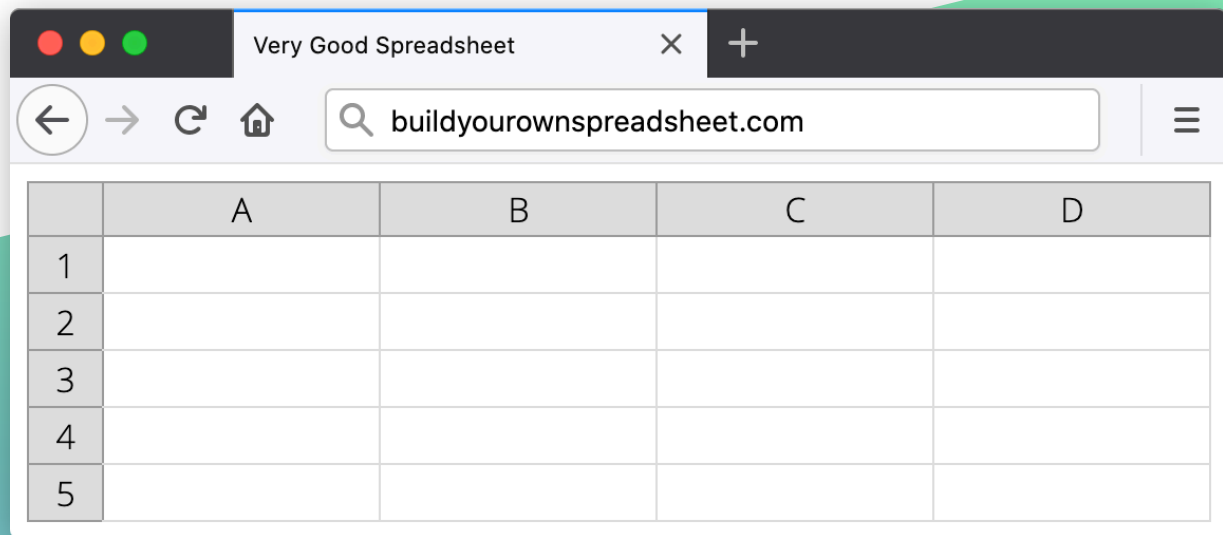


Build Your Own Spreadsheet

Example Chapters



A Text-Based Pairing Session Using Test-Driven Development

Contents

Colophon	2
Overview	3
Building a Spreadsheet	4
Requirements and Approach	4
Technical Setup	7
The First Test: Columns of a Spreadsheet	11
A Note on Purist TDD Practices	14
Getting it Green	15
Testing Rows	17
Testing Cells	18
Refactoring to Move Forward More Easily	19
Making it Pass	21
Recap	25

Colophon

Copyright © 2021 by Christoph Gockel

Names, characters and companies mentioned in this book are either the product of the author's imagination or are used fictitiously. Any resemblance to actual persons, living or dead, or companies is entirely coincidental.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including (photo)copying or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law. For permission requests, please contact byos@christophgockel.de.

All trademarks and copyrighted items mentioned and shown in pictures are the property of their respective owners.

<https://buildyourownspreadsheet.com>

Overview

Thank you for your interest in *Building Your Own Spreadsheet!*

The example content in this file contains the first few chapters of the full book. No changes have been made to the content you find here compared to what you get with the full version of the book. This overview chapter is the only page that is being added separately and is not included in the book itself. I want to give you as much context as possible to what the full book is like, so that you have enough details to make a decision whether you'd like to read more.

The introduction chapters found here include the discussion about what we're going to work on throughout the book. Followed by an overview of the technical setup. We won't spend too much time setting up our environment, but instead dive right into the code.

Have a read and please let me know what you think either via Twitter ([@ChristophGockel](#)) or E-Mail (byos@christophgockel.de). If you like what you're reading in these example chapters, you can head over to <https://buildyourownspreadsheet.com> and get the full version of the book!

Thank you and enjoy!

Christoph

Building a Spreadsheet

Okay, now that we have talked about *what* a spreadsheet is and what it contains, we're going to pair up to implement one ourselves! We'll start by discussing some expectations and constraints around what we're going to implement. Then we're going to set up the project. For an easier start we're going to begin our discussion using the starting template that came with the downloaded archive.

Once the initial set up is done and confirmed to be working, we're going to write the very first test for the spreadsheet, which we're then going to make pass right after. This cycle of discussing and writing a test followed by implementing the necessary changes is the theme throughout the whole book. No skipping details, no prepared code that has been written outside of what is being explained here, no cheating. We are in this together. There is, however, a safety net available in the form of the supporting code that comes with the book. At the end of every chapter the full source code of all created or modified files will be available in the book itself. Plus, the code archive that comes with the book contains code at all stages of the book. Each chapter will hint towards the appropriate subdirectory that contains the code relevant to that chapter.

Requirements and Approach

Here at Very Good Company we want to enable our users to explore their analytical data more easily. Our product managers had many conversations with customers and our sales and support teams that all tell the same story: our product would help them more by providing spreadsheet functionality in the application itself, instead of having to download a `.csv` file and then use it in another tool like Excel, Numbers or Sheets.

We from engineering sat together with our product manager Melissa to discuss an approach to make that happen. It's clear to them and us that we won't be able to support all the features people are used to from their spreadsheet applications. At least not in a reasonable amount of time. Luckily for us, this is also not required. We are not trying to gain market share here to become another spreadsheet vendor. We want to improve the lives of our users—at least when it comes to using our application.

Melissa has consolidated lots of feedback and requests from users and told us that having a way of viewing the data in a tabular form inside the application would already be a huge improvement over the current process users have to follow. The next topic she identified was calculations on that data. Users like to verify some assumptions by doing “napkin calculations” like adding and multiplying a couple of values from the spreadsheet next to the data itself. These calculations also ideally support functions like summing up a whole column or a range of values. Additionally supporting text values as well, in order to be able to concatenate a few words from different cells.

This is where we have left things for now. We agreed with Melissa that we would start working on a simple view of data in a spreadsheet-like user interface that also supports basic arithmetic. Supporting functions on the data is too much of an unknown for us at this point and we agreed to look at that again, once we have a working example of our spreadsheet.

So essentially we are going to provide a new spreadsheet-like user interface that will display data it's been given. Once we can display data, we are going to allow users to enter simple formulas into cells that work with the values that are being displayed.

Great! How do we start, though?

Luckily, our design team had a stab at this spreadsheet already and provided us with a handful of designs and mockups. This is what they came up with for the main user interface.

	A	B	C	D
1				
2				
3				
4				
5				

Figure 1: The user interface our spreadsheet should have.

Nothing too surprising here. They did mention, however, that it was unclear to them how many rows or columns are needed, so they only went with five rows and four columns. I don't have more details about that myself either at this point. Other existing spreadsheet applications seem to allow for almost infinite rows and columns. I think we'd make our lives difficult if we try to support that from the get-go. So I'd say for now, we are going to stick with five rows and four columns, too. Once we have a working solution for a spreadsheet of constrained sizes, it'll be easier to support more rows and columns eventually, instead of trying to support too much, too early.

We also got mockups for the different states a cell can be in. For example a highlighted cell should have a green background colour as shown in this mockup:

	A	B	C	D
1				
2		highlighted cell		
3				
4				
5				

Figure 2: Mockup showing a highlighted cell.

Cells that contain an error should be shown with a red border. When highlighting an erroneous cell we can also show a more detailed message about the reason of the error.

	A	B	C	D
1	#ERROR			
2		#ERROR		
3		A descriptive error message when highlighted		
4				
5				

Figure 3: Mockup showing cells with errors.

Okay, now that we've discussed what we are going to do, let's get started with the implementation of it!

Technical Setup

You can use your own project or setup to follow along the content of this book. As long as you have a test runner like Vitest, Jest, Jasmine, Mocha or similar available, the tests shown in the book should be able to run, as we're not using Vitest-specific features. Using Vitest will give you the highest chance of not running into issues, though. The other requirement is having TypeScript set up and React together with React Testing Library (including the `@testing-library/jest-dom` and `@testing-library/user-event` companion libraries) added to the project's dependencies.

To help you get started, however, the code subdirectory `00-technical-setup` contains the bare project setup with all necessary dependencies. It comes with Vite, Vitest, TypeScript, React, ESLint and Prettier already configured. Using the provided code will ensure the smoothest experience for you. The only tool you need to have installed already is Node in version 18.19.0 or higher. All code examples in this book assume you're following along with the provided project.

The initial code setup includes a `package.json` file that contains all necessary dependencies, as well as additional configuration for TypeScript, ESLint, Prettier and such. Feel free to explore the configuration of these tools. We're not spending much time in the book itself with that, but take for granted what is set up already. This is because the book tries to focus on the code we need to write and not how to configure Node or TypeScript.

The `package.json` file defines some additional scripts that we'll be using throughout the book.

- `start` runs a development server for us to see the spreadsheet in a browser
- `test` runs all tests
- `lint` runs ESLint across all source files

Prefixing any of these scripts with `npm run` will run them, e.g. `npm run start`. The `test` script can take an additional path to a specific test file in case we want to run only that one file instead of the whole suite: `npm run test path/to/test-file.ts`

Using the initial setup directory provided, we now need to run `npm install` first, to get all necessary dependencies installed. Once that is completed, we add a new file named `spreadsheet.test.tsx` in the `test` subdirectory as we're about to write our very first test for the spreadsheet!

The directory layout should look like this now (or similar if you're using your own project setup):

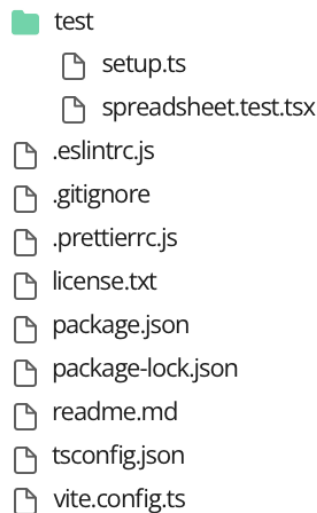


Figure 4: Directory layout with first test file.

Now, what is a good first test to write? I often start with something that is called a “pipe test” (or plumbing test) to make sure that the environment is properly set up and working as expected:

```
1 test("environment is working", () => {
2   expect(1).toEqual(2);
3 });
```

This test contains two things that we should explain a bit further before moving on. The first being that there is a function called `test` that we haven't defined anywhere. This function is provided by Vitest itself. It takes a description as the first argument and an anonymous function as the second, which contains the code to execute. When running the test file, Vitest will look for usages of the function `test` and execute them, as we will see in the next step. The other thing I want to explain is that this test does not contain a typo. I have deliberately set up this test to fail with an error that the number two does not equal the number one. I use a test like this to make sure that all testing tools are properly set up in the environment and can report errors. Running this with `npm run test` will show a test failure message:

```
> a-very-good-spreadsheet@0.1.0 test
> vitest

RUN v1.3.1 /Users/christoph/development/build-your-own-spreadsheet/code

> test/spreadsheet.test.tsx (1)
  × environment is working

Failed Tests 1

FAIL test/spreadsheet.test.tsx > environment is working
AssertionError: expected 1 to deeply equal 2

- Expected
+ Received

- 2
+ 1

> test/spreadsheet.test.tsx:2:13
  1| test("environment is working", () => {
  2|   expect(1).toEqual(2);
    |           ^
  3| });
  4|

[1/1]

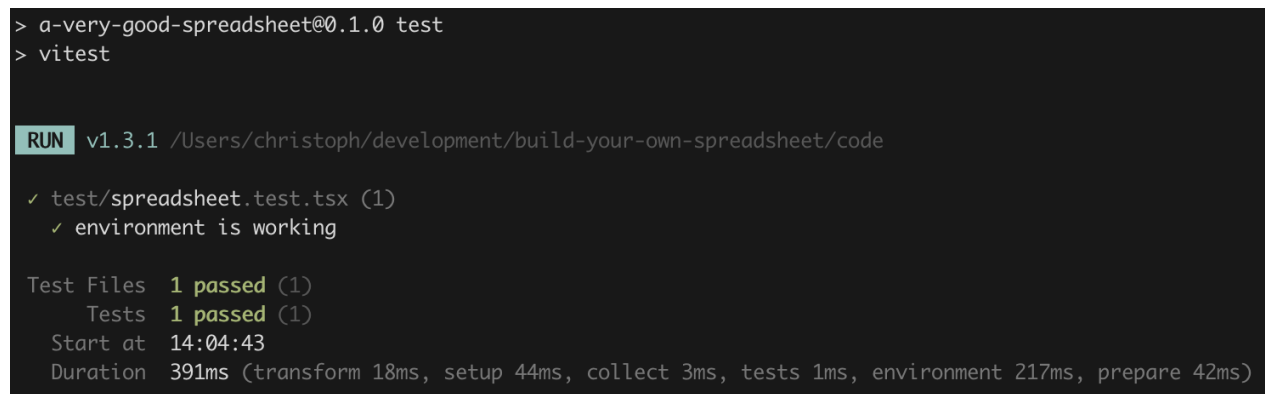
Test Files  1 failed (1)
Tests       1 failed (1)
Start at    14:03:54
Duration    391ms (transform 17ms, setup 42ms, collect 3ms, tests 4ms, environment 215ms, prepare 41ms)
```

Figure 5: The pipe test failure message.

Which is exactly the error we were expecting, great! Now we change the actual value to match the expected:

```
1 1 test("environment is working", () => {
2 -   expect(1).toEqual(2);
2 +   expect(1).toEqual(1);
3 3 });
```

Running the test again will then show a success message similar to the following:



```
> a-very-good-spreadsheet@0.1.0 test
> vitest

RUN v1.3.1 /Users/christoph/development/build-your-own-spreadsheet/code

✓ test/spreadsheet.test.tsx (1)
  ✓ environment is working

Test Files  1 passed (1)
Tests      1 passed (1)
Start at   14:04:43
Duration   391ms (transform 18ms, setup 44ms, collect 3ms, tests 1ms, environment 217ms, prepare 42ms)
```

Figure 6: Expected failure message.

We don't need to keep this test, though. It was only to gain confidence that our environment is working as expected.

```
1 -test("environment is working", () => {
2 -   expect(1).toEqual(1);
3 -});
```

This really is the core of testing tools like Vitest and similar. They provide functions for us to call, to which we pass executable pieces of code, which they then report on. There's nothing "magic" or special about testing tools like this. But they enable us to execute small pieces of *our* application code, so that we can verify that these small pieces work as expected. With the initial test done, let's think about what our first real test might look like.

The First Test: Columns of a Spreadsheet

Now that we know our environment is working, we're going to test-drive the whole implementation of our spreadsheet. There's not a whole lot going on implementing the static aspects of the user interface, but nevertheless we can write tests for it.

Looking once again at the mockups from earlier, we can see columns with headers, rows with headers and cells. All of these parts can be verified in a test. Keeping the UI we want in mind we can write a test that the component we're about to implement contains at least the expected columns.

That seems worthwhile to verify. Let's capture this in a test.

```
1 describe("Spreadsheet", () => {
2   it("consists of columns labelled with a letter", () => {
3     render(<Spreadsheet />);
4
5     expect(screen.getByText("A")).toBeInTheDocument();
6     expect(screen.getByText("B")).toBeInTheDocument();
7     expect(screen.getByText("C")).toBeInTheDocument();
8     expect(screen.getByText("D")).toBeInTheDocument();
9   });
10 });
```

Here, we're using Vitest's `describe` and `it` functions in order to group our tests together. The `it` function is very similar to the single `test` function we've seen earlier. I prefer grouping related tests together rather than having seemingly unrelated `test()` functions scattered throughout the test file. Every `it` function inside a `describe` will test behaviour of the "thing" named in the `describe` function. This distinction works purely on a semantic level, however. It helps us and future readers of the code to group together tests for certain behaviour.

There are two or three more things we should discuss before trying to make the test green. The test uses `render` and `screen` from *React Testing Library*, and it's trying to render a `Spreadsheet` component. The `render` function takes a React component and will render it for us, so that we can assert on certain things and behaviours of that component. React Testing Library's `screen` is an abstraction that represents the browser window or the *screen* an end user sees. One of React Testing Library's benefits is that it interacts with our app the same way a user does. There's no way to look

behind the scenes and verify some internal state of a component as there is with other testing tools in the React testing space. This has the benefit that our tests will focus on *observable behaviour* instead of verifying implementation details.

An expectation like `expect(screen.getByText("A")).toBeInTheDocument();` essentially says that there has to be the letter "A" *somewhere* on the screen. Which is somewhat of a broad assumption as it would be okay for the test if the HTML contains something like this:

```
<h1>A Very Good Spreadsheet</h1>
```

But we don't have any other content on the screen or in the document, so it is okay for us to rely on this expectation. The argument to `expect` is often referred to as the *code under test*, or the result of the code under test as in our example. Then on the return value of `expect` we can execute so-called *matchers*. We've already seen the `toEqual` matcher in the previous example, where we expected one number to equal another number. In this test, however, we now want to verify that a specific string is rendered in the document.

In order to make that test pass we need to update our imports like this:

```
1 +import * as React from "react";
2 +import { render, screen } from "@testing-library/react";
3 +
4 describe("Spreadsheet", () => {
5   it("consists of columns labelled with a letter", () => {
6     render(<Spreadsheet />);
```

Then, the next step is to create the `Spreadsheet` component. Eventually this component should live in a file called `spreadsheet.tsx` inside the `src` directory. To keep focussing on the test at hand, we can take a shortcut and create the component directly in the test file itself. This will save some time until we get the test green.

```
1 1 import * as React from "react";
2 2 import { render, screen } from "@testing-library/react";
3 3
4 +function Spreadsheet(): React.ReactElement {
5 +  return <table />;
6 +}
7 +
```

```

4 8 describe("Spreadsheet", () => {
5 9   it("consists of columns labelled with a letter", () => {
6 10     render(<Spreadsheet />);

```

Adding the component anywhere inside the test file will work. Running the test again should result in a failure with the message that the expected column "A" can't be found.

```

> a-very-good-spreadsheet@0.1.0 test
> vitest

RUN v1.3.1 /Users/christoph/development/build-your-own-spreadsheet/code

> test/spreadsheet.test.tsx (1)
  > Spreadsheet (1)
    × consists of columns labelled with a letter

Failed Tests 1

FAIL test/spreadsheet.test.tsx > Spreadsheet > consists of columns labelled with a letter
TestingLibraryElementError: Unable to find an element with the text: A. This could be because the text is broken up by multiple elements. In this case, you can provide a function for your text matcher to make your matcher more flexible.

<body>
  <div>
    <table />
  </div>
</body>
> Object.getElementError node_modules/@testing-library/dom/dist/config.js:37:19
> node_modules/@testing-library/dom/dist/query-helpers.js:90:38
> node_modules/@testing-library/dom/dist/query-helpers.js:62:17
> node_modules/@testing-library/dom/dist/query-helpers.js:111:19
> test/spreadsheet.test.tsx:12:19
   10|     render(<Spreadsheet />);
   11|
   12|     expect(screen.getByText("A")).toBeInTheDocument();
      |                       ^
   13|     expect(screen.getByText("B")).toBeInTheDocument();
   14|     expect(screen.getByText("C")).toBeInTheDocument();

[1/1]

Test Files  1 failed (1)
Tests       1 failed (1)
Start at    15:16:38
Duration    489ms (transform 19ms, setup 42ms, collect 85ms, tests 14ms, environment 219ms, prepare 38ms)

```

Figure 7: Expected failure message.

Great! This is exactly the error we were expecting. So let's make it green!

A Note on Purist TDD Practices

One thing worth mentioning is that TDD purists might correctly frown upon starting with a test as big as the one above.

The more correct (or strict) way would have been to start smaller and verify the existence of such a thing like a `Spreadsheet` component. I think it's okay to not get hung up on these details once you've written tests like that a couple of times and know *why* you're skipping it.

We *know* we want a spreadsheet component that renders a set of columns. So why not start verifying it? The alternative could have been to start writing something like:

```
1 describe("Spreadsheet", () => {
2   it("exists", () => {
3     render(<Spreadsheet />);
4   });
5 });
```

This test, however, neither verifies any behaviour nor does it contain any meaningful expectation.

It only forces us to create the `Spreadsheet` component. By *forcing us* I mean that the test will—correctly—fail, mentioning that the identifier `Spreadsheet` is unknown. Which is a valid test failure, but it feels artificial and forced. If you've written a test like that once or twice I think it is okay to jump ahead and combine it with the test you *actually* want to write.

Using TDD to write code encourages us to develop in small steps. This is very powerful and I don't want to discourage us from doing that. There is, however, a threshold of value or benefits provided by tests. Sometimes tests that are too small don't provide the most benefit to us. On the other hand, tests that are too large might become too brittle too quickly. This *threshold* is, like many other things in software development, a trade-off that we need to continue to consider in order to get the most out of the tests we write.

Getting it Green

With the first test failing for the right reason, we can now focus on getting it to pass. What's the simplest way to do that? Maybe it's good enough to hard code the four columns that are expected.

Let's try that by changing the `Spreadsheet` component as follows:

```
2 2   import { render, screen } from "@testing-library/react";
3 3
4 4   function Spreadsheet(): React.ReactElement {
5 5   -   return <table />;
6 6   +   return (
7 7     +   <table>
8 8       +   <thead>
9 9         +   <tr>
10 10          +   <th aria-label="empty header" />
11 11          +   <th>A</th>
12 12          +   <th>B</th>
13 13          +   <th>C</th>
14 14          +   <th>D</th>
15 15        +   </tr>
16 16      +   </thead>
17 17    +   </table>
18 18  +   );
19 19  }
20 20  describe("Spreadsheet", () => {
```

The spreadsheet itself is built with a table and we're using semantically correct header definitions. Let's see what our test says.

```
> a-very-good-spreadsheet@0.1.0 test
> vitest

RUN v1.3.1 /Users/christoph/development/build-your-own-spreadsheet/code
✓ test/spreadsheet.test.tsx (1)
  ✓ Spreadsheet (1)
    ✓ consists of columns labelled with a letter

Test Files  1 passed (1)
Tests       1 passed (1)
Start at    15:17:55
Duration    498ms (transform 21ms, setup 44ms, collect 87ms, tests 15ms, environment 217ms, prepare 38ms)
```

Figure 8: First test is passing!

It's passing! Well done, us!

Now, with the test passing, it is usually the time for refactorings or other improvements. I think it's too early to tell if and what we could improve in the implementation so far, but it's a good time to reorganise the code a bit.

Instead of continuing the implementation inside the test file, let's create a `src` directory and create a file called `spreadsheet.tsx` inside. Then we move the `Spreadsheet` component from the test file to the newly created file inside `src`.

```
1 +import * as React from "react";
2 +
3 +export function Spreadsheet(): React.ReactElement {
4 +  return (
5 +    <table>
6 +      <thead>
7 +        <tr>
8 +          <th aria-label="empty header" />
9 +          <th>A</th>
10 +          <th>B</th>
11 +          <th>C</th>
12 +          <th>D</th>
13 +        </tr>
14 +      </thead>
15 +    </table>
16 +  );
17 +}
```

Once moved, the test file needs to correctly import the component now.

```
1 1 import * as React from "react";
2 2 import { render, screen } from "@testing-library/react";
3 -
4 -function Spreadsheet(): React.ReactElement {
5 -  return (
6 -    <table>
7 -      <thead>
8 -        <tr>
9 -          <th aria-label="empty header" />
10 -          <th>A</th>
11 -          <th>B</th>
12 -          <th>C</th>
13 -          <th>D</th>
14 -        </tr>
15 -      </thead>
16 -    </table>
17 -  );
```

```

18 -}
19 3 +import { Spreadsheet } from "../src/spreadsheet";
20 4
21 5 describe("Spreadsheet", () => {
22 6     it("consists of columns labelled with a letter", () => {

```

After that extraction, let's make sure everything is still working by running the test again. There should be no errors being reported.

This concludes our first test and we can continue thinking about the next one to write.

Testing Rows

What's a good next test to write? From the initial user interface we've seen earlier we still have to verify that our spreadsheet also contains rows and cells.

Let's start by verifying it contains rows.

```

14
15 it("consists of five numbered rows", () => {
16     render(<Spreadsheet />);
17
18     expect(screen.getByText("1")).toBeInTheDocument();
19     expect(screen.getByText("2")).toBeInTheDocument();
20     expect(screen.getByText("3")).toBeInTheDocument();
21     expect(screen.getByText("4")).toBeInTheDocument();
22     expect(screen.getByText("5")).toBeInTheDocument();
23 });

```

Running this test will fail with an error message like the following:

```

TestingLibraryElementError: Unable to find an element with the text: 1.
This could be because the text is broken up by multiple elements.
In this case, you can provide a function for your text matcher to make your matcher
more flexible.

```

There's no element with the value of "1" in our document. That's correct, so let's add it!

```

12 12         <th>D</th>
13 13     </tr>
14 14 </thead>
15 15 + <tbody>

```

```

16 +     <tr>
17 +       <th scope="row">1</th>
18 +     </tr>
19 +     <tr>
20 +       <th scope="row">2</th>
21 +     </tr>
22 +     <tr>
23 +       <th scope="row">3</th>
24 +     </tr>
25 +     <tr>
26 +       <th scope="row">4</th>
27 +     </tr>
28 +     <tr>
29 +       <th scope="row">5</th>
30 +     </tr>
31 +   </tbody>
15 32 </table>
16 33   );
17 34 }

```

For now, we're hardcoding the rows again. Running the test will show that this will make it pass, great!

Testing Cells

Currently each row only has its header cell to denote its number. Another test we're going to write is to verify that our spreadsheet contains cells, too.

```

24
25   it("consists of cells", () => {
26     render(<Spreadsheet />);
27
28     expect(screen.getByTestId("A1")).toBeInTheDocument();
29     expect(screen.getByTestId("D5")).toBeInTheDocument();
30   });

```

This test introduces a new query function of React Testing Library: `getByTestId`. It allows us to assert on non-visible parts of our user interface. Since we don't have any content to display in the cells (yet), we can add a custom data attribute to an element in order to verify its existence. The custom attribute is called `data-testid`. Using `getByTestId` is something we should only do sparingly, because it breaks React Testing Library's original intent to *test an application the way users interact with it*. A user cannot see the data attribute of a DOM node, so we're entering a grey area between observable behaviour and implementation details. It's not bad to use `getByTestId`,

but it shouldn't always be the first query-function to reach for when trying to get hold of elements.

Instead of asserting on every cell individually, the test asserts only on the top-left cell and the bottom-right one. This assumes that the implementation will work in the same way across the other cells. It's a trade-off for not over-specifying the test. If we're not facetious in the implementation of `Spreadsheet` by deliberately leaving out some cells, this test provides the appropriate amount of safety and assurance while not being too overly specific.

Running the test we will see a failure containing a message like this:

```
TestingLibraryElementError: Unable to find an element by: [data-testid="A1"]
```

Now, how can we make this test pass? So far we've hard coded the rows in our implementation and we could continue doing so for the cells. But it would be quite repetitive and harder to work with in the long run. We could, however, refactor our current implementation to be more dynamic to help implement the cells more easily.

For that, let's hold off with the current test and skip it by changing the test function from `it` to `it.skip`.

```
22 22     expect(screen.getByText("5")).toBeInTheDocument();
23 23   });
24 24
25 25 - it("consists of cells", () => {
26 26 + it.skip("consists of cells", () => {
27 27     render(<Spreadsheet />);
28 28     expect(screen.getByTestId("A1")).toBeInTheDocument();
```

We're going to refactor our production code first, then come back to this test.

Refactoring to Move Forward More Easily

What we're doing now happens frequently while test-driving code. By adding a new test, tensions and shortcomings in the code become more visible. Instead of trying to refactor the code while making a new test pass it's good practice to hold off on the new test and requirements and improve the structure or design of the current code so that

the new behaviour can be added more easily. The improvements we're introducing have a goal in mind: to support the new behaviour in a better or easier manner.

Kent Beck has captured this approach quite well:

for each desired change, make the change easy (warning: this may be hard), then make the easy change

Kent Beck, on Twitter in 2012

And this is exactly what we're going to do now. With two tests passing and one test skipped, let's have a look at the production code. It contains quite a bit of structural duplication. Each column and each row is written out literally. We could introduce loops to reduce some of that duplication.

We can start by refactoring the table body to use loops instead of hard coded rows. Since we know we're going to focus on only five rows for now, we introduce a constant for that:

```
1 1  import * as React from "react";
2 2
3 3  +const NUMBER_OF_ROWS = 5;
4 4  +
5 5  export function Spreadsheet(): React.ReactElement {
6 6  return (
7 7  <table>
```

The individual rows inside the `tbody` can be replaced with a loop like this:

```
15 15     </tr>
16 16     </thead>
17 17     <tbody>
18 18     -     <tr>
19 19     -         <th scope="row">1</th>
20 20     -     </tr>
21 21     -     <tr>
22 22     -         <th scope="row">2</th>
23 23     -     </tr>
24 24     -     <tr>
25 25     -         <th scope="row">3</th>
26 26     -     </tr>
27 27     -     <tr>
28 28     -         <th scope="row">4</th>
```

```

29 -     </tr>
30 -     <tr>
31 -       <th scope="row">5</th>
32 -     </tr>
18 +     {...Array(NUMBER_OF_ROWS)].map((_, rowIndex) => (
19 +       <tr key={rowIndex.toString()}>
20 +         <th scope="row">{rowIndex + 1}</th>
21 +       </tr>
22 +     ))}
33 23   </tbody>
34 24   </table>
35 25   );

```

The anonymous array `[...Array(NUMBER_OF_ROWS)]` is a way to allow for `n` iterations in JavaScript/TypeScript instead of having to write a `for` or `while` loop. This will keep the current tests passing and enables us to add more `td` elements for the spreadsheet's cells more easily! So let's get back to the skipped test.

Making it Pass

With the refactored code in place, let's unskip the third test and change the `it.skip` back to an `it`.

```

22 22     expect(screen.getByText("5")).toBeInTheDocument();
23 23   });
24 24
25 -   it.skip("consists of cells", () => {
25 +   it("consists of cells", () => {
26 26     render(<Spreadsheet />);
27 27
28 28     expect(screen.getByTestId("A1")).toBeInTheDocument();

```

Running the test again will fail for the same reason it did before – it cannot find the expected cell. But now we can reap the benefits of our previous refactoring and add a table cell to the JSX markup.

```

18 18     {...Array(NUMBER_OF_ROWS)].map((_, rowIndex) => (
19 19       <tr key={rowIndex.toString()}>
20 20         <th scope="row">{rowIndex + 1}</th>
21 +         <td data-testid={`A${rowIndex + 1}`} />
21 22       </tr>
22 23     ))}
23 24   </tbody>

```

Running the tests again will still fail, but if you look closely it fails for a different reason now.

```
TestingLibraryElementError: Unable to find an element by: [data-testid="D5"]
```

That's a good thing! Even if we can't immediately solve the current test failure, at least we should be able to change the failure message to give us feedback if we're moving in the right direction. And we are moving in the right direction as the test fails now because there is no cell in the fourth column. This is different to the previous test failure where it couldn't find the first column.

We can make this pass reasonably quickly, by adding cells for the remaining columns:

```
19 19         <tr key={rowIndex.toString()}>
20 20             <th scope="row">{rowIndex + 1}</th>
21 21             <td data-testid={`A${rowIndex + 1}`} />
+          <td data-testid={`B${rowIndex + 1}`} />
+          <td data-testid={`C${rowIndex + 1}`} />
24 +          <td data-testid={`D${rowIndex + 1}`} />
22 25         </tr>
23 26     )})
24 27 </tbody>
```

This will get our third test green!

```
> a-very-good-spreadsheet@0.1.0 test
> vitest

RUN v1.3.1 /Users/christoph/development/build-your-own-spreadsheet/code

✓ test/spreadsheet.test.tsx (3)
  ✓ Spreadsheet (3)
    ✓ consists of columns labelled with a letter
    ✓ consists of five numbered rows
    ✓ consists of cells

Test Files  1 passed (1)
Tests       3 passed (3)
Start at    15:19:40
Duration    508ms (transform 25ms, setup 41ms, collect 93ms, tests 28ms, environment 216ms, prepare 41ms)
```

Figure 9: Third test is passing!

With all our tests passing, we have the opportunity again to clean up code or refactor

parts we'd like to improve. I think we could improve some repetitiveness around the column handling inside the table body and header.

For that, let's define an array of the columns we want in the spreadsheet to appear.

```
1 1 import * as React from "react";
2 2
3 +const COLUMNS = ["A", "B", "C", "D"];
4 const NUMBER_OF_ROWS = 5;
5 5
6 export function Spreadsheet(): React.ReactElement {
```

Then, instead of our current column definition, we can iterate through this array to create the `th` elements.

```
9 9 <thead>
10 10 <tr>
11 11 <th aria-label="empty header" />
12 - <th>A</th>
13 - <th>B</th>
14 - <th>C</th>
15 - <th>D</th>
12 + {COLUMNS.map((letter) => (
13 + <th key={letter}>{letter}</th>
14 + ))}
16 15 </tr>
17 16 </thead>
18 17 <tbody>
```

Rerunning the tests after every refactoring step keeps us from making mistakes and not realising that something unrelated has changed or broken the test expectations. We can continue using the new `COLUMNS` array for the cells, too.

```
18 18 {[...Array(NUMBER_OF_ROWS)].map((_, rowIndex) => (
19 19 <tr key={rowIndex.toString()}>
20 20 <th scope="row">{rowIndex + 1}</th>
21 - <td data-testid={`A${rowIndex + 1}`} />
22 - <td data-testid={`B${rowIndex + 1}`} />
23 - <td data-testid={`C${rowIndex + 1}`} />
24 - <td data-testid={`D${rowIndex + 1}`} />
21 + {COLUMNS.map((columnLetter) => (
22 + <td
23 + key={columnLetter}
24 + data-testid={`${columnLetter}${rowIndex + 1}`}
25 + />
26 + ))}
```

```

25 27         </tr>
26 28     )})
27 29 </tbody>

```

After this change we verify again that our tests still pass.

The `spreadsheet.tsx` module now looks like this:

```

1  import * as React from "react";
2
3  const COLUMNS = ["A", "B", "C", "D"];
4  const NUMBER_OF_ROWS = 5;
5
6  export function Spreadsheet(): React.ReactElement {
7      return (
8          <table>
9              <thead>
10                 <tr>
11                     <th aria-label="empty header" />
12                     {COLUMNS.map((letter) => (
13                         <th key={letter}>{letter}</th>
14                     ))}
15                 </tr>
16             </thead>
17             <tbody>
18                 {[...Array(NUMBER_OF_ROWS)].map((_, rowIndex) => (
19                     <tr key={rowIndex.toString()}>
20                         <th scope="row">{rowIndex + 1}</th>
21                         {COLUMNS.map((columnLetter) => (
22                             <td
23                                 key={columnLetter}
24                                 data-testid={`-${columnLetter}-${rowIndex + 1}`}>
25                             </td>
26                         ))}
27                     </tr>
28                 ))}
29             </tbody>
30         </table>
31     );
32 }

```

There's less duplication than before but also more complex logic with three loops. The current implementation will enable our future selves to extend the columns or rows with minimal effort and no duplication. When comparing that against the original implementation, which hard coded the four columns, I think it was a worthwhile refactoring.

Overall it's up to us and our fellow developers which refactorings are worthwhile and,

more importantly, how much refactoring is *enough*. “Enough” in this instance is also context specific. Maybe there’s deadline pressure or other parts of our project are more important at the time than four duplicated columns. It is okay to hold off or not do any of these refactorings at all. If the component never needs updating or to support more than four columns, the original approach of four hard coded columns can be good enough. We shouldn’t let our personal opinion of code aesthetics influence the complexity of the code we write.

As the saying goes: *done is better than perfect*.

The code up to this point is available in subdirectory `01-testing-cells` of the code archive that came with the book.

Recap

What we’ve seen and experienced up to this point is the basic flow of test-driven development. We start with a failing test defining the expectations regarding the code we’re about to write, followed by *just enough* implementation to make this one test pass and then we go back to writing another test for the next part of the code we’d like to see.

A common occurrence during this cycle is the temporary skipping of a new test when we see opportunities to refactor some of our code in order to implement the new behaviour in an easier way. Once the refactoring is finished, we unskip the last test and can then focus on making it pass—which is usually easier now because of the improvements we made. Skipping the test is important because we do not want to implement new behaviour *and* refactor code at the same time. If for example the test didn’t pass after our changes, we can’t know for certain whether it still fails because of the new logic we implemented or because of the refactoring we applied. By having a new test skipped, we can then focus on only the refactoring without having to think of improving code *and* implement new behaviour.

When working on getting a test to green, I always ask myself “*what’s the least amount of code I can get away with?*” This is also sometimes referred to as “TDD Golf”. Hard coding things helps to keep our code simpler. Instead of immediately reaching for a more complex or clever solution I prefer to wait until there is definite a need for that complex logic to exist. For example, I wouldn’t mind continuing to have four hard coded

columns A, B, C and D in the production code as we didn't have a need or requirement to support more columns. But another test exposed a tension with this approach, so we refactored it to a loop—still with only four columns, but a little bit more dynamic than it was before.

This concludes the initial setup and implementation of our spreadsheet. We have tested the structure of the user interface, but we haven't really paid much attention to the design and layout of the user interface yet. This will be the next step.