Template Engines (Mustache, Handlebars etc)

**When To Use a JavaScript Templating Engine?**

You should use a JavaScript templating engine like Handlebars.js when:

* You use a JavaScript front-end framework like Backone.js, Ember.js, and the like; most front-end JavaScript frameworks rely on templating engines
* The application’s view (the HTML page or portions of the page) will be updated frequently, especially as a result of changes to the data either from the server via a REST API or from data on the client
* You have multiple tech stacks that depend on your data from the server and you want all the tech stacks to process the same data
* Your application has much interactivity and it is very responsive
* You are developing a single-page web application with multiple views
* You want to easily manage your HTML content; you don’t want your JavaScript code to contain important HTML markup.

**What is Mustache?**

Mustache is a simple template system that you can use when building websites and web apps. By using a template system, you can keep your back-end code separate from the markup that displays the front end to the user. This clean separation gives you many advantages. For example, it makes it easy for a designer to work on a website's visual design without the risk of messing up the site's code. It also makes it easy for you to change the design at a later point without impacting the back-end code.

One of Mustache's big plus points is that is logic-less, which means it keeps your templates very neat and tidy. There are no messy if ... then or looping constructs embedded within a Mustache template; it's all just markup and simple Mustache tags. All the logic is hidden away inside your data objects (and the code that creates or fetches them).

Another advantage of Mustache is that you're not tied to any particular language. Mustache processors are available in many languages, including JavaScript, PHP, Perl, Ruby, Python, Java and lots more.

You can use Mustache for practically any kind of template, including config files and even source code. However, this article concentrates on using Mustache to build HTML templates, which is a very popular use of the system.

Mustache is used by a lot of popular websites and web apps, including Twitter and OMGPOP (maker of the ever-popular Draw Something game).[Here's a list](https://github.com/janl/mustache.js/wiki/beard-competition) of sites that use Mustache.js, the JavaScript version of Mustache.

A basic Mustache example

What does a Mustache template look like? Here's a simple example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <h4>Product Info: {{name}}</h4>    <ul>    <li>Product: {{name}}</li>    <li>Colour: {{colour}}</li>    <li>Price: ${{price}}</li>  </ul> |

As you can see, this template is essentially HTML mixed with a few special Mustache tags. A Mustache tag begins with two opening braces ({{) and ends with two closing braces (}}).

As you might have guessed, the {{ and }} delimiters are where Mustache gets its name from!

Inside each tag is the name of a variable (in this case, name, colour, andprice). When Mustache processes the template, it replaces each variable name with an actual value.

But where does Mustache get the values for the variables in the template? The answer is that, when you pass the template to the Mustache processor, you also pass an object or hash containing the variable names and their associated values:

|  |  |
| --- | --- |
| 1  2  3  4  5 | {    "name": "SuperWidget",    "colour": "Green",    "price": "19.99"  } |

The exact format of the object or hash depends on the language you're using. The above example is a JavaScript (or JSON) object. But essentially you pass Mustache a list of name/value pairs.

When you pass the template and object to the Mustache processor, it combines the two and returns the final HTML, with the variable names replaced by their values:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <h4>Product Info: SuperWidget</h4>    <ul>    <li>Product: SuperWidget</li>    <li>Colour: Green</li>    <li>Price: $19.99</li>  </ul> |

Installing Mustache

The Mustache processor is available in a [wide range of languages](http://mustache.github.com/). As a general rule, you install Mustache like this:

Visit the GitHub page for your chosen language (for example, [JavaScript](https://github.com/janl/mustache.js)or [PHP](https://github.com/bobthecow/mustache.php)).

Click the ZIP button to download the repository Zip file.

Unzip the downloaded file, and move the resulting folder to your website.

Include the relevant library file (such as mustache.js or Mustache.php) in your web pages or server-side scripts.

There are some subtle differences between the Mustache implementations for different languages. Make sure you read the documentation specific to the Mustache processor you're using.

Running the Mustache processor

The syntax for calling the Mustache processor depends on the language you are using. Essentially though, you call the Mustache class's or object'srender() method, passing in the template string followed by the data object. The processor then combines the template with the data object to produce the final markup string, which it returns.

Here are some examples:

JavaScript:

|  |  |
| --- | --- |
| 1 | output = Mustache.render( template, data ); |

PHP:

|  |  |
| --- | --- |
| 1  2 | $m = new Mustache;  $output = $m->render( $template, $data ); |

Perl:

|  |  |
| --- | --- |
| 1  2 | use Template::Mustache;  $output = Template::Mustache->render( $template, $data ); |

A Mustache demo page

Let's get going with a simple demo page that shows Mustache in action. We'll use mustache.js — the JavaScript implementation of Mustache — so that we can run the demo straight in the browser, without needing to write any server-side code.

The JavaScript version of Mustache is very handy for Ajax-driven web apps. Your app can pull JSON data from the server using Ajax, then combine the data with a Mustache template to create the final markup for displaying in the page. You'll learn how to do this in the complete example at the end of the tutorial.

Grab mustache.js

To start building the demo page, you first need to install the JavaScript Mustache processor:

Create a folder somewhere on your computer

Download and unzip the [JavaScript Mustache processor](https://github.com/janl/mustache.js)

Copy the resulting mustache.js file into your folder

Expressions and functions

With most languages, you're not limited to using literal values in your Mustache data objects. You can also use expressions and functions as values. This means your object can contain properties such as:

|  |  |
| --- | --- |
| 1 | "price": parseFloat(netPrice+tax).toFixed(2) |

...or:

|  |  |
| --- | --- |
| 1 | "price": calcPrice |

(where calcPrice() is a function). Mustache then replaces the variable tag with the result of the expression, or the return value of the function, in the output.

One notable exception to this rule is JSON, which doesn't allow expressions as property values.

Accessing object properties and methods

Your data object can also contain other objects — for example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {    "name": "SuperWidget",    "colour": "Green",    "price": {      "regular": "19.99",      "discount": "14.99"    }  } |

To access a property or method of an object from within your Mustache template, you can use the dot (.) notation, like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <h4>Product Info: {{name}}</h4>    <ul>    <li>Product: {{name}}</li>    <li>Colour: {{colour}}</li>    <li>Regular Price: ${{price.regular}}</li>    <li>Discount Price: ${{price.discount}}</li>  </ul> |

Combining the above object and template produces the following output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <h4>Product Info: SuperWidget</h4>    <ul>    <li>Product: SuperWidget</li>    <li>Colour: Green</li>    <li>Regular Price: $19.99</li>    <li>Discount Price: $14.99</li>  </ul> |

Missing variables

If you use a variable name in your template that doesn't appear in the corresponding data object, Mustache simply replaces the tag with an empty string. For example, this Mustache template:

|  |  |
| --- | --- |
| 1 | Hello, {{first}} {{last}}! |

when combined with this data object:

|  |  |
| --- | --- |
| 1  2  3 | {    "first": "John",  } |

produces this result:

|  |  |
| --- | --- |
| 1 | Hello, John ! |

HTML escaping

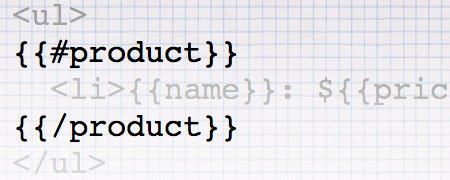
Mustache automatically replaces certain HTML characters, such as < and >, with their equivalent HTML entities, such as &lt; and &gt;. If you don't want Mustache to HTML-escape a value, put triple braces around the variable name instead of double braces, like this:

|  |  |
| --- | --- |
| 1 | {{{name}}} |

Alternatively, you can place an ampersand after the opening double brace, like this:

|  |  |
| --- | --- |
| 1 | {{&name}} |

Working with sections



Sections let you add more power to your templates. Using sections, you can display a chunk of markup only if a certain condition is true (or false). You can also create repeating sections, which let you display lists and tables of data.

The basic syntax for a section looks like this:

|  |  |
| --- | --- |
| 1  2  3 | {{#sectionName}}    (more markup in here)  {{/sectionName}} |

Conditional sections

A conditional section is a block of markup that is displayed only if a certain condition is true. Here's an example.

Say your template looks like this:

|  |  |
| --- | --- |
| 1  2  3 | {{#inStock}}    <li><a href="#">Buy Now!</a></li>  {{/inStock}} |

If your data object looks like this:

|  |  |
| --- | --- |
| 1  2  3 | {    "inStock": true  } |

...then the output will look like this:

|  |  |
| --- | --- |
| 1 | <li><a href="#">Buy Now!</a></li> |

On the other hand, if your data object looks like this:

|  |  |
| --- | --- |
| 1  2  3 | {    "inStock": false  } |

Or like this:

|  |  |
| --- | --- |
| 1  2 | {  } |

...then nothing will be output.

Inverted sections

Inverted sections are the opposite of conditional sections. With an inverted section, the section's content is only output if the section's variable is false.

You create an inverted section by replacing the "#" character with a "^"character. For example:

|  |  |
| --- | --- |
| 1  2  3 | {{^inStock}}    <li>Sorry, out of stock.</li>  {{/inStock}} |

The li element in the above example is only output if the value of theinStock variable is false.

To try out conditional sections and inverted sections, press the button below:

[View Demo »](http://www.elated.com/res/File/articles/development/easy-html-templates-with-mustache/sections-conditional.html)

Press the Process Template button. Since the value of inStock is true, the "Buy Now!" link is displayed. Now try changing inStock to false in thedata object:

|  |  |
| --- | --- |
| 1 | "inStock": false |

Now when you press the Process Template button, the "Sorry, out of stock" message is displayed instead.

Repeating sections

Repeating sections are handy when you want to display a list or table of related data in the page. You create a repeating section like this:

Add a section ( for example, {{#mySection}} ... {{/mySection}} ) to your Mustache template.

Inside the section, place the markup and any variable tags that you want to display for each item in the list.

Add a list (or array) property to your data object. Give the property the same name as the section (for example, mySection).

Each item in the list should be an object containing the properties corresponding to the variable tags you added in Step 2.

The Mustache processor then loops through the objects in the list. For each object, it replaces the variable tags in the section with the properties of the object, and outputs the section's markup.

Here's an example. First, the Mustache template:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <ul>  {{#product}}    <li>{{name}}: ${{price}}</li>  {{/product}}  </ul> |

Next, the data object:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | {    "product": [      {        "name": "SuperWidget",        "price": "19.99"      },      {        "name": "WonderWidget",        "price": "24.99"      },      {        "name": "MegaWidget",        "price": "29.99"      }    ]  } |

And finally, the resulting output:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <ul>    <li>SuperWidget: $19.99</li>    <li>WonderWidget: $24.99</li>    <li>MegaWidget: $29.99</li>  </ul> |

You can try out this example by pressing the button below:

[View Demo »](http://www.elated.com/res/File/articles/development/easy-html-templates-with-mustache/sections-repeating.html)

Adding comments in Mustache

To insert a comment in a Mustache template, use the following syntax:

|  |  |
| --- | --- |
| 1 | {{! this is a comment }} |

The whole comment tag is ignored by the Mustache processor.

Including Mustache templates with partials

Partials allow you to include one Mustache template inside another. This lets you keep your templates modular and organized.

In the JavaScript version of Mustache, you create an object containing all your partials:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var partials = {    myPartial: "templateString",    anotherPartial: "templateString"    ...  }; |

Then, to insert a partial at a given point in a template, you use the syntax:

{{>partialName}}

So to include the partial called "myPartial", you'd write:

|  |  |
| --- | --- |
| 1 | {{>myPartial}} |

You can also include a partial within another partial, which lets you make nested includes.

Then, when you run the Mustache processor, you pass your partials object as the third argument, like this:

|  |  |
| --- | --- |
| 1 | html = Mustache.render( template, data, partials ); |

Here's an example of partials in action. First, we'll create a couple of partials,productInfo and buyLink:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var partials = {      productInfo: "                \    <ul>                          \      <li>Name: {{name}}</li>     \      <li>Colour: {{colour}}</li> \      <li>Price: ${{price}}</li>  \      {{>buyLink}}                \    </ul>                         \    ",      buyLink: "                            \      {{#inStock}}                        \        <li><a href='#'>Buy Now!</a></li> \      {{/inStock}}                        \      {{^inStock}}                        \        <li>Sorry, out of stock.</li>     \      {{/inStock}}                        \    "  }; |

productInfo displays a product's name, colour and price, It also includes the buyLink partial, which displays the "Buy Now!" link (or the "Sorry, out of stock" message if the product is out of stock).

Now let's create our template:

|  |  |
| --- | --- |
| 1 | var template = "<h3>Product: {{name}}</h3> {{>productInfo}}"; |

This template displays the product name inside an h3 element, then includes the productInfo partial to display the product info.

Here's the data object that we'll use for our product data:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var data = {    "name": "SuperWidget",    "colour": "Green",    "price": "19.99",    "inStock": true  } |

Finally, we call the Mustache render() method to create the final markup:

|  |  |
| --- | --- |
| 1  2 | html = Mustache.render( template, data, partials );  alert( html ); |

This displays the following alert box:

<h3>Product: SuperWidget</h3>

<ul>

<li>Name: SuperWidget</li>

<li>Colour: Green</li>

<li>Price: $19.99</li>

<li><a href='#'>Buy Now!</a></li>

</ul>

Why Use Handlebars.js (out of the eight or more templating engines)?

Handlebars.js is an extension of the Mustache JavaScript templating language; it supersedes Mustache.js.

The reasons you should use Handlerbars.js follow:

Handlebars is one of the most advanced (pre-compiling and the like), feature-rich, and popular of all the JavaScript templating engines, and it has the most active community.

Handlebars is a logic-less templating engine, which means there is little to no logic in your templates that are on the HTML page. The most important use of Handlebars, and any templating engine, is to keep your HTML pages simple and clean and decoupled from the logic-based JavaScript files, and Handlebars serves this purpose well. Indeed, Dust.js is also a logic-less templating engine and a [worthy alternative](http://engineering.linkedin.com/frontend/client-side-templating-throwdown-mustache-handlebars-dustjs-and-more) to Handlebars.js.

Moreover, the cutting-edge JavaScript frameworks Meteor.js and Derby.js, are expected to become mainstream in the coming months, and both use Handlebars.js. To be clear: Meteor.js uses Handlebars.js and Derby.js “template syntax is largely based on Handlebars” template syntax. And Ember.js uses Handlebars, too.

While Backbone.js is packaged with Underscore.js templating engine, it is super easy to use Handlebars.js with Backbone.js.

Therefore, the experience and knowledge you will have gained from learning Handlebars.js now will be well worth it, if you use, or plan to use, any of the noted JS frameworks.

Handlebars.js Overview

Now that we have seen how to use Handlebars in a simple application, let’s study Handlebars in detail.

How Handlebars.js Works?

As noted in the introduction: Handlebars.js is a compiler built with JavaScript that takes any HTML and Handlebars expression and compiles them to a JavaScript function. This derived JavaScript function then takes one parameter, an object—your data—and it returns an HTML string with the object properties’ values inserted (interpolated) into the HTML. So, you end up with a string (HTML) that has the values from the object properties inserted in the relevant places, and you insert the string on a page.

This sounds way more complex that it is, so let’s take a closer look.

The 3 Main Parts of Handlebars Templating

To use Handlebars, first you link to the Handlebars.js file in the head block of your HTML page, just like you do for jQuery or any .js files.. Then there are 3 main pieces of code you use for Handlebars templating:

Handlebars.js Expressions   
A simple Handlebars expression is written like this (where “content” can be a variable or a helper function with—or without—parameters:

|  |  |
| --- | --- |
|  | {{ content }} |

Or like this, in the case of Handlebars block expressions (which we will discuss in detail later):

|  |  |
| --- | --- |
|  | {{#each}} ​ |
|  | HTML content and other Handlebars expresion go here. |
|  | {{/each}} |

Below is a Handlebars expression with HTML. The customerName variable is the property that will be interpolated (its values will be inserted in place) by the Handlebars.compile function:

|  |  |
| --- | --- |
|  | <div> Name: {{ customerName }} </div> |

The output will be the following (if the customerName variable has the value “Richard”):

Richard

Since you have to pass the Handlebars expression (with any containing HTML) to the Handlebars.compile function, a script tag is used to enclose each Handlebars template when they are on the HTML page. Indeed, the script tag is not necessary when a template is in its own HTML file, but it is necessary when the Handlebars template is on the page along with other Handlebars template and other HTML content.

— Script Tag  
Handlebars templates are embedded in script tags (where the script tag’s type property is set to “text/x-handlebars-template”). The script tag is similar to the script tag you normally use to embed JavaScript in the HTML page, except the type attribute is different.   You retrieve the HTML content from the script tag and pass it to the Handlebars compiler.

Here is an example of the Handlebars script tag:

|  |  |
| --- | --- |
|  | <script id="header" type="text/x-handlebars-template">​ |
|  | <div> Name: {{ headerTitle }} </div>​ |
|  | ​</script> |

Data (or Context)  
The second piece of code in Handlebars templating is the data you want to display on the page. You pass your data as an object (a regular JavaScript object) to the Handlebars function. The data object is called the context. And this object can be comprised of arrays, strings, numbers, other objects, or a combination of all of these.

If the data object has an array of objects, you can use Handlebars each helper (more on helpers later) function to iterate the array, and the current context is set to each item in the array.

Here are examples of setting up the data object and how to iterate it in a Handlebars template.

— Data object with array of objects

|  |  |
| --- | --- |
|  | //The customers object has an array of objects that we will pass to Handlebars: |
|  | ​var theData = {customers:[{firstName:”Michael”, lastName:”Alexander”, age:20}, {firstName:”John”, lastName:”Allen”, age:29}]}; |

You can use the each helper to iterate the customer’s object like this:

|  |  |
| --- | --- |
|  | <script id="header" type="text/x-handlebars-template">​ |
|  | {{#each customers}} // Note the reference to the customers object​ |
|  | <li> {{ firstName }} {{ lastName }} </li>​ |
|  | {{/each}} |
|  | ​</script> |

Or, since we are passing the customer’s object as an array of objects, we can use a block helper (more on block helpers later) statement like this and reference the customers directly:

|  |  |
| --- | --- |
|  | <script id="header" type="text/x-handlebars-template">​ |
|  | {{#customers}} // In this example, because the customer’s ​ |
|  | <li> {{ firstName }} {{ lastName }} </li>​ |
|  | {{/customers}} |
|  | ​</script> |

— Data object with Strings

|  |  |
| --- | --- |
|  | // In this example, the data object contains properties with strings:​ |
|  | var theData = {headerTitle:"Shop Page", weekDay:”Wednesday”}; |
|  | ​ |
|  | ​<script id="header" type="text/x-handlebars-template">​ |
|  | <div> {{ headerTitle }} </div>​ |
|  | Today is {{weekDay}} |
|  | ​</script> |

The Handlebars Compile Function  
The last piece of code we need for Handlebars templating is actually a two-step execution:  
1. Compile the template with the Handlebars.compile function.  
2. Then use that compiled function to invoke the data object passed to it (it takes a data object as its sole parameter). And this will return an HTML string with the interpolated object values inserted into the HTML.

In short:  
The Handlebars.compile function takes the template as a parameter and it returns a JavaScript function. We then use this compiled function to execute the data object and return a string with HTML and the interpolated object values. Then we can insert the string into the HTML page.

 Here are the 3 pieces together:   
1. On the HTML age: Setup the templates by using Handlebars expressions, and add the templates to a script tag (if using script tags: templates in individual HTML files don’t need script tags):

|  |  |
| --- | --- |
|  | <script id="header" type="text/x-handlebars-template">​ |
|  | <div> {{ headerTitle }} </div>​ |
|  | Today is {{weekDay}} |
|  | ​</script> |

  2. In the JavaScript file: Initialize the data object

|  |  |
| --- | --- |
|  | var theData = {headerTitle:"Shop Page", weekDay:”Wednesday”}; |
|  | ​ |
|  | ​// Retrieve the HTML from the script tag we setup in step 1​ |
|  | // We use the id (header) of the script tag to target it on the page​ |
|  | var theTemplateScript = $("#header").html(); |

 3. Also in the JavaScript file: Then we use the Handlebars compile function to compile the templates.  
Compile the template retrieved from the script tag:

|  |  |
| --- | --- |
|  | // The Handlebars.compile function returns a function to theTemplate variable​ |
|  | var theTemplate = Handlebars.compile (theTemplateScript); |

Use the theTemplate () function returned by the compile function to generate the final string with interpolated object values. We pass the object data as a parameter. Then attach the resulting string with HTML to the page:

|  |  |
| --- | --- |
|  | $(document.body).append (theTemplate (theData)); |

This will return our HTML with the values from the object inserted in place, and the result will look like this:   
\_\_\_\_\_\_\_\_\_\_   
Shop Page  
Today is Wednesday  
\_\_\_\_\_\_\_\_\_\_

Learn Handlebars.js Syntax

Handlebars.js Expressions  
We saw the Handlebars Expressions above. Handlebars expressions are written like this (a double stash before, followed but the content to be evaluated, followed a double closing double stash):

|  |  |
| --- | --- |
|  | <div>{{ content goes here }} |

The customerName variable is the property (the expression to be evaluated by the Handlebars compiler) that will be interpolated (its values will be inserted in place) by the Handlebars compiled function, when it executes:

|  |  |
| --- | --- |
|  | <div> Name: {{ customerName }} </div> |

Comments  
This is how you add comments in a Handlebars template:

|  |  |
| --- | --- |
|  | {{! Whatever is inside this comment expression will not be outputted }} |

And you can also use regular HTML comments, but they will be outputted on the HTML page source, as HTML comments of course:

|  |  |
| --- | --- |
|  | <!-- Regular HTML comments will be in the output --> |

Blocks  
Blocks in Handlebars are expression that has a block, an opening {{# }} followed by a closing {{/ }}.

We cover blocks in detail later; this is the syntax for a block:

|  |  |
| --- | --- |
|  | {{#each}} Content goes here. {{/each}} |

Here is an if block

|  |  |
| --- | --- |
|  | {{#if someValueIsTrue}} Content goes here {{/if}} |

The words block and helper are sometimes used interchangeably because most built-in helpers are blocks, although there are function helpers and block helpers.

Paths (with dot notation)  
A path in Handlebars is a property lookup. If we have a name property that contains an object, such as:

|  |  |
| --- | --- |
|  | var objData = {name: {firstName: "Michael", lastName:"Jackson"}} |

  We can use nested paths (dot notation) to lookup the property you want, like this:

|  |  |
| --- | --- |
|  | {{name.firstName}} |

Parent Path ../  
Handlebars also has a parent path ../ to lookup properties on parents of the current context. Thus:  
With a data object such as this:

|  |  |
| --- | --- |
|  | var shoesData = {groupName:"Celebrities", users:[{name:{firstName:"Mike", lastName:"Alexander" }}, {name:{firstName:"John", lastName:"Waters" }} ]}; |
|  |  |
|  | We can use the parent path ../ to get the groupName property: |
|  | ​<script id="shoe-template" type="x-handlebars-template">​ |
|  | {{#users}}​ |
|  | <li>{{name.firstName}} {{name.lastName}} is in the {{../groupName}} group.</li>​ |
|  | {{/users}} |
|  | ​</script> |

The rendered HTML will be:  
Mike Alexander is in the Celebrities group.  
John Waters is in the Celebrities group.

Context  
Handlebars refers to the object you passed to its function as the context. Throughout this article, we use “object data” and sometimes “data” or “object” to refer to the context object. All these words are used interchangeably from time to time, but you will no doubt understand that we are referring to the object being passed into the Handlebars function.

Triple Stash {{{ }}} For Non-escape HTML  
Ordinarily, you use Handlebars’s double stash {{ }} for expressions, and by default, Handlebars content in this standard double stash is escaped to “protect you against accidental XSS problems caused by malicious data passed from the server as JSON.” 1This ensures that nefarious code in the HTML cannot be injected into the page.   But sometimes you want the raw (un-escaped) HTML instead. For this, you can use Handlebars’s triple stash instead {{{ }}}. The triple stash tag signifies to handlebars to do not escape the HTML content contained in the triple stash.

Partials (sub-templates)  
Sometimes you have to render a section of a template within a larger template. You use Partials to do this in Handlebars, and this is the partials expression:

|  |  |
| --- | --- |
|  | {{> partialName}} |

Let’s add a partial template to the basic Handlebars project we built earlier. We will addcolor and size under each shoe.

Links

<http://javascriptissexy.com/handlebars-js-tutorial-learn-everything-about-handlebars-js-javascript-templating/>

<http://www.elated.com/articles/easy-html-templates-with-mustache/>