JavaScript Regex: Capturing groups

Summary: in this tutorial, you'll learn about JavaScript regex capturing groups and how to use them to create subgroups for a match.

Introduction to the JavaScript regex capturing groups

Suppose you have the following path:

```
resource/id
```

For example:

```
posts/10
```

In the path, the resource is `posts` and `id` is 10. To match the path, you use the following regular expression:

```
/\w+\/\d+/
```

> In this pattern:

- `\w+` is a word character set with a quantifier (+) that matches one or more word characters.
- `/` matches the forward slash (`/`). The backslash (`\`) escapes the forward slash,
- `\d+` is the combination of the digit character set and a quantfifer (`+`), which matches one or more digits.

The following uses the regular expression `/\w+\/\d+/` pattern to match the string ' `posts/10'` :

```
const path = 'posts/10';
const pattern = /\w+\/\d+/;

const match = path.match(pattern);
console.log(match);
```

Output:

```
[ 'posts/10', index: 0, input: 'posts/10', groups: undefined ]
```

To get the id 10 from the path `posts/10`, you use a capturing group. To create a capturing group for a rule, you place that rule in parentheses like this:

```
(rule)
```

The following example creates a capturing group that captures the `id` value from the path `posts/10`:

```
'\w+/(\d+)'
```

In this pattern, we place the rule `\d+` inside the parentheses `()`. If you run the program with the new pattern, you'll see that it displays one match:

```
const path = 'posts/10';
const pattern = /\w+\/(\d+)/;

const match = path.match(pattern);
console.log(match);
```

Output:

```
[ 'posts/10', '10', index: 0, input: 'posts/10', groups: undefined ]
```

The match() method returns an array of matches. The first element is the whole match (`'posts/10'`) while the second one (`'10'`) is the value of the capturing group.

Note that the `String.match()` method will not return groups if you set the `g` flag e.g., `/\w+\/(\d+)/g` . If you use the `g` flag, you need to use the `String.matchAll()` method instead to get the groups.

Multiple capturing groups

To capture both the resource (`posts`) and id (`10`) of the path (`post/10`), you use multiple capturing groups in the regular expression as follows:

```
/(\w+)\/(\d+)/
```

The regex has two capturing groups one for `\w+` and the other for `\d+` .

The following script shows the entire match and all the subgroups:

```
const path = 'posts/10';
const pattern = /(\w+)\/(\d+)/;

const match = path.match(pattern);
console.log(match);
```

Output:

```
['posts/10', 'posts', '10', index: 0, input: 'posts/10', groups: undefined]
```

To access the first and second subgroups, you use `match[1]` and `match[2]` . Note that the `match[0]` returns the entire match. For example:

```
const path = 'posts/10';
const pattern = /(\w+)\/(\d+)/;
const match = path.match(pattern);
console.log(match[0], match[1], match[2]);
```

Output:

```
posts/10 posts 10
```

Named capturing groups

To access a subgroup in a match, you use an index. However, you may want to access a subgroup by a meaningful name to make it more convenient.

To do that, you use the **named capturing group** to assign a name to a group. The following shows the syntax for assigning a name to a capturing group:

```
(?<name>rule)
```

In this syntax:

- `()` indicates a capturing group.
- `?<name>` specifies the name of the capturing group.
- `rule` is a rule in the pattern.

For example, the following creates the names:

```
/?<resource>\w+)\/(?<id>\d+/
```

In this syntax:

- The `resource` is the name for the first capturing group
- The `id` is the name for the second capturing group.

For example:

```
const path = 'posts/10';
const pattern = /(?<resource>\w+)\/(?<id>\d+)/;

const match = path.match(pattern);
console.log(match);
```

Output:

```
[
  'posts/10',
  'posts',
  '10',
  index: 0,
  input: 'posts/10',
  groups: [Object: null prototype] { resource: 'posts', id: '10' }
]
```

The match has a `groups` property that is an object. The `match.groups` object has properties whose names are the capturing group name and values are the capturing values. For example:

```
const path = 'posts/10';
const pattern = /(?<resource>\w+)\/(?<id>\d+)/;

const match = path.match(pattern);
for (const name in match.groups) {
  console.log(name, match.groups[name]);
}
```

Output:

```
resource posts
id 10
```

More named capturing group example

The following regular expression matches the path `posts/2022/02/18`

```
/\w+\/d{4}\/d{2}\/d{2}/
```

To capture the resource (post), year (2022), month (02), and day (18), you use the named capturing groups like this:

```
/(?<resource>\w+)\/(?<year>\d{4})\/(?<month>\d{2})\/(?<day>\d{2})/
```

This program uses the patterns to match the path and shows all the subgroups:

```
const path = 'posts/2022/02/18';
const pattern =
  /(?<resource>\w+)\/(?<year>\d{4})\/(?<month>\d{2})\/(?<day>\d{2})/;

const match = path.match(pattern);
console.log(match.groups);
```

Output:

```
{resource: 'posts', year: '2022', month: '02', day: '18'}
```

Summary

- Place a rule in parentheses `()` to create a capturing group. A regular expression can have multiple capturing groups.
- Use the `(?<capturingGroupName>rule)` to create a named capturing group for the rule in a pattern.