Test report

Lokesh Podipiredy
MD Rashad

April 2017

Contents

1 Introduction

The purpose of this document is to go describe the various testing methodologies that went into testing the system, the result of using these methodologies and also discuss the various improvements that can be made for testing this platform.

A couple different approaches were used to test out the system. Due to the client-server architecture nature of the system, we need to establish two different approaches to testing. One approach should test the back end of the system while the other should test the front end of the system.

The way we tested the backend of the system is through the use of automated testing. This approach ensures that the backend doesn't have any issues when it comes to making server requests as well as database requests

The way we tackled client testing is via the unit testing to ensure the individual javascript functions are working as well. We conducted usability testing to ensure that the html elements are working for various screen sizes as well and various browsers.

2 Automated Testing

Automated testing was used to test backend of the system from the backend server to the saving the information in the database, this automated testing system ensures the application is following procedure.

Usually for each database table, there is an associated controller and model that is responsible for manipulating the table. For example when we make a request in this application to save a drawing. The server makes a request to the drawing controller which has a method called "create". When the server hits this controller method, the client will pass all the parameters needed to save a certain drawing instance to the controller in order to save it to the database. When the request reaches the controller, it will span off a new instance of the "Drawing" model, update the attributes of the newly created drawing instance with the parameters that were just passed into the controller and run the save method on the newly created "Drawing" instance to save it to the database. Now that we have an idea of the server works, we can discuss how to test it The way testing works is that each controller and model will have a test file. In this case we are more concerned with the controller since the controller access the model. The controller test file will have a test method for each method in the controller. In our case we are testing out index, create, update and delete for methods for each controller.
The reason for this testing it that it is automated, so you can call all the test cases by running a single command once the file is setup, and it test the controller, the model and the database due to how they all interact making it almost a system wide testing procedure that is also automated.

```
require 'test_helper'

class BezierCurvesControllerTest < ActionController::TestCase
  setup do
    @bezier_curf = bezier_curves(:one)
  end
```

Figure 1: bezier curve test 1

From The above picture, we can see that before the testing starts, it sets up a testing object to test the controller with. The object is populated with all atomic values. i.e if all the fields are integers it will be populated with 1.

From The above picture, we are testing out he create method which saves the bezier curve to the database. The string right beside the keyword test is used to describe the behaviour of that testing procedure. The assert difference looks for the difference in the count in the table once the object has been created and saved to the database. The code within the assert creates a post request with the parameters setup from figure 1. Below is a table of various controller testing. Once you have the test file setup, you can call "rake test" within the

```
test "should create bezier_curf" do
  assert_difference('BezierCurve.count') do
    post :create, bezier_curf: { cp1_x: @bezier_curf.cp1_x, cp1_y: @bezier_curf.cp1_y,
  end

  assert_redirected_to bezier_curf_path(assigns(:bezier_curf))
end
```

Figure 2: bezier curve test 2

terminal to run the test.

	BezierCurve	Drawing	User
Create	Pass	Pass	Pass
Update	Pass	Pass	Pass
Delete	Pass	Pass	Pass
Index	Pass	Pass	Pass

2.1 Issues with this automated testing

Some of the issues that we ran into while testing was with associations. When you have associations between tables, such as a user has many drawings, it creates problem with the testing because you need to spawn of an instance of the drawing in order to save the bezier curve. There are a few ways to solve this. You can have dependency injection whenever there is such an association. For example, each drawing has a user, you can inject the user model to deal with this issue within the test file. This seemed too complicated so we manually used a user id that was already saved in the database to deal with this issues. it seemed much simpler than going through the process of dependency injection.

3 Front end Testing

The Front end was tested in various ways. It was tested for browser compatibility. Various browsers were tested with the system to see if the functionality and look is consistent throughout the browsers. Below is are tables describing this.

	Home	Login	Dashboard	Drawing
Internet Explorer	Pass	Pass	Pass	Fail
Google Chrome	Pass	Pass	Pass	Pass
Firefox	Pass	Pass	Pass	Pass
Safari	Pass	Pass	Pass	Pass

Almost all testing passed except for Internet Explorer due to the lack of WebGL in the browser.

Another form of testing that was done was the usability testing. This testing is to see if the application works under various conditions. One of the problems, we ran into was when users were manipulating the bezier curves using the handles. At first they weren't given proper feedback about the handles not connecting properly. We solved this issue by programming an alert to the user that they are selecting the wrong.
Another issue we ran into, is that users would render shapes that were not connected. This becomes problematic because the STL triangles were still being rendered and it was still able to output it onto a STL parser but wasn't going 3D printer compatible. The way solved this issue, is by detecting a discontinuation in the shape and alerting the user of this problem when this heppens

4 Avoidance of Unit Testing

We avoided unit testing, due to time contraints. Since there were so many components to test out within the system and also test out the entire system in terms of dependency against one another, we felt that it would eat up in the development time as well. We felt system wide testing, functionality testing and usability testing needed to priortized in order to meet deadlines as well. We also went through a few iterations of User Interface meaning that many of the javascript functions were getting thrown away as they were being created.

5 Going Forward

We would like to perform more rigrous usability testing of the system. We believe there are still issues to be solved in terms of giving users proper feedback about errors. If more time were allocated, we definately would also do unit testing on individual functions in order ensure against garbage inputs.