Reading JSON Test Data in Playwright with TypeScript: Detailed Notes

A simple, step-by-step guide to reading data from a JSON file into your Playwright tests.

1. What Is JSON and Why Use It for Test Data?

JSON is a plain text format for storing data. It is easy for people to read and easy for TypeScript to understand.

Why keep test data in a JSON file instead of writing it inside the test:

- You change the data in **one place**, without touching test code.
- You can run the **same test with many sets of data**.
- Tests stay short and clean.
- The data file can be reviewed in Git, because it is plain text.

2. JSON File

```
{
  "name": "Asha",
  "age": 30,
  "active": true,
  "skills": ["playwright", "typescript"],
  "address": { "city": "Mumbai" },
  "manager": null
}
```

Key Points	**Example**
Keys must be in **double quotes**	"name" (not name or 'name')
Text values use **double quotes**	"Asha" (single quotes are not allowed)
Numbers, true, false, and null have **no quotes**	30, true, null
Items are separated by commas, with **no comma after the last item**	{ "a": 1, "b": 2 }
No comments are allowed	// note will break the file

Tip: VS Code highlights JSON mistakes with red underlines, so open the file in VS Code and fix them before running tests.

3. The Simple Approach: 5 Steps

1. Create a test-data folder and put a .json file in it.
2. Create a small helper function that reads the file.
3. Describe the shape of the data with a TypeScript type.
4. Call the helper in your test file.
5. Use the data inside your tests.

4. Step 1: Create the JSON File

```
playwright-ts-demo/
├── test-data/
│   └── users.json
├── utils/
│   └── jsonReader.ts
├── tests/
│   └── login.spec.ts
└── playwright.config.ts
```

test-data/users.json:

```
[
  { "username": "standard_user", "password": "secret_sauce", "expected": "success" },
  { "username": "locked_out_user", "password": "secret_sauce", "expected": "locked" },
  { "username": "standard_user", "password": "wrong_password", "expected": "invalid" }
]
```

This file is a **list** (square brackets) of **objects** (curly brackets). Each object is one row of test data.

5. Step 2: Create the Reader Helper

utils/jsonReader.ts:

```
import fs from 'fs';
import path from 'path';
export function readJson<T>(fileName: string): T {
  const filePath = path.resolve('test-data', fileName);
  if (!fs.existsSync(filePath)) {
    throw new Error(`Test data file not found: ${filePath}`);
  }
```

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


  try {
    return JSON.parse(text) as T;
  } catch (error) {
    throw new Error(`Invalid JSON in ${fileName}: ${(error as Error).message}`);
  }
}
```

Line by line:

Code	What it does
import fs from 'fs'	Node's built-in tool for reading files
import path from 'path'	Node's built-in tool for building file paths
path.resolve('test-data', fileName)	Builds the full path. It starts from the folder where you run the tests (your project root)
fs.existsSync(filePath)	Checks that the file is there, so you get a clear error if not
fs.readFileSync(filePath, 'utf-8')	Reads the file content as text
JSON.parse(text)	Converts the text into real JavaScript data
as T	Tells TypeScript what shape the data has
try / catch	Gives a clear message when the JSON has a typing mistake

You write this helper once and reuse it in every test file.

6. Step 3: Describe the Data with a Type

A TypeScript type describes what each row looks like. It gives you autocomplete and warns you about spelling mistakes.

```
type LoginData = {
  username: string;
  password: string;
  expected: 'success' | 'locked' | 'invalid';
};
```

Important: the type is only a promise to TypeScript. It does **not** check the real file when the test runs. If the JSON has a different shape, you will find out at run time (see Section 12 for a simple check).

7. Steps 4 and 5: Read and Use the Data

7.1 Use data in a single test

```
import { test, expect } from '@playwright/test';
import { readJson } from '../utils/jsonReader';

type LoginData = { username: string; password: string; expected: string };

const users = readJson<LoginData[]>('users.json');

test('login with the first user', async ({ page }) => {
  const user = users[0];

  await page.goto('https://www.saucedemo.com/');
  await page.getByPlaceholder('Username').fill(user.username);
  await page.getByPlaceholder('Password').fill(user.password);
  await page.getByRole('button', { name: 'Login' }).click();

  await expect(page).toHaveURL(/inventory/);
});
```

readJson<LoginData[]> means: read the file, and treat it as a list of LoginData.

7.2 One test for every row (data-driven)

Use a for loop to create a separate test for each object in the file.

```
import { test, expect } from '@playwright/test';
import { readJson } from '../utils/jsonReader';

type LoginData = {
  username: string;
  password: string;
  expected: 'success' | 'locked' | 'invalid';
};

const users = readJson<LoginData[]>('users.json');
```

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

    if (user.expected === 'success') {
      await expect(page).toHaveURL(/inventory/);
    } else if (user.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.spec.ts
```

The report shows **three tests**, one for each row. To add a new case, you only add a new object to users.json.

Rules for generated tests:

- Each test title must be **unique**, so add the row number (index + 1) or an ID.
- The file is read **once**, when the test file loads, which keeps tests fast.

8. Different Shapes of JSON Data

8.1 An object with named entries

Useful when you want one specific item, such as a user by role.

test-data/accounts.json:

```
{
  "standard": { "username": "standard_user", "password": "secret_sauce" },
  "locked":   { "username": "locked_out_user", "password": "secret_sauce" }
}
```

```
type Account = { username: string; password: string };
type Accounts = Record<string, Account>;

const accounts = readJson<Accounts>('accounts.json');

test('login with the standard account', async ({ page }) => {
  await page.goto('https://www.saucedemo.com/');
  await page.getByPlaceholder('Username').fill(accounts.standard.username);
  await page.getByPlaceholder('Password').fill(accounts.standard.password);
  await page.getByRole('button', { name: 'Login' }).click();
  await expect(page).toHaveURL(/inventory/);
});
```

Record<string, Account> means: an object whose keys are text and whose values are Account objects.

8.2 A list of expected values to verify on the page

test-data/products.json:

```
[
  { "name": "Sauce Labs Backpack", "price": "$29.99" },
  { "name": "Sauce Labs Bike Light", "price": "$9.99" },
  { "name": "Sauce Labs Fleece Jacket", "price": "$49.99" }
]
```

```
type Product = { name: string; price: string };
const products = readJson<Product[]>('products.json');

test('product prices are correct', async ({ page }) => {
  await page.goto('https://www.saucedemo.com/');
```

```
  await page.getByPlaceholder('Username').fill('standard_user');
  await page.getByPlaceholder('Password').fill('secret_sauce');
  await page.getByRole('button', { name: 'Login' }).click();

  for (const product of products) {
    const card = page.locator('.inventory_item').filter({ hasText: product.name });
    await expect(card.locator('.inventory_item_price')).toHaveText(product.price);
  }
});
```

Here the JSON holds the **expected results**, and the test compares them with the page.

8.3 Nested data

JSON can have objects inside objects.

```
{
  "customer": {
    "firstName": "Asha",
    "lastName": "Rao",
    "address": { "city": "Mumbai", "zip": "400001" }
  },
  "items": ["Sauce Labs Backpack", "Sauce Labs Bike Light"]
}
type OrderData = {
  customer: {
    firstName: string;
    lastName: string;
    address: { city: string; zip: string };
  };
  items: string[];
};

const order = readJson<OrderData>('order.json');
```

```
console.log(order.customer.address.city);   // Mumbai
console.log(order.items.length);            // 2
```

8.4 Settings for different environments

test-data/environment.json:

```
{
  "qa":     { "baseUrl": "https://qa.example.com",      "user": "qa_user" },
  "staging": { "baseUrl": "https://staging.example.com", "user": "staging_user" }
}
type Env = { baseUrl: string; user: string };

const allEnvs = readJson<Record<string, Env>>('environment.json');
const envName = process.env.TEST_ENV ?? 'qa';       // default is qa
const env = allEnvs[envName];

if (!env) {
  throw new Error(`Unknown environment: ${envName}`);
}
```

Choose the environment when you run the tests:

```
# macOS and Linux
TEST_ENV=staging npx playwright test

# Windows PowerShell
$env:TEST_ENV='staging'; npx playwright test
```

9. Another Way: Import the JSON File Directly

Turn on JSON support in tsconfig.json (create this file in the project root if you do not have one):

```
{
  "compilerOptions": {
    "resolveJsonModule": true,
    "esModuleInterop": true
  }
}
```

Then import the file like any other module:

```
import users from '../test-data/users.json';


test('first user name', async () => {
  console.log(users[0].username);
});
```

Points	**readJson() helper**	**Direct import**
Setup needed	None	resolveJsonModule in tsconfig.json
Types	You write them	TypeScript guesses them from the file
Clear error for a missing or broken file	Yes, from your helper	Basic error
Works with ES module projects	Yes	May need extra import syntax

For beginners, the **helper function is the simplest and most reliable choice**.

10. Best Practices

- Keep JSON files in the test-data folder and give them clear names, such as users.json.
- Use one helper (readJson) for the whole project.
- Write a TypeScript type for every JSON shape.
- Give each generated test a unique title.
- Keep files small and focused. Do not put all data into one giant file.
- Do **not** store real passwords, tokens, or secrets in JSON files that go to Git. Use environment variables for secrets.
- Use the same property names in JSON and TypeScript, with the same spelling and capital letters.
- Check the JSON for errors in VS Code before running.