Playwright

Who am I?

- Roy de Kleijn
  - > 15 years experience in software testing
  - Freelancer also work for BrowserStack
- Website: https://testsmith.io
- Email: roy@testsmith.io
- LinkedIn: https://www.linkedin.com/in/roydekleijn/

Online Training Rules

1. Camera on please 📹
   So we can add a little more interactivity

2. Phone off 📴

3. Eat during breaks 🍕

4. Switch off your microphone 🎤 or 📳
   Because, we are with many people

5. Please, raise your hand if you have any question 🖐

Timetable

9:30 🕘	Start
12:00 🕛 - 13:00 🕐	Lunch
16:30 🕟	End

- Regularly 10-minute breaks. 🚻

Training material

After/during this training all training material (slides and code) will be distributed.

Introduction

- What do you do?
- How did you get into testing?
- Background & experience
  - Testing
  - Programming
  - Automation, tools
- Why are you here?
  - Your objectives
  - Problems in testing and questions you have

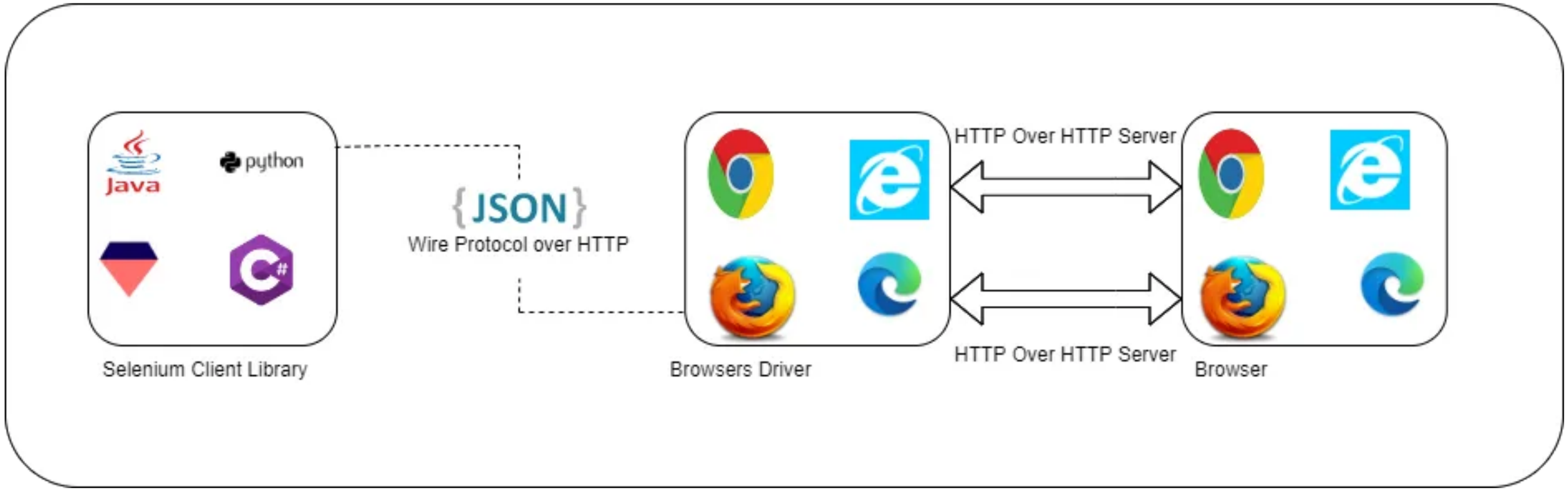
Agenda

- Introduction to Playwright
- Playwright Installation
  - Run the Tests
  - Show Results
- Playwright Configuration
- Web Test Automation
- Test Generation with Codegen
- Trace Viewer
- UI Mode
- Visual Testing
- Improve Maintainability (Page Objects)
- API Test Automation

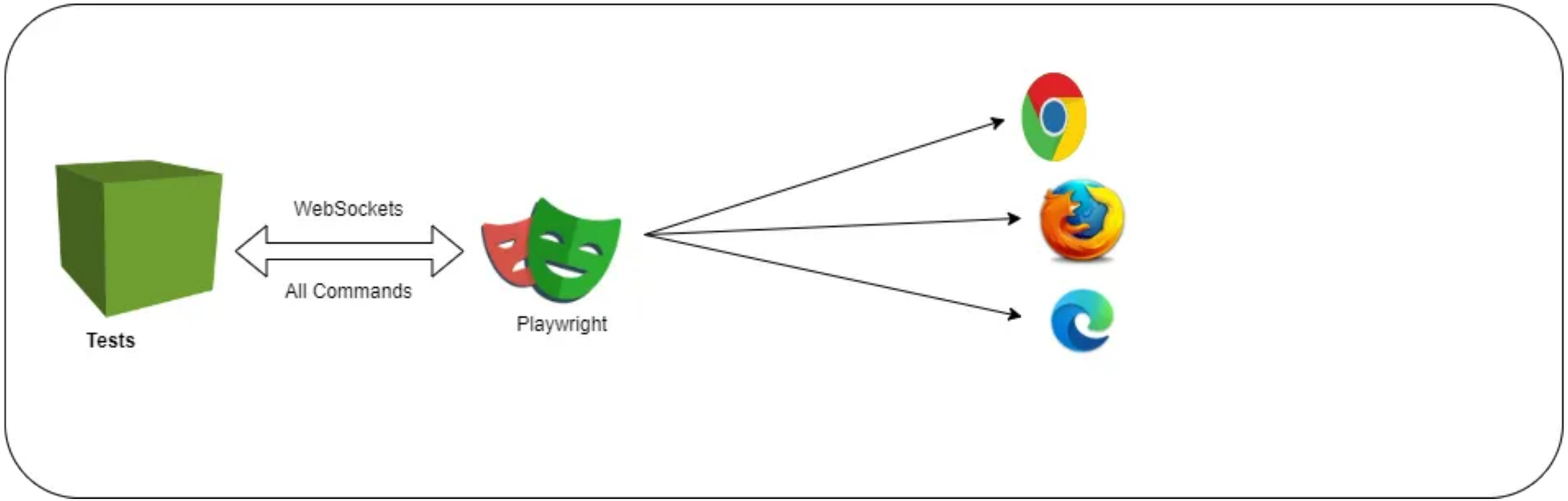
Introduction to Playwright

- Designed for speed
- Developer (Tester) - friendly tools
- multiple languages
  - Java, JS / TS, Python, .NET
- Mobile emulation
- Works well with tabs and iframes

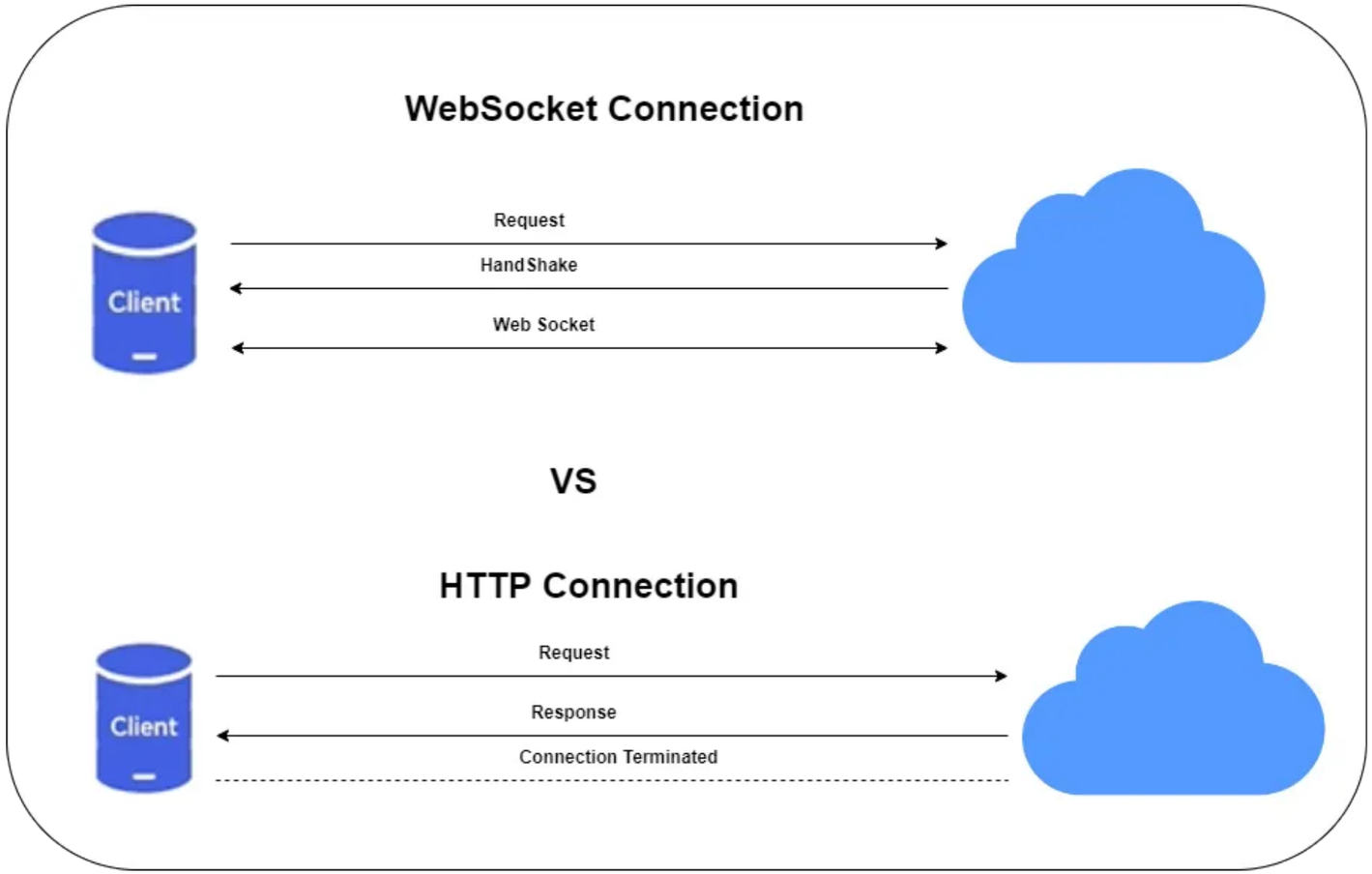
Playwright vs. Selenium



Playwright vs. Selenium



Playwright vs. Selenium



Playwright Installation

- Separate project for Test Automation:
  - In an empty folder execute: `npm init playwright@latest`
- As part of application code:
  - In the project folder execute: `npm init playwright@latest`
- Playwright creates a default project structure

Run the Tests

By default in all 3 browsers and in headless mode:

```
npx playwright test
```

Or developer friendly mode with time travel debugging, watch mode.

```
npx playwright test --ui
```

With browser window

```
npx playwright test --headed
```

With browser window & parallel = 1

```
npx playwright test --headed --workers=1
```

Show Results

```
npx playwright show-report
```

VSCode Extension

- Playwright Test for VSCode
  - Install Playwright -> from the command panel
  - Run Single Test
  - Run Multiple Tests
  - Debug
  - Record new tests

Playwright Test Annotations

- `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.

Playwright Configuration

- baseURL
- trace
- testIdAttribute
- screenshot
- projects --> browser configuration
- filtering, `testIgnore` or `testMatch`

Web Test Automation

- Locate Elements
- Frequently Used Actions
- Frequently Used Assertions

Locate Elements

- `page.getByRole()` to locate by explicit and implicit accessibility attributes.
  - Locate elements by their ARIA role, ARIA attributes and accessible name.
- `page.getByText()` to locate by text content.
- `page.getByLabel()` to locate a form control by associated label's text.
- `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 attribute.
- `page.getByTestId()` to locate an element based on its data-testid attribute (or other attribute).

Set Different attribute

In code:

```
JavaScript
selectors.setTestIdAttribute('data-test');
```

In Configuration file:

```
use: {
    testIdAttribute: 'data-test'
}
```

Locate by CSS or Xpath

```
await page.locator('css=button').click();
await page.locator('xpath=//button').click();

await page.locator('button').click();
await page.locator('//button').click();
```

Filtering

```
await page
    .getByRole('listitem')
    .filter({ hasText: 'Product 2' })
    .getByRole('button', { name: 'Add to cart' })
    .click();
```

Frequently Used Actions

page.goto()	Navigate to a specific page
locator.check()	Check the input checkbox
locator.click()	Click the element
locator.uncheck()	Uncheck the input checkbox
locator.hover()	Hover mouse over the element
locator.fill()	Fill the form field (fast)
locator.focus()	Focus the element
locator.press()	Press single key
locator.setInputFiles()	Pick files to upload
locator.selectOption()	Select option in the drop down
locator.type()	Type text character by character (slow)

Frequently Used Assertions

expect(locator).toBeChecked()	Checkbox is checked
expect(locator).toBeEnabled()	Control is enabled
expect(locator).toBeVisible()	Element is visible
expect(locator).toContainText()	Element contains text
expect(locator).toHaveAttribute()	Element has attribute
expect(locator).toHaveCount()	List of elements has given length
expect(locator).toHaveText()	Element matches text
expect(locator).toHaveValue()	Input element has value
expect(page).toHaveTitle()	Page has title
expect(page).toHaveURL()	Page has URL
expect(page).toHaveScreenshot()	Page has screenshot

Test Generation with Codegen

Gives some nice inspiration, but code cleanup is needed!

```
npx playwright codegen <url>
```

url is optional.

```
npx playwright codegen https://practicesoftwaretesting.com
```

Faker-js

Faker to generate (random) test data

- **Add package:**
  `npm i @faker-js/faker --save-dev`
- **Using Faker:**

```
//first line in file

import { faker } from '@faker-js/faker';

faker.name.firstName();

faker.internet.email();
```

https://github.com/faker-js/faker#api

Exercises

- Implement tests (register, login, contact, search, navigation)
- Try to use codegen
- Run the tests

Trace Viewer

You can get the following insights: action logs, snapshots, network logs, meta data

Configure tracing in `playwright.config.ts`, using one of the 4 options:

- `off` - Do not record a trace.
- `on` - Record a trace for each test.
- `retain-on-failure` - Record a trace for each test, but remove it from successful test runs.
- `on-first-retry` - Record a trace only when retrying a test for the first time.

Trace Viewer (example config)

```
use: {

    /* Collect trace when retrying the failed test. See https://playwright.dev/docs/trace-viewer */
    trace: 'retain-on-failure'
},
```

Auto-waiting

It auto-waits for all the relevant checks to pass and only then performs the requested action.

Action	Attached	Visible	Stable	Receives Events	Enabled	Editable
click	Yes	Yes	Yes	Yes	Yes	-
fill	Yes	Yes	-	-	Yes	Yes
selectOption	Yes	Yes	-	-	Yes	-
type	Yes	-	-	-	-	-

https://playwright.dev/docs/actionability

Visual Testing

- `await expect(page).toHaveScreenshot();`
- `await expect(page).toHaveScreenshot('landing.png');`
- First time Playwright will generate reference screenshot.
- options
  - fullPage
  - maxDiffPixels
  - omitBackground
- `await expect(page).toHaveScreenshot({ maxDiffPixels : 100 });`
- config:

```javascript
export default defineConfig({
expect: {
  toHaveScreenshot: { maxDiffPixels: 100 },
},
});
```

Exercise

Create a visual test for the product detail page.

Page Object Model

Pages (homepage)

```
import { expect, Locator, Page } from '@playwright/test';

export class HomePage {
    readonly url = "https://practicesoftwaretesting.com";
    readonly page: Page;
    readonly logo: Locator;
    readonly contactMenu: Locator;

    constructor(page: Page) {
        this.page = page;
        this.logo = page.locator('#logo');
        this.contactMenu = page.getByTestId('nav-contact');
    }

    async goto() {
        await this.page.goto(this.url);
    }
    async clickOnContact() {
        await this.contactMenu.waitFor({ state: "visible" });
        await this.contactMenu.click();
    }
}
```

Test

```
import { HomePage } from './pages/home.page';

test('Navigate to contact page', async ({ page }) => {
  const homepage = new HomePage(page);
  await homepage.goto();
  await homepage.clickOnContact();
});
```

Pages (contactpage)

```
import { expect, Locator, Page } from '@playwright/test';

export class ContactPage {
    readonly page: Page;
    readonly message: Locator;

    constructor(page: Page) {
        this.page = page;
        this.message = page.getByTestId('message');
    }

    async getMessage(): Promise<string> { {
        await this.contactMenu.waitFor({ state: "visible" });
        return await this.message.textContent();
    }
}
```

Exercise

Implement a page object model for the contact and login test.

API Testing (GET)

```
test('should retrieve all brands', async ({ request }) => {
  const response = await request.get(`https://api.practicesoftwaretesting.com/brands`);
  expect(response.ok()).toBeTruthy();
});
```

API Testing (POST)

```
test('should authenticate', async ({ request }) => {
  const tokenResponse = await request.post(`https://api.practicesoftwaretesting.com/users/login`, {
    data: {
      email: 'customer@practicesoftwaretesting.com',
      password: 'welcome01',
    }
  });
  expect(tokenResponse.ok()).toBeTruthy();
});
```

API testing parse response

```
const body = await response.json();
```

API Testing (log response)

```
console.log((await response.json()));
```

API Testing (protected endpoint)

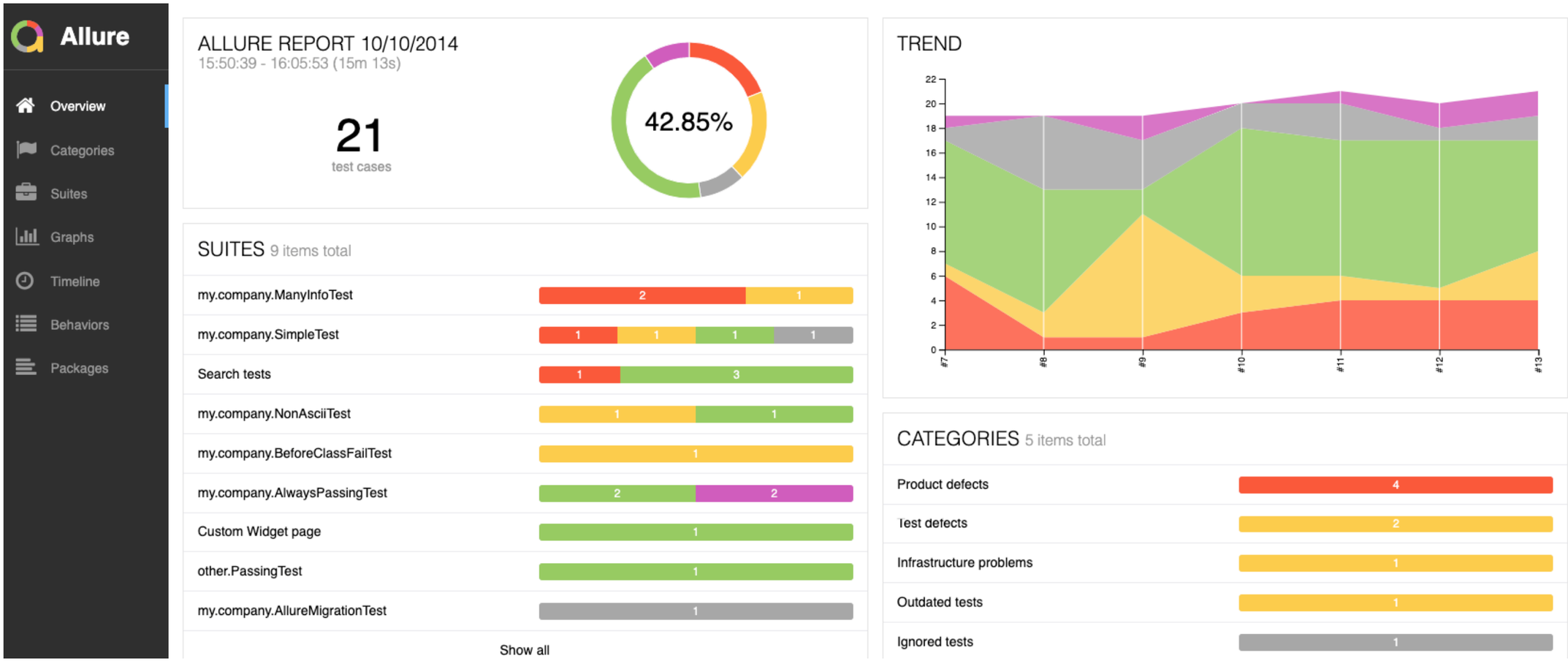
```
test('should authenticate', async ({ request }) => {
  const response = await request.post(`https://api.practicesoftwaretesting.com/users/login`, {
    data: {
      email: 'customer@practicesoftwaretesting.com',
      password: 'welcome01',
    }
  });
  expect(response.ok()).toBeTruthy();

  const tokenResponse = await response.json();

  const invoices = await request.get(`https://api.practicesoftwaretesting.com/invoices`, {
    headers: { Authorization: `Bearer ${tokenResponse.access_token}` },
  });
  expect(invoices.ok()).toBeTruthy();
});
```

Exercise

Implement a few different (GET / POST / PUT) API tests.

Implement an API test for a protected endpoint.

Allure Reporting



Add Allure package

```
npm i -D allure-playwright
```

Source: https://www.npmjs.com/package/allure-playwright

Adjust Playwright config

File: `playwright.config.ts`

```
import { testPlanFilter } from "allure-playwright/dist/testplan";

export default defineConfig({
  .....
  grep: testPlanFilter(),
  reporter: [["html"], ["line"], ["allure-playwright"]],

});
```

Add extra reporting information

```
import { allure } from "allure-playwright";

await allure.label("labelName", "labelValue");
await allure.issue("Issue Name", "/issues/352");
await allure.story("Some Story");
```

Execute test

```
npx playwright run
```

Generate report

Install Allure:

```
npm i -D allure-commandline
```

Assuming allure is already installed:

- serve report based on current "allure-results" folder: allure serve
- generate new report based on current "allure-results" folder: allure generate
- open generated report from "allure-report" folder: allure open
- https://docs.qameta.io/allure/#_windows

```
npx allure serve
npx allure generate
```