A Cypress tree grows in FNZ: End-to-end testing automation framework considerations

Charles Bruderer Wise
Master of Software Development degree candidate
Victoria University of Wellington - Te Herenga Waka
Wellington, New Zealand
charles@charlesbwise.com

***Abstract*—Creating successful frameworks for automating end-to-end test cases of web applications requires careful consideration of maintenance, reliability, and reporting. While writing high-quality end-to-end test cases is an important part of the automation battle, how successful these tests are at coping with changes to user interfaces and web application elements owes more to the choice of testing framework than simply to good behaviour-driven development (BDD) practices.**

Seeking alternatives to their current in-house automation testing framework based around the Selenium WebDriver, financial services and technology company FNZ proposed a proof of concept project for six master's students to use the Cypress JavaScript end-to-end testing framework to automate certain manual "misuse" tests, integrate the framework into their existing continuous integration/continuous delivery (CI/CD) pipelines, and potentially replace all or part of the legacy testing framework with regression test suites written in Cypress. The project ran for a five-sprint development cycle and produced an automated misuse testing suite written in Cypress and capable of being deployed through existing CI/CD pipelines. Writing more complex integration and end-to-end tests in Cypress proved to be difficult, however, owing to the complexity, brittleness, and flakiness of existing tests within the in-house framework. Additional Cypress limitations were identified around programming language and browser support. Finally, the in-house testing framework was critically assessed and some suggestions for improvements put forward.

***Index Terms*—Automated Software Testing, FNZ, Selenium WebDriver, Cypress, End-to-End Testing, Web Application Testing**

I. Introduction

Web applications are a ubiquitous part of daily financial life. Whether banking, commerce, or securities, browser-based platforms have become an essential part of transacting personal and professional business. Considering the large role that financial web applications play in the lives and fortunes of their end-users and the regulatory scrutiny they inspire, end-to-end testing of these web applications is crucial for ensuring their ongoing stability and security [1]. End-to-end testing is based on the scenario testing approach [2] [1], where a scenario is a story that describes a particular way the system might be used.

Given the complexity and importance of web application end-to-end testing, organisations have been particularly keen to automate these tests so as to reap the benefits of reusability, repeatability, and efficiency [3]. Surveys of grey literature have found that the best practices for developing high-quality end-to-end tests are writing test code of high structural quality, making use of continuous integration/continuous delivery (CI/CD) pipelines, engaging in good design patterns, and crafting robust GUI element locators [4].

At FNZ, a financial services and technology company that builds and provides web-based investment platforms, testing automation is haphazardly implemented through an in-house testing framework called Kratos that uses the Selenium WebDriver[1]. The framework has several drawbacks, however, such as high maintenance costs, tests with non-determinism problems (more colloquially known as "flakiness"), and slowness due to build and compilation times and Selenium WebDriver design limitations.

To address these issues, FNZ proposed a web testing automation proof of concept project for six master's of software development students at Victoria University of Wellington - Te Herenga Waka using Cypress[2], a newer JavaScript-based end-to-end testing framework. Broadly, FNZ wanted to answer the following questions about automation testing within the company:

- *Is Cypress suitable for automating certain manual "misuse" tests?*
- *Is Cypress capable of being run as part of a CI/CD pipeline?*
- *Could Cypress be used to make regression testing a regular, automated process?*

To answer these questions, this paper introduces FNZ and its development and automation testing landscape in Section II, and then introduces the Cypress proof of concept project and its outcomes in Section III. These outcomes are analysed and discussed in Section IV before reflecting on the final results in Section V.

II. Industry Placement Overview

A. FNZ

FNZ is a financial services and technology company that creates and manages investor platforms for financial institutions. The company was founded in 2003 by Adrian Durham in Wellington, New Zealand [5]. Durham, who had been working at Credit Suisse NZ as an analyst since 1996 [6], recounts that he was inspired to start the company after his parents emerged from a meeting with a financial planner with an incomprehensible contract stipulating a high management fee [7]. The company expanded into the United Kingdom (UK) in 2005 [8], and even moved its headquarters to Edinburgh [6].

Amidst the Great Financial Crisis, the company was sold to the private equity group HIG Europe for NZ$34 million in 2009, with the current management retaining a controlling

[1] https://www.selenium.dev/documentation/webdriver/
[2] https://www.cypress.io/

interest [9]. Expansion into Australia began during the same year, followed by the Czech Republic in 2010, Singapore in 2015, China in 2016, and most recently the United States of America (USA) in 2021 [8].

In 2018, Canada's second-largest pension fund Caisse de dépôt et placement du Québec (CDPQ) and the Anglo-American private equity group Generation Investment Management LLP (Generation) purchased in partnership a majority stake in the company for £1.65 billion [10]. Further investment followed from Temasek, a Singaporean state-owned holding company, in 2020 [11], and a record US$1.4 billion investment from the Canada Pension Plan Investment Board (CPP Investments) and Motive Partners in 2022 [12].

At present, the company has over US$1.5 trillion assets under administration (AUM). Its customers include over 650 large financial institutions and over 8,000 wealth management firms spanning 21 countries [13].

B. Service Platforms

FNZ offers four different propositions that allow customers to target investors with different levels of wealth and financial goals [14]:

- Retail Direct - Known as D2C (direct-to-consumer), these platforms allow clients to offer investments directly to consumers who wish to make their own investment decisions. Common tools include robo-advisers and risk questionnaires.
- Retail Advised - Aimed at retail financial services organisations who want to offer a platform to financial advisers, who in turn will use the platform to manage their customers' portfolios and information. These customers have read-only access to their accounts, and all transactions are made by the advisers. Common tools include reporting, client management, task automation, and fee calculation.
- Wealth - Platforms offered to discretionary wealth managers, private banks, and stockbrokers who typically deal with high net worth individuals who demand a more bespoke portfolio tailored to achieve certain investment strategies. Common tools include reporting and discretionary portfolio model generation and maintenance.
- Corporate - Allows companies to offer their employees financial instruments and may incorporate other financial benefits that an employer may provide to their employees (e.g., insurance, vouchers, travel passes). The platform provides access to all of these accounts in one place. Common tools include viewing multiple accounts, forecasting, and bulk actions.

The core of the FNZ service platform is FNZ One, which underpins almost all of the company's propositions . FNZ One consists of client, account, and portfolio management services for a comprehensive range of asset and liability types, and acts as a unified customer asset register [15]. It is also designed to integrate with customers' existing systems and other FNZ platforms. Discretionary investment management services are provided by the FNZ X-Hub service, which makes it possible to adjust investment propositions for client-specific constraints and preferences such as capital gains taxation, sustainability, risk, and asset allocation.

FNZ makes ample use of the .NET framework. The standard codebase for FNZ One is called FNZ One X, and is written in C# and Visual Basic for generating ASP.NET web pages. The core One X codebase consists of typical global investor management tools, with features for specific jurisdictions and customers implemented as separate extensions. All projects in One X share the same branch - there are no distinct branches for customers. The goal of this architecture decision is to reunite the disparate elements of the legacy FNZ One codebase into a common code architecture that promotes reusability and rigour.

The outcome of this decision is that an increasing number of customer projects are implemented in the One X codebase. Each of these projects is run by a customer delivery team, which has its own developers, testers, and business analysts [16]. Architecturally, the codebase is broken down into globally useful core code, which is highly configurable and changes infrequently. The core code does not know about customer projects; each customer project has its own bootstrapper/installer in the core code that references code specific to jurisdictions (e.g. United Kingdom, Germany) and customer-specific user interfaces (UIs) and application programming interfaces (APIs). The bootstrapper in the core code places all these references into a single container, which becomes a deployment package specific to each project that builds all relevant code and fetches all required libraries [16].

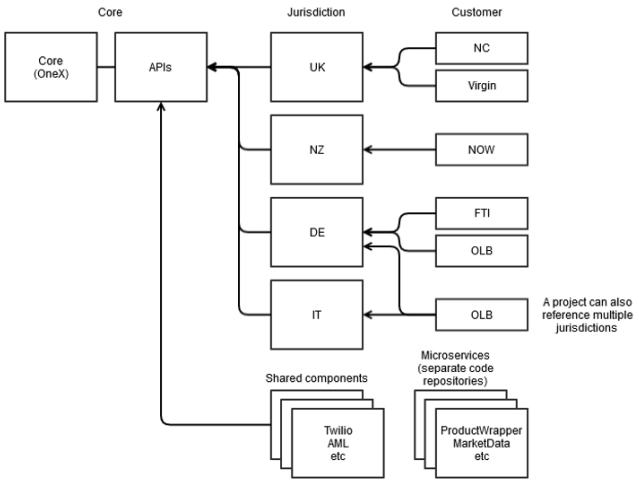

Fig. 1. A high-level diagram of the FNZ One X codebase architecture [16].

C. Test Automation

One of the challenges of the FNZ development landscape is that there are numerous customer platforms that share a lot of functionality, but potentially have completely different UIs and slight processing variations [17]. For example, most of FNZ's Australian clients provide superannuation services to their customers, but offer distinct web portals with client branding and other customisations. While the One X codebase is the future direction of customer on-boarding at FNZ, this still leaves the question of how to efficiently and effectively write tests for these differentiated platforms that have some degree of shared functionality. The answer is Kratos, an in-house framework for test automation written in C#.

Kratos is intended to maximise code reusability by splitting the codebase up into five main levels [18]:

1) Kratos Core library - a separate repository of high-level code consisting of generic functionality, base

classes, and interfaces. Key features include classes for constructing randomised test data (e.g., names and addresses) and generic UI navigation tasks such as selecting an item from a drop-down menu.
2) Systems - generic implementations of code at a systems level (e.g., One X, XHub, databases) including page groups and objects, validators, test data builders, and features representing key system functions (e.g., user administration, deposits).
3) Platforms - similar to the Systems level, but pertaining to specific clients. Systems level functionality can be overridden at this level if needed.
4) Test Apps - contain specific tests as feature files and their related step definitions/glue code [19]. It is also possible to specify which environment tests should be run in (e.g., testing, user acceptance).
5) Unit Tests - tests specific to the functionality of the Kratos framework

The heart of Kratos is at the Platforms level, where behaviour-driven development (BDD) [19], [20] tests exist as feature files. BDD envisions writing acceptance tests [21] as easily readable examples, allowing for feedback and collaboration from non-technical stakeholders such as business analysts and testers. To accomplish this, feature files in Kratos are written in Gherkin, which is human-readable syntax that documents examples of behaviour for testing. Each feature file is made up of a collection of scenarios, which define a flow of events. Individual events are defined as steps, which use familiar BDD keywords such as “given”, “when”, “then”, etc., to define and document the action being tested in a clear and easily understandable form. Test data is supplied in a table within the file, and the scenario is run against each row therein.

```
Feature: Prevent unauthenticated access to
    restricted pages

  An unauthenticated user should be prevented
      from accessing a restricted page

  Scenario: An unauthenticated user attempts to
       access a restricted page

    Given the user is not authenticated

    When the user tries to access a
        <restrictedPage>

    Then the user should be redirected to an
        <accessDeniedPage>

    Examples:
      | restrictedPage | accessDeniedPage    |
      | secrets.aspx   | access-denied.aspx |
```

Fig. 2. An example of a BDD feature file demonstrating Gherkin syntax with examples.

Each step in a BDD scenario is linked to a step definition, which hold the actual code to execute the step event. In Kratos, these step definitions are C# methods.

Kratos uses the the SpecFlow[3] BDD framework, which is an open source .NET port of Cucumber[4]. Kratos manages

[3]https://specflow.org/
[4]https://cucumber.io/

```
[When(@"the user tries to access a (.*)")]
public void WhenTheUserTriesToAccessA(
    string restrictedPage)
{
  PageObject.GoTo(restrictedPage);
}
```

Fig. 3. A pseudo-code example of a step definition underpinning the ”When” statement from Fig. 2.

tests using SpecFlow hooks, which define events to run before and after scenarios and steps such as obtaining login credentials. Kratos manages all data relating to environments in its own dedicated database, which are determined by environment code per client at the Test App level.

For steps that involve assertions (the ”Then” step) against web elements, Kratos makes use of the Selenium WebDriver. Selenium is an open source testing framework that was first released in 2004. The WebDriver component was released in 2008, and makes use of a browser-dependent binary file to implement native automation support. Selenium identifies web elements using a locator strategy, which can look for elements using attributes ids, CSS (Cascading Style Sheets) selectors, or XPath (XML Path Language) queries [22]. Database assertions are performed using a database validator native to Kratos.

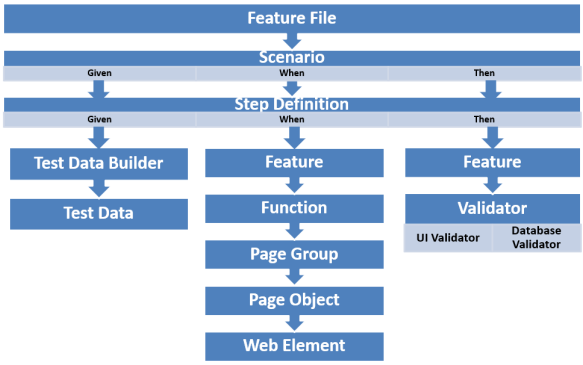

Fig. 4. A high-level summary of the different generic methods, interfaces, objects, features, and validators used to prepare, execute, and assert BDD test cases in Kratos [17].

From an infrastructure perspective, Kratos is the source control part of a full testing automation framework. The Kratos codebase is a git repository hosted by Microsoft Azure DevOps under the auspices of the Global Automation team. Although it technically has all the BDD feature files needed to run tests, test case management and reporting is managed through the Atlassian issue tracking platform Jira. Tests are explained in Jira as both manual instructions and copies of the BDD feature files. Continuous integration and deployment (CI/CD) of tests as part of test sets or plans is handled by TeamCity[5], which is a JetBrains CI/CD product that uses build agents for pipelines. As most FNZ codebases are still running .NET 4.7.2 or less, the build agents are all Microsoft Windows Server virtual machines. Within TeamCity, all clients live under the ”KratosForAll” project and have environment sub-projects that run the following build steps during each deployment:

[5]https://www.jetbrains.com/teamcity/

1) Pull the latest commits from Kratos
2) Download all BDD test case features from Jira using the in-house JiraExtractor tool
3) Re-build the Kratos code
4) Run the tests

During the development process, analyst developers work on the Kratos codebase and merge reviewed pull requests to the remote repository on Azure DevOps. For each deployment, TeamCity then selects a suitable build agent based on the environment sub-project configuration and uses that virtual machine to complete all the steps above. Test statuses are reported back to a parent Jira issue and within each build in TeamCity. Test failures are reported in TeamCity through GUI screen-shots, which are called artefacts, and in a verbose build log that includes stack traces.

D. Unit and Misuse Testing

FNZ promotes unit test creation as part of the development process. Unit tests accompany code in specific codebases, with an expectation of a minimum of 10 percent code coverage. Analyst developer on-boarding and training includes a substantial unit testing module consisting of lecture material as well as a complete training environment with senior oversight. Unit test classes generally extend from a specific base class of the code under test (CUT) and rely on the .NET Framework tool xUnit[6]. Test methods begin by declaring the CUT instance and instantiating an instance of it, potentially with mocked or stubbed data using Rhino Mocks[7], and invoking it to produce a result. Next, one writes observations that cover specific testing scenarios, which are methods that validate expected results against the actual results object, and continues until all scenarios and their observations cover all aspects of the CUT [23].

Part of the acceptance criteria for platform releases is misuse testing, which depending on the platform and/or client may or may not be partially automated through Kratos. Misuse testing is specifically about identifying security issues by sending unauthorised or inappropriate requests to certain platform web pages and forms and verifying that they are correctly handled. This can take the form of user session destruction, re-direction to ”access denied” pages, or other warning messages. For example, platforms should prevent end-users from successfully uploading files with unauthorised extensions such as *.exe*, as this could result in the execution of unauthorised code within the back-end.

On some platforms, certain misuse test cases have already been automated as test suites within Kratos written as BDD feature files with corresponding step definitions. In many cases, though, the majority of misuse tests must be run manually by analyst testers within a project using security testing tools such as Burp Suite[8] to send malformed requests to lists of platform URIs. This is an inefficient and time-consuming process that significantly slows down releases and suffers from a lack of consistency as far as what is being tested and how. The FNZ Information Security team have attempted to document the manual steps to run each test within internal wikis and Jira [24], but there are still significant drawbacks to this approach.

[6]https://xunit.net/
[7]https://hibernatingrhinos.com/oss/rhino-mocks
[8]https://portswigger.net/burp

III. Industry Project

FNZ employed six master's of software development students from Victoria University of Wellington - Te Herenga Waka as intern analyst developers to produce a proof of concept for performing web application GUI testing using the Cypress[9] framework. FNZ's Head of Development for Australia and New Zealand oversaw the project, which lasted for 12 weeks. The internship fulfilled the industry placement requirement of the software engineering (SWEN) 589 course ”Industry Research and Development Project”, in which all six students were enrolled.

A. Cypress

Cypress is a JavaScript end-to-end testing framework following a client-server architecture that implements tests inside a browser [22]. Its goal is to make it easy to write unit, integration, and end-to-end tests for web applications by providing a simpler alternative to more complex testing frameworks [26], and is predicated on the idea that aversion to testing is caused by the limitations of previous solutions such as Selenium [25]. Cypress consists of a locally installed open source test runner that includes Mocha[10] BDD syntax, the Chai[11] assertions library, and an Electron[12] browser based on Chromium[13], and a commercial dashboard [27]. The test runner can be executed as a GUI or headlessly via the command line.

Cypress differs from Selenium in several key ways. Firstly, Cypress tests are written in JavaScript, and the framework's functionality can be extended using plugins that interact with the Cypress API, whereas Selenium supports all major development languages [26]. While Selenium's WebDriver interacts with the Document Object Model (DOM) using JSON messages over HTTP (now the W3C WebDriver standard) [22] and requires a different driver for each browser type, Cypress uses a custom universal driver that works directly in the browser, meaning it can intercept commands directly [26]. However, unlike Selenium, Cypress currently lacks support for WebKit[14]-based browsers such as Safari and cannot interact with multiple browser tabs [26].

B. Project Organisation and Implementation

The main goals of the Cypress proof of concept project were:

- Address problems with the Kratos test automation framework, specifically:
  - Difficulty of maintenance due to size and complexity
  - Increasing compilation times with the addition of more tests
  - Slow performance of the Selenium WebDriver
- Automate manual misuse testing
- Integrate Cypress into TeamCity CI/CD processes
- Automate full regression testing

The project group formed a self-organising team with a scrum master and agreed to treat the FNZ project overseer

[9]https://www.cypress.io/
[10]https://mochajs.org/
[11]https://www.chaijs.com/
[12]https://www.electronjs.org/
[13]https://www.chromium.org/Home/
[14]https://webkit.org/

as a product owner. FNZ provided each team member with a laptop and a local virtual machine for development, and set up a git repository for the project within Azure DevOps. At the request of the team, the product owner modified the repository to prevent commits being made directly to the master branch and require pull request reviews prior to merging. The team also agreed to use Atlassian Trello as a Kanban board and sprint planning tool.

The team held daily stand-ups via Microsoft Teams or in-person at the FNZ offices in Wellington, New Zealand, and scheduled regular afternoon meetings with the product owner on Tuesday and Friday. Based on this schedule, the team agreed to end sprints on Tuesday morning so that working demonstrations could be presented at the meeting in the afternoon. Of the five sprints the team did, sprint retrospectives were completed for all but the fourth one, which was not done due to time pressure to complete work on TeamCity integration.

Sprint	***Overview***
1	Project kick-off, focus on reusability by creating custom fixture files for clients, start writing misuse tests in Cypress for a client platform in the user acceptance testing (UAT) environment
2	Finish implementing all unit tests in Cypress, exploration of using Cucumber feature files in Gherkin syntax, start writing some simple end-to-end tests in Cypress
3	Key decision to develop using feature files with step definitions going forward, addition of a second client platform in a different environment, shift away from fixtures towards custom configurations on a per-client basis
4	Clean-up feature file descriptions and language, continue trying to implement certain end-to-end tests in Cypress, focus on reusability of step definitions between clients
5	Implement all misuse tests for the FNZ One X platform, create PowerShell wrapper for TeamCity usage, create TeamCity project and write build steps

Fig. 5. A table summarising the project sprints.

The team faced some challenges initially around installing Cypress on the provided virtual machines due to proxy settings in place within FNZ. The solution was to use a ZIP archive copy of Cypress version 9.5.4 as the installation source by setting the environment variable `CYPRESS_INSTALL_BINARY` to its location. and to register FNZ's custom Node.js repository with the Node.js package manager (npm) so that other dependencies could be installed. Although Cypress is available as a Node.js package, it was not possible to directly install it from the FNZ Node.js repository because the package is just a wrapper around a script that attempts to download the file from an external location, which was prevented by the proxy settings. Once installed, Cypress was invoked from the command line in either a GUI or in headless mode.

GUI:
```
npx cypress open
```

Headless:
```
npx cypress run
```

During the first two sprints, the team focused on getting acquainted with Cypress by writing misuse tests exclusively as JavaScript files for a single client platform. Cypress fixtures were used to store static data such as environment URLs and login credentials in JSON format. Some members of the team also started implementing simple integration tests after being urged to do so by the product owner, and other team members started looking at creating tests as Cucumber feature files written in Gherkin syntax using the cypress-cucumber-preprocessor[15] Node.js package.

During the third sprint, the rest of the team converted their existing Cypress tests into Cucumber feature files referencing JavaScript step definitions. The team also considered whether it made sense to fully embrace this practice for future tests. One consideration was whether it was possible to separate out the "Examples" data from the feature file for greater reusability, but it was discovered that this is strongly discouraged by BDD practitioners, as it defeats the purpose of the feature files serving as complete BDD test cases [28]. After an extraordinary meeting, the team concluded that Cucumber feature files with step definitions were better from a testing readability and documentation standpoint, and could be designed so that step definitions were reusable across multiple client platforms.

Having decided to proceed using Cucumber feature files, the fourth sprint involved fixing reusability issues with step definitions after adding support for misuse testing of a second client platform. The biggest issue was how to provide context for Cypress to run tests on different platforms with different URLs, login credentials, and GUI differences. The solution was to convert the existing JSON fixture files into client-specific Cypress JSON configuration files[16] , making it possible to change important parameters like the base URL for page navigation. Another challenge of working with a second client was that some common steps (see Fig. 6) such as authenticating via POST request stopped working due to technical differences, requiring a code refactor to get them working across both platforms.

For the fifth sprint, the team divided into two smaller groups, with one group working on adding misuse test support for the FNZ One X platform, and the other group working on the TeamCity integration. The latter faced numerous challenges around how to configure the Cypress project within TeamCity, eventually settling on an environment-based sub-project structure similar to Kratos. To simplify the execution steps in the TeamCity environment, the command line invocations for each client and environment were wrapped in a PowerShell script (see Fig. 6), which took significant inspiration from the FNZ One X build script method. Additionally, none of the existing TeamCity build agents had PowerShell Core or Node.js installed, requiring a manual installation process on multiple Windows virtual machines to make agents available for running tests.

C. Outcomes

At the conclusion of the fifth sprint, the project had achieved the following milestones:

- Automation of all misuse test cases in Cypress across three different client platforms

[15] https://github.com/badeball/cypress-cucumber-preprocessor
[16] Cypress 10.0 removed support for JSON configuration files

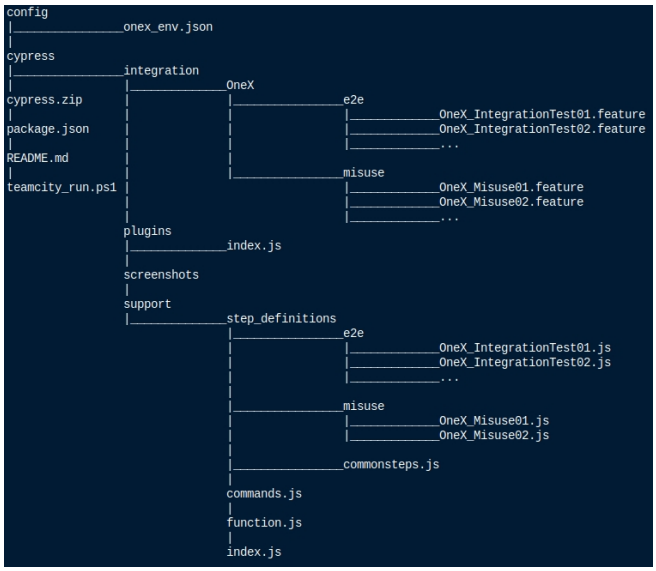

Fig. 6. The FNZ One X Cypress tests directory structure. Within TeamCity, the testing suite build is invoked via PowerShell:
`.\teamcity_run.ps1 setup`
This sets `CYPRESS_INSTALL_BINARY` to the included `cypress.zip` archive. Test execution runs Cypress headlessly and uses the following arguments, which in this case are mapped to the `onex_env.json` configuration file and the `misuse` suite:
`.\teamcity_run.ps1 run onex misuse env`
During test runs, artefacts are stored in the `screenshots` directory and made available through the TeamCity dashboard.

- Creation of some proof of concept Cypress integration and end-to-end tests
- Integration of the misuse test pack into TeamCity for each client environment

Within a week of finishing the final sprint, the team presented the project to an audience of FNZ developers, testers, and Global Automation team members. The overall reception of the project was largely positive, with particular interest shown in the advantages in using Cypress for web application GUI testing relative to Selenium and the wider availability of Cypress as a testing suite relative to an in-house framework such as Kratos. Concerns were raised about Cypress's lack of support for WebKit-based browsers and multiple tabs, as well as the non-trivial task of writing full integration and end-to-end tests in Cypress. The product owner articulated a vision of potentially using Cypress in conjunction with a separate test database instance for each client platform, allowing for smaller unit tests and integration tests to replace the complex end-to-end tests typical in Kratos, but acknowledged that the project would most likely complement Kratos going forward instead of seeking to completely replace it. It is expected that ongoing work on the project will be directed by interest from client teams in adopting it for specific testing goals and will eventually be placed under the control of the Global Automation team.

Referring to the original project goals, the team presented a testing framework proof of concept that addressed a number of issues presented by the existing Kratos testing framework, specifically ease of maintenance due to the nature of the misuse test cases and unencumbered by compilation concerns due to JavaScript being an interpreted language. As a result, the Cypress browser tests were much faster to complete than their comparable Kratos integration tests and demonstrated less flakiness overall. The project did reveal that replacing complex integration and end-to-end tests using Cypress would not be a trivial task, as the team's attempts to write these tests often encountered obstacles such as needing to prepare complex objects to fulfill the existing assertions using a different language than the original code. Therefore, the project did not achieve the goal of automating regression testing of client platforms. However, the project did satisfactorily demonstrate that misuse testing could be successfully automated within a resilient framework with reusability across multiple client platforms, and that these test packs could be successfully run via TeamCity build agents as part of a CI/CD pipeline.

IV. Analysis and Discussion

A. Testing Automation

The benefits of automating manual tests such as the FNZ misuse cases are well documented in academic literature. In their systematic literature review (SLR) of the benefits and limitations of automated software testing, Rafi *et al.* [3] reported that their practitioner survey found the main benefits thereof to be reusability, repeatability, and effort saved, particularly in regards to repeated regression testing. The latter point was also cited by Ricca *et al.* [29] as a key benefit of adopting test automation, as well as identifying bugs during early stages of development. While the FNZ misuse testing by itself cannot be classified as a full regression test suite, it is nevertheless treated as such by the business and is part of the acceptance criteria for the new release of a client platform. This points to another clear advantage of automated testing, which is the ability to scale. As Badal and Grünler [30] note, paying employees to manually test something repeatedly is not only expensive but a waste of resources. Prior to beginning the Cypress proof of concept project, the team was asked to assist with manual misuse testing of a client platform in a UAT platform. The process took almost a week to complete and was frustrating to implement given the sheer amount of repetition involved. A fully automated misuse testing pack initiated through a CI/CD pipeline would have taken approximately 30-60 minutes to run, and any follow-up tasks or bugs could have been easily distributed to the existing team to address well within the expected delivery time-frame.

In the context of direct web application GUI testing, aspects of the Cypress framework also contradict some of the limitations cited by Rafi *et al.* [3], specifically the high initial cost involved in designing test cases. Through the use of Cucumber feature files, the Cypress framework would be instantly recognizable to analyst testers at FNZ given that the Kratos framework also uses feature files via the .NET port of Cucumber SpecFlow. Additionally, prospective employees of FNZ may also already have experience with using Cypress and/or Cucumber to write tests, and the interpreted nature of JavaScript as a programming language suggests a lower barrier to entry from a development perspective compared to the object-oriented knowledge required to work with C#. However, further adoption of Cypress beyond misuse testing starts to stray into high initial cost territory, specifically because Cypress supports only JavaScript, whereas FNZ is heavily reliant on .NET. Although it is a direct competitor

with the Selenium WebDriver, FNZ's C# implementation thereof cannot be swapped out for Cypress. Therefore, a serious commitment to using Cypress would also mean a serious commitment to refactoring end-to-end tests present in Kratos as well as browser automation and page object code in Kratos Core.

B. Kratos Considerations

In favour of a Kratos refactor is the argument that the framework has too many end-to-end tests. This stands in opposition to the test automation pyramid model (see Fig. 7) proposed by Cohn [31], whereby automated UI tests should be done as little as possible because they are prone to breakage with small changes, expensive to develop and write, and are time-consuming to run. Fowler [32] also points out that UI end-to-end tests are more prone to non-determinism problems, more colloquially known as "flakiness".

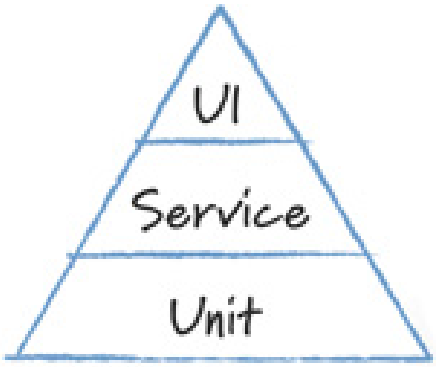

Fig. 7. The test automation pyramid model indicating the relative amount of each type of testing that should be done [31].

Test flakiness is a concern in Kratos for two reasons. Firstly, the complexity of Kratos tests means that identifying the root cause of the issue is a time-consuming task. Secondly, the Selenium WebDriver tends towards flakiness because of its dependence on waits due to its design [22]. It is no surprise that these shortcomings are some of the key challenges in GUI test automation identified by Nass *et al.* [33], specifically that application changes can break test execution and a lack of synchronisation between the test and the SUT may cause assertions to fail. Referring back to the test automation pyramid (see Fig. 7), Cohn [31] argues that the general solution is to cement a testing automation framework on a broad base of simple unit tests. As discussed in Section II-D, FNZ already has an established culture of writing unit tests, however, these tests are not part of the Kratos framework, and it is not clear whether they are automated in some form already. Creating automated test packs of unit tests for clients would help to balance out the ratio of automated test types. More specifically, Ricca *et al.* [29] recommend generating more robust XPath locators using the Robula+ algorithm, which carefully combines XPath predicates to creator short, compact locators with low fragility. This solution could be adopted directly into the Kratos Core codebase for existing `By.XPath()` methods. Another finding regarding Selenium waits that may improve existing Kratos performance comes from Presler-Marshall *et al.* [34], who analysed known Selenium test failures in an academic CI/CD test pipeline and determined that tests with Explicit Waits (where Selenium tells the driver to wait for a certain amount of time or until a condition is satisfied) were more flaky, but that the flakiness itself was more predictable. Ensuring a preference for Explicit Waits in Kratos Core could also be more easily adopted than a wholesale refactor.

Another suggestion from the literature for maintaining end-to-end test cases is the usage of the page object pattern [1]. In practice, a web page is broken down into a series of objects so that the page features are encapsulated into methods, making the page objects a bridge between the actual UI and the test code. The result is that web page elements can now be easily referenced, called, reused, or asserted against within a test. If the UI is updated, then refactoring only needs to concern itself with those elements that have changed. Kratos already makes ample use of the page object pattern to handle web elements.

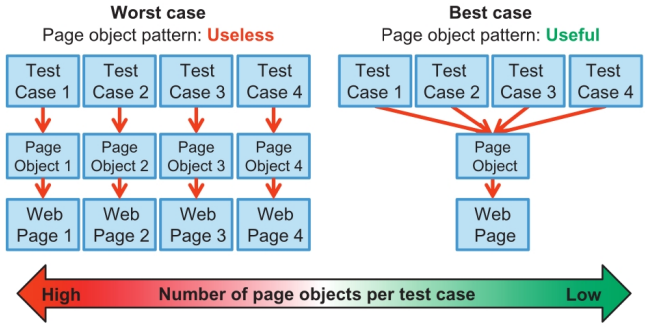

Fig. 8. A diagram illustrating the reusability benefits of the page object pattern for end-to-end testing [1].

Beyond the Kratos codebase, improvements could also be made to the environments in which it compiles and runs. Most of the TeamCity build agents appear to be running Windows Server 2012, which is approaching end-of-life and requires a Microsoft license. As June 2021, Kratos is running on .NET framework 4.7.2 [35], which does not yet have native support for compilation on GNU/Linux. Updating the codebase to the .NET 5.0 (formerly .NET Core) would make it possible to build and compile Kratos on specialised Docker containers running lightweight GNU/Linux operating systems, which would improve build and compile times and lower the infrastructure costs of maintaining TeamCity build agents. Additionally, the Selenium data gathered by Presler-Marshall *et al.* [34] used three different hardware types for testing and found that the Windows system took up to seven times longer to build and compile, and that Windows had the worst Selenium test performance overall in the form of long run-times, flakiness, or both.

C. Cypress Limitations

As previously mentioned, Cypress lacks support for WebKit browsers and multiple browser tabs [26]. The former is a concern for FNZ, as it is expected that client platforms will be tested on WebKit browsers such as Safari. However, FNZ currently does not perform any automated testing of WebKit browsers and instead relies on manual checks, so this particular limitation may not be a significant factor. Kratos does not include the Selenium Safari driver at this time.

Both Selenium and Cypress are also criticised by Moon *et al.* [25] for requiring additional time and maintenance to run

tests in parallel execution, which is a key factor in automated test scalability. As both Selenium and Cypress use a server-client model for test management, multiple instances of these must be configured and ready, and so they conclude that the costs would preclude either framework from being practical in an industry setting. Moon *et al.* [25] offer an alternative approach that argues strongly in favour of deploying native implementations of testing frameworks to match the code being tested so that difficult context switches are not required. By this logic, front-end web application tests should be written in JavaScript without requiring any specialised client libraries. This particular solution is actually indirectly cited by Fowler [32] when discussing what a well-balanced test portfolio should look like. At its most basic, this solution is that like should test like - a web application GUI that makes significant use of JavaScript should use a JavaScript unit testing solution, whereas complex business rules written in C# should avoid calling a browser and instead test the components directly using a C# testing suite. While this line of thinking qualifies Cypress as an appropriate testing suite for misuse testing, it excludes it from most further unit and end-to-end testing because this would require a context switch to C#. The Selenium WebDriver, in contrast, does support most major languages, including JavaScript and C#. An alternative framework that also includes JavaScript and .NET support is Playwright[17], a Node.js library created by Microsoft that has full cross-browser support [22], which may be of interest for replacing Selenium with a more performant browser automation suite.

Finally, Cypress is a third-party tool developed outside of FNZ, which means that its development cycle and decisions are beyond the control of the company. While third-party status does mean that prospective employees may already possess experience with Cypress prior to being hired, and the learning curve for Kratos as an in-house testing framework is steep, Kratos has also grown and evolved over the years as FNZ's business and testing automation needs have dictated. As noted earlier, the Cypress proof of concept project used version 9.5.4. However, at the beginning of June 2022, the Cypress team released Cypress 10.0, which departs from the previous Cypress structure and design in several significant ways that would automatically break the entire proof of concept were Cypress to be updated to the latest version [36]. This means that technically the project is no longer fit for purpose due to external changes beyond the control of the company, which is a serious concern from a stability and resourcing perspective. Therefore, it is important to consider the development priorities and cadences of any third-party tooling being used for testing automation to ensure it aligns with the those of the business in question.

V. Conclusion

Automation testing frameworks should aspire to support best practices of end-to-end testing. Fundamentally, automated end-to-end testing should be reusable, repeatable, and efficient. To counteract the tendency towards brittleness, long run-times, and high maintenance costs of automated end-to-end tests, several considerations must be given when selecting a test tool.

[17]https://playwright.dev/

FNZ's internal automation testing framework Selenium WebDriver makes good use of recommendations such as preferring Explicit Wait and improving test maintenance by relying on the page object model to manage web elements. Selenium also supports both JavaScript and C#, making it a suitable testing suite by the principle of like testing like to reduce context switching and third-party library dependencies. However, the client-server driver design of the Selenium WebDriver may contribute to a loss of synchronisation between the CUT and the test code, and a lack of robust locators used by the XPath may cause easy test breakage with even minor UI updates due to a lack of resiliency. Additionally this imposes high costs for automation upfront as well as ongoing technical debt that has led to difficulty consistently maintaining testing suites for some client platforms.

As part of their testing automation proof of concept project, the team found several advantages to using the Cypress testing framework over Selenium, specifically the ease of writing and maintaining tests and its suitability for automating misuse tests that were being performed manually. Due to being written in JavaScript, Cypress tests executed faster since they only needed to be interpreted, not compiled. Cypress's direct browser driving also reduced amount of flakiness in tests thanks to its improved design over Selenium with better support for synchronisation between the CUT and the test code itself. Cypress was also easy to integrate into the existing TeamCity CI/CD pathways for client platforms in specific environments. The framework also proved capable of handling some simple integration tests, but struggled with larger, more complex end-to-end tests due to a lack of direct .NET support. Other potential business concerns include a lack of support for WebKit browsers and testing in multiple browser tabs. Furthermore, the recent release of Cypress 10.0 illustrates the drawbacks of relying on third-party tools for major frameworks, as this release involved a major refactor of the code that broke the project implementation described in this paper.

Overall, the Cypress testing framework may be a suitable companion to the internal Kratos framework, but will certainly not replace it wholesale due to a lack of support for .NET, WebKit browsers, and multiple browser tabs. However, the prevalence of expensive end-to-end test in the Kratos framework suggests that it should be refactored towards smaller unit and integration tests in line with the test automation pyramid model. Despite performance concerns around compilation and build times, there is evidence to suggest that moving Kratos to .NET 5 would bring substantial benefits in this area, specifically because it would allow the framework to be compiled and run on GNU/Linux Docker containers, saving both time and money. Finally, the alternative testing framework Playwright from Microsoft may prove to be a more suitable Selenium successor given native support for .NET and all major browsers.

References

[1] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Approaches and tools for automated end-to-end web testing," in *Advances in Computers*, vol. 101, A. M. Memon Ed., Cambridge, MA, USA: Academic, 2016, pp. 193-237, doi: 10.1016/bs.adcom.2015.11.007.

[2] I. Sommerville, "Software testing," in *Software Engineering*. Harlow, U.K.: Pearson Education Limited, 2016, pp. 226-254.

[3] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th Int. Workshop Automat. Softw. Test (AST)*. IEEE, Jun. 2012. doi: 10.1109/iwast.2012.6228988.

[4] F. Ricca and A. Stocco, "Web test automation: insights from the grey literature," in *47th Int. Conf. Current Trends Theory Pract. Comput. Sci.*, in SOFSEM 2021: Theory and Practice of Computer Science, vol. 12607, LNCS, Jan. 2021, pp. 472-485, doi: 10.1007/978-3-030-67731-2_35.

[5] R. Stock. "FNZ was founded by Kiwi Adrian Durham in 2003. Now it's worth $3.35 billion." Stuff.co.nz, 2018. [Online]. Available: https://www.stuff.co.nz/business/money/107727511/fnz-was-founded-by-kiwi-adrian-durham-in-2003-now-its-worth-335-billion [Accessed May 30, 2022].

[6] I. Martin. "Face to Face: Adrian Durham, FNZ." Citywire, 2009. [Online]. Available: https://citywire.com/new-model-adviser/news/face-to-face-adrian-durham-fnz/a353963 [Accessed May 30, 2022].

[7] FNZ. *FNZ Overview by Adrian Durham, Founder & Group CEO.* (Aug 26, 2021). Accessed: May 30, 2022. [Unpublished Online Video].

[8] FNZ. *FNZ Timeline.* (Aug. 26, 2021). Accessed: May 30, 2022. [Unpublished Online Video].

[9] "FNZ goes for $34 million." Good Returns, 2009. [Online]. Available: https://www.goodreturns.co.nz/article/976494790/fnz-goes-for-34-million.html [Accessed May 30, 2022].

[10] A. Peyton. "CDPQ buys stake in fintech provider FNZ for £1.65bn." FinTech Futures, 2018. [Online]. Available: https://www.fintechfutures.com/2018/10/cdpq-buys-stake-in-fintech-provider-fnz-for-1-65bn/ [Accessed May 30, 2022].

[11] T. Andreasyan. "Wealthtech FNZ secures investment from Temasek, acquires IPSI." FinTech Futures, 2020. [Online]. Available: https://www.fintechfutures.com/2020/02/wealthtech-fnz-secures-investment-from-temasek-acquires-ipsi/ [Accessed May 30, 2022].

[12] A. Pugh. "Wealthtech FNZ secures $1.4bn investment." FinTech Futures, 2022. [Online]. Available: https://www.fintechfutures.com/2022/02/wealthtech-fnz-secures-1-4bn-investment/ [Accessed May 30, 2022].

[13] "FNZ raises US$1.4bn in new capital from CPP Investments and Motive Partners to accelerate transformation in the global wealth industry." FNZ, 2022. [Online]. Available: https://www.fnz.com/news/fnz-raises-usdollar14bn-in-new-capital-from-cpp-investments-and-motive-partners-to-accelerate-transformation-in-the-global-wealth-industry [Accessed May 30, 2022].

[14] FNZ. (2021). Understanding the FNZ Propositions [Unpublished Online Course].

[15] FNZ. *About FNZ Platforms in General - Legacy Vs FNZ One & Proposition Types.* (May 05, 2020). Accessed: Jun. 01, 2022. [Unpublished Online Video].

[16] FNZ. (2021). FNZ One X 101 [Unpublished PowerPoint Slides].

[17] FNZ Global Automation Team. (April 2020). Kratos Test Automation [Unpublished PowerPoint Slides].

[18] FNZ Product Development. (2020). Introduction to Kratos [Unpublished Online]. [Accessed Jun. 05, 2022].

[19] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Characterising the quality of behaviour driven development specifications," in *21st Int. Conf. Agile Softw. Develop. (XP 2020)*, in Agile Processes in Software Engineering and Extreme Programming, vol. 383, LNBIP, Jun. 2020, pp. 87–102, doi: 10.1007/978-3-030-49392-9_6.

[20] M. Wynne, A. Hellesøy, and S. Tooke, "Why Cucumber?," in *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, 2nd ed., J. Carter Ed. Raleigh, NC, USA: Pragmatic Programmers, LLC, 2017 [Online]. Available: O'Reilly.

[21] P. Bourque and R. E. Fairley, Eds., "Software testing," in *SWEBOK: Guide to the Software Engineering Body of Knowledge*, Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014 [Online], pp. 4-1-4-22. Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering [Accessed Jun. 05, 2022].

[22] B. García, M. Gallego, F. Gortázar, and M. Munoz-Organero, "A survey of the Selenium ecosystem," *Electronics*, vol. 9, no. 7, p. 1067, Jun. 2020, doi: 10.3390/electronics9071067.

[23] FNZ Turbine. (2011). Developer Guide to Unit Testing [Unpublished Online]. [Accessed Jun. 06, 2022].

[24] FNZ Information Security. (2016). Misuse case testing [Unpublished Online]. [Accessed Jun. 06, 2022].

[25] J. Moon, B. Farnsworth, and R. Smith, "The effectiveness of client-side JavaScript testing," in *Proc. IEEE/ACM 1st Int. Conf. Automat. Softw. Test (AST '20)*, Oct. 2020, pp. 101-102, doi: 10.1145/3387903.3389314.

[26] W. Mwaura, "Differences between Selenium WebDriver and Cypress," in *End-To-End Web Testing with Cypress: Explore Techniques for Automated Frontend Web Testing with Cypress and JavaScript*. Birmingham, U.K.: Pakt Publishing, Limited, 2021 [Online]. Available: O'Reilly.

[27] A. Vuorjoki, "A developer-friendly automated web GUI test strategy," M.S.(tech.), Dept. Comp. Sci., Aalto Univ., Espoo, Finland, 2021. [Online]. Available: https://urn.fi/URN:NBN:fi:aalto-2021121910959.

[28] D. North, "Let your examples flow," Dan North & Associates Ltd, blog, Jun. 30, 2008 [Online]. Available: https://dannorth.net/2008/06/30/let-your-examples-flow/ [Accessed Jun. 06, 2022].

[29] F. Ricca, M. Leotta, and A. Stocco, "Three open problems in the context of E2E web testing and a vision: NEONATE," in *Advances in Computers*, vol. 113, A. M. Memon Ed. Cambridge, MA, USA: Academic, 2019, pp. 89-133, doi: 10.1016/bs.adcom.2018.10.005.

[30] L. Badal and D. Grünler, "Automated end-to-end testing: Useful practice or frustrating time sink?," Apr. 2021, [Online Unpublished]. Available: https://raw.githubusercontent.com/KTH/devops-course/2021/contributions/essay/badal-grunler/E2E_Testing_Essay.pdf [Accessed Jun. 05, 2022].

[31] M. Cohn, "Quality", in *Succeeding with Agile: Software Development Using Scrum*. Boston, MA: Pearson Education, Inc., 2010 [Online]. Available: O'Reilly.

[32] M. Fowler, "TestPyramid," martinFowler.com, blog, May 01, 2012 [Online]. Available: https://martinfowler.com/bliki/TestPyramid.html [Accessed: Jun. 07, 2022].

[33] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with GUI test automation (will) remain," *Inf. Softw. Technol.*, vol. 138., p. 106625, Oct. 2021, doi: 10.1016/j.infsof.2021.106625.

[34] K. Presler-Marshall, E. Horton, S. Heckman, and K. Stolee, "Wait, wait. No, tell me. Analyzing Selenium configuration effects on test flakiness," in *2019 IEEE/ACM 14th Int. Workshop Automat. Softw. Test (AST)*, May 2019, pp. 7-13, doi: 10.1109/AST.2019.000-1.

[35] FNZ Product Development. (2021). .Net update [Unpublished Online]. [Accessed Jun. 07, 2022].

[36] "Changelog," Cypress Documentation, Jun. 2022 [Online]. Available: https://docs.cypress.io/guides/references/changelog#10-0-0 [Accessed: Jun. 07, 2022].