Playwright Interview Questions and Answers

Complete Guide (2026)

Preparing for a Playwright interview means more than memorizing syntax, interviewers want real understanding. This guide covers 120 Playwright interview questions and answers, from fundamentals to advanced scenarios, matched to your experience level, whether you're a fresher or an experienced SDET. Use it to walk in ready to explain concepts in your own words, not repeat memorized lines.

Book Free Demo Class | WhatsApp For More Details

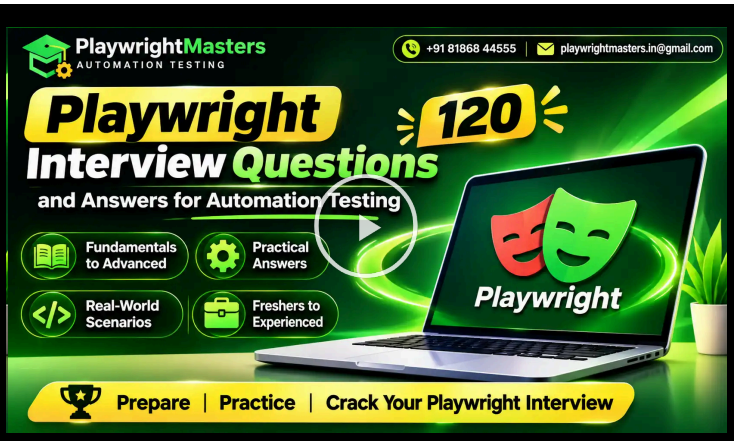
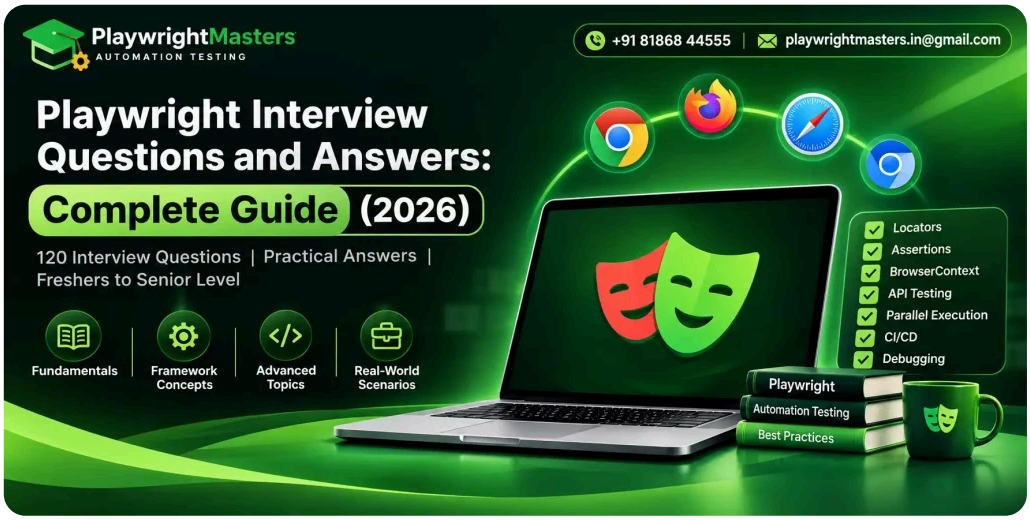


Table of Contents



00:00 00:00

Book Free Demo Class

If you are preparing for a Playwright automation testing interview, knowing syntax alone is not enough. Interviewers can test your understanding of browser automation, locators, assertions, fixtures, browser contexts, Page Object Model, API testing, debugging, parallel execution, CI/CD, and real-world test design.

This guide brings together 120 Playwright interview questions and answers, progressing from fundamentals to advanced and scenario-based topics. It is designed for freshers, automation testers, SDETs, QA engineers, and experienced professionals.

Playwright currently supports automation across Chromium, Firefox, and WebKit, with language support for TypeScript, JavaScript, Python, Java, and .NET. Its testing ecosystem includes features such as auto-waiting, assertions, tracing, parallel execution, and test isolation.

Whether you are preparing for your first interview or a senior automation testing role, use the questions below to identify what you know, what you need to revise, and how well you can explain Playwright in practical situations.

How to Use This Playwright Interview Guide

Do not try to memorize all 120 answers word for word.

A better approach is to understand the concept behind each answer and practice explaining it in your own words.

Preparation level	Focus on
Fresher	Fundamentals, locators, assertions, browser concepts, basic coding
1 to 2 years	Framework concepts, fixtures, POM, waits, debugging, API testing
3+ years	Architecture, scalability, parallelism, CI/CD, authentication, advanced debugging
SDET / Senior QA	Framework design, trade-offs, reliability, performance, CI/CD and real-world scenarios

Part 1: Playwright Fundamentals

These questions establish whether you understand what Playwright is, why teams use it, and how it fits into automation testing.

1. What is Playwright?

Playwright is a free, open-source framework for automating web browsers, built and maintained by Microsoft. It lets developers and QA engineers write automated tests that run consistently across all major browser engines, including Chromium, Firefox, and WebKit, from a single codebase.

It can be used for:

- End-to-end testing
- UI automation
- API testing
- Cross-browser testing
- Mobile browser emulation
- Network interception
- Authentication testing
- Debugging and test reporting

2. What is Playwright used for?

Playwright is primarily used to automate and test modern web applications.

For example, a tester can automate a complete e-commerce flow:

- Open the website
- Log in
- Search for a product
- Add it to the cart
- Complete checkout
- Verify the order confirmation

This makes Playwright a solid fit for regression, smoke, functional, end-to-end, and cross-browser testing alike.

3. Why is Playwright popular for automation testing?

Playwright provides several features that make modern browser automation easier:

- Built-in auto-waiting
- Web-first assertions
- Browser isolation through contexts
- Cross-browser testing

- Parallel test execution
- Network interception
- API testing
- Trace Viewer
- Test retries
- Multiple language bindings
- CI/CD support

These features reduce the amount of custom infrastructure testers need to build.

4. Which browsers does Playwright support?

Browser engine	**Playwright support**
Chromium	Yes
Firefox	Yes
WebKit	Yes
Chrome	Supported through Chromium-based browser channels
Microsoft Edge	Supported through Chromium-based browser channels
Safari	Tested through Playwright's WebKit implementation rather than the branded Safari browser

Playwright's official documentation specifically distinguishes its WebKit browser from branded Safari.

5. Which programming languages does Playwright support?

Playwright provides bindings for:

- TypeScript
- JavaScript
- Python
- Java
- .NET

The available testing ecosystem differs by language. For instance, Node.js projects typically rely on Playwright Test as the go-to runner, whereas Python developers more often reach for the Playwright Pytest plugin.

6. Should Playwright be classified as a testing framework, or is it more of a general browser automation tool?

It can be viewed as both, depending on how it is used.

The Playwright library provides browser automation capabilities, while Playwright Test provides a complete testing framework for JavaScript and TypeScript projects.

Playwright Test includes features such as:

- Test runner
- Assertions
- Fixtures
- Parallel execution
- Reporting
- Retries
- Tracing
- Test isolation

7. What is end-to-end testing?

End-to-end testing verifies an application from the user's perspective across a complete workflow.

For example:

Login -> Search -> Add product -> Checkout -> Payment -> Order confirmation

Instead of testing one function in isolation, an E2E test checks whether different parts of the application work together correctly.

8. What is cross-browser testing?

Cross-browser testing verifies that an application works correctly across different browser engines.

With Playwright, the same test suite can be configured to run against Chromium, Firefox, and WebKit.

This helps identify browser-specific issues.

9. What is browser automation?

Browser automation lets a script open, click, and navigate a web browser on its own, no human clicking required.

For example:

```
await page.goto('https://example.com');

await page.getByRole('button', { name: 'Login' }).click();
```

Instead of manually clicking the button, Playwright performs the action automatically.

10. What are the main advantages of Playwright?

The major advantages include:

- Cross-browser automation
- Auto-waiting
- Reliable locators
- Browser context isolation
- Parallel execution

- API testing
- Network interception
- Built-in debugging tools
- Trace Viewer
- CI/CD integration
- Multiple programming language options

The official documentation emphasizes auto-waiting, test isolation, resilient locators, parallelism, and sharding as important Playwright capabilities

Part 2: Playwright Setup, Architecture and Core Concepts

Understanding the internal building blocks of Playwright is important because interviewers often move from basic definitions to architecture questions.

11. What is the basic architecture of Playwright?

A simplified Playwright architecture can be understood as:

Test -> Playwright API -> Browser -> Browser Context -> Page -> Web Application

The test sends commands through Playwright's API. Playwright controls the browser, creates isolated browser contexts, and interacts with pages inside those contexts.

12. What is a Browser in Playwright?

A Browser represents a browser instance such as Chromium, Firefox, or WebKit.

Example:

```
const browser = await chromium.launch();
```

The browser can then be used to create one or more browser contexts.

13. What is Browser Context?

A BrowserContext is an isolated browser session.

It functions much like a brand-new, untouched browser profile. Cookies, local storage, session storage, and other session information can be isolated between contexts.

Example:

```
const context = await browser.newContext();

const page = await context.newPage();
```

This isolation is one of the important reasons Playwright tests can run independently.

14. What is a Page in Playwright?

A Page represents a browser tab or page.

You use the Page object to:

- Navigate to URLs
- Locate elements
- Click buttons
- Fill forms
- Read content
- Handle dialogs
- Interact with frames

Example:

```
await page.goto('https://example.com');
```

15. What is the difference between Browser, BrowserContext and Page?

Object	**Meaning**
Browser	Browser instance
BrowserContext	Isolated browser session
Page	Individual browser tab

A common relationship is:
Browser -> Context -> Page
One browser can contain multiple contexts, and each context can contain multiple pages.

16. Why is BrowserContext important in Playwright?

Browser contexts provide test isolation.

For example, Test A can have one user's cookies while Test B has another user's cookies. Their sessions do not need to interfere with each other.

This makes parallel and independent testing easier.

17. What is Playwright Test?

Playwright Test is the test runner provided for Playwright's Node.js ecosystem.

It provides:

- Test organization
- Assertions
- Fixtures
- Hooks
- Parallel execution
- Retries
- Projects
- Reporting

- Tracing
- Configuration

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut elit tellus, luctus nec ullamcorper mattis, pulvinar dapibus leo.

18. How do you install Playwright?

For a typical Node.js project, you can initialize a Playwright project using:

```
npm init playwright@latest
```

The setup can create the configuration, test directory, example test, and browser installation.

19. What is playwright.config.ts?

playwright.config.ts is the central configuration file for a Playwright Test project.

It can define:

- Browsers
- Projects
- Base URL
- Retries
- Workers
- Timeouts
- Reporters
- Screenshots
- Videos
- Traces
- Test directories

This keeps project-level settings centralized.

20. What is a Playwright project?

A project is a configuration that allows the same test suite to run under different settings.

For example, you can create projects for:

- Chromium
- Firefox
- WebKit
- Mobile devices
- Different environments

This is particularly useful for cross-browser testing.

Part 3: Locators and Selectors

Locators are one of the most important Playwright interview topics. Playwright recommends user-facing locators such as role, label, text, placeholder, and test ID where appropriate.

21. What are Playwright locators?

Locators are objects used to identify elements on a web page.

Examples include:

- getByRole()
- getByText()
- getByLabel()
- getByPlaceholder()
- getByTestId()
- locator()

Locators are central to Playwright's auto-waiting and retry behavior.

22. What is the difference between a locator and a selector?

A selector is a way of describing how to find an element, such as:

```
#login
```

Example:

```
const loginButton = page.getByRole('button', {

 name: 'Login'

});
```

The locator can then be used for actions and assertions.

23. What are the recommended Playwright locators?

Locator	**Best use**
getByRole()	Accessible UI elements
getByLabel()	Form controls
getByText()	Visible text
getByPlaceholder()	Inputs with placeholders
getByAltText()	Images
getByTitle()	Elements with titles
getByTestId()	Explicit testing contracts

25. When should you use getByTestId()?

Use getByTestId() when an element has a stable testing attribute such as:

```
<button data-testid="submit-order">
  Submit Order
</button>
```

Then:

```
await page.getByTestId('submit-order').click();
```

It is particularly useful when the UI text or structure changes frequently.

26. Can Playwright use CSS selectors?

Yes.

Example:

```
await page.locator('#username').fill('admin');
```

CSS selectors are supported, but for many situations Playwright's user-facing locators are preferable because they can be more resilient and readable.

27. Can Playwright use XPath?

Yes. Playwright supports XPath selectors.

Example:

```
await page.locator('#username').fill('admin');
```

However, XPath should not automatically be the first choice. A role-based or other user-facing locator may be easier to maintain.

28. How do you locate an input by its label?

```
await page.getByLabel('Email').fill('user@example.com');
```

This is useful when the input is associated with a visible form label.

29. How do you locate a button by its role?

```
await page.getByRole('button', {
  name: 'Login'
}).click();
```

The role identifies the element type, while the accessible name identifies the specific button.

30. What happens if a locator matches multiple elements?

For actions that require a unique target, Playwright generally expects the locator to resolve to one element. If it resolves to multiple matching elements, the action can fail due to strictness.

Instead of using a broad locator, narrow it down using:

- filter()
- first()
- last()
- nth()

But the best solution is usually to create a locator that uniquely identifies the intended element.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut elit tellus, luctus nec ullamcorper mattis, pulvinar dapibus leo.

Part 4: Auto-Waiting, Assertions and Synchronization

31. What is auto-waiting in Playwright?

Auto-waiting means Playwright automatically waits for an element to become actionable before performing certain actions.

For example, before click(), Playwright checks conditions such as visibility, stability, receiving events, and enabled state.

32. Why is auto-waiting important?

Modern web applications often load content dynamically.

Without proper synchronization, a test might attempt to click an element before it is ready.

Auto-waiting reduces the need for arbitrary delays and can make tests more reliable.

33. What are web-first assertions?

Web-first assertions automatically retry until the expected condition is met or the assertion times out.

Example:

```
await expect(
  page.getByText('Order successful')
).toBeVisible();
```

Instead of checking once, Playwright waits for the condition to become true.

34. What is the difference between expect() and a normal value check?

A Playwright assertion such as:

```
await expect(locator).toBeVisible();
```

is designed for asynchronous web conditions and automatically retries.

A normal JavaScript check such as:

```
if (value === 'Success')
```

does not provide Playwright's automatic retry behavior.

35. What is a timeout in Playwright?

A timeout defines how long Playwright waits for an operation or assertion before failing.

Different timeout settings can exist for:

- Tests
- Actions
- Assertions
- Navigation

Timeouts should be used thoughtfully rather than simply increased whenever a test fails.

36. Should you use waitForTimeout() regularly?

No.

Using arbitrary delays such as:

```
await page.waitForTimeout(5000);
```

can make tests slower and still unreliable.

Prefer:

- Locators
- Assertions
- Explicit event waiting
- Appropriate navigation waiting
- Application-specific conditions

37. How would you wait for an element to become visible?

```
await expect(
  page.getByRole('button', { name: 'Submit' })
).toBeVisible();
```

This is generally better than adding a fixed delay.

38. How do you verify a page title?

```
await expect(page).toHaveTitle('Dashboard');
```

The assertion waits for the expected condition instead of checking only once.

39. How do you verify the current URL?

```
await expect(page).toHaveURL(/dashboard/);
```

You can use either an exact URL or a regular expression depending on the requirement.

40. How do you verify text on a page?

```
await expect(
  page.getByRole('heading')
).toContainText('Dashboard');
```

This verifies the expected content while using Playwright's retryable assertion mechanism.

Part 5: Browser Actions and Web Elements

41. How do you enter text into an input?

```
await page.getByLabel('Username').fill('admin');
```

fill() is commonly used to replace the existing value with the specified text.

42. How do you click an element?

```
await page.getByRole('button', {
  name: 'Login'
}).click();
```

Playwright performs actionability checks before clicking.

43. How do you select an option from a dropdown?

For a standard HTML <select>:

```
await page.getByLabel('Country')
  .selectOption('India');
```

For custom dropdowns, you generally interact with the UI using appropriate locators.

44. How do you upload a file?

```
await page

 .getByLabel('Upload file')

 .setInputFiles('resume.pdf');
```

Playwright provides direct support for file uploads.

45. How do you download a file?

```
const downloadPromise = page.waitForEvent('download');

await page.getByText('Download').click();

const download = await downloadPromise;

await download.saveAs('output.pdf');
```

The important point is to start waiting for the download event before triggering the action

46. How do you handle browser dialogs?

```
page.on('dialog', async dialog => {

 await dialog.accept();

});
```

This can handle JavaScript dialogs such as:

- Alert
- Confirm
- Prompt

47. How do you handle checkboxes?

```
await page.getByLabel('Accept terms').check();
```

You can also verify the state:

```
await expect(

 page.getByLabel('Accept terms')

).toBeChecked();
```

48. How do you handle radio buttons?

```
await page.getByLabel('Male').check();
```

Using a label is usually clearer than depending on a complex CSS selector.

49. How do you hover over an element?

```
await page.getByText('Products').hover();
```

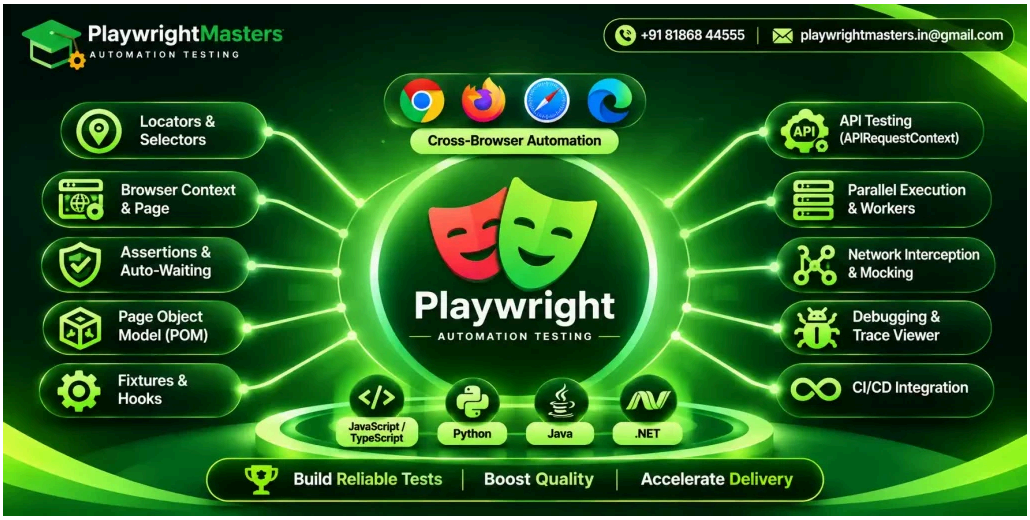
This is useful for menus, tooltips, and hover-based UI interactions.

50. How do you press keyboard keys?

```
await page.getByLabel('Search').press('Enter');
```

Playwright also supports keyboard combinations such as:

```
await page.keyboard.press('Control+A');
```



Part 6: Frames, Tabs, Popups and Windows

51. How do you handle an iframe in Playwright?

Use frameLocator() when interacting with elements inside an iframe.

Example:

```
const frame = page.frameLocator('#payment-frame');



await frame.getByLabel('Card number').fill('1234');
```

This allows you to locate elements within the frame.

52. What is FrameLocator?

FrameLocator provides a convenient way to locate elements inside an iframe.

Instead of manually retrieving the frame and then locating an element, you can chain operations:

```
page
```

```
.frameLocator('#login-frame')

.getByLabel('Username');
```

53. How do you handle a new tab?

```
const pagePromise = context.waitForEvent('page');

await page.getByText('Open Report').click();

const newPage = await pagePromise;

await newPage.waitForLoadState();
```

The important concept is to wait for the new page event before triggering the action.

54. How do you handle a popup?

```
const popupPromise = page.waitForEvent('popup');



await page.getByText('Open').click();



const popup = await popupPromise;
```

A popup belongs to the same browser context as its parent page.

55. What is the difference between a new Page and a new BrowserContext?

Page	BrowserContext
Represents a tab	Represents an isolated session
Shares context state	Has independent state
Useful for multiple tabs	Useful for test isolation
Lightweight	Provides session-level isolation

Part 7: Page Object Model and Framework Design

56. What is Page Object Model?

Page Object Model, or POM, is a design pattern where page-specific locators and actions are organized into classes or objects.

For example:

```
class LoginPage {

 constructor(private page: Page) {}



 username = this.page.getByLabel('Username');

 password = this.page.getByLabel('Password');

 loginButton = this.page.getByRole('button', {

  name: 'Login'

 });



 async login(user: string, pass: string) {

  await this.username.fill(user);

  await this.password.fill(pass);

  await this.loginButton.click();

 }

}
```

This keeps test scenarios cleaner and improves reuse.

57. Why use Page Object Model in Playwright?

POM can help with:

- Reusability
- Maintainability
- Separation of test logic and page interaction
- Centralized locators
- Cleaner test cases

It becomes especially useful as the automation suite grows.

58. Is Page Object Model mandatory in Playwright?

No.

Playwright does not require POM.

For a small test suite, simple test files may be sufficient. For larger projects, POM or another well-designed abstraction can make the framework easier to maintain.

59. What should you avoid in Page Object Model?

Avoid making page objects excessively complicated.

For example, a page object should not contain every possible business workflow in one huge class.

Keep responsibilities clear and reusable.

60. How would you design a scalable Playwright framework?

A scalable structure may contain:

```
tests/

pages/

fixtures/

utils/

test-data/

api/

config/

reports/
```

You can separate:

- Test scenarios
- Page interactions
- Fixtures
- Test data
- API utilities
- Configuration
- Reporting

The exact structure should depend on project size and team requirements.

Part 8: Fixtures and Hooks

61. What are fixtures in Playwright?

Fixtures provide reusable setup and resources for tests.

Examples include:

- test
- page
- context
- browser

You can also create custom fixtures for things such as:

- Login sessions
- Page objects
- Test data
- API clients

62. Why are fixtures useful?

Fixtures make it easier to centralize setup logic and reuse it across tests.

Instead of repeating login or initialization logic in every test, you can provide it through a fixture.

63. What is beforeEach()?

beforeEach() runs before each test in a test scope.

Example:

```
test.beforeEach(async ({ page }) => {
  await page.goto('/login');
});
```

It is useful for common setup.

64. What is afterEach()?

afterEach() runs after each test.

It can be used for cleanup or test-specific post-processing.

65. What is the difference between beforeAll() and beforeEach()?

Hook	Runs
beforeAll()	Once before tests in its scope
beforeEach()	Before every test

Use beforeAll() carefully because sharing state between tests can reduce test isolation.

66. What is afterAll()?

afterAll() runs once after the tests in its scope complete.

It can be used for final cleanup activities.

67. When should you create a custom fixture?

Create a custom fixture when multiple tests need the same setup or resource.

For example:

- authenticatedPage

- adminUser
- apiClient
- productPage

This can reduce repeated setup code.

68. How can fixtures improve test isolation?

Fixtures can create fresh resources for individual tests or workers.

For example, a fixture can provide a new page or context for each test, reducing unwanted state sharing.

Part 9: Authentication and Test Data

69. How does Playwright handle authentication?

Playwright can authenticate through UI flows or API requests and save authentication state.

Authentication state can include cookies and local storage.

The saved state can then be reused when creating browser contexts.

70. What is storageState?

storageState represents saved browser authentication-related state such as cookies and local storage.

Example:

```
await page.context().storageState({

 path: ‘auth.json’

});
```

The state can later be supplied when creating a context.

Playwright officially supports reusing authentication state to avoid logging in repeatedly.

71. Why reuse authentication state?

It can:

- Reduce execution time
- Avoid repeated UI login
- Simplify authenticated tests
- Make test setup more efficient

However, authentication state files should be treated as sensitive because they may contain credentials or session information.

72. How would you test an application with multiple user roles?

Create separate authentication states or fixtures for the required roles.

For example:

- admin
- manager
- customer

Then configure tests to use the appropriate state.

This avoids repeatedly logging in through the UI.

73. How would you test two users interacting with the same application?

Use separate BrowserContexts.

For example:

- Context A -> User 1
- Context B -> User 2

Each context can have its own authentication state and cookies.

74. How do you handle test data?

Test data can be managed through:

- Fixtures
- JSON files
- Factory functions
- API setup
- Database utilities
- Environment variables

The right approach depends on how the application is structured.

75. Why should tests avoid depending on previous tests?

Tests should generally be independent.

If Test B only works because Test A ran first, failures become harder to diagnose and parallel execution becomes difficult.

Independent tests are easier to retry, debug, and maintain.

Part 10: API Testing and Network Interception

Playwright can interact with REST APIs through APIRequestContext, including using API calls to prepare application state or verify server-side