

“The Future of Testing:

Visual Testing and Its Influence"

Document Version / Details: Ver. 1.0/ 14-Sept-2023

**White Paper**

**Table of Contents**

Visual Testing overview 2

1.1 Trigger for Visual Testing 2

1.2 Visual Testing Tools evaluation criteria 3

1.3 Visual Testing Workflow 4

1.4 Visual testing with Cypress 6

1.5 Visual testing with WDIO 8

1.6 Visual testing with Selenium 21

1.7 Visual Testing Tools Comparison 26

1.8 Visual Testing Benefits, Challenges, and learnings 27

1.9 Visual Testing Best Practices 28

1.10 Visual Testing References 28

**1. Visual Testing overview**

Visual testing is an automated process that enables user interface (UI) testing of a web/mobile application to ensure that the software/application is represented to the end-users in the exact visual way it is intended to. This approach supports QA (Quality Assurance) with the objective of identifying visual errors that can negatively impact the application's usability and end user experience.

But why do we need Visual Testing? When running a functional test, QA validates that the behavior (output) corresponds to the action by executing a functional test (input). Typically, this works out effectively when fully automated. The UI's visual aspects are not being tested; therefore, it does not test whether an element is shifted far off the screen or is blocked by another element on the page. functional test would not pick this up.

So how does one proceed with the approach? It works by comparing snapshots of your UI against baselines to see if pixels have changed. By looking at the pixels rather than the code underneath, visual testing makes it easy to see exactly what your UI looks like before deploying and to catch visual regressions.

## Trigger for Visual Testing

To validate a functional user interface and workflow, test assertions should be present in multiple automated instances. Especially, UI visual appeal test assertions, which vary based on the number of UI elements are present on each page of the application being tested and which make the code base bulky and challenging to maintain as functional enhancements continue to improve. And there is no support in the current automation framework for concurrent cross-platform and cross-browser testing.

A computer screen with a couple of black text

Description automatically generated with medium confidence

(Source: Visual triggers, Internet Image)

To do smoke and regression tests, the testing team must manually check the UI aesthetics and any other features that are not automatable based on daily/weekly releases. Performing visual tests uncovers visual defects more quickly and efficiently, ensuring that they do not make it into production. As a result, visual testing improves end-user experiences, product confidence, and productivity.

## Visual Testing Tool Evaluation Criteria

**Skills:** To build and run test scripts in many automation technologies, especially open-source software, testers need to have a reasonable level of programming expertise.

**Cost:** Does the tool fall under team budget?

**Integration:** How complex is the process of integration with existing platform, scripts, browsers and repositories? The chosen automation solution must be able to integrate with CI/CD pipelines and other platforms.

**Support:** Technical support and upgrades of the tool along with Customer Support. Also includes bug resolution and quick response time.

**Reporting functionality:** Reports of test results in detail, especially those that specify the precise steps where failures take place. Snapshots and screenshots of the steps are also very beneficial. Reports should be able to be exported in a variety of formats so that users can easily view them.

**Script Reusability:** Applications with large number of test cases require scripts to be utilized for multiple executions. Reusability becomes a key factor in saving time during scripts maintenance and migration.

## Visual Testing Workflow

1. **Write the Visual testing scripts/Tests -**

Visual Testing can be integrated with almost all tools and technologies generally used for automation. Below sections will cover the integration topics. Once the integration has been completed, the first step would be to create the tests and scripts to perform visual testing.

2.  **First Run -**

The first run of the scripts will be as per user expectations, and it will capture the ideal state of the visible UI.

3. **Establish baselines -**

The screenshots captured from the first run will establish the ideal state of the application under test and will work as a baseline for consecutive test runs. Scripts will capture screenshots each time they are executed, and we can use the baseline to compare newly captured screenshots with the ideal state of the application.

4. **Execute scripts again to compare -**

Once the UI receives changes, scripts executed will generate a new set of screenshots. Some of them will fall under new categories, these are the screenshots from the UI where changes were made.

5. **Update Baseline - Accept/Reject changes -**

With the new set of screenshots, we can either Accept or Reject the changed ones based on the requirements of the stories. Below are the states explained for the screenshots –

* Workflow –

|  |  |
| --- | --- |
| **UI State** | **Description** |
| New | Has not been reviewed yet. |
| Changed | Has changed visually when compared to its accepted baseline. |
| Accepted | Is visually the same as its baseline OR has been reviewed and accepted by a team member. |
| Rejected | Has been reviewed, found to have defects, and rejected by a team member. |

Diagram

Description automatically generated

## Visual testing with Cypress

1. **Ways to integrate Visual Testing plugin in Cypress:** There are multiple ways to use –

* Using SmartBear with Cypress –

**Requirement:** Cypress version 6.7.0 and above should be installed to avail this plugin.

**Steps to integrate:**

**Step-1:**

* + - Install plugin using the command below –
    - npm install @smartbear/visualtest-cypress@latest.
    - npx visualtest-setup
    - Installing the plugin will create visualTest.config.js file.

**Step-2:**

* + - Import and add the smartbear command.
    - **Add:** require('@smartbear/visualtest-cypress’) (module); at the bottom of cypress.config.js
* **Add:** import '@smartbear/visualtest-cypress/commands' at the bottom of cypress/support/e2e.js

**Step - 3**

* Substitute **"PROJECT TOKEN"** in your newly created visualTest.config.js file with your project token.
* Project Token is issued when you first create the project and can be found in project settings -

**module.exports = { projectToken: 'PROJECT\_TOKEN' }**

**Step - 4**

* Use the capture method at each location during your functional test where you want to capture an image as shown below -
* cy. sbvtCapture ('Home Page'); // Full-page screenshot
* cy. sbvtCapture ('Home Page Viewport', {capture: 'viewport'}); // Viewport screenshot
* cy.get('.menu').sbvtCapture('Menu'); // Element screenshot

**Step - 5**

* Visual Test generates a console log output, which includes a link for your team to approve and review any picture differences discovered in your UI.
* Use Argos with cypress –

Argos utilizes the Cypress command to enhance its functionality and snap screenshots.

**Step- 1**

- Setup import "@argos-ci/cypress/support”;

**Step- 2**

- Import and add argos command

  - Add this line to your cypress/support/index.js file.

- import "@argos-ci/cypress/support";

**Step -3**

- Use cy.argosScreenshot to take screenshot

describe("Homepage", () => {

it("screenshots the page", () => {

// Screenshot a full page

cy.argosScreenshot("home");

// Screenshot a component

cy.get("main div.breadcrumb").argosScreenshot("home-breadcrumb");

});

});

- Screenshots taken will be stored in cypress/screenshots folder by default

**Step - 5**

- Configure your CI to send screenshots to Argos when using the Argos CLI command.

- **Command** : npx @argos-ci/cli upload cypress/screenshots, you must specify the folder where Cypress stores screenshots “cypress/screenshots”’

* Use Percy cypress -

Percy helps organizations with visual testing automation. It takes screenshots, analyzes them with the starting point, and identifies visual changes.

**Step - 1**

**Setup**

* + - npm install --save-dev@percy/cli @percy/cypress.

**Step - 2**

- Import and add percy command

- Import the following package into cypress/support/e2e.js file

* + - import '@percy/cypress’.

**Step-3**

- Use cy.percySnapshot() command to take screenshot -

describe ('Integration test with visual testing', function () {

it ('Loads the homepage', function () {

// Load the page or perform any other interactions with the app.

cy.visit(<URL under test>);

// Take a snapshot for visual diffing

cy. percySnapshot().

});

});

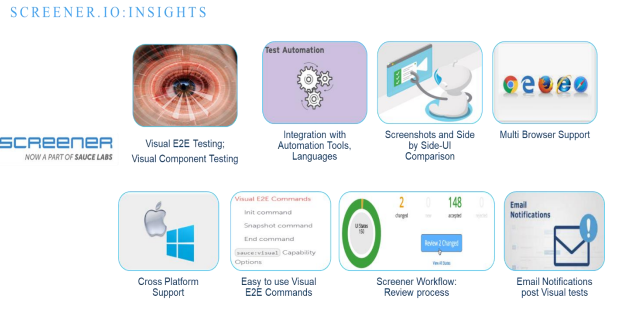
* + - Use the percy exec command to enclose your test runner command. By doing this, a Percy agent will be launched, which will collect screenshots from your Cypress tests and upload rendered screenshots to your Percy dashboard.

## Visual testing with WDIO

**Sauce Visual Overview:**

Sauce Visual a.k.a. Screener.io has been acquired by Sauce Labs in 2019.

Key Features –



**Trigger for Sauce Visual Integration:**

• In test automation, it’s essentials to have test assertions at multiple instances within script to validate UI, functional workflow. Especially, UI aesthetics-based test assertions, which varies as per number of UI elements available on each screen of application under test and which makes code base bulky and difficult to maintain as functional enhancements continue to enhance.

• With existing automation framework, there is no provision to perform cross platform and cross browser tests in parallel.

• Testing team has to spend manual efforts for verification of UI aesthetics or whichever features are not automobile based on daily/weekly builds as part of smoke, regression tests

• UI deviations sometimes are time consuming to resolve exact root cause which lead to increase in UI Failure resolution timeframe.

• Application which needs to upgrade with new software version on which it has been build and not required any functional change. Testing team must ensure that upgrade has not impacted on any UI change.

Sauce Visual helps us to mitigate all above listed challenges in a seamless manner.

**Pre-Requisite:**

Ensure to have access to Screener.io portal and Sauce Labs.

**End to End application navigation script (without assertions)**

Step 1:

- Access Screener.io interface and create a web driver project

- Access Screener.io application (https://screener.io/)

- Click on + icon in the dashboard

- Click on New WebDriver Project and create a respective project in screener.io

**Note:** New Storybook Project is used by developers to do component level development

and testing

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Step 2**: **Sauce Visual-Webdriver IO Framework Integration**

* 1. **:** Create **webdriver.shared.conf.ts file** in the existing Webdriver IO-Cucumber framework to include SauceLab and Sauce Visual configuration.

A screen shot of a computer code

Description automatically generated

* Add Screener configuration.

A computer screen with text on it

Description automatically generated

* Add SauceOptions to define Sauce Lab credentials and execution tunnel details

A computer screen with text

Description automatically generated

* Add Visual Options to define Screener credentials, Web driver project configured in sauce visual.

A computer screen with text

Description automatically generated

* Add details of Sauce Visual based E2E feature file, which needs to be executed/has been created.

A computer screen with green text

Description automatically generated

* Add Configuration specific to cross browser, cross platform tests along with desired resolution and parallel testing configured with number of instances.

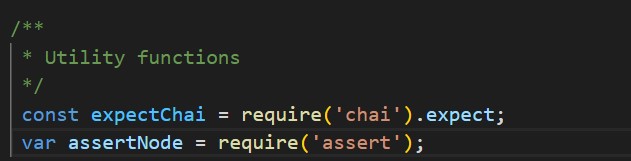
A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

* 1. **:** Utilities-**keywords.ts file**: Incorporate sauce visual commands, listed as below into existing keywords.ts file



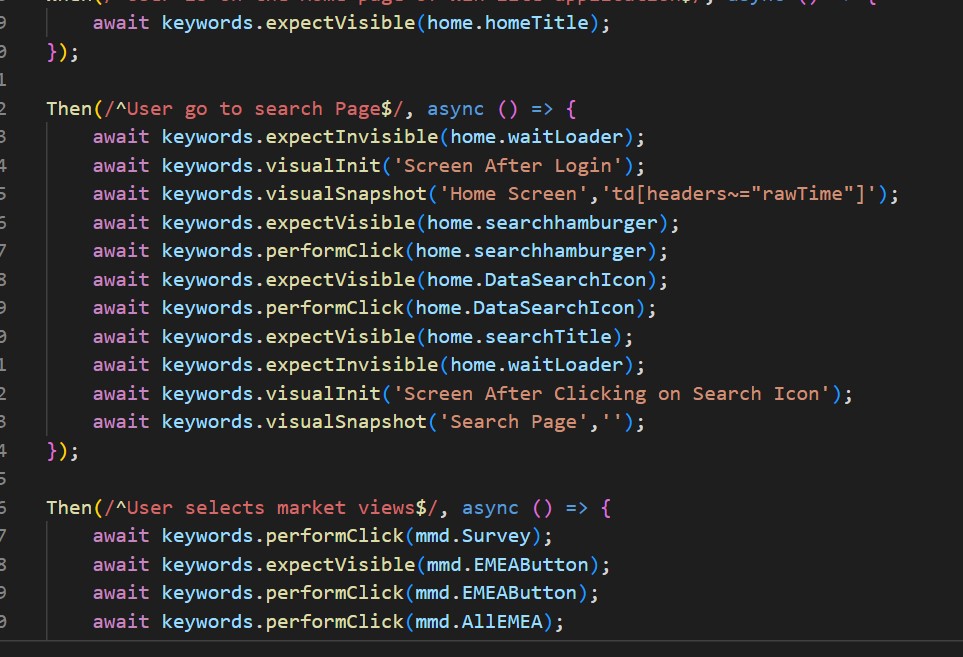
* 1. : Create Feature file and associated step definition file and incorporate visual test methods for each screen.

**Feature File:**

A screen shot of a computer

Description automatically generated

**Step Definition file** (Included Sauce visual command/methods as defined)



* 1. : **Package.json file**:
     1. : Install Sauce Services, mocha framework (optional) and edgedriver service (in case cross browser tests needs to be done using Microsoft edge browser)

A screenshot of a computer program

Description automatically generated

* + 1. : Add test execution command like smoke and regression test as below -

**"screener\_qa":**

"npx wdio wdio.shared.conf.ts --env=qa --sauce\_username=<<sauce labs username here>>--sauce\_access\_key=<<sauce lab access key here>> --

screener\_access\_key=<<screener access key here>>",

A computer code on a black background

Description automatically generated

**Step # 3. Initiate Visual test and tracking test results.**

* + 1. : Initiate Visual test in VS Code console with command: npm run screener\_qa

A screen shot of a computer

Description automatically generated

* + 1. **:** User can track test execution in console/terminal:

A screenshot of a computer program

Description automatically generated

* 1. **Access Screener.io** to track real time tracking of test execution.

A screenshot of a computer

Description automatically generated

* 1. Access View log in Sauce lab link to see real time test execution on Sauce Labs

A screenshot of a computer

Description automatically generated

* 1. **:** Post Execution of visual test, user can see test results in VS Code terminal.

A screenshot of a computer program

Description automatically generated

* 1. : Post Execution of visual test, Sauce visual triggers notification to users associated with project.

A screenshot of a email

Description automatically generated

**3.7**: User can also access video recording of visual test in Sauce Labs

A screenshot of a computer

Description automatically generated

**Step 4: Test Results Review Process in Sauce Visual:**

* 1. Post execution of visual tests, in case there are instances where deviation with respect to baseline screenshots have been observed, user can review such deviation and either accept (in case of new UI enhancements) or reject (in case of defect) .

A screenshot of a computer

Description automatically generated

* 1. **Understanding Visual States:**

**New:** In case visual tests executed first time, all screenshot gets captured as new. User can access these and accept or reject per results. In case user accepts any or all screenshots, those will be considered as baseline screenshots.

**Changed:** In case of any deviation with respect to baseline screenshot of same screen, screenshot gets captured as changed so that user can view deviations and accept or reject per results. In case user accepts any or all screenshots, those will be considered as baseline screenshots.

**Accepted:** In case of no deviation with respect to baseline screenshots and all screenshots accepted by user gets categorized as accepted

**Rejected:** In case of deviation with respect to baseline screenshot and if it’s legit defect, user can reject such screenshot and such instances captured as rejected.

* 1. **: Viewing Change Details for E2E Tests in Sauce Visual**

Changed UI states will include highlights of visual changes directly overlayed on the screenshots. Screener automatically organizes these changes into four categories:

* **Structure:** structural changes are when new visual items are added, or old items are removed.
* **Layout:** layout changes are when visual items change position or dimension. o Style: style changes are when CSS styling gets updated (e.g., font size, color). o Content: content changes are when text or graphics get updated.

A screenshot of a computer

Description automatically generated

In above screenshot, user can see which type of UI deviation, where is deviation occurred and with these specific, **it will help development time to provide resolution quickly.**

**4.4: Side by Side comparison (Baseline vs current state of application)**

A screenshot of a computer

Description automatically generated

**4.5: User can accept, reject, or ignore changes**

A screenshot of a computer

Description automatically generated

**4.6 Post reviewing all screenshots, user can refresh Sauce visual dashboard.**

A screenshot of a computer

Description automatically generated

**4.7:** User can also track visual test executions in the past using **Build History**

A screenshot of a computer

Description automatically generated

**4.8:** Post completion of review process, user receives automated email notification

A screenshot of a email

Description automatically generated

## Visual testing with Selenium

Selenium is an open-source test automation framework that allows web apps to be tested across different browsers & operating systems. It is compatible with different programming languages.

**Visual Testing by Selenium** will help the QA to detect when the layout of a web page has changed. Instead of comparing pixels from screenshots, layout differences can be detected by comparing HTML tags and attributes with a baseline. If a change is detected, it could mean that something broke, the web page was redesigned, or dynamic content changed.

**Step by Step Tutorial of Selenium Visual Testing:**

**Pre- Requsites :**

* Install [Eclipse](https://nodejs.org/en/download/)
* Install Selenium WebDriver
* Framework Maven TestNG

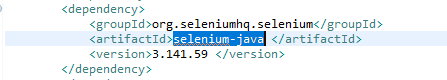
**Step- 1:**

**Sauce Visual-Selenium Framework Integration:**

* Create a “saucelab.properties” file.
* Add sauce\_userName, sauce\_accessKey, screenerIO\_url, screener\_apiKey, ProjectName in saucelab. properties file.
* Also add sauce\_url, parent\_tunnel, tunnel\_identifier, Browser, platform and ScreenResolution.
* Add all parameters as listed, which are required for Sauce Visual integration in the file.

**Step- 2:**

**Update selenium-java to latest version in pom.xml**



A picture containing text

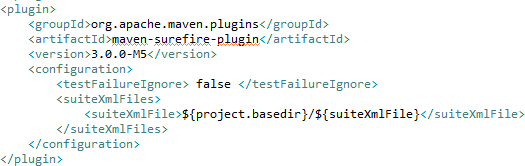
Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated



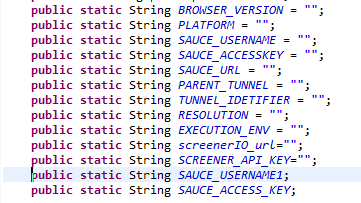
Text

Description automatically generated

**Step- 3:**

Initialize the Saucelabs.Properties Parameters in InitTest.Java file and get the vaules from property file (Saucelab property)

Text, letter

Description automatically generated

**Step- 4:**

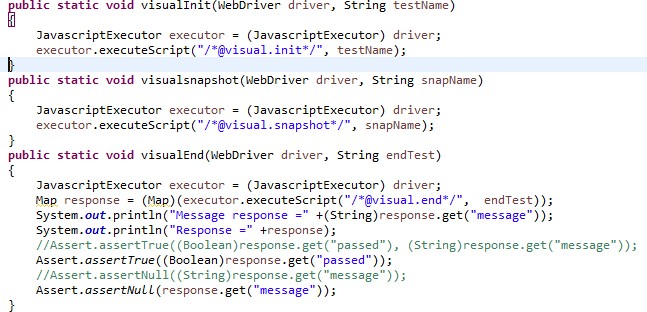
Add an else if condition when the execution environment is ‘screenerIO’ in **Driver.Java** for all browser types and set the desired capabilities, Set the **WebDriver Capabilities** to connect to remote hub (i.e. given ScreenerIO\_url) Refer the image below



**Step- 5:**

**Visual Commands to perform UI snapshot Verification.**

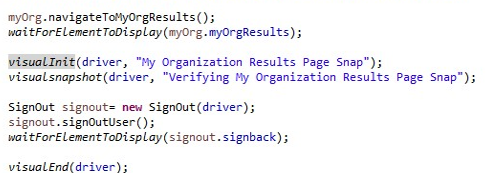
Create a **visualInit**, **Visualsnapshot** and **VisualEnd** methods in **Driver.Java** having respective commands to initialize the visual test, to take visual UI screenshot and to end visual test respectively. Refer the image below.



**Step- 6:**

**Write your first Java-Selenium WebDriver Visual Test**

Add a new file example **sauceVisualTest.java.** Refer the image to add Visual Init, Visual snapshot and Visual End Commands in the Test class.



**Step- 7:**

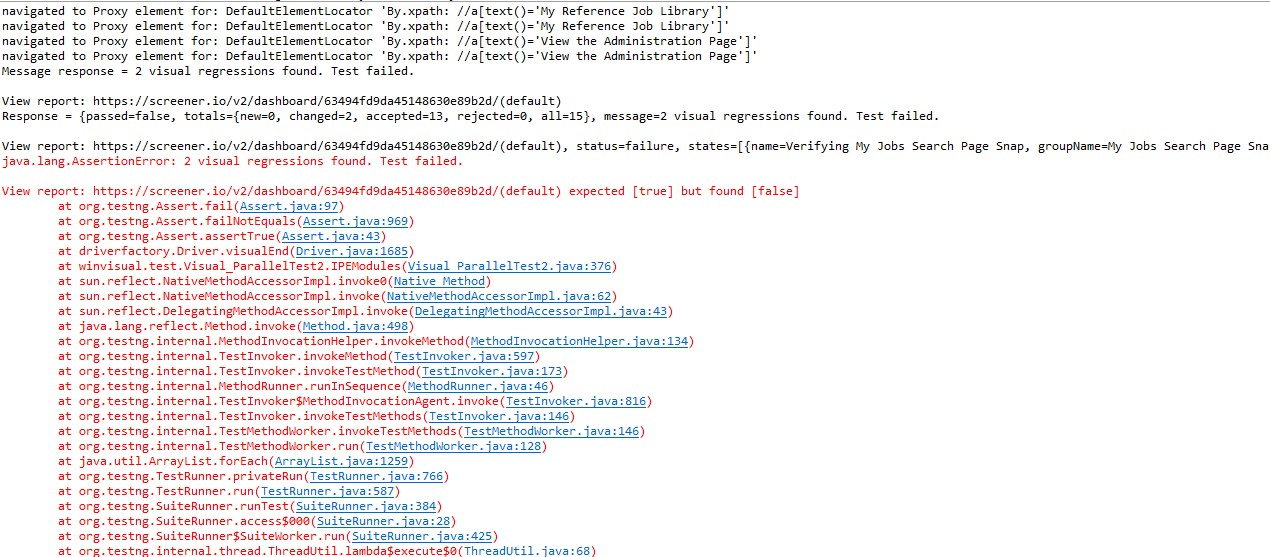
**Execute the sauceVisualTest.Java by creating a .xml file and use the command below**

* clean install test -DsuiteXmlFile=sauceVisualTest.xml -Denv=QA

**Step- 8:**

**Real time tracking and review results in Dashboard**

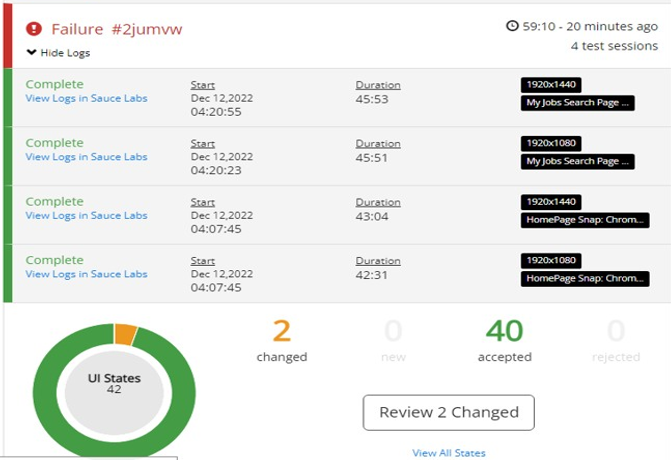
**Console output:**



**In Screener. Io dashboard**

Users can see the script run by clicking on View Logs in Sauce Labs.

Once the execution Completes in screenrIO. Users can see the screenshots in the **new** option. A notification email is sent to all associate project users to review screenshots.



* Click on **new/View All States/Review New options**, Review the screenshots depending on the preferences and overall visual impression user can accept or reject the screenshot.
* **Accepted/Rejected** Visual snapshots show the number of accepted & rejected numbers in the screener.io dashboard.
* After accepting the screenshots, those first execution of visual snapshots becomes the Baseline to compare with next executions visual snapshot, depending on the comparison changes user can review and accordingly can **accept or reject** the visual snapshot.
* The **accepted, rejected screenshot** **counts** user can see on the screener.io Dashboard.
* If any deviation is present in the next execution, then the screenshot will appear in the **changed** option in the screener IO Dashboard. (Changed Visual snapshots count showing in the above picture)
* Review the **changed** snapshots, After the review, user gets the Build Accepted mail and dashboard shows as successful.

## Visual Testing Tools Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | **ScreenerIO** | **Java-Selenium (non-visual)** | **Cypress(non-visual)** | **WebdriverIO (non-visual)** |
| Open Source/ No Licensing cost | YES | YES | YES | YES |
| Parallel Execution | YES | YES | YES | YES |
| Scripts maintenance | NO | YES | YES | YES |
| Test creation |  |  |  |  |
| Scripting required | YES | YES | YES | YES |
| Cross-browser capability | YES | YES | YES | YES |
| Multi Platforms (Windows/Mac) | YES | YES | YES | YES |
| Desktop app testing | NO | NO | NO | NO |
| Reusability | YES | YES | YES | YES |
| Testing types (Regression, functional, Smoke, etc) | All | All | All | All |
| Data-driven testing | YES | YES | YES | YES |
| Learning Curve: | YES | NO | YES | YES |
| CI/CD integration | YES (inbuilt | YES | YES | YES |
| Reporting | Very Good | Basic | Good | Good |
| Record and Playback | YES (on cloud) | NO | NO | NO |
| Technical support | YES | Well-developed community | Decent | Minimal |
| Cost | License (provided by client) | NO | NO | NO |

## Visual Testing Benefits, Challenges, and learnings

**Benefits:**

* The number of assertions in a script will be replaced with one visual snapshot and in case of any changes in the applications user can compare the two snapshots and accept the new changes.
* Cross platform and Cross Browser parallel execution can be achieved.

**Challenges/Disadvantages:**

* Users must manually review and update the baseline each time if there is deviation in the application UI.
* If the test execution cross more than 30minutes, the test will show as Timeout in the Screener IO dashboard.

Sauce Visual Limitation: Test Execution timeframe: In case Visual test exceeds beyond 30 mins, visual test will fail with timeout error. So in order to avoid such situation, it’s advised to split test into multiple tests.

**Advantages of Sauce Visual Integration:**

• Number of assertions in a script will be replace with one visual snapshot and in case of any changes in the applications we can compare the two snapshots and accept the new changes.

• It automatically spots the UI inconsistencies across different platforms.

• Increases test coverage by performing both visual and functional testing.

• Allows tester to run tests in parallel for fast execution across multiple browsers, platforms, and screen resolutions.

• It supports continuous integration/continuous deployment (CI/CD) platform.

## Visual Testing Best Practices

Web application testing should be done early in the development stages to speed up the app-building process and reduce risks. Testing can be performed automatically with a software program or manually with a human tester.

**Best practices for visual testing include:**

* Performing system tests first
* Creating small specs
* Performing multiple visual checks within a single test
* Using automated screenshot comparison
* Testing a wide range of visual elements, including text, images, colors, and layout

**When selecting a tool, you can consider features like:**

* The ability to capture screenshots of the whole page or specific elements.
* Comparing screenshots pixel by pixel or using AI-based algorithms
* Integrating with your existing testing framework and CI/CD pipeline
* Handling dynamic content, animations, and responsive design

## Visual Testing References

1. <https://docs.saucelabs.com/visual/e2e-testing/setup/>
2. <https://www.ndt.net/article/wcndt00/papers/idn276/idn276.htm>
3. https://docs.cypress.io/guides/tooling/visual-testing#\_\_docusaurus\_skipToContent\_fallback

**About the Author:**

* + - 1. Kavya Bhat (10667587).
      2. Ashish Namdeo (10670517).
      3. Roja K (10676861).
      4. Suhani Khandelwal (10684155).
      5. Shivani Boda (10676813).

