Spring | 09

[RealmCrash] – Test Specification Report

Atasay Gökkaya | 1448661
Cemre Güngör | 1448737
Erbil Karaman | 1395144
Yiğit Boyar | 1448463

[RealmLab Inc.]

Table of Contents

1. INTRODUCTION

RealmCrash is a social gaming framework, which is abstracted from a single realm and can be extracted to multiple themes. This framework lets users to connect and share virtual activities. This is a Massively Multiplayer Online Game concept where different realms interact with each other within their own reality, which is RealmGroups in our definition. This platform is currently reachable from Facebook and iPhone as a prototype, and will be reachable from most popular social networks.

It is obvious that this system, given its goals, needs carefully chosen and adequately run testing methodologies.

1.1 Objectives

Since RealmLab employs an agile approach, RealmCrash gets developed with a very fast pace. New features are added and code gets refactored all the time with continuous integration. The speed of development combined with the small group size, that leads to each member having a wide area of responsibility, makes efficient testing an important necessity.

Because of our constraints, we can't aim to solve all the bugs. We use data driven development and exception driven development to fix the problems in order of scale and severity.

The objectives of our testing process is to keep RealmCrash functional at all times, and to effectively utilize our resources to keep balance between fixing bugs and doing new development.

We will have several different concerns while testing, such as:

- the independent modules behaving correctly (unit testing)
- the independent modules (and external entities) communicating correctly with each other (integration testing)
- all requirements are met for a satisfactory end product (validation testing)
- the product runs properly under high load and with lots of users (stress testing)

1.2 Scope of the Document

This document will explain the measures RealmLab is taking to make RealmCrash a well-functioning, robust and well-maintained project. To do this, we will elaborate on

- what we will test
- what are our constraints while testing
- how will we handle severe, user-facing bugs
- how will we find internal bugs of lower severity

- the automated tools that help us conduct testing
- how our testing relates to our schedule
- how team members are responsible for different tests

1.3 Major Constraints

1.3.1 Time

Since the project is being developed at our remaining time from studies, it hasn't been possible to devote full-time attention to RealmCrash except our winter coding camp in February. A fully-functioning version of RealmCrash is as of date already running on the internet, so our time constraints apply mainly to the remaining work, namely Turkcell integration.

1.3.2 Staff

RealmLab consists of four people, thus each of us has a wide area of responsibility. While this makes development more robust because of low overhead, it also creates the risk of code not being much peer-reviewed. Although we tried to overcome this by pair-programming the more important parts of our project, the low overlap between the work of different team members constitutes a constraint.

1.3.3 Hardware

RealmCrash is a web-bases application thus neither client- nor server-side hardware are relevant.

1.3.4 Software

The server-side of RealmCrash is relatively easier to test, as the servers are in our own control and observation. However a large part of code is client-side. Not only the JavaScript functionality, but the whole frontend of the project is bound for testing within lots of different environments. Our biggest constraint in this aspect is the different rendering engines of different browsers of different platforms. Luckily Facebook, the platform that we run on, actively encourages users to upgrade to A-grade browsers, so we do not expect to encounter lots of rendering bugs.

2. TESTING PLAN

2.1 Overview

Proper testing plan is crucial for RealmLab's success. We have partially integrated planning into our development and deployment lifecycle since we practice Agile development. This semi-automated process is driven by developers. In a constantly changing environment it's best to keep efforts as effective as possible. Creating global rules for testing in such an environment blocks productivity and creativity opportunities. So we concentrate more on atomic testing cases. Below you can find more information about those.

2.2 Testing Strategies and Methodologies

2.2.1 Unit Testing

Unit testing is a very suitable testing methodology for Agile teams. We try to cover the core elements with Unit Tests. The PHP framework that we are using (Zend Framework) supplies a tool called PHPUnit. It's a derivative of JUnit framework. It organizes the tests into cases, which consist of many independent tests with public methods. To create fixtures we use the setUp() and tearDown() methods of PHPUnit. Fixtures are a very good way of isolating independent test cases. They are destroyed after each test. The assertions work pretty much the same way that they work in JUnit and are the standard way of confirming expected values.

Besides Zend Framework integrated this suit into its own MVC so that many additional capabilities are available to test Zend Framework applications. It's enough to extend Zend_Test_PHPUnit_ControllerTestCase for this purpose. This class is exteremly powerful and gives us 5 new assertion classes to test the actual sandboxed form of our web application.

Those are:

CSS Selector Assertions and XPath Assertions to verify artifacts in the response content even if they contain JavaScript/AJAX.

Redirect Assertions to test if the routing works the way its defined in the controller. Often routing depends on stored user profile and request parameters, so this is very handy to match cases where it would be so hard to test with any other methodology.

Response Header Assertions to verify response headers sent back to client or Facebook Proxy. In Facebook Proxy case, there are certain response headers that they do not accept, so this is a good way to catch such cases.

Request Assertions to test routing based on request parameters. The MVC structure of Zend Framework works with routing pattern based on GET parameters. User defined dispatchers can change the behavior and it becomes very hard to test. We wrote a state machine request dispatcher for certain cases, and this assertion class makes it very easy to test our dispatcher rules.

For projects that relay on remote interfaces (DB, Web Services, Email etc.) Mocking is a very useful method. PHPUnit supplies Mock Object creation interfaces that enable us to write very sophisticated Test cases. With mock objects many side effects of a function can be hide from the tester. Isolation yields to less false-positives and it's significantly important since our deployments heavily relay on those tests.

To be able to run the Unit Tests we use the command line tool

phpunit <TestCaseName>

It runs the given test case and reports to STDOUT. This is very useful for our automated testing needs.

Even if the test first/implement later is widely considered as a standard in TDD (Test Driven Development), we decided not to go strictly that way until we clarify the base API. In an environment that we are not totally familiar and constantly changing, we believe that starting implementation as soon as possible with minimal design shows us where we gave wrong decisions so that we can change as fast as we can to make the design better. We see that worked very well for us up to now, and we gained enough experience to continue with better/robust design. So we are trying to move in the direction that we test more before we implement.

Another important point that we had discovered is to be able to write effective Test Cases, we need to have good encapsulation and code craftsmanship. Functions that are too long and do multiple things together are more vulnerable than small and simple functions since it's harder to test.

As a result, unit testing yields reduced number of bugs in production code, reduced development time in long tail, repeatedly testable functionalities, re-producible test cases, more freedom for developers on others code if they are well covered, better overall software design, measurable development efforts and well integrated automated deployment environments.

2.2.2 Integration Testing

Instead of re-inventing everything, we use a lot of external technologies wherever we need. Most of them are web services so can be easily mocked (during Unit Testing). However Mocking assumes that those services work without any problem. In real life, unfortunately that is not the case. Usually the documents are out of date or the services are not as robust as they can be. We try to encapsulate every integration point to a well designed adapter/manager class, so that we can catch any abnormalities.

Our approach is very similar to Big Bang approach, where we let our tests run in test environment with real users, and we watch for exceptions and logs from adapter classes for the integration points. This is called Exception Driven approach and pretty new in this area. We use Splunk to watch exceptions and logs, and those are automatically indexed so that we can search by certain parameters to dig into the problem.

This is the fastest way to do Integration Testing and becoming more and more popular in start-up world that does Agile development. Facebook also uses this technique in a very similar way.

2.2.3 System Testing

System testing constitutes the tests that are run just after Integration tests. After Integration testing, the software modules are validated to communicate with each other correctly at least as described within those tests. System testing, in fact, is conducted on a complete, integrated system to see if the combination of modules comply with the specified requirements. Its goal is to detect any defects

within the software modules, connections between modules and also the system as a whole.

As we had described in our previous design reports the RealmCrash system is composed of several layers and helper modules that are used for functional computations or operations. Since System testing has several types (can be automated or not) that are used to detect the defects on Functional Requirement Specification and/or System Requirement Specification throughly, we decided to implement necessary important System testing types, such as Usability testing, Load and Stress testing, and Scalability testing. The reasons to use these methodologies, our approaches and plans are described below in detail.

2.2.3.1 Usability Testing

RealmCrash needs to be very usable, as we follow our idea, a user without any coding skills and without a predisposition to games (i.e. in gaming literature, from blue ocean) should be able to create his or her own game using our system. To be able to do that for an initial setup, we requested several people from different backgrounds to participate in our Usability tests. We watched people both on eye (psychological behaviors) and automatically (measuring the time they spent on the atomic activities). We have greatly improved our system from the results of these tests.

2.2.3.2 Load and Stress Testing

Although Load and Stress testing are concerned with different aspects of testing, we decided to approach them together as they are strongly related. Load testing is basically putting extensive demand on a system and measuring its response. On the other hand, Stress testing is concerned with the robustness, availability and error handling of a system under heavy load.

RealmCrash's mission is to bring millions of users from different Social Networks together to let them engage in their favorite worlds, realms. To achieve this goal, RealmCrash needs to stand safe under very heavy load and scale out when needed. Standing safe means responding deterministically and being robust in our context.

We have examined several Load testing suites and tools. We plan to use Selenium suite to automate our Load testing across many platforms. Selenium has a Firefox (browser) extension IDE to generate test cases for web applications and other tools to run those tests and to distribute tests on multiple servers.

We plan to assert the response of our system under heavy load for Stress testing. We can use Selenium tests for verifying the correctness of the responses under high demand.

2.2.3.3 Scalability Testing

We think that Scalability testing is also very important for providing our users a reliable and qualitative service under very high demands. Scalability testing is concerned with measuring a systems capability and righteousness to scale up or

scale out in terms of any non-functional capability, like user load supported per server, number of connections, data volume etc.

RealmCrash will scale out for processing power according to a metric which will define the necessity of scaling. We are using Amazon Web Services - Elastic Computing Cloud virtual server instances. AWS provides the necessary services to use and scale out our system when necessary. The Scalability tests will allow us to detect possible defects during scaling.

2.2.4 Acceptance Testing and Quality Assurance

As we have described that we are using Extreme Programming, one of the Agile Software Development methodologies, we believe that the quality of the service is very important and a release should be accepted or not using a functional testing of a user story that was defined by the software development team during the implementation phase.

2.2.4.1 Alpha - Beta Testing

We have released alpha and beta versions of our system components, namely a game instance, and our creator. Alpha release was held within our project group and close friends, which in original takes place in developers' site as we did. Then we have released public beta versions of these components and shared them with our friends and game experts. We have improved the usability features and fixed several functional defects in our system thanks to our public beta users and their kindness to give us very helpful feedback.

2.2.4.2 User Acceptance Testing

We have created several user Acceptance tests involving several user stories such as creating a game using our generator, and playing or doing some actions within a game that is running our game engine, etc. These tests are simple and give a boolean result, either accepted or rejected, no degree of success or failure. If all of the tests are accepted than it will assure the quality of our service (release). If we think that the current tests are not enough for judging the quality of our system, we will define extra user stories to test our quality.

3. TESTING TOOLS

Continuos integration is a key concept for us which we need to achieve as soon as possible. To reach continuos integration, automated testing is a must since any unstable commit should be prevented from going to the production repository.

To test server side PHP codes, we will use Zend_Unit, which is a unit testing abstraction based on PHP Unit; which is widely used in many php projects. We also need to test client side codes since most of the usability functionality is implemented in client side. For client side test, we will use selenium server.

We need a centralized mechanism to control testing tools; which are subject to grow in number. For this task, we will use BuildBot, which is a widely used open source compile/test cycle software.

3.1 Zend_Test

(http://framework.zend.com/manual/en/zend.test.html)

Zend_Test provides tools to facilitate unit testing of your Zend Framework applications. It only provides MVC testing tools but since we also use MVC, it works for us.

This testing utility also provides interface to mimic user requests via controller but since we use facebook, our sessions are dependent on facebook; controllor testing will be a little tricky.

3.2 Selenium RC

(http://seleniumhq.org/projects/remote-control/)

Selenium Remote Control (RC) is a test tool that allows you to write automated web application UI tests in any programming language against any HTTP website using any mainstream JavaScript-enabled browser.

Selenium Remote Control is great for testing complex AJAX-based web user interfaces under a Continuous Integration system. It works on 3 major browsers, namely firefox, internet explorer and safari. To use it, you start a selenium server which you can connect via php, java and some other programming languages.

To use selenium RC with continuous integration, we will wake up two virtual machine instances, one mac and one windows XP. On these instances, two selenium servers will run which are controlled by Build Bot. Any written UI test will be run on firefox, safari and internet explorer 7 & 8.

A sample Selenium test written in Java looks like as follows:

```
selenium.open("/");
selenium.type("Bugzilla_login", "admin");
selenium.type("Bugzilla_password", "admin");
selenium.click("log_in");
selenium.waitForPageToLoad("30000");
selenium.click("link=Reports");
selenium.waitForPageToLoad("30000");
```

3.3 BuildBot

(http://buildbot.net)

The BuildBot is a system to automate the compile/test cycle required by most software projects to validate code changes. By automatically rebuilding and testing the tree each time something has changed, build problems are pinpointed quickly, before other developers are inconvenienced by the failure.

We will integrate buildbot to our svn with a hook. When an employee commits some code, this hook will wake up buildbot to run test cases. If any of the test cases fail, buildbot sends an email to the commit owner and to our mail group with a list of people to blame.

3.4 Nagios

(http://www.nagios.org/)

Nagios is not a testing tool but a production monitoring software via which we will be sure our production environment works fine.

We will setup a Nagios instance on server which will monitor database, memcached and server instances. If any of these components have problems (e.g. high db load, high cpu/io load, memcached down etc); it will either restart the component or run the rescue script we provide it. Nagios also has capabilities like sending sms, email which will inform us as soon as disaster occurs.

4. TESTING SCHEDULE

4.1 Continuous Testing

We have 3 playground groups, namely dev, test and production. All codes are developed on dev. When decide to deploy them on prod, all codes are first moved to test which has almost same settings with production. If it works fine, codes are pushed to production.

Dev environment is controlled by BuildBot and production is monitored by Nagios.

BuildBot runs tests using Zend_Test and Selenium RC on dev when a new commit is sent to svn. When we decide to push a new release to production, we first push it to test environment and tell BuildBot to run tests on test environment. If everything is fine, code is pushed to production and BuildBot runs tests on production.

Since we run unit tests periodically on dev, there is a very low probability of having a failure on release but our test-move structure saves us from deployment crisis which is quite common in web projects.

4.2 General Version Testing Schedule

General Version Test	Dates	Completed
RealmCrash Engine and Sample Game (Alpha Tests)	10.1.2009 - 15.2.2009	Yes
RealmCrash Engine and Sample Game (Beta Tests)	16.2.2009 - …	Continues
RealmCrash Creator (Alpha Tests)	1.5.2009 - 7.5.2009	Yes
RealmCrash Creator (Beta Tests)	8.5.2009 - …	Continues

4.3 Testing Implementation Schedule

Automated or half-automated testing schemes that are implemented/going to be implemented are as follows. We do not mention our other periodic/release driven tests (user acceptance tests, alpha – beta tests) here that were explained above.

Testing Methodology	Start Date	Completed
Unit Testing	15.2.2009	Half Coverage, continues
Integration Testing	6.4.2009	Half Coverage, continues
Usability Testing (System)	7.5.2009	Yes, will continue with new functionalities
Load and Stress Testing (System)	30.6.2009	No
Scalability Testing (System)	10.7.2009	No

5. REFERENCES

- Wikipedia, *http://en.wikipedia.org/*