# UNIVERSITY DEPOT

# *Name: Dustin Annarumo, Heidi Bednarz, Zack Colich, Jenna Jossalyn, Cody Kuntz, Joshua Malone, Steve Simpson*

# *Date Completed: October 27th 2022*

# *Application Build Number/Version: 2.0*

# *Database Version: 2.0*

# 

# 

# 

# 

# *December 6th, 2022*

# Table of Contents

# 

# University Depot Testing Plan........................ 4

# Overview of Testing............................... 4

# Instructions.............................................. 4

# Setting up Spring Maven.................... 5

# Unit Testing.......................................... 5

# Classes to be Tested............................. 5

# EmailController...................... 6

# UserDetailsController....................... 7

# IndexController.................................................... 7

# MarketListingDomainController........................... 8

# ConfirmPurchasePageController........................ 8

# TransactionDetailsPageController........................ 9

# SellerRatingTest................................................. 10

# Integration Testing...................... 11

# Black Box Testing............................ 11

# White Box Testing............................ 12

# User Interface Testing........................ 12

# Input File Testing.................................. 12

# Boundary Analysis.......................... 13

# Regression Testing......................... 13

# 

# University Depot Testing Plan

# **Overview of Testing**

# This document will describe the testing procedures for the University Depot system. Types of testing will include:

# · Human Factor Testing

# · Unit Testing

# · Integration Testing

# · Black Box Testing

# · White Box Testing

# · Boundary Analysis

# · Regression Testing

# Assumptions: We assume that the interface capabilities of Java Spring Maven have been tested and will not be included in this document.

# **Instructions**

# Please follow each step checking off when completed. Unique instructions will be provided for each test type. Any question, problems, or discrepancies may be noted in the margins or on the reverse side of this test plan.

# **Setting up Spring Maven**

## **Registering Spring**

### Default Setup

# \_\_1. Click on this link: [Spring Initializr](https://start.spring.io/)

# \_\_2. Under Project, select Maven Project

# \_\_3. Under Language, select Java

# \_\_4. Under Spring Boot, select 2.7.5

# \_\_5. Under Project Metadata, create a Group name, a Name and a Package name

# \_\_6. Under Packaging, select Jar

# \_\_7. Under Java, select 17

# \_\_8. Under Dependencies, add any dependencies relevant to the project

# \_\_9. Click Generate

# Unit Testing

# Unit testing involves creating tests to assure each method is functioning correctly for each unit or class. Our group used a mix of Junit testing and SpringBoot testing.

# **Classes to be Tested**

This is only a small percentage of the classes to be tested

## **EmailControllert**

# · Test getJavaMailSender

# o Test to see if the mail is loaded and sent correctly

# o If it throws an exception if matches false

# · Test verificationEmail

# o Tests to see if a user got a verification email with a code

# o Throws an exception if the verification doesn’t load

# · Test messageEmail

# o Test to see if the email was sent and received to the user

# o Throws an exception if the message doesn’t send or isn’t received

# · Test usernameRecovery

# o Tests to see that a user can recover their username

# o Throws an exception if it can’t find the email attached to the username

# · Test applicationAccepted

# o Tests to see if the application loads and matches with what the test creates for it

# o Throws an exception if one of the strings don’t match up

# · Test applicationRejected

# o Tests to see if the application loads and matches with what the test creates for it

# o Throws an exception if one of the strings doesn’t load

# · Test updateTicketStatus

# o Tests to see if the system can update the ticket

# o Throws an exception if the ticket info doesn’t load

## **UserDetailsController**

# · Test updatePaymentDetailsSuccess

# o Test each results of each valid field one by one in order to ensure that each field has working validation

# · Test updatePaymentDetailsFailure

# o Test each results of each invalid field one by one in order to ensure that each field has working validation

# · Test updateDirectDepositDetailsSuccess

# o Tests that updateDepositDetails can add new DirectDepositDetails

# · Test updateDirectDepositDetailsFailure

# o Tests that invalid PaymentDetails submissions are rejected

## **IndexController**

# · Test indexStatusTests

# o Tests that invalid indexes are rejected

# · Test applicationPostTest

# o Tests that invalid applications are rejected

## **MarketListingDomainController**

# · Test purchaseQuantityValid

# o Tests the marketListingPurchaseUpdate function when the quantity purchased is valid

# · Test PurchaseQuantityInvalid

# o Tests the marketListingPurchaseUpdate function when the quantity purchased is invalid

# · Test PurchaseQuantityListingDeleted

# o Verifies that marketListingPurchaseUpdate fails when the listing is deleted

## **ConfirmPurchasePageController**

# · Test OpenPage

# o Tests that the page opens properly

# · Test CCPurchaseSuccess

# o Tests that a purchase is successful when using valid payment details

o @throws Exception is thrown if the test fails

# · Test CCPurchaseFailure

# o Tests that a purchase fails for all invalid cases

# o @throws Exception is thrown if the test fails

# · Test PayPalPurchaseSuccess

# o Tests that a Paypal purchase is successful if the Paypal details are valid

o @throws Exception is thrown if the test fails

# · Test PayPalPurchaseFailure

# o Tests that a Paypal purchase fails if the Paypal details are invalid

o @throws Exception is thrown if the test fails

# · Test CancelPurchase

# o Tests that cancel purchase is functioning

o @throws Exception is thrown if the test fails

## **TransactionDetailsPageController**

# · Test openPage

# o Tests the the page opens successfully assuming the required data is set

# o @throws Exception

# · Test submitShippingUpdateSuccess

# o Tests that the shipping information is updated successfully when valid information is provided

# o Tests from the perspective of the seller

# o @throws Exception

# · Test submitShippingUpdateFailure

# o Tests that all of the failure conditions for updating shipping information prevent updates from being made

# o Tests from the perspective of the seller

# o @throws Exception

# · Test deleteTransactionSuccessful

# o Tests that the Transaction can be deleted under valid conditions, and that the deletion is applied

# o @throws Exception

# · Test deleteTransactionFailure

# o Tests that failure conditions for deleting a Transaction are successful, and that the deletion is not applied

# o @throws Exception

## **SellerRatingTest**

# · Test id

# o Test to see if the id is successfully added

# · Test ratingNameTest

# o Test to see if the name is successfully added

# · Test minPercent

# o Test to see if the minimum percentage is added

# · Test maxPercent

# o Test to see if the maximum percentage is added

# Integration Testing

# **Our Testing Method**

# This type of testing involves various modules/classes of the software that are developed as a group to see if they function together or not. Our group has been using the Incremental Approach which is where files are tested in increments. Our group has been testing using the Bottom-Up method which tests lower-level modules before high-level ones. This has made testing more efficient and comprehensible for us.

# Black Box Testing

# **Our Testing Method**

# This type of testing involves testing a system with absolutely no prior knowledge of its internal workings. When the group first got the project we observed how the code reacted to what buttons we pressed or what we clicked on. Our main goal was to identify how the system responded and what we could do as a group in order to mitigate it. We tried our best to simulate a user by clicking on anything we could in order to attempt to break the system. As testers, having the user’s point of view in the forefront is one of the most important aspects of testing to us.

# White Box Testing

# **User Interface Testing**

# Intensive testing will be executed to push the available buttons in every order possible in order to crash the system.

# **Input File Testing**

# The Input file will be manipulated in order to test the parsing functions and the feasibility of the problem.

# Boundary Analysis

# **Master Server Boundary Testing**

# Test to see what would happen if multiple instances of the master server class were running on one system. Our current boundary is one master server running at a time.

# **Solution Boundary Testing**

# Test to see how many problems you can solve at one time. Our current boundary is one problem per instance of the master server.

# **Customer Population Testing**

# Test to see the maximum number of customers that the system can solve for in one problem. The maximum currently tested is 100 as there are no data files readily available for more customers than that.

# Regression Testing

# **Changes to Inter-Agent Communication Code**

# For all changes to code that communicates with other agents, Unit testing and Integration testing will be run to see if there are no errors.

# **Changes to Intra-Agent Communication Code**

# For all changes that do not affect the communication between two or more agents, only Unit testing will be done

# **Changes to User Interface Code**

# For all changes to the user interface, Human Factor and White Box testing will be done to test the validity of the new code.

# 