Guide to ExcelJS: An Excel Workbook Manager

ExcelJS is a helpful JavaScript package that acts as an Excel workbook manager. This guide will help you read, manipulate and write spreadsheet data to XLSX and JSON.

Written by Wai Foong Ng
Published on Dec. 21, 2022

Image: Shutterstock / Built In

ExcelJS is a JavaScript package that acts as an Excel workbook manager. `exceljs` can read, manipulate and write spreadsheet data and styles to XLSX and JSON, according to its official documentation. It's reverse-engineered from Excel spreadsheet files as a project.

- Creating a worksheet.
- Handling headers and footers.
- Setting frozen or split views.
- Setting auto filters.
- Data manipulation on rows and columns.
- Adding data validation.
- Adding styles.
- Inserting images to a workbook.

In addition, `exceljs` is frequently updated and available for free. This tutorial covers the step-by-step installation process and a few examples of some of the most basic and commonly used features. Also, code snippets will be provided as reference to the readers.

WHAT IS EXCELJS?

ExcelJS is a JavaScript package that works as an Excel workbook manager. It can read, manipulate and write spreadsheet data and styles to XLSX and JSON.

Let's proceed to the next section and start installing `exceljs`.

How to Install ExcelJS

The recommended installation process is via `npm`, which mainly works for node.js projects. If you intend to use it directly within a browser for a project outside of node.js, you have to import the package using the official content delivery network (CDN) link. Let's have a look at the installation process for both methods.

NPM

Run the following command to install `npm` in your project:

```
npm install exceljs
```

Then, import it as follows:

```
const ExcelJS = require('exceljs');
```

For a node.js that is more than 10 years old, you have to import using ES5 transpiled code. Also, you need to import a few polyfills, as well:

```
// polyfills required by exceljs
require('core-js/modules/es.promise');
require('core-js/modules/es.string.includes');
require('core-js/modules/es.object.assign');
require('core-js/modules/es.object.keys');
require('core-js/modules/es.symbol');
require('core-js/modules/es.symbol.async-iterator');
require('regenerator-runtime/runtime');

const ExcelJS = require('exceljs/dist/es5');
```

CDN

For using it directly on your browser, simply head over to the official CDN page for ExcelJS and copy the corresponding script tag based on your needs:

FILESAVER.JS

Some of the features in `exceljs` are highly dependent on node.js built-in packages. For example, the save file functionality requires the `fs` module. This will throw an error when you call it within a non-node.js project.

Having said that, you can still utilize the file saving feature with just a few additional tricks. All you need to do is output the binary data as a buffer and save it using an external JavaScript library, such as FileSaver.js. Head over to the repository and copy the script tag based on your use cases.

Then, insert the script tag in your HTML as follows:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/FileSaver.js/2.0.5/FileSaver.min.js"
```

How to Use ExcelJS

In this section, you will learn the fundamentals concepts and the basics behind `exceljs`.

CREATING A NEW WORKBOOK

Let’s start by creating a new workbook, as follows:

WORKBOOK PROPERTIES

Once you have a workbook, you can play around with some of its properties:

```
workbook.creator = 'Ng Wai Foong';
workbook.lastModifiedBy = 'Bot';
workbook.created = new Date(2021, 8, 30);
workbook.modified = new Date();
workbook.lastPrinted = new Date(2021, 7, 27);
```

Adding a New Worksheet

Next, you can add a new worksheet to the workbook that you created earlier via the following code:

```
const worksheet = workbook.addWorksheet('New Sheet');
```

WORKSHEET SETTINGS

You can initialize some configurations for your sheet as well by passing in an object:

```
const worksheet = workbook.addWorksheet('New Sheet', {views: [{showGridLines: fal

// freeze first row and column
const worksheet = workbook.addWorksheet('New Sheet', {views:[{state: 'frozen', xS

// pageSetup settings for A4 - landscape
const worksheet =  workbook.addWorksheet('New Sheet', {
  pageSetup:{paperSize: 9, orientation:'landscape'}
});

// headers and footers
const worksheet = workbook.addWorksheet('New Sheet', {
  headerFooter: {oddFooter: "Page &P of &N";, oddHeader: 'Odd Page'}
});
```

You can also set the configurations manually after you have initialized the worksheet object. For example, you can set a footer for odd pages, as follows:

```
//Set footer (default centered), result: "Page 2 of 16"
worksheet.headerFooter.oddFooter = "Page &P of &N";
```

`&P` and `&N` are part of the script commands.

How to Set Auto Filters in ExcelJS

Moreover, there is support for auto filters as well. The following code snippet allows you to set an auto filter from A1 to C1:

standard form:

```
worksheet.autoFilter = {
 from: 'A1',
 to: 'C1',
}
```

CREATING COLUMNS

It's good practice to set the columns with the corresponding mapping header and key.

```
worksheet.columns = [
 { header: 'Id', key: 'id' },
 { header: 'Name', key: 'name' },
 { header: 'Age', key: 'age' }
];
```

Header represents the text that will be displayed, while key represents the property name mapping when you add a new row using an object. Please note that the order is important, as it will affect the final output when you save the workbook as an excel file.

ACCESSING A COLUMN

You can access each column individually via:

- `key`: The mapping key declared when creating the column (ID, name, age, etc.).
- `letter`: The corresponding letter for the column (A, B, C, etc.).

```
const nameCol = worksheet.getColumn('B');
const ageCol = worksheet.getColumn(3);
```

ITERATING EACH CELL

You can easily iterate all current cells in the column that is not empty:

```
ageCol.eachCell(function(cell, rowNumber) {

});
```

If you want to iterate over empty cells, simply set the includeEmpty property to true as follows:

```
dobCol.eachCell({ includeEmpty: true }, function(cell, rowNumber) {
});
```

How to Add a Row in ExcelJS

Once you have declared the corresponding columns, you can start to add data to it by calling the `addRow` function. There are multiple ways to add a row to the worksheet.

ADDING A NEW ROW USING KEY-VALUE OBJECT

For example, you can pass in key-value object that matched the header columns:

ADDING A NEW ROW USING ARRAY

You can use an array as well to add new data. It will assign to the columns in order (A, B, C):

```
const row = worksheet.addRow([2, 'Mary Sue', 22]);
```

ADDING ROWS OF DATA

In addition, you can call `addRows` function and pass in an array of row objects instead. This allows you to add rows of data with just a single line of code:

```
const rows = [
 [3,'Alex','44'],
 {id:4, name: 'Margaret', age: 32}
];
worksheet.addRows(rows);
```

ADDING PAGE BREAK

There is also a function called addPageBreak that adds a new page break below the row. You can call it as follows:

```
row.addPageBreak();
```

```
worksheet.eachRow(function(row, rowNumber) {
});
```

ITERATING EACH CELL

Similar to column, you can iterate over each cell of the row via the following

```
row.eachCell(function(cell, colNumber) {
});
```

METRICS

There are a few built-in metrics for getting the number of rows and total number of cells:

```
const rowSize = row.cellCount;
const numValues = row.actualCellCount;
```

DATA VALIDATION

You can easily incorporate data validation to your worksheet programmatically via `dataValidation`. For example, you can set a dropdown with a list of values via the following code:

```
};
```

Here’s another example that limits the input to be in between 2.5 and seven. You can add a tooltip message to guide the users by setting the prompt property.

```
worksheet.getCell('A1').dataValidation = {
  type: 'decimal',
  operator: 'between',
  allowBlank: true,
  showInputMessage: true,
  formulae: [2.5, 7],
  promptTitle: 'Decimal',
  prompt: 'The value must between 2.5 and 7'
};
```

Styling in ExcelJS

`exceljs` supports a rich set of styling and formatting for cells, rows and columns. It comes with the following properties:

- `numFmt`
- `font`
- `alignment`
- `border`
- `fill`

```
// display value as '1 3/5'
worksheet.getCell('A1').value = 1.6;
worksheet.getCell('A1').numFmt = '# ?/?';
// display value as '1.60%'
worksheet.getCell('B1').value = 0.016;
worksheet.getCell('B1').numFmt = '0.00%';
```

With the exception of `numFmt`, which accepts a string, the rest of the style takes in a JavaScript object.

FONT

For example, you can set the font via the following code snippet:

```
worksheet.getCell('A1').font = {
 name: 'Arial Black',
 color: { argb: 'FF00FF00' },
 family: 2,
 size: 14,
 italic: true
};
```

ALIGNMENT

On the other hand, you can easily set the alignment to top right, as follows:

```
// set cell indent to 1
worksheet.getCell('A1').alignment = { indent: 1 };
```

BORDER

To style the border of a cell based on your desired color, use the following code instead:

```
worksheet.getCell('A1').border = {
 top: {style:'double', color: {argb:'FF00FF00'}},
 left: {style:'double', color: {argb:'FF00FF00'}},
 bottom: {style:'double', color: {argb:'FF00FF00'}},
 right: {style:'double', color: {argb:'FF00FF00'}}
};
```

The underlying style property accepts the following:

- `thin`
- `dotted`
- `dashDot`
- `hair`
- `dashDotDot`
- `slantDashDot`
- `mediumDashed`
- `mediumDashDotDot`
- `mediumDashDot`
- `medium`
- `double`

FILLS

For filling a particular cell, simply pass in an object with the following property:

```
worksheet.getCell('A1').fill = {
 type: 'pattern',
 pattern:'darkTrellis',
 fgColor:{argb:'FFFFFF00'},
 bgColor:{argb:'FF0000FF'
};
```

`fgColor` refers to the foreground color, while `bgColor` refers to the back end color.

The complete list for pattern types is as follows:

- `none`
- `solid`
- `darkGray`
- `mediumGray`
- `lightGray`
- `gray125`
- `gray0625`
- `darkHorizontal`
- `darkVertical`
- `darkDown`
- `darkUp`
- `darkGrid`
- `darkTrellis`
- `lightHorizontal`
- `lightVertical`

INHERITANCE

Please note that when you set a specific style to a row or column, `exceljs` will internally apply the same style to all existing cells in that row or column. Newly created cells will inherit the style as well.

As a result,it's recommended to add cells before you set the style. The following code snippet illustrates an example of setting all the headers to bold after cells creation:

```
// arrayObj is an array of objects
worksheet.addRows(arrayObj);
worksheet.getRow(1).font = { bold: true };
```

File I/O

`exceljs` provides a few helper functions for reading and loading a file depending on the use cases. Generally, it's categorized into:

- `file`
- `stream`
- `buffer`

Please note that File I/O related operations return a `Promise` and have to be called together with the `await` keyword. If you are using it inside a function, make sure to set

Assuming that you have a local excel file, you can perform reading and writing as follows:

```
// file reading
await workbook.xlsx.readFile(filename);
// file writing
await workbook.xlsx.writeFile(filename);
```

STREAM

If you have a stream of data, use the following code snippet:

```
// stream reading
await workbook.xlsx.read(stream);
// stream writing
await workbook.xlsx.write(stream);
```

BUFFER

When dealing with buffer, you should call load and `writeBuffer` function instead:

```
// buffer reading
await workbook.xlsx.load(data);
// buffer writing
const buffer = await workbook.xlsx.writeBuffer();
const blob = new Blob([fileList[0]], { type: 'application/vnd.openxmlformats-offi
const buffer = await blob.arrayBuffer();

const workbook = new ExcelJS.Workbook();
await workbook.xlsx.load(buffer);
```

A tutorial on how to work with XLSX in JavaScript. | Video: Vincent Lab

MORE ON JAVASCRIPT
JavaScript Call Stacks: An Introduction

FILE I/O FOR A NON-NODE.JS PROJECT

As mentioned earlier, the read and write file functions rely on the `fs` module, which will throw an error if you're using it on a browser. In order to resolve this, simply call the `writeBuffer` function and save the output buffer data into a BLOB. Then, save it using FileSaver.js.

```
  await workbook.xlsx.load(buffer);
```

Make sure you import the appropriate `script` tag for FileSaver.js. Check the installation section for more information on this.

Data Science Expert Contributors

Software Engineering Perspectives