Pug.js tutorial

last modified October 18, 2023

The pug.js tutorial presents the Pug template engine.

Pug

Pug is a JavaScript template engine. It is is a high-performance template engine heavily influenced by Haml and implemented with JavaScript for Node.js and browsers. Pug was formerly calle Jade.

Template engine

A template engine or template processor is a library designed to combine templates with a data model to produce documents. Template engines are often used to generate large amounts of emails, in source code preprocessing, or producing dynamic HTML pages.

We create a template engine, where we define static parts and dynamic parts. The dynamic parts are later replaced with data. The rendering function later combines the templates with data.

Setting up Pug.js

First, we install Pug.js.

```
$ npm init -y
```

We initiate a new Node application.

```
$ npm i pug
```

We install `pug` module with `nmp i pug`.

Pug.js rendering from string

We start with a very simple example that renders from a string.

simple.js

```
import { render } from 'pug';

const template = 'p #{name} is a #{occupation}';

const data = { 'name': 'John Doe', 'occupation': 'gardener' };
const output = render(template, data);

console.log(output);
```

The example shows output from a string template.

```
import { render } from 'pug';
```

We load the `render` function from pug module.

```
const template = 'p #{name} is a #{occupation}';
```

This is our simple string template. The first value is the tag to be rendered. In addition, we add two variables: `name`and `occupation`. To output the variables we use the `#{}` syntax.

```
const data = { 'name': 'John doe', 'occupation': 'gardener' };
```

This is the data that we pass to the template engine.

```
const output = render(template, data);
```

The `render` function takes a template string and the context data. It compiles both into the final string output.

```
$ node app.js
<p>John Doe is a gardener</p>
```

Pug.js compileFile

The `compileFile` function compiles a Pug template from a file to a function which can be rendered multiple times with different locals.

template.pug

```
p Hello #{name}!
```

This is the template file; it has a `.pug` extension.

app.js

```
import { compileFile } from 'pug';

const cfn = compileFile('template.pug');

const res = cfn({'name': 'John Doe'});
console.log(res);

const res2 = cfn({'name': 'Roger Roe'});
console.log(res2);
```

We compile the template to a function and call the function with two different local data.

```
$ node app.js
<p>Hello John Doe!</p>
<p>Hello Roger Roe!</p>
```

Pug.js renderFile

The `renderFile` function compiles a Pug template from a file and render it with locals to HTML string.

template.pug

```
doctype html
html
  body
    ul
      li Name: #{name}
      li Occupation: #{occupation}
```

In the tempalte, we have a small HTML document with an unordered list. We have two variables.

app.js

```
import { renderFile } from 'pug';

const options = { 'pretty': true }
const locals = { 'name': 'John doe', 'occupation': 'gardener', };

const res = renderFile('template.pug', Object.assign(locals, options));

console.log(res);
```

We merge the locals and the options with `Object.assign`;

```
const options = { 'pretty': true }
```

In the options map, we set the pretty printing. (Note that this option is deprecated.)

```
$ node app.js
<!DOCTYPE html>
<html>
  <body>
    <ul>
      <li>Name: John doe</li>
      <li>Occupation: gardener</li>
    </ul>
  </body>
</html>
```

Pug.js passing a list of data

In the following example, we pass a list of data to the template engine and process it.

template.pug

```
doctype html
html
  body
    ul
      each e in names
        li= e
```

In the template, we use the `each` form to go over the list of data passed to the template.

app.js

```
import { renderFile } from 'pug';

const names = ['John Doe', 'Roger Roe', 'Paul Smith', 'Rebecca Jordan'];
const res = renderFile('template.pug', { 'names': names });

console.log(res);
```

We define a list of names. The list is passed to the template engine in the options with the `names` option.

Pug.js conditions

Conditions are created with `if/else` keywords.

template.pug

```
doctype html
html
  head
    style.
      .emp { background: lightGreen }

  body
    if emp
      p.emp Today is #{today}
    else
      p Today is #{today}
```

In the template, we show en emphasized paragraph depending on the `emp` option.

```
style.
```

With the dot syntax, we can pass a block of text to the tag.

app.js

```
import { renderFile } from 'pug';
import { writeFileSync } from 'fs';

const today = new Date().toLocaleDateString()
const emp = false;

const output = renderFile('template.pug', {'today': today, 'emp': emp});

writeFileSync('index.html', output);
```

We get the current data and pass it along the `emp` variable to the template engine. The `emp` determines whether the output is emphasized. The resulting output is written to a file with `writeFileSync`.

Pug.js table

In the following example, we read data from a CSV file and render it in an HTML table.

```
$ npm i csv
```

We use the `csv` module to process the CSV data.

cars.csv

```
id,name,price
1,Audi,52642
```

```
2,Mercedes,57127
3,Skoda,9000
4,Volvo,29000
5,Bentley,350000
6,Citroen,21000
7,Hummer,41400
8,Volkswagen,21600
9,Toyota,26700
```

This CSV data is rendered in an HTML table.

template.pug

```
doctype html
html

  body
    table
      thead
        tr
          each header in headers
            th= header
      tbody
          each field in fields
            tr
              td= field.id
              td= field.name
              td= field.price
```

The data is displayed in an HTML table.

```
td= field.id
```

The `td=` syntax interpolates the field.

app.js

```
import { renderFile } from 'pug';
import { readFileSync, writeFileSync } from 'fs';
import pkg from 'csv';
const { parse } = pkg;

const csvData = readFileSync('cars.csv').toString();

parse(csvData, { columns: true }, (e, records) => {

    const headers = Object.keys(records[0]);
    const template = 'template.pug';

    const options = { 'fields': records, 'headers': headers };
    const res = renderFile(template, options);
    writeFileSync('index.html', res);
});
```

We read and parse the CSV data. We seperate the CSV data into headers and records.

Pug.js with Express.js

In the next example, we integrate Pug.js with Express.js web framework.

views/index.pug

```
html
  body
    p Today is #{today}
```

The template is placed in the `views` directory.

app.js

```
import express from "express";

const app = express();
app.set('view engine', 'pug');

app.get("/today", (req, res) => {

    let today = new Date();
    res.render("index", {today: today});
});

app.use((req, res) => {
    res.statusCode = 404;
    res.end("404 - page not found");
});

app.listen(3000, () => {

    console.log("Application started on port 3000");
});
```

The example is a small web application which shows the current date.

```
app.set('view engine', 'pug');
```

We tell Express to use Pug.

```
app.get("/today", (req, res) => {
    let today = new Date();
    res.render("index", {today: today});
});
```

We render the `index` template for the `/today` route.

In this article we have used Pug.js to generate HTML documents from Pug templates and data.

Author

My name is Jan Bodnar and I am a passionate programmer with many years of programming experience. I have been writing programming articles since 2007. So far, I have written over 1400 articles and 8 e-books. I have over eight years of experience in teaching programming.

List all JavaScript tutorials.