SE 3XA3: Test Plan
Spann

Team 5
Christopher Stokes — stokescd
Varun Hooda — hoodav

December 8, 2016

Contents

1 Functional Requirements Evaluation **1**
1.0.1 Front End Tests 1
1.0.2 Server API Tests 2
1.0.3 Back End C# Server 2
1.0.4 Back end C# server 6

2 Nonfunctional Requirements Evaluation **6**
2.0.1 Front End JavaScript UI 6
2.0.2 API /Back End Tests 7
2.0.3 Usability 8

3 Comparison to Existing Implementation **8**

4 Unit Testing **9**

5 Changes Due to Testing **9**

6 Automated Testing **10**

7 Trace to Requirements **10**

8 Trace to Modules **12**

9 Code Coverage Metrics **14**

List of Tables

1 **Revision History** i
2 Trace to Requirements 11
3 Trace to Modules 13

Table 1: **Revision History**

Date	**Version**	**Notes**
Dec 7, 2016	1.0	Added content

This document summaries the various testing strategies employed in this project to ensure the application meets the specified requirements, both functional and non-functional. Specifically, this document will outline all the tests that were performed, along with the results of each test, and any changes to the application that came from those results.

1 Functional Requirements Evaluation

1.0.1 Front End Tests

1. SHA512

   Initial State: Function, has no state.

   Input: UI data objects with a string to be hashed.

   Expected: Correct hash string,

   Result: PASS – The module results the correct hash string.

2. Encryption utils

   Initial State: Function, has no state.

   Input: UI data objects of a string and a salt.

   Expect: UI response objects with the hash string

   Result: PASS – Module returns correctly formatted object with the hash string.

3. Login Screen

   Initial State: Function, has no state.

   Input: Set of correct user credentials and a set of incorrect user credentials.

   Expect: (returns true) Successful authentication with correct credentials and (return false) no authentication with incorrect credentials.

   Result: PASS – module does not allow unauthorized access when given incorrect username and password, but does allow access when correct username and password are supplied.

1.0.2 Server API Tests

1. Invalid Requests

   Type: Functional, Dynamic

   Initial State: Function, has no state.

   Input: Invalid user authentication request, invalid file/project request.

   Expect: Response object from server indicating that the request is invalid.

   Result: PASS – Server returns object indicating an incorrect/invalid request is made and no action can be performed.

1.0.3 Back End C# Server

1. SQL Queries

   Initial State: Function, has no state.

   Input: Database request object.

   Expect: A valid SQL query that corresponds to the request.

   Result: PASS – The module provides a correct and valid SQL query.

   Example SQL Tests

   (a) Create

   Test 1

   Name: Domain Model Object Data

   Output: valid SQL code for creating that object in the database.

   Expected Results: SQL code with random data inserted into correct spot

Result: PASS

(b) Update

Test 1

Name: Test Generate Update With Where

Initial State: No initial state. Code is a Pure Function and has no internal sate data.

Input:
1. DomianModel object data.
2. Inline LINQ code for the C# representation of the where statement

Output: valid SQL code for updating that object in the database with the C# LINQ code transformed into an expression tree then into the correct select statement.

Result: PASS

Test 2:

Name: Test Generate Create With ID

Initial State: No initial state. Code is a Pure Function and has no internal sate data.

Input:
1. DomianModel object data.
2. ID of object to update

Output: valid SQL code for updating that object in the database with select clause for that ID.

Expected Results: SQL code with random data inserted into correct spot.

Result: PASS

(c) Load

Test 1

Name: Test Generate Load With Where

Initial State: No initial state. Code is a Pure Function and has no internal state data

Input:
1. Domain Model Object Data.
2. Inline LINQ code for the C# representation of the where statement.

Output: valid SQL code for loading that object in the database with the C# LINQ code transformed into an expression tree then into the correct select statement.

Result: PASS

Test 2:

Name: Test Generate Load With ID

Initial State: No initial state. Code is a Pure Function and has no internal sate data.

Input:
1. Domain Model object data.
2. ID of object to update

Output: valid SQL code for loading that object in the database with select clause for that ID.

Expected Results: SQL code with random data inserted into correct spot.

Result: PASS

(d) Delete

Name: Test Generate Delete With Where

Initial State: No initial state. Code is a Pure Function and has no internal sate data.

Input:
1. Domain Model object data.
2. Inline LINQ code for the C# representation of the where statement

Output: valid SQL code for removing that object in the database with the C# LINQ code transformed into an expression tree then into the correct select statement.

Expected Results: SQL code with random data inserted into correct spot.

Result: PASS

2. Domain Model Tests

Initial State: Function, has no state.

Input: Domain Model Transfer Object.

Expect: Data Model (DM) object with the correct corresponding metadata.

Result: PASS – Module returns a correct DM with the correct meta-data.

1.0.4 Back end C# server

1. Python Execution

   Initial State: Function, has no state.

   Input: String of valid python code and a string of invalid python code.

   Expect: String containing the expected output for a valid piece of python code and a string containing the expected error's for a invalid piece of python code.

   Result: PASS – Module returns expected code output for valid code and a string of error/exceptions/stack trace for an invalid string of code.

2 Nonfunctional Requirements Evaluation

2.0.1 Front End JavaScript UI

1. UI performance

   Initial State: Application running inside browser.

   Input: User input using mouse clicks and keyboard input.

   Expect: The application should feel responsive to any user actions and actions performed should be performed so that the user is kept engaged.

   Result: PASS – The application's response times are unnoticeable while in use. The longest delay in any action performed is when the actual python code is sent to the server to be executed (the delay is a result of the actual python code execution, as opposed to any request processing delay).

Look and Feel

1. UI visual inspection

   Initial State: Application running inside browser.

   Input: Mouse inputs for navigating the application and keyboard for any textual input for the purposes of seeing text/code.

   Expect: The rendered DOM (html elements) should be as the developer intended. There should not be any anomalies, bugs or incorrect properties in any of the elements rendered to the screen. The results should be as similar as possible on different browsers (browsers inherently render elements differently, so results should be as minimal as possible across browsers).

   Result: PASS – All elements are rendered as the developer who designed them intended. The rendering across browsers has no noticeable differences.

2.0.2 API /Back End Tests

Stress Test

1. API/Server Stress

   Initial State: Function, has no state.

   Input: String of python code from multiple concurrent connections.

   Expect: The server should process each request and respond with the correct result with minimal delay. Increasing the stress on the application should have a proportional delay in response time.

   Result: PASS – The application handles a large number of concurrent requests without noticeable delays in performance/response time. It was noted that the results do differ when a less powerful machine is used to perform the tests and host the server, since the hardware is less powerful and can't handle the stresses of newer/more powerful machines. Thus, Modern machines are well capable of hosting the application without any performance issues.

2.0.3 Usability

The usability of the project was manually determined using the usability survey questions (as specified in the Test Plan document).

After manually using the application as a regular user would, it was deemed that the application is quite usable and there are not any aspected that hinder the application or present a scenario/situation in which the application fails to be usable (under regular, expected usage).

3 Comparison to Existing Implementation

The existing implementation did not have a set of tests to compare the results against. But the manual comparison of the two implementation can be made.

Functionality

Both projects has very similar functionality. The major difference is that Spann supports projects and multi file projects while the existing implementation does not. The python execution and results of the same code on both platform are, as expected, identical.

Performance

Both implementations a fast and responsive. It can be noted, since the existing implementation is running on what can be assumed to be faster hardware, the python code executes faster that on Spann.

Stress Test

The existing implementation is hosted in a public facing server and it can be assumed thousands of concurrent requests can be made without any delays. Testing this would be very difficult for the developers of this, Spann, project.

As noted in the stress test section 1, this application is able to support concurrent requests without noticeable delays or issues regarding performance. But the performance overall is dictated by the hardware the application is running on, hence, running on a modern laptop computer, this application can't support thousands of concurrent requests. But, should this application be hosted on a more powerful server, a larger number of concurrent requests could be supported. Testing on a powerful server is currently not feasible for this project and team.

4 Unit Testing

There were two sets of unit tests performed. The Jasmine, JavaScript based, unit tests for front-end facing code were performed using the NodeJS Jasmine module. The Nunit, C# based, unit test for server components and modules were performed using C# and the Nunit testing framework.

The Jasmine tests using the ”spec” prefix to specify which component/module they are testing. A configuration file is used to indicate the location of the project code and the test code. Jasmine uses this configuration file to automatically perform the tests.

The C# tests are performed using Visual Studio. This method of tests are very similar to the well known Java JUnit method of testing. Visual Studio handles the majority of the work in regards to testing, the developer needs only write the test and specify the code to be run before and after each test/suite of tests (as it is in JUnit).

5 Changes Due to Testing

Summary of the changes as a result of testing,

- Refactoring UI JavaScript UI components to allow code inside callbacks and inline function definitions, to be exposed to the testing framework.
- Refactoring server C# code for better modularity and individual testing. Reducing the coupling between pieces of code to allow tests to test individual components.
- Editing CSS/UI specification in JavaScript to allow for elements to be rendered as similarly as the browsers allow when the application is running on different browsers.
- Handling user input in the python console interpreter. The python console interpreter is a heavily modified version of the regular editor used throughout this application for writing code. The testing, manual

testing since that is the only feasible means of testing this, revealed subtle bug in the way the components performed and handled input.

- UI/overall front end design to accommodate when the framework allowed and the different browsers allowed to be rendered correctly.

6 Automated Testing

The automated testing was performed using, as noted previously, Jasmine and NUnit. The other automated testing that was performed, API testing, was performed using the Postman chrome web application. Postman allows the developers to write scripts that perform automated tests for the API.

All the front end tests for UI components were performed using Jasmine. All the API tests were performed using Postman. All the server tests for C# server components was performed using NUnit.

7 Trace to Requirements

Req.	**Modules**
Mode	Login Screen 3
	SHA512 1
	Encryption Utils 2
Editing	UI Performance 1
	UI Visual Inspection 1
Editing Support	UI Performance 1
	UI Visual Inspection 1
File Handling	Invalid Request 1
	SQL Queries 1
	Domain Model 2
Code Execution	Python Execution 1
	Domain Model 2
	API/Server Stress 1
Shell Interpreter	Python Execution 1
	API/Server Stress 1
	UI Performance 1
	UI Visual Inspection 1
Networking	Invalid Request 1
	Stress Test 1
Accounts	Invalid Request 1
	Domain Model 2
Account Management	Invalid Request 1
	Domain Model 2
Look and Feel	UI Performance 1
	UI Visual Inspection 1
Usability	UI Performance 1
	UI Visual Inspection 1
Performance	UI Performance 1
	API/Server Stress 1
Security	Login Screen 3
	SHA512 1
	Encryption Utils 2
Health and Safety	UI Visual Inspection 1

Table 2: Trace to Requirements

8 Trace to Modules

Req.	**Modules**
Hardware-Hiding Module	UI Performance 1 UI Visual Inspection 1 API/Server Stress 1
API Controllers	SQL Queries 1 Domain Model 2 Python Execution 1
Python Console	API/Server Stress 1 Python Execution 1 UI Performance 1 UI Visual Inspection 1
Python Console Manager	Domain Model 2 Python Execution 1
Python Runners	Python Execution 1
Repository Model	Domain Model 2 SQL Queries 1
Domain Models	Domain Model 2 SQL Queries 1
UI Screens	UI Performance 1 UI Visual Inspection 1

Table 3: Trace to Modules

9 Code Coverage Metrics

Visual Studio was used to measure code coverage for all C# code. Visual studio reports 26% coverage. A reason for not having a higher coverage is that a lot of the testable code inherits from a select from classes. A better approach to testing, in reflection, would be to have tests that incrementally tests each class and become more specific with each child class. This would result in better code coverage and a more through test suite.

For the API test, this project achieved 100% code coverage, as reported by Postman (the chrome application this project uses for testing the API). API for this project is very simple and quite small. This allows the developers to test all aspects of the API and achieved 100% coverage.

The front end JavaScript was not measured for code coverage due to infeasibility. The front end relies heavily on the JJS UI framework. Measuring code coverage is also made more difficult by the nature of JavaScript and the heavy use of closures in both this project's JavaScript code and in JJS's code. A rough estimate of code coverage by inspection would yield a number of around 25%, but this should not be considered a reliable figure.