Automation Testing

Locators

A Day in the Life of an Automation Test Engineer

Mrak is now aware of performing end to end testing using Cypress.

Now, he has been asked to write HTML page as a test application using Cypress Locators with CSS Selector.

To achieve this, he will learn CSS selectors to identify any of the web elements which will help him to find a solution for the given scenario.

Learning Objectives

By the end of this lesson, you will be able to:

- Describe ButtonText Locator
- Define Binding and Model Locators
- Explain Repeat Locator
- Comprehend Page Objects

Cypress Locators

Defining Cypress Locators

A Selector or Locator is an object that searches for and returns web page items/elements depending on a query. The Locator or Selector aids in the location of an element on a webpage.

Elements By Id

Recognize HTML Elements Using 'Id' Cypress Locator

A sample HTML page is shown demonstrating the ID syntax in HTML:

Command:

```
<ul id="lists">
<li id="first"> Option 1 </li>
<li id="second"> Option 2 </li>
<li id="third"> Option 3 </li>
</ul>
```

Cypress Id Locator Demonstration

The following is an example of how to utilize the ID locator in Cypress:

Command:

```
describe('Cypress Locators in action', ()=>{
beforeEach(()=>{
cy.visit('http://automationpractice.com')
})
it('Interact with Elements using ID selector', ()=>{
cy.get('#search_query_top') //ids are identified using #
prefix
.type('Dress')
.type('{enter}')
.get('#center_column')
.should('be.visible')
})
})
```

Working with Locator Functions

Problem Statement:

You are asked to work with locator functions.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to work with locator functions are:

1. Work with locator functions

Element By Class

Recognize HTML Elements Using ‘by Class’ in Cypress

The HTML class property is mostly used to style an element or a set of components. Class attributes, unlike ID, do not have to be unique, which is useful when we need to apply a standard style to several components.

Command:

```
<a
href="http://automationpractice.com/index.php?id_category=3&contr
oller=category" title=“Women” class=“sf-with-ul”>Women</a>
```

Element By Tag Name

Recognize HTML Elements Using ‘Name’ Cypress Locator

The HTML name property is commonly used by form>, button>, and input> elements to identify field information when submitted to the server.

Command:

```
describe('Cypress Locators in action', ()=>{
beforeEach(()=>{
cy.visit('http://automationpractice.com')
})
it('Interact with Elements using name attribute',()=>{
cy.get('input[name="search_query"]') //observe how we use name
attribute
.type('Tshirt{enter}')
.get('#center_column')
.should('be.visible')
})
})
```

Recognize HTML Elements Using ‘Name’ Cypress Locator

In the Code walkthrough, the test "Interact with Elements using name attribute" begins by identifying elements using the name attribute ('search query') and then sends the search value to the text box using the type() method after the beforeEach() execution.

Command:

```
it('Interact with Elements using name attribute',()=>{
cy.get('input[name="search_query"]') //observe how we use name
attribute
.type('Tshirt{enter}')
```

Element By Attribute

Recognize HTML Elements Using ‘Attributes’ in Cypress

Attributes are methods for specifying a specific property of an HTML element. They also aid in the definition of the components' behavior when a certain condition is set and activated.

Command:

Syntax to use attributes in Cypress

```
cy.get(<tag>[attribute_name=attribute_value])
```

Recognize HTML Elements by ‘Link Text’ in Cypress

Links are methods of connecting web resources (or web pages) by using anchor tags (i.e., a>) and href attributes. For example, My Orders is the link text displayed on the website, as shown below.

Command:

```
<a href="http://automationpractice.com/index.php?controller=history">My orders</a>
```

CSS Selectors with Regular Expressions

Rules for Writing a CSS Selector

The steps for creating a CSS selector are listed below:

Using the class name attribute, the (.) sign is used to select items based on their unique class name.

With the assistance of id attributes, the (#) sign is used to select components based on their unique identifier.

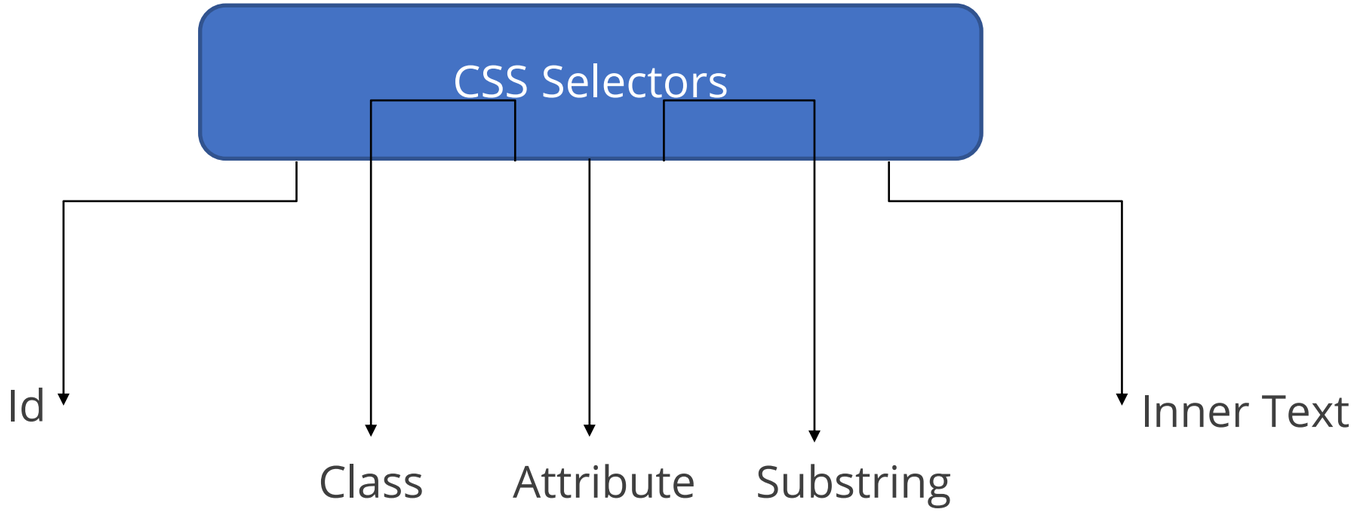
Rules for Writing a CSS Selector

The steps for creating a CSS selector are listed below:

- With the help of the tag name and id or class name attributes, the (#) sign is used to select the components based on their unique identifier.
- With the help of tag name and any characteristics, the items are chosen based on any properties that have a distinct value.
- By travelling from the parent, to child tag name separated by space, the personalized CSS expression must be (div input).

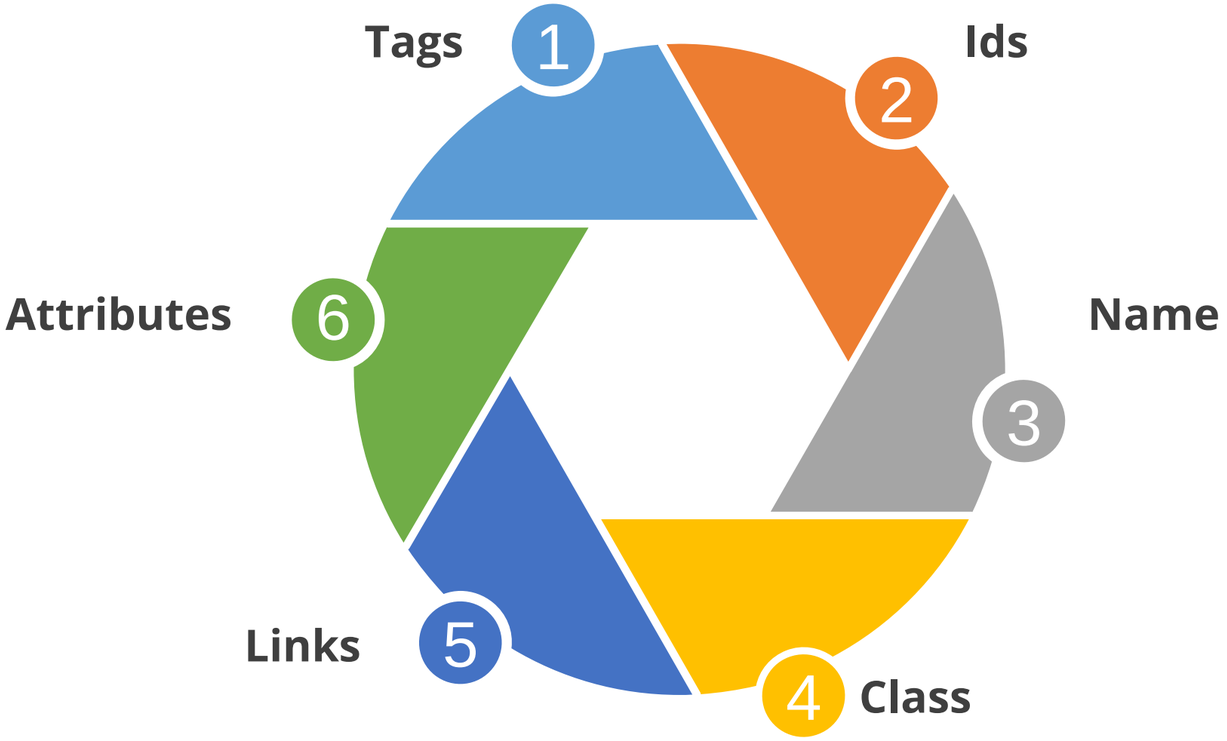
Use Locators in Cypress

For locating elements in the DOM, Cypress only supports CSS selectors out of the box.



Default Configurations Timeouts by Cypress

While searching for numerous items in Cypress, locators are often divided under the following categories:



AjaxElementLocatorFactory

In Selenium, AjaxElementLocatorFactory is a lazy loading notion of Page Factory.

Example

```
public int TimeoutValue = 20;
public SearchResultsPage(Webdriver driver) {
PageFactory.initElements(new AjaxElementLocatorFactory(driver,
TimeoutValue), this);
}
```

Working with Multiple Elements

Working with Multiple Elements

Cypress offers a convenient .each() function to iterate through array-like structures with multiple elements.

Example

```
cy.get("#primary-menu(id*=menu-item]").each((item, index, list)=>
{
expect(list).to.have.length(6);
});
```

ButtonText Locator

By.buttonText()

buttonThe text locator will attempt to match with an element that has the same text as the specified locator, or one of its sub-elements within the button tag.

Example:

```
/HTML code
<button>Save</button>
// usage of locator
element(by.buttonText('Save'));
//HTML code
<button>
<label>Save As   //matches
</label>
</button>
// usage of locator
element(by.buttonText('Save As'));
```

By.partialButtonText()

The partialButtonText locator looks for elements that have a partial match in the text of the button element.

Example:

```
//HTML code
<button>Save As Text </button>  //matches
// usage of locator
element(by.buttonText('Save'));
//HTML code
<button>
<label>Save As File  //matches
</label>
</button>
// usage of locator
element(by.buttonText('Save As'));
```

Binding and Model Locators

exactBinding

exactBinding is likewise used to locate the elements using the ng-bind locator, but with an exact string/value.

Example:

```
<p ng-bind="example.name"></p>
```

Binding Locators

The binding locator is used to locate the element based on the bind property value.

Example:

```
<p ng-bind="example.name"></p>
```

Model Locators

The model finds the element based on the model attribute value.

Example:

```
<input type="text" ng-model="example.name">
```

Repeater Locator

Repeater Locator

The repeater locator is used to find elements that have the ng-repeat attribute.

Example:

```
var eleID = element(by.repeater('product info').row(0));
expect(eleID.getText()).toBe('101');

var eleName = element(by.repeater('product info').row(1));
expect(eleName.getText()).toBe('Simplilearn');
```

by.exactRepeater

The exactRepeater locator finds the element with the precise text linked with the ng-repeat attribute. It will not search the text for incomplete matches.

Example:

```
<li ng-repeat="dev in developer_names"></li>

<li ng-repeat="test in tester_names"></li>
expect(element(by.exactRepeater('dev in developer_names')).isPresent()).toBe(true);
```

by.Repeater

The repeater locator is used to locate the elements with the ng-repeat attribute. It also aids in locating the text that is a partial match, i.e. if an attribute has some matching with a particular locator, our locator will find this element and return the necessary matching elements.

Example:

```
<tr ng-repeat="developer_info">
<td>{{dev.id}}</td>
<td>{{dev..name}}</td>
<td>{{dev.salary}}</td>
</tr>
var devID = element(by.repeater('developer_info').row(0));
expect(devID.getText()).toBe('2');

var devName = element(by.repeater('developer_info').row(1));
expect(devName.getText()).toBe('Mark');
```

by.exactRepeater

The exactRepeater locator finds the element with the precise text linked with the ng-repeat attribute. It will not search the text for incomplete matches.

Example:

```
// /* The XML input type contains the text with the bind
                    attribute */ //

      <li ng-repeat="dev in developer_names"></li>

       <li ng-repeat="test in tester_names"></li>
```

Creating and Executing Test Suites

Create a Test Case Grouping as a Test Suite

There are three methods to Create Test Case Grouping as Test Suite in Cypress Automation Framework.

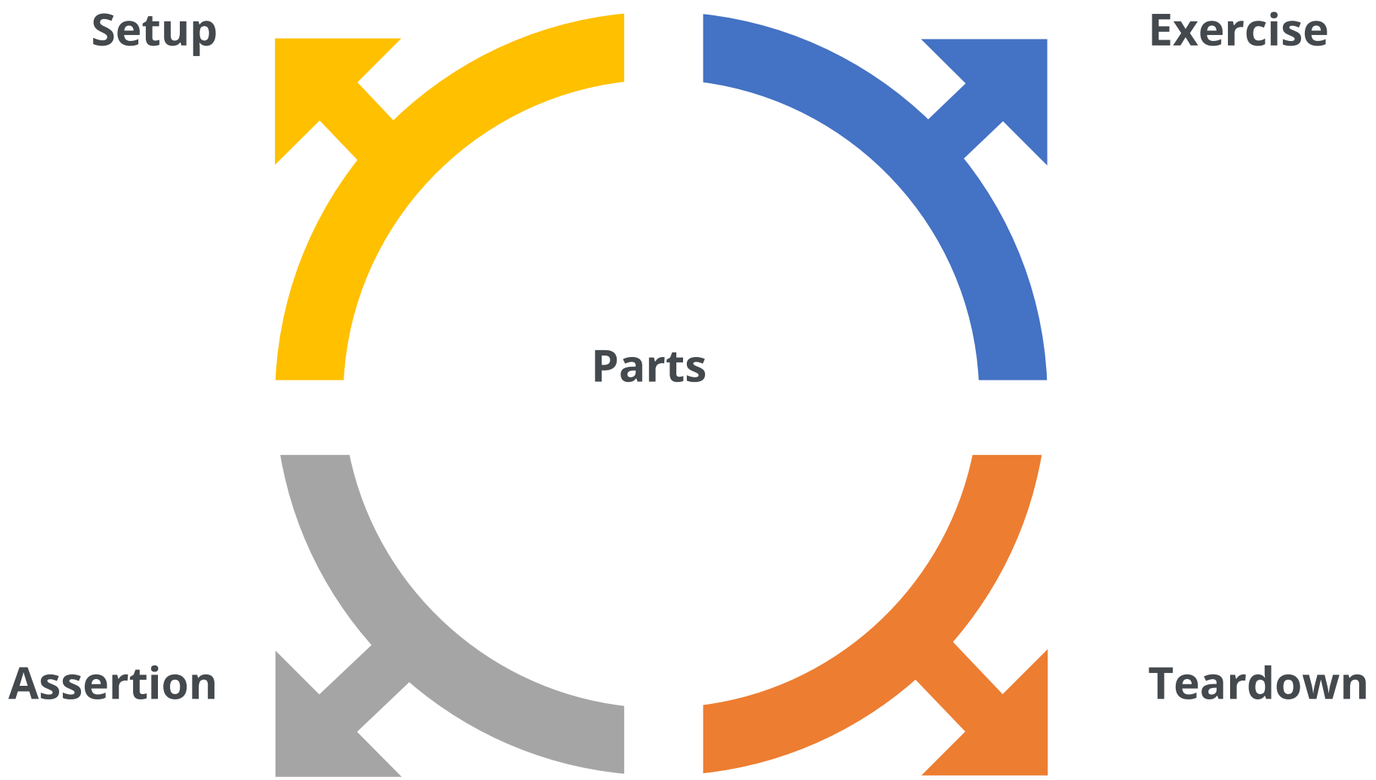
Method 1: Using --spec Options in Your Cypress Command Line

Method 2: Organizing the Test Script Folder as a Test Suite in Cypress

Method 3: Configuring the Support/Index.js and the Environment Variable

Steps to Write a Cypress Test Case

The test case should be divided into three to four logical parts:



Methods to Reduce Flakiness in Cypress Tests

Commands to reduce flakiness in Cypress tests:

- Reduce inconsistent interactions in the DOM
- Reduce request inconsistency
- Use the Cypress dashboard

.get() and .find()

Methods to Reduce Flakiness in Cypress Tests

Commands to reduce flakiness in Cypress tests:

- Reduce inconsistent interactions in the DOM
- Reduce request inconsistency
- Use the Cypress dashboard

cy.intercept()

Methods to Reduce Flakiness in Cypress Tests

Commands to reduce flakiness in Cypress tests:

- Reduce inconsistent interactions in the DOM
- Reduce request inconsistency
- Use the Cypress dashboard

Premium Dashboard Feature

Tooling: Reporters

Mocha comes with Cypress. As a result, any reports that can be created for Mocha can also be used with Cypress.

Tooling: Mochawesome Report

The Mochawesome report is one of the most important reports in Cypress.

Install mochawesome

Example

```
npm install mochawesome --save-dev
```

Tooling: Mochawesome Report

The Mochawesome report is one of the most important reports in Cypress.

Install mocha

Example

```
npm install mocha --save-dev
```

Tooling: Mochawesome Report

The Mochawesome report is one of the most important reports in Cypress.

Merge mochawesome json reports

Example

```
npm install mochawesome-merge --save-dev
```

Tooling: Mochawesome Report

The Mochawesome report is one of the most important reports in Cypress.

Merge multiple reports in a single report

Example

```
npm run combine-reports
```

Tooling: JUnit Report

Cypress provides another type of report known as the JUnit report.

Install the package for JUnit report

Example

```
npm install cypress-junit-reporter --save-dev
```

Tooling: Teamcity Report

Cypress offers another type of report known as the teamcity report.

Install the package for teamcity report

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Tooling: TypeScript

Cypress includes TypeScript official type declarations to support the framework.

Install TypeScript

Example

```
npm install --save-dev typescrip
```

Tooling: TypeScript

Users should create a tsconfig.json file in the Cypress folder with the following settings:

Configure tsconfig.json

Example

```
{
"compilerOptions": {
"target": "es6",
"lib": ["es6", "dom"],
"types": ["cypress", "node"]
},
"include": ["**/*.ts"]
}
```

Tooling: TypeScript

To avoid TypeScript issues while adding custom commands to the cy object, users may manually add their types.

Types for Custom Commands

Example

```
Cypress.Commands.add('dataCy', (value) => {
    return cy.get(`[data-cy=${value}]`)
})
```

Tooling: Visual Regression Testing

Visual Regression Testing is one of Cypress's extensible features. There are several plugins available in Cypress that may be used to record and compare visual pictures.

Cypress Visual Regression

Prerequisites

Install NodeJS

Install Visual Studio Code

Install Cypress

Setting Up Cypress Visual Regression

Install the cypress image diff npm package

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Setting Up Cypress Visual Regression

Configure image diff plugin

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Setting Up Cypress Visual Regression

Import and add the Cypress image command

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Setting Up Cypress Visual Regression

Configure reporter

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Setting Up Cypress Visual Regression

Write the first Cypress Visual Test

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Setting Up Cypress Visual Regression

Run your first Visual Regression Test with Cypress

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Setting Up Cypress Visual Regression

View report

Example

```
npm install cypress-teamcity-reporter --save-dev
```

Why Use LambdaTest Cypress CLI?

Here are some of the primary reasons why users should use LambdaTest Cypress CLI instead of the Cypress test automation framework:

Browser coverage has been improved

Parallel test execution that is accelerated

Detailed logging and video capture

Why Use LambdaTest Cypress CLI?

Browser coverage has been improved

The Cypress testing framework enables users to run tests concurrently by grouping them based on browsers, test labels, and other factors. Getting ideal browser coverage with a local Grid infrastructure, on the other hand, is impossible since the test infrastructure must be regularly updated.

Why Use LambdaTest Cypress CLI?

Parallel test execution that is accelerated

The LambdaTest Cypress CLI enables users to execute Cypress parallel testing on the LambdaTest Grid using a variety of browser and operating system combinations.

Why Use LambdaTest Cypress CLI?

Detailed logging and video capture

Cypress test automation on the LambdaTest grid includes full test logs, screencasts (or video captures of executed tests), and other features. These are useful tools for diagnosing problems in the existing tests.

Debugging Protractor Tests

Problems to Debug Protractor Tests

Certain modules' quality may be inadequate, or there may be browser compatibility testing concerns. Users may encounter the following issues along the way:

Testing a web application is difficult.

For cross-browser testing, WebDrivers can be separted for different operating systems and browsers.

The Selenium test automation scenarios leads to output dependency.

Problems to Debug Protractor Tests

Certain modules' quality may be inadequate, or there may be browser compatibility testing concerns. Users may encounter the following issues along the way:

Long error messages will be difficult to understand.

Differentiating mistakes from difficulties is difficult.

Types of Failures to Debug in Protractor Tests

There are four types of failures to debug in protractor tests which can be listed as:

Types of Failures

- Failure of Expectation
- Failure of the WebDriver and unexpected Failure of the WebDriver
- Protractor Angular Failure
- Protractor Timeout Failure

Debugging Protractor Tests In Selenium

There are three methods to debug protractor tests in Selenium which are listed as:

Pause method

Screen shot method

Debugger method

Methods To Debug Protractor Tests

There are two methods to debug protractor tests:

- Debugger Method
- Screenshot Method

Key Takeaways

- Cypress gives users the ability to dynamically change environment variables and configuration parameters from the Cypress setup.
- Each specification file is run separately by Cypress; the browser is closed in between specifications.
- End-to-end testing (E2E) concentrates on what ultimately matters: ensuring that users will be able to accomplish their goals within the application.
- E2E testing files must end with spec.js to be presented correctly.