Mick

[Email address]

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

Arowana Design

Requirements and Specification

Table of Contents

[Overview 1](#_Toc506001492)

[Technologies 1](#_Toc506001493)

[IDE 1](#_Toc506001494)

[Storage 1](#_Toc506001495)

[Appendix 1. 2](#_Toc506001496)

[Build Tools 2](#_Toc506001497)

[xUnit 2](#_Toc506001498)

[Cake (C# Make) 2](#_Toc506001499)

[Rationale 3](#_Toc506001500)

[Continuous Integration 3](#_Toc506001501)

[Release Builds 4](#_Toc506001502)

[Unit Tests 5](#_Toc506001503)

[Difficulties when coding using unit tests 5](#_Toc506001504)

[Integration Tests 6](#_Toc506001505)

[Acceptance Tests 6](#_Toc506001506)

[Appendix 2. 8](#_Toc506001507)

# Overview

# Technologies

The goal of this project is to have a set of projects that will work on both .NET Framework 4.6 and .NET Core 2.0. This doesn’t mean that all of the functionality is available in each framework but it should have similar functionality. For instance, .NET 4.6 will make use of the Windows Services functionality for the background applications. The .NET Core will need to use daemon services on non-Windows based boxes. But in the end, both versions provide the same functionality even though the implementation is different.

## IDE

Microsoft Visual Studio will be used to develop

# Storage

There are two types of storage:

* Loosely packed – Individual files reside on the file system.
* Containerized – The files for the series\study are containerized within a compressed file. The file may span several files. Think of this like a compressed file without the high level of compression.

# 

This section is details the build processes and the tools used. The general principal of the build processes is to automate nearly everything possible. However, there will always be a human intervention required through out the process. For instance, the promotion to the final release will be performed by someone. Also, there will always be tests that need to be performed that are not automated or cannot be automated.

## Build Tools

### xUnit

xUnit documentation is found here: <http://xunit.github.io/>

The unit and integration tests use the xUnit framework to execute the tests. xUnit has different runners that can be used by multiple different applications.

xUnit is included in the projects using the NuGet package manager.

#### Conventions

Unit test projects are designated by naming the project as **[assemblyname].unittests**.

Integration test projects are designated by naming the project as **[assemblyname].integrationtests**.

This is done so that the build scripts can easily identify the type of tests the assembly contains so that it can run at the proper time.

## Cake (C# Make)

Cake documentation is found here: <https://cakebuild.net/>

Cake is a C# make build scripting platform. It can execute on multiple different build systems. The following link is useful if you are not running on a Windows machine. It specifies the methods to get the bootstrapper script. <https://cakebuild.net/docs/tutorials/setting-up-a-new-project>

There are two parts to the execution of the cake scripts:

* The build bootstrapper (build.ps1 or build.sh) – This is the command script that ensures that ensures all of the components required for the build have been downloaded from NuGet sources. The final step of this script is to execute the cake executable with the build.cake as the input.
* The actual C# build script (build.cake) – This is a C# file that is compiled in cake that runs the steps required for creating the build.

NOTE: Custom tasks that are specific to the build process are kept separate from the Arowana solutions and projects. They have specific functionality for the build process and should not be coupled with the Arowana solutions.

### Rationale

The choice to use Cake was so that the developer can execute the build scripts directly from their development machine as would the build server. This will hopefully allow for better build experiences.

## Continuous Integration

There are 3 different types of continuous integration (CI) builds.

* **Code commit** – Whenever a code commit is performed, the build server will build the base solution using the script detailed in the section, Build Tools. The only artifact of this build is the report that is generated in the last step. This build should perform the following 3 steps:
  + Compile the code without the installer(s).
  + Run the unit tests.
  + Generate a detailed report on acceptance of the code commit. If the build fails, emails will be sent out indicating the issues.
* **Nightly** – Every night the code will be built using the most current version of the accepted **code commit** build. If no new build has been performed since the last **nightly** build, the build will not occur. The only artifact of this build is the report that is generated in the last step. This build should perform the following 4 steps:
  + Compile the code without the installer(s).
  + Run the unit tests.
  + Run the integration tests. These tests include performance and memory profiling. This step does not install the software. It runs the tests from the development environment.
  + Generate a detailed report on acceptance of the nightly build. If the build fails, emails will be sent out indicating the issues.
* **Quality Assurance (QA) acceptance** – On the successfully completion of the **nightly** build, this build will be started using the version of the **nightly** build. The artifacts of this build are the report that is generated and the installer(s). This build should perform the following steps:
  + Compile the code generating the installer(s).
  + For each set of acceptance test groups[[1]](#footnote-1):
    - Deploy the installer(s) on the acceptance server(s) based on the test group.
    - Configure the server(s) based on the individual server requirements for the test group.
    - Run the acceptance tests within the test group. Items to monitor:
      * Test results.
      * Memory profile
        + Number of handles.
        + Number of threads.
      * Performance profile
        + CPU
        + Code hotspot detection
        + Network
        + Disk I\O
  + Generate a detailed report on the acceptance of the QA acceptance build. If the build fails, emails will be sent out indicating the issues.

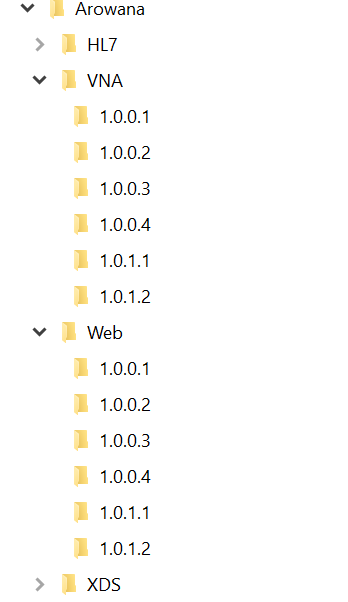
NOTE: The reports shall be stored for long term review capabilities. However, they don’t need to be online accessible. It is important to be able to look back through the reports for tends.

## Release Builds

In the release build process, the code isn’t rebuilt. It uses the artifacts from the **QA acceptance** build. The first phase of the release build process is to run manual testing. This includes the tests that deemed as ‘smoke tests’ that are not automated and functional testing of defects and new implementation features.

Once these manual tests are completed successfully, the **QA acceptance** build is ready to be published to the end-users. The final step is to manually kick of the job that copies of the QA acceptance build artifacts to the software share repository. This build does the following steps:

* Takes the most current successful **QA acceptance** builds artifacts and copies them to the QA testing folder. The structure of this folder should include the product and build version.



* Branches the source repository for the appropriate version of code based on the release build.
* Pins the NuGet packages to the versions of the packages there were used in the build. This way future patch releases will use the same version of the NuGet packages unless they are manually updated by the developer based on need and risk analysis.

## Unit Tests

Unit testing is for low level testing of individual pieces of code. Please refer online to what unit testing should accomplish and how to properly develop unit tests. One way to look at this is that you have a single method within a class that you are exercising its implementation. There might be a lot going on under the covers but you are only looking at the functionality of the method.

Typically, the hosting application of the unit test is software that is being tested. Meaning the Arowana executable(s) will not be running the code that is being tested.

General philosophy taken should be that it tests a single operation. If there are more than one operation being tested, it is likely an integration test. For instance, say you have the following implementation:

The following test isn’t a unit test as it is testing multiple different operations:

### Difficulties when coding using unit tests

To design code that is easily unit tested, the class\method under testing needs to be loosely coupled with the other instances. For example, that talk about the DICOM C-Store operation. There within the implementation there will be the need for determining the configuration, storing the file to the file system, updating database information about the file, and much more. If the code for the C-Store operation is responsible for determining the configuration like reading from the registry or connecting to the database to get information, insert, or update data, then setting up the unit test will become burdensome. However, if the C-Store method is passed interfaces that abstract this functionality, the unit test can create mock classes that will simulate the different aspects of the implementation so that the actual code within the C-Store implementation can be tested.

Reasons for unit testing include but not limited to:

* Exercise the normal functionality of an operation (method, property, constructor).
* Edge-case testing. Making sure that the edge cases that may occur are handled.
* Exception handling testing. This is likely covered in the above to bullet points but it is here because many issues occur within the exception handling code that go undetected since the exception handling code doesn’t get tested in normal operations. For instance, referencing a null-pointer within the exception handling will throw an exception and the original exception will be lost.
* Code coverage. Running the unit tests under code coverage software will high light areas of the code that the unit tests are not exercising. This can be used to help analysis the risk of the release.
* Design time critical thinking. Writing the unit tests for the implementation while writing the implementation helps in identifying weakness in the design at coding time instead of later in the process like integration.

Reasons for not creating unit tests include but not limited to:

* The code is highly coupled with other code that requires extensive mocking or setup to accomplish the task.
  + The code could be refactored to decouple the other code.
  + Manual or integration testing will be required to exercise this code.
* The implementation is very simple.
  + For instance, the class is a basic ‘structure’ holding properties for ORM to the database. However, if there is business logic that is included within the class like validation methods, these methods will need some kind of unit testing.

## Integration Tests

The integration tests are in a way the opposite of the unit tests. While the unit tests are micro in nature, the integration tests are macro in nature. For instance, performing a DICOM C-Store from the Arowana executable and verifying that the C-Store was successful, would be an example of an integration test. Instead of using the mock implementation as the unit test does, it will use the actual implementation of the configuration, file system, and database and execute the test using all of the components together.

The goal of the integration tests is to run the tests from the development environment so that developers can run them before committing code to the source control repository. The complete installation should not have to be installed on the machine to run these tests.

## Acceptance Tests

The acceptance tests are automated test that are performed on a “test system”. The software is installed on systems to represent the “system under test”. Configured using a set of known information that is specific to the tests. And then the series of test instructions are executed and the desired outcome is monitored.

These types of tests may take an extremely long time to execute all of the tests. Therefore, the following are needed within the functionality of this build job:

* A way to track the nightly build that is being tested so that the promoted of this build grabs the proper artifacts.
* A way to cancel the job so that a new nightly build can be tested instead.
* This job will execute mutually exclusive to itself. Meaning that there will be only one instance of the job running at a time.
* If multiple nightly jobs are successfully while this job is executing, the next time this job runs it will use the most current successful nightly job as the testing point. The other nightly jobs will never have the acceptance test executed on them.

These tests are a sanity check so that QA personnel are not wasting time testing a fundamentally flawed build. At a minimum these tests:

* Basic installation – required files are present and at the correct version.
* Basic functionality

## Cake Tasks

This section details the functionality of each of the Cake tasks.

### Build-Information

Details the information about the build.

# 

1. For details on the acceptance test groups, see section Acceptance Tests. [↑](#footnote-ref-1)