{{ Mustache }}

1. Въведение:

Mustache е проста система за темплейти с имплементация на ActionScript, C++, Clojure, CoffeeScript, ColdFusion, D, Delphi, Erlang, Fantom, Go, Haskell, Io, Java, JavaScript, Julia, Lua, .NET, Objective-C, Perl, PHP, Pharo, Python, Racket, Ruby, Rust, Scala, Swift, CFEngine and XQuery.

1. Тагове

Таговете се индикират с двойни къдрави скоби: {{person}} е таг, както и {{#person}}. И в двата примера реферираме person като ключ или като таг на ключа.

* 1. Променливи

Най-простите за употреба вид тагове. Тагът {{name}} в разов html документ би опитал да открие името на ключа в текущия контекст. Ако няма такова име, ще провери в контекста на родителя рекурсивно. Ако при достигане на най-външния контекст променливата все още не е открита, нищо няма да се рендерира.

Всички променливи се ескейпват по подразбиране в HTML документа. Ако искате да върнете неескейпнат HTML, използвайте тройни къдрави скоби: {{{name}}}.

За да предотвратите ескейпването на променливи, може да използвате &: {{& name}}. Това може да е полезно, когато се променят разделителите.

По подразбиране липсваща променлива връща празен стринг. Обикновено това може да се промени в локалния конфигурационен файл на библитеката Mustache. Например: версията за Ruby поддържа хвърлянето на грешка в този случай.

Темплейт:

\* {{name}}

\* {{age}}

\* {{company}}

\* {{{company}}}

Хеш таблица:

{

"name": "Chris",

"company": "<b>GitHub</b>"

}

Резултат:

\* Chris

\*

\* &lt;b&gt;GitHub&lt;/b&gt;

\* <b>GitHub</b>

Източник: [http://mustache.github.io/mustache.5.html#Sections](http://mustache.github.io/mustache.5.html" \l "Sections)

* 1. Секции

Секциите рендерират блокове от текст един или повече пъти в зависимост от стойността на ключа в текущия контекст.

Секцията започва с "#" и завършва с "/". Например тагът {{#person}} започва секцията "person", а тагът {{/person}} я завършва.

Поведението на секцията се определя от стойността на ключа.

Стойност false или празен списък:

Ако ключът person съществува и има стойност false или празен лист, HTML-ът между "#" и "/" няма да се визуализира.

* 1. Непразни списъци

Ако ключът person съществува и има стойност, различна от false, HTML-ът между “#” и “/” ще се визуализира един или повече пъти.

Когато стойността е непразен списък, текстът в блока ще се визуализира по един път за всеки елемент в списъка. Контекстът на блока ще се запазва във всеки елемент за всяка итерация. По този начин се осъществява обхождане на колекциите.

Template:

{{#repo}}

<b>{{name}}</b>

{{/repo}}

Hash:

{

"repo": [

{ "name": "resque" },

{ "name": "hub" },

{ "name": "rip" }

]

}

Output:

<b>resque</b>

<b>hub</b>

<b>rip</b>

* 1. Lambda изрази

Когато стойността е обект, който можем да изпълним, например – функция или lamda функции, обектът ще се изпълни и ще се предаде на контекста в блока.

When the value is a callable object, such as a function or lambda, the object will be invoked and passed the block of text. The text passed is the literal block, unrendered. {{tags}} will not have been expanded - the lambda should do that on its own. In this way you can implement filters or caching.

Темплейт:

{{#wrapped}}

{{name}} is awesome.

{{/wrapped}}

Хеш таблица:

{

"name": "Willy",

"wrapped": function() {

return function(text, render) {

return "<b>" + render(text) + "</b>"

}

}

}

Резултат:

<b>Willy is awesome.</b>

* 1. Non-False стойности

Когато стойността на променлива е non-false, но не е списък, тя ще бъде използвана еднократно в блока при рендерирането си.

Темплейт:

{{#person?}}

Hi {{name}}!

{{/person?}}

Хеш таблица:

{

"person?": { "name": "Jon" }

}

Резултат:

Hi Jon!

* 1. Обърнати секции

Обърнатите секции започват с “^” и завършват с “/”. Например {{^person}} започва обърната секция “person”, {{/person}} я завършва.

Докато секциите могат да се използват един или повече пъти спрямо стойността на ключа, то обърнатите секции могат да рендерират текстът един път в зависимост от обърнатата стойност на ключа.

Тоест ако ключът не съществува, ще бъде рендерирано като false или като празен списък.

Темплейт:

{{#repo}}

<b>{{name}}</b>

{{/repo}}

{{^repo}}

No repos :(

{{/repo}}

Хеш таблица:

{

"repo": []

}

Резултат:

No repos :(

* 1. Коментари

Коментарите започват с “!” и се игнорират при рендерирането на HTML.

Коментарите могат да съдържат нови редове.

Например следният темплейт:

<h1>Today{{! ignore me }}.</h1>

Ще се рендерира по следния начин:

<h1>Today.</h1>

* 1. Partials

Partial-ите започват с “>” като {{> box}}.

Partial-ите се рендерират по време на изпълнение и е възможно да има рекурсивни части, стига да се избягват безкрайните цикли.

Частите наследяват контекста, от който са извикани. Например в [**eRuby**](http://en.wikipedia.org/wiki/ERuby) (Embedded Ruby) може да имаме следния пример:

<%= partial :next\_more, :start => start, :size => size %>

Mustache изисква само това:

{{> next\_more}}

Защо? Защото файлът next\_more.mustache ще наследи методите size и start от извикващия контекст.

По този начин можете да си мислите за partial-ите като includes, imports, разширители на темплейти, вгнездени темплейти, подтемплейти и др.

Например този темплейт и partial-ът:

base.mustache:

<h2>Names</h2>

{{#names}}

{{> user}}

{{/names}}

user.mustache:

<strong>{{name}}</strong>

Могат да се напишат като един разширен темплейт:

<h2>Names</h2>

{{#names}}

<strong>{{name}}</strong>

{{/names}}

* 1. Поставяне на разделители

Поставянето на разделители започва с “=” и променя разделителите за тагове от “{{“ и “}}” към собствено дефинирани стрингове.

Например:

\* {{default\_tags}}

{{=<% %>=}}

\* <% erb\_style\_tags %>

<%={{ }}=%>

\* {{ default\_tags\_again }}

Тук имаме списък от 3 елемента. Първият елемент използва стила на таговете по подразбиране, вторият използва erb стила, както е дефинирано според тагът за разделител, и третият елемент връща първоначалния стил след още една декларация на таг за разделител.

Според [ctemplates](http://google-ctemplate.googlecode.com/svn/trunk/doc/howto.html) това е „полезно за езици като TeX, в които двойните къдрави скоби може да се появят в текста“.

Собствено дефинираните разделители не могат да съдържат спейсове или “=”.

1. Имплементация за различни езици
   1. JavaScript

Кратък пример как да използваме mustache.js:</p>

var view = {

title: "Joe",

calc: function () {

return 2 + 4;

}

};

var output = Mustache.render("{{title}} spends {{calc}}", view);

В този пример функцията Mustache.renderприема 2 параметъра: темплейтът [mustache](http://mustache.github.com/) и обектът, който съдържа данните и кодът, за да се рендерира темплейта.

* 1. Pystache in Python

Как да го използваме?

>>> import pystache

>>> print pystache.render('Hi {{person}}!', {'person': 'Mom'})

Hi Mom!

Може също да създадем класове, които да пазят логиката за темплейтите.

Например създаваме класа SayHello (в .../examples/readme.py):

class SayHello(object):

def to(self):

return "Pizza"

Инициализираме го:

>>> from pystache.tests.examples.readme import SayHello

>>> hello = SayHello()

Тогава в темплейтите say\_hello.mustache (по подразбиране в същата директория, където е дефиниран класа) използваме:

Hello, {{to}}!

Всичко заедно:

>>> renderer = pystache.Renderer()

>>> print renderer.render(hello)

Hello, Pizza!

За по-добър контрол при рендерирането (например използване на собствена директория за темплейти) може да използваме класът Renderer както по-горе. Темплейтът може да подаде атрибути на конструктура на Render класа или да ги присвои на негова инстанция. Ако искаме да персонализираме темплейта, който се зарежда за определено view, е необходимо да наследим TemplateSpec. Виж документацията на класа [Renderer](https://github.com/defunkt/pystache/blob/master/pystache/renderer.py) и [TemplateSpec](https://github.com/defunkt/pystache/blob/master/pystache/template_spec.py) за повече информация.

Можем също и да създаваме темплейт, който да приема аргументи:

>>> parsed = pystache.parse(u"Hey {{#who}}{{.}}!{{/who}}")

>>> print parsed

[u'Hey ', \_SectionNode(key=u'who', index\_begin=12, index\_end=18, parsed=[\_EscapeNode(key=u'.'), u'!'])]

И тогава:

>>> print renderer.render(parsed, {'who': 'Pops'})

Hey Pops!

>>> print renderer.render(parsed, {'who': 'you'})

Hey you!

* 1. Ruby

С Mustache view-тата, които пишем, се разделят на 2 части: Ruby клас и HTML темплейт.

Цялата логика, решения и код се съхраняват във view-то, а всичкият markup - в темплейта.По този начин темплейтът не прави нищо друго освен да реферира методи в нашите view-та.

Стриктното разделение прави по-лесно писането на чисти темплейти, което ги прави по-лесно разширяеми, тестваеми и гъвкави.

Кратък пример:

>> require 'mustache'

=> true

>> Mustache.render("Hello {{planet}}", :planet => "World!")

=> "Hello World!"

Ето един каноничен пример, който е само малка част от всички [примери](https://github.com/mustache/mustache/tree/master/examples):

class Simple < Mustache

def name

"Chris"

end

def value

10\_000

end

def taxed\_value

value \* 0.6

end

def in\_ca

true

end

end

Просто създаваме обикновен Ruby клас и дефинираме методите му. Някои методи реферират други, някои връщат стойности, други - булеви стойности..

Сега да напишем и темплейта:

Hello {{name}}

You have just won {{value}} dollars!

{{#in\_ca}}

Well, {{taxed\_value}} dollars, after taxes.

{{/in\_ca}}

Този темплейт реферира нашите методи-view-та. За да съберем всичко заедно, ето как всъщност изглежда кодът, който ще рендерира HTML:

Simple.render

Което връща следното:

Hello Chris

You have just won 10000 dollars!

Well, 6000.0 dollars, after taxes.

Просто :)

* 1. Java

Примерен темплейтен файл:

{{#items}}

Name: {{name}}

Price: {{price}}

{{#features}}

Feature: {{description}}

{{/features}}

{{/items}}

Така подаваме данните на темплейта:

public class Context {

List<Item> items() {

return Arrays.asList(

new Item("Item 1", "$19.99", Arrays.asList(new Feature("New!"), new Feature("Awesome!"))),

new Item("Item 2", "$29.99", Arrays.asList(new Feature("Old."), new Feature("Ugly.")))

);

}

static class Item {

Item(String name, String price, List<Feature> features) {

this.name = name;

this.price = price;

this.features = features;

}

String name, price;

List<Feature> features;

}

static class Feature {

Feature(String description) {

this.description = description;

}

String description;

}

}

И получаваме:

Name: Item 1

Price: $19.99

Feature: New!

Feature: Awesome!

Name: Item 2

Price: $29.99

Feature: Old.

Feature: Ugly.

# Mstch for C++

## **Основна употреба**

#include <iostream>

#include <mstch/mstch.hpp>

int main() {

std::string view{"{{#names}}Hi {{name}}!\n{{/names}}"};

mstch::map context{

{"names", mstch::array{

mstch::map{{"name", std::string{"Chris"}}},

mstch::map{{"name", std::string{"Mark"}}},

mstch::map{{"name", std::string{"Scott"}}},

}}

};

std::cout << mstch::render(view, context) << std::endl;

return 0;

}

Резултатът е:

Hi Chris!

Hi Mark!

Hi Scott!

### **Структура на данните**

### Използваните типове данни в примера са mstch::array и mstch::map,които всъщност са alias-и за стандартните типове:

using map = std::map<const std::string, node>;

using array = std::vector<node>;

mstch::node е boost::variant, което може да съдържа std::string, int, double, bool, mstch::lambda или astd::shared\_ptr<mstch::object>, също масив или map, който работи като JSON обект.

Важно е да се отбележи, че когато използваме std::string като стойност, трябва да поясним експлицитно типа, докато const char\*, например "foobar" ще бъде неявно конвертирано до bool.

1. За mustache

Mustache може да се използва в HTML, конфигурационни файлове, файлове с код - навсякъде. Mustache работи като разширява таговете в темплейта, използвайки стойности от хеш таблици или обекти.

Нарича се "logic-less" темплейт, тоест темплейт, който не съдържа логика - няма if условия, else клаузи или for цикли. Вместо тях се използват само тагове. Някои тагове се заместват със стойност, други с нищо, трети - с поредица от стойности.