**C868 – Software Capstone Project Summary**

**Task 2 – Section C: Testing Plan and Results**



|  |  |
| --- | --- |
| **Capstone Proposal Project Name:** | FS InfoCat – File System Information Cataloging Tool |
| **Student Name:** | Leonard T. Erwine |

Table of Contents

[Test Plan 2](#_Toc83555265)

[Introduction 2](#_Toc83555266)

[Testing External Assumptions 3](#_Toc83555267)

[Data Driven Unit Testing 3](#_Toc83555268)

[Test Implementation 4](#_Toc83555269)

[Deliverables / Outcomes 5](#_Toc83555270)

[Volume Identifier Data Type Unit Test 5](#_Toc83555271)

[Test Overview 5](#_Toc83555272)

[Test for Construction from VSN 6](#_Toc83555273)

[Test for Construction from UUID 7](#_Toc83555274)

[Test for Construction from URI object 8](#_Toc83555275)

[Serialization Test 10](#_Toc83555276)

[Reconstruction Test 11](#_Toc83555277)

[Volume Entity Unit Tests 13](#_Toc83555278)

[Unit Test Overview 13](#_Toc83555279)

[Empty Display Name Test 13](#_Toc83555280)

[Empty Volume Identifier Test 15](#_Toc83555281)

[File System Entity Omission Test 17](#_Toc83555282)

# Test Plan

## Introduction

I operate on a philosophical basis that something can only be as good as the sum of its parts. Therefore, I decided to take a top-down approach when considering possible subjects for unit tests, and a bottom-up approach for prioritization. It made sense to me that I would test the reliability of any dependencies before I tested the dependent entity. Dependencies which do not serve as inputs to the logical decision making within the application have diminished priority.

Another consideration factored into the testing plan is the factors that indicate when a unit test needs to be re-executed. There are three generalized reasons by which I determine that a unit test needs to be executed: Direct change, change within a dependency, and environmental change. The first and obvious indicator is when there has been a direct change to the subject of the unit test. Next, unit tests which involve a dependency upon the subject of the latter should be performed as well. Additionally, all unit tests should be reconducted after any updates or modifications to the overall programming environment.

### Testing External Assumptions

I have also created some unit tests which do not directly test any aspect of the application, but rather tests assumptions about dependent systems and Application Programming Interfaces (APIs). Documentation for the behavior and results of system components and APIs external to the application are sometimes incomplete or it was not feasible to include all possible scenarios. These kinds of tests serve to assert that I, the programmer, correctly understands the technology being utilized. Additionally, tests that validate assumptions can be useful to detect when the behavior of external dependencies differs from past results and may have an impact on the currently released version of the application.

### Data Driven Unit Testing

Many of the tests employed the concept of data driven unit testing. These tests are used in cases where it is necessary to evaluate multiple combinations of input parameters. Creating separate unit tests for each variance of the input parameters would be difficult to maintain. Additionally, it would result in many separate test result entries for the same subject, making collective test results more difficult to read. Embedding multiple assertions within a single test would still be disadvantageous because the test method would be lengthy and difficult to maintain, with increased risk of human error. By using data driven testing, the test code is easier to maintain and makes it easier to logically group test results by subject.

### Test Implementation

I decided to use MS Test, which a testing framework created by Microsoft that requires minimal setup and configuration and is useful for resolving failures while also having the ability to produce reports that are meaningful to the customer. In the planning phase skeleton methods were created for all unit tests. Test methods were annotated with attributes that indicated the priority of that unit test. By doing so, the test reports will show an accounting of all unit tests, including which ones have been implemented. Tests which have not yet been implemented are annotated with an attribute which tells the test evaluation engine to skip that test.

Due to time limitations, not all unit tests were implemented. In the final part of the planning phase, I determined which tests were required so I could be reasonably confident that unit test results would indicate releasability. All other unit tests were annotated with the attribute that causes it to be skipped, which I will remove after implementation.

Unit tests are placed in the unit test project. Each test class should have only one object type as its primary target type. Individual unit test methods are implemented as test methods within the test class:

1. Create a static method which returns the test parameters. This method should return an enumerable series of object array. The values in the object array will be passed to the test method. If the unit test is not data driven, this step will be omitted.
2. Create the instance method that will implement the unit test. For data driven tests, create parameters that match test input data types. Otherwise, there should be no parameters.
3. Implement the test code using assertions.
4. From the ‘Test’ dropdown menu, select ‘Run All Tests’, which will compile the solution and discover newly added tests.
5. Open the test explorer window to examine the results.

## Deliverables / Outcomes

When unit tests are conducted, the project is configured to record the results in two different files. The first file, named “TestResults.trx”, is an XML-based log file which can be consumed by other developer tools as needed. The second file, named “TestResults.html”, is a human-readable report that begins with a summary and is followed by the detailed results.

# Volume Identifier Data Type Unit Test

This is an example of a unit test where the subject is a dependency of a database entity. The custom Volume Identifier data type is one of the most critical unit test targets. This data type is responsible for correlating file storage volumes to entities within the Volume database table. The Volume Identifier type is an abstraction of the actual value by which the storage volume uniquely identifies itself. Any failure to accurately translate and distinguish volume-native identifier types and values would make the entire application unusable. Furthermore, this is translated to a URI string when it is stored the database, so conversion to and from the database-native value needs to be thoroughly tested as well.

## Unit Test Overview

All Volume Identifier unit tests that produce a constructed “VolumeIdentifier” value use the same input data type to indicate the expected result. Any other test input parameters are used as part of the evaluation. The property named “IsArgumentOutOfRangeException” is set to true when the input argument should cause the subject of the test to throw an exception. Otherwise, the remaining properties correspond to the expected values of identically named properties on the result “VolumeIdentifier” value.

public record ConstructorResultExpected

{

    public bool IsArgumentOutOfRangeException { get; }

    public bool IsEmpty { get; }

    public uint? SerialNumber { get; }

    public Guid? UUID { get; }

    public Uri Location { get; }

}

Figure 1 - Volume Identifier Unit Test Expectation Parameter Type

For the volume identifier, there will be 5 conversions that need to be tested. The first test verifies that a VolumeIdentifier value can be created from a 32-bit unsigned integer value. The second test asserts that a VolumeIdentifier value can be created from a UUID value. The third test ensures that a VolumeIdentifier value can be created from a URI object. The fourth test verifies that the Volume Identifier data type is property serialized as a URI string for storage in the database. The final test will verify that URI strings will reconstruct the Volume Identifier data type as expected.

## Test for Construction from VSN

This test will construct a “VolumeIdentifier” object from a 32-bit unsigned integer value, which is the same type that is used as Volume Serial Numbers. There will be 3 input values applied to this test: A zero value, the maximum possible 32-bit unsigned integer value and an arbitrary where none of the bytes that make up the 32-bit value are the same. A second value will be passed to the test method which contains the expected result property values. The test passes when all expected values match. Figure 2 is a snippet of code containing the method that implements this test.

public static IEnumerable<object[]> GetConstructorSerialNumberTestData()

{

    yield return ConstructorResultExpected.CreateTestData(0x094f622a5u);

    yield return ConstructorResultExpected.CreateTestData(uint.MaxValue);

    yield return ConstructorResultExpected.CreateTestData(uint.MinValue);

}

[DataTestMethod, Priority(1)]]

[DynamicData(nameof(GetConstructorSerialNumberTestData), DynamicDataSourceType.Method)]

public void ConstructorSerialNumberTestMethod(uint serialNumber,

        ConstructorResultExpected expected)

{

    VolumeIdentifier volumeIdentifier = new(serialNumber);

    Assert.IsFalse(volumeIdentifier.IsEmpty());

    Assert.IsNotNull(volumeIdentifier.Location);

    Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.Location.ToString());

    Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.ToString());

    Assert.AreEqual(expected.SerialNumber, volumeIdentifier.SerialNumber);

    Assert.IsFalse(volumeIdentifier.UUID.HasValue);

}

Figure 2 - Unit test code for constructor that takes a VSN

## Test for Construction from UUID

This constructs a VolumeIdentifier from a 128-bit UUID that is stored in a Guid object. There will be three input values applied to this test: A UUID with all bits unset, another with all bits set, and another whereby none of the byte values comprising the UUID have the same value. A second value will be passed to the test method containing the expected result property values. The test passes when all expected values match. Figure 3 is a snippet of code containing the method that implements this test.

public static IEnumerable<object[]> GetConstructorUuidTestData()

{

    yield return ConstructorResultExpected.CreateTestData(

        Guid.Parse("91502fe2-cb4b-4274-a8ad-8b70074132c3"));

    yield return ConstructorResultExpected.CreateTestData(

        Guid.Parse("ffffffff-ffff-ffff-ffff-ffffffffffff"));

    yield return ConstructorResultExpected.CreateTestData(Guid.Empty);

}

[DataTestMethod, Priority(1)]]

[DynamicData(nameof(GetConstructorUuidTestData), DynamicDataSourceType.Method)]

public void ConstructorUuidTestMethod(Guid uuid,

        ConstructorResultExpected expected)

{

    VolumeIdentifier volumeIdentifier = new(uuid);

    Assert.IsFalse(volumeIdentifier.IsEmpty());

    Assert.IsNotNull(volumeIdentifier.Location);

    Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.Location.ToString());

    Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.ToString());

    Assert.AreEqual(expected.UUID, volumeIdentifier.UUID);

    Assert.IsFalse(volumeIdentifier.SerialNumber.HasValue);

}

Figure 3 - Unit test code for constructor that takes a UUID

## Test for Construction from URI object

In this unit test, a VolumeIdentifier is constructed from a URI object. This has a large number of test input sets where character casing is modified, and different URI schemes are evaluated. This test passes when all the value of the object representing expected results match the corresponding properties of the target VolumeIdentifier. Figure 4 shows the code which implements this test.

public static IEnumerable<object[]> GetConstructorUriTestData()

{

    yield return ConstructorResultExpected.CreateEmptyTestData((Uri)null);

    yield return ConstructorResultExpected.CreateEmptyTestData(new Uri("", UriKind.Relative));

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateTestData(

        new Uri("urn:uuid:91502fe2-cb4b-4274-a8ad-8b70074132c3", UriKind.Absolute),

        Guid.Parse("91502fe2-cb4b-4274-a8ad-8b70074132c3"));

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateTestData(

        new Uri("file://desktop-10538/Users/jlynn/ANWR", UriKind.Absolute),

        new Uri("file://desktop-10538/Users/jlynn/ANWR", UriKind.Absolute));

    yield return ConstructorResultExpected.CreateTestData(

        new Uri("file://DESKTOP-10538/USERS/JLYNN/ANWR", UriKind.Absolute),

        new Uri("file://desktop-10538/USERS/JLYNN/ANWR", UriKind.Absolute));

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateArgumentOutOfRangeExceptionTestData(

        new Uri("file://desktop-10538", UriKind.Absolute));

    yield return ConstructorResultExpected.CreateArgumentOutOfRangeExceptionTestData(

        new Uri("file:///desktop-10538/Users/jlynn/ANWR", UriKind.Absolute));

    … Omitted for brevity …

}

[DataTestMethod, Priority(1)]]

[DynamicData(nameof(GetConstructorUriTestData), DynamicDataSourceType.Method)]

public void ConstructorUriTestMethod(Uri uri, ConstructorResultExpected expected)

{

    if (expected.IsArgumentOutOfRangeException)

        Assert.ThrowsException<ArgumentOutOfRangeException>(() => new VolumeIdentifier(uri));

    else

    {

        VolumeIdentifier volumeIdentifier = new(uri);

        Assert.AreEqual(expected.IsEmpty, volumeIdentifier.IsEmpty());

        Assert.IsNotNull(volumeIdentifier.Location);

        Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.Location.ToString());

        Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.ToString());

        Assert.AreEqual(expected.SerialNumber, volumeIdentifier.SerialNumber);

        Assert.AreEqual(expected.UUID, volumeIdentifier.UUID);

    }

}

Figure 4 - Unit test for constructor that takes a URI object

## Serialization Test

This test ensures that a VolumeIdentifier produces the expected, well-formed URI string for storage in the database. VolumeIdentifiers that represent a VSN should start with “urn:volume:id” followed by the two most significant bytes in hexadecimal notation, followed by a dash and then by the two least significant bytes in hexadecimal notation. Those that represent a UUID should start with “urn:uuid:” followed by the UUID format string variant which contains dashes and no curly brackets. Those that represent a UNC path should use the “file” URI schema that specifies a host, has at least one path segment beyond the root segment, and does not end with a path separator. Figure 5 contains the source code for this test.

public static IEnumerable<object[]> GetSerializedValueTestData()

{

    yield return new object[] { new VolumeIdentifier(), "" };

    yield return new object[] { new VolumeIdentifier(0x270d32a3),

        "urn:volume:id:270D-32A3" };

    yield return new object[] { new VolumeIdentifier(

        Guid.Parse("aacbef27-5451-4964-aba3-e4c2c5118a87")),

        "urn:uuid:aacbef27-5451-4964-aba3-e4c2c5118a87" };

    yield return new object[] { new VolumeIdentifier(

        new Uri("file://servicenowdiag479.file.core.windows.net/testazureshare", UriKind.Absolute)),

        "file://servicenowdiag479.file.core.windows.net/testazureshare" };

}

[DataTestMethod, Priority(1)]]

[DynamicData(nameof(GetSerializedValueTestData), DynamicDataSourceType.Method)]

public void SerializedValueTestMethod(VolumeIdentifier target, string expected)

{

    string actual = target.ToString();

    Assert.AreEqual(expected, actual);

}

Figure 5 - Unit test for serialization to string value

## Reconstruction Test

This test ensures that a valid URI string can be used to reconstruct a VolumeIdentifier value. The first parameter passed to the test method is the URI string. The second is an object which represents the expected property values after reconstruction. This test succeeds when all property values match. Figure 5 contains the source code for this test method.

public static IEnumerable<object[]> GetParseTestData()

{

    yield return ConstructorResultExpected.CreateEmptyTestData((string)null);

    yield return ConstructorResultExpected.CreateEmptyTestData("");

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateTestData(

        "urn:volume:id:94F6-22A5", 0x094f622a5u);

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateTestData(

        "urn:uuid:91502fe2-cb4b-4274-a8ad-8b70074132c3",

        Guid.Parse("91502fe2-cb4b-4274-a8ad-8b70074132c3"));

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateTestData(

        "file://servicenowdiag479.file.core.windows.net/testazureshare",

        new Uri("file://servicenowdiag479.file.core.windows.net/testazureshare", UriKind.Absolute));

    … Omitted for brevity …

    yield return ConstructorResultExpected.CreateArgumentOutOfRangeExceptionTestData(

        "file://desktop-10538");

    yield return ConstructorResultExpected.CreateArgumentOutOfRangeExceptionTestData(

        "file:///Users/jlynn/ANWR");

    … Omitted for brevity …

}

[DataTestMethod, Priority(1)]

[DynamicData(nameof(GetParseTestData), DynamicDataSourceType.Method)]

public void ParseTestMethod(string text, ConstructorResultExpected expected)

{

    if (expected.IsArgumentOutOfRangeException)

        Assert.ThrowsException<ArgumentOutOfRangeException>(() =>

            VolumeIdentifier.Parse(text));

    else

    {

        VolumeIdentifier volumeIdentifier = VolumeIdentifier.Parse(text);

        Assert.AreEqual(expected.IsEmpty, volumeIdentifier.IsEmpty());

        Assert.IsNotNull(volumeIdentifier.Location);

        Assert.AreEqual(expected.Location.ToString(),

            volumeIdentifier.Location.ToString());

        Assert.AreEqual(expected.Location.ToString(), volumeIdentifier.ToString());

        Assert.AreEqual(expected.SerialNumber, volumeIdentifier.SerialNumber);

        Assert.AreEqual(expected.UUID, volumeIdentifier.UUID);

    }

}

Figure 6 - Unit test for reconstructing VolumeIdentifiers

# Volume Entity Unit Tests

This is an example of a unit test for a database entity. The Volume entity is very important for the application because this is what distinguishes one file source from another. All database entity objects support property validation. Some forms of validation, however, occur within the database, such as making sure there are no duplicate volume identifiers.

## Unit Test Overview

There will be 3 tests for this entity type. The first test checks display name validation. The second test ensures that the volume will have a valid identifier. Lastly, the third test asserts that the entity has a related file system entity.

## Test for Display Name Validation

This unit test validates two facets. First, it makes sure that the object is in an error state when the display name is empty. Second, it ensure that an exception will be thrown if an attempt is made to save an entity to the database with an empty display name. This test passes when non-empty display names can be saved to the database, and an empty display name has an error message and an exception is thrown when trying to save it to the database.

[DataTestMethod, Priority(20)]

[DynamicData(nameof(GetDisplayNameTestData), DynamicDataSourceType.Method)]

public void DisplayNameTestMethod(string displayName, string expected, string errorMessage)

{

    using IServiceScope serviceScope = Services.ServiceProvider.CreateScope();

    using LocalDbContext dbContext = serviceScope.ServiceProvider

        .GetRequiredService<LocalDbContext>();

    Microsoft.EntityFrameworkCore.Storage.IDbContextTransaction dbContextTransaction =

        dbContext.Database.BeginTransaction();

    Volume target = new() { DisplayName = displayName };

    Assert.IsTrue(target.IsChanged());

    target.FileSystem = dbContext.FileSystems.Find(

        Guid.Parse("0af7fe3e-3bc2-41ac-b6b1-310ad5fc46cd"));

    target.Identifier = new VolumeIdentifier(Guid.NewGuid());

    string actualValue = target.DisplayName;

    Assert.IsNotNull(actualValue);

    Assert.AreEqual(expected, actualValue);

    Collection<ValidationResult> validationResults = new();

    bool isValid = Validator.TryValidateObject(target, new ValidationContext(target),

        validationResults, true);

    dbContext.Volumes.Add(target);

    if (string.IsNullOrWhiteSpace(errorMessage))

    {

        Assert.IsTrue(isValid);

        dbContext.SaveChanges();

    }

    else

    {

        Assert.IsFalse(isValid);

        Assert.AreEqual(1, validationResults.Count);

        Assert.AreEqual(validationResults[0].ErrorMessage, errorMessage);

        string actualMemberName = validationResults[0].MemberNames.FirstOrDefault();

        Assert.IsNotNull(actualMemberName);

        Assert.IsFalse(validationResults[0].MemberNames.Skip(1).Any());

        Assert.AreEqual(nameof(Volume.DisplayName), actualMemberName);

        Assert.ThrowsException<AggregateException>(() => dbContext.SaveChanges());

    }

}

Figure 7 - Display name validation unit test

## Test for Volume Identifier Validation

This unit test is responsible for three general assertions. If the volume has an empty identifier, the object will be in an error state. Any attempt to save a volume entity to the database with an empty identifier must throw an exception. Finally, no duplicate volume identifiers should be allowed to exist in the database. This test succeeds when valid volume identifiers are saved to the database and empty ones have an error message and an exception is thrown when saving to the database.

[DataTestMethod, Priority(20)]

[DynamicData(nameof(GetIdentifierTestData), DynamicDataSourceType.Method)]

public void IdentifierTestMethod(string identifier, string expected, string errorMessage)

{

    using IServiceScope serviceScope = Services.ServiceProvider.CreateScope();

    using LocalDbContext dbContext = serviceScope.ServiceProvider

        .GetRequiredService<LocalDbContext>();

    Microsoft.EntityFrameworkCore.Storage.IDbContextTransaction dbContextTransaction =

        dbContext.Database.BeginTransaction();

    Volume target = new() { Identifier = string.IsNullOrWhiteSpace(identifier) ? default :

        VolumeIdentifier.Parse(identifier) };

    Assert.IsTrue(target.IsChanged());

    target.DisplayName = "Test";

    target.FileSystem = dbContext.FileSystems.Find(

Guid.Parse("0af7fe3e-3bc2-41ac-b6b1-310ad5fc46cd"));

    VolumeIdentifier expectedValue = string.IsNullOrWhiteSpace(expected) ? default :

        VolumeIdentifier.Parse(expected);

    VolumeIdentifier actualValue = target.Identifier;

    Assert.AreEqual(expectedValue, actualValue);

    Collection<ValidationResult> validationResults = new();

    bool isValid = Validator.TryValidateObject(target, new ValidationContext(target),

validationResults, true);

    dbContext.Volumes.Add(target);

    if (string.IsNullOrWhiteSpace(errorMessage))

    {

        Assert.IsTrue(isValid);

        dbContext.SaveChanges();

    }

    else

    {

        Assert.IsFalse(isValid);

        Assert.AreEqual(1, validationResults.Count);

        Assert.AreEqual(validationResults[0].ErrorMessage, errorMessage);

        string actualMemberName = validationResults[0].MemberNames.FirstOrDefault();

        Assert.IsNotNull(actualMemberName);

        Assert.IsFalse(validationResults[0].MemberNames.Skip(1).Any());

        Assert.AreEqual(nameof(Volume.Identifier), actualMemberName);

        Assert.ThrowsException<AggregateException>(() => dbContext.SaveChanges());

    }

}

## Test for File System Association Validation

Related entities are lazy-loaded when using the EF Core entity framework, which means that the existence of a related file system type cannot be verified before changes are saved to the database. Nevertheless, the database is supposed to throw an exception if a volume entity does not have a related file system type. This test passes when a volume which an associated file system type can be saved to the database, and an attempt to save a volume without a file system type fails with an exception.

[TestMethod("FileSystem FileSystem"), Priority(20)]

public void FileSystemTestMethod()

{

    using IServiceScope serviceScope = Services.ServiceProvider.CreateScope();

    using LocalDbContext dbContext = serviceScope.ServiceProvider

        .GetRequiredService<LocalDbContext>();

    Microsoft.EntityFrameworkCore.Storage.IDbContextTransaction dbContextTransaction =

        dbContext.Database.BeginTransaction();

    Guid expectedFileSystemId = Guid.Parse("88a3cdb9-ed66-4778-a33b-437675a5ae38");

    Volume target = new() { FileSystemId = expectedFileSystemId };

    Assert.IsTrue(target.IsChanged());

    Assert.IsNull(target.FileSystem);

    target.DisplayName = "Test";

    target.Identifier = new VolumeIdentifier(Guid.NewGuid());

    Guid actualId = target.FileSystemId;

    Assert.AreEqual(expectedFileSystemId, actualId);

    dbContext.Volumes.Add(target);

    dbContext.SaveChanges();

    expectedFileSystemId = Guid.Parse("bd64e811-2c25-4385-8b99-1494bbb24612");

    FileSystem expectedValue = dbContext.FileSystems.Find(expectedFileSystemId);

    target.FileSystem = expectedValue;

    actualId = target.FileSystemId;

    FileSystem actualValue = target.FileSystem;

    Assert.IsNotNull(actualValue);

    Assert.AreEqual(expectedFileSystemId, actualId);

    Assert.AreSame(expectedValue, actualValue);

    dbContext.Volumes.Update(target);

    dbContext.SaveChanges();

    target.FileSystem = null;

    dbContext.Volumes.Update(target);

    Assert.ThrowsException<DbUpdateException>(() => dbContext.SaveChanges());

}

# References

**There are no sources in the current document.**