Handlebars example template

Today, most of the web consists of dynamic applications where data changes frequently. As a result, there is a constant need to update the data rendered in the browser. This is where JavaScript templating engines come to the rescue and become so useful. They simplify the process of manually updating the view while improving the structure of the application, allowing developers to separate business logic from the rest of the code. Some of the most famous JavaScript template engines are Mustache, Underscore, EJS, and Handlebars. In this article, we will focus our attention on the steering wheel, discussing its main features. Handlebars: What it is and why to use it steering wheel is without the logic of templating the engine, which dynamically generates an HTML page. This is a Mustache extension with several additional features. The mustache is fully logic less, but the steering wheel adds minimal logic by using some helpers (such as if, with, probably, everyone and more) that we will discuss further in this article. In fact, we can say that the steering wheel is a stuffed mustache. The steering wheel can be loaded into the browser just like any other JavaScript file: <script src=/path/to/handlebars.min.js></script> If you ask why you should employ this template engine and not another one, you should take a look at its advantages. Here's a short list: Keeps the HTML page clean and separates templates without logic from business logic in JavaScript files. Thus improving the structure of the application (as well as its ease of maintenance and scalability) Simplifies the task of manually updating the data in the view It is used in popular frameworks and platforms such as Ember.js, Meteor.js, Derby.js and Ghost I hope this short summary will help you decide whether it is worth using the steering wheel or not. How does it work? As shown in the diagram above, the way the steering wheel works can be summarized as follows: The handlebar takes the template with variables and compiles it into the function This function is then executed by passing the JSON object as an argument. This JSON object is known as context and contains the values of variables used in the Template On its execution, the function returns the required HTML code when you replace the template variables with the appropriate values to understand the above process, let's start with a demo that explains in detail all the above steps. Template templates can be saved both in an HTML file and separately. In the first case, they appear inside <script> tag with a type=text/x-handlebars-template attribute and an ID. The variables are written in double curly braces {{}} and are known as expressions. Here is an example: <script id=handlebars-demo type=text/x-handlebars-template> <div> My name is {{name}}. I am a {{occupation}}. </div> </script> with this tag in place, we can see what we need to do to use it. In a JavaScript file, we must first download the template from an HTML document. In the following example, we'll use the template ID to do this. After downloading the template, we can compile it using the Handlebars.compile() method, which returns the function. This function is then executed by passing the context as an argument. When execution is complete, the function returns the desired HTML code with all variables replaced by the corresponding values. At this point, we can inject HTML into our website. Converting this description to code results in the following code snippets: template var = $('#handlebars-demo').html(); var templateScript = Handlebars.compile(template); var context = { name : Ritesh Kumar, occupation : developer }; var html = templateScript(context); $(document.body).append(html); A live demo of this code can be found in this Codepen Demo Syntax Now it's time to dive a little deeper into the steering wheel. We will go through some important terms and syntax that make up the core of the steering wheel. Expressions We have already seen expressions in the above section. Variables used inside templates are surrounded by double braces {{}} and are called expressions: My name is {{name}} Html Escaping Handlebars can dissuasive from the value returned by the expression. For example, the character < is= converted= into= &lt.= if= you= don't= want= handlebars= to= escape= a= value,= you= have= to= surround= the= variable= using= triple= curly= braces= {{{variablename}}}.= for= example,= when= the= following = template:= i= am= learning= {{language}}.= it= is= {{{adjective}}}.= is= used= with= the= context= variable= reported= below:= var= context={ language= :=><h3>Managers</h3>, adjective: <h3>Amazing</h3> } The resulting HTML code will be: I'm learning <h3>Managers</h3>. It is recommended that the <h3>Amazing</h3> Live Demo that shows this feature can be found in this Codepen demo we can also post comments inside the steering wheel templates. The syntax for comments on the steering wheel is {{! TypYourCommentHere}}. However, any comment that has }} in it or any other symbol that has special meaning in the steering wheel should be saved in the form {{!--TypeYourCommentHere--}}. Comments on the steering wheel are not visible in HTML format, but if you want to show them, you can use the standard HTML comment: <!--comments--> . If we apply all these concepts to the template we use, we can come up with the code shown below: I'm learning {{language}}. This is {{!--adjective--}} If you use the previous template with the context variable reported below: var context = { language : Handlebars, adjective: awesome } THE HTML output will be: I'm learning the steering wheel. This will cause the browser to not display the content in the standard HTML comment. A demo of this example can be found here. Blocks in the steering wheel are expressions that open a block ({{# }}) and close ({{/}}). We will study this topic in depth later, on the helpers. For now, take a look at how the if block is written: {{#if boolean}} Some content content {{/if}} The Steering Wheel Path supports both normal and nested paths, so you can search for nested properties below the current context. The steering wheel also supports .. / path segment. This segment refers to the scope of the parent template, not to one level up in the context. To better understand this topic, we'll use the following example, where we use each helper (discussed in detail later). As you might expect, the latter iterates over the elements of the array. In this example, we'll use the following template: This article is available in {{website.name}}.
 {{#each name}} I'm {{.. /occupation}}. My name is {{firstName}} {{lastName}}.
 {{/everyone}} By providing this context variable: context var = { occupation : developer, website : { name : sitepoint } names : [{firstName : Ritesh, lastName : Kumar}, {firstName : John , lastName : Doe}] } we will get the output shown below: This article is available on the page. I'm a developer. My name is Ritesh Kumar. I'm a developer. My name is John Doe. Like other small fragments we've created so far, this example is available on Codepen Helpers Even though the steering wheel is a template engine with no logic, you can execute simple logic using helpers. The steering wheel helper is a simple identifier that can be followed by parameters (separated by a space), as shown below: {{#helperName parameter1 parameter2 ...}} Content here {{/helperName}} Each parameter is a steering wheel expression. These helpers can be obtained from any context in the template. Block, helper, and block helper terms are sometimes used interchangeably because most built-in helpers are blocks, although there are feature helpers that differ slightly from block helpers. We'll discuss them while dealing with custom helpers. Some built-in helpers are, if, everyone, I guess, and with. Let's find out more. Each helper each helper is used to iterate through an array. The helper syntax is {{#each ArrayName}} YourContent {{/each}} . We can refer to individual elements of an array by using the keyword that's inside the block. You can render an array element index by using {{@index}}. The following example illustrates the use of each helper. If we use the following template: {{#each countries}} {{@index}} : {{this}}
 {{/everyone}} {{#each names}} Name : {{firstName}} {{lastName}}
 {{/each}} in combination with this context variable: context var = { countries:[Russia,,,USA], names : [{firstName:Ritesh,lastName:Kumar}, {firstName:John,lastName:Doe}] } then the output will be: 0 : Russia 1 : India 2 : USA Name : Ritesh Kumar Name: John Doe Live Demo in this example can be found on Codepen. if the If helper is similar to the if statement. If the condition is truey, the handlebar will render the block. You can also specify a section of the template known as else section, using {{else}}. Unless the midfielder is the inverse of the midfielder if. if. renders the block when the condition is evaluated at the falsa value. To show how the if helper works, consider the following template: {{#if}} Countries are present. {{else}} Countries are not present. {{/if}} If you enter the following context variable: var context = { countries: [] } We get the result reported below: Countries are not present. This is because an empty array is a falsa value. If you want to play with helpers, you can take a look at the live demo I created on Codepen. Custom helpers can create their own helpers to execute complex logic by using the expression system that provides the handlebars. There are two types of helpers: function helpers and block helpers. The first definition is intended for a single expression, while the latter is used for block expressions. The arguments supplied to the callback function are parameters written after the helper name, separated by a space. Helpers are created using the handlebars.registerHelper() method: Handlebars.registerHelper(HelperName, function(arguments){ }) Custom function helper the function helper syntax is {{helperName parameter1 parameter2 ...}}. To better understand how to continue the implementation, let's create a function helper named studyStatus that returns the string that will be passed if passingYear < 2015= and= not= passed= if= passingyear=> = 2015: Handlebars.registerHelper(studyStatus, function(passingYear) { if(passingYear < 2015)= {= return= passed;= }= else= {= return= not= passed;= }= })= in= our= example= the= parameter= is= only= one.= however,= if= we= want= to= pass= more= parameters= to= the= helper's= callback= function= we= can= write= them= in= the= template= after= the= first= parameter= separated= by= a= space.= let's= develop= an= example= with= this= template:= {{#each= students}}= {{name}}= has= {{studystatus=>
 {{name}}= has= {{studystatus=>
 {{/each}} and with the following context variable: { students:[{name : John, passingYear : 2013}, {name : Doe , passingYear : 2016}] } In this case, the output will be as follows: John passed. Doe did not pass. A live presentation of this example is available here. Custom helper block custom block helpers are used in the same way as function helpers, but the syntax is slightly different. The block helper syntax is{ {{#helperName parameter1 parameter2 ...}} Your content here {{/helperName}} When you register a custom block helper, the handlebar automatically adds the option object as the last parameter to the callback function. This option object has an fn() method that allows us to temporarily change the context of an object to access a specific property. Let's change the example of the previous section with a block helper named studyStatus, but with the same context variable: Handlebars.registerHelper(studyStatus, function(data, options){ var len = data.length; var returnData=; for(var i=0;i<len;i++){ 2015) ? passed : not passed; returnData = returnData + 2015)= passed= := not= passed;= returndata=returnData +=></ 2015) ? passed : not passed; returnData = returnData + > > } return data; }); var context = { students:[{name : John, passingYear : 2013}, {name : Doe , passingYear : 2016}] } If this code is used in conjunction with the template defined below {{#studyStatus students}} {{name}} has {{passingYear}} {{/studyStatus}} we get the following result: John passed. Doe did not pass. And here's a codepen demo. Partial partial steering wheel templates are templates that can be shared by different templates. They are saved as {{> partialName}}. Before using them in html, we need to register partial using the Handlebars.registerPartial() method. The following example will help you understand how to register a partialTemplate name: Handlebars.registerPartial('partialTemplate', '{{language}} is {{adjective}}. You are reading this article on {{website}}..); var context={ language : Handlebars, adjective: awesome } When used with the template defined below {{> partialTemplate website=sitepoint}}
 {{> partial web page=www.sitepoint.com}} This will give the following result: The steering wheel is amazing. You are reading this article on sitepoint steering wheel is awesome. You are reading this article www.sitepoint.com live demo of this code can be found in this Codepen demo. Precompilation As we've seen, the first thing the steering wheel does is build the template into a function. This is one of the most expensive operations to perform on the client. We can improve application performance if you precompile templateScript and then send the compiled version to the client. In this case, the only task that must be performed on the client will be to perform this function. Because a precompiled file is a script, we can load the script in HTML as a regular file. Let's see how all this can be done. First, you need to install the steering wheel globally using npm to install the steering wheel -g. Make sure that different templates are saved in separate files with different file names and with .handlebars extensions (for example, demo.handlebars). No need to use <script></script>instead: <script src=handlebars.runtime.js></script> <script src=path/to/templatesCompiled.js></script> Now we can use the template that was originally present in demo.handlebars using the following code: var context = { name : Ritesh Kumar, occupation : developer } var templateScript = Handlebars.demo(context); $(document.body)append(templateScript); The final output will be as follows: My name is Ritesh Kumar. I'm a developer. This method will greatly improve application performance and page load time also decreases as we use runtime build managers, which is lighter than the entire library. The code for this entire pre-build demo is available on GitHub. ConclusionS In this article, we discussed the basic concepts of steering. We also examined its frequently used functions and syntax. I hope you liked this tutorial and you will use the demos included to have a good understanding of this topic. I look forward to your comments. If you don't want to download the library but still want to try it out, you can play with the steering wheel online . .