Testing web applications using Applitools Eyes

Source-code for this tutorial

- code is available in GitHub

Overview

Playwright is a recent browser automation tool that provides an alternative to Selenium.

Applitools Eyes is a visual AI test automation tool that have an SDK available that you can add to your test project allowing visual validations.

Prerequisites

For this example we will use Playwright Test Runner and Applitools Eyes SDK.

We will need:

- Access to a demo site that we aim to test
- Node.js environment with Playwright and Playwright Test Runner
- Applitools Eyes SDK

To start using the Playwright Test Runner please follow the Get Started documentation.

The tests consist in validating 3 features of the demo site: Home link, Find owners functionality and veterinarians link.

We want to add visual validations to these tests, so we have included the Applitools Eyes SDK to be able to use the comparison abilities of the tool.

Before coding the tests start by registering in the Applitools Eyes site and obtain an API-KEY (that is what we will use in the test execution to ship screenshots to the tool for comparison), more information on how to do it here.

We started by defining *PageObjects* that will represent the pages we will interact with, we have defined three, as we see below:

./models/owners.js

```javascript
const config = require ("../config/config.json");

class OwnersPage {

    constructor(page) {
      this.page = page;
    }

    async navigate() {
      await this.page.goto(config.endpoint);
    }

    async click_find_owners_button(){
        await this.page.click(config.find_owners_button);
    }

  }
  module.exports = { OwnersPage };
```

./models/home.js

```javascript
const config = require ("../config/config.json");

class HomePage {

    constructor(page) {
      this.page = page;
    }

    async navigate() {
      await this.page.goto(config.endpoint);
    }

    async getMenuEntry(){
        return await this.page.locator(config.top_menu_entry).first();
    }

    async getHomeText(){
        return config.home_text;
    }

  }
  module.exports = { HomePage };
```

./models/veterinarians.js

```
const config = require ("../config/config.json");

class VetsPage {

    constructor(page) {
      this.page = page;
    }

    async navigate() {
      await this.page.goto(config.endpoint);
    }

    async getTopMenuEntry(){
        return this.page.locator(config.vet_menu_entry).first();
    }

    async getVetsText(){
        return config.vet_text;
    }

  }
  module.exports = { VetsPage };
```

Plus a configuration file where we have the identifiers that will match the elements in the page, this will add an extra abstraction layer to the tests allowing us to redefine locators or text without changing the code.

config.json

```
{
    "endpoint" : "https://xray-essentials-petclinic.herokuapp.com/",
    "owners_link" : "a[title=\"find owners\"]",
    "top_menu_entry" :"//*[@id=\"main-navbar\"]/ul/li[1]/a",
    "vet_menu_entry" : "//*[@id=\"main-navbar\"]/ul/li[3]/a",
    "find_owners_button" : "a[title=\"find owners\"]",
    "vet_text" : "Veterinarians",
    "home_text" : "Home"
}
```

We added an helper file that will parse returned information and add valuable information returned by Applitools Eyes to the Junit report.

helper.js

```javascript
class Helper {

  constructor() {

  }

  handleTestResults(summary){
    let ex = summary.getException();
    if (ex != null ) {
        console.log("System error occurred while checking target.\n");
    }
    let result = summary.getTestResults();
    if (result == null) {
        console.log("No test results information available\n");
    } else {
        console.log("[Eyes URL|%s] \\\\ AppName = %s \\\\ testname = %s
\\\\ status = %s \\\\ different = %s \\\\ Browser = %s \\\\ OS = %s \\\\
viewport = %dx%d \\\\ matched = %d \\\\ mismatched = %d \\\\ missing = %
d\\\\ aborted = %s\\\\",
            result.getUrl(),
            result.getAppName(),
            result.getName(),
            result.getStatus(),
            result.getIsDifferent(),
            result.getHostApp(),
            result.getHostOS(),
            result.getHostDisplaySize().getWidth(),
            result.getHostDisplaySize().getHeight(),
            result.getMatches(),
            result.getMismatches(),
            result.getMissing(),
            (result.getIsAborted() ? "aborted" : "no"));
            let steps = result.getStepsInfo();
            steps.forEach(step => {
              console.log("StepName = %s, different = %s\\\\", step.
getName(), step.getIsDifferent());
            });
        }

  };

  }
  module.exports = { Helper };
```

The tests that validate if the features are behaving as expected are below, notice that we are using the Applitools Eyes SDK and adding checks on the tests in the moments we want to have visual validations.

For the tutorial purpose we will focus in the *Owners* validations (the others will be similar with more or less actions).

login.spec.ts

```
import { test, expect } from '@playwright/test';
import { OwnersPage } from "../models/owners";
import { Helper } from "../models/helper"
const { Eyes, ClassicRunner, Target , Configuration, BatchInfo,
MatchLevel, TestResultContainer, TestResults} = require('@applitools/eyes-
playwright')

test.describe("PetClinic validations", () => {
  let eyes, runner;//, default_url;

  test.beforeEach(async () => {

    // Initialize the Runner for your test.
    runner = new ClassicRunner();

    // Create Eyes object with the runner
    eyes = new Eyes(runner);

    // Initialize the eyes configuration
    const configuration = new Configuration();

    // create a new batch info instance and set it to the configuration
    configuration.setBatch(new BatchInfo('PetClinic Batch - Playwright -
Classic'));

    // Define the match level we need for our tests
    eyes.setMatchLevel(MatchLevel.Strict);

    // Set the configuration to eyes
    eyes.setConfiguration(configuration);

  });

  test('Validate find owners link', async ({ page }) => {
    const ownersPage = new OwnersPage(page);
    await ownersPage.navigate();
    await eyes.open(page, 'PetClinic', 'FindOwnersLink', { width: 800,
height: 600 });
    await ownersPage.click_find_owners_button();
    await eyes.check(Target.window().fully());
    await eyes.close();
  });

  test.afterEach(async () => {
    const helper = new Helper();
    // If the test was aborted before eyes.close was called, ends the test
as aborted.
    await eyes.abort();

    // We pass false to this method to suppress the exception that is
thrown if we
    // find visual differences
    const results = await runner.getAllTestResults(false);

    results.getAllResults().forEach(result  => {
      helper.handleTestResults(result);
    });

  });
})
```

Looking to this class in more details we can see different areas:

- *test.beforeEach()*
- *test()*

- *test.afterEach()*

In the "*test.beforeEach*" we are configuring the runner that will be used by the Eyes instance, as you can see, we are using the classic one (Eyes have another available called "*Visual Grid Runner*" that interacts with the Eyes Ultrafast Grid server to render the checkpoint images in the cloud).

We defined a configuration object that will hold the configuration for the instance, we are defining the Batch named 'PetClinic Batch - Playwright - Classic', defining the match level (in our case we are using the recommended one *Strict* but there are more available).

In the test itself we have a normal Playwright test with additions from the Applitools Eyes SDK, let's look into those in more detail:

- *await eyes.open(page, 'PetClinic', 'FindOwnersLink', { width: 800, height: 600 });* - to start a test, before calling any of the check methods and we are defining the AppName, TestName and ViewPortSize.
- *await eyes.check(Target.window().fully());* - Run a checkpoint. Uses Fluent arguments to specify the various parameters.
- *await eyes.close();* - Call this method at the end of the test. This terminates the sequence of checkpoints, and then waits synchronously for the test results and returns them.

Finally in the "*test.afterEach*" we make sure to close all eyes instances by calling the "abort" method and process the results that are returned by the runner.

Once the code is implemented, we will run it to define the baseline (A baseline stores a sequence of reference images), that will be used to compare to the next tests. We achieve that with the following command:

```
APPLITOOLS_API_KEY="API_KEY" PLAYWRIGHT_JUNIT_OUTPUT_NAME=results.xml npx
playwright test ./tests/* --browser=chromium --reporter=junit,line
```
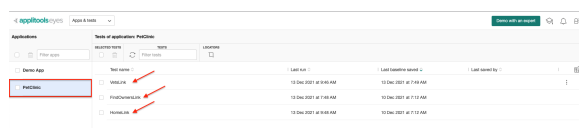
> ⚠️ If the APPLITOOLS_API_KEY is not defined the tests will be executed but the screenshots will not be sent to Applitools Eyes.

The output generated shows how many tests have been executed, produces a Junit report and returns the link to check the visual assertions.
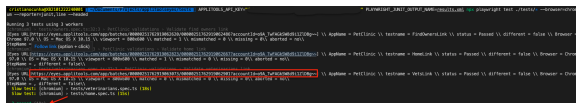
In Applitools Eyes interface we can see that a new application was created with 3 tests:


If we navigate to the test results we will see the three tests properly named, information about the OS, Browser and Viewport used, a screenshot taken and the notion if is new or not and a date.
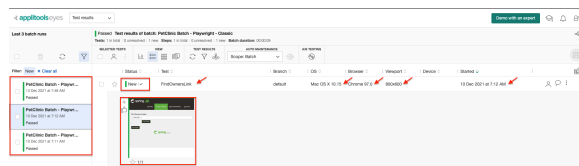
At this point we have generated our baseline and the tests are behaving as expected, we will now introduce a change in the application and remove strings from the Owners test that will make the test to succeed but the visual validation will fail as it will not match the baseline and thus failing the tests overall.

After the second execution the output terminal will have the following information:
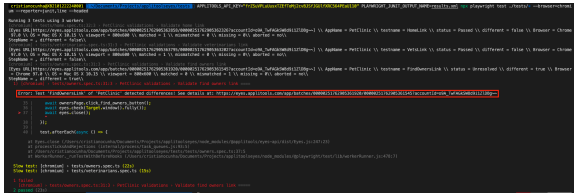
The report generated will contain the following information:

Junit Report

```xml
<testsuites id="" name="" tests="3" failures="1" skipped="0" errors="0"
time="23.226">
<testsuite name="tests/home.spec.ts" timestamp="1639395430782" hostname=""
tests="1" failures="0" skipped="0" time="14.955" errors="0">
<testcase name="PetClinic validations Validate home link" classname="
[chromium] > tests/home.spec.ts:32:3 > PetClinic validations > Validate
home link" time="14.955">
<system-out>
[Eyes URL|https://eyes.applitools.com/app/batches/00000251762905362858
/00000251762905362326?accountId=o9A_TwFAGkSW8d9i1ZlDBg~~] \\ AppName =
PetClinic \\ testname = HomeLink \\ status = Passed \\ different = false
\\ Browser = Chrome 97.0 \\ OS = Mac OS X 10.15 \\ viewport = 800x600 \\
matched = 1 \\ mismatched = 0 \\ missing = 0\\ aborted = no\\
StepName = , different = false\\

</system-out>
</testcase>
</testsuite>
<testsuite name="tests/owners.spec.ts" timestamp="1639395430782"
hostname="" tests="1" failures="1" skipped="0" time="21.788" errors="0">
<testcase name="PetClinic validations Validate find owners link"
classname="[chromium] > tests/owners.spec.ts:31:3 > PetClinic validations
> Validate find owners link" time="21.788">
<failure message="owners.spec.ts:31:3 Validate find owners link" type="
FAILURE">
  [chromium] > tests/owners.spec.ts:31:3 > PetClinic validations >
Validate find owners link =======

    Error: Test 'FindOwnersLink' of 'PetClinic' detected differences! See
details at: https://eyes.applitools.com/app/batches/00000251762905361920
/00000251762905361545?accountId=o9A_TwFAGkSW8d9i1ZlDBg~~

      35 |      await ownersPage.click_find_owners_button();
      36 |      await eyes.check(Target.window().fully());
    > 37 |      await eyes.close();
         |        ^
      38 |    });
      39 |
      40 |    test.afterEach(async () => {

        at Eyes.close (/Users/cristianocunha/Documents/Projects
/applitoolseyes/node_modules/@applitools/eyes-api/dist/Eyes.js:247:23)
        at processTicksAndRejections (internal/process/task_queues.js:93:5)
        at /Users/cristianocunha/Documents/Projects/applitoolseyes/tests
/tests/owners.spec.ts:37:5
        at WorkerRunner._runTestWithBeforeHooks (/Users/cristianocunha
/Documents/Projects/applitoolseyes/node_modules/@playwright/test/lib
/workerRunner.js:478:7)

</failure>
<system-out>
[Eyes URL|https://eyes.applitools.com/app/batches/00000251762905361920
/00000251762905361545?accountId=o9A_TwFAGkSW8d9i1ZlDBg~~] \\ AppName =
PetClinic \\ testname = FindOwnersLink \\ status = Unresolved \\ different
= true \\ Browser = Chrome 97.0 \\ OS = Mac OS X 10.15 \\ viewport =
800x600 \\ matched = 0 \\ mismatched = 1 \\ missing = 0\\ aborted = no\\
StepName = , different = true\\
```

```
</system-out>
</testcase>
</testsuite>
<testsuite name="tests/veterinarians.spec.ts" timestamp="1639395430782"
hostname="" tests="1" failures="0" skipped="0" time="15.489" errors="0">
<testcase name="PetClinic validations Validate veterinarians link"
classname="[chromium] > tests/veterinarians.spec.ts:31:3 > PetClinic
validations > Validate veterinarians link" time="15.489">
<system-out>
[Eyes URL|https://eyes.applitools.com/app/batches/00000251762905363795
/0000025176290536302?accountId=o9A_TwFAGkSW8d9i1ZlDBg~~] \\ AppName =
PetClinic \\ testname = VetsLink \\ status = Passed \\ different = false
\\ Browser = Chrome 97.0 \\ OS = Mac OS X 10.15 \\ viewport = 800x600 \\
matched = 1 \\ mismatched = 0 \\ missing = 0\\ aborted = no\\
StepName = , different = false\\

</system-out>
</testcase>
</testsuite>
</testsuites>
```

When we access the link provided by Applitools Eyes we can see the visual changes detected:
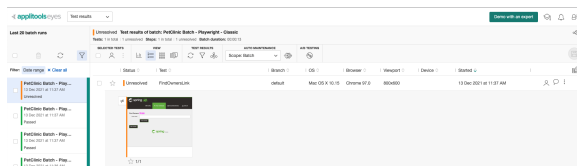
When accessing the details we can see the actual differences detected between the baseline and the latest test side by side:
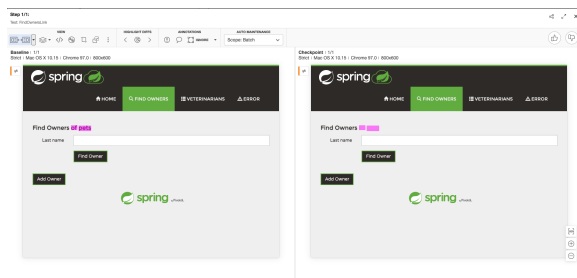
Notes:

- Applitools will let you analyse further the problem by filtering the view in different layers.
- Applitools have available the possibility to define regions, regions to be ignore for example if the content is too dynamic.
- Applitools let you add annotations, such as remarks or setting a bug.
- By default Playwright will execute tests for the 3 browser types available (that is why we are forcing to execute only for one browser)
- By default all the tests will be executed in headless mode
- Folio command line will search and execute all tests in the format: *"**/?(*.)+(spec|test).[jt]s"*
- In order to get the Junit test report please follow this section

Integrating with Xray

As we saw in the above example, where we are producing Junit reports with the result of the tests, it is now a matter of importing those results to your Jira instance, this can be done by simply submitting automation results to Xray through the REST API, by using one of the available CI/CD plugins (e.g. for Jenkins) or using the Jira interface to do so.

API

API

Once you have the report file available you can upload it to Xray through a request to the REST API endpoint for JUnit, and for that the first step is to follow the instructions in v1 or v2 (depending on your usage) to obtain the token we will be using in the subsequent requests.

JUnit XML results

We will use the API request with the definition of some common fields on the Test Execution, such as the target project, project version, etc.

In the first version of the API, the authentication used a login and password (not the token that is used in Cloud).

```
curl -H "Content-Type: multipart/form-data" -u admin:admin -F
"file=@results.xml" 'http://<LOCAL_JIRA_INSTANCE>/rest/raven/1.0/import
/execution/junit?projectKey=COM&testPlanKey=COM-81'
```

With this command we are creating a new Test Execution in the referred Test Plan with a generic summary and two tests with a summary based on the test name.

Looking closer to the failed test, if we click on it to check details we will see that in the comment section we have the details returned by the Applitools call, in there we find the link that will open the Applitools application with the details of the visual validations. The definition is automatically filled up with an auto-generated identifier that uniquely identifies the test and in the output section we have the stack trace returned when we executed the test.

Jira UI

Jira UI

1

Create a Test Execution for the test that you have

2

Fill in the necessary fields and press "*Create*"

3

Open the Test Execution and import the JUnit report

4 Choose the results file and press *"Import"*

5 The Test Execution is now updated with the test results imported

Tests implemented using Playwright will have a corresponding Test entity in Xray. Once results are uploaded, Test issues corresponding to the Jest tests are auto-provisioned, unless they already exist.

Xray uses a concatenation of the suite name and the test name as the the unique identifier for the test.

In Xray, results are stored in a Test Execution, usually a new one. The Test Execution contains a Test Run per each test that was executed using playwright-test runner.

Detailed results, including logs and exceptions reported during execution of the test, can be seen on the execution screen details of each Test Run, accessible through the *Execution details*:
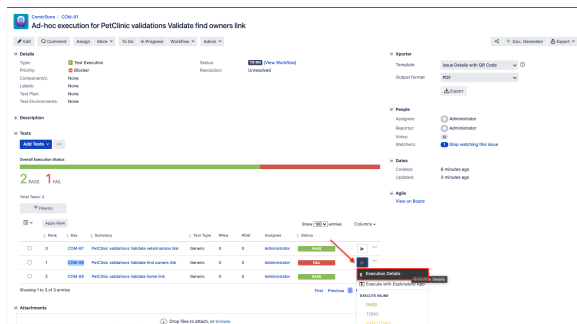
Looking to the details of the failed test, we see that in the comment section we have the details returned by the Applitools call. We can find the link that will open the Applitools application with the details of the visual validations. The definition is automatically filled up with an auto-generated identifier that uniquely identifies the test and in the output section we have the stack trace returned when we executed the test.
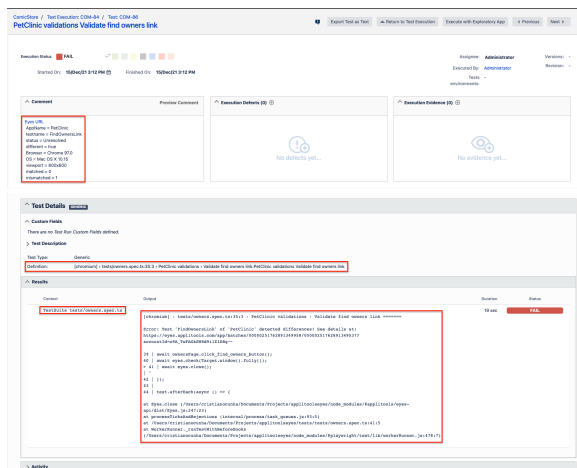
Tips

- after results are imported, in Jira Tests can be linked to existing requirements/user stories, so you can track the impacts on their coverage.
- results from multiple builds can be linked to an existing Test Plan, to facilitate the analysis of test result trends across builds.
- results can be associated with a Test Environment, in case you want to analyze coverage and test results by that environment later on. A Test Environment can be a testing stage (e.g. dev, staging, prepod, prod) or a identifier of the device/application used to interact with the system (e. g. browser, mobile OS).

References

- https://github.com/microsoft/playwright-test/blob/master/README.md
- https://playwright.dev/
- https://playwright.tech/blog/using-jest-with-playwright
- https://applitools.com/docs/index.html
- https://applitools.com/docs/topics/sdk/sdk.html