User Interface Testing

This section is primarily geared towards web-based UIs, but the guidance is similar for mobile and OS based applications.

Applicability

UI Testing is not always going to be applicable, for example applications without a UI or parts of an application that require no human interaction. In those cases unit, functional and integration/e2e testing would be the primary means. UI Testing is going to be mainly applicable when dealing with a public facing UI that is used in a diverse environment or in a mission critical UI that requires higher fidelity. With something like an admin UI that is used by just a handful of people, UI Testing is still valuable but not as high priority.

Goals

UI testing provides the ability to ensure that users have a consistent visual user experience across a variety of means of access and that the user interaction is consistent with the function requirements.

- Ensure the UI appearance and interaction satisfy the functional and non-functional requirements
- Detect changes in the UI both across devices and delivery platforms and between code changes
- Provide confidence to designers and developers the user experience is consistent
- Support fast code evolution and refactoring while reducing the risk of regressions

Evidence and Measures

Integrating UI Tests in to your CI/CD is necessary but more challenging than unit tests. The increased challenge is that UI tests either need to run in headless mode with something like Puppeteer or there needs to be more extensive orchestration with Azure DevOps or GitHub that would handle the full testing integration for you like BrowserStack

Integrations like `BrowserStack` are nice since they provide Azure DevOps reports as part of the test run.

That said, Azure DevOps supports a variety of test adapters, so you can use any UI Testing framework that supports outputting the test results to one of the output formats listed at Publish Test Results task.

If you're using an Azure DevOps pipeline to run UI tests, consider using a self hosted agent in order to manage framework versions and avoid unexpected updates.

General Guidance

The scope of UI testing should be strategic. UI tests can take a significant amount of time to both implement and run, and it's challenging to test every type of user interaction in a production application due to the large number of possible interactions.

Designing the UI tests around the functional tests makes sense. For example, given an input form, a UI test would ensure that the visual representation is consistent across devices, is accessible and easy to interact with, and is consistent across code changes.

UI Tests will catch 'runtime' bugs that unit and functional tests won't. For example if the submit button for an input form is rendered but not clickable due to a positioning bug in the UI, then this could be considered a runtime bug that would not have been caught by unit or functional tests.

UI Tests can run on mock data or snapshots of production data, like in QA or staging.

Writing Tests

Good UI tests follow a few general principles:

- Choose a UI testing framework that enables quick feedback and is easy to use
- Design the UI to be easily testable. For example, add CSS selectors or set the id on elements in a web page to allow easier selecting.
- Test on all primary devices that the user uses, don't just test on a single device or OS.
- When a test mutates data ensure that data is created on demand and cleaned up after. The consequence of not doing this would be inconsistent testing.

Common Issues

UI Testing can get very challenging at the lower level, especially with a testing framework like Selenium. If you choose to go this route, then you'll likely encounter timeouts, missing elements, and you'll have significant friction with the testing framework itself. Due to many issues with UI testing there have been a number of free and paid solutions that help alleviate certain issues with frameworks like Selenium. This is why you'll find Cypress in the recommended frameworks as it solves many of the known issues with Selenium.

This is an important point though. Depending on the UI testing framework you choose will result in either a smoother test creation experience, or a very frustrating and time-consuming one. If you were to choose just Selenium the development costs and time costs would likely be very high. It's better to use either a framework built on top of Selenium or one that attempts to solve many of the problems with something like Selenium.

Note there that there are further considerations as when running in headless mode the UI can render differently than what you may see on your development machine, particularly with web applications. Furthermore, note that when rendering in different page dimensions elements may disappear on the page due to CSS rules, therefore not be selectable by certain frameworks with default options out of the box. All of these issues can be resolved and worked around, but the rendering demonstrates another particular challenge of UI testing.

Specific Guidance

Recommended testing frameworks:

- Web
  - BrowserStack
  - Cypress
  - Jest
  - Selenium
  - Appium
- OS/Mobile Applications
  - Coded UI tests (CUITs)
  - Xamarin.UITest
  - BrowserStack
  - Appium

> Note that the framework listed above that is paid is BrowserStack, it's listed as it's an industry standard, the rest are open source and free.

Last update: March 20, 2024