backbone.js

```
Backbone.js 0.9.10
```

```
(c) 2010-2012 Jeremy Ashkenas, DocumentCloud Inc.
Backbone may be freely distributed under the MIT license.
For all details and documentation:
http://backbonejs.org
```

Initial Setup

```
(function(){
```

Save a reference to the global object (`window` in the browser, `exports` on the server).

```
  var root = this;
```

Save the previous value of the `Backbone` variable, so that it can be restored later on, if `noConflict` is used.

```
  var previousBackbone = root.Backbone;
```

Create a local reference to array methods.

```
  var array = [];
  var push = array.push;
  var slice = array.slice;
  var splice = array.splice;
```

The top-level namespace. All public Backbone classes and modules will be attached to this. Exported for both CommonJS and the browser.

```
  var Backbone;
  if (typeof exports !== 'undefined') {
    Backbone = exports;
  } else {
    Backbone = root.Backbone = {};
  }
```

Current version of the library. Keep in sync with `package.json`.

```
  Backbone.VERSION = '0.9.10';
```

Require Underscore, if we're on the server, and it's not already present.

```
  var _ = root._;
  if (!_ && (typeof require !== 'undefined')) _ = require('underscore');
```

For Backbone's purposes, jQuery, Zepto, or Ender owns the `$` variable.

```
  Backbone.$ = root.jQuery || root.Zepto || root.ender;
```

Runs Backbone.js in *noConflict* mode, returning the `Backbone` variable to its previous owner. Returns a reference to this Backbone object.

```
  Backbone.noConflict = function() {
    root.Backbone = previousBackbone;
    return this;
  };
```

Turn on `emulateHTTP` to support legacy HTTP servers. Setting this option will fake `"PUT"` and `"DELETE"` requests via the `_method` parameter and set a `X-Http-Method-Override` header.

```
Backbone.emulateHTTP = false;
```

Turn on `emulateJSON` to support legacy servers that can't deal with direct `application/json` requests ... will encode the body as `application/x-www-form-urlencoded` instead and will send the model in a form param named `model`.

```
Backbone.emulateJSON = false;
```

Backbone.Events

Regular expression used to split event strings.

```
var eventSplitter = /\s+/;
```

Implement fancy features of the Events API such as multiple event names `"change blur"` and jQuery-style event maps `{change: action}` in terms of the existing API.

```
var eventsApi = function(obj, action, name, rest) {
  if (!name) return true;
  if (typeof name === 'object') {
    for (var key in name) {
      obj[action].apply(obj, [key, name[key]].concat(rest));
    }
  } else if (eventSplitter.test(name)) {
    var names = name.split(eventSplitter);
    for (var i = 0, l = names.length; i < l; i++) {
      obj[action].apply(obj, [names[i]].concat(rest));

    }
  } else {
    return true;
  }
};
```

Optimized internal dispatch function for triggering events. Tries to keep the usual cases speedy (most Backbone events have 3 arguments).

```
var triggerEvents = function(events, args) {
  var ev, i = -1, l = events.length;
  switch (args.length) {
  case 0: while (++i < l) (ev = events[i]).callback.call(ev.ctx);
  return;
  case 1: while (++i < l) (ev = events[i]).callback.call(ev.ctx, args[0]);
  return;
  case 2: while (++i < l) (ev = events[i]).callback.call(ev.ctx, args[0], args[1]);
  return;
  case 3: while (++i < l) (ev = events[i]).callback.call(ev.ctx, args[0], args[1], args[2]);
  return;
  default: while (++i < l) (ev = events[i]).callback.apply(ev.ctx, args);
  }
};
```

A module that can be mixed in to *any object* in order to provide it with custom events. You may bind with `on` or remove with `off` callback functions to an event; `trigger`-ing an event fires all callbacks in succession.

```
var Events = Backbone.Events = {
```

```
var object = {};
_.extend(object, Backbone.Events);
object.on('expand', function(){ alert('expanded'); });
object.trigger('expand');
```

Bind one or more space separated events, or an events map, to a `callback` function. Passing `"all"` will bind the callback to all events fired.

```
on: function(name, callback, context) {
  if (!(eventsApi(this, 'on', name, [callback, context]) && callback)) return this;
  this._events || (this._events = {});
  var list = this._events[name] || (this._events[name] = []);
  list.push({callback: callback, context: context, ctx: context || this});
  return this;
},
```

Bind events to only be triggered a single time. After the first time the callback is invoked, it will be removed.

```
once: function(name, callback, context) {
  if (!(eventsApi(this, 'once', name, [callback, context]) && callback)) return this;
  var self = this;
  var once = _.once(function() {
    self.off(name, once);
    callback.apply(this, arguments);
  });
  once._callback = callback;
  this.on(name, once, context);
  return this;
},
```

Remove one or many callbacks. If `context` is null, removes all callbacks with that function. If `callback` is null, removes all callbacks for the event. If `name` is null, removes all bound callbacks for all events.

```
off: function(name, callback, context) {
  var list, ev, events, names, i, l, j, k;
  if (!this._events || !eventsApi(this, 'off', name, [callback, context])) return this;
  if (!name && !callback && !context) {
    this._events = {};
    return this;
  }

  names = name ? [name] : _.keys(this._events);
  for (i = 0, l = names.length; i < l; i++) {
    name = names[i];
    if (list = this._events[name]) {
      events = [];
      if (callback || context) {
        for (j = 0, k = list.length; j < k; j++) {
          ev = list[j];
          if ((callback && callback !== ev.callback &&
                           callback !== ev.callback._callback) ||
              (context && context !== ev.context)) {
            events.push(ev);
          }
        }
      }
      this._events[name] = events;
    }
  }

  return this;
```

Trigger one or many events, firing all bound callbacks. Callbacks are passed the same arguments as `trigger` is, apart from the event name (unless you're listening on `"all"`, which will cause your callback to receive the true name of the event as the first argument).

```
  },

  trigger: function(name) {
    if (!this._events) return this;
    var args = slice.call(arguments, 1);
    if (!eventsApi(this, 'trigger', name, args)) return this;
    var events = this._events[name];
    var allEvents = this._events.all;
    if (events) triggerEvents(events, args);
    if (allEvents) triggerEvents(allEvents, arguments);
    return this;
  },
```

An inversion-of-control version of `on`. Tell *this* object to listen to an event in another object ... keeping track of what it's listening to.

```
  listenTo: function(obj, name, callback) {
    var listeners = this._listeners || (this._listeners = {});
    var id = obj._listenerId || (obj._listenerId = _.uniqueId('l'));
    listeners[id] = obj;
    obj.on(name, typeof name === 'object' ? this : callback, this);
    return this;
  },
```

Tell this object to stop listening to either specific events ... or to every object it's currently listening to.

```
  stopListening: function(obj, name, callback) {
    var listeners = this._listeners;
    if (!listeners) return;
    if (obj) {
      obj.off(name, typeof name === 'object' ? this : callback, this);
      if (!name && !callback) delete listeners[obj._listenerId];
    } else {
      if (typeof name === 'object') callback = this;
      for (var id in listeners) {
        listeners[id].off(name, callback, this);
      }
      this._listeners = {};
    }
    return this;
  }
};
```

Aliases for backwards compatibility.

```
Events.bind   = Events.on;
Events.unbind = Events.off;
```

Allow the `Backbone` object to serve as a global event bus, for folks who want global "pubsub" in a convenient place.

```
_.extend(Backbone, Events);
```

Backbone.Model

Create a new model, with defined attributes. A client id (`cid`) is automatically generated and assigned for you.

```
var Model = Backbone.Model = function(attributes, options) {
  var defaults;
  var attrs = attributes || {};
  this.cid = _.uniqueId('c');
```

```
  this.attributes = {};
  if (options && options.collection) this.collection = options.collection;
  if (options && options.parse) attrs = this.parse(attrs, options) || {};
  if (defaults = _.result(this, 'defaults')) {
    attrs = _.defaults({}, attrs, defaults);
  }
  this.set(attrs, options);
  this.changed = {};
  this.initialize.apply(this, arguments);
};
```

Attach all inheritable methods to the Model prototype.

```
_.extend(Model.prototype, Events, {
```

A hash of attributes whose current and previous value differ.

```
  changed: null,
```

The default name for the JSON `id` attribute is `"id"`. MongoDB and CouchDB users may want to set this to `"_id"`.

```
  idAttribute: 'id',
```

Initialize is an empty function by default. Override it with your own initialization logic.

```
  initialize: function(){},
```

Return a copy of the model's `attributes` object.

```
  toJSON: function(options) {
    return _.clone(this.attributes);
  },
```

Proxy `Backbone.sync` by default.

```
  sync: function() {
    return Backbone.sync.apply(this, arguments);
  },
```

Get the value of an attribute.

```
  get: function(attr) {
    return this.attributes[attr];
  },
```

Get the HTML-escaped value of an attribute.

```
  escape: function(attr) {
    return _.escape(this.get(attr));
  },
```

Returns `true` if the attribute contains a value that is not null or undefined.

```
  has: function(attr) {
    return this.get(attr) != null;
  },
```

Set a hash of model attributes on the object, firing `"change"` unless you choose to silence it.

```
  set: function(key, val, options) {
    var attr, attrs, unset, changes, silent, changing, prev, current;
    if (key == null) return this;
```

Handle both `"key", value` and `{key: value}` -style arguments.

```
if (typeof key === 'object') {
  attrs = key;
  options = val;
} else {
  (attrs = {})[key] = val;
}

options || (options = {});
```

Run validation.

```
if (!this._validate(attrs, options)) return false;
```

Extract attributes and options.

```
unset           = options.unset;
silent          = options.silent;
changes         = [];
changing        = this._changing;
this._changing  = true;
```

```
if (!changing) {
  this._previousAttributes = _.clone(this.attributes);
  this.changed = {};
}
current = this.attributes, prev = this._previousAttributes;
```

Check for changes of `id`.

```
if (this.idAttribute in attrs) this.id = attrs[this.idAttribute];
```

For each `set` attribute, update or delete the current value.

```
for (attr in attrs) {
  val = attrs[attr];
  if (!_.isEqual(current[attr], val)) changes.push(attr);
  if (!_.isEqual(prev[attr], val)) {
    this.changed[attr] = val;
  } else {
    delete this.changed[attr];
  }
  unset ? delete current[attr] : current[attr] = val;
}
```

Trigger all relevant attribute changes.

```
if (!silent) {
  if (changes.length) this._pending = true;
  for (var i = 0, l = changes.length; i < l; i++) {
    this.trigger('change:' + changes[i], this, current[changes[i]], options);
  }
}

if (changing) return this;
if (!silent) {
  while (this._pending) {
    this._pending = false;
    this.trigger('change', this, options);
  }
}
```

Remove an attribute from the model, firing `"change"` unless you choose to silence it. `unset` is a noop if the attribute doesn't exist.

```
  }
  this._pending = false;
  this._changing = false;
  return this;
},
```

```
unset: function(attr, options) {
  return this.set(attr, void 0, _.extend({}, options, {unset: true}));
},
```

Clear all attributes on the model, firing `"change"` unless you choose to silence it.

```
clear: function(options) {
  var attrs = {};
  for (var key in this.attributes) attrs[key] = void 0;
  return this.set(attrs, _.extend({}, options, {unset: true}));
},
```

Determine if the model has changed since the last `"change"` event. If you specify an attribute name, determine if that attribute has changed.

```
hasChanged: function(attr) {
  if (attr == null) return !_.isEmpty(this.changed);
  return _.has(this.changed, attr);
},
```

Return an object containing all the attributes that have changed, or false if there are no changed attributes. Useful for determining what parts of a view need to be updated and/or what attributes need to be persisted to the server. Unset attributes will be set to undefined. You can also pass an attributes object to diff against the model, determining if there *would be* a change.

```
changedAttributes: function(diff) {
  if (!diff) return this.hasChanged() ? _.clone(this.changed) : false;
  var val, changed = false;
  var old = this._changing ? this._previousAttributes : this.attributes;
  for (var attr in diff) {
    if (_.isEqual(old[attr], (val = diff[attr]))) continue;
    (changed || (changed = {}))[attr] = val;
  }
  return changed;
},
```

Get the previous value of an attribute, recorded at the time the last `"change"` event was fired.

```
previous: function(attr) {
  if (attr == null || !this._previousAttributes) return null;
  return this._previousAttributes[attr];

},
```

Get all of the attributes of the model at the time of the previous `"change"` event.

```
previousAttributes: function() {
  return _.clone(this._previousAttributes);
},
```

Fetch the model from the server. If the server's representation of the model differs from its current attributes, they will be overriden, triggering a `"change"` event.

```
fetch: function(options) {
  options = options ? _.clone(options) : {};
  if (options.parse === void 0) options.parse = true;
  var success = options.success;
  options.success = function(model, resp, options) {
    if (!model.set(model.parse(resp, options), options)) return false;
```

```
    if (success) success(model, resp, options);
  };
  return this.sync('read', this, options);
},
```

Set a hash of model attributes, and sync the model to the server. If the server returns an attributes hash that differs, the model's state will be `set` again.

```
save: function(key, val, options) {
  var attrs, success, method, xhr, attributes = this.attributes;
```

Handle both `"key", value` and `{key: value}` -style arguments.

```
  if (key == null || typeof key === 'object') {
    attrs = key;
    options = val;
  } else {
    (attrs = {})[key] = val;
  }
```

If we're not waiting and attributes exist, save acts as `set(attr).save(null, opts)`.

```
  if (attrs && (!options || !options.wait) && !this.set(attrs, options)) return false;
```

Do not persist invalid models.

```
  options = _.extend({validate: true}, options);

  if (!this._validate(attrs, options)) return false;
```

Set temporary attributes if `{wait: true}`.

```
  if (attrs && options.wait) {
    this.attributes = _.extend({}, attributes, attrs);
  }
```

After a successful server-side save, the client is (optionally) updated with the server-side state.

```
  if (options.parse === void 0) options.parse = true;
  success = options.success;
  options.success = function(model, resp, options) {
```

Ensure attributes are restored during synchronous saves.

```
    model.attributes = attributes;
    var serverAttrs = model.parse(resp, options);
    if (options.wait) serverAttrs = _.extend(attrs || {}, serverAttrs);
    if (_.isObject(serverAttrs) && !model.set(serverAttrs, options)) {
      return false;
    }
    if (success) success(model, resp, options);
  };
```

Finish configuring and sending the Ajax request.

```
  method = this.isNew() ? 'create' : (options.patch ? 'patch' : 'update');
  if (method === 'patch') options.attrs = attrs;
  xhr = this.sync(method, this, options);
```

Restore attributes.

```
  if (attrs && options.wait) this.attributes = attributes;

  return xhr;
},
```

Destroy this model on the server if it was already persisted. Optimistically removes the model from its collection, if it has one. If `wait: true` is passed, waits for the server to respond before removal.

```
destroy: function(options) {
  options = options ? _.clone(options) : {};
  var model = this;
  var success = options.success;

  var destroy = function() {
    model.trigger('destroy', model, model.collection, options);
  };

  options.success = function(model, resp, options) {
    if (options.wait || model.isNew()) destroy();
    if (success) success(model, resp, options);
  };

  if (this.isNew()) {
    options.success(this, null, options);
    return false;
  }

  var xhr = this.sync('delete', this, options);
  if (!options.wait) destroy();
  return xhr;
},
```

Default URL for the model's representation on the server -- if you're using Backbone's restful methods, override this to change the endpoint that will be called.

```
url: function() {
  var base = _.result(this, 'urlRoot') || _.result(this.collection, 'url') || urlError();
  if (this.isNew()) return base;
  return base + (base.charAt(base.length - 1) === '/' ? '' : '/') + encodeURIComponent(this.id);
},
```

parse converts a response into the hash of attributes to be `set` on the model. The default implementation is just to pass the response along.

```
parse: function(resp, options) {
  return resp;
},
```

Create a new model with identical attributes to this one.

```
clone: function() {
  return new this.constructor(this.attributes);
},
```

A model is new if it has never been saved to the server, and lacks an id.

```
isNew: function() {
  return this.id == null;
},
```

Check if the model is currently in a valid state.

```
isValid: function(options) {
  return !this.validate || !this.validate(this.attributes, options);
},
```

Run validation against the next complete set of model attributes, returning `true` if all is well. Otherwise, fire a general `"error"` event and call the error callback, if specified.

```
_validate: function(attrs, options) {
  if (!options.validate || !this.validate) return true;
  attrs = _.extend({}, this.attributes, attrs);
  var error = this.validationError = this.validate(attrs, options) || null;
```

```
    if (!error) return true;
    this.trigger('invalid', this, error, options || {});
    return false;
  }

});
```

Backbone.Collection

Provides a standard collection class for our sets of models, ordered or unordered. If a `comparator` is specified, the Collection will maintain its models in sort order, as they're added and removed.

```
var Collection = Backbone.Collection = function(models, options) {
  options || (options = {});
  if (options.model) this.model = options.model;
  if (options.comparator !== void 0) this.comparator = options.comparator;
  this.models = [];
  this._reset();
  this.initialize.apply(this, arguments);
  if (models) this.reset(models, _.extend({silent: true}, options));
};
```

Define the Collection's inheritable methods.

```
_.extend(Collection.prototype, Events, {
```

The default model for a collection is just a **Backbone.Model**. This should be overridden in most cases.

```
  model: Model,
```

Initialize is an empty function by default. Override it with your own initialization logic.

```
  initialize: function(){},
```

The JSON representation of a Collection is an array of the models' attributes.

```
  toJSON: function(options) {
    return this.map(function(model){ return model.toJSON(options); });
  },
```

Proxy `Backbone.sync` by default.

```
  sync: function() {
    return Backbone.sync.apply(this, arguments);
  },
```

Add a model, or list of models to the set.

```
  add: function(models, options) {
    models = _.isArray(models) ? models.slice() : [models];
    options || (options = {});
    var i, l, model, attrs, existing, doSort, add, at, sort, sortAttr;
    add = [];
    at = options.at;
    sort = this.comparator && (at == null) && options.sort != false;
    sortAttr = _.isString(this.comparator) ? this.comparator : null;
```

Turn bare objects into model references, and prevent invalid models from being added.

```
    for (i = 0, l = models.length; i < l; i++) {
      if (!(model = this._prepareModel(attrs = models[i], options))) {
        this.trigger('invalid', this, attrs, options);
        continue;
```

If a duplicate is found, prevent it from being added and optionally merge it into the existing model.

```
    }

    if (existing = this.get(model)) {
      if (options.merge) {
        existing.set(attrs === model ? model.attributes : attrs, options);
        if (sort && !doSort && existing.hasChanged(sortAttr)) doSort = true;
      }
      continue;
    }
```

This is a new model, push it to the `add` list.

```
    add.push(model);
```

Listen to added models' events, and index models for lookup by `id` and by `cid`.

```
    model.on('all', this._onModelEvent, this);
    this._byId[model.cid] = model;
    if (model.id != null) this._byId[model.id] = model;
  }
```

See if sorting is needed, update `length` and splice in new models.

```
  if (add.length) {
    if (sort) doSort = true;
    this.length += add.length;
    if (at != null) {
      splice.apply(this.models, [at, 0].concat(add));
    } else {
      push.apply(this.models, add);
    }
  }
```

Silently sort the collection if appropriate.

```
  if (doSort) this.sort({silent: true});

  if (options.silent) return this;
```

Trigger `add` events.

```
  for (i = 0, l = add.length; i < l; i++) {
    (model = add[i]).trigger('add', model, this, options);
  }
```

Trigger `sort` if the collection was sorted.

```
  if (doSort) this.trigger('sort', this, options);

  return this;
},
```

Remove a model, or a list of models from the set.

```
remove: function(models, options) {
  models = _.isArray(models) ? models.slice() : [models];
  options || (options = {});
  var i, l, index, model;
  for (i = 0, l = models.length; i < l; i++) {
    model = this.get(models[i]);
    if (!model) continue;
    delete this._byId[model.id];
    delete this._byId[model.cid];
```

```
    index = this.indexOf(model);
    this.models.splice(index, 1);
    this.length--;
    if (!options.silent) {
      options.index = index;
      model.trigger('remove', model, this, options);
    }
    this._removeReference(model);
  }

  return this;
},
```

Add a model to the end of the collection.

```
push: function(model, options) {
  model = this._prepareModel(model, options);
  this.add(model, _.extend({at: this.length}, options));
  return model;
},
```

Remove a model from the end of the collection.

```
pop: function(options) {
  var model = this.at(this.length - 1);
  this.remove(model, options);
  return model;
},
```

Add a model to the beginning of the collection.

```
unshift: function(model, options) {
  model = this._prepareModel(model, options);
  this.add(model, _.extend({at: 0}, options));
  return model;
},
```

Remove a model from the beginning of the collection.

```
shift: function(options) {
  var model = this.at(0);
  this.remove(model, options);
  return model;
},
```

Slice out a sub-array of models from the collection.

```
slice: function(begin, end) {
  return this.models.slice(begin, end);
},
```

Get a model from the set by id.

```
get: function(obj) {
  if (obj == null) return void 0;
  this._idAttr || (this._idAttr = this.model.prototype.idAttribute);
  return this._byId[obj.id || obj.cid || obj[this._idAttr] || obj];
},
```

Get the model at the given index.

```
at: function(index) {
  return this.models[index];
},
```

Return models with matching attributes. Useful for simple cases of `filter`.

```
where: function(attrs) {
  if (_.isEmpty(attrs)) return [];
  return this.filter(function(model) {
    for (var key in attrs) {
      if (attrs[key] !== model.get(key)) return false;
    }
    return true;
  });
},
```

Force the collection to re-sort itself. You don't need to call this under normal circumstances, as the set will maintain sort order as each item is added.

```
sort: function(options) {
  if (!this.comparator) {
    throw new Error('Cannot sort a set without a comparator');
  }
  options || (options = {});
```

Run sort based on type of `comparator`.

```
  if (_.isString(this.comparator) || this.comparator.length === 1) {
    this.models = this.sortBy(this.comparator, this);
  } else {
    this.models.sort(_.bind(this.comparator, this));
  }

  if (!options.silent) this.trigger('sort', this, options);
  return this;
},
```

Pluck an attribute from each model in the collection.

```
pluck: function(attr) {
  return _.invoke(this.models, 'get', attr);
},
```

Smartly update a collection with a change set of models, adding, removing, and merging as necessary.

```
update: function(models, options) {
  options = _.extend({add: true, merge: true, remove: true}, options);
  if (options.parse) models = this.parse(models, options);
  var model, i, l, existing;
  var add = [], remove = [], modelMap = {};
```

Allow a single model (or no argument) to be passed.

```
  if (!_.isArray(models)) models = models ? [models] : [];
```

Proxy to `add` for this case, no need to iterate...

```
  if (options.add && !options.remove) return this.add(models, options);
```

Determine which models to add and merge, and which to remove.

```
  for (i = 0, l = models.length; i < l; i++) {
    model = models[i];
    existing = this.get(model);
    if (options.remove && existing) modelMap[existing.cid] = true;
    if ((options.add && !existing) || (options.merge && existing)) {
      add.push(model);
    }
  }
  if (options.remove) {
    for (i = 0, l = this.models.length; i < l; i++) {
```

```
      model = this.models[i];
      if (!modelMap[model.cid]) remove.push(model);
    }
  }
```

Remove models (if applicable) before we add and merge the rest.

```
  if (remove.length) this.remove(remove, options);
  if (add.length) this.add(add, options);
  return this;
},
```

When you have more items than you want to add or remove individually, you can reset the entire set with a new list of models, without firing any `add` or `remove` events. Fires `reset` when finished.

```
reset: function(models, options) {
  options || (options = {});
  if (options.parse) models = this.parse(models, options);
  for (var i = 0, l = this.models.length; i < l; i++) {
    this._removeReference(this.models[i]);
  }
  options.previousModels = this.models.slice();
  this._reset();
  if (models) this.add(models, _.extend({silent: true}, options));
  if (!options.silent) this.trigger('reset', this, options);
  return this;
},
```

Fetch the default set of models for this collection, resetting the collection when they arrive. If `update: true` is passed, the response data will be passed through the `update` method instead of `reset`.

```
fetch: function(options) {
  options = options ? _.clone(options) : {};
  if (options.parse === void 0) options.parse = true;
  var success = options.success;
  options.success = function(collection, resp, options) {
    var method = options.update ? 'update' : 'reset';
    collection[method](resp, options);
    if (success) success(collection, resp, options);
  };
  return this.sync('read', this, options);
},
```

Create a new instance of a model in this collection. Add the model to the collection immediately, unless `wait: true` is passed, in which case we wait for the server to agree.

```
create: function(model, options) {
  options = options ? _.clone(options) : {};
  if (!(model = this._prepareModel(model, options))) return false;
  if (!options.wait) this.add(model, options);
  var collection = this;
  var success = options.success;
  options.success = function(model, resp, options) {
    if (options.wait) collection.add(model, options);
    if (success) success(model, resp, options);
  };
  model.save(null, options);
  return model;
},
```

parse converts a response into a list of models to be added to the collection. The default implementation is just to pass it through.

```
parse: function(resp, options) {
  return resp;
```

Create a new collection with an identical list of models as this one.

```
  },

  clone: function() {
    return new this.constructor(this.models);
  },
```

Reset all internal state. Called when the collection is reset.

```
  _reset: function() {
    this.length = 0;
    this.models.length = 0;
    this._byId  = {};
  },
```

Prepare a model or hash of attributes to be added to this collection.

```
  _prepareModel: function(attrs, options) {
    if (attrs instanceof Model) {
      if (!attrs.collection) attrs.collection = this;
      return attrs;
    }
    options || (options = {});
    options.collection = this;
    var model = new this.model(attrs, options);
    if (!model._validate(attrs, options)) return false;
    return model;
  },
```

Internal method to remove a model's ties to a collection.

```
  _removeReference: function(model) {
    if (this === model.collection) delete model.collection;
    model.off('all', this._onModelEvent, this);
  },
```

Internal method called every time a model in the set fires an event. Sets need to update their indexes when models change ids. All other events simply proxy through. "add" and "remove" events that originate in other collections are ignored.

```
  _onModelEvent: function(event, model, collection, options) {
    if ((event === 'add' || event === 'remove') && collection !== this) return;
    if (event === 'destroy') this.remove(model, options);
    if (model && event === 'change:' + model.idAttribute) {
      delete this._byId[model.previous(model.idAttribute)];
      if (model.id != null) this._byId[model.id] = model;
    }
    this.trigger.apply(this, arguments);
  },

  sortedIndex: function (model, value, context) {
    value || (value = this.comparator);
    var iterator = _.isFunction(value) ? value : function(model) {
      return model.get(value);
    };
    return _.sortedIndex(this.models, model, iterator, context);
  }

});
```

Underscore methods that we want to implement on the Collection.

```
var methods = ['forEach', 'each', 'map', 'collect', 'reduce', 'foldl',
```

Mix in each Underscore method as a proxy to `Collection#models`.

```
  'inject', 'reduceRight', 'foldr', 'find', 'detect', 'filter', 'select',
  'reject', 'every', 'all', 'some', 'any', 'include', 'contains', 'invoke',
  'max', 'min', 'toArray', 'size', 'first', 'head', 'take', 'initial', 'rest',
  'tail', 'drop', 'last', 'without', 'indexOf', 'shuffle', 'lastIndexOf',
  'isEmpty', 'chain'];

_.each(methods, function(method) {
  Collection.prototype[method] = function() {
    var args = slice.call(arguments);
    args.unshift(this.models);
    return _[method].apply(_, args);
  };
});
```

Underscore methods that take a property name as an argument.

Use attributes instead of properties.

```
var attributeMethods = ['groupBy', 'countBy', 'sortBy'];

_.each(attributeMethods, function(method) {
  Collection.prototype[method] = function(value, context) {
    var iterator = _.isFunction(value) ? value : function(model) {
      return model.get(value);
    };
    return _[method](this.models, iterator, context);
  };
});
```

Backbone.Router

Routers map faux-URLs to actions, and fire events when routes are matched. Creating a new one sets its `routes` hash, if not set statically.

```
var Router = Backbone.Router = function(options) {
  options || (options = {});
  if (options.routes) this.routes = options.routes;
  this._bindRoutes();
  this.initialize.apply(this, arguments);
};
```

Cached regular expressions for matching named param parts and splatted parts of route strings.

```
var optionalParam = /\((.*?)\)/g;
var namedParam    = /(\(\?)?:\w+/g;
var splatParam    = /\*\w+/g;
var escapeRegExp  = /[\-{}\[\]+?.,\\\^$|#\s]/g;
```

Set up all inheritable **Backbone.Router** properties and methods.

```
_.extend(Router.prototype, Events, {
```

Initialize is an empty function by default. Override it with your own initialization logic.

```
  initialize: function(){},
```

Manually bind a single named route to a callback. For example:

```
this.route('search/:query/p:num', 'search', function(query, num) {
  ...
});
```

```
  route: function(route, name, callback) {
    if (!_.isRegExp(route)) route = this._routeToRegExp(route);
    if (!callback) callback = this[name];
    Backbone.history.route(route, _.bind(function(fragment) {
      var args = this._extractParameters(route, fragment);
```

Simple proxy to `Backbone.history` to save a fragment into the history.

Bind all defined routes to `Backbone.history`. We have to reverse the order of the routes here to support behavior where the most general routes can be defined at the bottom of the route map.

Convert a route string into a regular expression, suitable for matching against the current location hash.

Given a route, and a URL fragment that it matches, return the array of extracted parameters.

Backbone.History

Handles cross-browser history management, based on URL fragments. If the browser does not support `onhashchange`, falls back to polling.

Ensure that `History` can be used outside of the browser.

Cached regex for stripping a leading hash/slash and trailing space.

```
      callback && callback.apply(this, args);
      this.trigger.apply(this, ['route:' + name].concat(args));
      this.trigger('route', name, args);
      Backbone.history.trigger('route', this, name, args);
    }, this));
    return this;
  },

  navigate: function(fragment, options) {
    Backbone.history.navigate(fragment, options);
    return this;
  },

  _bindRoutes: function() {
    if (!this.routes) return;
    var route, routes = _.keys(this.routes);
    while ((route = routes.pop()) != null) {
      this.route(route, this.routes[route]);
    }
  },

  _routeToRegExp: function(route) {
    route = route.replace(escapeRegExp, '\\$&')
                 .replace(optionalParam, '(?:$1)?')
                 .replace(namedParam, function(match, optional){
                   return optional ? match : '([^\/]+)';
                 })
                 .replace(splatParam, '(.*?)');
    return new RegExp('^' + route + '$');
  },

  _extractParameters: function(route, fragment) {
    return route.exec(fragment).slice(1);
  }

});

var History = Backbone.History = function() {
  this.handlers = [];
  _.bindAll(this, 'checkUrl');

  if (typeof window !== 'undefined') {
    this.location = window.location;
    this.history = window.history;
  }
};

var routeStripper = /^[#\/]|\s+$/g;
```

Cached regex for stripping leading and trailing slashes.

```
var rootStripper = /^\/+|\/+$/g;
```

Cached regex for detecting MSIE.

```
var isExplorer = /msie [\w.]+/;
```

Cached regex for removing a trailing slash.

```
var trailingSlash = /\/$/;
```

Has the history handling already been started?

```
History.started = false;
```

Set up all inheritable **Backbone.History** properties and methods.

```
_.extend(History.prototype, Events, {
```

The default interval to poll for hash changes, if necessary, is twenty times a second.

```
  interval: 50,
```

Gets the true hash value. Cannot use location.hash directly due to bug in Firefox where location.hash will always be decoded.

```
  getHash: function(window) {
    var match = (window || this).location.href.match(/#(.*)$/);
    return match ? match[1] : '';
  },
```

Get the cross-browser normalized URL fragment, either from the URL, the hash, or the override.

```
  getFragment: function(fragment, forcePushState) {
    if (fragment == null) {
      if (this._hasPushState || !this._wantsHashChange || forcePushState) {
        fragment = this.location.pathname;
        var root = this.root.replace(trailingSlash, '');
        if (!fragment.indexOf(root)) fragment = fragment.substr(root.length);
      } else {
        fragment = this.getHash();
      }
    }
    return fragment.replace(routeStripper, '');
  },
```

Start the hash change handling, returning `true` if the current URL matches an existing route, and `false` otherwise.

```
  start: function(options) {
    if (History.started) throw new Error("Backbone.history has already been started");
    History.started = true;
```

Figure out the initial configuration. Do we need an iframe? Is pushState desired ... is it available?

```
    this.options          = _.extend({}, {root: '/'}, this.options, options);
    this.root             = this.options.root;
    this._wantsHashChange = this.options.hashChange !== false;
    this._wantsPushState  = !!this.options.pushState;

    this._hasPushState    = !!(this.options.pushState && this.history && this.history.pushState);
    var fragment          = this.getFragment();
    var docMode           = document.documentMode;
    var oldIE             = (isExplorer.exec(navigator.userAgent.toLowerCase()) && (!docMode || docMode <= 7));
```

Normalize root to always include a leading and trailing slash.

```
    this.root = ('/' + this.root + '/').replace(rootStripper, '/');
```

Depending on whether we're using pushState or hashes, and whether 'onhashchange' is supported, determine how we check the URL state.

```
  if (oldIE && this._wantsHashChange) {
    this.iframe = Backbone.$('<iframe src="javascript:0" tabindex="-1" />').hide().appendTo('body')[0].contentWindow;
    this.navigate(fragment);
  }

  if (this._hasPushState) {
    Backbone.$(window).on('popstate', this.checkUrl);
  } else if (this._wantsHashChange && ('onhashchange' in window) && !oldIE) {
    Backbone.$(window).on('hashchange', this.checkUrl);
  } else if (this._wantsHashChange) {
    this._checkUrlInterval = setInterval(this.checkUrl, this.interval);
  }
```

Determine if we need to change the base url, for a pushState link opened by a non-pushState browser.

```
  this.fragment = fragment;
  var loc = this.location;
  var atRoot = loc.pathname.replace(/[^\/]$/, '$&/') === this.root;
```

If we've started off with a route from a `pushState`-enabled browser, but we're currently in a browser that doesn't support it...

```
  if (this._wantsHashChange && this._wantsPushState && !this._hasPushState && !atRoot) {
    this.fragment = this.getFragment(null, true);
    this.location.replace(this.root + this.location.search + '#' + this.fragment);
```

Return immediately as browser will do redirect to new url

```
    return true;
```

Or if we've started out with a hash-based route, but we're currently in a browser where it could be `pushState`-based instead...

```
  } else if (this._wantsPushState && this._hasPushState && atRoot && loc.hash) {
    this.fragment = this.getHash().replace(routeStripper, '');
    this.history.replaceState({}, document.title, this.root + this.fragment + loc.search);
  }

  if (!this.options.silent) return this.loadUrl();
},
```

Disable Backbone.history, perhaps temporarily. Not useful in a real app, but possibly useful for unit testing Routers.

```
stop: function() {
  Backbone.$(window).off('popstate', this.checkUrl).off('hashchange', this.checkUrl);
  clearInterval(this._checkUrlInterval);
  History.started = false;
},
```

Add a route to be tested when the fragment changes. Routes added later may override previous routes.

```
route: function(route, callback) {
  this.handlers.unshift({route: route, callback: callback});
},
```

Checks the current URL to see if it has changed, and if it has, calls `loadUrl`, normalizing across the hidden iframe.

```
checkUrl: function(e) {
  var current = this.getFragment();
  if (current === this.fragment && this.iframe) {
    current = this.getFragment(this.getHash(this.iframe));
  }
  if (current === this.fragment) return false;
  if (this.iframe) this.navigate(current);
  this.loadUrl() || this.loadUrl(this.getHash());
}
```

Attempt to load the current URL fragment. If a route succeeds with a match, returns `true`. If no defined routes matches the fragment, returns `false`.

```
},
```

```
loadUrl: function(fragmentOverride) {
  var fragment = this.fragment = this.getFragment(fragmentOverride);
  var matched = _.any(this.handlers, function(handler) {
    if (handler.route.test(fragment)) {
      handler.callback(fragment);
      return true;
    }
  });
  return matched;
},
```

Save a fragment into the hash history, or replace the URL state if the 'replace' option is passed. You are responsible for properly URL-encoding the fragment in advance.

The options object can contain `trigger: true` if you wish to have the route callback be fired (not usually desirable), or `replace: true`, if you wish to modify the current URL without adding an entry to the history.

```
navigate: function(fragment, options) {
  if (!History.started) return false;
  if (!options || options === true) options = {trigger: options};
  fragment = this.getFragment(fragment || '');
  if (this.fragment === fragment) return;
  this.fragment = fragment;
  var url = this.root + fragment;
```

If pushState is available, we use it to set the fragment as a real URL.

```
  if (this._hasPushState) {
    this.history[options.replace ? 'replaceState' : 'pushState']({}, document.title, url);
```

If hash changes haven't been explicitly disabled, update the hash fragment to store history.

```
  } else if (this._wantsHashChange) {
    this._updateHash(this.location, fragment, options.replace);
    if (this.iframe && (fragment !== this.getFragment(this.getHash(this.iframe)))) {
```

Opening and closing the iframe tricks IE7 and earlier to push a history entry on hash-tag change. When replace is true, we don't want this.

```
      if(!options.replace) this.iframe.document.open().close();
      this._updateHash(this.iframe.location, fragment, options.replace);
    }
```

If you've told us that you explicitly don't want fallback hashchange-based history, then `navigate` becomes a page refresh.

```
  } else {
    return this.location.assign(url);
  }
  if (options.trigger) this.loadUrl(fragment);
},
```

Update the hash location, either replacing the current entry, or adding a new one to the browser history.

```
_updateHash: function(location, fragment, replace) {
  if (replace) {
    var href = location.href.replace(/(javascript:|#).*$/, '');
    location.replace(href + '#' + fragment);
  } else {
```

Some browsers require that `hash` contains a leading #.

```
    location.hash = '#' + fragment;
  }
}
```

Create the default Backbone.history.

Backbone.View

Creating a Backbone.View creates its initial element outside of the DOM, if an existing element is not provided...

Cached regex to split keys for `delegate`.

List of view options to be merged as properties.

Set up all inheritable **Backbone.View** properties and methods.

The default `tagName` of a View's element is `"div"`.

jQuery delegate for element lookup, scoped to DOM elements within the current view. This should be prefered to global lookups where possible.

Initialize is an empty function by default. Override it with your own initialization logic.

render is the core function that your view should override, in order to populate its element (`this.el`), with the appropriate HTML. The convention is for **render** to always return `this`.

Remove this view by taking the element out of the DOM, and removing any applicable Backbone.Events listeners.

Change the view's element (`this.el` property), including event re-delegation.

```
});

Backbone.history = new History;



var View = Backbone.View = function(options) {
  this.cid = _.uniqueId('view');
  this._configure(options || {});
  this._ensureElement();
  this.initialize.apply(this, arguments);
  this.delegateEvents();
};

var delegateEventSplitter = /^(\S+)\s*(.*)$/;

var viewOptions = ['model', 'collection', 'el', 'id', 'attributes', 'className', 'tagName', 'events'];

_.extend(View.prototype, Events, {

  tagName: 'div',

  $: function(selector) {
    return this.$el.find(selector);
  },

  initialize: function(){},

  render: function() {
    return this;
  },

  remove: function() {
    this.$el.remove();
    this.stopListening();
    return this;
  },

  setElement: function(element, delegate) {
    if (this.$el) this.undelegateEvents();
    this.$el = element instanceof Backbone.$ ? element : Backbone.$(element);
    this.el = this.$el[0];
    if (delegate !== false) this.delegateEvents();
    return this;
```

Set callbacks, where `this.events` is a hash of

{"event selector": "callback"}

```
{
  'mousedown .title':  'edit',
  'click .button':     'save'
  'click .open':       function(e) { ... }
}
```

pairs. Callbacks will be bound to the view, with `this` set properly. Uses event delegation for efficiency. Omitting the selector binds the event to `this.el`. This only works for delegate-able events: not `focus`, `blur`, and not `change`, `submit`, and `reset` in Internet Explorer.

Clears all callbacks previously bound to the view with `delegateEvents`. You usually don't need to use this, but may wish to if you have multiple Backbone views attached to the same DOM element.

Performs the initial configuration of a View with a set of options. Keys with special meaning *(model, collection, id, className)*, are attached directly to the view.

Ensure that the View has a DOM element to render into. If `this.el` is a string, pass it through `$()`, take the first matching element, and re-assign it to `el`. Otherwise, create an element from the `id`, `className` and `tagName` properties.

Backbone.sync

Map from CRUD to HTTP for our default `Backbone.sync`

```
  },

  delegateEvents: function(events) {
    if (!(events || (events = _.result(this, 'events')))) return;
    this.undelegateEvents();
    for (var key in events) {
      var method = events[key];
      if (!_.isFunction(method)) method = this[events[key]];
      if (!method) throw new Error('Method "' + events[key] + '" does not exist');
      var match = key.match(delegateEventSplitter);
      var eventName = match[1], selector = match[2];
      method = _.bind(method, this);
      eventName += '.delegateEvents' + this.cid;
      if (selector === '') {
        this.$el.on(eventName, method);
      } else {
        this.$el.on(eventName, selector, method);
      }
    }
  },

  undelegateEvents: function() {
    this.$el.off('.delegateEvents' + this.cid);
  },

  _configure: function(options) {
    if (this.options) options = _.extend({}, _.result(this, 'options'), options);
    _.extend(this, _.pick(options, viewOptions));
    this.options = options;
  },

  _ensureElement: function() {
    if (!this.el) {
      var attrs = _.extend({}, _.result(this, 'attributes'));
      if (this.id) attrs.id = _.result(this, 'id');
      if (this.className) attrs['class'] = _.result(this, 'className');
      var $el = Backbone.$('<' + _.result(this, 'tagName') + '>').attr(attrs);
      this.setElement($el, false);
    } else {
      this.setElement(_.result(this, 'el'), false);
    }
  }

});

var methodMap = {
```

implementation.

```
  'create': 'POST',
  'update': 'PUT',
  'patch':  'PATCH',
  'delete': 'DELETE',
  'read':   'GET'
};

Backbone.sync = function(method, model, options) {
  var type = methodMap[method];
```

Override this function to change the manner in which Backbone persists models to the server. You will be passed the type of request, and the model in question. By default, makes a RESTful Ajax request to the model's `url()`. Some possible customizations could be:

- Use `setTimeout` to batch rapid-fire updates into a single request.
- Send up the models as XML instead of JSON.
- Persist models via WebSockets instead of Ajax.

Turn on `Backbone.emulateHTTP` in order to send `PUT` and `DELETE` requests as `POST`, with a `_method` parameter containing the true HTTP method, as well as all requests with the body as `application/x-www-form-urlencoded` instead of `application/json` with the model in a param named `model`. Useful when interfacing with server-side languages like **PHP** that make it difficult to read the body of `PUT` requests.

Default options, unless specified.

```
  _.defaults(options || (options = {}), {
    emulateHTTP: Backbone.emulateHTTP,
    emulateJSON: Backbone.emulateJSON
  });
```

Default JSON-request options.

```
  var params = {type: type, dataType: 'json'};
```

Ensure that we have a URL.

```
  if (!options.url) {
    params.url = _.result(model, 'url') || urlError();
  }
```

Ensure that we have the appropriate request data.

```
  if (options.data == null && model && (method === 'create' || method === 'update' || method === 'patch')) {
    params.contentType = 'application/json';
    params.data = JSON.stringify(options.attrs || model.toJSON(options));
  }
```

For older servers, emulate JSON by encoding the request into an HTML-form.

```
  if (options.emulateJSON) {
    params.contentType = 'application/x-www-form-urlencoded';
    params.data = params.data ? {model: params.data} : {};
  }
```

For older servers, emulate HTTP by mimicking the HTTP method

```
  if (options.emulateHTTP && (type === 'PUT' || type === 'DELETE' || type === 'PATCH')) {
```

with `_method` And an `X-HTTP-Method-Override` header.

Don't process data on a non-GET request.

Make the request, allowing the user to override any Ajax options.

Set the default implementation of `Backbone.ajax` to proxy through to `$`.

Helpers

Helper function to correctly set up the prototype chain, for subclasses. Similar to `goog.inherits`, but uses a hash of prototype properties and class properties to be extended.

The constructor function for the new subclass is either defined by you (the "constructor" property in your `extend` definition), or defaulted by us to simply call the parent's constructor.

Add static properties to the constructor function, if supplied.

```
    params.type = 'POST';
    if (options.emulateJSON) params.data._method = type;
    var beforeSend = options.beforeSend;
    options.beforeSend = function(xhr) {
      xhr.setRequestHeader('X-HTTP-Method-Override', type);
      if (beforeSend) return beforeSend.apply(this, arguments);
    };
  }

  if (params.type !== 'GET' && !options.emulateJSON) {
    params.processData = false;
  }

  var success = options.success;
  options.success = function(resp) {
    if (success) success(model, resp, options);
    model.trigger('sync', model, resp, options);
  };

  var error = options.error;
  options.error = function(xhr) {
    if (error) error(model, xhr, options);
    model.trigger('error', model, xhr, options);
  };

  var xhr = options.xhr = Backbone.ajax(_.extend(params, options));
  model.trigger('request', model, xhr, options);
  return xhr;
};

Backbone.ajax = function() {
  return Backbone.$.ajax.apply(Backbone.$, arguments);
};

var extend = function(protoProps, staticProps) {
  var parent = this;
  var child;

  if (protoProps && _.has(protoProps, 'constructor')) {
    child = protoProps.constructor;
  } else {
    child = function(){ return parent.apply(this, arguments); };
  }

  _.extend(child, parent, staticProps);
```

Set the prototype chain to inherit from `parent`, without calling `parent`'s constructor function.

Add prototype properties (instance properties) to the subclass, if supplied.

Set a convenience property in case the parent's prototype is needed later.

Set up inheritance for the model, collection, router, view and history.

Throw an error when a URL is needed, and none is supplied.

```
    var Surrogate = function(){ this.constructor = child; };
    Surrogate.prototype = parent.prototype;
    child.prototype = new Surrogate;

    if (protoProps) _.extend(child.prototype, protoProps);

    child.__super__ = parent.prototype;

    return child;
  };

  Model.extend = Collection.extend = Router.extend = View.extend = History.extend = extend;

  var urlError = function() {
    throw new Error('A "url" property or function must be specified');
  };

}).call(this);
```