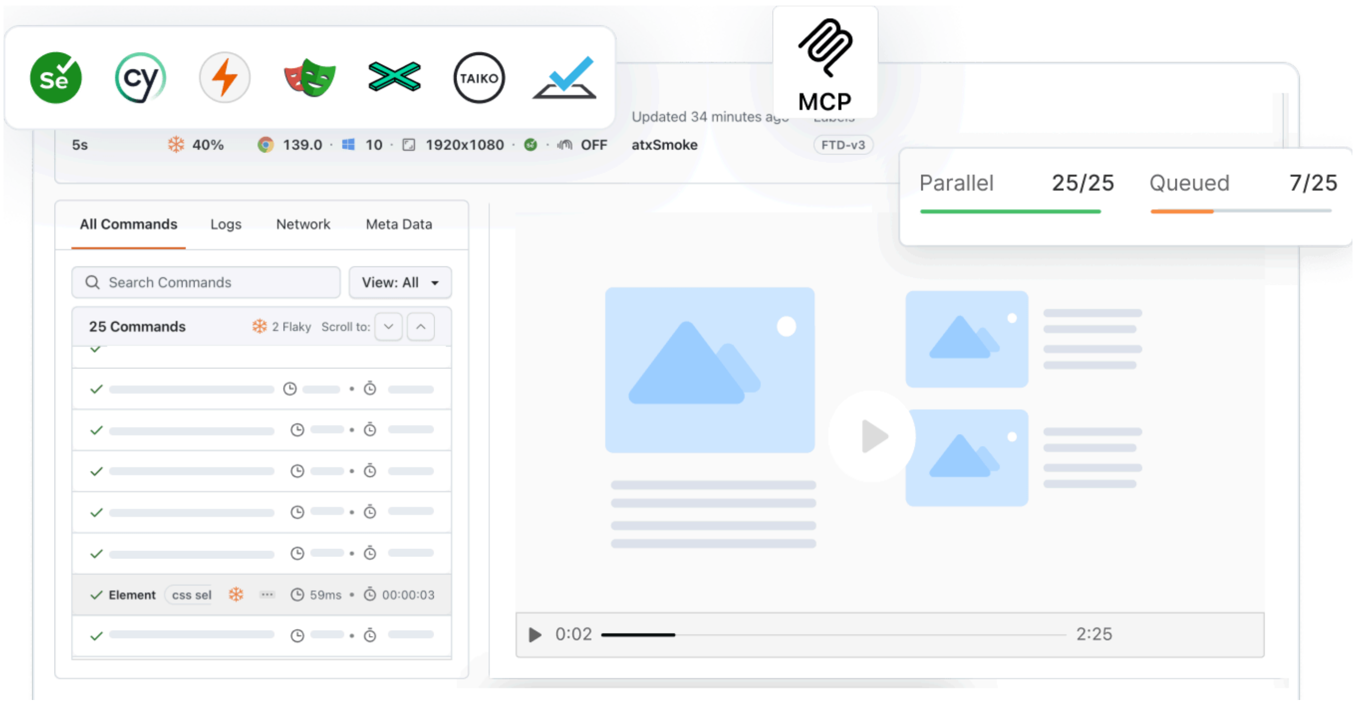
Top 25+ Playwright Interview Questions

lambdatest.com/learning-hub/playwright-interview-questions



Prepare for your Playwright interview with the most common Playwright interview questions & Playwright automation interview questions, with detailed answers.

Playwright is a Node.js library that automates Chromium, WebKit, and Firefox with a single API. Playwright was released in 2020, and in such a short time, it gained huge popularity. As per the State of JS, in just 3 years, awareness of Playwright reached 51%, and more than 74% of the tester and developers showed interest in learning it. It shows how fast organizations are adapting to this new framework.

In that case, it is essential to have the right knowledge about Playwright framework to clear the interview. If you are preparing for a Playwright automation interview, this list of Playwright interview questions can help you get a list of the most asked Playwright interview questions with detailed answers and examples for better understanding.

Here are the Playwright's interview questions with detailed answers.

Playwright Interview Questions
Note: We have compiled all Playwright Interview Questions Sheet for you in a template format. Feel free to comment on it. Check it out now!!

Basic Playwright Interview Questions

1. What is the Playwright framework, and what are its key features?

Playwright is a Node.js library developed by Microsoft that automates Chromium, Firefox, and WebKit with a single API. Developers writing JavaScript code can use these APIs to build new browser pages, navigate to URLs, and interact with page elements. Furthermore, because Microsoft Edge is based on the open-source Chromium web framework, Playwright may automate Microsoft Edge.

Some key features make Playwright different:

- Cross-browser development on Chromium, WebKit, and Firefox is supported, including Chrome, Edge, Firefox, Opera, and Safari. Cross-platform execution is possible on Windows, Linux, and macOS.
- Cross-language testing, including JavaScript, TypeScript, Python, Java, and .NET, allows you to choose the best environment for you while covering all regions and formats.
- Keep track of logs and videos effortlessly using auto-wait, clever assertions that retry until an element is located, and test data tracing.
- Playwright frameworks take full-page screenshots of the webpage, which makes visual testing easier.
- It is built with current architecture and no constraints, allowing you to interact with multi-page, multi-tab websites like a genuine user, and effortlessly handles frames and browser events.
- Because the Playwright framework is built on the architecture of modern browsers, it does not have the constraints of an in-process test runner.

Note: Accelerate Playwright Automation testing with LambdaTest.

2. How does Playwright differ from other testing frameworks?

Released in 2020, Playwright is gaining popularity every day. The Playwright is registering 1.3 million downloads every week on NPM as of April 2023. The reason is it has many advantages over other frameworks. It enables you to test complex web applications across several browsers. It gives superior test coverage and reliable findings. Additionally, there are several benefits to automating your web testing. Such as:

- It includes capabilities for step-by-step debugging, exploring selectors, and recording new tests. It is accessible as a VS Code addon.
- To examine test execution results in a browser, Playwright creates an HTML report. Visual discrepancies and test artifacts like screenshots, traces, error logs, and video recordings are included.
- Playwright installation just takes a few minutes to be configured. However, the installation procedure may vary depending on the programming language you use with Playwright to perform the tests.

- Playwright offers a high-level API that frees you from dealing with low-level browser interactions so that you may design tests that are easier to comprehend and manage.
- Playwright includes built-in parallelism support, allowing you to run tests simultaneously across many browsers or computers, which can help you save time and expedite your test runs.
- You can test your application on various mobile devices and screen sizes because of Playwright's ability to emulate mobile devices.

Be sure to check out our comprehensive guide on Top Asked mobile testing interview questions to further strengthen your preparation.

3. What programming languages does Playwright support?

Here is the list of programming languages Playwright supports:

- JavaScript/Node.js
- TypeScript
- Python
- C#
- Java

4. Is there any Webdriver dependency in Playwright?

No, Playwright is not dependent on WebDriver. Playwright communicates with the browser via a different protocol than WebDriver-based automation tools. It uses the DevTools protocol, a more advanced and effective method of controlling the browser.

5. Can you describe Playwright's architecture?

To understand Playwright architecture better, we will compare it with the Selenium structure. Selenium sends each command as a separate HTTP request, and it receives JSON replies in return. Then, a separate HTTP request is issued for each interaction, such as opening a browser window, selecting an object, or typing text into a text field.

As a result, we must wait longer for responses, and the likelihood of mistakes rises.

Playwright uses a single WebSocket connection to communicate with all drivers, which is kept open until testing is complete, instead of having separate connections for each driver. This lowers the number of potential failure sites by enabling rapid command transmission via a single link.

6. Can I perform tests for Playwright using Jasmine?

Yes, we can use Jasmine as the test runner for Playwright since Jest and Jasmine have nearly identical syntax.

7. What is the difference between Playwright and Puppeteer?

Playwright is the first release as a fork of Puppeteer. Puppeteer is a node library that uses JavaScript to automate Chromium-based browsers. Even though Playwright started as one fork of Puppeteer, there are many key differences.

Cross-browser interoperability is Playwright's key differentiator. It can power Firefox, Chromium, and WebKit (the Safari browser engine). Along with this, Playwright was able to enhance ergonomics in ways that might damage Puppeteer by beginning a new library.

Playwright's **page.click**, for instance, waits for an element to be visible. Puppeteer still has room for improvement, although Playwright is sometimes a better option.

The Playwright automation platform's more robust browser context features, which let you simulate many devices with a single instance, are the final notable difference. Multiple sites may be nested within a single browser context, and each context is independent of the others in terms of cookies and local storage.

Contrarily, browser automation can be started quickly using Puppeteer testing. This is partially attributable to its simplicity of usage, as it manages Chrome using the unusual DevTools interface, makes it simple to intercept network requests, and offers more capabilities than Selenium.

Playwright Automation interview questions

8. How can I install Playwright with NodeJS?

You can install Playwright easily from NodeJS. Here are the steps to follow:

- Launch a command prompt or terminal window.
- Go to the project directory where Playwright should be installed.
- To install the Playwright package, do the following command:

```
npm install Playwright
```

The most recent version of Playwright and its dependencies will be installed in your project directory.

When the installation is finished, import Playwright into your JavaScript code:

```
const { chromium } = require('Playwright');
```

This imports the Playwright library's Chrome browser object, which you can use to automate browsing activities and tests.

9. Can I run tests parallelly in Playwright?

Yes, Playwright enables parallel test execution, enabling you to speed up your test suite by running several tests concurrently. A test runner that supports parallel execution, such as Jest or Mocha, can be used to execute tests in parallel.

For instance, If you are using Jest, then **--runInBand** option allows you to run tests concurrently. The following code must be added to your jest.config.js file to activate this option:

```
module.exports = {
  // ...
  testRunner: "jest-circus/runner",
  maxConcurrency: 5, // set the maximum number of parallel tests
};
```

10. How does Playwright handle browser automation and testing?

Through a single API, the Playwright library offers cross-browser automation. Chromium, Firefox, and WebKit can all be automated using a single API, thanks to the Playwright Node.js module. Playwright makes cross-browser online automation that is evergreen, competent, dependable, and quickly possible.

Here are the steps you can follow to handle browser automation in Playwright:

- Install the Playwright test to perform a test on your website:

  ```
  npm i -D @Playwright/test
  ```

- Install browsers on which you want to run a test with the command given below:

  ```
  npx Playwright install
  ```

- Write a tiny script for automation testing. Here is an example of running the Playwright test in Chrome browser.

```
const { chromium } = require('Playwright');


(async () => {
  // Launch a Chrome browser instance
  const browser = await chromium.launch();


  // Create a new page
  const page = await browser.newPage();


  // Navigate to a URL
  await page.goto('https://www.lambdatest.com/');


  // Get the page title and log it to the console
  const title = await page.title();
  console.log('The page title is: ${title}');


  // Close the browser
  await browser.close();
})();
```

Also Read: A list of 70 Cucumber Interview Questions and Answers

11. Can you explain how Playwright's page object model works?

A design technique called page object modeling enables us to model a group of objects to a certain page type. Despite its name, a page is only sometimes what it appears to be; it could be a part of the website. Here is how Page object model in Playwright works:

- Define page object classes: Create classes that contain methods that correspond to the many interactions a user might have with the page, such as clicking buttons, typing text, and checking the page's status.
- Encapsulate various page elements: The various web page components that your methods interact with can be contained within the page object class.
- Create the page object: To instantiate the object, create a new instance of the page object class in your test code. This grants you access to the page object class's methods.
- Use the Page object method for testing: Once your Page object model is generated, you can use those classes for test automation.

12. How does Playwright handle asynchronous operations?

To handle asynchronous actions and run the test phases sequentially, Playwright employs the async/await keywords.

Playwright's methods and functions return promises that resolve after the process is finished when you interact with a web page. For instance, when an element is clicked, the click method provides a promise that resolves. Before performing the next operation, you can use the await keyword to wait for the promise to resolve.

Here is a code snippet for better understanding:

```
const { chromium } = require('Playwright');


(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();


  await page.goto('https://www.lambdatest.com');


  // Wait for the element to be visible before clicking it
  const button = await page.waitForSelector('#my-button');
  await button.click();




  // Wait for an animation to finish before taking a screenshot
  await page.waitForTimeout(1000); // Wait for 1 second
  await page.screenshot({ path: 'lambdatest.png' });


  await browser.close();
})();
```

13. How does Playwright integrate with Continuous Integration (CI) tools?

When code updates are posted to your repository, Playwright works seamlessly with Continuous Integration (CI) tools like Jenkins, Travis CI, CircleCI, and others so that your tests run automatically. Here is the way you can integrate Playwright with CI tools:

- Verify that a browser can be executed on the CI agent. The Linux agent's Playwright Docker image Utilizes command-line tools to install your dependencies.
- Examine the documentation for the tools or frameworks you intend to integrate; many of these include thorough instructions on how to do so with the Playwright framework.
- Utilize the Playwright automation framework's most recent version. The framework has been updated to make it easier to integrate with other systems.
- Verify that the two frameworks' necessary dependencies are installed and configured.

14. What are some common challenges you've faced when using Playwright, and how did you overcome them?

Playwright has certain drawbacks, such as not supporting Legacy Microsoft Edge or IE11, using emulators instead of real devices, and many more. However, here are some major common challenges with Playwrights with a possible solution:

- **Setting up and keeping the test environment up to date:**
  Keeping the test environment up to date is one of the key Playwright difficulties. It can involve setting up browsers, installing dependencies, and configuring the test runner. The use of containerization solutions like Docker, which can provide a consistent testing environment across many computers and environments, is advised to address this difficulty.

- **Playwright test debugging:**
  Playwright test debugging can be difficult, particularly when dealing with complex situations or intermittent failures. Using debugging tools, such as the Playwright CLI's debug command, which enables you to pause test execution and investigate the current state of the page and the test code, is one way to get around this problem.

- **Managing asynchronous activity:**
  Asynchronous behavior is frequently included in Playwright testing, which can be difficult to manage. It is advised to use async/awaits syntax to get around this problem and make sure that the tests wait for the page to load and for the required behavior to happen before moving on to the next stage.

15. How to automate date pickers in Playwright?

Playwright offers two major methods to automate date pickers or calendars:

- **Fill method**
  Using the fill() command to automate date pickers is a quicker and simpler process. The fill() function only requires the date to be sent as an argument. These are the procedures for this approach. Here is the code snippet as an example:

```
import { test } from "@Playwright/test";


 test("Calendar demo using fill function", async ({ page }) => {
     await page.goto("https://www.lambdatest.com/selenium-playground/bootstrap-
date-picker-demo");
     let date = "1994-12-04"

     await page.fill("id=birthday", date);
     await page.waitForTimeout(3000)
 })
```

- **Moment method**
  An open-source and cost-free JavaScript tool called Moment.js aids in solving date and time-related problems. It is a collection of native JavaScript date objects that makes it simple to change or show date and time in JavaScript.

  This method allows you to create perfect methods to test date pickers or calendars. Here is the code snippet for better understanding.

  ```
  async function selectDate(date: number, dateToSelect: string) {
  await page.click("//input[@placeholder='Start date']")

  const mmYY = page.locator("(//table[@class='table-
  condensed']//th[@class='datepicker-switch'])[1]");
  const prev = page.locator("(//table[@class='table-condensed']//th[@class='prev'])
  [1]");
  const next = page.locator("(//table[@class='table-condensed']//th[@class='next'])
  [1]");

  // let dateToSelect: string = "May 2019";
  const thisMonth = moment(dateToSelect, "MMMM YYYY").isBefore();
  console.log("this month? " + thisMonth);
  while (await mmYY.textContent() != dateToSelect) {
  if (thisMonth) {
  await prev.click();
  } else {
  await next.click();
  }
  }
  ```

16. How do you debug tests in Playwright?

Playwright offers various methods to debug tests, as mentioned below:

- **VS Code debugger:**
  If you are using VS code extension, you can step through your tests while debugging them, view error messages, and create breakpoints. Here are the steps you can follow. If your test fails, VS Code will display error messages directly in the editor that include a call log, what was anticipated, and what was received.

  In VS Code, you may actively debug your test. Click on any of the locators in VS Code after running a test with the Show Browser checkbox selected, and it will be highlighted in the Browser window. You can see if there are several matches, thanks to Playwright.

- **Using the Playwright Command-Line Interface (CLI):**
  Playwright has a command-line interface (CLI) with a debug command that enables you to halt the test execution and examine the current state of the page and the test code. Simply add the --debug flag to your test command to activate the debug command, and Playwright will halt test execution if it comes across a debugger statement in your test code.

- **Using the Playwright Inspector:**
  While your test is running, you can interact with the page and examine its state using Playwright's interactive debugging tool, the Playwright Inspector. By adding the --inspect flag to your test command and then navigating to the given URL in your browser, you can start the Playwright Inspector.

17. Can you explain how Playwright handles cookies and local storage?

During test execution, Playwright offers APIs to manage cookies and local storage in the browser. An overview of Playwright's cookie and local storage policies is as below:

- **Handling cookies in Playwright:**
  To obtain all of the cookies for the current page, use the cookies() function on the page object. The setCookie() method can also be used to create a new cookie or make changes to an existing cookie. The deleteCookie() can be used to remove a cookie.

```
await page.setCookie({
  name: 'myCookie',
  value: 'myValue',
  domain: 'example.com',
  path: '/',
  expires: Date.now() + 86400000, // expires in 1 day
  httpOnly: true,
  secure: true
});
```

- **Handling cookies in Playwright:**
  To obtain all of the cookies for the current page, use the cookies() function on the page object. The setCookie() method can also be used to create a new cookie or make changes to an existing cookie. The deleteCookie() can be used to remove a cookie.

```
await page.setCookie({
  name: 'myCookie',
  value: 'myValue',
  domain: 'example.com',
  path: '/',
  expires: Date.now() + 86400000, // expires in 1 day
  httpOnly: true,
  secure: true
});
```

- **Handling Local storage:**
  To run JavaScript code in the browser context and work with local storage, use the evaluate() function on the page object. Here is the code snippet for it:

```
await page.evaluate(() => {
  localStorage.setItem('myKey', 'myValue');
});
```

18. What are some best practices for writing efficient and maintainable tests with Playwright?

Here are some best practices for writing efficient test scripts with Playwright:

- Use relevant test names that are descriptive and reflect the functionality when naming your tests. This makes comprehending the test's goal simpler, especially if you have a sizable test suite.
- A test should be independent and atomic, meaning it should only test one object and not depend on the results of other tests. When a test fails, it is simpler to identify and address problems.
- Your tests will be easier to comprehend and manage if page objects are used to encapsulate page-specific behavior. Page objects offer an abstraction layer that keeps the test code distinct from the page's implementation specifics.
- The waitForSelector() and waitForNavigation() functions in Playwright are just two of the mechanisms available for waiting for elements and actions to finish. Utilize these techniques to manage timing concerns and guarantee the consistency and dependability of your testing.
- Create logical groupings for your tests depending on the features or components. Create logical groups for your tests. It makes execution simpler to particular subsets of tests and helps comprehend the general structure of the test suite.
- Use version control to coordinate with other team members and manage your test code. To automate your testing procedure and guarantee that your tests are run consistently and dependably, use continuous integration tools.

19. What is waitFor() in Playwright?

The waitFor() method in Playwright enables you to postpone the script's execution while you wait for a particular condition to hold true. It only accepts one argument, a function that yields a boolean value, and it iteratively executes that function until it returns true.

Here is an illustration of how to use Playwright's waitFor() function to wait for an element to become visible:

```
await page.waitFor(() => {
    const element = document.querySelector('#my-element');
    return element && element.isVisible();
  });
```

20. Which Selectors are used in Playwright?

To find elements on a web page, Playwright offers several selectors. Here is the list of the most popular selectors in Playwright with an example:

- CSS Selector:

  ```
  const buttons = await page.$$('button');
  ```

- XPath Selector:

```
const input = await page.$x('//input[@name="username"]');
```

- ID Selector:

```
const element = await page.$('#my-element');
```

- Text Selector:

```
const link = await page.$('a[href="https://www.lambdatest.com/"]');
```

21. How do you get the CSS value of an element?

Here are two methods to get the CSS value of an element:

- **elementHandle.$eval() method:**
  Playwright's elementHandle.$eval() method allows you to obtain an element's CSS value. This method calls a function while taking the page's context into account, then returns the result to the caller.

  Here is a code snippet for using elementHandle.$eval()

  ```
  const element = await page.$('#my-element');
  const backgroundColor = await element.$eval('body', el =>
  getComputedStyle(el).backgroundColor);
  console.log('Background color:', backgroundColor);
  ```

- **elementHandle.evalute() method:**
  You can also run a function that returns the computed style of an element by using the elementHandle.evaluate() method. Here's an illustration of how to use evaluate() to determine an element's font-size:

  ```
  const element = await page.$('#my-element');
  const fontSize = await element.evaluate(el =>
  parseFloat(getComputedStyle(el).fontSize));
  console.log('Font size:', fontSize);
  ```

22. Can you walk me through setting up a new Playwright project?

You can follow the steps below to set up a new Playwright project:

- Ensure Node.js is set up on your computer. Running **node -v** in your terminal or command prompt will show you whether it is installed or not.
- In your terminal or command prompt, create a new directory for your project and go there.
- Start a new Node.js project by launching **npm init** and following the on-screen instructions. For your project, this will produce a **package.json** file.
- Run npm install Playwright to add Playwright as a dependency.
- Install the corresponding browser driver(s) as devDependencies for the browser(s) you want to automate with Playwright. For instance, you may install the driver by executing **npm install Playwright-chromium** if you wish to automate Chromium.

- In the tests directory at the project's root, create a new file named **Playwrighttest1.spec.js** (or whatever name you like). Your test code will be in this file.

23. How does Playwright ensure cross-browser compatibility?

Web applications can be tested with Playwright on a variety of browsers, including Gecko-based Mozilla Firefox, WebKit-based Apple Safari, and Chromium-based Google Chrome, and the new Microsoft Edge. Playwright 1.32.3 is currently accessible on NPM.

With a foundation that prioritizes dependable and quick execution, Playwright can automate a wide variety of scenarios across several browsers with a single API. Here is an example of it:

```
const { chromium, firefox, webkit } = require('Playwright');


(async () => {
  // Define the test code
  const runTest = async (browserType) => {
    const browser = await browserType.launch();
    const context = await browser.newContext();
    const page = await context.newPage();
    await page.goto('https://www.lambdatest.com/');
    // Add test code here
    await browser.close();
  };


  // Run the test in multiple browsers
  await Promise.all([
    runTest(chromium),
    runTest(firefox),
    runTest(webkit),
  ]);
})();
```

24. What are some common errors you might encounter when using Playwright, and how do you troubleshoot them?

Here is the list of errors you might encounter while using Playwright with solutions to solve it:

- **TimeoutError:** When a Playwright operation takes longer to complete than the default timeout (30 seconds), this error may appear. If the operation is stuck because of a race condition or another problem, you can raise the timeout by giving a timeout option to the appropriate method.
- **SelectorTimeoutError:** This error will appear when Playwright cannot locate an element on the page that matches a given selector. You can solve it by verifying that the selection is accurate and that the element is on the page when the test is run.
- **NavigationError:** It appears during a DNS resolution error or a server fault, which can result in a navigation error. You can troubleshoot by verifying the URL and whether the server is up and operating.

- **ActionabilityError:** This error appears when Playwright cannot carry out a specific action on an element, like clicking a button or entering text into a text field. You can solve it by checking whether the element is visible and in the proper condition for the operation to be performed can help you troubleshoot.

25. How do you stay up to date with new features and updates to the Playwright framework?

There are various sources to be updated about Playwright automation frameworks, such as:

- Official Documentation
- Release notes
- Community forum
- Conference and events
- GitHub repository
- Twitter

Conclusion

Playwright testing frameworks are growing, and in the near future, there will be a huge demand for testers with knowledge about Playwright. Here in this questionnaire, we have discussed some most asked Playwright interview questions with answers and practical examples, it is not only helpful for clearing the interview, but those are general concepts related to Playwright automation frameworks that will be applicable throughout the software testing career.

Playwright is continuously evolving as well, so for testers, it is mandatory being kept yourself updated. Also, it is essential to have a strong fundamental about automation testing and Playwright frameworks. By keeping all those things in mind, you can distinguish yourself as a worthy candidate and leverage your testing career. Best of Luck!