



SOA Testing Framework Developer Guide

|  |  |
| --- | --- |
| Author: | Ladislav Jech |
| Owner: | Irish Water Programme |
| Document Date: |  |
| Document Reference: |  |
| Version: | 0.1 |

Document History

Document Location

This is a snapshot of an on-line document. Paper copies are valid only on the day they are printed. Refer to the author if you are in any doubt about the currency of this document.

Most up-to-date version is available at SVN repository:

<http://10.19.12.41:8080/svn/repos_int1/m2/trunk/soa_testing_framework/doc/user-guide>

Revision History

|  |  |
| --- | --- |
| Date of this revision: 04.11.2013 | Date of next revision *31.12.2013* |

|  |  |  |  |
| --- | --- | --- | --- |
| Revision Number | Revision Date | Summary of Changes | Changes marked |
| 0.1 | 04.11.2013 | Initial version |  |
|  |  |  |  |
|  |  |  |  |

Approvals

This document requires following approvals:

|  |  |
| --- | --- |
| Name | Title |
| Kieran Murphy | Integration Architect |

Distribution

This document has been distributed to

|  | Title | Name | Title |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

1. Introduction 4

2. High Level Overview 5

2.1 High Level Framework Design 5

2.2 Framework Entry point 5

2.3 Configuration Load & Configuration Unmarshalling Process 6

2.4 Loading of GUI 6

2.5 Flow Executor 7

3. Detail process flow 8

3.1 Components and Operations 9

3.1.1 Extending SOATF - Adding operations to existing components 9

3.1.2 Extending SOATF - Adding new components 9

3.2 Testing process 11

3.3 IBM Irish Water Programme Integration flow patterns 12

4. General JAVA code guidelines 14

4.1 Object OperationResult 14

4.2 Logging 16

4.3 Exception handling 17

5. Build process 18

5.1 Building in NetBeans IDE 7.4 18

5.2 Building with Ant 1.9.1 20

5.3 Build process customization 21

6. Tips & Tricks 22

6.1 Entity Reference and Message Reference 22

6.2 What do I do when I want to add something new to the XSD schema(s)? 24

# Introduction

SOA Testing Framework (SOATF) project has been created within the Irish Water Programme as an evolutional mechanism requirement which should help to transform the software development life cycle to the way of continuous integration and delivery, both becoming standards in those days. Framework itself provides mechanisms to test defined interfaces and create testing reports.

The prime purpose of SOATF is to save time of developers/testers on interface testing phases.

Framework is fully implemented using Java Standard Edition 7 and supports XML based configuration with 2 runtime access modes:

* Graphical User Interface

Used from developer/tester private laptop to provide user friendly control interface

* Command Line

Used by Hudson server for automatic execution of tests

This document's purpose is to help the developer with understanding the structure of the code.

# High Level Overview

This chapter describes high level process flow and design of the framework.

## High Level Framework Design

Inside the framework there exists support for endpoints of various technology types. Framework provides a set of the most common operations used in testing scenarios for each technology. Every operation can be seen as atomic from integration testing point of view. Following picture shows the basic architecture:

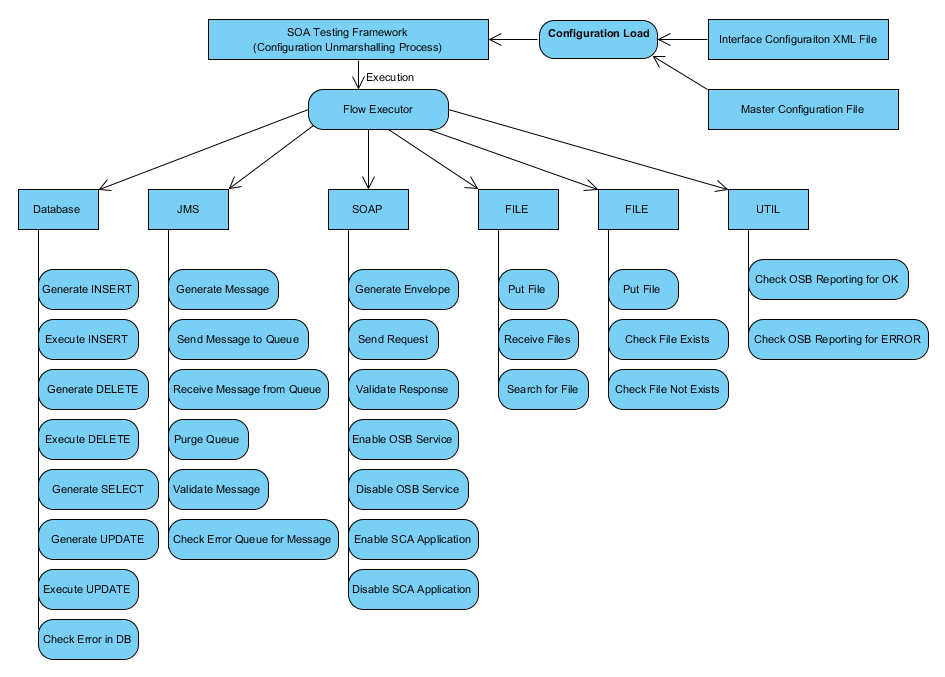


Figure - High Level Framework Design

## Framework Entry point

Entry point of the SOA Testing Framework is SOATestingFramework class located in com.ibm.soatf package. First step while running the framework is to decide whether to run in GUI mode or CLI mode. This decision is based on the given command line arguments. For GUI mode, only the -gui argument is needed. Arguments -env and -i or -p with corresponding names have to be passed, in order to run the framework in CLI mode. Otherwise the execution will finish with printing the usage information. While running in CLI mode, framework is executing always at interface level and the execution is stopped after the selected interface test is done. This is not true for the GUI mode, where user can select any combination of interface, environment and test level and run the tests repeatedly.

## Configuration Load & Configuration Unmarshalling Process

One of the main init processes started on execution of the framework itself. These general procedures reads XML based framework metadata storage files and transform them to real JAVA class instances or let's say objects.

Classes handling all configuration related tasks are located in the package com.ibm.soatf.config.

ConfigurationManager class is the one responsible for loading and unmarshalling the framework configuration. It follows the singleton pattern. Current instance of this class is accessible anywhere in the code simply by calling a static method ConfigurationManager.getInstance(). After calling init()method, configuration manager is ready to return any of the loaded configurations by calling:

* public getMasterConfig(),
* public getInterfaceConfig(String interfaceName), or
* public getFrameworkConfig()

Each type of configuration holder class has its own init() method, which validates, and loads the data needed from XML file. MasterConfiguration is initialized by ConfigurationManager init() method and InterfaceConfiguration is initialized usually when first accessed.

DirectoryStructureManager class is responsible for creating / cleaning all the directories holding the result data for every test run. It accesses the coresponding InterfaceConfiguration, so the DirectoryStructureManager is the place where InterfaceConfiguration gets validated and loaded.

ConfigurationManager is initialized in the very first step when running SOATF in CLI mode, followed by initialization of DirectoryStructureManager as the second step.

Both steps are invoked as AsyncTask right after initialization of GUI components in GUI mode. Nested class AdditionalInitAsyncTask serves this purpose in the class SOATestingFrameworkGUI.

In GUI mode, initialization of DirectoryStructureManager loops through every interface mentioned in MasterConfiguration, while in CLI mode, only provided interface gets initialized.

## Loading of GUI

When the framework is executed with -gui option, application's GUI is loaded by creating the instance of SOATestingFrameworkGUI class, which is subclass of JFrame. This JFrame holds all graphic components of the application. Every action performed by the user in GUI is executed asynchronously in a separate thread. For example FlowExecutor class is instantiated on the btnExecuteActionPerformed event. To show the results of execution on the fly, the frame registers its own listeners to the FlowExecutor class and then executes it in separate thread.

## Flow Executor

This core process takes care about the test scenario execution. This is good place where to explain high-level design of the test scenario architecture. SOA Testing Framework come with common object hierarchy design.

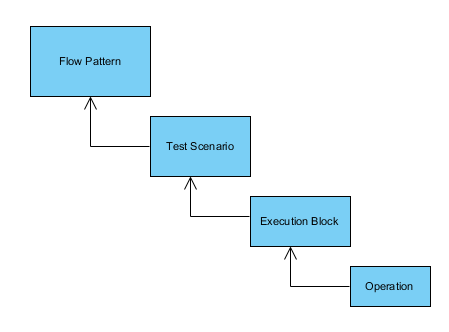


Figure - Common Flow Pattern Object Hierarchy

Sequence of operations is nested in execution block. One execution block can be bound with only one source and one target endpoint of some technology type. This is how operation can identify endpoint configuration uniquely. Sequence of execution blocks are nested in test scenario. There are also optional pre/post execution blocks in a scenario. Sequence of scenarios are grouped in flow patterns and flow patterns instances are bound to interfaces in MasterConfiguration. Multiple different flow patterns can be bound to one interface. In config.xml for this interface then should all these flow pattern instances be implemented (i.e. all blocks paired with the endpoints).

After choosing an interface (passing the CLI argument, or selecting it in GUI) SOATF can identify all the operations and their order of execution to test the interface.

Detailed description of every layer of configuration can be found in user guide.

# Detail process flow

In the FlowExecutor class, there is only one public method for executing the flow:

public void execute()

This method is executed whether by pressing execute button in GUI or automatically follows initialization steps when running in CLI mode.

There are multiple overloaded methods which are chained together and the actual execution flow is based on the selected level. For example if we are executing from the level of operation, the execution chain goes directly down to the last method in the hierarchy - executeOperation which is responsible for actual component creation and operation execution on the targeted endpoint. But when we are executing from the level of execution block, second last method in the chain will iterate through all the operations in the selected execution block and call executeOperation multiple times. The execution is stopped when the operation throws an exception, or user pressed stop button in guy (then the flow is stopped at the end of the actual iteration).

There are few more rules taken into account in the described process:

* scenario pre/post blocks are executed only when execution was started at scenario (or higher) level - before first or after last regular execution block in particular scenario
* flow pattern pre/post blocks are executed only when execution was started at scenario (or higher) level - before first and after last ran scenario (or before/after one selected scenario)
* post operations are run even if operation threw exception
* operations with the flag 'continueOnFailure' (which can be set in flow-pattern configuration) doesn't cause flow to break
* all pre/post operations are treated as continueOnFailure
* at the beginning of the test, actual system date is stored in FlowExecutor instance - this can affect many things during the test, for example name of the generated files. This date remains unchanged as long as the test scenario is not switched during the test. This allows users to run operations one by one and still be able to perform one consistent test.
* report - as post flow pattern operation is generated only when executing at least one scenario as whole. If run from flow pattern level, one report is generated for the whole flow pattern containing results from every scenario.

FlowExecutor provides 2 more public methods - for registering and unregistering listeners for flow events. There are 3 events to which can listener respond - start of the operation execution, end of the operation execution and switching between scenarios or flow patterns. For this purpose FlowExecutionListener class exists with corresponding 3 abstract methods.

When constructing the component and executing the operation in executeOperation method, FlowExecutor selects the appropriate component from the prefix of the operation name in a switch-case statement. Then it extracts the part of the configuration based on the type of component and target of the operation. Configuration is passed to the component constructor together with component working directory and other vital arguments. Operation is executed right after component initialization.

Scenario pre/post operations are treated in the same way as standard operations, except they are coupled with standard execution block before identifying their target endpoint. This is done of 2 reasons:

* pre/post operations should only work with targets used in that particular scenario
* this coupling is done once, in MasterConfiguration in definition of flow pattern. There is no need for user to identify resources for pre/post operation

Flow pattern pre/post operations are not targeting any endpoint. At the moment only the report and some of the the util operations can be used as flow pattern pre/post operations.

## Components and Operations

Every operation known to SOATF should be enumerated in MasterConfiguration schema, in the type OperationName. All operations that are coupled with one component should share the same prefix (part of the name before \_ ). This is how FlowExecutor class decides which component to load and execute the operation on.

Component classes are not discovered automatically. For every new component introduced, new case branch should be added to the FlowExecutor. This can be one of the possible future improvements of the SOATF framework.

For every operation execution, there is a new component object instantiated. This is based on our original design - that every operation should be independent and atomic set of commands to achieve the goal and collect the result. But there are already few exceptions to this rule. Although there is a new component instance created, it is not forbidden to save the state of execution in the component's static fields. For example DatabaseComponent will store rowId returned by any execution of the insert statement. For any subsequent db operations targeting the same endpoint, the affected row is identified by its rowId instead of the combination of configured values. Another example of statefull behaviour of the components is UtilComponent. The operation util\_clear\_reporting obtains an actual timestamp from the db for future use. Next time the util\_check\_reporting\_\* operations are called, this timestamp is used as limiting condition for the returned data.

### Extending SOATF - Adding operations to existing components

The entry point for any operation execution is the components executeOperation(Operation operation) method. The type Operation holds information about the operation name, and operation target, which both are of type enum, and their values are defined in master configuration schema. These are the only two properties required to properly execute an operation within the component context. While the target enum should be immutable, the operationName enum should be extended to contain names of all new operations added to the framework. All operations within one component must share the same prefix.

There are 2 more optional properties in the Operation type. Description and expectedResult are both text fields that have no real use in the framework flow and are passed only to the output reports.

It is a good practice to provide separate method for any operation in the component and implement one switch-case block in the executeOperation method.

The initialized component already holds all required data needed to perform any operation, so the implementation of the operation should be straightforward. Every operation is required to fill the OperationResult object of the component (usualy named cor), using the addMsg() method and call the markSuccessful() method after successful execution. Any operation can throw FrameworkExecutionException or any of its subclasses.

Component can also implement operations, that are meant to be executed in pre/post flow pattern execution blocks, but these operations can't access any endpoint configuration. For these cases the simplified component constructor is recommended.

### Extending SOATF - Adding new components

#### change in the code

Framework component discovery is not implemented yet. Therefore, slight change in the FlowExecutor class itself is needed, for any new component added. First, an update of master configuration schema is needed and all component's new operation names (known at the moment) should be added to the type OperationName. All the operation names must share the same prefix and this prefix will be used in the FlowExecutor's switch block in executeOperation(...) method. In the case statement the following steps should occur:

1. extraction of the required data (endpoint configuration - table, file, entry, etc..) from the interface configuration
2. extraction of the required meta-data (hosts,ports,credentials) from the master configuration
3. initialization of the component - passing these 2 data sets to the components constructor, usually with the component working directory path
4. execution of the operation on the component instance

In the FlowExecutor instance both configuration files are represented in form of java classes and are accessible via properties ICFG & MCFG. Required data can be extracted using these classes methods and passing the environment name, execution block id,and operation's executeOn flag.

Objects visible inside the executeOperation switch case:

* ICFG - interface configuration object, unmarshalled for the actual interface configuration
* MCFG - unmarshaled master configuration object
* envName - actual environment name
* workingDir/patternWorkingDir/rootWorkingDir - string path to the directory for storing component's results, flow pattern's results or scenario's results
* interfaceExecutionBlock - unmarshalled flow pattern instance block with list of endpoint references
* operation - actual operation to be executed unmarshalled from master configuration flow pattern configuration
* ifaceId - name of the selected interface (for which the ICFG object was unmarshalled)

The new component class should extend the AbstractSoaTFComponent class and its constructor should accept the subset of master & interface configuration and also the working directory path, which is usually vital for the components functionality. The component should acquire an instance of OperationResult object upon its creation (calling OperationResult.getInstance()) and use it to note any relevant results within the operation execution.

If the new component implements operations that will be used in flow pattern pre/post operation blocks, then it should contain a simplified constructor which doesn't require the interface endpoint data to be initialized. In the FlowExecutor class, the same steps as for executeOperation method should be repeated in the executeFlowPatternOperation method, except the step 1 - extraction of endpoint data from the interface configuration, because these operations are not bound with any endpoint.

#### Change in the schema

If the new component is designated to be paired with a new technology, or endpoint type, new structures describing those data needs to be added to the master and interface configuration schemas.

In the master configuration schema, the new technology should be described by its own type, extended from master: AbstractMasterConfigEnvironmentInstance, with one required attribute - environment. List of such elements (configuration of the same technology host, for different environment) is then grouped in the type (usualy plural form of the original type name) with one required attribute - identificator. Finally, list of these groups is nested in the Environments section of the master configuration, together with all other technologies. For example, if the technology is FTP, identificator can be 'client data FTP', or 'web images FTP'. For both of these hosts then the same set of environment specifications should be added. For example environment 'dev' or 'prod' then results in 'client data for dev', 'client data for prod', 'web images for dev', 'web images for prod'.

## Testing process

SOATF testing process is designed to verify the integration process step by step. The whole flow is split to separate stages, and each of them has source and the target endpoint defined. The transition between two endpoints itself is the subject of the test. One transition is usually tested within one execution block of the flow pattern, and the number of execution blocks then represents the number of transitions in the integration flow. All testing flow patterns are forged to pause the execution of the next stage in the integration flow, until the actual stage testing is finished and verified.

Within SOATF, the transition is tested indirectly. Either by verifying the data in the target endpoint, or by analysing reporting events of the fusion platform itself. The reporting events of the services involved in the transition are being analysed too, therefore all services should follow the same rules of creating report events. Our UtilComponent was tailored for the needs of the Irish Water programme, utilizing the fact, that all proxy services are sending report pairs ENTRY/EXIT, respectively ENTRY/ERROR upon their execution.

Final target endpoint is always tested using 'black box' approach. It means that SOATF is not accessing and or validating data stored in the final target system. Outcome of the final transition is given by the integration platform reporting events.

Testing process usually involves one positive scenario and set of negative scenarios. For any target endpoint in the flow there should be 2 negative tests included. One test simulating the endpoint being inaccessible for the integration platform and the second test producing invalid data for the endpoint to consume.

## Flow patterns

SOATF comes with a set of predefined flow patterns implemented in the master configuration schema. These flow patterns overlaps and fits with the integration patterns identified within IBM Irish Water Programme.

### Creating new flow pattern

There are 2 determining factors while creating new flow pattern in the master configuration:

1. source endpoint technology
2. number of transitions in the integration flow

The biggest difference between all flow patterns is in the type of the source endpoint and how the source data is constructed or picked up and fed to the flow. There are various components in the SOATF supporting all the known source of data - FTP, SOAP, DB. In the beginning of the test scenario, proper data should be generated using the components method and then fed to the system, where they can be picked up by the first service can pick them up.

If there is more than one transition in the flow, before initiating the first transition, the subsequent service must be paused in order to gather results of the partial flow.

After injecting the data and before collecting the results, framework should wait for a specified amount of time for the transition to take place.

In the beginning of any subsequent transition, the next one in the chain must be paused before the actual one is resumed. Rest of the flow is similar to the first pattern.

Number of transitions determines the number of execution blocks in the flow pattern's successful scenario and number of negative tests.

From the SOATF point of view, if there are multiple services chained with callouts, it is still considered as one transition, with multiplied number of ENTRY/EXIT reports in the flow.

Pseudo scenario for flow pattern with one queue in the middle of 2 transitions would contain these steps:

First execution block:

1. TECHNOLOGY\_GENERATE\_DATA
2. SOAP\_DISABLE\_QUEUE\_CONSUMING\_SERVICE
3. TECHNOLOGY\_FEED\_DATA
4. UTIL\_WAIT\_TECHNOLOGY\_POLLER\_DELAY
5. JMS\_READ\_QUEUE\_MESSAGE
6. JMS\_VALIDATE\_MESSAGE
7. UTIL\_CHECK\_REPORTING\_EVENTS

Second execution block:

1. UTIL\_RESET\_REPORT\_CHECKING\_TIMESTAMP
2. SOAP\_ENABLE\_QUEUE\_CONSUMING\_SERVICE
3. UTIL\_WAIT\_JMS\_POLLER\_DELAY
4. UTIL\_CHECK\_REPORTING\_EVENTS

Scenario like this will perform better with some pre/post operations added to prepare/cleanup the environment for the test:

Scenario pre execution block:

1. JMS\_RESUME\_QUEUE

Scenario post execution block:

1. TECHNOLOGY\_PURGE\_DATA
2. JMS\_DELETE\_QUEUE\_MESSAGES
3. SOAP\_ENABLE\_QUEUE\_CONSUMING\_SERVICE

### IBM Irish Water Programme flow patterns

Based on the information provided in the section 3.2 and 3.3, there has been following patterns prepared for the Irish Water programme:

1. Database -> Queue -> Database/SOAP
2. Database -> Queue -> Queue -> Database/SOAP
3. FTP -> Queue -> Database/SOAP
4. File -> Queue -> SOAP/File
5. SOAP -> Queue -> Database/SOAP/FTP
6. SOAP -> Queue -> Queue -> SOAP
7. SOAP -> Database/SOAP
8. Database -> Database

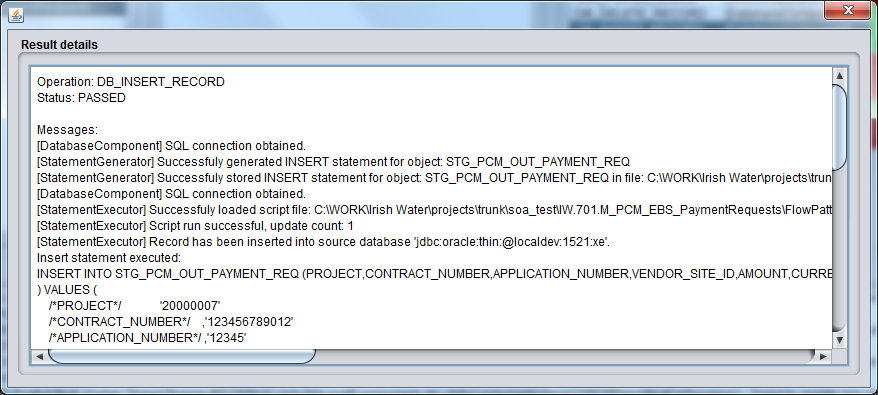
# General JAVA code guidelines

## Object OperationResult

Serves as an indicator of success or failure at the operation's execution end and also holds a list of all messages gathered during the operation's execution. The purpose of the success/failure indicator is obvious - you can tell whether the operation was successful or not (such as if the DB operation DB\_INSERT\_RECORD succeeded at inserting a record to the database). The list of messages is the main source of the data that goes to the test report and also, in case of the GUI, it is used to display what was going on during the operation's execution in a step-by-step manner (check the screenshot below).

Current instance of this class is accessible anywhere in the code simply by calling a static method OperationResult.getInstance(). This instance lives from the very beginning of the operation's execution to its very end. FlowExecutor class takes care of "resetting" it at each operation execution start. No other class outside of com.ibm.soatf.flow package can reset the state of it. By resetting we mean creating entirely new instance of OperationResult, effectively setting the overall success to false and clearing any messages it contained prior to reset.

public void addMsg(String msg):

* use it everywhere you want to record a certain event which will then be displayed in the operation result details window (after double-clicking the row in the results table in GUI)
* each added message automatically contains a name of the class from which you called the method (in brackets): 
* it is considered a good practice to use OperationResult.addMsg() wherever you also log message using the framework's logging mechanism but not necessarily everywhere. It's entirely up to you what you want to appear in the details (maybe you want the log to contain a TRACE message of opening/closing a database connection but it's not really necessary in the report or details window)
* always use it in the place where you are catching a non-framework exception (such as SQLException, IOException, ...), wrapping it in FrameworkException (or its subclass) and rethrowing it. Try to also include the original exception's message
* similarly, use it wherever you are creating and throwing a brand new FrameworkException

public boolean isSuccessful():

* tells if operation's execution ended in success (true) or failure (false)

public void markSuccessful():

* this method sets the operation's execution result to success
* you call this method at the point where you consider the operation succeeded

NOTE: markSuccessful() method introduces a possible danger: once it is called, there is no possibility to change the operation's result back to failure anymore so you should be really careful to call it at the very end of the operation

## Logging

Framework uses Log4j2 as its logging facility so standard logging methods are used. Each class that uses logging should instantiate its own Logger instance at the class body beginning, for example:

public class DatabaseComponent **extends** SOATFComponent **{**

private static final Logger logger **=** LogManager**.**getLogger**(**DatabaseComponent**.**class**);**

**...**

* when you log a message via logger, always consider to call also the OperationResult.addMsg() at the same time
* DO NOT use logger in the catch blocks unless you don't propagate the exception further - this exception is always logged at the end of the operation's execution automatically

## Exception handling

The main exception class is FrameworkException. It has 2 subclasses: FrameworkExecutionException and FrameworkConfigurationException. All these exceptions are checked exceptions.

Each checked exception that is not FrameworkException (or its subclass) - that is, each checked exception thrown by the standard JDK or 3rd party libraries - should be caught and handled appropriately. Unless there are special circumstances you should always rethrow this exception wrapped in one of the framework's exceptions so it eventually propagates to the highest level in the execution chain, FlowExecutor.execute() method. This method is handled automatically and if there was exception thrown by it, it will be logged in the log.

In this example you see SQLException and IOException being handled. Key points here are:

* catch block where message is recorded in the OperationResult object
* new FrameworkExecutionException wrapping the original is created and rethrown
* the method is declared to be throwing FrameworkExecutionException

public static void generateInsertStatement**(**Connection conn**,** DatabaseComponent**.**DbObjectConfig config**,** File file**)** **throws** FrameworkExecutionException **{**

OperationResult cor **=** OperationResult**.**getInstance**();**

**try** **{**

//...

//some logic is here...

//...

String msg **=** "Successfuly stored INSERT statement for object: " **+** objectName **+** " in file: " **+** outputScriptFilePath**;**

logger**.**debug**(**msg**);**

cor**.**addMsg**(**msg**);**

cor**.**markSuccessful**();**

**}** **catch** **(**SQLException ex**)** **{**

String msg **=** String**.**format**(**"Failed to generate INSERT statement. Reason: %s"**,** ex**.**getMessage**());**

cor**.**addMsg**(**msg**);**

**throw** **new** FrameworkExecutionException**(**msg**,** ex**);**

**}** **catch** **(**IOException ex**)** **{**

String msg **=** String**.**format**(**"Failed to save INSERT statement file %s. Reason: %s"**,** outputScriptFilePath**,** ex**.**getMessage**());**

cor**.**addMsg**(**msg**);**

**throw** **new** FrameworkExecutionException**(**msg**,** ex**);**

**}**

**}**

# Build process

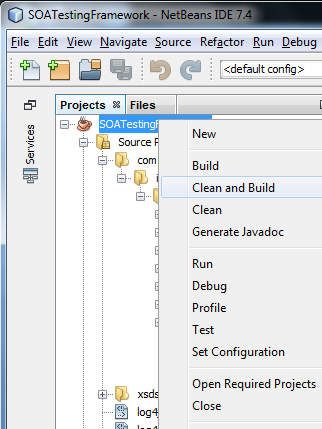
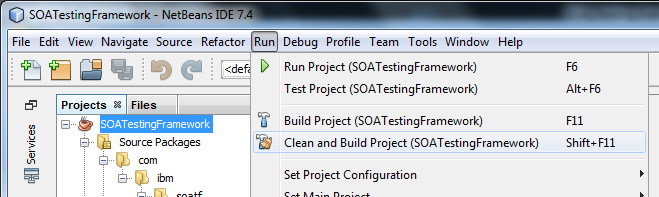
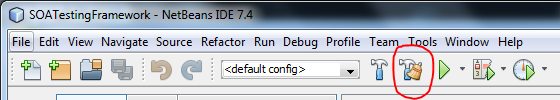
**Requirements:** Ensure that the soa\_testing\_framework\_bin directory exists at the same level as the yout NetBeans project. You can check it out from SVN URL:

http://10.19.12.41:8080/svn/repos\_int1/m2/trunk/soa\_testing\_framework\_bin

Reason for this is because the generated JAR file and other resources (like libraries, XSD schemas) are copied there after the build is finished.

## Building in NetBeans IDE 7.4

Several options here:

1. Right-click on the project in the Projects tab and choose Clean and Build option
2. Use the menu bar at the top and select Run -> Clean and Build Project (SOATestingFramework)
3. Click the "Hammer and Broom" located on the toolbar
4. The quickest method: memorize the shortcut **Shift + F11**, you'll use it a lot ;)

## Building with Ant 1.9.1

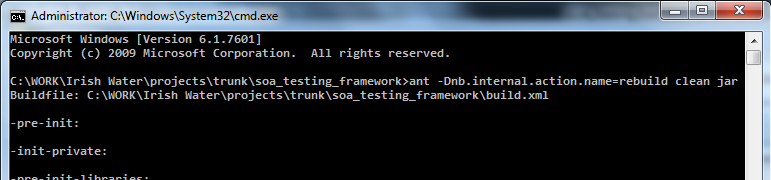
**Requirements:** Ant 1.9.1 (NetBeans IDE has Ant built-in, usually in <NetBeans\_install\_folder>\extide or <NetBeans\_install\_folder>\ide folder

Navigate to your project root level. When you see the file called build.xml (not build-impl.xml), you know you are at the correct place.

From the project root directory run

ant -Dnb.internal.action.name=rebuild clean jar

Example:



## Build process customization

You can customize the build process by defining various targets in the build.xml. Refer to the file itself, each target is documented there. For example, the post-jar stage is added to copy everything needed to the soa\_testing\_framework\_bin directory:

<target name=**"-post-jar"**>

<echo message=**"Deleting ${bin.dist.lib.dir} dir"**/>

<delete dir=**"${bin.dist.lib.dir}"**/>

<echo message=**"Deleting ${bin.dist.schema.dir} dir"**/>

<delete dir=**"${bin.dist.schema.dir}"**/>

<echo message=**"Deleting ${bin.dist.schema.dir} dir"**/>

<delete dir=**"${bin.dist.schema.dir}"**/>

<echo message=**"Deleting ${bin.dist.reporting.dir} dir"**/>

<delete dir=**"${bin.dist.reporting.dir}"**/>

<echo message=**"Deleting ${bin.dist.jar} file"**/>

<delete file=**"${bin.dist.jar}"**/>

<echo message=**"Copying ${schema.dir} dir to ${dist.dir}"**/>

<copy todir=**"${dist.dir}"**>

<fileset dir=**"${basedir}"** includes=**"${schema.dir}/\*\*"**/>

</copy>

<echo message=**"Copying ${reporting.dir} dir to ${dist.dir}"**/>

<copy todir=**"${dist.dir}"**>

<fileset dir=**"${basedir}"** includes=**"${reporting.dir}/\*\*"**/>

</copy>

<echo message=**"Copying ${dist.dir} dir to ${bin.dist.dir}"**/>

<copy todir=**"${bin.dist.dir}"**>

<fileset dir=**"${dist.dir}"** includes=**"\*\*"**/>

</copy>

</target>

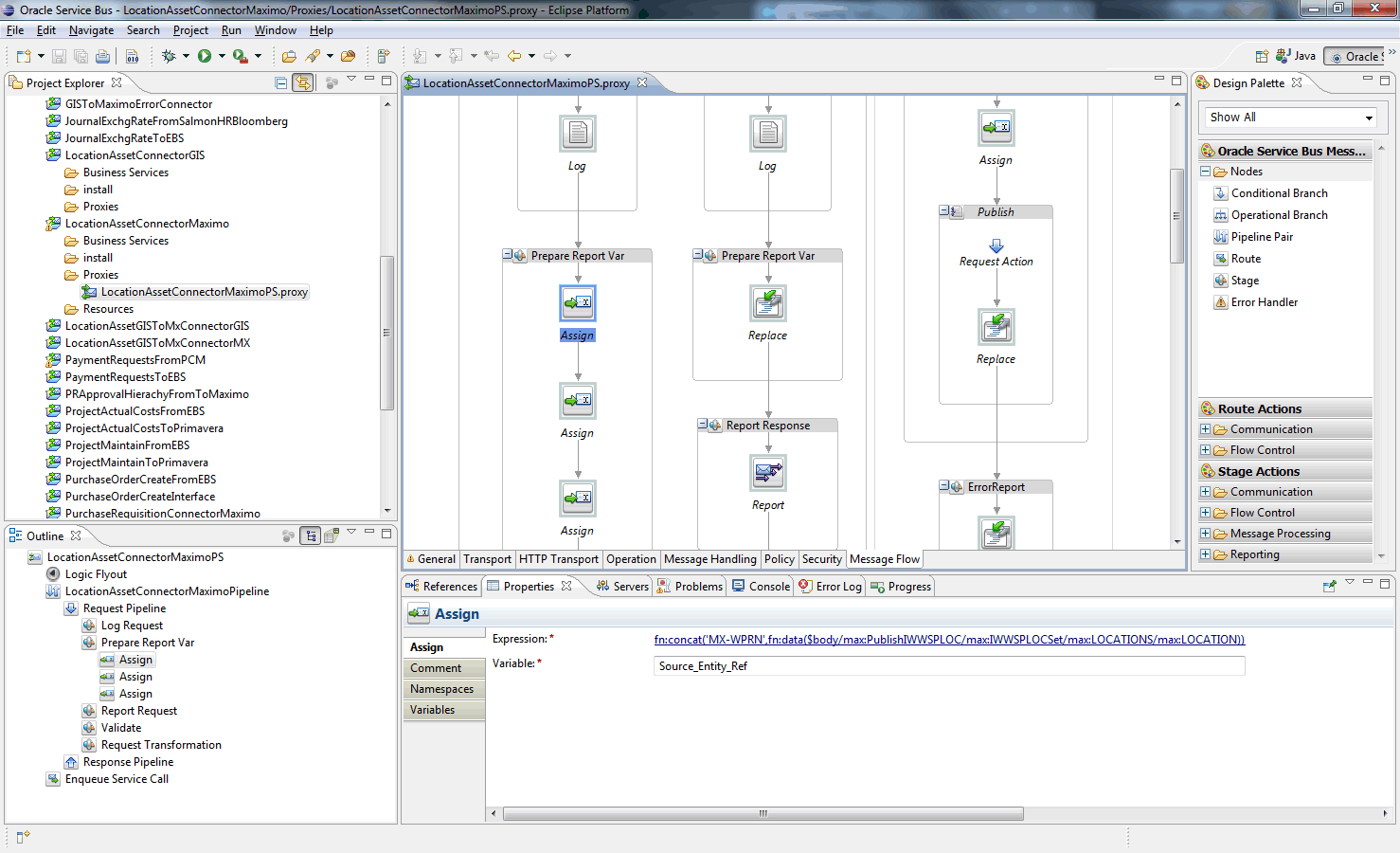
# Tips & Tricks

## Entity Reference and Message Reference

This tip deals with a non-standard way of consructing and dealing with the Source\_Entity\_Ref and/or Source\_Message\_Ref that can cause us grief with the UTIL\_CHECK\_REPORTING operations.

The standard way is either a single field taken from the source message or a concatenated combination of fields delimited by a hyphen character (-) but unfortunately it's not always the case.

This tip is to show you how entity\_ref is constructed, concatenating a constant string MX\_WPRN (static) with the field taken from the envelope (dynamic):



Source\_Entity\_Ref is constructed as fn:concat('MX-WPRN',fn:data(**$body**/max:PublishIWWSPLOC/max:IWWSPLOCSet/max:LOCATIONS/max:LOCATION))

So you see it's a constant concatenated with a dynamic value from the message. There's no way to define it in the config.xml the "standard" way, i.e. using the sourceEntityRef="true" flag for the field, because:

1. the value is NOT concatenated using hyphen (-) as a delimiter (which is a standard way of doing this)

2. one of the values is a constant, and not a value taken from the message field

The only thing you can do is to define the entityRef completely manually here at the root element of the envelope config:

<soapconf:defaultEnvelopeConfig **sourceEntityRef="MX\_WPRN1360302"**>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/@messageID" sourceMessageId="true" elementValue="1383565717919330777"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/@action" sourceEntityRef="false" elementValue="Replace"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:LOCATION" sourceEntityRef="true" elementValue="1360302"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:LOCATIONSID" sourceEntityRef="true" elementValue="2979518"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:IWMETERGISPARAM1" sourceEntityRef="false" elementValue="-6.2499545"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:IWMETERGISPARAM2" sourceEntityRef="false" elementValue="53.3526186"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:STATUS/@maxvalue" sourceEntityRef="false" elementValue="NOT READY"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:STATUS" sourceEntityRef="false" elementValue="PLANNED"/>  
 <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:STATUSDATE" sourceEntityRef="false" elementValue="2013-07-06T03:27:29+01:00"/>  
 </soapconf:defaultEnvelopeConfig>

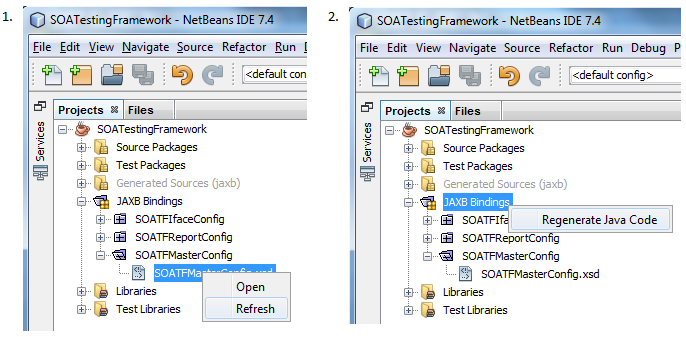
This is not the recommended approach and **SHOULD NOT BE USED** if standard approach can be used (i.e. marking the fields with the sourceEntityRef="true" flag) because you have to keep in mind that all the dynamic fields should be in sync with your manual value, so in this example be careful about the <soapconf:element elementXpath="//\*:PublishIWWSPLOC/\*:IWWSPLOCSet/\*:LOCATIONS/\*:LOCATION" sourceEntityRef="true" elementValue="1360302"/> field.

## What do I do when I want to add something new to the XSD schema(s)?

If you searched thoroughly, you already found that there are actually 3 locations that you can find the XSDs for the framework but only 1 is the correct one to edit in order to introduce some changes. It is in the source code directory:

<http://10.19.12.41:8080/svn/repos_int1/m2/trunk/soa_testing_framework/schema>

When you are satisfied with the change you made to the schema, you need to refresh it and regenerate the JAXB code so that your change (e.g. newly added XML attribute) is available for use in the NetBeans code. To do that:



The last thing to do is do a clean build, so that the changes are transferred to the soa\_testing\_framework\_bin folder as well.