LifeYield, LLC

2 Financial Center, 60 South Street, 7th Floor
Boston, MA 02111
http://www.lifeyield.com

Testing Suite

Version 1.0
July 2015
Release Notes

Kevin Lyons
Software Engineer Intern
Mobile: (781) 812-5423
E-Mail: kevin.lyons@lifeyield.com
Web: http://www.lifeyield.com

Table of Contents

Application Introduction

LifeYield's financial planning software for both the retail and professional investor markets requires a common service to link all incoming requests. These requests arise from numerous different front-end applications, including a custom iPad application and Tax and Social Security Optimization services in partnership with Quicken. No matter the origin of these requests, however, the response returned must be constant.

For instance, a user entering an annual salary of $27,000 in the iPad application should receive the same return value for a PIA calculation as he would running through the Quicken interface. At times, updates to new releases or build models of LifeYield's tools can produce differences between the expected and actual gateway responses. Avoiding these discrepancies in data ensures customer satisfaction with the product and back-end stability throughout numerous platform releases across organizations.

Previous Methods

Prior to the development of the current application, methods were in place among the Development and Quality Assurance teams to test the gateway service. Figure 1 below shows the XmlServiceClient application, which required manual input of raw XML to receive a response, then separate differentiation of the files to observe any potential changes.

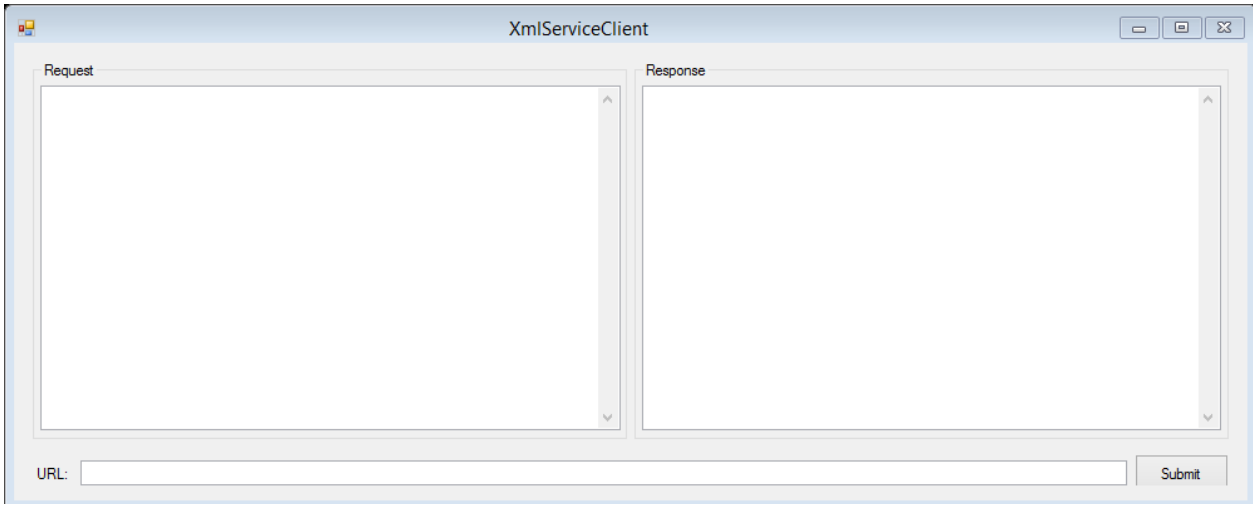


Figure 1 The XMLServiceClient achieved its desired core functionality yet was not practical in dealing with a large number of request files.

For one or perhaps several requests, this implementation was more than enough. Yet with the company's ongoing expansion to multiple organizations and current management of dozens of platform builds, a new solution must replace this application to ensure productivity and security within the gateway service.

Task Definition

The LifeYield Testing Suite application must replace this manual differentiation of request-response pairs with automated tool for selection, customization and iteration of *n* requests. The tool must also recognize the need for UI functionality and clarity, for those who will implement desire a simple, streamlined interface with debugging options. This application is assigned in a four-week period of July 2015, with several intermittent meetings throughout the process to ensure cooperation among several different departments. Once complete, the application is released to those parties who require it as a common source-controlled directory is established on the network for continuity across machines.

Functionality

In deciding the proper application type for the task (whether it be Windows Forms, WPF, HTML-5 Web Application, or simple console application), the following factors were taken into consideration by the developer:

- User-friendliness as a means to promote speed and efficient results
- Reporting services for differentiation between files after an error occurs
- Visual tool to select any number of requests
- Storage of non-confidential user information for reuse across multiple application settings

After weighing all tasks and goals at hand, the decision was made to go forward with the design and implementation of a Windows Form user interface. Microsoft Visual Studio was the IDE of choice with several third-party applications providing support when needed. ASP.NET and C# services make up the application's core.

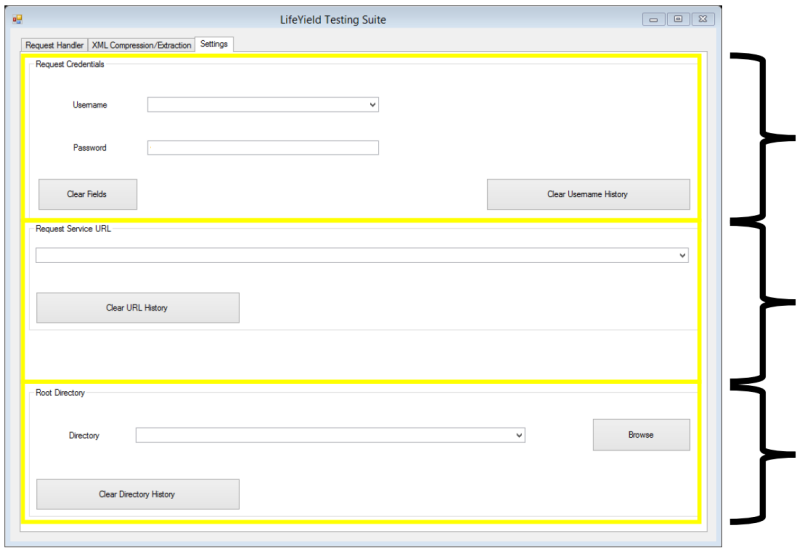
Multi-Page Overview

The application currently consists of three pages: Settings, XML Compression/Extraction and the Request Handling Service. All three of these pages are outlined in detail below, with figures for reference.

Page 1 – Settings

The Settings page is a basic layout for the user to automatically load settings from previous sessions, enter new credentials and select runtime configurations and directory preferences. It utilizes simple drop down boxes to populate stored user data and provides options for data removal if the user wishes to do so.

Startup Page - Settings



Request Credentials
User credentials for the current session, with saved usernames from previous sessions

Request Service URL
Preference to change the XML Service through which all requests will pass

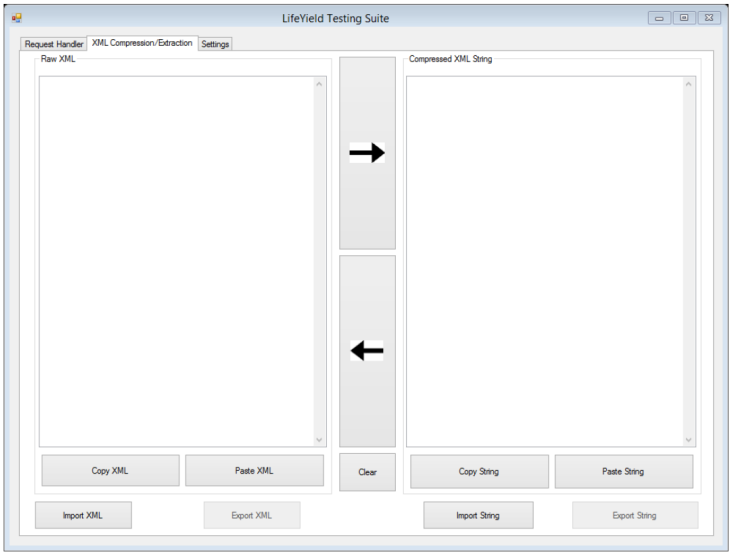
Root Directory
Set the root directory where all Application files are located in their proper location

Figure 2 The Settings page implements a simple UI to achieve core functionality. Switching between many different test users is a much easier task here, as opposed to manual XML editing.

Page 2 – XML Compression/Extraction

The application's center page is a simple development tool that converts raw XML document data into compressed G-Zip binary strings, and vice versa. If at any time during the development process an application returns a compressed value, we now possess an in-house tool to extract meaningful code for analysis. For reporting services, file export is built-in with this tool.

XML Compression/Extraction Tool



Features

- Readily convert between raw XML document and its compressed G-Zip binary string

- Import and export converted files with ease, with streamlined access to the Clipboard

- Allow Development, QA and Operations teams to quickly debug compressed data into a readable format

Figure 3 The XML conversion tool is a necessary component of LifeYield's development team testing services, as so much of their platform hinges upon XML input and output across a variety of applications.

Page 3 – Request Handling

As the core manager of the application, the Request Handling page is the most critical in terms of functionality and performance. Numerous improvements and fixes were implemented to achieve the end goals described in the task definition. Ultimately, it consists of two main stages: *setup* and *run*.

Stage 1 – Setup

This stage allows the user to select the requests he wants to run within the current configuration. The complex tree-structure is a direct representation of system directories being merged into custom structures. Familial relationships ensure its constant updating and layout.

Other personal options exist for the user, as seen in Figure 4, including layout changes and personal configuration management. For instance, if the user has a certain preference to running all of request A on organization X, he can save those settings to a file `requestA_organizatonX.xml`. Several application sessions later, when he is prepared to return to that configuration again, that file can be imported into the application to immediately populate the tree view. This approach avoids tedious checking by the developer for a growing number of custom request sets.

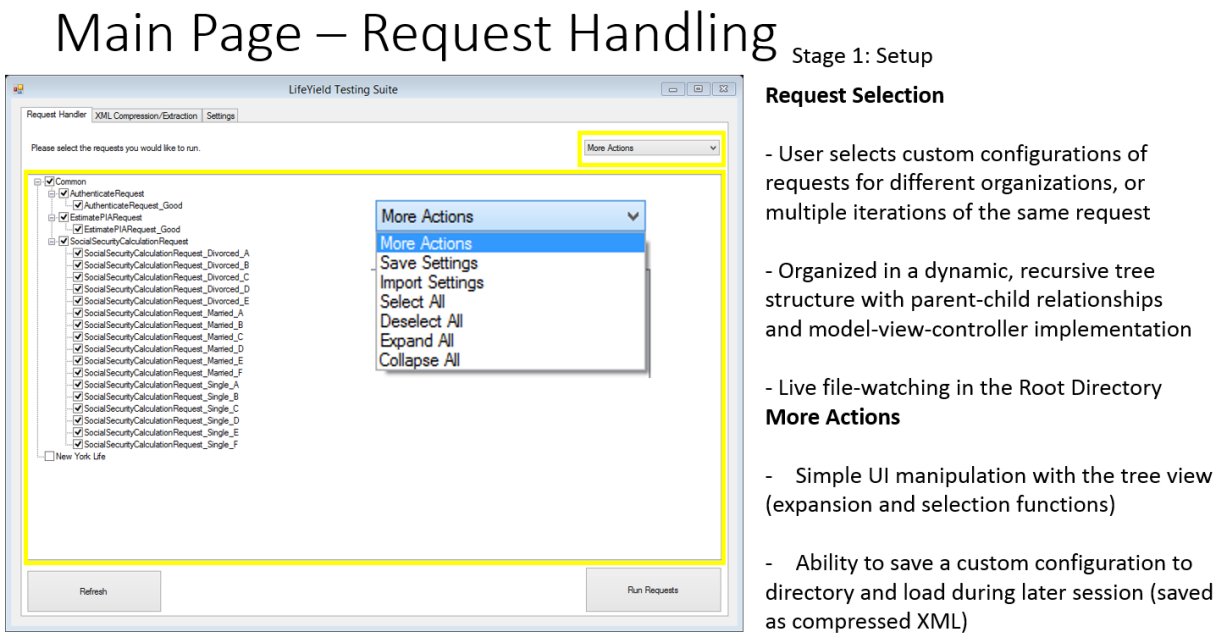
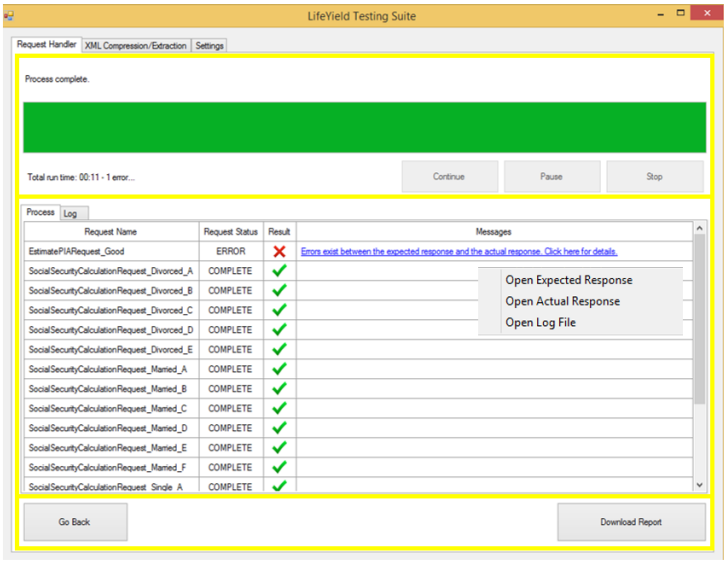


Figure 4 The *Setup* stage of the Request Handling tool seeks to combine a sleek UI with responsive data structures working in code to keep both the model and the view in a constant one-to-one relationship.

Stage 2 – Run

Running the requests is the ultimate goal of the application; a clear graphical layout of the running processes is critical. Keeping with application-wide MVC conventions, a table-based layout of each request, its current status and associated messages corresponds directly to the request structures being iterated with asynchronous management. Following task completion, the user has the option to view reporting and logging services, or return to Stage 1 for a new request set.

Main Page – Request Handling Stage 2: Runtime



Status Header
View the current status of the request operation with ability to pause/stop asynchronous iteration

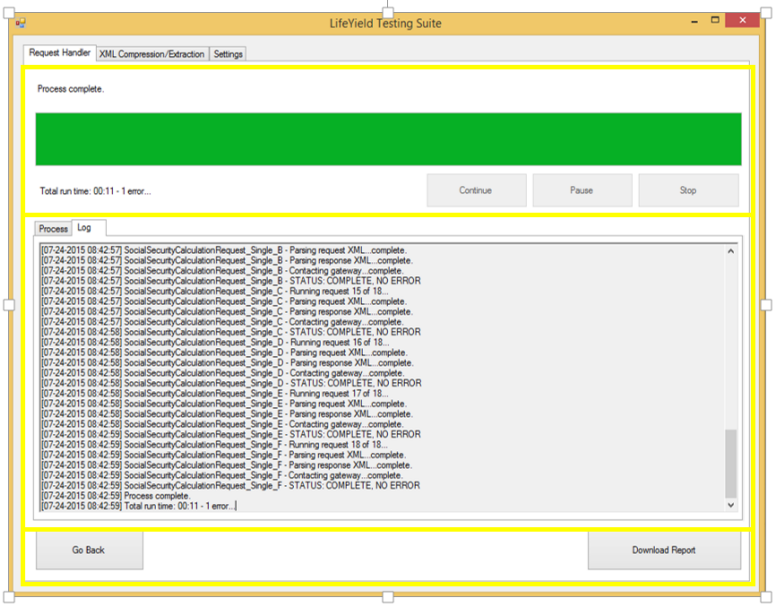
Running Pane
- *View the status of all requests*

- *If errors occur, access associated XMLs and logs for debugging – will need to differentiate between expected response in directory with actual received gateway response*

Completion Tools
- *Download report as *.cvs file*
- *Return to setup stage*

Figure 5a The running portion displays current progress, messages and other tools to clearly represent all of the data operations running in the backend.

Main Page – Request Handling Stage 2: Runtime



Logging Capabilities
Constant logging to specified paths in root directory, as well as directly to the UI for debugging

Figure 5b Logging service allow for clear documentation of larger tests that may need to be shared with multiple individuals or departments.

The three-tabbed application achieves all of its desired goals in a functional, streamlined UI that can be easily extended if need be.

Implementation

The design and development process leading up to the end result shown above was an intuitive and dynamic process, with many bug fixes and direction shifts along the way. A clear visual and programmatic layout from the outset did provide a strong footing upon which all other work would build, however. A key to the application's success was pre-planning to a large degree and dealing with error scenarios that had not been anticipated.

Design Process

Visually, the Testing Suite provides core functionality with minimal flavor and elaborate design scheming. The designing of layouts and interfaces became far more programmatic than creative as the process continued, as UI views must operate in a logical, hierarchical format for back-end code to its job efficiently. Several sketches were rendered before any code was written, though. Supervisor critique and approval was required before implementation could commence.

Structure Development

The bare UI controls and methods offered by Microsoft's Windows Form interface and ASP .NET SDK would not suffice for the custom implementations necessary for application functionality. Thus, I developed numerous custom structures and classes to deal with all such scenarios. Some demonstrate polymorphic properties of inheritance, while all follow the MVC model to be discussed. See Figure 6 below for a reference to these structures.

Figure 6 Custom Structures and their Functions within the Testing Suite

`Compressed Structure`	`class`	This class represents the structure of an already XML-serialized `SettingsStructure` object. It holds one instance variable, content, which in turn holds the one compressed G-Zip string responsible for deploying the entire application's user settings. It is saved at `ROOT\Settings\settings.xml`.
`EDifferenceToMake`	`enum`	This enumeration file processes the change needed in order to synchronize the UI and/or back-end data following a user selection of a node in the Request Handling tree view.
`ENodeStructureCheckedEntryPoint`	`enum`	This enumeration file processes the entry point of a check/uncheck event in the tree view, which could be one of the following: `USER`, `FORCEDBYCHILD` or `FORCEDBYPARENT`.
`ENodeStructureType`	`enum`	This enumeration file processes the type of a given `NodeStructure`, which could be

		one of the following: PARENT, CHILD or BOTH.
NodeStructure	class	Represents the most basic unit, the node, in the tree view. A universal application, the node must implement parent-child relationships and object associations. Each node has the potential to be a parent, child or both in addition to possessing numerous other properties.
NodeStructureManger	class	Handles many basic operations associated with a node, such as adding children, selecting and processing.
NodeStructureList	class	An inherited class from List<T>. Represents a list of nodes with family relationships and all other respective properties.
NodeStructureListManager	class	Handles many basic list operations including addition, removal and sorting. Works closely with many iterations of NodeStructureManager.
RequestHandlingManager	class	Handles file input from associated directories and merging of an existing NodeStructureList to load the tree view accordingly.
RequestStructure *also contains…* *RequestStructureManger* *RequestStructureList* *RequestStructureListManger*	class	The basic unit of a RequestStructureList, this structure represents a request to be submitted into the gateway with asynchronous list iteration. It contains the associated files for the request and status messages for necessary reporting services on both the UI and local file system.
SettingsStructure *also contains…* *SettingsStructureManager* *DefaultSettingsStructure : SettingsStructure* *RequestHandlerSettingsStructure : SettingsStructure*	class	Represents the settings for the entire application and innately houses all the above structures within its single <content> attribute in the CompressedStructure serialized XML object.

Model-View-Controller Approach

Whenever working with a complex UI application of this nature, the programming scheme of Model-View-Controller separation is critical in maintaining functionality and clarity in written code. The name of this approach describes exactly its purpose: to establish a three-layered interface where developer

and user interact through specific managers that operate at runtime. The models, in the case of the Testing Suite, are represented by all classes that contain a custom prefix followed by `Structure`, and their associated `Lists`. The managers are defined clearly in Figure 6 above, and are responsible for handling all operations done onto their respective models. Finally, the views of this application are visible objects the user can interact with, including but not limited to text fields, buttons, labels, drop down boxes, links and menus.

The unison of these three entities with streamlined logical code and inheritance/`using` statements where necessary represents the foundation of this and numerous other visual applications. See Figure 7 for a diagram relating the three in terms of the `NodeStructure` model.

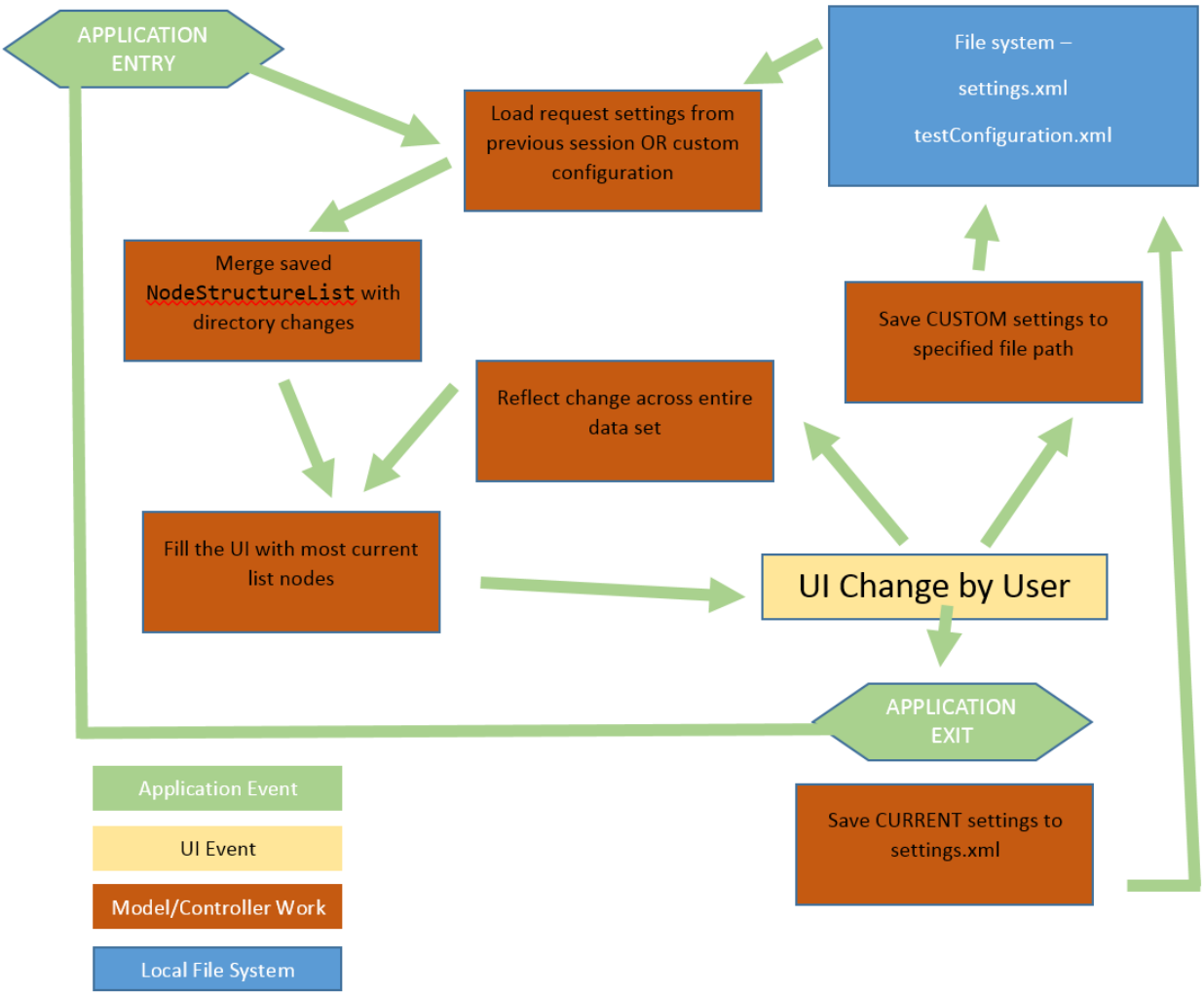


Figure 7 A look at the Model-View-Controller approach in action as a NodeStructureList is used to populate a tree view with automatic synchronization across the entire UI.

Storage Management

As previously mentioned, conserving storage from file production in such an application is critical for its long-term success and productivity. Throughout the development process, several measures were taken to drastically reduce memory quotas, beyond clearing out memory heaps after application

termination. The most effective in which we achieved these goals was XML compression and extraction across the interface. A native part of the LifeYield service, this quality enables seamless key-value data transmission into custom objects. This allows the application to, quite literally, load all of its settings from a single G-Zip binary string that holds the ability to populate all text fields and the entire Request Handling tree view.

Individual requests could not be compressed in the file system as they had to be viewed for differentiation purposes. Compression of these files is a feature to be implemented in Version 1.1.

Error Handling

In dealing with complex system process such as cross-thread UI invocation during asynchronous thread work and directory parsing, error handling and debugging became the ultimate determinant of success for the application. With the use of log4net's robust logging tools, we are able to view many of these errors immediately for quick fixes. Throwing `Exception`s when there is even the slightest chance of an error prevents the chance of an application shut down. As the lead developer of the Testing Suite, I will in fact not be in the office during its heaviest use. Thus, it is imperative that beyond clear documentation, catches are in place for potential setbacks.

Conventions

To run an application with such dynamic data coming from an local directory, certain conventions must be in place in order to avoid system errors and ensure proper file input and output. There are four main areas where specific guidelines must be in place: directory management, settings storage, file naming and XML creation.

Directory Conventions

Please see Figure 8 below for the structure of directories that must be established **before** running the application for the first use.

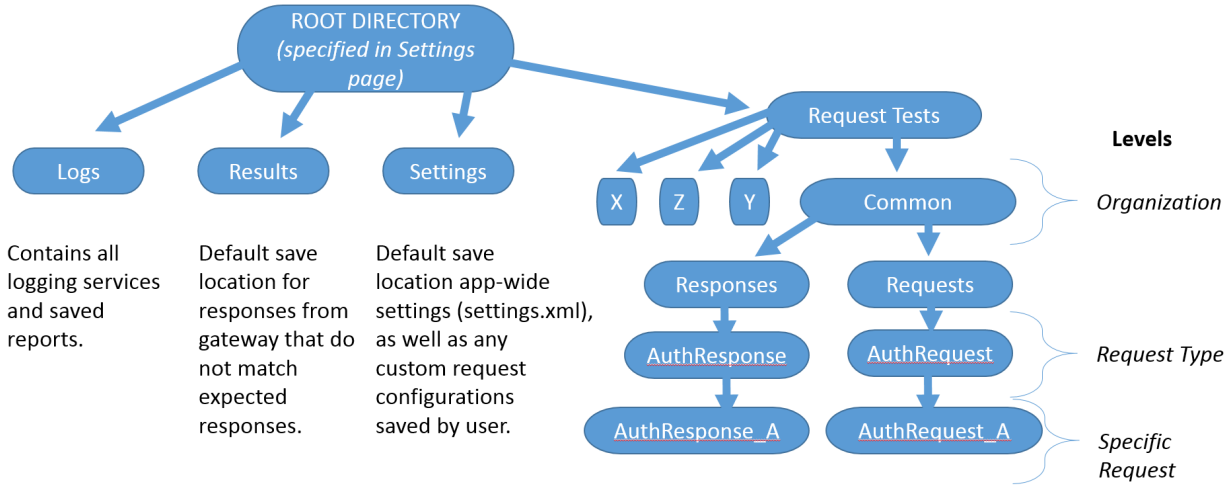


Figure 8 The above directory structure must be followed closely for finding requests and utilizing native reporting services.

Settings Conventions

The application settings must live in the following path: ROOT\Settings\settings.xml, where ROOT is the root directory of your application files.

This settings file is of type CompressedStructure and appears in the following format:

```
<CompressedStructure>
      <content>SOMECOMPRESSEDGZIPSTRING</content>
</CompressedStructure>
```

See the reference to First Use Information for getting your settings.xml established before your first startup.

Naming Conventions

As seen in Figure 8, specific naming conventions must be used when dealing with requests and responses in the directory. **There must be no difference between any given request and response folder and file other than that the word "Request" is replaced by "Response."** Here is an example folder and file structure that correctly pairs a request and its expected gateway response:

Request File Path – ROOT\Request Tests\Common\Requests\AuthenticateRequest\AuthenticateRequest_Good.xml
Response File Path – ROOT\Request Tests\Common\Responses\AuthenticateResponse\AuthenticateResponse_Good.xml

Failure to follow these conventions may lead to errors at compile or run-time.

XML Conventions

The dependence upon XML compression and extraction throughout the program requires a strict set of parameters to be followed when developing new XML in the ROOT\Request Tests directory. Thus, the following rules must be followed to ensure successful program iteration:

- All requests must contain keyword placeholders for the username, @USERNAME, password, @PASSWORD and organization, @ORGANIZATION, within the XML. The specified username and password from the Settings pane will be substituted in for these values at runtime.
- Requests and responses must not contain any AuthTokens. This is due to their time-sensitive nature and relative expirations that shall throw unexpected ProcessingErrors despite otherwise identical XMLs. The application does account for the presence of AuthTokens for safety purposes, yet it is best to leave this value out of any raw XML files.

Code Definitions

The Testing Suite application is comprised of numerous classes, enumerations and managers that live within `TestingSuiteApplication.sln`. Below a complete reference of all classes and associated methods for future reference in the event of debugging or upgrading the application.

Class Hierarchy

See Figure 9 for a class hierarchy of the application.

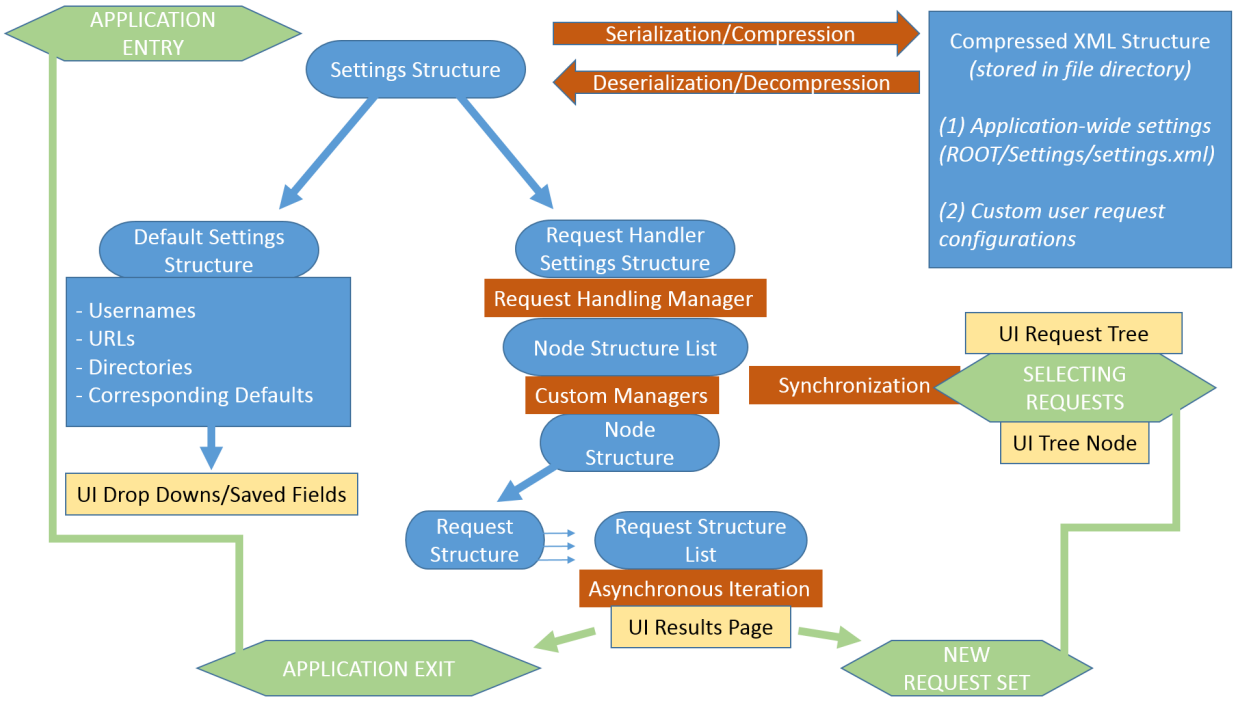


Figure 9 The application flow with respect to the numerous models, controllers and corresponding UI views that connect to ultimately test the gateway service's core functionality.

Method Definitions

Organized by class, the following methods are critical components of the Testing Suite and can be easily modified for future use. By developing with a deliberate flexible design in mind, structures and functions suited for one specific use can now easily be applied in other formats to achieve a similar end result.

```
public enum EDifferenceToMake
// Represents the difference needed to synchronize a TreeNode and a
//  NodeStructure with varying states for being checked – either to check
//  both or uncheck both.  Depends upon input path – user or forced by
//  system.

public enum ENodeStructureCheckedEntryPoint
// Represents the manner in which a call to check a structure is initiated –
//  a user UI click results in a different behavior than a method iterating
//  through all children of a TreeNode

public enum ENodeStructureType
// Represents the state of a NodeStructure in terms of its family relations –
//  a parent, child, both or neither

public partial class Form1
// Represents the main class for the Windows Form.  Contains all the standard
//  UI action methods, and some additional helper methods, which are listed
//  below.  Please also note the #region UpdateUIRegion.  This region
//  contains all of the necessary delegate callbacks in order to update the
//  UI thread while a separate thread (likely that of the BackgroundWorker)
//  is busy.

public void initializeBackgroundWorker(BackgroundWorker backgroundWorker)
// Params:
//   backgroundWorker – the BackgroundWorker to initialize
// Precondition:
//   backgroundWorker is not null
// Postcondition:
//   backgroundWorker has been properly initialized with necessary properties
//    for cancellation, as well as associated event handlers for thread
//    operation.

public static void findDifferences(string a, string b, RequestStructure
      requestStructure)
// Params:
//   a – first string to differentiate
//   b – second string to differentiate
//   requestStructure – the RequestStructure to update with any necessary
//    error messages
// Precondition:
//   no params are null
```

```
// Postcondition:
//   The errors of requestStructure show the differences between strings a
//    and b, which likely represent the expected XML response and actual
//    gateway response.


public void createPanel()
// Postcondition:
//   The requestHandlerTableLayoutPanel has been repopulated to fit the needs
//    of the new RequestStructureList that shall fill it.  This includes
//    resizing and filling lists of controls to be added to the table.

public static void selectIndex(ComboBox comboBox, int homeIndex)
// Params:
//   comboBox – the ComboBox whose SelectedIndex property will be changed
//   homeIndex – the index to set for comboBox
// Precondition:
//   no params are null; homeIndex does not throw and
//    IndexOutOfBoundsException for comboBox
// Postcondition:
//   comboBox.SelectedIndex = homeIndex and the UI view updates accordingly.

public static void populateComboBox(ComboBox comboBox, List<string> values)
// Params:
//   comboBox – the ComboBox to populate with string elements
//   values – the list of strings with which comboBox will be filled
// Precondition:
//   no params are null
// Postcondition:
//   comboBox's drop down menu now contains the strings in values and can be
//    selected accordingly within the UI.

public static string getRequestString(int number)
// Params:
//   number – the number of requests that will be running on the
//    BackgroundWorker thread
// Precondition:
//   no params are null; number > 0
// Return:
//   a string that displays the word "request," but either in its single or
//    plural form based on the value of number

public TreeNode convertPathToTreeNode(string fullPath)
// Params:
//   fullPath – the string containing the full path of the TreeNode in its
//    respective TreeView, with specific dot notation to demonstrate its
//    relative indexes
```

```
// Precondition:
//   fullPath is within the expected string format for proper parsing
// Return:
//   a TreeNode that exists at the path specified within fullPath.  This node
//    is often the one to scroll to and become the top node of the TreeView.

public void PopulateTableLayoutPanel(RequestStructureList
      requestStructureList)
// Params:
//   requestStructureList – the list of requests that shall be passed in to
//    populate the view with names and corresponding statuses and messages.
// Precondition:
//   the TableLayoutPanel to populate has been properly initialized (likely
//    through CreatePanel()); no params are null;
//    requestStructureList.Count > 0
// Postcondition:
//   The TableLayoutPanel view is populated column by column with request
//    names, statuses, results and any messages needed.  UI suspension and
//    resuming still allows a bit of a UI flicker during this process, yet it
//    is the only such action within the entire request handling process.

private void GoBack()
// Postcondition:
//   Returns the user back to the SETUP stage of the request handling process
//    and ensures that all application resources are properly dealt with in
//    the event of further request running.

public class NodeStructure
// Represents the most basic unit of a NodeStructureList and data, with the
//  ability to represent a complex object containing numerous properties and
//  familial relations to other local NodeStructures.

public class NodeStructureManager
// Handles all operations for managing and editing individual NodeStructures.

public static void addChild(NodeStructure parentNodeStructure, NodeStructure
      childNodeStructure)
// Postcondition:
//   parentNodeStructure and childNodeStructure both contain the proper
//    assocations necessary in a parent-child relationship that shall be
//    accessed in all other areas of the application.

public static void check(NodeStructureList nodeStructureList, NodeStructure
      nodeStructure, ENodeStructureCheckedEntryPoint
      nodeStructureCheckedEntryPoint)
// Params:
//   nodeStructureList – the highest level list containing all NodeStructures
//    for a given application – root nodes are high-level parents that have
//    progressively more levels of children below
//   nodeStructure – the NodeStructure to check
```

```
//   nodeStructureCheckedEntryPoint – the action forcing nodeStructure to be
//    checked – must differentiate between a parent instructing check and a
//    child doing so as well.
//Precondition:
//   no params are null
//Postcondition:
//   properly checks the current NodeStructure as well as all associated
//    NodeStructures in the overarching NodeStructureList

public static void uncheck(NodeStructureList nodeStructureList, NodeStructure
      nodeStructure, ENodeStructureCheckedEntryPoint
      nodeStructureCheckedEntryPoint)
// Same functionality as check, just opposite bool values.

public static bool allChildrenChecked(NodeStructure parentNodeStructure)
// Params:
//   parentNodeStructure – the NodeStructure to be analyzed for children's
//    checked status
// Precondition:
//   parentNodeStructure is a parent and therefore has some number n of
//    children
// Return:
//   true if all NodeStructures in parentNodeStructure.children are checked,
//    false otherwise

public static TreeNode toTreeNode(NodeStructure nodeStructure)
// Params:
//   nodeStructure – the NodeStructure to convert to a TreeNode
// Precondition:
//   no params are null
// Return:
//   a TreeNode that contains the corresponding proerties of nodeStructure.
//    this is the main connector between model and view on the individual
//    node level.

public static void getName(NodeStructure nodeStructure, ENodeStructureType
nodeStructureType)
// Params:
//   nodeStructure – the NodeStructure whose name must be determined
// Precondition:
//   no params are null
// Postcondition:
//   the name property of nodeStructure is set to its acutal name, which is
//    extracted from its file path.  The ENodeStructureType is used with
//    string formatting to determine the proper substring to utilize and
//    whether file extensions are in play.

public static NodeStructure getParentNodeStructure(NodeStructure
      nodeStructure, NodeStructureList largeList)
// Params:
```

```
//   nodeStructure – the child NodeStructure for whom the method must return
//    the parent
//   largeList - the highest level list containing all NodeStructures
//    for a given application – root nodes are high-level parents that have
//    progressively more levels of children below
// Precondition:
//   nodeStructure is a child with a given parent in largeList; no params are
//    null
// Return:
//   the NodeStructure who is the parent of nodeStructure – often used for
//    list iteration purposes and changing family relations from a UI event.

public static ENodeStructureType getStructureType(NodeStructure
      nodeStructure)
// Params:
//   nodeStructure – the NodeStructure whose type must be determined
// Precondition:
//   no params are null
// Return:
//   an ENodeStructureType that describes the family relations contained by
//    nodeStructure, either as a parent, child, both or neither.

public static void assignParentNames(NodeStructure parentNodeStructure)
// Params:
//   parentNodeStructure – the NodeStructure whose property parentName will
//    be assigned
// Precondition:
//   no params are null
// Postcondition:
//   the parentName property of parentNodeStructure is set to a value
//    determiened before the method's invocation.  This parentName property
//    is used to find the parent in the above mentioned
//    getParentNodeStructure() method.

public class NodeStructureList : List<NodeStructure>
// Represents the list of NodeStructures that is used to populate the first
//  pane's TreeView and is constantly updated to respond to UI changes.
//  Shows parent/child relationships with nested lists.

public class NodeStructureListManager
// Handles all operations for managing and editing entire NodeStructureLists.

public static void addNodeStructure(NodeStructure nodeStructure,
      NodeStructureList nodeStructureList)
// Postcondition:
//   Adds nodeStructure to nodeStructureList and resorts the list accordingly
//    to repsond to this addition.

public static void removeNodeStructure(NodeStructure nodeStructure,
      NodeStructureList nodeStructureList)
```

```
// Same functionality as add, just removing elements instead.

public static NodeStructureList alphabetize(NodeStructureList
      nodeStructureList)
// Return:
//   the contents of nodeStructureList sorted in alphabetical order by their
//    name property.

public static NodeStructureList reindex(NodeStructureList nodeStructureList)
// Precondition:
//   nodeStructureList has already been alphabetized immediately before this
//    method's invocation
// Return
//   the contents of nodeStructureList with each NodeStructure's index
//    property updated to match its current position in the list.

public static void uncheckAllNodeStructures(NodeStructureList
      nodeStructureList)
// Postcondition:
//   all NodeStructures in nodeStructureList, including sublists, are
//    unchecked

public static bool containsNodeStructure(NodeStructureList nodeStructureList,
      NodeStructure nodeStructure)
// Return:
//   true if nodeStructureList already contains a NodeStructure with the same
//    name as the nodeStructure passed in as a parameter, false otherwise.
//    Used when determining whether or not to add a NodeStructure to a
//    NodeStructureList.

public static NodeStructureList getCheckedNodes(NodeStructureList
      nodeStructureList)
// Return:
//   a NodeStructureList that contains only the checked bottom-level nodes in
//    nodeStructureList.  This list is used to populate a
//    RequestStructureList for actual request submission to the gateway.

public static NodeStructureList removeDuplicateNodes(NodeStructureList
      nodeStructureList)
// Return:
//   a NodeStructureList that contains the contents of nodeStructureList,
//    with duplicate elements (based on name) removed.

public static NodeStructure getNodeStructureWithName(NodeStructureList
      nodeStructureList, string name)
// Precondition:
//   nodeStructreList contains a NodeStructure named with the input string
//    name
```

```
// Return
//   the NodeStructure with the given name.  Used when a list already
//    contains a particular NodeStructure during request loading from the
//    application directory at startup or refresh.

public static NodeStructureList mergeNodeStructureLists(NodeStructureList
      highList, NodeStructureList lowList)
// Params:
//   highList – the existing, priority list that shall be updated with any
//    applicable changes observed in lowList
//   lowList – a new NodeStructureList that may reflect some of its
//    properties into highList as a means of updating existing list content
// Precondition:
//   no params are null
// Return:
//   highList such that its contents contain any necessary updates seen
//    within lowList (i.e. differences in expansion or checking states).
//    Used when the user imports old list configurations that need to be
//    reconciled with the current working directory.
```

public class RequestHandlingManager

```
// Handles all operations having to do with getting new requests from the
//  directory and updating the UI's views accordinly on the first pane of the
//  application.
```

Please note that the params for some methods will not be repeated numerous times. Please see below for params that appear in numerous method definitons as well as their roles.

```
nodeStructureList – the highest level NodeStructureList containing all
      NodeStructures necessary for updating and UI propogation
treeView – the TreeView in the UI that is updated with new data
rootDirectory – the ROOT path of the application, as specified in the
      settings pane
isFresh – bool specifying whether or not the NodeStructureList should start
      anew and remove all previous associations.  Mainly used for debugging
      purposes.
occurance – either 0, specifying the first population of data at application
      startup, or 1, specifying an update during the application's normal
      runtime
treeNode – used to indicate the top node of treeView, for scrolling purposes
      and UI functionality (the view always scrolls to the previous scroll
      point after updating its data)

public static NodeStructureList getNewRequests(<params...>)
// Return:
//   updated NodeStructureList that contains all old data including new files
//    from the directory and excluding those that were changed.
```

```
public static void updateRequests(<params...>)
// Postcondition:
//   Synchronizes the UI view and backend NodeStructureList model in order to
//    reflect all familial changes of checking/unchecking a particular
//    TreeNode.

public static void fillTreeView(<params...>)
// Postcondition:
//   The UI TreeView is filled with all nodes and subnodes, with associated
//    expanded and checked states, as represented in a NodeStructureList.

public static void expandProperNodes(<params...>)
// Postcondition:
//   The nodes in the UI TreeView are expanded back to a state saved within a
//    NodeStructureList.  This ensures that the entire view does not reshift
//    for every UI check/uncheck by the user.

public static EDifferenceToMake differencesExist(NodeStructure nodeStructure,
      TreeNode treeNode)
// Precondition:
//   nodeStructure is the object represented within the Tag property of
//    treeNode
// Return
//   an EDifferenceToMake indicating the necessary action to synchronize the
//    two structures, whether it be to check both or uncheck both.

public static void synchronize(<params...>)
// Postcondition:
//   The TreeNode and NodeStructure resync themselves to match their checked
//    states, either forcing both to be false or true based on the entry
//    point of the invocation (USER, FORCEDBYCHILD or FORCEDBYPARENT).

public static void saveExpandedState(NodeStructureList nodeStructureList,
TreeView treeView)
// Postcondition:
//   The expanded property of all NodeStructures in nodeStructureList,
//    including all children, are updated to match their respective partners
//    on the UI.  These properties are then used to refresh the view
//    accordingly, as seen in the expandProperNodes(<params...>) method
above.

public class RequestStructure
// Represents a request that will be submitted to the gateway service through
//  asynchronous iteration within a BackgroundWorker through a
//  RequestStructureList.

public class RequestStructureManager
// Handles all operations for managing and editing individual
//  RequestStructures.
```

```
public static string getResponseFilePath(RequestStructure requestStructure)
// Return:
//   The responseFilePath for requestStructure using substring and
//    replacement techniques within directory file paths.  It is these such
//    methods that require such a precise naming scheme in order for proper
//    program function to occur.

public class RequestStructureList
// Represents a list of RequestStructures that enters a BackgroundWorker for
//  processing.

public class RequestStructureListManager
// Handles all operations for managing and editing entire
//  RequestStructureLists.

public static RequestStructureList
      populateRequestStructureList(RequestStructureList requestStructureList,
      NodeStructureList nodeStructureList)
// Precondition:
//   nodeStructureList represents the list of checked lowest-level nodes in
//    the current application data set.
// Postcondition:
//   Returns a RequestStructureList with all lowest-level node structures
//    converted into RequestStructures with corresponding property values
//    (i.e. filePath).

*** Addition, removal, sorting and indexing processes are identical to those
seen in NodeStructureManager.

public enum ERequestStructureState
// Represents the current state of a RequestStructure, either WAITING,
//  RUNNING, COMPLETED OR ERROR.

public class SettingsStructure
// Represents a set of settings used to load the application.  Contains 2
//  children (DefaultSettingsStructure and RequestHandlerSettingsStructure)
//  that fit within its broader contents.

public static SettingsStructure downloadSettings(string
      settingsStructureFilePath, ESettingsStructureType
      settingsStructureType)
// Params:
//   settingsStructureFilePath – the file path of the XML settings, whether
//    it be application wide settings at ROOT\Settings\settings.xml or some
//    specific request handler list configuration at
//    ROOT\Settings\testConfiguration.xml
//   settingsStructureType – the type at hand, whether it be ROOT, DEFAULT OR
//    REQUESTHANDLER
// Precondition:
```

```
//   There exists a valid XML file at settingsStructureFilePath that contains
//    the XML to be deserialized into an instance of SettingsStructure or one
//    of its children.
// Return:
//   a SettingsStructure instance that contains all the necessary data values
//    to deploy the application and fill drop down boxes, tree views and
//    other components.

public static void saveSettings(SettingsStructure settingsStructure)
// Postcondition:
//   Serializes the given SettingsStructure instance to XML and writes that
//    XML to its internal file path.  Also differentiates between the ROOT
//    SettingsStructure and RequestHandlerSettingsStructure types before
//    serialization occurs.
```

References

Consult the following information for application support and debugging tools, as well as future contact information.

First Use Information

Before the Testing Suite is deployed for the first time, several features must be in place on the local system to avoid errors. Please follow these steps to ensure proper startup.

1. Establish a location where the application's supporting files will live. This will be considered the ROOT directory.
2. Create the proper folders for Logs, Results, Settings and Request Tests. See Figure 8 for details. **If you are to add any organizations to the Request Tests directory, be sure to add its respective Requests and Responses folders within.**
3. Within the Logs folder, create a new file `requestHandlerLog.log`.
4. Attached to these Release Notes you will find the file `settings.xml`, which represents the default settings for the application. Place this file within the Settings folder.
5. In the application's bin, open the `.config` file. Make the following changes:
   - In the <userSettings> tag, change the value of "settingsFilePath" to match the path of the `settings.xml` file you just copied to your local system.
   - In the <userSettings> tag, change the value of "logFilePath" to match the path of the `requestHandlerLog.log` file you just created in your local system.
   - In the <log4net> tag, change the path value given in the <param "File" …> tag to match the path of the `requestHandlerLog.log` file you just created in your local system.
6. Deploy `TestingSuiteApplication.exe`.

Contact Information

Following July 31st, 2015, my contact information is as follows:

Kevin Lyons
Weymouth High School, Class of 2016
Mobile: (781) 812-5423
E-Mail: contact@kevinalyons.com | 16kalyons@weymouthstudents.org (if before June 2016)
Web: http://www.kevinalyons.com

Professional References

I received guidance and advocacy from the following individuals during my tenure at LifeYield in the summer of 2015. Please see their contact information below for reference.

Michael Benedek
President and CFO
LifeYield, LLC.
(617) 502-5711
michael.benedek@lifeyield.com

Mark Hoffman
CEO
LifeYield, LLC.
(617) 502-5710
mark.hoffman@lifeyield.com

Joseph Dynes
Software Engineer
LifeYield, LLC.
(617) 502-5668
joseph.dynes@lifeyield.com