НЕТОЛОГИЯ

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

АЛЕКСЕЙ СУДНИЧНИКОВ / ВЕБ-РАЗРАБОТЧИК

АЛЕКСЕЙ СУДНИЧНИКОВ

Веб-разработчик

@avsudnichnikov

ПЛАН ЗАНЯТИЯ

- Задача
- Регулярные выражения
- Методы объекта String
- Методы объекта Regexp
- Новые возможности регулярных выражений
- Когда не стоит использовать Regexp

ЗАДАЧА

ЗАДАЧА

Перед нами поставили следующую задачу: нам надо произвести валидацию адреса электронной почты: проверить, что адрес электронной почты содержит знак `@`.

Каким способом можно это сделать?

ПОСИМВОЛЬНЫЙ ПЕРЕБОР

Можно перебрать каждый символ (самое неправильное решение):

```
function validateEmail(emailStr) {
  for (const itemSymbol of emailStr) {
    if (itemSymbol === '@') {
      return true;
    }
  }
  return false;
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('supportnetology.ru'));

// -> true
// -> false
```

INDEXOF

Можно воспользоваться `indexOf`:

```
function validateEmail(emailStr) {
  return emailStr.indexOf('@') !== -1;
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('supportnetology.ru'));

// -> true
// -> false
```

INCLUDES

А ещё лучше сразу воспользоваться `includes`:

```
function validateEmail(emailStr) {
  return emailStr.includes('@');
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('supportnetology.ru'));

// -> true
// -> false
```

REALITY

В реальных задачах редко требуется найти совпадение лишь по одному символу, зачастую требуется, чтобы строка нескольким условиям.

Например, чтобы понять, насколько жизнеспособно каждое из наших решений, давайте введем еще одно правило - в адресе электронной почты после собаки должна присутствовать точка.

На самом деле, по спецификации email не обязан содержать точку, но в большинстве случаев вы встретите именно такое требование.

Первое выражение примет прямо-таки страшный вид:

```
function validateEmail(emailStr) {
  let foundCommercialAt = false;
  for (const itemSymbol of emailStr) {
    if (itemSymbol === '@') {
      foundCommercialAt = true;
    }
    if (itemSymbol === '.' && foundCommercialAt) {
      return true;
    }
  }
  return false;
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('supportnetology.ru'));
```

Второе решение попроще:

```
function validateEmail(emailStr) {
  const commercialAtPos = emailStr.indexOf('@');
  return commercialAtPos !== -1 && commercialAtPos < emailStr.indexOf('.');
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('supportnetology.ru'));
```

Решение с `includes` придётся свести к `indexOf`, т.к. теперь имеет значение позиция.

СЛОЖНОСТИ

Если же еще ввести требования, что собака не является первым символом адреса, а точка не является последней...

Первый и второй способ станут еще сложнее.

Первый способ (не делайте такого, пожалуйста, в своей работе):

```
function validateEmail(emailStr) {
  let foundCommercialAt = false;
  const cuttedEmailStr = emailStr.substring(1, emailStr.length - 1);
  for (const itemSymbol of cuttedEmailStr) {
    if (itemSymbol === '@') {
      foundCommercialAt = true;
    }
    if (itemSymbol === '.' && foundCommercialAt) {
      return true;
    }
  }
  return false;
}
```

```
console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));

// -> true
// -> false
// -> false
// -> false
// -> false
// -> false
```

Второй способ:

```
function validateEmail(emailStr) {
  const commercialAtPos = emailStr.indexOf("@");
  const dotPos = emailStr.indexOf(".");
  if (commercialAtPos <= 0) {
    return false;
  }
  if (dotPos < commercialAtPos) {
    return false;
  }
  if (dotPos === emailStr.length - 1) {
    return false;
  }
  return true;
}
```

```
console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));

// -> true
// -> false
// -> false
// -> false
// -> false
// -> false
```

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

ЧТО ТАКОЕ "РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ"?

MDN Web Docs:

Регулярные выражения - это шаблоны используемые для сопоставления последовательностей символов в строках.

А также частное название самих выражений, на которых основана технология.

REGEXP

Давайте начнём сразу с простого, проверим, есть ли `@` в строке:

```
function validateEmail(emailStr) {
  return /@/.test(emailStr);
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('supportnetology.ru'));
```

REGEXP

В JS регулярные выражения (как и, например, массивы), можно записывать двумя способами:

1. `/regexp/` - литерал
2. `new Regexp(regexp)` - через вызов функции-конструктора

КАК ЭТО РАБОТАЕТ?

Фактически, движок регулярных выражений сканирует всю строку (например, `support@netology.ru`) пока не найдёт совпадения с символом `@` (за это отвечает метод `test`).

REGEXP

Давайте а теперь `@`, за которой идут любые символы, кроме точки, а затем `.`:

```
function validateEmail(emailStr) {
  return /@[^.]+\./.test(emailStr);
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('supportnetology.ru'));
```

Третий же притерпит всего пару изменений:

```
function validateEmail(emailStr) {
  return /@[^.]+\.\w/.test(emailStr);
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));

// -> true
// -> false
// -> false
// -> false
// -> false
// -> false
```

СИНТАКСИС

Синтаксис регулярок (регулярных выражений) достаточно прост.

`/регулярное выражение/` записывается между символами "/" (в виде литерала)

Например: `/my_regexp/`

По умолчанию, каждый символ означает самого себя (например `@`).

СИНТАКСИС

Но есть и исключения:

- `[abc]` - подразумевает любой из символов группы
- `[^abc]` - подразумевает любой **не** из символов группы
- `[a-z]` - подразумевает любой из символов a и z

Если необходимо отменить специальное действие [, тогда это записывается как `\[`.

Для того, чтобы отменить специальное действие \, его нужно продублировать: `\\`.

СИНТАКСИС

Чтобы не писать каждый раз всё в скобках придумали специальные последовательности:

- `.` - любой символ
- `\s` - любой пробельный символ
- `\S` - любой непробельный символ
- `\d` - любая цифра
- `\D` - любая нецифра
- `\w` - любой буквенный символ
- `\W` - любой небуквенный символ

СИНТАКСИС

А чтобы не повторять много раз, придумали квантификаторы:

- `a?` - символ a должен встречаться 0 или 1 раз
- `a*` - символ a должен встречаться 0 или более раз подряд
- `a+` - символ a должен встречаться 1 или более раз подряд
- `a{5}` - символ a должен встречаться ровно 5 раз подряд
- `a{3,5}` - символ a должен встречаться ровно от 3 до 5 раз подряд

Эти квантификаторы можно применять не только к единичным символам, например: `[a-z]*`

ГРУППИРОВКА

Для того, чтобы несколько символов выступали "как одно целое" применяется группировка (круглые скобки): `(abc)?` - строка abc может встречать 0 или 1 раз.

Внутри группы мы можем также использовать вариации: `(a|b)?` - строка a или b может встречать 0 или 1 раз.

Можно усложнять: `((a|b)c)?` - строка ac или bc может встречать 0 или 1 раз.

При этом сами круглые скобки "захватывают" (capture) те символы, которые в них попадают, что позволяют их затем извлекать при помощи API.

ФЛАГИ

По умолчанию, регулярные выражения "не перескакивают" за символы разделения строк, ищут только первое и учитывают регистр.

Всё это настраивается с помощью флагов, которые пишутся:

- `/regex/flags` - в литеральной форме
- `new Regex('regex', 'flags')` - при создании через функцию-конструктор

КОНЕЦ И НАЧАЛО СТРОКИ

Кроме того, можно использовать спец.символы привязки к началу и концу строки: `/^@[^.]+\.\w$/`

Где:

- `^` сигнализирует о начале строки
- `$` сигнализирует о конце строки

Их можно использовать как вместе, так и по отдельности.

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

В регулярных выражениях очень легко ошибиться: `/@[^.]+\.\w/`

Например наше текущее выражение разрешает пробелы, символы # и другие выражения после `@`.

Мы можем его "ужесточить" до `/\w@\w+\.\w/`, но тогда `support@netology-code.ru` уже не пройдёт.

И так наше выражение будет потихоньку разрастаться: `/\w@[\d\w][-_\w\d]+\.\w/`.

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

У регулярных выражений достаточно большое количество различных возможностей, все их осветить в рамках данной лекции не удастся, однако изучить их стоит.

Сам язык регулярных выражений во многих языках программирования единый, поэтому освоить его стоит в любом случае.

МЕТОДЫ STRING

STR.SEARCH(REGEXP)

Этот метод возвращает позицию первого совпадения с шаблоном регулярного выражения.

Если совпадения нет, то результатом выполнения будет `-1`

Синтаксис:

```
str.search(RegExp);
```

STR.SEARCH(REGEXP)

```
1  function validateEmail(emailStr) {
2    return emailStr.search(/\w@[\d\w][-_\w\d]+\.\w/) !== -1;
3  }
4
5  console.log(validateEmail('support@netology.ru'));
6  console.log(validateEmail('support@netologyru'));
7  console.log(validateEmail('@supportnetology.ru'));
8  console.log(validateEmail('support@netologyru.'));
9  console.log(validateEmail('supportnetology.ru'));
10 console.log(validateEmail('supportnetologyru'));
```

В этом примере производится поиск совпадения с шаблоном. Если поиск удачен, то возвращается число большее или равное нулю, на что и производится проверка в условии.

STR.MATCH(REGEXP)

Возвращает результат совпадения с шаблоном.

Попробуем записать предыдущую проверку через `match()`. Для наглядности пока просто выведем результат выполнения `match()`

```
function validateEmail(emailStr) {
  return emailStr.match(/\w@[\d\w][-_\w\d]+\.\w/);
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));
```

REGEX

Возвращает результат совпадения с шаблоном.

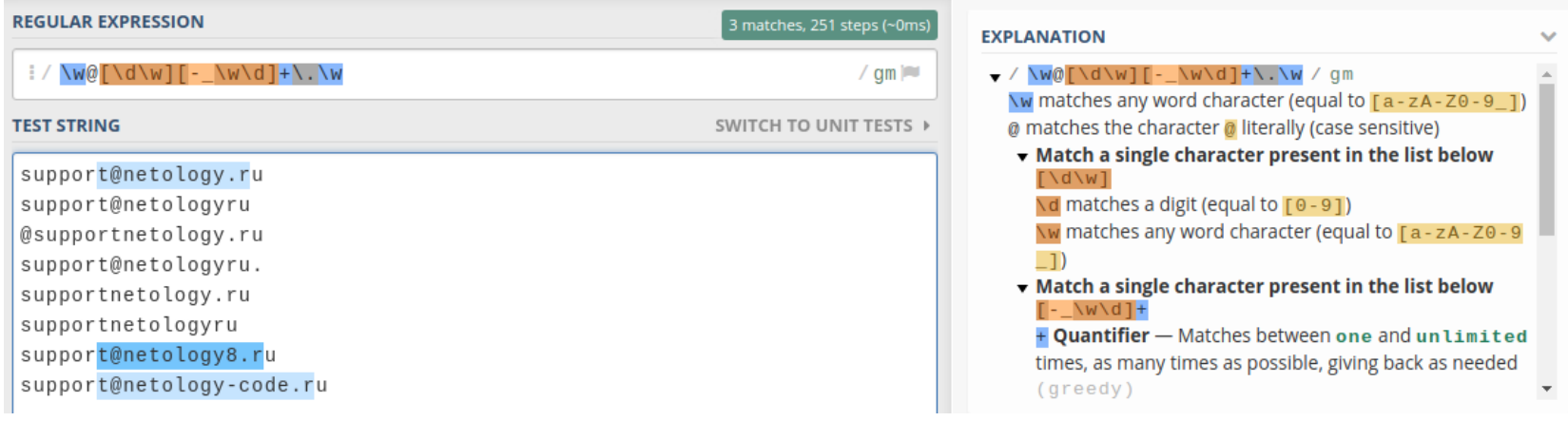
Попробуем записать предыдущую проверку через `match()`. Для наглядности пока просто выведем результат выполнения `match()`

```
function validateEmail(emailStr) {
  return emailStr.match(/\w@[\w\d][-_\w\d]+\.\w/);
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));
```

REGEX101.COM

В работе с регулярными выражениями отлично помогает сайт regex101.com



Кроме того, регулярки - это отличный кандидат на табличные тесты с Jest.

STR.MATCH(REGEXP)

Если `search()` оповещает только о позиции совпадения, то `match()` нам показывает и сам результат сверки с шаблоном.

```
> function validateEmail(emailStr) {
    return emailStr.match(/\w@[\w\d][-_\w\d]+\.\w/);
  }

  console.log(validateEmail('support@netology.ru'));
  console.log(validateEmail('support@netologyru'));
  console.log(validateEmail('@supportnetology.ru'));
  console.log(validateEmail('support@netologyru.'));
  console.log(validateEmail('supportnetology.ru'));
  console.log(validateEmail('supportnetologyru'));
  ▼ ["t@netology.r", index: 6, input: "support@netology.ru", groups: undefined] i
      0: "t@netology.r"
      groups: undefined
      index: 6
      input: "support@netology.ru"
      length: 1
    ▶ __proto__: Array(0)
  null
  null
  null
  null
  null
```

Т.е., если совпадений не найдено, то результатом `match()` будет *null*

STR.MATCH(REGEXP)

Проверка приобретает такой вид:

```
function validateEmail(emailStr) {
  return emailStr.match(/\w@[\w\d][-_\w\d]+\.\w/) !== null;
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));
```

GROUPS

Обратите внимание, если мы поставим круглые скобки, то под индексом `1` у нас появятся значения (не в `groups`):

```
> function validateEmail(emailStr) {
    return emailStr.match(/\w@([\w\d][-_\w\d]+)\.\w/);
  }

  console.log(validateEmail('support@netology.ru'));
  console.log(validateEmail('support@netologyru'));
  console.log(validateEmail('@supportnetology.ru'));
  console.log(validateEmail('support@netologyru.'));
  console.log(validateEmail('supportnetology.ru'));
  console.log(validateEmail('supportnetologyru'));
```

```
▼(2) ["t@netology.r", "netology", index: 6, input: "support@netology.ru", groups: undefined]
    0: "t@netology.r"
    1: "netology"
    groups: undefined
    index: 6
    input: "support@netology.ru"
    length: 2
  ▶ __proto__: Array(0)
null
null
null
null
null
```

STR.SPLIT(REGEXP|STR, LIMIT)

Функция `split()` позволяет разбить строку по какому-либо разделителю.

Как разделитель можно использовать как строку, так и регулярку.

`limit` - необходимое количество полученных элементов массива, по умолчанию - неограничено.

STR.SPLIT(REGEXP|STR, LIMIT)

Разобьем предложение на слова:

```
function separatePhrase(phrase) {
  return phrase.split(/[^(a-яёa-z@\.)]+/i);
}
console.log(separatePhrase('Support@netology.ru \
- адрес технической поддержки Нетологии.'));
```

`^` внутри скобок используется для отрицания

`[]` группа возможных символов

`а-я` указывает на диапазон символов (от "а" до "я")

Флаг `i` указывает на игнорирование регистра.

STR.REPLACE(REGEXP, STR|FUNC)

Чудо, а не функция.

Позволяет произвести определенные операции с определенными участками текста.

STR.REPLACE(REGEXP, STR|FUNC)

Давайте напишем функцию перевода на "кирпичный язык".

Кто помнит, как перевести на "кирпичный"? Можно просто пример.

STR.REPLACE(REGEXP, STR|FUNC)

В круглых скобках указывается группа сиволов.

Для получения выбранной группы используем `$1`

```
function transferToBrick(phrase) {
  return phrase.toUpperCase().replace(/([АЯЭЕОЁУЮЫИ])/, '$1К$1');
}
console.log(transferToBrick('Привет, мир!'));
```

STR.REPLACE(REGEXP, STR|FUNC)

Флаг `g` указывает на то, что поиск будет производиться по всей фразе.

Без этого флага будет изменен только первый символ.

```
function transferToBrick(phrase) {
  return phrase.toUpperCase().replace(/([АЯЭЕОЁУЮЫИ])/g, '$1К$1');
}
console.log(transferToBrick('Привет, мир!'));
```

МЕТОДЫ REGEXP

REGEXP.TEST(STR)

Функция, схожая с `str.search(regexp) !== -1`. Проверяет, есть ли хоть одно совпадение.

```
function validateEmail(emailStr) {
  return /\w@[\w\d][-_\w\d]+\.\w/.test(emailStr);
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));
```

REGEXP.EXEC(STR)

Функция, похожая на `str.match(RegExp)`.

Возвращает совпадение с шаблоном

```
function validateEmail(emailStr) {
  return /\w@[\w\d][-_\w\d]+\.\w/.exec(emailStr) !== null;
}

console.log(validateEmail('support@netology.ru'));
console.log(validateEmail('supp@rt@netologyru'));
console.log(validateEmail('$upport@netologyru'));
console.log(validateEmail('support@netologyru'));
console.log(validateEmail('@supportnetology.ru'));
console.log(validateEmail('support@netologyru.'));
console.log(validateEmail('supportnetology.ru'));
console.log(validateEmail('supportnetologyru'));
```

НОВЫЕ ВОЗМОЖНОСТИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

ВОЗМОЖНОСТИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ В СТАНДАРТЕ ES2018

- именованные группы;
- lookbehind, lookahead;
- флаг `s`;
- паттерны для работы с Unicode;

ИМЕНОВАННЫЕ ГРУППЫ

Появилась возможность давать группам наименования.

```
function findEmail(emailStr) {
  // мы упростили выражение для наглядности
  return /(?<emailGroup>\w+@\w+\.\w+)/.exec(emailStr);
}

const textStr = 'Support@netology.ru \
- адрес технической поддержки Нетологии.';
const emailStr = findEmail(textStr);
console.log(emailStr.groups.emailGroup);

// -> Support@netology.ru
```

LOOKBEHIND, LOOKAHEAD

"Взгляд назад" и "Взгляд вперёд":

- `x(?=y)` - ищет соответствие паттерну x, когда он идёт перед y (положительная опережающая проверка);
- `x(?!y)` - ищет соответствие паттерну x, когда он идёт не перед y (негативная опережающая проверка);
- `(?<=y)x` - ищет соответствие паттерну x, когда он идёт после y (положительная ретроспективная проверка);
- `(?<!y)x` - ищет соответствие паттерну x, когда он идёт не после y (негативная ретроспективная проверка);

LOOKBEHIND, LOOKAHEAD

```
function findEmail(emailStr) {
  return emailStr.match(/\w+(?=@)/g);
}

textStr = 'admin@netology, email: support@netology.ru';
console.log(findEmail(textStr));

// -> ["admin", "support"]
```

ФЛАГ `s` (DOTALL)

Хотя и считается, что символ точки соответствует любому одиночному символу, он не соответствует некоторым символам, например, символу перевода строки `\n`.

```
function matchPhrase(phraseStr) {
  return /Нетология.онлайн-школа/.exec(phraseStr);
}

const textStr = 'Нетология\nонлайн-школа';
console.log(matchPhrase(textStr));

// -> null
```

ФЛАГ `s` (DOTALL)

Флаг `s` позволяет видеть как точку абсолютно любой символ:

```
function matchPhrase(phraseStr) {
  return /Нетология.онлайн-школа/s.exec(phraseStr);
}

const textStr = 'Нетология\nонлайн-школа';
console.log(matchPhrase(textStr));

// -> ["Нетология↵онлайн-школа", index: 0,
// input: "Нетология↵онлайн-школа", groups: undefined]
```

ПАТТЕРНЫ ДЛЯ РАБОТЫ С UNICODE-СИМВОЛАМИ

Были расширены возможности для работы с Unicode:

```
1  console.log(/\p{Emoji}/u.test('😀ΩU'));
2  console.log(/\p{Script=Greek}/u.exec('😀ΩU'));
3
4  // -> true
5  // -> ["Ω", index: 0, input: "😀ΩU", groups: undefined]
```

КОГДА НЕ СТОИТ ИСПОЛЬЗОВАТЬ

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ...

Регулярные выражения - удобный, полезный, высокопроизводительный и простой инструмент.

Однако, важно помнить, что:

- как и любой другой инструмент, он НЕ универсален;
- регулярное выражение сложно читаемо.

Достаточно часто разработчики могут создать "дьявольское" регулярное выражение.

Так же, среди разработчиков бытует мнение, что "регулярки пишутся в одну сторону" - то есть их пишут, но не читают, регулярное выражение проще написать с нуля, чем разобраться в готовом.

Поэтому,

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

1. Есть более подходящие инструменты.

Например, для проверки кода есть большое количество готовых синтаксических анализаторов. Неправильно проверять "правильно ли сформирован JSON" собственной регуляркой.

Например, если требуется только определить наличие единственного символа в строке, с этим справится и функция `str.indexOf()`.

Поэтому,

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

1. Есть более подходящие инструменты.

Например, Вам требуется найти наличие двух подстрок в тексте. Может быть, код для поиска двух подстрок будет эффективнее, чем одна регулярка?

Например, не пытайтесь проверить текст на цензурность с помощью одной регулярки. Эта проблема эффективнее решается несколькими различными инструментами, где регулярным выражениям отведена вспомогательная роль.

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

2. Вы пытаетесь решить одной регуляркой задачу, которую НУЖНО решать несколькими регулярными выражениями

Например, Вам требуется проверить серию и номер документа. Если у вас допускается два документа (например, паспорт и свидетельство о рождении), не стоит пытаться объединить два шаблона для проверки в один. Даже если Вы получите выигрыш в производительности (что маловероятно), Вы существенно потеряете в читаемости кода.

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

3. Регулярное выражение нечитаемо.

Например, с вероятностью 99,9% другой разработчик не станет разбираться в смысле следующей регулярки:

```
(([0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}
|([0-9a-fA-F]{1,4}:){1,7}:|([0-9a-fA-F]{1,4}:)
{1,6}:[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,5}
(:[0-9a-fA-F]{1,4}){1,2}|([0-9a-fA-F]{1,4}:){1,4}
(:[0-9a-fA-F]{1,4}){1,3}|([0-9a-fA-F]{1,4}:){1,3}
(:[0-9a-fA-F]{1,4}){1,4}|([0-9a-fA-F]{1,4}:){1,2}
(:[0-9a-fA-F]{1,4}){1,5}|[0-9a-fA-F]{1,4}:
((:[0-9a-fA-F]{1,4}){1,6})|:((:[0-9a-fA-F]{1,4})
{1,7}|:)|fe80:(:[0-9a-fA-F]{0,4}){0,4}
%[0-9a-zA-Z]{1,}|::(ffff(:0{1,4}){0,1}:){0,1}
((25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])\.){3,3}
(25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])|
([0-9a-fA-F]{1,4}:){1,4}:((25[0-5]|(2[0-4]|1{0,1}
[0-9]){0,1}[0-9])\.){3,3}(25[0-5]|(2[0-4]|1{0,1}
[0-9]){0,1}[0-9]))
```

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ - ЗЛО, ЕСЛИ:

4. Вы не умеете писать регулярные выражения или Ваш код "грязный".

Если регулярное выражение не работает - оно вредит.

Если регулярное выражение слишком требовательное к ресурсам - оно вредит.

Если в мешанине кода всплывает регулярное выражение - оно вредит.

ИТАК, ПОДВЕДЁМ ИТОГИ

На этой лекции были рассмотрены регулярные выражения и их возможности в JS.

ИНТЕРЕСНОЕ ЧТИВО

Регулярные выражения:

- Шпаргалка по регулярным выражениям
- regex101.com - проверить свою регулярку
- MDN - сводка по RegExp
- MDN - Руководство по JavaScript: Регулярные выражения
- Пример, так делать НЕЛЬЗЯ

НЕТОЛОГИЯ

Спасибо за внимание!!! Жду ваших вопросов 🙂

АЛЕКСЕЙ СУДНИЧНИКОВ

@avsudnichnikov