CoffeeScript, Meet Backbone.js

CoffeeScript, Meet Backbone.js is a simple Backbone.js tutorial written in CoffeeScript comprised of self-explanatory "hello world" examples of increasing complexity. It was designed to provide a smoother transition from zero to the popular Todos example The bulk of this tutorial is a rewrite of the original hello-backbonejs tutorial.

Backbone.js offers a lean MVC framework for organizing your Javascript application. It leads to more maintainable code by untangling the "spaghetti" of callbacks tied to different parts of the DOM and the backend server that often arises in rich client-side applications.

The tutorial starts with a minimalist View object, and progressively introduces event binding/handling, Models, and Collections.

Once in the tutorial, use the navigation menu in the top-right corner to view other examples. Example numbers are in order of increasing complexity.

START THE TUTORIAL

Fork me on GitHub

The only non-Javascript part of the examples is the fol[illegible]ing HTML template (with some minimal styling):

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CoffeeScript, Meet Backbone.js: Part N</title>
  <link rel="stylesheet" href="/coffeescript-meet-backbonejs/style.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js"></script>
  <script src="http://ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
  <script src="http://ajax.cdnjs.com/ajax/libs/underscore.js/1.1.6/underscore-min.js"></script>
  <script src="http://ajax.cdnjs.com/ajax/libs/backbone.js/0.3.3/backbone-min.js"></script>
  <script src="script.js"></script>
</head>
<body>
  <header>
    <h1>CoffeeScript, Meet Backbone.js: Part N</h1>
    <p>Looking for <a href="docs/script.html">the documentation</a>?</p>
  </header>
</body>
</html>
```

script.coffee

Extending Backbone.View

First, we'll extend Backbone.View to create a very minimal unordered list view. Take a look at the implementation of this step so you'll know what we're building.

The `jQuery` wrapper isn't necessary since we know this script is called after jQuery has been included, but it explicitly shows our dependence on jQuery. Clarity is a good thing.

```
jQuery ->
```

Our main application view.

```
  class ListView extends Backbone.View
```

We'll be using the `body` element for our view in all our examples. All views have a DOM element at all times. `el` is our view's connetion to the DOM.

```
    el: $ 'body'
```

`initialize()` is automatically called upon instantiation.

```
    initialize: ->
```

We're using Underscore.js's bindAll method to bind all the view's methods to this instance of our view.

```
      _.bindAll @
      @render()
```

`render()` renders the view in `@el`. It must be called by the us; we called it at the end of our `initialize()` function.

```
    render: ->
      $(@el).append '<ul><li>Hello, Backbone!</li></ul>'
```

Lastly, we instantiate our main app view.

Onward to Part 2.

```
    list_view = new ListView
```

JUMP TO ...

script.coffee

Binding DOM Events to View Methods

So, part one was pretty boring. Let's spice it up by binding DOM events to our view's methods. The implementation of this step is slighlty more exciting than the last.

```
jQuery ->

  class ListView extends Backbone.View

    el: $ 'body'

    initialize: ->
      _.bindAll @
      @counter = 0
      @render()
```

We'll add a button and an empty list to our view.

```
    render: ->
      $(@el).append '<button>Add List Item</button>'
      $(@el).append '<ul></ul>'
```

`addItem()` will be called via the `click` event on the button we added in our `render` method.

```
    addItem: ->
      @counter++
      $('ul').append "<li>Hello, Backbone #{@counter}!</li>"
```

`events` is a JSON object where DOM events are bound to view methods. Backbone doesn't have a separate controller to handle event bindings; it all

```
    events: 'click button': 'addItem'
```

happens in a view.

Onward to Part 3.

```
    list_view = new ListView
```

JUMP TO ...

script.coffee

Using a Collection of Models

So far we've used the view as our view and model. We'll extend Backbone.Model and Backbone.Collection to separate our model from our view.

You might want to wiew the implementation of this step before diving in.

```
jQuery ->
```

The model is the heart of any application. We have a very small heart in this example.

```
  class Item extends Backbone.Model
```

`defaults` is a JSON object used to specify the default attributes for our model.

```
    defaults:
      part1: 'Hello'
      part2: 'Backbone'
```

Our collection is an ordered set of the previously defined `Item`s.

```
  class List extends Backbone.Collection

    model: Item

  class ListView extends Backbone.View

    el: $ 'body'
```

`initialize()` now instantiates a collection and binds the `add` event to the `appendItem()` method.

```
    initialize: ->
      _.bindAll @
      @collection = new List
      @collection.bind 'add', @appendItem

      @counter = 0
      @render()

    render: ->
      $(@el).append '<button>Add List Item</button>'
      $(@el).append '<ul></ul>'
```

`addItem()` now deals solely with models/collections. View updates are delegated to the `add` event bound to `appendItem()` when we `initialize`ed our list_view.

```
  addItem: ->
    @counter++
```

Instantiate a new Item,

```
    item = new Item
```

and modify its second part.

```
    item.set part2: "#{item.get 'part2'} #{@counter}"
```

Then, add it to our collection.

```
    @collection.add item
```

`appendItem()` is triggered by the collection event `add` and handles updates to the interface.

```
  appendItem: (item) ->
    $('ul').append "<li>#{item.get 'part1'} #{item.get 'part2'}!</li>"

  events: 'click button': 'addItem'

list_view = new ListView
```

Onward to Part 4.

script.coffee

Using a Dedicated View for a Model

We added a model and collection in the previous example, but our main application view still held the structure for our model. We'll add a dedicated view, `ItemView`, for our `Item` model. The implementation for this part looks exactly the same as the implementation of Part 3.

```
jQuery ->

  class Item extends Backbone.Model

    defaults:
      part1: 'Hello'
      part2: 'Backbone'


  class List extends Backbone.Collection

    model: Item
```

The `ItemView` is now responsible for rendering each `Item`.

```
  class ItemView extends Backbone.View
```

`tagName` is used to create our `@el`. You can also add `className` and `id` properties, but `tagName` is enough for this simple example.

```
    tagName: 'li'

    initialize: ->
      _.bindAll @

    render: ->
      $(@el).html "<span>#{@model.get 'part1'} #{@model.get 'part2'}!</span>"
```

Returning `@` is considered a good practice. It let's us chain method calls (i.e., `item_view.render().el`).

```
      @
```

```
class ListView extends Backbone.View

  el: $ 'body'

  initialize: ->
    _.bindAll @

    @collection = new List
    @collection.bind 'add', @appendItem

    @counter = 0
    @render()

  render: ->
    $(@el).append '<button>Add List Item</button>'
    $(@el).append '<ul></ul>'

  addItem: ->
    @counter++
    item = new Item
    item.set part2: "#{item.get 'part2'} #{@counter}"
    @collection.add item
```

`appendItem()` no longer renders an individual `Item`. Rendering is delegated to the `render()` method of each `ItemView` instance.

```
  appendItem: (item) ->
```

Instantiate a new `ItemView` using `item` as the `model`.

```
    item_view = new ItemView model: item
    $('ul').append item_view.render().el

  events: 'click button': 'addItem'


list_view = new ListView
```

Onward to Part 5.

JUMP TO ...

script.coffee

Adding Actions to a View

Our models have been pretty lifeless so far. We'll attach some actions to our `Item`s for some dynamic goodness. Take a look at the implementation before getting started.

```
jQuery ->

  class Item extends Backbone.Model

    defaults:
      part1: 'Hello'
      part2: 'Backbone'


  class List extends Backbone.Collection

    model: Item


  class ItemView extends Backbone.View

    tagName: 'li'
```

`initialize()` binds `change` and `remove` to `@render` and `@unrender`, respectively.

```
    initialize: ->
      _.bindAll @

      @model.bind 'change', @render
      @model.bind 'remove', @unrender
```

`render()` now includes two extra `span`s for swapping and deleting an

```
    render: =>
```

item.

```
    $(@el).html """
      <span>#{@model.get 'part1'} #{@model.get 'part2'}!</span>
      <span class="swap">swap</span>
      <span class="delete">delete</span>
    """
    @
```

`unrender()` removes the calling list item from the DOM. This uses jQuery's `remove()` method.

```
  unrender: =>
    $(@el).remove()
```

`swap()` interchanges an `Item`'s attributes. The `set()` model function triggers the `change` event.

```
  swap: ->
    @model.set
      part1: @model.get 'part2'
      part2: @model.get 'part1'
```

`remove()` calls the model's `destroy()` method, removing the model from its collection. `destroy()` would normally delete the record from its persistent storage, but we'll override this in `Backbone.sync` below.

```
  remove: -> @model.destroy()
```

`ItemView`s now respond to two click actions for each `Item`.

```
  events:
    'click .swap': 'swap'
    'click .delete': 'remove'
```

We no longer need to modify the `ListView` because `swap` and `delete` are called on each `Item`.

```
class ListView extends Backbone.View

  el: $ 'body'

  initialize: ->
    _.bindAll @

    @collection = new List
    @collection.bind 'add', @appendItem
    @counter = 0
    @render()

  render: ->
    $(@el).append '<button>Add Item List</button>'
    $(@el).append '<ul></ul>'

  addItem: ->
    @counter++
    item = new Item
    item.set part2: "#{item.get 'part2'} #{@counter}"
    @collection.add item

  appendItem: (item) ->
    item_view = new ItemView model: item
    $('ul').append item_view.render().el

  events: 'click button': 'addItem'
```

We'll override `Backbone.sync` since we're not making any calls to the server when we change our model.

```
Backbone.sync = (method, model, success, error) ->
```

Perform a NOOP when we successfully change our model. In our example, this will happen when we remove each Item view.

```
  success()
```

```
list_view = new ListView
```