applitools
AI-Powered Visual Testing & Monitoring

Automated Visual Testing Best Practices Guide

By Angie Jones

Automated Visual Testing Best Practices Guide

Many of the top high-performing companies in software, financial services, and healthcare verticals are using Applitools Eyes for their visual test automation needs. We've talked with these companies to gain a better understanding of how they are using the product and to identify common practices that are working well for teams. This guide provides a set of best practices and our recommendations for implementing and maintaining automated visual tests with Applitools based on our customers' experiences.

Summary of Recommendations

Here is a summary of our top recommendations contained within this guide. Details on each of these, including when to detour from the recommendation can be found within the following sections of the guide.

1. **Use Eyes for both visual and functional assertions**
   The Applitools Eyes API should be used to complement or encompass functional assertions with less lines of code and more test coverage. **READ MORE**

2. **Verify the entire page**
   While there are various options to visually verify your application with Applitools Eyes, we recommend verifying the entire page of your application to obtain the most coverage from your visual testing. **READ MORE**

3. **Use the Strict match level**
   By default, Applitools Eyes uses the Strict match level algorithm (our recommended comparison method), which employs AI to compare your baseline image with the current image from a regression run. **READ MORE**

4. **Use Ignore annotation for data that is not relevant to the test**
   We recommend using our Ignore annotation to ignore parts of your application's page that are not pertinent to the test. **READ MORE**

5. **Programmatically apply annotations when possible**
   When annotations are needed, use fluent checks to programmatically apply them to baseline images before the test is executed. **READ MORE**

6. **Use steps to include multiple visual checkpoints in a single test**
   It's recommended to perform multiple visual checks where there are multiple visual states that need to be verified in a given scenario. **READ MORE**

7. **Group related tests into batches to enable easier management as well as automated maintenance of baselines**
   For related tests, it's recommended to use batches so that the tests display together on the Applitools Dashboard within an aggregated view. **READ MORE**

8. **Utilize a wrapper class to encapsulate visual checks**
   It's a good practice to create a wrapper class that isolates the Eyes API calls so that if there are any changes to the API, the required changes to the test code will be limited to an individual class. **READ MORE**

9. **Use Layout mode for early unverified localization testing**
   Before your localized app has been verified by a specialist, use Layout mode. After you're sure the content is correct, use Strict mode. **READ MORE**

10. **Tag visual tests and execute them any time the UI is under test**
    Utilize tagging capabilities of your test runner to easily include and exclude visual tests when executing. **READ MORE**

11. **Use the Ultrafast Grid to run cross-platform tests**
    Applitools' Ultrafast Grid enables you to execute your tests across many different browsers, viewports, and devices at a fraction of the time it takes with other solutions. **READ MORE**

12. **Gate your builds with relevant visual checks**
    With the accuracy of Eyes API, we feel confident recommending that you have your visual tests follow the same gating strategy used for your functional tests. **READ MORE**

13. **Visual checks can be used when using feature files**
    When practicing BDD, continue to write feature files without implementation detail and use the step definition code to execute visual checks where necessary. **READ MORE**

14. **Use the Applitools dashboard to collaborate**
    The Applitools Dashboard should be used to annotate screenshots of failures with bug regions, assign failures to developers to review, and remark on questionable changes. **READ MORE**

15. **Integrate Applitools with Slack for faster collaboration on failed visual tests**
    If your team already uses Slack for collaboration, we recommend integrating Applitools with Slack, which makes sharing and resolving failures easier. **READ MORE**

What Is Visual Testing

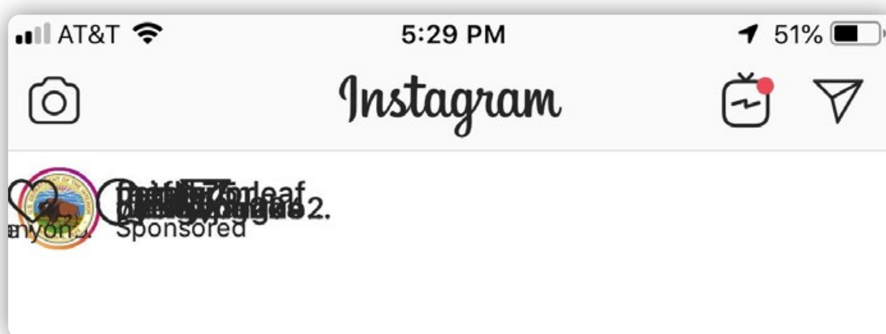
Automated visual testing is a technique to verify that your application appears to the user as you intended. The Applitools Eyes API can be used to complement or **encompass functional assertions** with less lines of code and more test coverage.

Applitools Eyes is both flexible and powerful, and makes a wonderful addition to any UI or mobile test automation toolkit.

Visual testing can be used whenever there's a need to verify the display of data on a web or mobile application.

Functional assertion libraries verify information in the **DOM**, but are incapable of verifying how that data is actually presented to the user.

For example, here's a visual bug on Instagram. Functional assertions that check if the text exists would pass because yes, it does indeed exist in the DOM. However, it misses the fact that all of the text is overlapping and therefore is unreadable by the user.



To avoid missing such bugs, Applitools should be used in your mobile and web UI tests.

Where to Use Visual Testing

Our recommendation is to use Applitools Eyes as a superset for both visual and functional assertions.

Just like functional assertions, visual assertions are used at the point of verification. An automation tool (e.g. Selenium, Cypress, Appium, etc) is used to interact with the application and then Applitools Eyes is used to visually verify the outcome.

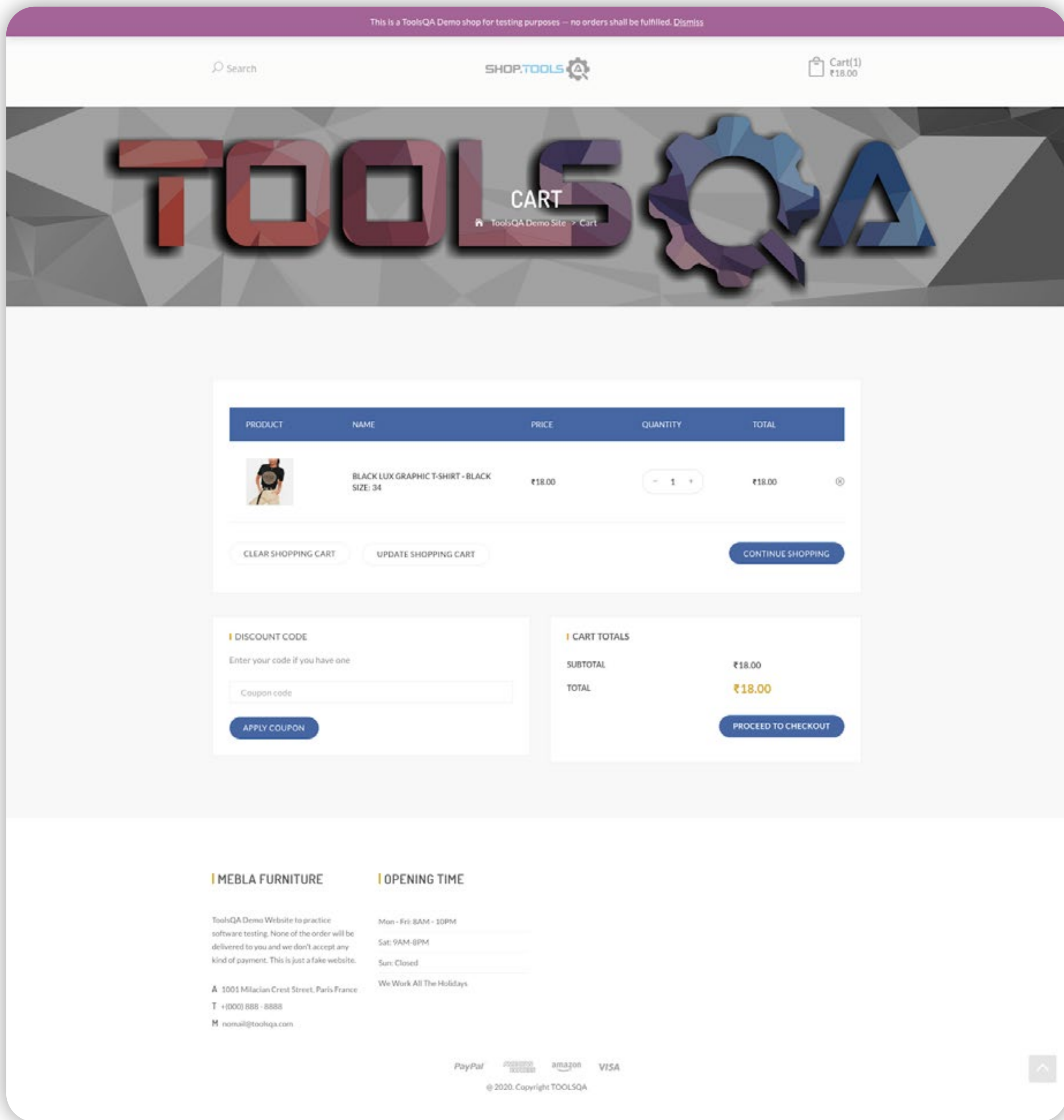
Also, just like with functional assertions, it's important to make the visual assertion only when the application is ready to be verified. For example, if your app needs to load data, your test code will need to wait until that data is loaded before verifying it. This is true of both functional and visual assertions.

Some users create separate tests for their functional and visual checks. They utilize the functional tests to assert that all data is accurate, and they use the visual tests to assert that there are no visual bugs.

While this is certainly a viable option, it is redundant. Not only is the visual check asserting that there are no visual bugs, but it is also asserting that everything on the page is correct, including the data.

Therefore, visual checks are a superset of functional checks, and in many cases can do the job of both.

For example, consider a test that adds an item to a shopping cart, and then wants to verify that the item was properly added.



Including functional assertions for everything here is unnecessary if using a visual assertion. Not only will the visual assertion make sure all of the items are displayed properly, it is inherently also making sure that the product title, size, color, quantity, price, total, etc are also correct.

An exception to this suggestion would be when it's necessary to verify things that are not visible on the UI, for example for backend databases or web service calls. In these cases, we recommend using visual assertions for the UI and coupling these with functional assertions for the backend.

Visual Testing Techniques

Full Page

While there are various options to visually verify your application with Applitools Eyes, we recommend verifying the entire page of your application to obtain the most coverage from your visual testing. This is the most common usage of visual testing.

```
1    @Test
2    public void testCart(){
3      productPage.addToCart();
4      app.goToCart();
5      eyes.check(Target.window().fully());
6    }
```

The benefit of using this approach is that everything on the entire screen will be captured and verified so you don't have to worry about missing key assertion points.

For web tests, we recommend setting the viewport size when calling Eyes' open method and specifying a size that's at least 50 pixels below the maximum possible viewport size. This ensures that your test is run against the same window size each time. Otherwise, browser viewport variances can result in a new baseline being created.

```
1    eyes.open(driver,
2              "My App",
3              "My Test",
4              new RectangleSize(800,600));
5
```

When using Applitools Eyes to check the entire page, we also recommend setting the API to capture a full page screenshot. This will ensure that the entire screen will be captured, even if Applitools has to scroll to capture it in its entirety. When requesting a full page screenshot, Eyes will capture a screenshot of the visible portion of the page, then scroll and capture a screenshot of the newly visible portion of the page. It will continue this until the entire page has been captured. Once done, Eyes will stitch together each of the individual images to make one single image of the entire page. In most cases, this is all that's needed. However, on applications with sticky headers (meaning the header of the page is visible even when scrolling), then you don't want to have the header duplicated in every portional screenshot and appearing multiple times in the stitched version of the full image. To avoid this, you can make a call to the Eyes command to **set the stitch level** to CSS.

```
1    public void verifyFullWindow(){
2
3        eyes.open(driver,
4                  "My App",
5                  "sticky header full page",
6                  new RectangleSize(800, 600));
7
8        eyes.setStitchMode(StitchMode.CSS);
9        eyes.check(Target.window().fully());
10       eyes.closeAsync();
11   }
```

While validating the entire page is the most common and thorough approach, we do realize that in some cases, this may be too much for your needs.

Examples:

- page is undergoing active development so is not yet stable
- page contains information that is irrelevant to the test

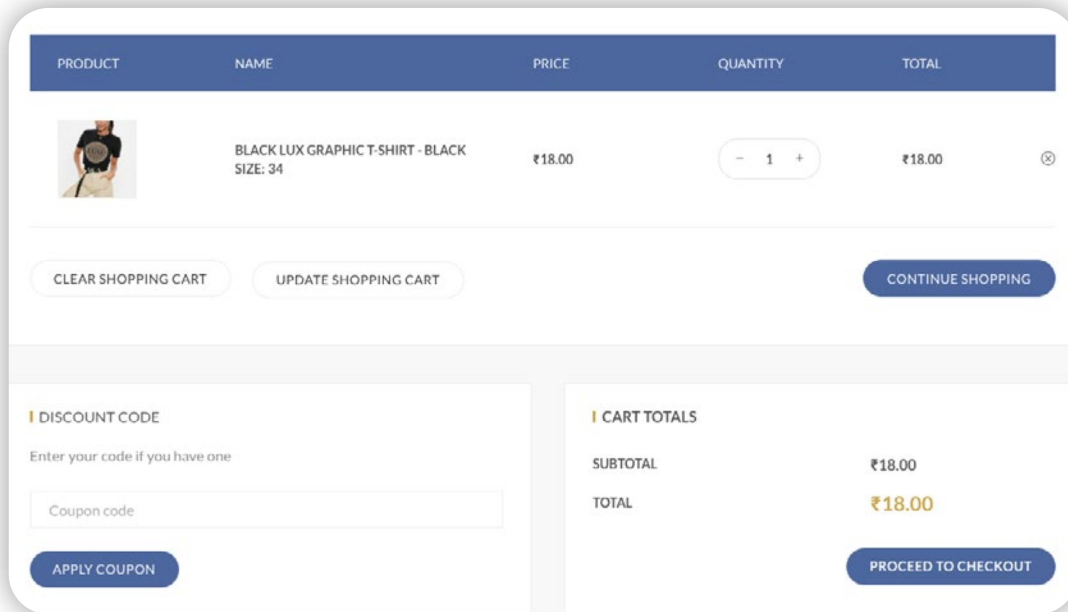
In these cases, you can scope your visual test down to a specific element on the page.

Applitools is extremely flexible and allows you to visually verify as much or as little as you desire. Let's look at when it may be better to use alternative techniques.

Element

Applitools provides a method to allow you to check a specific element, region, or frame of the page. This is very useful for those cases when your test does not need to verify the entire page.

For example, for a test that adds something to a shopping cart, perhaps the only thing you'd like to verify is the actual cart.



This is possible by calling the Applitools Eyes method that checks a specific element.

This allows you to only verify the part of the page that you care about.

```
1    @Test
2    public void testCart(){
3        productPage.addToCart();
4        app.goToCart();
5        eyes.check(Target.region(By.id("cart")));
6    }
```

When using this approach, be careful to think about what you could be missing. Are there any other elements on the page that are also important? Perhaps the cart icon up top that is showing how many items are present? You can add an additional visual assertion to this test to verify this element as well.

Be careful not to scope your test too narrowly. For example, scoping the cart test just to the product line would miss other important information about the cart like the cart totals and button states. Scope wide enough to catch everything you need to test, but narrow enough that you're not capturing parts of the screen that would be problematic to include in your test.

Also consider coupling your visual assertions with functional ones when using this approach if you need additional coverage.

Components

If creating frontend components, you can also use visual validation to unit test these components. We recommend testing the visual states (hover, focused, open, etc) of all of your components separately from testing the pages of your application. We recommend using Storybook for building a component library with all their states, along with the **Applitools Storybook SDK** for testing it.

Match Levels

In addition to being able to specify what to visually validate, you can also specify how it should be validated. Applitools provides various comparison algorithms called match levels which can be applied on either the entire page, or to specific regions of the page.

Strict

By default, Applitools Eyes uses the Strict match level. This is our recommended strategy for verifying stable and static pages. This algorithm will compare your baseline image with the current image from a regression run and use AI to detect the things that the human eye would. The Strict match level is the most widely used match level.

Content

The Content match level is similar to the Strict match level except that it ignores color differences. You may have an application where you're only interested in verifying the content and not necessarily the color of it. For example, if colors are customizable by your users (e.g. profiles), your test may not be certain what colors it will see, and this may be irrelevant to what is being verified. In this case, Content match level is a great solution.

However, our **Baseline Variation feature** is recommended if the color differences are because of A/B testing of your application.

Layout

The Layout match level allows you to verify dynamic content. It does not verify the actual data of the tested area but it does utilize AI to determine the pattern of the layout of your page. Customers who utilize this approach are verifying applications such as news sites and social media applications, where the data is dynamic and will always change but the structure of the page is the same.

The Layout match level will ensure there are no visual bugs, but with this approach, you lose the ability to also verify the functional data, as this technique does not consider the text, images, or other data on the UI; it will only verify that structurally the page looks ok.

As for full page testing, we do not recommend using Layout mode as your complete automation solution. Layout mode is great for light-weight testing such as smoke tests to ensure your application is up and loaded properly. It should be followed by additional tests that verify the actual functionality and data of your application.

However, it is valid to use Layout mode on regions for greater test stability.

Exact

The Exact match level uses pixel-to-pixel to compare the images. We highly discourage you from using this technique, as in general, pixel-to-pixel comparisons are highly flaky and prone to false negatives. This match level is mostly in the product for demonstration purposes of what not to do.

Annotations

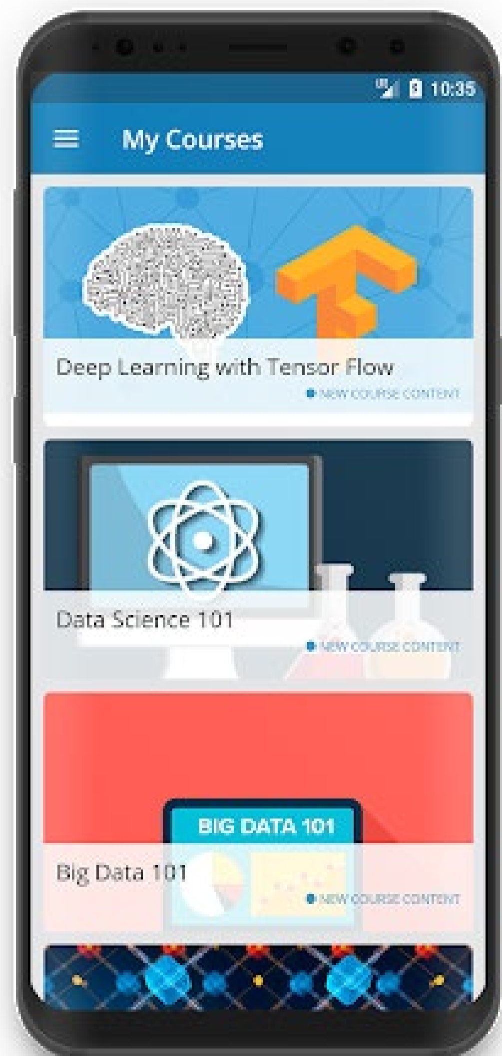
In addition to various match levels and check types, Applitools Eyes also allows you to annotate your images for even more flexibility.

The Ignore annotation is a customer favorite and is the most popular annotation used. It allows you to mask any part of your screenshots to be ignored.

We recommend using the Ignore annotation for data that is not relevant to the test. For example, with mobile app testing, there's usually a status bar visible at the top of the application which displays dynamic content such as the time, battery level, notifications, etc.

This data is highly dynamic, and more importantly is irrelevant for the tests, so we recommend using an Ignore annotation for such cases.

Annotations can be applied in two ways: programmatically and via the **Dashboard**.



We recommend programmatically applying the annotation by using a fluent check. The fluent check will allow you to specify your target (window, element, frame, etc) as well as any annotations that should be applied to specific regions of that target.

```
1    @Test
2    public void testWithFluentCheck() {
3        eyes.check(Target.window()
4                .ignore(By.id("status-bar")));
5    }
```

By using this technique, the annotation will be applied before the verification is done on the baseline and checkpoint images thus eliminating the need for you to annotate and pass each test manually in the Dashboard.

Adding annotations programmatically also allows you to easily mask areas across different browsers, viewport sizes, and operating systems with one line of code, thus saving maintenance time.

Other annotations include Floating, Strict, Layout, and Content. Each of these can be used on specific regions of the screenshot. They can even be mixed and matched on a given image, thus enabling really flexible tests to meet your needs.

For example, the Layout annotation comes in very handy for use on dynamic portions of the page such as timestamps, prices, etc. This is the recommended approach for when the area has a known layout but the data is dynamic.

```
1    @Test
2    public void testDynamicAd() {
3        eyes.check(Target.window()
4                .layout(By.id("banner-ad")));
5    }
```

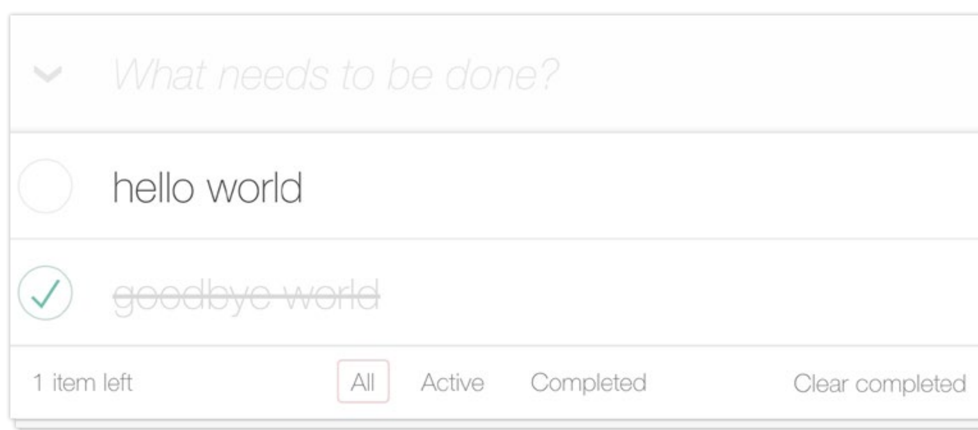
And the **Floating annotation** enables a stricter verification than Ignore and Layout annotations, as it will verify the content of the specified region and ensure it's within a given scope of the image. The Floating annotation is recommended in cases where the content can shift on the page (e.g. an item within a list of search results).

Multiple Checks Within a Test

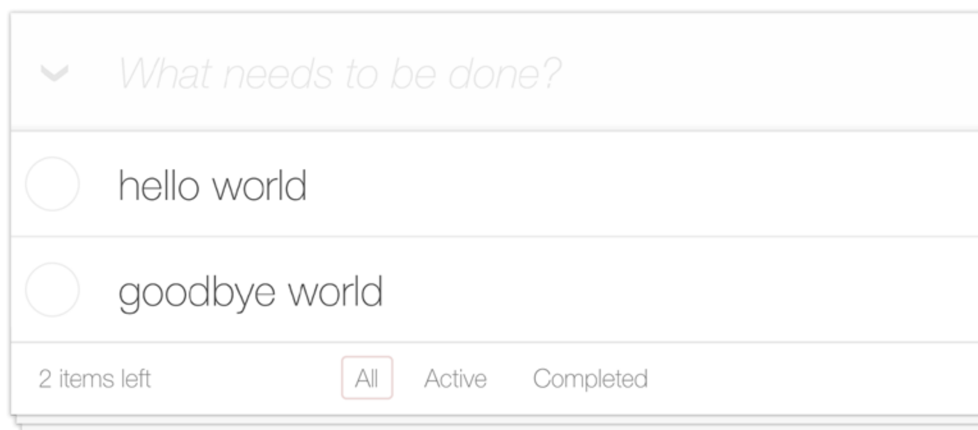
Tests are not limited to a single visual check. You can perform multiple visual checks within a single test and we recommend doing that when there are multiple visual states you need to verify for a given scenario.

For example, in a test where you want to reinstate a completed item from a Todo list, you may want to verify that the item is actually marked completed before verifying its reinstatement. Otherwise, you could miss a bug if the item was never marked completed in the first place.

These multiple checkpoints are known as **steps** within the test.



Step 1: marking item complete

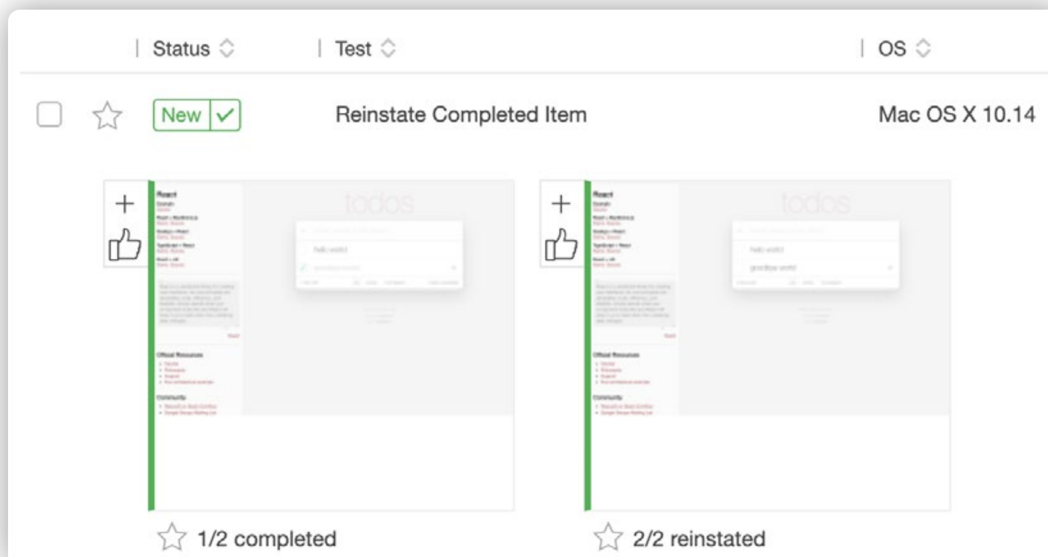


Step 2: reinstating completed item

To complete all steps within one test, call the check commands between Eyes' open and close commands.

```
@Test
public void testReinstateCompletedItem(){

    eyes.open(driver,
            "Todo",
            "Reinstate Completed Item",
            new RectangleSize(800,600));
    todo.addItem("hello world");

    String item = "goodbye world";
    todo.addItem(item);

    //Mark as completed
    todo.clickItem(item);
    eyes.check("completed", Target.window().fully());

    //Reinstate
    todo.clickItem(item);
    eyes.check("reinstated", Target.window().fully());

    eyes.closeAsync();
}
```

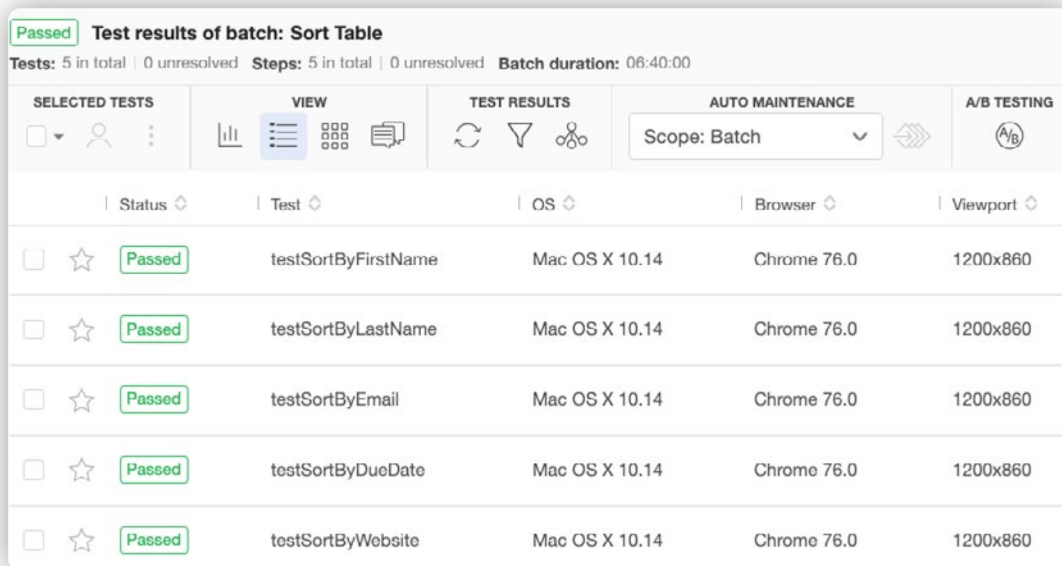
In the Applitools Dashboard, the steps will show as multiple images within a single test, and can be managed and maintained individually as well as collectively.



Grouping Tests

You may have multiple tests in which it makes sense to group them into a suite. With Applitools, this is known as a batch. We recommend using **batches** for tests that have page elements in common (e.g. Shopping Cart tests).

In the Applitools Dashboard, when tests are run as part of a batch, they will all display together as individual tests within one aggregate view.



In addition to being able to see related tests in one view, a major benefit of using batches is the automated maintenance provided within the Applitools Dashboard. For example, if a common element (e.g. title of the page) changes within multiple tests within the batch, you only need to update one of the tests and by default, Applitools will apply that update to all tests within the batch, thereby automating the maintenance and reducing the manual headache of doing so.

In addition to functional groupings, a batch can be further grouped by commonalities such as device, browser, operating system, viewport, etc. For example, the following batch contains 40 tests.

Passed **Test results of batch: Sort Table**

Tests: 40 in total | 0 unresolved **Steps:** 40 in total | 0 unresolved **Batch duration:** 09:43:20

SELECTED TESTS | VIEW | TEST RESULTS | AUTO MAINTENANCE | A/B TESTING

Scope: Batch

Status	Test	OS	Browser	Viewport	Device
Passed	testSortByFirstName	Linux	Chrome 74.0	640x360	Galaxy S5 (Chr...
Passed	testSortByFirstName	Linux	Chrome 74.0	800x600	Desktop
Passed	testSortByFirstName	Linux	Chrome 74.0	812x375	iPhone X (Chro...
Passed	testSortByFirstName	Linux	Firefox 68.0	800x600	Desktop
Passed	testSortByFirstName	Linux	Firefox 68.0	1200x800	Desktop
Passed	testSortByFirstName	Linux	Chrome 74.0	1200x800	Desktop
Passed	testSortByFirstName	Linux	Chrome 74.0	375x812	iPhone X (Chro...
Passed	testSortByFirstName	Linux	Chrome 74.0	360x640	Galaxy S5 (Chr...
Passed	testSortByLastName	Linux	Firefox 68.0	800x600	Desktop
Passed	testSortByLastName	Linux	Firefox 68.0	1200x800	Desktop
Passed	testSortByLastName	Linux	Chrome 74.0	812x375	iPhone X (Chro...
Passed	testSortByLastName	Linux	Chrome 74.0	375x812	iPhone X (Chro...

This is a lot to look through, and many of the tests have the same name but differ by the environment. You can use the dashboard to group this batch by device, for example.

While Applitools offers many grouping options by default, you can also create custom ones.

Test Code Design

While calls to the Eyes API can be made directly within a test function, we recommend abstracting the calls into a separate utility class. In doing so, if there are any changes to the API, the required changes to the test code will be limited to an individual class.

The wrapper class can serve to not only encapsulate visual checks, but also initializing and configuring the Eyes SDK.

An example of this would be **such a class** which contains wrapper methods to make calls to the Eyes functions.

Before Tests

It's recommended to utilize a prerequisite method (e.g. @BeforeClass) to set up the Eyes object and set the API key before all tests run, and another prerequisite method (e.g. @Before) to make a call to Eyes' open method before each visual test is executed.

Within Tests

Inside of the test itself, we recommend calling any Eyes check* methods necessary. You do not need to open Eyes before every visual check; once is enough per test, even if there are multiple checks within the test.

After Tests

We recommend utilizing postrequisite methods (e.g. @After) to close the Eyes object after each test. In addition, it's also recommended to make a call to Eyes' abortIfNotClosed method to ensure that no visual tests hang without completion.

Framework Classes

It is not recommended to use Eyes within framework classes such as ones that utilize the Page Object Model design pattern. These classes are typically reused across multiple tests, and adding visual checks within the page object methods could cause ambiguity, and be called in events where visual testing is not yet required.

However, creating reusable utility methods to execute the visual checks is encouraged.

Localization Testing

When utilizing Applitools for visual testing, we recommend approaching this in stages. If your translated site has not yet been manually verified by a localization specialist, we recommend you use the Layout mode of Applitools eyes for basic sanity checking. This will ensure that the translated content has not broken the layout of your application.

After the translated site has been verified by a localization expert and you're sure that the content is correct, we then recommend switching to Strict mode.

Executing Visual Tests

Tagging

Not every test in your framework will be a visual test. So, we recommend using the tagging capabilities of your test runner to denote your visual tests (e.g. @visual). This will allow you to easily include or exclude visual tests from your executions based on the features under test.

This tag can be coupled with any other tags such as ones denoting functional areas (e.g. @search). Given this, if testing the UI of the search functionality, you can also include the visual tests. Alternatively, if testing search on the backend, the @visual tests can be skipped.

Disabling

We also recommend using properties or environment variables to denote when not to run visual tests at all. Given the state of the property/variable, you can call the setIsDisabled method of Eyes passing in a boolean (true/false) value indicating whether to run the visual checks or not.

Frequency

We recommend executing your visual tests any time your UI tests run, or a UI-related feature check-in is made. By using the tagging convention above, this will reduce the number of visual checks needed to execute.

Platforms

Applitools Ultrafast Grid enables you to execute your UI tests across many different browsers, devices, and viewports at a fraction of the time it takes with other solutions. We recommend using the Grid for all cross-platform and responsive-width web tests and also for any mobile tests that are not specifically verifying native gestures (swipe, pinch, etc).

There is no need to strategically choose which of your supported environments to execute against, as the duration will be roughly the same no matter if you run against 10 or 20 configurations.

Gating Deployments

A common question is should development builds be gated by visual checks. With the accuracy of the Eyes API, we feel confident recommending that you treat your visual tests just as you would your functional ones. If your UI, integration, and end-to-end tests should work before any new code is integrated, then you can include the visual checks for these tests in your pipeline to gate check-ins as well.

Collaboration

API Keys

When using Teams within Applitools, we recommend every member of the team using their own unique Applitools API key as opposed to having a public one for the team. This is important for security purposes, so that if a user leaves the company, their private API key can be disabled and will not affect any other user.

Behavior-Driven Development (BDD)

When practicing BDD, there's **no need to consider the implementation details** of how you will automate the tests. This remains true when using visual validation. Continue to write your feature files as normal, and use the step definition code to call the Eyes API.

```
1    public void verifyTransactionRecord() {
2      NavigationUtils.goToAccountActivity(accountId);
3      eyes.check(Target.window().fully());
4    }
```

Bugs

When your visual tests run, the results are stored in the Applitools dashboard. Within the dashboard, there are several collaboration features, including the ability to annotate regions of the screenshots as bugs.

We strongly recommend indicating the bugs and using the Root Cause Analysis dashboard feature to capture and specify the exact cause of the bug. The bug annotations can be assigned to specific developers, if integrated with a bug tracking system such as Jira. Clearly annotated bugs will allow for easier debugging and faster fixes by the developers.

Remarks

We recommend utilizing the Remarks annotation within the dashboard to pose questions to developers about changes within the screenshot. Remark regions are great for changes where you aren't quite sure if the change is intended or not.

Slack Integration

Applitools integrates with Slack. If your team already uses Slack for collaboration, we recommend integrating Applitools with Slack, which makes sharing and resolving failures easier.

This integration can be configured to send notifications to slack on every batch completion, or for only the batches that result in an Unresolved state - meaning visual differences were detected.

About the Author

Angie Jones

Sr. Developer Advocate & Director of Test Automation University
angiejones.tech

Angie Jones is a Senior Developer Advocate who specializes in test automation strategies and techniques. She shares her wealth of knowledge by speaking and teaching at software conferences all over the world, as well as writing tutorials and blogs on angiejones.tech.

Angie loves teaching. In fact, in addition to working as an automation engineer, she also was an adjunct college professor where she taught Java programming courses.

As a Master Inventor, Angie is known for her innovative and out-of-the-box thinking style which has resulted in more than 25 patented inventions in the US and China. In her spare time, Angie volunteers with Black Girls Code to teach coding workshops to young girls in an effort to attract more women and minorities to tech.