SUPER HEROES IN TRAINING

My Academic Pyramid

Test Plan

CECS 491A Sec 05

December 11, 2018

Team Leader: Krystal Leon, 013986607
Arturo Peña Contreras, 010914811
Luis Julian, 007472593
Hyunwoo Kim, 014392909
Victor Kim, 012016990
Trong Nguyen, 016208983

Revision History

Date	**Version**	**Description**
12/11/18	1.0	First draft.

Table of Contents

1 Introduction

This document provides a description of the tests that are used in My Academic Pyramid web application.

1.1 Purpose of the Document

This document states what kind of testing methodologies are used for some specific features of My Academic Pyramid Web application. Additionally, all of possible scenarios are listed and they are being tested using selected testing methodologies.

2 Scope

2.1 Unit Testing

Each functionality of the Authorization, Password Checking, and User Management features will be tested.

2.1.1 Authorization Functions

- *bool CheckClaims(List<Claim> requiredClaims):*
    - Checks that user has the required claim in the requiredClaims. It would throw the exception, if the requireClaims is null. If the required claim is in the requiredClaims, it would return true and user would be able to use the feature that

user requested to use. If the required claim is not in the requiredClaims, it would return false and user wouldn't be able to use the feature.

- *int FindHeight(User user)*
    - Finds the level of the user by traversing back to the root using the referenced parent Id.
- *bool HashHigherPrivilege(User callingUser, User targetedUser)*
    - Checks if the user who made the request is at a higher level than the targeted user.

2.1.2 Password Checking Functions

- *PasswordStatus CheckPasswordCount(int count)*
    - Checks the vulnerability of a password based on the number of times it has been breached.
- *int FindHash(string hashValue, Dictionary<string, int> hashes)*
    - Finds a specified hash value within a dictionary as a key, and its count as its value.
- *string GetHashValue(string input)*
    - Generates a hash value using a SHA1 hash function.
- *Dictionary<string,int> JsonToDictionary(string list)*
    - Deserializes a JSON string into a dictionary object.
- *Task<string> RequestData(Uri uri)*
    - An HTTP GET Request of the specified uri.

2.1.3 User Management Functions

- *void CreateUser(User user)*
    - Creates a user account.
- *void DeleteUser(User user)*
    - Deletes a user account.
- *void UpdateUser(User user)*
    - Updates a user account
- *void FindUserbyUserName(string userName)*
    - Finds a user by providing a user name.
- *Void AddClaim(User user, Claim claim)*
    - Adds a claim to the user account.
- *void RemoveClaim(User user, Claim claim)*
    - Removes a claim to the user account.

2.1.4 User Tree Functions

- *bool AddChild(User user)*
    - Adds a direct child to the node by specifying a user.
- *bool AddChild(Node user)*
    - Adds a direct child to the node by specifying a user node.
- *int Depth()*
    - Gets the depth of a node in a tree.
- *Node FindUser(User user)*

    - This finds a user below the current node.
- *Node FindNodeAtLevel(int level)*
    - Finds node based on a specified depth.
- *bool IsDirectParentOf(User user)*
    - Checks if current node is a direct parent of a user.
- *int CompareTo(Node user)*
    - Compares the depth of current node to another node.
- *bool IsAbove(Node user)*
    - Checks if current node's depth is lower than another node's
- *void PrintTree(string indent, bool last)*
    - Prints the tree in the console for demo and testing purposes.

2.2 Integration testing

Not available

2.3 End to end testing

Not available

2.4 Performance testing

The runthrough of the Password Checking feature as a whole will be tested.

2.4.1 Password Checking Functions

- *PasswordStatus Validate(string password)*
    - Validates the security of a password based on the number of times it has been breached according to the Pwned Passwords service.

2.5 Penetration testing

Not available

3 Test Data

The repeatable data sets that were used to test each feature.

3.1 Password Checking

1. SHA1HashFunction sha = new SHA1HashFunction()

    An instance of the SHA1 hash function.

2. string url = “https://api.pwnedpassword.com/range/”

    The url for the Pwned Passwords API service.

3. PwnedPasswordsValidation pv = new PwnedPasswordsValidation(sha, url)

    An instance of Pwned Passwords password validation implementation.

4. IHttpClient HttpClientMethods = new HttpClientString()

    An instance of and Http Client.

5. string hashValue = “1E4C9B93F3F0682250B6CF8331B7EE68FD8”

The hash value for the string ‘password’. This password is expected to be breached.

3.2 User Tree

1. *Node root = new Node(“Root”)*

   The root node for a user tree.

2. *User krystal = new User(“Krystal”)*

   A user of the user tree.

3. *User luis = new User(“Luis”)*

   A user of the user tree.

4 Test Case Scenarios

4.1 Authorization

1. *AuthorizationManager_CheckClaims_ClaimFoundShouldReturnTrue()*

   If a claim exists, it should be found and return true.

2. *AuthorizationManager_CheckClaims_ClaimNotFoundShouldReturnFalse()*

   If a claim does not exist, it should not be found and return false.

3. *AuthorizationManager_CheckClaims_DuplicatedRightClaimShouldReturnTrue()*

   If duplicate claims exists, it should still be found and return true.

4. *AuthorizationManager_CheckClaims_DuplicatedWrongClaimShouldReturnFalse()*

If a claim does not exists, it should not be found, even when other claims are duplicated and return false.

5. *AuthorizationManager_CheckClaims_MultipleClaimFoundShouldReturnTrue()*

   If multiple matching claims exists, they should all be found and return true;

6. *AuthorizationManager_CheckClaims_HasPointsClaimNotFoundShouldReturnFalse()*

   While checking for multiple claims, if at least one does not exist, it should not be found and return false.

7. *AuthorizationManager_CheckClaims_MultipleClaimNotFoundTwoClaimShouldReturnFalse()*

   While checking for multiple claims, if more than one does not exist, it should not be found and return false.

8. *AuthorizaionManager_CheckClaims_DuplicatedMultipleClaimFoundShouldReturnTrue()*

   While checking for multiple claims, if all exist, they should be found and return true.

9. *AuthorizationManager_FindHeight_ShouldReturnCorrectLevel()*

   If the user exists, the correct height should be returned.

10. *AuthorizationManager_FindHeigth_NullUserShouldThrowException()*

    If a null user is passed, throw an exception.

11. *AuthorizationManager_HasHigherPrivilege_ShouldReturnTrue(User parent, User child)*

    If the parent has a higher privilege than the child, return true.

12. *AuthorizationManager_HasHigherPrivilege_ShouldReturnFalse(User parent, User child)*

    If the parent does not have a higher privilege than the child, return false;

13. *AuthorizationManager_HasHigherPrivilege_NullUserShouldThrowException()*

    If one null parent or child is passed, throw an exception.

14. *AuthorizationManager_HasHigherPrivilege_BothUSersNullShouldThrowException()*

    If both null parent and null child are passed, throw an exception.

4.2 Password Checking

1. *PwnedPassword_FindHash_FoundShouldReturnCount()*

    If the hash value is found as a key in the dictionary, return the count value.

2. *PwnedPassword_FindHash_NotFoundShouldReturnZero()*

    If the hash value is not found as a key in the dictionary, return zero.

3. *PwnedPasswordValidation_FindHash_InvalidHashValueShouldThrowException()*

    If a null hshValue parameter is passed, throw an exception.

4. *PwnedPasswordValidation_FindHashValue_InvalidHashesShouldThrowExecption()*

    If a null hashes parameter is passed, throw an exception.

5. *PwnedPasswordValidation_FindHash_AllNullValuesShouldThrowException()*

    If all null parameters are passed, throw an exception.

6. *SHA1HashFunctions_GetHashValue_ValidStringShouldReturnHashValue()*

    If a valid string input is passed, return the correct SHA1 hash value.

7. *SHA1HAshFunctions_GetHashValue_InvalidStringShouldReturnException()*

If a null string parameter is passed, throw an exception.

8. *PwnedPassword_JsonToDictionary_ValidStringShouldReturnDictionary()*

    If a correctly formatted JSON string is passed, return a dictionary of hash values and their counts.

9. *PwnedPassword_JsonToDictionary_InvalidStringShouldThrowException()*

    If an incorrectly formatted JSON string is passed, throw an exception.

10. *PwnedPassword_JsonToDictionary_NullStringShouldThrowException()*

    If a null string is passed, throw an exception.

11. *HttpClientString_RequestData_ShouldPass()*

    If a valid uri is passed, return the string response.

12. *HttpClientString_RequestData_InvalidUrlShouldFail()*

    If an invalid uri is passed, throw an exception.

13. *PasswordCheckingBR_CheckPasswordCount_ShouldReturnValidStatus(PasswordStatus e, PasswordStatus a)*

    If a valid count is passed, return the correct password status.

14. *PwnedPasswordsValidation_Validate_ShouldBeFasterThanOneSecond()*

    The feature as a whole should be faster than one second.

4.3 User Management

1. *UserManagementServices_Constructor_ShouldReturnArgumentNullException()*

    If a null argument is passed in the constructor, throw an exception.

2. *UserManagementServices_CreateUser_ShouldAbleFindUserAfterCreation()*

   A user should exist after being created.

3. UserManagementServices_CreateUser_NullUserShouldRaiseExcception()

   If a null user is passed, throw an exception.

4. *UserManagementServices_CreateUserWIthDupplicateUserName_ShouldReturnOnlyOne UserObjectWhenFinding()*

   If creating a duplicate user is attempted, throw an exception.

5. *UserManagementServices_DeleteUser_UserShouldNotExistAfterDeletion()*

   A user should not exist after being deleted.

6. *UserManagementServices_DeleteUser_NullUserShouldRaiseException()*

   If a null user is passed, throw an exception.

7. *UserManagementServices_DeleteUser_UserNotFoundShouldRaiseException()*

   If the user is not found, no user is deleted and an exception is thrown.

8. *UserManagementServices_UpdateUser_UserNameShouldGetUpdate()*

   A user's username should correctly change after being updated.

9. *UserManagementServices_AddClaim_ClaimsShouldBeAddToUser()*

   Claims should accurately exist with a user after being added to the user.

10. *UserManagementServices__RemoveClaim_ClaimsShouldBeRemovedFromUser()*

   A claim should not exist with a user after being deleted from the user.

4.4 User Tree

1. *Node_AddChild_InputUser_UniqueChildShouldReturnTrue()*

If a user does not already exist in the tree, it should be added.

2. *Node_AddChild_InputUser_DuplicateChildShouldReturnFalse()*

   If a user already exists in the tree, it should not be added.

3. *Node_AddChild_InputUser_NullInputShouldThrowException()*

   If a null user is passed, throw an exception.

4. *Node_AddChild_InputNode_UniqueChildShouldReturnTrue()*

   If a user does not already exist in the tree, it should be added.

5. *Node_AddChild_InputNode_DuplicateChildShouldReturnFalse()*

   If a user already exists in the tree, it should not be added.

6. *Node_AddChild_InputNode_NullInputShouldThrowException()*

   If a null user is passed, throw an exception.

7. *Node_Depth_RootShouldReturnZero()*

   The depth of the root node should return zero.

8. *Node_Depth_ChildShouldReturnValidDepth()*

   The depth of any child should return the correct depth.

9. *Node_Depth_NullShouldThrowException()*

   If a null node is passed, throw an exception.

10. *Node_FindUser_ExistingUserShouldReturnUser()*

    If a user is found in the tree, the user should be returned.

11. *Node_FindUser_NonExistingUserShouldReturnNull()*

    If a user is not found in the tree, return a null user.

12. *Node_FindUser_NullShouldThrowException()*

Passing a null user should throw an exception.

13. *Node_FindNodeAtLevel_ValidShouldReturnNode()*

    If a valid existing depth is passed, the first node found existing at that depth should be returned.

14. *Node_FindNodeAtLevel_InvalidShouldReturnNull()*

    If an invalid depth is passed, return a null node.

15. *Node_IsDirectParentOf_DirectParentShouldReturnTrue()*

    If the current node is the direct parent of the passed node, return true.

16. *Node_IsDirectParentOf_NotDirectParentShouldReturnFalse()*

    If the current node is not the direct parent of the passed node, return false.

17. *Node_IsDirectParentOf_NullShouldReturnFalse()*

    If a null node is passed, return false.

18. *Node_CompareTo_LesserDepthShouldReturnPositive()*

    If the depth of the current node is less than the depth of the passed node, return a positive number.

19. *Node_CompareTo_GreaterDepthShouldReturnNegative()*

    If the depth of the current node is greater than the depth of the passed node, return a negative number.

20. *Node_CompareTo_EqualDepthShouldReturnZero()*

    If the depth of the current node is equal to the depth of the passed node, return zero.

21. *Node_CompareTo_NullShouldThrowException()*

If a null node is passed, throw an exception.

22. *Node_IsAbove_AboveShouldReturnTrue()*

    If the current node's level is above the passed node, return true.

23. *Node_IsAbove_BelowShouldReturnFalse()*

    If the current node's level is below the passed node, return false;

24. *Node_IsAbove_NullShouldThrowException()*

    If a null node is passed, throw an exception.

25. *Tree_AddUser_ByParent_ExistingParentShouldReturnTrue()*

    If the user does not already exist in the tree and the parent does exist, add the user to the tree.

26. *Tree_AddUser_ByParent_NonExistingParentShouldReturnFalse()*

    If the parent does not exist in the tree, do not add the user.

27. *Tree_AddUser_ByParent_NullShouldThrowException()*

    If a null parent is passed, throw an exception.

28. *Tree_AddUser_ByLevel_ValidLevelShouldReturnTrue()*

    If the user does not exist in the tree and the level is valid and exists in the tree, add the user.

29. *Tree_AddUser_ByLevel_InvalidLevelReturnFalse()*

    If the level does not exist in the tree, do not add the user.

30. *Tree_DeleteUser_ExistingUserShouldReturnTrue()*

    If the user exists in the tree, delete the user.

31. *Tree_DeleteUser_NonExistingUserShouldReturnFalse()*

If the user does not exists in the tree, do not delete the user.

32. *Tree_DeleteUser_NullInputShouldThrowException()*

If a null user is passed, throw an exception.