AnnotationsPlaywright Test supports test annotations to deal with failures, flakiness, skip, focus and tag tests:test.skip() marks the test as irrelevant. Playwright Test does not run such a test. Use this annotation when the test is not applicable in some configuration.test.fail() marks the test as failing. Playwright Test will run this test and ensure it does indeed fail. If the test does not fail, Playwright Test will complain.test.fixme() marks the test as failing. Playwright Test will not run this test, as opposed to the fail annotation. Use fixme when running the test is slow or crashes.test.slow() marks the test as slow and triples the test timeout.Annotations can be used on a single test or a group of tests. Annotations can be conditional, in which case they apply when the condition is truthy. Annotations may depend on test fixtures. There could be multiple annotations on the same test, possibly in different configurations.Focus a test You can focus some tests. When there are focused tests, only these tests run.test.only('focus this test', async ({ page }) => { // Run only focused tests in the entire project.});Skip a test Mark a test as skipped.test.skip('skip this test', async ({ page }) => { // This test is not run});Conditionally skip a test You can skip certain test based on the condition.test('skip this test', async ({ page, browserName }) => { test.skip(browserName === 'firefox', 'Still working on it');});Group tests You can group tests to give them a logical name or to scope before/after hooks to the group.import { test, expect } from '@playwright/test';test.describe('two tests', () => { test('one', async ({ page }) => {   // ... }); test('two', async ({ page }) => {   // ... });});Tag tests Sometimes you want to tag your tests as @fast or @slow and only run the tests that have the certain tag. We recommend that you use the --grep and --grep-invert command line flags for that:import { test, expect } from '@playwright/test';test('Test login page @fast', async ({ page }) => { // ...});test('Test full report @slow', async ({ page }) => { // ...});You will then be able to run only that test:npx playwright test --grep @fastOr if you want the opposite, you can skip the tests with a certain tag:npx playwright test --grep-invert @slowTo run tests containing either tag (logical OR operator):npx playwright test --grep "@fast|@slow"Or run tests containing both tags (logical AND operator) using regex lookaheads:npx playwright test --grep "(?=.*@fast)(?=.*@slow)"Conditionally skip a group of tests For example, you can run a group of tests just in Chromium by passing a callback.example.spec.tstest.describe('chromium only', () => { test.skip(({ browserName }) => browserName !== 'chromium', 'Chromium only!'); test.beforeAll(async () => {   // This hook is only run in Chromium. }); test('test 1', async ({ page }) => {   // This test is only run in Chromium. }); test('test 2', async ({ page }) => {   // This test is only run in Chromium. });});Use fixme in beforeEach hook To avoid running beforeEach hooks, you can put annotations in the hook itself.example.spec.tstest.beforeEach(async ({ page, isMobile }) => { test.fixme(isMobile, 'Settings page does not work in mobile yet'); await page.goto('http://localhost:3000/settings');});test('user profile', async ({ page }) => { await page.getByText('My Profile').click(); // ...});Custom annotations It's also possible to add custom metadata in the form of annotations to your tests. Annotations are key/value pairs accessible via test.info().annotations. Many reporters show annotations, for example 'html'.example.spec.tstest('user profile', async ({ page }) => { test.info().annotations.push({ type: 'issue', description: 'https://github.com/microsoft/playwright/issues/<some-issue>' }); // ...});

Command lineHere are the most common options available in the command line.Run all the testsnpx playwright testRun a single test filenpx playwright test tests/todo-page.spec.tsRun a set of test filesnpx playwright test tests/todo-page/ tests/landing-page/Run files that have my-spec or my-spec-2 in the file namenpx playwright test my-spec my-spec-2Run tests that are in line 42 in my-spec.tsnpx playwright test my-spec.ts:42Run the test with the titlenpx playwright test -g "add a todo item"Run tests in headed browsersnpx playwright test --headedRun all the tests against a specific projectnpx playwright test --project=chromiumDisable parallelizationnpx playwright test --workers=1Choose a reporternpx playwright test --reporter=dotRun in debug mode with Playwright Inspectornpx playwright test --debug-- Run tests in interactive UI mode, with a built-in watch mode (Preview)npx playwright test --uiAsk for helpnpx playwright test --helpReference Complete set of Playwright Test options is available in the configuration file. Following options can be passed to a command line and take priority over the configuration file:OptionDescription--headedRun tests in headed browsers. Useful for debugging.--browserRun test in a specific browser. Available options are "chromium", "firefox", "webkit" or "all" to run tests in all three browsers at the same time.--debugRun tests with Playwright Inspector. Shortcut for PWDEBUG=1 environment variable and --timeout=0 --max-failures=1 --headed --workers=1 options.-c <file> or --config <file>Configuration file. If not passed, defaults to playwright.config.ts or playwright.config.js in the current directory.-c <dir> or --config <dir>Configuration file. If not passed, defaults to playwright.config.ts or playwright.config.js in the current directory.--forbid-onlyWhether to disallow test.only. Useful on CI.-g <grep> or --grep <grep>Only run tests matching this regular expression. For example, this will run 'should add to cart' when passed -g "add to cart".--grep-invert <grep>Only run tests not matching this regular expression. The opposite of --grep.--global-timeout <number>Total timeout for the whole test run in milliseconds. By default, there is no global timeout. Learn more about various timeouts.--listlist all the tests, but do not run them.--max-failures <N> or -xStop after the first N test failures. Passing -x stops after the first failure.--output <dir>Directory for artifacts produced by tests, defaults to test-results.--pass-with-no-testsAllows the test suite to pass when no files are found.--project <name>Only run tests from one of the specified projects. Defaults to running all projects defined in the configuration file.--quietWhether to suppress stdout and stderr from the tests.--repeat-each <N>Run each test N times, defaults to one.--reporter <reporter>Choose a reporter: minimalist dot, concise line or detailed list. See reporters for more information.--retries <number>The maximum number of retries for flaky tests, defaults to zero (no retries).--shard <shard>Shard tests and execute only selected shard, specified in the form current/all, 1-based, for example 3/5.--timeout <number>Maximum timeout in milliseconds for each test, defaults to 30 seconds. Learn more about various timeouts.--trace <mode>Force tracing mode, can be on, off, on-first-retry, on-all-retries, retain-on-failure--ignore-snapshotsWhether to ignore snapshots. Use this when snapshot expectations are known to be different, e.g. running tests on Linux against Windows screenshots.--update-snapshots or -uWhether to update snapshots with actual results instead of comparing them. Use this when snapshot expectations have changed.--workers <number> or -j <number>The maximum number of concurrent worker processes that run in parallel.
EmulationWith Playwright you can test your app on any browser as well as emulate a

real device such as a mobile phone or tablet. Simply configure the devices you would like to emulate and Playwright will simulate the browser behavior such as "userAgent", "screenSize", "viewport" and if it "hasTouch" enabled. You can also emulate the "geolocation", "locale" and "timezone" for all tests or for a specific test as well as set the "permissions" to show notifications or change the "colorScheme".Devices Playwright comes with a registry of device parameters using playwright.devices for selected desktop, tablet and mobile devices. It can be used to simulate browser behavior for a specific device such as user agent, screen size, viewport and if it has touch enabled. All tests will run with the specified device parameters. TestLibraryplaywright.config.tsimport { defineConfig, devices } from '@playwright/test'; // import devicesexport default defineConfig({ projects: [{ name: 'chromium', use: { ...devices['Desktop Chrome'], }, }, { name: 'Mobile Safari', use: { ...devices['iPhone 13'], }, },],});const { chromium, devices } = require('playwright');const browser = await chromium.launch();const iphone13 = devices['iPhone 13'];const context = await browser.newContext({ ...iphone13,});Viewport The viewport is included in the device but you can override it for some tests with page.setViewportSize().TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Viewport used for all pages in the context. viewport: { width: 1280, height: 720 }, },});// Create context with given viewportconst context = await browser.newContext({ viewport: { width: 1280, height: 1024 }});Test file:TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ viewport: { width: 1600, height: 1200 },});test('my test', async ({ page }) => { // ...});// Create context with given viewportconst context = await browser.newContext({ viewport: { width: 1280, height: 1024 }});// Resize viewport for individual pageawait page.setViewportSize({ width: 1600, height: 1200 });// Emulate high-DPIconst context = await browser.newContext({ viewport: { width: 2560, height: 1440 }, deviceScaleFactor: 2,});The same works inside a test file.TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.describe('specific viewport block', () => { test.use({ viewport: { width: 1600, height: 1200 } }); test('my test', async ({ page }) => { // ... });});// Create context with given viewportconst context = await browser.newContext({ viewport: { width: 1600, height: 1200 }});const page = await context.newPage();isMobile Whether the meta viewport tag is taken into account and touch events are enabled.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { isMobile: false, },});Locale & Timezone Emulate the user Locale and Timezone which can be set globally for all tests in the config and then overridden for particular tests.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Emulates the user locale. locale: 'en-GB', // Emulates the user timezone. timezoneId: 'Europe/Paris', },});TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ locale: 'de-DE', timezoneId: 'Europe/Berlin',});test('my test for de lang in Berlin timezone', async ({ page }) => { await page.goto('https://www.bing.com'); // ...});const context = await browser.newContext({ locale: 'de-DE', timezoneId: 'Europe/Berlin',});Permissions Allow app to show system notifications.TestLibraryplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Grants specified permissions to the browser context. permissions: ['notifications'], },});const context =

await browser.newContext({ permissions: ['notifications'],});Allow notifications for a specific domain.TestLibrarytests/example.spec.tsimport { test } from '@playwright/test';test.beforeEach(async ({ context }) => { // Runs before each test and signs in each page. await context.grantPermissions(['notifications'], { origin: 'https://skype.com' });});test('first', async ({ page }) => { // page has notifications permission for https://skype.com.});await context.grantPermissions(['notifications'], { origin: 'https://skype.com' });Revoke all permissions with browserContext.clearPermissions().// Libraryawait context.clearPermissions();Geolocation Grant "geolocation" permissions and set geolocation to a specific area.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: {   // Context geolocation geolocation: { longitude: 12.492507, latitude: 41.889938 },   permissions: ['geolocation'], },});TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ geolocation: { longitude: 41.890221, latitude: 12.492348 }, permissions: ['geolocation'],});test('my test with geolocation', async ({ page }) => { // ...});const context = await browser.newContext({ geolocation: { longitude: 41.890221, latitude: 12.492348 }, permissions: ['geolocation']});Change the location later:TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ geolocation: { longitude: 41.890221, latitude: 12.492348 }, permissions: ['geolocation'],});test('my test with geolocation', async ({ page, context }) => { // overwrite the location for this test await context.setGeolocation({ longitude: 48.858455, latitude: 2.294474 });});await context.setGeolocation({ longitude: 48.858455, latitude: 2.294474 });Note you can only change geolocation for all pages in the context.Color Scheme and Media Emulate the users "colorScheme". Supported values are 'light', 'dark', 'no-preference'. You can also emulate the media type with page.emulateMedia().playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: {   colorScheme: 'dark', },});TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ colorScheme: 'dark' // or 'light'});test('my test with dark mode', async ({ page }) => { // ...});// Create context with dark modeconst context = await browser.newContext({ colorScheme: 'dark' // or 'light'});// Create page with dark modeconst page = await browser.newPage({ colorScheme: 'dark' // or 'light'});// Change color scheme for the pageawait page.emulateMedia({ colorScheme: 'dark' });// Change media for pageawait page.emulateMedia({ media: 'print' });User Agent The User Agent is included in the device and therefore you will rarely need to change it however if you do need to test a different user agent you can override it with the userAgent property.TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ userAgent: 'My user agent' });test('my user agent test', async ({ page }) => { // ...});const context = await browser.newContext({ userAgent: 'My user agent'});Offline Emulate the network being offline.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: {   offline: true },});JavaScript Enabled Emulate a user scenario where JavaScript is disabled.TestLibrarytests/example.spec.tsimport { test, expect } from '@playwright/test';test.use({ javaScriptEnabled: false });test('test with no JavaScript', async ({ page }) => { // ...});const context = await browser.newContext({ javaScriptEnabled: false});
FixturesPlaywright Test is based on the concept of test fixtures. Test fixtures are used to establish environment for each test, giving the test everything it needs and nothing else.

Test fixtures are isolated between tests. With fixtures, you can group tests based on their meaning, instead of their common setup.Built-in fixtures You have already used test fixtures in your first test.import { test, expect } from '@playwright/test';test('basic test', async ({ page }) => { await page.goto('https://playwright.dev/'); await expect(page).toHaveTitle(/Playwright/);});The { page } argument tells Playwright Test to setup the page fixture and provide it to your test function.Here is a list of the pre-defined fixtures that you are likely to use most of the time:FixtureTypeDescriptionpagePageIsolated page for this test run.contextBrowserContextIsolated context for this test run. The page fixture belongs to this context as well. Learn how to configure context.browserBrowserBrowsers are shared across tests to optimize resources. Learn how to configure browser.browserNamestringThe name of the browser currently running the test. Either chromium, firefox or webkit.requestAPIRequestContextIsolated APIRequestContext instance for this test run.Without fixtures Here is how typical test environment setup differs between traditional test style and the fixture-based one.TodoPage is a class that helps interacting with a "todo list" page of the web app, following the Page Object Model pattern. It uses Playwright's page internally.Click to expand the code for the TodoPageTypeScriptJavaScripttodo-page.tsimport type { Page, Locator } from '@playwright/test';export class TodoPage { private readonly inputBox: Locator; private readonly todoItems: Locator; constructor(public readonly page: Page) {   this.inputBox = this.page.locator('input.new-todo');   this.todoItems = this.page.getByTestId('todo-item'); } async goto() {   await this.page.goto('https://demo.playwright.dev/todomvc/'); } async addToDo(text: string) {   await this.inputBox.fill(text);   await this.inputBox.press('Enter'); } async remove(text: string) {   const todo = this.todoItems.filter({ hasText: text });   await todo.hover();   await todo.getByLabel('Delete').click(); } async removeAll() {   while ((await this.todoItems.count()) > 0) {   await this.todoItems.first().hover();   await this.todoItems.getByLabel('Delete').first().click();   } }}todo-page.jsexport class TodoPage { /**   * @param {import('@playwright/test').Page} page   */ constructor(page) {   this.page = page;   this.inputBox = this.page.locator('input.new-todo');   this.todoItems = this.page.getByTestId('todo-item'); } async goto() {   await this.page.goto('https://demo.playwright.dev/todomvc/'); } /**   * @param {string} text   */ async addToDo(text) {   await this.inputBox.fill(text);   await this.inputBox.press('Enter'); } /**   * @param {string} text   */ async remove(text) {   const todo = this.todoItems.filter({ hasText: text });   await todo.hover();   await todo.getByLabel('Delete').click(); } async removeAll() {   while ((await this.todoItems.count()) > 0) {   await this.todoItems.first().hover();   await this.todoItems.getByLabel('Delete').first().click();   } }}todo.spec.tsconst { test } = require('@playwright/test');const { TodoPage } = require('./todo-page');test.describe('todo tests', () => { let todoPage; test.beforeEach(async ({ page }) => {   todoPage = new TodoPage(page);   await todoPage.goto();   await todoPage.addToDo('item1');   await todoPage.addToDo('item2'); }); test.afterEach(async () => {   await todoPage.removeAll(); }); test('should add an item', async () => {   await todoPage.addToDo('my item');   // ... }); test('should remove an item', async () => {   await todoPage.remove('item1');   // ... });});With fixtures Fixtures have a number of advantages over before/after hooks:Fixtures encapsulate setup and

teardown in the same place so it is easier to write.Fixtures are reusable between test files - you can define them once and use in all your tests. That's how Playwright's built-in page fixture works.Fixtures are on-demand - you can define as many fixtures as you'd like, and Playwright Test will setup only the ones needed by your test and nothing else.Fixtures are composable - they can depend on each other to provide complex behaviors.Fixtures are flexible. Tests can use any combinations of the fixtures to tailor precise environment they need, without affecting other tests.Fixtures simplify grouping. You no longer need to wrap tests in describes that set up environment, and are free to group your tests by their meaning instead.Click to expand the code for the TodoPageTypeScriptJavaScripttodo-page.tsimport type { Page, Locator } from '@playwright/test';export class TodoPage { private readonly inputBox: Locator; private readonly todoItems: Locator; constructor(public readonly page: Page) {   this.inputBox = this.page.locator('input.new-todo');   this.todoItems = this.page.getByTestId('todo-item'); } async goto() {   await this.page.goto('https://demo.playwright.dev/todomvc/'); } async addToDo(text: string) {   await this.inputBox.fill(text);   await this.inputBox.press('Enter'); } async remove(text: string) {   const todo = this.todoItems.filter({ hasText: text });   await todo.hover();   await todo.getByLabel('Delete').click(); } async removeAll() {   while ((await this.todoItems.count()) > 0) {   await this.todoItems.first().hover();   await this.todoItems.getByLabel('Delete').first().click();   } }}todo-page.jsexport class TodoPage { /**   * @param {import('@playwright/test').Page} page   */ constructor(page) {   this.page = page;   this.inputBox = this.page.locator('input.new-todo');   this.todoItems = this.page.getByTestId('todo-item'); } async goto() {   await this.page.goto('https://demo.playwright.dev/todomvc/'); } /**   * @param {string} text   */ async addToDo(text) {   await this.inputBox.fill(text);   await this.inputBox.press('Enter'); } /**   * @param {string} text   */ async remove(text) {   const todo = this.todoItems.filter({ hasText: text });   await todo.hover();   await todo.getByLabel('Delete').click(); } async removeAll() {   while ((await this.todoItems.count()) > 0) {   await this.todoItems.first().hover();   await this.todoItems.getByLabel('Delete').first().click();   } }}TypeScriptJavaScriptexample.spec.tsimport { test as base } from '@playwright/test';import { TodoPage } from './todo-page';// Extend basic test by providing a "todoPage" fixture.const test = base.extend<{ todoPage: TodoPage }>({ todoPage: async ({ page }, use) => {   const todoPage = new TodoPage(page);   await todoPage.goto();   await todoPage.addToDo('item1');   await todoPage.addToDo('item2');   await use(todoPage);   await todoPage.removeAll(); },});test('should add an item', async ({ todoPage }) => { await todoPage.addToDo('my item'); // ...});test('should remove an item', async ({ todoPage }) => { await todoPage.remove('item1'); // ...});todo.spec.jsconst base = require('@playwright/test');const { TodoPage } = require('./todo-page');// Extend basic test by providing a "todoPage" fixture.const test = base.test.extend({ todoPage: async ({ page }, use) => {   const todoPage = new TodoPage(page);   await todoPage.goto();   await todoPage.addToDo('item1');   await todoPage.addToDo('item2');   await use(todoPage);   await todoPage.removeAll(); },});test('should add an item', async ({ todoPage }) => { await todoPage.addToDo('my item'); // ...});test('should remove an item', async ({ todoPage })

=> { await todoPage.remove('item1'); // ...});Creating a fixture To create your own fixture, use test.extend() to create a new test object that will include it.Below we create two fixtures todoPage and settingsPage that follow the Page Object Model pattern.Click to expand the code for the TodoPage and SettingsPageTypeScriptJavaScripttodo-page.tsimport type { Page, Locator } from '@playwright/test';export class TodoPage { private readonly inputBox: Locator; private readonly todoItems: Locator; constructor(public readonly page: Page) {   this.inputBox = this.page.locator('input.new-todo');   this.todoItems = this.page.getByTestId('todo-item'); } async goto() {   await this.page.goto('https://demo.playwright.dev/todomvc/'); } async addToDo(text: string) {   await this.inputBox.fill(text);   await this.inputBox.press('Enter'); } async remove(text: string) {   const todo = this.todoItems.filter({ hasText: text });   await todo.hover();   await todo.getByLabel('Delete').click(); } async removeAll() {   while ((await this.todoItems.count()) > 0) {   await this.todoItems.first().hover();   await this.todoItems.getByLabel('Delete').first().click();   } }}todo-page.jsexport class TodoPage { /**   * @param {import('@playwright/test').Page} page   */ constructor(page) {   this.page = page;   this.inputBox = this.page.locator('input.new-todo');   this.todoItems = this.page.getByTestId('todo-item'); } async goto() {   await this.page.goto('https://demo.playwright.dev/todomvc/'); } /**   * @param {string} text   */ async addToDo(text) {   await this.inputBox.fill(text);   await this.inputBox.press('Enter'); } /**   * @param {string} text   */ async remove(text) {   const todo = this.todoItems.filter({ hasText: text });   await todo.hover();   await todo.getByLabel('Delete').click(); } async removeAll() {   while ((await this.todoItems.count()) > 0) {   await this.todoItems.first().hover();   await this.todoItems.getByLabel('Delete').first().click();   } }}SettingsPage is similar:TypeScriptJavaScriptsettings-page.tsimport type { Page } from '@playwright/test';export class SettingsPage { constructor(public readonly page: Page) { } async switchToDarkMode() {   // ... }}settings-page.jsexport class SettingsPage { /**   * @param {import('@playwright/test').Page} page   */ constructor(page) {   this.page = page; } async switchToDarkMode() {   // ... }}TypeScriptJavaScriptmy-test.tsimport { test as base } from '@playwright/test';import { TodoPage } from './todo-page';import { SettingsPage } from './settings-page';// Declare the types of your fixtures.type MyFixtures = { todoPage: TodoPage; settingsPage: SettingsPage;};// Extend base test by providing "todoPage" and "settingsPage".// This new "test" can be used in multiple test files, and each of them will get the fixtures.export const test = base.extend<MyFixtures>({ todoPage: async ({ page }, use) => {   // Set up the fixture.   const todoPage = new TodoPage(page);   await todoPage.goto();   await todoPage.addToDo('item1');   await todoPage.addToDo('item2');   // Use the fixture value in the test.   await use(todoPage);   // Clean up the fixture.   await todoPage.removeAll(); }, settingsPage: async ({ page }, use) => {   await use(new SettingsPage(page)); },});export { expect } from '@playwright/test';my-test.jsconst base = require('@playwright/test');const { TodoPage } = require('./todo-page');const { SettingsPage } = require('./settings-page');// Extend base test by providing "todoPage" and "settingsPage".// This new "test" can be used in multiple test files, and each of them will get the fixtures.exports.test = base.test.extend({ todoPage: async ({ page }, use) => {   // Set up the fixture.   const todoPage = new TodoPage(page);   await todoPage.goto();   await todoPage.addToDo('item1');   await

todoPage.addToDo('item2'); // Use the fixture value in the test. await use(todoPage); // Clean up the fixture. await todoPage.removeAll(); }, settingsPage: async ({ page }, use) => { await use(new SettingsPage(page)); },});exports.expect = base.expect;noteCustom fixture names should start with a letter or underscore, and can contain only letters, numbers, underscores.Using a fixture Just mention fixture in your test function argument, and test runner will take care of it. Fixtures are also available in hooks and other fixtures. If you use TypeScript, fixtures will have the right type.Below we use the todoPage and settingsPage fixtures defined above.TypeScriptJavaScriptimport { test, expect } from './my-test';test.beforeEach(async ({ settingsPage }) => { await settingsPage.switchToDarkMode();});test('basic test', async ({ todoPage, page }) => { await todoPage.addToDo('something nice'); await expect(page.getByTestId('todo-title')).toContainText(['something nice']);});const { test, expect } = require('./my-test');test.beforeEach(async ({ settingsPage }) => { await settingsPage.switchToDarkMode();});test('basic test', async ({ todoPage, page }) => { await todoPage.addToDo('something nice'); await expect(page.getByTestId('todo-title')).toContainText(['something nice']);});Overriding fixtures In addition to creating your own fixtures, you can also override existing fixtures to fit your needs. Consider the following example which overrides the page fixture by automatically navigating to some baseURL:import { test as base } from '@playwright/test';export const test = base.extend({ page: async ({ baseURL, page }, use) => { await page.goto(baseURL); await use(page); },});Notice that in this example, the page fixture is able to depend on other built-in fixtures such as testOptions.baseURL. We can now configure baseURL in the configuration file, or locally in the test file with test.use().example.spec.tstest.use({ baseURL: 'https://playwright.dev' });Fixtures can also be overridden where the base fixture is completely replaced with something different. For example, we could override the testOptions.storageState fixture to provide our own data.import { test as base } from '@playwright/test';export const test = base.extend({ storageState: async ({}, use) => { const cookie = await getAuthCookie(); await use({ cookies: [cookie] }); },});Worker-scoped fixtures Playwright Test uses worker processes to run test files. Similarly to how test fixtures are set up for individual test runs, worker fixtures are set up for each worker process. That's where you can set up services, run servers, etc. Playwright Test will reuse the worker process for as many test files as it can, provided their worker fixtures match and hence environments are identical.Below we'll create an account fixture that will be shared by all tests in the same worker, and override the page fixture to login into this account for each test. To generate unique accounts, we'll use the workerInfo.workerIndex that is available to any test or fixture. Note the tuple-like syntax for the worker fixture - we have to pass {scope: 'worker'} so that test runner sets up this fixture once per worker.TypeScriptJavaScriptmy-test.tsimport { test as base } from '@playwright/test';type Account = { username: string; password: string;};// Note that we pass worker fixture types as a second template parameter.export const test = base.extend<{}, { account: Account }>({ account: [async ({ browser }, use, workerInfo) => { // Unique username. const username = 'user' + workerInfo.workerIndex; const password = 'verysecure'; // Create the account with Playwright. const page = await browser.newPage(); await page.goto('/signup'); await page.getByLabel('User

Name').fill(username); await page.getByLabel('Password').fill(password); await page.getByText('Sign up').click(); // Make sure everything is ok. await expect(page.getByTestId('result')).toHaveText('Success'); // Do not forget to cleanup. await page.close(); // Use the account value. await use({ username, password }); }, { scope: 'worker' }], page: async ({ page, account }, use) => { // Sign in with our account. const { username, password } = account; await page.goto('/signin'); await page.getByLabel('User Name').fill(username); await page.getByLabel('Password').fill(password); await page.getByText('Sign in').click(); await expect(page.getByTestId('userinfo')).toHaveText(username); // Use signed-in page in the test. await use(page); },});export { expect } from '@playwright/test';my-test.jsconst base = require('@playwright/test');exports.test = base.test.extend({ account: [async ({ browser }, use, workerInfo) => { // Unique username. const username = 'user' + workerInfo.workerIndex; const password = 'verysecure'; // Create the account with Playwright. const page = await browser.newPage(); await page.goto('/signup'); await page.getByLabel('User Name').fill(username); await page.getByLabel('Password').fill(password); await page.getByText('Sign up').click(); // Make sure everything is ok. await expect(page.locator('#result')).toHaveText('Success'); // Do not forget to cleanup. await page.close(); // Use the account value. await use({ username, password }); }, { scope: 'worker' }], page: async ({ page, account }, use) => { // Sign in with our account. const { username, password } = account; await page.goto('/signin'); await page.getByLabel('User Name').fill(username); await page.getByLabel('Password').fill(password); await page.getByText('Sign in').click(); await expect(page.getByTestId('userinfo')).toHaveText(username); // Use signed-in page in the test. await use(page); },});exports.expect = base.expect;Automatic fixtures Automatic fixtures are set up for each test/worker, even when the test does not list them directly. To create an automatic fixture, use the tuple syntax and pass { auto: true }.Here is an example fixture that automatically attaches debug logs when the test fails, so we can later review the logs in the reporter. Note how it uses TestInfo object that is available in each test/fixture to retrieve metadata about the test being run.TypeScriptJavaScriptmy-test.tsimport * as debug from 'debug';import * as fs from 'fs';import { test as base } from '@playwright/test';export const test = base.extend<{ saveLogs: void }>({ saveLogs: [async ({}, use, testInfo) => { // Collecting logs during the test. const logs = []; debug.log = (...args) => logs.push(args.map(String).join('')); debug.enable('myserver'); await use(); // After the test we can check whether the test passed or failed. if (testInfo.status !== testInfo.expectedStatus) { // outputPath() API guarantees a unique file name. const logFile = testInfo.outputPath('logs.txt'); await fs.promises.writeFile(logFile, logs.join('\n'), 'utf8'); testInfo.attachments.push({ name: 'logs', contentType: 'text/plain', path: logFile }); } }, { auto: true }],});export { expect } from '@playwright/test';my-test.jsconst debug = require('debug');const fs = require('fs');const base = require('@playwright/test');exports.test = base.test.extend({ saveLogs: [async ({}, use, testInfo) => { // Collecting logs during the test. const logs = []; debug.log = (...args) => logs.push(args.map(String).join('')); debug.enable('myserver'); await use(); // After the test we can check whether the test passed or failed. if (testInfo.status !== testInfo.expectedStatus) { // outputPath() API guarantees a unique file name.

const logFile = testInfo.outputPath('logs.txt'); await fs.promises.writeFile(logFile, logs.join('\n'), 'utf8'); testInfo.attachments.push({ name: 'logs', contentType: 'text/plain', path: logFile }); } }, { auto: true }],});Fixture timeout By default, fixture shares timeout with the test. However, for slow fixtures, especially worker-scoped ones, it is convenient to have a separate timeout. This way you can keep the overall test timeout small, and give the slow fixture more time.TypeScriptJavaScriptimport { test as base, expect } from '@playwright/test';const test = base.extend<{ slowFixture: string }>({ slowFixture: [async ({}, use) => { // ... perform a slow operation ... await use('hello'); }, { timeout: 60000 }]});test('example test', async ({ slowFixture }) => { // ...});const { test: base, expect } = require('@playwright/test');const test = base.extend({ slowFixture: [async ({}, use) => { // ... perform a slow operation ... await use('hello'); }, { timeout: 60000 }]});test('example test', async ({ slowFixture }) => { // ...});Fixtures-options noteOverriding custom fixtures in the config file has changed in version 1.18. Learn more.Playwright Test supports running multiple test projects that can be separately configured. You can use "option" fixtures to make your configuration options declarative and type-checked. Learn more about parametrizing tests.Below we'll create a defaultItem option in addition to the todoPage fixture from other examples. This option will be set in configuration file. Note the tuple syntax and { option: true } argument.Click to expand the code for the TodoPageTypeScriptJavaScripttodo-page.tsimport type { Page, Locator } from '@playwright/test';export class TodoPage { private readonly inputBox: Locator; private readonly todoItems: Locator; constructor(public readonly page: Page) { this.inputBox = this.page.locator('input.new-todo'); this.todoItems = this.page.getByTestId('todo-item'); } async goto() { await this.page.goto('https://demo.playwright.dev/todomvc/'); } async addToDo(text: string) { await this.inputBox.fill(text); await this.inputBox.press('Enter'); } async remove(text: string) { const todo = this.todoItems.filter({ hasText: text }); await todo.hover(); await todo.getByLabel('Delete').click(); } async removeAll() { while ((await this.todoItems.count()) > 0) { await this.todoItems.first().hover(); await this.todoItems.getByLabel('Delete').first().click(); } }}todo-page.jsexport class TodoPage { /** * @param {import('@playwright/test').Page} page */ constructor(page) { this.page = page; this.inputBox = this.page.locator('input.new-todo'); this.todoItems = this.page.getByTestId('todo-item'); } async goto() { await this.page.goto('https://demo.playwright.dev/todomvc/'); } /** * @param {string} text */ async addToDo(text) { await this.inputBox.fill(text); await this.inputBox.press('Enter'); } /** * @param {string} text */ async remove(text) { const todo = this.todoItems.filter({ hasText: text }); await todo.hover(); await todo.getByLabel('Delete').click(); } async removeAll() { while ((await this.todoItems.count()) > 0) { await this.todoItems.first().hover(); await this.todoItems.getByLabel('Delete').first().click(); } }}TypeScriptJavaScriptmy-test.tsimport { test as base } from '@playwright/test';import { TodoPage } from './todo-page';// Declare your options to type-check your configuration.export type MyOptions = { defaultItem: string;};type MyFixtures = { todoPage: TodoPage;};// Specify both option and fixture types.export const test = base.extend<MyOptions & MyFixtures>({ // Define an option and provide a default value. // We can later override it in the config. defaultItem: ['Something nice', { option: true }], // Our "todoPage" fixture depends on the option. todoPage: async ({ page, defaultItem }, use) => { const todoPage = new

TodoPage(page); await todoPage.goto(); await todoPage.addToDo(defaultItem); await use(todoPage); await todoPage.removeAll(); },});export { expect } from '@playwright/test';my-test.jsconst base = require('@playwright/test');const { TodoPage } = require('./todo-page');exports.test = base.test.extend({ // Define an option and provide a default value. // We can later override it in the config. defaultItem: ['Something nice', { option: true }], // Our "todoPage" fixture depends on the option. todoPage: async ({ page, defaultItem }, use) => { const todoPage = new TodoPage(page); await todoPage.goto(); await todoPage.addToDo(defaultItem); await use(todoPage); await todoPage.removeAll(); },});exports.expect = base.expect;We can now use todoPage fixture as usual, and set the defaultItem option in the config file.TypeScriptJavaScriptplaywright.config.tsimport { defineConfig } from '@playwright/test';import type { MyOptions } from './my-test';export default defineConfig<MyOptions>({ projects: [{ name: 'shopping', use: { defaultItem: 'Buy milk' }, }, { name: 'wellbeing', use: { defaultItem: 'Exercise!' }, },]});playwright.config.ts// @ts-checkconst { defineConfig } = require('@playwright/test');module.exports = defineConfig({ projects: [{ name: 'shopping', use: { defaultItem: 'Buy milk' }, }, { name: 'wellbeing', use: { defaultItem: 'Exercise!' }, },]});Execution order Each fixture has a setup and teardown phase separated by the await use() call in the fixture. Setup is executed before the fixture is used by the test/hook, and teardown is executed when the fixture will not be used by the test/hook anymore.Fixtures follow these rules to determine the execution order:When fixture A depends on fixture B: B is always set up before A and teared down after A.Non-automatic fixtures are executed lazily, only when the test/hook needs them.Test-scoped fixtures are teared down after each test, while worker-scoped fixtures are only teared down when the worker process executing tests is shutdown.Consider the following example:TypeScriptJavaScriptimport { test as base } from '@playwright/test';const test = base.extend<{ testFixture: string, autoTestFixture: string, unusedFixture: string,}, { workerFixture: string, autoWorkerFixture: string,} >({ workerFixture: [async ({ browser }) => { // workerFixture setup... await use('workerFixture'); // workerFixture teardown... }, { scope: 'worker' }], autoWorkerFixture: [async ({ browser }) => { // autoWorkerFixture setup... await use('autoWorkerFixture'); // autoWorkerFixture teardown... }, { scope: 'worker', auto: true }], testFixture: [async ({ page, workerFixture }) => { // testFixture setup... await use('testFixture'); // testFixture teardown... }, { scope: 'test' }], autoTestFixture: [async () => { // autoTestFixture setup... await use('autoTestFixture'); // autoTestFixture teardown... }, { scope: 'test', auto: true }], unusedFixture: [async ({ page }) => { // unusedFixture setup... await use('unusedFixture'); // unusedFixture teardown... }, { scope: 'test' }],});test.beforeAll(async () => { /* ... */ });test.beforeEach(async ({ page }) => { /* ... */ });test('first test', async ({ page }) => { /* ... */ });test('second test', async ({ testFixture }) => { /* ... */ });test.afterEach(async () => { /* ... */ });test.afterAll(async () => { /* ... */ });const { test: base } = require('@playwright/test');const test = base.extend({ workerFixture: [async ({ browser }) => { // workerFixture setup... await use('workerFixture'); // workerFixture teardown... }, { scope: 'worker' }], autoWorkerFixture: [async ({ browser }) => { // autoWorkerFixture setup... await use('autoWorkerFixture'); // autoWorkerFixture teardown... }, { scope: 'worker', auto: true }], testFixture: [async ({ page, workerFixture }) => { // testFixture setup... await

use('testFixture'); // testFixture teardown... }, { scope: 'test' }], autoTestFixture: [async () => { // autoTestFixture setup... await use('autoTestFixture'); // autoTestFixture teardown... }, { scope: 'test', auto: true }], unusedFixture: [async ({ page }) => { // unusedFixture setup... await use('unusedFixture'); // unusedFixture teardown... }, { scope: 'test' }],});test.beforeAll(async () => { /* ... */ });test.beforeEach(async ({ page }) => { /* ... */ });test('first test', async ({ page }) => { /* ... */ });test('second test', async ({ testFixture }) => { /* ... */ });test.afterEach(async () => { /* ... */ });test.afterAll(async () => { /* ... */ });Normally, if all tests pass and no errors are thrown, the order of execution is as following.worker setup and beforeAll section:browser setup because it is required by autoWorkerFixture.autoWorkerFixture setup because automatic worker fixtures are always set up before anything else.beforeAll runs.first test section:autoTestFixture setup because automatic test fixtures are always set up before test and beforeEach hooks.page setup because it is required in beforeEach hook.beforeEach runs.first test runs.afterEach runs.page teardown because it is a test-scoped fixture and should be teared down after the test finishes.autoTestFixture teardown because it is a test-scoped fixture and should be teared down after the test finishes.second test section:autoTestFixture setup because automatic test fixtures are always set up before test and beforeEach hooks.page setup because it is required in beforeEach hook.beforeEach runs.workerFixture setup because it is required by testFixture that is required by the second test.testFixture setup because it is required by the second test.second test runs.afterEach runs.testFixture teardown because it is a test-scoped fixture and should be teared down after the test finishes.page teardown because it is a test-scoped fixture and should be teared down after the test finishes.autoTestFixture teardown because it is a test-scoped fixture and should be teared down after the test finishes.afterAll and worker teardown section:afterAll runs.workerFixture teardown because it is a workers-scoped fixture and should be teared down once at the end.autoWorkerFixture teardown because it is a workers-scoped fixture and should be teared down once at the end.browser teardown because it is a workers-scoped fixture and should be teared down once at the end.A few observations:page and autoTestFixture are set up and teared down for each test, as test-scoped fixtures.unusedFixture is never set up because it is not used by any tests/ hooks.testFixture depends on workerFixture and triggers its setup.workerFixture is lazily set up before the second test, but teared down once during worker shutdown, as a worker-scoped fixture.autoWorkerFixture is set up for beforeAll hook, but autoTestFixture is not.

Global setup and teardownThere are two ways to configure global setup and teardown: using a global setup file and setting it in the config under globalSetup or using project dependencies. With project dependencies, you define a project that runs before all other projects. This is the recommended way to configure global setup as with Project dependencies your HTML report will show the global setup, trace viewer will record a trace of the setup and fixtures can be used.Project Dependencies Project dependencies are a list of projects that need to run before the tests in another project run. They can be useful for configuring the global setup actions so that one project depends on this running first. Using dependencies allows global setup to produce traces and other artifacts.Setup First we add a new project with the name 'setup'. We then give it a testProject.testMatch property in order to match the file called

global.setup.ts:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: / global.setup\.ts/, }, // { // other project // }]});Then we add the testProject.dependencies property to our projects that depend on the setup project and pass into the array the name of of our dependency project, which we defined in the previous step:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: / global.setup\.ts/, }, { name: 'chromium', use: { ...devices['Desktop Chrome'] }, dependencies: ['setup'], },]});Setup Example This example will show you how to use project dependencies to create a global setup that logins into an application and saves the state in storage state. This is useful if you want to run multiple tests that require a sign sign-in state and you want to avoid login for each test.The setup project will write the storage state into an 'playwright/.auth/user.json' file next to your playwright.config. By exporting a const of STORAGE_STATE we can then easily share the location of the storage file between projects with the StorageState method. This applies the storage state on the browser context with its cookies and a local storage snapshot.In this example the 'logged in chromium' project depends on the setup project whereas the 'logged out chromium' project does not depend on the setup project, and does not use the storageState option.playwright.config.tsimport { defineConfig } from '@playwright/test';export const STORAGE_STATE = path.join(__dirname, 'playwright/.auth/user.json');export default defineConfig({ use: { baseURL: 'http://localhost:3000/', }, projects: [{ name: 'setup', testMatch: / global.setup\.ts/, }, { name: 'logged in chromium', testMatch: '**/ *.loggedin.spec.ts', dependencies: ['setup'], use: { ...devices['Desktop Chrome'], storageState: STORAGE_STATE, }, }, { name: 'logged out chromium', use: { ...devices['Desktop Chrome'] }, testIgnore: ['**/ *loggedin.spec.ts'] },],});We then create a setup test, stored at root level of your project, that logs in to an application and populates the context with the storage state after the login actions have been performed. By doing this you only have to log in once and the credentials will be stored in the STORAGE_STATE file, meaning you don't need to log in again for every test. Start by importing the STORAGE_STATE from the Playwright config file and then use this as the path to save your storage state to the page's context.global.setup.tsimport { test as setup, expect } from '@playwright/ test';import { STORAGE_STATE } from '../playwright.config';setup('do login', async ({ page }) => { await page.goto('/'); await page.getByLabel('User Name').fill('user'); await page.getByLabel('Password').fill('password'); await page.getByText('Sign in').click(); // Wait until the page actually signs in. await expect(page.getByText('Hello, user!')).toBeVisible(); await page.context().storageState({ path: STORAGE_STATE });});tests/menu.loggedin.spec.tsimport { test, expect } from '@playwright/test';test.beforeEach(async ({ page }) => { await page.goto('/');});test('menu', async ({ page }) => { // You are signed in!});For a more detailed example check out our blog post: A better global setup in Playwright reusing login with project dependencies or check the v1.31 release video to see the demo.Teardown You can teardown your setup by adding a testProject.teardown property to your setup project. This will run after all dependent projects have run.First we add a new project into the projects array and give it a name such as 'cleanup db'. We then

give it a testProject.testMatch property in order to match the file called global.teardown.ts:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [// { // setup project // }, { name: 'cleanup db', testMatch: /global.teardown\.ts/, }, // { // other project // }]});Then we add the testProject.teardown property to our setup project with the name 'cleanup db' which is the name we gave to our teardown project in the previous step:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup db', testMatch: /global\.setup\.ts/, teardown: 'cleanup db', }, { name: 'cleanup db', testMatch: /global.teardown\.ts/, }, // { // other project // }]});Teardown Example Start by creating a global.setup.ts file at the root level of your project. This will be used to seed the database with some data before all tests have run.global.setup.ts// seed the database with some dataThen create a global.teardown.ts file at the root level of your project. This will be used to delete the data from the database after all tests have run.global.teardown.ts// delete the data from the databaseIn the Playwright config file:Add a project into the projects array and give it a name such as 'setup db'. Give it a testProject.testMatch property in order to match the file called global.setup.ts.Create another project and give it a name such as 'cleanup db'. Give it a testProject.testMatch property in order to match the file called global.teardown.ts.Add the testProject.teardown property to our setup project with the name 'cleanup db' which is the name given to our teardown project in the previous step. Finally add the 'chromium' project with the testProject.dependencies on the 'setup db' project.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup db', testMatch: /global\.setup\.ts/, teardown: 'cleanup db', }, { name: 'cleanup db', testMatch: /global\.teardown\.ts/, }, { name: 'chromium', use: { ...devices['Desktop Chrome'] }, dependencies: ['setup db'], },]});Configure globalSetup and globalTeardown You can use the globalSetup option in the configuration file to set something up once before running all tests. The global setup file must export a single function that takes a config object. This function will be run once before all the tests.Similarly, use globalTeardown to run something once after all the tests. Alternatively, let globalSetup return a function that will be used as a global teardown. You can pass data such as port number, authentication tokens, etc. from your global setup to your tests using environment variables.noteUsing globalSetup and globalTeardown will not produce traces or artifacts. If you want to produce traces and artifacts, use project dependencies.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ globalSetup: require.resolve('./global-setup'), globalTeardown: require.resolve('./global-teardown'),});Example Here is a global setup example that authenticates once and reuses authentication state in tests. It uses the baseURL and storageState options from the configuration file.global-setup.tsimport { chromium, type FullConfig } from '@playwright/test';async function globalSetup(config: FullConfig) { const { baseURL, storageState } = config.projects[0].use; const browser = await chromium.launch(); const page = await browser.newPage(); await page.goto(baseURL!); await page.getByLabel('User Name').fill('user'); await page.getByLabel('Password').fill('password'); await page.getByText('Sign in').click(); await page.context().storageState({ path: storageState as string }); await

browser.close();}export default globalSetup;Specify globalSetup, baseURL and storageState in the configuration file.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ globalSetup: require.resolve('./global-setup'), use: { baseURL: 'http://localhost:3000/', storageState: 'state.json', },});Tests start already authenticated because we specify storageState that was populated by global setup.import { test } from '@playwright/test';test('test', async ({ page }) => { await page.goto('/'); // You are signed in!});You can make arbitrary data available in your tests from your global setup file by setting them as environment variables via process.env.global-setup.tsimport type { FullConfig } from '@playwright/test';async function globalSetup(config: FullConfig) { process.env.FOO = 'some data'; // Or a more complicated data structure as JSON: process.env.BAR = JSON.stringify({ some: 'data' });}export default globalSetup;Tests have access to the process.env properties set in the global setup.import { test } from '@playwright/test';test('test', async ({ page }) => { // environment variables which are set in globalSetup are only available inside test(). const { FOO, BAR } = process.env; // FOO and BAR properties are populated. expect(FOO).toEqual('some data'); const complexData = JSON.parse(BAR); expect(BAR).toEqual({ some: 'data' });});Capturing trace of failures during global setup In some instances, it may be useful to capture a trace of failures encountered during the global setup. In order to do this, you must start tracing in your setup, and you must ensure that you stop tracing if an error occurs before that error is thrown. This can be achieved by wrapping your setup in a try...catch block. Here is an example that expands the global setup example to capture a trace.global-setup.tsimport { chromium, type FullConfig } from '@playwright/test';async function globalSetup(config: FullConfig) { const { baseURL, storageState } = config.projects[0].use; const browser = await chromium.launch(); const context = await browser.newContext(); const page = await context.newPage(); try { await context.tracing.start({ screenshots: true, snapshots: true }); await page.goto(baseURL!); await page.getByLabel('User Name').fill('user'); await page.getByLabel('Password').fill('password'); await page.getByText('Sign in').click(); await context.storageState({ path: storageState as string }); await context.tracing.stop({ path: './test-results/setup-trace.zip', }); await browser.close(); } catch (error) { await context.tracing.stop({ path: './test-results/failed-setup-trace.zip', }); await browser.close(); throw error; }}export default globalSetup;

Parallelism and shardingPlaywright Test runs tests in parallel. In order to achieve that, it runs several worker processes that run at the same time. By default, test files are run in parallel. Tests in a single file are run in order, in the same worker process.You can configure tests using test.describe.configure to run tests in a single file in parallel.You can configure entire project to have all tests in all files to run in parallel using testProject.fullyParallel or testConfig.fullyParallel.To disable parallelism limit the number of workers to one.You can control the number of parallel worker processes and limit the number of failures in the whole test suite for efficiency.Worker processes All tests run in worker processes. These processes are OS processes, running independently, orchestrated by the test runner. All workers have identical environments and each starts its own browser.You can't communicate between the workers. Playwright Test reuses a single worker as much as it can to make testing faster, so multiple test files are usually run in a single worker one after another.Workers are always shutdown after a test

failure to guarantee pristine environment for following tests.Limit workers You can control the maximum number of parallel worker processes via command line or in the configuration file.From the command line:npx playwright test --workers 4In the configuration file:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Limit the number of workers on CI, use default locally workers: process.env.CI ? 2 : undefined,});Disable parallelism You can disable any parallelism by allowing just a single worker at any time. Either set workers: 1 option in the configuration file or pass --workers=1 to the command line.npx playwright test --workers=1Parallelize tests in a single file By default, tests in a single file are run in order. If you have many independent tests in a single file, you might want to run them in parallel with test.describe.configure().Note that parallel tests are executed in separate worker processes and cannot share any state or global variables. Each test executes all relevant hooks just for itself, including beforeAll and afterAll.import { test } from '@playwright/test';test.describe.configure({ mode: 'parallel' });test('runs in parallel 1', async ({ page }) => { /* ... */ });test('runs in parallel 2', async ({ page }) => { /* ... */ });Alternatively, you can opt-in all tests into this fully-parallel mode in the configuration file:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ fullyParallel: true,});You can also opt in for fully-parallel mode for just a few projects:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // runs all tests in all files of a specific project in parallel projects: [{ name: 'chromium', use: { ...devices['Desktop Chrome'] }, fullyParallel: true, },]});Serial mode You can annotate inter-dependent tests as serial. If one of the serial tests fails, all subsequent tests are skipped. All tests in a group are retried together.noteUsing serial is not recommended. It is usually better to make your tests isolated, so they can be run independently.import { test, type Page } from '@playwright/test';// Annotate entire file as serial.test.describe.configure({ mode: 'serial' });let page: Page;test.beforeAll(async ({ browser }) => { page = await browser.newPage();});test.afterAll(async () => { await page.close();});test('runs first', async () => { await page.goto('https://playwright.dev/');});test('runs second', async () => { await page.getByText('Get Started').click();});Shard tests between multiple machines Playwright Test can shard a test suite, so that it can be executed on multiple machines. For that, pass --shard=x/y to the command line. For example, to split the suite into three shards, each running one third of the tests:npx playwright test --shard=1/3npx playwright test --shard=2/3npx playwright test --shard=3/3That way your test suite completes 3 times faster.Limit failures and fail fast You can limit the number of failed tests in the whole test suite by setting maxFailures config option or passing --max-failures command line flag.When running with "max failures" set, Playwright Test will stop after reaching this number of failed tests and skip any tests that were not executed yet. This is useful to avoid wasting resources on broken test suites.Passing command line option:npx playwright test --max-failures=10Setting in the configuration file:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Limit the number of failures on CI to save resources maxFailures: process.env.CI ? 10 : undefined,});Worker index and parallel index Each worker process is assigned two ids: a unique worker index that starts with 1, and a parallel index that is between 0 and workers - 1. When a worker is restarted, for example after a failure, the new worker process has the same parallelIndex and a new workerIndex.You can read

an index from environment variables process.env.TEST_WORKER_INDEX and process.env.TEST_PARALLEL_INDEX, or access them through testInfo.workerIndex and testInfo.parallelIndex.Control test order Playwright Test runs tests from a single file in the order of declaration, unless you parallelize tests in a single file.There is no guarantee about the order of test execution across the files, because Playwright Test runs test files in parallel by default. However, if you disable parallelism, you can control test order by either naming your files in alphabetical order or using a "test list" file.Sort test files alphabetically When you disable parallel test execution, Playwright Test runs test files in alphabetical order. You can use some naming convention to control the test order, for example 001-user-signin-flow.spec.ts, 002-create-new-document.spec.ts and so on.Use a "test list" file dangerTests lists are discouraged and supported as a best-effort only. Some features such as VS Code Extension and tracing may not work properly with test lists.You can put your tests in helper functions in multiple files. Consider the following example where tests are not defined directly in the file, but rather in a wrapper function.feature-a.spec.tsimport { test, expect } from '@playwright/test';export default function createTests() { test('feature-a example test', async ({ page }) => {   // ... test goes here });}feature-b.spec.tsimport { test, expect } from '@playwright/test';export default function createTests() { test.use({ viewport: { width: 500, height: 500 } }); test('feature-b example test', async ({ page }) => {   // ... test goes here });}You can create a test list file that will control the order of tests - first run feature-b tests, then feature-a tests. Note how each test file is wrapped in a test.describe() block that calls the function where tests are defined. This way test.use() calls only affect tests from a single file.test.list.tsimport { test } from '@playwright/test';import featureBTests from './feature-b.spec.ts';import featureATests from './feature-a.spec.ts';test.describe(featureBTests);test.describe(featureATests);Now disable parallel execution by setting workers to one, and specify your test list file.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ workers: 1, testMatch: 'test.list.ts',});noteDo not define your tests directly in a helper file. This could lead to unexpected results because your tests are now dependent on the order of import/require statements. Instead, wrap tests in a function that will be explicitly called by a test list file, as in the example above. Parameterize testsYou can either parameterize tests on a test level or on a project level.Parameterized Tests example.spec.tsconst people = ['Alice', 'Bob'];for (const name of people) { test(`testing with ${name}`, async () => {   // ... }); // You can also do it with test.describe() or with multiple tests as long the test name is unique.} Parameterized Projects Playwright Test supports running multiple test projects at the same time. In the following example, we'll run two projects with different options.We declare the option person and set the value in the config. The first project runs with the value Alice and the second with the value Bob.TypeScriptJavaScriptmy-test.tsimport { test as base } from '@playwright/test';export type TestOptions = { person: string;};export const test = base.extend<TestOptions>({ // Define an option and provide a default value. // We can later override it in the config. person: ['John', { option: true }],});my-test.jsconst base = require('@playwright/test');exports.test = base.test.extend({ // Define an option and provide a default value. // We can later override it in the config. person: ['John', { option: true }],});We can use this option in the test, similarly to fixtures.example.spec.tsimport { test } from './my-test';test('test 1',

async ({ page, person }) => { await page.goto(`/index.html`); await expect(page.locator('#node')).toContainText(person); // ...});Now, we can run tests in multiple configurations by using projects.TypeScriptJavaScriptplaywright.config.tsimport { defineConfig } from '@playwright/test';import type { TestOptions } from './my-test';export default defineConfig<TestOptions>({ projects: [{   name: 'alice',   use: { person: 'Alice' },   },   {   name: 'bob',   use: { person: 'Bob' },   },]});playwright.config.ts// @ts-checkmodule.exports = defineConfig({ projects: [{   name: 'alice',   use: { person: 'Alice' },   },   {   name: 'bob',   use: { person: 'Bob' },   },]});We can also use the option in a fixture. Learn more about fixtures.TypeScriptJavaScriptmy-test.tsimport { test as base } from '@playwright/test';export type TestOptions = { person: string;};export const test = base.extend<TestOptions>({ // Define an option and provide a default value. // We can later override it in the config. person: ['John', { option: true }], // Override default "page" fixture. page: async ({ page, person }, use) => {   await page.goto('/chat');   // We use "person" parameter as a "name" for the chat room.   await page.getByLabel('User Name').fill(person);   await page.getByText('Enter chat room').click();   // Each test will get a "page" that already has the person name.   await use(page); },});my-test.jsconst base = require('@playwright/test');exports.test = base.test.extend({ // Define an option and provide a default value. // We can later override it in the config. person: ['John', { option: true }], // Override default "page" fixture. page: async ({ page, person }, use) => {   await page.goto('/chat');   // We use "person" parameter as a "name" for the chat room.   await page.getByLabel('User Name').fill(person);   await page.getByText('Enter chat room').click();   // Each test will get a "page" that already has the person name.   await use(page); },});noteParameterized projects behavior has changed in version 1.18. Learn more.Passing Environment Variables You can use environment variables to configure tests from the command line.For example, consider the following test file that needs a username and a password. It is usually a good idea not to store your secrets in the source code, so we'll need a way to pass secrets from outside.example.spec.tstest(`example test`, async ({ page }) => { // ... await page.getByLabel('User Name').fill(process.env.USERNAME); await page.getByLabel('Password').fill(process.env.PASSWORD);});You can run this test with your secret username and password set in the command line.BashPowerShellBatchUSERNAME=me PASSWORD=secret npx playwright test$env:USERNAME=me$env:PASSWORD=secretnpx playwright testset USERNAME=meset PASSWORD=secretnpx playwright testSimilarly, configuration file can also read environment variables passed through the command line.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: {   baseURL: process.env.STAGING === '1' ? 'http://staging.example.test/' : 'http://example.test/', }});Now, you can run tests against a staging or a production environment:BashPowerShellBatchSTAGING=1 npx playwright test$env:STAGING=1npx playwright testset STAGING=1npx playwright test.env files To make environment variables easier to manage, consider something like .env files. Here is an example that uses dotenv package to read environment variables directly in the configuration file.playwright.config.tsimport { defineConfig } from '@playwright/test';import dotenv from 'dotenv';import path from 'path';// Read from default ".env" file.dotenv.config();// Alternatively, read from "../my.env" file.dotenv.config({ path:

path.resolve(__dirname, '..', 'my.env') });export default defineConfig({ use: { baseURL: process.env.STAGING === '1' ? 'http://staging.example.test/' : 'http://example.test/', }});Now, you can just edit .env file to set any variables you'd like.# .env fileSTAGING=0USERNAME=mePASSWORD=secretRun tests as usual, your environment variables should be picked up.npx playwright testCreate tests via a CSV file The Playwright test-runner runs in Node.js, this means you can directly read files from the file system and parse them with your preferred CSV library.See for example this CSV file, in our example input.csv:"test_case","some_value","some_other_value""value 1","value 11","foobar1""value 2","value 22","foobar21""value 3","value 33","foobar321""value 4","value 44","foobar4321"Based on this we'll generate some tests by using the csv-parse library from NPM:test.spec.tsimport fs from 'fs';import path from 'path';import { test } from '@playwright/test';import { parse } from 'csv-parse/sync';const records = parse(fs.readFileSync(path.join(__dirname, 'input.csv')), { columns: true, skip_empty_lines: true});for (const record of records) { test(`foo: ${record.test_case}`, async ({ page }) => { console.log(record.test_case, record.some_value, record.some_other_value); });}
ProjectsA project is logical group of tests running with the same configuration. We use projects so we can run tests on different browsers and devices. Projects are configured in the playwright.config.ts file and once configured you can then run your tests on all projects or only on a specific project. You can also use projects to run the same tests in different configurations. For example, you can run the same tests in a logged-in and logged-out state.By setting up projects you can also run a group of tests with different timeouts or retries or a group of tests against different environments such as staging and production, splitting tests per package/functionality and more.Configure projects for multiple browsers By using projects you can run your tests in multiple browsers such as chromium, webkit and firefox as well as branded browsers such as Google Chrome and Microsoft Edge. Playwright can also run on emulated tablet and mobile devices. See the registry of device parameters for a complete list of selected desktop, tablet and mobile devices.import { defineConfig, devices } from '@playwright/test';export default defineConfig({ projects: [{ name: 'chromium', use: { ...devices['Desktop Chrome'] }, }, { name: 'firefox', use: { ...devices['Desktop Firefox'] }, }, { name: 'webkit', use: { ...devices['Desktop Safari'] }, }, /* Test against mobile viewports. */ { name: 'Mobile Chrome', use: { ...devices['Pixel 5'] }, }, { name: 'Mobile Safari', use: { ...devices['iPhone 12'] }, }, /* Test against branded browsers. */ { name: 'Microsoft Edge', use: { ...devices['Desktop Edge'], channel: 'msedge' }, }, { name: 'Google Chrome', use: { ...devices['Desktop Chrome'], channel: 'chrome' }, },],});Run projects Playwright will run all projects by default.npx playwright testRunning 7 tests using 5 workers ' [chromium] › example.spec.ts:3:1 › basic test (2s) ' [firefox] › example.spec.ts:3:1 › basic test (2s) ' [webkit] › example.spec.ts:3:1 › basic test (2s) ' [Mobile Chrome] › example.spec.ts:3:1 › basic test (2s) ' [Mobile Safari] › example.spec.ts:3:1 › basic test (2s) ' [Microsoft Edge] › example.spec.ts:3:1 › basic test (2s) ' [Google Chrome] › example.spec.ts:3:1 › basic test (2s)Use the --project command line option to run a single project.npx playwright test --project=firefoxRunning 1 test using 1 worker ' [firefox] › example.spec.ts:3:1 › basic test (2s)The VS Code test

runner runs your tests on the default browser of Chrome. To run on other/multiple browsers click the play button's dropdown from the testing sidebar and choose another profile or modify the default profile by clicking Select Default Profile and select the browsers you wish to run your tests on.Choose a specific profile, various profiles or all profiles to run tests on.Configure projects for multiple environments By setting up projects we can also run a group of tests with different timeouts or retries or run a group of tests against different environments. For example we can run our tests against a staging environment with 2 retries as well as against a production environment with 0 retries.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ timeout: 60000, // Timeout is shared between all tests. projects: [{ name: 'staging', use: { baseURL: 'staging.example.com', }, retries: 2, }, { name: 'production', use: { baseURL: 'production.example.com', }, retries: 0, },],});Splitting tests into projects We can split tests into projects and use filters to run a subset of tests. For example, we can create a project that runs tests using a filter matching all tests with a specific file name. We can then have another group of tests that ignore specific test files.Here is an example that defines a common timeout and two projects. The "Smoke" project runs a small subset of tests without retries, and "Default" project runs all other tests with retries.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ timeout: 60000, // Timeout is shared between all tests. projects: [{ name: 'Smoke', testMatch: /.*smoke.spec.ts/, retries: 0, }, { name: 'Default', testIgnore: /.*smoke.spec.ts/, retries: 2, },],});Dependencies Dependencies are a list of projects that need to run before the tests in another project run. They can be useful for configuring the global setup actions so that one project depends on this running first. When using project dependencies, test reporters will show the setup tests and the trace viewer will record traces of the setup. You can use the inspector to inspect the DOM snapshot of the trace of your setup tests and you can also use fixtures inside your setup.In this example the chromium, firefox and webkit projects depend on the setup project.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: '**/*.setup.ts', }, { name: 'chromium', use: { ...devices['Desktop Chrome'] }, dependencies: ['setup'], }, { name: 'firefox', use: { ...devices['Desktop Firefox'] }, dependencies: ['setup'], }, { name: 'webkit', use: { ...devices['Desktop Safari'] }, dependencies: ['setup'], },],});Running Sequence When working with tests that have a dependency, the dependency will always run first and once all tests from this project have passed, then the other projects will run in parallel.Running order:Tests in 'setup' project runTests in 'chromium', 'webkit' and 'firefox' projects run in parallelIf there are more than one dependency then these project dependencies will be run first and in parallel. If the tests from a dependency fails then the tests that rely on this project will not be run.Running order:Tests in 'Browser Login' and 'DataBase' projects run in parallel'Browser Login' passes'L 'DataBase' fails!“e2e tests” is not run!Teardown You can also testProject.teardown your setup by adding a teardown property to your setup project. This will run after all dependent projects have run. See the teardown guide for more information.Custom project parameters Projects can be also used to parametrize tests with your custom configuration - take a look at this separate guide.

ReportersPlaywright Test comes with a few built-in reporters for different needs and ability to provide custom reporters. The easiest way to try out built-in reporters is to pass --reporter command line option.npx playwright test --reporter=lineFor more control, you can specify reporters programmatically in the configuration file.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'line',});Multiple reporters You can use multiple reporters at the same time. For example you can use 'list' for nice terminal output and 'json' to get a comprehensive json file with the test results.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['list'],   ['json', { outputFile: 'test-results.json' }]],});Reporters on CI You can use different reporters locally and on CI. For example, using concise 'dot' reporter avoids too much output. This is the default on CI.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Concise 'dot' for CI, default 'list' when running locally reporter: process.env.CI ? 'dot' : 'list',});Built-in reporters All built-in reporters show detailed information about failures, and mostly differ in verbosity for successful runs.List reporter List reporter is default (except on CI where the dot reporter is default). It prints a line for each test being run.npx playwright test --reporter=listplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'list',});Here is an example output in the middle of a test run. Failures will be listed at the end.npx playwright test --reporter=listRunning 124 tests using 6 workers 1 ' should access error in env (438ms) 2 ' handle long test names (515ms) 3 x 1) render expected (691ms) 4 ' should timeout (932ms) 5   should repeat each: 6 ' should respect enclosing .gitignore (569ms) 7   should teardown env after timeout: 8   should respect excluded tests: 9 ' should handle env beforeEach error (638ms)10   should respect enclosing .gitignore:You can opt into the step rendering via passing the following config option:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['list', { printSteps: true }]],});Line reporter Line reporter is more concise than the list reporter. It uses a single line to report last finished test, and prints failures when they occur. Line reporter is useful for large test suites where it shows the progress but does not spam the output by listing all the tests.npx playwright test --reporter=lineplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'line',});Here is an example output in the middle of a test run. Failures are reported inline.npx playwright test --reporter=lineRunning 124 tests using 6 workers 1) dot-reporter.spec.ts:20:1 › render expected ==   Error: expect(received).toBe(expected) // Object.is equality   Expected: 1   Received: 0[23/124] gitignore.spec.ts - should respect nested .gitignoreDot reporter Dot reporter is very concise - it only produces a single character per successful test run. It is the default on CI and useful where you don't want a lot of output.npx playwright test --reporter=dotplaywright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: 'dot',});Here is an example output in the middle of a test run. Failures will be listed at the end.npx playwright test --reporter=dotRunning 124 tests using 6 workers·····F·····HTML reporter HTML reporter produces a self-contained folder that contains report for the test run that can be served as a web page.npx playwright test --reporter=htmlBy default, HTML report is opened

automatically if some of the tests failed. You can control this behavior via the open property in the Playwright config. The possible values for that property are always, never and on-failure (default).You can also configure host and port that are used to serve the HTML report.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['html', { open: 'never' }]],});By default, report is written into the playwright-report folder in the current working directory. One can override that location using the PLAYWRIGHT_HTML_REPORT environment variable or a reporter configuration.In configuration file, pass options directly:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['html', { outputFolder: 'my-report' }]],});If you are uploading attachments from data folder to other location, you can use attachmentsBaseURL option to let html report where to look for them.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['html', { attachmentsBaseURL: 'https://external-storage.com/' }]],});A quick way of opening the last test run report is:npx playwright show-reportOr if there is a custom folder name:npx playwright show-report my-reportThe html reporter currently does not support merging reports generated across multiple --shards into a single report. See this issue for available third party solutions.JSON reporter JSON reporter produces an object with all information about the test run.Most likely you want to write the JSON to a file. When running with --reporter=json, use PLAYWRIGHT_JSON_OUTPUT_NAME environment variable:BashPowerShellBatchPLAYWRIGHT_JSON_OUTPUT_NAME=results.json npx playwright test --reporter=json$env:PLAYWRIGHT_JSON_OUTPUT_NAME="results.json"npx playwright test --reporter=jsonset PLAYWRIGHT_JSON_OUTPUT_NAME=results.jsonnpx playwright test --reporter=jsonIn configuration file, pass options directly:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['json', { outputFile: 'results.json' }]],});JUnit reporter JUnit reporter produces a JUnit-style xml report.Most likely you want to write the report to an xml file. When running with --reporter=junit, use PLAYWRIGHT_JUNIT_OUTPUT_NAME environment variable:BashPowerShellBatchPLAYWRIGHT_JUNIT_OUTPUT_NAME=results.xml npx playwright test --reporter=junit$env:PLAYWRIGHT_JUNIT_OUTPUT_NAME="results.xml"npx playwright test --reporter=junitset PLAYWRIGHT_JUNIT_OUTPUT_NAME=results.xmlnpx playwright test --reporter=junitIn configuration file, pass options directly:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['junit', { outputFile: 'results.xml' }]],});GitHub Actions annotations You can use the built in github reporter to get automatic failure annotations when running in GitHub actions.Note that all other reporters work on GitHub Actions as well, but do not provide annotations.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // 'github' for GitHub Actions CI to generate annotations, plus a concise 'dot' // default 'list' when running locally reporter: process.env.CI ? 'github' : 'list',});Custom reporters You can create a custom reporter by implementing a class with some of the reporter methods. Learn more about the Reporter API.my-awesome-reporter.tsimport type { FullConfig, FullResult, Reporter, Suite, TestCase, TestResult }

from '@playwright/test/reporter';class MyReporter implements Reporter { onBegin(config: FullConfig, suite: Suite) { console.log(`Starting the run with ${suite.allTests().length} tests`); } onTestBegin(test: TestCase, result: TestResult) { console.log(`Starting test ${test.title}`); } onTestEnd(test: TestCase, result: TestResult) { console.log(`Finished test ${test.title}: ${result.status}`); } onEnd(result: FullResult) { console.log(`Finished the run: ${result.status}`); }}export default MyReporter;Now use this reporter with testConfig.reporter.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ reporter: './my-awesome-reporter.ts',});Third party reporter showcase AllureMonocartTesultsReportPortalCurrentsSerenity/JS RetriesTest retries are a way to automatically re-run a test when it fails. This is useful when a test is flaky and fails intermittently. Test retries are configured in the configuration file.Failures Playwright Test runs tests in worker processes. These processes are OS processes, running independently, orchestrated by the test runner. All workers have identical environments and each starts its own browser.Consider the following snippet:import { test } from '@playwright/test';test.describe('suite', () => { test.beforeAll(async () => { /* ... */ }); test('first good', async ({ page }) => { /* ... */ }); test('second flaky', async ({ page }) => { /* ... */ }); test('third good', async ({ page }) => { /* ... */ });});When all tests pass, they will run in order in the same worker process.Worker process startsbeforeAll hook runsfirst good passessecond flaky passesthird good passesShould any test fail, Playwright Test will discard the entire worker process along with the browser and will start a new one. Testing will continue in the new worker process starting with the next test.Worker process #1 startsbeforeAll hook runsfirst good passessecond flaky failsWorker process #2 startsbeforeAll hook runs againthird good passesIf you enable retries, second worker process will start by retrying the failed test and continue from there.Worker process #1 startsbeforeAll hook runsfirst good passessecond flaky failsWorker process #2 startsbeforeAll hook runs againsecond flaky is retried and passesthird good passesThis scheme works perfectly for independent tests and guarantees that failing tests can't affect healthy ones.Retries Playwright supports test retries. When enabled, failing tests will be retried multiple times until they pass, or until the maximum number of retries is reached. By default failing tests are not retried.# Give failing tests 3 retry attemptsnpx playwright test --retries=3You can configure retries in the configuration file:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Give failing tests 3 retry attempts retries: 3,});Playwright Test will categorize tests as follows:"passed" - tests that passed on the first run;"flaky" - tests that failed on the first run, but passed when retried;"failed" - tests that failed on the first run and failed all retries.Running 3 tests using 1 worker ' example.spec.ts:4:2 › first passes (438ms) x example.spec.ts:5:2 › second flaky (691ms) ' example.spec.ts:5:2 › second flaky (522ms) ' example.spec.ts:6:2 › third passes (932ms) 1 flaky example.spec.ts:5:2 › second flaky 2 passed (4s)You can detect retries at runtime with testInfo.retry, which is accessible to any test, hook or fixture. Here is an example that clears some server-side state before a retry.import { test, expect } from '@playwright/test';test('my test', async ({ page }, testInfo) => { if (testInfo.retry) await cleanSomeCachesOnTheServer(); // ...});You can specify retries for a specific group of tests or a single file with test.describe.configure().import { test, expect } from

'@playwright/test';test.describe(() => { // All tests in this describe group will get 2 retry attempts. test.describe.configure({ retries: 2 }); test('test 1', async ({ page }) => {   // ... }); test('test 2', async ({ page }) => {   // ... });});Serial mode Use test.describe.serial() to group dependent tests to ensure they will always run together and in order. If one of the tests fails, all subsequent tests are skipped. All tests in the group are retried together.Consider the following snippet that uses test.describe.serial:import { test } from '@playwright/test';test.describe.configure({ mode: 'serial' });test.beforeAll(async () => { /* ... */ });test('first good', async ({ page }) => { /* ... */ });test('second flaky', async ({ page }) => { /* ... */ });test('third good', async ({ page }) => { /* ... */ });When running without retries, all tests after the failure are skipped:Worker process #1:beforeAll hook runsfirst good passessecond flaky failsthird good is skipped entirelyWhen running with retries, all tests are retried together:Worker process #1:beforeAll hook runsfirst good passessecond flaky failsthird good is skippedWorker process #2:beforeAll hook runs againfirst good passes againsecond flaky passesthird good passesnoteIt is usually better to make your tests isolated, so they can be efficiently run and retried independently.Reuse single page between tests Playwright Test creates an isolated Page object for each test. However, if you'd like to reuse a single Page object between multiple tests, you can create your own in test.beforeAll() and close it in test.afterAll().TypeScriptJavaScriptexample.spec.tsimport { test, type Page } from '@playwright/test';test.describe.configure({ mode: 'serial' });let page: Page;test.beforeAll(async ({ browser }) => { page = await browser.newPage();});test.afterAll(async () => { await page.close();});test('runs first', async () => { await page.goto('https://playwright.dev/');});test('runs second', async () => { await page.getByText('Get Started').click();});example.spec.js// @ts-checkconst { test } = require('@playwright/test');test.describe.configure({ mode: 'serial' });/** @type {import('@playwright/test').Page} */let page;test.beforeAll(async ({ browser }) => { page = await browser.newPage();});test.afterAll(async () => { await page.close();});test('runs first', async () => { await page.goto('https://playwright.dev/');});test('runs second', async () => { await page.getByText('Get Started').click();});
TimeoutsPlaywright Test has multiple configurable timeouts for various tasks.TimeoutDefaultDescriptionTest timeout30000 msTimeout for each test, includes test, hooks and fixtures:Set defaultconfig = { timeout: 60000 }Overridetest.setTimeout(120000)Expect timeout5000 msTimeout for each assertion:Set defaultconfig = { expect: { timeout: 10000 } }Overrideexpect(locator).toBeVisible({ timeout: 10000 })Test timeout Playwright Test enforces a timeout for each test, 30 seconds by default. Time spent by the test function, fixtures, beforeEach and afterEach hooks is included in the test timeout.Timed out test produces the following error:example.spec.ts:3:1 › basic test ===========================Timeout of 30000ms exceeded.The same timeout value also applies to beforeAll and afterAll hooks, but they do not share time with any test.Set test timeout in the config playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ timeout: 5 * 60 * 1000,});API reference: testConfig.timeout.Set timeout for a single test import { test, expect } from '@playwright/test';test('slow test', async ({ page }) => { test.slow(); // Easy way to triple the default timeout // ...});test('very slow test', async ({ page }) =>

{ test.setTimeout(120000); // ...});API reference: test.setTimeout() and test.slow().Change timeout from a beforeEach hook import { test, expect } from '@playwright/test';test.beforeEach(async ({ page }, testInfo) => { // Extend timeout for all tests running this hook by 30 seconds. testInfo.setTimeout(testInfo.timeout + 30000);});API reference: testInfo.setTimeout().Change timeout for beforeAll/afterAll hook beforeAll and afterAll hooks have a separate timeout, by default equal to test timeout. You can change it separately for each hook by calling testInfo.setTimeout() inside the hook.import { test, expect } from '@playwright/test';test.beforeAll(async () => { // Set timeout for this hook. test.setTimeout(60000);});API reference: testInfo.setTimeout().Expect timeout Web-first assertions like expect(locator).toHaveText() have a separate timeout, 5 seconds by default. Assertion timeout is unrelated to the test timeout. It produces the following error:example.spec.ts:3:1 › basic test ==========================Error: expect(received).toHaveText(expected)Expected string: "my text"Received string: ""Call log: - expect.toHaveText with timeout 5000ms - waiting for "locator('button')"Set expect timeout in the config playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ expect: {   timeout: 10 * 1000, },});Global timeout Playwright Test supports a timeout for the whole test run. This prevents excess resource usage when everything went wrong. There is no default global timeout, but you can set a reasonable one in the config, for example one hour. Global timeout produces the following error:Running 1000 tests using 10 workers 514 skipped 486 passed Timed out waiting 3600s for the entire test runYou can set global timeout in the config.// playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ globalTimeout: 60 * 60 * 1000,});API reference: testConfig.globalTimeout.Advanced: low level timeouts These are the low-level timeouts that are pre-configured by the test runner, you should not need to change these. If you happen to be in this section because your test are flaky, it is very likely that you should be looking for the solution elsewhere.TimeoutDefaultDescriptionAction timeoutno timeoutTimeout for each action:Set defaultconfig = { use: { actionTimeout: 10000 } } Overridelocator.click({ timeout: 10000 })Navigation timeoutno timeoutTimeout for each navigation action:Set defaultconfig = { use: { navigationTimeout: 30000 } } Overridepage.goto('/', { timeout: 30000 })Global timeoutno timeoutGlobal timeout for the whole test run:Set in configconfig = { globalTimeout: 60*60*1000 }beforeAll/afterAll timeout30000 msTimeout for the hook:Set in hooktest.setTimeout(60000)Fixture timeoutno timeoutTimeout for an individual fixture:Set in fixture{ scope: 'test', timeout: 30000 }Set timeout for a single assertion import { test, expect } from '@playwright/test';test('basic test', async ({ page }) => { await expect(page.getByRole('button')).toHaveText('Sign in', { timeout: 10000 });});Set action and navigation timeouts in the config playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: {   actionTimeout: 10 * 1000, navigationTimeout: 30 * 1000, },});API reference: testOptions.actionTimeout and testOptions.navigationTimeout.Set timeout for a single action import { test, expect } from '@playwright/test';test('basic test', async ({ page }) => { await page.goto('https://playwright.dev', { timeout: 30000 }); await page.getByText('Get Started').click({ timeout: 10000 });});Fixture timeout By default, fixture shares timeout with the test. However, for slow fixtures, especially worker-scoped ones, it is convenient to have a separate

timeout. This way you can keep the overall test timeout small, and give the slow fixture more time.TypeScriptJavaScriptimport { test as base, expect } from '@playwright/test';const test = base.extend<{ slowFixture: string }>({ slowFixture: [async ({}, use) => {   // ... perform a slow operation ...   await use('hello'); }, { timeout: 60000 }]});test('example test', async ({ slowFixture }) => { // ...});const { test: base, expect } = require('@playwright/test');const test = base.extend({ slowFixture: [async ({}, use) => {   // ... perform a slow operation ...   await use('hello'); }, { timeout: 60000 }]});test('example test', async ({ slowFixture }) => { // ...});API reference: test.extend().
TypeScriptPlaywright supports TypeScript out of the box. You just write tests in TypeScript, and Playwright will read them, transform to JavaScript and run.tsconfig.json Playwright will pick up tsconfig.json for each source file it loads. Note that Playwright only supports the following tsconfig options: paths and baseUrl.We recommend setting up a separate tsconfig.json in the tests directory so that you can change some preferences specifically for the tests. Here is an example directory structure.src/ source.tstests/   tsconfig.json # test-specific tsconfig   example.spec.tstsconfig.json # generic tsconfig for all typescript sourcesplaywright.config.tstsconfig path mapping Playwright supports path mapping declared in the tsconfig.json. Make sure that baseUrl is also set.Here is an example tsconfig.json that works with Playwright Test:
{ "compilerOptions": {   "baseUrl": ".", // This must be specified if "paths" is.   "paths": {   "@myhelper/*": ["packages/myhelper/*"] // This mapping is relative to "baseUrl".   } }}You can now import using the mapped paths:example.spec.tsimport { test, expect } from '@playwright/test';import { username, password } from '@myhelper/credentials';test('example', async ({ page }) => { await page.getByLabel('User Name').fill(username); await page.getByLabel('Password').fill(password);});Manually compile tests with TypeScript Sometimes, Playwright Test will not be able to transform your TypeScript code correctly, for example when you are using experimental or very recent features of TypeScript, usually configured in tsconfig.json.In this case, you can perform your own TypeScript compilation before sending the tests to Playwright.First add a tsconfig.json file inside the tests directory:{   "compilerOptions": {   "target": "ESNext",   "module": "commonjs",   "moduleResolution": "Node",   "sourceMap": true,   "outDir": "../tests-out",   }}In package.json, add two scripts:
{ "scripts": {   "pretest": "tsc --incremental -p tests/tsconfig.json",   "test": "playwright test -c tests-out" }}The pretest script runs typescript on the tests. test will run the tests that have been generated to the tests-out directory. The -c argument configures the test runner to look for tests inside the tests-out directory.Then npm run test will build the tests and run them.Transpilation issues Using dynamic imports inside a function passed to various evaluate() methods is not supported. This is because Playwright uses Function.prototype.toString() to serialize functions, and transpiler will sometimes replace dynamic imports with require() calls, which are not valid inside the web page.To work around this issue, use a string template instead of a function:await page.evaluate(`async () => {  const { value } = await import('some-module');  console.log(value);}`);
UI ModeUI Mode let's you explore, run and debug tests with a time travel experience complete with watch mode. All test files are loaded into the testing sidebar where you can expand each file and describe block to individually run, view, watch and debug

each test. Filter tests by text or @tag or by passed, failed and skipped tests as well as by projects as set in your playwright.config file. See a full trace of your tests and hover back and forward over each action to see what was happening during each step and pop out the DOM snapshot to a separate window for a better debugging experience.Running tests in UI Mode To open UI mode, run the following command:npx playwright test --uiFiltering tests Filter tests by text or @tag or by passed, failed or skipped tests. You can also filter by projects as set in your playwright.config file. If you are using project dependencies make sure to run your setup tests first before running the tests that depend on them. The UI mode will not take into consideration the setup tests and therefore you will have to manually run them first.Running your tests Once you launch UI Mode you will see a list of all your test files. You can run all your tests by clicking the triangle icon in the sidebar. You can also run a single test file, a block of tests or a single test by hovering over the name and clicking on the triangle next to it. Viewing test traces Traces are shown for each test that has been run, so to see the trace, click on one of the test names. Note that you won't see any trace results if you click on the name of the test file or the name of a describe block.Actions and metadata In the Actions tab you can see what locator was used for every action and how long each one took to run. Hover over each action of your test and visually see the change in the DOM snapshot. Go back and forward in time and click an action to inspect and debug. Use the Before and After tabs to visually see what happened before and after the action. Next to the Actions tab you will find the Metadata tab which will show you more information on your test such as the Browser, viewport size, test duration and more.Source, console, log and network As you hover over each action of your test the source code for the test is highlighted below. Click on the source tab to see the source code for the entire test. Click on the console tab to see the console logs for each action. Click on the log tab to see the logs for each action. Click on the network tab to see the network logs for each action.Attachments The "Attachments" tab allows you to explore attachments. If you're doing visual regression testing, you'll be able to compare screenshots by examining the image diff, the actual image and the expected image. When you click on the expected image you can use the slider to slide one image over the other so you can easily see the differences in your screenshots.Pop out and inspect the DOM Pop out the DOM snapshot into it's own window for a better debugging experience by clicking on the pop out icon above the DOM snapshot. From there you can open the browser DevTools and inspect the HTML, CSS, Console etc. Go back to UI Mode and click on another action and pop that one out to easily compare the two side by side or debug each individually.Timeline view At the top of the trace you can see a timeline view of each action of your test. Hover back and forth to see an image snapshot for each action.Pick locator Click on the pick locator button and hover over the DOM snapshot to see the locator for each element highlighted as you hover. Click on an element to save the locator into the pick locator field. You can then copy the locator and paste it into your test.Watch mode Next to the name of each test in the sidebar you will find an eye icon. Clicking on the icon will activate watch mode which will re-run the test when you make changes to it. You can watch a number of tests at the same time be clicking the eye icon next to each one or all tests by clicking the eye icon at the top of the sidebar. If you are using VS Code then you can easily open your test by clicking on the file icon next to the eye icon. This will open your test in VS Code right at the line

of code that you clicked on.Docker & GitHub Codespaces For Docker and GitHub Codespaces environments, you can run UI mode in the browser. In order for an endpoint to be accessible outside of the container, it needs to be bound to the 0.0.0.0 interface:npx playwright test --ui-host=0.0.0.0In the case of GitHub Codespaces, the port gets forwarded automatically, so you can open UI mode in the browser by clicking on the link in the terminal.To have a static port, you can pass the --ui-port flag:npx playwright test --ui-port=8080 --ui-host=0.0.0.0noteBe aware that when specifying the --ui-host=0.0.0.0 flag, UI Mode with your traces, the passwords and secrets is accessible from other machines inside your network. In the case of GitHub Codespaces, the ports are only accessible from your account by default.
CI GitHub ActionsWhen installing Playwright you are given the option to add a GitHub Actions. This creates a playwright.yml file inside a .github/workflows folder containing everything you need so that your tests run on each push and pull request into the main/master branch.What you will learn:GitHub ActionsCreate a Repo and Push to GitHubOpening the WorkflowsViewing Test LogsHTML ReportDownloading the HTML ReportViewing the HTML ReportViewing the TraceWhat's NextGitHub Actions Tests will run on push or pull request on branches main/master. The workflow will install all dependencies, install Playwright and then run the tests. It will also create the HTML report.name: Playwright Testson: push: branches: [main, master] pull_request: branches: [main, master]jobs: test: timeout-minutes: 60 runs-on: ubuntu-latest steps: - uses: actions/checkout@v3 - uses: actions/setup-node@v3 with: node-version: 18 - name: Install dependencies run: npm ci - name: Install Playwright Browsers run: npx playwright install --with-deps - name: Run Playwright tests run: npx playwright test - uses: actions/upload-artifact@v3 if: always() with: name: playwright-report path: playwright-report/ retention-days: 30Create a Repo and Push to GitHub Create a repo on GitHub and create a new repository or push an existing repository. Follow the instructions on GitHub and don't forget to initialize a git repository using the git init command so you can add, commit and push your code.Opening the Workflows Click on the Actions tab to see the workflows. Here you will see if your tests have passed or failed.On Pull Requests you can also click on the Details link in the PR status check.Viewing Test Logs Clicking on the workflow run will show you the all the actions that GitHub performed and clicking on Run Playwright tests will show the error messages, what was expected and what was received as well as the call log.HTML Report The HTML Report shows you a full report of your tests. You can filter the report by browsers, passed tests, failed tests, skipped tests and flaky tests.Downloading the HTML Report In the Artifacts section click on the playwright-report to download your report in the format of a zip file.Viewing the HTML Report Locally opening the report will not work as expected as you need a web server in order for everything to work correctly. First, extract the zip, preferably in a folder that already has Playwright installed. Using the command line change into the directory where the report is and use npx playwright show-report followed by the name of the extracted folder. This will serve up the report and enable you to view it in your browser.npx playwright show-report name-of-my-extracted-playwright-reportTo learn more about reports check out our detailed guide on HTML ReporterViewing the Trace Once you have served the report using npx playwright show-report, click on the trace icon next to the test's file name as seen in the image above.

You can then view the trace of your tests and inspect each action to try to find out why the tests are failing.To learn more about traces check out our detailed guide on Trace Viewer.To learn more about running tests on CI check out our detailed guide on Continuous Integration.What's Next Learn how to use LocatorsLearn how to perform ActionsLearn how to write Assertions
Getting started - VS CodePlaywright Test was created specifically to accommodate the needs of end-to-end testing. Playwright supports all modern rendering engines including Chromium, WebKit, and Firefox. Test on Windows, Linux, and macOS, locally or on CI, headless or headed with native mobile emulation of Google Chrome for Android and Mobile Safari. Get started by installing Playwright and generating a test to see it in action. Alternatively you can also get started and run your tests using the CLI.Installation Install the VS Code extension from the marketplace or from the extensions tab in VS Code.Once installed, open the command panel and type:Install PlaywrightSelect Test: Install Playwright and Choose the browsers you would like to run your tests on. These can be later configured in the playwright.config file. You can also choose if you would like to have a GitHub Actions setup to run your tests on CI.Running Tests You can run a single test by clicking the green triangle next to your test block to run your test. Playwright will run through each line of the test and when it finishes you will see a green tick next to your test block as well as the time it took to run the test.Run Tests and Show Browsers You can also run your tests and show the browsers by selecting the option Show Browsers in the testing sidebar. Then when you click the green triangle to run your test the browser will open and you will visually see it run through your test. Leave this selected if you want browsers open for all your tests or uncheck it if you prefer your tests to run in headless mode with no browser open.Use the Close all browsers button to close all browsers.View and Run All Tests View all tests in the testing sidebar and extend the tests by clicking on each test. Tests that have not been run will not have the green check next to them. Run all tests by clicking on the white triangle as you hover over the tests in the testing sidebar.Run Tests on Specific Browsers The VS Code test runner runs your tests on the default browser of Chrome. To run on other/multiple browsers click the play button's dropdown and choose another profile or modify the default profile by clicking Select Default Profile and select the browsers you wish to run your tests on.Choose various or all profiles to run tests on multiple profiles. These profiles are read from the playwright.config file. To add more profiles such as a mobile profile, first add it to your config file and it will then be available here.Debugging Tests With the VS Code extension you can debug your tests right in VS Code see error messages, create breakpoints and live debug your tests.Error Messages If your test fails VS Code will show you error messages right in the editor showing what was expected, what was received as well as a complete call log.Live Debugging You can debug your test live in VS Code. After running a test with the Show Browser option checked, click on any of the locators in VS Code and it will be highlighted in the Browser window. Playwright will highlight it if it exists and show you if there is more than one resultYou can also edit the locators in VS Code and Playwright will show you the changes live in the browser window.Run in Debug Mode To set a breakpoint click next to the line number where you want the breakpoint to be until a red dot appears. Run the tests in debug mode by right clicking on the line next to the test you want to run. A browser window will open and the test will run and pause at where

the breakpoint is set. You can step through the tests, pause the test and rerun the tests from the menu in VS Code.Debug in different Browsers By default debugging is done using the Chromium profile. You can debug your tests on different browsers by right clicking on the debug icon in the testing sidebar and clicking on the 'Select Default Profile' option from the dropdown.Then choose the test profile you would like to use for debugging your tests. Each time you run your test in debug mode it will use the profile you selected. You can run tests in debug mode by right clicking the line number where your test is and selecting 'Debug Test' from the menu.To learn more about debugging, see Debugging in Visual Studio Code.Generating Tests CodeGen will auto generate your tests for you as you perform actions in the browser and is a great way to quickly get started. The viewport for the browser window is set to a specific width and height. See the configuration guide to change the viewport or emulate different environments.Record a New Test To record a test click on the Record new button from the Testing sidebar. This will create a test-1.spec.ts file as well as open up a browser window. In the browser go to the URL you wish to test and start clicking around. Playwright will record your actions and generate a test for you. Once you are done recording click the cancel button or close the browser window. You can then inspect your test-1.spec.ts file and see your generated test. Your browser does not support the video tag.Record at Cursor To record from a specific point in your test file click the Record at cursor button from the Testing sidebar. This generates actions into the existing test at the current cursor position. You can run the test, position the cursor at the end of the test and continue generating the test.Picking a Locator Pick a locator and copy it into your test file by clicking the Pick locator button form the testing sidebar. Then in the browser click the element you require and it will now show up in the Pick locator box in VS Code. Press 'enter' on your keyboard to copy the locator into the clipboard and then paste anywhere in your code. Or press 'escape' if you want to cancel.Playwright will look at your page and figure out the best locator, prioritizing role, text and test id locators. If the generator finds multiple elements matching the locator, it will improve the locator to make it resilient and uniquely identify the target element, so you don't have to worry about failing tests due to locators.What's next Write tests using web first assertions, page fixtures and locatorsSee test reportsSee a trace of your tests Release notesVersion 1.36 Ø<ßÝþ Summer maintenance release.Browser Versions Chromium 115.0.5790.75Mozilla Firefox 115.0WebKit 17.0This version was also tested against the following stable channels:Google Chrome 114Microsoft Edge 114Version 1.35 Highlights UI mode is now available in VSCode Playwright extension via a new "Show trace viewer" button:UI mode and trace viewer mark network requests handled with page.route() and browserContext.route() handlers, as well as those issued via the API testing:New option maskColor for methods page.screenshot(), locator.screenshot(), expect(page).toHaveScreenshot() and expect(locator).toHaveScreenshot() to change default masking color:await page.goto('https://playwright.dev');await expect(page).toHaveScreenshot({ mask: [page.locator('img')], maskColor: '#00FF00', // green});New uninstall CLI command to uninstall browser binaries:$ npx playwright uninstall # remove browsers installed by this installation$ npx playwright uninstall --all # remove all ever-install Playwright browsersBoth UI mode and trace viewer now could be opened in a browser tab:$ npx playwright test --ui-port 0 # open UI mode in a tab on a random port$ npx playwright show-trace --port 0 # open trace

viewer in tab on a random port& þ Breaking changes playwright-core binary got renamed from playwright to playwright-core. So if you use playwright-core CLI, make sure to update the name:$ npx playwright-core install # the new way to install browsers when using playwright-coreThis change does not affect @playwright/test and playwright package users.Browser Versions Chromium 115.0.5790.13Mozilla Firefox 113.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 114Microsoft Edge 114Version 1.34 Highlights UI Mode now shows steps, fixtures and attachments: New property testProject.teardown to specify a project that needs to run after this and all dependent projects have finished. Teardown is useful to cleanup any resources acquired by this project.A common pattern would be a setup dependency with a corresponding teardown:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{   name: 'setup',   testMatch: /global.setup\.ts/,   teardown: 'teardown',   },   {   name: 'teardown',   testMatch: /global.teardown\.ts/,   },   {   name: 'chromium',   use: devices['Desktop Chrome'],   dependencies: ['setup'],   },   {   name: 'firefox',   use: devices['Desktop Firefox'],   dependencies: ['setup'],   },   {   name: 'webkit',   use: devices['Desktop Safari'],   dependencies: ['setup'],   },],});New method expect.configure to create pre-configured expect instance with its own defaults such as timeout and soft.const slowExpect = expect.configure({ timeout: 10000 });await slowExpect(locator).toHaveText('Submit');// Always do soft assertions.const softExpect = expect.configure({ soft: true });New options stderr and stdout in testConfig.webServer to configure output handling:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Run your local dev server before starting the tests webServer: {   command: 'npm run start',   url: 'http://127.0.0.1:3000',   reuseExistingServer: !process.env.CI,   stdout: 'pipe',   stderr: 'pipe', },});New locator.and() to create a locator that matches both locators.const button = page.getByRole('button').and(page.getByTitle('Subscribe'));New events browserContext.on('console') and browserContext.on('dialog') to subscribe to any dialogs and console messages from any page from the given browser context. Use the new methods consoleMessage.page() and dialog.page() to pin-point event source.& þ Breaking changes npx playwright test no longer works if you install both playwright and @playwright/test. There's no need to install both, since you can always import browser automation APIs from @playwright/test directly:automation.tsimport { chromium, firefox, webkit } from '@playwright/test';/* ... */Node.js 14 is no longer supported since it reached its end-of-life on April 30, 2023.Browser Versions Chromium 114.0.5735.26Mozilla Firefox 113.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 113Microsoft Edge 113Version 1.33 Locators Update Use locator.or() to create a locator that matches either of the two locators. Consider a scenario where you'd like to click on a "New email" button, but sometimes a security settings dialog shows up instead. In this case, you can wait for either a "New email" button, or a dialog and act accordingly:const newEmail = page.getByRole('button', { name: 'New email' });const dialog = page.getByText('Confirm security settings');await expect(newEmail.or(dialog)).toBeVisible();if (await dialog.isVisible()) await page.getByRole('button', { name: 'Dismiss' }).click();await newEmail.click();Use new options hasNot and hasNotText in locator.filter() to find elements that do not match certain conditions.const rowLocator =

page.locator('tr');await rowLocator .filter({ hasNotText: 'text in column 1' }) .filter({ hasNot: page.getByRole('button', { name: 'column 2 button' }) }) .screenshot();Use new web-first assertion expect(locator).toBeAttached() to ensure that the element is present in the page's DOM. Do not confuse with the expect(locator).toBeVisible() that ensures that element is both attached & visible.New APIs locator.or()New option hasNot in locator.filter()New option hasNotText in locator.filter()expect(locator).toBeAttached()New option timeout in route.fetch()reporter.onExit()& þ Breaking change The mcr.microsoft.com/playwright:v1.33.0 now serves a Playwright image based on Ubuntu Jammy. To use the focal-based image, please use mcr.microsoft.com/playwright:v1.33.0-focal instead.Browser Versions Chromium 113.0.5672.53Mozilla Firefox 112.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 112Microsoft Edge 112Version 1.32 Introducing UI Mode (preview) New UI Mode lets you explore, run and debug tests. Comes with a built-in watch mode.Engage with a new flag --ui:npx playwright test --uiNew APIs New options updateMode and updateContent in page.routeFromHAR() and browserContext.routeFromHAR().Chaining existing locator objects, see locator docs for details.New property testInfo.testId.New option name in method tracing.startChunk().& þ Breaking change in component tests Note: component tests only, does not affect end-to-end tests.@playwright/experimental-ct-react now supports React 18 only.If you're running component tests with React 16 or 17, please replace @playwright/experimental-ct-react with @playwright/experimental-ct-react17.Browser Versions Chromium 112.0.5615.29Mozilla Firefox 111.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 111Microsoft Edge 111Version 1.31 New APIs New property testProject.dependencies to configure dependencies between projects.Using dependencies allows global setup to produce traces and other artifacts, see the setup steps in the test report and more.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ projects: [{ name: 'setup', testMatch: /global.setup\.ts/, }, { name: 'chromium', use: devices['Desktop Chrome'], dependencies: ['setup'], }, { name: 'firefox', use: devices['Desktop Firefox'], dependencies: ['setup'], }, { name: 'webkit', use: devices['Desktop Safari'], dependencies: ['setup'], },],});New assertion expect(locator).toBeInViewport() ensures that locator points to an element that intersects viewport, according to the intersection observer API.const button = page.getByRole('button');// Make sure at least some part of element intersects viewport.await expect(button).toBeInViewport();// Make sure element is fully outside of viewport.await expect(button).not.toBeInViewport();// Make sure that at least half of the element intersects viewport.await expect(button).toBeInViewport({ ratio: 0.5 });Miscellaneous DOM snapshots in trace viewer can be now opened in a separate window.New method defineConfig to be used in playwright.config.New option Route.fetch.maxRedirects for method route.fetch().Playwright now supports Debian 11 arm64.Official docker images now include Node 18 instead of Node 16.& þ Breaking change in component tests Note: component tests only, does not affect end-to-end tests.playwright-ct.config configuration file for component testing now requires calling defineConfig.// Beforeimport { type PlaywrightTestConfig, devices } from '@playwright/experimental-ct-react';const config: PlaywrightTestConfig = { // ... config goes here ...};export default config;Replace config variable definition with defineConfig call://

Afterimport { defineConfig, devices } from '@playwright/experimental-ct-react';export default defineConfig({ // ... config goes here ...});Browser Versions Chromium 111.0.5563.19Mozilla Firefox 109.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 110Microsoft Edge 110Version 1.30 Browser Versions Chromium 110.0.5481.38Mozilla Firefox 108.0.2WebKit 16.4This version was also tested against the following stable channels:Google Chrome 109Microsoft Edge 109Version 1.29 New APIs New method route.fetch() and new option json for route.fulfill():await page.route('**/api/settings', async route => { // Fetch original settings. const response = await route.fetch(); // Force settings theme to a predefined value. const json = await response.json(); json.theme = 'Solorized'; // Fulfill with modified data. await route.fulfill({ json });});New method locator.all() to iterate over all matching elements:// Check all checkboxes!const checkboxes = page.getByRole('checkbox');for (const checkbox of await checkboxes.all()) await checkbox.check();locator.selectOption() matches now by value or label:<select multiple> <option value="red">Red <option value="green">Green <option value="blue">Blue </select>await element.selectOption('Red');Retry blocks of code until all assertions pass:await expect(async () => { const response = await page.request.get('https://api.example.com'); await expect(response).toBeOK();}).toPass();Read more in our documentation.Automatically capture full page screenshot on test failure:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { screenshot: { mode: 'only-on-failure', fullPage: true, } }});Miscellaneous Playwright Test now respects jsconfig.json.New options args and proxy for androidDevice.launchBrowser().Option postData in method route.continue() now supports Serializable values.Browser Versions Chromium 109.0.5414.46Mozilla Firefox 107.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 108Microsoft Edge 108Version 1.28 Playwright Tools Record at Cursor in VSCode. You can run the test, position the cursor at the end of the test and continue generating the test.Live Locators in VSCode. You can hover and edit locators in VSCode to get them highlighted in the opened browser.Live Locators in CodeGen. Generate a locator for any element on the page using "Explore" tool.Codegen and Trace Viewer Dark Theme. Automatically picked up from operating system settings.Test Runner Configure retries and test timeout for a file or a test with test.describe.configure().// Each test in the file will be retried twice and have a timeout of 20 seconds.test.describe.configure({ retries: 2, timeout: 20_000 });test('runs first', async ({ page }) => {});test('runs second', async ({ page }) => {});Use testProject.snapshotPathTemplate and testConfig.snapshotPathTemplate to configure a template controlling location of snapshots generated by expect(page).toHaveScreenshot() and expect(snapshot).toMatchSnapshot().playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ testDir: './tests', snapshotPathTemplate: '{testDir}/__screenshots__/{testFilePath}/{arg}{ext}',});New APIs locator.blur()locator.clear()android.launchServer() and android.connect()androidDevice.on('close')Browser Versions Chromium 108.0.5359.29Mozilla Firefox 106.0WebKit 16.4This version was also tested against the following stable channels:Google Chrome 107Microsoft Edge 107Version 1.27 Locators With these new APIs writing locators is a joy:page.getByText() to locate by text

content.page.getByRole() to locate by ARIA role, ARIA attributes and accessible name.page.getByLabel() to locate a form control by associated label's text.page.getByTestId() to locate an element based on its data-testid attribute (other attribute can be configured).page.getByPlaceholder() to locate an input by placeholder.page.getByAltText() to locate an element, usually image, by its text alternative.page.getByTitle() to locate an element by its title.await page.getByLabel('User Name').fill('John');await page.getByLabel('Password').fill('secret-password');await page.getByRole('button', { name: 'Sign in' }).click();await expect(page.getByText('Welcome, John!')).toBeVisible();All the same methods are also available on Locator, FrameLocator and Frame classes.Other highlights workers option in the playwright.config.ts now accepts a percentage string to use some of the available CPUs. You can also pass it in the command line:npx playwright test --workers=20%New options host and port for the html reporter.import { defineConfig } from '@playwright/test';export default defineConfig({ reporter: [['html', { host: 'localhost', port: '9223' }]],});New field FullConfig.configFile is available to test reporters, specifying the path to the config file if any.As announced in v1.25, Ubuntu 18 will not be supported as of Dec 2022. In addition to that, there will be no WebKit updates on Ubuntu 18 starting from the next Playwright release.Behavior Changes expect(locator).toHaveAttribute() with an empty value does not match missing attribute anymore. For example, the following snippet will succeed when button does not have a disabled attribute.await expect(page.getByRole('button')).toHaveAttribute('disabled', '');Command line options --grep and --grep-invert previously incorrectly ignored grep and grepInvert options specified in the config. Now all of them are applied together.Browser Versions Chromium 107.0.5304.18Mozilla Firefox 105.0.1WebKit 16.0This version was also tested against the following stable channels:Google Chrome 106Microsoft Edge 106Version 1.26 Assertions New option enabled for expect(locator).toBeEnabled().expect(locator).toHaveText() now pierces open shadow roots.New option editable for expect(locator).toBeEditable().New option visible for expect(locator).toBeVisible().Other highlights New option maxRedirects for apiRequestContext.get() and others to limit redirect count.New command-line flag --pass-with-no-tests that allows the test suite to pass when no files are found.New command-line flag --ignore-snapshots to skip snapshot expectations, such as expect(value).toMatchSnapshot() and expect(page).toHaveScreenshot().Behavior Change A bunch of Playwright APIs already support the waitUntil: 'domcontentloaded' option. For example:await page.goto('https://playwright.dev', { waitUntil: 'domcontentloaded',});Prior to 1.26, this would wait for all iframes to fire the DOMContentLoaded event.To align with web specification, the 'domcontentloaded' value only waits for the target frame to fire the 'DOMContentLoaded' event. Use waitUntil: 'load' to wait for all iframes.Browser Versions Chromium 106.0.5249.30Mozilla Firefox 104.0WebKit 16.0This version was also tested against the following stable channels:Google Chrome 105Microsoft Edge 105Version 1.25 VSCode Extension Watch your tests running live & keep devtools open.Pick selector.Record new test from current page state.Test Runner test.step() now returns the value of the step function:test('should work', async ({ page }) => { const pageTitle = await test.step('get title', async () => {   await page.goto('https://playwright.dev');   return await page.title(); }); console.log(pageTitle);});Added test.describe.fixme().New 'interrupted' test

status.Enable tracing via CLI flag: npx playwright test --trace=on.Announcements Ø<ß• We now ship Ubuntu 22.04 Jammy Jellyfish docker image: mcr.microsoft.com/playwright:v1.34.0-jammy.Ø>Þ¦ This is the last release with macOS 10.15 support (deprecated as of 1.21).Ø>Þ¦ This is the last release with Node.js 12 support, we recommend upgrading to Node.js LTS (16).& þ Ubuntu 18 is now deprecated and will not be supported as of Dec 2022.Browser Versions Chromium 105.0.5195.19Mozilla Firefox 103.0WebKit 16.0This version was also tested against the following stable channels:Google Chrome 104Microsoft Edge 104Version 1.24 Ø<ß Multiple Web Servers in playwright.config.ts Launch multiple web servers, databases, or other processes by passing an array of configurations:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ webServer: [{ command: 'npm run start', url: 'http://127.0.0.1:3000', timeout: 120 * 1000, reuseExistingServer: ! process.env.CI, }, { command: 'npm run backend', url: 'http://127.0.0.1:3333', timeout: 120 * 1000, reuseExistingServer: ! process.env.CI, }], use: { baseURL: 'http://localhost:3000/', },});Ø=Ü Debian 11 Bullseye Support Playwright now supports Debian 11 Bullseye on x86_64 for Chromium, Firefox and WebKit. Let us know if you encounter any issues!Linux support looks like this:Ubuntu 20.04Ubuntu 22.04Debian 11Chromium' ' WebKit' ' Firefox' ' Ø=Ýuþ Anonymous Describe It is now possible to call test.describe() to create suites without a title. This is useful for giving a group of tests a common option with test.use().test.describe(() => { test.use({ colorScheme: 'dark' }); test('one', async ({ page }) => { // ... }); test('two', async ({ page }) => { // ... });});Ø>Ýé Component Tests Update Playwright 1.24 Component Tests introduce beforeMount and afterMount hooks. Use these to configure your app for tests.For example, this could be used to setup App router in Vue.js:src/component.spec.tsimport { test } from '@playwright/experimental-ct-vue';import { Component } from './mycomponent';test('should work', async ({ mount }) => { const component = await mount(Component, { hooksConfig: { /* anything to configure your app */ } });});playwright/index.tsimport { router } from '../router';import { beforeMount } from '@playwright/experimental-ct-vue/hooks';beforeMount(async ({ app, hooksConfig }) => { app.use(router);});A similar configuration in Next.js would look like this:src/component.spec.jsximport { test } from '@playwright/experimental-ct-react';import { Component } from './mycomponent';test('should work', async ({ mount }) => { const component = await mount(<Component></Component>, { // Pass mock value from test into `beforeMount`. hooksConfig: { router: { query: { page: 1, per_page: 10 }, asPath: '/posts' } } });});playwright/index.jsimport router from 'next/router';import { beforeMount } from '@playwright/experimental-ct-react/hooks';beforeMount(async ({ hooksConfig }) => { // Before mount, redefine useRouter to return mock value from test. router.useRouter = () => hooksConfig.router;});Version 1.23 Network Replay Now you can record network traffic into a HAR file and re-use this traffic in your tests.To record network into HAR file:npx playwright open --save-har=github.har.zip https://github.com/microsoftAlternatively, you can record HAR programmatically:const context = await browser.newContext({ recordHar: { path: 'github.har.zip' }});// ... do stuff ...await context.close();Use the new methods page.routeFromHAR() or browserContext.routeFromHAR() to serve matching responses from the HAR file:await context.routeFromHAR('github.har.zip');Read more in our documentation.Advanced Routing You can now use route.fallback() to defer

routing to other handlers.Consider the following example:// Remove a header from all requests.test.beforeEach(async ({ page }) => { await page.route('**/*', async route => { const headers = await route.request().allHeaders(); delete headers['if-none-match']; route.fallback({ headers }); });});test('should work', async ({ page }) => { await page.route('**/*', route => { if (route.request().resourceType() === 'image') route.abort(); else route.fallback(); });});Note that the new methods page.routeFromHAR() and browserContext.routeFromHAR() also participate in routing and could be deferred to.Web-First Assertions Update New method expect(locator).toHaveValues() that asserts all selected values of <select multiple> element.Methods expect(locator).toContainText() and expect(locator).toHaveText() now accept ignoreCase option.Component Tests Update Support for Vue2 via the @playwright/experimental-ct-vue2 package.Support for component tests for create-react-app with components in .js files.Read more about component testing with Playwright.Miscellaneous If there's a service worker that's in your way, you can now easily disable it with a new context option serviceWorkers:playwright.config.tsexport default { use: { serviceWorkers: 'block', }};Using .zip path for recordHar context option automatically zips the resulting HAR:const context = await browser.newContext({ recordHar: { path: 'github.har.zip', }});If you intend to edit HAR by hand, consider using the "minimal" HAR recording mode that only records information that is essential for replaying:const context = await browser.newContext({ recordHar: { path: 'github.har', mode: 'minimal', }});Playwright now runs on Ubuntu 22 amd64 and Ubuntu 22 arm64. We also publish new docker image mcr.microsoft.com/playwright:v1.34.0-jammy.& þ Breaking Changes & þ WebServer is now considered "ready" if request to the specified url has any of the following HTTP status codes:200-299300-399 (new)400, 401, 402, 403 (new)Version 1.22 Highlights Components Testing (preview)Playwright Test can now test your React, Vue.js or Svelte components. You can use all the features of Playwright Test (such as parallelization, emulation & debugging) while running components in real browsers.Here is what a typical component test looks like:App.spec.tsximport { test, expect } from '@playwright/experimental-ct-react';import App from './App';// Let's test component in a dark scheme!test.use({ colorScheme: 'dark' });test('should render', async ({ mount }) => { const component = await mount(<App></App>); // As with any Playwright test, assert locator text. await expect(component).toContainText('React'); // Or do a screenshot Ø=Þ€ await expect(component).toHaveScreenshot(); // Or use any Playwright method await component.click();});Read more in our documentation.Role selectors that allow selecting elements by their ARIA role, ARIA attributes and accessible name.// Click a button with accessible name "log in"await page.locator('role=button[name="log in"]').click();Read more in our documentation.New locator.filter() API to filter an existing locatorconst buttons = page.locator('role=button');// ...const submitButton = buttons.filter({ hasText: 'Submit' });await submitButton.click();New web-first assertions expect(page).toHaveScreenshot() and expect(locator).toHaveScreenshot() that wait for screenshot stabilization and enhances test reliability.The new assertions has screenshot-specific defaults, such as:disables animationsuses CSS scale optionawait page.goto('https://playwright.dev');await expect(page).toHaveScreenshot();The new expect(page).toHaveScreenshot() saves screenshots at the same location as

expect(snapshot).toMatchSnapshot().Version 1.21 Highlights New role selectors that allow selecting elements by their ARIA role, ARIA attributes and accessible name.// Click a button with accessible name "log in"await page.locator('role=button[name="log in"]').click();Read more in our documentation.New scale option in page.screenshot() for smaller sized screenshots.New caret option in page.screenshot() to control text caret. Defaults to "hide".New method expect.poll to wait for an arbitrary condition:// Poll the method until it returns an expected result.await expect.poll(async () => { const response = await page.request.get('https://api.example.com'); return response.status();}).toBe(200);expect.poll supports most synchronous matchers, like .toBe(), .toContain(), etc. Read more in our documentation.Behavior Changes ESM support when running TypeScript tests is now enabled by default. The PLAYWRIGHT_EXPERIMENTAL_TS_ESM env variable is no longer required.The mcr.microsoft.com/playwright docker image no longer contains Python. Please use mcr.microsoft.com/playwright/python as a Playwright-ready docker image with pre-installed Python.Playwright now supports large file uploads (100s of MBs) via locator.setInputFiles() API.Browser Versions Chromium 101.0.4951.26Mozilla Firefox 98.0.2WebKit 15.4This version was also tested against the following stable channels:Google Chrome 100Microsoft Edge 100Version 1.20 Highlights New options for methods page.screenshot(), locator.screenshot() and elementHandle.screenshot():Option animations: "disabled" rewinds all CSS animations and transitions to a consistent stateOption mask: Locator[] masks given elements, overlaying them with pink #FF00FF boxes.expect().toMatchSnapshot() now supports anonymous snapshots: when snapshot name is missing, Playwright Test will generate one automatically:expect('Web is Awesome <3').toMatchSnapshot();New maxDiffPixels and maxDiffPixelRatio options for fine-grained screenshot comparison using expect().toMatchSnapshot():expect(await page.screenshot()).toMatchSnapshot({ maxDiffPixels: 27, // allow no more than 27 different pixels.});It is most convenient to specify maxDiffPixels or maxDiffPixelRatio once in testConfig.expect.Playwright Test now adds testConfig.fullyParallel mode. By default, Playwright Test parallelizes between files. In fully parallel mode, tests inside a single file are also run in parallel. You can also use --fully-parallel command line flag.playwright.config.tsexport default { fullyParallel: true,};testProject.grep and testProject.grepInvert are now configurable per project. For example, you can now configure smoke tests project using grep:playwright.config.tsexport default { projects: [{ name: 'smoke tests', grep: /@smoke/, },],};Trace Viewer now shows API testing requests.locator.highlight() visually reveals element(s) for easier debugging.Announcements We now ship a designated Python docker image mcr.microsoft.com/playwright/python. Please switch over to it if you use Python. This is the last release that includes Python inside our javascript mcr.microsoft.com/playwright docker image.v1.20 is the last release to receive WebKit update for macOS 10.15 Catalina. Please update MacOS to keep using latest & greatest WebKit!Browser Versions Chromium 101.0.4921.0Mozilla Firefox 97.0.1WebKit 15.4This version was also tested against the following stable channels:Google Chrome 99Microsoft Edge 99Version 1.19 Playwright Test Update Playwright Test v1.19 now supports soft assertions. Failed soft assertionsdo not terminate test execution, but mark the test as failed.// Make a few checks that will not stop the test when failed...await

expect.soft(page.locator('#status')).toHaveText('Success');await expect.soft(page.locator('#eta')).toHaveText('1 day');// ... and continue the test to check more things.await page.locator('#next-page').click();await expect.soft(page.locator('#title')).toHaveText('Make another order');Read more in our documentationYou can now specify a custom error message as a second argument to the expect and expect.soft functions, for example:await expect(page.locator('text=Name'), 'should be logged in').toBeVisible();The error would look like this: Error: should be logged in Call log: - expect.toBeVisible with timeout 5000ms - waiting for "getByText('Name')" 2 | 3 | test('example test', async({ page }) => { > 4 | await expect(page.locator('text=Name'), 'should be logged in').toBeVisible(); | ^ 5 | }); 6 |Read more in our documentationBy default, tests in a single file are run in order. If you have many independent tests in a single file, you can now run them in parallel with test.describe.configure().Other Updates Locator now supports a has option that makes sure it contains another locator inside:await page.locator('article', { has: page.locator('.highlight'),}).click();Read more in locator documentationNew locator.page()page.screenshot() and locator.screenshot() now automatically hide blinking caretPlaywright Codegen now generates locators and frame locatorsNew option url in testConfig.webServer to ensure your web server is ready before running the testsNew testInfo.errors and testResult.errors that contain all failed assertions and soft assertions.Potentially breaking change in Playwright Test Global Setup It is unlikely that this change will affect you, no action is required if your tests keep running as they did.We've noticed that in rare cases, the set of tests to be executed was configured in the global setup by means of the environment variables. We also noticed some applications that were post processing the reporters' output in the global teardown. If you are doing one of the two, learn moreBrowser Versions Chromium 100.0.4863.0Mozilla Firefox 96.0.1WebKit 15.4This version was also tested against the following stable channels:Google Chrome 98Microsoft Edge 98Version 1.18 Locator Improvements locator.dragTo()expect(locator).toBeChecked({ checked })Each locator can now be optionally filtered by the text it contains:await page.locator('li', { hasText: 'my item' }).locator('button').click();Read more in locator documentationTesting API improvements expect(response).toBeOK()testInfo.attach()test.info()Improved TypeScript Support Playwright Test now respects tsconfig.json's baseUrl and paths, so you can use aliasesThere is a new environment variable PW_EXPERIMENTAL_TS_ESM that allows importing ESM modules in your TS code, without the need for the compile step. Don't forget the .js suffix when you are importing your esm modules. Run your tests as follows:npm i --save-dev @playwright/test@1.18.0-rc1PW_EXPERIMENTAL_TS_ESM=1 npx playwright testCreate Playwright The npm init playwright command is now generally available for your use:# Run from your project's root directorynpm init playwright@latest# Or create a new projectnpm init playwright@latest new-projectThis will create a Playwright Test configuration file, optionally add examples, a GitHub Action workflow and a first test example.spec.ts.New APIs & changes new testCase.repeatEachIndex APIacceptDownloads option now defaults to trueBreaking change: custom config options Custom config options are a convenient way to parametrize projects with different values. Learn more in this guide.Previously, any fixture introduced through test.extend() could be overridden in the

testProject.use config section. For example,// WRONG: THIS SNIPPET DOES NOT WORK SINCE v1.18.// fixtures.jsconst test = base.extend({ myParameter: 'default',});// playwright.config.jsmodule.exports = { use: {   myParameter: 'value', },};The proper way to make a fixture parametrized in the config file is to specify option: true when defining the fixture. For example,// CORRECT: THIS SNIPPET WORKS SINCE v1.18.// fixtures.jsconst test = base.extend({ // Fixtures marked as "option: true" will get a value specified in the config, // or fallback to the default value. myParameter: ['default', { option: true }],});// playwright.config.jsmodule.exports = { use: {   myParameter: 'value', },};Browser Versions Chromium 99.0.4812.0Mozilla Firefox 95.0WebKit 15.4This version was also tested against the following stable channels:Google Chrome 97Microsoft Edge 97Version 1.17 Frame Locators Playwright 1.17 introduces frame locators - a locator to the iframe on the page. Frame locators capture the logic sufficient to retrieve the iframe and then locate elements in that iframe. Frame locators are strict by default, will wait for iframe to appear and can be used in Web-First assertions.Frame locators can be created with either page.frameLocator() or locator.frameLocator() method.const locator = page.frameLocator('#my-iframe').locator('text=Submit');await locator.click();Read more at our documentation.Trace Viewer Update Playwright Trace Viewer is now available online at https://trace.playwright.dev! Just drag-and-drop your trace.zip file to inspect its contents.NOTE: trace files are not uploaded anywhere; trace.playwright.dev is a progressive web application that processes traces locally.Playwright Test traces now include sources by default (these could be turned off with tracing option)Trace Viewer now shows test nameNew trace metadata tab with browser detailsSnapshots now have URL barHTML Report Update HTML report now supports dynamic filteringReport is now a single static HTML file that could be sent by e-mail or as a slack attachment.Ubuntu ARM64 support + more Playwright now supports Ubuntu 20.04 ARM64. You can now run Playwright tests inside Docker on Apple M1 and on Raspberry Pi.You can now use Playwright to install stable version of Edge on Linux:npx playwright install msedgeNew APIs Tracing now supports a 'title' optionPage navigations support a new 'commit' waiting optionHTML reporter got new configuration optionstestConfig.snapshotDir optiontestInfo.parallelIndextestInfo.titlePathtestOptions.trace has new optionsexpect.toMatchSnapshot supports subdirectoriesreporter.printsToStdio()Version 1.16 Ø<ß- Playwright Test API Testing Playwright 1.16 introduces new API Testing that lets you send requests to the server directly from Node.js! Now you can:test your server APIprepare server side state before visiting the web application in a testvalidate server side post-conditions after running some actions in the browserTo do a request on behalf of Playwright's Page, use new page.request API:import { test, expect } from '@playwright/test';test('context fetch', async ({ page }) => { // Do a GET request on behalf of page const response = await page.request.get('http://example.com/foo.json'); // ...});To do a stand-alone request from node.js to an API endpoint, use new request fixture:import { test, expect } from '@playwright/test';test('context fetch', async ({ request }) => { // Do a GET request on behalf of page const response = await request.get('http://example.com/foo.json'); // ...});Read more about it in our API testing guide.Response Interception It is now possible to do response interception by combining API Testing with request interception.For example, we can blur all the images on the page:import { test, expect } from '@playwright/test';import jimp from

'jimp'; // image processing librarytest('response interception', async ({ page }) =>
{ await page.route('**/*.jpeg', async route => {   const response = await
page._request.fetch(route.request());   const image = await jimp.read(await
response.body());   await image.blur(5);   route.fulfill({   response,   body: await
image.getBufferAsync('image/jpeg'),   }); }); const response = await page.goto('https://
playwright.dev'); expect(response.status()).toBe(200);});Read more about response
interception.New HTML reporter Try it out new HTML reporter with either --
reporter=html or a reporter entry in playwright.config.ts file:$ npx playwright test --
reporter=htmlThe HTML reporter has all the information about tests and their failures,
including surfacing trace and image artifacts.Read more about our reporters.Ø<ß- Playwright
Library locator.waitFor Wait for a locator to resolve to a single element with a given state.
Defaults to the state: 'visible'.Comes especially handy when working with lists:import
{ test, expect } from '@playwright/test';test('context fetch', async ({ page }) => { const
completeness = page.locator('text=Success'); await completeness.waitFor();
expect(await page.screenshot()).toMatchSnapshot('screen.png');});Read more about
locator.waitFor().Docker support for Arm64 Playwright Docker image is now published
for Arm64 so it can be used on Apple Silicon.Read more about Docker integration.Ø<ß-
Playwright Trace Viewer web-first assertions inside trace viewerrun trace viewer with
npx playwright show-trace and drop trace files to the trace viewer PWAAPI testing is
integrated with trace viewerbetter visual attribution of action targetsRead more about
Trace Viewer.Browser Versions Chromium 97.0.4666.0Mozilla Firefox 93.0WebKit
15.4This version of Playwright was also tested against the following stable
channels:Google Chrome 94Microsoft Edge 94Version 1.15 Ø<ß- Playwright Library Ø=Ý±þ Mouse
Wheel By using Page.mouse.wheel you are now able to scroll vertically or horizontally.Ø=ÜÜ
New Headers API Previously it was not possible to get multiple header values of a
response. This is now possible and additional helper functions are
available:Request.allHeaders()Request.headersArray()Request.headerValue(name:
string)Response.allHeaders()Response.headersArray()Response.headerValue(name:
string)Response.headerValues(name: string)Ø<ß Forced-Colors emulation Its now possible
to emulate the forced-colors CSS media feature by passing it in the context options or
calling Page.emulateMedia().New APIs Page.route() accepts new times option to
specify how many times this route should be matched.Page.setChecked(selector:
string, checked: boolean) and Locator.setChecked(selector: string, checked: boolean)
was introduced to set the checked state of a checkbox.Request.sizes() Returns
resource size information for given http request.BrowserContext.tracing.startChunk() -
Start a new trace chunk.BrowserContext.tracing.stopChunk() - Stops a new trace chunk.
Ø<ß- Playwright Test Ø>Ý test.parallel() run tests in the same file in parallel
test.describe.parallel('group', () => { test('runs in parallel 1', async ({ page }) => { });
test('runs in parallel 2', async ({ page }) => { });});By default, tests in a single file are run
in order. If you have many independent tests in a single file, you can now run them in
parallel with test.describe.parallel(title, callback).Ø=Þà Add --debug CLI flag By using npx
playwright test --debug it will enable the Playwright Inspector for you to debug your
tests.Browser Versions Chromium 96.0.4641.0Mozilla Firefox 92.0WebKit 15.0Version
1.14 Ø<ß- Playwright Library &¡þ New "strict" mode Selector ambiguity is a common problem in
automation testing. "strict" mode ensures that your selector points to a single element
and throws otherwise.Pass strict: true into your action calls to opt in.// This will throw if

you have more than one button!await page.click('button', { strict: true });Ø=ÜÍ New Locators API Locator represents a view to the element(s) on the page. It captures the logic sufficient to retrieve the element at any given moment.The difference between the Locator and ElementHandle is that the latter points to a particular element, while Locator captures the logic of how to retrieve that element.Also, locators are "strict" by default!const locator = page.locator('button');await locator.click();Learn more in the documentation.Ø>Ýé Experimental React and Vue selector engines React and Vue selectors allow selecting elements by its component name and/or property values. The syntax is very similar to attribute selectors and supports all attribute selector operators.await page.locator('_react=SubmitButton[enabled=true]').click();await page.locator('_vue=submit-button[enabled=true]').click();Learn more in the react selectors documentation and the vue selectors documentation.'(New nth and visible selector engines nth selector engine is equivalent to the :nth-match pseudo class, but could be combined with other selector engines.visible selector engine is equivalent to the :visible pseudo class, but could be combined with other selector engines.// select the first button among all buttonsawait button.click('button >> nth=0');// or if you are using locators, you can use first(), nth() and last()await page.locator('button').first().click();// click a visible buttonawait button.click('button >> visible=true');Ø<ß- Playwright Test ' Web-First Assertions expect now supports lots of new web-first assertions.Consider the following example:await expect(page.locator('.status')).toHaveText('Submitted');Playwright Test will be re-testing the node with the selector .status until fetched Node has the "Submitted" text. It will be re-fetching the node and checking it over and over, until the condition is met or until the timeout is reached. You can either pass this timeout or configure it once via the testProject.expect value in test config.By default, the timeout for assertions is not set, so it'll wait forever, until the whole test times out.List of all new assertions:expect(locator).toBeChecked()expect(locator).toBeDisabled()expect(locator).toBeEditable()expect(locator).toBeEmpty()expect(locator).toBeEnabled()expect(locator).toBeFocused()expect(locator).toBeHidden()expect(locator).toBeVisible()expect(locator).toContainText(text, options?)expect(locator).toHaveAttribute(name, value)expect(locator).toHaveClass(expected)expect(locator).toHaveCount(count)expect(locator).toHaveCSS(name, value)expect(locator).toHaveId(id)expect(locator).toHaveJSProperty(name, value)expect(locator).toHaveText(expected, options)expect(page).toHaveTitle(title)expect(page).toHaveURL(url)expect(locator).toHaveValue(value)&Ó Serial mode with describe.serial Declares a group of tests that should always be run serially. If one of the tests fails, all subsequent tests are skipped. All tests in a group are retried together.test.describe.serial('group', () => { test('runs first', async ({ page }) => { /* ... */ }); test('runs second', async ({ page }) => { /* ... */ });});Learn more in the documentation.Ø=Ü> Steps API with test.step Split long tests into multiple steps using test.step() API:import { test, expect } from '@playwright/test';test('test', async ({ page }) => { await test.step('Log in', async () => {   // ... }); await test.step('news feed', async () => {   // ... });});Step information is exposed in reporters API.Ø<ß Launch web server before running tests To launch a server during the tests, use the webServer option in the configuration file. The server will wait for a given url to be available before running the tests, and the url will be passed over to Playwright as a baseURL when creating a context.playwright.config.tsimport { defineConfig } from '@playwright/test';export default

defineConfig({ webServer: {   command: 'npm run start', // command to launch   url: 'http://127.0.0.1:3000', // url to await for   timeout: 120 * 1000,   reuseExistingServer: ! process.env.CI, },});Learn more in the documentation.Browser Versions Chromium 94.0.4595.0Mozilla Firefox 91.0WebKit 15.0Version 1.13 Playwright Test &¡þ Introducing Reporter API which is already used to create an Allure Playwright reporter.&úþ New baseURL fixture to support relative paths in tests.Playwright Ø=Ý– Programmatic drag-and-drop support via the page.dragAndDrop() API.Ø=Ý Enhanced HAR with body sizes for requests and responses. Use via recordHar option in browser.newContext().Tools Playwright Trace Viewer now shows parameters, returned values and console.log() calls.Playwright Inspector can generate Playwright Test tests.New and Overhauled Guides IntroAuthenticationChrome ExtensionsPlaywright Test AnnotationsPlaywright Test ConfigurationPlaywright Test FixturesBrowser Versions Chromium 93.0.4576.0Mozilla Firefox 90.0WebKit 14.2New Playwright APIs new baseURL option in browser.newContext() and browser.newPage()response.securityDetails() and response.serverAddr()page.dragAndDrop() and frame.dragAndDrop()download.cancel()page.inputValue(), frame.inputValue() and elementHandle.inputValue()new force option in page.fill(), frame.fill(), and elementHandle.fill()new force option in page.selectOption(), frame.selectOption(), and elementHandle.selectOption()Version 1.12 &¡þ Introducing Playwright Test Playwright Test is a new test runner built from scratch by Playwright team specifically to accommodate end-to-end testing needs:Run tests across all browsers.Execute tests in parallel.Enjoy context isolation and sensible defaults out of the box.Capture videos, screenshots and other artifacts on failure.Integrate your POMs as extensible fixtures.Installation:npm i -D @playwright/testSimple test tests/foo.spec.ts:import { test, expect } from '@playwright/test';test('basic test', async ({ page }) => { await page.goto('https://playwright.dev/'); const name = await page.innerText('.navbar__title'); expect(name).toBe('Playwright');});Running:npx playwright testØ=ÜI Read more in Playwright Test documentation.Ø>Ýß &Bþ Introducing Playwright Trace Viewer Playwright Trace Viewer is a new GUI tool that helps exploring recorded Playwright traces after the script ran. Playwright traces let you examine:page DOM before and after each Playwright actionpage rendering before and after each Playwright actionbrowser network during script executionTraces are recorded using the new browserContext.tracing API:const browser = await chromium.launch();const context = await browser.newContext();// Start tracing before creating / navigating a page.await context.tracing.start({ screenshots: true, snapshots: true });const page = await context.newPage();await page.goto('https://playwright.dev');// Stop tracing and export it into a zip archive.await context.tracing.stop({ path: 'trace.zip' });Traces are examined later with the Playwright CLI:npx playwright show-trace trace.zipThat will open the following GUI:Ø=ÜI Read more in trace viewer documentation.Browser Versions Chromium 93.0.4530.0Mozilla Firefox 89.0WebKit 14.2This version of Playwright was also tested against the following stable channels:Google Chrome 91Microsoft Edge 91New APIs reducedMotion option in page.emulateMedia(), browserType.launchPersistentContext(), browser.newContext() and browser.newPage()browserContext.on('request')browserContext.on('requestfailed')browserContext.on('requestfinished')browserContext.on('response')tracesDir option in browserType.launch() and browserType.launchPersistentContext()new browserContext.tracing API namespacenew download.page() methodVersion 1.11 Ø<ß¥

New video: Playwright: A New Test Automation Framework for the Modern Web (slides)We talked about PlaywrightShowed engineering work behind the scenesDid live demos with new features '(Special thanks to applitools for hosting the event and inviting us!Browser Versions Chromium 92.0.4498.0Mozilla Firefox 89.0b6WebKit 14.2New APIs support for async predicates across the API in methods such as page.waitForRequest() and othersnew emulation devices: Galaxy S8, Galaxy S9+, Galaxy Tab S4, Pixel 3, Pixel 4new methods:page.waitForURL() to await navigations to URLvideo.delete() and video.saveAs() to manage screen recordingnew options:screen option in the browser.newContext() method to emulate window.screen dimensionsposition option in page.check() and page.uncheck() methodstrial option to dry-run actions in page.check(), page.uncheck(), page.click(), page.dblclick(), page.hover() and page.tap()Version 1.10 Playwright for Java v1.10 is now stable!Run Playwright against Google Chrome and Microsoft Edge stable channels with the new channels API.Chromium screenshots are fast on Mac & Windows.Bundled Browser Versions Chromium 90.0.4430.0Mozilla Firefox 87.0b10WebKit 14.2This version of Playwright was also tested against the following stable channels:Google Chrome 89Microsoft Edge 89New APIs browserType.launch() now accepts the new 'channel' option. Read more in our documentation.Version 1.9 Playwright Inspector is a new GUI tool to author and debug your tests.Line-by-line debugging of your Playwright scripts, with play, pause and step-through.Author new scripts by recording user actions.Generate element selectors for your script by hovering over elements.Set the PWDEBUG=1 environment variable to launch the InspectorPause script execution with page.pause() in headed mode. Pausing the page launches Playwright Inspector for debugging.New has-text pseudo-class for CSS selectors. :has-text("example") matches any element containing "example" somewhere inside, possibly in a child or a descendant element. See more examples.Page dialogs are now auto-dismissed during execution, unless a listener for dialog event is configured. Learn more about this.Playwright for Python is now stable with an idiomatic snake case API and pre-built Docker image to run tests in CI/CD.Browser Versions Chromium 90.0.4421.0Mozilla Firefox 86.0b10WebKit 14.1New APIs page.pause().Version 1.8 Selecting elements based on layout with :left-of(), :right-of(), :above() and :below().Playwright now includes command line interface, former playwright-cli.npx playwright --helppage.selectOption() now waits for the options to be present.New methods to assert element state like page.isEditable().New APIs elementHandle.isChecked().elementHandle.isDisabled().elementHandle.isEditable().elementHandle.isEnabled().elementHandle.isHidden().elementHandle.isVisible().page.isChecked().page.isDisabled().page.isEditable().page.isEnabled().page.isHidden().page.isVisible().New option 'editable' in elementHandle.waitForElementState().Browser Versions Chromium 90.0.4392.0Mozilla Firefox 85.0b5WebKit 14.1Version 1.7 New Java SDK: Playwright for Java is now on par with JavaScript, Python and .NET bindings.Browser storage API: New convenience APIs to save and load browser storage state (cookies, local storage) to simplify automation scenarios with authentication.New CSS selectors: We heard your feedback for more flexible selectors and have revamped the selectors implementation. Playwright 1.7 introduces new CSS extensions and there's more coming soon.New website: The docs website at playwright.dev has been updated and is now built with Docusaurus.Support for Apple Silicon: Playwright browser binaries for WebKit and Chromium are now built for Apple Silicon.New APIs browserContext.storageState() to

get current state for later reuse.storageState option in browser.newContext() and browser.newPage() to setup browser context state.Browser Versions Chromium 89.0.4344.0Mozilla Firefox 84.0b9WebKit 14.1
Canary releasesPlaywright for Node.js has a canary releases system.It permits you to test new unreleased features instead of waiting for a full release. They get released daily on the next NPM tag of Playwright.It is a good way to give feedback to maintainers, ensuring the newly implemented feature works as intended.note Using a canary release in production might seem risky, but in practice, it's not. A canary release passes all automated tests and is used to test e.g. the HTML report, Trace Viewer, or Playwright Inspector with end-to-end tests. npm install -D @playwright/test@nextNext npm Dist Tag For any code-related commit on main, the continuous integration will publish a daily canary release under the @next npm dist tag.You can see on npm the current dist tags:latest: stable releasesnext: next releases, published dailybeta: after a release-branch was cut, usually a week before a stable release each commit gets published under this tagUsing a Canary Release npm install -D @playwright/test@nextDocumentation The stable and the next documentation is published on playwright.dev. To see the next documentation, press Shift on the keyboard 5 times.
Test configurationPlaywright has many options to configure how your tests are run. You can specify these options in the configuration file. Note that test runner options are top-level, do not put them into the use section.Basic Configuration Here are some of the most common configuration options.import { defineConfig, devices } from '@playwright/test';export default defineConfig({ // Look for test files in the "tests" directory, relative to this configuration file. testDir: 'tests', // Run all tests in parallel. fullyParallel: true, // Fail the build on CI if you accidentally left test.only in the source code. forbidOnly: !!process.env.CI, // Retry on CI only. retries: process.env.CI ? 2 : 0, // Opt out of parallel tests on CI. workers: process.env.CI ? 1 : undefined, // Reporter to use reporter: 'html', use: {   // Base URL to use in actions like `await page.goto('/')`.   baseURL: 'http://127.0.0.1:3000',   // Collect trace when retrying the failed test.   trace: 'on-first-retry', }, // Configure projects for major browsers. projects: [{   name: 'chromium',   use: { ...devices['Desktop Chrome'] },   },], // Run your local dev server before starting the tests. webServer: {   command: 'npm run start',   url: 'http://127.0.0.1:3000',   reuseExistingServer: !process.env.CI, },});OptionDescriptiontestConfig.forbidOnlyWhether to exit with an error if any tests are marked as test.only. Useful on CI.testConfig.fullyParallelhave all tests in all files to run in parallel. See /Parallelism and sharding for more details.testConfig.projectsRun tests in multiple configurations or on multiple browserstestConfig.reporterReporter to use. See Test Reporters to learn more about which reporters are available.testConfig.retriesThe maximum number of retry attempts per test. See Test Retries to learn more about retries.testConfig.testDirDirectory with the test files.testConfig.useOptions with use{}testConfig.webServerTo launch a server during the tests, use the webServer optiontestConfig.workersThe maximum number of concurrent worker processes to use for parallelizing tests. Can also be set as percentage of logical CPU cores, e.g. '50%'.. See /Parallelism and sharding for more details.Filtering Tests Filter tests by glob patterns or regular expressions.playwright.config.tsimport { defineConfig } from '@playwright/test';export

default defineConfig({ // Glob patterns or regular expressions to ignore test files. testIgnore: '*test-assets', // Glob patterns or regular expressions that match test files. testMatch: '*todo-tests/*.spec.ts',});OptionDescriptiontestConfig.testIgnoreGlob patterns or regular expressions that should be ignored when looking for the test files. For example, '*test-assets'testConfig.testMatchGlob patterns or regular expressions that match test files. For example, '*todo-tests/*.spec.ts'. By default, Playwright runs .*(test|spec).(js|ts|mjs) files.Advanced Configuration playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ // Folder for test artifacts such as screenshots, videos, traces, etc. outputDir: 'test-results', // path to the global setup files. globalSetup: require.resolve('./global-setup'), // path to the global teardown files. globalTeardown: require.resolve('./global-teardown'), // Each test is given 30 seconds. timeout: 30000,});OptionDescriptiontestConfig.globalSetupPath to the global setup file. This file will be required and run before all the tests. It must export a single function.testConfig.globalTeardownPath to the global teardown file. This file will be required and run after all the tests. It must export a single function.testConfig.outputDirFolder for test artifacts such as screenshots, videos, traces, etc.testConfig.timeoutPlaywright enforces a timeout for each test, 30 seconds by default. Time spent by the test function, fixtures, beforeEach and afterEach hooks is included in the test timeout.Expect Options Configuration for the expect assertion library.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ expect: {   // Maximum time expect() should wait for the condition to be met.   timeout: 5000,   toHaveScreenshot: {   // An acceptable amount of pixels that could be different, unset by default.   maxDiffPixels: 10,   },   toMatchSnapshot: {   // An acceptable ratio of pixels that are different to the total amount of pixels, between 0 and 1.   maxDiffPixelRatio: 0.1,   }, },});OptionDescriptiontestConfig.expectWeb first assertions like expect(locator).toHaveText() have a separate timeout of 5 seconds by default. This is the maximum time the expect() should wait for the condition to be met. Learn more about test and expect timeouts and how to set them for a single test.expect(page).toHaveScreenshot()Configuration for the expect(locator).toHaveScreeshot() method.expect(snapshot).toMatchSnapshot()Configuration for the expect(locator).toMatchSnapshot() method.Add custom matchers using expect.extend You can extend Playwright assertions by providing custom matchers. These matchers will be available on the expect object.In this example we add a custom toBeWithinRange function in the configuration file. Custom matcher should return a message callback and a pass flag indicating whether the assertion passed.TypeScriptJavaScriptplaywright.config.tsimport { expect, defineConfig } from '@playwright/test';expect.extend({ toBeWithinRange(received: number, floor: number, ceiling: number) {   const pass = received >= floor && received <= ceiling;   if (pass) {   return {   message: () => 'passed',   pass: true,   };   } else {   return {   message: () => 'failed',   pass: false,   };   } },});export default defineConfig({});playwright.config.tsconst { expect, defineConfig } = require('@playwright/test');expect.extend({ toBeWithinRange(received, floor, ceiling) {   const pass = received >= floor && received <= ceiling;   if (pass) {   return {   message: () => 'passed',   pass: true,   };   } else {   return {   message: () => 'failed',   pass: false,   };   } },});module.exports = defineConfig({});Now we

can use toBeWithinRange in the test.example.spec.tsimport { test, expect } from '@playwright/test';test('numeric ranges', () => { expect(100).toBeWithinRange(90, 110); expect(101).not.toBeWithinRange(0, 100);});noteDo not confuse Playwright's expect with the expect library. The latter is not fully integrated with Playwright test runner, so make sure to use Playwright's own expect.For TypeScript, also add the following to your global.d.ts. If it does not exist, you need to create it inside your repository. Make sure that your global.d.ts gets included inside your tsconfig.json via the include or compilerOptions.typeRoots option so that your IDE will pick it up.You don't need it for JavaScript.global.d.tsexport {};declare global { namespace PlaywrightTest { interface Matchers<R, T> { toBeWithinRange(a: number, b: number): R; } }}
Test use optionsIn addition to configuring the test runner you can also configure Emulation, Network and Recording for the Browser or BrowserContext. These options are passed to the use: {} object in the Playwright config.Basic Options Set the base URL and storage state for all tests:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Base URL to use in actions like `await page.goto('/')`. baseURL: 'http://127.0.0.1:3000', // Populates context with given storage state. storageState: 'state.json', },});OptionDescriptiontestOptions.baseURLBase URL used for all pages in the context. Allows navigating by using just the path, for example page.goto('/settings').testOptions.storageStatePopulates context with given storage state. Useful for easy authentication, learn more.Emulation Options With Playwright you can emulate a real device such as a mobile phone or tablet. See our guide on projects for more info on emulating devices. You can also emulate the "geolocation", "locale" and "timezone" for all tests or for a specific test as well as set the "permissions" to show notifications or change the "colorScheme". See our Emulation guide to learn more.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Emulates `'prefers-colors-scheme'` media feature. colorScheme: 'dark', // Context geolocation. geolocation: { longitude: 12.492507, latitude: 41.889938 }, // Emulates the user locale. locale: 'en-GB', // Grants specified permissions to the browser context. permissions: ['geolocation'], // Emulates the user timezone. timezoneId: 'Europe/Paris', // Viewport used for all pages in the context. viewport: { width: 1280, height: 720 }, },});OptionDescriptiontestOptions.colorSchemeEmulates 'prefers-colors-scheme' media feature, supported values are 'light', 'dark', 'no-preference'testOptions.geolocationContext geolocation.testOptions.localeEmulates the user locale, for example en-GB, de-DE, etc.testOptions.permissionsA list of permissions to grant to all pages in the context.testOptions.timezoneIdChanges the timezone of the context.testOptions.viewportViewport used for all pages in the context.Network Options Available options to configure networking:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Whether to automatically download all the attachments. acceptDownloads: false, // An object containing additional HTTP headers to be sent with every request. extraHTTPHeaders: { 'X-My-Header': 'value', }, // Credentials for HTTP authentication. httpCredentials: { username: 'user', password: 'pass', }, // Whether to ignore HTTPS errors during

navigation. ignoreHTTPSErrors: true, // Whether to emulate network being offline. offline: true, // Proxy settings used for all pages in the test. proxy: { server: 'http://myproxy.com:3128', bypass: 'localhost', }, },});OptionDescriptiontestOptions.acceptDownloadsWhether to automatically download all the attachments, defaults to true. Learn more about working with downloads.testOptions.extraHTTPHeadersAn object containing additional HTTP headers to be sent with every request. All header values must be strings.testOptions.httpCredentialsCredentials for HTTP authentication.testOptions.ignoreHTTPSErrorsWhether to ignore HTTPS errors during navigation.testOptions.offlineWhether to emulate network being offline.testOptions.proxyProxy settings used for all pages in the test.noteYou don't have to configure anything to mock network requests. Just define a custom Route that mocks the network for a browser context. See our network mocking guide to learn more.Recording Options With Playwright you can capture screenshots, record videos as well as traces of your test. By default these are turned off but you can enable them by setting the screenshot, video and trace options in your playwright.config.js file. Trace files, screenshots and videos will appear in the test output directory, typically test-results.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Capture screenshot after each test failure. screenshot: 'only-on-failure', // Record trace only when retrying a test for the first time. trace: 'on-first-retry', // Record video only when retrying a test for the first time. video: 'on-first-retry' },});OptionDescriptiontestOptions.screenshotCapture screenshots of your test. Options include 'off', 'on' and 'only-on-failure'testOptions.tracePlaywright can produce test traces while running the tests. Later on, you can view the trace and get detailed information about Playwright execution by opening Trace Viewer. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry'testOptions.videoPlaywright can record videos for your tests. Options include: 'off', 'on', 'retain-on-failure' and 'on-first-retry'Other Options playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { // Maximum time each action such as `click()` can take. Defaults to 0 (no limit). actionTimeout: 0, // Name of the browser that runs tests. For example `chromium`, `firefox`, `webkit`. browserName: 'chromium', // Toggles bypassing Content-Security-Policy. bypassCSP: true, // Channel to use, for example "chrome", "chrome-beta", "msedge", "msedge-beta". channel: 'chrome', // Run browser in headless mode. headless: false, // Change the default data-testid attribute. testIdAttribute: 'pw-test-id', },});OptionDescriptiontestOptions.actionTimeoutTimeout for each Playwright action in milliseconds. Defaults to 0 (no timeout). Learn more about timeouts and how to set them for a single test.testOptions.browserNameName of the browser that runs tests. Defaults to 'chromium'. Options include chromium, firefox, or webkit.testOptions.bypassCSPToggles bypassing Content-Security-Policy. Useful when CSP includes the production origin. Defaults to false.testOptions.channelBrowser channel to use. Learn more about different browsers and channels.testOptions.headlessWhether to run the browser in headless mode meaning no browser is shown when running tests. Defaults to true.testOptions.testIdAttributeChanges the default data-testid attribute used by Playwright locators.More browser and context options Any options accepted by browserType.launch() or browser.newContext() can be put into launchOptions or

contextOptions respectively in the use section.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { launchOptions: { slowMo: 50, }, },});However, most common ones like headless or viewport are available directly in the use section - see basic options, emulation or network.Explicit Context Creation and Option Inheritance If using the built-in browser fixture, calling browser.newContext() will create a context with options inherited from the config:playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { userAgent: 'some custom ua', viewport: { width: 100, height: 100 }, },});An example test illustrating the initial context options are set:import { test, expect } from '@playwright/test';test('should inherit use options on context when using built-in browser fixture', async ({ browser }) => { const context = await browser.newContext(); const page = await context.newPage(); expect(await page.evaluate(() => navigator.userAgent)).toBe('some custom ua'); expect(await page.evaluate(() => window.innerWidth)).toBe(100); await context.close();});Configuration Scopes You can configure Playwright globally, per project, or per test. For example, you can set the locale to be used globally by adding locale to the use option of the Playwright config, and then override it for a specific project using the project option in the config. You can also override it for a specific test by adding test.use({}) in the test file and passing in the options.playwright.config.tsimport { defineConfig } from '@playwright/test';export default defineConfig({ use: { locale: 'en-GB' },});You can override options for a specific project using the project option in the Playwright config.import { defineConfig, devices } from '@playwright/test';export default defineConfig({ projects: [{ name: 'chromium', use: { ...devices['Desktop Chrome'], locale: 'de-DE', }, },],});You can override options for a specific test file by using the test.use() method and passing in the options. For example to run tests with the French locale for a specific test:import { test, expect } from '@playwright/test';test.use({ locale: 'fr-FR' });test('example', async ({ page }) => { // ...});The same works inside a describe block. For example to run tests in a describe block with the French locale:import { test, expect } from '@playwright/test';test.describe('french language block', () => { test.use({ locale: 'fr-FR' }); test('example', async ({ page }) => { // ... });});