Препроцессоры

LoftSchool
от мыслителя к создателю

Содержание

1

О ПРЕПРОЦЕССОРАХ

В широком смысле слова, ***препроцессор*** – это программа, которая производит некоторые манипуляции с первоначальным текстом программы перед тем, как он подвергается компиляции.

Но что же такое препроцессор в контексте HTML и CSS?
Можно сказать, что препроцессоры компилируют код, который мы пишем на процессорном языке в чистый html (css) код.

Препроцессор – это надстройка над html (css), которая позволяет использовать новые синтаксические конструкции, такие как переменные, или, например, операторы и функции.

Задача препроцессора состоит в том, чтобы предоставить разработчику возможности, которые ускоряют разработку и упрощают поддержку стилей в проектах.

Препроцессоры, как и любой другой инструмент, имеет свои плюсы и минусы.

Положительные стороны:

- Модульность и структурированность
- Повышение производительности
- Читабельность для разработчика

Отрицательные стороны:

- Большое количество инструментов
- Высокий уровень абстракции для простых элементов
- Дополнительные возможности имеют ограничения, если используются неразумно

На сегодняшний день, препроцессоры продолжают развиваться, и разработка без использования препроцессоров считается плохой практикой. Эволюция препроцессоров повлекло за собой создание различных руководств по написанию стилей.

Говоря о большом количестве инструментов, можно привести в пример некоторые из них: Less, Stylus, Sass, ClosureStylesheets и др.

В данной методичке будут рассмотрены две технологии, которые являются наиболее популярными в своей среде. Для HTML это препроцессор Pug (Jade), а для CSS это препроцессор Sass.

2

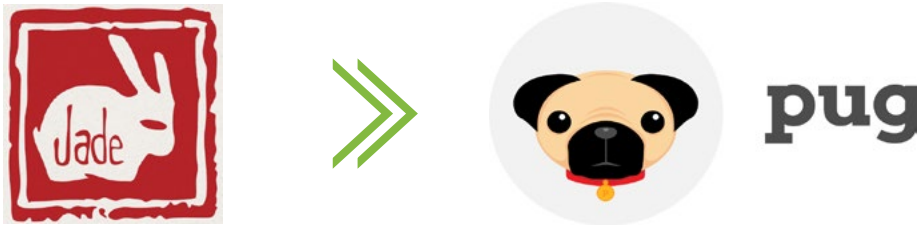
PUG (JADE)

Что такое Pug (Jade) и способ его установки

Jade — это препроцессор html или язык шаблонов, реализованный на JavaScript для Node.js.

Jade ещё называют высокопроизводительным шаблонизатором.

Источником вдохновения для создания Jade послужил язык разметки Haml. С недавних пор, из-за разногласий с одноимённой компанией по поводу торговой марки, было решено переименовать препроцессор jade в pug.



Перечислим основные возможности и преимущества Pug:

- Возможность работы на стороне клиента
- Простое описание тегов
- Схожесть синтаксиса с CSS
- Использование системы отступов, обеспечивающей правильный каскад
- Возможность объявлять переменные, функции (mixins)
- Шаблонизация (includes)
- Разные способы компиляции

Ссылки на официальную документацию и гитхаб:

http://jade-lang.com/

https://github.com/pugjs/pug

Установка

Как уже говорилось, pug работает на node.js, поэтому для работы с pug необходимо предварительно **произвести установку node.js** на ваш компьютер.

Установка самого препроцессора pug производится через npm (идёт в комплекте с node.js) в консоли следующей командой:

```
$ npm install pug –global
```

2.1.Синтаксис Pug

Синтаксис Pug немного напоминает синтаксис, который используется в CSS. Далее рассмотрим по порядку все синтаксические конструкции Pug.

Теги, классы, id и комментарии

Ниже показано, как в pug объявляются классы, id, теги и комментарии (здесь и далее пример **html-кода** будет иметь `светло-серый фон`, пример **pug** – `светло-синий`):

```
// объявили класс
.class-example
nav.navigation-example
// объявили несколько классов
p.first-class.second-class
// объявили id
div#id-example
```

Заметим, что вложенность в pug соблюдается благодаря системе отступов. Вышеуказанный пример при компиляции преобразуется в html:

```
<!-- объявили класс-->
<div class="class-example">
<nav class="navigation-example"></nav>
<!-- объявили несколько классов-->
<p class="first-class second-class"></p>
<!-- объявили id-->
<div id="id-example"></div>
```

Если текст, например, в параграфе, слишком большой, благодаря символу «|» мы можем его перенести на новую строку без учёта отступа:

```
p.first-class.second-class
| Мы перенесли текст
| на новую строку
```

```
<p class="first-class second-class">
  Мы перенесли текст
  на новую строку
</p>
```

Pug так же поддерживает блочные и небуферизируемые комментарии:

```
//- для того, что бы комментарий не компилировался,
достаточно поставить в начале дефис
// блочные комментарии задаются отступом
  .comments-example
    .block-comment
```

Атрибуты и раскрытие блоков

Атрибут в pug указываются после объявления тега или класса в круглых скобках:

```
// атрибуты в pug
a.attribute-example(href='loftschool.ru',
title='PugAtr')
```

```
<!--атрибуты в pug-->
<a href="loftschool.ru" title="PugAtr"
class="attribute-example"></a>
```

В puge сть возможность писать атрибуты с новой строки. Так же стоит отметить, что pug поддерживает одиночные атрибуты, например, в теге input атрибут checked:

```
input(type='checkbox'
name='checkbox-example'
checked='checked')
```

```
<input type="checkbox" name="checkbox-example"
checked="checked">
```

Раскрытие блоков позволяет писать компактные однострочные конструкции вложенных тегов. Покажем на примере списка:

```
// список вложенный тегов
ul.block-list
li.block-item-first
a(href='#') Я ссылка
li.block-item-second
a(href='#') И Я ссылка
```

```
// тот же самый список с использованием раскрытия блоков
ul.block-list
li.block-item-first: a(href='#') Я ссылка
li.block-item-second: a(href='#') И Я ссылка
```

Переменные

В любом препроцессоре есть переменные, и pug этому не исключение. Объявление и использование переменных является одной из главных особенностей всех препроцессоров. В pug переменные объявляются двумя способами. Первый способ – это объявление переменной непосредственно в самом файле, с которым мы работаем. Второй способ заключается в том, что мы можем вынести наши переменные в отдельный файл, и при помощи зарезервированного слова **«include»** (которое мы рассмотрим чуть позже) подключить их в том месте, где мы их будем использовать.

Давайте теперь рассмотрим использование переменных на примерах.

Для объявления переменной внутри файла используется зарезервированное слово **«var»**:

```
// самое простое объявление и использование переменной
- var pig = 'Peppa'
h1 Hello, my name is #{pig}
```

```
<h1>Hello, my name is Peppa</h1>
```

Помним, что переменные нам не нужно компилировать, и поэтому перед «var» ставим дефис, чтобы не буферизировать их объявление.

Переменные в pug могут объявляться как до их вызова, так и после. Рассмотрим ещё один способ вызова переменных:

```
// мы можем объявить массив переменных
- var personalInfo = {gender: 'girl', age: '5', skinColor:
'pink'}
h2= personalInfo.gender
h3 I am #{personalInfo.age} years old. My skin is
#{personalInfo.skinColor}
```

```
<h2>girl</h2>
<h3>I am 5 years old. My skin is pink</h3>
```

Переменные можно объявлять внутри другой переменной:

```
- var personalInfo = {name: 'Peppa', gender: 'girl', age:
'5', skinColor: 'pink'}
// используем переменную в новой переменной myName
- var myName = 'Awesome, my name is ' + personalInfo.name
h2= myName
```

```
<h2>Awesome, my name is Peppa</h2>
```

Как вы уже поняли, мы можем использовать переменные везде, где только они могут нам понадобиться. Посмотрим на последний пример использования переменной в атрибутах:

```
- var inputAtr = {type: 'password', text: 'please, enter
your password'}
input(type = inputAtr.type, placeholder = inputAtr.text)
```

```
<input type="password" placeholder="please, enter your
password">
```

Условия

Рассмотрим, как реализованы условия в препроцессоре pug. В pug существуют следующие условные конструкции – **«if/else»**, **«when»**, **«while»**, **«for/each»**. Далее рассмотри пример использования вышеперечисленных условий по порядку.

```
- var name = 'Peter Parker'
if name == 'Peter Parker'
p Amazing Spider-Man!
else
p Hello, I am #{name}
```

```
<p>Amazing Spider-Man!</p>
```

Перед «if» и «else»дефис можно не ставить. Pug поймёт, что вы хотите использовать в данном случае.

«When» используется в сочетании с «case», и немного отличается от «if/else» по выполнению условия.Впрочем, пример скажет всё сам за себя:

```
- var superhero = 'Peppa'
case superhero
when 'Bruce Wayne'
p Incredible Batman!
when 'Peter Parker'
p Amazing Speder-Man!
default
p #{superhero} Pig
```

```
<p>Peppa Pig</p>
```

Каждый раз, при помощи «when», мы проверяли, совпадает ли переменная, указанная после «case», с указанным именем. Если нет, то переходим к следующему условию до тех пор, пока не появится совпадение или условие по умолчанию.

«While». В pug работает подобно одноимённому оператору перебора, который есть практически во всех языках программирования:

```
- var n = 0
ul.list
while n <3
li.item= n++
```

```
<ul class="list">
<li class="item">0</li>
<li class="item">1</li>
<li class="item">2</li>
</ul>
```

«Each». Используется, когда нам необходимо по порядку перебрать все имеющиеся элементы:

```
// пройдёмся с помощью each по всем элементам массива и
выведем вместе с данными номер каждой итерации в атрибуте
тега li
ul
  each val, index in ['Peppa', 'Bruce Wayne', 'Peter
Parker']
li(value=index)= index + ': ' + val
```

```
// дополнительно покажем, что есть возможность указать
свой ключ
each val, index in {1: 'Peppa', 2: 'Bruce Wayne', 3:
'Peter Parker'}
li(value=index)= index + ': ' + val
```

```
<ul>
<li value="0">0: Peppa</li>
<li value="1">1: Bruce Wayne</li>
<li value="2">2: Peter Parker</li>
<!-- дополнительно покажем, что есть возможность указать
свой ключ-->
<li value="1">1: Peppa</li>
<li value="2">2: Bruce Wayne</li>
<li value="3">3: Peter Parker</li>
</ul>
```

«For». Работает аналогично любому одноимённому циклу в языках программирования. Давайте попробуем переписать предыдущий пример цикла, только теперь используя «for»:

```
- var superheroes = ['Peppa', 'Bruce Wayne', 'Peter
Parker'];
ul
- for (var i=0; i<=2; i++) {
li= superheroes[i]
  - }
```

```
<ul>
<li>Peppa</li>
<li>Bruce Wayne</li>
<li>Peter Parker</li>
</ul>
```

Примеси (mixins)

Перейдём к, пожалуй, самой важной возможности препроцессора jade– примесям, или как их ещё называют, миксинам.

Говоря о примесях, можно сказать, что они сродни функциям. Примеси, также, как и функции, могут принимать параметры в качестве входных данных и генерируют соответствующую разметку. Для того что бы объявить такую «функцию» в pug, необходимо использовать ключевое слово mixin. Примеси дают разработчику возможность не дублировать свой код.

```
// объявим миксин, который будет выводить название команды
mixinnameOfTeam(name)
p#{name} team

// вызовем наш миксин
+nameOfTeam('GeorgeMartin')

// объявим миксин, который будет выводить участника
команды
mixinteam(person, post)
li#{person} : #{post}
ul.teamList
  +team('JonSnow', 'director')
  +team('DaenerysTargaryen', 'front-end')
  +team('NedStark', 'back-end')
  +team('TyrionLannister', 'manager')
  +team('AryaStark', 'designer')
```

```
<p>George Martin team</p>
<ul class="teamList">
<li>Jon Snow : director</li>
<li>Daenerys Targaryen : front-end</li>
<li>Ned Stark : back-end</li>
<li>Tyrion Lannister : manager</li>
<li>Arya Stark : designer</li>
</ul>
```

Из примера видно – для того, чтобы вызвать наш миксин, после того как он был оглашён, необходимо использовать символ «+». Часто примеси используются, когда необходимо просто часто дублировать какой-то относительно большой кусок кода.

Входные параметры примесей так же могут использоваться в качестве данных атрибутов. В следующем примере покажем это, а также посмотрим, как можно использовать миксин с неизвестным заранее количеством входных параметров:

```
mixin warriors(name, ...items)
ul(class=name)
each item in items
// &attributes (atributes) – позволяет нам после вызова
миксина добавлять атрибуты, не ломая наш код
li&attributes(attributes)= 'Number ' + item
+warriors('samurais', 145, 146, 147)(class='samurai')
```

```
<ul class="samurais">
<li class="samurai">Number 145</li>
<li class="samurai">Number 146</li>
<li class="samurai">Number 147</li>
</ul>
```

Из вышепоказанного примера видно, что примеси позволяют использовать любое количество параметров, какое нам может понадобиться в процессе разработки проекта. Для этого достаточно поставить «...» перед именем нашего параметра.

2.2. Шаблонизация страниц в pug

Говоря о шаблонизации страниц в pug, можно отдельно выделить две, так называемые, директивы – **include** и **extends**. Давайте по порядку разберём, что каждая из этих директив собой представляет и какие у них функции.

Include

Зачастую разработчики сталкиваются с тем, что в процессе работы необходимо изменить на странице какой-нибудь элемент. Давайте представим, что у нас в проекте 5 разных страниц, но в каждой из них присутствует одинаковый элемент (такими элементами могут быть header, footer, навигация и др.). Менять на каждой странице один и тот же элемент занимает много времени, когда есть возможность поменять один раз в одном месте. Вот с такой задачей и справляются в pug рассматриваемые нами инклюды. Можно сказать, что инклюды это аналог import в CSS.
Pug имеет возможность выделять общие блоки в отдельные файлы и подключать их на странице через include. Если говорить проще, то инклюды позволяют вставлять контент из одного pug-файла в другой.

Рассмотрим пример в котором у нас есть папка **_includes**. Как вы уже догадались, в ней мы храним те pug файлы, которые будем подключать с помощью include в наш основной файл:

```
//- _includes/_head.pug – здесь лежит шаблон тега head
head
  style
//- обратите внимание, что pug позволяет вставлять в
документ не только pug-файлы, но и необходимый код стилей
css или javascript
include ../../style/style.css
title Site about everything

//- _includes/_head.pug – здесь лежит блок header
header.header
h2 Hello, I am header
```

Подключаем эти файлы на нашу общую страницу *my.pug*:

```
doctype html
html
//- подключаем файл _head
include ../_includes/_head
body
//- подключаем файл _header
include ../_includes/_header
p 42!
p You know all about includes
```

В итоге весь pug-файл скомпилируется в следующий html-файл:

```
<!DOCTYPE html>
<html>
<head>
<style>
/* style.css */
h2 { color: green; }
</style>
<title>Site about everything</title>
</head>
<body>
<header class="header">
<h2>Hello, I am header</h2>
</header>
<p>42!</p>
<p>You know all about includes</p>
</body>
</html>
```

Вот и всё. Работать с инклюдами проще простого!

Extends

Давайте теперь рассмотрим директиву extends, которую в pug называют ещё расширением. Можно сказать, что extends – это блоки, которые нужны для создания шаблонов (лейаутов). При помощи extends мы можем просто наследовать некий шаблон станицы. Лучше всего, применение расширений, показать на новом примере.

Давайте возьмём наш прошлый пример, где мы подключали кусочки кода в основной файл **my.pug**. Пускай у нас будет вторая страница **cat.pug**. Эти две страницы абсолютно идентичны, кроме того, что у **cat.pug** другой **title**. Посмотрим теперь на примере, как extends нам теперь поможет создать общий шаблон этих двух страниц, которые его унаследуют (для простоты примера поместим наш блок **head** в общий шаблон **general.pug**):

```
//- general.pug - общий шаблон, от которого наследуются
наши страницы
doctype html
html
  head
//- объявляем наш блок
block title
//- title по умолчанию
title General default title
body
    block content

//- my.pug - наследуется от общего шаблона general
extends ../general

block title
title this is My title

block content
include ../_includes/_header
p 42!
p You know all about includes

//- cat.pug - также как и my.pug наследуется от общего
шаблона general
extends ../general

block title
title this is Cat title. Mew

block content
//- подключаем файл _header
include ../_includes/_header
```

```
p 42MEW!
p mew-mew-mew
```

В итоге на выходе получаем два html файла (my.html и cat.html соответственно):

```
<!DOCTYPE html>
<html>
<head>
<title>this is My title</title>
</head>
<body>
<header class="header">
<h2>Hello, I am header</h2>
</header>
<p>42!</p>
<p>You know all about includes</p>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
<title>this is Cat title. Mew</title>
</head>
<body>
<header class="header">
<h2>Hello, I am header</h2>
</header>
<p>42MEW!</p>
<p>mew-mew-mew</p>
</body>
</html>
```

С помощью **extends** и конструкции **block** мы можем легко изменять необходимый нам контент, унаследовав общий шаблон.

2.3. Фильтры

Фильтры в pug позволяют использовать другие языки в шаблонах. Фильтры обозначаются префиксом «:». Например, фильтр, который позволяет использовать облегчённый язык разметки markdown, пишется так «:markdown». После объявления фильтра, ему передаётся на обработку последующий блок текста. Приведём пример использования фильтра markdown в pug:

```
:markdown
My answer: **42!**
  # You know all about includes
  ## You know all about includes
  ### You know all about includes
```

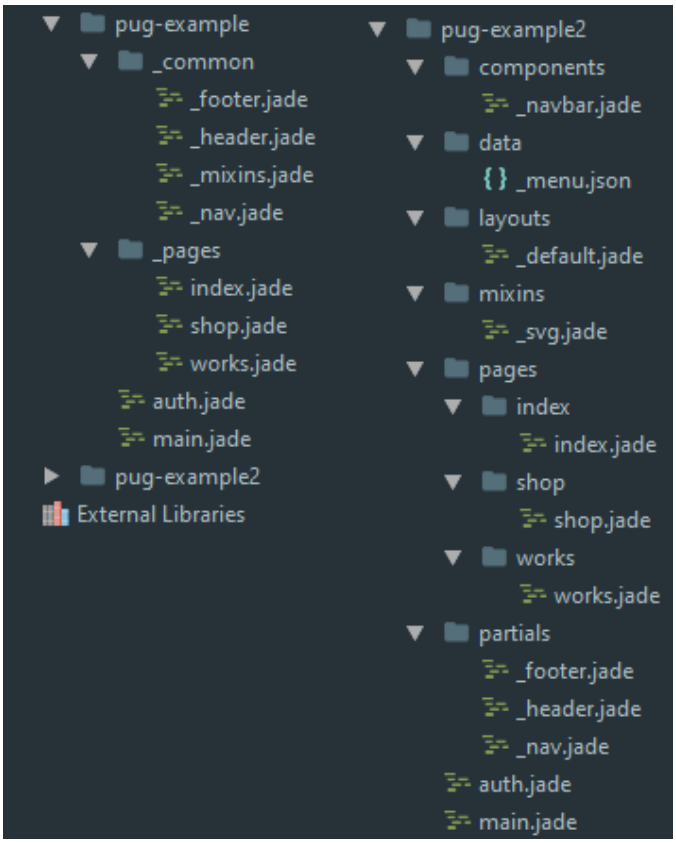
```
<p>My answer: <strong>42!</strong></p>
<h1 id="you-know-all-about-includes">You know all about
includes</h1>
<h2 id="you-know-all-about-includes">You know all about
includes</h2>
<h3 id="you-know-all-about-includes">You know all about
includes</h3>
```

Ниже представлен список наиболее популярных фильтров pug:

- :sass
- :less
- :markdown
- :coffee-script
- :babel

2.4. Организация файловой структуры проекта в pug

Используя любой препроцессор в проекте, важно задаться вопросом — как изначально построить структуру, чтобы облегчить дальнейшую разработку и поддержку проекта? И препроцессор pug не исключение.
Хотя, по своей сути, построение структуры никак не зависит от того или иного препроцессора. Как и какую структуру использовать мы выбираем сами, в зависимости от наших потребностей и предпочтений. Давайте посмотрим два примера разных структур, которые можно использовать для работы с pug.



Две организации файловой структуры проекта в pug

Если мы посмотри на структуры первого и второго скриншота, то увидим, что первая структура является намного проще.

На первом примере в директории **_common** хранятся все наши файлы-части шаблонов вместе с необходимыми миксинами, которые записаны в отдельный файл. В папке **_pages** хранятся непосредственно наши страницы, которые наследуются от шаблонов в корне проекта – **auth.jade** и **main.jade**. Всё очень просто.

Во втором примере организация немного сложнее:

1) Директория **components** – здесь находятся такие компоненты, из которых будет собираться вся страница. Например, блок социальных кнопок, dropdown-меню, глобальная навигация и другие.

1) Директория **data** – содержит в себе какие-либо необходимые данные. В pug есть возможность использовать на страницах данные jsonформата

1) Директория **mixins** – здесь хранятся все необходимые нам примеси

1) Директория **pages** – содержит основную часть каждой страницы, представленную файлом **_main.jade** и директорией с именем, совпадающим с именем файла точки входа страницы. Отдельные директории для каждой страницы создаются для того, если появится необходимость вынести содержимое страницы в отдельные файлы (например, страница содержит большое количество текста, который можно вынести в отдельный markdown-файл).

1) Директория **partials** – в ней хранятся «кирпичики» шаблона, которые мы используем на разных страницах.

3

SASS

Что такое Sass

На сегодняшний день ***Sass (syntactically awesome stylesheets)*** является самым мощным css-препроцессором. Препроцессор Sass основан в 2007 году как модуль для HAML и написан на языке программирования Ruby.

Sass имеет два синтаксиса:

1) **sass** – упрощённый синтаксис css, который отличается отсутствием фигурных скобок, а разделение свойств точкой с запятой заменяется новой строкой;
2) **scss** (SassyCSS) – основан на стандартном синтаксисе CSS, как и CSS использует фигурные скобки

Перечислим основные преимущества sass:

- Полная совместимость с CSS – возможность использовать любые доступные библиотеки CSS;
- Большое количество функциональных возможностей;
- Возможности препроцессора расширяются за счёт многофункциональности разных библиотек, построенных на sass – Compass, Bourbon, Susy и другие;
- Большое сообщество разработчиков и поддержка индустрией

Установка

Перед использованием Sass, для пользователей операционных систем Windows и Linux, необходимо установить ***Ruby*** на ваш компьютер.

Для пользователей Linux (установка через пакетные менеджеры apt, rbenv или rvm):

```
sudo su –c "gem install sass"
```

Для пользователей Windows используется **установщик Ruby**.

Если вы пользователь Mac, то в установке Ruby нет необходимости, так как он уже предустановлен по умолчанию.

Установка самого Sass происходит через командную строку. Ruby использует gem для управления различными пакетами. Для установки пакета Sass необходимо выполнить следующую команду:

```
gem install sass
```

Теперь пакет Sass установлен на ваш компьютер. Для проверки необходимо выполнить команду, которая должна вывести в терминале текущую версию Sass:

```
sass –v
```

Ссылка на официальную документацию на русском языке:

http://sass-scss.ru/documentation/

Компиляция

Самый прямой и быстрый способ скомпилировать ваш sass (scss) код в css – это использовать терминал. Сразу после того как вы установили препроцессор Sass, вам стала доступна консольная программа «sass».Вы можете компилировать ваш sassфайл в css аналогично этому примеру:

```
sass input.scss output.css
```

Компилировать файлы sass можно используя ключ **–*--watch***. Этот ключ говорит программе следить за указанными файлами. Т.е. один раз запустив команду, sass пересоберёт все необходимые css-файлы:

```
sass––watch app/sass:public/stylesheets
```

В данном примере sass отслеживает все изменения файлов в каталоге **app/sass**, а результат сохраняет в каталоге **public/stylesheets**. Для большей наглядности, данную конструкцию можно обозначить так: **sass –watch [что]:[куда]**.

3.1. Синтаксис, директивы и миксины Sass

В этой части рассмотрим синтаксис и примеры основных возможностей препроцессора Sass.

Вложенность

Препроцессор Sass позволяет вкладывать CSS селекторы иерархично. Таким образом, вложенность делает ваш код более читабельным. Посмотрим на простом примере, как работает вложенность sass (здесь и далее показан синтаксис scss):

```
.parent {
  .first-children {
background: #000;
  }
  .second-children {
background: #fff;
  }
}
```

```
.parent .first-children {
background: #000; }

.parent .second-children {
background: #fff; }
```

В этом и последующих примерах синтаксис scss имеет светло-синий фон, а сгенерированный css-код, который получаем на выходе, имеет светло-серый фон (аналогично примерам в части про pug).

Вложенные правила помогают избежать повторения родительских селекторов, и делают код более лаконичным.

Иногда нам необходимо использовать специальные стили, например, псевдокласс hover. Для этого мы можем использовать символ «&» - ссылка на родителя элемента. С помощью него мы можем указать где будет вставлен родительский селектор:

```
.parent {
color: silver;
&:hover {
color: gold;
  }
}
```

```
.parent {
color: silver; } \
  .parent:hover {
color: gold; }
```

Символ «&» также удобно использовать, когда необходимо добавить к родительскому селектору последующий суффикс:

```
ul .main-list {
position: relative;
li &__item {
position: absolute;
top: 0;
  }
}
```

```
ul .main-list {
position: relative; }
li ul .main-list__item {
position: absolute;
top: 0; }
```

Интересное свойство вложенности можно наблюдать в каком-либо пространстве имён. Например, рассмотрим пространство имён ***font***:

```
.link-example {
font: {
family: "Roboto Mono";
size: 20em;
weight: 700;
  }
}
```

```
.link-example {
font-family: "Roboto Mono";
font-size: 20em;
font-weight: 700; }
```

Видим, что Sass позволяет упростить процесс написания кода – в примере мы единожды написали пространство имён, а внутри уже пишем вторичное свойство.

Переменные и типы данных

Прежде чем перейдём к рассмотрению переменных, узнаем, что такое SassScript.

SassScript – это дополнительный набор расширений, поддерживаемый Sass. Благодаря ему появляется возможность использовать переменные и арифметические функции, создавать селекторы и имена свойств, что полезно при написании миксинов, которые будут рассмотрены чуть позже.

Переменные – это самое простое, что поддерживается SassScript. Для задания переменной используется символ «$». Переменные доступны только в пределах того уровня вложенности селекторов, на котором они определены:

```
// объявим переменную главного цвета
$mainColorWhite: #f8f8ff;
// используем нашу переменную внутри селектора
.variable-example {
color: $mainColorWhite;
}
```

```
.variable-example {
color: #f8f8ff; }
```

SassScript поддерживает семь основных типов данных:

- Числа (21px, 21, 2.1)
- Строки (“string”, ‘string’, string)
- Цвета (gold, #ffd700, rgba(255, 215, 0, 1))
- Булевы значения (true, false)
- Null
- Списки значений (Roboto Mono, sans-serif; 1.5em 1em 0 2em)
- Массивы (key1: value1, key2: value2)

Стоит упомянуть, что каждый тип данных поддерживает свой набор определённых операций (операции равенства, неравенства, сложения, деления...). Например, мы можем перемножить два числовых значения или сложить два разных значения цветов. Подробнее об операциях над типами данных можно посмотреть в документации, ссылка на которую была приведена выше.

Функции

SassScript включает в себя некоторые полезные функции, которые можно вызвать в процессе написания стилей:

```
// покажем пример использования функции random
// объявим переменную, которая будет задавать лимит для
функции случайного числа
$randomLimit: 9;
// используем нашу переменную внутри функции random
.function-random {
opacity: 0 + #{'.'} + random($randomLimit);
}
```

```
.function-random {
opacity: 0.3; }
```

Полный список доступных функций можно посмотреть на странице **английской документации Sass**.

Правила и директивы

В Sass есть поддержка всех существующих CSS @-правил и поддержка дополнительных правил, которые именуются «директивами».

«@import». Директива import позволяет импортировать sass файлы, которые могут быть в дальнейшем объединены в один css файл. Переменные и миксины, объявленные в импортированном файле, могут использоваться в главном файле. По-умолчанию import ищет Sass-файлы. Если расширения файлов не указаны, то Sass будет пытаться искать файлы по имени с расширением .sass или .scss, которые необходимо импортировать:

```
// импортирует весь код указанных файлов
@import'_auth.scss';
@import'_first.scss'';
// можно не указывать расширение – Sass так же определит
нужные файлы
@import'_auth';
@import'_first';
```

«@media». Директива media работает практически также, как и в привычном CSS. Если директива вложена в css-правило, то при компиляции она будет поднята наверх таблицы стилей, а все селекторы в которых была директива переместятся внутрь.

Медиа-запросы в Sass могут вкладываться друг в друга. В таком случае, после компиляции они просто объединятся оператором «and»:

```
// пример директивы media и использования в ней
объявленной переменной
$widthOfDesktop: 1800px;
$mainMedia: screen;
@media #{$mainMedia} {
  .media-example {
@media (width: $widthOfDesktop) {
margin-top: 20px;
    }
  }
}

@media screen and (width: 1800px) {
  .media-example {
margin-top: 20px; } }
```

Как вы уже поняли из примера, удобно использовать все возможности SassScript (в данном случае использовали переменную) в медиа запросах.

«@extend». Эта директива позволяет выполнять функцию наследования в Sass. Используя exted, можно наследовать наборы свойств от одного селектора к другому. Это позволяет держать Sass-файлы «чистоте»:

```
.example-extend {
border: 1px solid #000;
color: #000;
}
.good-message {
@extend .example-extend;
border-color: green;
}
.error-message {
@extend .example-extend;
border-color: red;
}
```

```
.example-extend, .good-message, .error-message {
border: 1px solid #000;
color: #000; }

.good-message {
border-color: green; }

.error-message {
border-color: red; }
```

В нашем примере код позволяет взять свойства CSS из example-extend и применить их в good-message и error-message.

Эти три рассмотренные директивы чаще всего используются при работе с Sass. Так же существуют такие директивы как **@at-root**, которое изымает одно или несколько правил из родительского селектора в корневой уровень документа, **@debug**, **@warn** (выводят значения выражений Sass средствами стандартного ввода ошибок) и **@error** (отображает значения выражений и функций как фатальную ошибку).

Выражения

SassScript поддерживает так называемые управляющие директивы и выражения для подключения стилей при определённых условиях или в циклах с изменениями.Обычно такие директивы целесообразно использовать при создании миксинов. Но для полного понимания давайте рассмотрим каждую из них.

@if.

Данная директива принимает какое-то выражение. Если это выражение возвращает true, то условие выполняются и стили, вложенные в неё, применяются. Обратим внимание, что в примере показана также возможность использовать выражение ***@elseif***:

```
// пример использования директивы @if
$variableColor: firstRainbowColor;
.if-example {
@if $variableColor == secondRainbowColor {
color: orange;
  } @else if $variableColor == thirdRainbowColor {
color: yellow;
  } @else if $variableColor == firstRainbowColor {
color: red;
  } @else {
color: violet;
  }
}
```

```
.if-example{
color: red; }
```

@for.

Данная директива реализует стандартный цикл. Директива имеет две формы:

1) @for $var from <начало>through <конец>
2) @for $var from <начало>to <конец>

Первое выражение отличается от второго тем, что слово ***through*** говорит нам, что значение <конец> включается в диапазон значение, а ***to-*** наоборот не включено. Приведём пример, в котором будем проходиться от 1-го до 3-ёх и выводить разный результат для каждого элемента:

```
// пример использования директивы @for
@for $i from 1 through 3 {
  .list-item-#{$i} { width: 20px * $i; }
}

.list-item-1 {
width: 20px; }

.list-item-2 {
width: 40px; }

.list-item-3 {
width: 60px; }
```

@each.

Директива each также, как и for по своему функционалу является циклом. Each проходится по каждому значению массива или данных, которые переданы в неё. Обычно имеет следующую форму:

@each $varin<список или карта значений>

Приведём простой пример:

```
// пример использования директивы @each
// пройдёмся по списку цветов радуги
@each $color in red, orange, yellow, green {
  .#{$color}-col {
color: $color;
  }
}

.red-col {
color: red; }

.orange-col {
color: orange; }

.yellow-col {
color: yellow; }

.green-col {
color: green; }
```

@while.

Эта директива циклично будет выводить вложенные в неё стили, до тех пор, пока выполняется заданное условие. Эта директива может быть использована для сложных циклов, но на практике используется реже остальных.

Миксины (@mixin).

Миксины, или как их ещё называют, примеси позволяют определить стили, которые могут быть использованы повторно в разных файлах и разных местах документов. Миксины, как и функции, могут принимать аргументы и содержать целые css правила. Миксины в Sass очень гибкие в работе и являются одним из самых важных инструментов.

Чтобы объявить миксин, используется директива ***@mixin***, после которого стоит имя нашего миксина. Что бы вызвать наш объявленный миксин, используется директива ***@include***, которая принимает имя миксина.

Приведём пример создания примеси для простоты работы со свойством text-decoration:

```
// создадим миксин, укажем необходимые аргументы и
присвоим им значения по умолчанию
@mixin example-style-link($color: blue, $line: underline)
{
text-decoration: {
color: $color;
line: $line;
style: double;
  }
}
// для вызова нашего миксина используем @include
a {
  .first-link {
@include example-style-link(red);
  }
}
a {
  .second-link {
@include example-style-link(green, none);
  }
}
```

```
a .first-link {
-webkit-text-decoration-color: red;
text-decoration-color: red;
-webkit-text-decoration-line: underline;
text-decoration-line: underline;
-webkit-text-decoration-style: double;
text-decoration-style: double; }

a .second-link {
-webkit-text-decoration-color: green;
text-decoration-color: green;
-webkit-text-decoration-line: none;
text-decoration-line: none;
-webkit-text-decoration-style: double;
text-decoration-style: double; }
```

По мере работы над разными проектами, у вас будет накапливаться своя библиотека миксинов, которые вы чаще всего будете использовать. Так же не стоит забывать про такие библиотеки, как Compass, которые уже имеют огромное количество рабочих примесей.