Alex Wentz
Posted on Jun 24

❤️ 3

Username and Password Validation Using Regex

#regex #tutorial #javascript

When I was first learning web development, one of the tools that consistently eluded my understanding was regex. It's been about a year and a half since I got my first engineering job, and that level of understanding persisted until very recently.

I hadn't really *needed* to learn regex before, but a recent work project (mobile app, React Native) required frontend validation on a user registration page for the username and password. The use case was complicated enough to require learning more than I knew, but easy enough to be attainable. By the end, it felt like a great example to talk through in a tutorial, so here we are!

The Problem

Usernames have a minimum number of characters, require at least one number, and so on. Passwords often do the same, usually adding an assessment of password strength.

For our example here, we want the following to be true:

- Username: between 6 and 16 characters, alphanumeric only
- Valid Password: between 8 and 32 characters, at least one letter and one number
- Strong Password: valid, plus a combination of uppercase and lowercase letters and special characters

The Solution

Part A: the regex we need

Username

While some exposure to regex is useful here, the following explanation is intended to be clear enough for anyone to understand.

Let's start with the username. We need it to contain only letters and numbers, and it must be between 6 and 16 characters long. This will do the trick:

```
/^[0-9A-Za-z]{6,16}$/
```

Let's break this statement up into its parts.

- `/.../` wrap/denote a regex statement
- `^` indicates the start of a string, while `$` indicates the end. Basically, this is ensuring that the entire string follows our rules, rather than only a subset of the string.
- `[...]` indicates a particular set of valid characters, otherwise called a **character class**; `0-9` allows numbers, `A-Z` allows uppercase letters, `a-z` allows lowercase. There are other indicators, and you can find a complete list in regex documentation.
- `{6,16}` indicates the allowed number of characters. If you just used `{6}`, you're testing for a length of exactly 6, while `{6,}` tests for minimum length.

Password

When we move on to test for a valid password, the requirements change. Instead of being *limited* to letters and numbers only, now we need *at least one each* of different kinds of characters. This requires us to implement a specific check for each of them, which looks like:

```
/^(?=.*?[0-9])(?=.*?[A-Za-z]).{8,32}$/
```

Let's break down the new symbols here.

- `(...)` is a **capture group**. You can use them for capturing particular characters in specific orders.
- `?=` is a **positive lookahead**. The search moves rightward through the string from the location in your regex you make this assertion in.
- `.` signifies any character is possible, while `*` means 'zero or more' of them.
- The extra question mark in `?=.*?` makes the search **lazy**, which essentially means 'stop looking after the first time this requirement is met'.

Translated into plain English, the first part of our statement `^(?=.*?[0-9])` means 'from the start of the string, find a number that is preceded by zero or more of any character'.

Adding `(?=.*?[A-Za-z])` means do the same for any letter, or 'from the start of the string, find a letter that is preceded by zero or more of any character'. This allows us to confirm the presence of a specified kind of character within the total set of what is allowed without regard to where it occurs in the string.

The last part of our statement `.{8,32}$` builds on our understanding of `.` usage. We don't want to limit what kinds of characters the actual password is allowed to be. In contrast, if limiting to letters and numbers only, you'd use `[0-9A-Za-z]{8,32}$`.

In addition to the valid password requirements, a strong password requires at least one uppercase letter, at least one lowercase letter, and at least one special character. We can just build on the same regex statement:

```
// valid, but not strong
/^(?=.*?[0-9])(?=.*?[A-Za-z]).{8,32}$/
// strong
/^(?=.*?[0-9])(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[^0-9A-Za-z]).{8,32}$/
```

There's one more new meaning to unpack here. Translating to English first will make the meaning clear:

- find one number, `(?=.*?[0-9])`
- one uppercase letter, `(?=.*?[A-Z])`
- one lowercase letter, `(?=.*?[a-z])`
- and one character that is NOT alphanumeric `(?=.*?[^0-9A-Za-z])`
- in a string with any characters between 8 and 32 characters long `.{8,32}`

When used at the start of a character class `^` means 'NOT', or everything excluding. A basic understanding of sets helps here. A simple example is `[^B]`, which means 'any character except B'.

Part B: testing for valid inputs

The JavaScript RegExp class has a very handy method called `test()` that can be used to test strings against a specified regex statement. For instance,

```
const validUsername = /^[0-9A-Za-z]{6,16}$/;

console.log(validUsername.test('username')) // valid
// prints true
console.log(validUsername.test('test1')) // too short
// prints false
console.log(validUsername.test('@testing1')) // invalid character
// prints false
```