A Journey into Test Frameworks

With the creation of the Development Product Team and quality being at the forefront of this project it was paramount to select a testing framework and infrastructure best suited to the project. The main criteria to take into consideration when selecting a testing framework are; **multiple browser** support, **mobile browser/OS** support, **continuous integration** support, **community** support, **embedded keywords** support and **readability**.

1. Geb and Spock testing Framework

Geb is based on the Groovy language and can be utilized with Spock, JUnit, TestNG and Cucumber. Geb used in conjunction with Spock provides the most effective combination allowing for tests to be written using the simple Gherkin Syntax allowing the test to be well structured, as can be seen in Figure 1.

```
import geb.Page
import geb.spock.GebSpec

class LoginSpec extends GebSpec {
    def "login to admin section"() {
        given:
        to LoginPage

        when:
        loginForm.with {
            username = "admin"
            password = "password"
        }

        and:
        loginButton.click()

        then:
        at AdminPage
    }
}
```

Figure 1, Geb and Spock example

Another benefit of using Geb along with Spock is the page object pattern support, allowing a page to be described as an object and Geb will wait for the element specified only once the page has loaded.

```
import geb.Page

class LoginPage extends Page {
    static url = "http://myapp.com/login"
    static at = { heading.text() == "Please Login" }
    static content = {
        heading { $("h1") }
        loginForm { $("form.login") }
        loginButton(to: AdminPage) { loginForm.login() }
    }
}

class AdminPage extends Page {
    static at = { heading.text() == "Admin Section" }
    static content = {
        heading { $("h1") }
    }
}
```

```
import geb.Browser

Browser.drive {
    to LoginPage
    assert at(LoginPage)
    loginForm.with {
        username = "admin"
        password = "password"
    }
    loginButton.click()
    assert at(AdminPage)
}
```

Figure 3, import and use of page object

After looking into Spock's GitHub repository it was found there were numerous issues/outstanding pull requests, suggesting the support provided is not at the standard preferred. https://github.com/spockframework/spock/issues

2. Gauge

Guage is relatively new and follows the flow of defining a specification file along with a class file. The specification file displaying clearly the steps of the test being undertaken and the class file containing the exact commands required to execute each step.

```
Signup
======

The below Scenarios usees the datastore on scenario level.
A new customer gets registered, its name gets stored and reused.

Register a customer
-------------------
Use tags to enrich information about:
Which interface is used? user, admin, manager, ...
What functionality is tested?  signup, login, serach, ...
What is the status of the implementation? inprogress, final, dericated, ...
What is the execution priority / iteration? low, medium, high, smoke, nightly, ...
e.g.:
tags: user, signup, high, final, smoke

* Sign up a new customer
* On the customer page
* Just registered customer is listed
```

Figure 4, Gauge specification file example

Gauge allows for spec files to be broken down in to concepts file (.cpt) for better understanding of a tests flow.

```
# Sign up a new customer

* On signup page
* Fill in and send registration form
```

Figure 5, Gauge concept file example

```
import com.thoughtworks.gauge.Step;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.PageFactory;

import java.util.UUID;


public class UserSpec {
    private final WebDriver driver;

    public UserSpec() {
        this.driver = DriverFactory.getDriver();
    }

    public String localPart() {
        // Creating a random local part of an email address also used as username
        return UUID.randomUUID().toString();
    }

    @Step("On signup page")
    public void navigateToSignUpPage() {
        driver.get(SignUpPage.SignUpUrl);
    }

    @Step("Fill in and send registration form")
    public void searchUser() {
        String username = localPart();
        SignUpPage signUpPage = PageFactory.initElements(driver, SignUpPage.class);
        signUpPage.user_username.sendKeys(username);
        signUpPage.user_email.sendKeys(username.concat("@domain.com"));
        signUpPage.user_password.sendKeys("qweqwe");
        signUpPage.user_password_confirmation.sendKeys("qweqwe");
        signUpPage.commit.submit();
        // store generated username
        signUpPage.storeStringToScenarioDataStore("currentUser", username);
    }
}
```

Figure 6, Gauge class file example

A benefit of Gauge is large sets of test data can be imported from text and csv files and so minimizing the time spent during the test run of creating test data.

3. Protractor

Protractor testing framework is predominantly used to test AngularJS apps. One major advantage of Protractor is the automatic waiting it provides and so when writing/running a test no concern needn't be given about waiting for the test and the webpage to sync. This is a major advantage as many testing frameworks suffer from this issue and many workarounds have to be put in place.

However if an app is not developed fully in AngularJS when using Protractor the automatic wait can be a hindrance and workarounds are required; similar to those required in other testing frameworks.

Protractor tests are written in Javascript and can follow the page object pattern.

```
fit("1. Try to login with correct username and incorrect password", function () {
    loginPage.go();
    loginPage.attemptLogin(angular.user.email, angular.wrongPassword);
    expect(loginPage.isLoginErrorDisplayed()).toEqual(true);
});

fit("2. Try to login with incorrect username and correct password", function () {
    loginPage.attemptLogin(angular.wrongEmail, angular.user.password);
    expect(loginPage.isLoginErrorDisplayed()).toEqual(true);
});

fit("3. Try to login with incorrect username and incorrect password", function () {
    loginPage.attemptLogin(angular.wrongEmail, angular.wrongPassword);
    expect(loginPage.isLoginErrorDisplayed()).toEqual(true);
});
```

Figure 7, protractor test following page object pattern

```
attemptLogin: {
    value: function (username, password) {
        this.isLoginPageDisplayed();
        this.fillUsername(username);
        this.fillPassword(password);
        this.clickLoginButton();
    }
},
```

Figure 8, protractor test following page object pattern

```
isLoginPageDisplayed: {
  value: function(){
    var loginForm = element(by.css("form.login-form"));
    Util.waitUntilElementIsVisible(loginForm);
    Util.waitUntilElementIsVisible(this.txtUsername);
    Util.waitUntilElementIsVisible(this.txtPassword);
  }
},
```

Figure 9, protractor test following page object pattern

4.Cucumber

The Cucumber testing framework is written in a behavior-driven development style using the simple syntax language Gherkin. Cucumber is written in Ruby can be written in Ruby along with many others programming languages.

```
Feature: Search courses
  In order to ensure better utilisation of courses
  Potential students should be able to search for courses

  Scenario: Search by topic
    Given there are 240 courses which do not have the topic "biology"
    And there are 2 courses A001, B205 that each have "biology" as one of the topics
    When I search for "biology"
    Then I should see the following courses:
      | Course code |
      | A001        |
      | B205        |
```

Figure 10, example Gherkin script

When used in conjunction with Capybara, a library written in Ruby, embedded keywords can be used to easily simulate how a user interacts with an application.

```ruby
When /^I want to add/ do
    fill_in 'a', :with => 100
    fill_in 'b', :with => 100
    click_button 'Add'
end
```

Figure 11, Capybara with Cucumber example

5. Robot

Robot framework utilizes the keyword-driven testing approach and follows acceptance test-driven development. Test libraries, written in Python or Java, can be extended with users able to extend these further or create their own.

```
***Test Cases***

1.Login Page Should Have User and Password Required
  [Tags]  Test
  Wait Until Username field is Visible
  Wait Until Password field is Visible
  Check the tooltip message for Username is  Required
  Check the tooltip message for Password is  Required

2.Username must be at least 4 character long
  Wait Until Username field is Visible
  Type into username field  A
  Check the tooltip message for Username is  ${short_username_error_message}
  Type into username field  AB
  Check the tooltip message for Username is  ${short_username_error_message}
  Type into username field  ABC
  Check the tooltip message for Username is  ${short_username_error_message}
  Type into username field  ABCD
  Check the tooltip message for Username is not visible
```

Figure 12, Robot framework test steps example

```
Wait Until Username field is Visible
  Wait Until Page Contains Element  data-automation-id=username

Wait Until Password field is Visible
  Wait Until Page Contains Element  data-automation-id=password

Type into username field  [Arguments]  ${text_to_type}
  Input Text  data-automation-id=username  ${text_to_type}

Type into password field  [Arguments]  ${text_to_type}
  Input Text  data-automation-id=password  ${text_to_type}
```

Figure 13, Robot framework keyword example

6. Intern

Tests are written in JavaScript, it is best for testing JavaScript code but can test other programming languages. Intern's execution model is well-suited to those that follow a test-last development approach and want to prevent regressions using continuous integration.

Intern always uses Promise objects whenever an asynchronous operation needs to occur. All suite, test, and reporter functions can return a Promise, which will pause the test system until the Promise resolves.

```
// the login function accepts username and password
// and returns a promise that resolves to `true` on
// success or rejects with an error on failure
login: function (username, password) {
  return this.remote
    // first, we perform the login action, using the
    // specified username and password
    .findById('login')
    .click()
    .type(username)
    .end()
    .findById('password')
    .click()
    .type(password)
    .end()
    .findById('loginButton')
    .click()
    .end()
    // then, we verify the success of the action by
    // looking for a login success marker on the page
    .setFindTimeout(5000)
    .findById('loginSuccess')
    .then(function () {
      // if it succeeds, resolve to `true`; otherwise
      // allow the error from whichever previous
      // operation failed to reject the final promise
      return true;
    });
```

Figure 14, Intern example script

Conclusion

	Geb	Protractor	Intern	Robot	Cucumber	Gauge
Multiple browser support (IE, Chrome, Firefox, Safari)	✔	✔	✔	✔	✔	✔
Mobile OS/browser support	Appium					
Continous integration support	✔	✔	✔	✔	✔	✔
Community support (0-5)	0	4	5	5	5	2
Embedded keywords	X	✔	✔	✔	✔	X
Ability to run failed tests only	X	X	✔	✔	✔	X
Gherkin syntax	✔	X	X	X	✔	X
Readable syntax	✔	X	X	✔	✔	X
Test data import	X	X	X	X	X	✔
Asynchronous testing framework	X	✔	✔	X	X	X
Automatic test synch with webpage (wait)	X	✔	X	X	X	X

Figure 15, testing framework comparison

Figure 15 displays the main criteria discussed earlier along with others which need to be taken into consideration.

For multiple browser support, mobile support via Appium (Appium drives iOS and Android apps and is an open source test automation framework for use with native, hybrid and mobile web apps) and continuous integration support all the testing frameworks investigated are able to cover these criteria.

With regards to community support, this will provide huge assistance when creating the automated tests. With little or no community support this is evidence the testing framework at some point may not updated in parallel with browsers, meaning the testing suite will become obsolete. This applies to Geb, Guage and Protractor, to an extent.

The ability to run failed tests would save much needed time at runtime of an automated suite instead of running a full suite again due to false failures. This is only available on a select few frameworks, Intern, Robot and Cucumber.

Use of a Gherkin or readable syntax would aid speed at which automated tests can be wrote, reviewed and maintained. Also depending on the stakeholders involved a Gherkin or readable syntax may have to used. Currently Geb, Cucumber and Robot offer Gherkin or readable syntax.

Test data import support depends on the project, only if large sets of test data are required to be imported is when this would be advantageous.

Asynchronous testing frameworks can be advantageous but from experience it was also found they can also be a hindrance. On most occasions the steps within a test are dependent on its previous step e.g. create user via API, login with user etc.

Other items needing to be taken into consideration, which are not easily measured, are maintenance of code and ability to calculate test coverage.

Taking all the above into account the Robot testing framework proves to be the most effective to use to fulfill an automated tests ultimate role, test a system/app

and provide feedback. The main advantages being the readability of the test scripts, the ease and speed of creating a test script, the amount of embedded keywords and the fact it is not an asynchronous framework. However the ability of the framework to automatically synch the test with the webpage is a great feature Protractor offers that Robot does not there. Robot offers many embedded keywords to use instead e.g. ‘Wait Until Page Contains Element’.