**What is an Autovalidating form?**

This lesson gives a brief overview of auto validating forms and a roadmap of the rest of the chapter.

**We'll cover the following**

* + [They’re everywhere!](https://www.educative.io/courses/intermediate-javascript/3jwwj7w9P2n#theyre-everywhere)
  + [Understanding validation complexity](https://www.educative.io/courses/intermediate-javascript/3jwwj7w9P2n#understanding-validation-complexity)
  + [Our target](https://www.educative.io/courses/intermediate-javascript/3jwwj7w9P2n#our-target)

**They’re everywhere!**[#](https://www.educative.io/courses/intermediate-javascript/3jwwj7w9P2n#theyre-everywhere)

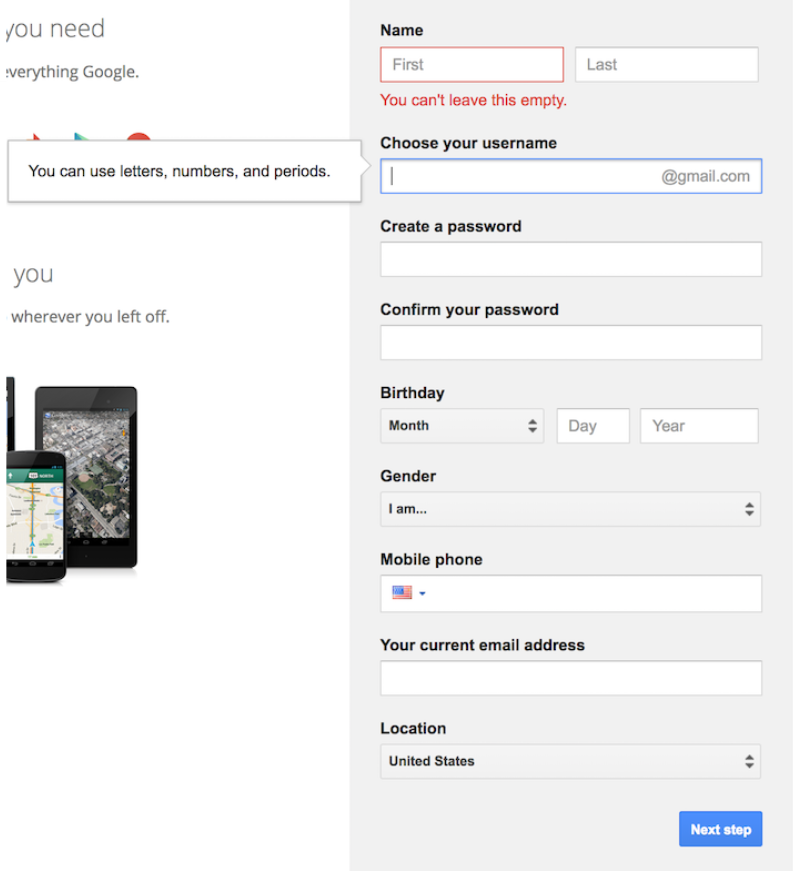
In this chapter, we’ll be building an auto validating form. You see this in a lot of places across the web, so you might hardly notice it anymore. However, poorly done forms can easily give users a negative impression of your product. When they’re already filling out a form, it means they’re engaged and giving some information. This is not an area of your app that you want them to form negative impressions.

**Understanding validation complexity**[#](https://www.educative.io/courses/intermediate-javascript/3jwwj7w9P2n#understanding-validation-complexity)

Forms that do more than just give you some input fields and let you submit freeform data has a surprising amount of complexity. For example, you rarely get to think about what constitutes a valid email address until you implement one of these. Do you just want to accept any string of characters, followed by @ followed by another string with a dot? Or do you want completeness, which could be loosely achieved with a regex like the [this](http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html)?

**Our target**[#](https://www.educative.io/courses/intermediate-javascript/3jwwj7w9P2n#our-target)

At one point or another in your career as a frontend engineer, customized inputs will almost certainly be something you’ll have to think about. We’ll be building out Gmail’s signup form together, and I believe it’ll cover enough cases that you’ll be able to extend it to your ​forms. ​



When you’re ready, let’s get started.

# Rules for User Input Categories

In this lesson, let's choose which rules will be used to validate various user input types such as birthday, name and username.

###### We'll cover the following

* + - * [What is the goal of adding auto validation to forms?](https://www.educative.io/courses/intermediate-javascript/m2MKX5jVOzE#what-is-the-goal-of-adding-auto-validation-to-forms)
      * [What to avoid](https://www.educative.io/courses/intermediate-javascript/m2MKX5jVOzE#what-to-avoid)
      * [Rules](https://www.educative.io/courses/intermediate-javascript/m2MKX5jVOzE#rules)

#### What is the goal of adding auto validation to forms? [**#**](https://www.educative.io/courses/intermediate-javascript/m2MKX5jVOzE#what-is-the-goal-of-adding-auto-validation-to-forms)

When forms validate inputs for users before they hit submit, it’s to prevent frustration when they later hit submit with improper inputs and are rejected. The second worst user experience you can provide is to wait until the user has filled out all the information and hits submit, only to be rejected for an unclear reason and forced to re-enter information- I’ll get to the worst in a second.

Therefore, let’s only validate when the user appears to be finished with that input.

There are some exceptions to this rule, and that’s where the requirements are ambiguous. Everyone knows what emails are supposed to look like, but different websites can have different requirements for passwords depending on their audience, risk of security breaches (it’s not the end of the world if someone gains access to your yoga membership account, ​but it could be disastrous if they do gain access to your bank account). It’s reasonable to show the requirements of inputs with varying rules before the user enters them – or better yet, as the user enters them:

So we’ll validate after the user is complete for fields with universal rules. Otherwise, we want to guide them. We also want to choose the best way to do it. For example, for birthdays, giving individual fields for month, date, and year removes ambiguities for region-specific formats. You want to create as little chance as possible for errors when they hit submit.

#### What to avoid [**#**](https://www.educative.io/courses/intermediate-javascript/m2MKX5jVOzE#what-to-avoid)

Adding client-side validation can also accidentally have the opposite effect of preventing good input or adding undue confusion. The validation shouldn’t be too stringent. For example, some forms will validate as the user is typing, showing an error before they’re even finished working on the field.

The **worst** user experience, however, is when valid input can’t be submitted. I’ve encountered sites where an email field wouldn’t accept emails that didn’t end with .com.

So now let’s establish a set of rules for the form we’re building (these won’t necessarily be Gmail’s):

#### Rules [**#**](https://www.educative.io/courses/intermediate-javascript/m2MKX5jVOzE#rules)

* Name: A non-empty​ string of alpha characters
  + Validate after the field is blurred (cursor no longer in the field)
* Username: A string of characters which are either numbers, letters, periods, or underscores.
  + Validate after the field is blurred
* Password: A string of alphanumeric characters greater than length 6
  + Validate as typed
  + Rated by the following:
    - Weak: Less than 6 length
    - Fair: Greater than 6 length
    - Good: Greater than 6 length and has a mix of letters and numbers
* Birthday: Numbers
  + Validate after the field is blurred
  + Extra guidance provided by:
    - inability to continue typing past character limit (2 for day, 4 for year)
    - inability to type non-numeric characters (typing ‘a’ does nothing)
* Phone number: Numbers with hyphens and parentheses
  + Parentheses can only surround the initial set of numbers
  + Hyphens must appear between numbers
* Current email address: String of ASCII characters with @ and . existing between any characters (can’t be at the beginning or end)​

Let’s proceed forward with actually setting up our own autovalidating form.

**Types of tests**[#](https://www.educative.io/courses/intermediate-javascript/YQW33n6DD0K#types-of-tests)

There are three broad categories of tests that can be automated in web apps.

1. **Unit tests**

* These test individual functions. For example, if you have a function that’s supposed to return true given a certain input and false otherwise, that’s easily testable with unit tests.
* It only covers JavaScript code

1. **Integration tests**

* These test functions that are working together. For example, when one function calls another and uses that result in a different function call.
* It only covers JavaScript code. A lot of business logic can be tested here.

1. **Acceptance Tests**

* These test functionality in the web app as a whole through the client interface. For example, when the user clicks on a button, a popup appears.
* A lot of the user experience guarantees are tested here. JavaScript should not be called directly; these tests are concerned with the outcome of user-taken actions.

People have different interpretations of how tests should be split, an​d there’s no standard approach here. It doesn’t matter, though. Labels on tests aren’t as important as simply having tests that cover the critical parts of your app. You also shouldn’t write tests for the sake of writing tests. Having 100% code coverage (mean​ing you have tests that hit every line of code) is a mostly meaningless goal. I’d much rather have a concentrated set of tests on the parts that are prone to bugs than an equal distribution of tests that cover 100% of my code.

So let’s try some test-driven development! This just means we write the tests before we have the code to make them pass.

In a real application, you’ll want to use a testing framework. These make it much more pleasant to write and run your tests. There are many to choose from for each category of tests, but the differences are mostly syntactical. Writing the actual tests is easy in any framework or library. **The hard part is choosing what to test and how to test it.**

# Unit Tests

In this lesson, we'll learn what Unit Tests are and how to build and run them.

Unit tests should be straightforward and answer, “given these inputs, what are the expected outputs.”

function runner({inputs, expectedOutputs, func}) {

  assert(inputs.length === expectedOutputs.length);

  for (let i = 0; i < inputs.length; i++) {

    assert(func(inputs[i]) === expectedOutputs[i]);

  }

}

function firstNameTest() {

  const invalidInputs = ["@", "", "blah$", "123"];

  const validInputs = ["asdf", "Alfred", "ALFRED"];

  runner({

    inputs: validInputs,

    expectedOutputs: validInputs.map(\_ => true),

    func: isValidName

  });

  runner({

    inputs: invalidInputs,

    expectedOutputs: invalidInputs.map(\_ => false),

    func: isValidName

  });

}

function emailTest() {

  const invalidEmails = ["@asdf.com", "what@what", "", ".."];

  const validEmails = ["asdf@asdf.com", "what@what.au", "a@a.c"];

  runner({

    inputs: validEmails,

    expectedOutputs: validEmails.map(\_ => true),

    func: isValidEmail

  });

  runner({

    inputs: invalidEmails,

    expectedOutputs: invalidEmails.map(\_ => false),

    func: isValidEmail

  });

}

assert is a Node thing, which, as its name indicates, checks that the argument evaluates to true and throws an error if it’s not.

We’ve structured our tests such that there’s a runner function, and every test feeds a set of inputs, expected outputs, and the function used. The line, x.map(\_ => true), is just creating an array of equal size with the values true for every element. Remember, map applies a function to every element, \_ is the variable we use when we want to convey that it’s unused, and => is just shorthand for function() { return true }.

Since assert isn’t available to us, we can display test results on the client instead. We can create a new HTML page, import the source code JavaScript, and the test code, and instead of assert, we’ll display the results on the page. (This is similar to what a testing framework called [QUnit](https://qunitjs.com/" \t "_blank) does.)

# Output

Test Results

isValidName(asdf) === true: Fail  
isValidName(Alfred) === true: Fail  
isValidName(ALFRED) === true: Fail  
  
isValidName(@) === false: Fail  
isValidName() === false: Fail  
isValidName(blah$) === false: Fail  
isValidName(123) === false: Fail  
  
isValidEmail(asdf@asdf.com) === true: Fail  
isValidEmail(what@what.au) === true: Fail  
isValidEmail(a@a.c) === true: Fail  
  
isValidEmail(@asdf.com) === false: Fail  
isValidEmail(what@what) === false: Fail  
isValidEmail() === false: Fail  
isValidEmail(..) === false: Fail

# html

<!DOCTYPE html>

<html>

<head>

</head>

<body>

  <h1>Test Results</h1>

  <div class="results"></div>

</body>

</html>

# Js

function isValidName(name) {

  // TODO

}

function isValidEmail(name) {

  // TODO

}

function validate(event) {

  const inputElement = event.target;

  inputElement.classList.add('signup\_\_field\_\_input--error');

  const errorMessageElement = event.target.parentElement.getElementsByClassName('signup\_\_field\_\_error')[0];

  errorMessageElement.innerHTML = 'Sample Error';

}

const inputs = document.getElementsByClassName('signup\_\_field\_\_input');

for (const input of inputs) {

  input.onblur = validate;

}

// Unit Tests

function runTests() {

  firstNameTest();

  emailTest();

}

runTests();

function runner({inputs, expectedOutputs, func}) {

  let results = "";

  for (let i = 0; i < inputs.length; i++) {

    const passFailString = func(inputs[i]) === expectedOutputs[i] ?

      'Pass' :

      '<span style="color: red">Fail</span>';

    const result = `${func.name}(${inputs[i]}) === ${expectedOutputs[i]}: ${passFailString}`;

    results += (result + "<br>");

  }

  const resultsElement = document.getElementsByClassName('results')[0];

  resultsElement.innerHTML += (results + "<br>");

}

function firstNameTest() {

  const invalidInputs = ["@", "", "blah$", "123"];

  const validInputs = ["asdf", "Alfred", "ALFRED"];

  runner({

    inputs: validInputs,

    expectedOutputs: validInputs.map(\_ => true),

    func: isValidName

  });

  runner({

    inputs: invalidInputs,

    expectedOutputs: invalidInputs.map(\_ => false),

    func: isValidName

  });

}

function emailTest() {

  const invalidEmails = ["@asdf.com", "what@what", "", ".."];

  const validEmails = ["asdf@asdf.com", "what@what.au", "a@a.c"];

  runner({

    inputs: validEmails,

    expectedOutputs: validEmails.map(\_ => true),

    func: isValidEmail

  });

  runner({

    inputs: invalidEmails,

    expectedOutputs: invalidEmails.map(\_ => false),

    func: isValidEmail

  });

}

**Integration Tests**

In this lesson, we'll learn what integration tests are and see one in practice.

**We'll cover the following**

* + [What are integration tests?](https://www.educative.io/courses/intermediate-javascript/qV2YByK3Ror#what-are-integration-tests)
  + [Example](https://www.educative.io/courses/intermediate-javascript/qV2YByK3Ror#example)

**What are integration tests?**[#](https://www.educative.io/courses/intermediate-javascript/qV2YByK3Ror#what-are-integration-tests)

We’ve covered what unit tests look like, but what about integration tests?

They’re extremely similar. You write integration tests to test that functions work well together. It’s possible to have functions that pass comprehensive sets of unit tests but be part of failing integration tests. They usually involve multiple systems interacting together.

**Example**[#](https://www.educative.io/courses/intermediate-javascript/qV2YByK3Ror#example)

Consider our form and how it might be sent to a server. We can have a function called serialize, which gives a representation that can be stored as a string and a deserialize function, which reads data in that form and populates our field with that data.

serialize() {

  // Get form fields

  ...

  return `form\_${firstName}\_${lastName}...`;

}

deserialize(serializedForm) {

  // Extract form fields and populate form

  ...

}

For example, you save a form as form\_walt\_disney, and when they come to a page with a form, we can help them fill out the first and last name with “Walt” and “Disney​.”

We can run all sorts of tests to make sure that all these methods work. Also, let’s say we have a function that sends HTTP requests:

function httpRequest({url, data}) {

  return this.request(url + `?data=data`);

}

And the many tests for httpRequest all pass too. However, URLs have a character limit. So if someone exclusively wrote unit tests and shipped the product, one day someone will come along and enter a first name that is longer than the character limit, serialize it, and when they try to retrieve it again, they’ll fail.

You need an integration test to catch that:

function TestBugIntegration() {

  let longName = '';

  for (let i = 0; i < 3000; i++) {

    longName += 'blah';

  }

  const firstNameInput = document.getElementsByClassName('form\_\_input--firstName')[0];

  firstNameInput.value = longName;

  httpRequest({

    url: '/saveForm',

    data: serialize()

  });

  const serializedForm = httpRequest({

    url: '/getForm',

  });

  const form = deserialize(serializedForm);

  // This will fail!

  assert(form.firstName === longName);

}

# This test creates a really long name, sets the input value as such, serializes it, sends the request to an endpoint that saves it, and retrieves it from another endpoint. At no point do we question the integrity of serialize, deserialize, or httpRequest, since we assume there are​ ample unit tests for them. It’s the interaction of the functions which surface the bug.

# Acceptance Tests

In this lesson, let's discuss what acceptance tests are and see one in practice

###### We'll cover the following

* + - [Software Testing](https://www.educative.io/courses/intermediate-javascript/3jYKmrVAPGQ#software-testing)

Acceptance tests look a lot more different. Instead of having a set of inputs map to a set of expected outputs, acceptance tests typically look at the state of the world after a set of actions have been taken.

For example, to test the form’s password guidance modal, w​e can do something like:

function TestPasswordGuide() {

  const passwordInput = document.getElementsByClassName('form\_\_input--password')[0];

  passwordInput.focus();

  const passwordGuideMessage = document.getElementsByClassName('password\_\_guide\_\_message')[0];

  const badPassword = 'foo';

  const goodPassword = badPassword + 'bar123';

  for (const character of badPassword) {

    const keypress = new Event('keydown');

    keypress.key = character;

    document.dispatchEvent(keypress);

  }

  assert(passwordGuideMessage.innerHTML === 'Bad');

  for (const character of badPassword) {

    const keypress = new Event('keydown');

    keypress.key = character;

    document.dispatchEvent(keypress);

  }

  assert(passwordGuideMessage.innerHTML === 'Good');

}

This test focuses on the password input (simulate the act of a user clicking on that input), sends keypress events (simulate a user typing), and checks what the password guide message is set to.

### Software Testing [**#**](https://www.educative.io/courses/intermediate-javascript/3jYKmrVAPGQ#software-testing)

The more mature your software/application is, and the more people come to rely on it, the more important tests become. Frontend testing typically gets less code coverage, and one of the reasons for that is because it’s often hard to write tests for web apps. If your app has an animation, do you include a delay for it to finish? How do you test that a message shows up if you hover over something?

It can be tempting to handwave that a feature works because you tested it on your machine and saw​ that it works, or assume your change didn’t cause a bug because you don’t see one. However, if you include good tests that evolve with your source code, they’ll likely help catch regressions and force you to think rigorously about edge cases.

Now, let’s take a look at the unique errors for each field in our form.

**Unique Errors for Each Field**

Back to our form! In this lesson, we'll learn how to distinguish each field instead of showing the same error for each one.

**We'll cover the following**

* + [Selecting the right validator](https://www.educative.io/courses/intermediate-javascript/gx2yBQ3NBzk#selecting-the-right-validator)

Returning to our component, we last left it in a state that was ready to accept validation rules. If you’ll remember, we wrote some basic code to show an error on each input field upon blur, as if validation failed.

**Selecting the right validator**[#](https://www.educative.io/courses/intermediate-javascript/gx2yBQ3NBzk#selecting-the-right-validator)

To correctly validate each field, of course, we have to differentiate which one is being blurred and apply the rules for that specific field.

Take a moment to inspect the code above. Without changing anything, how would you know which validation rule to apply in the validate function?

To answer this question, we should consider what’s important to us.

As much as possible, you should keep the **view layer** separate from the **logic layer**, which is where you do computations (people also call this the business or controller layer). This is for **maintainability** of your code as it evolves. If you’re building something that lasts, you should expect adding and removing features, doing redesigns, and other sorts of changes. If every time you make changes to your HTML, you also have to go and make changes in your JavaScript code that encodes the business logic, it’s extra work and is an unnecessary dependency. And dependencies expose more points for bugs to occur.

Given that, would you change your answer to​ the quiz? The order is subject to revision – maybe you’ll want to collect additional data in the very near future or remove one due to some legal implications. The placeholder’s purpose is entirely presentational and can easily change. The accessibility label should be human-readable and isn’t suitable either for being the source of truth for this input. The class name is the most reasonable out of the choices since it doesn’t change often, and we have to rely on it already in JavaScript to select elements. It’s not the best, though. What if we want to wrap inputs around another element later, and we change the class name to reflect the new hierarchy? Instead, we can give elements of our own ​attributes for this purpose. These tags are just there for the logic layer consumption, so no matter what changes occur in the HTML (view layer), they stay the same.

So let’s add a data-field to each one. data- is just a prefix that the HTML specs decided is valid for adding custom properties. So instead of like <div field='name'>, we use data-field=name.

**Introduction to Regular Expressions**

This lesson explains and introduces Regular Expressions.

**We'll cover the following**

* + [Regular Expressions](https://www.educative.io/courses/intermediate-javascript/RMkLmkkLjW0#regular-expressions)
    - [Readability](https://www.educative.io/courses/intermediate-javascript/RMkLmkkLjW0#readability)

Time to apply specific rules so that each field only errors if the user input breaks the rules for that particular field.

**Regular Expressions**[#](https://www.educative.io/courses/intermediate-javascript/RMkLmkkLjW0#regular-expressions)

Our rules are for strings, and the *only* robust method of string validation is through **regular expressions**. If you haven’t heard of it, a regular expression (shortened to regex) is a powerful tool in many programming languages that let us define patterns to match strings. They can look a bit unfriendly at first, but I’m certain you’ll find them extremely useful and worth familiarizing yourself with.

The basic concept is you have two delimiters that mark the beginning and end of your regular expression, /. In the middle, you can use groups and symbols to specify rules. Let’s just learn through example.

There’s one thing to keep in mind as we’re going through these examples: while regex is very powerful, we want to use everything at our disposal to make things **human readable**.

**Readability**[#](https://www.educative.io/courses/intermediate-javascript/RMkLmkkLjW0#readability)

Tasks in JavaScript are often not expensive enough for frontend developers to concern themselves with performance metrics, or at least not as often as backend developers. We’re writing code for one browser and one user interacting with that browser. Of course, this isn’t true as a blanket statement across all frontend tasks and doesn’t mean you shouldn’t care to learn about that aspect of programming, but it does mean that when you’re thinking about tradeoffs, readability should be weighted heavily. Your inputs are probably not big enough to make your hand-written binary search make a noticeable performance difference compared to for-looping search, but it will be less readable. Pure regex can be more concise, but not as readable as other methods.

We’ll learn how to put these Regular Expressions into practice for user input validation in the next lesson.

**Validation of User Input with Regex**

Let's apply the rules to individual fields now so they can validate appropriately

**We'll cover the following**

* + - [Name](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#name)
    - [Username](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#username)
    - [Password](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#password)
    - [Birthday](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#birthday)
    - [Phone Number](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#phone-number)

**Name**[#](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#name)

*Original rule: Non-empty string of alpha characters*

Without regex, the non-empty part can be enforced by .length and the alpha part can be enforced by checking if the character is within the list ['a', 'b', ..., 'z', 'A', 'B', ..., 'Z']. With regex, the alpha part is matched by /[a-zA-Z]/ (don’t forget capital letters!), and the nonempty part through the symbol + (which means one or more). So that gives us:

function isValidName(name) {

  const nameRegex = /^[a-zA-Z]+$/;

  return name.test(nameRegex);

}

​The ^ means “start” and $ means “end.” Since test checks that our regex has a match in any part of the string and we want to validate on the entirety of the string, we need to add those. Otherwise, inputs like @#$@#$#@ username would pass.

**Username**[#](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#username)

*Original rule: Non-empty string of number, letter, period, or underscore characters*.

This is pretty much the same thing, except we include numbers, 0-9, the period, ., and the underscore, \_.

function isValidUsername(username) {

  const usernameRegex = /^[a-zA-Z0-9.\_]+$/

  return username.test(usernameRegex);

}

### Password [**#**](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#password)

Original rule: A string of alphanumeric characters with the following categories.

* Weak: Less than six-length
* Fair: Greater than six-length
* Good: Greater than six-length and has a mix of letters and numbers

This one’s a little different in that the return is not a boolean true/false, and we have a strict number requirement.

Notice we have the following property: **Inputs that satisfy a higher grade category also satisfy the ones below it**.

So a good password is also fair and weak, and a fair password is also weak, technically. We can create 3 different rules and just apply them from the highest grade to lowest, and the first one matched is the category.

As for the strict number requirement, regex has a symbol for that! {x} specifies the exact number of occurrences, and {x,} specifies at least that number.

The hard part is the “mix of letters and numbers.” If we have /[a-zA-Z0-9]+/, this enforces that it’s a non-empty string with either numbers or letters. The way we do this in pure regex is: (?=.\*[a-zA-Z])(?=.\*[0-9])[a-zA-Z0-9]+

Whoa, right? Briefly, the ?= is a **lookahead** symbol, which checks characters ahead of the ​current match, the . means “anything,” and the \* means “any number of.” So the (?=.\*[a-zA-Z]) means, “for everything in this input, check that there’s a letter.” The group following it does the same thing with numbers, and the last section is the actual check for a non-empty string of **either** numbers or letters.

But that’s **extremely** messy, and I bet if you were a developer who stumbled upon that piece of code your colleague wrote, you’d have trouble parsing it and be even more cautious making changes. Instead, let’s just run multiple, separate validations! It has to pass /[a-zA-Z]+/, /[0-9]+/, and /^[a-zA-Z0-9]+$. The first two check that there are one or more letters and numbers, and the third one checks that the input consists only of numbers or letters

const PASSWORD\_CATEGORIES = {

  GOOD: 'password\_good',

  FAIR: 'password\_fair',

  WEAK: 'password\_weak',

}

function getPasswordCategory(password) {

  const hasLettersRegex = /[a-zA-Z]+/

  const hasNumbersRegex = /[0-9]+/

  const hasOnlyLettersAndNumbersRegex = /^[a-zA-Z0-9]{6,}$/

  function isGoodPassword() {

    return hasLettersRegex.test(password) &&

      hasNumbersRegex.test(password) &&

      hasOnlyLettersAndNumbersRegex.test(password);

  }

  function isFairPassword() {

    return hasOnlyLettersAndNumbersRegex.test(password);

  }

  if (isGoodPassword()) {

    return PASSWORD\_CATEGORIES.GOOD;

  }

  if (isFairPassword()) {

    return PASSWORD\_CATEGORIES.FAIR;

  }

  return PASSWORD\_CATEGORIES.WEAK;

}

Defining PASSWORD\_CATEGORIES allows unambiguous communication to this password validating interface. Since it’s not true or false, we use arbitrary strings, so there’s no mixup of what’s being returned.

### Birthday [**#**](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#birthday)

Original rule: Numeral characters with enforced character limits of 2 for day and 4 for year.

The month will be a dropdown menu, so we don’t need to worry about that.

I think you can do this one yourself!

### Phone Number [**#**](https://www.educative.io/courses/intermediate-javascript/JYlM1noxJ9o#phone-number)

Original rule: Numbers with hyphens and parentheses, where parentheses can only surround initial set of numbers and hyphens must be between numbers.

This one is a little more interesting! So far, we’ve only had a singular group to work with. For this requirement, we need to split things up a little.

In more precise English, the requirement is:

* The input can either begin with a number or an opening parentheses
* If it begins with an opening parenthesis, a closing parenthesis must appear before a hyphen but after one or more numbers
* The input can also include hyphens in between numbers

That’s going to be a very​ messy regex. To simplify​, let’s split our validation logic for phone numbers into one branch for inputs that have formatting and one that doesn’t.

FORMATTING\_CHARACTERS = ['(', ')', '-'];

for (const formattingCharacter of FORMATTING\_CHARACTERS) {

  if phoneNumber.includes(formattingCharacter) {

    // Do the validation for formatted numbers

  }

}

// Do the validation for non-formatted numbers

# The validation for non-formatted numbers is trivial. For formats, we don’t want to be too strict and not allow users in a country with unique rules.

function isValidFormattedNumber(phoneNumber) {

  const regex = /^[0-9(]{1}[0-9)-]+[0-9]$/;

  const hasOpeningParentheses = phoneNumber.includes('('));

  const hasClosingParentheses = phoneNumber.includes(')'));

  if (hasOpeningParentheses && !hasClosingParentheses) {

    return false;

  }

  return regex.test(phoneNumber);

}

# Conditionals are more easily tested outside regexes, and this is much cleaner than whatever ​the alternative would’ve been.

# Displaying Actual Errors

The first step to getting error messages to show up when validation fails

###### We'll cover the following

* + [Changelist](https://www.educative.io/courses/intermediate-javascript/7n9N5JMAjgQ#changelist)
  + [Changelist](https://www.educative.io/courses/intermediate-javascript/7n9N5JMAjgQ#changelist-2)

We now have a basic form, some JavaScript to execute validation on each input at the correct time, a way to distinguish which field is which, and a set of validation rules to apply to each one. Let’s put the pieces together now!

## Changelist [#](https://www.educative.io/courses/intermediate-javascript/7n9N5JMAjgQ#changelist)

* Create a unique error class
  + This lets us do things only with expected errors that arise from validation by checking instanceof.
* Introduce a try/catch for validation.
  + The catch clause is where the error view is rendered.
  + We could have used enums and passed back something like {status: 'error', message: 'blah'} for errors and {status: 'success'} to signal passing validation, but if the set of return possibilities are many different ways of specifying a type of error and one way of specifying success, we should make use of error handling which covers this exact case. We want to avoid defining a new ad-hoc object, its schema (what fields it contains) is just another thing to keep track of.
* Each input maps to the validation rule through a mapping.
  + Cleaner than a switch statement.
* validate functions that match the previous lesson, except they throw an error when validation fails instead​ of returning false.

Hopefully, this all makes sense. I skipped password and confirmPassword for now because that uses some special handling. Try playing around with this form and check for yourself.

When you implement a new rule, how do you check that it works? Testing it manually doesn’t give us confidence, so let’s adjust our test and see if it passes.

The source code JavaScript is in a hidden file, but note that this test runner HTML would import js files from the source before the test​ so that the test can reference functions in the source.

## Changelist [#](https://www.educative.io/courses/intermediate-javascript/7n9N5JMAjgQ#changelist-2)

* Instead of passing in a function that returns a boolean, properly pass in the validateX functions.
* Instead of calling the functions in the runner and interpreting them as boolean results, we use the try branch for success and catch for failure.
* Messages are wrapped in a span that colors the text red if the ​test failed.
* A test failure means the passing of validation does not match what actually happened. This occurs when the try block runs completely, but failure was expected, or the catch block runs, but success was expected.