JIRAE, Vol. 4, No. 2, October 2019, 46-50
e-ISSN 2407-7259
DOI: 10.9744/jirae.4.2.46-50

Test Driven Development in OWOW's Full-stack Web Development

Albert Pratomo[1,2,a], Erik van der Schriek[2,b], Thomas van der Veen[3,c]
[1] Department of Informatics, Petra Christian University, Jl. Siwalankerto 121-131, Surabaya 60236, Indonesia
[2] Information and Communication Technology, Fontys University of Applied Sciences, Eindhoven, the Netherlands
[3] OWOW Agency, Eindhoven, the Netherlands
[a]albertpratomo21@gmail.com, [b]e.vanderschriek@fontys.nl, [c]thomas@owow.io

Abstract. OWOW is a digital agency which operates in the context of software development, design, and digital marketing. One of the key services it provides is full-stack web development. OWOW separates its full-stack web development into back-end and front-end. OWOW had done some automated testing in back-end, but none in front-end. This was not ideal because the quality of web apps developed could not be easily and thoroughly ensured. OWOW believes Test Driven Development (TDD) might be the solution for the situation. TDD is a software development process where test code are written before the implementation code. Through this research, OWOW would like to start applying TDD into its full-stack web development. Research had been done to investigate how to apply TDD in OWOW's current workflow. In the research, the TDD approach was defined. Afterwards, its application on current back-end and front-end development was investigated. Furthermore, its relation to Continuous Integration was explored. The research findings were then implemented on an ongoing web app project called RentIt. This implementation had been delivered as a proof-of-concept application of TDD in OWOW's full-stack web development. It is concluded that OWOW had been introduced to apply TDD in its full-stack web development.

Keywords: Test driven development, full-stack web development, laravel, Vue.js, continuous integration.

1. Introduction

The goal in Software Engineering is to create a good quality software. Developers expect the software they build to meet all requirements and have proper functionalities. Bugs should be prevented as much as possible, or detected and fixed as soon as possible.

Automated testing is invented to ensure softwares' quality. In simple terms, automated testing is writing code that can test other code. This is especially helpful because the tests can be run automatically by a computer, whenever it is specified to.

As automated testing practices matured, there emerged a new software development workflow called Test Driven Development (TDD). TDD is a workflow where software is developed by writing test before the implementation code [1].

OWOW is a digital agency which operates in the context of software development, design, and digital marketing. One of its main products is web app, in which it does full-stack web development.

OWOW would like to start applying TDD into its full-stack web development. It believes doing so will increase the web apps quality and decrease the time to fix bugs.

This paper documents the research process of investigating how to apply TDD in OWOW's context. Also, how the research findings are implemented on an ongoing OWOW web app project (RentIt), as a pilot case.

2. Background

2.1 Initial Situation

OWOW separates its full-stack web development into back-end and front-end. Back-end is responsible for interacting with database and providing Application Programming Interface (API) endpoints. Front-end is responsible for interacting with users and communicating with the API. Currently OWOW uses Laravel (PHP) as its back-end framework and Vue.js (Javascript) for the front-end side.

OWOW has done some automated testing in the back-end development. It uses PHPUnit testing tool. Most of the tests were written after implementation code, but some were written before. In other words, OWOW has applied partial TDD on its back-end development.

OWOW has not done any automated testing in its front-end development. This means no automated tests to check user interaction with the app, components behaviour, or communication with the API. If anything, a few manual tests are performed by Project Manager to check new features which are being developed.

2.2 Goal

The goal is to get OWOW start applying TDD into its full-stack web development.

It is expected that as an outcome, OWOW knows how to apply TDD into its full-stack web development. OWOW will have its development environment set up for TDD and Continuous Integration (CI). OWOW will also have a TDD pilot case implemented on an ongoing web app project called RentIt.

2.3 Approach

To achieve the goal, an approach has been made. It is carried out in 2 major phases:

1. Research: Investigate on how to apply TDD into OWOW's full-stack web development.
2. Implementation: Apply research findings on an OWOW web app project.

All the learnings from these 2 phases are then shared with OWOW developers to introduce them to apply TDD in their workflow.

3. Research

3.1 Back-end

Back-end development is an aspect of full-stack development which processes data and interacts with the database. Back-end's main responsibility is to provide data to the front-end (which handles user interaction). This is generally done by providing Application Programming Interface (API) for the front-end to communicate with.

For the web app projects, OWOW's back-end development produces web API in the form of web services. Simply put, the back-end provides a set of URLs as communication endpoints for the front-end. The front-end would then use the HTTP protocol to send request to the endpoint. The back-end receives the request, processes it, and returns back a response.

OWOW uses Laravel as its back-end development framework. The programming language is PHP and the database management system is MySQL. Laravel follows Model-View-Controller architecture pattern. The model defines abstraction of data that is stored in database. The view renders HTML pages that are the user interface. The controller handles requests and responses from the client.

However, OWOW does not use Laravel's view to render HTML but instead handles UI in the front-end development. In addition, OWOW implements a more complex back-end architecture: The Manager and Repository design pattern. Figure 1 below shows overview of the back-end architecture pattern used.

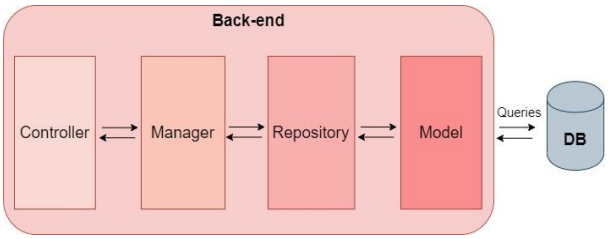

Figure 1. Back-end Architecture Pattern

Testing Situation

Currently OWOW implements Feature and Unit Testing in its back-end development. This is following the recommendation by Laravel through their official documentation.

Feature testing is checking a specific functionality of an app. In the case of back-end Laravel app, this mainly means checking that the web API is working properly. This is done through simulating HTTP request to the endpoint, and then asserting the response or changes made to the database. In doing this, several objects are being tested on how they behave and interact with each other.

Unit testing is testing a very small, isolated portion of the code. While feature testing tests several objects interacting with each other, unit testing focuses only on one. In fact, it mostly tests only a single function of the object at a time. This made unit tests very fast to run and able to directly point a specific part of the code when an error happens.

3.2 Front-end

Front-end development is an aspect of full-stack development which manages the User Interface (UI). Front-end's main responsibility is to handle user interactions. It receives user input and gives back output. In order to do that, front-end needs to communicate with the back-end.

The communication with back-end is generally done by sending request to API endpoints, receiving the response, and then showing information to the user via a web browser. The widely used technology for API communication are HTTP requests. While the underlying technologies for the UI are HTML, CSS, and Javascript (JS).

OWOW uses Vue.js as its front-end development framework. Vue.js is a progressive Javascript framework focused on the view layer, and designed to be easy to adopt and integrate with other libraries. Figure 2 below shows overview of the front-end architecture pattern used.

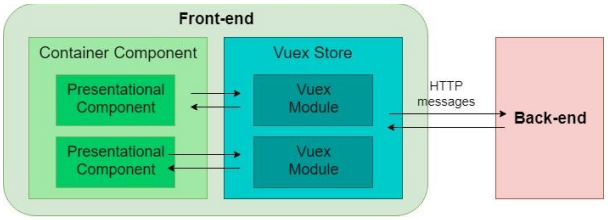

Figure 2. Front-end Architecture Pattern

Testing Types

Vue.js through their official guide and cookbook encourages Unit Testing, specifically for Vue components. It is described that this practice brings some benefits: extra documentation, less bugs, improved design, and easier refactoring. Vue.js also mentions vue-test-utils as their official unit testing library [2].

Edd Yerburgh is a Vue.js core team member and main creator of vue-test-utils. Through his book "Testing Vue.js Application", Edd recommends Unit, Snapshot, and End-to-end Testing for Vue.js development [1].

Vue through their documentation page provides plugins for Unit and End-to-end Testing. For Unit Testing, it provides Jest and Mocha. While for End-to-end Testing, it provides Cypress and Nightwatch [2]. It can be inferred that Vue recommends these 2 types of testing.

Thus far, 3 types of testing are recommended by Vue.js authorities: Unit, Snapshot, and End-to-end Testing.

Unit testing is the process of running tests against the smallest parts of an app. In a Vue.js app, components are the units to test [1]. This is agreeable, because essentially a Vue.js app is a combination of components interacting with each other. In order to guarantee the app's quality, its components must be ensured to behave properly.

Snapshot testing is a method of unit testing that is focused in the visual part of the component. It takes a snapshot of the component's HTML output and uses it as a comparison for future outputs. This way, the developer is always informed whenever there are changes to the component's presentation and can decide if the changes are intentional or accidental.

End-to-end testing is checking an application's behaviour by automating a browser to interact with the running application [1]. In other words, it is automated manual testing. When someone is manually testing an app, he opens the app, clicks through some action, and checks if the app responds correctly. End-to-end testing is the same, except a program, not a human, interacts with the app.

Testing Tools

Unit Testing

For unit testing, the tools needed are Vue utility library and test runner. The main Vue utility library that is officially recommended is vue-test-utils. It provides methods to mount the component in isolation, mock the necessary inputs (props, slots, data), and assert the outputs (rendered HTML, emitted events).

There are currently no visible competitors of vue-test-utils in terms of unit testing utilities for Vue.js development. It is officially recommended by Vue.js and widely found across testing tutorials on the Internet. For these reasons, vue-test-utils will be used.

For the test runner, Vue.js official documentation stated to have compared possible options and as a result recommended Jest and Mocha [2]. These 2 runners are further investigated in terms of feature, performance, and adoptability.

Jest vs Mocha Comparison

Jest is an open-source test runner developed by Facebook. It was initially built for React.js, but has been extended to work seamlessly with other Javascript frameworks, including Vue.js. Mocha is also an open-source Javascript test runner. It is older and more mature compared to Jest, but not backed by a big tech company [3].

Feature

In terms of features, Jest comes with built-in mocking and assertion abilities. It requires little configuration and supports snapshot testing by default. On the other hand, Mocha does not come with built-in mocking, assertion, or snapshotting abilities. Instead, it allows developers flexibility to choose the libraries they want to use (with more configuration). In other words, Jest is more fully featured while Mocha is more flexible.

Performance

Performance test was done on both runners by conducting 5 runs of Btn.spec.js test file, which consists of 7 tests. The result: Jest averagely requires 1.59s and Mocha requires 1.05s. Mocha is averagely faster by 34%. The report can be seen in Table 1 below.

Table 1. Jest vs Mocha Performance Test (Btn)

	#1	#2	#3	#4	#5	Avg	Time Ratio
Jest	1.57s	1.64s	1.58s	1.57s	1.58s	1.59s	1
Mocha	1.04s	1.03s	1.07s	1.04s	1.06s	1.05s	0.66

Edd Yerburgh also did a similar experiment. He measured time required for different amounts of tests [4]. Each amount is run 10 times and the average time is calculated, as shown in Table 3 below. The result: for a single test, Jest averagely requires 0.0197s and Mocha 0.0090s. Mocha is averagely faster by 55%. These 2 experiments reveal that Mocha has faster performance than Jest.

Table 2. Jest vs Mocha Performance Test (Yerburgh)

	10 tests	100 tests	1000 tests	5000 tests	Avg Time/test	Time Ratio
Jest	2.44s	4.50s	21.84s	91.91s	0.0197s	1
Mocha	2.32s	3.07s	10.79s	38.97s	0.0090s	0.45

Adoptability

Adoptability plainly means the level of difficulty to adopt the test runner. This factor is strongly related to setup process and learning resources. Regarding setup proces s, Vue provides plugins for both Jest and MochaUsing these plugins, both runners can be setup easily into an existing Vue.js codebase.

In terms of learning resources, vue-test-utils provides practical guide for Jest and Mocha. Both runners also have extensive documentation and good quantity of Internet tutorials. However, the main book used in this project, "Testing Vue.js Application", uses Jest. Additionally, Vue Testing Handbook by Lachlan Miller also uses Jest [5]. For these reasons, Jest has better adoptability than Mocha.

In conclusion, Jest has more features and better adoptability, while Mocha has more flexibility and faster performance. Jest is decided to be chosen, as it is more suitable for OWOW's current situation as a start-up agency.

Snapshot Testing

Jest, the chosen unit test runner, supports snapshot testing by default. It provides a simple method to take the snapshot, save it in a designated folder, compare the changes, and update the saved ones. This feature is actually a big upside of Jest. vue-test-utils and "Testing Vue.js Application" also use Jest for snapshot testing. With that being said, Jest is chosen as the snapshot testing tool.

End-to-end Testing

End-to-end testing checks an app by automating browser and simulating user interactions. In order to do this, it requires a tool that is able to control the browser. Vue in their official documentation recommends Nightwatch and Cypress for end-to-end tests [2].

Nightwatch is an open-source test framework for automating browsers started in 2012. It comes with a built-in test runner and provides API to perform commands and assertions on the browser. Apart from the main feature, Nightwatch also offers custom assertions, built-in test reporters, Page Object support, and parallel test runs [6].

Cypress is a younger open-source test framework for browser automation, started in 2015. It also comes with a built-in test runner and provides API to control the browser. Additionally, Cypress offers a GUI dashboard for tests management, automatic browser waiting, real-time reloads, and parallel test runs [7].

Over the last 2-year, Cypress has been gaining more popularity compared to Nightwatch. Developers are advocating for it, mainly due to the dashboard GUI and improved developer experience [8]. However, Cypress has a big downside compared to Nightwatch. Since it is based on Chromium, it currently only supports tests in Chromium-based browsers. Nightwatch, which is based on Selenium, supports tests across common browsers (Chrome, Firefox, Internet Explorer, and Edge).

OWOW's web apps are expected to run properly across common browsers. End-to-end testing in these browsers will greatly help to ensure the app's browsers compatibility. For this sole reason, Nightwatch is chosen to be the end-to-end testing tool.

3.3 Continuous Integration

Continuous Integration (CI) is the practice of routinely integrating code changes into the main branch of a repository, building and testing it, as early and often as possible [9]. This practice is strongly related to code version controlling.

OWOW uses Git as its version control system and Bitbucket as the version control tool. This means OWOW puts its Git repositories on the Bitbucket server.

A key part of CI that is lacking in OWOW is Test Automation. Test Automation is running software tests automatically by a machine in a repeatable way, without the need of human intervention [9].

OWOW has done frequent code integration through its Git workflow, but no Test Automation yet. Therefore, the aim is to implement CI, focusing on Test Automation. Because otherwise, the tests resulted from TDD will not be utilized maximally, as they have to be manually run.

After discussion with OWOW developers, it was decided that Test Automation should be run:

- Whenever there is a pull-request made.
- Whenever there is a merge made to develop, staging, and master branch.

As OWOW uses Bitbucket, for simplicity reason, the CI tool selected is Bitbucket Pipelines. It is able to do Test Automation with effortless integration, configurable settings, informative UI, and reasonable budget.

4. Implementation

4.1 RentIt

RentIt is a web app that serves as a machine renting platform. The main idea is that a construction company can create a rent request, which rental companies can reply to. The construction company can then select which offer it prefers and proceed the order. The purpose of RentIt is to facilitate easier renting process in the construction industry.

A part of RentIt is selected as the TDD pilot case, which is the "Admin Manages Machines" Epic (AMM). This Epic is about giving an admin the ability to manage machines of the whole platform. Each machine has a name, a description, multiple types, and multiple options. This Epic is broken down into several User Stories:

- Admin able to see machines index.
- Admin able to create a machine.
- Admin able to see a machine's details.
- Admin able to edit a machine.
- Admin able to archive a machine.

AMM Epic is implemented in both back-end (called rentit-api) and front-end (called rentit-ui).

4.2 Back-end

As previously stated, the back-end's responsibility is to provide web API to communicate with front-end. rentit-api is expected to provide web API for all of AMM Epic user stories. For each user story, this TDD process had been performed:

1. Specify requirements of the user story.
2. For each requirement, write a feature test.
3. When writing the feature test, if a unit test is deemed necessary, write a unit test.
4. Write implementation code to pass the unit test.
5. Write implementation code to pass the feature test.
6. Refactor test and implementation code.
7. Repeat step 2-6 until all requirements are met.
8. Do a manual test with Postman.

User story requirements specification was done during the sprint meeting. The user story's web API request and response data were specified. Furthermore, the main use case (happy flow) and edge cases were described.

For each user story, a feature test file was written. The file contains multiple feature tests. Each simulates the HTTP request and asserts the response and/or the database.

The last step of developing the user story is manual testing with Postman. Postman is a web API client tool. It is used to manually send HTTP requests to the web API and check the response returned.

Once all user stories are done, Continuous Integration was implemented for rentit-api and test code coverage was measured. Table 3 below shows the code coverage results for rentit-api.

Table 3. Back-end Code Coverage

	Code Coverage
All Code	**16.42%**
Machine Store Request	100%
Machine Update Request	100%
Machine Controller	100%
Machine Manager	100%
Machine Repository	37.5%
Machine Model	100%
Machine Code (Avg)	**90%**

4.3 Front-end

As previously stated, the front-end's responsibility is to provide UI to handle user interactions. rentit-ui is expected to provide web API for all of AMM Epic user stories. For each user story, this TDD process had been performed:

1. Break down all the components needed in a user story.
2. For each component, specify its requirements.
3. For each requirement, write a unit test.
4. Write implementation code to pass the unit test.
5. Repeat step 3-4 until all requirements are met.
6. Manually check the component in browser and style it.
7. Write snapshot test of the component.
8. Go back to step 2, until all components are done.
9. Write end-to-end test of the user story.
10. Refactor test and implementation code.

Each user story has a corresponding UI page designed. The page was broken down into components, considering modularity and reusability. Requirements of each component are mostly evident from the design, thus can be inferred straightforwardly by the developer. Occasionally some components' requirements are discussed during the sprint meeting.

There are 2 types of components: container and presentational. For each user story, the TDD process started with the container component (parent) and continue to the presentational components (children). Unit tests were also written for the Vuex store instance, which handles the communication with back-end API.

After all requirements of the component are met, the component was checked in browser and styled. Once proper HTML output was achieved, the component snapshot test was written.

Finally after all components of the user story were done, end-to-end test was written to make sure everything works together correctly.

Once all user stories were done, Continuous Integration was setup for rentit-ui and code coverage was measured. Table 4 below shows the code coverage results for rentit-ui.

Table 4. Front-end Code Coverage

	Code Coverage
All Code	**16.74%**
components/machines/Create	100%
components/machines/Edit	100%
components/machines/Form	100%
components/machines/Index	100%
components/machines/Table	100%
components/machines/Table Row	41.67%
components/common/(Machibe)*	66.46%
store/modules/Machine	75%
library/store/Model Factory	89.58%
Machine Code (Avg)	**86%**

(Machine)* = The common components that are used in AMM Epic

5. Conclusions and Recommendations

Research had been done to investigate how to apply TDD in OWOW's full-stack web development. The findings had been implemented in a web project called RentIt. It is concluded that OWOW has been introduced to apply TDD in its full-stack web development.

The TDD application was separated between back-end and front-end development. In back-end, TDD was applied by writing feature and unit tests, writing implementation code, and refactoring.

In front-end, TDD was applied by writing unit tests, writing implementation code, writing snapshot and end-to-end tests, then refactoring. TDD is related to Continuous Integration, specifically Test Automation. Test Automation had been researched and implemented in back-end and front-end of RentIt. Therefore, the tests written in the TDD process will be automatically run according to the trigger events specified.

For future improvements, it is recommended that in front-end OWOW adopts TDD gradually. OWOW developers should get familiar with front-end testing first. Additionally, TDD application should be determined in a case-by-case basis. Despite all the pros TDD are advocated for, it can slow development. OWOW should attentively decide in which projects or features TDD shall be applied

References

1. Yerburgh, E. Testing Vue.js Applications. New York: Manning, 2018.
2. Vue.js Official Documentation. Available from: https://vuejs.org/
3. Wheeler, K. Jest vs Mocha: Which Should You Choose? Available from https://blog.usejournal.com/jest-vs-mocha-whats-the-difference-235df75ffdf3
4. Yerburgh, E. Vue Unit Test Performance Comparison. Available from: https://github.com/eddyerburgh/vue-unit-test-perf-comparison
5. Miller, L. Vue Testing Handbook. Available from: https://lmiller1990.github.io/vue-testing-handbook/
6. Nightwatch Official Documentation. Available from: https://nightwatchjs.org/gettingstarted/
7. Cypress Official Documentation. Available from: https://docs.cypress.io/guides/getting-started/why-cypress.html
8. NPM Trends. Nightwatch vs Cypress. Available from: https://www.npmtrends.com/nightwatch-vs-cypress
9. Radigan, D. Continuous Integration, explained. Available from: https://www.atlassian.com/continuous-delivery/continuous-integration