read-excel-file TS

5.8.8 • Public • Published 8 days ago

read-excel-file

Read small to medium `*.xlsx` files in a browser or Node.js. Parse to JSON with a strict schema.

Demo

Also check out `write-excel-file` for writing simple `*.xlsx` files.

Install

```
npm install read-excel-file --save
```

If you're not using a bundler then use a standalone version from a CDN.

Use

Browser

```
<input type="file" id="input" />


import readXlsxFile from 'read-excel-file'

// File.
const input = document.getElementById('input')
input.addEventListener('change', () => {
  readXlsxFile(input.files[0]).then((rows) => {
    // `rows` is an array of rows
    // each row being an array of cells.
  })
})

// Blob.
fetch('https://example.com/spreadsheet.xlsx')
  .then(response => response.blob())
  .then(blob => readXlsxFile(blob))
  .then((rows) => {
    // `rows` is an array of rows
    // each row being an array of cells.
  })

// ArrayBuffer.
// https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global
//
// Could be obtained from:
// * File
// * Blob
// * Base64 string
//
readXlsxFile(arrayBuffer).then((rows) => {
  // `rows` is an array of rows
  // each row being an array of cells.
})
```

Note: Internet Explorer 11 requires a `Promise` polyfill. **Example**.

Node.js

```
const readXlsxFile = require('read-excel-file/node')

// File path.
readXlsxFile('/path/to/file').then((rows) => {
  // `rows` is an array of rows
  // each row being an array of cells.
})

// Readable Stream.
readXlsxFile(fs.createReadStream('/path/to/file')).then((rows) => {
  // `rows` is an array of rows
  // each row being an array of cells.
})

// Buffer.
readXlsxFile(Buffer.from(fs.readFileSync('/path/to/file'))).then((rows) =>
  // `rows` is an array of rows
  // each row being an array of cells.
})
```

Web Worker

```
const worker = new Worker('web-worker.js')

worker.onmessage = function(event) {
  // `event.data` is an array of rows
  // each row being an array of cells.
  console.log(event.data)
}

worker.onerror = function(event) {
  console.error(event.message)
}
```

```
const input = document.getElementById('input')

input.addEventListener('change', () => {
  worker.postMessage(input.files[0])
})
```

web-worker.js

```
import readXlsxFile from 'read-excel-file/web-worker'

onmessage = function(event) {
  readXlsxFile(event.data).then((rows) => {
    // `rows` is an array of rows
    // each row being an array of cells.
    postMessage(rows)
  })
}
```

JSON

To read spreadsheet data and then convert it to an array of JSON objects, pass a `schema` option when calling `readXlsxFile()`. In that case, instead of returning an array of rows of cells, it will return an object of shape `{ rows, errors }` where `rows` is gonna be an array of JSON objects created from the spreadsheet data according to the `schema`, and `errors` is gonna be an array of errors encountered while converting spreadsheet data to JSON objects.

Each property of a JSON object should be described by an "entry" in the `schema`. The key of the entry should be the column's title in the spreadsheet. The value of the entry should be an object with properties:

- `property` — The name of the object's property.
- `required` — (optional) Required properties of the object could be marked as such.
  - `required: boolean` — `true` or `false`.
  - `required: (object) => boolean` — A function returning `true` or `false` depending on some other properties of the object.

- `validate(value)` — (optional) Cell value validation function. Is only called on non-empty cells. If the cell value is invalid, it should throw an error with the error message set to the error code.
- `type` — (optional) The type of the value. Defines how the cell value will be parsed. If no `type` is specified then the cell value is returned "as is": as a string, number, date or boolean. A `type` could be a:
  - Built-in type:
    - `String`
    - `Number`
    - `Boolean`
    - `Date`
  - "Utility" type exported from the library:
    - `Integer`
    - `Email`
    - `URL`
  - Custom type:
    - A function that receives a cell value and returns a parsed value. If the value is invalid, it should throw an error with the error message set to the error code.

Note on missing columns or empty cells

When converting cell values to object properties, by default, it skips any missing columns or empty cells, which means that property values for such cells will be `undefined`. To be more specific, first it interprets any missing columns as if those columns existed but had empty cells, and then it interprets all empty cells as `undefined`s in the output objects.

In some cases thought that default behavior is not appropriate.

For example, spreadsheet data might be used to update an SQL database using Sequelize ORM library, and Sequelize completely ignores any `undefined` values. In order for Sequelize to set a certain field value to `NULL` in the database, it must be passed as `null` rather than `undefined`.

So for Sequelize use case, property values for any missing columns should stay `undefined` but property values for any empty cells should be `null`. That could be achieved by passing two parameters to `read-excel-file`: `schemaPropertyValueForMissingColumn: undefined` and `schemaPropertyValueForEmptyCell: null`.

An additional option that could be passed in that case would be `schemaPropertyShouldSkipRequiredValidationForMissingColumn: (column, { object`

`}) => true` : it would skip `required` validation for columns that're missing from the spreadsheet.

There's also a legacy parameter `includeNullValues: true` that could be replaced with the following combination of parameters:

- `schemaPropertyValueForMissingColumn: null`
- `schemaPropertyValueForEmptyCell: null`
- `getEmptyObjectValue = () => null`

`errors`

If there were any errors while converting spreadsheet data to JSON objects, the `errors` property returned from the function will be a non-empty array. An element of the `errors` property contains properties:

- `error: string` — The error code. Examples: `"required"`, `"invalid"`.
  - If a custom `validate()` function is defined and it throws a `new Error(message)` then the `error` property will be the same as the `message` value.
  - If a custom `type()` function is defined and it throws a `new Error(message)` then the `error` property will be the same as the `message` value.
- `reason?: string` — An optional secondary error code providing more details about the error. Currently, it's only returned for "built-in" `type` s. Example: `{ error: "invalid", reason: "not_a_number" }` for `type: Number` means that "the cell value is *invalid* because it's *not a number*".
- `row: number` — The row number in the original file. `1` means the first row, etc.
- `column: string` — The column title.
- `value?: any` — The cell value.
- `type?: any` — The schema `type` for this column.

An example of using a `schema`

```
// An example *.xlsx document:
// ----------------------------------------------------------------------
// | START DATE | NUMBER OF STUDENTS | IS FREE | COURSE TITLE |    CONTACT
// ----------------------------------------------------------------------
// | 03/24/2018 |         10         |   true  |  Chemistry   | (123) 456-7
// ----------------------------------------------------------------------

const schema = {
```

```
    'START DATE': {
      // JSON object property name.
      prop: 'date',
      type: Date
    },
    'NUMBER OF STUDENTS': {
      prop: 'numberOfStudents',
      type: Number,
      required: true
    },
    // Nested object example.
    // 'COURSE' here is not a real Excel file column name,
    // it can be any string — it's just for code readability.
    'COURSE': {
      // Nested object path: `row.course`
      prop: 'course',
      // Nested object schema:
      type: {
        'IS FREE': {
          prop: 'isFree',
          type: Boolean
        },
        'COURSE TITLE': {
          prop: 'title',
          type: String
        }
      }
    },
    'CONTACT': {
      prop: 'contact',
      required: true,
      // A custom `type` can be defined.
      // A `type` function only gets called for non-empty cells.
      type: (value) => {
        const number = parsePhoneNumber(value)
        if (!number) {
```

```
          throw new Error('invalid')
        }
        return number
      }
    },
    'STATUS': {
      prop: 'status',
      type: String,
      oneOf: [
        'SCHEDULED',
        'STARTED',
        'FINISHED'
      ]
    }
  }

  readXlsxFile(file, { schema }).then(({ rows, errors }) => {
    // `errors` list items have shape: `{ row, column, error, reason?, value?
    errors.length === 0

    rows === [{
      date: new Date(2018, 2, 24),
      numberOfStudents: 10,
      course: {
        isFree: true,
        title: 'Chemistry'
      },
      contact: '+11234567890',
      status: 'SCHEDULED'
    }]
  })
```

Separate use

The function for converting input data rows to JSON objects using a schema is exported independently as `read-excel-file/map` , if anyone's interested.

```
import mapToObjects from "read-excel-file/map"

const { rows, errors } = mapToObjects(data, schema, options)
```

Maps a list of rows — `data` — into a list of objects — `rows` — using a `schema` as a mapping specification.

- `data` — An array of rows, each row being an array of cells. The first row should be the list of column headers and the rest of the rows should be the data.
- `schema` — A "to JSON" convertion schema (see above).
- `options` — (optional) Schema conversion parameters of `read-excel-file`:
  - `schemaPropertyValueForMissingColumn` — By default, when some of the `schema` columns are missing in the input `data`, those properties are set to `undefined` in the output objects. Pass `schemaPropertyValueForMissingColumn: null` to set such "missing column" properties to `null` in the output objects.
  - `schemaPropertyValueForNullCellValue` — By default, when it encounters a `null` value in a cell in input `data`, it sets it to `undefined` in the output object. Pass `schemaPropertyValueForNullCellValue: null` to make it set such values as `null`s in output objects.
  - `schemaPropertyValueForUndefinedCellValue` — By default, when it encounters an `undefined` value in a cell in input `data`, it it sets it to `undefined` in the output object. Pass `schemaPropertyValueForUndefinedCellValue: null` to make it set such values as `null`s in output objects.
  - `schemaPropertyShouldSkipRequiredValidationForMissingColumn: (column: string, { object }) => boolean` — By default, it does apply `required` validation to `schema` properties for which columns are missing in the input `data`. One could pass a custom `schemaPropertyShouldSkipRequiredValidationForMissingColumn(column, { object })` to disable `required` validation for missing columns in some or all cases.
  - `getEmptyObjectValue(object, { path? })` — By default, it returns `null` for an "empty" resulting object. One could override that value using `getEmptyObjectValue(object, { path })` parameter. The value applies to both top-level object and any nested sub-objects in case of a nested schema, hence the additional (optional) `path?: string` parameter.
  - `getEmptyArrayValue(array, { path })` — By default, it returns `null` for an "empty" array value. One could override that value using `getEmptyArrayValue(array, { path })` parameter.

Returns a list of "mapped objects".

When parsing a schema property value, in case of an error, the value of that property is gonna be `undefined`.

When a "mapped object" is empty, i.e. when all property values of it are `null` or `undefined`, it is returned as `null` rather than an object.

Schema: Tips and Features

- ▶ **Custom `type`** example.
- ▶ **Ignoring empty rows**.
- ▶ How to fix spreadsheet data before `schema` parsing. For example, **how to ignore irrelevant rows**.
- ▶ A **React component for displaying errors** that occured during schema parsing/validation.

JSON (mapping)

Same as above, but simpler: without any parsing or validation.

Sometimes, a developer might want to use some other (more advanced) solution for schema parsing and validation (like `yup`). If a developer passes a `map` option instead of a `schema` option to `readXlsxFile()`, then it would just map each data row to a JSON object without doing any parsing or validation. Cell values will remain "as is": as a string, number, date or boolean.

```
// An example *.xlsx document:
// ------------------------------------------------------------
// | START DATE | NUMBER OF STUDENTS | IS FREE | COURSE TITLE |
// ------------------------------------------------------------
// | 03/24/2018 |         10         |   true  |  Chemistry   |
// ------------------------------------------------------------

const map = {
  'START DATE': 'date',
  'NUMBER OF STUDENTS': 'numberOfStudents',
  'COURSE': {
    'course': {
      'IS FREE': 'isFree',
      'COURSE TITLE': 'title'
    }
```

```
    }
  }

  readXlsxFile(file, { map }).then(({ rows }) => {
    rows === [{
      date: new Date(2018, 2, 24),
      numberOfStudents: 10,
      course: {
        isFree: true,
        title: 'Chemistry'
      }
    }]
  })
```

Multiple Sheets

By default, it reads the first sheet in the document. If you have multiple sheets in your spreadsheet then pass either a sheet number (starting from `1`) or a sheet name in the `options` argument.

```
readXlsxFile(file, { sheet: 2 }).then((data) => {
  ...
})


readXlsxFile(file, { sheet: 'Sheet1' }).then((data) => {
  ...
})
```

By default, `options.sheet` is `1`.

To get the names of all sheets, use `readSheetNames()` function:

```
readSheetNames(file).then((sheetNames) => {
  // sheetNames === ['Sheet1', 'Sheet2']
})
```

Dates

XLSX format originally had no dedicated "date" type, so dates are in almost all cases stored simply as numbers (the count of days since `01/01/1900`) along with a **"format"** description (like `"d mmm yyyy"`) that instructs the spreadsheet viewer software to format the date in the cell using that certain format.

When using `readXlsx()` with a `schema` parameter, all schema columns having type `Date` are automatically parsed as dates. When using `readXlsx()` without a `schema` parameter, this library attempts to guess whether a cell contains a date or just a number by examining the cell's "format" — if the "format" is one of the **built-in date formats** then such cells' values are automatically parsed as dates. In other cases, when date cells use a non-built-in format (like `"mm/dd/yyyy"`), one can pass an explicit `dateFormat` parameter to instruct the library to parse numeric cells having such "format" as dates:

```
readXlsxFile(file, { dateFormat: 'mm/dd/yyyy' })
```

Trim

By default, it automatically trims all string values. To disable this feature, pass `trim: false` option.

```
readXlsxFile(file, { trim: false })
```

Parse Numbers

By default, it parses numeric cell values from strings. In some rare cases though, javascript's **inherently limited** floating-point number precision might become an issue. An example might be finance and banking domain. To work around that, this library supports passing a custom `parseNumber(string)` function option.

```
// Arbitrary-precision numbers in javascript.
import Decimal from 'decimal.js'

readXlsxFile(file, {
```

```
    parseNumber: (string) => new Decimal(string)
  })
```

Transform

Sometimes, a spreadsheet doesn't exactly have the structure required by this library's `schema` parsing feature: for example, it may be missing a header row, or contain some purely presentational / empty / "garbage" rows that should be removed. To fix that, one could pass an optional `transformData(data)` function that would modify the spreadsheet contents as required.

```
readXlsxFile(file, {
  schema,
  transformData(data) {
    // Add a missing header row.
    return [['ID', 'NAME', ...]].concat(data)
    // Remove empty rows.
    return data.filter(row => row.filter(column => column !== null).length
  }
})
```

Limitations

Performance

There have been some **reports** about performance issues when reading very large `*.xlsx` spreadsheets using this library. It's true that this library's main point have been usability and convenience, and not performance when handling huge datasets. For example, the time of parsing a file with 2000 rows / 20 columns is about 3 seconds. So, for reading huge datasets, perhaps use something like `xlsx` package instead. There're no comparative benchmarks between the two, so if you'll be making one, share it in the Issues.

Formulas

Dynamically calculated cells using formulas (`SUM` , etc) are not supported.

TypeScript

I'm not a TypeScript expert, so the community has to write the typings (and test those). See `example index.d.ts` .

CDN

One can use any npm CDN service, e.g. **unpkg.com** or **jsdelivr.net**

```
<script src="https://unpkg.com/read-excel-file@5.x/bundle/read-excel-file.m

<script>
  var input = document.getElementById('input')
  input.addEventListener('change', function() {
    readXlsxFile(input.files[0]).then(function(rows) {
      // `rows` is an array of rows
      // each row being an array of cells.
    })
  })
</script>
```

TypeScript

This library comes with TypeScript "typings". If you happen to find any bugs in those, create an issue.

References

Uses `xmldom` for parsing XML.

GitHub

On March 9th, 2020, GitHub, Inc. silently **banned** my account (erasing all my repos, issues and comments, even in my employer's private repos) without any notice or explanation. Because of that, all source codes had to be promptly moved to GitLab. The **GitHub repo** is now only used as a backup (you can star the repo there too), and the primary repo is now the **GitLab one**. Issues can be reported in any repo.

License

MIT

Keywords

excel xlsx browser json

Install

```
> npm i read-excel-file
```

Repository

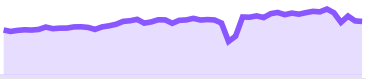
gitlab.com/catamphetamine/read-excel-file

Homepage

gitlab.com/catamphetamine/read-excel-file#readme

Weekly Downloads

127,173



Version	License
5.8.8	MIT

Unpacked Size	Total Files
1.12 MB	244

Last publish

8 days ago

Collaborators

>_Try on RunKit

Report malware

Support

Help

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy