4-25-2016

Software Development Using Test-Driven Development and Automated Build Systems

Kaleb James Tyler Smart

Software Development Using Test-Driven Development and Automated Build Systems

by

Kaleb James Tyler Smart

An Honors Capstone
submitted in partial fulfillment of the requirements
for the Honors Diploma
to

The Honors College

of

The University of Alabama in Huntsville

19-04-2016

Honors Capstone Director: Dr. Rick Coleman
Lecturer

Kaleb Smart	19-04-2016
Student (signature)	Date
[illegible]	4-25-2016
Director (signature)	Date
[illegible]	4-26-2016
Department Chair (signature)	Date
[illegible]	4-26-2016
Honors College Dean (signature)	Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

Kaleb Smart

Student Name (printed)

[illegible]

Student Signature

19-04-2016

Date

Table of Contents

Abstract

Software development is an ever changing discipline, where engineers are encouraged and expected to find new, maintainable, and efficient ways to write new software. One of the more recent development mantras is test driven development.

In this agile method, unit tests for software behaviour are written before the actual code is. This simplifies testing, solidifies the intent of the functionality in question, and facilitates the refactoring of existing code at a later point to incorporate new features. In this discipline, programmers are expected to build code frequently to ensure that bugs are quickly found and dealt with before the software is prepared to release. To accommodate such rapid development, automated build systems are often deployed. These build systems will regularly poll a software repository for changes, and upon finding an update, it will build the software and run the test suite.

In this project, a bank management utility will be implemented using test driven development with the Jenkins open source automation server as the build system. The Jenkins server will allow rapid software development for this project by providing frequent software builds and tests in order to quickly notify a member of this development team if a code submission breaks existing functionality, thus allowing fast, efficient, and robust software development.

Motivation

Bugs in software code are highly common. Code problems in large projects are particularly susceptible to issues that may not have been anticipated such as interface connection issues. When developers or testers retroactively write tests for their code, it is very easy for the test code to become entangled and dependent upon the functional code. Test-Driven Development seeks to alleviate this issue by forcing developers and testers to test behaviour, not the code.

Tests are generally simple and specific to a specific task that needs to happen, and as such they are easily portable from one language to another. If an architectural decision is made by a team to change their programming language, the tests may be quickly reimplemented into the target language, and the code may be written to pass those tests to guarantee that all the previous functionality has been migrated. Furthermore, the Object Oriented Programming paradigm lends itself quite nicely to Test-Driven Development due to the prevalence of the SOLID principles, a set of design principles that encourage the development of extensible software.

The SOLID principles are a fundamental set of principles in Object Oriented Development first referenced by Robert Martin, a co-author of the Agile Manifesto, and they lead developers to think about a class's interface separately from the internal mechanisms, thereby causing developers to consider the behaviour of a class, or a group of classes (Martin). These behaviours can easily be extrapolated into unit tests.

Automated build systems can be a useful auxiliary tool to Test-Driven Development because it can automatically verify new code submissions to a software repository on a third-party platform. This allows developers to worry less about whether their programming environment, which is often highly customized, is the only environment a program will run on. It is also a good indicator of the actual dependencies required for a specific software solution. This paper investigates the potential synergy between Test-Driven Development and automated build systems in order to determine whether this logistical setup is feasible.

History of Test-Driven Development

Test-Driven Development as we know it today is credited as being developed by Kent Beck, another co-author of the Agile Manifesto, in his software development practice Extreme Programming in 1999. Extreme Programming suggests using automated tests to “monitor the progress of development, to allow the system to evolve, and to catch defects early,” (Beck 2).

Test-Driven Development has been largely accepted and embraced by the Agile Development community, particularly Robert Martin (21). Martin also helped to popularize Test-Driven Development in his book *Agile Principles, Patterns, and Practices.* He writes that this development methodology greatly facilitates refactoring, which should be done extremely frequently to prevent code rot (Martin 44).

In the past decade, Test-Driven Development has seen a large spike in interest within the professional world, with surveyed usage spiking to 49% in 2008. The most recent survey by VersionOne reveals that currently 33% of business are using Test-Driven Development (10). Although the numbers have dropped in the past 7 years, it still constitutes a large number of users.

The Project

The project was to create a multiplatform bank management system which is accessible to both bank employees and customers. The objective of it was to develop a system which will maintain all information on user accounts (Coleman). This information includes basic user data such as username, password, login tokens, and the role of the user for group-based permissions; account data such as account balance, account owner, and account types which detail interest accrued on balances if applicable; and transaction data which contain the accounts involved, the amount transferred, and contact information if the transaction was a bill.

The project required both a standalone application which would function as an ATM and a web application for customers so that they may manage their money online. The multiplatform requirement for the standalone application prompted our team to choose Java for its development. For the web application, we chose the AngularJS platform due to its extensive support for libraries. We assigned developers to each project according to their level of expertise and desire to work with that section.

Our risk analysis of the project inclined us to implement a RESTful API which acted as a broker to the actual database so that in the event of a member being unable to complete their portion of the project, one of the other members of our team may step in and quickly begin implementing functionality without having to explore the nuances of database API's for each specific language we were working in and only have to research HTTP request API's. Furthermore, since the RESTful API provides a layer of abstraction for the database, we would be able to replace our database software with another without fear of breaking any compatibility with the end-user applications.

Since PHP is developed to be used as a website backend programming language, we agreed to choose the most popular PHP web framework, Laravel Framework. Laravel allows easy development of RESTful API's through its Model-View-Controller architecture. Since our Laravel server was to function as the backend, we did not bother utilizing the View functionality of Laravel. We agreed that this would allow better decoupling of the Model-View-Controller responsibilities if we assigned the responsibilities of the Views to the standalone Java and AngularJS web applications.

Another useful feature of Laravel is its seamless integration with PHPUnit, one of the most popular PHP unit testing frameworks. Laravel's integration allows developers to not only test controller and model functionality that is not available via its HTTP route definitions, but also test the front-facing API. This functionality is extremely useful for our project architecture because most of the logic is happening in the backend, and the validation that the HTTP routes work properly provide confidence to developers of any of the three sections of our project.

The Automated Build System

Jenkins is a Java-based open source continuous integration tool. It is self-contained, extensible, and portable platform which makes it highly suitable to be utilized as an automated software building and testing system (Jenkins). It supports a multitude of operating systems such as the lightweight Docker containers, Windows, Mac OS X, and numerous distributions of Linux.

Jenkins allows the user to create projects, provides status on software builds not only from a single build scope but also a project history scope, and has plug-ins for GitHub to allow for automated building and testing upon pushing Git commits to the software repository. These features allow developers to get rapid feedback of their latest code submissions and see the overall health of a build. Its containerized approach allows the entire platform to be easily migration into another system.

We created a project for our development branch using a GitHub authentication token. We enabled the GitHub plugin to allow a hook to be sent to the server upon code submission that will begin building our code. The Jenkins project was also configured to tag the commits on GitHub with the build state results to indicate errors or successes.

Our team included build steps for the software to automatically run database migrations, seed the database with some base data, and then execute unit tests. If any of these build steps failed, an error would be recorded. Jenkins provides the console output of the build steps, so it is easy to track down what went wrong with a build in the event of failure.

Initially, the Jenkins automated build server was hosted on a developer's personal computer, but we migrated it to our production server so that we could maximize up-time and capitalize on the rapid feedback of code submissions. The lightweight nature of the system allowed unit tests to be ran while the actual system was being utilized by other members of the team as well.

Approach

As is with the general process of Test-Driven Development, we began by creating a new test class before writing any of the functionality. Then, database migrations were created to implement the behaviour expected in the tests. Laravel provides many easy-to-use tools to create, perform, and rollback database migrations, as well as generating model and controller skeletons.

In the first development iteration, we were concerned with creating some base RESTful functionality from the Laravel backend while the GUI's were being mocked up. We designed the basic Create-Read-Update-Destroy (CRUD) methods for the Users, Accounts, and Transactions. Once we had written all of the intended functionality and verified that it was passing the unit tests, the code was submitted to the repository. Upon pushing our code to the GitHub repository, the automated build system would fetch the new code and run unit tests. This helped get rapid feedback on whether our code only worked on our system, or if it worked on our generalized platform.

When code was pushed that failed tests, the errors were given in detail in the console output section of the specific build in the project. This allowed us to quickly identify and investigate the issue which could stem from a multitude of issues, such as typos that were ignored due to the out-of-date class loaders in Laravel or differences in versions of PHP. The latter example happened rather frequently, as each developer was running a different operating system that provided a different version of PHP.

Our second development iteration was concerned with updated the RESTful actions to incorporate user roles and account types. For example, a customer should not be allowed to see another customer's accounts. Updating our tests to support role-based vision of information was quite easy, as the bulk of the code from our original tests was able to be extracted into auxiliary functions.

Our third development iteration was to implement user authentication, so that a user could login, be given a token, and use that token as a basis for identity. This functionality was the first internal implementation that was tested. Laravel provided the ability to call functions in controllers not defined in the HTTP routes so that we could maintain a separation of responsibilities within the tests.

Finally, our fourth development iteration implemented the final requirements such as bill pay. Once the tests were passing, it was verified on the Jenkins server for portability. The testing allowed us to quickly write up and update the documentation for the backend API, as the inputs for the tests could be looked at as a reference instead of having to dig through and examine code.

Results

The Jenkins automated build server was able to integrate effortlessly with our unit tests. It provided the same detailed error and failure information that we saw from running the tests on our individual machines. The existence of a third party build system assisted in us being able to promote code from our development branch into our mainline with confidence and without hassle.

When developers working on the user-facing applications encountered issues communicating with the RESTful API, the unit tests were able to remove the possibility of the backend as the culprit, and developers were able to focus on their area of expertise instead of mulling through unfamiliar code. If developers were able to consider or find an edge case that was not covered in the testing, additions to the tests and controllers could be added rapidly, and it ensured that the changes to the controllers did not break other parts of code.

Jenkins provides a build history for each project, so if multiple commits were made that resulted in multiple builds and one caused a build to fail, the failure could be quickly tracked to a specific commit and analyzed from that scope. This is much more efficient in finding code bugs instead of scouring each commit changes to create hypotheses for which change broke the code. The build details also contain data for the new commits since the last build along with who made each commit, so it is easier to inform the developer who broke the build that they should look into it.

Conclusion

Test-Driven Development has been an extremely popular and efficient method of Agile development. It bestows confidence in developers when they are making changes or additions to their code and software functionality. The method also enables developers to change software architecture as well as port code to other platforms or programming languages due to it being focused on the behaviour of the software instead of the actual code implementation.

With a small amount of configuration, our team was able to create comprehensive coverage for a RESTful API that made software development nearly trivial. All issues were quickly addressed, and we were able to rapidly deploy stable software not only to our group but also to our customer with confidence that the code was in working order. The documentation for our software was complete and coincided directly with the unit tests.

While much of our project's code did not need to be actually built, an automated build system would be even more useful to developers writing in compiled languages such as C or C++. Developers working on a team can receive feedback on tests, and they also will receive output on whether the build failed or not.

Automated build systems accentuate the efficiency of Test-Driven Development by providing a third party platform that is able to provide rapid feedback on the state of your code. It removes the developer's personal setup out of the equation for whether code works or not. The Jenkins automation platform is a free tool that is infinitely extendable and configurable, and it is a must-have product in any software engineer's toolbox.

References

Beck, Kent. Extreme Programming EXplained: Embrace Change. Reading, MA: Addison-Wesley, 2000. Print.

Martin, Robert C. Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ: Prentice Hall, 2003. Print.

VersionOne. VersionOne 10th Annual State of Agile Report (n.d.): n. pag. 2016. Web. 19 Apr. 2016.

Coleman, Rick. "CS 499 Team 4 Project." Message to the author. 12 Jan. 2016. E-mail.

Jenkins. "Jenkins Automation Server." Jenkins. Jenkins, n.d. Web. 24 Apr. 2016.

Otwell, Taylor. "Documentation." Laravel. N.p., n.d. Web. 24 Apr. 2016.

Otwell, Taylor. "Laravel API." Laravel. N.p., n.d. Web. 24 Apr. 2016.

Bergmann, Sebastian. "PHPUnit Manual." PHPUnit Manual. Sebastian Bergmann, 2016. Web. 24 Apr. 2016.