# Client-side form validation

## What does it matter?

*You don't want to validate all your form information on the server side. If there is an error, this will take longer to correct and inform the client that something is wrong and needs to be updated. It is best practice to do what you can on the client side and then try to do a bit more of a robust check on the server side just in case someone could get past your client-side validation.*

## Form Validation:

*It is easier to use than JS validation, and you can set the parameters in line.*

*I assume it is slightly quicker since it doesn't need to run a script.*

## Main reasons for validation:

- *We want to get the right data in the correct format. Our applications will only work correctly if our users' data is stored in the correct format, incorrect, or omitted altogether.*
- *We want to protect our users' data. Forcing our users to enter secure passwords makes it easier to protect their account information.*
- *We want to protect ourselves. There are many ways that malicious users can misuse unprotected forms to damage the application. See Website security.*

### There are two different types of client-side validation that you'll encounter on the web:

- *Built-in form validation uses HTML form validation features, which we've discussed in many places throughout this module. This validation generally doesn't require much JavaScript. Built-in form validation performs better than JavaScript but is less customizable than JavaScript validation.*
- *JavaScript validation is coded using JavaScript. This validation is entirely customizable, but you must create it all (or use a library).*

### Validation attributes (common ones):

- *Required: Specifies whether a form field needs to be filled in before the form can be submitted.*
- *Minlength and maxlength: Specifies textual data's minimum and maximum length (strings).*
- *Min and max: Specifies numerical input types' minimum and maximum values.*
- *Type: Specifies whether the data needs a number, an email address, or a specific preset type.*
- *Pattern: Specifies a regular expression that defines a pattern the entered data needs to follow.*

# Fetch API:

*The Fetch API provides a JavaScript interface for accessing and manipulating parts of the protocol, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.*

*A basic fetch request is really simple to set up. Have a look at the following code:*

```
fetch('http://example.com/movies.json')
  .then((response) => response.json())
  .then((data) => console.log(data));
```

## Uploading JSON data

*Use fetch() to POST JSON-encoded data.*

```
const data = { username: 'example' };

fetch('https://example.com/profile', {
  method: 'POST', // or 'PUT'
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data),
})
  .then((response) => response.json())
  .then((data) => {
    console.log('Success:', data);
  })
  .catch((error) => {
    console.error('Error:', error);
  });
```

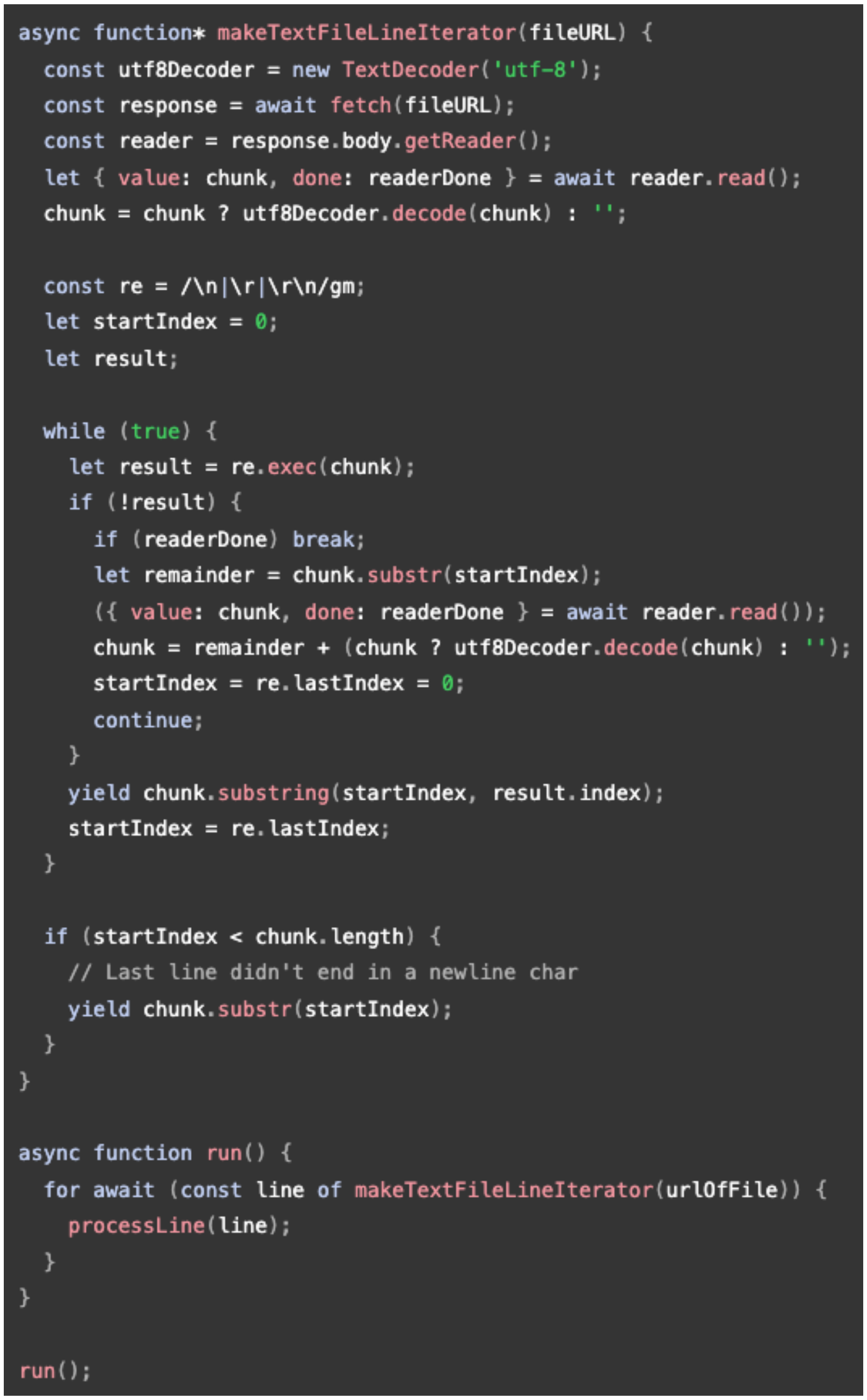
## Uploading a file

*Files can be uploaded using an HTML <input type="file" /> input element, FormData() and fetch().*

## Uploading multiple files

*Files can be uploaded using an HTML <input type="file" multiple /> input element, FormData() and fetch().*

## Processing a text file line by line



```
async function* makeTextFileLineIterator(fileURL) {
  const utf8Decoder = new TextDecoder('utf-8');
  const response = await fetch(fileURL);
  const reader = response.body.getReader();
  let { value: chunk, done: readerDone } = await reader.read();
  chunk = chunk ? utf8Decoder.decode(chunk) : '';

  const re = /\n|\r|\r\n/gm;
  let startIndex = 0;
  let result;

  while (true) {
    let result = re.exec(chunk);
    if (!result) {
      if (readerDone) break;
      let remainder = chunk.substr(startIndex);
      ({ value: chunk, done: readerDone } = await reader.read());
      chunk = remainder + (chunk ? utf8Decoder.decode(chunk) : '');
      startIndex = re.lastIndex = 0;
      continue;
    }
    yield chunk.substring(startIndex, result.index);
    startIndex = re.lastIndex;
  }

  if (startIndex < chunk.length) {
    // Last line didn't end in a newline char
    yield chunk.substr(startIndex);
  }
}

async function run() {
  for await (const line of makeTextFileLineIterator(urlOfFile)) {
    processLine(line);
  }
}

run();
```