BACKBONE.JS

Backbone.js gives structure to web 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 API 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, an example application, a list of tutorials and a long list of real-world projects that use Backbone. Backbone is available for use under the MIT software license.

You can report bugs and discuss features on the GitHub issues page, on Freenode IRC in the `#documentcloud` channel, post questions to the Google Group, add pages to the wiki or send tweets to @documentcloud.

Backbone is an open-source component of DocumentCloud.

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

Development Version (1.1.2)	*60kb, Full source, tons of comments*
Production Version (1.1.2)	*6.5kb, Packed and gzipped* (Source Map)
Edge Version (master)	*Unreleased, use at your own risk* build passing

Backbone's only hard dependency is Underscore.js (>= 1.5.0). For RESTful persistence, history support via Backbone.Router and DOM manipulation with Backbone.View, include **jQuery**, and **json2.js** for older Internet Explorer support. *(Mimics of the Underscore and jQuery APIs, such as Lo-Dash and Zepto, will also tend to work, with varying degrees of compatibility.)*

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 often 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 state can be notified of the change, so that they are able to respond accordingly, re-rendering themselves with the new information. In a finished Backbone app, 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.

Philosophically, Backbone is an attempt to discover the minimal set of data-structuring (models and collections) and user interface (views and URLs) primitives that are generally useful when building web applications with JavaScript. In an ecosystem where overarching, decides-everything-for-you frameworks are commonplace, and many libraries require your site to be reorganized to suit their look, feel, and default behavior — Backbone should continue to be a tool that gives you the *freedom* to design the full experience of your web application.

If you're new here, and aren't yet quite sure what Backbone is for, start by browsing the list of Backbone-based projects.

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

Upgrading to 1.1

Backbone **1.1** should be a fairly painless upgrade from the **0.9.X** series. If you're upgrading from an older version, be sure to check out the change log. In brief, a few of the larger breaking changes are:

○ If you want to smartly update the contents of a Collection, adding new models, removing missing ones, and merging those already present, you now call set (previously named "update"), a similar operation to calling `set` on a Model. This is now the default when you call fetch on a collection. To get the old behavior, pass `{reset: true}`.

○ If you have characters in your URL segments that require URL encoding, Backbone will now decode them for you (normalizing the behavior cross-browser) before your route handlers receive them as arguments.

○ In **0.9.x**, Backbone events gained two new methods: listenTo and stopListening, which make it easier to create Views that have all of their observers unbound when you want to remove the view.

○ Model validation is now only enforced by default in save — not in set unless the `{validate:true}` option is passed. Model validation now fires an `"invalid"` event instead of `"error"`.

○ In 1.1, Backbone Views no longer have the `options` argument attached as `this.options` automatically. Feel free to continue attaching it if you like.

○ In 1.1, The **Collection** methods `add`, `remove`, `set`, `push`, and `shift` now return the

model (or models) added or removed from the collection.

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

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

For example, to make a handy event dispatcher that can coordinate events among different areas of your application: `var dispatcher = _.clone(Backbone.Events)`

on   `object.on(event, callback, [context])`   *Alias: bind*

Bind a **callback** function to an object. The callback will be invoked whenever the **event** 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"` . The event string may also be a space-delimited list of several events...

```
book.on("change:title change:author", ...);
```

To supply a **context** value for `this` when the callback is invoked, pass the optional third argument: `model.on('change', this.render, this)`

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

All Backbone event methods also support an event map syntax, as an alternative to positional arguments:

```
book.on({
  "change:title": titleView.update,
  "change:author": authorPane.update,
  "destroy": bookView.remove
});
```

off   `object.off([event], [callback], [context])`   *Alias: unbind*

Remove a previously-bound **callback** function from an object. If no **context** is specified, all of the versions of the callback with different contexts will be removed. If no callback is specified, all callbacks for the **event** will be removed. If no event is specified, callbacks for *all* events will be removed.

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

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

// Removes the `onChange` callback for all events.
object.off(null, onChange);

// Removes all callbacks for `context` for all events.
object.off(null, null, context);

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

Note that calling `model.off()` , for example, will indeed remove *all* events on the model — including events that Backbone uses for internal bookkeeping.

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

Trigger callbacks for the given **event**, or space-delimited list of events. Subsequent arguments to **trigger** will be passed along to the event callbacks.

once   `object.once(event, callback, [context])`

Just like on, but causes the bound callback to only fire once before being removed. Handy for saying "the next time that X happens, do this".

listenTo   `object.listenTo(other, event, callback)`

Tell an **object** to listen to a particular event on an **other** object. The advantage of using this form, instead of `other.on(event, callback, object)` , is that **listenTo** allows the **object** to keep track of the events, and they can be removed all at once later on. The **callback** will always be called with **object** as context.

```
view.listenTo(model, 'change', view.render);
```

stopListening   `object.stopListening([other], [event], [callback])`

Tell an **object** to stop listening to events. Either call **stopListening** with no arguments to have the **object** remove all of its registered callbacks ... or be more precise by telling it to remove just the events it's listening to on a specific object, or a specific event, or just a specific callback.

```
view.stopListening();
```

```
view.stopListening(model);
```

listenToOnce   `object.listenToOnce(other, event, callback)`

Just like listenTo, but causes the bound callback to only fire once before being removed.

Catalog of Events

Here's the complete list of built-in Backbone events, with arguments. You're also free to trigger your own events on Models, Collections and Views as you see fit. The `Backbone` object itself mixes in `Events`, and can be used to emit any global events that your application needs.

○ **"add"** (model, collection, options) — when a model is added to a collection.

○ **"remove"** (model, collection, options) — when a model is removed from a collection.

○ **"reset"** (collection, options) — when the collection's entire contents have been replaced.

○ **"sort"** (collection, options) — when the collection has been re-sorted.

○ **"change"** (model, options) — when a model's attributes have changed.

○ **"change:[attribute]"** (model, value, options) — when a specific attribute has been updated.

○ **"destroy"** (model, collection, options) — when a model is destroyed.

○ **"request"** (model_or_collection, xhr, options) — when a model or collection has started a request to the server.

○ **"sync"** (model_or_collection, resp, options) — when a model or collection has been successfully synced with the server.

○ **"error"** (model_or_collection, resp, options) — when model's or collection's request to remote server has failed.

○ **"invalid"** (model, error, options) — when a model's validation fails on the client.

○ **"route:[name]"** (params) — Fired by the router when a specific route is matched.

○ **"route"** (route, params) — Fired by the router when *any* route has been matched.

○ **"route"** (router, route, params) — Fired by history when *any* route has been matched.

○ **"all"** — this special event fires for *any* triggered event, passing the event name as the first argument.

Generally speaking, when calling a function that emits an event (`model.set`, `collection.add`, and so on...), if you'd like to prevent the event from being triggered, you may pass `{silent: true}` as an option. Note that this is *rarely*, perhaps even never, a good idea. Passing through a specific flag in the options for your event callback to look at, and choose to ignore, will usually work out better.

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 domain-specific 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.on('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.apply(this, arguments);
    ...
  }
});
```

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

When creating an instance of a model, you can pass in the initial values of the **attributes**, which will be <u>set</u> 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"
});
```

In rare cases, if you're looking to get fancy, you may want to override **constructor**, which allows you to replace the actual constructor function for your model.

```
var Library = Backbone.Model.extend({
  constructor: function() {
    this.books = new Books();
    Backbone.Model.apply(this, arguments);
  },
  parse: function(data, options) {
    this.books.reset(data.books);
    return data.library;
  }
});
```

If you pass a `{collection: ...}` as the **options**, the model gains a `collection` property that will be used to indicate which collection the model belongs to, and is used to help compute the model's <u>url</u>. The `model.collection` property is normally created automatically when you first add a model to a collection. Note that the reverse is not true, as passing this option to the constructor will not automatically add the model to the collection. Useful, sometimes.

If `{parse: true}` is passed as an **option**, the **attributes** will first be converted by <u>parse</u> before being <u>set</u> on the model.

get   `model.get(attribute)`

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

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

Set a hash of attributes (one or many) on the model. If any of the attributes change the

model's state, a `"change"` event will be triggered on the model. Change events for specific attributes are also triggered, and you can bind to those as well, for example: `change:title`, and `change:content`. You may also pass individual keys and values.

```
note.set({title: "March 20", content: "In his eyes she eclipses..."});

book.set("title", "A Scandal in Bohemia");
```

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'));
```

has   `model.has(attribute)`

Returns `true` if the attribute is set to a non-null or non-undefined value.

```
if (note.has("title")) {
  ...
}
```

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, including the `id` attribute. 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.

idAttribute   `model.idAttribute`

A model's unique identifier is stored under the `id` attribute. If you're directly communicating with a backend (CouchDB, MongoDB) that uses a different unique key, you may set a Model's `idAttribute` to transparently map from that key to `id`.

```
var Meal = Backbone.Model.extend({
  idAttribute: "_id"
});

var cake = new Meal({ _id: 1, name: "Cake" });
alert("Cake id: " + cake.id);
```

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.

attributes   `model.attributes`

The **attributes** property is the internal hash containing the model's state — usually (but
not necessarily) a form of the JSON object representing the model data on the server.
It's often a straightforward serialization of a row from the database, but it could also be
client-side computed 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 `_.clone(model.attributes)`
instead.

Due to the fact that Events accepts space separated lists of events, attribute names should not include
spaces.

changed   `model.changed`

The **changed** property is the internal hash containing all the attributes that have
changed since the last set. Please do not update **changed** directly since its state is
internally maintained by set. A copy of **changed** can be acquired from
changedAttributes.

defaults   `model.defaults or model.defaults()`

The **defaults** hash (or function) 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'));
```

Remember that in JavaScript, objects are passed by reference, so if you include an object as a default

*value, it will be shared among all instances. Instead, define **defaults** as a function.*

toJSON   `model.toJSON([options])`

Return a shallow copy of the model's attributes for JSON stringification. This can be used for persistence, serialization, or for augmentation before being sent to the server. 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));
```

sync   `model.sync(method, model, [options])`

Uses Backbone.sync to persist the state of a model to the server. Can be overridden for custom behavior.

fetch   `model.fetch([options])`

Resets the model's state from the server by delegating to Backbone.sync. Returns a jqXHR. 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 both passed `(model, response, options)` as arguments.

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

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

Save a model to your database (or alternative persistence layer), by delegating to Backbone.sync. Returns a jqXHR if validation is successful and `false` otherwise. The **attributes** hash (as in set) should contain the attributes you'd like to change — keys that aren't mentioned won't be altered — but, a *complete representation* of the resource will be sent to the server. As with `set`, you may pass individual keys and values instead of a hash. 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`).

If instead, you'd only like the *changed* attributes to be sent to the server, call `model.save(attrs, {patch: true})`. You'll get an HTTP `PATCH` request to the server with just the passed-in attributes.

Calling `save` with new attributes will cause a `"change"` event immediately, a `"request"` event as the Ajax request begins to go to the server, and a `"sync"` event after the server has acknowledged the successful change. Pass `{wait: true}` if you'd like to wait for the server before setting the new attributes on the model.

In the following example, notice how our overridden version of `Backbone.sync` receives a `"create"` request the first time the model is saved and an `"update"` request the second time.

```
Backbone.sync = function(method, model) {
  alert(method + ": " + JSON.stringify(model));
  model.set('id', 1);
};

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

book.save();

book.save({author: "Teddy"});
```

save accepts `success` and `error` callbacks in the options hash, which will be passed the arguments `(model, response, options)`. If a server-side validation fails, return a non-`200` HTTP response code, along with an error response in text or JSON.

```
book.save("author", "F.D.R.", {error: function(){ ... }});
```

destroy   `model.destroy([options])`

Destroys the model on the server by delegating an HTTP `DELETE` request to Backbone.sync. Returns a jqXHR object, or `false` if the model isNew. Accepts `success` and `error` callbacks in the options hash, which will be passed `(model, response, options)`. Triggers a `"destroy"` event on the model, which will bubble up through any collections that contain it, a `"request"` event as it begins the Ajax request to the server, and a `"sync"` event, after the server has successfully acknowledged the model's deletion. Pass `{wait: true}` if you'd like to wait for the server to respond before removing the model from the collection.

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

Underscore Methods (6)

Backbone proxies to **Underscore.js** to provide 6 object functions on **Backbone.Model**. They aren't all documented here, but you can take a look at the Underscore documentation for the full details…

○ keys

- values

- pairs

- invert

- pick

- omit

```
user.pick('first_name', 'last_name', 'email');

chapters.keys().join(', ');
```

validate   `model.validate(attributes, options)`

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. By default **validate** is called before `save`, but can also be called before `set` if `{validate:true}` is passed. The **validate** method is passed the model attributes, as well as the options from `set` or `save`. If the attributes are valid, don't return anything from **validate**; if they 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. If **validate** returns an error, `save` will not continue, and the model attributes will not be modified on the server. Failed validations trigger an `"invalid"` event, and set the `validationError` property on the model with the value returned by this method.

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

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

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

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

`"invalid"` events are useful for providing coarse-grained error messages at the model or collection level.

validationError   `model.validationError`

The value returned by validate during the last failed validation.

isValid   `model.isValid()`

Run <u>validate</u> to check the model state.

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

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

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

if (!one.isValid()) {
  alert(one.get("title") + " " + one.validationError);
}
```

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.url]/[id]"` by default, but you may override by specifying an explicit `urlRoot` if the model's collection shouldn't be taken into account.

Delegates to <u>Collection#url</u> to generate the URL, so make sure that you have it defined, or a <u>urlRoot</u> property, if all models of this class share a common root URL. A model with an id of `101`, stored in a <u>Backbone.Collection</u> with a `url` of `"/documents/7/notes"`, would have this URL: `"/documents/7/notes/101"`

urlRoot   `model.urlRoot or model.urlRoot()`

Specify a `urlRoot` if you're using a model *outside* of a collection, to enable the default <u>url</u> function to generate URLs based on the model id. `"[urlRoot]/id"`
Normally, you won't need to define this. Note that `urlRoot` may also be a function.

```
var Book = Backbone.Model.extend({urlRoot : '/books'});

var solaris = new Book({id: "1083-lem-solaris"});

alert(solaris.url());
```

parse   `model.parse(response, options)`

parse is called whenever a model's data is returned by the server, in <u>fetch</u>, and <u>save</u>. The function is passed the raw `response` object, and should return the attributes hash

to be <u>set</u> 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 that has a version prior to 3.1, you'll notice that its 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.

hasChanged   `model.hasChanged([attribute])`

Has the model changed since the last <u>set</u>? If an **attribute** is passed, returns `true` if that specific attribute has changed.

Note that this method, and the following change-related ones, are only useful during the course of a `"change"` *event.*

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

changedAttributes   `model.changedAttributes([attributes])`

Retrieve a hash of only the model's attributes that have changed since the last <u>set</u>, or `false` if there are none. 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.on("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 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.

Any event that is triggered on a model in a collection will also be triggered on the collection directly, for convenience. This allows you to listen for changes to specific attributes in any model in a collection, for example:

`documents.on("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 reset, and the attributes will be converted into a model of the proper type.

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

A collection can also contain polymorphic models by overriding this property with a constructor that returns a model.

```
var Library = Backbone.Collection.extend({

  model: function(attrs, options) {
    if (condition) {
      return new PublicDocument(attrs, options);
    } else {
      return new PrivateDocument(attrs, options);
    }
  }
```

```
});
```

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

When creating a Collection, you may choose to pass in the initial array of **models**. The collection's comparator may be included as an option. Passing `false` as the comparator option will prevent sorting. If you define an **initialize** function, it will be invoked when the collection is created. There are a couple of options that, if provided, are attached to the collection directly: `model` and `comparator`.

```
var tabs = new TabSet([tab1, tab2, tab3]);
var spaces = new Backbone.Collection([], {
  model: Space
});
```

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([options])

Return an array containing the attributes hash of each model (via toJSON) 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));
```

sync   collection.sync(method, collection, [options])

Uses Backbone.sync to persist the state of a collection to the server. Can be overridden for custom behavior.

Underscore Methods (28)

Backbone proxies to **Underscore.js** to provide 28 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 (collect)

○ reduce (foldl, inject)

○ reduceRight (foldr)

```
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, firing an `"add"` event. If a model
property is defined, you may also pass raw attributes objects, and have them be vivified
as instances of the model. Returns the added (or preexisting, if duplicate) models. Pass

`{at: index}` to splice the model into the collection at the specified `index` . If you're adding models to the collection that are *already* in the collection, they'll be ignored, unless you pass `{merge: true}` , in which case their attributes will be merged into the corresponding models, firing any appropriate `"change"` events.

```
var ships = new Backbone.Collection;

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

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

Note that adding the same model (a model with the same `id` *) to a collection more than once is a no-op.*

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

Remove a model (or an array of models) from the collection, and returns them. Fires a `"remove"` event, which you can use `silent` to suppress. The model's index before removal is available to listeners as `options.index` .

reset   `collection.reset([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 **reset** to replace a collection with a new list of models (or attribute hashes), triggering a single `"reset"` event at the end. Returns the newly-set models. For convenience, within a `"reset"` event, the list of any previous models is available as `options.previousModels` .

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

```
<script>
  var accounts = new Backbone.Collection;
  accounts.reset(<%= @accounts.to_json %>);
</script>
```

Calling `collection.reset()` without passing any models as arguments will empty the entire collection.

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

The **set** method performs a "smart" update of the collection with the passed list of models. If a model in the list isn't yet in the collection it will be added; if the model is already in the collection its attributes will be merged; and if the collection contains any models that *aren't* present in the list, they'll be removed. All of the appropriate `"add"` ,

`"remove"` , and `"change"` events are fired as this happens. Returns the touched models in the collection. If you'd like to customize the behavior, you can disable it with options: `{add: false}` , `{remove: false}` , or `{merge: false}` .

```
var vanHalen = new Backbone.Collection([eddie, alex, stone, roth]);

vanHalen.set([eddie, alex, stone, hagar]);

// Fires a "remove" event for roth, and an "add" event for "hagar".
// Updates any of stone, alex, and eddie's attributes that may have
// changed over the years.
```

get   `collection.get(id)`

Get a model from a collection, specified by an id, a cid, or by passing in a **model**.

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

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.

push   `collection.push(model, [options])`

Add a model at the end of a collection. Takes the same options as add.

pop   `collection.pop([options])`

Remove and return the last model from a collection. Takes the same options as remove.

unshift   `collection.unshift(model, [options])`

Add a model at the beginning of a collection. Takes the same options as add.

shift   `collection.shift([options])`

Remove and return the first model from a collection. Takes the same options as remove.

slice   `collection.slice(begin, end)`

Return a shallow copy of this collection's models, using the same options as native Array#slice.

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** for 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` . A comparator can be defined as a sortBy (pass a function that takes a single argument), as a sort (pass a comparator function that expects two arguments), or as a string indicating the attribute to sort by.

"sortBy" comparator functions take a model and return a numeric or string value by which the model should be ordered relative to others. "sort" comparator functions take two models, and return `-1` if the first model should come before the second, `0` if they are of the same rank and `1` if the first model should come after. *Note that Backbone depends on the arity of your comparator function to determine between the two styles, so be careful if your comparator function is bound.*

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 = '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'));
```

Collections with a comparator will not automatically re-sort if you later change model attributes, so you may wish to call `sort` *after changing model attributes that would affect the order.*

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 will sort itself whenever a model is added. To disable sorting when adding a model, pass `{sort: false}` to `add` . Calling **sort** triggers a `"sort"` event on the collection.

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([
  {name: "Curly"},
  {name: "Larry"},
  {name: "Moe"}
]);

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

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

where   `collection.where(attributes)`

Return an array of all the models in a collection that match the passed **attributes**. Useful for simple cases of `filter`.

```
var friends = new Backbone.Collection([
  {name: "Athos",      job: "Musketeer"},
  {name: "Porthos",    job: "Musketeer"},
  {name: "Aramis",     job: "Musketeer"},
  {name: "d'Artagnan", job: "Guard"},
]);

var musketeers = friends.where({job: "Musketeer"});

alert(musketeers.length);
```

findWhere   `collection.findWhere(attributes)`

Just like where, but directly returns only the first model in the collection that matches the passed **attributes**.

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, options)`

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;
  }
});
```

clone   `collection.clone()`

Returns a new instance of the collection with an identical list of models.

fetch   `collection.fetch([options])`

Fetch the default set of models for this collection from the server, setting them on the collection when they arrive. The **options** hash takes `success` and `error` callbacks which will both be passed `(collection, response, options)` as arguments. When the model data returns from the server, it uses set to (intelligently) merge the fetched models, unless you pass `{reset: true}`, in which case the collection will be (efficiently) reset. Delegates to Backbone.sync under the covers for custom persistence strategies and returns a jqXHR. 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();
```

The behavior of **fetch** can be customized by using the available set options. For example, to fetch a collection, getting an `"add"` event for every new model, and a `"change"` event for every changed existing model, without removing anything: `collection.fetch({remove: false})`

jQuery.ajax options can also be passed directly as **fetch** options, so to fetch a specific page of a paginated collection: `Documents.fetch({data: {page: 3}})`

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.

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 new model. If client-side validation failed, the model will be unsaved, with validation errors. In order for this to work, you should set the model property of the collection. The **create** method can accept either an attributes hash or an existing, unsaved model object.

Creating a model will cause an immediate `"add"` event to be triggered on the collection, a `"request"` event as the new model is sent to the server, as well as a `"sync"` event, once the server has responded with the successful creation of the model. Pass `{wait: true}` if you'd like to wait for the server before adding the new model to the collection.

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

var nypl = new Library;

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

Backbone.Router

Web applications often provide linkable, bookmarkable, shareable URLs for important locations in the app. Until recently, hash fragments (`#page`) were used to provide these permalinks, but with the arrival of the History API, it's now possible to use standard URLs (`/page`). **Backbone.Router** provides methods for routing client-side pages, and connecting them to actions and events. For browsers which don't yet support the History API, the Router handles graceful fallback and transparent translation to the fragment version of the URL.

During page load, after your application has finished creating all of its routers, be sure to call `Backbone.history.start()` , or `Backbone.history.start({pushState: true})` to route the initial URL.

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

Get started by creating a custom router class. Define actions that are triggered when certain URL fragments are matched, and provide a routes hash that pairs routes to actions. Note that you'll want to avoid using a leading slash in your route definitions:

```
var Workspace = Backbone.Router.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   `router.routes`

The routes hash maps URLs with parameters to functions on your router (or just direct

function definitions, if you prefer), 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. Part of a route can be made optional by surrounding it in parentheses `(/:optional)`.

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.

A route of `"docs/:section(/:subsection)"` will match `#docs/faq` and `#docs/faq/installing`, passing `"faq"` to the action in the first case, and passing `"faq"` and `"installing"` to the action in the second.

Trailing slashes are treated as part of the URL, and (correctly) treated as a unique route when accessed. `docs` and `docs/` will fire different callbacks. If you can't avoid generating both types of URLs, you can define a `"docs(/)"` matcher to capture both cases.

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 router, and be notified. In the following example, visiting `#help/uploading` will fire a `route:help` event from the router.

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

```
router.on("route:help", function(page) {
  ...
});
```

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

When creating a new router, 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   `router.route(route, name, [callback])`

Manually create a route for the router, 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. If the `callback` argument is omitted `router[name]` will be used instead. Routes added later may override previously declared routes.

```
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" to this.open
  this.route(/^(.*?)\/open$/, "open");

},

open: function(id) { ... }
```

navigate   `router.navigate(fragment, [options])`

Whenever you reach a point in your application that you'd like to save as a URL, call
navigate in order to update the URL. If you wish to also call the route function, set the
trigger option to `true`. To update the URL without creating an entry in the browser's
history, set the **replace** option to `true`.

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

# Or ...

app.navigate("help/troubleshooting", {trigger: true});

# Or ...

app.navigate("help/troubleshooting", {trigger: true, replace: true});
```

execute   `router.execute(callback, args)`

This method is called internally within the router, whenever a route matches and its
corresponding **callback** is about to be executed. Override it to perform custom parsing
or wrapping of your routes, for example, to parse query strings before handing them to
your route callback, like so:

```
var Router = Backbone.Router.extend({
  execute: function(callback, args) {
    args.push(parseQueryString(args.pop()));
    if (callback) callback.apply(this, args);
  }
});
```

Backbone.history

History serves as a global router (per frame) to handle `hashchange` events or
`pushState`, match the appropriate route, and trigger callbacks. You shouldn't ever have
to create one of these yourself since `Backbone.history` already contains one.

pushState support exists on a purely opt-in basis in Backbone. Older browsers that don't support `pushState` will continue to use hash-based URL fragments, and if a hash URL is visited by a `pushState`-capable browser, it will be transparently upgraded to the true URL. Note that using real URLs requires your web server to be able to correctly render those pages, so back-end changes are required as well. For example, if you have a route of `/documents/100`, your web server must be able to serve that page, if the browser visits that URL directly. For full search-engine crawlability, it's best to have the server generate the complete HTML for the page ... but if it's a web application, just rendering the same content you would have for the root URL, and filling in the rest with Backbone Views and JavaScript works fine.

start   `Backbone.history.start([options])`

When all of your Routers have been created, and all of the routes are set up properly, call `Backbone.history.start()` to begin monitoring `hashchange` events, and dispatching routes. Subsequent calls to `Backbone.history.start()` will throw an error, and `Backbone.History.started` is a boolean value indicating whether it has already been called.

To indicate that you'd like to use HTML5 `pushState` support in your application, use `Backbone.history.start({pushState: true})`. If you'd like to use `pushState`, but have browsers that don't support it natively use full page refreshes instead, you can add `{hashChange: false}` to the options.

If your application is not being served from the root url `/` of your domain, be sure to tell History where the root really is, as an option: `Backbone.history.start({pushState: true, root: "/public/search/"})`

When called, if a route succeeds with a match for the current URL, `Backbone.history.start()` returns `true`. If no defined route matches the current URL, it returns `false`.

If the server has already rendered the entire page, and you don't want the initial route to trigger when starting History, pass `silent: true`.

Because hash-based history in Internet Explorer relies on an `<iframe>`, be sure to only call `start()` after the DOM is ready.

```
$(function(){
  new WorkspaceRouter();
  new HelpPaneRouter();
  Backbone.history.start({pushState: true});
});
```

Backbone.sync

Backbone.sync is the function that Backbone calls every time it attempts to read or save a model to the server. By default, it uses `jQuery.ajax` to make a RESTful JSON

request and returns a jqXHR. 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, [options])`

- **method** – the CRUD method (`"create"`, `"read"`, `"update"`, or `"delete"`)
- **model** – the model to be saved (or collection to be read)
- **options** – success and error callbacks, and all other jQuery request options

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.

Whenever a model or collection begins a **sync** with the server, a `"request"` event is emitted. If the request completes successfully you'll get a `"sync"` event, and an `"error"` event if not.

The **sync** function may be overriden globally as `Backbone.sync`, or at a finer-grained level, by adding a `sync` function to a Backbone collection or to an individual model.

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 integrating Rails versions prior to 3.1 is to disable the default namespacing for `to_json` calls on models by setting `ActiveRecord::Base.include_root_in_json = false`

ajax   `Backbone.ajax = function(request) { ... };`

If you want to use a custom AJAX function, or your endpoint doesn't support the

jQuery.ajax API and you need to tweak things, you can do so by setting `Backbone.ajax`.

emulateHTTP `Backbone.emulateHTTP = true`

If you want to work with a legacy web server that doesn't support Backbone'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`, setting the `X-HTTP-Method-Override` header with the true method. If `emulateJSON` is also on, the true method will be passed as an additional `_method` parameter.

```
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() {
    this.listenTo(this.model, "change", this.render);
  },

  render: function() {
    ...
  }

});
```

Properties like `tagName`, `id`, `className`, `el`, and `events` may also be defined as a function, if you want to wait to define them until runtime.

constructor / initialize   new View([options])

There are several special options that, if passed, will be attached directly to the view:
`model`, `collection`, `el`, `id`, `className`, `tagName`, `attributes` and `events`. 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`, `id` and `attributes` properties, if specified. If not, **el** is an empty `div`.

```
var ItemView = Backbone.View.extend({
  tagName: 'li'
});

var BodyView = Backbone.View.extend({
  el: 'body'
});

var item = new ItemView();
var body = new BodyView();

alert(item.el + ' ' + body.el);
```

$el   view.$el

A cached jQuery object for the view's element. A handy reference instead of re-wrapping
the DOM element all the time.

```
view.$el.show();

listView.$el.append(itemView.el);
```

setElement   view.setElement(element)

If you'd like to apply a Backbone view to a different DOM element, use **setElement**,
which will also create the cached `$el` reference and move the view's delegated events
from the old element to the new one.

attributes   view.attributes

A hash of attributes that will be set as HTML DOM element attributes on the view's `el`
(id, class, data-properties, etc.), or a function that returns such a hash.

$ (jQuery)   view.$(selector)

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

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

template   view.template([data])

While templating for a view isn't a function provided directly by Backbone, it's often a
nice convention to define a **template** function on your views. In this way, when rendering
your view, you have convenient access to instance data. For example, using Underscore
templates:

```
var LibraryView = Backbone.View.extend({
  template: _.template(...)
});
```

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({
  template: _.template(...),
```

```
      render: function() {
        this.$el.html(this.template(this.model.attributes));
        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 prefer simple interpolated-JavaScript style templates.

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()`

Removes a view from the DOM, and calls stopListening to remove any bound events
that the view has listenTo'd.

delegateEvents   `delegateEvents([events])`

Uses jQuery's `on` 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"}`. The callback may be
either the name of a method on the view, or a direct function body. 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.

The `events` property may also be defined as a function that returns an **events** hash, to
make it easier to programmatically define your events, as well as inherit them from
parent views.

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.attributes));
  return this;
},

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

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

...

});
```

undelegateEvents   undelegateEvents()

Removes all of the view's delegated events. Useful if you want to disable or remove a view from the DOM temporarily.

Utility

Backbone.noConflict   var backbone = Backbone.noConflict();

Returns the `Backbone` object back to its original value. You can use the return value of `Backbone.noConflict()` to keep a local reference to Backbone. Useful for embedding Backbone on third-party websites, where you don't want to clobber the existing Backbone.

```
var localBackbone = Backbone.noConflict();
var model = localBackbone.Model.extend(...);
```

Backbone.$ Backbone.$ = $;

If you have multiple copies of `jQuery` on the page, or simply want to tell Backbone to use a particular object as its DOM / Ajax library, this is the property for you.

F.A.Q.

Why use Backbone, not [other framework X]?

If your eye hasn't already been caught by the adaptability and elan on display in the above list of examples, we can get more specific: Backbone.js aims to provide the common foundation that data-rich web applications with ambitious interfaces require — while very deliberately avoiding painting you into a corner by making any decisions that you're better equipped to make yourself.

- The focus is on supplying you with helpful methods to manipulate and query your data, not on HTML widgets or reinventing the JavaScript object model.

- Backbone does not force you to use a single template engine. Views can bind to HTML constructed in your favorite way.

- It's smaller. There are fewer kilobytes for your browser or phone to download, and less *conceptual* surface area. You can read and understand the source in an afternoon.

- It doesn't depend on stuffing application logic into your HTML. There's no embedded JavaScript, template logic, or binding hookup code in `data-` or `ng-` attributes, and no need to invent your own HTML tags.

- Synchronous events are used as the fundamental building block, not a difficult-to-reason-abou run loop, or by constantly polling and traversing your data structures to hunt for changes. And you want a specific event to be asynchronous and aggregated, no problem.

- Backbone scales well, from embedded widgets to massive apps.

- Backbone is a library, not a framework, and plays well with others. You can embed Backbone widgets in Dojo apps without trouble, or use Backbone models as the data backing for D3 visualizations (to pick two entirely random examples).

- "Two way data-binding" is avoided. While it certainly makes for a nifty demo, and works for the most basic CRUD, it doesn't tend to be terribly useful in your real-world app. Sometimes you want to update on every keypress, sometimes on blur, sometimes when the panel is closed, and sometimes when the "save" button is clicked. In almost all cases, simply serializing the form to JSON is faster and easier. All that aside, if your heart is set, go for it.

- There's no built-in performance penalty for choosing to structure your code with Backbone. An if you do want to optimize further, thin models and templates with flexible granularity make it easy to squeeze every last drop of potential performance out of, say, IE8.

There's More Than One Way To Do It

It's common for folks just getting started to treat the examples listed on this page as some sort of gospel truth. In fact, Backbone.js is intended to be fairly agnostic about many common patterns in client-side code. For example...

References between Models and Views can be handled several ways. Some people like to have direct pointers, where views correspond 1:1 with models (`model.view` and `view.model`). Others prefer to have intermediate "controller" objects that orchestrate the creation and organization of views into a hierarchy. Others still prefer the evented approach, and always fire events instead of calling methods directly. All of these styles work well.

Batch operations on Models are common, but often best handled differently depending on your server-side setup. Some folks don't mind making individual Ajax requests. Others create explicit resources for RESTful batch operations: `/notes/batch /destroy?ids=1,2,3,4`. Others tunnel REST over JSON, with the creation of "changeset" requests:

```
  {
    "create":  [array of models to create]
    "update":  [array of models to update]
    "destroy": [array of model ids to destroy]
  }
```

Feel free to define your own events. Backbone.Events is designed so that you can mix it in to any JavaScript object or prototype. Since you can use any string as an event, it's often handy to bind and trigger your own custom events: `model.on("selected:true")` or `model.on("editing")`

Render the UI as you see fit. Backbone is agnostic as to whether you use Underscore templates, Mustache.js, direct DOM manipulation, server-side rendered snippets of HTML, or jQuery UI in your `render` function. Sometimes you'll create a view for each model ... sometimes you'll have a view that renders thousands of models at once, in a tight loop. Both can be appropriate in the same app, depending on the quantity of data involved, and the complexity of the UI.

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.on("reset", this.updateCounts);
  },

  ...

});

var inbox = new Mailbox;

// And then, when the Inbox is opened:

inbox.messages.fetch({reset: true});
```

If you're looking for something more opinionated, there are a number of Backbone plugins that add sophisticated associations among models, available on the wiki.

Backbone doesn't include direct support for nested models and collections or "has many" associations because there are a number of good patterns for modeling structured data on the client side, and *Backbone should provide the foundation for implementing any of them.* You may want to…

○ Mirror an SQL database's structure, or the structure of a NoSQL database.

○ Use models with arrays of "foreign key" ids, and join to top level collections (a-la tables).

○ For associations that are numerous, use a range of ids instead of an explicit list.

○ Avoid ids, and use direct references, creating a partial object graph representing your data set

○ Lazily load joined models from the server, or lazily deserialize nested models from JSON documents.

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 reset to populate your collections with the initial data. At DocumentCloud, in the ERB template for the workspace, we do something along these lines:

```
<script>
  var accounts = new Backbone.Collection;
  accounts.reset(<%= @accounts.to_json %>);
  var projects = new Backbone.Collection;
  projects.reset(<%= @projects.to_json(:collaborators => true) %>);
</script>
```

You have to escape `</` within the JSON string, to prevent javascript injection attacks.

Extending Backbone

Many JavaScript libraries are meant to be insular and self-enclosed, where you interact with them by calling their public API, but never peek inside at the guts. Backbone.js is *not* that kind of library.

Because it serves as a foundation for your application, you're meant to extend and enhance it in the ways you see fit — the entire source code is annotated to make this easier for you. You'll find that there's very little there apart from core functions, and most of those can be overriden or augmented should you find the need. If you catch yourself adding methods to `Backbone.Model.prototype`, or creating your own base subclass, don't worry — that's how things are supposed to work.

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.Router** – 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, its value for `this` is lost. When dealing with events and callbacks in Backbone, you'll often find it useful to rely on listenTo or the optional `context` argument that many of Underscore and Backbone's methods use to specify the `this` that will be used when the callback is later invoked. (See _.each, _.map, and object.on, to name a few). View events are automatically bound to the view's context for you. You may also find it helpful to use _.bind and _.bindAll from Underscore.js.

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

  initialize: function() {
    var messages = this.collection;
    messages.on("reset", this.render, this);
    messages.on("add", this.addMessage, this);
    messages.on("remove", this.removeMessage, this);

    messages.each(this.addMessage, this);
  }

});

// Later, in the app...

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

Working with Rails

Backbone.js was originally extracted from a Rails application; getting your client-side (Backbone) Models to sync correctly with your server-side (Rails) Models is painless, but there are still a few things to be aware of.

By default, Rails versions prior to 3.1 add an extra layer of wrapping around the JSON representation of models. You can disable this wrapping by setting:

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

... in your configuration. Otherwise, override parse to pull model attributes out of the wrapper. Similarly, Backbone PUTs and POSTs direct JSON representations of models, where by default Rails expects namespaced attributes. You can have your controllers filter attributes directly from `params`, or you can override toJSON in Backbone to add

the extra wrapping Rails expects.

Examples

The list of examples that follows, while long, is not exhaustive. If you've worked on an app that uses Backbone, please add it to the wiki page of Backbone apps.

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.

<p align="center">Todos</p>

DocumentCloud

The DocumentCloud workspace is built on Backbone.js, with *Documents*, *Projects*, *Notes*, and *Accounts* all as Backbone models and collections. If you're interested in history — both Underscore.js and Backbone.js were originally extracted from the DocumentCloud codebase, and packaged into standalone JS libraries.

<p align="center">DocumentCloud Workspace</p>

USA Today

USA Today takes advantage of the modularity of Backbone's data/model lifecycle — which makes it simple to create, inherit, isolate, and link application objects — to keep the codebase both manageable and efficient. The new website also makes heavy use of the Backbone Router to control the page for both pushState-capable and legacy browsers. Finally, the team took advantage of Backbone's Event module to create a PubSub API that allows third parties and analytics packages to hook into the heart of the app.

<p align="center">USA Today</p>

Rdio

New Rdio was developed from the ground up with a component based framework based on Backbone.js. Every component on the screen is dynamically loaded and rendered, with data provided by the Rdio API. When changes are pushed, every component can update itself without reloading the page or interrupting the user's music. All of this relies on Backbone's views and models, and all URL routing is handled by Backbone's Router.

When data changes are signaled in realtime, Backbone's Events notify the interested components in the data changes. Backbone forms the core of the new, dynamic, realtime Rdio web and *desktop* applications.

<div align="center">Rdio</div>

Hulu

Hulu used Backbone.js to build its next generation online video experience. With Backbone as a foundation, the web interface was rewritten from scratch so that all page content can be loaded dynamically with smooth transitions as you navigate. Backbone makes it easy to move through the app quickly without the reloading of scripts and embedded videos, while also offering models and collections for additional data manipulation support.

<div align="center">Hulu</div>

Quartz

Quartz sees itself as a digitally native news outlet for the new global economy. Because Quartz believes in the future of open, cross-platform web applications, they selected Backbone and Underscore to fetch, sort, store, and display content from a custom WordPress API. Although qz.com uses responsive design for phone, tablet, and desktop browsers, it also takes advantage of Backbone events and views to render device-specific templates in some cases.

<div align="center">Quartz</div>

Earth

Earth.nullschool.net displays real-time weather conditions on an interactive animated globe, and Backbone provides the foundation upon which all of the site's components are built. Despite the presence of several other javascript libraries, Backbone's non-opinionated design made it effortless to mix-in the Events functionality used for distributing state changes throughout the page. When the decision was made to switch to Backbone, large blocks of custom logic simply disappeared.

<div align="center">Earth</div>

Vox

Vox Media, the publisher of SB Nation, The Verge, Polygon, Eater, Racked, Curbed, and Vox.com, uses Backbone throughout Chorus, its home-grown publishing platform. Backbone powers the liveblogging platform and commenting system used across all Vox Media properties; Coverage, an internal editorial coordination tool; SB Nation Live, a live

event coverage and chat tool; and Vox Cards, Vox.com's highlighter-and-index-card inspired app for providing context about the news.

Vox

Gawker Media

Kinja is Gawker Media's publishing platform designed to create great stories by breaking down the lines between the traditional roles of content creators and consumers. Everyone — editors, readers, marketers — have access to the same tools to engage in passionate discussion and pursue the truth of the story. Sharing, recommending, and following within the Kinja ecosystem allows for improved information discovery across all the sites.

Kinja is the platform behind Gawker, Gizmodo, Lifehacker, io9 and other Gawker Media blogs. Backbone.js underlies the front-end application code that powers everything from user authentication to post authoring, commenting, and even serving ads. The JavaScript stack includes Underscore.js and jQuery, with some plugins, all loaded with RequireJS. Closure templates are shared between the Play! Framework based Scala application and Backbone views, and the responsive layout is done with the Foundation framework using SASS.

Gawker

Flow

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`.

Flow

Gilt Groupe

Gilt Groupe uses Backbone.js to build multiple applications across their family of sites. Gilt's mobile website uses Backbone and Zepto.js to create a blazing-fast shopping experience for users on-the-go, while Gilt Live combines Backbone with WebSockets to display the items that customers are buying in real-time. Gilt's search functionality also uses Backbone to filter and sort products efficiently by moving those actions to the client-side.

Gilt Groupe

Enigma

Enigma is a portal amassing the largest collection of public data produced by governments, universities, companies, and organizations. Enigma uses Backbone Models and Collections to represent complex data structures; and Backbone's Router gives Enigma users unique URLs for application states, allowing them to navigate quickly through the site while maintaining the ability to bookmark pages and navigate forward and backward through their session.

Enigma

NewsBlur

NewsBlur is an RSS feed reader and social news network with a fast and responsive UI that feels like a native desktop app. Backbone.js was selected for a major rewrite and transition from spaghetti code because of its powerful yet simple feature set, easy integration, and large community. If you want to poke around under the hood, NewsBlur is also entirely open-source.

Newsblur

WordPress.com

WordPress.com is the software-as-a-service version of WordPress. It uses Backbone.js Models, Collections, and Views in its Notifications system. Backbone.js was selected because it was easy to fit into the structure of the application, not the other way around. Automattic (the company behind WordPress.com) is integrating Backbone.js into the Stats tab and other features throughout the homepage.

WordPress.com Notifications

Foursquare

Foursquare is a fun little startup that helps you meet up with friends, discover new places, and save money. Backbone Models are heavily used in the core JavaScript API layer and Views power many popular features like the homepage map and lists.

Foursquare

Bitbucket

Bitbucket is a free source code hosting service for Git and Mercurial. Through its models and collections, Backbone.js has proved valuable in supporting Bitbucket's REST API, as well as newer components such as in-line code comments and approvals for pull requests. Mustache templates provide server and client-side rendering, while a custom Google Closure inspired life-cycle for widgets allows Bitbucket to decorate existing DOM trees and insert new ones.

Bitbucket

Disqus

Disqus chose Backbone.js to power the latest version of their commenting widget. Backbone's small footprint and easy extensibility made it the right choice for Disqus' distributed web application, which is hosted entirely inside an iframe and served on thousands of large web properties, including IGN, Wired, CNN, MLB, and more.

Disqus

Delicious

Delicious is a social bookmarking platform making it easy to save, sort, and store bookmarks from across the web. Delicious uses Chaplin.js, Backbone.js and AppCache to build a full-featured MVC web app. The use of Backbone helped the website and mobile apps share a single API service, and the reuse of the model tier made it significantly easier to share code during the recent Delicious redesign.

Delicious

Khan Academy

Khan Academy is on a mission to provide a free world-class education to anyone anywhere. With thousands of videos, hundreds of JavaScript-driven exercises, and big plans for the future, Khan Academy uses Backbone to keep frontend code modular and organized. User profiles and goal setting are implemented with Backbone, jQuery and Handlebars, and most new feature work is being pushed to the client side, greatly increasing the quality of the API.

Khan Academy

IRCCloud

IRCCloud is an always-connected IRC client that you use in your browser — often leaving it open all day in a tab. The sleek web interface communicates with an Erlang backend via websockets and the IRCCloud API. It makes heavy use of Backbone.js events, models, views and routing to keep your IRC conversations flowing in real time.

IRCCloud

Pitchfork

Pitchfork uses Backbone.js to power its site-wide audio player, Pitchfork.tv, location

routing, a write-thru page fragment cache, and more. Backbone.js (and Underscore.js) helps the team create clean and modular components, move very quickly, and focus on the site, not the spaghetti.

Pitchfork

Spin

Spin pulls in the latest news stories from their internal API onto their site using Backbone models and collections, and a custom `sync` method. Because the music should never stop playing, even as you click through to different "pages", Spin uses a Backbone router for navigation within the site.

Spin

ZocDoc

ZocDoc helps patients find local, in-network doctors and dentists, see their real-time availability, and instantly book appointments. On the public side, the webapp uses Backbone.js to handle client-side state and rendering in search pages and doctor profiles. In addition, the new version of the doctor-facing part of the website is a large single-page application that benefits from Backbone's structure and modularity. ZocDoc's Backbone classes are tested with Jasmine, and delivered to the end user with Cassette.

ZocDoc

Walmart Mobile

Walmart used Backbone.js to create the new version of their mobile web application and created two new frameworks in the process. Thorax provides mixins, inheritable events, as well as model and collection view bindings that integrate directly with Handlebars templates. Lumbar allows the application to be split into modules which can be loaded on demand, and creates platform specific builds for the portions of the web application that are embedded in Walmart's native Android and iOS applications.

Walmart Mobile

Groupon Now!

Groupon Now! helps you find local deals that you can buy and use right now. When first developing the product, the team decided it would be AJAX heavy with smooth transitions between sections instead of full refreshes, but still needed to be fully linkable and shareable. Despite never having used Backbone before, the learning curve was incredibly quick — a prototype was hacked out in an afternoon, and the team was able

to ship the product in two weeks. Because the source is minimal and understandable, it was easy to add several Backbone extensions for Groupon Now!: changing the router to handle URLs with querystring parameters, and adding a simple in-memory store for caching repeated requests for the same data.

<div align="center">Groupon Now!</div>

Basecamp

37Signals chose Backbone.js to create the calendar feature of its popular project management software Basecamp. The Basecamp Calendar uses Backbone.js models and views in conjunction with the Eco templating system to present a polished, highly interactive group scheduling interface.

<div align="center">Basecamp Calendar</div>

Slavery Footprint

Slavery Footprint allows consumers to visualize how their consumption habits are connected to modern-day slavery and provides them with an opportunity to have a deeper conversation with the companies that manufacture the goods they purchased. Based in Oakland, California, the Slavery Footprint team works to engage individuals, groups, and businesses to build awareness for and create deployable action against forced labor, human trafficking, and modern-day slavery through online tools, as well as off-line community education and mobilization programs.

<div align="center">Slavery Footprint</div>

Stripe

Stripe provides an API for accepting credit cards on the web. Stripe's management interface was recently rewritten from scratch in CoffeeScript using Backbone.js as the primary framework, Eco for templates, Sass for stylesheets, and Stitch to package everything together as CommonJS modules. The new app uses Stripe's API directly for the majority of its actions; Backbone.js models made it simple to map client-side models to their corresponding RESTful resources.

<div align="center">Stripe</div>

Airbnb

Airbnb uses Backbone in many of its products. It started with Airbnb Mobile Web (built in six weeks by a team of three) and has since grown to Wish Lists, Match, Search, Communities, Payments, and Internal Tools.

SoundCloud Mobile

SoundCloud is the leading sound sharing platform on the internet, and Backbone.js provides the foundation for SoundCloud Mobile. 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.

Art.sy

Art.sy is a place to discover art you'll love. Art.sy is built on Rails, using Grape to serve a robust JSON API. The main site is a single page app written in CoffeeScript and uses Backbone to provide structure around this API. An admin panel and partner CMS have also been extracted into their own API-consuming Backbone projects.

Pandora

When Pandora redesigned their site in HTML5, they chose Backbone.js to help manage the user interface and interactions. For example, there's a model that represents the "currently playing track", and multiple views that automatically update when the current track changes. The station list is a collection, so that when stations are added or changed, the UI stays up to date.

Inkling

Inkling is a cross-platform way to publish interactive learning content. Inkling for Web uses Backbone.js to make hundreds of complex books — from student textbooks to travel guides and programming manuals — engaging and accessible on the web. Inkling supports WebGL-enabled 3D graphics, interactive assessments, social sharing, and a system for running practice code right in the book, all within a single page Backbone-driven app. Early on, the team decided to keep the site lightweight by using only Backbone.js and raw JavaScript. The result? Complete source code weighing in at a mere 350kb with feature-parity across the iPad, iPhone and web clients. Give it a try with this excerpt from JavaScript: The Definitive Guide.

Inkling

Code School

Code School courses teach people about various programming topics like CoffeeScript, CSS, Ruby on Rails, and more. The new Code School course challenge page is built from the ground up on Backbone.js, using everything it has to offer: the router, collections, models, and complex event handling. Before, the page was a mess of jQuery DOM manipulation and manual Ajax calls. Backbone.js helped introduce a new way to think about developing an organized front-end application in JavaScript.

Code School

CloudApp

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.

CloudApp

SeatGeek

SeatGeek's stadium ticket maps were originally developed with Prototype.js. Moving to Backbone.js and jQuery helped organize a lot of the UI code, and the increased structure has made adding features a lot easier. SeatGeek is also in the process of building a mobile interface that will be Backbone.js from top to bottom.

SeatGeek

Easel

Easel is an in-browser, high fidelity web design tool that integrates with your design and development process. The Easel team uses CoffeeScript, Underscore.js and Backbone.js for their rich visual editor as well as other management functions throughout the site. The structure of Backbone allowed the team to break the complex problem of building a visual editor into manageable components and still move quickly.

Easel

Jolicloud

Jolicloud is an open and independent platform and operating system that provides music

playback, video streaming, photo browsing and document editing — transforming low cost computers into beautiful cloud devices. The new Jolicloud HTML5 app was built from the ground up using Backbone and talks to the Jolicloud Platform, which is based on Node.js. Jolicloud works offline using the HTML5 AppCache, extends Backbone.sync to store data in IndexedDB or localStorage, and communicates with the Joli OS via WebSockets.

Jolicloud

Salon.io

Salon.io provides a space where photographers, artists and designers freely arrange their visual art on virtual walls. Salon.io runs on Rails, but does not use much of the traditional stack, as the entire frontend is designed as a single page web app, using Backbone.js, Brunch and CoffeeScript.

Salon.io

TileMill

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. Note that the gorgeous MapBox homepage is also a Backbone.js app.

TileMill

Blossom

Blossom is a lightweight project management tool for lean teams. Backbone.js is heavily used in combination with CoffeeScript to provide a smooth interaction experience. The app is packaged with Brunch. The RESTful backend is built with Flask on Google App Engine.

Blossom

Trello

Trello is a collaboration tool that organizes your projects into boards. A Trello board holds many lists of cards, which can contain checklists, files and conversations, and may be voted on and organized with labels. Updates on the board happen in real time. The site was built ground up using Backbone.js for all the models, views, and routes.

<div align="center">

Trello

</div>

Tzigla

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, routers 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.

<div align="center">

Tzigla

</div>

Change Log

1.1.2 — *Feb. 20, 2014* — Diff — Docs

○ Backbone no longer tries to require jQuery in Node/CommonJS environments, for better compatibility with folks using Browserify. If you'd like to have Backbone use jQuery from Node, assign it like so: `Backbone.$ = require('jquery');`

○ Bugfix for route parameters with newlines in them.

1.1.1 — *Feb. 13, 2014* — Diff — Docs

○ Backbone now registers itself for AMD (Require.js), Bower and Component, as well as being a CommonJS module and a regular (Java)Script. Whew.

○ Added an `execute` hook to the Router, which allows you to hook in and custom-parse route arguments, like query strings, for example.

○ Performance fine-tuning for Backbone Events.

○ Better matching for Unicode in routes, in old browsers.

1.1.0 — *Oct. 10, 2013* — Diff — Docs

○ Made the return values of Collection's `set`, `add`, `remove`, and `reset` more useful. Instead of returning `this`, they now return the changed (added, removed or updated) model or list of models.

○ Backbone Views no longer automatically attach options passed to the constructor as `this.options` and Backbone Models no longer attach `url` and `urlRoot` options, but you can do it yourself if you prefer.

○ All `"invalid"` events now pass consistent arguments. First the model in question, then the error object, then options.

○ You are no longer permitted to change the **id** of your model during `parse`. Use `idAttribute` instead.

○ On the other hand, `parse` is now an excellent place to extract and vivify incoming nested JSON into associated submodels.

○ Many tweaks, optimizations and bugfixes relating to Backbone 1.0, including URL overrides, mutation of options, bulk ordering, trailing slashes, edge-case listener leaks, nested model parsing...

1.0.0 — *March 20, 2013* — Diff — Docs

○ Renamed Collection's "update" to set, for parallelism with the similar `model.set()`, and contrast with reset. It's now the default updating mechanism after a fetch. If you'd like to continue using "reset", pass `{reset: true}`.

○ Your route handlers will now receive their URL parameters pre-decoded.

○ Added listenToOnce as the analogue of once.

○ Added the findWhere method to Collections, similar to where.

○ Added the `keys`, `values`, `pairs`, `invert`, `pick`, and `omit` Underscore.js methods to Backbone Models.

○ The routes in a Router's route map may now be function literals, instead of references to methods, if you like.

○ `url` and `urlRoot` properties may now be passed as options when instantiating a new Model.

0.9.10 — *Jan. 15, 2013* — Diff — Docs

○ A `"route"` event is triggered on the router in addition to being fired on `Backbone.history`

○ Model validation is now only enforced by default in `Model#save` and no longer enforced by default upon construction or in `Model#set`, unless the `{validate:true}` option is passed.

○ `View#make` has been removed. You'll need to use `$` directly to construct DOM elements now.

○ Passing `{silent:true}` on change will no longer delay individual `"change:attr"` events, instead they are silenced entirely.

○ The `Model#change` method has been removed, as delayed attribute changes as no longer available.

○ Bug fix on `change` where attribute comparison uses `!==` instead of `_.isEqual`.

○ Bug fix where an empty response from the server on save would not call the success function.

○ `parse` now receives `options` as its second argument.

○ Model validation now fires `invalid` event instead of `error`.

0.9.9 — *Dec. 13, 2012* — Diff — Docs

○ Added listenTo and stopListening to Events. They can be used as inversion-of-control flavors of `on` and `off`, for convenient unbinding of all events an object is currently listening to. `view.remove()` automatically calls `view.stopListening()`.

○ When using `add` on a collection, passing `{merge: true}` will now cause duplicate models to have their attributes merged in to the existing models, instead of being ignored.

○ Added update (which is also available as an option to `fetch`) for "smart" updating of sets of models.

○ HTTP `PATCH` support in save by passing `{patch: true}`.

○ The `Backbone` object now extends `Events` so that you can use it as a global event bus, if you like.

○ Added a `"request"` event to Backbone.sync, which triggers whenever a request begins to be made to the server. The natural complement to the `"sync"` event.

○ Router URLs now support optional parts via parentheses, without having to use a regex.

- Backbone events now supports `once`, similar to Node's `once`, or jQuery's `one`.

- Backbone events now support jQuery-style event maps `obj.on({click: action})`.

- While listening to a `reset` event, the list of previous models is now available in `options.previousModels`, for convenience.

- Validation now occurs even during "silent" changes. This change means that the `isValid` method has been removed. Failed validations also trigger an error, even if an error callback is specified in the options.

- Consolidated `"sync"` and `"error"` events within Backbone.sync. They are now triggered regardless of the existence of `success` or `error` callbacks.

- For mixed-mode APIs, `Backbone.sync` now accepts `emulateHTTP` and `emulateJSON` as inline options.

- Collections now also proxy Underscore method name aliases (collect, inject, foldl, foldr, head, tail, take, and so on...)

- Removed `getByCid` from Collections. `collection.get` now supports lookup by both `id` and `cid`.

- After fetching a model or a collection, *all* defined `parse` functions will now be run. So fetching a collection and getting back new models could cause both the collection to parse the list, and then each model to be parsed in turn, if you have both functions defined.

- Bugfix for normalizing leading and trailing slashes in the Router definitions. Their presence (or absence) should not affect behavior.

- When declaring a View, `options`, `el`, `tagName`, `id` and `className` may now be defined as functions, if you want their values to be determined at runtime.

- Added a `Backbone.ajax` hook for more convenient overriding of the default use of `$.ajax`. If AJAX is too passé, set it to your preferred method for server communication.

- `Collection#sort` now triggers a `sort` event, instead of a `reset` event.

- Calling `destroy` on a Model will now return `false` if the model `isNew`.

- To set what library Backbone uses for DOM manipulation and Ajax calls, use `Backbone.$ = ...` instead of `setDomLibrary`.

- Removed the `Backbone.wrapError` helper method. Overriding `sync` should work better for those particular use cases.

- To improve the performance of `add`, `options.index` will no longer be set in the `add` event callback. `collection.indexOf(model)` can be used to retrieve the index of a model as necessary.

- For semantic and cross browser reasons, routes will now ignore search parameters. Routes like `search?query=…&page=3` should become `search/…/3`.

- `Model#set` no longer accepts another model as an argument. This leads to subtle problems and is easily replaced with `model.set(other.attributes)`.

0.9.2 — *March 21, 2012* — Diff — Docs

- Instead of throwing an error when adding duplicate models to a collection, Backbone will now silently skip them instead.

- Added push, pop, unshift, and shift to collections.

- A model's changed hash is now exposed for easy reading of the changed attribute delta, since the model's last `"change"` event.

- Added where to collections for simple filtering.

○ You can now use a single <u>off</u> call to remove all callbacks bound to a specific object.

○ Bug fixes for nested individual change events, some of which may be "silent".

○ Bug fixes for URL encoding in `location.hash` fragments.

○ Bug fix for client-side validation in advance of a `save` call with `{wait: true}`.

○ Updated / refreshed the example <u>Todo List</u> app.

0.9.1 — *Feb. 2, 2012* — <u>Diff</u> — <u>Docs</u>

○ Reverted to 0.5.3-esque behavior for validating models. Silent changes no longer trigger validation (making it easier to work with forms). Added an `isValid` function that you can use to check if a model is currently in a valid state.

○ If you have multiple versions of jQuery on the page, you can now tell Backbone which one to use with `Backbone.setDomLibrary`.

○ Fixes regressions in **0.9.0** for routing with "root", saving with both "wait" and "validate", and the order of nested "change" events.

0.9.0 — *Jan. 30, 2012* — <u>Diff</u> — <u>Docs</u>

○ Creating and destroying models with `create` and `destroy` are now optimistic by default. Pass `{wait: true}` as an option if you'd like them to wait for a successful server response t proceed.

○ Two new properties on views: `$el` — a cached jQuery (or Zepto) reference to the view's element, and `setElement`, which should be used instead of manually setting a view's `el`. It will both set `view.el` and `view.$el` correctly, as well as re-delegating events on the new DOM element.

○ You can now bind and trigger multiple spaced-delimited events at once. For example: `model.on("change:name change:age", ...)`

○ When you don't know the key in advance, you may now call `model.set(key, value)` as well as `save`.

○ Multiple models with the same `id` are no longer allowed in a single collection.

○ Added a `"sync"` event, which triggers whenever a model's state has been successfully synced with the server (create, save, destroy).

○ `bind` and `unbind` have been renamed to `on` and `off` for clarity, following jQuery's lead. The old names are also still supported.

○ A Backbone collection's `comparator` function may now behave either like a <u>sortBy</u> (pass a function that takes a single argument), or like a <u>sort</u> (pass a comparator function that expects two arguments). The comparator function is also now bound by default to the collection — so you can refer to `this` within it.

○ A view's `events` hash may now also contain direct function values as well as the string name of existing view methods.

○ Validation has gotten an overhaul — a model's `validate` function will now be run even for silent changes, and you can no longer create a model in an initially invalid state.

○ Added `shuffle` and `initial` to collections, proxied from Underscore.

○ `Model#urlRoot` may now be defined as a function as well as a value.

○ `View#attributes` may now be defined as a function as well as a value.

○ Calling `fetch` on a collection will now cause all fetched JSON to be run through the collection's model's `parse` function, if one is defined.

○ You may now tell a router to `navigate(fragment, {replace: true})`, which will either use `history.replaceState` or `location.hash.replace`, in order to change the URL without adding a history entry.

○ Within a collection's `add` and `remove` events, the index of the model being added or removed is now available as `options.index`.

○ Added an `undelegateEvents` to views, allowing you to manually remove all configured event delegations.

○ Although you shouldn't be writing your routes with them in any case — leading slashes (`/`) are now stripped from routes.

○ Calling `clone` on a model now only passes the attributes for duplication, not a reference to the model itself.

○ Calling `clear` on a model now removes the `id` attribute.

0.5.3 — *August 9, 2011* — Diff — Docs

A View's `events` property may now be defined as a function, as well as an object literal, making it easier to programmatically define and inherit events. `groupBy` is now proxied from Underscore as a method on Collections. If the server has already rendered everything on page load, pass `Backbone.history.start({silent: true})` to prevent the initial route from triggering. Bugfix for pushState with encoded URLs.

0.5.2 — *July 26, 2011* — Diff — Docs

The `bind` function, can now take an optional third argument, to specify the `this` of the callback function. Multiple models with the same `id` are now allowed in a collection. Fixed a bug where calling `.fetch(jQueryOptions)` could cause an incorrect URL to be serialized. Fixed a brief extra route fire before redirect, when degrading from `pushState`.

0.5.1 — *July 5, 2011* — Diff — Docs

Cleanups from the 0.5.0 release, to wit: improved transparent upgrades from hash-based URLs to pushState, and vice-versa. Fixed inconsistency with non-modified attributes being passed to `Model#initialize`. Reverted a **0.5.0** change that would strip leading hashbangs from routes. Added `contains` as an alias for `includes`.

0.5.0 — *July 1, 2011* — Diff — Docs

A large number of tiny tweaks and micro bugfixes, best viewed by looking at the commit diff. HTML5 `pushState` support, enabled by opting-in with: `Backbone.history.start({pushState: true})`. `Controller` was renamed to `Router`, for clarity. `Collection#refresh` was renamed to `Collection#reset` to emphasize its ability to both reset the collection with new models, as well as empty out the collection when used with no parameters. `saveLocation` was replaced with `navigate`. RESTful persistence methods (save, fetch, etc.) now return the jQuery deferred object for further success/error chaining and general convenience. Improved XSS escaping for `Model#escape`. Added a `urlRoot` option to allow specifying RESTful urls without the use of a collection. An error is thrown if `Backbone.history.start` is called multiple times. `Collection#create` now validates before initializing the new

model. `view.el` can now be a jQuery string lookup. Backbone Views can now also take an `attributes` parameter. `Model#defaults` can now be a function as well as a literal attributes object.

0.3.3 — *Dec 1, 2010* — Diff — Docs

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* — Diff — Docs

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* — Diff — Docs

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* — Diff — Docs

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* — Diff — Docs

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* — Diff — Docs

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* — <u>Diff</u> — <u>Docs</u>

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

0.1.0 — *Oct 13, 2010* — <u>Docs</u>

Initial Backbone release.