How the Page Object Model and Promises Work Together

Hal Deranek

Solution Architect, Slalom_build

deranek@gmail.com

Abstract

The Page Object Model is one of the bedrocks of modern website test automation. The model, based on Object Oriented Programming concepts, helps a test automation engineer break website pages down from large dynamic sets of elements into sections and patterns. Rather than having to account for 100+ elements on a webpage, we can account for the discernible sections and patterns. Place the elements within those sections and patterns. The POM is easier to implement, understand, and maintain.

This all worked well when page content wasn't terribly dynamic. With the recent rise of asynchronous platforms like React and Angular, test automation has become more difficult because of the one word that strikes fear in the hearts of most automation engineers: PROMISES! Perhaps that's hyperbolic, but I have seen enough testers struggle to work with Promises to know that it's a large problem. Fortunately, I've been able to understand Promises, when they are used, and how to resolve them within a test automation suite.

My presentation will focus on these points - describing the Page Object Model and its benefits, explaining what Promises are and how they work, and best practices around using the Page Object Model when testing against an asynchronous platform. I will provide context and examples that the attendees will be able to understand and reproduce. I will also be sure to move around, speak with energy and verve, and interact with the attendees so that everyone has a good time while learning.

Bio

Hal Deranek has worked in testing for over a decade at companies like Tableau, Slalom, and Blue Nile. He started with writing test automation suites for websites, mobile apps, and other platforms. He has grown to become an expert in test strategy and organization. Hal has led several testing teams in various projects across disparate industries. He enjoys mentoring burgeoning testers and helping them reach their potential. Finally, Hal enjoys being part of the testing community in his home city of Seattle by attending, facilitating, and presenting at various Meetup groups.

Introduction

The Page Object Model (POM) is one of the bedrocks of modern website test automation. Rather than having to account for 100+ elements on a webpage, we can account for them using discernible sections and patterns. The POM is easier to implement, understand, and maintain than code without it.

This all worked well when page content wasn't terribly dynamic. With the rise of asynchronous platforms like React and Angular, test automation has become more difficult because of the one word that strikes fear in the hearts of most automation engineers: PROMISES! Perhaps that's hyperbolic, but I have seen enough testers struggle to work with Promises to know that it's a large problem. Fortunately, I have come to understand Promises, when to use them, and how to resolve them within a test automation suite.

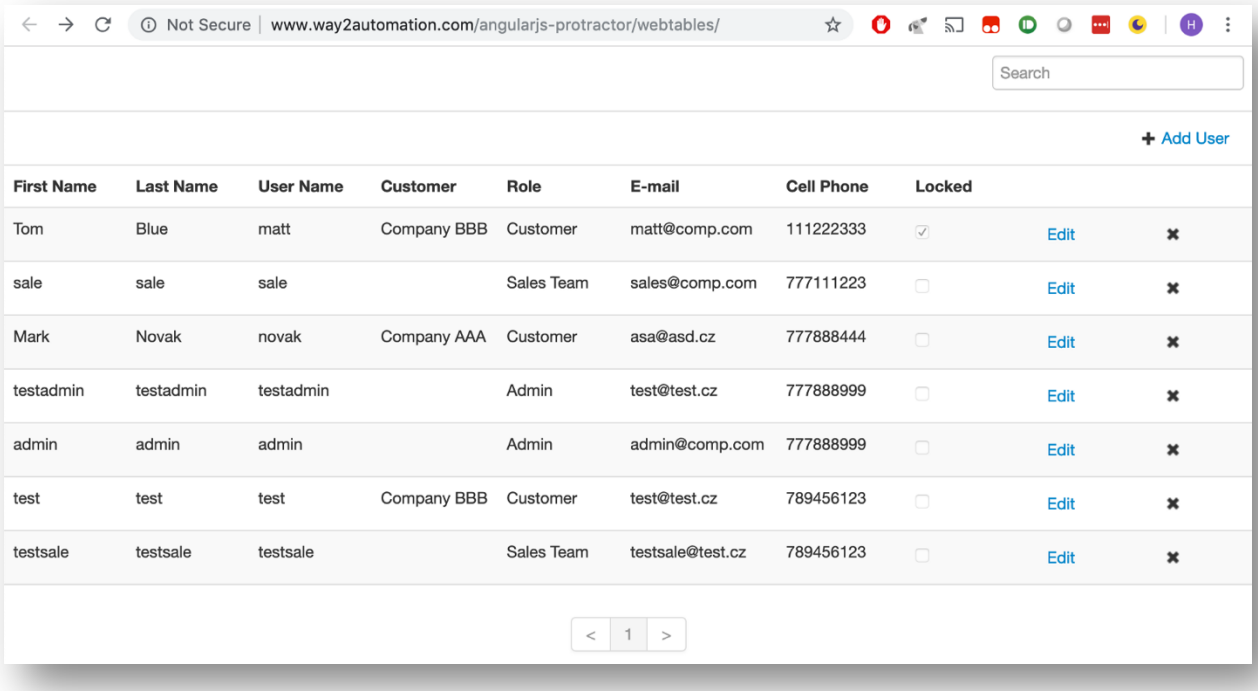
By the end of this paper, I will go through the following points:

- Describing the Page Object Model and its benefits
- Explaining what promises are and how they work
- Best practices for using the POM when testing an asynchronous platform

Explaining the Page Object Model

"A page object is an object-oriented class that serves as an interface to a page of your [application under test]. The tests then use the methods of this page object class whenever they need to interact with the UI of that page. The benefit is that if the UI changes for the page, the tests themselves don't need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place." — Seleniumhq.org

There are different methods of creating automated tests. The most obvious is to take a page and define everything in one file/class. As an example, take a look at this webpage:

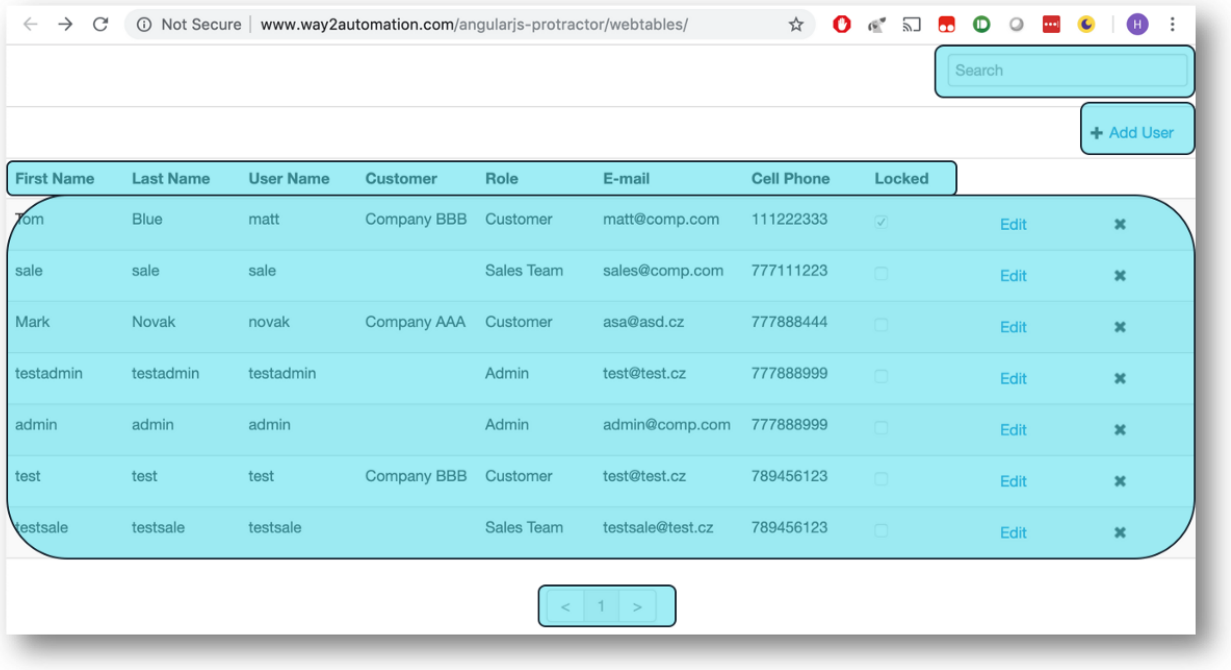


http://www.way2automation.com/angularjs-protractor/webtables/

How many different visible elements do you think are on this page? Counting the search input box, each user name, each Edit button, etc., there are 84 elements visible on the page. You could include them all in one large class if this page never changed, however this is a dynamic website. You can add, remove, or edit records, meaning the page will change and you can never define it in automation by declaring each object. There must be a better way!

There is! The Page Object Model is ideal for situations such as these. Rather than seeing the page as singular elements, you can look at it more as exploitable sections and patterns. The sections and patterns you define will contain the objects.

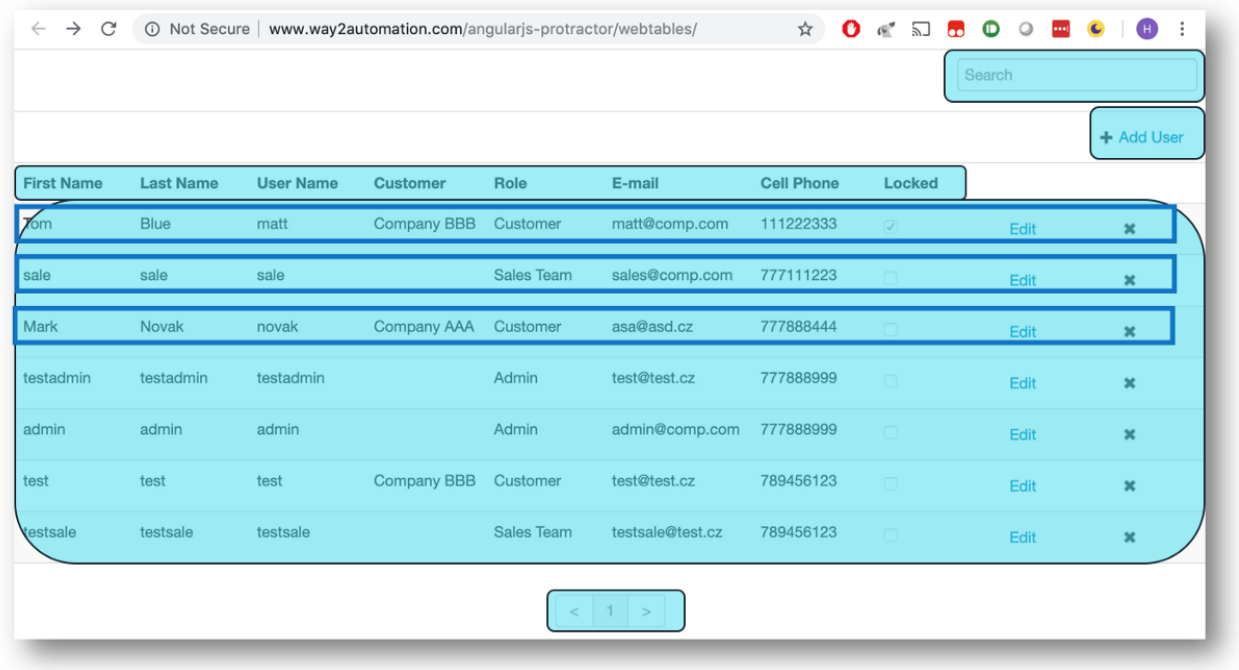
As an example, let's take a look at the website again. This time, let's break it down by sections…



Looking at the page, there are five sections we can define:

- Search
- Add User
- Table Headers
- Table Records
- Pagination

This is a great start but the Table Records section still has a vast amount of elements. Also, the number of records is likely to fluctuate at any time. There's a distinctive pattern that emerges when you look at the records themselves: the rows.



If you create a class to represent this pattern, you can grab each row's container and let the pattern handle the rest.

Code Examples

Using the section examples above, we can start coding our classes. Let's start with the Dashboard Page. We have five sections which will need their own classes. Here's what the DashboardPage class would look like:

```
export class DashboardPage {
    constructor() {}

    public get searchBar(): SearchBarSection {
        return new SearchBarSection(element(by.css('td.global-search')));
    }

    public get addUserButton(): AddUserSection {
        return new AddUserSection(element(by.css('td.actions-add-url')));
    }

    public get userTableTitles(): UserTableTitles {
        return new UserTableTitles(element(by.css('tr.table-header-row')));
    }

    public get userTableRecords(): UserTableRecords {
        return new UserTableRecords(element(by.css('tbody')));
    }

    public get userTablePagination(): Pagination {
        return new Pagination(element(by.css('.pagination')));
    }
}
```

The code above includes all five previously identified sections. One thing to note is that each returned class instance has an element passed in. This is the container. You use the container to ensure you're only looking for elements within the correct section. This is especially important for patterns like the records rows.

```
export class SearchBarSection {
    constructor(private container: ElementFinder) {}

    public get inputBox(): ElementFinder {
        return container.element(by.css('input'));
    }
}
```

Here's the class for the first section seen on the page—SearchBarSection. Since there's only one element currently in the section—an inputBox—it's not terribly complicated. Note how we search for the 'input' tag after the container. This ensures we only look for an 'input' tag within the passed-in container.

To see something more complicated, let's think about the records table. Before we work on the class of the table itself, let's think about the pattern of the record rows:

```
export class UserTableRecord {
    constructor(private container: ElementFinder) {}

    public get firstName(): ElementFinder {
        return container.element(by.css('td.firstName')));
    }

    public get lastName(): ElementFinder {
        return container.element(by.css('td.lastName')));
    }

    public get userName(): ElementFinder {
        return container.element(by.css('td.userName')));
    }

    public get customer(): ElementFinder {
        return container.element(by.css('td.customer')));
    }

    public get role(): ElementFinder {
        return container.element(by.css('td.role')));
    }
    ...
```

This class defines each element with a basic pattern, including using the container. Now that we've got the row pattern, we can start thinking about finding all the records:

```
export class UserTableRecords {
    constructor(private container: ElementFinder) {}

    public async getUserRecords(): Promise<UserRecord[]> {
        const records: UserRecord[] = [];
        for(const record of (await container.element.all(by.css('tr')))) {
            records.push(new UserRecord(record);
        }
        return records;
    }
}
```

Here we find the user rows by this code:

```
await container.element.all(by.css('tr'))
```

This will find all 'tr' tags within the scope of the container. The next line will push a new instance of a UserRecord (as defined in the previous code example) into an array. Finally, the method returns all found row elements.

…

The examples above show how you can use the POM to help with test automation – find basic patterns in the page and build them into sections. However, as previously stated, interacting with these page elements can be difficult when working with asynchronous frameworks, especially when you may have to nest these elements. Let's take a moment to look at how to work with asynchronous frameworks. Let's take a moment to discuss promises!

Explaining Promises

"A Promise is an asynchronous operation. It's called a "Promise" because we are promised a result at a future time." - https://toddmotto.com/promises-angular-q

Let's talk about promises. To translate the statement above, think of a promise as a request to perform an action. Once you've identified an element on a page, you'll want to perform an action on it – click it, get its text, determine if it's displayed, etc. These actions are promises. (Note: finding a single element itself does not return a promise. Finding multiple elements will invoke a promise.)

Why are Promises difficult?

Asynchronous frameworks don't wait for actions to complete before proceeding to the next action. This allows for quick and dynamic websites. For example, let's say a page has 30 images to load. Traditional websites wait for the images to load in a linear fashion. An asynchronous site will ask for those images all at once and display them as they are available.

Unfortunately, automated tests almost always expect actions to occur in a linear fashion. You're going to run into problems if your test asks for an action (a promise) without accounting for the asynchronous nature of the framework. You must resolve the promises on one action before continuing on to the next.

Let's look at an example of this. In this test, I want to determine if an element is present on the webpage. I start by declaring the element in the first line and then asking to print text to the console on the second line:

```
let newElement = element(by.css('div'));
console.log(`Is the element present? ${newElement.isPresent()}`);
```

Here is the output I expect to see:

```
Is the element present? true
```

Here is the output I will actually see:

```
Is the element present? ManagedPromise::428 {[[PromiseStatus]]: "pending"}
```

What happened? Instead of "true", there's a bunch of nonsense about promises. What's going on?

Resolving Promises – the "await" keyword

What's going on is that I didn't resolve the promises in my code. "isPresent" is an action that I performed on the element ("newElement"). Because the framework is asynchronous, the console.log event executes before the "isPresent" promise resolves. This is why we see that the promise is "pending" – it hasn't completed the action when we asked for its output. Fortunately, there's a way to resolve the promise before moving on. Let's see how…

Here's the code I wrote before:

```
let newElement = element(by.css('div'));
console.log(`Is the element present? ${newElement.isPresent()}`);
```

To make the promise to finish executing before I proceed, I use the "await" command like so:

```
console.log(`Is the element present? ${await newElement.isPresent()}`);
```

Notice the “await” keyword before the “isPresent” action on the “newElement” element. This is telling the framework to wait for the "isPresent" action to complete before continuing on. Once "isPresent" finishes, the console.log method will execute. There’s the output we now see:

```
Is the element present? true
```

Best practices for implementing the Page Object Model & in an asynchronous framework

Now that we know how to force promises to resolve on our command, let’s look at how to implement this within a framework.

Using the “get” syntax

```
export class UserTableRecord {
    constructor(private container: ElementFinder) {}

    public get firstName(): ElementFinder {
        return container.element(by.css('td.firstName'));
    }

    public get lastName(): ElementFinder {
        return container.element(by.css('td.lastName'));
    }

    public get userName(): ElementFinder {
        return container.element(by.css('td.userName'));
    }

    public async getFullName(): Promise<string> {
        return `${await this.firstName.getText()} ${await this.lastName.getText()}`;
    }
...
```

Here is an example of using the ‘get’ syntax. You’ll notice that the async/await keywords aren’t used in these methods since they are only identifying the element they’re returning. Actions are not performed so no asynchronous calls (promises) are made. Another note on using the "async" keyword on "get" methods: it’s not allowed when declaring the method, though you can return a promise (see the next section).

Looking at the “getFullName” method, though, we see the keywords appear. The “async” keyword appears before the method name to declare it as asynchronous. We use “await” before the two “getText” actions to ensure they resolve before the method returns the string.

A note on constructors: by design, constructors cannot execute complex procedures. This means you cannot resolve a promise within the context of the constructor. Having the “get” methods find and return a new instance of the element each time they're called ensures you do not run into difficulties with stale elements such as when a page’s data dynamically refreshes.

Returning promises from a “get”

There are times where a “get” statement requires either returning a promise or doing some error checking before returning. For example, you might want to validate a dynamic element is present before returning it

or throw an error if it isn't present. In the code below, the method is getting all record rows. Since "element.all" returns a promise, the "get" needs to handle this:

```
export class UserTableRecords {
    constructor(private container: ElementFinder) {}

    public get userRecords(): Promise<UserRecord[]> {
        const records: UserRecord[] = [];
        return(async () => {
            for(const record of (await container.element.all(by.css('tr')))) {
                records.push(new UserRecord(record);
            }
            return records;
        })();
    }
}
```

You use the "await" keyword to ensure all the row elements have been found before you proceed to the next step.

Chaining promises

When one line of code needs to resolve multiple promises, you'll need to chain your promises with "await". Unfortunately, "await" is not a cure all – one "await" will not resolve all promises in one line of code. As an example, let's say you wanted to get the first name from the first record in a table. The code would look like this:

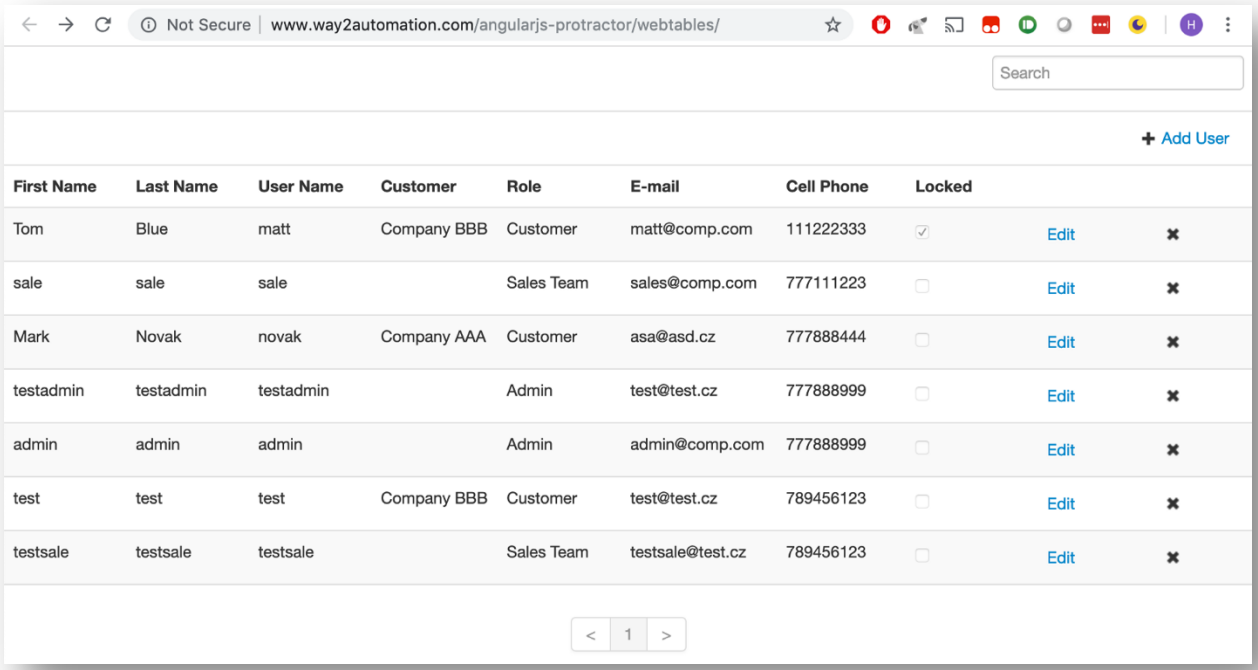
```
const firstName = await (await table.userRecords)[0].firstName.getText();
```

The action breaks down like this:

1. Call and execute the "userRecords" get method on the table object, returning an array of userRecord objects. Use "await" to ensure the method finishes before moving on.
2. Select the object in that userRecord array with index "[0]"
3. Select the "firstName" object in the "userRecord" object
4. Performs the "getText" action on the "firstName" object. Use "await" to ensure the method finishes before moving on.
5. Assign the text returned by "getText" to the constant "firstName" property

Writing a test

Now that we've gone over the Page Object Model, promises, and how to work with them in concert, we can finish up by writing a test to bring this all together. Here's the webpage again:



Let's write a basic test. I want to write a test that will validate that the first name of the third record is "Mark". To do that, I will need to:

- Find the page (dashboard)
- Find all the user records on the page
- Find the third record from the user records array
- Get the text from the First Name of that record
- Ensure that the record's First Name is "Mark"

Here's the code for that test:

```
describe('User records', () => {
    let dashboard: DashboardPage;
    let userRecords: UserTableRecord[];

    it('should use "Mark" for the third record', () => {
        dashboard = new DashboardPage();
        userRecords = await dashboard.userTableRecords.userRecords;

        const recordName= await userRecords[2].firstName.getText();
        const expectedName = 'Mark';
        expect(recordName).toEqual(expectedName,
`First name of the third record should be '${expectedName}' - is '${recordName}'`);
    }
}
```

The interesting things to note here:

- We use "await" when setting the "userRecords" property because calling the "get" method for "userTableRecords.userRecords" returns a promise

- “await” is also used within the scope of the “expect” to ensure we get the text of the “firstName” property before determining if it equals ‘Mark’
- The string, `` `First name of the third record should be '${expectedName}' - is '${recordName}'` ``, is a custom error message created for this specific expectation.
    - Creating specific error messages are not required, however it is helpful from a debugging standpoint. Why? Without this, the error message seen would be something like ‘`Expected Frank to equal Mark`’ (assuming “Frank” is the name seen). Writing a specific message allows for more clear debugging. That said, it can be onerous to add a custom message to each and every assertion/expectation. You’ll need to determine which option works best for your situation.
- Even though “toEqual” returns a promise, we do no put an “await” before the “expect”. Why not? In this instance, the code was written in Protractor using the Jasmine testing framework. Though Protractor doesn’t inherently handle promises otherwise, it does resolve promises here by default.

In Conclusion

Let's summarize what's been said above:

- Use the Page Object Model to increase code functionality, reduce repetition, and make maintenance easier
- Any action performed on an element will generate a promise
- Use the await keyword to force promises to resolve before continuing to the next action
- Use the `get` syntax to return elements in your POM class
- Chained promises need to use await within those chains

While the Page Object Model is fairly straight forward, promises are not. Using the information above, I've explained how both work on their own and when intertwined. I've also given examples to demonstrate their effectiveness. You should now be able to write automated tests for modern frameworks without issue. Good luck in all your endeavors!

Glossary:

Action - an interaction with an element. Examples of an action on an element: clicking, getting its text, determining if it's displayed, sending text, etc.

Async – a keyword used in a method's declaration, requiring the method to return a Promise.

Asynchronous Call – an action that initiates a promise.

Await – a keyword placed before an Asynchronous Call to tell the framework to wait for it to resolve before continuing on.

Container - an element that contains other elements. These other elements can use the container's scope to help with location. Example of usefulness: if there is a data table, each row of data will contain elements that likely use similar identifiers. If you use the <tr> tag of each row as a container, you can access each row's columns without fear of accessing data from other rows.

Element – a singular Page Object seen on a page of your application (text, input box, button, link, etc.).

Page Object - an object-oriented class that serves as an interface to a page of your [application under test].

Page Object Model – a method for representing patterns within the architecture of your application. The patterns are collections of page objects that can be captured in a class for easy reuse.

Promise - an asynchronous operation. These are most often (but not exclusively) raised when actions are performed on Page Objects (click, sendKeys, isDisplayed, etc.).

Resolving - this term refers to a promise completing its execution.