Reading Excel and CSV Test Data in Playwright with TypeScript: Detailed Notes

A simple, step-by-step guide to reading data from Excel (.xlsx) and CSV (.csv) files into your Playwright tests.

1. Why Use Excel or CSV for Test Data?

Excel and CSV files store data in **rows and columns**, like a table. They are a good choice when:

- You have many rows of similar data, such as 50 login cases.
- Testers or business people who do not write code maintain the data.
- You want to run the **same test with many rows of data** (data-driven testing).

	CSV (.csv)	**Excel (.xlsx)**
File type	Plain text	Binary (zipped XML)
Opens in	Any text editor, Excel	Excel, Google Sheets
Multiple sheets	No (one table per file)	Yes
Formatting, formulas	No	Yes
Easy to review in Git	Yes (changes are readable)	No
Extra library needed	Yes, or use the same library as Excel	Yes

Rule of thumb: use **CSV** for simple tables, and **Excel** when you need several sheets or the business team edits the data in Excel.

2. How Data Files Are Laid Out

The **first row is the header**. It contains the column names. Every row after it is one test case.

Excel sheet named Login:

username	**password**	**expected**	**run**
standard_user	secret_sauce	success	Y
locked_out_user	secret_sauce	locked	Y
standard_user	wrong_password	invalid	Y
problem_user	secret_sauce	success	N

The same data as a CSV file:

username,password,expected,run

standard_user,secret_sauce,success,Y

locked_out_user,secret_sauce,locked,Y

standard_user,wrong_password,invalid,Y

problem_user,secret_sauce,success,N

In CSV, values are separated by commas, and each line is a row. If a value contains a comma, put it in double quotes, for example "Rao, Asha".

The extra run column is a useful trick: it lets anyone switch a row on (Y) or off (N) without changing code.

3. The Simple Approach: 5 Steps

1. Put the data file in a test-data folder.
2. Install a small library to read it.
3. Create a helper function that returns the rows.
4. Describe a row with a TypeScript type.
5. Read the rows at the top of your test file and create one test per row.

4. Project Layout

playwright-ts-demo/

├── test-data/

│ ├── login-data.xlsx

│ └── login-data.csv

├── utils/

│ └── dataReader.ts

├── tests/

│ └── login-data.spec.ts

└── playwright.config.ts

Paths in the code use path.resolve('test-data', fileName), which starts from the folder where you run npx playwright test (normally the project root).

5. Install the Libraries

npm install xlsx # reads Excel and also CSV

npm install csv-parse # optional: a dedicated CSV reader

Library	**Reads**	**Style**	**Notes**
xlsx (SheetJS)	.xlsx, .xls, .csv	Synchronous	One library for both formats
csv-parse	.csv	Synchronous (csv-parse/sync)	Many options for tricky CSV files
papaparse	.csv	Synchronous on text	Popular in web projects

exceljs	.xlsx	Asynchronous (await)	Cell-level control. Cannot create tests at file level

Note about xlsx: the copy on the npm registry is an older release. SheetJS publishes newer builds on its own download site, and its documentation explains how to install them. Check the installation page and choose what fits your project and company rules.

Why synchronous matters: to create **one test per row**, the data must be ready when the test file loads, so use a synchronous reader.

6. Reading an Excel File (xlsx)

6.1 The reader helper

```
// utils/dataReader.ts

import * as XLSX from 'xlsx';

import path from 'path';


export function readExcel<T>(fileName: string, sheetName?: string): T[] {

  const filePath = path.resolve('test-data', fileName);

  const workbook = XLSX.readFile(filePath);


  const name = sheetName ?? workbook.SheetNames[0];   // first sheet if none is given

  const sheet = workbook.Sheets[name];

  if (!sheet) {

    throw new Error(`Sheet "${name}" was not found in ${fileName}`);

  }


  return XLSX.utils.sheet_to_json<T>(sheet, { defval: '' });

}
```

Line by line:

Code	What it does
XLSX.readFile(filePath)	Opens the workbook
workbook.SheetNames	The list of sheet names, in order
workbook.Sheets[name]	One sheet
sheet_to_json<T>()	Turns each row into an object, with the header row as property names

defval: ''	Fills empty cells with an empty string, so the property always exists

6.2 Useful options for sheet_to_json

Need	**Option**
Get cell text exactly as Excel shows it	{ raw: false }
Fill blank cells	{ defval: '' }
Get each row as an array, not an object	{ header: 1 }
Read only part of the sheet	{ range: 'A1:D10' }
Read dates as real dates	XLSX.readFile(path, { cellDates: true })

If you do not use cellDates, Excel dates arrive as serial numbers such as 45123.

6.3 Reading a single cell

```
const workbook = XLSX.readFile(path.resolve('test-data/login-data.xlsx'));

const sheet = workbook.Sheets['Login'];


console.log(sheet['A2']?.v);   // raw value of cell A2: standard_user
```

7. Reading a CSV File

7.1 Option A: use xlsx (simplest, no extra library)

The same xlsx library can open a CSV file. A CSV is treated as a workbook with one sheet, so you can use readExcel() from above:

```
const rows = readExcel<LoginRow>('login-data.csv');
```

This is the quickest way when you already have xlsx installed.

7.2 Option B: use csv-parse (recommended for CSV)

```
// utils/dataReader.ts (add to the same file)

import fs from 'fs';

import { parse } from 'csv-parse/sync';


export function readCsv<T>(fileName: string): T[] {

  const filePath = path.resolve('test-data', fileName);

  const text = fs.readFileSync(filePath, 'utf-8');


  return parse(text, {
```

```
    columns: true,          // use the first row as property names
    skip_empty_lines: true,  // ignore blank lines
    trim: true,             // remove spaces around values
    bom: true,              // handle the hidden mark that Excel adds to some CSV files
  }) as T[];
}
```

Options you may need:

Option	Purpose
columns: true	Return objects keyed by the header names
skip_empty_lines: true	Ignore empty lines at the end of the file
trim: true	Remove extra spaces
bom: true	Remove the byte order mark that some programs put at the start of a file
delimiter: ';'	Use a semicolon, which some regional settings produce
cast: true	Convert numbers and similar values from text automatically

7.3 Option C: the manual way (not recommended)

Splitting each line on commas (line.split(',')) is tempting, but it breaks when a value contains a comma inside quotes, such as "Rao, Asha". Use a library instead.

8. Describe a Row with a TypeScript Type

```
type LoginRow = {
  username: string;
  password: string;
  expected: 'success' | 'locked' | 'invalid';
  run: string;   // 'Y' or 'N'
};
```

This gives you autocomplete and warns about spelling mistakes. Remember that it is only a promise to TypeScript. It does **not** check the real file at run time.

9. Create One Test Per Row (Data-Driven)

```
// tests/login-data.spec.ts
import { test, expect } from '@playwright/test';
import { readExcel, readCsv } from '../utils/dataReader';
```

```
type LoginRow = {
  username: string;
  password: string;
  expected: string;
  run: string;
};

// Choose ONE of these sources
const allRows = readExcel<LoginRow>('login-data.xlsx', 'Login');
// const allRows = readCsv<LoginRow>('login-data.csv');

// Keep only rows marked Y
const rows = allRows.filter((row) => String(row.run).toUpperCase() === 'Y');

for (const [index, row] of rows.entries()) {
  test(`login #${index + 1}: ${row.username} should be ${row.expected}`, async ({ page }) => {
    await page.goto('https://www.saucedemo.com/');
    await page.getByPlaceholder('Username').fill(String(row.username));
    await page.getByPlaceholder('Password').fill(String(row.password));
    await page.getByRole('button', { name: 'Login' }).click();

    if (row.expected === 'success') {
      await expect(page).toHaveURL(/inventory/);
    } else if (row.expected === 'locked') {
      await expect(page.locator('[data-test="error"]')).toContainText('locked out');
    } else {
      await expect(page.locator('[data-test="error"]')).toContainText('do not match');
    }
  });
}
```

Run it:

npx playwright test tests/login-data.spec.ts

With the sample data above, three tests run, and the row marked N is skipped. To add a new case, you add a row to the file. No code changes are needed.

Points to remember:

- Every generated test needs a **unique title**, so the example includes the row number.
- String(row.username) protects you when Excel stores a value as a number, for example 12345.
- Data is read once, when the file loads.

10. Converting Values

CSV values are always **text**. Excel values can be numbers, text, booleans, or dates. Convert them when you need a specific type.

```
const quantity = Number(row.quantity);                    // '3' becomes 3

const isActive = String(row.active).toLowerCase() === 'true'; // 'TRUE' becomes true

const zip = String(row.zip).padStart(6, '0');             // 400 becomes '000400'
```

If a field such as a zip code or phone number loses its leading zero in Excel, format the column as **Text** in Excel before typing the values.

11. Reading Data Inside a Test or Hook

You can also read the file inside beforeAll or a test. This works for normal tests, but you **cannot** create one test per row this way, because test titles are needed before tests run.

```
let rows: LoginRow[];


test.beforeAll(() => {
  rows = readCsv<LoginRow>('login-data.csv');
});


test('uses the first row', async ({ page }) => {
  console.log(rows[0].username);
});
```

12. Using Excel Data with exceljs (Async Alternative)

Use exceljs when you need cell-level control, such as reading a single cell, styles, or merged cells. It reads asynchronously, so use it inside a test or hook.

npm install exceljs

```
import { test } from '@playwright/test';
```

```
import * as ExcelJS from 'exceljs';
import path from 'path';

test('read Excel with exceljs', async () => {
  const workbook = new ExcelJS.Workbook();
  await workbook.xlsx.readFile(path.resolve('test-data/login-data.xlsx'));

  const sheet = workbook.getWorksheet('Login');
  if (!sheet) throw new Error('Login sheet not found');

  sheet.eachRow((row, rowNumber) => {
    if (rowNumber === 1) return;              // skip the header row
    console.log(row.getCell(1).value);        // column A
  });
});
```

13. A Simple Safety Check

Catch bad data early with a small check before the tests are created.

```
function checkRows(rows: LoginRow[], source: string): void {
  if (rows.length === 0) {
    throw new Error(`${source} has no data rows`);
  }
  rows.forEach((row, i) => {
    if (!row.username || !row.password) {
      throw new Error(`${source} row ${i + 2} is missing username or password`);
    }
  });
}

checkRows(allRows, 'login-data.xlsx');
```

The row number is shown as i + 2 because row 1 is the header, which matches what the person sees in Excel.

14. Best Practices

- Keep data files in test-data and give them clear names.
- Use **one helper file** for all reading code.
- Keep column names simple, with no spaces, such as username, expected, and run.
- Match the header names to your TypeScript property names exactly, including capital letters.
- Add a run column to switch rows on or off.
- Give every generated test a unique title.
- Do not store real passwords or secrets in data files that go to Git.
- Keep Excel files small and simple. Avoid merged cells, blank rows in the middle of data, and formulas that depend on other files.
- Save CSV files with **UTF-8** encoding.