CONTENTS

RELATED

CodeIgniter: Getting Started With a Simple Example

View

How To Install Express, a Node.js Framework, and Set Up Socket.io on a VPS

View

Tutorial Series: How To Code in JavaScript

34/37 Understanding Template ...

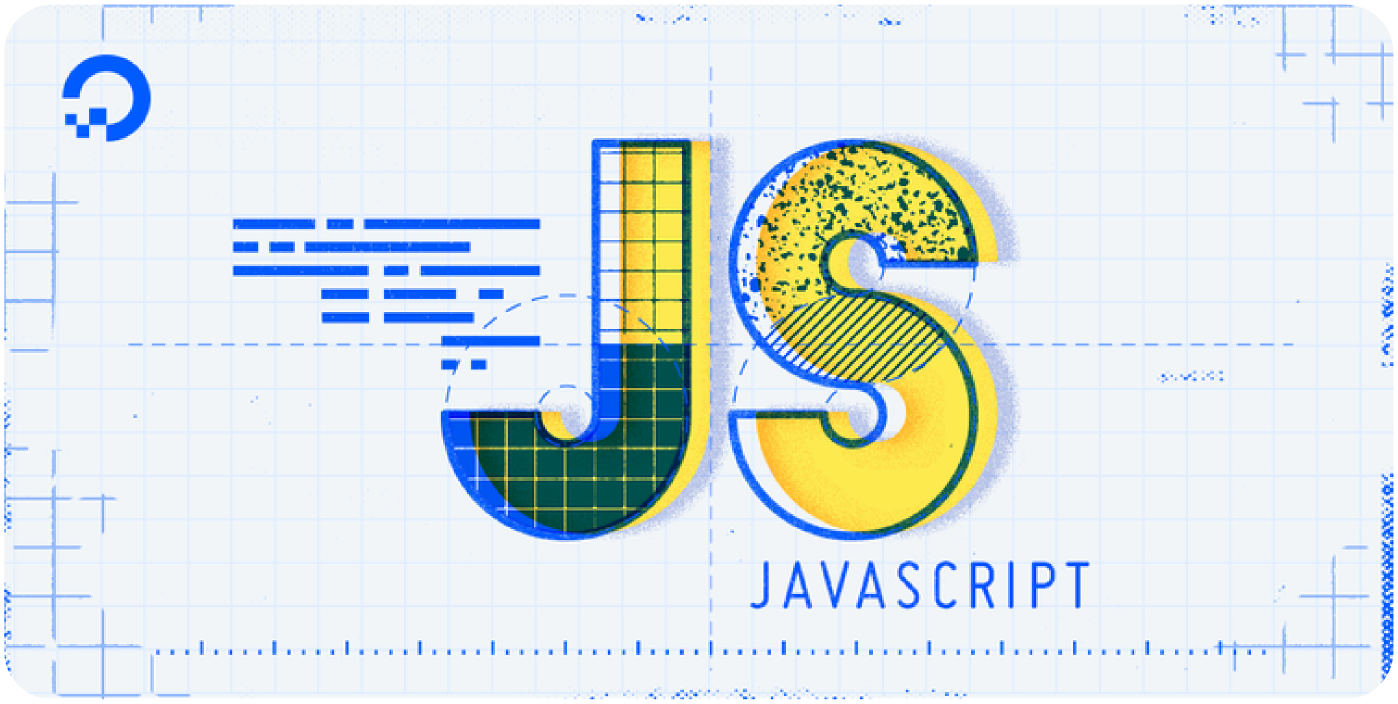
35/37 Understanding Arrow Fun...

Understanding Template Literals in JavaScript

Published on June 30, 2020 · Updated on August 27, 2021

JavaScript Development

By Tania Rascia



The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

# Introduction

The 2015 edition of the ECMAScript specification (ES6) added *template literals* to the JavaScript language. Template literals are a new form of making strings in JavaScript that add a lot of powerful new capabilities, such as creating multi-line strings more easily and using placeholders to embed expressions in a string. In addition, an advanced feature called *tagged template literals* allows you to perform operations on the expressions within a

In this article, you will go over the differences between single/double-quoted strings and template literals, running through the various ways to declare strings of different shape, including multi-line strings and dynamic strings that change depending on the value of a variable or expression. You will then learn about tagged templates and see some real-world examples of projects using them.

# Declaring Strings

This section will review how to declare strings with single quotes and double quotes, and will then show you how to do the same with template literals.

In JavaScript, a string can be written with single quotes (`' '`):

```
const single = 'Every day is a good day when you paint.'
```

Copy

A string can also be written with double quotes (`" "`):

```
const double = "Be so very light. Be a gentle whisper."
```

Copy

There is no major difference in JavaScript between single- or double-quoted strings, unlike other languages that might allow *interpolation* in one type of string but not the other. In this context, interpolation refers to the evaluation of a placeholder as a dynamic part of a string.

The use of single- or double-quoted strings mostly comes down to personal preference and convention, but used in conjunction, each type of string only needs to escape its own type of quote:

Copy

```
// Escaping a single quote in a single-quoted string
const single = '"We don\'t make mistakes. We just have happy accidents." - Bob Ro

// Escaping a double quote in a double-quoted string
const double = "\"We don't make mistakes. We just have happy accidents.\" - Bob R

console.log(single);
console.log(double);
```

```
"We don't make mistakes. We just have happy accidents." - Bob Ross
"We don't make mistakes. We just have happy accidents." - Bob Ross
```

Template literals, on the other hand, are written by surrounding the string with the backtick character, or grave accent (`):

```
const template = `Find freedom on this canvas.`
```

Copy

They do not need to escape single or double quotes:

```
const template = `"We don't make mistakes. We just have happy accidents." R
```

Copy

However, they do still need to escape backticks:

```
const template = `Template literals use the \` character.`
```

Copy

Template literals can do everything that regular strings can, so you could possibly replace all strings in your project with them and have the same functionality. However, the most common convention in codebases is to only use template literals when using the additional capabilities of template literals, and consistently using the single or double quotes for all other simple strings. Following this standard will make your code easier to read if examined by another developer.

Now that you've seen how to declare strings with single quotes, double quotes, and backticks, you can move on to the first advantage of template literals: writing multi-line strings.

# Multi-line Strings

In this section, you will first run through the way strings with multiple lines were declared before ES6, then see how template literals make this easier.

Originally, if you wanted to write a string that spans multiple lines in your text editor, you would use the concatenation operator. However, this was not always a straight-forward process. The following string concatenation seemed to run over multiple lines:

This might allow you to break up the string into smaller lines and include it over multiple lines in the text editor, but it has no effect on the output of the string. In this case, the strings will all be on one line and not separated by newlines or spaces. If you logged `address` to the console, you would get the following:

```
Output
Homer J. Simpson742 Evergreen TerraceSpringfield
```

You can use the backslash character (`\`) to continue the string onto multiple lines:

```
const address =
  'Homer J. Simpson\
  742 Evergreen Terrace\
  Springfield'
```

Copy

This will retain any indentation as whitespace, but the string will still be on one line in the output:

```
Output
Homer J. Simpson  742 Evergreen Terrace  Springfield
```

Using the newline character (`\n`), you can create a true multi-line string:

```
const address =
  'Homer J. Simpson\n' +
  '742 Evergreen Terrace\n' +
  'Springfield'
```

Copy

When logged to the console, this will display the following:

```
Output
Homer J. Simpson
742 Evergreen Terrace
Springfield
```

```
const address = `Homer J. Simpson
742 Evergreen Terrace
Springfield`
```

Copy

The output of logging this to the console is the same as the input:

```
Output
Homer J. Simpson
742 Evergreen Terrace
Springfield
```

Any indentation will be preserved, so it’s important not to indent any additional lines in the string if that is not desired. For example, consider the following:

```
const address = `Homer J. Simpson
                 742 Evergreen Terrace
                 Springfield`
```

Copy

Although this style of writing the line might make the code more human readable, the output will not be:

```
Output
Homer J. Simpson
                 742 Evergreen Terrace
                 Springfield
```

With multi-line strings now covered, the next section will deal with how expressions are interpolated into their values with the different string declarations.

# Expression Interpolation

In strings before ES6, concatenation was used to create a dynamic string with variables or expressions:

```
const method = 'concatenation'
```

Copy

```
This string is using concatenation.
```

With template literals, an expression can be embedded in a *placeholder*. A placeholder is represented by `${}`, with anything within the curly brackets treated as JavaScript and anything outside the brackets treated as a string:

```
const method = 'interpolation'
const dynamicString = `This string is using ${method}.`
```

Copy

When `dynamicString` is logged to the console, the console will show the following:

```
Output
This string is using interpolation.
```

One common example of embedding values in a string might be for creating dynamic URLs. With concatenation, this can be cumbersome. For example, the following declares a function to generate an OAuth access string:

```
function createOAuthString(host, clientId, scope) {
  return host + '/login/oauth/authorize?client_id=' + clientId + '&scope=' + scop
}

createOAuthString('https://github.com', 'abc123', 'repo,user')
```

Copy

Logging this function will yield the following URL to the console:

```
Output
https://github.com/login/oauth/authorize?client_id=abc123&scope=repo,user
```

Using string interpolation, you no longer have to keep track of opening and closing strings and concatenation operator placement. Here is the same example with template literals:

```
function createOAuthString(host, clientId, scope) {
  return `${host}/login/oauth/authorize?client_id=${clientId}&scope=${scope}`
}
```

Copy

```
Output
https://github.com/login/oauth/authorize?client_id=abc123&scope=repo,user
```

You can also use the `trim()` method on a template literal to remove any whitespace at the beginning or end of the string. For example, the following uses an arrow function to create an HTML `<li>` element with a customized link:

```
const menuItem = (url, link) =>
  `
<li>
  <a href="${url}">${link}</a>
</li>
`.trim()

menuItem('https://google.com', 'Google')
```

Copy

The result will be trimmed of all the whitespace, ensuring that the element will be rendered correctly:

```
Output
<li>
  <a href="https://google.com">Google</a>
</li>
```

Entire expressions can be interpolated, not just variables, such as in this example of the sum of two numbers:

```
const sum = (x, y) => x + y
const x = 5
const y = 100
const string = `The sum of ${x} and ${y} is ${sum(x, y)}.`

console.log(string)
```

Copy

This code defines the `sum` function and the variables `x` and `y`, then uses both the function and the variables in a string. The logged result will show the following:

This can be particularly useful with ternary operators, which allow conditionals within a string:

```
const age = 19
const message = `You can ${age < 21 ? 'not' : ''} view this page`
console.log(message)
```

Copy

The logged message here will change depnding on whether the value of `age` is over or under `21`. Since it is `19` in this example, the following output will be logged:

```
Output
You can not view this page
```

Now you have an idea of how template literals can be useful when used to interpolate expressions. The next section will take this a step further by examining tagged template literals to work with the expressions passed into placeholders.

# Tagged Template Literals

An advanced feature of template literals is the use of *tagged template literals*, sometimes referred to as *template tags*. A tagged template starts with a *tag function* that parses a template literal, allowing you more control over manipulating and returning a dynamic string.

In this example, you'll create a `tag` function to use as the function operating on a tagged template. The string literals are the first parameter of the function, named `strings` here, and any expressions interpolated into the string are packed into the second parameter using rest parameters. You can console out the parameter to see what they will contain:

```
function tag(strings, ...expressions) {
  console.log(strings)
  console.log(expressions)
}
```

Copy

Use the `tag` function as the tagged template function and parse the string as follows:

```
(4) ["This is a string with ", " and ", " and ", " interpolated inside."]
(3) [true, false, 100]
```

The first parameter, `strings`, is an array containing all the string literals:

- `"This is a string with "`
- `" and "`
- `" and "`
- `" interpolated inside."`

There is also a `raw` property available on this argument at `strings.raw`, which contains the strings without any escape sequences being processed. For example, `/n` would just be the character `/n` and not be escaped into a newline.

The second parameter, `...expressions`, is a rest parameter array consisting of all the expressions:

- `true`
- `false`
- `100`

The string literals and expressions are passed as parameters to the tagged template function `tag`. Note that the tagged template does not need to return a string; it can operate on those values and return any type of value. For example, we can have the function ignore everything and return `null`, as in this `returnsNull` function:

Copy

```
function returnsNull(strings, ...expressions) {
  return null
}

const string = returnsNull`Does this work?`
console.log(string)
```

Logging the `string` variable will return:

```
Output
null
```

```
function bold(strings, ...expressions) {
  let finalString = ''

  // Loop through all expressions
  expressions.forEach((value, i) => {
    finalString += `${strings[i]}<strong>${value}</strong>`
  })

  // Add the last string literal
  finalString += strings[strings.length - 1]

  return finalString
}

const string = bold`This is a string with ${true} and ${false} and ${100} interpo

console.log(string)
```

Copy

This code uses the `forEach` method to loop over the `expressions` array and add the bolding element:

```
Output
This is a string with <strong>true</strong> and <strong>false</strong> and <stron
```

There are a few examples of tagged template literals in popular JavaScript libraries. The `graphql-tag` library uses the `gql` tagged template to parse GraphQL query strings into the abstract syntax tree (AST) that GraphQL understands:

```
import gql from 'graphql-tag'

// A query to retrieve the first and last name from user 5
const query = gql`
  {
    user(id: 5) {
      firstName
      lastName
    }
  }
`
```

Copy

```
import styled from 'styled-components'

const Button = styled.button`
  color: magenta;
`

// <Button> can now be used as a custom component
```
Copy

You can also use the built-in `String.raw` method on tagged template literals to prevent any escape sequences from being processed:

```
const rawString = String.raw`I want to write /n without it being escaped.`
console.log(rawString)
```
Copy

This will log the following:

```
Output
I want to write /n without it being escaped.
```

# Conclusion

In this article, you reviewed single- and double-quoted string literals and you learned about template literals and tagged template literals. Template literals make a lot of common string tasks simpler by interpolating expressions in strings and creating multi-line strings without any concatenation or escaping. Template tags are also a useful advanced feature of template literals that many popular libraries have used, such as GraphQL and `styled-components`.

To learn more about strings in JavaScript, read How To Work with Strings in JavaScript and How To Index, Split, and Manipulate Strings in JavaScript.

Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases.

Tutorial Series: How To Code in JavaScript

JavaScript is a high-level, object-based, dynamic scripting language popular as a tool for making webpages interactive.

Subscribe

JavaScript Development

Browse Series: 37 articles

Expand to view all

Tania Rascia Author

Timothy Nolan Editor
Senior Technical Editor

Editor at DigitalOcean, fiction writer and podcaster elsewhere, always searching for the next good nautical pun!

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

Leave a comment

B I U S H1 H2 H3

Leave a comment...

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

Try DigitalOcean for free

Click below to sign up and get **$200 of credit** to try our products over 60 days!

Sign up →

Popular Topics

Ubuntu

Linux Basics

JavaScript

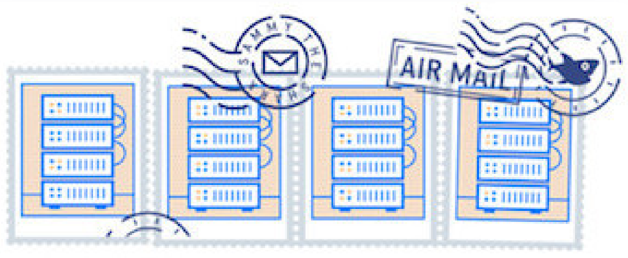
Python

MySQL

Docker

Kubernetes

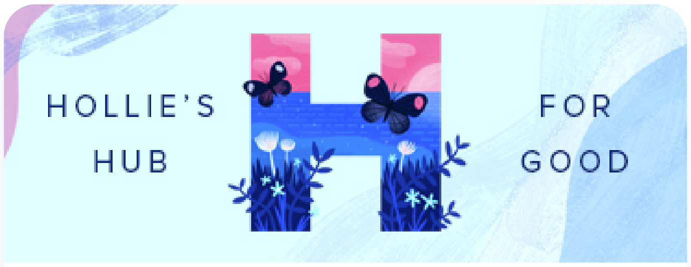
All tutorials →



Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

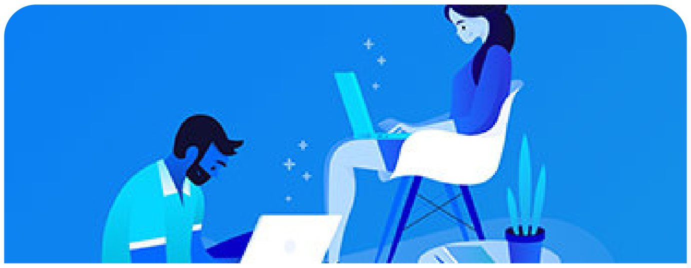
Sign up →



Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →



Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

[Kubernetes Course](#) [Learn Python 3](#) [Machine Learning in Python](#)
[Getting started with Go](#) [Intro to Kubernetes](#)

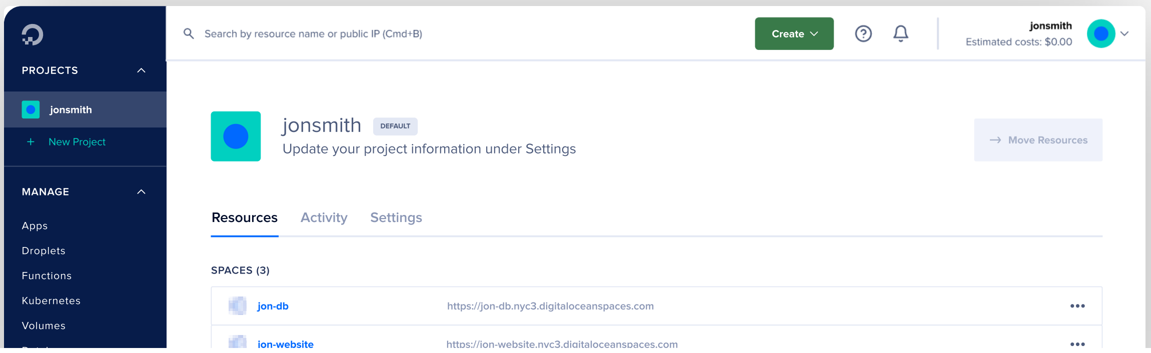
DigitalOcean Products

[Cloudways](#) [Virtual Machines](#) [Managed Databases](#) [Managed Kubernetes](#)
[Block Storage](#) [Object Storage](#) [Marketplace](#) [VPC](#) [Load Balancers](#)

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn more →



Partners

Channel Partners

Referral Program

Affiliate Program

Press

Legal

Security

Investor Relations

DO Impact

Functions

Cloudways

Managed Databases

Spaces

Marketplace

Load Balancers

Block Storage

Tools & Integrations

API

Pricing

Documentation

Release Notes

Uptime

Currents Research

Hatch Startup Program

deploy by DigitalOcean

Shop Swag

Research Program

Open Source

Code of Conduct

Newsletter Signup

Meetups

Streaming

VPN

SaaS Platforms

Cloud Hosting for Blockchain

Startup Resources