# *The Obix Configuration Framework: Developer and User Guide*

By

Obi Ezechukwu

May 06

## Abstract

The Obix Configuration Framework—referred to in this document simply as “the framework”—is an XML and Java based configuration framework which provides developers with the ability to easily and quickly develop configurable software applications. To paraphrase, it provides facilities for utilising XML-based configuration files in modern application software, by providing standardised XML schema definitions for configuration data, and, easy to use API for loading/auto-reloading and general management of such data. This document is a user and developer guide which describes the framework in detail, its features, API, structure and design rationale, and the various options for integrating it into software applications. This guide is divided into three main chapters, the first of which covers the basic concepts of the framework, providing simple and practical “quickstart” examples to demonstrate its features, and how it can be integrated into application software. The second chapter covers the structure of the framework, providing greater detail on the layout of configuration documents, as well as a “beneath the scenes guide” to the implementation packaged in the downloadable “base” distribution—including main interfaces and Java classes. Finally, the third chapter focuses exclusively on the alternative approaches for integrating the framework into application code. This chapter covers the basic code-based approach as well as more advanced approaches, applicable to J2EE and JMX environments, which require little or no coding. It details the various framework extensions, provided in the base distribution, which facilitate the use of open source frameworks such as Apache Log4j and Hibernate.

TABLE OF CONTENTS

# 1 Introduction

Majority of Java applications, regardless of size, will to some extent utilize a configuration mechanism that enables the application's behaviour to be modified without code changes. In fact one of the major reasons for the use of configuration is to externalize values, which are likely to change frequently, and which if embedded directly into code would lead to all-to-frequent code changes, re-compilation and re-deployment. Examples of such values abound, and they include: database connection strings; switching/control values—to enable or disable a certain piece of functionality; environment (machine) specific entries—file paths, resource locators etc. Externalizing such values simplify application support and reduce costs, as well as increasing development throughput and minimizing system outages.

If you are reading this document, then you probably already realise the importance of configurable software, and are in the process of implementing/modifying such software; consequently we will not bore you with the rationale behind software configuration. We will instead proceed to tell you, briefly, in comparison to other approaches for implementing configurable software, what this framework offers you/your organization.

## 1.1 Feature Summary

- ***Use of XML configuration files.*** XML not only provides a mechanism for encoding data so that it is portable across applications and environments, but also supports a tree-structure which, when compared to other encoding mechanisms, allows for denser and more structured data. Many Java frameworks and tools such as Log4J and ANT already utilise XML files as their standard configuration mechanism—and for good reason too. Not only is the configuration data better structured—and consequently easier to understand—but XML enables data-relationships to be better represented. The framework mandates that configuration data be specified in XML format, for similar reasons (and more) as will become clearer later on in this document.
- ***Support for componentization/modularization of configuration data.*** The framework recognizes that in a number of cases, it is necessary to modularize/componentize configuration data; as such, it provides facilities to support this. The framework organises a configuration set—a set of related configuration files—around the concept of modules, where each module can contain other modules. The child modules can either be defined inline—explicitly within child XML elements—or as references to other documents. A configuration document is thus a high-level module of configuration which can contain other modules—either defined within the same document or referenced from an external source. There is no limit to the depth of modules, so child modules can have their own children, and so on. At the top of the configuration tree is a root module, which is the ancestor of all modules in the configuration set.
- ***Easy to use API.*** The framework's API is designed around a "KISS"—"*keep it simple stupid"*—philosophy, making it as intuitive as possible; the objective being to flatten the learning curve involved in using it. Thus the mechanisms for integrating it into application code are relatively simple.
- ***Support for Java specifications and other open source API.*** The framework provides extensions which are built around Java standards such

as the servlet specification, JMX[1] and JNDI. These utilities generally provide alternative (and code free) means of integrating the framework into application code. Other utilities, also provided in the obix distribution, simplify the use (specifically initialization) of other open source toolkits, such as Log4J and Hibernate (see section 3.3 for concrete examples).

- ***Extensibility.*** The framework provides event notification interfaces, which enable application developers to build upon, and extend its functionality. It is also divided into two core parts, a specification, and an implementation, where the specification governs interaction with application code, and the implementation remains hidden but manages the details of the interaction e.g. data loading, reloading, and JNDI binding etc. As such, developers are able to use their own implementation if they wish to do so, or, to extend the implementation provided in the obix distribution. The fact that its source is freely obtainable also facilitates this.
- ***Free use of binaries and open access to source code.*** The framework is completely free to use, and there are no restrictions on access to, or modification of its source files or binaries. In other words, it is truly open source without any strings attached.

## 1.2 Quick-Start Examples

In this subsection we illustrate, by example, how to integrate the framework—using a basic approach—into a Java application. We provide three example applications, the first of which consists of a simple Java class and a basic configuration file. The purpose of this application is to demonstrate how to create a simple configuration file, and how to access its constituent data values in a Java program. The subsequent example applications are refinements of the first, and are aimed at simulating more realistic real-world scenarios. The entire source code and corresponding configuration files for these examples are included in the binary archives, downloadable from the project website http://obix-framework.sourceforge.net.

### 1.2.1 A Basic Configuration File and Java Application

We start by creating a very basic configuration file—illustrated below—which holds simple database connection information.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <entry entryKey="database.url">
   <value>jdbc:mysql://localhost:2500/mySQLdb</value>
 </entry>
 <entry entryKey="database.userId">
  <value>appuser</value>
 </entry>
 <entry entryKey="database.password">
  <value>nopassword</value>
 </entry>
</configuration>
```

As you can see, the configuration file has three entries, each with a single value. To run this example, you will have to save this configuration file to a location, which we assume to be *"C:/ObixExampleApp/config/simple-configuration.xml"*—if you wish, you can change this to another location that is more convenient for you.

All we need now is a Java application which accesses the configuration values. For simplicity, we provide a simple class—shown in the following listing—which just reads the configuration values and prints them out.

[1] *Java Management Extensions*

```
import org.obix.configuration.Configuration;
import org.obix.configuration.ConfigurationAdapter;
import org.obix.configuration.ConfigurationAdapterFactory;

public class ConfigUser
{
 public static void main(String[] args)
 {
  String configLocation = "file:/C:/ObixExampleApp/config/simple-
configuration.xml";

  try{

  ConfigurationAdapterFactory
adapterFactory=ConfigurationAdapterFactory.newAdapterFactory();

  ConfigurationAdapter adapter=adapterFactory.create(null);

  adapter.setValidate(false);
  adapter.adaptConfiguration(Configuration.getConfiguration(),

                                    configLocation);

  String dbURL =
   Configuration.getConfiguration().getStringValue("database.url");

  String dbUserId =
   Configuration.getConfiguration().getStringValue("database.userId");


  String dbPassword =
      Configuration.getConfiguration().getStringValue("database.password");

  System.out.println ("DB URL : " + dbURL);
  System.out.println ("DB UserId :  " + dbUserId);

  System.out.println ("DB Password : " + dbPassword);

  }catch (Exception exce){
    exce.printStackTrace();
  }
 }//end method main(..)

}//end class
```

To run this class, you will need the framework library, ***obix-framework.jar***, which can be found in the *'lib'* folder of the obix distribution. You will also need the following third-party open-source libraries, ***dom.jar***, ***jaxen-full.jar***, ***sax.jar***, ***saxpath.jar*** and ***xercesImpl.jar***, which can be found in the *'lib/thirdPartyLibs'* folder of the obix distribution.

When executed, the class will produce the following output.

```
DB URL :jdbc:mysql://localhost: 2500/mySQLdb
DB UserId :appuser
DB Password :nopassword
```

To explain the inner workings of this example, we examine the example class *(ConfigUser.java)* in more detail. The class starts with a string declaration encapsulating the location of the configuration document (**note** that this declaration depends on the path to which the supplied document is saved). Next we create an adapter-factory instance:

```
  ConfigurationAdapterFactory
adapterFactory=ConfigurationAdapterFactory.newAdapterFactory();
```

The above statement returns a new instance of the default factory—packaged with the distribution. This default behaviour can be overridden by specifying an adapter-factory implementation at execution time—as the value of the runtime property *'org.obix.configuration.AdapterFactory'.*

Next we create an adapter instance as shown below.

```
ConfigurationAdapter adapter=adapterFactory.create(null);
adapter.setValidate(false);
```

Validation is disabled in this example, which means that the adapter will not use a validating xml parser to parse the configuration document. This is not recommended—at least not for development—but is used here for simplicity.

Using the adapter instance, we read the configuration data into a configuration object.

```
adapter.adaptConfiguration(Configuration.getConfiguration(),
                                     configLocation);
```

Notice that the configuration data is read into the global/static configuration instance—obtained by the invocation *'Configuration.getConfiguration()'*. This is ideal for scenarios where you want the configuration data to be accessible statically across your application. In other situations this may not be ideal; it may be more appropriate to create a new (non-static) configuration instance. An example of where not to use the static instance is a load-balanced/distributed J2EE environment—where the static context may be meaningless, and, where it may be more ideal to use an instance bound into a JNDI context (see section 3.2 for options).

### 1.2.2 Modules and Document-Import (Include) Example

For this application, assume that the task is to create an order-management system of some sort, where there are several databases as opposed to a single one. Assume also that for this application, there is a need to cache the static-data [from each individual database] on the file-system so as to remove the need for database fetches; and that achieving this involves specifying temporary file paths in the configuration set—mapped to the relevant database by grouping it with the connection information for the database.

We proceed by modifying the configuration file for example, so as to create a configuration set where the connection information for the global database is stored in the root module, and, where there are sub-modules for the other databases—called ***price***, ***cust*** (for customer data), and ***order***. The resulting file is as follows (**note**: the new entries/changes are shown in bold font):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <entry entryKey="global.database.url">
    <value>jdbc:mysql://localhost:2500/globalDb</value>
  </entry>
  <entry entryKey="database.userId">
    <value>appuser</value>
  </entry>
  <entry entryKey="database.password">
    <value>nopassword</value>
```

```
      </entry>

      <!--Connection info for prices database-->
      <configuration-module moduleId="price.db.connection.info">
        <entry entryKey="database.url">
          <value>jdbc:mysql://localhost:2501/pricesDb</value>
        </entry>
        <entry entryKey="database.userId">
          <value>pricedbuser</value>
        </entry>
        <entry entryKey="database.password">
          <value>nopassword</value>
        </entry>

        <!--sub configuration module which holds temp
           filespace details-->
        <configuration-module moduleId="price.tmp.space">
          <entry entryKey="data.cache">
             <value>C:/ObixExampleApp/data/price.cache</value>
          </entry>
        </configuration-module>
      </configuration-module>

      <!--Connection info for customer database-->
      <configuration-module moduleId="cust.db.connection.info">
        <entry entryKey="database.url">
          <value>jdbc:mysql://localhost:2502/custDb</value>
        </entry>
        <entry entryKey="database.userId">
          <value>custdbuser</value>
        </entry>
        <entry entryKey="database.password">
          <value>nopassword</value>
        </entry>

        <!--import the configuration file
            which holds the temp filespace details
        -->
        <configuration-import isRelativeLink="true"
          moduleId="cust.tmp.space">
          cust.cache.configuration.xml
        </configuration-import>
      </configuration-module>

      <!--Connection info for the order database-->
      <configuration-import isRelativeLink="true"
          moduleId="order.db.connection.info">
          order.db.configuration.xml
      </configuration-import>
    </configuration>
```

Rather than delving straight into the Java class which makes use of this document, we will first of all dissect it, exploring its different elements and what they mean; and for each element we present corresponding Java code for accessing its contents. We will start with the connection information for the *price* database, which is declared as a module. The declaration for this module (also shown above) is re-iterated as follows:

```
    <!--Connection info for prices database-->
      <configuration-module moduleId="price.db.connection.info">
                                             .
                                             .
                                             .
        <!--sub configuration module which holds temp
           filespace details-->
        <configuration-module moduleId="price.tmp.space">
```

```
                                        .
                                        .
        </configuration-module>
      </configuration-module>
```

To access this module programmatically, we use the following code:

```
  Configuration priceDbConfig = Configuration.getConfiguration().
                         getModule("price.db.connection.info");
```

This call returns the configuration module (node) with id *"price.db.connection.info"*—as you will notice all configuration modules are encapsulated by instances of the class *'org.obix.configuration.Configuration',* thus each instance is effectively a tree node which can have descendants. Note that in the configuration file, the module *"price.db.connection.info"* also consists of an ***inline*** module, with id *"price.tmp.space"*, encapsulated within a child *<configuration-module>* element. A reference to this child module can be obtained as follows:

```
       Configuration priceTempSpaceConfig =priceDbConfig.
                                           getModule("price.tmp.space");
```

There is no programmatic limit to the depth of a configuration tree, so a configuration file can consist of several child modules, which themselves can also have descendants. Thus a *<configuration-module>* element can contain several other *<configuration-module>* nodes, which in themselves can also have child nodes. Thus the following structure is possible.

```
    < configuration-module …>
     <configuration-module …>
       <configuration-module …>
           …….
       </configuration-module>
     </configuration-module>
    </configuration-module >
```

The full text of the test method which accesses and prints out the configuration values related to the *price* database is as follows:

```
public static void printPriceDbConfigInfo()
{
  Configuration priceDbConfig = Configuration.getConfiguration().
                         getModule("price.db.connection.info");

  Configuration priceTempSpaceConfig =
     priceDbConfig.getModule("price.tmp.space");

  System.out.println ("Price DB URL : " + priceDbConfig.getStringValue
                                              ("database.url"));

  System.out.println ("Price DB UserId : " + priceDbConfig.getStringValue
                                                              ("database.userId"));

  System.out.println ("Price DB Password : " + priceDbConfig.getStringValue

                                ("database.password"));

  System.out.println ("Price DB File Cache : " +
priceTempSpaceConfig.getStringValue
                                                                   ("data.cache"));

 }
```

Next, consider the configuration information for the *cust* database, which is encapsulated within the following element.

```
<!--Connection info for customer database-->
<configuration-module moduleId="cust.db.connection.info">
                                                .
                                                .
                                                .
  <!--import the configuration file
      which holds the temp filespace details
  -->
  <configuration-import isRelativeLink="true"
       moduleId="cust.tmp.space">
    cust.cache.configuration.xml
  </configuration-import>
</configuration-module>
```

This node is largely the same as the element encapsulating the data for the *price* database, with the main distinction being that we use a *<configuration-import>* node to include the configuration data for the file cache, as opposed to an inline module [declared within a *<configuration-module>* node]. The *<configuration-import>* element can occur as a child of either a
*<configuration>* or *<configuration-module>* node. It requires two attributes, the first of which [the *'isRelativeLink'* attribute] determines whether or not the import path —the node's body content—is to be resolved relative to the directory (the URI context) of the encapsulating configuration document [in this case *'C:/ObixExampleApp/config/'*] or as an absolute URI. Considering that a configuration document can be imported at several points, and possibly with different connotations—i.e. re-used in several scenarios—the second (and mandatory) attribute *'moduleId'* provides a unique id [within the scope of the importing element] for the imported configuration-data.

The imported module can be accessed as follows:

```
Configuration custDbConfig =Configuration.getConfiguration().
                           getModule("cust.db.connection.info");

Configuration custTempSpaceConfig =custDbConfig.getModule("cust.tmp.space");
```

As you can see from the above code segment, there is no difference—from the perspective of the Java application—between an inline module and an imported one. The use of either is really down to choice, but an import is probably best suited for large amounts of data which can be externalised so as to make the importing file more legible. Next, we examine, the imported file '*cust.cache.configuration.xml'* listed next:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <entry entryKey="data.cache">
   <value>C:/ObixExampleApp/data/cust.cache1</value>
   <value>C:/ObixExampleApp/data/cust.cache2</value>
   <value>C:/ObixExampleApp/data/cust.cache3</value>
   <value>C:/ObixExampleApp/data/cust.cache4</value>
 </entry>
</configuration>
```

Notice that the entry *'data.cache'* has four values as opposed to a single one; this demonstrates that a configuration-entry is not restricted to a single value. Actually, **it can have one or more** values, thus rendering it to situations where you need to associate multiple values to the same key/identifier. Examination of the method listed below demonstrates how these values can be accessed. Note that the first value, with index '0', is the default value, and as such we don't have to supply an index when accessing it—although you can if you wish to.

```
public static void printCustDbConfigInfo()
{
  Configuration custDbConfig =Configuration.getConfiguration().
                         getModule("cust.db.connection.info");

  Configuration custTempSpaceConfig = custDbConfig.getModule("cust.tmp.space");

  System.out.println ("Cust DB URL : " +
                 custDbConfig.getStringValue("database.url"));

  System.out.println ("Cust DB UserId : " +
                 custDbConfig.getStringValue("database.userId"));

  System.out.println ("Cust DB Password : " +
                 custDbConfig.getStringValue("database.password"));

  System.out.println ("Cust DB File Cache 1: " +
                 custTempSpaceConfig.getStringValue("data.cache"));

  System.out.println ("Cust DB File Cache 2: " +
                custTempSpaceConfig.getStringValue("data.cache",1));

  System.out.println ("Cust DB File Cache 3: " +
                custTempSpaceConfig.getStringValue("data.cache",2));

  System.out.println ("Cust DB File Cache 4: " +
                custTempSpaceConfig.getStringValue("data.cache",3));

}
```

When invoked, this method should produce the following output:

```
Cust DB URL : jdbc:mysql://localhost:2502/custDb
Cust DB UserId : custdbuser
Cust DB Password : nopassword
Cust DB File Cache 1: C:/ObixExampleApp/data/cust.cache1
Cust DB File Cache 2: C:/ObixExampleApp/data/cust.cache2
Cust DB File Cache 3: C:/ObixExampleApp/data/cust.cache3
Cust DB File Cache 4: C:/ObixExampleApp/data/cust.cache4
```

Finally, we examine the configuration information for the *orders* database, revisited in the following listing:

```
<!--Connection info for the order database-->
<configuration-import isRelativeLink="true"
      moduleId="order.db.connection.info">
      order.db.configuration.xml
</configuration-import>
```

As should already be clear, the node <*configuration-import*>, imports the contents of the file *'order.db.configuration.xml'* as a configuration module with identifier *'order.db.connection.info'*. The contents of the imported document are listed next.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <entry entryKey="database.url">
   <value>jdbc:mysql://localhost:2503/orderDb</value>
 </entry>
 <entry entryKey="database.userId">
  <value>ordermanager</value>
 </entry>
```

```
    <entry entryKey="database.password">
     <value>nopassword</value>
    </entry>

    <configuration-module moduleId="order.tmp.space">
     <entry entryKey="data.cache">
      <value>C:/ObixExampleApp/data/order.cache</value>
     </entry>
    </configuration-module>
   </configuration>
```

The following method accesses and prints out the values of the *'order.db.connection.info'* module.

```
public static void printOrderDbConfigInfo()
{
  Configuration orderDbConfig =
          Configuration.getConfiguration().
            getModule("order.db.connection.info");

  Configuration orderTempSpaceConfig =orderDbConfig.
                              getModule("order.tmp.space");

  System.out.println ("Order DB URL : " +
                 orderDbConfig.getStringValue("database.url"));

  System.out.println ("Order DB UserId : " +
               orderDbConfig.getStringValue("database.userId"));

  System.out.println ("Order DB Password : " +
               orderDbConfig.getStringValue("database.password"));

  System.out.println ("Order DB File Cache : " +
              orderTempSpaceConfig.getStringValue("data.cache"));

 }
```

As is evident from the above method, the imported module is accessed in the same way as an inline module; thus demonstrating that the Java API is independent of the module definition approach used. This method, when invoked, should produce the following output.

```
Order DB URL : jdbc:mysql://localhost:2503/orderDb
Order DB UserId : ordermanager
Order DB Password : nopassword
Order DB File Cache : C:/ObixExampleApp/data/order.cache
```

**NOTE:** The Java class for this application is *ConfigUserV2.java* which is included in the examples archive, as are the referenced configuration documents.

### 1.2.3 Auto-Reload Example

The framework provides auto-reload capabilities for configuration data, thus making it possible to make "hot" configuration changes. This is provided for applications where the ability to change configuration entries at runtime is required. To paraphrase, it is provided for applications which require that configuration changes are detected and applied without the application having to be restarted. Without this feature, the alternative of course would be to restart the application each time the configuration data is amended, which is far from desirable in scenarios where system downtime is intolerable.

To demonstrate this feature, we present a simple configuration document—listed below—which we assume will be saved as *'C:/ObixExampleApp/config/editable-configuration.xml'*.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration reloadOnModify="true"  reloadInterval="1000">
 <entry entryKey="test-value">
   <value>Initial Value</value>
 </entry>
</configuration>
```

Notice that the *<configuration>* node has two new attributes (highlighted in bold); the value *'true'* assigned to first one, *'reloadOnModify'*, instructs the framework to monitor the configuration document for changes; the second attribute *'reloadInterval'* is non-mandatory as it has a default value—discussed later—but we specify it here for completeness. The value of this attribute dictates the interval (in milliseconds) at which the framework should inspect the configuration document for changes. Evidently the shorter the interval, the higher the strain it places on virtual-machine resources; hence for any given application, the value used is a judgment-call on the part of the developer/system-administrator.

The following code listing presents a simple class, which we use to demonstrate the framework's ability to detect and re-apply configuration changes at runtime. The class reads, and prints out, the initial value of the configuration-entry *'test-value'* and loops continuously (with a wait interval of 998 milliseconds). In each iteration of the loop, it re-reads the value of the entry, and when it detects that a new value has been applied by the framework, it prints out the new value and breaks from the loop.

```
import java.util.Calendar;
import org.obix.configuration.Configuration;
import org.obix.configuration.ConfigurationAdapter;
import org.obix.configuration.ConfigurationAdapterFactory;

public class ConfigUserV3
{

 public static void main(String[] args)
 {
  String configLocation = "F:/ObixExampleApp/config/editable-configuration.xml";

  try
  {
   ConfigurationAdapterFactory adapterFactory=
        ConfigurationAdapterFactory.newAdapterFactory();

   ConfigurationAdapter adapter=adapterFactory.create(null);

   adapter.setValidate(false);
     adapter.adaptConfiguration(
       Configuration.getConfiguration(),configLocation);

   String initialValue= Configuration.
            getConfiguration().getStringValue("test-value");

   System.out.println("Value at time '" +
                  Calendar.getInstance().getTimeInMillis() +
             "' is '" + initialValue+"'");

   int sleepInterval = 998;
   String newValue;
   while (true)
   {
    Thread.sleep(sleepInterval);

    newValue= Configuration.getConfiguration().getStringValue("test-value");

    if (!initialValue.equals(newValue))
    {
     System.out.println("Value at time '" +
              Calendar.getInstance().getTimeInMillis() +"' is '"
                                                +newValue+"'");
     break;
    }

   }
  }
  catch (Exception exce)
  {
   exce.printStackTrace();
  }
 }//end method main(..)

}//end class
```

If we execute this class, it should produce initial output similar to the following:

```
Value at time '1129846864390' is 'Initial Value'
```

To test the auto-reload feature, we modify (using a suitable text/xml editor) the value of the entry with id *'test-value'* to something like "Modified Value", as shown in the file listing below (**note:** new value shown in bold):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration reloadOnModify="true"  reloadInterval="1000">
 <entry entryKey="test-value">
   <value>Modified Value</value>
 </entry>
</configuration>
```

On saving/committing the change, the test-application will produce output similar to the following, and then exit.

```
Value at time '1129846896328' is 'Modified Value'
```

### 1.2.4 Examples Summary

In this section, we have used three example applications to demonstrate how the framework can be easily integrated into a Java application. We have also explored the structure of configuration documents, demonstrating how configuration data can be organised into modules, and how external self-contained documents can be imported as configuration modules. It has also been demonstrated that the API for accessing configuration modules is independent of the manner in which the modules are declared. Finally, we have demonstrated that the framework is capable of automatically detecting and applying changes to configuration data, without the need for application restarts.

# 2 Framework Structure

In order to aid extension, and to sow the seeds of re-use, the framework is organised around the concepts of a specification and an implementation. The specification effectively defines the interaction between the framework and applications, whilst an implementation manages the details of this interaction. This is to enable developers to either supply their own implementations, or to modify the default implementation, so as to suit their requirements.

The framework is packaged with a default implementation codenamed *Janex* (see section 2.2), and as such there is no need to develop your own. The Janex implementation should meet most requirements, however, where this is not the case, it is possible to either build extensions on top of it, or replace it completely without impacting application code.

The following subsections explain the specification in greater detail, and describe the constituent parts of the Janex implementation.

## 2.1 Specification Overview

The specification, as already mentioned, defines the framework's core interfaces and classes—including the responsibilities of implementing classes and extensions—which in turn define how applications interact with it. The specification also defines the structure of configuration documents, thus establishing the rules by which configuration documents are governed irrespective of the underlying implementation.

### 2.1.1 Configuration Document Structure

This section explores the structure of a configuration document. To paraphrase, it provides a textual (and functional) description of the configuration XML schema/doctype. The aim is to give the reader a sufficient understanding of the elemental structure of configuration documents, by describing the practical functions of each element type. We begin by describing how to create a skeletal configuration document.

#### 2.1.1.1 Document Skeleton

A configuration document begins with the root element <*configuration*>, and, as such, it will have the following basic (and mandatory) structure.

```
<configuration>
        ………..
</configuration>
```

The <*configuration*> element supports the following attributes:

| name | value-type | use | default | description |
|---|---|---|---|---|
| reloadOnModify | boolean | optional | false | Indicates whether or not the configuration document should be monitored for changes. When set to *'true'* all changes to the document are synchronized with the corresponding *org.obix.configuration.Configuration* instance. |
| reloadInterval | long | optional | N/A | Only valid where the attribute *reloadOnModify* has a value of *true*. It indicates the interval, in milliseconds, at which the document should be examined for changes. The lower the interval, the more frequent the document is inspected for changes; the higher the interval, the less the frequency. Where a value is not specified, the implementation will use a default value—which for the *Janex* implementation is 60000 milliseconds. |

Example usage of this element is as follows:

```
<configuration reloadOnModify="true" reloadInterval="86400000">
 ……
</configuration>
```

#### 2.1.1.2 Lifecycle Listener Declarations

The framework enables application developers to specify listeners in a configuration document, which are invoked during the loading of the document. All listeners must implement the interface *org.obix.configuration.ConfigurationLifecycleListener*. Listeners are either invoked before the configuration data is loaded into memory, or afterwards; consequently, and logically, listeners are declared either at the start, or end, of the configuration document. Note that, where listeners are declared at the start of the document, they will not have access to the configuration-data in the document, as the data would not have been loaded into memory when they are invoked. On the other hand, listeners that are declared at the end of a document will have access to all the data in the document (including imported modules), as the data would have been loaded pre the listener invocation.

Listeners to be invoked pre document loading are declared under the element *<execute-before-adapt>*, which is valid at the start of the configuration document. Each listener is instantiated with a *<listener>* element. Listeners to be invoked post document loading are declared under the element *<execute-after-adapt>* (which is valid at the end of the configuration document) and are also instantiated using *<listener>* elements.

The *<listener>* element has a single attribute, described in the following table:

| name | value-type | use | default | Description |
|---|---|---|---|---|
| class | string | mandatory | N/A | The fully qualified class-name of the listener. The class must implement the interface *org.obix.configuration.ConfigurationLifecycleListener* |

It is possible to supply arguments/parameters to a listener using the *<parameters>* element. Each parameter has a single key, can have multiple values, and is encapsulated in a *<parameter>* element. The tag has a single attribute, described next:

| name | value-type | use | default | description |
|---|---|---|---|---|
| entryKey | string | mandatory | N/A | The key/identifier for the argument. Note that arguments are supplied to the listener implementation as a *java.util.Map* implementation, and as such, each list of values have to be keyed against a name/key/id. |

Listener parameters can have multiple values for each name/key/identifier, where each value is encapsulated by a *<value>* tag.

Below is an example of a listener declaration, which causes the Apache Log4J extension (implemented as a listener) to be invoked before the configuration data, in the given document, is loaded. The Apache Log4J listener is discussed in greater detail in section 3.3.1.

```
<configuration>
 <execute-before-adapt>
  <listener class="org.obix.janex.ext.log4j.ObixLog4jAdapter">
   <parameters>
    <parameter entryKey="log4j.configuration.file">
     <value>log4j-config.xml</value>
    </parameter>
    <parameter entryKey="log4j.configuration.file.resolution.policy">
     <value>RELATIVE</value>
    </parameter>
   </parameters>
  </listener>
  </execute-before-adapt>
```

```
            ……………………………………………………………………………………………………..

                                 [CONFIGURATION DATA GOES HERE]

            ……………………………………………………………………………………………………..


    </configuration>
```

The next example, demonstrates a declaration of a listener (highlighted in bold) which is to be invoked only after the data contained within the configuration document is loaded. For the listener implementation, we use the Hibernate extension which is described in section 3.3.2, and which is implemented as a lifecycle listener.

```
   <configuration reloadOnModify="true" reloadInterval="86400000">
    <execute-before-adapt>
     <listener class="org.obix.janex.ext.log4j.ObixLog4jAdapter">
      <parameters>
       <parameter entryKey="log4j.configuration.file">
        <value>log4j-config.xml</value>
       </parameter>
       <parameter entryKey="log4j.configuration.file.resolution.policy">
        <value>RELATIVE</value>
       </parameter>
      </parameters>
     </listener>
     </execute-before-adapt>

           ……………………………………………………………………………………………………..

                                 [CONFIGURATION DATA GOES HERE]

           ……………………………………………………………………………………………………..
     <execute-after-adapt>
      <listener
   class="org.obix.janex.ext.hibernate.HibernateConfigurationInitializer">
       <parameters>
        <parameter entryKey="hibernate.mapped.classes">
         <value>com.myorg.phonebook.Address</value>
         <value> com.myorg.phonebook.Person</value>
        </parameter>
       </parameters>
      <listener>
     </execute-after-adapt>

    </configuration>
```

### 2.1.1.3 Configuration Entries

Configuration entries are denoted by the *<entry>* element, which can occur as a child of either: the root *<configuration>* element or a *<configuration-module>* element. In either case, it must occur before the declaration of modules—inline or imported. Hence if it occurs under the root *<configuration>* element, it must occur just after the *<execute-before-adapt>* element and before the first *<configuration-module>* or *<configuration-import>* element. If it occurs under a *<configuration-module>* node, it must do so before the first child *<configuration-module>* or *<configuration-import>* element.

Each entry must specify a unique—within the context of the encapsulating module alone, and excluding child modules—key/identifier against which its values are stored. The key is specified as the value of the *entryKey* attribute, which is described below:

| name | value-type | use | default | Description |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| entryKey | string | mandatory | N/A | The key to which the entry’s values should be associated in the corresponding org.obix.configuration.Configuration instance. The key is the parameter-value supplied to a *getStringValue(….)* method invoked on the instance, in order to return the value(s) for the entry. The key only needs to be unique in the local context of the entry’s parent module. As such, the module’s parent or child can have an entry with the same key. |

An entry can have several values, each encapsulated with a *<value>* tag—which does not have any attributes and expects a string body-content.

The following example demonstrates the use of the *<entry>* element (highlighted in bold).

```
<configuration reloadOnModify="true" reloadInterval="86400000">
  <execute-before-adapt>
                                                    ……………………….

  </execute-before-adapt>

  <entry entryKey="database.url">
     <value>jdbc:mysql://localhost:2503/orderDb</value>
   </entry>
   <entry entryKey="database.userId">
    <value>ordermanager</value>
   </entry>
   <entry entryKey="database.password">
    <value>nopassword</value>
   </entry>

   <configuration-module moduleId="order.tmp.space">
    <entry entryKey="data.cache">
     <value>C:/ObixExampleApp/data/order.cache</value>
    </entry>

    <configuration-module moduleId="order.secondary.space">
     <entry entryKey="data.cache">
      <value>C:/ObixExampleApp/data/order.cache1</value>
      <value>C:/ObixExampleApp/data/order.cache2</value>
      <value>C:/ObixExampleApp/data/order.cache3</value>
      <value>C:/ObixExampleApp/data/order.cache4</value>
     </entry>

     <configuration-import isRelativeLink="true"
               moduleId="import.module.1">
      imported-config.xml
     </configuration-import>

    </configuration-module>

   </configuration-module>


   <execute-after-adapt>
                                                    ……………………….
   </execute-after-adapt>

 </configuration>
```

Although this should be apparent from the quickstart examples, it is important to state that an entry is accessible, programmatically, only through the *org.obix.configuration.Configuration* instance which corresponds to the module—the *<configuration>* [where applicable referenced by the *moduleId* of the encapsulating *<configuration-import>* element], or *<configuration-module>* element—under which it is declared.

### 2.1.1.4 Modularizing Configuration Data

Configuration modules can either be declared ‘inline’ within a configuration document, or an external document can be referenced as an ‘import’ module. For simplicity, we describe an ‘inline’ module as one whose contents and child modules are specified within the same configuration document; whilst an ‘import’ module is one whose contents are taken whole from an externally referenced document. Note that an inline module can contain child modules specified as import modules.

Each module corresponds to an instance of the class *org.obix.configuration.Configuration.* For a given configuration set, instances of this class are effectively built into a module-tree, where the root configuration document corresponds to the root instance. For a given parent module, a child module can be retrieved by calling the method *getModule(...)* specifying the identifier of the child as the argument. This method is applicable regardless of how the module is declared i.e. regardless of whether or not the module is declared inline or via an import.
An inline module, depending on where it occurs, can be defined using the *<configuration-module>* tag. The resulting element can contain configuration entries, and other modules—either inline or import modules. This element-type supports a single attribute which is described below:

| name | value-type | use | default | description |
|---|---|---|---|---|
| moduleId | string | mandatory | N/A | A unique identifier for the module. This is the value that is passed to the *getModule(...)* method, when invoked on the parent module, so as to obtain a reference to the *org.obix.configuration.Configuration* instance corresponding to the module. |

The following example illustrates how the *<configuration-module>* element can be used.

```
<configuration reloadOnModify="true" reloadInterval="86400000">
  <execute-before-adapt>
                                   …………………….
  </execute-before-adapt>

   <configuration-module moduleId="parent.inline.module">

    <configuration-module moduleId="child.inline.module">

       <configuration-module moduleId="grandchild.inline.module">

                                   …………………….

       </ configuration-module>

    </configuration-module>

   </configuration-module>


  <execute-after-adapt>
                                   …………………….
  </execute-after-adapt>

 </configuration>
```

In some scenarios, it is helpful to have the data for a configuration module externalized in a separate document; in other words, it can sometimes be useful to include an entire configuration document as a child module. This can be achieved using the *<configuration-import>* element, which can occur as a child of either the

*<configuration>* or *<configuration-module>* element. If it occurs as a direct child of a *<configuration>* node, it must be placed after all *<entry>* and *<configuration-module>* elements and before the listener block *<execute-after-adapt>*. Where it occurs as a child of a *<configuration-module>* element, it must occur after all *<entry>* and child *<configuration-module>* elements.

The *<configuration-import>* element's body (text) is expected to contain a locator (such as a filename or URI) to the document to be imported. This element-type supports two attributes which are described below:

| name | value-type | use | default | description |
|---|---|---|---|---|
| moduleId | string | mandatory | N/A | A unique identifier for the module defined by the import i.e. the identifier of the module into which the imported data will be loaded. This is the value that is passed to the *getModule(…)* method, when invoked on the parent module, so as to obtain an object reference to the imported module. |
| isRelativeLink | boolean | optional | true | Indicates whether or not the body-content (text) of the element should be interpreted as a relative or absolute URI. If a value of *true* is specified, the adapter will attempt to find the imported document within the location where the importing document is stored (e.g. in the case of filepaths, this would be the same directory). If a value of *false* is specified, then the element's body text is interpreted as an absolute URI. |

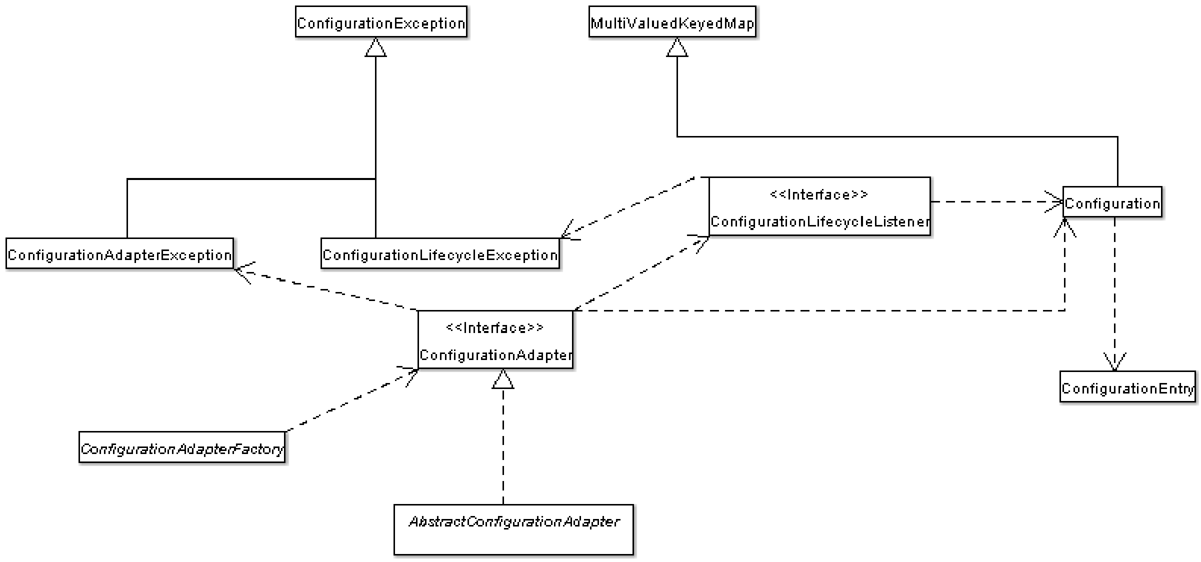
The following example illustrates the use of the *<configuration-import>* element.

```
<configuration reloadOnModify="true" reloadInterval="86400000">
  <execute-before-adapt>
                                     …………………….
  </execute-before-adapt>

   <configuration-module moduleId="parent.inline.module">

    <configuration-module moduleId="child.inline.module">

       <configuration-module moduleId="grandchild.inline.module">

                                     …………………….

       </ configuration-module>
       <configuration-import moduleId="grandchild.module.import">
          a-relative-file.xml
       </configuration-import>
    </configuration-module>

    <configuration-import moduleId="child.module.import" isRelativeLink="false">
          file://server-root/an-absolute-filepath.xml
    </configuration-import>

   </configuration-module>

    <configuration-import moduleId="parent.module.import"
                         isRelativeLink="false">
          http://www.mywebsite.com/config/an-absolute-url.xml
    </configuration-import>

  <execute-after-adapt>
                                          …………………….
  </execute-after-adapt>

 </configuration>
```

### 2.1.2 Java API

An application's interaction with the framework is governed by the classes and interfaces defined in the package *'org.obix.configuration'*, which is illustrated in the following diagram.

In this subsection, we present a summary of the objects and interfaces, which constitute this package, including their roles and responsibilities. This section should be read to gain a deeper insight into the workings of the framework, as well as a pre-cursor to extending the base distribution or creating your own implementation.


ConfigurationException
MultiValuedKeyedMap
<<Interface>> ConfigurationLifecycleListener
Configuration
ConfigurationAdapterException
ConfigurationLifecycleException
<<Interface>> ConfigurationAdapter
ConfigurationEntry
ConfigurationAdapterFactory
AbstractConfigurationAdapter

#### 2.1.2.1 Adapters and Adapter-Factories

Configuration data is loaded and managed by an adapter, where an adapter is a class that implements the interface *org.obix.configuration.ConfigurationAdapter*.

An adapter instance is obtained through an adapter-factory—an instance of the abstract class *org.obix.configuration.ConfigurationAdapterFactory*. This class provides the static method *'newAdapterFactory(…)'* which returns the adapter-factory implementation that is applicable to the current runtime. It determines the implementation by examining the system/runtime property "*org.obix.configuration.AdapterFactory*"; the value of this property must be the fully qualified name of a Java class which extends the abstract class. Where this property is not specified, the method returns an instance of the default factory, *org.obix.janex.j2seadapter.J2SEConfigurationAdapterFactoryImpl*.

Note that some of the extensions provided by the Janex implementation enable adapters to be instantiated and configured through constructs such as JMX, or J2EE application descriptors, which we will discuss later (see section 3.2).

An adapter's role is simply to load a configuration set, specified by the location of the root configuration document, into an instance of the class *org.obix.configuration.Configuration*; where this instance corresponds to the root configuration-document i.e. the root of the module tree. If the root-document defines further child modules, it is the responsibility of the adapter to encapsulate each of these in a child object. Thus each module will correspond to an instance of the class *org.obix.configuration.Configuration*, and every instance (except for the root) will have the root instance as its ancestor—i.e. can be traced back to the root instance.

In order to simplify the task of implementing custom adapters, and to provide access to package-protected constructs in the package *'org.obix.configuration'*, the framework provides an abstract adapter class *org.obix.configuration.AbstractConfigurationAdapter*. It is recommended that developers, who wish to implement custom adaptors, do so by extending this class.

**NOTE:** An adapter or factory instance is not necessarily thread-safe; however there is no limit to the number of adapters or factories that can be instantiated by an application.

#### 2.1.2.2 Configuration and Configuration-Entries

As previously mentioned, configuration modules are represented by instances of the class *org.obix.configuration.Configuration*. The root configuration document—located via the URI supplied to the *org.obix.configuration.Adapter* instance—represents the root *org.obix.configuration.Configuration* object. Where this document defines modules, these modules are child *org.obix.configuration.Configuration* instances, which can be obtained by invoking the method *getModule(…)* on the parent, specifying the child module's identifier as its argument. Where these child modules also have child modules, their corresponding instances will also have child instances, which can be referenced in similar fashion. Consequently, instances of this class form a tree structure, mirroring the module structure of the configuration data-set, with the root-configuration-document represented by the root instance. The root instance can be determined by the return value of the method *isRoot(…)*, which should return *'true'* if the instance corresponds to the root of the module tree.

The specification provides a static instance of the class *org.obix.configuration.Configuration* which can be obtained by invoking the static method *getConfiguration()* on the class. The adapter implementation will generally expect that an instance is specified, into which the configuration data will be loaded, and it is perfectly allowable to use the static instance. In fact, it is made available for environments which do not have complex data-segregation requirements, and where data can easily be shared using a static context. In some other environments this is not possible or perhaps just not desirable, and in such scenarios application developers can elect for the configuration data to be loaded into other [non-static] instances. More advanced adaptors (see 3.2) allow for new instances to be created and bound into a naming-context—applicable in environments where JNDI is supported. Such strategies may be better suited to enterprise environments where more comprehensive data sharing strategies are provided.

Configuration entries are encapsulated by instances of the class *org.obix.configuration.ConfigurationEntry*; however to reduce the call depth of applications, convenience methods are provided in *org.obix.configuration.Configuration* which, to a great extent, remove the need to directly reference entry instances.

#### 2.1.2.3 Lifecycle Listeners

A lifecycle listener is a class which implements the interface, *org.obix.configuration.ConfigurationLifecycleListener*, and consequently is able to receive notification of events at the start, and end, of the configuration document loading process. The listener interface is provided to enable developers to easily perform application initialization tasks at the same time as the loading of configuration data. For example instantiating database-connection pools, or environment-specific setup such as performing system/environment diagnostics.

Listeners are declared within a configuration-document, and are invoked each time the document is loaded. As such, if a given document is imported (as a module) at *x* different points in a configuration set, the listeners related to it are executed *x* times each. If the *'reloadOnModify'* feature is enabled in a document, the listeners declared in the document will be executed each time it is reloaded. Each listener declaration results in a new instance of the listener being created by the relevant adapter, and as such, multiple declarations of the same listener-class will result in multiple instances of the given type.

Listeners can be declared at two points in a configuration document, either at the start under the node *<execute-before-adapt>*, or at the end under the node *<execute-after-adapt>*. When listeners are declared at the start of a configuration document, they will be invoked before the data contained within the document is loaded into a corresponding configuration instance, and consequently will not have access to the data. Where the listener is declared at the end of a file, it will be executed after the data contained in the document is loaded; hence it will have access to the data defined in the document.

It is worth mentioning that listener notification is not necessarily asynchronous; the listener invocation can be performed by the same thread that reads the document contents. Also, an unhandled listener exception/error will cause the load process to be aborted.

#### 2.1.2.4 Exceptions

It is important to mention that all exceptions raised by the framework during the course of an operation will cause the operation to be halted. For example, if an exception is raised the first time the configuration data is being loaded, perhaps at system-startup, the load sequence will be aborted—with the consequence that the application will not have access to the configuration data. In the same manner, if an exception occurs whilst the configuration data is being reloaded, as a result of modification(s) to the source document(s), the reload sequence will be aborted but the application will still have access to the last loaded set of configuration data.

The framework specifies three exception classes, which are:
*org.obix.configuration.ConfigurationException*;
*org.obix.configuration.ConfigurationLifecycleException;*
*org.obix.configuration.ConfigurationAdapterException.*

All exceptions specified in the framework extend *org.obix.configuration.ConfigurationException*. All implementation developers are obliged to ensure that all exceptions they propagate through the framework are subclasses of this class. As such, application developers can regard it as the base class for all framework exceptions. It is provided for the simple reason of simplifying application error handling.

Exceptions of the type *org.obix.configuration.ConfigurationLifecycleException* are thrown by lifecycle listeners to indicate a listener failure. On the failure of a listener, no subsequent listeners will be executed and the operation being performed (loading/reloading of data) will be aborted.

Adapters and adapter factories throw exceptions of the type *org.obix.configuration.ConfigurationAdapterException* when an operation fails. This indicates that: an error has occurred during the course of instantiating or accessing an adapter or a factory; or that an error has occurred during the process of loading or reloading configuration data. As with all other framework exceptions, when an instance of this exception is thrown, the operation being performed is aborted.

## 2.2 The Janex Implementation

*Janex* is the codename of the default implementation supplied with the framework. Its role is to manage configuration data in a manner transparent to client applications. Application developers should not have to reference classes/interfaces in the implementation programmatically; rather, their interaction should be restricted solely to specification interfaces and classes.

The Janex implementation provides sufficient functionality to meet the needs of most applications, and as such, there should be little or no need to extend it, nor to write your own custom implementation—except for highly specialised requirements. It provides a factory implementation and a host of adapters for both standard and enterprise application environments—supporting standards such as J2EE and JMX. In addition, it also provides a continuously evolving suite of extensions to ease the integration of other frameworks such as Log4J and Hibernate. In cases where developers wish to extend its functionality, it provides a common set of utilities to simplify this.

The following subsections summarise, briefly, the chief constituents of the implementation, so as to provide developers with greater knowledge of the facilities it provides.

### 2.2.1 Adapters and Factories

This implementation provides a single adapter-factory *org.obix.janex.j2seadapter.J2SEConfigurationAdapterFactoryImpl*, which creates instances of the adapter *org.obix.janex.j2seadapter.J2SEConfigurationAdapterImpl*. This adapter is most suited for J2SE environments, and for this reason, Janex also provides two additional adapters: *org.obix.janex.ext.jmx.JMXConfigurationAdapterImpl*, and *org.obix.janex.ext.j2ee.ObixServletConfigLoader*, which are better suited to enterprise environments.

### 2.2.2 Lifecycle Listeners

Janex provides listeners to simplify the initialisation of other frameworks. The listener implementations *org.obix.janex.ext.log4j.ObixLog4jAdapter*, and *org.obix.janex.ext.hibernate.HibernateConfigurationInitializer* enable the initialisation of Apache Log4J and Hibernate respectively, during loading/reloading of configuration data. These listeners remove the need for application developers to programmatically initialise these frameworks in their application code, thus further simplifying application development. For more information on how to use these listeners, please see section 3.3.

# 3 Application Integration

The base distribution provides a number of options for integrating the framework into software applications. The route chosen is probably a trade-off between flexibility, the environment in which the application is to be deployed, and the amount code the application developer(s) is willing to write.

In this section, the available integration options are categorised under the environments to which we feel they are best suited. This is done on the basis that they offer the best—most achievable—level of flexibility in those environments, as well as reducing/removing the need for specific application integration code when used accordingly.

## 3.1 Basic Integration (J2SE)

In a typical J2SE application, be it an applet, desktop application or standalone process, there will most likely be an initialization block where the application creates/sets-up the resources it requires. As opposed to an enterprise environment, there are generally no standard container facilities in J2SE, where resource setup can be accomplished extant to application code. For that reason, the application developer has to call a configuration adapter to actually initiate the process of loading the configuration data. This should not take more than a few lines of code as is illustrated below—this approach is also used for the example applications described in section 1.2.

```
 ConfigurationAdapterFactory adapterFactory=
         ConfigurationAdapterFactory.newAdapterFactory();

 ConfigurationAdapter adapter=adapterFactory.create(map);

adapter.adaptConfiguration(Configuration.getConfiguration(),
                                        configLocation);
```

This example can be distilled into the following distinct steps:

1. Obtain an adapter-factory by calling the method *ConfigurationAdapterFactory.newAdapterFactory()*. The factory instance returned by this method will be the default adapter *org.obix.janex.j2seadapter.J2SEConfigurationAdapterFactoryImpl*, except you specify an implementation name as the value of the Java runtime parameter (using java –D"param-name=param-value") "*org.obix.configuration.AdapterFactory"*. Note that the value of this parameter must be a fully qualified name of the factory implementation i.e. a class that extends the abstract class *org.obix.configuration.ConfigurationAdapterFactory*.
2. Create an adapter by invoking the method *create(java.util.Map)* on the factory instance. The map is used to pass parameters to the factory instance, and its contents are implementation specific. As such, if the implementation being used does not require any parameters, it is perfectly acceptable to specify a *null* argument.
3. Invoke one of the methods, *adaptConfiguration (org.obix.configuration.Configuration, java.lang.String)*, *adaptConfiguration (org.obix.configuration.Configuration, java.io.File), or adaptConfiguration (org.obix.configuration.Configuration, java.net.URL)* to load the configuration data at the specified location into the specified configuration instance. Where the location can either be specified as a URL, file-path or URI string. A configuration instance, into which the data is to be loaded, must also be

supplied; however a static instance is provided and this may be ideal for J2SE environments where information can be shared easily by using a static context. To use the static instance, simply pass its reference to the *adaptConfiguration(….)* method, where the reference is obtained by calling the method *Configuration.getConfiguration().* Alternatively, you may wish to use your own instance, which can be instantiated like any other Java object using the *new* operator.

## 3.2 Advanced (Enterprise) Integration

The advantage of an enterprise environment—such as J2EE—is that there are standard services provided by the container; thus reducing/removing the need for developers to write infrastructure code.

The main advantage of an enterprise environment, from the perspective of the framework, is the ability to completely externalise the loading of configuration data from application logic. In simple terms, within an enterprise environment, there is little or no need to actually write code to manually invoke a configuration adapter. In addition to this, an enterprise environment also provides naming facilities (via JNDI), which in turn means that configuration data/instances can be more easily shared across an application's component landscape. Application-developers do not have to rely on a static context or write custom code to share configuration instances across different enterprise components. The framework provides a set of enterprise adapters to enable configuration data to be automatically bound into the JNDI context of the executing application. This means that configuration data can be treated, by the application, as any other resource—e.g. datasources—accessible via JNDI.

### 3.2.1 Web Application Adaptor

The web-application-adapter is provided for servlet environments. It relies on the use of the servlet-context-listener (cum adapter implementation) *org.obix.janex.ext.j2ee.ObixServletConfigLoader*, which loads the configuration data at context initialisation.

This listener class is specified in the web-application deployment descriptor, and makes use of servlet-context parameters; as a consequence, it does not have to be referenced explicitly in application code. In other words, the application does not have to contain any code-logic to specifically load configuration data, as this is achieved entirely in the deployment descriptor.

We illustrate the use of this adapter/context-listener through an example. Assume that the configuration file is located in a web-application archive (war) at the path "*/WEB-INF/config/application-config.xml*", and that we wish to load the contents of this file into a configuration object which we want to bind to the naming (JNDI) location *"myWebApp/configuration"*. The corresponding web-application deployment descriptor would look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
 <display-name>My Web Application</display-name>

 <context-param>
  <param-name>obix.configuration.file</param-name>
  <param-value>/WEB-INF/config/application-config.xml</param-value>
 </context-param>
 <context-param>
  <param-name>obix.bind.configuration.to.location</param-name>
  <param-value>obixweb/configuration</param-value>
 </context-param>

 <listener>
  <listener-class>
   org.obix.janex.ext.j2ee.ObixServletConfigLoader
  </listener-class>
 </listener>

 <servlet>

   ..........

 </servlet>

  ...................


</web-app>
```

In the above listing, we first of all specify the location of the configuration-document as a context-parameter with name "*obix.configuration.file*". The value of this parameter is assumed to be relative to the web-application root, however this can be overridden by the use of a context-parameter which we will describe later on. The next parameter in the deployment descriptor is the JNDI location to which the *org.obix.configuration.Configuration* instance—that results from loading the configuration data—will be bound. If this parameter is not specified, then the adapter will load the data into the static instance. Where the JNDI location is specified, note that the resulting instance will be given the web-application's name as its identifier; however it is possible to specify a custom identifier via a context-parameter which we will describe shortly. The final and most crucial part—from our perspective—of the web-application deployment-descriptor is the declaration of the servlet-context listener *org.obix.janex.ext.j2ee.ObixServletConfigLoader*. This class is an extension of the J2SE adapter *org.obix.janex.j2seadapter.J2SEConfigurationAdapterImpl*, which also implements the *javax.servlet.ServletContextListener* interface. As such, in order to use this extension, you will have to place all the jar files in the base distribution, plus the jar files in the extension download into the web-application's classpath.

The following table lists the context parameters which can be used in conjunction with this adapter/context-listener.

| name | use | default | description |
|---|---|---|---|
| obix.validate.configuration.file | optional | false | A value of *true* indicates that the adapter should use a validating XML parser to read the configuration document. |
| obix.configuration.file | required | N/A | The location of the configuration document to be loaded by the adapter. This location is assumed to be relative to the root of the web-application archive, except a value of *EXTERNAL* is specified for the context-parameter *'obix.configuration.filelocation.type'*. See below. |

| | | | |
|---|---|---|---|
| obix.configuration.filelocation.type | optional | N/A | A value of *EXTERNAL* indicates that the configuration-document is external to the web-application. As a consequence, its path will be treated as an absolute path i.e. the value of the context-parameter *'obix.configuration.file'* is treated as an absolute URI—URLs are also supported by implication.<br><br>All values of this parameter, other than *EXTERNAL*, are ignored. |
| obix.bind.configuration.to.location | optional | N/A | When a value is specified for this parameter, the configuration data is loaded into a new *org.obix.configuration.Configuration* instance, and bound into the application's default JNDI context under the specified value. In essence, the value of this parameter specifies the name under which the new instance will be bound. Where this parameter is not specified, the configuration data is loaded into the static configuration instance. |
| obix.configuration.identifier | optional | Servlet Context (Web Application) name. | This parameter is only valid in conjunction with the parameter *'obix.bind.configuration.to.location'*. In other words, it is only valid when a new configuration instance is to be created as part of the configuration-data load. If a value is not specified for this parameter, the new instance is assigned the web-application's name—the context name—as its identifier. |

### 3.2.2 JMX Adaptor

The JMX adaptor is provided for environments which support the JMX specification, and allow for custom services to be deployed as standard JMX MBeans. This adaptor is implemented as an MBean, with the aim of providing a means to manage configuration data as a container-level resource in the same way that datasources are managed at container level.

The adapter is provided mainly for applications where the 'Web Application Adapter' (see previous section) can't be easily used e.g. a middleware application which consists entirely of EJBs (Enterprise Java Beans). It is also provided for environments where system-administrators prefer to manually manage configuration data in the same manner in which other container resources—e.g. datasources—are managed.

It provides methods which can be invoked, either through an MBean administration interface, or programmatically, to load configuration data. It is recommended that the MBean be deployed separately from the application, and that the relevant configuration data be loaded before the application is deployed, started, or invoked. This is really a common-sense suggestion aimed at ensuring that the configuration data which an application requires is available when the application is executed/invoked. In this scenario, the configuration data would have to be adapted/loaded in the same way that the system-administrator has to initialise container resources—e.g. mail sessions, JMS connectors, datasources etc—before applications are deployed, started or executed.

The MBean provides four MBean operations which are summarised below. For further details, please consult the framework's Javadoc at http://obix-framework.sourceforge.net/docs/apidocs/index.html, or the MBean information—viewable through the MBean administration interface.

| operation name | description |
|---|---|
| *loadConfigurationFromURL(...)* | Loads configuration data from a given URL and binds the resulting *org.obix.configuration.Configuration* instance into the default JNDI context under a specified name. This method is provided for scenarios where it is inappropriate to share the static/global configuration instance across an application. **Note** that this operation always creates a new configuration instance, and does not update the static/global instance in any way. |
| *loadGlobalConfigurationFromURL(...)* | Loads configuration data from a URL, and, into the global/static configuration instance. **Note** that this method alters the state of the global/static instance i.e. it overwrites its contents with the data loaded from the given URL. Sharing the static/global configuration instance across many components in a large application is likely to lead to problems; hence, it is strongly discouraged. |
| *loadConfigurationFromFile(...)* | Loads configuration data from a given file-path and binds the resulting *org.obix.configuration.Configuration* instance into the default JNDI context under a specified name. This method is provided for scenarios where it is inappropriate to share the static/global configuration instance across an application. **Note** that this operation always creates a new configuration instance, and does not update the static/global instance in any way. |
| *loadGlobalConfigurationFromFile(...)* | Loads configuration data from a file, and, into the global/static configuration instance. **Note** that this method alters the state of the global/static instance i.e. it overwrites its contents with the data loaded from the given URL. Sharing the static/global configuration instance across many components in a large application is likely to lead to problems; consequently, it is not recommended. |

## 3.3 Examples of Initialisation via Listeners

There are certain tasks, which whilst not directly related to the task of loading configuration data, can be performed more easily at the same time. These tasks are generally related to the initialization of application resources, and are more commonly performed at application start-up.

The framework attempts to simplify these tasks through the use of configuration-lifecycle-listeners. This section presents two listeners to illustrate how the initialization of the Apache Log4J logging framework, and the Hibernate object-to-relational mapping framework, can be simplified.

A **note of caution** on the use of listeners: each time a configuration document is loaded—be it at first load, or a reload—any listener's declared within it are invoked. Whilst in most cases, this is not a problem; it is still something that should be borne in mind when you are deploying your application.

### 3.3.1 Log4J

In order to make effective use of the Log4J logging framework, developers generally have to initialise it at runtime using a Log4J configuration document. The most common way to achieve this is to invoke one of the static *configure(...)* methods on the class *org.apache.log4j.xml.DOMConfigurator*. It makes sense that this should be done at the time of application initialisation/start-up so as to make logging functionality fully available to subsequent operations.

Whilst this mode of initialisation is very simple, effective and also efficient, the Janex distribution provides a configuration lifecycle listener *org.obix.janex.ext.log4j.ObixLog4jAdapter*, which altogether eliminates the need to write code to initialise Log4J. The listener is simply declared at the start of an obix-configuration-document, with the Log4J configuration-document-location specified as its parameter. This has the effect of initialising the Log4J framework as part of the obix-configuration loading process.

The following example configuration document demonstrates the use of this listener.

```
<configuration>
 <execute-before-adapt>
  <listener class="org.obix.janex.ext.log4j.ObixLog4jAdapter">
   <parameters>
    <parameter entryKey="log4j.configuration.file">
     <value>log4j-config.xml</value>
    </parameter>
    <parameter entryKey="log4j.configuration.file.resolution.policy">
     <value>RELATIVE</value>
    </parameter>
   </parameters>
  </listener>
 </execute-before-adapt>-->
 ....................
 ....................
 ....................

</configuration>
```

In the above document, we have simply instructed the adapter to execute the Janex Log4J adapter before the data defined in the document is loaded. The adapter expects that we supply the location of the Log4J-XML-configuration-file as the value of the parameter *'log4j.configuration.file'*. An additional and optional parameter—'log4j.configuration.file.resolution.policy'—can be specified to indicate how this path should be resolved. This listener's applicable parameters are explained in greater detail in the table below.

| name | use | description |
|---|---|---|
| log4j.configuration.file | required | The path (URI) of the Log4J configuration document. The value of this parameter can either be treated as an absolute or relative URI, or the path to a resource in the application's classpath. The manner in which it is treated depends on the value of the parameter *'log4j.configuration.file.resolution.policy'*; however where this parameter is not specified, heuristics, as described below, are used to resolve the URI/resource-path. |

| | | |
|---|---|---|
| log4j.configuration.file.resolution.policy | optional | Determines how the value of the parameter *'log4j.configuration.file'* should be treated. A value of *'RELATIVE'* indicates that the value is a URI relative to the location of the obix-configuration-document being loaded. A value of *'ABSOLUTE'* indicates that the value is an absolute URI. Whilst a value of *'CLASSPATH'* indicates that the value is the path to a resource in the application's classpath.<br><br>Where no value is specified for this parameter, the listener sequentially applies the values (and their related rules) *'ABSOLUTE', 'CLASSPATH'* and *'RELATIVE'* in a bid to locate the Log4J configuration file. It adopts the first successful scheme i.e. it stops as soon as it has found a matching scheme [located the file]; where all three schemes fail, the listener raises an exception. |

### 3.3.2 Hibernate

In order to manage object-to-relational mapping easily via a Hibernate session, it may be necessary that the classes (types) of the objects to be managed are specified during the construction of the Hibernate *org.hibernate.SessionFactory* instance. Achieving this programmatically involves adding each type, via the *addClass(…)* method, to the *org.hibernate.cfg.Configuration* instance with which the SessionFactory is created. Even in the smallest of applications, this can lead to long initialization blocks.

The Janex distribution provides a lifecycle listener *org.obix.janex.ext.hibernate.HibernateConfigurationInitializer*, which simplifies this process, and removes the need to write initialization code. The following obix-configuration-document illustrates its use.

```
<configuration>

 ....................

 <execute-after-adapt>
  <listener
     class="org.obix.janex.ext.hibernate.HibernateConfigurationInitializer">
   <parameters>
    <parameter entryKey="hibernate.mapped.classes">
     <value>com.example.phonebook.Address</value>
     <value>com.example.phonebook.Person</value>
     <value>com.example.phonebook.ContactDetails</value>
    </parameter>
   </parameters>
  </listener>

</configuration>
```

The listener expects that the names of the classes, to be managed by the resulting Hibernate session, are specified as the values of the parameter *'hibernate.mapped.classes'.*

Although Hibernate provides support for JNDI, thus allowing the SessionFactory to be bound into the application's JNDI context, this listener also provides facilities to enable the factory to be bound into the obix configuration instance that encapsulates the contents of the document being loaded. This is achieved by specifying the listener parameter *'hibernate.sessionfactory.config.key'*. The value of this parameter is the configuration key under which the SessionFactory will be stored. It is recommended that you ensure the key is unique in the context of the configuration instance. This facility is provided for environments which do not provide JNDI support.

# 4 Conclusion

This document has described the Obix Configuration Framework (the framework). It has provided a quick overview of it, as well as “quickstart” examples to “flatten” the learning curve involved in using it. For more advanced users, it has described the framework’s structure and the role played by the various components included in the base distribution. Finally, the document has detailed the different ways in which the framework can be integrated into application code, as well as utilities provided by the framework to simplify the use of other frameworks such as Apache Log4J and Hibernate.

# 5 Acknowledgements

- Ugo Ezechukwu *(editing)*
- Yinka Erinle *(review)*
- Lewis Foti *(editing)*
- Ian Gerry *(review)*