles to JavaScript applications. However, there isn't much basis for comparison. SproutCore and Cappuccino provide rich UI widgets, vast core libraries, and determine the structure of your HTML for you. Both frameworks measure in the hundreds of kilobytes when packed and gzipped, and megabytes of JavaScript, CSS, and images when loaded in the browser — there's a lot of room underneath for libraries of a more moderate scope. Backbone is a *4 kilobyte* include that provides just the core concepts of models, events, collections, views, controllers, and persistence.

Change Log

0.3.3 — *Dec 1, 2010*

Backbone.js now supports Zepto, alongside jQuery, as a framework for DOM manipulation and Ajax support. Implemented Model#escape, to efficiently handle attributes intended for HTML interpolation. When trying to persist a model, failed requests will now trigger an `"error"` event. The ubiquitous `options` argument is now passed as the final argument to all `"change"` events.

0.3.2 — *Nov 23, 2010*

Bugfix for IE7 + iframe-based "hashchange" events. `sync` may now be overridden on a per-model, or per-collection basis. Fixed recursion error when calling `save` with no changed attributes, within a `"change"` event.

0.3.1 — *Nov 15, 2010*

All `"add"` and `"remove"` events are now sent through the model, so that views can listen for them without having to know about the collection. Added a `remove` method to Backbone.View. `toJSON` is no longer called at all for `'read'` and `'delete'` requests. Backbone routes are now able to load empty URL fragments.

0.3.0 — *Nov 9, 2010*

Backbone now has Controllers and History, for doing client-side routing based on URL fragments. Added `emulateHTTP` to provide support for legacy servers that don't do `PUT` and `DELETE`. Added `emulateJSON` for servers that can't accept `application/json` encoded requests. Added Model#clear, which removes all attributes from a model. All Backbone classes may now be seamlessly inherited by CoffeeScript classes.

0.2.0 — *Oct 25, 2010*

Instead of requiring server responses to be namespaced under a `model` key, now you can define your own parse method to convert responses into attributes for Models and Collections. The old `handleEvents` function is now named delegateEvents, and is automatically called as part of the View's constructor. Added a toJSON function to Collections. Added Underscore's chain to Collections.

0.1.2 — *Oct 19, 2010*

Added a Model#fetch method for refreshing the attributes of single model from the server. An `error` callback may now be passed to `set` and `save` as an option, which will be invoked if validation fails, overriding the `"error"` event. You can now tell backbone to use the `_method` hack instead of HTTP methods by setting `Backbone.emulateHTTP = true`. Existing Model and Collection data is no longer sent up unnecessarily with `GET` and `DELETE` requests. Added a `rake lint` task. Backbone is now published as an NPM module.

0.1.1 — *Oct 14, 2010*

Added a convention for `initialize` functions to be called upon instance construction, if defined. Documentation tweaks.

0.1.0 — *Oct 13, 2010*

Initial Backbone release.

A DocumentCloud Project