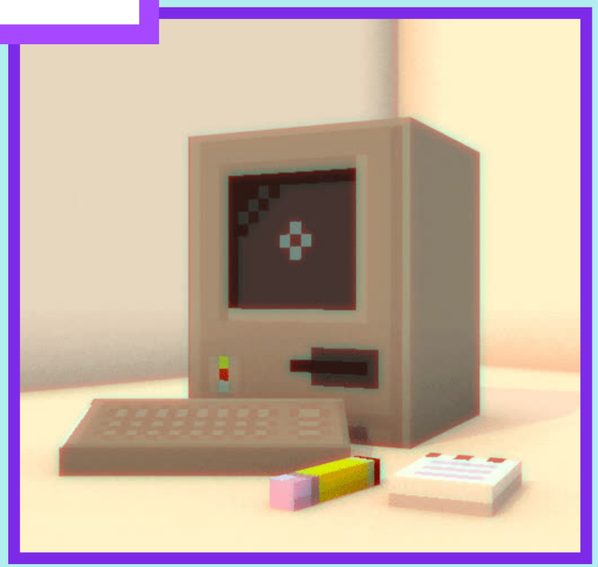
Automated UI Testing



Presented by
Tom Isles

Canva

What We'll Talk About

- Recap on some theory
- What is integration testing?
- How do we perform integration testing in a browser environment?
- Framework #1 (Theory): Selenium, Webdriver and DOM Selectors
- Flakiness
- Framework #2 (Practice): Cypress
- Demo

Recap:
Why test?

Testing is the ultimate guardrail against breaking software.

It allows us to understand our code, and make changes without fear of introducing bugs.

It improves the quality of your code immensely and the speed at which you can write new code.

A practical take...

Without tests you have no way of knowing that your application works.

That means it could break at any time.

And if it breaks at any time, who is going to have to fix it?
You.

When are you going to have to fix it?
As soon as possible. Sometimes midnight. Sometimes during dinner. On the weekend. You could get called back from holiday.

A practical take...

Testing isn't just about the quality of your code. It's also about improving the quality of your life.

Ideally we'd all like to keep a good balance between our work and our life.

Testing helps us do that!

Recap: Black box vs white box testing

Black box testing does not have any knowledge about the underlying implementation of an application.

It's main goal is to test the behaviour of the software. It is focused on the end-user perspective. It generally tests the output of a piece of software.

White box testing tests with knowledge of the program's structure.

It's main goal is to test the code structure, conditions, paths, and so on.

Black Box Testing

Q

How do we black box test a browser-based application?

A

Black box testing generally operates on the compiled output of an application.

In our case, that's the DOM! So, to black box test our code, we should test directly against the DOM.

Recap: Integration Testing

Integration testing is a method of testing the boundaries between different components of our application.

Instead of testing individual units of code (like classes, functions) with unit tests, we test how they interact.

Integration tests are not as cheap as unit tests. They are slower to run, take more time to write, and can be prone to breaking.

However, their expensiveness is offset by how effective they are. Integration tests cover more code and are far more effective at detecting breakages (so long as we do them correctly!)

Recap:
UI Testing

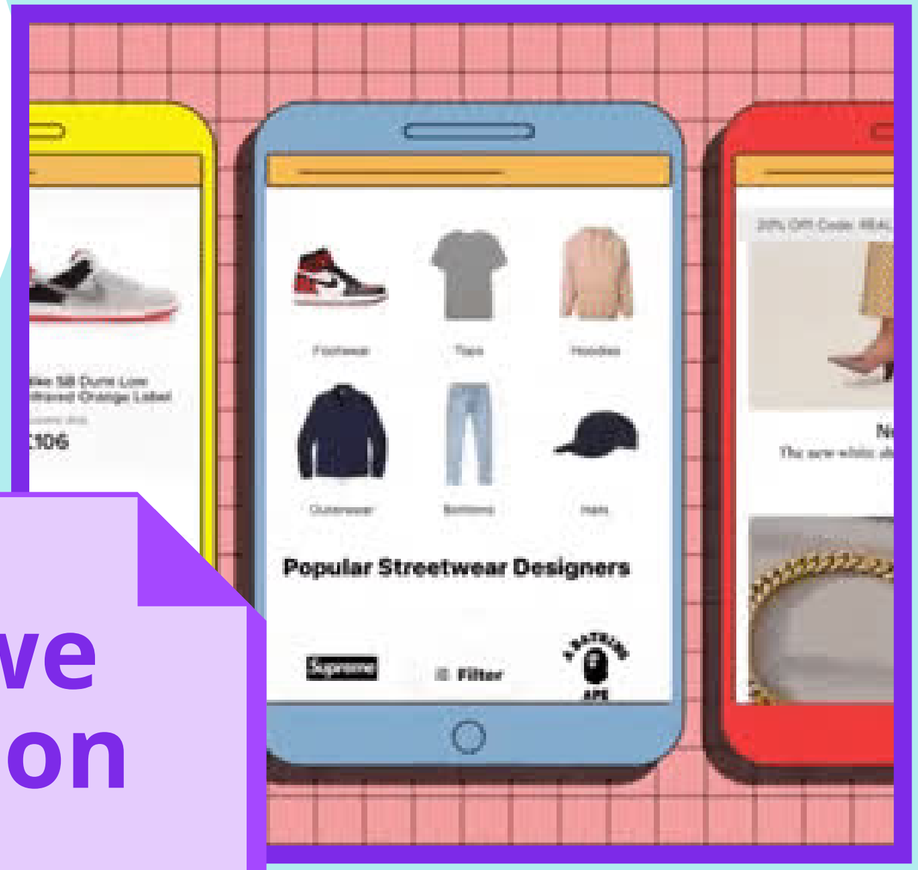
UI Testing is a component of integration testing.

They are black box tests that test against the DOM output of our application.

UI testing allows us to simulate how a user might interact with our web page. We can simulate click events, typing, and can react to changes how a user might.

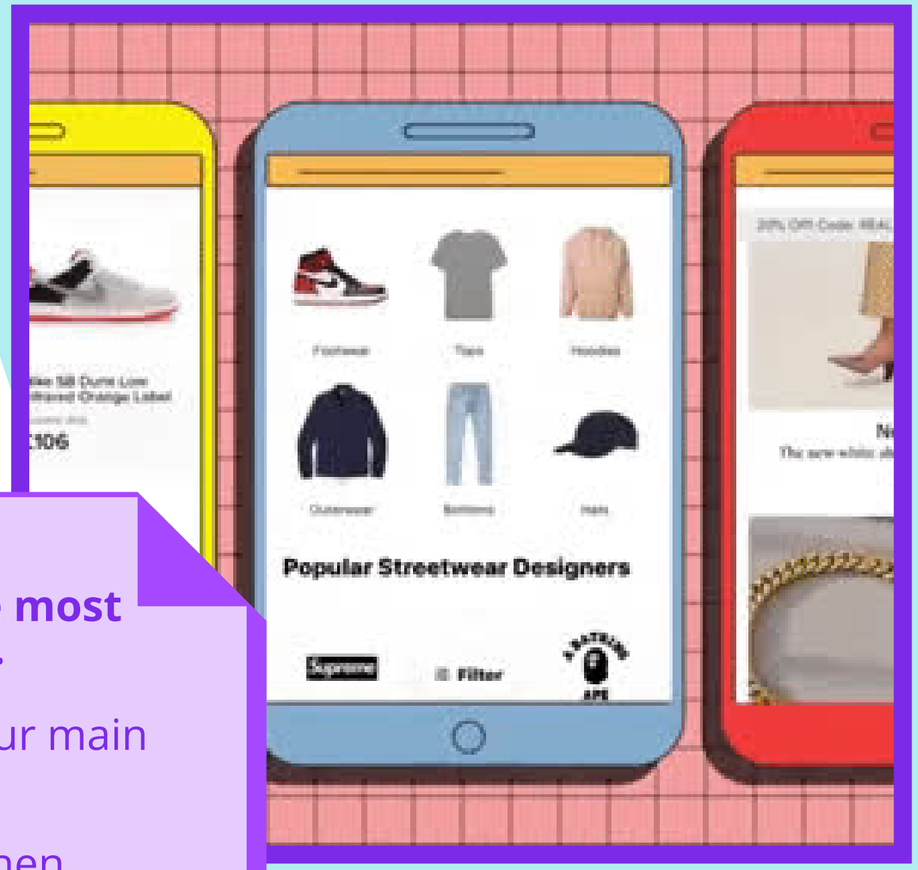
UI tests are important because **they cover what a user is most likely to notice.**

Q



When should we write integration / UI tests?

A



Start with the test that covers the most common path that users traverse.

For example, the "happy path" of your main signup flow.

This means testing the UI flow for when signup completes successfully.

Happy paths and core user flows

In software engineering, our most valuable resource is often our time.

Integration tests are expensive. So it would be a poor use of our time to write integration tests that cover UI flows that our users barely go down.

Happy paths and core user flows

A "**happy path**" is a path traversed by users where everything goes right.

A "**core flow**" is the main way your users would interact with your application. For example, Google's core flow is to perform a search.

If the happy path does not work on your core flow, your application is basically broken.

In a workplace environment that would mean a severity 0 incident which you'd be asked to do overtime to fix.

Automating UI Tests

Q

How do I write an automated UI test?

A

We can simulate the browser! In our previous lectures, we spoke about fake DOMs and using the browser.

In this lecture, we'll be testing against real browsers and real DOMs.

Browser Automation

All mainstream browsers can be automated. It is possible to simulate any user event you can think of using code.

Selenium is the project responsible for automating the browser. At the core of Selenium is a specification known as **Webdriver.**

Webdriver is an interface first implemented by Selenium that allows browsers to provide a common set of APIs used by programming languages for automation.

Webdriver

While Webdriver was formerly a Selenium-specific implementation, it has since been adopted by the W3:

https://www.w3.org/TR/webdriver/

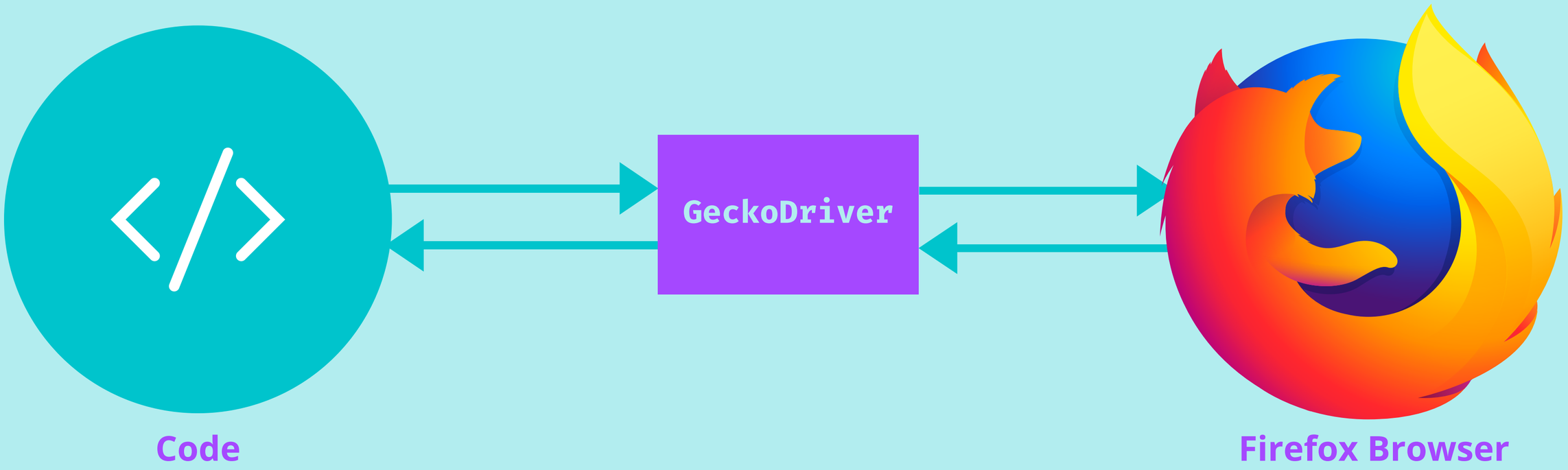
as a standard. So you can reasonably expect that all browsers that are standards-compliant will provide an interface for you to automate.

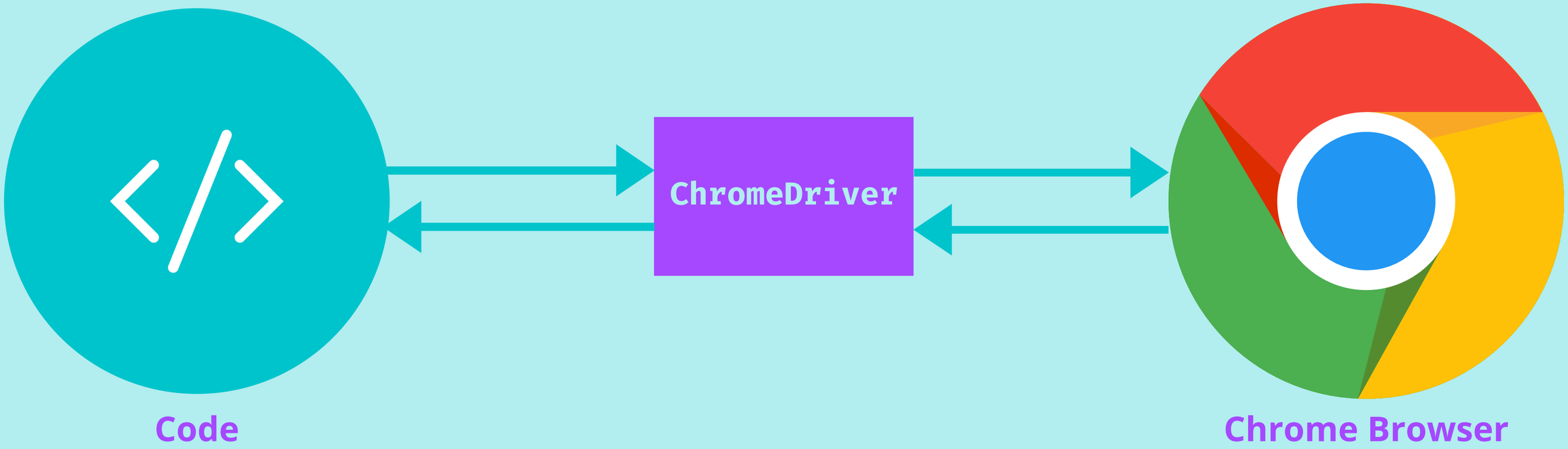
Webdriver

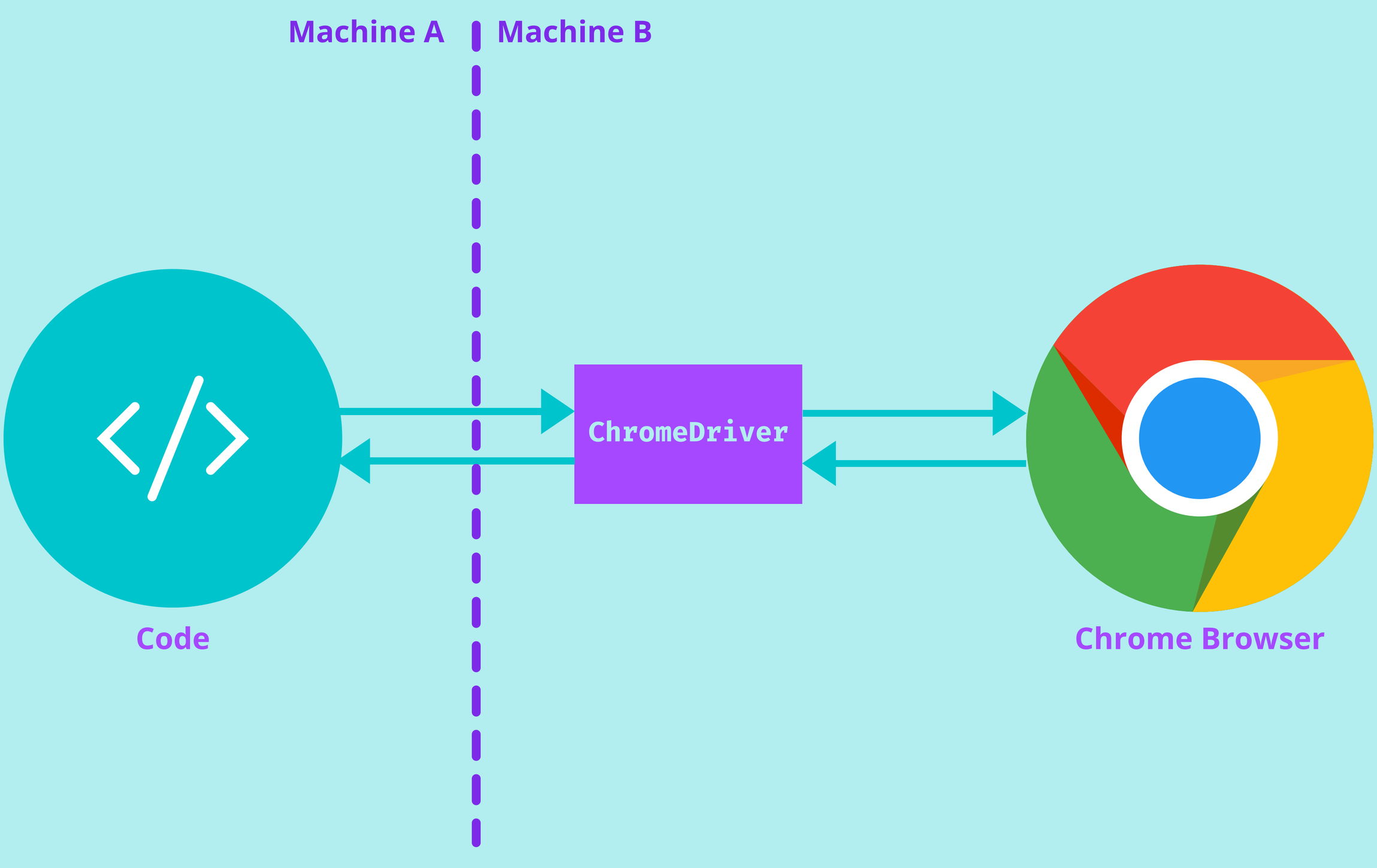
At the core level, **WebDriver talks to a browser through a driver.** The driver sits between the browser and your code and acts like a proxy between the two. The browser can send information back to WebDriver through the same driver.

Drivers are browser specific. Chromedriver is used for Chrome, Geckodriver for Firefox, and so on.

The driver must always run on the same machine as the browser it is speaking to. You can run your code on the same machine, or you can run test against the machine remotely.


GeckoDriver
Code
Firefox Browser


ChromeDriver
Code
Chrome Browser


Machine A
Machine B
ChromeDriver
Code
Chrome Browser

Testing w/ Webdriver

Webdriver does not know anything about testing. It only cares about remote browser automation.

However, testing frameworks can run and execute Webdriver code.

To perform UI tests, a common architecture is for the testing framework to wrap Webdriver code, allowing you to call to it to perform actions and verify the results against the browser.

WebDriver Example: Javascript

```
const {Builder, By, Key, until} = require('selenium-webdriver');

(async function example() {
    let driver = await new Builder().forBrowser('firefox').build();
    try {
        // Navigate to Url
        await driver.get('https://www.google.com');
        // Enter text "cheese" and perform keyboard action "Enter"
        await driver.findElement(By.name('q')).sendKeys('cheese',
Key.ENTER);
        let firstResult = await
            driver.wait(until.elementLocated(By.css('h3>div')), 10000);
        console.log(await firstResult.getAttribute('textContent'));
    }
    finally {
        driver.quit();
    }
})();
```

WebDriver Example: Python

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support.expected_conditions import
presence_of_element_located

#This example requires Selenium WebDriver 3.13 or newer
with webdriver.Firefox() as driver:
    wait = WebDriverWait(driver, 10)
    driver.get("https://google.com/ncr")
    driver.find_element(By.NAME, "q").send_keys("cheese" +
Keys.RETURN)
    first_result =
wait.until(presence_of_element_located((By.CSS_SELECTOR, "h3>div")))
    print(first_result.get_attribute("textContent"))
```

WebDriver Example: Javascript

```
const {Builder, By, Key, until} = require('selenium-webdriver');

(async function example() {
    let driver = await new Builder().forBrowser('firefox').build();
    try {
        // Navigate to Url
        await driver.get('https://www.google.com');
        // Enter text "cheese" and perform keyboard action "Enter"
        await driver.findElement(By.name('q')).sendKeys('cheese',
Key.ENTER);
        let firstResult = await
            driver.wait(until.elementLocated(By.css('h3>div')), 10000);
        console.log(await firstResult.getAttribute('textContent'));
    }
    finally {
        driver.quit();
    }
})();
```

WebDriver Example: Javascript

```
const {Builder, By, Key, until} = require('selenium-webdriver');

(async function example() {
    let driver = await new Builder().forBrowser('firefox').build();
    try {
        // Navigate to Url
        await driver.get('https://www.google.com');
        // Enter text "cheese" and perform keyboard action "Enter"
        await driver.findElement(By.name('q')).sendKeys('cheese',
Key.ENTER);
        let firstResult = await
            driver.wait(until.elementLocated(By.css('h3>div')), 10000);
        console.log(await firstResult.getAttribute('textContent'));
    }
    finally {
        driver.quit();
    }
})();
```

Locators

```
const {Builder, By, Key, until} = require('selenium-webdriver');

(async function example() {
    let driver = await new Builder().forBrowser('firefox').build();
    try {
        // Navigate to Url
        await driver.get('https://www.google.com');
        // Enter text "cheese" and perform keyboard action "Enter"
        await driver.findElement(By.name('q')).sendKeys('cheese',
Key.ENTER);
        let firstResult = await
            driver.wait(until.elementLocated(By.css('h3>div')), 10000);
        console.log(await firstResult.getAttribute('textContent'));
    }
    finally {
        driver.quit();
    }
})();
```

Locators

Locators allow us to navigate and traverse through the DOM.

Remember that the DOM is the model we test against, so we need a way to perform verifications and interact with the DOM. Locators are our toolkit for this purpose.

A successful locator will return an element that can be interacted with.

There are many different locators available to us.

Locator Types

By CSS ID:
method name: find_element_by_id

By CSS Class Name:
method_name: find_element_by_class_name

By 'name' attribute:
method_name: find_element_by_name

By HTML tag name:
method_name: find_element_by_tag_name

Locator Types

By link text:
method_name: find_element_by_link_text

By partial link text:
method_name: find_element_by_partial_link_text

By xpath:
method_name: find_element_by_xpath

Locator example

```
const {Builder, By, Key, until} = require('selenium-webdriver');

(async function example() {
    let driver = await new Builder().forBrowser('firefox').build();
    try {
        // Navigate to Url
        await driver.get('https://www.google.com');
        // Enter text "cheese" and perform keyboard action "Enter"
        await driver.findElement(By.name('q')).sendKeys('cheese',
Key.ENTER);
        let firstResult = await
            driver.wait(until.ele    ated(By.css('h3>div')), 10000);
        console.log(await firstR    getAttribute('textContent'));
    }
    finally {
        driver.quit();
    }
})();
```

Xpath

If you can't find an element using its CSS ID, CSS classnames, names or so on, we can use Xpath.

XPath is a powerful syntax that allows us to define a path through an XML file. HTML has enough in common with XML that we can use it.

```
//form[@id='loginForm']/input[1]
```

Find a form element with an id attribute of 'loginForm', then extract the first input element inside the form.

Recap:
Flaky Tests

A test is flaky if it sometimes succeeds and sometimes doesn't.

Tests should be **deterministic**. A deterministic test always produces the same result, provided that the inputs are the same. It is easy to write a deterministic unit test. It is harder to write a deterministic integration test or a deterministic UI test.

Let's examine why.

Backends

Integration and UI testing involves testing our user flows.

In web applications, all our flows will involve communicating with some sort of backend server.

For functions like signup, or even retrieving a list of items, we often call out to a backend.

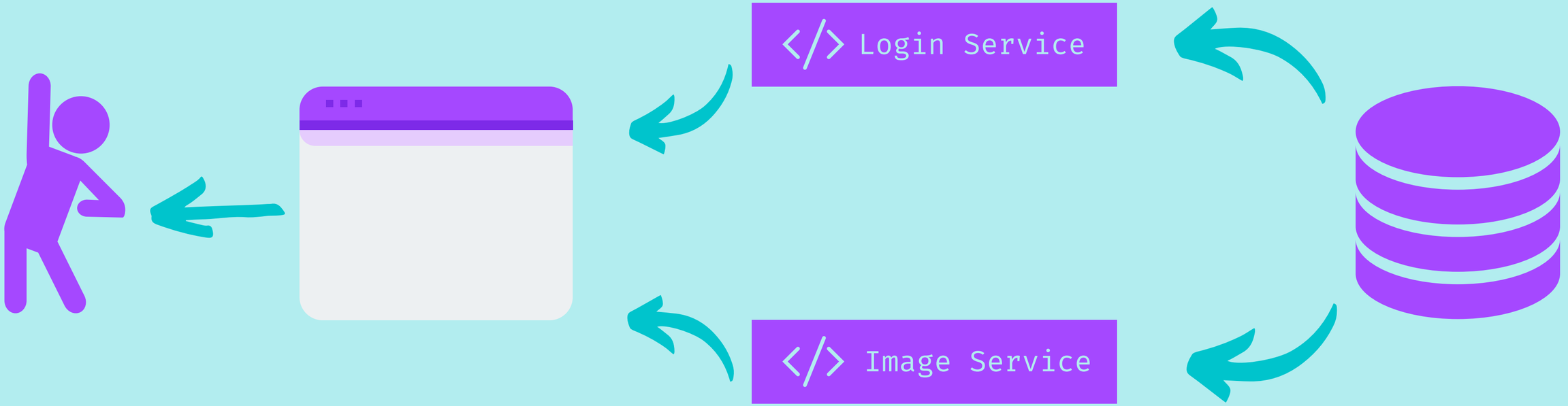
Therefore, **if the backend produces a result we don't expect, our test will flake.**

We can't create a deterministic test if we can't produce reliable inputs.

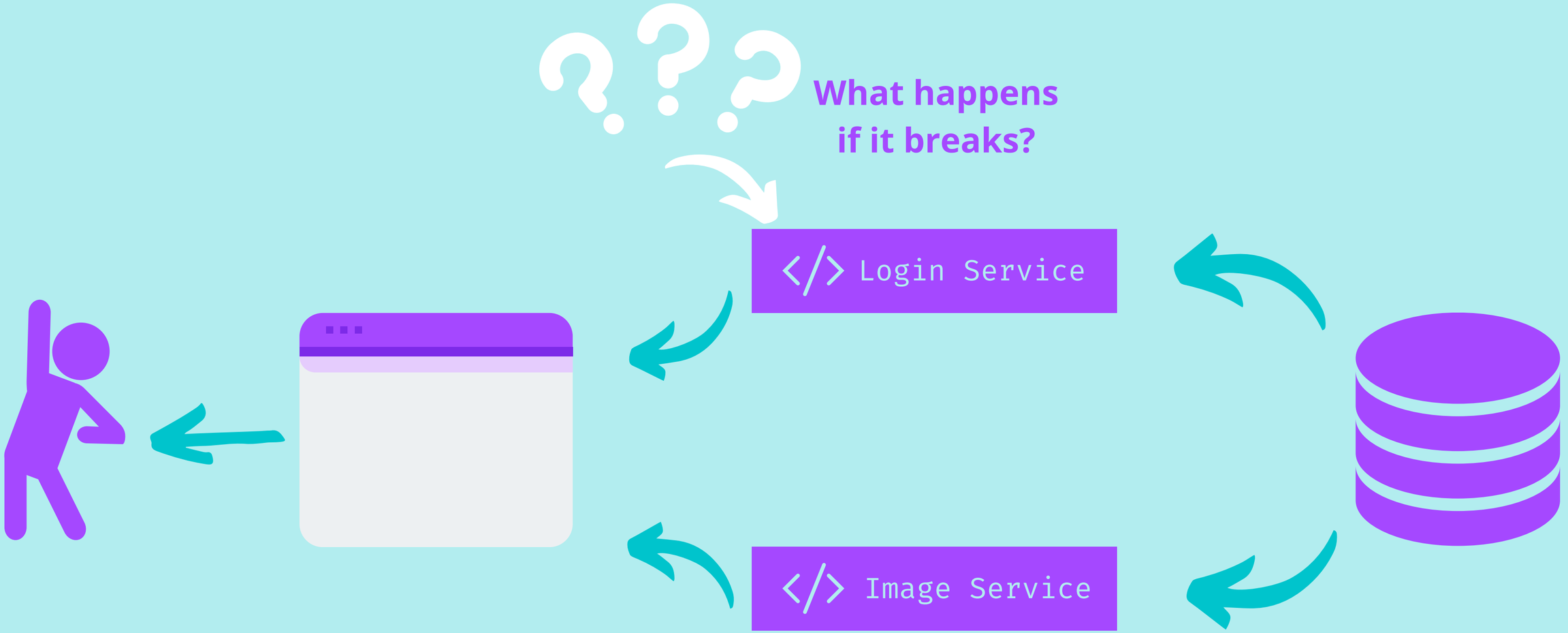
Backends

Also, if a bug occurs on the backend and the test fails, that's not really our page's fault, right?

But our test fails anyway.



Typical Webpage Architecture



Typical Webpage Architecture

Stubs

When doing UI testing, we should isolate our page from the backend so that we don't get unreliable results.

This means that our tests only cover our frontend architecture and don't rely on our backend services at all.

We can do this using **stubs!**

Instead of calling out to our backend services, we can create stub implementations that always produce reliable results.

Stub Services

Stubs are objects that provide fake responses on every endpoint. They implement the interface of an object exactly, such that they can be switched in and out.

```
class LoginService {
   signup() {
      // ...Make remote call to login service
      return {
         result,
         user,
      }
   }
}

class MockLoginService {
   signup() {
      // Don't make a remote call
      return {
         result: 'success',
         user: 'stubUser',
      }
   }
}
```

End to End Tests

Q

But how do I test if the frontend integrates correctly with the backend?

A

If your stubs produce results that conform to what the backend sends, that's a start.

End-to-end testing is a form of testing that integrates both the backend and frontend together. They are very expensive.

Randomness

Your runtime code may rely on some form of random number generation or other dynamic values.

You may think this is unlikely, but consider the ***Date*** module which is time-based. If your application makes calculations based on the date and time, and your test attempts to verify these, you may have created a flaky test.

Examples from real life:

- An integration test that failed consistently at 10am each day.

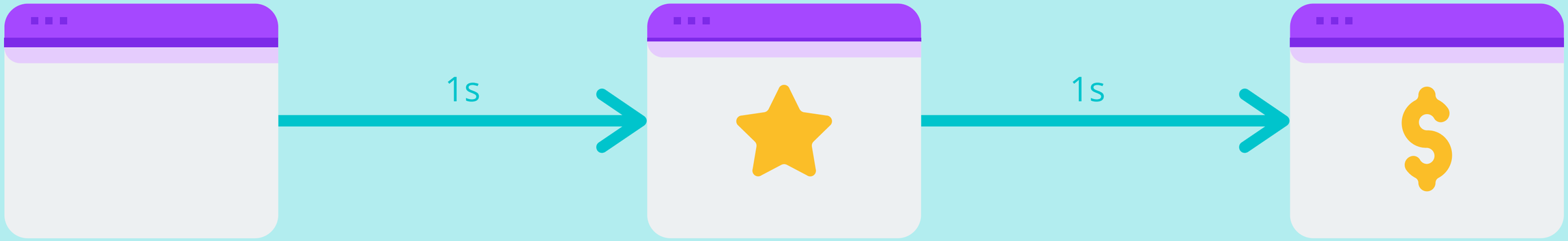
Browser Performance

UI tests are performed in the browser.

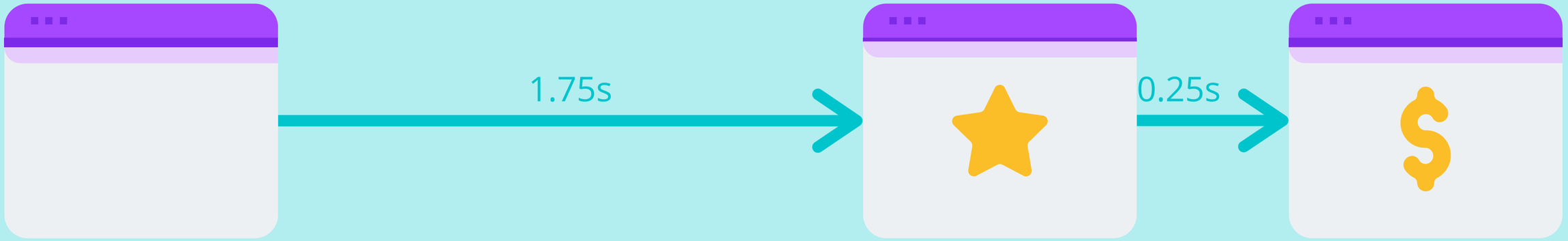
This means that **conditions which affect the browser can affect the reliability of your tests.**

Your browser might be subject to constraints on:

- CPU / processing power
- Memory allocation
- Network bandwidth



CPU, network and memory constraints do not affect our test. For every verification step, the browser will always be in the correct state.



Due to CPU being throttled, the browser took longer to render the star. "Test star is visible" executes the test before the browser has rendered, leading to a failure.

Browser Performance Problems

Q

Why can't we wait until the browser renders the star before verifying?

A

Because we are black box testing.

We do not know the underlying details of our application. We can't just wait for a render to fire and react to it because that implies knowledge of the codebase.

Performance Solutions

It is significantly harder to solve flakiness problems caused by poor browser performance. Poor browser performance can have multiple causes.

There are a couple of levers we can pull to improve this.

Timeouts

Test steps can provide a timeout threshold, after which the test will fail. This provides browsers some leeway in how long it takes to render, giving slow tests time to catch up.

If a test step is failing consistently, you can increase the timeout on the test step to provide more time for rendering to occur.

However, doing so increases the overall time taken for a test to complete. Increasing timeouts can lead to a very slow testing cycle and only kicks the can down the road, but it can be helpful.

Execution Environment

Your test execution environment might not have the resources it needs or be in an inconsistent state.

Examples of execution environment problems include:

- You are running tests on a VM with insufficient resources. E.G 512mb of RAM, leading to memory allocation problems in the browser.
- Your tests rely on environment specific configurations ie environment variables which are not set consistently for each test.

Drivers

Sometimes, flakiness can be caused by a bug in the Driver that sits between your code and the browser.

Looking at you Safari...

Retry

To solve any of these problems, let's not discount the simple solution of re-running the test to see if it succeeds.

Sometimes the "dumbest" solution is the one that works the best!

Summary

So far we've spoken about:

- UI Tests and black box testing
- Webdriver, Selenium and Locators
- Problems with integration tests ie unreliability and how to solve flakiness.
- Backend stubbing, test execution environments.

Examples

</>

Today, we'll be using a framework to write some automated UI tests for a sample application.

Cypress

</>

The framework we will use is called Cypress. Cypress is a framework for automated UI testing. It was built to make end to end testing easier, but can be used for automated testing.

Cypress does NOT use Selenium. 😅

Why not use Selenium?

Why spend all that time teaching the architecture of Selenium when the example doesn't use it?

- Selenium is very commonly used, BUT is harder to configure.
- The broad principles we discussed will always apply - you will always use locators, you will always need to write deterministic tests, you will always have an execution environment, whether thats your local machine or remotely.

Cypress

Cypress consists of two components:

- A test runner.
- A dashboard service.

Some features:

- Snapshots for each step that is run in your test.
- No need to add waits or timeouts to tests - automatically handles this for you.
- Automatic screenshots and videos.
- Easy setup.

Cypress

</>

Things like timeouts, and so on need to be configured manually in Selenium, but Cypress handles all of this for us!

Some caveats:

- Cypress uses **mocha** and **chai** to handle test assertions. These tools fill broadly the same role as Jest.

Getting started with Cypress

Today we'll be writing some tests in Cypress. In the interests of time, this lecture will use a pre-written sample project. However, you can install Cypress on your own project like so:

```
yarn add -D cypress
```

We can then run cypress to open the test runner. The first time you run the command, it will configure cypress in your project and create a **cypress/** folder at the root level of your project.

```
yarn run cypress open
```

Example Cypress Code

cypress/integrations/actions.spec.js

Get an object with CSS classname 'action-focus', focus it, then check if it has the focus class and if the immediately preceding sibling of the element has an inline style of 'color: orange'.

```
it('.focus() - focus on a DOM element',
  () => {
    // https://on.cypress.io/focus
    cy.get('.action-focus')
      .focus()
      .should('have.class', 'focus')
      .prev()
      .should('have.attr', 'style',
              'color: orange;');
  }
);
```

Demo

BDD

One final topic - BDD and Cucumber.

BDD (Behaviour Driven Development) is a method of development that is meant to encourage developers, QA and business people to formalise a shared understanding of an application's behaviour.

It has its own language, which is often used as an interface to write automated tests.

```
Scenario: Items returned for refund should be added to inventory.

Given that a customer previously bought a black sweater from me
and I have three black sweaters in inventory,
when they return the black sweater for a refund,
then I should have four black sweaters in inventory.
```

Well Done!