Client-side Templating

We need a more efficient way of creating DOM elements than what we've been using.

Template: an HTML fragment that accepts a JavaScript object to fill in its missing data pieces. A templating engine will use the template and the data to generate markup which can then be returned from a server request or inserted into the DOM. This is essentially what every template library does, from Underscore, to Handlebars, to Mustache, to Dust, to Jade, to HAML.

Most Backbone applications will use templating, even though Backbone doesn't provide any built-in support for it. Underscore, which is a "hard" dependency for Backbone does support micro-templating. Backbone can work with any templating engine, however, so we will look at two, Underscore and, more briefly, Handlebars.

Most template libraries, including Underscore and Handlebars, use a two-step process:

1. compile the template - pass the original template markup to the template function which will return a function
2. call that function, passing in the data the template will use, and it will return the generated markup

One very common approach to templates is to place them in script blocks and put ID's on the script elements

```
// specifying type as "text/template" prevents the browser from executing as scrip
<script id="latLngTemplate" type="text/template">
    <li class="location">
        <span class="latLng_value"><%= lat %> latitude, <%= lng %> longitude</span
    </li>
</script>
```

The template is then selected from the DOM, compiled, then markup generated and inserted into a DOM fragment or into the document itself.

```
// using a template engine like Underscore means a two-step process
var View = Backbone.View.extend({
    template: '#latLngTemplate',
    render: function () {

        var data = { lat: -27, lng: 153 };

        // retrieve the template markup
        var markup = $(this.template).html();

        // compile the template using the template markup
        var templateFunc = _.template(markup);

        // insert the generated markup into the DOM fragment
        this.$el.html(templateFunc(data));

        return this;
    }
});
```

why use templates?

```
// anti-pattern - so this:
var li = '<li class="book">';
li += 'Title: Finnegan\'s Wake';
li += 'Artist: James Joyce';
li += '</li>';

// can be replaced with this
<script type='text/template' id='book_template'>
  <li class='book'>
    Title: <%= title %>
    Artist: <%= author %>
  </li>
</script>
```

Underscore templates consist of HTML and 3 different kinds of executable code blocks:

1. `<% ... %>` - allows you to execute arbitrary code, set variables, loop over collections, or conditionally branch
2. `<%= ... %>` - evaluate an expression and render the result inline
3. `<%- ... %>` - evaluate an expression and render the HTML escaped inline

Executing arbitrary code inside `<% ... %>`

```
// just JavaScript inside the template to loop over an array
<script id="latLngTemplate" type="text/template">
    <p>Latitude: <%= lat %></p>
    <p>Longitude: <%= lng %></p>
    <% [1, 2, 3].forEach(function (number) { %>
        <p>
            <%= number %>
        </p>
    <% }); %>
</script>
```

Handlebars: a templating engine based on Mustache

- can execute a small set of simple constructs like conditionals and loops - anything more complicated lives in helpers
- philosophically opposed to code in templates - the authors believe templates should be as simple as possible and the view should be separated from application code
- code blocks are delimited by `{{ ... }}`

simple substitution identical to Underscore

```
// templating delimiters for simple substitution
<p>Latitude: {{lat}}</p>
<p>Longitude: {{lng}}</p>
```

Handlebars has a number built-in helpers to facilitate common tasks normally done with code, such as looping:

```
{{#each aList}}
    <p>
        {{this}} // refers to the current item in the collection
    </p>
{{/each}}
```

Handlebars conditional

```
div class="entry">
  {{!-- only output author name if an author exists --}}
  {{#if author}}
    <h1>{{firstName}} {{lastName}}</h1>
  {{/if}}
</div>
```

Like Underscore, rendering in Handlebars is a two stage process: compile, then execute

```
// compile:
var source = '<p>Latitude: {{lat}}</p>';
var compiled = Handlebars.compile(source);

// then execute: input to compiled function is data
var rendered = compiled({ lat: -27 });

// rendered output: <p>Latitude: -27</p>
```

Selecting templates from the DOM via script elements is an anti-pattern that should be avoided in production. Selecting from the DOM is an expensive action and it's far more performant to do things in memory. One way to address this problem is to use pre-compilation.

The reason to favor a strategy like pre-compilation is because the first step can be done in advance. It isn't necessary to wait to do that step when you're trying to render content.

```javascript
// given a template in a script element with an id of bookTemplate
$(function () {
    'use strict';

    var data = {
        title: 'Wool',
        author: 'Hugh Howey'
    };

    // this step can be done ahead of time
    var templateFunc = Handlebars.compile($('#bookTemplate').html());

    var markup = templateFunc(data);
    $('.book_list').append(markup);
});
```

Handlebars includes support for pre-compiling file-based templates:

```
//usage
handlebars <input> -f <output>
```

The pre-compilation is typically handled by a file-system watcher, a build script (Grunt, Gulp, or Webpack), or a post-build event if you are working with Visual Studio. For the handlebars executable to work, you need to install the handlebars npm module.

```
// install it globally so it can be used anywhere
npm install handlebars -g

// usage handlebars templates/ -f templates.js
```