Guia del motor de plantillas PUG

Fuente: https://www.youtube.com/playlist?list=PLROIqh_5RZeCFvLRw6_L7IGXFU_GahWhL

Pug es un motor de plantillas basado en javascript destinado a facilitar la escritura de código html. El código pug requiere de un compilador que convierta el código a html, ya que el navegador por si solo no lo interpreta.

Las principales ventajas a la hora de usar pug son:

- Tiene una sintaxis mas amigable que html y otros motores de plantillas como hbs o ejs.
- Esta totalmente integrado con NodeJs y tecnologías como Express y React.
- Permite incluir código puro de html.

Como desventaja podría mencionarse que al no controlarse el código mediante el cierre de etiquetas sino por indentación puede conducir a errores difíciles de detectar.

Las principales características a tener en cuenta de pug son:

- Sintaxis.
- Interpolación de variables.
- Condicionales.
- Bloques, includes, templates, etc.
- Mixins

1 – Instalación y ejecución

Antes de empezar la práctica con pug es necesario instalar algunas cosas.

1. Tener instalado NodeJS y npm
2. Instalar pug de forma global y verificar la instalación

```
npm install pug -g
pug -V
```

Para compilar un archivo **index.pug** y generar un documento html en la misma carpeta se usa el comando

```
pug index.pug
```

Si no queremos tener que estar ejecutandolo cada vez que se hace una modificación se puede agregar el flag watch, que rederiza el documento cada vez que se guarda un cambio

```
pug --watch index.pug
```

Este genera el archivo comprimido, para verlo con formato hay que agregar el flag **--pretty**.

```
pug --watch index.pug --pretty
```

para separar los archivos en carpetas distintas es necesario especificar la ruta del archivo y, con el flag **out**, la carpeta de destino.

```
pug --watch 'pug/index.pug' --out './html' --pretty
```

2 – Sintaxis

La sintaxis básica de pug tiene las siguientes reglas:

- Las **etiquetas** no requieren los símbolos de **<>** ni un cierre.
- Para **anidar** elementos la forma básica es mediante indentación.
- Los **atributos** del elemento html se indican entre **paréntesis** y se pueden separar con comas o con espacios.
- Las **calses** se indican con un **punto**.
- El **id** se indica con un **numeral**.
- Pug soporta sintaxis propia de html.

Teniendo en cuenta estas reglas una estructura html5 básica con un título en pug se escribe de la siguiente manera.

Index.pug

```
<!DOCTYPE html>
html(lang="en")
    head
        meta(charset="UTF-8")
        meta(name="viewport"
            content="width=device-width, initial-scale=1.0")
        title pug | Sintaxis basica
    body

        h1#titulo1.h1 Titulo
```

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>pug | Sintaxis basica</title>
  </head>
  <body>
    <h1 class="h1" id="titulo1">Titulo</h1>
  </body>
</html>
```

Para **intercalar elementos** no alcanza con poner la etiqueta ya que al no tener una etiqueta de cierre el motor no sabe donde empieza y termina el elemento anidado. Por ejemplo, si queremos incluir un enlace en un parrafo, al agregar un **a** dentro del mismo lo toma como una letra.

```
p Lorem ipsum dolor sit amet a(href="#") consectetur adipisicing elit. Numquam, voluptatum?
```

Por otro lado, si se hace un salto de linea deja de considerar el resto del texto como parte del párrafo y la siguiente palabra la considera un etiqueta.

```
p Lorem ipsum dolor sit amet
    a(href="#")
consectetur adipisicing elit. Numquam, voluptatum?
```

Esto se soluciona agregando una barra horizontal antes del resto del texto. Es importante tener en cuenta que este debe estar al mismo nivel que la etiqueta hijo, en este caso a.

```
p Lorem ipsum dolor sit amet
    a(href="#")
    | consectetur adipisicing elit. Numquam, voluptatum?
```

Otra modificación al anidamiento de elementos es la **expansión en bloque**. Esta permite anidar

elementos en una misma linea. Es usado generalmente en listas y menues Vemos como de ambas formas se obtiene el mismo resultado.

```
ul.menu
    li.item1
        a.link elemento

ul.menu: li.item1: a.link elemento
```

```
<ul class="menu">
  <li class="item1"><a class="link">elemento</a></li>
</ul>

<ul class="menu">
  <li class="item1"><a class="link">elemento</a></li>
</ul>
```

Por último, dos casos particulares son los **estilos** css y los **scripts** implementados dentro del documento. Mientras los que se indican con la referencia a un archivo externo se indican con etiquetas de igual manera que en html

```
html(lang="en")
    head
        meta(charset="UTF-8")
        title pug | Sintaxis basica
        link(rel="stylesheet", href="style.css")
    body

        script(src="index.js")
```

```
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <title>pug | Sintaxis basica</title>
    <link rel="stylesheet" href="style.css"/>
  </head>
  <body>
    <script src="index.js"></script>
  </body>
</html>
```

para implementarlos dentro del documento es necesario agregar un punto a la etiqueta.

```
html(lang="en")
    head
        meta(charset="UTF-8")
        title pug | Sintaxis basica
        style.
            body{
                color: darkorange;
            }
    body
        h1 pug | Sintaxis básica
        script.
            alert('hola mundo')
```

```
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <title>pug | Sintaxis basica</title>
    <style>
      body{
        color: darkorange;
      }
    </style>
  </head>
  <body>
    <h1>pug | Sintaxis básica</h1>
    <script>alert('hola mundo')</script>
  </body>
</html>
```

3 – Código js embebido

Además de la sintaxis, una de las características principales de los motores de plantillas es que permiten embeber código javascript. Esto brinda tres funciones escenciales.

- Manejo de variables.
- Condicionales
- Bucles

3.1 - Variables

Al manejo de variables en pug se lo llama **interpolación**. Las **variables**, tanto let como const se declaran con un **guion** , el tipo de variable y el nombre, y pueden utilizarse de dos formas, con un **numeral** seguido de un par de llaves, o con un símbolo **igual.**

* Para declarar variables de tipo objeto se debe hacer un salto de linea e indentación luego del guión.

```
-let persona = {
        nombre: "Carlos",
        apellido:"Martinez",
        edad: 29
    }
```

→ NO FUNCIONA!

Numeral (#)

Este sirve no solo para mostrar variables, sino que también puede ejecutarse funciones de javascript.

```
body
    h1 pug | Javascript
    -let nombre = "Carlos"
    p Hola #{nombre.toUpperCase()}
```

```
<body>
  <h1>pug | Javascript</h1>
  <p>Hola CARLOS</p>
</body>
```

Igual (=)

Esta es la forma mas usual y recomendada por dos motivos. Por un lado porque permite utilizar template strings, y por el otro porque es la forma en que se renderiza información recibida del servidor en NodeJs.

```
body
    h1 pug | Javascript
    -
        let persona = {
            nombre: "Carlos",
            apellido:"Martinez",
            edad: 29
        }

    p= ` ${persona.nombre} ${persona.apellido} tiene ${persona.edad} años`
```

```
<body>
  <h1>pug | Javascript</h1>
  <p> Carlos Martinez tiene 29 años</p>
</body>
```

3.2 – Condicionales

Los condicionales tienen tres formas principales de utilización

- If / else
- Unless
- Operador ternario

El condicional tradicional **if/else** se escribe de la siguiente manera .

```
-let logueado = false

if logueado
    p Bienvendido
else if logueado == null
    p No se regustró un usuario
else
    p usuario o contraseña incorrectos
```

```
<body>
  <h1>pug | Javascript</h1>
  <p>usuario o contraseña incorrectos</p>
</body>
```

Unless equivale a un if negado

```
-let logueado = false

unless logueado
    p ingrese usuario y contraseña
else
    p bienvenido
```

```
<body>
  <h1>pug | Javascript</h1>
  <p>ingrese usuario y contraseña</p>
</body>
```

Por último, el **operador ternario** puede ser utilizado para asignar clases según información llegada del servidor.

```
-let fondo = true;

div(class= fondo ? "bgDark" : "bgLight")
    p Hola mundo
```

```
<body>
  <h1>pug | Javascript</h1>
  <div class="bgLight">
    <p>Hola mundo</p>
  </div>
</body>
```

3.3 – Bucles

Si bien en pug existen dos tipos de bucles, el foreach y el while, este último generalmente no es utilizado, por lo que nos concentramos en el primero.

La sintaxis básica del **each** en pug es

```
-let personas = ["carlos", "Sebastián", "Martin"];

ul
    each persona in personas
        li= persona
```

```
<ul>
  <li>carlos</li>
  <li>Sebastián</li>
  <li>Martin</li>
</ul>
```

En caso de recibir un array vacío puede agregarse un condicional else al final

```
-let personas = [];

ul
    each persona in personas
        li= persona
    else
        p no existen usuarios
```

```
<ul>
  <p>no existen usuarios</p>
</ul>
```

El condicional también puede ser usado para recorrer objetos, y recibir un segundo parámetro, que representa la clave. En caso de ser un array el segundo parámetro es el indice.

```
-let personas = ["Carlos", "Sebastián", "Martin"];
each persona, indice in personas
    p= `Usuario numero ${indice+1}: ${persona}`

-
    let usuario ={
        nombre:"Carlos",
        apellido: "Martinez",
        edad: 29
    }
each valor, clave in usuario
    p= ` ${clave}: ${valor}`
```

```
<h1>pug | Javascript</h1>
<p>Usuario numero 1: Carlos</p>
<p>Usuario numero 2: Sebastián</p>
<p>Usuario numero 3: Martin</p>

<p> nombre: Carlos</p>
<p> apellido: Martinez</p>
<p> edad: 29</p>
```

4 – Templates y partials

Pug ofrece la posibilidad de evitar repetir código e interactuar dinámicamente entre distintos documentos html mediante tres mecanismos

- extends.
- Block.
- Include.

4.1 – Extends

Este es utilizado generalmente para reutilizar el contenido de la etiqueta **head**. Así por ejemplo no es necesario incluir en cada documento la referencia al archivo de estilos css. Para esto hay que crear un archivo comúnmente llamado **template.pug** (a veces también es layout.pug) en una carpeta de orden superior a donde se encuentran las vistas y estas extienden del mismo.

Template.pug

```pug
<!DOCTYPE html>
html(lang="en")
    head
        meta(charset="UTF-8")
        title Document
        link(rel="stylesheet", href="style.css")
    body
```

ndex.pug

```pug
extends ../templates/template.pug
```

index.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <title>Document</title>
    <link rel="stylesheet" href="style.css"/>
  </head>
  <body></body>
</html>
```

* Al incluir un template en un archivo pug, luego de este no se puede agregar contenido a no ser que este sea en un bloque o un mixin.

4.2 – Blocks

Para agregar contenido a un template es necesario especificarlo en el mismo indicando que el mismo contiene un **bloque**. Generalmente si va a haber un solo bloque este se llama **content**.

Así, ahora podemos agregar el siguiente contenido al documento.

pug > index.pug

```pug
extends ../templates/template.pug

block content
    h1 pug | Bloques
    p Lorem ipsum dolor sit amet consectetur adipisicing elit. Iste,
    nesciunt!
```

html > index.html > ...

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"/>
    <title>Document</title>
    <link rel="stylesheet" href="style.css"/>
  </head>
  <body></body>
  <h1>pug | Bloques</h1>
  <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Iste,
  nesciunt!</p>
</html>
```

Si uno quiere forzar que los documentos que extienden de una plantilla sigan cierta estructura, puede declarar en la misma las etiquetas generales y luego en cada documento su contenido.

Por ejemplo, queremos que todos los documentos que extienden de **template.pug** tengan un encabezado, cuerpo y pie de página, entonces luego del contenido propio del template se ponen las etiquetas **header**, **div.container** (usando la clase de bootstrap) y **footer**, cada una con un bloque dentro. Luego el contenido de cada una de las etiquetas se rrellena en el bloque dentro del documento que importa el template.

Template.pug

templates > template.pug

```pug
body

    header.dark
        block header

    div.container
        block content

    footer.dark
        block footer
```

index.pug

pug > index.pug

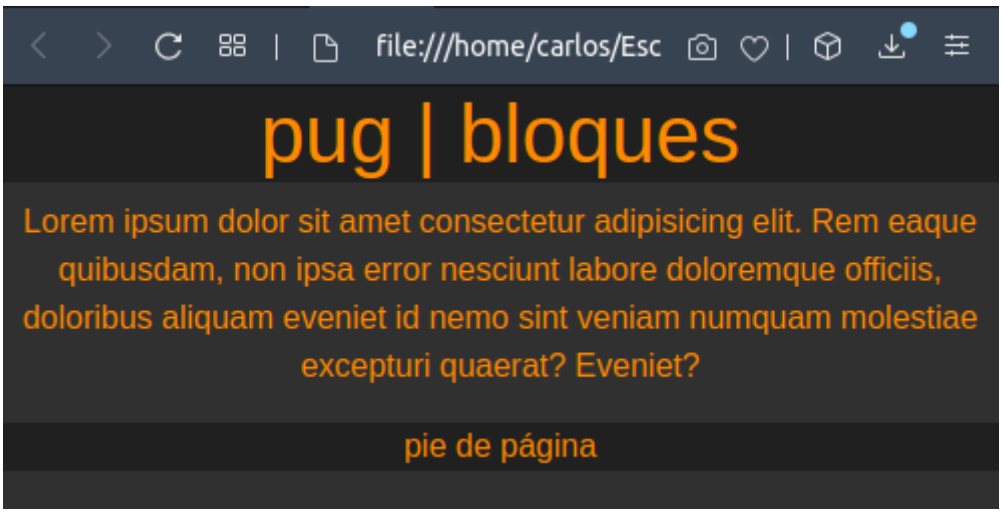
```pug
extends ../templates/template.pug

block header
    h1.text-center pug | bloques
block content
    .row.text-center
        p Lorem ipsum dolor sit amet consectetur adipisicing elit. Rem eaque quibusdam, non ipsa error nesciunt labore doloremque officiis, doloribus aliquam eveniet id nemo sint veniam numquam molestiae excepturi quaerat? Eveniet?
block footer
    p.text-center pie de página


```

index.html

html > index.html > ...

```html
  <body>
    <header class="dark">
      <h1 class="text-center">pug | bloques</h1>
    </header>
    <div class="container">
      <div class="row text-center">
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Rem eaque quibusdam, non ipsa error nesciunt labore doloremque officiis, doloribus aliquam eveniet id nemo sint veniam numquam molestiae excepturi quaerat? Eveniet?</p>
      </div>
    </div>
    <footer class="dark">
      <p class="text-center">pie de página</p>
    </footer>
  </body>
</html>
```

De este modo, aplicando algunos estilos la página queda asi.



Por último, los bloques también pueden tener **contenido por defecto,** declarándolo como contenido en el template. Es importante aclarar que si en el documento que importa el template se usa la palabra **block**, este va a sobreescribir el contenido por defecto, entonces para mantenerlo se llama al bloque usando **append**, si se quiere agregar contenido al bloque luego del que ya está, o **prepend** si se quiere agregar antes.

Entonces para el siguiente cuerpo del template

```
15      body
16          block content
17              h1 titulo 1
18
```

Se obtienen los siguientes resultados

```
3   block content
4       h2 titulo 2
```

```
17  <body>
18    <h2>titulo 2</h2>
19  </body>
```

```
3   append content
4       h2 titulo 2
```

```
17  <body>
18    <h1>titulo 1</h1>
19    <h2>titulo 2</h2>
20  </body>
```

```
3   prepend content
4       h2 titulo 2
```

```
17  <body>
18    <h2>titulo 2</h2>
19    <h1>titulo 1</h1>
20  </body>
```

4.3 – Includes

Los blocks son una forma de dividir el código, pero mas orientada a darle un orden y estructura a los documentos. Si lo que se necesita es reutilizar código en ciertas partes del mismo, pero sin importar dónde deben usarse los includes. Estos son muy útiles por ejemplo para usar los mismos menues o navegaciones (como vemos en el siguiente apunte de nodejs – primera página web).

En este ejemplo vemos como se puede usar para emular una base de datos. Creamos el archivo data.pug, y dentro un array de objetos.

Ejemplo

Creamos una carpeta **partials**, y dentro un archivo **data.pug**. Este archivo contiene un array de objetos, supongamos artículos de un kiosco.

partials > data.pug

```pug
-
    let articulos = [{
                        nombre: "caramelo",
                        precio: 2
                    },
                    {
                        nombre: "chicle",
                        precio: 5
                    },
                    {
                        nombre: "alfajor",
                        precio: 10
                    }
                ]
```

Esta información puede ser requerida desde cualquier parte del código indicando su ruta con la siguiente sintaxis

```pug
include ../partials/data.pug
```

Entonces, en el archivo index.pug tenemos este código para generar una tabla con los estilos de bootstrap (el cdn está importado desde el template.pug)

pug > index.pug

```pug
extends ../templates/template.pug

block content
    include ../partials/data.pug
    .container
        h1 Listado de articulos
        br
        table.table.table-striped
            thead
                tr
                    th #
                    th articulo
                    th precio
            tbody
                each articulo, i in articulos
                    tr
                        th= i
                        td= articulo.nombre
                        td= articulo.precio
```

Con este código se obtiene como resultado la siguiente vista

4.4 – Mixins

Otra forma de reutilizar el código en pug es mediante los mixins. Esto son funciones que reciben parámetros y los utilizan para hacer algo. Un ejemplo básico pero que permite mostrar la forma de implementar un mixin es crear uno que reciba un array de elementos y los liste.

Creamos un archivo llamado listar.pug en la carpeta mixins y en este ponemos el siguiente código.

```
mixin listar(array)
    each element in array
        li= element
```

Y en index.pug incluimos los archivos de datos y funciones, y los usamos para generar la lista de personas.

```
block content
    include ../partials/data.pug
    include ../mixins/listar
    .container.text-center
        ul
            +listar(personas)
```

Asi queda el html

```
<body>
  <div class="container text-center">
    <ul>
      <li>Carlos</li>
      <li>Sebastián</li>
      <li>Martín</li>
    </ul>
  </div>
</body>
```