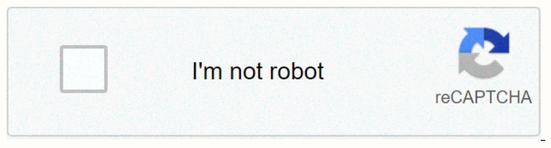

Continue

Js template literal variable

The literals of the model are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them. They were called model strings in earlier editions of the ES2015 specification. 'String text' 'string text' 'string text line 1 string text line 2' 'string text ${expression} string text'string '{expression} string template literals are included by the backtick character (' ') (accent grave) instead of double or single quotes. Template literals can contain placeholders. These are indicated by dollar sign and keys (${expression}). Expressions in placeholders and text between backticks are passed to a function. The default function only concatenates the parts into a single sequence. If there is an expression before the literal template (tag here), this is called the marked template. In this case, the tag expression (usually a function) is called with the literal template, which you can manipulate before exiting. To escape a backtick on a literal model, place a backbar (\) before the backtick. '\" === '" // --> true multi-line strings Any new line characters inserted into the font are part of the literal model. Using normal strings, you would have to use the following syntax to get multi-line strings: console.log ('string text line 1' + 'string text line 2'); string text line 1 // string text line 2 Using template literals, you can do the same: console.log ('string text line 1 line of string text 2'); string line 1 // string text line 2 Expression interpolation To embed expressions within normal strings, you would use the following syntax: leave the = 5; leave b = 10; console.log ('Fifteen is ' + (a + b) + ' andnot ' + (2 * a + b) + '.'); Fifteen is 15 years old and// not 20. Now, with the model literals, you are able to make use of syntactic sugar, making substitutions as this more readable: leave the = 5; leave b = 10; console.log ('Fifteen is ${a + b} and not ${2 * a + b}.'); Fifteen is 15 years old and// not 20. Nesting models In certain cases, nesting a model is the easiest (and perhaps most readable) way to have configurable strings. Within an indented model, it is simple to allow internal backticks by simply using them within a ${ } placeholder within the model. For example, if the condition is true, then return this literal template. In ES5: leave classes = 'header'; += classes (isLargeScreen() ? '' : item.isCollapsed? 'icon-expander': 'icon-collapser'); In ES2015 with template literals and no nesting: const classes = 'header ${ isLargeScreen() ? '' : (item.isCollapsed? 'icon-expander' : 'icon-collapser') }'; In ES2015 with nested template literals: const classes = 'header ${ isLargeScreen() ? '' : 'icon-${item.isCollapsed ? 'expander' : 'collapser'}' }'; Tagged templates A more advanced form of literal models are marked templates. Tags allow you to analyze model literals with a one The first argument of a tag function contains an array of sequence values. The other arguments are related to expressions. The tag function can then perform any operations on these arguments as it wants and return the manipulated sequence. (Alternatively, it may return something completely different, as described in one of the following examples.) The name of the function used for the tag can be whatever you want. leave the person = 'Mike'; leave age = 28; function myTag (strings, personExp, ageExp) { let str0 = strings[0]; // That let str1 = strings[1]; is a // There is technically a string after // the final expression (in our example), // but it is empty (), so disregard. leave str2 = strings[2]; leave ageStr; if (ageExp > 99){ ageStr = 'centennial'; } another { ageStr = 'young'; } // We can even return a string constructed using a literal return of model '${str0}${personExp}${str1}${ageStr}'; } leave output = myTag'That ${ person } is an age ${}'; console.log (output); That Mike is a young tag functions doesn't even need to return a rope! function model (strings, ... keys) { return (function(... values) { let dict = values[values.length - 1] || {}; leave result = [strings[0]; keys.forEach (function(key, i) { leave the value = Number.isInteger(key) ? values[key] : dict[key]; result.push(value, strings[i +1]); }); return.join(''); }); } leave t1Closure = model'${0}${1}${0}!'; let t1Closure = model([,,!,0,1,0); t1Closure('Y', 'A'); Yay! leave t2Closure = model'${0} ${'foo'}!'; let t2Closure = template([, !,0,foo); t2Closure('Hello', {foo: 'World'}); Hello World! let t3Closure = template 'I'm ${'name'}. I'm almost ${'old'} years old.'; let t3Closure = model([I am, . I'm almost, years old.], name, age); t3Closure('foo', {name: 'MDN', age: 30}); I'm MDN. I'm almost 30. t3Closure ({name: 'MDN', age: 30}); I'm MDN. I'm almost 30. Raw Strings The special raw property, available in the first argument for the tag function, allows you to access the raw strings as they have been inserted, without processing escape sequences. function tag (strings) { console.log(strings.raw[0]); } tag'string text line 1 string text line 2'; registers string text line 1 string text line 2 / // including the two characters '\' and 'n' In addition, the String.raw() method exists to create raw strings — just as the default function of the model and the string concatenation would create. let str = String.raw'Hi${2+3}!'; Hi5! str.length; 6 Array.from(str).join(''); H,i,\,n,5,! Tagged models and escape sequences behavior ES2016 From ECMAScript 2016, tagged models are in accordance with the rules of the following escape sequences: Unicode leaks initiated by \u, e.g. Unicode code point leaks indicated by \u{}, for example \u{2F804} Hexadecimal leaks initiated by \x, e.g. \xA9 Octal literal leaks initiated by \0o and followed by a more digits, e.g. \0o251 \0o251 means that a model marked as the following is problematic, because, by ECMAScript grammar, an analyzer looks for valid unicode escape sequences, but finds malformed syntax: latex'\unicode' // Launches in older ECMAScript versions (ES2016 and earlier) // SintaxError: ES2018 malformed Unicode string leak sequence revision of illegal escape sequences Tagged models should allow the embedding of languages (e.g. DSLs, or LaTeX), where other escape sequences are common. The ECMAScript Template Literal Revision proposal removes the syntax constraint of the ECMAScript escape sequences from tagged templates. However, illegal escape sequences should still be represented in the baked representation. They will appear as an undefined element in the baked array: latex(str) { return { cooked: str[0], raw: str.raw[0] } } latex'\unicode' // { baked: undefined, raw: \\unicode } Note that the restriction of the escape sequence is only discarded from marked models — not from unmarked model literals: leave bad = 'bad escape sequence: \\unicode'; Browser compatibility Browser compatibility Update on GitHubDesktopMobileServerChromeEdgeFirefoxInternetExplorerSafariSafariAndroid webviewChrome for AndroidFirefox for AndroidOpera for AndroidSafari on iOSSamsung InternetNode.jsTemplate liteIralsChrome Full support 41Edge Full support 1 Full support 2 Full support Firefox 34IE No Support Full Support NoOpera 28Safari Full support 9WebView Full support 41Chrome Android Full support 41Firefox Full support 34Opera Android Full support 28Safari iOS Full support 9Samsung Internet Android Full support 4.0nodejs Support full 4.0.0 Landscape sequences allowed in model tagged literals Chrome Full support 62Edge Full support 79Firefox Full support 53IE Unsupported in Opera Full support 49Safari Full support 11WebView Full support 62Chrome Android Full support 62Firefox Android Full support 5 Full support android 46Safari iOS Full support 11Samsung Internet Full support 8.0nodejs Total support 8.10.0 Total support 8.10.0 Total support 8.0.0Disabled Starting version 8.0.0 : This feature is behind the runtime flag --harmony. Full Support Full Support No Support Without Experimental Support. Expect behavior to change in the future. Experimental. Expect behavior to change in the future. The user must explicitly enable this feature. The user must explicitly enable this feature. See also Thomas Jensen's photo on UnsplashTemplate strings are a powerful feature of modern JavaScript released in ES6. It allows us to insert/interpolate variables and expressions into strings without having to concatenate as in older versions of JavaScript. It allows us to create strings that are complex and contain dynamic elements. Another great thing that comes with strings from are the tags. Tags are functions that take a string and decomposed parts of the string as parameters and are great for converting strings to Entities. The syntax for creating template strings is by using backticks to delimit them. For example, we can write:This is a very simple example of a sequence of templates. All content is constant and there are no variables or expressions in it. To add variables and or expressions to a string, we can do the following. Interpolating variables and expressionsWe add a variable or expression by adding a dollar sign of $ and keys {} in the string '${variable}'. This is much better than the alternative in old javaScript where we had to concatenate a string like the following:As we can see it is easy to make syntax errors with the old concatenation syntax if we have complex variables and expressions. Therefore, the strings of the model are a great improvement over what we had before. If we want to use the backtick character in the string contents just put a \ before the backtick character in the string. => 'will have a literal tick-back\"Multiline Strings Another great feature of model strings is that we can have strings with multiple lines for better code readability. This cannot be done easily with the old style of strings. With the old style of strings, we had to concatenate each line of the string to put long strings in several lines as the following examples:As we can see, this will become really painful is that we have many more lines. It's very frustrating to have all these more signals and divide the string into multiple lines. With template strings, we don't have this problem:This is much better than concatenating strings together. It takes much less time to write code and a lower probability of syntax errors. It is also much more readable. Note that this method adds a new real like to the string, and if you don't want the string to have multiple lines in its final format, just put a \at the end of the line. Nesting modelsTemplata strings can be nested in each other. This is a great feature because many people want to create dynamic strings. Model strings allow us to nest regular strings or model sequences within model strings. For example, we can write the following:We can write the same thing with the expression interpolated in the string with template strings instead:This case is simple, but with complex combinations of strings, it is very powerful to be able to add logic to the way we build our strings. Photo of Brett Jordan in UnsplashTagged TemplatesWith template strings, we can add something called tags in front of a sequence of templates. They are functions that take a rope and decomposed parts of the string as parameters. A marked function decomposes the sequence of models into an array as the first parameter, with each array item as the constant parts of the sequence. The additional parameters of the function are all variables and expressions in the order in which they appear in the sequence. Tagged models are great for strings in anything you want, since it's just a regular function. However, it is a one function because the first parameter is an array containing the constant parts of the string. The rest of the parameters contain the returned values that each expression returns. This is great for manipulating the string and transforming the returned values to what we want. The return value of tags can be whatever you want. So we can return strings, matrices, objects or anything else. As we see in the function above, on the string we put next to the oldtag person, first we interpolate the name, then the age, then the sex. Thus, in the parameters, they also appear in this order — nameExp, ageExp, and genderExp. They are the evaluated versions of the name, age and gender following models. Since tags are functions, we can return whatever we want:As you can see, we can manipulate the data to return a boolean instead of a sequence. Template tags are a feature available only for template strings. To do the same thing with the old string style, we have to decommin the strings with our own string manipulation code and the built-in string manipulation functions, which is not as flexible as using tags. It decomposes the parts of a sequence into arguments for you that are passed in the template tags. Raw StringsTemplate strings have a special raw property that you can get from tags. The gross property gives us the string without escaping the special characters, so the property is called raw. To access the raw property of a string, we can follow the example:This is useful for viewing the string as it is with the special characters intact. There is also the String tag.raw where we can take a template sequence and return a string with the escape characters without actually escaping them to see the string in its full form. For example, with the String tag.raw, we can type:As we can see, we have all the characters of the string with the evaluated expressions, but we keep the escape characters as is. Support for wide-ranging SAssass template strings is supported by almost all browsers that are maintained regularly today. The only important thing that doesn't support this out of the box is Internet Explorer. However, browsers that do not support it can add it with Babel. Node.js also supports it in version 4 or later. Template strings are the new standard for JavaScript strings. It is much better than strings before ES6 in that it supports interpolation of expressions within strings. We also have marked templates that are just functions, with the decomposed parts of the model sequence as parameters. The constant sequence in the first parameter is the string in a decomposed array shape, and the evaluated versions of the expressions in the order in which they appeared in the sequence are the remaining arguments. Marked model functions can return any type of variable. With the raw property of a string, which is available in tags, you can get the string with the non-leak characters Escape. Escape.