The Visual Testing Revolution

[Slide]

Hello everyone! Thanks for attending my talk today. My name is Andrew Knight, but you can call me “Pandy” for short. I’m the Automation Panda. Historically, I’ve been a Software Engineer in Test, building high-scale solutions for testing problems. And, as many of you know, for the past half a year, I’ve been *the* Developer Advocate at Applitools.

Another cool thing I do at Applitools is serve as Director of Test Automation University. TAU is one of the best platforms online for learning about testing, automation, and software quality. I’ve taught a few courses myself on Python.

[Slide]

Today, I’m going to talk about testing. Surprise, surprise! Who would’ve thought?

In particular, though, I’m going to talk about *visual testing*. Visual testing techniques are an incredible way to maximize the value of your functional tests. Instead of coding a bunch of assertions based on text or attributes, visual testing captures a full snapshot of a page and looks for visual differences over time.

My goal today is to “open your eyes” to see how visual testing can revolutionize how you approach software quality. This isn’t just another “nice-to-have” feature that’s on the bleeding edge of technology. It’s a tried-and-true technique that anyone can use, and it makes testing easier! I want you to see things in new ways.

[Slide]

We all know that there are several different kinds of testing. What are some of the kinds of testing y’all have done? Shout it out! [Wait and solicit]

You name it, there’s a test for it. We could play buzzword bingo if we wanted. But what is “testing”?

[Slide]

In simplest terms, testing is interaction plus verification. That’s it! [Slide] You do something, and [Slide] you make sure it works. Every kind of testing reduces to this formula.

Manual testing accomplishes both interactions and verifications by direct human action. Somebody needs to bang on a keyboard to drive the test. Automation drives interactions and verifications with a script. We like to think that automation is so great because it doesn’t need

any human intervention, but we all know that's not true. Humans still need to write the tests, develop the scripts, and fix them when they break. Paradoxically, test automation isn't fully autonomous.

[Slide]

Visual testing helps to change that. Although humans still need to figure out interactions, visual testing techniques make verifications autonomous. Teams take snapshots of their views and look for changes over time. They catch more kinds of problems while ironically simplifying test code. A tester doesn't need to think critically about what specific elements to check on a page. Snapshots capture everything!

[Slide]

Unfortunately, many folks seem to have the impression that visual testing is an "advanced" technique that requires a high level of testing maturity to be valuable. That's not the case at all. In fact, I want to flip the script entirely: *Visual test automation is easier than traditional test automation.* It isn't some bleeding-edge technology useful only to FAANG companies. It isn't out of reach for teams just starting their test automation journey. Visual testing makes functional testing easier and stronger. It's something teams should do first before attempting to automate longer, more complicated tests with traditional techniques.

Big claims, right? Let's see what I mean. We are going to automate a web test together in Java with Selenium WebDriver using traditional interactions and verifications, and then we will supercharge it with visual testing techniques using Applitools.

[Slide]

First, we need a web app to test. We could test an app of any size, but I'm going to choose a small one for the sake of our demo. This is Applitools' demo site. It mimics a banking application. You can try it yourself at https://demo.applitools.com/.

The login page has a main icon, username and password fields, and a sign-in button. Since this is a demo site, you can enter any username or password to log in.

[Slide]

After clicking the sign-in button, the main page loads. There's a lot of stuff on the main page. The top bar has the name of the app, a search field, and icons for your account. The main part of the page shows financial data. The left sidebar shows different account types.

[Slide]

We could write a basic login test for this app in four steps:

1. Load the login page.
2. Verify that the login page loads correctly.
3. Log into the app.
4. Verify that the main page loads correctly.

This could be a smoke test. There's nothing fancy here. The trickiest part for automation would be deciding which elements to check on the loaded pages.

[Slide]

We could automate this test in Java using Selenium WebDriver. Technically, we could automate it using any popular language and tool. Personally, right now, I really like Python and Playwright, but based on different reports I've seen, Java is still one of the most popular languages for test automation, and Selenium WebDriver remains the most popular browser automation tool. JavaScript, C#, and Ruby are other popular languages for test automation, and Cypress is a very popular alternative to Selenium.

In Java, we can write a JUnit test class named "LoginTest" and create a test case method named "login". This test case method calls four helper methods – one for each step.

[Slide]

The first method, "load login page", loads the demo app's login page in the browser.

[Slide]

The second method, "verify login page", verifies the appearance of five critical elements on the page: the logo, the username field, the password field, the sign-in button, and the remember-me checkbox. It waits for each of these elements to appear using a helper method named "wait for appearance".

[Slide]

The third method, "perform login", enters a username and password, and then clicks the sign-in button. So far, so good. Nothing too bad.

[Slide]

The fourth method, "verify main page", is a doozy. Remember all the things on that page? Well, they'll need several assertions to verify. Some assertions merely check the appearance of elements.

Others need to perform text matching. For example, [Slide] to check the banner at the top that says, "Your nearest branch closes in X minutes," we need to find the element, get its text, and

then perform a regular expression match. [Slide] Checking the account types and status fields require getting lists of elements, mapping their text values, and transforming the resulting data for comparisons.

[Slide] Despite this heavy lifting, this page still doesn't check everything on the main page. Dates, amounts, and descriptions are all ignored as a risk-based tradeoff. We could add more assertions, but they would lengthen this method even more. They could also be difficult to write and become brittle over time. Tests can't cover everything.

[Slide]

If we run this login test against our web app, it should pass without a problem. But what if the page changes? [Slide] Here's a different version of the same page with some slight visual differences. Can you see what they are? Let me go back and forth a few times for you to see. [Slide back; Slide forward]. Will our login test still work? Will it pass or fail? *Should* it pass or fail?

[Slide]

Looking at these two pages side-by-side makes comparison easier. The logos are different, and the sign-in buttons are different. While I'd probably ask the developers about the sign-in button change, I'd categorically consider that logo change a bug. Unfortunately, as long as the page structure doesn't change, our login test will still pass. It wouldn't detect these changes. We probably wouldn't find out about these changes if we relied exclusively on traditional test automation.

[Slide]

The step to verify that the login page loaded correctly only checks for the appearance of five elements by locators. These assertions will pass as long as these locators find elements somewhere on the page, regardless of where they appear or what they look like.

[Slide]

Technically, this login page would still pass the test, even though we can clearly see it's broken. Traditional functional testing hinges on the most basic functionality of web pages. If it clicks, it works! It completely misses visuals. Those are *huge* test gaps. Adding more assertions probably wouldn't catch this kind of problem, either.

[Slide]

So, what if we *could* visually inspect this page? That would easily catch any changes on the page. We could take a baseline snapshot that we consider "good," and every time we run our tests, we take a new "checkpoint" snapshot. Then, we can compare the two side-by-side to detect any changes.

[Slide]

This is what we call *visual testing*. If a picture is worth a thousand words, then a snapshot is worth a thousand assertions. Automated visual testing is what Applitools does!

[Slide]

One visual snapshot captures *everything* on the page. As a tester, you don't need to explicitly state what to check: a snapshot implicitly covers layout, color, size, shape, and styling. That's a huge advantage over traditional functional test automation.

To be honest, testers have been doing visual testing since computers first had screens. Anyone can manually bang on a keyboard and look at the screen to see what changes. That's arguably the first kind of testing anyone does! It's super valuable to take a quick glance at a page to see what's wrong. Humans can intuitively judge if a page is good or bad in a few seconds.

[Slide]

Unfortunately, human reviews don't scale well. Modern apps have several screens worth checking, and Continuous Integration systems deploy changes multiple times a day. Humans make mistakes. They get tired. They miss things. They also have limited time.

This reminds me of the legend of John Henry, a folk hero from the United States. Has anyone heard the legend of John Henry? [Pause] [If anyone answers, ask them what it is.]

As the legend goes, John Henry was a railroad worker on the Great Bend Tunnel along the C&O Railway in West Virginia. When the railroad company brought in a steam drill, he competed against it head-to-head with a ten-pound hammer in each hand. He drilled deeper than the steam engine could, technically winning the contest, but he died from exhaustion afterwards. The legend of John Henry serves as a parable that even the strongest, sharpest human is inevitably no match for a machine.

[Slide]

To be relevant in a modern software shop, visual testing must be automated, but that's easier said than done. Programming a tool to capture snapshots and perform pixel-by-pixel comparisons isn't too difficult, but determining if those changes matter is. A good visual testing tool should ignore changes that don't matter – like small padding differences – and focus on changes that do matter. Otherwise, human testers will need to review every single result, nullifying any benefit of automating visual tests.

Take a look at these two pictures. They show a cute underwater scene. There are a total of ten differences between the two pictures. Can you find them? I'll give you a few seconds to look. [Pause]

[Slide]

Unfortunately, a pixel-to-pixel comparison won't find any of them. I ran these two pictures through Applitools using an exact pixel-to-pixel comparison, and this is what happened. Except for the whitespace on the sides, every pixel was different. As humans, we can clearly see that these images are very similar, but because they were a few pixels off on the sides, automation failed to pinpoint meaningful differences.

[Slide]

This is where AI really helps. Applitools uses Visual AI to detect meaningful changes that humans would see and ignore inconsequential differences that just make noise. Here, I used Applitools' "strict" comparison, which pinpointed each of the ten differences. Take a look. Did you find all ten yourself? Do you see any that you missed? I'll pause a moment for y'all to look. [Pause]

[Slide]

That's the second advantage of good automated visual testing: Visual AI, like what Applitools does, focuses on meaningful changes to avoid noise. Visual test results shouldn't waste testers' time over small pixel shifts or things a human wouldn't even notice. They should highlight what matters, like missing elements, different colors, or skewed layouts. Visual AI is a differentiator for Applitools's brand of visual testing. Not all visual testing tools rise above pixel-to-pixel comparisons.

[Slide]

So, let's update our login test to do visual testing with Applitools. First, we need to create an Applitools account. Anyone can create one for free at the link I'm showing here. You can use your GitHub account or an email address, and you don't need to enter a credit card.

The account will come with an API key that must be set as an environment variable for testing.

[Slide]

Next, we need to add the Applitools Eyes SDK to our project. Since we are using Selenium WebDriver with Java, we need to add the "com.applitools.eyes-selenium-java3" Maven dependency to our pom.xml file.

[Slide]

Then, we need to set up visual testing objects in a test setup method. We will still use the WebDriver object, but we also need a VisualGridRunner to upload snapshots to Applitools Ultrafast Grid, and we need an Eyes object to take visual snapshots during test execution. The Configuration object sets the API key, the test batch name, and the browsers we want to test.

[Slide]

The steps for our login test are the same, but before and after the test steps, we must “open” and “close” Applitools Eyes so that we can take snapshots during the test. Opening requires the WebDriver reference, the app name, and the test name.

[Slide]

The interaction methods can remain unchanged, but we need to update the verification methods. Traditional assertions can be completely replaced by one-line snapshot calls. Take a look at “load login page”: five lines reduced to one, and the visual snapshot technically has far greater coverage.

[Slide]

The impact on “verify main page” is far greater. One visual snapshot eliminates the need for several lines of assertions.

[Slide]

This is the third major advantage visual testing has over traditional functional testing: visual snapshots greatly simplify assertions. Instead of spending hours deciding what to check, figuring out locators, and writing transformation logic, you can make one concise snapshot call and be done. As an engineer myself, I cannot understate the cognitive load this removes from the automation coding process. I said it before, and I’ll say it again: If a picture is worth a thousand words, then a snapshot is worth a thousand assertions.

[Slide]

So, what about cross-browser and cross-device testing? It’s great if my app works on my machine, but it also needs to work on everyone else’s machine. The major browsers these days are Chrome, Edge, Firefox, and Safari. The two main mobile platforms are iOS and Android. That might not sound like too much hassle at first, but then consider:

- All the versions of each browser – typically, you want to verify that your app works on the last two or three releases
- All the screen sizes – modern web apps have responsive designs that change based on viewport

- All the device types – desktops and laptops have various operating systems, and phones and tablets come in a plethora of models

We have a combinatorial explosion! Traditional functional tests must be run start-to-finish in their entirety on each of these platforms. Most teams will pick a few of the most popular combinations to test and skip the rest, but that could still require lots of test execution.

[Slide]

Visual testing simplifies things here, too. We already know that visual testing captures snapshots of pages in our applications to look for differences over time. Note how I used the word “snapshot” and not “screenshot.” That was deliberate. A *screenshot* is merely a rasterized capture of pixels reflecting an instantaneous view. It’s frozen in time and in size. A *snapshot*, however, captures everything that makes up the page: the HTML structure, the CSS styling, and the JavaScript code that brings it to life.

Snapshots are more powerful than screenshots because snapshots can be rerendered. For example, I could run my test one time on my local machine using Google Chrome, and then I could re-render any snapshots I capture from that test on Firefox, Safari, or Edge. I wouldn’t need to run the test from start to finish three more times – I just need to rerender the snapshots in the new browsers and run the Visual AI checker. I could re-render them using different versions and screen sizes, too, because I have the full page, not just a flat screenshot. This works for web apps as well as mobile apps.

[Slide]

Visually-based cross-platform testing is lightning fast. A typical UI test case takes about a minute to run. It could be more or less, but from my experience, 1 minute is a rough industry average. A visual checkpoint backed by Visual AI takes only a few seconds to complete. Do the math: if you have a large test suite with hundreds to thousands of tests that you need to test across multiple configurations, then visual testing could save you hours, if not days, of test execution time per cycle. Plus, if you use a service like Applitools Ultrafast Test Cloud, then you won’t need to set up all those different configurations yourself. You’ll spend less time and money on your full test efforts.

[Slide]

I’ve made a lot of bold claims about visual testing, so let’s see it in action!

[Screen sharing]

[Off-script:
1. Show the Java test project
2. Run the visual test once to establish baselines

3. Run it again to show passing tests
4. Change the URL to point to the broken site
5. Run the test again to show visual differences
6. Show the differences and thumbs-down
7. Explain cross-browser testing with the Applitools Ultrafast Grid
8. Explain snapshots-not-screenshots
9. Show the code to test any browser
10. Rerun the tests

]

[Slide]

Before I conclude this talk, there is one more thing I want y'all to consider: when a team should adopt visual testing. I can't tell you how many times folks have told me, "Andy, that visual testing thing looks soooo cool and sooooo helpful, but I don't think my team will ever get there. We're just getting started, and we're new to automation, and automation is so hard, and I don't think we'll ever be mature enough to use a tool like Applitools." I just smack myself in the face because visual testing makes automation easier!

I really think teams should do visual testing from the start. Consider this strategy: start by automating a smoke test that navigates to different pages of an app and captures snapshots of each. The interaction code would be straightforward, and the snapshots are one-liners. That would provide an immense amount of value for relatively little automation work. It's the 80/20 rule: 80% of the value for 20% of the work. Then, later, when the team has more time or more maturity, they can expand the automation project with larger tests that use both traditional and visual assertions.

[Slide]

Test automation is hard, no matter what tool or what language you use. Teams struggle to automate tests in time and to keep them running. Visual testing simplifies implementation and execution while catching more problems. It offers the advantage of making functional testing easier. It's not a technique only for those on the bleeding edge. It's here today, and it's accessible to anyone doing test automation.

[Slide]

Visual testing is a winning strategy. It has several advantages over traditional functional testing. Please note, however, that visual testing does not *replace* functional testing. Instead, it *supercharges* it. With Applitools Eyes, you can do visual testing in any major language or test framework you like, and with Applitools Ultrafast Grid, you can do visual testing using any major browser configuration. If you want to give it a try, you can sign up for a free account. You can also clone the example project I showed today to try running tests yourself.

By the way, this is the best slide for taking a picture. I'll wait a moment. [Pause]

[Slide]

Thank you all for attending my talk today. Again, my name is Pandy Knight, and I'm the Automation Panda. Be sure to read my blog and follow me on Twitter. Check out Test Automation University to learn just about anything about testing, and give visual testing a try. Thanks!