

[Back to Home](#)

Papa Parse

DOCUMENTATION

Parsing strings

`$.parse(inputString[, config])`

Returns a *parse results* object.

Examples

With default settings:

```
var results = $.parse(csvString);
```

With custom *config*:

```
var results = $.parse(csvString, {  
  delimiter: "\t",  
  header: false,  
  dynamicTyping: false,  
  preview: 10,  
  step: function(data, file, inputElem) {  
    console.log(data.results);  
  }  
});
```

Notes

- Parsing strings does not utilize jQuery.
- This is simply a wrapper over the internal "Parser" function that does the heavy lifting.

- If any bad **config settings** are passed in, that setting's default will be used instead.
- If you specify a `step` callback, the input will be streamed and `step` will be executed after each row is parsed.

Parsing files

`$(selector).parse(settings)`

Where `selector` selects file input elements and `settings` is an object as described below.

Example

You can parse one or more files from one or more `<input type="file">` elements like so, where each property is optional:

```
$('#input[type=file]').parse({
  config: {
    // base config to use for each file
  },
  before: function(file, inputElem)
  {
    // executed before parsing each file begins;
    // what you return here controls the flow
  },
  error: function(err, file, inputElem)
  {
    // executed if an error occurs during loading the file,
    // or if the file being iterated is the wrong type,
    // or if the input element has no files selected
  },
  complete: function(results, file, inputElem, event)
  {
    // executed when parsing each file completes;
    // this function receives the parse results
  }
});
```

Notes

- **config** should be a **config object** as described below.
 - **before** is an optional callback that lets you inspect each file before parsing begins. Return:
 - "skip" to skip parsing just that file.
 - false to abort parsing this and all other files in the queue.
 - a config object to alter the options for parsing that file only
 - anything else, including undefined, to continue without any changes
 - **error** is executed when there is a problem getting the file ready to parse. (Parse errors are not reported here.) It receives an object that implements the **DOMError** interface, the File object at hand, and the <input> element from which the file was selected. Errors can occur before reading the file if:
 - the HTML file input element has no files chosen
 - a user-defined callback function ("before") aborted the process
- Otherwise, errors are invoked by FileReader when opening the file. (Loading a non-text file may result in undefined behavior.)
- **complete** is invoked when parsing a file completes. It receives the results of the parse (including parse errors), the File object, the <input> element from which the file was chosen, and the FileReader-generated event.
 - Read about **streaming** for large files.

The config object

Use a config object to specify the parser's behavior.

Any time you invoke the parser, you may customize its behavior using a "config" object. It supports these properties:

- **delimiter** The delimiting character. Leave blank to auto-detect. If you specify a delimiter, it must be a string of length 1, and cannot be `\n`, `\r`, or `"`.
- **header** If true, the first row of parsed data will be interpreted as column titles (fields). Fields are returned separately from the rows, and each data point will be keyed to its field name. If false, the parser simply returns an array of arrays, including the first row.
- **dynamicTyping** If true, fields that are only numeric will be converted to a number type. If false, each parsed datum is returned as a string.
- **preview** If `preview > 0`, only that many rows will be parsed.
- **step** To use a stream, **define a callback function** here which receives the data, row-by-row, as each row is parsed. If parsing a file, step also receives the source file and file input element. Return `false` to abort the process.
- **encoding** UTF-8 is the default encoding for parsing files, but you can specify another. Applies only to reading files.

Default config object

```
{
  delimiter: ",",
  header: true,
  dynamicTyping: true,
  preview: 0,
  step: undefined,
  encoding: "UTF-8"
}
```

Notes

- If using a header row, duplicate field names would be problematic.
- Dynamic typing comes at a slight performance hit, only noticeable for large files. If you don't need it, disable it.
- Step through results by defining a "step" callback function that receives data from the parser after each row is parsed.

Parse results (output)

Structure

Parse output is always an object like this:

```
{
  results: // parse results
  errors:  // parse errors, keyed by row
  meta:    // other info, such as the delimiter used
}
```

Notes

- `results` will be an array of arrays if header row is disabled, or an array of objects if header row is enabled.
- If no delimiter is specified and a delimiter cannot be auto-detected, an error keyed by "config" will be produced and a default delimiter will be chosen.
- With a header row, the field count must be the same on each row or a FieldMismatch error will be produced for that row. (Without a header row, lines can have variable number of fields without errors.)

Example 1

With default config (header row and dynamic typing **enabled**):

```
{
  "results": {
    "fields": [
      "Item",
      "SKU",
      "Cost",
      "Quantity"
    ],
    "rows": [
      {
        "Item": "Book",
        "SKU": "ABC1234",
        "Cost": 10.95,
        "Quantity": 4
      },
      {
        "Item": "Movie",
        "SKU": "DEF5678",
        "Cost": 29.99,
        "Quantity": 3
      }
    ]
  },
}
```

- With a header row, field names are returned as an array, separate from the rows of actual data which are returned as an array of objects. The values are keyed to their field names, which is much easier to work with than index positions.
- Using the header row can degrade performance with really large inputs.
- Notice how dynamic typing turned numeric values into Number types. This also comes at a slight performance hit (you'd only notice with really big inputs).
- Because header row is enabled, errors will be raised for each row that has a different field count from the first (header) row.

Example 2

With header row and dynamic typing **disabled**:

```
{
  "results": [
    [
      "Item",
      "SKU",
      "Cost",
      "Quantity"
    ],
    [
      "Book",
      "ABC1234",
      "10.95",
      "4"
    ],
    [
      "Movie",
      "DEF5678",
      "29.99",
      "3"
    ]
  ],
  "errors": {
    "length": 0
  }
}
```

- The results are returned as an array of arrays because the header row is disabled. You'll have to use plain ol' non-descript index positions to access the values.
- Mismatching field counts will not produce **errors** without a header row.
- This is a fast configuration. With header row and dynamic typing disabled, you should see faster performance for large inputs.

Parse errors

Structure

Parse errors are returned in this format, keyed by the row number, alongside the

"length" property (shown above) which is included for convenience:

```
{
  type: "",      // A generalization of the error
  code: "",      // Standardized error code
  message: "",   // Human-readable details
  line: 0,       // Line of original input
  row: 0,        // Row index of parsed data where error is
  index: 0       // Character index within original input
}
```

Notes

- If no delimiter is specified and a delimiter cannot be auto-detected, an error keyed by "config" will be produced and a default delimiter will be chosen.
- With a header row, the field count must be the same on each row or a FieldMismatch error will be produced for that row. (Without a header row, lines can have variable number of fields without errors.)
- The `type` will be one of "Abort", "Quotes", "Delimiter", or "FieldMismatch".
- The `code` may be:
 - ParseAbort
 - MissingQuotes
 - UnexpectedQuotes
 - UndetectableDelimiter
 - TooFewFields
 - TooManyFields
- The `index` will be the character index across the entire input where the error occurred; it is not the index of the offending character on that line.
- In the event of an error, the Parser makes its best attempt to continue parsing as correctly as possible. For example, if a header row is used and extra fields are found on a line, they will be put into an array keyed by the field name "__parsed_extra".

Streaming files

Papa can load and parse very large files by using streams.

Can Papa load and parse huge text files?

Yes. By defining a **step** callback function, you're able to receive parsed results, row-by-row, as the data is collected. This dramatically reduces memory usage and prevents browsers from crashing.

What is a stream and when should I stream files?

A stream is a unique data structure which, given infinite time, gives you infinite space. So if you're short on memory (as client computers often are), use a stream.

Wait, does that mean streaming takes more time?

Yes and no. Typically, when we gain speed, we pay with space. The opposite is true, too. Streaming uses significantly less memory with large inputs, but since the reading happens in chunks and results are processed at each row instead of at the very end, yes, it can be slower.

But consider the alternative: upload the file to a remote server, open and process it there using a (hopefully) fast and accurate parser, then compress it and have the client download the results. How long does it take you to upload a 500 MB or 1 GB file? Then consider that the server still has to open the file and read its contents, which is what the client would have done minutes ago. The server might parse it faster with natively-compiled binaries, but only if its resources are dedicated to the task and isn't already parsing files for many other users.

So unless your clients have **a fiber line** and you have a scalable cloud application, local parsing by streaming is nearly guaranteed to be faster.

How do I use the `step` function?

Simple example:

```
$('#input[type=file]').parse({
  config: {
    step: function(data, file, inputElem) {
      console.log("Row data:", data.results);
    }
  }
});
```

```
        console.log("Row errors:", data.errors);
    }
},
complete: function() {
    console.log("All done!");
}
});
```

Notice that the function receives data, which has the same structure as the **output described above**.

How do I get all the results together after streaming?

You don't. Unless you assemble it manually. And really, don't do that... it defeats the purpose of using a stream. Just take the bits you need as they come through.

How big should a file be before streaming it?

In some very unscientific testing (with the fastest 2013 Macbook Pro), we were able to load files of about 250 MB for parsing in Chrome without crashing the tab. Beyond that, Chrome started to choke. Actual performance may vary widely. But keep in mind that file size may not be the only factor for choosing to stream.

Why wouldn't I stream the input?

Getting parsed results one row at a time is usually less convenient to work with; it's hard to see the big picture. (But the big picture might be really big.) As results stream in, you can tabulate stats or keep track of whatever you need to, but you wouldn't want to reassemble all the data...

Why should I stream large files, even if they fit in memory?

The space required by the parsed results is often much larger than that of the original input file. The convenience of Javascript objects afforded by 64-bit pointers (to make each value quickly accessible) takes up a lot more space than globbing it together like a file does (at the cost of accessibility). In other words, the output may not fit in memory even if the input does.

Can I stream text without using a file?

Yes, though that's often not necessary. Input that comfortably fits in a textarea usually is small enough that it doesn't need to be streamed.

Does Papa use a true stream?

Papa uses HTML 5's FileReader API to load files, which uses a stream to read in the data. FileReader doesn't technically allow us to hook into the underlying stream (other than providing occasional progress reports), but it does let us load the file in chunks/blobs. Don't worry about that though, because if you want to stream, you'll still get results, row-by-row, into your **step** function.

Dynamic typing

Papa can convert numeric values to true numbers for you

What is dynamic typing?

By default, parsed values are returned as strings. Dynamic typing is a feature built into Papa that converts numeric values to a Number type. When dynamic typing is enabled, parsed values that resemble a number will be converted to one.

Do I need dynamic typing?

If you're performing mathematical operations on the data, then yes, it'll be very helpful. (You'd probably rather add two numbers than concatenate them, right?)

What's the trade-off for using dynamic typing?

Performance, as usual. Each parsed value is matched against a regular expression to determine its numerality. You probably won't notice the degraded performance except with very large inputs. Even then, it may not be significantly slower in many cases. But if you absolutely need the best performance possible, turn off dynamic typing (and header row).

What kinds of numbers does it recognize?

Papa can convert numbers like: 0, "1", -2, 1.23, -4.56, .123, 1., 2., 1.23e4, 5.67E+7, -1.23e4, 5.67e-7, etc.

Does whitespace affect dynamic typing?

If, for some reason, the data is padded by whitespace, it will be ignored. Within the actual data, however, whitespace is significant. For example, floats represented using

scientific notation should not have spaces around the "e" character.

Contribute

Help make Papa better

How to contribute

Please, feel free to [fork Papa on GitHub](#) and submit a [pull request](#). Remember, the Parser component is [under test](#), so if you're making changes to the actual parsing mechanisms, be sure to add a test case to validate your change.

Feedback

You can [open an issue](#) on GitHub to ask questions or start discussion, or you can hashtag [#PapaParse](#) on Twitter.

[Papa Parse](#) is brought to you by [these contributors](#). Thanks!