Handlebars.js – a Behind the Scenes Look

Gabriel Manricks on Jul 26th 2013 with 11 Comments

Tutorial Details

-
- **Framework**: Handlebars.js
- **Difficulty**: Medium
- **Estimated Completion Time** 15 Minutes

Handlebars has been gaining popularity with its adoption in frameworks like Meteor and Ember.js, but what is really going on behind the scenes of this exciting templating engine?

In this article we will take a deep look through the underlying process Handlebars goes through to compile your templates.

This article expects you to have read my previous introduction to Handlebars and as such assumes you know the basics of creating Handlebar templates.

When using a Handlebars template you probably know that you start by compiling the template's source into a function using `Handlebars.compile()` and then you use that function to generate the final HTML, passing in values for properties and placeholders.

But that seemingly simple compile function is actually doing quite a few steps behind the scenes, and that is what this article will really be about; let's take a look at a quick breakdown of the process:

- Tokenize the source into components.
- Process each token into a set of operations.
- Convert the process stack into a function.
- Run the function with the context and helpers to output some HTML.

The Setup

In this article we will be building a tool to analyze Handlebars templates at each of these steps, so to display the results a bit better on screen, I will be using the prism.js syntax highlighter created by the one and only Lea Verou. Download the minified source remembering to check JavaScript in the languages section.

The next step is to create a blank HTML file and fill it with the following:

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/html">
    <head>
        <title>Handlebars.js</title>
        <link rel="stylesheet" href="prism.css"></p>

        <script src="prism.js" data-manual></script>
        <script src="handlebars.js"></script>
    </head>
    <body>
        <div id="analysis">
            <div id="tokens"><h1>Tokens:</h1></div>
            <div id="operations"><h1>Operations:</h1></div>
            <div id="output"><h1>Output:</h1></div>
            <div id="function">
                <h1>Function:</h1>
                <pre><code class="language-javascript" id="source"
            </div>
        </div>
        <script id="dt" type="template/handlebars">
        </script>

        <script>
            //Code will go here
        </script>
    </body>
</html>
```

It’s just some boilerplate code which includes handlebars and prism and then set’s up some divs for the different steps. At the bottom, you can see two script blocks: the first is for the template and the second is for our JS code.

I also wrote a little CSS to arrange everything a bit better, which you are free to add:

```
body{
    margin: 0;
    padding: 0;
    font-family: "opensans", Arial, sans-serif;
    background: #F5F2F0;
    font-size: 13px;
}
#analysis {
    top: 0;
    left: 0;
    position: absolute;
    width: 100%;
    height: 100%;
    margin: 0;
    padding: 0;
}
#analysis div {
    width: 33.33%;
    height: 50%;
    float: left;
    padding: 10px 20px;
    box-sizing: border-box;
    overflow: auto;
}
#function {
    width: 100% !important;
}
```

Next we need a template, so let’s begin with the simplest template possible, just some static text:

```
<script id="dt" type="template/handlebars">
```

```
    Hello World!
</script>

<script>
    var src = document.getElementById("dt").innerHTML.trim();

    //Display Output
    var t = Handlebars.compile(src);
    document.getElementById("output").innerHTML += t();
</script>
```

Opening this page in your browser should result in the template being displayed in the output box as expected, nothing different yet, we now have to write the code to analyze the process at each of the other three stages.

Tokens

The first step handlebars performs on your template is to tokenize the source, what this means is we need to break the source apart into its individual components so that we can handle each piece appropriately. So for example, if there was some text with a placeholder in the middle, then Handlebars would separate the text before the placeholder placing it into one token, then the placeholder itself would be placed into another token, and lastly all the text after the placeholder would be placed into a third token. This is because those pieces need to both retain the order of the template but they also need to be processed differently.

This process is done using the `Handlebars.parse()` function, and what you get back is an object that contains all the segments or ′statements′.

To better illustrate what I am talking about, let′s create a list of paragraphs for each of the tokens taken out:

```
//Display Tokens
var tokenizer = Handlebars.parse(src);
var tokenStr = "";
for (var i in tokenizer.statements) {
    var token = tokenizer.statements[i];
    tokenStr += "<p>" + (parseInt(i)+1) + ") ";
    switch (token.type) {
        case "content":
            tokenStr += "[string] - \"" + token.string + "\"";
            break;
        case "mustache":
            tokenStr += "[placeholder] - " + token.id.string;
            break;
        case "block":
            tokenStr += "[block] - " + token.mustache.id.string;
    }
}
document.getElementById("tokens").innerHTML += tokenStr;
```

So we begin by running the templates source into `Handlebars.parse` to get the list of tokens. We then cycle through all the individual components and build up a set of human readable strings based on the segment's type. Plain text will have a type of "content" which we can then just output the string wrapped in quotes to show what it equals. Placeholders will have a type of "mustache" which we can then display along with their "id" (placeholder name). And last but not least, block helpers will have a type of "block" which we can then also just display the blocks internal "id" (block name).

Refreshing this now in the browser, you should see just a single 'string' token, with our template's text.

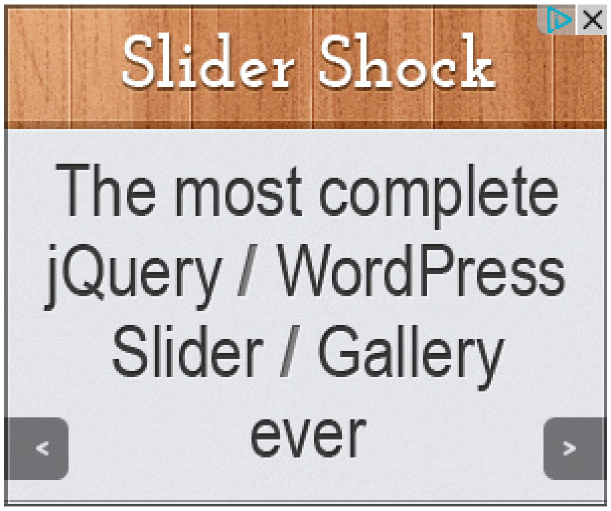
Operations

Once handlebars has the collection of tokens, it cycles through each one and "generates" a list of predefined operations that need to be performed for the template to be compiled. This process is done using the `Handlebars.Compiler()` object, passing in the token object from step 1:

```
//Display Operations
var opSequence = new Handlebars.Compiler().compile(tokenizer, {});
var opStr = "";
for (var i in opSequence.opcodes) {
    var op = opSequence.opcodes[i];
    opStr += "<p>" + (parseInt(i)+1) + ") - " + op.opcode;
}
document.getElementById("operations").innerHTML += opStr;
```

Here we are compiling the tokens into the operations sequence I talked about, and then we are cycling through each one and creating a similar list as in the first step, except here we just need to print the opcode. The opcode is the "operation’s" or the function’s ’name’ that needs to be run for each element in the sequence.

Back in the browser, you now should see just a single operation called ’appendContent’ which will append the value to the current ’buffer’ or ’string of text’. There are a lot of different opcodes and I don’t think I am qualified to explain some of them, but doing a quick search in the source code for a given opcode will show you the function that will be run for it.



The Function

The last stage is to take the list of opcodes and to convert them into a function, it does this by reading the list of operations and smartly concatenating code for each one. Here is the code required to get at the function for this step:

```
//Display Function
var outputFunction = new Handlebars.JavaScriptCompiler().compile(op
document.getElementById("source").innerHTML = outputFunction.toStri
Prism.highlightAll();
```

The first line creates the compiler passing in the op sequence, and this line will return the final function used for generating the template. We then convert the function to a string and tell Prism to syntax highlight it.

With this final code, your page should look something like so:

This function is incredibly simple, since there was only one operation, it just returns the given string; let's now take a look at editing the template and seeing how these individually straight forward steps, group together to form a very powerful abstraction.

Examining Templates

Let's start with something simple, and let's simply replace the word 'World' with a placeholder; your new template should look like the following:

```
<script id="dt" type="template/handlebars">
    Hello {{name}}!
</script>
```

And don't forget to pass the variable in so that the output looks OK:

```
//Display Output
var t = Handlebars.compile(src);
document.getElementById("output").innerHTML += t({name: "Gabriel"})
```

Running this, you will find that by adding just one simple placeholder, it complicates the process quite a bit.

> The complicated if/else section is because it doesn't know if the placeholder is in fact a placeholder or a helper method

If you were still unsure about what tokens are, you should have a better idea now; as you can see in the picture, it split out the placeholder from the strings and created three individual components.

Next, in the operations section, there are quite a few additions. If you remember from before, to simply output some text, Handlebars uses the 'appendContent' operation, which is what you can now see on the top and bottom of the list (for both "Hello " and the "!"). The rest in the middle are all the operations needed to process the placeholder and append the escaped content.

Finally, in the bottom window, instead of just returning a string, this time it creates a buffer variable, and handles one token at a time. The complicated if/else section is because it doesn't know if the placeholder is in fact a placeholder or a helper method. So it tries to see if a helper method with the given name exists, in which case it will call the helper method and set 'stack1' to the value. In the event it is a placeholder, it will assign the value from the context passed in (here named 'depth0') and if a function was passed in it will place the result of the function into the variable 'stack1'. Once that is all done, it escapes it like we saw in the operations and appends it to the buffer.

For our next change, let's simply try the same template, except this time without escaping the results (to do this, add another curly brace `"{{{name}}}"`)

Refreshing the page, now you will see it removed the operation to escape the variable and instead it just appends it, this bubbles down into the function which now simply checks to make sure the value isn't a falsy value (besides `0`) and then appends it without escaping it.

So I think placeholders are pretty straight forward, lets now take a look at using helper functions.

Helper Functions

There is no point in making this more complicated then it has to be, let's just create a simple function that will return the duplicate of a number passed in, so replace the template and add a new script block for the helper (before the other code):

```
<script id="dt" type="template/handlebars">
    3 * 2 = {{{doubled 3}}}
</script>

<script>
    Handlebars.registerHelper("doubled", function(number){
        return number * 2;
    });
</script>
```

I have decided to not escape it, as it makes the final function slightly simpler to read, but you can try both if you like. Anyways, running this should produce the following:

Here you can see it knows it is a helper, so instead of saying 'invokeAmbiguous' it now says 'invokeHelper' and therefore also in the function there is no longer an if/else block. It does still however make sure the helper exists and tries to fall back to the context for a function with the same name in the event it doesn't.

Another thing worth mentioning is you can see the parameters for helpers get passed in directly, and are actually hard coded in, if possible, when the function get's generated (the number 3 in the doubled function).

The last example I want to cover is about block helpers.

Block Helpers

Block helpers allow you to wrap other tokens inside a function which is able to set its own context and options. Let's take a look at an example using the default 'if' block helper:

```
<script id="dt" type="template/handlebars">
    Hello
    {{#if name}}
        {{{name}}}
    {{else}}
        World!
    {{/if}}
</script>
```

Here we are checking if "name" is set in the current context, in which case we will display it, otherwise we output "World!". Running this in our analyzer, you will see only two tokens even though there are more; this is because each block is run as its own 'template' so all the tokens inside it (like `{{{name}}}`) will not be part of the outer call, and you would need to extract it from the block's node itself.

Besides that, if you take a look at the function:

You can see that it actually compiles the block helper's functions into the template's function. There are two because one is the main function and the other is the inverse function (for when the parameter doesn't exist or is false). The main function: "program1" is exactly what we had before when we just had some text and a single placeholder, because like I mentioned, each of the block helper functions are built up and treated exactly like a regular template. They are then run through the "if" helper to receive the proper function which it will then append to the outer buffer.

Like before, it is worth mentioning that the first parameter to a block helper is the key itself, whereas the 'this' parameter is set to the entire passed in context, which can come in handy when building your own block helpers.

Conclusion

In this article we may not have taken a practical look at how to accomplish something in Handlebars, but I hope you got a better understanding of what exactly is going on behind the scenes which should allow you to build better templates and helpers with this new found knowledge.

I hope you enjoyed reading, like always if you have any questions feel free to contact me on Twitter (@GabrielManricks) or on the Nettuts+ IRC (#nettuts on freenode).

Like 51 people like this. Be the first of your friends.

Tags: handlebars.js

By Gabriel Manricks
This author has yet to write their bio.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more